Содержание

Введение

В современном мире многие **TODO** *общественные явления* приобретают цифровую форму. Так, например: банковские операции, беспилотные автомобили, умные дома, хранилища информации о частной жизни людей имеют программно-информационные аспекты. В связи с этим становится очевидным актуальность обеспечения информационной безопасности.

Зачастую злоумышленники получают доступ к конфиденциальным данным по средствам нахождения уязвимостей в программном обеспечении. Также злоумышленники могут изменять исходный код программного обеспечения для устранения частей, отвечающих за защиту, с целью дальнейшего незаконного распространения данной программы. И в первом, и во втором случае применяется программа, называемая отладчиком.

Изначально разработанные для упрощения процесса поиска ошибок в собственных программах, отладчики получили широкое распространение как инструмент взлома.

Целью данной работы является разработка методов обеспечения защиты программы от отладчика. Данные методы должны как обеспечивать обнаружение факта работы программы под отладчиком, так и препятствовать самому процессу отладки.

В связи с этим были поставлены следующие задачи:

- Изучить внутреннее устройство программы отладчика.
- Разработать алгоритм защиты от отладчика.
- Реализовать полученный алгоритм, обеспечив при этом механизм достаточным уровнем скрытности.
- Провести тестирование полученной системы защиты.

ГЛАВА 1. ОБЩИЕ СВЕДЕНИЯ О РАБОТЕ ОТЛАДЧИКА

Данная глава включает в себя общие сведения о работе отладчика, которые будут затем использованы для защиты программ от него. Помимо этого также рассматривается структура исполняемого файла в операционной системе Windows.

1.1. Принципы работы отладчика

Отладчик — это программа, которая упрощает разработку программного обеспечения, предоставляя разработчику способы поиска ошибок. Обычно функционал отладчика предоставляет следующие возможности:

- Поставить точку останова. Например, пометить инструкцию, дойдя до которой, программа должна остановить свое выполнение и передать управление отладчику.
- Трассировать программу. То есть, последовательно выполнять инструкции, и после каждой останавливать выполнение программы и передавать управление отладчику.
- Отобразить состояние регистров процессора на момент останова.
- Отобразить состояние стека процесса на момент останова.

Возможности предоставляемые отладчиком могут быть использованы злоумышленниками для изучения уязвимостей программного обеспечения и обхода ограничений и защиты. Для лучшего понимания способов защиты от отладчика рассмотрим, как работает каждая из предоставляемых им возможностей более подробно.

Отладчик может самостоятельно запустить отлаживаемый процесс. В рассматриваемой системе Microsoft Windows для этого нужно вызвать функцию `CreateProcess` с указанием ей в качестве параметра `fdwCreate` константы `DEBUG_PROCESS`. И также отладчик может подключиться к уже работающему процессу. Для этого ему необходимо получить идентификатор процесса при помощи функции `OpenProcess`, после чего вызвать `DebugActiveProcess` и таким образом подключиться к нему. Как правило, отладчик открывает процесс с доступом на чтение и запись в виртуальную память процесса.

Затем отладчик в цикле обрабатывает события отладки, используя функцию `WaitForDebugEvent`. После завершения обработки очередного со-

бытия отладки вызывает функцию ContinueDebugEvent. Общую схему работу отладчика можно представить следующим образом:

```
CreateProcess("FileName.exe", ..., DEBUG_PROCESS, ...);
for (;;) {
  WaitForDebugEvent(&dbgEv, INFINITE);
  switch(dbgEv.dwDebugEventCode)
  {
  case EXCEPTION_DEBUG_EVENT:
  ...
  }
  ContinueDebugEvent( dbgEv.dwProcessId,
                      dbgEv.dwThreadId,
                      dwContinueStatus );
}
```

Перейдем к рассмотрению конкретных возможностей предоставляемых отладчиком.

1.2. Трассировка

Трассировка — последовательное выполнение программы, при котором после каждой инструкции управление передается отладчику. В этом режиме программист может детально отследить изменения значений всех параметров процесса. Обеспечение режима пошагового выполнения программы предусмотрено на аппаратном уровне.

В архитектуре x86 есть регистр флагов (EFLAGS), состоящий из 32-х бит, каждый из которых отображает состояние процессора (рис. 1.1). Восьмой из них является флагом трассировки (trap flag). Если этот флаг равен 1, то процессор будет выполнять прерывание с номером 1 после каждой инструкции. При выполнении прерывания 1 процессор выполняет передачу управления в обработчик прерывания, который помещает состояние всех регистров процессора в стек, после чего передает необходимую информацию отладчику. Обработчик прерывания в конце своей работы может либо оставить флаг трассировки равным 1, либо перевести его значение в 0.

Пример установки флага трассировки:

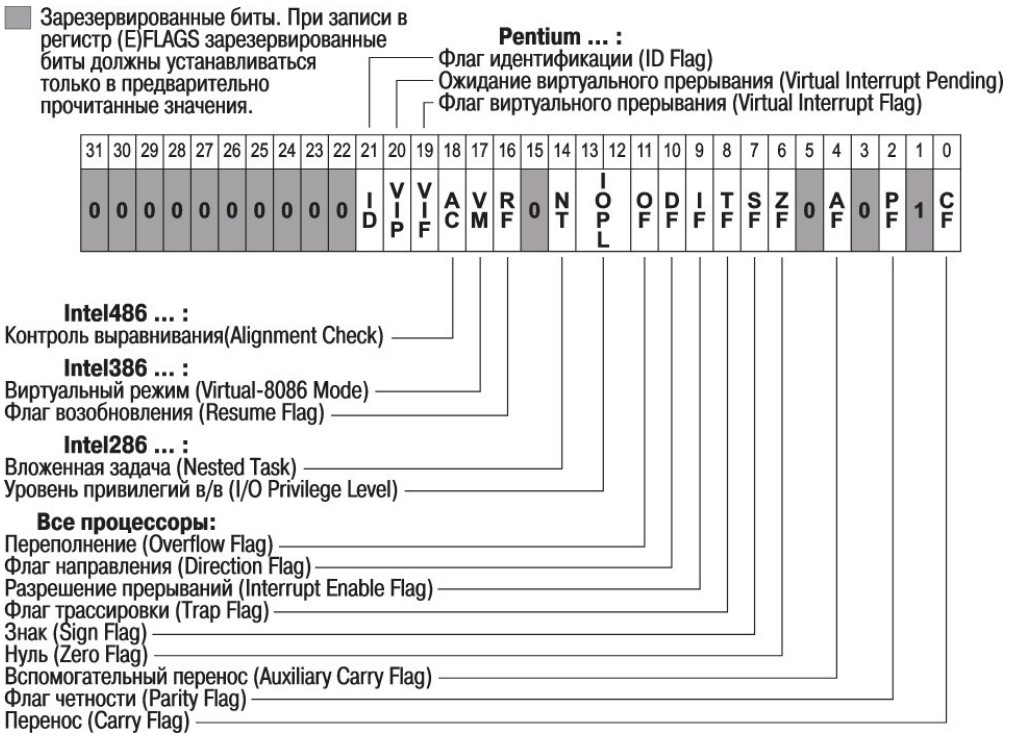


Рис. 1.1 Регистр флагов процессора

```
pushf                     ; Помещаем регистр флагов в стек
mov EBP, ESP              ; Сохраняем адрес вершины стека
or  WORD PTR[EBP], 0100h  ; Устанавливаем флаг TF
popf                      ; Восстанавливаем регистр флагов
```

Соответственно, чтобы снять флаг трассировки достаточно заменить инструкцию `or` на инструкцию:

```
and  WORD PTR[EBP], FEFFh
```

Таким образом, можно заметить, что при проведении трассировки исходный код отлаживаемой программы никак не меняется.

