АНТИОТЛАДОЧНЫЕ ТРЮКИ

Умному не надо, глупый не поймет…

Abstract: Распаковка как искусство - это серьезное испытание для умственных способностей и одна из самых захватывающих игр для ума в области обратной инженерии. В некоторых случаях, реверсер должен хорошо знать внутреннее устройство операционной системы, чтобы обойти или решить очень хитрые уловки, используемые упаковщиками/протекторами. Терпение и сообразительность тоже играют не последнюю роль в распаковке. Это занятие заставляет состязаться друг с другом исследователей, создающих упаковщики и с другой стороны, исследователей, собирающихся обойти эти защиты.

Главная цель этой работы состоит в том, чтобы показать методы защиты от исследования, используемые упаковщиками/протекторами, а также обсудить методы и доступные инструменты, которые могут использоваться, чтобы обойти или снять это защиты. Эта информация позволит исследователям, особенно, malware-аналитикам идентифицировать эти методы используемые в упакованном вредоносном программном коде, и затем обойти их и приступить к обычному анализу. Представленная информация может также использоваться разработчиками, которые планируют усилить защиту в их программном обеспечении и запретить реверсерам анализировать их защищенный код. Хотя, конечно, квалифицированного, знающего и решительно настроенного реверсера вряд ли что-то остановит  .

Эта статья, по сути, является переводом и компиляцией всего, что было мною найдено в сети по теме антиотладки. Писалось и переводилось все это в первую очередь для себя, но потом решил выложить сей труд для всеобщего обозрения. Матерому реверсеру вряд ли будет интересно это читать, поскольку никаких приватных трюков здесь не описано, а новичкам (каким я сам и являюсь) возможно, будет познавательно узнать, как ушлые шароварщики защищают свои творения от таких как мы.

Ключевые слова: reverse engineering, упаковщики, протекторы, антиотладка

СОДЕРЖАНИЕ

Введение

В области обратной инженерии упаковщики - одна из самых интересных загадок. В процессе решения этих головоломок реверсер получает ценные знания о куче разных вещей, таких как внутренности операционной системы, различные хитрые уловки, инструменты и методы исследования.

Упаковщики (термин, используемый в этой работе и для упаковщиков и для протекторов), создаются, чтобы защитить исполняемые файлы от анализа. Они используются законно коммерческими приложениями, чтобы предотвратить раскрытие кода, его изменение и пиратство. К сожалению, malware-программы также используют упаковщики по тем же самым причинам, но в злонамеренных целях.

Из-за большого количества упакованной малвари, исследователи и вирусные аналитики начали осваивать разные трюки, чтобы распаковывать образцы для анализа. Однако, поскольку время идет, новые методы защиты от анализа постоянно добавляются в упаковщики, чтобы препятствовать анализу и распаковке. И этот круг продолжается - придумываются и развиваются новые методы защиты, в то время как реверсеры с другой стороны баррикад придумывают способы и инструменты, чтобы обойти их.

Главная задача этой работы - показать способы защиты, используемые упаковщиками, а также инструменты и способы обхода/снятия этих защит. Конечно, некоторые упаковщики можно легко обойти, просто сделав дамп процесса, поэтому вся возня с этими защитными механизмами кажется ненужной. Однако, бывают случаи, когда код защиты должен быть полностью исследован и проанализирован, например:

• Части кода защиты должны быть изменены, чтобы получить возможность сделать нормальный дамп процесса и должным образом восстановить таблицу импорта;

• Полный анализ защитного кода, чтобы встроить распаковку в антивирусные программы.

Кроме того, понимание механизмов антиотладки также ценно в случаях, где они непосредственно применяются во вредоносном коде, чтобы предотвратить анализ разрушительных действий.

Антиотладочные приемы могут быть самого различного вида, в зависимости от того, для противостояния какому методу распаковки они предназначены. Распаковка может быть основана на *дампе памяти*, *отладчике*, *эмуляторе*, *буферизации кода*, или *W-X перехвате*.

- *Дампер памяти* делает дамп (выгружает содержимое памяти) текущего процесса без разбора кода в нем.

- *Отладчик* присоединяется к процессу, позволяя выполнение в пошаговом режиме, или размещение точек останова (брейкпоинтов, бряков) в ключевых местах программы, чтобы локализовать интересующие нас места. С помощью отладчика процесс может быть сдамплен с большей точностью, чем одним только дампером.

- *Эмулятор*, в этой работе под эмулятором подразумевается программная среда выполнения, обычно используемая anti-malware программами. Они помещают и запускают файл в собственном пространстве и наблюдают за специфическими действиями, выполняемыми программой.

- *Буфер кода* похож на отладчик. Он также присоединяется к процессу, но вместо того, чтобы выполнять команды процессора в реальном режиме, копирует каждую команду в собственный буфер и выполняет их оттуда. Это дает более точный контроль над выполнением программы. Буферы также более прозрачны чем отладчики, и работают быстрее чем эмулятор.

- *Перехватчик W-X (**W**rite-e**X**ecute)* предназначен для работы со страницами памяти, чтобы наблюдать за последовательностью записи-исполнения. Как правило, исполняемая область имеет атрибуты чтения и выполнения, а все остальные только чтения. Коду позволяют выполняться свободно. Перехватчик обрабатывает исключения, которые вызваны, записью в область с атрибутами

read-only, или выполнением страниц, которые не имеют атрибута executable страниц. Если аппаратные средства позволяют это, read-only страница будет заменена страницей с разрешением на запись, и запись продолжится. Иначе, пошаговое исключение используется, чтобы позволить записи завершиться, после чего страница будет восстановлена к ее исходному состоянию. В любом случае содержимое страницы будет сохранено в списке. В случае исключений, вызванных выполнением невыполнимых или несуществующих страниц, адрес страницы сравнивается со вхождениями в этом списке. Соответствие указывает на выполнение недавно записанного кода, и является расположением точки входа.

Эта работа не претендует на содержание полного списка антиотладочных методов, поскольку она содержит только часто используемые и интересные способы, найденные в упаковщиках. Любознательный читатель может обратиться к последней странице, которая содержит ссылки на информационные ресурсы, где можно узнать больше о других антиотладочных трюках.

Автор надеется, что читатель найдет этот материал полезным и пригодным к использованию. Счастливой распаковки!

1. Антидамп

1.1. SizeOfImage

SizeOfImage - показывает (в байтах), какое количество памяти должно быть выделено в адресном пространстве для загрузки образа исполняемого файла. Это сумма длин всех секций бинарного (исполняемого файла).

Определяется по формуле:
SizeOfImage = VirtualOffset последней секции + VirtualSize последней секции.

Самым простым антидамповым приемом является изменение параметра SizeOfImage в блоке операционного окружения процесса (PEB - process environment block).

Такой прием препятствует доступу к процессу, мешает отладчику присоединиться к процессу и обламывает такие дамперы как LordPE с настройками по умолчанию.
Пример кода:

```
mov eax,  fs:[30h]  ;Указатель на Teb.Peb
mov eax,  [eax+0Ch] ;Указатель на Peb.Ldr - PEB_LDR_DATA
mov eax,  [eax+0Ch] ;Указатель на Ldr.InLoadOrderModuleList.Flink
lea ebx,   [eax+20h] ;LDR_DATA_TABLE_ENTRY.SizeOfImage
add [ebx], 10000h   ;LDR_DATA_TABLE_ENTRY.SizeOfImage + 0x10000
```

Сейчас этот метод используется многими упаковщиками. Однако легко обойти эту уловку можно даже из юзер-модного режима. Мы можем просто проигнорировать значение SizeOfImage, и вместо этого вызывать функцию VirtualQuery(). VirtualQuery) возвращает число последовательных страниц виртуального адресного пространства с одинаковыми атрибутами. В памяти не может быть промежутка между секциями и страницы памяти могут быть подсчитаны, начиная с первой страницы после конца предыдущего диапазона. Подсчет начинается со страницы ImageBase и продолжается пока возвращается тип MEM_IMAGE. Страница, которая не имеет типа MEM_IMAGE, не относится к файлу.

1.2. Очистка заголовка

Некоторые распаковщики исследуют таблицу секций, чтобы собрать интересную информацию об образе. Стирание или изменение этой таблицы в заголовке PE может помешать сбору этой информации. Это, как правило, используется, чтобы противостоять такому инструменту как ProcDump, который полагается на таблицу секций, чтобы сдампить образ.
Пример кода:

```
;получить  image base
push 0
call GetModuleHandleA
push eax
push esp
push 4 ;PAGE_READWRITE
;Округлить в большую сторону размер
страниц
push 1
push eax
xchg edi, eax
call VirtualProtect
xor ecx, ecx
mov ch, 10h ;assume 4kb pages
;сохранить результат VirtualProtect
rep stosb
```

Эту технику использует Yoda's Crypter. Как сказано выше, функция VirtualQuery () может использоваться, чтобы узнать размер образа, и некоторые его свойства (то есть какие страницы -

исполняемые, какие перезаписываемы, и т.д.), но нет никакого способа восстановить таблицу секций, если она была стерта.

1.3. Наномиты (Nanomites)

Наномиты - более продвинутый способ антидампа. Они появились в протекторе Армадилло. Они работают, заменяя команды перехода на команды "int 3", и используя собственные таблицы в ходе распаковки, чтобы определить подробности. Подробностями в этом случае будет - является ли "int 3" наномитом или точкой останова; должен ли переход осуществиться, если это наномит; адрес перехода; и размер команды если переход не осуществился.

Процесс, который защищен наномитами, требует самоотладки (известный как "Debug Blocker" в Армадилле, см. раздел Anti-Debugging:Self-Debugging ниже), который использует копию этого же самого процесса. Это позволяет отладчику перехватывать исключения, которые являются наномитами. Когда исключение происходит в отладчике, он возвращает адрес исключения и ищет его адрес в таблице. Если совпадение имеется, то тип наномита ищется в таблице типов. Если флаги центрального процессора будут соответствовать типу, то переход будет сделан. Когда это происходит, адрес назначения берется из таблицы адресов, и выполнение продолжается с этого адреса. В противном случае, берется размер команды берется из таблицы размера, чтобы перепрыгнуть команду и продолжить выполнение.

1.4. Украденные байты (Stolen Bytes)

Украденные байты - команды, которые вырезаны со своего места и помещены в динамически выделяемую память, где они будут выполняться отдельно. Команда перехода помещена в начало украденных байтов, чтобы указать на перемещенный код. Еще одна команда перехода поставлена в конце перемещенного кода и указывает на конец украденных байтов. Остальная часть команд на месте украденной части кода заменена мусором. Перемещенный код может также быть разбавлен мусорными командами, чтобы сделать более трудным определение реальные команды. Это усложняет восстановление оригинального кода. Такая штука впервые была введена в ASProtect.

Вот пример оригинальной точки входа:

```
004011CB  MOV EAX,DWORD PTR
FS:[0]
004011D1  PUSH EBP
004011D2  MOV EBP,ESP
004011D4  PUSH -1
004011D6  PUSH 0047401C
004011DB  PUSH 0040109A
004011E0  PUSH EAX
004011E1  MOV DWORD PTR
FS:[0],ESP
004011E8  SUB ESP,10
004011EB  PUSH EBX
004011EC  PUSH ESI
004011ED  PUSH EDI
```

Тот же самый код, после упаковки его Enigma Protector. Первые две инструкции были украдены:

```
004011CB  POP EBX
004011CC  CMP EBX,EBX
004011CE  DEC ESP
004011CF  POP ES
004011D0  JECXZ SHORT 00401169
004011D2  MOV EBP,ESP
004011D4  PUSH -1
004011D6  PUSH 0047401C
004011DB  PUSH 0040109A
004011E0  PUSH EAX
004011E1  MOV DWORD PTR
FS:[0],ESP
004011E8  SUB ESP,10
004011EB  PUSH EBX
004011EC  PUSH ESI
004011ED  PUSH EDI
```

А вот типичный пример, в котором несколько инструкций были украдены упаковщиком ASProtect. Он также добавил команду перехода к процедуре, которая выполняет украденные инструкции. Украденные инструкции тогда переплетаются с мусорным кодом, чтобы усложнить восстановление украденных инструкций.

```
004011CB  JMP 00B70361
004011D0  JNO SHORT 00401198
004011D3  INC EBX
004011D4  ADC AL,0B3
004011D6  JL SHORT 00401196
004011D8  INT1
004011D9  LAHF
004011DA  PUSHFD
004011DB  MOV EBX,1D0F0294
004011E0  PUSH ES
004011E1  MOV EBX,A732F973
004011E6  ADC BYTE PTR DS:[EDX-E],CH
004011E9  MOV ECX,EBP
004011EB  DAS
004011EC  DAA
004011ED AND DWORD PTR
DS:[EBX+58BA76D7],ECX
```

1.5. Защищенные страницы (Guard Pages)

Защищенные страницы представляют собой защиту от доступа. Первый раз, когда к защищенной странице обращаются по любой причине, будет сгенерировано исключение EXCEPTION_GUARD_PAGE (0x80000001). Это может использоваться для разных целей, но в целом оно действует как система вызова страниц по требованию кода в ring3. Это достигается вызовом исключения EXCEPTION_GUARD_PAGE (0x80000001), и проверкой находится ли страница в пределах специфического диапазона (например, в пределах образа процесса); и если находится, то страница отображается в некотором пространстве.

Эта методика используется Shrinker, чтобы выполнять распаковку по требованию. Распаковывая только те страницы, к которым происходит обращение, время запуска значительно уменьшено. Потребление памяти может быть снижено, так как страницы, к которым не обращаются, не нуждаются ни в какой физической памяти. Однако, работа приложения в целом может также быть медленнее по сравнению с другими упаковщиками, которые распаковывают приложение полностью. Shrinker работает, перехватывая функцию KiUserExceptionDispatcher(), и следя за исключением EXCEPTION_GUARD_PAGE (0x80000001). Если исключение произойдет в пределах образа процесса, то Shrinker загрузит с диска ту страницу к которой обращаются, распакует ее, и затем продолжит выполнение.

Разновидность этого трюка используется Армадилло, выполняя расшифровку по требованию (известную как "CopyMem2"). Однако, как и с наномитами, это требует использования самоотладки. В отличие от Shrinker, который является полностью самостоятельным, Армадилло распаковывает все страницы в память сразу, вместо того, чтобы подгружать их с диска, когда к ним обращаются. Армадилло использует отладчик, чтобы отлавливать исключения EXCEPTION_GUARD_PAGE (0x80000001). Если исключение произойдет в пределах образа процесса, то Армадилло расшифрует страницу, к которой обращаются, и затем продолжит выполнение.

При выполнении приложений защищенных таким образом, в памяти всегда находится последняя страница, которая была загружена, и очищается когда исключение происходит для другой страницы (если адрес исключения не находится в близком диапазоне). Таким образом, в памяти не может находится одновременно больше чем две страницы. Фактически, Армадилло этого не делает, и это можно считать слабостью в реализации, потому что, просто вызывая все страницы по очереди, Армадилло расшифрует их все, и затем процесс может быть сдамплен полностью.

Защищенные страницы могут использоваться и для простого обнаружения отладчика. Сначала регистрируется обработчик исключений, Динамически выделяется страница памяти с атрибутами executable/writable, в нее записывается код операции "C3" (машинная команда "RET"), а затем защита страницы изменяется на PAGE_GUARD. После этого делается попытка выполнить машинную команду. Это должно привести к исключению EXCEPTION_GUARD_PAGE (0x80000001), которое передается в обработчик исключений, но если присутствует отладчик, он может прервать исключение и позволить программе продолжаться. Это справедливо для такого отладчика как OllyDbg (см. раздел Anti-debugging: OllyDbg ниже).

