УДК 004.492.3

АЛГОРИТМ ОБНАРУЖЕНИЯ И ОБХОДА АНТИОТЛАДОЧНЫХ И АНТИЭМУЛЯЦИОННЫХ ПРИЕМОВ

А. Е. Антонов,
аспирант
А. С. Федулов,
доктор техн. наук, профессор
Смоленский филиал Московского энергетического института (технического университета)

Проведен обзор принципов работы отладчиков и эмуляторов, их уязвимостей. Предложен новый алгоритм обнаружения антиотладочных и антиэмуляционных приемов, а также модификация отладчика для задач анализа вредоносного кода.

Ключевые слова *— эмулятор, отладчик, антиотладочные и антиэмуляционные приемы, вредоносное программное обеспечение.*

Введение

Одним из эффективных методов анализа вредоносного программного обеспечения (ПО) является выполнение кода в среде отладчика или эмулятора. Отладчики обычно используются как инструмент вирусного аналитика, в свою очередь эмуляторы встраиваются в антивирусные продукты для автоматического поиска опасного или нежелательного ПО. Действительно, ряд задач анализа вредоносного кода, например поиск полиморфных вирусов, можно решить, только запустив программный код на выполнение и отслеживая результаты его работы. Чтобы препятствовать работе отладчиков и эмуляторов, авторами вредоносного кода разработан ряд методов. В настоящее время поиск и деактивация таких методов может быть эффективно проведена только в ручном режиме при тщательном анализе алгоритма защиты. В работе предложен алгоритм для обнаружения защищенного от отладчиков и эмуляторов кода. Предлагаемый алгоритм не только облегчает поиск антиотладочных и антиэмуляционных приемов, но и может работать в автоматическом режиме (без участия аналитика).

Принципы работы и уязвимости отладчика и эмулятора

Отладчик — приложение, которое либо перехватывает окружение отлаживаемой программы во время ее выполнения, либо исполняет ее в виртуальной машине, таким образом помогая находить ошибки. Отладчик позволяет контролировать окружение выполнения (например, память), в котором функционирует отлаживаемая программа [1, с. 627]. В дальнейшем под отладчиком мы будем понимать приложение, работающее в том же окружении, что и отлаживаемая программа, и использующее отладочные возможности процессора и операционной системы (ОС).

Существуют две разновидности отладчиков — пользовательского режима и режима ядра [2, с. 143]. Первые в большинстве своем используют функции ОС, например интерфейс отладки *Debug API* (*Application Program Interface*) ОС Windows, вторые — непосредственно возможности отладки процессоров. Далее в работе рассматривается архитектура процессора x86 и ОС Windows как наиболее распространенные.

Любой отладчик должен обеспечивать трассировку приложения и установку точек останова. Трассировка обычно задается выбором специального режима процессора (при котором после каждой инструкции управление передается отладчику). Существует три типа точек останова [3]: аппаратные — используют специальные отладочные регистры, программные — вставляют в выполняемый код специальные команды останова (int 3) и на доступ к памяти — изменяют атрибут доступа к странице памяти и при обращении к ней производят обработку.

Для защиты от отладчика программа может использовать следующие методы [4, с. 226]:

1) проверку регистров, флагов процессора, отвечающих за аппаратную отладку, например проверку того, установлен ли флаг трассировки (*Trace Flag — TF*), а также изменение этих флагов и регистров, что может вести к аварийному завершению отладчика;

2) проверку целостности кода для защиты от программных точек прерывания;

3) поиск структур отладчика в памяти, например в контексте потока (*Thread Infomation Block*) — для этой цели можно использовать функцию *IsDebuggerPresent* (*Debug API* ОС Windows);

4) поиск конкретных отладчиков в системе.