1.3. Точки останова

Точка останова — место в коде программы, дойдя до которого, процессор должен прервать выполнение программы и передать управление отладчику. После этого программист может просмотреть параметры состояния программы, поставить или убрать другие точки останова или запустить трассировку. Точки останова бывают двух типов: программные и аппаратные.

Рассмотрим принцип работы каждой из них.

1.3.1. Программные точки останова

Программные точки останова реализованы следующим образом. Когда программист ставит точку останова на какой-либо инструкции, отладчик запоминает данную инструкцию в своей памяти, а затем заменяет данную инструкцию на

```
int 3 ; Генерация программного прерывания
```

Таким образом, когда процессор доходит до данной инструкции, возбуждается прерывание, управление переходит в обработчик прерывания, откуда затем информация передается в отладчик.

Инструкция `int 3` имеет специальный однобайтовый код операции (`0xCC`), в то время как, обычно прерывания имеют двухбайтовый код операции: `int x` $\rightarrow$ `0xCD x`. Это связано с тем, что может потребоваться заменить в коде однобайтовую операцию, например команду инкремента `inc`. В таком случае, если бы инструкция замены была больше одного байта, то повреждалась бы следующая инструкция. Это в свою очередь накладывает дополнительные расходы в ситуации, когда требуется из точки останова произвести трассировку.

Как видно, установка программной точки останова изменяет исходный код программы.

1.3.2. Аппаратные точки останова

В архитектуре x86 есть шесть регистров, предназначенных специально для отладки. Именуются они `DR0..DR7`, при этом регистры `DR4` и `DR5` не используются. Данные регистры позволяют устанавливать точки останова с различными условиями. Также они являются привилегированным ресурсом, следовательно инструкции, устанавливающие данные регистры, могут выполняться только с нулевого уровня защиты.

Регистры с `DR0` по `D3` содержат линейные адреса точек останова, каждая из которых связана с условием остановки. Условия определены в регистре `DR7`.

В регистре `DR6` содержится статус отладки. Он позволяет отладчику определить, какие условия отладки возникли. Первые четыре бита указыва-

ют, какая из четырех точек останова в регистрах DR0..DR3 сработала. Бит 13 указывает, что следующая инструкция обращается к регистрам отладки. Бит 14 указывает на пошаговое выполнение (включает Trap Flag в регистре EFLAGS).

Регистр DR7 предназначен для управления процессом отладки, он позволяет выборочно включать условия остановки для точек останова в регистрах DR0..DR3. Есть два режима включения регистра: локальный (биты 0, 2, 4, 6) и глобальный (биты 1, 3, 5, 7). При локальном включении процессор сбрасывает условия остановки при каждом переключении задачи. При глобальном включении условия остановки не сбрасываются, а следовательно они используются для всех задач. Биты 17:16; 21:20; 25:24 и 29:28 позволяют установить следующие условия срабатывания точек останова:

- 00b — При выполнении инструкции.
- 01b — При записи данных.
- 10b — При обращении к порту ввода/вывода.
- 11b — Чтение и запись данных.

Как можно заметить, установка аппаратных точек останова никак не меняет исходный код программы.

1.4. Наиболее распространенные отладчики

Для лучшего противодействия отладчику полезно рассмотреть какие отладчики используются на сегодняшний день. Исходя из этой информации, можно организовать простой способ защиты. Например, программа в процессе работы может просмотреть процессы запущенные в системе, и, если среди них встретится какой-либо знакомый отладчик, программа может изменить свое поведение или просто прервать выполнение.

1.4.1. SoftICE

Наиболее важным свойством отладчика SoftICE является то, что он работает на нулевом кольце защиты.

В архитектуре x86 есть три кольца защиты (ring-1, 2, 3). Данные кольца предназначены для ограничения взаимодействия выполняющихся программ между собой и с операционной системой. Как правило, на нулевом кольце

защиты выполняется сама операционная система, которая одна имеет доступ ко всем привилегированным операциям. Рассматриваемый отладчик также располагается на нулевом кольце защиты, что позволяет ему отлаживать не только пользовательские приложения, но также драйвера и саму операционную систему.

Поддержка отладчика разработчиками прекратилась 11 июля 2007 года.

1.4.2. IDA Pro

IDA Pro — интерактивный дизассемблер и универсальный отладчик.

Дизассемблер — программный инструмент, позволяющий получить из машинного кода код на языке ассемблера. По принципу работы они делятся на пассивные и интерактивные. Пассивные предоставляют пользователю готовый листинг программы, а интерактивные позволяют на ходу изменять правила, по которым производится трансляция.

Как дизассемблер IDA Pro способен создавать карты выполнения фрагментов программы, делая полученный код ассемблера еще более понятным человеком.В качестве отладчика IDA Pro охватывает все рассмотренные ранее возможности отладки, обеспечивает доступ ко всем сегментам пространства памяти процесса, а также обеспечивает подробную визуализацию.

Проект активно поддерживается и развивается.

1.4.3. WinDBG

WinDBG — отладчик предоставляемый фирмой Microsoft, предназначенный специально для работы в операционной среде Windows. Является более мощной альтернативой широко применяемому отладчику Visual Studio Debugger. Может использоваться как отладчик режима ядра. Имеет поддержку сторонних расширений. На данный момент является одним из самых применяемых, благодаря своей универсальности.

1.4.4. OllyDBG

OllyDBG — отладчик уровня приложений (ring-3). Помимо поддержки всех рассматриваемых ранее возможностей отладки, больша́я часть мощности OllyDBG заключается в расширениях, которые разрабатывают и выкладыва-

ют в сеть Internet пользователи этого отладчика. OllyDBG имеет бесплатное распространение, а также не требует установки.

Для тестирования разработанных методов в данной работе применяется именно этот отладчик.

1.5. PE формат

Исполняемые файлы в системе Windows имеют общую сигнатуру, называемую PE (Portable Executable). В PE формате содержится различная информация о исполняемом файле, как например: таблицы импорта и экспорта, информация о различных секциях, точка входа и так далее. Структура PE формата представлена на рисунке 1.2.

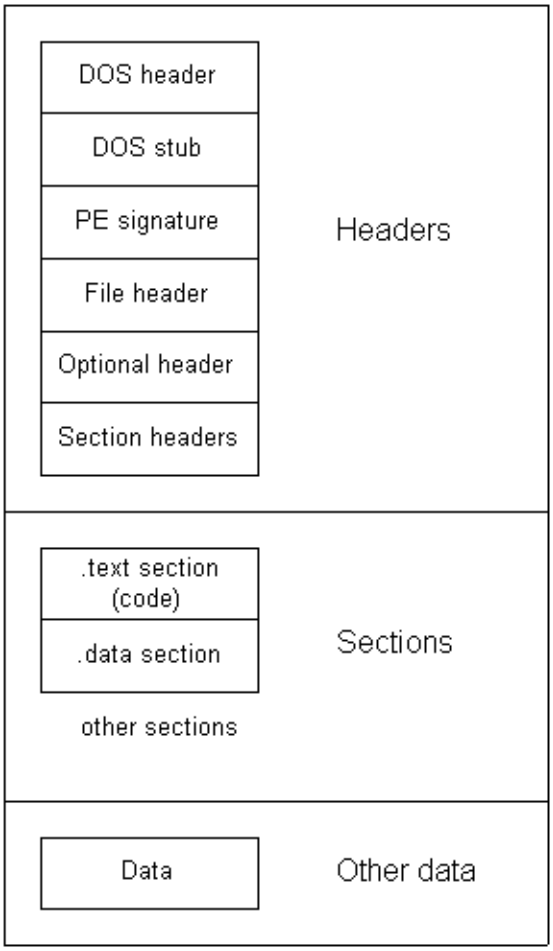


Рис. 1.2 Формат PE

Рассмотрим те части PE формата, которые будут использованы в данной работе. Вначале идет MS-DOS заголовок, первыми двумя байтами которого является сигнатура "MZ". Бóльшая часть полей данного заголовка предна-

значены для запуска из-под DOS. Для дальнейшей работы, кроме проверки сигнатуры, потребуется поле `e_lfanew`, в котором содержится указатель на начало PE-заголовка.