Пример кода:

```
    xor   ebx, ebx
    push 40h ;PAGE_EXECUTE_READWRITE
    push 1000h ;MEM_COMMIT
    push 1
    push ebx
    call VirtualAlloc
    mov   b [eax], 0c3h
    push eax
    push esp
    ;PAGE_EXECUTE_READWRITE
    ;+ PAGE_GUARD
    push 140h
    push 1
    push eax
    xchg ebp, eax
    call VirtualProtect
    push offset l1
    push dw fs:[ebx]
    mov   fs:[ebx], esp
    push offset being_debugged
    ; если RET выполнится, то
    ; отладчик присутствует
    jmp ebp
    ; исключение вернется сюда
l1:
```

Эта методика используется PC Guard.

1.6. Импорт

Список импортируемых функций может быть полезным, чтобы получить, по крайней мере, некоторое представление о том, что делает программа. Чтобы предотвратить это, некоторые упаковщики изменяют таблицу импорта после того, как импорт был использован. Изменение, как правило, заключается в полном стирании таблицы импорта, но есть разновидности метода, которые изменяют адреса импорта так, чтобы они указывали на скрытый буфер, который выделяется динамически. В буфере находится переход к реальному адресу функции. Этот буфер обычно не

дампится по умолчанию, так что, когда процесс завершается, информация о реальных адресах функций будет потеряна.

Переадресация API – хороший способ предотвратить восстановление таблицы импорта защищенного файла. Как правило, оригинальная таблица импорта стирается и вызовы API, переадресовываются в процедуру, расположенную в выделенной памяти, а уже из этих процедур происходит вызов оригинальной API.

В этом примере код вызывает API kernel32@CopyFileA():

```
00404F05  LEA EDI,DWORD PTR SS:[EBP-20C]
00404F0B  PUSH EDI
00404F0C  PUSH DWORD PTR SS:[EBP-210]
00404F12  CALL <JMP.&KERNEL32.CopyFileA>
```

Этот вызов является переходом на адрес в таблице импората:

```
004056B8  JMP DWORD PTR DS:[<&KERNEL32.CopyFileA>]
```

Однако, когда ASProtect переадресовывает kernel32@CopyFileA(), стаб заменяется вызовом процедуры из выделенной памяти, которая в конечном счете приводит к выполнению украденных инструкций kernel32@CopyFileA():

```
004056B8  CALL 00D90000
```

Ниже иллюстрация того, как размещаются украденные инструкции. Были скопированы первые 7 инструкций kernel32@CopyFileA(). Дополнительно, код, который вызывает процедуру 0x7C83005E также скопирован. Тогда, управление передается внутрь kernel32.dll в середину kernel32@CopyFileA() через RETN к 0x7C830063:

Украденные байты из kernel32@CopyFileA()

```
0D80003  MOV EDI,EDI
00D80005  PUSH EBP
00D80006  MOV EBP,ESP
00D80008  PUSH ECX
00D80009  PUSH ECX
00D8000A  PUSH ESI
00D8000B  PUSH DWORD PTR SS:[EBP+8]
00D8000E  JMP SHORT 00D80013
00D80011   INT 20
00D80013  PUSH 7C830063 ;return EIP
00D80018  MOV EDI,EDI
00D8001A  PUSH EBP
00D8001B  MOV EBP,ESP
00D8001D  PUSH ECX
00D8001E  PUSH ECX
00D8001F  PUSH ESI
00D80020  MOV EAX,DWORD PTR FS:[18]
00D80026  PUSH DWORD PTR SS:[EBP+8]
00D80029 LEA ESI ,DWORD PTR DS:[EAX+BF8]
00D8002F  LEA EAX,DWORD PTR SS:[EBP-8]
00D80032  PUSH EAX
00D80033  PUSH 7C80E2BF
00D80038  RETN
```

Настоящие байты из kernel32@CopyFileA()

```
7C830053  MOV EDI,EDI
7C830055  PUSH EBP
7C830056  MOV EBP,ESP
7C830058  PUSH ECX
7C830059  PUSH ECX
7C83005A  PUSH ESI
7C83005B  PUSH DWORD PTR SS:[EBP+8]
7C83005E  CALL kernel32.7C80E2A4
7C830063  MOV ESI ,EAX
7C830065  TEST ESI,ESI
7C830067  JE SHORT kernel32.7C8300A6
```

1.7. Виртуальные машины

Виртуальные машины - это возможно верхний предел в антидамповой защите, потому что исполняемый код ни в каком виде не появляется в памяти. Таблица импорта может содержать только

абсолютно необходимые функции (LoadLibrary() и GetProcAddress()), не оставляя подсказки относительно того, что делает программа. Дополнительно псевдокод может быть закодирован таким образом что два внешне идентичных куска кода могут выполнять совершенно разные функции. Эта методика используется в VMProtect.

Сам псевдокод может быть полиморфным, где мусорные команды вставлены в код таким же образом, как часто делается для простого машинного кода. Эта методика используется Themida.

Псевдокод может содержать ловушки антиотладки, такие как проверка определенных ячеек памяти на определенные значения (см. раздел антиотладки ниже). Эта методика используется в HyperUnpackMe2.

Интерпретатор псевдокода может быть запутан таким образом, что разбор его будет весьма и весьма сложен. Эта методика используется Themida и Virtual CPU.

II. АНТИОТЛАДКА

2.1. Поля PEB

Process Environment Block (PEB) это структура, расположенная в пользовательском пространстве, которая содержит данные окружающей среды процесса:

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

```
typedef struct _PEB
{
    BOOLEAN InheritedAddressSpace;
    BOOLEAN ReadImageFileExecOptions;
    BOOLEAN BeingDebugged;
    BOOLEAN Spare;
    HANDLE  Mutant;
    PVOID ImageBaseAddress;
    PPEB_LDR_DATA LoaderData;
     PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    PVOID SubSystemData;
    PVOID ProcessHeap;
    PVOID FastPebLock;
    PPEBLOCKROUTINE FastPebLockRoutine;
    PPEBLOCKROUTINE FastPebUnlockRoutine;
}
```

2.1.i. NtGlobalFlag

В структуре PEB есть поле, которое может использоваться для выявления отладчика. Это поле называется NtGlobalFlag и находится по смещению 0x68 от начала PEB. Значение в этом поле по умолчанию - ноль. Начиная с Windows 2000 и позже, введены специальные значения, которые, как правило, заносятся в это поле, когда запущен отладчик. Проверка этого значения не очень надежный показатель наличия отладчика (тем более, что это роле отсутствует в Windows NT). Однако такую проверку часто используют с целью антиотладки. Поле представлено рядом флагов. Значение, которое предполагает присутствие отладчика, складывается из значения следующих полей и поэтому обычно равно 0x70:

```
FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
FLG_HEAP_ENABLE_FREE_CHECK (0x20)
FLG_HEAP_VALIDATE_PARAMETERS (0x40)
```

Пример неправильного кода для проверки выглядит следующим образом:

```
mov eax, fs:[30h] ;PEB
;проверка NtGlobalFlag
cmp b [eax+68h], 70h
jne being_debugged
```

Эта методика используется ExeCryptor.

Команда "cmp" выше - частая встречаемая ошибка. Она подразумевает, что никакие другие флаги не могут быть установлены, что в принципе неверно. Эти три флага обычно устанавливаются для процесса, который создан отладчиком, но для процесса, к которому отладчик присоединяется, эти флаги не выставляются. Однако, есть три исключения для состояния этих флагов.

Первое исключение - то, что дополнительные флажки могут быть установлены для всех процессов, через системный реестр. Значение строки "GlobalFlag" ключа "HKLM\System\CurrentControlSet\Control\Session Manage".

Второе исключение - то, что всеми флагами может управлять на программно, изменяя значения системного реестра. Изменяемым значением опять же будет "GlobalFlag" (заметьте, что "Windows Anti-Debug Reference" от Николаса Фаллиери неправильно называет это "GlobalFlags") ключа реестра "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<filename>". <filename> должно быть заменено именем исполняемого файла (не DLL), к которому при запуске будут применены флаги. Пустое значение строки "GlobalFlag" не приведет ни к каким изменениям флагов.

Третье исключение - то, что, начиная с Windows 2000 и позже, всеми флагами можно управлять путем загрузки структуры конфигураций (Load Configuration Structure). Загрузка структуры конфигураций существует, начиная с Windows NT, но ее формат не был документирован Microsoft в спецификации PE/COFF до 2006 (и то неправильно). Структура была расширена, в основном, чтобы поддержать безопасную обработку исключений в Windows XP, но содержит два поля, важных и для нас: GlobalFlagsClear и GlobalFlagsSet. Поскольку их названия д вольно прозрачны, понятно что они могут использоваться, чтобы очистить и/или установить любую комбинацию битов в PEB> поле NtGlobalFlag. Флаги сначала сбрасываются полем GlobalFlagsClear, а затем выставляются GlobalFlagsSet. Это означает, что, даже если все флаги заданы полем GlobalFlagsClear, любые флаги, которые заданы полем GlobalFlagsSet, будут все еще установлены. Никакие современные упаковщики не поддерживают эту структуру.

Если FLG_USER_STACK_TRACE_DB (0x1000) задается или значением системного реестра "GlobalFlag", или полем GlobalFlagsSet, FLG_HEAP_VALIDATE_PARAMETERS будет автоматически установлен, даже если он задан в поле GlobalFlagsClear.

Таким образом, правильная реализация этого приема будет такой:

```
mov eax, fs:[30h] ;PEB
mov al, [eax+68h]
;NtGlobalFlag
and al, 70h
cmp al, 70h
je  being_debugged
```

Самый простой метод, чтобы обойти этот трюк - создать пустую строку "GlobalFlag".

2.2. Флаги кучи

Динамическая память - еще одно место где можно обнаружить присутствие отладчика. Основной указатель на динамическую память может быть найден с помощью функции kernel32@GetProcessHeap(). Некоторые упаковщики избегают использовать API и вместо этого считывают данные непосредственно на PEB.

Пример кода:

```
mov eax, fs:[30h] ;PEB
;получить указатель на кучу
mov eax, [eax+18h]
```

Динамическая память имеет два поля, представляющие интерес. PEB> поле NtGlobalFlags является основанием для значений в этих полях. Первое поле (Flags) находится по смещению 0x0c в динамической памяти, второе (ForceFlags) по смещению 0x10 в динамической памяти. Поле Flags показывает настройки, которые используются для текущего блока динамической памяти. Поле ForceFlags показывает настройки, которые будут использоваться для последующих действий с динамической памятью. Значение в первом поле сразу после инициализации по умолчанию равно двум, значение во втором поле - ноль. При выполнении программы под отладчиком эти поля принимают другие значения (Flags обычно принимает значение 0 x50000062, а ForceFlags - 0x40000060), но проверка этих полей не очень надежный вариант. Тем не менее, этот способ применяется довольно часто.

Поля представлены рядом флагов. Значение в первом поле (Flags), которое предполагает присутствие отладчика, состоит из следующих флагов:

HEAP_GROWABLE (0x02)
HEAP_TAIL_CHECKING_ENABLED (0x20)
HEAP_FREE_CHECKING_ENABLED (0x40)
HEAP_SKIP_VALIDATION_CHECKS (0x10000000)
HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)
Пример кода:

```
mov  eax, fs:[30h] ;PEB
;получить указатель на кучу
mov  eax, [eax+18h]
mov  eax, [eax+0ch] ;Flags
dec  eax
dec  eax
jne  being_debugged
```

Значение во втором поле (ForceFlags), которое предполагает присутствие отладчика, состоит из следующих флагов:

HEAP_TAIL_CHECKING_ENABLED (0x20)
HEAP_FREE_CHECKING_ENABLED (0x40)
HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)
Пример кода:

```
mov eax, fs:[30h] ;PEB
;получить указатель на кучу
mov eax, [eax+18h]
cmp [eax+10h], 0 ;ForceFlags
jne being_debuggeddebugged
```

Остальные флаги установлены в полях кучи, если установлен флаг FLG_HEAP_ENABLE_TAIL_CHECK в PEB> NtGlobalFlags. "Свободные" флаги установлены в полях кучи, если установлен флаг FLG_HEAP_ENABLE_FREE_CHECK в PEB> поле NtGlobalFlags. Флаub проверки установлены в полях динамической памяти если установлен флаг FLG_HEAP_VALIDATE_PARAMETERS в PEB> NtGlobalFlags. Однако, флагами динамической памяти можно управлять через значение "PageHeapFlags", тем же самым способом как и "GlobalFlag" описанным выше.

Есть два простых способа обойти этот прием:

- Запустить процесс, и приаттачиться к нему отладчиком.
- Установить NtGlobalFlags для процесса, который мы хотим отладить, через ключ реестра "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options": Создайте раздел с

именем процесса, и в нем строковый параметр "GlobalFlags" с пустым значением.

2.3. Куча

Проблема с простой очисткой флагов динамической памяти состоит в том, что изначально динамическая память будет инициализирована с активными флагами, и некоторые из них могут быть распознаны как обнаруженный отладчик. В любом случае, в конце блока динамической памяти будет одно определенное значение, и одно значение может быть, а может и не быть.

Флажок HEAP_TAIL_CHECKING_ENABLED заставляет последовательность 0xABABABAB всегда появляться дважды точно в конце распределенного блока.

Флажок HEAP_FREE_CHECKING_ENABLED заставляет последовательность 0xFEEEFEEE (или часть ее) появляться, если дополнительные байты заполняют все зарезервированное место до следующего блока.

Пример кода:

```
mov      eax, <heap ptr>
;найти неиспользуемые байты
movzx  ecx, b [eax-2]
movzx  edx, w [eax-8] ;size
sub      eax, ecx
lea      edi, [edx*8+eax]
mov      al, 0abh
mov      cl, 8
repe    scasb
je       being_debugged
```

Такая проверка реализована в Themida.

2.4. Специальные API

2.4.i. IsDebuggerPresent

Функция kernel32@IsDebuggerPresent() была введена в Windows 95. Она возвращает TRUE, если отладчик присутствует. На уровне архитектуры операционной системы это означает установка флага в поле PEB > BeingDebugged, который находится по смещению 0x02 от начала PEB.

Пример кода:

```
call IsDebuggerPresent
test al, al
jne   being_debugged
```

Некоторые упаковщики избегают использовать kernel32@IsDebuggerPresent() и считывают данные непосредственно из PEB.

Пример кода:

```
mov eax, fs:[30h] ;указатель на PEB
cmp b [eax+2], 0 ;проверка флага BeingDebugged
jne   being_debugged   ;если   флаг   установлен,   значит
присутствует отладчик
```

Или в другой вариации:

```
mov    eax, large fs:18h ;получаем указатель на TEB
mov    eax, [eax+30h] ;затем указатель на PEB
movzx  eax,  byte  ptr [ eax+2]  ;в  регистр EAX заносится
значение флага BeingDebugged
retn
```

Так как эти проверки очевидны, упаковщики запутывают путем использования мусорного кода мусора или других способов.

Обойти эту проверку можно только установив флаг PEB > BeingDebugged в FALSE. Чтобы сделать это в отладчике OllyDbg, в окне данных нажать Ctrl+G (Goto Expression), и впечатать fs:[30]. В окне отобразится PEB. Более удобным способом будет поместить контрольную точку на первую команду kernel32@IsDebuggerPresent() и подправить возвращаемое значение на FALSE.

2.4.ii. CheckRemoteDebuggerPresent

Функция kernel32@CheckRemoteDebuggerPresent() имеет следующие параметры:

```
BOOL CheckRemoteDebuggerPresent
(
HANDLE hProcess,
PBOOL pbDebuggerPresent.
)
```

Функция - обертка, вокруг функции ntdll@NtQueryInformationProcess(), которая была введена в Windows XP SP1, для запроса значения, которое существовало начиная с Windows NT. "Remote" означает, что функция обращается к удаленному процессу на той же самой машине. Функция устанавливает значение, на которое указывает pbDebuggerPresent в 0xffffffff, если отладчик присутствует. Механизм работы функции это возвращение значения из ntdll@NtQueryInformationProcess (класс ProcessDebugPort).

Пример кода:

```
push eax
push esp
push -1 ;GetCurrentProcess()
call CheckRemoteDebuggerPresent
pop eax
test eax, eax
jne being_debugged
```

Некоторые упаковщики не используют kernel32@CheckRemoteDebuggerPresent(), и вызывают непосредственно ntdll@NtQueryInformationProcess().

