электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.056.5

МЕТОДЫ ПРОТИВОДЕЙСТВИЯ АНАЛИЗУ ПРОГРАММ

***Карондеев А.М.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Информационная безопасность»

***Терещенко В.А.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Информационная безопасность»

Научный руководитель: Старчак С.Л., д.т.н., доцент, профессор отдела № 1 УВЦ ВИ
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана

Земляков В.Н., *начальник цикла отдела № 1 УВЦ ВИ*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана

Научный консультант: Шаменков Н.А., к.т.н., начальник лаборатории,
НИЦ РКО 4 ЦНИИ МО РФ, г. Москва, Россия
bauman@bmstu.ru

Введение

Современный этап развития общества характеризуется интенсивным ростом информатизации. Вместе с ним развиваются и технологии защиты программного обеспечения (ПО). Как правило, раньше или позже, после появления новых технологий защиты появляются методы её обхода. Поэтому проблема построения эффективной системы защиты со временем не теряет своей актуальности, равно как и сопряженные с решением проблемы комплекс задач оценки эффективности систем защиты ПО и разработка более эффективных и совершенных средств защиты. В рамках настоящего исследования рассмотрены методы противодействия анализу программ (МПАП).

Большинство атак на защищенные компьютерные системы начинается с анализа программной реализации средств защиты. До тех пор, пока нарушитель не произведет достаточно полный анализ программной реализации защиты, он не сможет использовать для преодоления защиты программные уязвимости атакуемой системы. Поскольку уязвимости ПО, как правило, являются наиболее удобными «входными воротами» для проникновения в систему, большинство хакеров рассматривают анализ программной реализации атакуемой системы как неотъемлемый этап её взлома [2].

МПАП позволяют предотвратить или существенно замедлить процесс анализа программ. МПАП применяются не только для повышения эффективности систем защиты,

но и для сохранения интересных программных решений от конкурирующих фирм. Как правило, новая технология, разработанная специалистами одной фирмы, очень быстро осваивается конкурентами, и анализ программной реализации новой технологии играет далеко не последнюю роль в этом процессе.

Разработчики программной системы защиты целью своей деятельности считают, как минимум, затруднить анализ разрабатываемой системы потенциальным нарушителем. Однако далеко не все разработчики включают в свои программы средства защиты от анализа. Это обусловлено следующими факторами:

- Использование МПАП может заметно усложнить отладку программы и тем самым снизить её надёжность.
- МПАП могут давать ложные срабатывания при запуске программы в необычной программно-аппаратной среде: на новой модели процессора, в новой версии операционной системы (ОС), в эмуляторе и т.п.
- Современные механизмы юридической защиты авторских прав снимают необходимость включать МПАП в широкий круг программ.

Принимая решение о том, включать ли в разрабатываемый программный продукт МПАП, разработчик должен решить для себя вопрос о целесообразности данного шага. Обычно МПАП оправдывают свое использование в программах оснащенных защитой от несанкционированного копирования. В основном, это программы, предназначенные для обработки строго конфиденциальной информации, или очень дорогие программы. Широко известная программа Skype является хорошим примером удачного использования МПАП. Skype оснащена мощнейшей защитой от анализа, которая, как ни странно, не снижает эксплуатационные качества программы и не вызывает заметных сбоев [2]. Стоит отметить, что МПАП также активно используются во вредоносных программах (malware) с целью сокрытия наличия разрушительной функции. Например, алгоритм функционирования вируса Sober удалось полностью восстановить лишь спустя четыре месяца после начала его распространения.

В ходе исследования авторами были проанализированы известные данные о средствах анализа программ и методах защиты от анализа. На основе результатов анализа составлена классификация МПАП, ориентированная на потребности разработчиков систем защиты ПО. Кроме того, выполнена опытно-экспериментальная работа, в ходе которой были разработаны тестовые программы на языке ассемблер, реализующие различные МПАП, а также изучено поведения различных средств анализа, таких как WinDBG, IDA Pro, OllyDBG, Syser, Immunity Debugger и т.д.

Обзор средств анализа программ

При исследовании программ специалисту приходится пользоваться различными инструментами, позволяющими более эффективно выполнять поставленные задачи. Эти инструменты способны поднять производительность труда аналитика в несколько раз. О существующих инструментах и их возможностях полезно знать и разработчикам средств безопасности, чтобы более эффективно создавать трудности аналитикам, которые будут пытаться найти дыры в защите. Под анализом программ понимается восстановление алгоритма работы программы по исполняемому модулю, загруженному в память. Существует множество разнообразных средств анализа программ, но все эти средства можно разделить на два класса:

- средства статического анализа программ;
- средства динамического анализа программ.

Средства статического анализа программ оперируют кодом программы как данными и строят её алгоритм без исполнения, что позволяет анализировать вредоносные программы без риска для системы. Исполняемые файлы программы обычно состоят из заголовка, в котором содержится необходимая для работы вспомогательная информация, и последовательности исполняемых команд, записанных в машинных кодах команд процессора. Средства статического анализа занимаются поиском данных необходимых для восстановления алгоритма работы программы и переводят их на язык, понятный аналитику. Основу такого перевода составляют программы дизассемблирования – дизассемблеры, которые переводят последовательность машинных кодов в листинг, близкий к исходному тексту программы на языке ассемблера. Дальнейшая работа после дизассемблирования сводится к анализу полученных листингов, поиску интересующих участков, и переводу описания процедур функционирования на понятный аналитику язык.