PE-заголовок также имеет свою сигнатуру, которую необходимо проверить на корректность, а именно четыре байта `"PE\0\0"`. В данном разделе хранится количество секций `NumberOfSection`, а также размер в байтах опционального заголовка `SizeOfOptionalHeader`.

Далее расположен опциональный заголовок. Данный раздел содержит:

- смещение адреса входа относительно базового адреса загрузки файла: `AddresOfEntryPoint`;
- рекомендуемый базовый адрес загрузки файла: `ImageBase`;
- количество элементов в таблице `DATA_DIRECTORY`: `NumberOfRvaAndSizes`.

`DATA_DIRECTORY` представляет собой таблицу, каждый элемент которой представляет из себя структуру из двух полей. А именно виртуального адреса данных, на который указывает данный элемент, и их размер.

Далее идет таблица секций. Для каждой секции в этой таблице приведена следующая информация:

- `Name` — имя секции.
- `VirtualAddress` — виртуальный адрес секции в памяти.
- `PointerToRawData` — указатель на данные в файле.
- `VirtualSize` — размер, занимаемый секцией в памяти.
- `Characteristics` — свойства секции. Так, например, секция, которую можно запустить на выполнение должна обладать свойством `IMAGE_SCN_CNT_CODE`.

В данной работе также будет использована секция `.reloc`, в которой содержится таблица базовых смещений. В целях безопасности исполняемый файл может быть загружен по случайному адресу. Соответственно, адреса, используемые в коде программы, необходимо изменить в соответствии с базовым адресом загрузки. Информация о всех местах, которые необходимо скорректировать, содержится с таблице базовых смещений.

ГЛАВА 2. ПРОЕКТИРОВАНИЕ МЕТОДОВ ЗАЩИТЫ ОТ ОТЛАДЧИКА

Данная глава посвящена разбору методов, которые могут обнаружить отладчик или препятствовать процессу отладки. Также в этой главе разбираются задачи, которые необходимо решить при реализации метода с подсчетом контрольной суммы.

2.1. Обзор методов защиты от отладчика

Как было отмечено в предыдущей главе, отладчик может предоставить злоумышленникам большой инструментарий по взлому программы. Следовательно, необходимо разработать механизм, который будет препятствовать работе отладчика. Рассмотрим несколько способов обнаружить, что программа работает под отладчиком.

2.1.1. Возможности предоставляемые операционной системой

Операционная система Windows предоставляет следующие функции:

```
BOOL IsDebuggerPresent();
BOOL CheckRemoteDebuggerPresent(
    [in]    HANDLE hProcess,
    [out]   PBOOL pbDebuggerPresent );
```

Первая функция позволяет понять находится ли процесс, который совершил вызов, под отладчиком. Если процесс выполняется под отладчиком, функция вернет `TRUE` значение, а иначе вернет `FALSE`.

Вторая функция принимает первым параметром дескриптор процесса, а вторым указатель на переменную типа `BOOL`, в которую функция занесет `TRUE`, если указанный процесс выполняется под отладчиком, или `FALSE`, если отладчика нет. Функция возвращает ненулевое значение в случае успеха и ноль, если произошла ошибка.

Недостаток метода основанного на данных функциях заключается в его заметности. Большинство отладчиков имеют расширения, которые позволяют обойти проверку данных функций.

Стоить отметить, что заметность данного метода можно снизить, если воспользоваться техникой, позволяющей скрыть от отладчика вызовы функ-

ций dll. Для этого необходимо найти в памяти процесса адреса функций `LoadLibrary` и `GetProcAddress` из KERNEL32.dll. При помощи неявного вызова данных функций, можно получить доступ к любой библиотечной функции незаметно для отладчика.

2.1.2. Замер времени выполнения

При отладке программы время выполнения инструкций многократно увеличивается, так как программа выполняется пошагово. Можно замерить время выполнения участка кода, и если время выполнения этого участка окажется сильно больше планируемого при штатной работе (например несколько секунд), то можно сделать вывод, что программа выполняется под отладчиком.

Есть несколько доступных способов замерить время выполнения участка программы:

- Использовать инструкцию `rdtsc`. Эта инструкция считывает текущее значение счетчика меток времени процессора (64-битный `MSR`) в регистры `EDX:EAX`. В регистр `EDX` загружаются старшие 32 бита `MSR`, а в регистр `EAX` младшие 32 бита. В коде такую проверку можно представить следующим образом:

```
rdtsc
xchg  esi, eax
mov   edi, edx
rdtsc
sub   eax, esi
subb  edx, edi
jne   _being_debugged
cmp   eax, elapsed ; elapsed содержит максимальное
                   ; время выполнения участка кода
jnbe  _being_debugged
```

- Использовать API операционной системы Windows:
  - Функции работы со временем `GetSystemTime` или `GetLocalTime`.

В этом случае придется переводить полученной время в 8-байтовое

число, чтобы можно было посчитать разницу между двумя значениями времени.

– Функции счетчики, которые возвращают количество миллисекунд (микросекунд) с момента запуска системы. Это соответственно функции `GetTickCount` и `QueryPerformanceCounter`. При использовании этих функций переводить время в число не требуется, что делает их более предпочтительными.

• Прочитать время с платы CMOS. Но, чтобы получить доступ к портам ввода/вывода, необходимо установить соответствующий драйвер.

Наилучшим вариантом из приведенного списка является использование инструкции `rdtsc`. Обращение к системным вызовам или портам ввода/вывода может быть слишком заметным при отладке.

2.1.3. Нахождение контрольной суммы участка кода

Реализации данного метода посвящена бóльшая часть этой работы. Суть метода заключается в том, чтобы посчитать контрольную сумму участка кода программы и занести в одну из переменных. При необходимости проверки наличия отладчика программа будет проверять соответствует ли текущее значение контрольной суммы кода сохраненному ранее значению.

Преимуществом данного метода является то, что он не только позволяет обнаружить отладчик, но и защищает программу от любых изменений вносимых в ее код. Также в работе будет рассмотрен способ нахождения контрольной суммы без обращения к системным вызовам, что дополнительно усложняет нахождение участка защиты.

Стоит отметить, что данный метод является наиболее сложным и трудозатратным в реализации. Рассмотрим проблемы и особенности, с которыми предстоит столкнуться при реализации данного метода.

2.2. Нахождение необходимой секции в памяти

Разрабатываемый метод будет вычислять контрольную сумму секции кода программы. Путем несложных изменений можно адаптировать метод так, чтобы он искал контрольную сумму секции данных, ресурсов или любой другой секции. Для этого необходимо найти начало и размер конкретной

секции в памяти. Чтобы решить эту задачи можно воспользоваться знаниями об устройстве PE формата.

Для нахождения нужной секции программа будет выполнять следующий алгоритм:

1. Прибавить к базовому адресу загрузки 60 и прочитать четыре байта по этому адресу. По смещению в шестьдесят байт от базового адреса загрузки находится относительный виртуальный адрес PE-заголовка.

2. Перейти к PE-заголовку, прибавив к базовому адресу прочитанные четыре байта.

3. Считать количество секций, которое содержится по смещению в 6 байт от PE-заголовка в двухбайтовом слове.

4. По смещению в 20 байт от PE-заголовка находится 2-байтовое число, в котором содержится размер опционального заголовка. Это значение нам нужно для того, чтобы перейти к разделу секций.

5. После этого необходимо перейти к опциональному заголовку, который находится по смещению в 24 байта от PE-заголовка.