В основе эмулятора лежит другой принцип работы. Эмулятор — это система, имитирующая работу процессора, оперативной памяти, аппаратного обеспечения и ОС [5, с. 279]. В общем случае эмулятор может имитировать ту же систему, на которой он запущен. Таким образом, его можно использовать при анализе вредоносного ПО, поскольку программа, исполняемая в эмуляторе, не может влиять на реальную систему [5]. Далее мы будем рассматривать эмуляторы для имитации выполнения программ переносимого формата исполняемых файлов (*Portable Executable*), работающих в ОС Windows.

Эмулятор загружает программный код в буфер и, читая последовательно инструкции, имитирует их выполнение. При этом все изменения происходят в переменных эмулятора (виртуальные регистры, виртуальный стек, буфер с программой), а не на реальной машине. Процесс имитации проходит в несколько этапов [6]:

- анализ — выявляет размер, параметры имитируемой инструкции;
- пересчет адресов — должен быть выполнен для всех инструкций, обращающихся к памяти, поскольку нет практической возможности загрузить программу в буфер по адресу, который бы она имела, исполняясь на реальной машине;
- проверка адресов — выявляет ошибки адресации;
- проверка дополнительных условий (например, для операции div — деление — эмулятор проверяет неравенство операнда нулю);
- непосредственно имитация инструкции, при которой эмулятор согласно коду инструкции меняет состояние виртуальных регистров, виртуального стека и буфера с кодом и данными программы;
- перевод указателя инструкций (*Extended Instruction Pointer — EIP*) на следующую команду.

Принципиально алгоритм работы эмулятора имеет два существенных недостатка: значительное время имитации команды по сравнению с ее выполнением на реальной машине и сложность правильного имитирования всевозможных инструкций процессора и функций ОС. Для полной эмуляции ОС фактически необходимо переписать все ее функции заново. Исходя из этих недостатков программа может предпринять следующие атаки на эмулятор, чтобы обнаружить его или прекратить эмуляцию [4]:

- использовать неизвестные эмулятору команды процессора (например, MMX, SSE, SSE2 или недокументированные инструкции). Не зная текущую команду, эмулятор не сможет продолжить выполнение кода;
- использовать команды, реализованные в эмуляторе, с ошибками (если команда выполняется ошибочно, то последующая работа программ будет некорректна);
- использовать неизвестные эмулятору возможности системы (имитировать все API-функции ОС Windows практически невозможно);
- реализовать длинные циклы, вычисляющие какой-либо параметр для исполняемого кода. На реальной машине такой код будет выполняться достаточно быстро, а эмуляция подобных циклов займет значительное время (для имитации одной команды процессора требуется выполнить десятки или даже сотни инструкций).

Антиотладочные и антиэмуляционные приемы в значительной мере распространены во вредоносном ПО. На данный момент для их обнаружения может быть использован сигнатурный анализ для поиска подозрительных мест по известным сигнатурам. Также в некоторых случаях эмулятор или отладчик может сообщить о потенциальном антиотладочном приеме. Эмулятор, например, может встретить неизвестную команду, которую расценит как попытку антиэмуляции. Отладчик может дополнительно проверять, не сбросила ли программа флаги и регистры отладки. Однако описанные методы не способны находить новые антиотладочные приемы, неизвестные на момент создания отладчика или эмулятора, что во многом затрудняет анализ вредоносного ПО.

Алгоритм обнаружения новых антиэмуляционных и антиотладочных приемов

Как видно из приведенного выше описания, методы защиты от отладчиков и эмуляторов используют различные механизмы. На основании этого предлагается алгоритм поиска новых антиотладочных и антиэмуляционных приемов, заключающийся в сравнении работы одного и того же кода в отладчике и эмуляторе. Действительно, приемы для обхода отладчика во многих случаях будут успешно выполняться в среде эмулятора. И наоборот, код, который невозможно ис-

полнить в эмуляторе, будет корректно работать в отладчике.