При отсутствии в программе специальных средств защиты от дизассемблирования средства статического анализа позволяют полностью восстановить алгоритм. Статический анализ даёт возможность понять структуру программы, способ вызова и взаимодействия отдельных модулей. Эффективность статического анализа не зависит от сложности алгоритма. Для применения средств статического анализа достаточно иметь в распоряжении лишь исследуемую программу. Не требуется, чтобы анализируемая программа, имеющаяся у аналитика, была работоспособна. Это удобно в тех случаях, когда предметом исследования является программный комплекс, требующий дорогостоящих средств, которые отсутствуют у аналитика. Статические методы в большей мере, чем другие подходы, допускают автоматизацию отдельных этапов анализа. Из всех средств статического анализа следует выделить наиболее успешный дизассемблер

IDA Pro. Он до определенной степени, умеет автоматически выполнять анализ кода, используя перекрестные ссылки, знание параметров вызовов функций стандартных библиотек, и другую информацию. Однако вся сила его проявляется в интерактивном взаимодействии с пользователем. В начале исследования дизассемблер выполняет автоматический анализ программы, а затем пользователь с помощью интерактивных средств IDA начинает давать осмысленные имена, комментировать, создавать сложные структуры данных и другим образом добавлять информацию в листинг, генерируемый дизассемблером, пока не станет ясно, что именно и как делает исследуемая программа. IDA Pro поддерживает большое количество форматов исполняемых файлов. Одной из отличительных особенностей IDA Pro является возможность дизассемблирования байт-кода виртуальных машин Java и .NET.

Средства динамического анализа программ изучают программу, интерпретируя ее в реальной или виртуальной вычислительной среде. Динамические средства строят алгоритм работы программы, на основе конкретной трассы полученной при определенных входных данных. Поэтому задача получения полного алгоритма программы в этом случае эквивалентна построению исчерпывающего набора тестов, что практически невозможно, для подтверждения правильности программы, поэтому при динамическом анализе обычно происходит восстановление лишь части алгоритма. Основным средством динамического анализа является отладчик. Отладчик – это программа, которая загружает в память другую программу и предоставляет пользователю возможность наблюдать за ходом выполнения этой программы. Механизм работы любого отладчика основан на использовании специальных средств, аппаратно реализованных в процессорах. Для процессоров Intel x86 к этим средствам относят: флаг трассировки, точка останова и отладочные регистры.

Флаг трассировки (Trace flag, TF) – это восьмой бит регистра флагов. Если этот бит равен единице, то процессор после выполнения каждой машинной команды вызывает прерывание 1. Флаг трассировки используется отладчиками для пошагового выполнения программы. В этом случае отладчик перехватывает прерывание 1, и после каждого выполнения машинной команды отлаживаемой программы отображает на экране состояние регистров процессора, флагов, сегментов памяти и т.д.

Точка останова (Breakpoint) – машинный код команды вызова прерывания 3 (int 3), в отличие от других прерываний, занимает всего один байт. Поэтому эта команда может быть вставлена в любое место отлаживаемой программы. Отладчик перехватывает прерывание 3, и после установки точки останова (замены первого байта машинной команды на байт CCh – код команды int 3) передает управление отлаживаемой программе. Перед выполнением команды, на которой стоит точка останова (команды, первый байт

которой заменен на CCh), управление передается отладчику. Перед повторным запуском программы отладчик восстанавливает исправленную команду.

Отладочные регистры (Debug register) – это специальные регистры процессора, которые позволяют установить в памяти ЭВМ четыре аппаратные точки останова. В отличие от обычных точек останова, аппаратные обладают более широкими возможностями. Отлаживаемая программа может быть остановлена не только при выполнении определённой команды, но и при обращении к определенному участку памяти. При останове программы аппаратной точкой останова вызывается прерывание 1.

Отладчики бывают трех основных типов: уровня пользователя, уровня ядра и эмулирующие [3]. Отладчики пользовательского уровня (User-level Debuggers) имеют практически те же возможности, что и отлаживаемая программа. Они используют Debugging API, входящий в состав операционной системы и с его помощью осуществляют контроль над объектом отладки. Среди наиболее распространённых следует выделить OllyDBG и Immunity Debugger. Они пригодны для исследования незащищенных программ, но могут быть легко обнаружены. Отладчики уровня ядра (Kernel-mode Debuggers) встраиваются внутрь операционной системы и имеют гораздо больше возможностей, чем отладчики пользовательского уровня. Из ядра операционной системы можно контролировать многие процессы, не доступные другими способами. Но и отладчики уровня ядра почти всегда могут быть обнаружены из программы, не имеющей доступа к ядру. Из отладчиков уровня ядра наиболее распространены WinDBG и Syser. Эмулирующие отладчики, пожалуй, являются самым мощным средством исследования кода программ. Такие отладчики эмулируют выполнение всех потенциально опасных действий, которые программа может использовать для выхода из-под контроля исследователя. Однако основная проблема создания эмулирующих отладчиков заключается в том, что иногда им приходится эмулировать реальное периферийное оборудование, а это чрезвычайно сложная задача.

По сравнению со статическим анализом динамический позволяет проводить поиск интересующего алгоритма более целенаправленно. Аналитик может запускать и просматривать именно те участки программы, в которых, как ему известно, используются интересующие его алгоритмы. Динамический анализ позволяет в момент останова наблюдать состояние процессора и оперативной памяти, что облегчает понимание реализуемых программой преобразований.

Классификация методов противодействия анализу программ

Методы противодействия анализу программ предназначены для предотвращения восстановления алгоритма работы программы по исполняемому модулю, загруженному в память. МПАП законно используются коммерческими приложениями, для предотвращения раскрытия кода, его модификации и пиратства. Стоит отметить, что вредоносные программы также используют МПАП, но в злонамеренных целях, для сокрытия наличия функции разрушения.

МПАП очень разнообразны и не похожи друг на друга, поэтому классифицировать их весьма сложно. Зачастую в литературе связанной с МПАП классификация вовсе не приводится, а идет простое описание тех или иных методов, поясняющих основные принципы. Существующие классификации МПАП, в основном, ориентированы на вирусных аналитиков и, в качестве основного признака разделения на классы, используют средство анализа, которому противодействует метод. В ходе исследования из существующих признаков для классификации МПАП выбраны наиболее важные с точки зрения разработчиков систем защиты ПО. На основе отобранных признаков составлена классификация МПАП, приведённая на рисунке 1.