6. По смещению в 96 байт от опционального заголовка находится массив 8-ми байтовых значений. Первые четыре байта содержат адрес секции, а вторые четыре байта содержат размер секции. Пятый элемент этого массива соответствует таблице базовых релокаций, которая нам понадобится при подсчете контрольной суммы. Эти значения необходимо сохранить.

7. После этого необходимо перейти к разделу секций, прибавив к началу опционального заголовка его размер, полученный на 4 этапе.

8. Раздел секций представляет собой массив структур, в которых находится информация о всех секциях, которые есть в программе. Необходимо циклом пройти по всем элементам данного массива, пока не встретится структура интересующей нас секции. Определить необходимую структуру можно по по полю характеристик секции. Так, например, секция кода содержит характеристику `IMAGE_SCN_CNT_CODE` (`0x00000020`).

9. При нахождении интересующей секции необходимо сохранить ее относительный виртуальный адрес и размер.

В итоге проделанных действий мы будем иметь относительные виртуальные адреса и размеры искомой секции и таблицы базовых релокаций.

2.3. Нахождение базового адреса загрузки

Чтобы найти расположение нужной секции, нам также нужно знать базовый адрес загрузки программы.

Самый очевидный способ получить базовый адрес загрузки — это вызвать соответствующую функцию Windows API. А именно, если вызвать функцию `GetModuleHandle` и в качестве параметра передать нулевой указатель, то функция вернет переменную типа `HMODULE`, значение которой будет соответствовать базовому адресу загрузки программы.

Для нашей задачи такой подход имеет недостаток в виде системного вызова. Так как отследить обращение к системным вызовам при помощи отладчика очень просто, такой подход может поставить защиту под угрозу.

Чтобы получить базовый адрес загрузки программы без обращения к функциям API Windows, обратимся к структуре PEB. PEB (process environment block) — это структура, которая содержит информацию о процессе, в памяти которого она хранится. Процесс может получить доступ к этой структуре при помощи регистра fs (для 64-битного процесса регистр gs). Получить адрес PEB можно, выполнив следующий код:

```
mov eax, fs:0x30
```

После чего в регистре `EAX` будет содержаться адрес PEB. По смещению в 8 байт в PEB хранится базовый адрес загрузки.

Таким образом можно получить базовый адрес загрузки без обращения к системным вызовам.

2.4. Таблица базовых релокаций

Как отмечалось ранее, операционная система Windows использует механизм ASLR. ASLR (Address space layout randomization) — это метод компьютерной безопасности, предназначенный для предотвращения использования уязвимостей, связанных с повреждением памяти. Данный прием может предотвратить простой переход злоумышленника, например, к конкретной функции в памяти. ASLR случайным образом упорядочивает позиции адресного пространства ключевых областей данных процесса, включая базовый адрес загрузки, позиции стека, кучи и загружаемых библиотек.

Из вышеизложенного следует, что адреса конкретных функций и переменных невозможно определить до запуска программы. Следовательно, должен быть механизм корректировки адресов в различных секциях программы. Например, необходимо изменить все абсолютные адреса из инструкций `jmp` в соответствии с конкретным базовым адресом.

В системе Windows для этого в каждом исполняемом файле, который поддерживает ASLR, есть таблица базовых релокаций. При помощи этой таблицы загрузчик приложений Windows может скорректировать участки программы в соответствии с адресом загрузки.

Таблица базовых релокаций содержит записи для всех исправлений в образе программы. Общий размер таблицы содержится в опциональном заголовке. Вся таблица разбита на блоки исправлений. Каждый блок содержит исправления для страницы размера 4 Кбайт и выровнен по 32-битной границе.

Каждый блок исправлений начинается со структуры описанной в таблице 1 в приложении 3.3.

После описанной структуры следуют поля таблицы. Каждое поле занимает 2 байта и имеет структуру, описанную в таблице 2 в приложении 3.3.

2.5. Отказ от классических функций

Реализуемый метод защиты подразумевает проверку контрольной суммы в различных местах работы программы. В классическом подходе подобные задачи решаются вынесением повторяющегося участка кода в отдельную функцию. Но конкретно в нашем случае такой подход имеет ряд недостатков.

Функция представляет собой участок памяти с кодом, в который передается управление при вызове данной функции. То есть, если программа в двух разных местах вызовет одну и ту же функцию, то процессор в обоих случаях передаст управление одному и тому же участку кода. Злоумышленнику будет достаточно изменить код в одном месте программы (в функции, осуществляющей защиту), чтобы обойти все проверки.

Бóльшую безопасность предоставляют встроенные (inline) функции. При вызове встроенной функции компилятор подставляет код данной функции на место каждого вызова. Вставка происходит только в том случае, если анализ затрат и преимуществ компилятора показывает, что это целесооб-

разно. Другими словами, компилятор может игнорировать ключевое слово `inline`. В компиляторе MSVC есть директива `__forceinline`, которая позволяет сделать функцию встраиваемой независимо от анализа компилятора.

В данной работе защита будет представлена в виде макроса, а не встроенной функции, так как это предоставит еще более широкие возможности по сокрытию кода, о которых написано в третьей главе.

ГЛАВА 3. РЕАЛИЗАЦИЯ

Данная глава посвящена реализации выбранного метода защиты от отладчика. А именно написанию макроса, реализующего нахождение контрольной суммы секции кода. Помимо макроса также были написаны вспомогательные модули, которые могут применяться при тестировании и отладке применяемой защиты. Принцип их работы также освещается в данной главе.

3.1. Алгоритм нахождения контрольной суммы

Реализуемый метод защиты заключается в том, что программа подсчитывает контрольную сумму участка кода и сравнивает с эталонным значением. Если контрольные суммы не совпадают, программа должна изменить свое поведение с учетом того, что она находится по угрозой взлома.

Алгоритм нахождения контрольной суммы должен быть быстрым, чтобы по задержке нельзя было определить место в программе, где осуществляется проверка. Также алгоритм не должен хранить в памяти большое количество промежуточных данных, так как это тоже упрощает нахождение блока защиты.

Исходя из этих условий был разработан алгоритм, блок-схема которого представлена на рисунке 3.1. Данный, алгоритм основан на том факте, что поля в таблице релокаций отсортированы по возрастанию. Алгоритм состоит из главного цикла, в котором суммируется каждый байт секции, кроме тех случаев, когда адрес проверяемого байта есть в таблице релокаций (в этом случае этот байт и три следующих за ним игнорируются). Если при проверке окажется, что адрес проверяемого байта больше, чем значение поля таблицы релокаций, то берется следующее поле. Таким образом полученный алгоритм имеет линейную сложность.

В данном подходе игнорируется указанный в поле таблицы релокации тип смещения. Это сделано для того, чтобы код алгоритма подсчета контрольной суммы имел минимальный размер.

Также перед началом алгоритма необходимо найти и сохранить в памяти адрес начала проверяемой секции, размер проверяемой секции, адрес, по которому размещена таблица базовых релокаций, и размер таблицы. В реализованном алгоритме данные значения хранятся в стеке процесса.