2.4.iii. NtQueryInformationProcess

Функция ntdll@NtQueryInformationProcess() принимает параметры:

```
NTSTATUS NTAPI NtQueryInformationProcess
(
 HANDLE           ProcessHandle,
 PROCESSINFOCLASS ProcessInformationClass,
 PVOID            ProcessInformation,
 ULONG            ProcessInformationLength,
 PULONG                   ReturnLength
)
```

Windows Vista поддерживает 45 классов информации ProcessInformationClass (вместо 38 в Windows XP), но только четыре из них документированы Microsoft. Один из них - ProcessDebugPort дает возможность сделать запрос на существования (не на значение) порта. Возвращаемое значение - 0xffffffff (-1), если процесс отлаживается. Функция проверяет ненулевое состояние поле EPROCESS > DebugPort.

Пример кода:

```
push eax
mov  eax, esp
push 0
push 4;ProcessInformationLength
push eax
push 7 ;ProcessDebugPort
push -1 ;GetCurrentProcess()
call NtQueryInformationProcess
pop  eax
test eax, eax
jne  being_debugged
```

Эта методика используется MSLRH. Так как эта информация исходит из ядра, из юзер-модного режима нет никакой возможности скрыть присутствие отладчика. Однако можно отследить вызов функции NtQueryInformationProcess и поправить возвращаемое значение.

Другое решение состоит в том, чтобы использовать системный драйвер, который перехватывал бы вызов ZwNtQueryInformationProcess.

NtQueryInformationProcess более продвинутый метод защиты от отладки чем, например, CheckRemoteDebuggerPresent или UnhandledExceptionFilter.

2.4.iv. Объекты отладки (Debug Objects)

В Windows XP появились "объекты отладки". Когда начинается сеанс отладки, создается объект, и связанный с ним хендл (дескриптор). Это дает возможность сделать запрос значения этого хендла, используя недокументированный класс ProcessDebugObjectHandle.

Недокументированный класс ProcessDebugFlags возвращает значение бита EPROCESS > NoDebugInherit. То есть, если возвращаемое значение - FALSE, отладчик присутствует.

Пример кода:

```
push eax
mov eax, esp
push 0
push 4 ;ProcessInformationLength
push eax
push 1fh ;ProcessDebugFlags
push -1 ;GetCurrentProcess()
call NtQueryInformationProcess
pop eax
test eax, eax
je being debugged
```

Этот трюк используется в HyperUnpackMe2. Так как эта информация исходит из ядра, из юзер-модного режима нет никакой возможности скрыть присутствие отладчика.

Еще один метод - класс SystemKernelDebuggerInformation, который поддерживается операционной системой ReactOS, но может не поддерживаться некоторыми современными версиями Windows.

Пример кода:

```
push  eax
mov    eax, esp
push  0
push  2 ;ProcessInformationLength
push  eax
;SystemKernelDebuggerInformation
push  23h
push  -1 ;GetCurrentProcess()
call   NtQueryInformationProcess
pop    eax
test  ah, ah
jne    being_debugged
```

Эта методика используется SafeDisc. Так как эта информация исходит из ядра, из юзер-модного режима нет никакой возможности скрыть присутствие отладчика.

Более подробную информацию о DebugObject можно найти в статьях Алекса Ионеску

2.4.v. NtQueryObject

Функция ntdll@NtQueryObject() принимает параметры:
HANDLE Handle,
OBJECT_INFORMATION_CLASS ObjectInformationClass,
PVOID ObjectInformation,
ULONG ObjectInformationLength,
PULONG ReturnLength.

NT-платфомы поддерживают пять классов информации ObjectInformationClass, но только два из них документированы Microsoft. И ни один из них не описывает ObjectAllTypesInformation, который нам больше всего интересен.

Как отмечено выше, когда Windows XP начинается сеанс отладки, создается объект отладки и связанный с ним хендл. Это дает возможность посмотреть список существующих объектов, и проверить число существующих объектов отладки. Эти API поддерживается платформами на основе Windows NT, но только Windows XP и выше возвращают подробные списки объектов отладки.

Пример кода:

```
   xor ebx, ebx
   push ebx
   push esp ;ReturnLength
   ;ObjectInformationlength of 0
   ;to receive required size
   push ebx
   push ebx
   ;ObjectAllTypesInformation
   push 3
   push ebx
   call NtQueryObject
   pop ebp
   push 4 ;PAGE_READWRITE
   push 1000h ;MEM_COMMIT
   push ebp
   push ebx
   call VirtualAlloc
   push ebx
   ;ObjectInformationLength
   push ebp
   push eax
   ;ObjectAllTypesInformation
   push 3
   push ebx
   xchg esi, eax
   call NtQueryObject
   lodsd ;handle count
   xchg ecx, eax
l1: lodsd ;string lengths
   movzx edx, ax ;length
   ;pointer to TypeName
   lodsd
   xchg esi, eax
   ;sizeof(L"DebugObject")
   ;avoids superstrings
   ;like "DebugObjective"
   cmp edx, 16h
   jne l2
   xchg ecx, edx
   mov edi, offset 13
   repe cmpsb
   xchg ecx, edx
   jne l2
   ;TotalNumberOfObjects
   cmp [eax], edx
   jne being_debugged
   ;point to trailing null
l2: add esi, edx
   ;round down to dword
   and esi, -4
   ;skip trailing null
   ;and any alignment bytes
   lodsd
   loop l1
   ...
l3: dw "D","e","b","u","g"
       dw "O","b","j","e","c","t"
```

Так как эта информация исходит из ядра, из юзер-модного режима нет никакой возможности скрыть присутствие отладчика.

2.4.vi. Скрытие потока (NtSetInformationThread)

Начиная с Windows 2000, введены специальные API-функции антиотладки, в форме информационного класса вызывающего HideThreadFromDebugger.

Например, функция ntdll@NtSetInformationThread(), которая является оберткой для ZwSetInformationThread. Вот ее прототип:

```
(
NTSYSAPI NTSTATUS NTAPI NtSetInformationThread
IN HANDLE ThreadHandle,
IN THREAD_INFORMATION_CLASS ThreadInformationClass,
IN PVOID ThreadInformation,
IN ULONG ThreadInformationLength
);
```

Если вызвать эту функцию с параметром ThreadInformationClass равным 0x11 (константа ThreadHideFromDebugger), то поток будет отсоединен от отладчика.

Пример кода:

```
push 0
push 0
;HideThreadFromDebugger
push 11h
push -2 ;GetCurrentThread()
call NtSetInformationThread
```

Когда функция вызывается, поток продолжит выполняться, но отладчик больше не будет получать сообщений, связанных с этим потоком. Если скрыт родительский поток, то не передается даже сообщение о завершении потока. Этот способ используется в HyperUnpackMe2.

2.4.vii. OpenProcess

По умолчанию запущенный процесс не имеет привилегий SeDebugPrivilege. Однако, когда процесс запущен под такими отладчиками как OllyDbg и WinDbg, он получает эту привилегию. Когда процесс получает привилегию SeDebugPrivilege, он получает полный контроль над CSRSS.EXE, даже несмотря на то, что CSRSS.EXE - системный процесс. Причина этого – привилегия SeDebugPrivilege позволяет процессу получить полный доступ к любому другому процессу, независимо от его уровня безопасности. Результат состоит в том, что если отлаживаемое приложение может получить идентификатор процесса для CSRSS.EXE, оно может открыть процесс через функцию kernel32@OpenProcess(). Идентификатор процесса может быть получен с помощью kernel32@CreateToolhelp32Snapshot() или kernel32@Process32Next() и л и ntdll@NtQuerySystemInformation (SystemProcessInformation (5)) (и ntdll@NtQuerySystemInformation() - как и kernel32@CreateToolhelp32Snapshot() эта функция доступна только на NT-платформах).

В качестве альтернативы, Windows XP имеет функцию ntdll@CsrGetProcessId(), которая более проста в использовании.

Пример кода:

```
call CsrGetProcessIdd
push eax
push 00
push 1f0fffh; PROCESS_ALL_ACCESSS
call OpenProcesss
test eax, eax
jne being_debuggedd
```

Это открывает дорогу (игра слов) к отказу в обслуживании на системном уровне, заставляя процесс CSRSS.EXE выполнить недопустимую операцию. Можно или создать поток в недопустимом адресе памяти, или поток, который выполняет бесконечный цикл. Однако, так как мы имеем полный доступ, приложение может создать поток в адресном пространстве CSRSS.EXE, и выполнить некоторые действие, которые приведут к повышению привилегий. Однако, это имеет небольшое

значение, обычно только Администраторы имеют возможность приобрести привилегию отладки, а Администратор и так уже имеет самые высокие привилегии. Эта методика была описана Piotr Bania[vi] в 2005.

И OllyDbg и WinDbg требуют, чтобы процесс имел привилегию отладки, а Turbo Debug не требует. Лучший способ победить этот финт состоит в том, чтобы не требовать привилегий отладки без достаточных оснований, и сохранять ее только пока это действительно необходимо.

2.4.viii. CloseHandle

Если в функцию kernel32@CloseHandle() (или непосредственно в ntdll@NtClose()) передается недопустимый хендл, и никакого отладчика не присутствует, то функция возвращает код ошибки. Однако, если присутствует отладчик, то будет сгенерировано исключение EXCEPTION_INVALID_HANDLE (0xc0000008). Это исключение может быть прервано обработчиком исключений, и является показателем присутствия отладчика.

Пример кода:

```
xor    eax, eax
push   offset being_debugged
push   dw fs:[eax]
mov    fs:[eax], esp
; любое недопустимое значение
; для Vista должно быть выровнено по dword
push   esp
call   CloseHandle
```

Обойти этот способ проще всего на Windows XP, где отладчик может установить FirstHandler векторного обработчика исключений, чтобы скрыть исключение и тихо продолжить выполнение. Конечно, есть проблема хука kernel32@AddVectoredExceptionHandler(), чтобы запретить какому-нибудь другому обработчику усановиться как первому. Однако, это все проще, чем хуки ntdll@NtClose() на Windows NT и Windows 2000, для регистрации структурного обработчика исключений.

2.4.ix. OutputDebugString

OutputDebugString. Эта малоизвестная функция имеет довольно простое назначение - она принимает строковый параметр и выводит его в программу.

Функция kernel32@OutputDebugString() имеет различное поведение, в зависимости от того, присутствует отладчик или нет. Самое заметное различие в поведении то, что вызываемая сразу kernel32@GetLastError() возвратит ноль, если отладчик будет присутствовать.

Пример кода:

```
push   0
push   esp
call   OutputDebugStringA
call   GetLastError
test   eax, eax
je     being_debugged
```

2.4.x. ReadFile

Функция kernel32@ReadFile() может применяться для самомодификации, при чтении содержимого файла в поток кода. Это эффективный метод, чтобы удалить программные точки останова (брейкпоинт), которые мог бы поставить отладчик. Эта методика обсуждалась еще в 1999, но была описано публично Piotr Bania только в 2007.

Пример кода:

```
xor   ebx, ebx
mov   ebp, offset l2
push  104h ;MAX_PATH
push  ebp
push  ebx ;self filename
call  GetModuleFileNameA
push  ebx
push  ebx
push  3 ;OPEN_EXISTING
push  ebx
push  1 ;FILE_SHARE_READ
push  80000000h ;GENERIC_READ
push  ebp
call  CreateFileA
push  ebx
push  esp
; чем больше байт, тем лучше
push  1
push  offset l1
push  eax
; заменить "M"
; в заголовка MZ
l1: int 3
...
l2: db 104h dup (?);MAX_PATH
```

Способ обхода состоит в том, чтобы использовать аппаратные контрольные точки вместо программных.

2.4.xi. WriteProcessMemory

Использование функции kernel32@WriteProcessMemory() – упрощенный вариант kernel32@ReadFile() описанный выше, но он требует, чтобы данные для записи уже присутствовали в памяти процесса..

Пример кода:

```
push  1
push  offset l1
push  offset l2
push  -1 ;GetCurrentProcess()
call  WriteProcessMemory
l1: nop
l2: int    3
```

Эта техника используется NsAnti. Способ обхода состоит в том, чтобы использовать аппаратные контрольные точки вместо программных.

2.4.xii. UnhandledExceptionFilter

Когда возникает исключение, Windows XP SP2, Windows 2003, и Windows Vista обычно совершают следующие действия:

- Управление выполнением передается в векторный обработчик исключений, если таковой имеется.

- Если исключение не обработано, управление передается на верхний обработчик SEH в поток, указанный регистром FS: [0] в потоке, который генерировал исключение. SEH расположены цепочкой и вызываются по очереди, если исключение не обрабатывается предыдущим обработчиком.

- Если исключение не было обработано ни одним из предыдущих обработчиков, вызывается конечного обработчик SEH (установленный системой). Вызывается он при помощи kernel32@UnhandledExceptionFilter(). Эта функция решает, что должно быть сделано в зависимости от того, находится процесс под отладкой или нет. Функция совершает вызов обработчика установленного

kernel32@SetUnhandledExceptionFilter(), но этот вызов не удастся если присутствует отладчик. Вместо этого исключение перейдет к отладчику. Присутствие отладчика определяется функцией ntdll@NtQueryInformationProcess (класс ProcessDebugPort). Так, для приложений, которые не знают о ntdll@NtQueryInformationProcess, пропущенное исключение может использоваться как индикатор отладчика.

- Если процесс не отлаживается, то произойдет вызов определенной пользователем функции (через kernel32@SetUnhandledExceptionFilter).
- Если процесс отлаживается, программа будет закончена.

Ниже пример, в котором первичный фильтр исключения устанавливается, используя SetUnhandledExceptionFilter(), и затем генерируется исключение. Если процесс находится под отладкой, то отладчик получит только второе случайное исключение; иначе, фильтр исключений установит CONTEXT.EIP и продолжит выполнение.

```
;set the exception filter
push  .exception_filter
call    [SetUnhandledExceptionFilter]
mov   [.original_filter],eax
;throw an exception
xor     eax,eax
mov   dword [eax],0
;restore exception filter
push  dword [.original_filter]
call    [SetUnhandledExceptionFilter]
   :::
.exception_filter:
 ;EAX = ExceptionInfo.ContextRecord
mov   eax,[esp+4]
mov   eax,[eax+4]
 …
;set return EIP upon return
add   dword [eax+0xb8],6
 …
;return EXCEPTION_CONTINUE_EXECUTION
mov   eax,0xffffffff
retn
```

Некоторые упаковщики устанавливают фильтр исключений непосредственно через вызов kernel32@BasepCurrentTopLevelFilter, вместо того, чтобы вызывать SetUnhandledExceptionFilter(), это может помешать исследованию, если реверсер ставит бряк на упомянутую API.

2.4.xiii. Блокировка ввода (Block Input)

Функция user32@BlockInput() блокирует сообщения от мыши и клавиатуры в отлаживаемом приложений. Это очень эффективный способ вывести из строя отладчики.

Типичным примером является случай, когда реверсер, устанавливает бряк на GetProcAddress(), затем запускает код распаковки, пока бряк не остановится. Однако, в скрытом в мусоре коде, упаковщик вызывает BlockInput(). И когда бряк на GetProcAddress() останавливается, реверсер внезапно теряет контроль над отладчиком, оставляя его озадаченным тем, что только случилось.

Пример кода:

```
; Block input
push    TRUE
call    [BlockInput]

; ...Unpacking code...

; Unblock input
push    FALSE
call    [BlockInput]
```

Применяется в Yoda's Protector.

2.5. Аппаратные трюки

2.5.i. Очередь предварительной выборки (Prefetch queue)

Рассмотрим этот код:

```
l1: call l3
l2:...
l3: mov al, 0c3h
    mov edi, offset l3
    or ecx, -1
    rep stosb
```

Что случается далее? Ответ зависит от нескольких вещей. Ясно, что код перезаписывает поверх себя новый код, который остановится, как только rep будет перезаписан. Если отладка выполняется в пошаговом режиме то это точно случится. Однако, если отладчика нет, то запись продолжится, пока не произойдет исключение. Какое именно исключение, зависит от размещения памяти в это время.