Классификация не является взаимоисключающей, то есть один МПАП может относиться одновременно к нескольким категориям. Как видно из рисунка 1, в основу классификации МПАП положено три основных признака, рассмотрим каждый из них с точки зрения разработчика системы защиты ПО. При создании конкретной системы защиты разработчик, как правило, привязан к определённым программным и аппаратным средствам. Описание особенностей, лежащих в основе МПАП, даёт разработчику краткое представление о механизме работы метода достаточное, чтобы понять применим ли данный метод для построения системы защиты ПО в конкретной программно-аппаратной среде. Использование МПАП существенно усложняет процесс отладки программы на стадии разработки, что приводит к дополнительным временным затратам. Описание типа поведения МПАП позволяет снизить эти дополнительные временные затраты.

Существует множество разнообразных средств анализа программ. Большинство МПАП не способны эффективно противодействовать одновременно нескольким средствам анализа. На практике эффективная защита от анализа достигается путем комбинирования нескольких МПАП. Описание средств анализа, которым противостоит метод, позволяет быстро произвести выбор нескольких МПАП из известных, комбинация которых будет эффективно противодействовать большинству средств анализа.

Рис. 1. Классификация МПАП

Сравнительный анализ типовых методов противодействия анализу программ

Для пояснения основных принципов, лежащих в основе большинства МПАП, и раскрытия сущности признаков, положенных в основе классификации, рассмотрим типовые МПАП. Под типовыми МПАП в рамках данной статьи понимаются наиболее часто упоминаемые в открытой литературе.

МПДАП обычно используют особенности анализирующего ПО. Далеко не все средства анализа способны полностью скрыть своё присутствие от отлаживаемой программы и как следствие могут быть обнаружены. Существует множество разнообразных методов обнаружения средств анализа, но все они используют те или особенности анализирующего программного обеспечеия. Большинство МПДАП

основаны, именно, на обнаружении средств анализа в системе блок схема таких методов изображена на рисунке 2.

Самым простым методом обнаружения отладчика является использование API функции IsDebuggerPresent. Если текущий процесс выполняется в контексте отладчика, то возвращаемое IsDebuggerPresent в регистр al значение будет равно единице (TRUE), в противном случаи регистр al будет содержать ноль (FALSE). Пример использования этого метода приведен в листинге 1.

Листинг 1

```
…
call IsDebuggerPresent
test al, al   ;# В случае наличия отладчика al содержит TRUE
jne being_dbg ;# На основе результата сравнения выбрать поведение
…
```

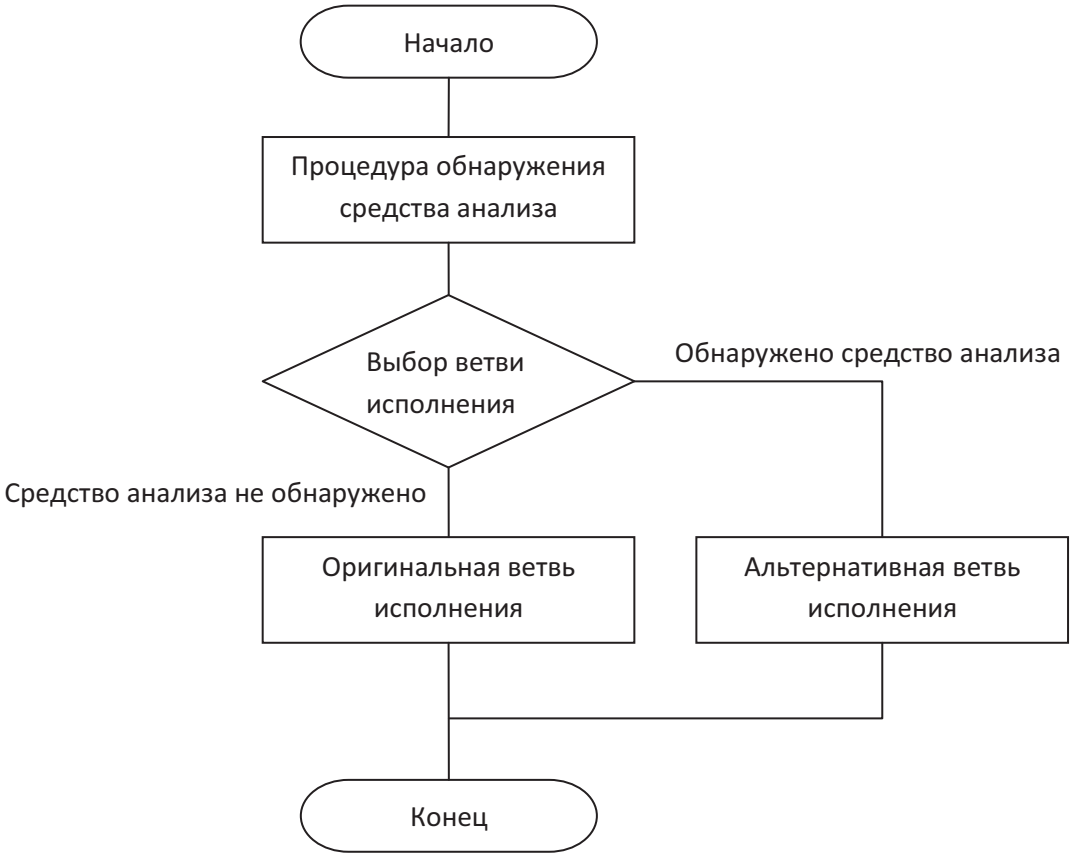

Рис. 2. Блок схема МПДАП основанных на обнаружении средств анализа

На рисунке 3 приведено поведения отладчика OllyDBG при анализе тестовой программы, использующей API функцию IsDebuggerPresent для обнаружения отладчика.

В таблице 1 приведено описание рассмотренного метода, основанное на предложенной классификации. Эта и все последующие таблицы составлены по следующему принципу. Первая строка содержит название, вторая характеризует тип анализа, которому противодействует метод, третья описывает тип поведения МПАП, а в четвертой описываются особенности, лежащие в основе метода.

Таблица 1

IsDebuggerPresent
МПДАП. Обнаруживает отладчик уровня пользователя.
Поведение активное с альтернативной ветвью исполнения.
Использует особенность ОС Windows – API функцию IsDebuggerPresent.