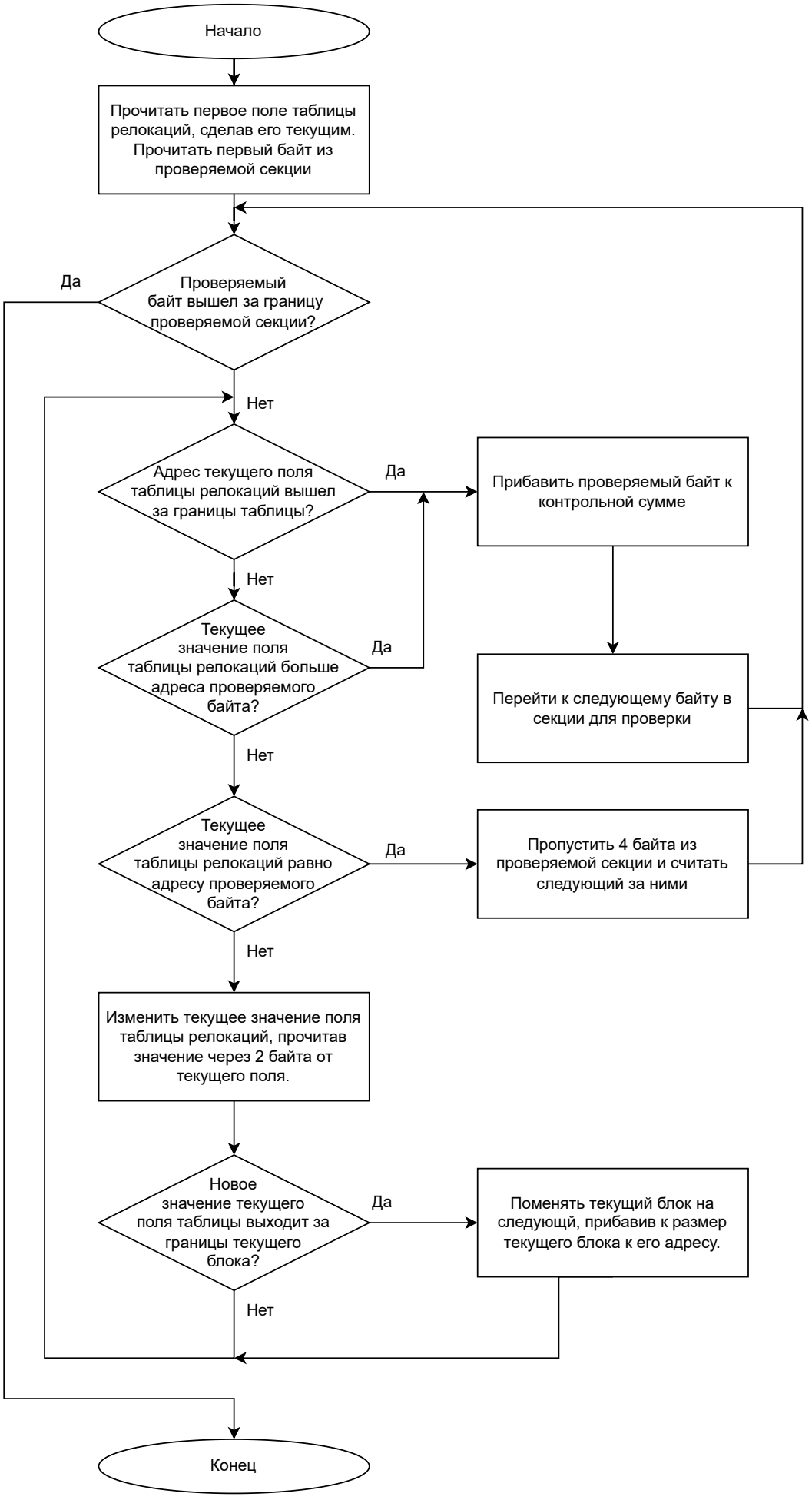


Рис. 3.1 Блок-схема алгоритма по нахождению контрольной суммы

3.2. Реализация алгоритма

В данной работе приведенный алгоритм реализован на языке ассемблера. Сам ассемблерный код реализован в виде макроса. Преимуществом такого подхода является то, что при вызове функции-макроса происходит не вызов функции, а подстановка кода функции на место вызова. То есть это работает аналогично встраиваемым функциям. Однако, если попытаться сделать функцию, содержащую код алгоритма, встраиваемой, то компилятор проигнорирует ключевое слово `inline`. Для компилятора MSVC данная проблема решается ключевым словом `__forceinline`, но, например, если поменять компилятор на gcc, то такое решение уже не подойдет. Для обеспечения универсальности кода по отношению к компилятору реализация алгоритма помещена в макрос.

Использование макроса предоставляет еще одну полезную возможность, а именно указать при вызове разный порядок регистров. Таким образом, вызывая макрос с разным порядком регистров, будут получаться разные участки кода. Если злоумышленник найдет и исправит один блок защиты, то найти остальные путем поиска повторяющихся элементов памяти у него не получится.

Объявление макроса выглядит следующим образом:

```
#define GET_CRC(reg_A, reg_B, reg_C, \
                reg_D, reg_SI, reg_DI, out_var) ...
```

В качестве регистра при вызове макроса необходимо передать букву, которая его обозначает. Для `eax` необходимо передать `a`, для `ebx` — `b` и так далее. Только для регистров `esi` и `edi` необходимо передать `si` и `di` соответственно. Это связано с тем, что при работе с этими регистрами невозможно обратиться к их младшему байту.

Внутри макроса имена регистров соединяются при помощи псевдооперации ##. Так, например, строчка кода

```
mov al, byte ptr [ebx]
```

принимает вид

```
mov reg_A##l, byte ptr [e##reg_B##x]
```

При оформлении кода ассемблера в виде макроса приходится учитывать следующие правила:

а. Заключить код в `__asm{...}` блок.

б. Поместить ключевое слово `__asm` перед каждой ассемблерной инструкцией.

в. Использовать только блочные комментарии (`/*...*/`).

Эти правила обусловлены тем, что при подстановке макроса он всегда занимает только одну строку.

Первое правило необходимо соблюдать, чтобы макрос можно было использовать в одной строке с другим кодом языка C или C++. Без закрывающей фигурной скобки компилятор не сможет понять, где заканчивается ассемблерный код, и будет воспринимать код C или C++ как ассемблерные инструкции.

Второе правило обусловлено тем, что ключевое слово `__asm` и перевод строки являются единственными разделителями операторов в ассемблерных вставках. Из-за того, что подстановка макроса происходит в одну строку, из доступных разделителей операторов остается только ключевое слово `__asm`.

Третье правило ограничивает использование строчных комментариев, так как первый строчный комментарий сделает всю оставшуюся часть макроса комментарием.

Также при разработке макроса пришлось столкнуться с проблемой невозможности установки меток внутри ассемблерной вставки. Чтобы установить метку, необходимо закрыть блок ассемблерной вставки, установить метку и открыть новую ассемблерную вставку.

Ниже приведен пример корректного макроса с ассемблерной вставкой.

```
#define EXAMPLE_MACRO(out_var) __asm \
{ \
  __asm mov eax, value    /* Place value in eax */ \
  __asm cmp eax, svalue   /* Compare first and second value */ \
  __asm je l1 \
  __asm mov out_var, eax \
  __asm jmp end \
} \
```

```
l1: \
__asm { \
  __asm add eax, 0x200    /* Increase eax by 200 */ \
  __asm mov out_var, eax \
} \
end:
```

3.3. Набор классов для нахождения контрольной суммы

Помимо макроса для нахождения контрольной суммы в ходе работы были реализованы модули, обеспечивающие нахождение контрольной суммы различных секций программ, загруженных в память и расположенных на диске.

Модули представляют из себя набор из трех классов, написанных на языке C++, который состоит из:

- `CRC_general`— базовый абстрактный класс, в котором реализованы основная логика работы с секциями и алгоритм нахождения контрольной суммы.
- `CRC_exe` — унаследованный от `CRC_general` класс, реализующий работу с исполняемыми файлами, расположенными на диске.
- `CRC_image` — унаследованный от `CRC_general` класс, реализующий работу с исполняемыми файлами, загруженными в оперативную память.

Различия в работе с секциями исполняемых файлов, один из который хранится на диске, а другой загружен в память, заключаются в разных значениях смещений внутри кода. В таблице секций PE-формата для каждой секции содержится два поля: `VirtualAddress` и `PointerToRawData`. В поле `VirtualAddress` содержится относительный виртуальный адрес секции при загрузке программы в памяти. В свою очередь, поле `PointerToRawData` содержит смещение относительно начала *файла*, по которому расположена данная секция.