Если, после кода, будет просто виртуальная область, к которой обратился отладчик, то произойдет исключение ошибки доступа (access violation), и программа завершится, если никакой обработчик исключений не был установлен.

В другом случае, если к виртуальной памяти не обращались, то *rep* остановится. Никаких ***видимых*** исключений не произойдет, но код операции "C3" (машинная команда "RET") будет выполнена, и управление будет возвращено к l2.

Почему? Ответ - очередь предварительной выборки. На x86 процессорах до Pentium очередь предварительной выборки не очищалась автоматически, когда происходила запись в адрес памяти, который соответствовал адресу байтов в очереди предварительной выборки. Однако, очередь очищалась всякий раз, когда происходило исключение, такое как пошаговое исключение, которое используется отладчиками для трассировки кода. Такое поведение учитывало все виды антиотладки, главным образом нацеленных на перезапись выполняемого кода. В отсутствии отладчика очередь предварительной выборки выполнила бы оригинальную машинную команду. В присутствии отладчика, который вызывает пошаговое исключение, очередь будет очищена, и изменение будет принято.

Intel считает такое поведение ошибкой, и это было устранено в Pentium и более поздних центральных процессорах, но с двумя исключениями, которые остаются по сей день: инструкции REP MOVS и REP STOS. Эти две команды центральный процессор все равно кэширует и продолжает выполнять их, даже когда последовательность машинной команды была перезаписана в памяти.

Выполнение продолжается до завершения, или пока не произойдет исключение. В рассмотренном выше случае, происходит исключение, из-за отсутствия страницы, когда значение в EDI достигает зарезервированной страницы памяти. Тогда, центральный процессор очищает и перезагружает очередь предварительной выборки, видит код операции "C3" (машинная команда "RET"), где ранее была машинная команда REP STOS, и выполняет ее.

Эта методика используется Invius.

Другой пример этого трюка не полагается на отсутствие страницы.
Пример кода:

```
l1: mov al, 90h
push 10h
pop ecx
cmov edi, offset l1
rep  stosb
```

Одни разновидности этой уловки содержат машинную команду JMP в пределах измененного диапазона; другие содержат машинную команду JECXZ за пределы измененного диапазона. Они

имеют противоположные эффекты.

В обоих случаях опкод "90" (машинная команда "NOP") в регистре AL используется, чтобы перезаписать REP STOSB и некоторые следующие байты. Неправильная эмуляция (или выполнение в пошаговом режиме) заставит REP завершиться преждевременно, позволяя выполниться командам следующим после машинной команды STOSB. В первом случае машинная команда JMP выполнится выдавая присутствие отладчика. Во втором случае значение в ECX будет нулем, только если REP STOSB завершится удачно. Если машинная команда JECXZ не выполняется, это показывает присутствие отладчика.

Версия с JECXZ этого способа используется Obsidium.

В Pentium Prom введено дополнительное поведение, названное "fast string", которое также поддерживается современными процессорами. Это доступно и для MOVS и для STOS. Требует следующих условий: префикс REP; EDI, выровненный ко множителю 8 байтов (и ESI для MOVS на Pentium 3); ESI и EDI по крайней мере 64 байта для Pentium 4 и более поздних процессоров, и 32 байта для более ранних процессоров) для MOVS; ECX по крайней мере 64, сброшенный D флаг в регистре EFLAGS. Также модельно-зависимые регистры (MSR) должен быть выстановлено соответственно (хотя по умолчанию это так и есть) - или 1A0 бит 0 или 1E0 бит 2. Особенно интересно то, что пошаговое исключение ***не может*** прервать операцию.

2.5.ii. Манипуляция отладочными регистрами (Hardware Breakpoints)

Отладочные регистры (DR0 - DR7) используются, для установки аппаратные бряков. Защита может проверить их и таким образом узнать установлены аппаратные бряки (и , следовательно, идет ли отладка); сбросить их или установить их в специальное значения, используемые проверкой позже. Упаковщик tElock использует регистры отладки, чтобы предотвратить их использование отладчиком.

Из непривилегированного режима к регистрам отладки невозможно получить доступ, используя привилегированную команду 'mov drx...'. Для этого существуют другие пути:

- Можно вызвать исключение, изменить контекст потока (во время исключения, он содержит регистры центрального процессора), и затем возобновить нормальное выполнение с новым контекстом.

```
; set up exception handler
push    .exception_handler
push    dword [fs:0]
mov     [fs:0], esp
; eax will be 0xffffffff if hardware breakpoints are identified
xor     eax,eax
; throw an exception
mov     dword [eax],0
; restore exception handler
pop     dword [fs:0]
add     esp,4
;test if EAX was updated (breakpoint identified)
test    eax,eax
jnz     .breakpoint_found
    :::
.exception_handler
;EAX = CONTEXT record
mov     eax,[esp+0xc]
;check if Debug Registers Context.Dr0-Dr3 is not zero
cmp     dword [eax+0x04],0
```

Также можно не проверяя регистры отладки, потихоньку просто сбросит все аппаратные бряки. В этом случае регистры не проверяются, а просто приравниваются к нулю:

```
push offset handler
push dword ptr fs:[0]
mov fs:[0],esp
xor eax, eax
div eax ;generate exception
pop fs:[0]
add esp, 4
;continue execution
;...
handler:
mov ecx, [esp+0Ch] ;skip div
add dword ptr [ecx+0B8h], 2 ;skip div
mov dword ptr [ecx+04h], 0 ;clean dr0
mov dword ptr [ecx+08h], 0 ;clean dr1
mov dword ptr [ecx+0Ch], 0 ;clean dr2
mov dword ptr [ecx+10h], 0 ;clean dr3
mov dword ptr [ecx+14h], 0 ;clean dr6
mov dword ptr [ecx+18h], 0 ;clean dr7
xor eax, eax
ret
```

- Другой путь состоит в том, чтобы использовать вызовы функций NtGetContextThread и NtSetContextThread syscalls (через kernel32 GetThreadContext и SetThreadContext).

Большинство протекторов использует первый, "неофициальный" путь.

Отладчики могут обойти этот трюк, если отладчик обнуляет значение регистров отладки до возобновления выполнения после исключения. Эта техника используется в ASProtect.

2.5.iii. Счетчик машинных команд

Подсчет машинных команд может быть выполнен обработчиком исключений, если установлены аппаратные бряки на специальных адресах. Когда программа дойдет до такого адреса, будет сгенерировано исключение EXCEPTION_SINGLE_STEP (0x80000004) . Это исключение попадет в обработчик исключений, который может скорректировать указатель на следующую инструкцию, и затем возобновить выполнение программы. Установка бряков требует доступа к контекстной структуре. Такой доступ может быть получен при вызове функции kernel32@GetThreadContext().

Альтернативно, контекстную структуру передают в обработчик исключений, таким образом, при возникновении исключения, контекст может быть получен более запутанным способом. Некоторые отладчики не могут правильно обработать аппаратные бряки, поэтому они не устанавливаются и, соответственно, не могут быть подсчитаны обработчиком исключений.

Примерный код выглядит так:

```
xor  eax, eax
xor  eax, eax
cdq
pushoffset l5
push dw fs:[eax]
mov fs:[eax], esp
int 3
l1: nop
l2: nop
l3: nop
l4: nop
div edx
cmp al, 4
jne being_debugged
...
l5: xor eax, eax
;ExceptionRecord
mov ecx, [esp+4]
;ContextRecord
mov edx, [esp+0ch]
;CONTEXT_Eip
inc b [edx+0b8h]
;ExceptionCode
mov ecx, [ecx]
;EXCEPTION_INT_DIVIDE_BY_ZERO
cmp ecx, 0c0000094h
jne l6
;CONTEXT_Eip
inc b [edx+0b8h]
mov [edx+4], eax ;Dr0
mov [edx+8], eax ;Dr1
mov [edx+0ch], eax ;Dr2
mov [edx+10h], eax ;Dr3
mov [edx+14h], eax ;Dr6
mov [edx+18h], eax ;Dr7
ret
;EXCEPTION_BREAKPOINT
l6: cmp ecx, 80000003h
jne l7
;Dr0
mov dw [edx+4], offset l1
;Dr1
mov dw [edx+8], offset l2
;Dr2
mov dw [edx+0ch], offset l3
;Dr3
mov dw [edx+10h], offset l4
;Dr7
mov dw [edx+18h], 155h
ret
;EXCEPTION_SINGLE_STEP
l7: cmp ecx, 80000004h
jne being_debugged
;CONTEXT_Eax
inc b [edx+0b0h]
ret
```

Эта техника применяется в tELock.

2.5.iv. Время выполнения

Когда отладчик присутствует, и выполняет пошаговую трассировку, есть существенная задержка между выполнением отдельных команд, по сравнению с обычным выполнением. Эта задержка может

быть измерена, при помощи нескольких способов. Эти способы включают в себя команду RDTSC (Read Time-Stamp Counter), функцию kernel32 GetTickCount(), и timeGetTime(). Однако, точность timeGetTime() функция является переменной, в зависимости от того, выполняет ли переход к kernel32 GetTickCount(), делая ее очень ненадежной для измерения маленьких интервалов.

Пример кода для RDTSC:

```
rdtsc
xchg  ecx, eax
rdtsc
sub   eax, ecx
cmp   eax, 500h
jnbe  being_debugged
```

Пример кода для kernel32@GetTickCount():

```
call  GetTickCount
xchg  ebx, eax
call  GetTickCount
sub   eax, ebx
cmp   eax, 1
jnb   being_debugged
```

Пример кода для winmm@timeGetTime():

```
call  timeGetTime
xchg  ebx, eax
call  timeGetTime
sub   eax, ebx
cmp   eax, 10h
jnb   being_debugged:
```

Предыдущие функции никак не задействуют ядро, а отображают счетчик в собственное пространство памяти.

Но существуют и таймеры для привилегированного режима. Это функция kernel32@QueryPerformanceCounter. Эта функция API вызывает ntdll@NtQueryPerformanceCounter, которая в свою очередь является оберткой для ZwQueryPerformanceCounterl.

2.5.v. EIP через исключения

Использование исключений, чтобы изменить значение eip обычная методика среди упаковщиков. Это служит эффективной методикой антиотладки, так как отладчики, как правило, перехватывают некоторые исключения (int 1 и int 3, например). Это также вносит некоторую путаницу, особенно если механизм возникновения исключения не очевиден.

Примерный код выглядит так:

```
   xor  eax, eax
   push offset l3
   push dw fs:[eax]
   mov  fs:[eax], esp
l1: call l1
l2: jmp  l2
l3: pop  eax
   pop   eax
   pop   esp
l4:
```

Выполнится когда-нибудь l2? Нет. В l1 происходит переполнение стека, вызывая исключение и передачу управления к l3. После того, как стек восстановится, выполнение продолжается с l4. Эта методика используется в PECompact.

2.5.vi. Прерывание Int3

Это - классическая антиотладка призванная одурачить простенькие отладчики. Смысл ее заключается во вставке кода операции INT3 в середине правильной последовательности команд. Когда INT3 выполняется, если программа не запущена под отладчиком, то управление передается заранее установленному обработчику исключений, и выполнение программы.

Поскольку инструкция INT3 используются отладчиками, чтобы установить программные бряки, вставляя коды операции INT3 этот прием может использоваться, чтобы обмануть отладчиком и заставить его поверить, чтоэто его бряки. Поэтому управление не будет передано в обработчик исключений, и напрвление программы будет изменено. Отладчики должны следить, где они расставляет программные бряки и избегать попадаться на эту уловку.

Так же заметьте, что INT3 может быть закодирован как 0xCD.
Пример кода:

```
push offset l1
push dword fs:[0]
mov fs:[0], esp
;...
db 0CCh
;if fall here, debugged
;...
l1:...;continue execution
```

2.5.vii. “Ice” breakpoint

Так называемая "точка заморозки" является одной из неописанных машинных команд Intel, код операции которой 0xF1. Она используется для обнаружения трассировки программ.

Выполнение этой машинной команды генерирует исключение SINGLE_STEP. Поэтому, если программа уже трассируется, то отладчик будет думать, что это нормальное исключение, сгенерированное при выполнении машинной команды с установленным флагом трассировки. Установленный обработчик исключений не получит управления, и выполнение программы продолжится не так как ожидается.

Обход этой уловки прост: можно выполнить программу через эту команду, и остановить выполнение в пошаговом режиме уже на ней. Исключение будет сгенерировано, но так как программа в это время не трассируется, отладчик поймет, что необходимо передать управление обработчику исключений.

Пример кода:

```
    push offset l1
    push dword fs:[0]
    mov  fs:[0], esp
    ;...
    db   0F1h
    ;if fall here, traced
    ;...
 l1: ... ;continue
```

2.5.viii. Прерывание 2Dh

Выполнение этого прерывания, если программа не отлаживается, генерирует исключение точки останова. Если программа запущена под отладчиком, и выполняется со сброшенным флагом трассировки, то никакого исключения не произойдет, и выполнение будет продолжаться в обычном режиме. Если программа отлаживается, и команда трассируется, то следующий байт будет пропущен, и выполнение продолжится через команду. Поэтому, использование INT 2Dh может использоваться как эффективная антиотладка и антитрассировка.

Пример кода:

```
    push offset l1
    push dword fs:[0]
    mov  fs:[0], esp
    ; ...
    db   02Dh
    mov  eax, 1 ;anti-tracing
    ; ...
l1: ... ;continue execution
```

2.5.ix. Ctrl-C

Если отлаживается консольная программа, сообщение Ctrl-C вызовет исключение EXCEPTION_CTL_C, тогда как в неотлаживаемой программе это сообщение непосредственно обрабатывается специальным обработчиком.

Пример кода:

```
    push offset l2
    push 1
    call RtlAddVectoredExceptionHandler
    push 1
    push l1
    call SetConsoleCtrlHandler
    push 0
    push CTRL_C_EVENT
    call GenerateConsoleCtrlEvent
    push 10000
    call Sleep
    push 0
    call ExitProcess
l1: ...
    ;check if EXCEPTION_CTL_C, if it is,
    ;debugger detected, should exit process
    ;...
l2: ... ;continue
```

2.5.ix. Popf и флаг трассировки

Флаг трассировки (TF), расположенный в регистре флагов (EFLSGS), управляет выполнением программы. Если этот флаг будет установлен, то после исполнения каждой машинной команды генерируется прерывание SINGLE_STEP (int 01h). Флагом трассировки можно управлять, чтобы мешать трассировке программы. Например, эта последовательность команд установит флаг трассировки:

```
pushf
mov dword [esp], 0x100
popf
```

Если программа будет трассироваться под отладчиком, то это не окажет никакого эффекта на регистр флагов, и отладчик обработает исключение, полагая, что так и должно быть. И обработчик исключений не получит управления. Хитрость этой уловки против трассировщика просто требует исполнения команды pushf в реальном режиме. Большинство отладчиков эмулируют команды pushf и popf и не попадаются на эту уловку. Поэтому более эффективно будет совместить ее со следующим трюком:

2.5.x. Регистр SS

Очень оригинальный антитрассировщик. Его можно встретить в упаковщике MarCrypt. Его мало кто знает, и еще меньше народу его использует.

Трюк состоит из трассировки следующего кода:

```
push ss
pop ss
pushf
nop
```

После выполнения pop ss, следующая машинная команда будет выполнена, но отладчик не остановится на ней, а остановится только на следующей машинной команде (в данном случае nop).

Marcrypt использует эту антиотладку следующим образом:

```
push ss
; junk
pop ss
pushf
; junk
pop eax
and eax, 0x100
or  eax, eax
jnz @debugged
; carry on normal execution
```

Фишка здесь в том, что если этот код трассируется в отладчике, команда popf будет выполнена, но отладчик на ней не остановится, и будет не в состоянии сбросить флаг трассировки в значение, помещенном в стек. Защита проверяет флаг трассировки и завершает программу, если его находит.