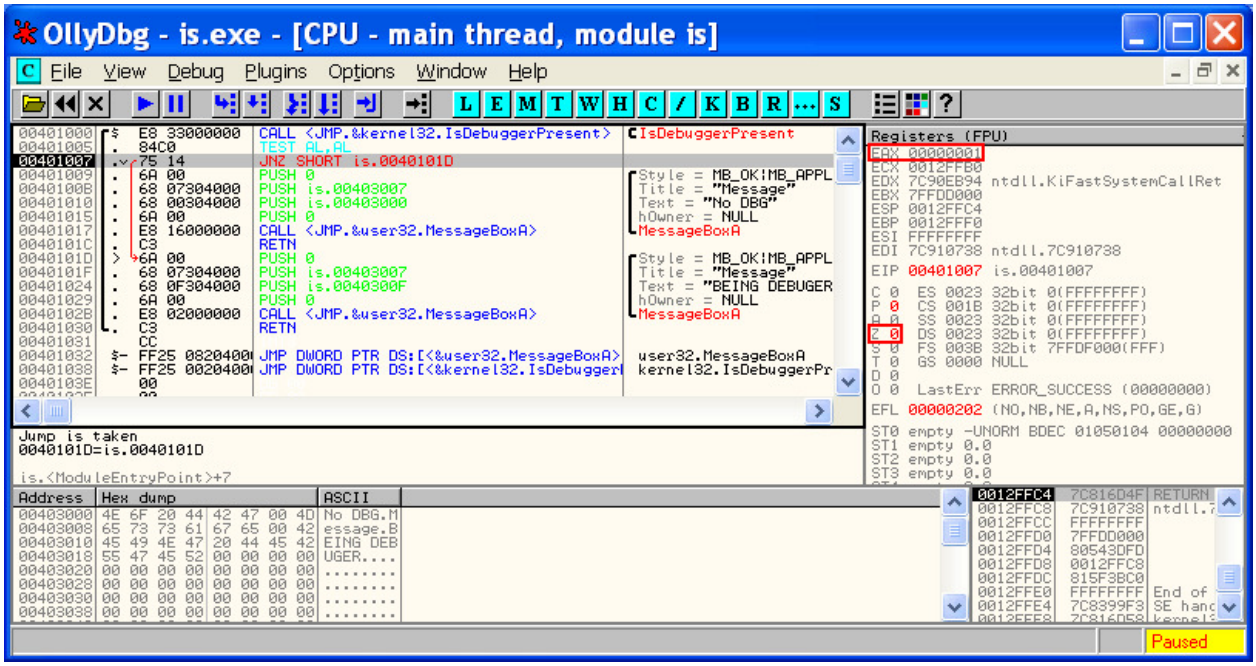

Рис. 3. Анализ результата API функции IsDebuggerPresent

Схожим образом можно определить наличие отладчика, считав данные непосредственно из структуры Process Enviroment Block (PEB). PEB – это структура, расположенная в пользовательском пространстве, которая содержит данные окружающей среды процесса:

- Переменные среды
- Список загруженных модулей
- Адреса в памяти Heap

PEB располагается в памяти по адресу fs:[30h]. По смещению 2 в PEB находится флаг наличия отладчика BeingDebugged, анализируя который можно узнать о присутствии

отладчика. Пример реализации МПАП, проверяющего наличие отладчика на основе данных из PEB, приведен в листинге 2, а соответствующее ему описание в таблице 2.

Рисунок 4 демонстрирует как программа, использующая рассматриваемый метод, обнаруживает отладчик Immunity Debuger.

Листинг 2

```
…
mov eax, fs:[30h]   ;# Поместить в регистр eax указатель на PEB
cmp b [eax+2], 0    ;# Проверить флаг BeingDebugged
jne being_dbg       ;# На основе результата сравнения выбрать поведение
…
```

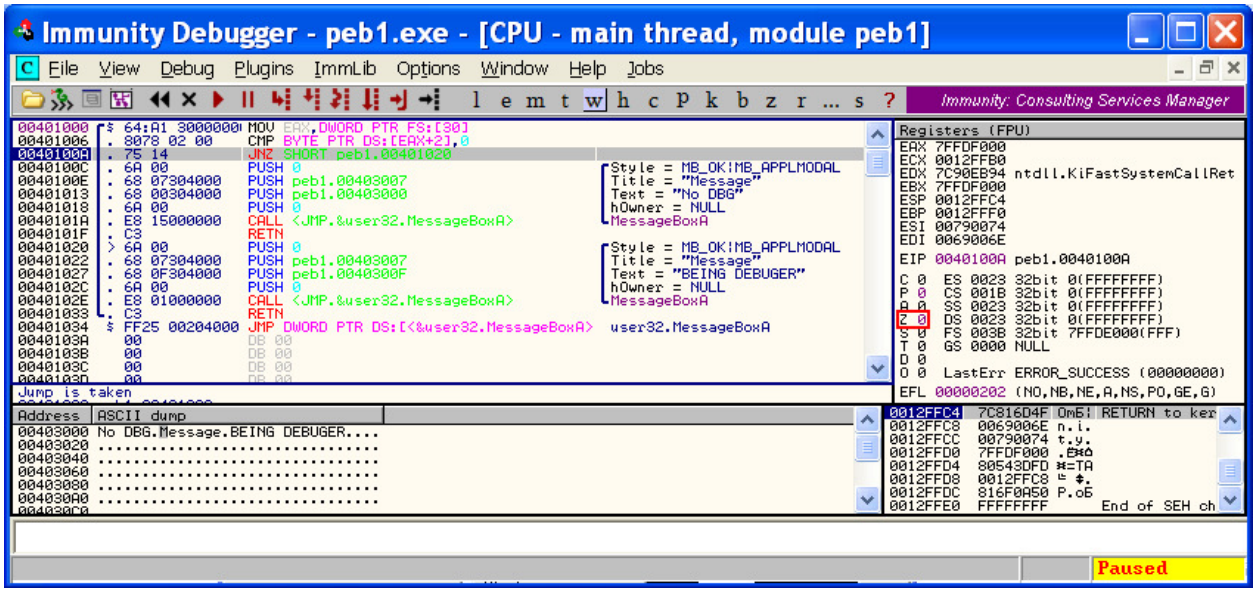

Рис. 4. Анализ флага BeingDebugged

Большинство отладчиков распознают имена экспортируемых функций, вызываемых библиотек динамической компоновки, и находят в программе места их использования. Поэтому МПАП, основанный на считывании флага BeingDebugged из структуры PEB, обнаружить и обойти сложнее, чем рассмотренный ранее МПАП, использующий библиотечную функцию IsDebuggerPresent.