Еще одно отличие заключается в том, что при работе с процессом, расположенном на диске, базовым адресом загрузки нужно считать начало файла. При этом необходимо учитывать, что адреса, вычисляемые по таблице релокаций, соответствуют смещению секций не внутри файла, а внутри

программы загруженной в память.

Реализованные модули следует использовать не для организации защиты приложения, а для скорее для отладки. В связи с этим реализованный в них алгоритм отличается от представленного на рисунке 3.1. Для нахождения контрольной суммы все адреса из таблицы релокаций заносятся в ассоциативный контейнер (`std::set`). При подсчеты контрольной суммы, если адрес проверяемого байта содержится в этом контейнере, то этот и следующие за ним три байта игнорируются. Также данные классы позволяют находить контрольную сумму любых секций. Для этого им необходимо передать битовую маску, соответствующую характеристикам секции.

Также в данных классах реализована система логирования, настраиваемая с помощью директив препроцессора. Так, например, можно указать файл, в который будут записаны все значения из таблицы релокаций, или все значения байт с их адресам, которые вошли в контрольную сумму.

Класс `CRC_exe` реализует еще одну важную функцию. Защищаемая программа после нахождения своей контрольной суммы должна сравнить ее с эталонным значением. Эталонное значение должно хранится в другой секции данной программы. Класс `CRC_exe` позволяет записать найденное значение контрольной суммы в файл расположенный на диске.

Защищаемая программа должна разместить в своей памяти ключ размером в четыре байта. Если защищаемая программа будет проверять целостность своей секции кода, то ключ можно расположить в секции инициализированных данных (.data). Сделать это можно путем объявления глобальной переменной, проинициализировав ее значением ключа. Так как в коде программы значение данной переменной меняться не будет, ее следует пометить как `volatile`. Иначе компилятор может попытаться оптимизировать обращение к данной переменной, и просто подставит значение данной переменной в код программы, вместо того чтобы читать это значение из адреса памяти. Пример объявления такой переменной:

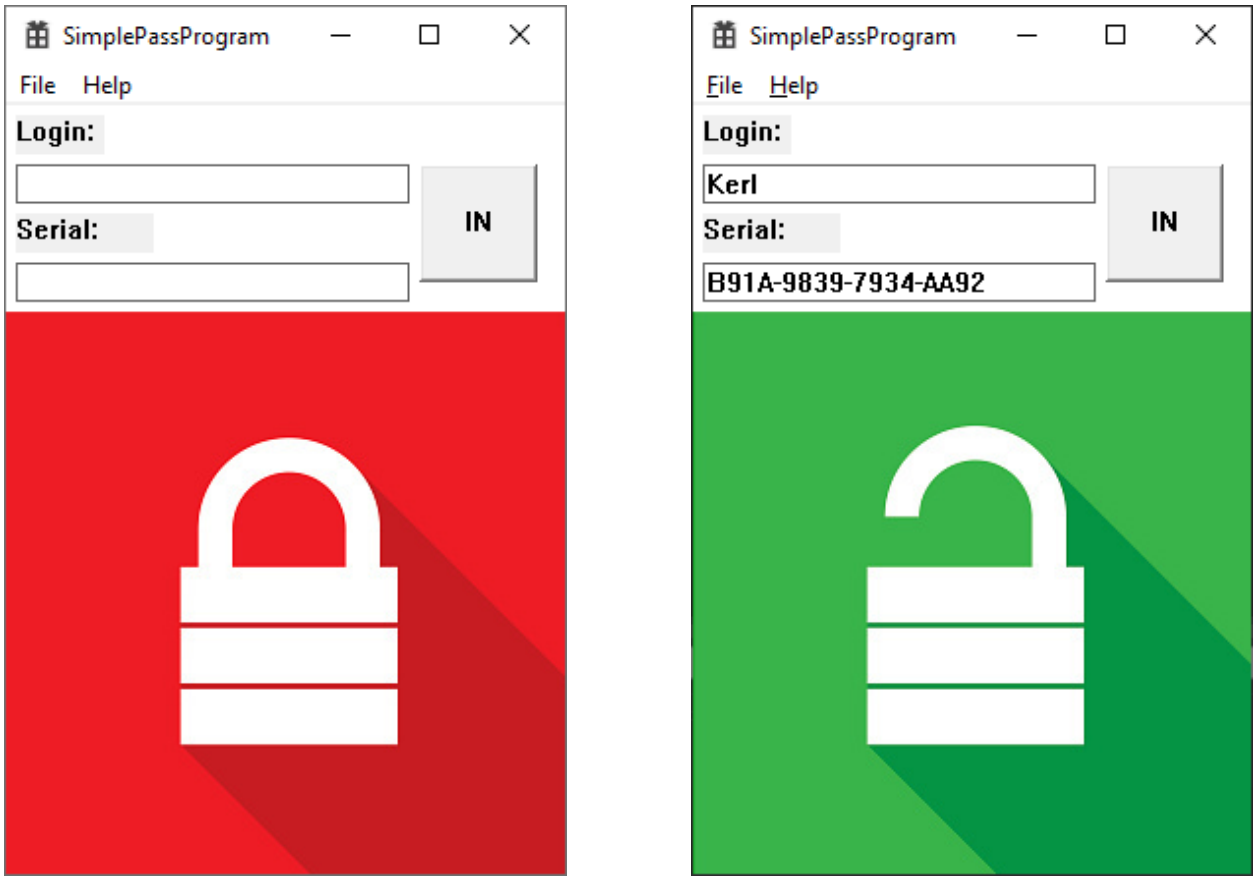
```
volatile uint32_t CRC = 0x4C52454B;
```

Класс `CRC_exe` позволяет записать значение контрольной суммы выбранной секции во все места, где в программе встретится ключевое значение. Для этого можно применить следующий код:

```
CRC_exe crc_finder;
uint32_t crc;
crc_finder.read_file("file_name.exe");
crc = crc_finder.get_sections_CRC(IMAGE_SCN_CNT_CODE);
crc_finder.replace_all_keys(KEY, crc);
crc_finder.write_file("new_file_name.exe");
```

ГЛАВА 4. ТЕСТИРОВАНИЕ

Для проведения тестов, была написана простая демонстрационная программа. Программа написана на языке C с использованием WinAPI. Ее интерфейс (рисунок 4.1) представляет из себя два поля `edit` для логина и для серийного ключа. Серийный ключ генерируется из имени пользователя в процессе работы программы.



а. б.

Рис. 4.1 Интерфейс демонстрационной программы. а. — заблокированное состояние; б. — разблокированное состояние

В случае, если введенный пользователем ключ не совпадает с тем, который сгенерировала программа, выводится окно сообщения, уведомляющее пользователя о неудаче. Данная уязвимость допущена здесь специально для большей наглядности примера.

Для взлома данной программы воспользуемся отладчиком OllyDBG. Изменим исходный код программы так, чтобы пользователь мог осуществить вход, введя любой пароль.

Для этого в памяти программы была найдена строка, которая выводится в окне сообщения при неверном вводе ключа. А именно: "Invalid activation code". Проследим по коду, предшествующему обращению к этой строке, условные переходы, которые приводят к ситуации, когда программа остается заблокированной. Таким образом легко обнаружить проверку двух строк на равенство. Очевидно в этом месте происходит проверка ключа. На рисунке 4.2 видно, что по адресу `00FF143F` располагается проверка признака корректности ключа, после чего идет условный переход на секцию кода, которая приводит к блокировке приложения.