Простой способ обойти это состоит в том, чтобы установить бряк на popf и выполнить программу (чтобы избежать установки флага трассировки).

2.6. Уловки процесса

2.6.i. Точка входа заголовка

Любая секция файла, атрибуты которой не включают IMAGE_SCN_MEM_WRITE (перезаписываемый) и/или IMAGE_SCN_MEM_EXECUTE (исполняемый), в отладчике будут доступны только для чтения. Это включает и заголовок PE (есть исключения, которые будут разобраны в разделе Anti-Emulating:File-Format ниже). Если точка входа, будет находится в таком разделе, то отладчик будет не в состоянии установить бряки, если сначала не вызовет kernel32@VirtualProtectEx(), чтобы сделать область памяти доступной для записи. Далее, если установить бряк не получилось, то отладчик не замечая этого, выполняет программу дальше. Дело обстоит так для Turbo Debugger. Эта методика используется MEW.

2.6.ii. Родительский процесс

Пользователи обычно запускают приложения вручную через окно, предоставленное оболочкой. В результате родительским процессом любого такого процесса будет Explorer.exe. Конечно, если приложение будет выполняться из командной строки, то командная строка будет родителем. Выполнение приложений из командной строки может быть проблемой для некоторых упаковщиков. Вот почему некоторые упаковщики проверяют имя родительского процесса, ожидая, что это будет "Explorer.exe", или сравнивают идентификатор родительского процесса (process ID) с Explorer.exe. Несоответствие в любом случае предполагает наличие отладчика, создавшего процесс.

Идентификатор процесса для Explorer.exe, и для родителя текущего процесса, может быть получен функциями kernel32@CreateToolhelp32Snapshot() и kernel32@Process32Next().

Примерный код выглядит так:

```
   xor  esi, esi
   xor  edi, edi
   push esi
   push 2 ;TH32CS_SNAPPROCESS
   call CreateToolhelp32Snapshot
   mov  ebx, offset l5
   push ebx
   push eax
   xchg ebp, eax
   call Process32First
l1: call GetCurrentProcessId
   ;th32ProcessID
   cmp  [ebx+8], eax
   ;th32ParentProcessID
   cmove edi, [ebx+18h]
   test esi, esi
   je   l2
   test edi, edi
   je   l2
   cmp  esi, edi
   jne  being_debugged
l2: lea  ecx, [ebx+24h] ;szExeFile
   push esi
   mov  esi, ecx
l3: lodsb
   cmp  al, "\"
   cmove ecx, esi
   or   b [esi-1], " "
   test al, al
   jne  l3
   sub  esi, ecx
   xchg ecx, esi
   push edi
   mov  edi, offset l4
   repe cmpsb
   pop  edi
   pop  esi
   ;th32ProcessID
   cmove esi, [ebx+8]
   push ebx
   push ebp
   call Process32Next
   test eax, eax
   jne  l1
   ...
l4: db      "explorer.exe "
   ;sizeof(PROCESSENTRY32)
l5: dd      128h
   db      124h dup (?)
```

Эта методика используется Yoda's Protector. Так как происходит обращение к ядру, нет простых способов противостоять обнаружению отладчика из юзер-модного режима. Однако, обычным способом будет заставить kernel32 Process32Next() вернуть значение FALSE, которое заставляет цикл завершиться. Должно вызывать подозрение, если Explorer.exe или текущие процессы не были найдены, но Yoda's Protector (и некоторые другие упаковщики) не содержит такой проверки.

Идентификатор процесса для Explorer.exe, и для родителя текущего процесса, может быть получен также функцией ntdll@NtQuerySystemInformation (SystemProcessInformation (5)).

Примерный код выглядит так:

```
   xor  ebp, ebp
   xor  esi, esi
   xor  edi, edi
   jmp  l2
l1: push 8000h ;MEM_RELEASE
   push esi
   push ebx
   call VirtualFreee
l2: xor  eax, eax
   mov  ah, 10h ;MEM_COMMIT
   add  ebp, eax ;4kb increments
   push 4 ;PAGE_READWRITE
   push eaxapush    ebp
   push esi
   call VirtualAlloc ;function does not return
   ;required length for this class
   push esi ;must calculate by brute-force
   push ebp
   push eax ;SystemProcessInformation
   push 5
   xchg ebx, eax
   call NtQuerySystemInformation ;STATUS_INFO_LENGTH_MISMATCH
   cmp eax, 0c0000004h
   je  l1
l3: call GetCurrentProcessId ;UniqueProcessId
   cmp [ebx+44h], eax ;InheritedFromUniqueProcessIdI
   move edi, [ebx+48h]
   test esi, esi
   je   l4
   test edi, edi
   je   l4
   cmp  esi, edi
   jne  being_debugged
l4: mov  ecx, [ebx+3ch];ImageNamemjecxz  l6
   push esi
   xor  eax, eax
   mov  esi, ecx
l5: lodsw
   cmp  eax, "\"
   cmove ecx, esi
   push ecx
   push eax
   call CharLowerW
   mov  w [esi-2], ax
   pop  ecx
   test eax, eax
   jne  l5
   sub  esi, ecx
   xchg ecx, esi
   push edi
   mov  edi, offset l7
   repe cmpsb
   pop  edi
   pop  esi ;UniqueProcessId
   cmove esi, [ebx+44h] ;NextEntryOffsete
l6: mov ecx, [ebx]
   add  ebx, ecx
   inc  ecx
   loop l3
   ...
   l7: dw "e","x","p","l","o","r"
   dw "e","r",".","e","x","e",0 0
```

Однако, идентификатор процесса Explorer.exe может быть получен более просто через user32@GetShellWindow() и user32@GetWindowThreadProcessId(). Идентификатор процесса родителя текущего процесса может быть получен через ntdll@NtQueryInformationProcess (ProcessBasicInformation (0)).

Примерный код выглядит так:

```
    call  GetShellWindow
    push  eax
    push  esp
    push  eax
    call  GetWindowThreadProcessId
    push  0
    ;sizeof(PROCESS_BASIC_INFORMATION)
    push  18h
    mov    ebp, offset l1
    push  ebp
    push  0
    ;ProcessBasicInformation
    push  -1
    ;GetCurrentProcess()
    call  NtQueryInformationProcess
    pop    eaxl
    ;InheritedFromUniqueProcessId
    cmp [ebp+14h], eax
    jne    being_debugged
    ...
    ; sizeof ROCESS_BASIC_INFORMATION)
l1: db 18h dup (?)
```

2.6.iii. Самовыполнение (Self-execution)

Один из самых простых способов сбежать из-под управления отладчиком, это запуск другой своей копии. Как правило, процесс использует синхронизацию объектов, такую как взаимное исключение (mutex) для предотвращения бесконечного выполнения. Первый процесс создаст взаимное исключение, и затем запускает вторую копию процесса. Второй процесс не будет виден отладчиком, даже если первый процесс отлаживается. Также он будет знать, что он - копия, так как существует взаимное исключение.

Примерный код выглядит так:

```
    xor ebx, ebx
    push  offset l2
    push  eax
    push  eax
    call  CreateMutexA
    call  GetLastError
    ;ERROR_ALREADY_EXISTS
    cmp   eax, 0b7h
    je    l1
    mov   ebp, offset l3
    push  ebp
    call  GetStartupInfoA
    call  GetCommandLineA
    ;sizeof(PROCESS_INFORMATION)
    sub   esp, 10h
    push  esp
    push  ebp
    push  ebx
    push  ebx
    push  ebx
    push  ebx
    push  ebx
    push  ebx
    push  eax
    push  ebx
    call  CreateProcessA
    pop   eax
    push  -1 ;INFINITE
    push  eax
    call  WaitForSingleObject
    call  ExitProcess
l1: ...
l2: db "my mutex", 0
    ;sizeof(STARTUPINFO)
l3: db 44h dup (?)
```

Частая ошибка - использование функции kernel32@Sleep(), вместо kernel32@WaitForSingleObject(), потому что это приводит к состоянию гонки. Проблема возникает, когда работа центрального процессора достаточно интенсивная. Это может быть из-за достаточно сложной защиты (или намеренных задержек) во втором процессе; но также и из-за действий, которые может выполнить пользователь, в то время как происходит выполнение, например серфинг по сети или извлечение файлов из архива. Результат в том, что второй процесс не может достигнуть проверки взаимного исключения перед окончанием задержки; и после опережения начинает думать, что он - первый процесс. После этого он запустит еще одну копию процесса. Такое может повторяться любое число раз, пока один из процессов не завершит проверку взаимного исключения успешно. Такая техника используется в MSLRH, и эта проблема там присутствует.

2.6.iv. Имя процесса

Как отмечалось выше, список имен процессов может быть получен функцией kernel32@CreateTool32Snapshot(), или ntdll@QuerySystemInformation(). В дополнение к обнаружению Explorer.exe или текущего имени процесса, некоторые упаковщики ищут другие процессы, особенно те, которые принадлежат anti-malware или специальным программам для борьбы с ними.

Пример кода для kernel32@CreateToolhelp32Snapshot():

```
   push 0
   push 2 ;TH32CS_SNAPPROCESS
   call CreateToolhelp32Snapshot
   mov  ebx, offset l5
   push ebx
   push eax
   xchg ebp, eax
   call Process32First
l1: lea ecx, [ebx+24h] ;szExeFile
   mov  esi, ecx
l2: lodsb
   cmp  al, "\"
   cmove ecx, esi
   or   b [esi-1], " "
   test al, al
   jne  l2
   sub  esi, ecx
   xchg ecx, esi
   mov  edi, offset l4
l3: push ecx
   push esi
   repe cmpsb
   je   being_debugged
   mov  al, " "
   not  ecx
   ;move to previous character
   dec  edi
   ;then find end of string
   repne scasb
   pop  esi
   pop  ecx
   cmp  [edi], al
   jne  l3
   push ebx
   push ebp
   call Process32Next
   test eax, eax
   jne  l1
   ...
l4: <array of space-terminated ASCII strings, space
to end>
   ;sizeof(PROCESSENTRY32)
l5: dd     128h
   db   124h dup (?)
```

Пример кода для ntdll@NtQuerySystemInformation():

```
    xor  ebp, ebp
    xor  esi, esi
    jmp  l2
l1: push    8000h ;MEM_RELEASE
    push esi
    push ebx
    call VirtualFree
l2: xor eax, eax
    mov  ah, 10h ;MEM_COMMIT
    add  ebp, eax ;4kb increments
    push  4 ;PAGE_READWRITE
    push  eax
    push  ebp
    push  esi
    call  VirtualAlloc
    ;function does not return
    ;required length for this class
    push  esi
    ;must calculate by brute-force
    push  ebp
    push  eax
    ;SystemProcessInformation
    push  5
    xchg  ebx, eax
    call  NtQuerySystemInformation
    ;STATUS_INFO_LENGTH_MISMATCH
    cmp   eax, 0c0000004h
    je    l1
l3: mov   ecx, [ebx+3ch] ;ImageName
    jecxz l6
    xor   eax, eax
    mov   esi, ecx
l4: lodsw
    cmp   eax, "\"
    cmove ecx, esi
    push  ecx
    push  eax
    call  CharLowerW
    mov   w [esi-2], ax
    pop   ecx
    test  eax, eax
    jne   l4
    sub   esi, ecx
    xchg  ecx, esi
    mov   edi, offset l7
l5: push ecx
    push  esi
    repe  cmpsb
    je    being_debugged
    not   ecx
    ;move  to previous character
    dec   edi
    ;force word-alignment
    and   edi, -2
    ;then  find end of string
    repne scasw
    pop   esi
    pop   ecx
    cmp   [edi], ax
    jne   l5
    ;NextEntryOffset
l6: mov  ecx, [ebx]
    add   ebx, ecx
    inc   ecx
    loop  l3
    ...
    ;must be word-aligned
    ;for correct scanning align 2
l7: <Unicode строка, с завершающим нулем на конце>
```

2.6.v. Потоки

Потоки используются некоторыми упаковщиками, чтобы выполнить такие действия как периодическая проверка присутствия отладчика, или обеспечение целостности основного кода. У использования потоков есть дополнительное преимущество, в том, что некоторые anti-malware эмуляторы не поддерживают потоки, позволяя упакованному файлу самовольно завершиться.

Примерный код выглядит так:

```
l1: xor eax, eax
    push eax
    push esp
    push eax
    push eax
    push offset l2
    push eax
    push eax
    call CreateThread
    …
l2: xor eax, eax
    cdq
    mov ecx, offset l4 - offset l1
    mov esi, offset l1
l3: lodsb
    ;simple sum
    ;to detect breakpoints
    add  edx, eax
    loop l3
    cmp  edx, <checksum>
    jne  being_debugged
    ;small delay then restart
    push 100h
    call Sleep
    jmp  l2
l4: ;code end
```

Такой прием имеется в PE-Crypt32.

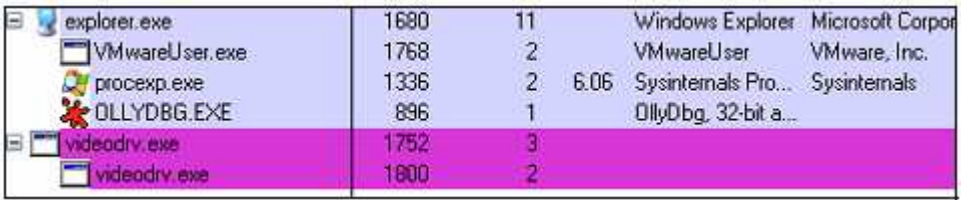
2.6.vi. Самоотладка (Self-debugging)

Самоотладка – это запуск копии процесса, и присоединение к нему этого же процесса в качестве отладчика. Эта уловка была введена в Armadillo, и препятствует присоединению отладчика к процессу. Так как только один отладчик может быть присоединен к процессу в один момент, процесс не может быть отлажен обычными средствами.

Пример кода:

```
        xor   ebx, ebx
        mov   ebp, offset l3
        push  ebp
        call  GetStartupInfoA
        call  GetCommandLineA
        mov   esi, offset l4
        push  esi
        push  ebp
        push  ebx
        push  ebx
        push  1 ;DEBUG_PROCESS
        push  ebx
        push  ebx
        push  ebx
        push  eax
        push  ebx
        call  CreateProcessA
        mov   ebx, offset l5
        jmp   l2
l1: push  10002h ;DBG_CONTINUE
        push  dw [esi+0ch] ;dwThreadId
        push  dw [esi+8] ;dwProcessId
        call  ContinueDebugEvent
l2: push  -1 ;INFINITE
        push  ebx
        call  WaitForDebugEvent
        cmp   b [ebx], 5
        ;EXIT_PROCESS_DEBUG_EVENT
        jne   l1
        ...
        ;sizeof(STARTUPINFO)
l3: db   44h dup (?)
        ;sizeof(PROCESS_INFORMATION)
l4: db   10h dup (?)
        ;sizeof(DEBUG_EVENT)
l5: db    60h dup (?)
```

Вот так это выглядит в менеджере процессов:



Так как защищенный процесс уже отлаживается, присоединение отладчика через kernel32@DebugActiveProcess() потерпит неудачу, и ntdll@NtDebugActiveProcess() возвратит STATUS_PORT_ALREADY_SET. На уровне архитектуры Операционной системы это означает, что поле DebugPort структуры EPROCESS уже установлено.

Обойти этот прием проще всего из режима ядра, обнулив поле EPROCESS > DebugPort. После этого к процессу можно присоединить другой отладчик. Также отлаживаемый процесс можно открыть с помощью функции kernel32@OpenProcess(), что позволит внедрить DLL в адресное пространство процесса. Как альтернативу в Windows XP и выше, можно использовать kernel32@DebugActiveProcessStop(), чтобы отсоединить отладчик. Последний прием можно выполнить, присоединив отладчик к родительскому процессу, и поставив бряк на kernel32@WaitForDebugEvent(), дождаться остановки на нем. После этого вызвать функцию DebugActiveProcessStop (ChildProcessPID) и можно присоединять отладчик к процессу.