Таблица 2

BeingDebugged
МПДАП. Обнаруживает отладчик уровня пользователя.
Поведение активное с альтернативной ветвью исполнения.
Использует особенность ОС Windows – флаг наличия отладчика BeingDebugged в PEB

Рассмотрим МПДАП, основанный на измерении времени выполнения инструкций. Под воздействием большинства средств динамического анализа ПО программа исполняется гораздо медленнее и посредством измерения времени исполнения блока инструкций можно сделать вывод о наличии средства анализа в системе. Произвести измерения времени можно посредством инструкции RDTSC (Read Time Stamp Counter), которая возвращает в регистрах EDX:EAX 64-битное количество тактов с момента последнего сброса процессора. В листинге 3, происходит измерение времени выполнения инструкции XCHG EBX, EAX и на основании сравнения с ожидаемой величиной выбирается ветвь исполнения.

Листинг 3

```
…
rdtsc             ;# Поместить текущие значение TSC=t1 в регистр eax
xchg ebx, eax     ;# Поместить t1 в регистр ebx
rdtsc             ;# Поместить TSC=t2 в регистр eax
sub eax, ebx      ;# Поместить d=t1-t2 в регистр eax
cmp eax, 100h     ;# Сравнить d с ожидаемым временем исполнения
jnb being_dbg     ;# На основе результата сравнения выбрать поведение
…
```

Из рисунка 5 видно, что отладчик вносит существенную временную задержку, которая позволяет тестовой программе определить наличие отладчика.

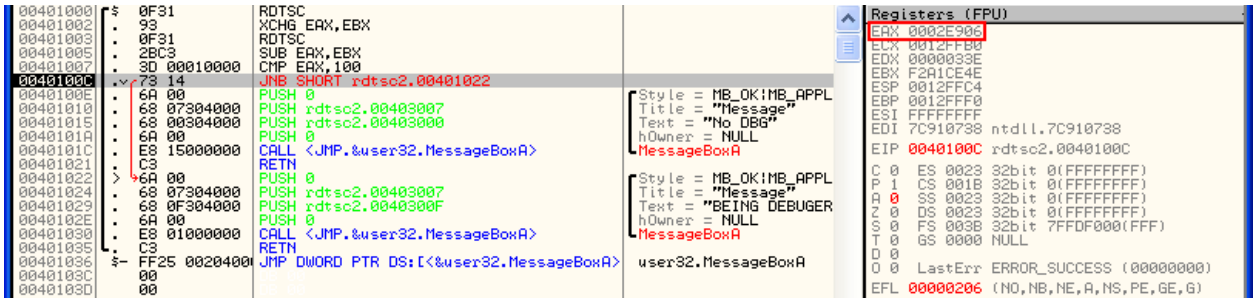

Рис. 5. Измерение времени выполнения команд.

Основным преимуществом приведённого в листинге 3 метода является то, что он не привязан к конкретной ОС, и, как следствие, может быть встроен в системы защиты ПО любой ЭВМ, построенной на базе процессоров Intel x86 вне зависимости от установленной на неё ОС. Таблица 3 содержит описание данного МПАП.

Интересной аппаратной особенностью процессоров x86 является очередь предварительной выборки. Она очищается автоматически при записи по адресу,

соответствующему адресу байтов в очереди предварительной выборки, но с двумя исключениями REP MOVS и REP STOS.

Таблица 3

RDTSC
МПДАП. Обнаруживает отладчик уровня пользователя, отладчик уровня ядра и эмулятор.
Поведение активное с альтернативной ветвью исполнения.
Использует особенность средств анализа ПО – внесение дополнительной задержки

Эти две команды процессор кэширует и продолжает выполнять их, даже после их перезаписи в памяти. Они завершат выполнение либо при обнулении регистра ECX, либо при генерации исключения, как, например, при исключении ошибки доступа (access violation). Пример МПДАП основанный на использовании особенности очереди предварительной выборки приведен в листинге 4. Данный метод перезаписывает часть своего кода, включая команду REP STOS, которая производит запись. Неправильная эмуляция (или выполнение в пошаговом режиме) заставит REP STOS завершиться преждевременно и значение в ECX будет отлично от нуля, что позволит сделать вывод о наличии средства анализа.

Листинг 4

```
…
l1: mov al, 90h      ;# поместить в регистр al код команды nop
push 20h             ;# положить в стек значение 20h
pop ecx              ;# извлечь из стека значение 20h в регистр ecx
cmov edi, offset l1  ;# поместить в edi адрес начала перезаписи l1
rep stosb            ;# произвести перезапись кода
…
jecxz being_debugged ;# на основании значения ECX выбрать поведение
…
```

На рисунках 6 и 7 приведен процесс анализа программы отладчиком OllyDBG.