Рис. 4.2 Окно отладчика при взломе программы без защиты

Так как наличие в регистре `eax` значения отличного от нуля свидетельствует о том, что введенный ключ не является корректным, оптимальным вариантом является замена инструкции:

```
test eax, eax
```

на инструкцию

```
xor eax, eax
```

Таким образом в регистр `eax` будет занесено значение 0, как и требует того алгоритм программы при вводе корректного ключа. А условный переход выполняться не будет .

Рис. 4.3 Работа взломанной программы

На рисунке 4.3 представлена работа полученной таким образом программы.

Теперь разместим в коде нашей программы разработанную ранее защиту от отладчика. Для этого создадим глобальную переменную:

```
static const volatile DWORD CRC = 0x4C52454B;
```

При помощи реализованных модулей вторая программа занесет в эту переменную значение, равное контрольной сумме секции кода. Защиту разместим в участке кода, который выполняется при нажатии на кнопку `IN`. Если полученная контрольная сумма не будет совпадать с эталонным значением, то будет выведено диалоговое окно, сообщающее, что целостность кода была нарушена, а проверка пароля выполняться не будет.

Стоит отметить, что уведомление пользователя об обнаружении отладчика или нарушении целостности программы само по себе является уязвимостью. Такое поведение реализовано исключительно в целях наглядности.

Исходный код программы претерпел незначительные изменения (рисунок 4.4). Признак корректности ключа заносится в регистр `ecx` по адресу `008B147B`.

Рис. 4.4 Окно отладчика при взломе программы с защитой

Для взлома программы заменим инструкцию

```
or ecx, 1
```

на инструкцию

```
xor ecx, ecx
```

Так как полученная инструкция занимает на один байт меньше памяти, освободившееся место заполним инструкцией `nop`. Сохраним полученную программу и попробуем ввести некорректное значение ключа.

Как видно из рисунка 4.5, разработанная система обеспечивает достойную защиту приложения.

Также стоит отметь, что если в отладчике установить точку останова, то он занесет в первый байт инструкции значение `CC`. Таким образом контрольная сумма кода изменится и разработанный алгоритм позволяет это обнаружить. Данная особенность защиты проиллюстрирована на рисунке 4.6.

Также в ходе тестирования было выявлено, что при передачи различных регистров в вызов макроса максимальная длина повторяющихся байт равна 9:

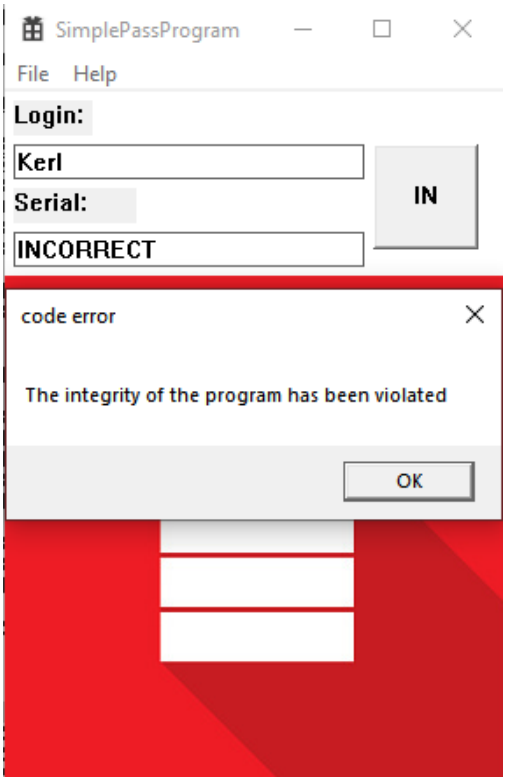


Рис. 4.5 Работа программы с защитой от взлома

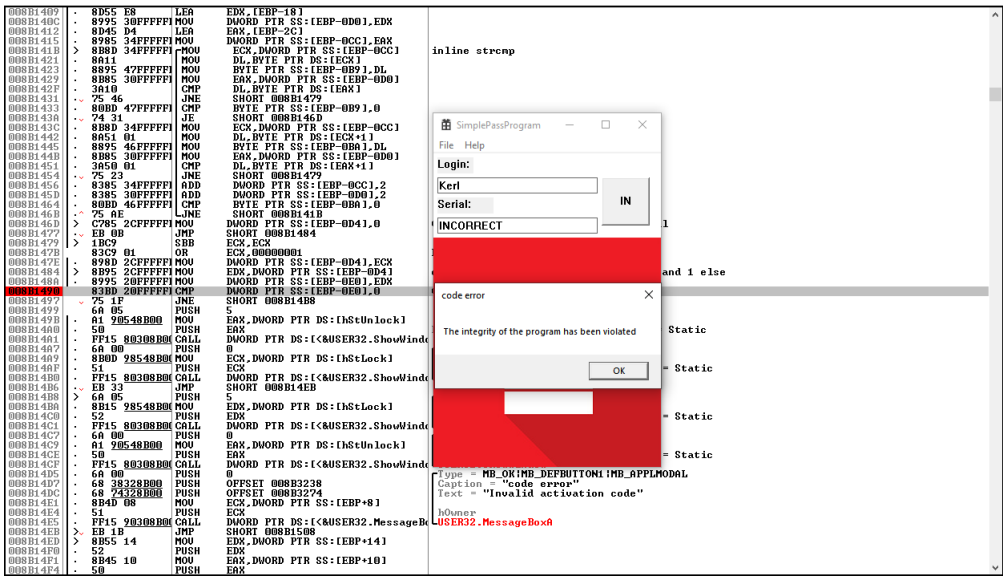


Рис. 4.6 Обнаружение системой защиты установленной точки останова

```
0F 84 8B 00 00 00 EB EA FF
```

Данные байты операции соответствуют двум инструкциям:

```
je  no_section
jmp section_loop
```

Если повторяющаяся последовательность длинной в 9 байт покажется критичной, можно вставить между этими инструкциями любую операцию с каким-либо регистром. Например:

```
xor ebx, ebx
```

Результаты тестирования показывают, что реализованный метод на должном уровне обеспечивает защиту программы от модификации исходного кода, а также препятствует процессу отладки.

Заключение

В результате выполнения работы была разработан метод защиты программы от изменения ее исходного кода, который также препятствует процессу отладки. Был разработан и представлен алгоритм нахождения контрольной суммы кода защищаемой секции. Составленный алгоритм был реализован и протестирован. Результаты тестирования показали, что разработанный метод обеспечивает защиту на достойном уровне.

Алгоритм был реализован на языке ассемблера, так как была необходимость иметь полный контроль над получающимся машинным кодом. Реализация на языке ассемблера позволила достичь независимости реализованного метода от компилятора.

Ассемблерный код был помещен в макрос языка C, что в свою очередь позволило решить две проблемы. Во-первых, код защиты можно вставлять в тело программы простым вызовом функции-макроса, но при этом в машинном коде будет не вызов функции с передачей управления, а подстановка кода защиты. Во-вторых, можно переназначать регистры процессора используемые при нахождении контрольной суммы.

Помимо этого в ходе работы были написаны программные модули на языке C++, которые обеспечивают удобную работу с секциями PE-файла, нахождением контрольной суммы для различных секций, а также замену заданной последовательности байт в файле.

Для тестирования полученного механизма была написана демонстрационная программа на языке C. В ходе тестирования механизм защиты смог обнаружить изменения в исходном коде программы, а также оказать затруднение процессу отладки. При этом код механизма защиты обладает незаметностью относительно остального кода программы.

Приложение А

(обязательное)

Таблицы с описанием структур из секции релокаций

Таблица 1

Структура блока исправлений

Смещение	**Размер**	**Поле**	**Значение**
0	4	Относительный виртуальный адрес страницы	Базовый адрес загрузки и относительный виртуальный адрес страницы прибавляется к каждому смещению в таблице, чтобы получить виртуальный адрес по которому необходимо провести исправление
4	4	Размер блока исправлений	Общее количество байтов, занимаемых блоком исправлений, включая эту структуру.