2.6.vii. Дизассемблирование

Обычной защитой, используемой упаковщиками, является CC-сканирование, нацеленное на обнаружение программных контрольных точек, установленных отладчиком. Если Вы хотите избежать таких неприятностей, Вы можете использовать или аппаратные бряки или использовать модифицированные бряки. Команда CLI (0xFA) является хорошим кандидатом, чтобы заменить классический код операции INT3. У этой машинной команды есть требования для нормальной работы: она генерирует привилегированное исключение, если выполняется программой из ring3, и занимает только 1 байт.

Некоторые упаковщики проверяют не только первые байты API на наличие бряков, но и активно дизассемблируют код функции. Есть несколько причин, почему они делают это. Одна причина - перехват API, после чего некоторые команды копируются в скрытый буфер и выполняются оттуда. Переход помещается в конце этих команд, и указывает на код после последней скопированной машинной команды в оригинальной API. Это обманывает бряки, которые ставятся куда-нибудь в начало кода API.

Другая причина - более надежный поиск бряков. При заданных границах машинных команд нет никакой вероятности появления бряков. Например, 0xb8 0xcc 0x00 0x00 0x00, кажется, содержит бряк (0xcc), но если машинная команда имеет жестко заданную длину в 5 байт, то после дизассемблирования получится - "MOV EAX, 000000CC".

Некоторые распаковщики проверяют, установлен ли брейкпоинт на функции
Пример кода:

```
    push offset l1
    call GetModuleHandleA
    push offset l2
    push eax
    call GetProcAddress
    cmp   b [eax], 0cch
    je     being_debugged
    ...
l1: db "kernel32", 0
l2: db "IsDebuggerPresent", 0
```

Некоторые упаковщики проверяют первый байт функции. Без установленного брейкпоинта он должен быть равен "64".
Пример кода:

```
    push offset l1
    call GetModuleHandleA
    push offset l2
    push eax
    call GetProcAddress
    cmp  b [eax], 64h
    jne  being_debugged
    ...
l1: db "kernel32", 0
l2: db "IsDebuggerPresent", 0
    je   being_debugged
```

В дополнение к поиску бряков некоторые упаковщики ищут обходы. Обходы - команды перехода, которые вставляются как первая машинная команда, указывающая на скрытое местоположение. Код в этом месте, как правило, создает лог-файл вызываемых API, хотя может быть любым. Проблема с обнаружением обходов состоит в том, что под эту категорию попадает любое изменение кода "на лету". Microsoft добавила фиктивную машинную команду ко многим функциям в Windows XP, который позволяет команде перехода быть помещенной чисто (то есть, без беспокойства о границах машинной команды, так как фиктивная машинная команда выполняет необходимое выравнивание). Эта команда перехода указывает на участок кода, который хучит функцию. Если упаковщик обнаружит обход и откажется работать, если обход какого-нибудь вида будет найден, то он также откажется работать, если функция была пропатчена.

Сюда же можно отнести такой способ защиты, как поиск изменений (патчей) с помощью проверки контрольной суммы. С помощью этого способа можно проверить изменен ли какой-нибудь байт в диапазоне адресов (чаще всего в коде защиты), или проверить установлен ли бряк в этом диапазоне. Принцип этого способа довольно простой, но реализация алгоритма проверки может быть довольно сложной и запутанной.

Пример очень простой проверки на изменение кода:

```
mov     esi,Protected_Code_Start
mov     ecx,Protected_Code_End - Protected_Code_Start
xor     eax,eax
.checksum_loop
movzx   ebx,byte [esi]
add     eax,ebx
rol     eax,1
inc     esi
loop    .checksum_loop
cmp     eax,dword [.dwCorrectChecksum]
jne   .patch_found
```

Чтобы не попадаться на такие проверки проще всего использовать железные/аппаратные/hardware бряки.

2.6.viii. Вызов TLS (Thread Local Storage – локальная память потока)

Это - способ, который позволяет выполнить определенный пользователем код перед выполнением основного кода еще до точки входа.

TIs Callbacks – это функции, которые загрузчик вызывает до вызова EP. Директория TLS описывает набор этих функций. Каждая из функций имеет следующий прототип:

```
typedef VOID
(
NTAPI *PIMAGE_TLS_CALLBACK) (
PVOID DllHandle,
DWORD Reason,
PVOID Reserved
   )
```

Этот параметр может принимать одно из следующих значений: DLL_PROCESS_ATTACH, DLL_THREAD_ATTACH, DLL_THREAD_DETACH, DLL_PROCESS_DETACH. Т.е. функция вызывается в зависимости от ситуации. Самое интересное значение DLL_PROCESS_ATTACH – вызов TLS функции, когда процесс только создался – даже до вызова EP.

Пример программы, использующей TLS:

```
format      PE GUI
include     'include\win32a.inc'
entry       $
            invoke ExitProcess,0
            ret
proc        callback,handle,reason,reserved
            cmp     [reason],DLL_PROCESS_ATTACH
            jnz     @f
            invoke  MessageBox,0,0,0,0
@@:         ret
endp
data        9
            dd a ; StartAddressOfRawData;
            dd a ; EndAddressOfRawData
            dd a ; AddressOfIndex
            dd c ; AddressOfCallBacks
a           dd 0 ;
c           dd callback ; Array Of Callbacks
            dd 0        ; NULL - end of Array Of
Callbacks
end data
section '.idata' import data readable

  library kernel,'KERNEL32.DLL',\
          user,'USER32.DLL'

  import kernel,\
         ExitProcess,'ExitProcess'
  import user,\
         MessageBox,'MessageBoxA'
```

TLS может быть идентифицирован, используя инструменты парсинга PE-файлов, такие как pedump. Pedump покажет, присутствует ли секция TLS в исполняемом файле:

```
Data Directory
  EXPORT              rva: 00000000  size: 00000000
  IMPORT              rva: 00061000  size: 000000E0
  :::
  TLS                 rva: 000610E0  size: 00000018
  :::
  IAT                 rva: 00000000  size: 00000000
  DELAY_IMPORT        rva: 00000000  size: 00000000
  COM_DESCRPTR        rva: 00000000  size: 00000000
  Unused              rva: 00000000  size: 00000000
```

Это - методика, которая обсуждалась конфиденциально в 2000, но была продемонстрировано публично Radim Picha несколько позже. Это использовалось ExeCryptor и другими с 2004.

2.6.ix. Имена устройств (Device names)

Инструменты, которые используют драйвера привилегированного режима, нуждаются в способе общения с этими драйверами. Общепринятым способом является использование именованных устройств. Таким образом, если попытаться открыть такое устройство, любая успешная попытка будет означать присутствие драйвера.

Пример кода:

```
    xor     eax, eax
    mov     edi, offset l2
l1: push    eax
    push eax
    Push 3 ;OPEN_EXISTING
    Push eax
    Push eax
    push eax
    push edi
    call CreateFileA
    inc eax
    jne being debugged
    or ecx, -1
    repne scasb
    cmp [edi], al
    jne l1
    ...
l2: <array of ASCIIZ strings, null to end>
```

Типичные имена, которые проверяются:
\\.\SICE
\\.\SIWVID
\\.\NTICE

Эти имена принадлежат отладчику SoftICE. Обратите внимание, что успешное открытие этих устройств не означает что SoftICE активен, а означает только его наличие в системет. Однако, это является достаточным для большинства людей. Первые два драйвера присутствуют на Windows 9x, третий - на Windows NT, но многие участки кода в упаковщиках пишутся методом copy/paste, поэтому имена всех трех драйверов иногда встречаются даже в тех упаковщиках, которые не работают под Windows 9x платформами.

Другими часто проверяемыми устройствами являются:
\\.\REGVXG
\\.\REGSYS

Эти имена принадлежат RegMon. Первое имя для Windows 9x, второе - для Windows NT.

\\.\FILEVXG
\\.\FILEM

Эти имена принадлежат FileMon. Первое имя для Windows 9x, второе - для Windows NT.

\\.\TRW

Это имя принадлежит TRW. TRW – отладчик для Windows 9x, но некоторые упаковщики проверяют его также и на Windows NT.

\\.\ICEEXT

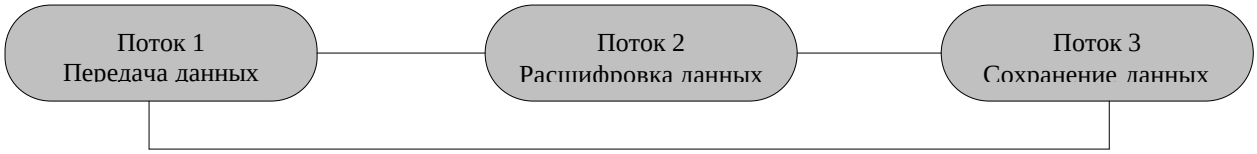
Это имя для дополнительного плагина к SoftICE.

2.6.x. Многопоточность

В многопоточных упаковщиках новый поток обычно создается, чтобы выполнить некоторые операции, например, такие как расшифровка защищенного кода.

Пример многопоточного упаковщика - PECrypt, он использует второй поток, чтобы выполнить декодирование данных, которые были переданы главным потоком, и эти потоки синхронизируются, используя объекты и события.

PECrypt управляет и синхронизирует свои потоки следующим образом:



2.6.xi. SuspendThread

Функция kernel32@SuspendThread() может быть эффективным способом отключения отладчиков непривилегированного уровня, таких как OllyDbg и Turbo Debug. Это может быть достигнуто путем перечисления процессов, как было описано выше, и остановкой основного потока родительского процесса, только если это не "Explorer.exe". Эта методика используется Yoda's Protector.

2.7. Анти-SoftICE

Много лет SoftICE был самым популярным отладчиком для платформы Windows. Это - отладчик, который использует драйвер привилегированного режима, чтобы поддержать отладку из кода и непривилегированного режима и привилегированного режима, включая переходы в любом направлении между ними.

SoftICE содержит много уязвимостей. Описание их выходит за рамки данной статьи. В будущем мы затронем эту тему подробно.

2.7.i. Информация о драйверах

Имена драйверов в системе могут быть перечислены с помощью ntdll NtQuerySystemInformation (SystemModuleInformation (0x0b)). Для каждого модуля, который найден, информация о версии файла может быть получена с помощью VerQueryValue (). Эта информация, как правило, включает названия продукта и авторское право право, которые могут быть направлены против определенных программ и компаний, таких как "SoftICE", "Compuware", и "NuMega".

2.7.ii. Прерывание 1

Прерывание 1 (int1) имеет уровень привилегий (DPL) 0, что означает, что "cd 01" код операции ("int 1") не может быть отслежен из третьего кольца. Попытка выполнить это прерывание приведет к нарушению защиты общего характера (исключение "int 0x0d") и будет отловлено центральным процессором, и в конечном счете приведет к EXCEPTION_ACCESS_VIOLATION (0xc0000005), генерируемому Windows.

Однако, если SoftICE запущен, он перехватывает прерывание 1 и корректирует DPL к 3, так, чтобы SoftICE мог трассировать код из непривилегированного режима. Это происходит внутри SoftICE, хотя команда "IDT", отображающая дескрипторную таблицу прерываний, покажет оригинальный адрес обработчика прерывания 1 с DPL 0, так, как если бы SoftICE не присутствовал.

Проблема состоит в том, что, когда происходит прерывание 1, SoftICE не проверяет, было ли это вызвано флагом трассировки или программным прерыванием. Результат такой, что SoftICE всегда вызывает оригинальный обработчик прерывания 1, и генерирует исключение EXCEPTION_SINGLE_STEP (0x80000004) вместо EXCEPTION_ACCESS_VIOLATION (0xc0000005), применяя простой метод обнаружения.

Пример кода:

```
    xor   eax, eax
    push offset l1
    push dw fs:[eax]
    mov   fs:[eax], esp
    int   1
    ...
    ;ExceptionRecord
l1: mov   eax, [esp+4]
    ;EXCEPTION_SINGLE_STEP
    cmp   dw [eax], 80000004h
    je     being_debugged
```

Эта методика используется SafeDisc. Обойти этот способ кажется очень просто, достаточно восстановить DPL прерывания 1. Однако, это не так просто. Проблема состоит в том, чтобы точно определить причину исключения на уровне прерывания 0x0d. Очередь машинных команд может быть исследована на "int 1", но флаг трассировки, тоже может быть установлен в это же время, даже не являясь активным. Это может случиться, если прерывания задерживаются на одну команду (через "pop ss", например), тогда флаг трассировки не будет причиной исключения, даже если он установлен. В будущем мы постараемся раскрыть несколько дополнительных аспектов этой проблемы.

2.8. Анти-OllyDbg

OllyDbg является самым популярным из отладчиков непривилегированного режима. Имеет поддержку плагинов. Некоторые упаковщики имеют специальную проверку на OllyDbg, поэтому имеется также много плагинов для сокрытия OllyDbg. Соответственно, новые упаковщики имеют проверки на обнаружение этих плагинов. Описание этих плагинов и уязвимостей в них, выходит а рамки данной статьи.

2.8.i. Искаженные файлы

OllyDbg слишком строго относится к формату Portable Executable - он откажется открыть файл, таблица секций которого не заканчиваются точно в конце опционального заголовка. Отладчик пытается выделить количество памяти, указанное в Export Directory Size, Base Relocation Directory Size, Export Address Table Entries, и PE> SizeOfCode , независимо от того, насколько там большие значения. Это может заставить файл подкачки значительно увеличиться, Что окажет существенное влияние на производительность системы.

2.8.ii. Начальное значение esi

Регистр esi имеет начальное значение 0xffffffff при запуске OllyDbg на Windows XP, которое является постоянным, позволяя некоторым разработчикам использовать это как обнаружение отладчика[x]. Фактически, это просто случайность (начальное значение Windows 2000 равно 0). Это значение - остаток структуры обработчика особых ситуаций, которую Windows XP создает во время вызова ntdll@RtlAllocateHeap(). Местоположение этого значения соответствует элементу esi в контекстной структуре, которая создается функцией kernel32@CreateProcess(). kernel32@CreateProcess() не инициализирует esi.

2.8.iii. OutputDebugString

OllyDbg передает заданные пользователем данные непосредственно в msvcrt _vsprintf (). Если эти данные содержат специальные символы форматирования строк, особенно если это несколько символов "%s", то есть вероятность, что один из них укажет на недопустимую область памяти и завершит OllyDbg. Эта ошибка существует в текущей версии OllyDbg (1.10) и возможно будет исправлена в дальнейших версиях.

```
push    .szFormatString
call    [OutputDebugStringA]
    :::
.szFormatString db "%s%s",0
```

2.8.iv. FindWindow

OllyDbg может быть найден через user32 FindWindow (), при нахождении имени класса "OLLYDBG".

Примерный код выглядит так:

```
push  0
push  offset l1
call  FindWindowA
test  eax, eax
jne    being_debugged ...
l1: db       "OLLYDBG", 0
```

2.8.v. Защищенные страницы

OllyDbg использует защищенные страницы, чтобы обнаруживать бряки на память. Как отмечено выше, если приложение помещает выполняемые команды в защищенную страницу, попытка выполнить их приводит к исключению, но в OllyDbg они будут выполняться нормально.

2.8.vi. HideDebugger-specific

HideDebugger это плагин для отладчика OllyDbg. Ранние версии HideDebugger перехватывали вызов kernel32 OpenProcess(). После перехвата совершался прыжок на шестой байт от начала функции kernel32 OpenProcess() function. Таким образом эта функция всегда возвращала отсутствие отладчика при включенном HideDebuggert.

Пример кода:

```
   push  offset l1
   call  GetModuleHandleA
   push  offset l2
   push  eax
   call  GetProcAddress
   cmp    b [eax+6], 0eah
   je       being_debugged
   ...
l1: db       "kernel32", 0
l2: db       "OpenProcess", 0
```

2.9. Анти-ImmunityDebugger

ImmunityDebugger сделан на основе OllyDbg с Python-командным интерфейсом. Фактически это копия OllyDbg. Соответственно он имеет все те же уязвимости и подвержен тем же способам обнаружения и противостояния.