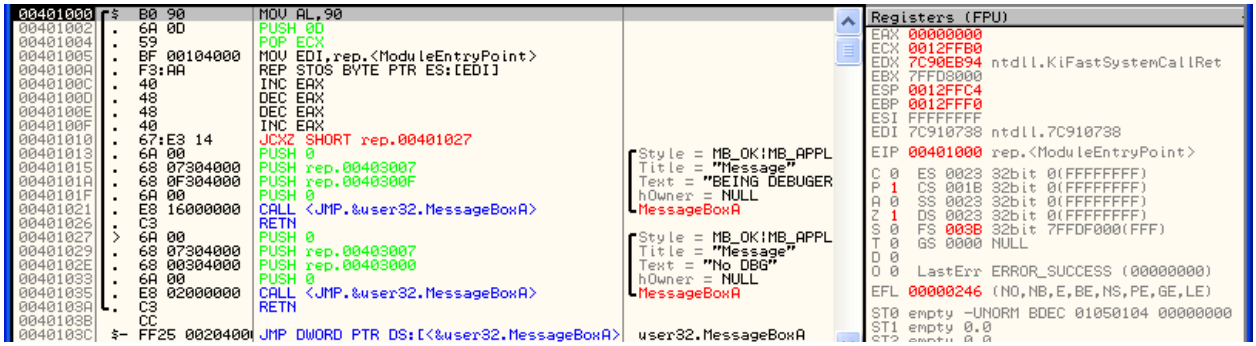

Рисунок 6. Программа до выполнения команды REP STOS

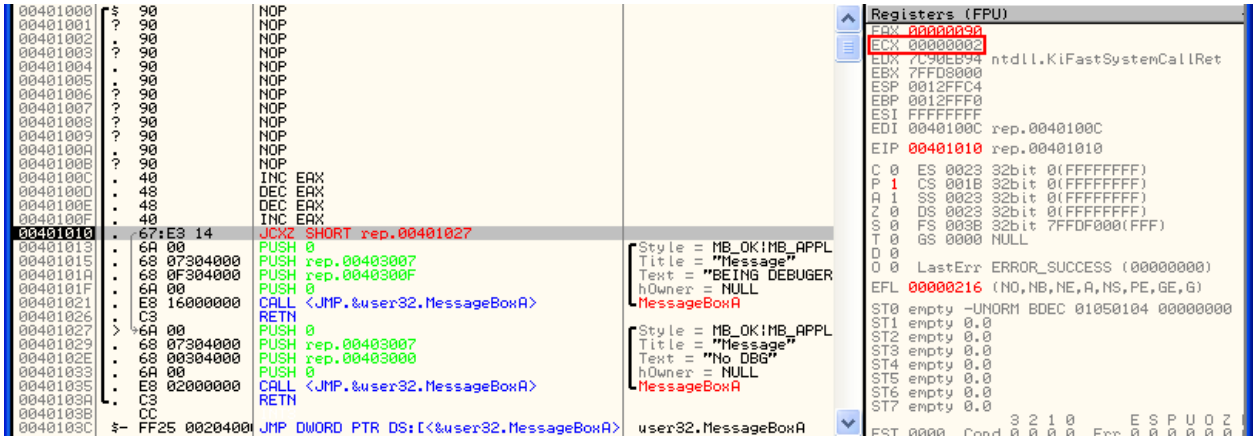

Рис. 7. Программа после выполнения команды REP STOS

Из рисунка 7 видно, что после перезаписи команды REP STOS процесс записи прекратился до того как обнулился регистр ECX, что позволило тестовой программе определить наличие отладчика.

МПАП, основанный на использовании очереди предварительной выборки, в сравнении с рассмотренными ранее методами обладает одним недостатком, а именно данный метод предполагает использование сегмента памяти с одновременно установленными флагами записи (Write W) и исполнения (Execute E). Многие средства анализа предоставляют сведенья о наличии таких сегментов, что может облегчить задачу анализа. Описание этого метода приведено в таблице 4.

Таблица 4

REP STOSB
МПДАП. Обнаруживает отладчик уровня пользователя.
Поведение активное с альтернативной ветвью исполнения.
Использует аппаратную особенность процессоров Intel x86 – очередь предварительной выборки.

Далее рассмотрим МПДАП, использующий программное прерывание 2D. Если программа не отлаживается, выполнение этого прерывания генерирует исключение точки останова (EXCEPTION_BREAKPOINT 0x80000003). Если программа запущена под отладчиком и выполняется со сброшенным флагом трассировки, то никакого исключения не произойдет, и выполнение будет продолжено в обычном режиме. Если программа отлаживается и команда трассируется, то следующий байт будет пропущен, и выполнение продолжится через команду [10]. Поэтому прерывание 2D можно использовать для обнаружения средств динамического анализа программ. Перед описанием метода рассмотрим общий порядок обработки исключений. Когда программа генерирует ошибку (например, ошибку деления на 0), процессор генерирует исключение. Далее, смотря на IDT (Interrupt Dispatch Table), процессор находит адрес обработчика прерывания (Interrupt Service Handler, ISR). Затем ISR ищет пользовательские обработчики исключений до тех пор, пока один из обработчиков не обработает ошибку. Если пользовательский обработчик возвращает 0 (ExceptionContinueExecution) – ISR возобновляет прерванный процесс пользователя. Если обработчик возвращает 1 (ExceptionContinueSearch) – ISR продолжает поиск следующего обработчика в связанном списке. Информация об обработчиках организована в виде связанного списка структур _EXCEPTION_REGISTRATION.

FS:[0] содержит адрес первой записи _EXCEPTION_REGISTRATION. В Win32, регистр FS всегда указывает на структуру данных TIB (Thread Information Block).

Листинг 5

```
_EXCEPTION_REGISTRATION struc
Prev dd       ?      ;# Указатель на предыдущую запись
handler dd    ?      ;# Адрес обработчика исключения
_EXCEPTION_REGISTRATION ends
```

В TIB хранится важная системная информация (такая как: вершина стека, последняя ошибка, идентификатор процесса) текущего потока. Таким образом, из FS:[0] ISR может вызывать пользовательские обработчики. В листинге 6 приведен пример использования int 2Dh для противодействия динамическому анализу.