Таблица 2

Структура поля таблицы базовых релокаций

Смещение	**Размер**	**Поле**	**Значение**
0	4 бита	Тип	Значение, размещенное в старших четырех битах слова, указывает на тип исправления. Всего типов исправлений может быть 16. Каждый тип указывает, как провести коррекцию значения в памяти.
0	12 бит	Смещение	Значение, размещенное в младших двенадцати битах слова, указывает смещение относительно относительного виртуального адреса страницы. Это смещение указывает место, где необходимо произвести коррекцию.

Приложение В

(обязательное)

Листинг макроса для нахождения контрольной суммы

```
#define GET_CRC(reg_A, reg_B, reg_C, reg_D, reg_SI, reg_DI, out_var) __asm \
{ \
    __asm mov e##reg_A##x, hInst /* base load address in eax ## */\
    __asm mov e##reg_B##x, e##reg_A##x\
    __asm add e##reg_B##x, 60 /* sixty is e_lfanew offset */\
    __asm mov e##reg_B##x, [e##reg_B##x] /* e_lfanew value is in ebx*/\
    __asm add e##reg_A##x, e##reg_B##x /* pe header is in eax */\
    __asm xor e##reg_C##x, e##reg_C##x \
    __asm xor e##reg_SI, e##reg_SI \
    __asm mov reg_C##x, [e##reg_A##x + 6] /* number of sections is in ecx */\
    __asm mov reg_SI, [e##reg_A##x + 20] /* Size of optional header is in esi */\
    __asm add e##reg_A##x, 24 /* Optional pe header is in eax */\
    __asm mov e##reg_B##x, e##reg_A##x \
    __asm add e##reg_B##x, 96 /* DataDirectory is in ebx */\
    __asm add e##reg_B##x, 5 * 8 /* Base relocation table field is in ebx */\
    __asm push[e##reg_B##x] /* Add reloc rva in stack */\
    __asm push[e##reg_B##x + 4] /* Add reloc size in stack */\
    __asm add e##reg_A##x, e##reg_SI /* Start of section table in eax */\
    __asm mov e##reg_SI, 1 \
    __asm mov e##reg_D##x, e##reg_A##x \
}\
sections_loop: \
__asm {\
    __asm mov e##reg_B##x, [e##reg_D##x + 36] /* Characteristics of section is in ebx */\
    __asm and e##reg_B##x, 0x20 /* if ebx is not zero, then the section is being executed */\
    __asm jnz section_found  \
    __asm add e##reg_D##x, 40 /* in edx next section table */\
    __asm inc e##reg_SI /* in esi number off current section */\
    __asm cmp e##reg_SI, e##reg_C##x /* if esi and ecx are equal, then there isn't code section */\
    __asm je no_section \
    __asm jmp sections_loop \
}\
section_found: \
__asm {\
    /* In edx code section table /* */\
```

```
36      __asm push [e##reg_D##x + 12] /* Add code section rva in stack */\
37      __asm push [e##reg_D##x + 8] /* Add code section rva in stack */\
38      /*============= Main algorithm =============*/\
39      __asm mov e##reg_DI, [esp + 12] /* reloc rva is in edi */\
40      __asm add e##reg_DI, hInst /* reloc address is in edi. So edi
    conatains current reloc block */\
41      __asm mov e##reg_SI, e##reg_DI /* esi contains current block */\
42      __asm add e##reg_SI, 8 /* esi contains current rf */\
43      __asm xor e##reg_C##x, e##reg_C##x /* In ecx will be iterator for main
     loop */\
44      __asm xor e##reg_D##x, e##reg_D##x /* In edx will be checksum */\
45  }\
46  main_loop: \
47  __asm {\
48      __asm cmp e##reg_C##x, [esp] /* Check is iterator less than code
    section size */\
49      __asm jge main_loop_end \
50  }\
51  find_reloc_loop: \
52  __asm{ \
53      /* Check is current rf in relocation table */\
54        __asm mov e##reg_B##x, [esp + 12] /* Reloc section rva in edx */\
55        __asm add e##reg_B##x, [esp + 8] /* rva of end address reloc table
    is in edx */\
56        __asm add e##reg_B##x, hInst /* end address reloc table is in edx */
    \
57        __asm cmp e##reg_SI, e##reg_B##x \
58        __asm jge end_reloc_loop \
59      /* Check equals current rf checked byte */\
60        __asm mov reg_A##x, [e##reg_SI] /* Place 2 bytes reloc in eax */\
61        __asm and e##reg_A##x, 0xFFF /* Remove type of relocation */\
62        __asm add e##reg_A##x, [e##reg_DI] /* Add section rva to value from
    reloc field. Reloc address is in eax */\
63        __asm mov e##reg_B##x, [esp + 4] /* code section rva in ebx */\
64        __asm add e##reg_B##x, e##reg_C##x /* Current checked byte rva in
    ebx */\
65        __asm cmp e##reg_A##x, e##reg_B##x /* Compare checked byte and
    current relocation */\
66        __asm jg  checked_is_not_in_reloc \
67        __asm je  checked_in_reloc \
68      /* Check is current rf in current block */\
69        __asm mov e##reg_B##x, e##reg_DI /* reloc block is in edx */\
70        __asm add e##reg_B##x, [e##reg_B##x + 4] /* end address of
    relocation block is in edx */\
71        __asm cmp e##reg_SI, e##reg_B##x /* compare current rf with end
    address of relocation block */\
```

```
72      __asm jl in_old_block \
73      __asm add e##reg_DI, [e##reg_DI + 4] /* Switch to next reloc block
    */\
74      __asm lea e##reg_SI, [e##reg_DI + 6] /* Current rf in new block */\
75 }\
76 in_old_block: \
77 __asm{ \
78      __asm add e##reg_SI, 2 /* Next rf */\
79      __asm jmp find_reloc_loop \
80 }\
81 checked_is_not_in_reloc: \
82 __asm{ \
83      __asm xor e##reg_A##x, e##reg_A##x /* if checked byte isn't in reloc
    , then eax contains 0 */\
84      __asm jmp end_reloc_loop \
85 }\
86 checked_in_reloc: \
87 __asm{ \
88      __asm mov e##reg_A##x, 1 /* if checked byte is in reloc, then eax
    contains 1 */\
89      __asm jmp end_reloc_loop /* TODO delete this instruction */\
90 }\
91 end_reloc_loop: \
92 __asm{ \
93     __asm test e##reg_A##x, e##reg_A##x  \
94     __asm jnz skip_byte \
95     __asm mov e##reg_B##x, hInst /* Place hInst in ebx */\
96     __asm add e##reg_B##x, [esp + 4] /* Add code section rva to hInst */\
97     __asm add e##reg_B##x, e##reg_C##x /* Add iterator in ebx. So ebx
    contains address of checked byte */\
98     __asm xor e##reg_A##x, e##reg_A##x \
99     __asm mov reg_A##l, byte ptr [e##reg_B##x] /* In eax low byte contains
     byte from code section for adding in crc */\
100    __asm add e##reg_D##x, e##reg_A##x /* Add value of checked byte to CRC
     */\
101    __asm inc e##reg_C##x /* Increment iterator */\
102    __asm jmp main_loop \
103 }\
104 skip_byte: \
105 __asm{ \
106    __asm add e##reg_C##x, 4 \
107    __asm jmp main_loop \
108 }\
109 main_loop_end: \
110 __asm{ \
111    __asm mov out_var, e##reg_D##x \
```

```
112     __asm add esp, 2*4 \
113 }\
114 no_section: \
115 __asm{ \
116     __asm add esp, 2*4 /* clear stack */\
117 }
```