Рассмотрим особенности алгоритма для поиска антиотладочных и антиэмуляционных приемов. На каждом шаге работы выполняется одна инструкция в среде эмулятора и отладчика и сравнивается результат ее выполнения. Инструкции процессора x86 могут влиять на регистры (в том числе на регистр флагов и на EIP), а также на содержимое оперативной памяти.

В простейшем случае нам достаточно сравнивать указатель инструкций. Его расхождение будет означать, что программа исполняется по разным веткам алгоритма, что в свою очередь говорит об ошибочности работы отладчика или эмулятора. Однако легко сформировать атаку против такого отладчика-эмулятора (листинг 1).

Листинг 1. *Алгоритм обнаружения отладчика-эмулятора, проверяющего только значение EIP.*

```
EAX = 1, если присутствует отладчик, иначе 0
ECX = 1, если обнаружен эмулятор, иначе 0
EAX = EAX + ECX
если EAX<>0, обнаружен отладчик или эмулятор
```

Следовательно, необходимо также сравнение и регистров общего назначения, и регистра флагов на каждом шаге. Для частных случаев возможно организовать обман предложенного метода, при котором значения регистров (в том числе регистра флагов и EIP) будут одинаковы, а различаться будут только состояния памяти (листинг 2). Поэтому для надежности и отсечения подобных вариантов следует производить сравнение содержимого памяти после некоторых, изменяющих ее, инструкций. Например, к «опасным» инструкциям можно отнести те, которые не содержат в качестве операнда регистры (stos, lods, movs, cmps и ряд других), а также сложные инструкции, которые могут быть реализованы в эмуляторе не полностью или с ошибками.

Листинг 2. *Пример обмана эмулятора-отладчика, проверяющего целостность кода и работоспособность инструкции add.*

```
Start:
    mov ecx,End-Start+4 ; +4, чтобы захватить переменную for_add
    mov edi,Start
    mov esi, DataCopy
    add dword [for_add],0FFFFFFFFh ; for_add = 0
    ;Предполагается, что инструкция
    ;ADD реализована в эмуляторе с ошибкой
    ;и в случае переполнения операнда не производит сложения.
    repe cmpsb ; Сравнивать байты по адресам EDI и ESI,
                ; пока они равны
    jnz EmulatorDebuggerDetected
    jmp NotDetected
End:
    for_add dd 1
CodeCopyStart:
    Точная копия байт от Start до End, для сравнения.
CodeCopyEnd:
    for_add_test dd 0
```

К достоинствам алгоритма для поиска антиотладочных и антиэмуляционных приемов относятся:

1) возможность находить новые, ранее неизвестные антиотладочные или антиэмуляционные приемы в автоматическом режиме;

2) локализация местонахождения приема, вызвавшего расхождение. Действительно, для обнаружения антиотладочного или антиэмуляционного приема аналитику достаточно проанализировать последовательность инструкций, выполненных до расхождения и ведущих к разным состояниям отладчика и эмулятора;

3) устойчивость к приемам, направленным против виртуальных машин.

Недостатки данного алгоритма:

1) он не способен без дополнительного анализа отличить антиотладочные приемы от антиэмуляционных. В качестве средств дополнительного анализа могут быть использованы, например, сигнатурный или эвристический анализ. Эвристический анализ может, в частности, учитывать тот факт, что исследуемая программа после обнаружения отладчика или эмулятора обычно сразу завершается, а в противном случае выполняет свои функции;

2) за антиэмуляционный прием может быть принята случайная ошибка в эмуляторе, ведущая к неправильному выполнению инструкции процессора и, как следствие, к расхождению выполнения кода в предложенном алгоритме. Однако вероятность такой ошибки уменьшается с улучшением качества эмулятора;

3) скорость выполнения кода при анализе в предложенном алгоритме меньше, чем в эмуляторе и отладчике. Действительно, после каждой операции необходимо проводить дополнительное сравнение полученных результатов, что требует определенных временных затрат. Поэтому алгоритм не способен обнаруживать атаки, использующие тот факт, что в эмуляторе код выполняется значительно медленнее, чем на реальной машине. Для обнаружения таких атак необходимо использовать другие методы.