Листинг 6

```
xor eax, eax         ;# очищаем регистр EAX

;# Устанавливаем новый обработчик исключений #
```

```
push offset l1        ;# помещаем в стек указатель на новый обработчик
push d fs:[eax]       ;# помещаем в стек указатель на предыдущую запись
;#теперь в стеке содержится новая запись _EXCEPTION_REGISTRATION

mov fs:[eax], esp     ;# помещаем в fs:[0] указатель на новую
                      ;# запись _Exception_Registration
;#
int 2dh         ;# прерывание => CPU должен вызвать обработчик l1
inc eax         ;# EAX = 1, пропускается, если присутствует отладчик
je being_dbg    ;# Если EAX=0, то отладчик присутствует
...
;# Обработчик исключений #
l1:
xor eax, eax    ;# обнуляем регистр EAX
ret             ;# EAX=0 (ExceptionContinueExecution) т.е. исключение
                ;# было  обработано и прерванный  процесс должен
                ;# возобновиться
```

Рисунок 8 демонстрирует то, как программа, использующая рассматриваемый метод, успешно обнаруживает отладчик Immunity Debuger.

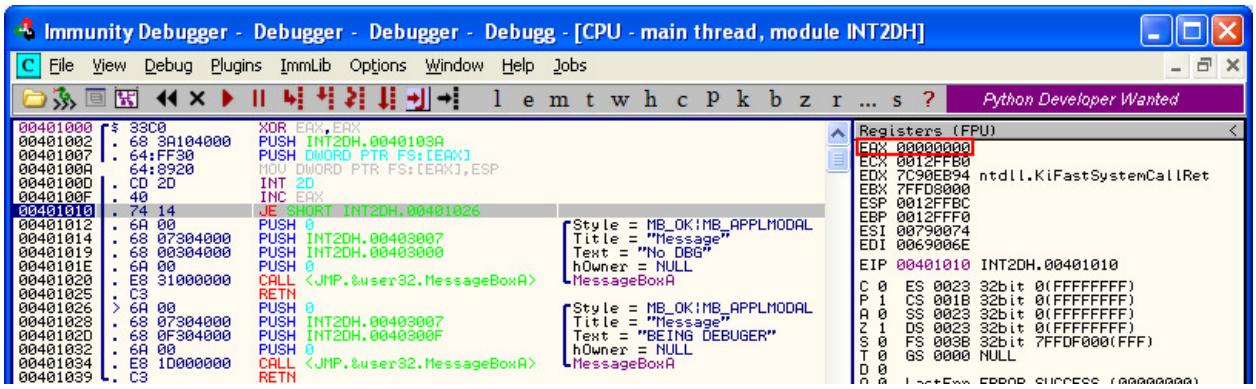

Рис. 8. Immunity Debuger после выполнения команды INT 2D

При отладке тестовой программы отладчиком уровня ядра WinDBG при выполнении команды INT 2D будет сгенерировано исключение точки останова, что показано на рисунке 9. После чего WinDBG обработает исключение, и программа продолжит свое выполнение. Однако при обычном запуске тестовой программы обрабатывать исключение будет обработчик установленный программой, что позволяет определить наличие отладчика уровня ядра.

http://sntbul.bmstu.ru/doc/636274.html

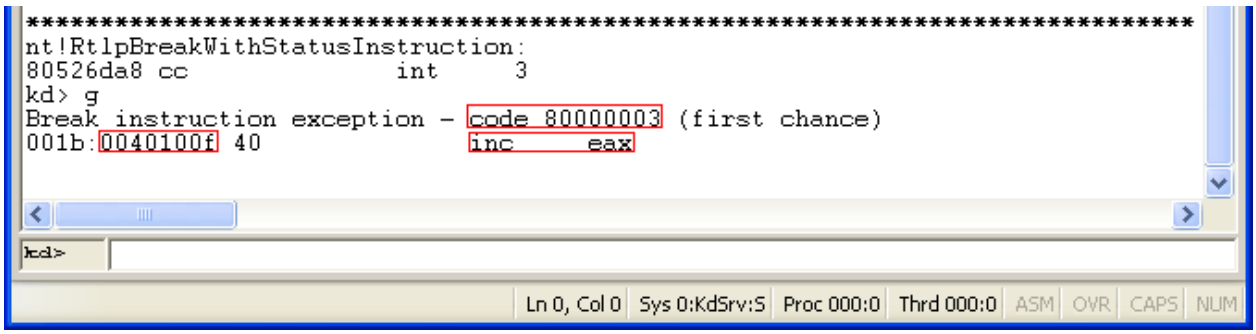

Рис. 9. WinDBG после выполнения команды INT 2D

В сравнении с методами, рассмотренными ранее, этот МПАП обладает существенным преимуществом. Даже обнаружив его, для его преодоления аналитик должен хорошо разбираться во внутренних механизмах функционирования ОС, что в ряде случаев способно существенно замедлить процесс анализа. Описание этого МПАП приведено ниже в таблице 5.

Таблица 5

INT 2Dh
МПДАП. Обнаруживает отладчик уровня пользователя и отладчик уровня ядра.
Поведение активное с альтернативной ветвью исполнения.
Использует особенность средств анализа ПО – перехват исключения точки останова Использует особенность ОС Windows – порядок обработки исключений

МПСАП обычно используют особенности анализирующего ПО и особенности человеческого восприятия. МПСАП основаны на обфускации кода, самораспаковке и модификации/стирании дополнительной информации о программе.

Под обфускацией кода понимается модификация кода, сохраняющая его функциональность, но затрудняющая его восприятие человеком. В простейшем случае обфускация реализуется добавлением “мусорных” инструкций.

Под самораспаковкой понимается модификация кода программы во время её исполнения, вводящая в заблуждение средства статического анализа программ. Простейший пример самораспаковки – хранение части кода в зашифрованном виде и расшифровка этой части во время выполнения программы. В качестве примера рассмотрим функцию расшифровки участка кода зашифрованного посредством сложения по модулю два с ключом фиксированной длины. Пример реализации данной функции приведен в листинге 7.