2.10. Анти-WinDbg

2.10.i. FindWindow

WinDbg может быть найден через user32 FindWindow(), при обнаружении имени "WinDbgFrameClass".

Примерный код выглядит так:

```
push  0
push  offset l1
call  FindWindowA
test  eax, eax
jne    being_debugged ...
l1: db  "WinDbgFrameClass", 0
```

2.11. Разные средства

2.11.i. FindWindow

Есть несколько менее распространенных инструментов, которые представляют интерес для упаковщиков, таких как "Import REConstructor v1.6 FINAL (C) 2001-2003 MackT/uCF" или "TESTDBG", "kk1, "Eew57", или "Shadow". Эти имена проверяет MSLRH.

2.11.ii. Антиотладка для Vista

Вот способ антиотладки, специфичный для Windows Vista, который был найден при сравнении дампов памяти программы, работающей с и без управления отладчиком. Может представлять некоторый интерес (проверено на Windows Vista 32 , SP0, английская версия).

Когда процесс отлаживается, TEB его основного потока, по смещению 0xBFC, содержит указатель на юникодовую ссылку на системную dll. Кроме того, строка следует сразу за этим указателем (то есть, расположена по смещению 0xC00 в TEB). Если процесс не отлаживается, указатель устанавливается в NULL, и строка отсутствует.

Пример кода:

```
call GetVersion
cmp al, 6
jne @NotVista
push offset _seh
push dword fs:[0]
mov fs:[0], esp
mov eax, fs:[18h] ; teb
add eax, 0BFCh
mov ebx, [eax] ; pointer to a unicode string
test ebx, ebx ; (ntdll.dll, gdi32.dll,...)
je @DebuggerNotFound
sub ebx, eax ; the unicode string follows the
sub ebx, 4 ; pointer
jne @DebuggerNotFound
;debugger detected if it reaches this point
;...
```

2.11.iii. Альтернативный рабочий стол (Alternative desktop)

NT-подобные платформы поддерживают несколько рабочих столов для одного сеанса. Это дает возможность выбирать разные активные рабочие столы, который буду невидимы для предыдущего рабочего стола, и скрытно переключиться назад на старый рабочий стол.

Пример кода:

```
    xor   eax, eax
    push eax
    ;DESKTOP_CREATEWINDOW
    ;+ DESKTOP_WRITEOBJECTS
    ;+ DESKTOP_SWITCHDESKTOP
    push 182h
    push eax
    push eax
    push eax
    push offset l1
    call CreateDesktopA
    push eax
    call SwitchDesktop
    ...
l1: db "mydesktop", 0
```

Используется в HyperUnpackMe2.

III. Антиэмуляция

3.1. Программные прерывания

3.1.i. Прерывание 3

Когда происходит исключение EXCEPTION_BREAKPOINT (0x80000003), регистр eip уже указывает на следующую машинную команду, таким образом, Windows необходимо поправить eip, чтобы он показывал на правильное место. Проблема состоит в том, что Windows считает, что исключение вызывается одно-байтовым кодом операции "CC" (короткая машинная команда "INT 3"). Если используется "CD 03" (полная машинная команда "INT 3"), чтобы вызвать исключение, то eip будет указывать на неправильное место. То же самое поведение отмечено, если какие-нибудь префиксы помещаются перед краткой машинной командой "INT 3". Эмулятор, который не понимают таких команд, будет обнаружены немедленно. Эта методика используется в TryGames.

3.2. Блокировка времени

Блокировка времени - очень эффективная методика антиэмуляции. Большинство anti-malware эмуляторов преднамеренно ограничивают время и/или количество команд центрального процессора, которые могут быть эмулированы, прежде, чем эмулятор будет обнаружен. Это требование ко всем эмуляторам, так как пользователь, как правило, не будет терпеливо ждать завершения эмулированного приложения (если это вообще произойдет), перед тем как обратиться к нему обычным способом. Это приводит к уязвимости, когда код содержит участок, который преднамеренно задерживает основное выполнение, обычно через фиктивный цикл, вынуждая эмулятор сдаться.

Примерный код выглядит так:

```
     mov ecx, 400000h
l1: loop l1
```

В некоторых случаях, такие фиктивные циклы могут быть распознаны и пропущены, но в этом случае надо проявлять осторожность, чтобы корректировать значения любых внутренних таймеров и регистры процессора, которые используются. Иначе, принудительный пропуск цикла может быть обнаружен.

Примерный код выглядит так:

```
call GetTickCount
xchg ebx, eax
mov  ecx, 400000h
l1: loop l1
call GetTickCount
sub  eax, ebx
cmp  eax, 1000h
jbe  being_debugged
```

Далее, цикл может не быть фиктивным, в том смысле, что его результаты могут использоваться для реальной цели, даже при том, что они, возможно, могут и не иметь отношения к циклу.

Реальный код ситуации выглядит следующим образом:

```
mov ebp, esp
mov ebp, [ebp+1ch] ;0ffffffffh
sub ebp, 5
l1: sub ebp, 0ah
dec eax
or  ebp, ebp
jne l1
```

В этом случае, расчетное значение используется как ключ, таким образом цикл не может быть пропущен. Эта методика используется вTibs.

3.3. Недопустимые параметры API

Много API возвращают коды ошибки, когда получают недопустимые параметры. Проблема для anti-malware эмуляторов состоит в том, что для простоты проверка таких ошибок не осуществляется. Это приводит к уязвимости, когда атакующий преднамеренно передаст заведомо недопустимые параметры в функцию, и ожидает, что вернется код ошибки. В некоторых случаях, этот код ошибки используется как ключ для расшифровки. Любой эмулятор, который не в состоянии возвращать код ошибки, не будет в состоянии расшифровать данные.

Примерный код выглядит так:

```
     push 1
     push 1
     call Beep
     call GetLastError
     ;ERROR_INVALID_PARAMETER (0x57)
     push 5 ;sizeof(l2)
     pop  ecx
     xchg edx, eax
     mov  esi, offset l2
     mov  edi, esi
 l1: lodsb
     xor  al, dl
     stosb
     loop l1
     ...
 l2: db 3fh, 32h, 3bh, 3bh, 38h
     ;secret message
```

Эта методика используется в Tibs.

3.4. GetProcAddress

kernel32 GetProcAddress () функция предназначается, чтобы возвращать адрес функции, экспортируемой указанной библиотекой. Существует потенциально неограниченное число возможных функций, которые могут быть возвращены из бесконечного числа библиотек, и невозможно эмулировать их все. Однако, даже некоторые распространенные функции могут отсутствовать в такой среде, поскольку они необязательны для нормальной работы системы, например такие как kernel32 GetTapeParameters(). Проблема состоит в том, что некоторые упаковщики завершаются, если не могут восстановить адреса всех функций. Чтобы обойти это, некоторые anti-malware эмуляторы всегда возвращают значение для kernel32 GetProcAddress(), независимо от передаваемых параметров. Это приводит к уязвимости, когда атакующий преднамеренно передаст заведомо недопустимые параметры, и ожидает, что никакого адреса функции не вернется. Любой эмулятор, который возвращает адрес в такой ситуации, будет раскрыт.

Примерный код выглядит так:

```
     push offset l1
     push 12345678h ;illegal value
     call GetProcAddress
     test eax, eax
     jne  being_debugged
     ...
 l1: db   "myfunction", 0
```

Эта методика используется в NsAnti. Это - частный случай описанной выше технологии недопустимых параметров API.

3.5. GetProcAddress(internal)

Некоторые anti-malware эмуляторы экспортируют специальные API, которые используются, чтобы общаться с основной средой. Эта методика была описана ранее[xii].

Примерный код выглядит так:

```
   push offset l1
   call GetModuleHandleA
   push offset l2
   push eax
   call GetProcAddress
   test eax, eax
   jne  being_debugged
   ...
l1: db   "kernel32", 0
l2: db   "Aaaaaa", 0
```

3.6. "Новые" команды процессора

У разных эмуляторов процессора есть разные возможности. Проблема для anti-malware эмуляторов состоит в том, что, для простоты, некоторые (а иногда и многие) команды центрального процессора не поддерживаются. Это может включать все классы, такие как сопроцессор, операций с плавающей точкой, *MMX*, и *SSE,* так же как менее обычные команды, такие как CMPXCHG8B. Кроме того, у некоторых команд есть нюансы выполнения, которые трудно учесть в эмулируемой среде, например, как машинная команда CMPXCHG, которая всегда пишет в память, независимо от результата. Многие команды имеют параметры (особенно флаги), которые отмечены как "неопределенные", но нет ничего неопределенного в аппаратных средствах. Процессор должен точно понять алгоритм, чтобы воспроизвести его.

Некоторые упаковщики используют сопроцессор для операций с плавающей точкой и команды *MMX* как пустые команды, но побочным эффектом является то, что anti-malware эмулятор может сдаться, не обнаружив ничего.

3.7. Недокументированные команды

Некоторые упаковщики используют недокументированные команды центрального процессора, по той причине, что они сделаны для современных процессоров. Таким образом, маловероятно, что toanti-malware эмулятор поддержит неизвестные ему команды или недокументированное использование известных команд, таким образом эмулятор сдастся не обнаружив ничего. Ссылка на список таких команд приведена в конце.

3.8. Селекторная проверка

Селекторная проверка используется, чтобы гарантировать, что таблица дескрипторов соответствует операционной системе, например возвращаемое значение функции kernel32@GetVersion(). На Windows 9x-, значение cs -селектора может быть больше 0xff, но на NT-платформах, значение всегда равно 0x1b для 3 кольца.

Примерный код выглядит так:

```
     call GetVersion
     test eax, eax
     ; Windows 9x
     js   l1
     mov  eax, cs
     xor  al, al
     test eax, eax
     jne  being_emulated
l1:  ...
```

Эта методика используется в MSLRH.

3.8.i. Схема размещения памяти

Бесспорно, что оперативная память структурирована, и всегда находится в предсказуемом местоположении. Один из таких участков - RTL_USER_PROCESS_PARAMETERS, при нормальных

обстоятельствах всегда находится по адресу 0x20000. В пределах этой структуры поле "DllPath" расположено по адресу 0x20498, а командная строка по адресу 0x205f8. Эта структура может быть перемещена, если PE > ImageBase равно 0x20000 или меньше. Причина является то, что сначала в память отображаются разделы PE, а только потом окружение (по адресу 0x10000 по умолчанию, и в диапазоне 64 КБ виртуальной памяти), а затем параметры процесса. При доступе к этим полям посредством API, таких как kernel32@GetCommandLine(), они не будут вызываться. Это может помешать собрать определенную информацию, и anti-malware эмуляторы не найдут эти структуры вообще. Эта методика используется в TryGames.

3.9. Трюки с форматом файла

Есть много известных трюков с форматом файла, и все же иногда появляются новые. Это может стать существенной проблемой для anti-malware эмуляторов, до тех пор пока эмулятор опираются на синтаксический анализ формата файла. Несовместимость может появиться из-за различий в эмулированной операционной системе. Например, *Windows 9x*-платформы использует жестко закодированное значение для размера опционального заголовка, и игнорируют поле PE> SizeOfOptionalHeader. Они также допускают промежутки в виртуальной памяти, описанной таблицей разделов. Платформы *на основе Windows NT* основываются на значение в PE> SizeOfOptionalHeader, и не позволяют промежутков.

Типичные уловки:

3.9.i. Невыравненный SizeOfImage

Документация формата файла сообщает, что значение в поле PE> SizeOfImage должно быть кратно значению в поле PE> SectionAlignment, но это не обязательное требование. Вместо этого Windows округлит значение в большую сторону как требуется.

3.9.ii. Перекрытие структур.

При корректировке значений определенных полей возможно получить структуры, которые накладываются на друг друга. Обычные цели – M Z > lfanew, выдает заголовок PE, который расположен в заголовке MZ; поле PE> SizeOfOptionalHeader создает таблицу разделов, которая появляется в массиве DataDirectory; и таблица импорта, чтобы получить таблицу импорта, у которой есть поля в заголовке PE.

3.9.iii. Нестандартный NumberOfRvaAndSizes

Частая ошибочно делают предположение, что значение в поле PE> NumberOfRvaAndSizes устанавливается в значение, которое точно определяет опциональный заголовок, и что таблица раздела следует сразу после него. Правильный метод вычисления местоположения таблицы разделов это использование поля PE> SizeOfOptionalHeader. *И SoftICE* и *OllyDbg попадаются на* эту ошибку.

3.9.iv. Невыравненный SizeOfRawData

Поле SizeOfRawData в таблице раздела - другое поле, которое подлежит автоматическому округлению в большую сторону. Если полагаться на это, то можно создать раздел, точка входа которого, как кажется, будет находиться в виртуальной памяти, но из-за округления, будет иметь физические данные.

3.9.v. Невыравненный PointerToRawData

Поле PointerToRawData в таблице разделов - поле, которое подлежит автоматическому округлению в меньшую сторону. Ели полагаться на это, то возможно, чтобы создать раздел, точка входа которого, будет указывать на другие данные кроме тех, которые будет фактически выполняться.

3.9.vi. Отсутствие таблицы разделов

Интересная вещь получается, если значение в поле PE > SectionAlignment сокращается 4кб или меньше. Обычно, раздел, который содержит заголовок PE, неперезаписываемый и невыполнимый, так как в него нет никакого прямого входа из таблицы разделов. Однако, если значение в PE > SectionAlignment составляет меньше чем 4 КБ, то заголовок PE помечается как перезаписываемый и выполнимый. Также содержание таблицы разделов становятся ***опциональными***. Таким образом, вся таблица разделов может быть обнулена, и файл будет отображен, как если бы это был один раздел,

размер которого равен значению PE> SizeOfImage..

3.9.vii. RVA точки входа равно 0

Некоторые упакованные файлы имеют RVA точки входа 0, что означает, что они начнут выполняться с 'MZ...', который соответствует 'dec ebx / pop edx ...'.

Само по себе это не антиотладка, но может отвлекать, если кто-то захочет остановиться на точке входа при использовании программного бряка.

Если Вы создадите приостановленный процесс, то устанавливая INT3 в RVA 0, Вы сотрете часть значения MZ ('M.'). Это поле уже было проверено, когда процесс был создан, но проверится снова ntdll, когда процесс продолжится (в надежде на достижение точки входа). В этом случае, будет сгенерировано исключение INVALID_IMAGE_FORMAT.

IV. Антипрерывания

4.1. Write> Exec

Некоторые инструментальные средства распаковки работают через прерывание выполнения при записи новых страниц памяти, примерно тогда, когда распаковщик завершит свою основную функцию и передаст управление собственно процессу. При записи и выполнении обманных инструкций распаковщик может принудительно завершить процесс.

Примерный код выглядит так:

```
mov [offset dest], 0c3h
call dest
```

Этот способ используется в ASPack. Однако, по каким-то причинам для многозадачных сред он вызывает очистку очереди процессора.

4.2. Write^Exec

Некоторые инструменты распаковки изменяют атрибуты перезаписываемо-выполнимой страницы на перезаписываемую или выполнимую, но не оба сразу. Эти изменения могут быть обнаружены. Простой метод это использовать функцию, которая использует ядро для записи по указанному адресу непривилегированного режима. Функция возвратит ошибку, если она будет не в состоянии записать что-либо по этому адресу. В этом случае, определяемый адрес, вероятно, принадлежит странице памяти у которой изменены атрибуты. Подходящая функция для этого - kernel32 VirtualQuery().

Примерный код выглядит так:

```
     ;sizeof(MEMORY_BASIC_INFORMATION)
     push  1ch
     mov    ebx, offset l1
     push  ebx
     push  ebx
     call  VirtualQuery
     test  eax, eax
     je       being_debugged
     ...
     ;sizeof(MEMORY_BASIC_INFORMATION)
  l1: db     1ch dup (?)
```

Kernel32@VirtualQuery(), не только делает запись по указанному адресу непривилегированного режима, но также возвращает оригинальное значение атрибутов страницы. Любое изменение в атрибутах - признак наличия перехватчика.