Итак, предлагаемый алгоритм способен в автоматическом режиме обнаруживать новые антиотладочные и антиэмуляционные приемы, а также помогает их локализовать для дополнительного анализа экспертом.

Реализация программы для поиска антиотладочных и антиэмуляционных приемов

Для демонстрации предлагаемого алгоритма была реализована программа, осуществляющая поиск антиотладочных и антиэмуляционных приемов в автоматическом режиме.

Для отладки анализируемого кода используется отладочной интерфейс *Windows Debug API*, выполняющий программу в пошаговом режиме (установлен *TF*). Для имитации инструкций используется модифицированный эмулятор от свободно распространяемого антивируса *Exploision Antivirus v.11*. Рассмотрим некоторые особенности реализации.

• Значение регистров эмулятора при запуске инициализируется регистрами отладчика. Это необходимо для последующей проверки регистров на эквивалентность.

• Эмулятор имитирует расположение стека по адресу стека отладчика. Это необходимо для эквивалентности указателя стека (*Extended Stack Pointer*).

• Эмулятор пересчитывает регистр EIP, имитируя расположение кода по базовому адресу образа (*ImageBase*), хотя реально код расположен по другому адресу.

• В процессе выполнения программа не сверяет *TF* регистра флагов.

• Для упрощения разработки сравнение участков памяти не осуществляется.

Проведем демонстрацию предложенного алгоритма обнаружения обманных приемов на примерах.

Пример 1. Защищенный код сверяет значение *TF* и, в случае если он установлен, сообщает об отладке (листинг 3). После сравнения *TF* исследуемый код продолжит выполняться по разным путям в отладчике и эмуляторе, что и обнаружит разработанное приложение.

Листинг 3. *Поиск отладчика по флагу трассировки.*

```
pushfd          ;сохраним регистр флагов в стек
pop eax         ;загрузим его в EAX
and eax,$100    ;сбросим все, кроме TF
test eax,eax    ;проверим, установлен ли TF
jnz .Debug      ;если установлен, нас отлаживают
```

Пример 2. В структуре блока переменных окружения процесса (*Process Enviroment Block — PEB*) есть поле *NtGlobalFlag*, которое может использоваться для выявления отладчика. Это поле находится по смещению 0x68 от начала *PEB*. Поле представлено рядом флагов. Значение, которое предполагает присутствие отладчика, складывается из значений следующих полей:

• FLG_HEAP_ENABLE_TAIL_CHECK (0x10)

• FLG_HEAP_ENABLE_FREE_CHECK (0x20)

• FLG_HEAP_VALIDATE_PARAMETERS (0x40)

Листинг 4. *Поиск отладчика с помощью NtGlobalFlag.*

```
mov eax,[fs:30h]
mov al, [eax+68h]
and al,70h
cmp al,70h
je .Debug
```

После команды mov al, [eax+68h] регистр al будет содержать разные значения в отладчике и эмуляторе, что обнаружит антиотладочный прием.

Пример 3. Воспользуемся приемом CALL-POP для определения реального адреса кода. Поскольку в эмуляторе код будет загружен по адресу, отличному от *ImageBase*, сравнивая «виртуальное» расположение кода с реальным, мы можем обнаружить наличие эмулятора (листинг 5). Реализованная программа успешно находит такой прием.

Листинг 5. *Обнаружение эмулятора по реальному адресу расположения кода.*

```
    call .Next      ;call-pop прием
.Next:
    pop eax         ;в EAX — адрес указателя .Next
    cmp eax,.Next   ;сравним EAX с реальным расположением
    jnz .Emul       ;переход, если мы в эмуляторе
```

Пример 4. Следующий прием (листинг 6) использует системное прерывание int 2e + inc edx для выявления отладчика. Поскольку эмулятор не может правильно выполнить команду, содержимое регистра eax будет отличаться в отладчике и эмуляторе, что обнаружит антиэмуляционный прием.