Листинг 7

```
…
XOR EAX, EAX                ;# Отчистить регистр EAX
MOV EBX, OFFSET KEY         ;# Поместить в EBX указатель на ключ
MOV EDI, OFFSET CTEXT       ;# Поместить в EDI указатель на шифртекст
;# Побайтовая функция расшифровки ###
;# Регистр EAX содержит номер байта ключа используемый
;# для расшифровки в текущей итерации
;# DS:[EDI] указывает на байт шифртекста, который
:# расшифровывается в текущей итерации
L1:
MOV DL, BYTE PTR DS:[EDI]     ;# Считать очередной байт шифртекста
XOR DL, BYTE PTR DS:[EAX+EBX] ;# XOR с ключом
MOV BYTE PTR DS:[EDI], DL     ;# Записать расшифрованный байт
INC EAX              ;# увеличиваем на 1 значение регистра EAX
;# Приводим EAX по модулю длины ключа
CMP EAX, 13   ;# Сравниваем EAX с длиной ключа
JL L2         ;# Если значение регистра EAX ≥ 13
XOR EAX, EAX  ;# обнуляем регистр EAX
L2:
INC EDI       ;# увеличиваем на 1 значение регистра EDI
LOOPD L1      ;# Переходим к следующей итерации
;#########################
…
```

МПАП, основанные на хранении части кода в зашифрованном виде, описаны в таблице 6. Зашифрованные части кода хранятся в исполняемом файле в искажённом виде и преобразуются к нормальному виду лишь в оперативной памяти в ходе выполнения программы. Поэтому такие методы препятствуют работе дизассемблера. Однако они совсем не противодействуют средствам динамического анализа. Стоит отметить, что в случае использования асимметричных криптографических алгоритмов, динамический анализ позволяет получить только открытый ключ, а без закрытого ключа модификация зашифрованного кода практически не возможна.

Таблица 6

Хранение части кода в зашифрованном виде
МПСАП. Предотвращает дизассемблирование.
Поведение пассивное.

Использует особенность средств анализа ПО – зашифрованная часть кода интерпретируется дизассемблером как данные.

Еще одним МПСАП является возврат на два уровня. Данный метод использует две вложенные функции. Первая функция содержит запутывающий код, которому предшествует вызов второй функции. Вторая функция перед завершением извлекает с вершины стека адрес возврата в первую функцию, в результате чего после завершения второй функции не происходит передача управления запутывающему коду, а осуществляется возврат сразу на два уровня назад. Данный метод препятствует построению графа вызовов. Пример реализации приведен в листинге 8.

Листинг 8

```
CALL L1   ;# Вызов L1. На вершине стека CODE2.
CODE2:
…
L1:
CALL L2   ;# Вызов L2. На вершине стека CODE1.
CODE1:    ;# Запутывающий код. Никогда не получает управление.
…
L2:
POP EAX   ;# Извлечение вершины стека. Теперь на вершине стека CODE2.
RETN      ;# Возврат к CODE2, а не к CODE1. CODE1 не будет выполнен.
```

Рассмотренный метод использует нетипичное применение инструкции RETN, что способно ввести в замешательство многие средства статического анализа. Описание этого метода приведено в таблице 7.

Таблица 7

Возврат на два уровня
МПСАП. Препятствует восстановлению оригинального графа потока исполнения.
Поведение пассивное.
Использует особенность средств анализа ПО – нетипичное применение инструкции RETN

Стоит отметить, что предложенная в работе классификация, полезна не только при выборе МПАП для построения противодействующих анализу систем защиты ПО, но и

может помочь в разработке новых МПАП. А именно из структуры классификации отчетливо видно, где стоит искать особенности для основы нового МПАП.

Заключение

На основе результатов произведенной исследовательской части работы, составлена классификация МПАП, ориентированная на разработчиков систем защиты ПО. Составленная классификация МПАП позволяет систематизировать уже существующие МПАП, облегчает процесс разработки противодействующих анализу систем защиты ПО, а также способна помочь при разработке новых МПАП.

В ходе выполнения опытно-экспериментальной части работы было выявлено, что рассмотренные в работе типовые МПАП позволяют успешно противодействовать наиболее популярным средствам анализа, таким как OllyDBG, WinDBG, IDA Pro, Syser, Immunity Debugger при настройках, установленных по умолчанию. Однако после установки дополнительных плагинов, скрывающих присутствие отладчика от системы и отлаживаемых процессов, некоторые из методов становятся неэффективными.

Список литературы

1. Пирогов В.Ю. Ассемблер и дизассемблирование. – СПб.: БХВ-Петербург, 2006. – 464 с.
2. Проскурин В.Г. Защита программ и данных – 2-е изд. – М.: Издательский центр «Академия», 2012. – 208с.
3. Скляров Д.В. Искусство защиты и взлома информации – СПб.: БХВ-Петербург, 2004. – 288 с.
4. Falliere N. Windows Anti-Debug Reference.2010.URL. http://www.symantec.com/connect/articles/windows-anti-debug-reference (дата обращения: 03.03.2013).
5. Ferrie P. Anti-unpacker tricks. 2008.URL. http://www.virusbtn.com/pdf/magazine/2008/200812.pdf (дата обращения: 03.03.2013)
6. Ferrie P. Anti-unpacker tricks – part one. Virus Bulletin, December 2008.URL. http://www.virusbtn.com/pdf/magazine/2008/200812.pdf(дата обращения – 03.03.2013)
7. Ferrie P. Anti-unpacker tricks – part two. Virus Bulletin, January 2009.URL. http://www.virusbtn.com/pdf/magazine/2009/200901.pdf(дата обращения – 03.03.2013)
8. Ferrie, P. Anti-unpacker tricks – part three. Virus Bulletin, February 2009.URL. http://www.virusbtn.com/pdf/magazine/2009/200902.pdf(дата обращения – 03.03.2013)

9. Ferrie, P. Anti-unpacker tricks – part four. Virus Bulletin, March 2009.URL. http://www.virusbtn.com/pdf/magazine/2009/200903.pdf(дата обращения – 03.03.2013)
10. Dr. Xiang Fu, Malware Analysis Tutorials: a Reverse Engineering Approach 2012.URL. http://fumalwareanalysis.blogspot.ru/p/malware-analysis-tutorials-reverse.html (дата обращения: 03.03.2013).