Листинг 6. *Обнаружение эмулятора по реальному адресу расположения кода.*

```
    mov eax,-1
    int 2eh ;это прерывание неправильно выполняется в эмуляторе
dt_:
    inc edx ;в отладчике int 2eh + inc edx выполнится как одна
команда (системное прерывание), и вернет значение 0xC000001C
    cmp eax, 0xc000001c ; STATUS_INVALID_SYSTEM_SERVICE
    jnz .aniEmu
;Переход если код выполняется в эмуляторе
```

Итак, разработанное приложение способно обнаруживать ряд обманных приемов.

Модификация отладчика для анализа вредоносного программного обеспечения

Поиск приемов обхода отладчика или эмулятора не всегда является целью. Во многих случаях (например, для поиска и анализа вредоносного кода) более важно имитировать правильное исполнение кода, обходя всевозможные защиты. В настоящее время некоторые отладчики (*OllyDebugger*, *Syser* [3]) применяют специальный метод анти-антиотладки. В процессе выполнения программы отладчик, используя сигнатурный анализ, пытается обнаружить известные ему антиотладочные приемы и подменяет результат их работы, таким образом не давая обнаружить свое присутствие. Однако таким методом можно обнаруживать только предопределенные антиотладочные приемы.

Для обхода антиотладочных приемов возможно дополнительно использовать предложенный

ранее алгоритм одновременного выполнения кода в отладчике и эмуляторе.

• Если программа отлаживается в ручном режиме, аналитику возможно выдавать сообщение о наличии расхождения регистров, памяти в отладчике или эмуляторе. Также возможно в диалоговом режиме предложить использовать значение регистров отладчика или эмулятора для дальнейшего исследования программы.

• Если исследование производится в автоматическом режиме, например для определения степени опасности ПО, то в случае наличия расхождения система может инициировать в памяти еще одну копию отладчика и эмулятора. Первая пара отладчик—эмулятор инициируется значениями регистров и памяти отладчика, вторая — эмулятора. Преимуществами данной модификации отладчика являются:

— незначительная, в сравнении с виртуальными машинами, ресурсоемкость;

— увеличенная устойчивость к антиотладочным приемам.

Недостатки обусловлены недостатками описанного в первой части статьи алгоритма:

— меньшая, по сравнению с отладчиком, скорость работы;

— подверженность случайным ошибкам, не являющимся антиотладочными или антиэмуляционными приемами, что может привести к увеличению затрат времени на дополнительный анализ.

Заключение

В работе проведен анализ наиболее существенных принципов построения и уязвимостей отладчиков и эмуляторов с точки зрения их использования во вредоносном ПО.

По итогам проведенного анализа предложен новый алгоритм детектирования антиотладочных и антиэмуляционных приемов, основанный на параллельном выполнении кода в отладчике и эмуляторе. Алгоритм был реализован программно. Рядом примеров проиллюстрирована его работоспособность.

Результаты работы могут быть использованы для более эффективного анализа вредоносного кода.

Литература

1. **Foster J. C., Price M.** Sockets, Shellcode, Porting, and Coding. — Syngress, 2005. — 667 p.
2. **Робинс Д.** Отладка приложений для Microsoft .NET и Microsoft Windows: пер. с англ. — М.: Русская Редакция, 2004. — 736 с.
3. **Касперски К.** Энциклопедия антиотладочных приемов // Хакер. 2008. № 11–16.
4. **Szor P.** The art of computer virus research and defense. — Addison-Wesley Professional, 2005. — 744 p.
5. **Mollin R. A.** Introduction to Cryptography. Second ed. — Taylor&Francis, 2006. — 413 p.
6. **Агафонов А.** Эмуляция программного кода // UINC.RU: Underground Information Center, 2004. http://www.uinc.ru/articles/47/ (дата обращения: 10.12.09).