Примерный код выглядит так:

```
      ;sizeof(MEMORY_BASIC_INFORMATION)
      push 1ch
      mov  ebx, offset l1
      push ebx
      push ebx
      call VirtualQuery
      ;PAGE_EXECUTE_READWRITE
      cmp   b [ebx+14h], 40h
      jne   being_debugged
      ;sizeof(MEMORY_BASIC_INFORMATION)
  l1: db  1ch dup (?)
```

Другой способ узнать атрибуты страницы функция kernel32@VirtualProtect(), так как она возвращает предыдущие атрибуты. Любое изменение - признак перехвата.

Примерный код выглядит так:

```
  l1: push eax
      push esp
      push 40h
      push 1
      push offset l1
      call VirtualProtect
      pop  eax
      ;PAGE_EXECUTE_READWRITE
      cmp  al, 40h
      ;PAGE EXECUTE READWRITE
      jne   being_debugged
```

V. Защита от анализа

5.1. Поддельные сигнатуры

Некоторые упаковщики имитируют код других популярных упаковщиков и инструментов, пытаясь ввести в заблуждение распаковщиков. Среди самых популярных из этих поддельных сигнатур код Microsoft Visual C, который был написан, чтобы одурачивать PEiD. Эта методика используется RLPack Professional.

5.2. Шифрование

Шифрование и сжатие - основные формы защиты от анализа. Основное их назначение это сделать так, чтобы реверсер не смог просто загрузить код в отладчик или дизассемблировать и спокойно исследовать его.

Шифрование. Упаковщики обычно шифруют и кодекс(код) защитника(протектора) и защищенное выполнимое. Алгоритмы шифрования очень сильно изменяются от упаковщика к упаковщику, и по уровню сложности могут быть, начиная от очень простых циклов XOR, до очень сложных циклов, которые используют криптографические способы. У полиморфных упаковщиков алгоритм шифрования изменяется для каждого упакованного файла, и процедура декодирования также изменяется для каждого отдельного случая. Это сделано для того, чтобы нельзя было опознать упаковщик и автоматизировать его анализ.

Процедуры расшифровки легко распознаются как циклы, которые выполняют характерные действия с данными. В основном они состоят из команд получения, обработки и сохранения данных. Ниже приведен пример простой процедуры декодирования, которая выполняет несколько операций XOR на зашифрованном значении DWORD.

```
0040A07C  LODS DWORD PTR DS:[ESI]
0040A07D  XOR EAX,EBX
0040A07F  SUB EAX,12338CC3
0040A084  ROL EAX,10
0040A087  XOR EAX,799F82D0
0040A08C  STOS DWORD PTR ES:[EDI]
0040A08D  INC EBX
0040A08E    LOOPD  SHORT  0040A07C
;decryption loop
```

А вот другой пример процедуры декодирования для полиморфного упаковщика:

```
00476056  MOV BH,BYTE PTR DS:[EAX]
00476058  INC ESI
00476059  ADD BH,0BD
0047605C  XOR BH,CL
0047605E  INC ESI
0047605F  DEC EDX
00476060  MOV BYTE PTR DS:[EAX],BH
00476062  CLC
00476063  SHL EDI,CL
::: More garbage code
00476079  INC EDX
0047607A  DEC EDX
0047607B  DEC EAX
0047607C  JMP SHORT 0047607E
0047607E  DEC ECX
0047607F  JNZ 00476056   ;decryption loop
```

И другая процедура расшифровки, сделанная тем же самым полиморфным упаковщиком:

```
0040C045  MOV CH,BYTE PTR DS:[EDI]
0040C047  ADD EDX,EBX
0040C049  XOR CH,AL
0040C04B  XOR CH,0D9
0040C04E  CLC
0040C04F  MOV BYTE PTR DS:[EDI],CH
0040C051  XCHG AH,AH
0040C053  BTR EDX,EDX
0040C056  MOVSX EBX,CL
::: More garbage code
0040C067  SAR EDX,CL
0040C06C  NOP
0040C06D  DEC EDI
0040C06E  DEC EAX
0040C06F  JMP SHORT 0040C071
0040C071  JNZ 0040C045    ;decryption loop
```

В последних двух примерах выделенные строки - основные инструкции декодирования, в то время как остающиеся инструкции - мусорный код, чтобы запутать исследователя. Заметьте, как изменяются регистры и как метод декодирования изменяется в этих двух примерах.

Сжатие. Главная цель сжатия состоит в том, чтобы уменьшить размер выполняемого файла, но поскольку все читаемые строки в файле также сжимаются, то можно в некотором роде считать сжатие частным случаем шифрования. Некоторые примеры алгоритмов сжатия, используемые упаковщиками, - NRV (Not Really Vanished) и LZMA (Lempel-Ziv-Markov chain-Algorithm) для UPX, aPLib для FSG, LZMA для Upack и LZO для yoda's Protector. Некоторые из этих алгоритмов бесплатны для некоммерческого использования, но требуют лицензию/регистрацию для коммерческого использования.

Решение проблемы. Декодирование и распаковку легко обойти, надо только знать, когда цикл

декодирования/декомпрессии заканчивает и затем установить бряк на инструкцию после этих процедур. Помните, некоторые упаковщики могут проверять наличие бряков в этих диапазонах адресов.

5.3. Мусорный код и перестановка (пермутация) кода.

Вставка мусорного кода в процедуру распаковки является эффективным способом запутать анализ кода. Этот трюк предназначен скрывать истинное назначение кода, будь то процедура расшифровки или какая-то защита. Мусорный код повышает эффективность всех остальных уловок, пряча их полезный код между большим количеством бесполезных, ничего не делающих команд. Кроме того, самый эффективный мусор - тот, который выглядит как рабочий код.

Ниже приведена процедура декодирования с несколькими мусорными инструкциями, вставленными между релевантными командами:

```
0044A21A  JMP SHORT sample.0044A21F
0044A21C  XOR DWORD PTR SS:[EBP],6E4858D
0044A223  INT 23
0044A225  MOV ESI,DWORD PTR SS:[ESP]
0044A228  MOV EBX,2C322FF0
0044A22D  LEA EAX,DWORD PTR SS:[EBP+6EE5B321]
0044A233  LEA ECX,DWORD PTR DS:[ESI+543D583E]
0044A239  ADD EBP,742C0F15
0044A23F  ADD DWORD PTR DS:[ESI],3CB3AA25
0044A245  XOR EDI,7DAC77F3
0044A24B  CMP EAX,ECX
0044A24D  MOV EAX,5ACAC514
0044A252  JMP SHORT sample.0044A257
0044A254  XOR DWORD PTR SS:[EBP],AAE47425
0044A25B  PUSH ES
0044A25C  ADD EBP,5BAC5C22
0044A262  ADC ECX,3D71198C
0044A268  SUB ESI,-4
0044A26B  ADC ECX,3795A210
0044A271  DEC EDI
0044A272  MOV EAX,2F57113F
0044A277  PUSH ECX
0044A278  POP ECX
0044A279  LEA EAX,DWORD PTR SS:[EBP+3402713D]
0044A27F  DEC EDI
0044A280  XOR DWORD PTR DS:[ESI],33B568E3
0044A286  LEA EBX,DWORD PTR DS:[EDI+57DEFEE2]
0044A28C  DEC EDI
0044A28D  SUB EBX,7ECDAE21
0044A293  MOV EDI,185C5C6C
0044A298  MOV EAX,4713E635
0044A29D  MOV EAX,4
0044A2A2  ADD ESI,EAX
0044A2A4  MOV ECX,1010272F
0044A2A9  MOV ECX,7A49B614
0044A2AE  CMP EAX,ECX
0044A2B0  NOT DWORD PTR DS:[ESI]
```

А вот, собственно, только рабочий код:

```
0044A225  MOV ESI,DWORD PTR SS:[ESP]
0044A23F  ADD DWORD PTR DS:[ESI],3CB
0044A268  SUB ESI,-4
0044A280  XOR DWORD PTR DS:[ESI],33B
0044A29D  MOV EAX,4
0044A2A2  ADD ESI,EAX
0044A2B0  NOT DWORD PTR DS:[ESI]
```

Пермутация кода - другая техника, используемая более продвинутыми упаковщиками. С кодовой пермутацией простые инструкции переводятся в ряд более сложных инструкций. Это требует, чтобы упаковщик понимал команды процессора и генерировал новый ряд инструкций, которые выполняют эквивалентную операцию.

Простым примером перестановки могут быть следующие инструкции:

```
mov  eax,ebx
test eax,eax
```

После перестановки стали иметь такой вид:

```
push ebx
pop  eax
or   eax,eax
```

Пермутация кода вместе с мусорными командами хороший способ запудрить мозги исследователям.

Ниже приведен код, который выполняет обнаружение отладчика. Но после написания к нему были применены описанные методы:

```
004018A3  MOV EBX,A104B3FA
004018A8  MOV ECX,A104B412
004018AD  PUSH 004018C1
004018B2  RETN
004018B3  SHR EDX,5
004018B6  ADD ESI,EDX
004018B8  JMP SHORT 004018BA
004018BA  XOR EDX,EDX
004018BC  MOV EAX,DWORD PTR DS:[ESI]
004018BE  STC
004018BF  JB SHORT 004018DE
004018C1  SUB ECX,EBX
004018C3  MOV EDX,9A01AB1F
004018C8  MOV ESI,DWORD PTR FS:[ECX]
004018CB  LEA ECX,DWORD PTR DS:[EDX+FFFF7FF7]
004018D1  MOV EDX,600
004018D6  TEST ECX,2B73
004018DC  JMP SHORT 004018B3
004018DE  MOV ESI,EAX
004018E0  MOV EAX,A35ABDE4
004018E5  MOV ECX,FAD1203A
004018EA  MOV EBX,51AD5EF2
004018EF  DIV EBX
004018F1  ADD BX,44A5
004018F6  ADD ESI,EAX
004018F8  MOVZX EDI,BYTE PTR DS:[ESI]
004018FB  OR EDI,EDI
004018FD  JNZ SHORT 00401906
```

А вот код до применения пермутации и замусоривания:

```
00401081  MOV EAX,DWORD PTR FS:[18]
00401087  MOV EAX,DWORD PTR DS:[EAX+30]
0040108A  MOVZX EAX,BYTE PTR DS:[EAX+2]
0040108E  TEST EAX,EAX
00401090  JNZ SHORT 00401099
```

Мусорный код и пермутация действительно способны отнять очень много времени у реверсера. Поэтому при анализе такого кода, важно определить что делает код, спрятанный в этом мусоре (например: выполняет расшифровку, инициализирует распаковку и т.д).

Один из способов избежать трассировку мусорного кода состоит в том, чтобы попытаться установить контрольные точки на API, которые активно используются упаковщиком (например, VirtualAlloc/VirtualProtect/LoadLibrary/GetProcAddress, и т.д.), или можно использовать утилиты следящие за вызываемыми функциями (API-шпионы) и сохраняющими лог. Если что-то пошло не так, как надо (например был обнаружен отладчик или бряки) необходимо анализировать лог и локализовать место неудачи. Дополнительная установка бряков на доступ/запись позволяет реверсеру точно определять, что инструкции пытаются изменить/получить доступ к интересным частям защищенного процесса, вместо того, чтобы трассировать массу кода.

Наконец, запуск OllyDbg под VMWare и при выполнении снимков сеанса отладки (snapshots) позволяет реверсеру возвращаться на определенное состоянии. И если что-то пошло не так, как надо, сессия отладки может вернуться назад к определенному состоянию.

5.4. Защита от дизассемблирования

Другой способ усложнить анализ состоит в том, чтобы запутать дизассемблер. Антиизассемблирование – не самый эффективный способ усложнить процесс исследования, но, те не менее, довольно популярный, и если он объединен с мусорным кодом и пермутацией, это делает его еще более эффективным.

Например, можно вставить один мусорный байт, а затем ввести условный переход, который будет передавать управление на этот мусор. Однако, условие для этого перехода всегда должно быть FALSE. Таким образом, мусор никогда не будет выполняться, но может обмануть дизассемблер, который дизассемблирование с мусорного байта, и в конечном счете приведет к неправильному результату. Другой способ – направить этот прыжок в середину команды. Это также может сбить дизассемблер с толку.

Прежде, чем обсуждать специфичные методики, хотелось бы кратко рассказать о двух подходах к дизассемблированию. Линейное дизассемблирование - тривиальный подход, который просто дизассемблирует машинные команды последовательно во всем файле. Рекурсивное дизассемблирование - более интеллектуальный подход, когда все команды анализируются, выявляются пересекающиеся команды, учитываются все условные и безусловные переходы. Рекурсивные дизассемблеры более надежны и намного более терпимы к различным уловкам антидизассемблирования.

В таблице описана методика дизассемблирования, используемая в самых распространенных инструментах исследования.

DISASSEMBLER/DEBUGGER NAME	DISSASEMBLY METHOD
OllyDbg	Рекурсивный метод
NuMega SoftICE	Линейный метод
Microsoft WinDbg	Линейный метод
IDA Pro	Рекурсивный метод
PEBrowse Professional (включая интерактивную версию)	Рекурсивный метод

Вот пример простой проверки флага PEB.BeingDebugged, с некоторым добавленным кодом. Выделенные строки - главные инструкции, остальные – мусорный код. Вставляется байт 0xff и добавляется поддельный условный переход на этот байт:

```
;Anti-disassembly sequence #1
    push    .jmp_real_01
    stc
    jnc     .jmp_fake_01
    retn
.jmp_fake_01:
    db      0xff
.jmp_real_01:
    ;-------------------------
    mov     eax,dword [fs:0x18]

;Anti-disassembly sequence #2
    push    .jmp_real_02
    clc
    jc      .jmp_fake_02
    retn
.jmp_fake_02:
    db      0xff
.jmp_real_02:
    ;-------------------------
    mov     eax,dword [eax+0x30]
    movzx   eax,byte [eax+0x02]
    test    eax,eax
    jnz     .debugger_found
```

Вот такой листинг выдает WinDbg:

```
0040194a 6854194000      push  0x401954
0040194f f9              stc
00401950 7301            jnb   image00400000+0x1953 (00401953)
00401952 c3              ret
00401953 ff64a118        jmp   dword ptr [ecx+0x18]
00401957 0000            add   [eax],al
00401959 006864          add   [eax+0x64],ch
0040195c 194000          sbb   [eax],eax
0040195f f8              clc
00401960 7201             jb    image00400000+0x1963 (00401963)
00401962 c3              ret
00401963 ff8b40300fb6    dec   dword ptr [ebx+0xb60f3040]
00401969 40              inc   eax
0040196a 0285c0750731    add   al,[ebp+0x310775c0]
```

А вот такой OllyDbg:

```
0040194A   68 54194000      PUSH 00401954
0040194F   F9               STC
00401950   73 01            JNB SHORT 00401953
00401952   C3               RETN
00401953   FF64A1 18        JMP DWORD PTR DS:[ECX+18]
00401957   0000             ADD BYTE PTR DS:[EAX],AL
00401959   0068 64          ADD BYTE PTR DS:[EAX+64],CH
0040195C   1940 00          SBB DWORD PTR DS:[EAX],EAX
0040195F   F8               CLC
00401960   72 01            JB SHORT 00401963
00401962   C3               RETN
00401963   FF8B 40300FB6    DEC DWORD PTR DS:[EBX+B60F3040]
00401969   40               INC EAX
0040196A   0285 C0750731    ADD AL,BYTE PTR SS:[EBP+310775C0]
```

А вот листинг из IDA:

```
0040194A            push    (offset loc_401953+1)
0040194F            stc
00401950            jnb     short loc_401953
00401952            retn
00401953 ; ---------------------------------------------------------------
00401953
00401953 loc_401953:                            ; CODE XREF: sub_401946+A
00401953                                        ; DATA XREF: sub_401946+4
00401953            jmp     dword ptr [ecx+18h]
00401953 sub_401946 endp
00401953
00401953 ; ---------------------------------------------------------------
00401957            db    0
00401958            db    0
00401959            db    0
0040195A            db  68h ; h
0040195B            dd offset unk_401964
0040195F            db 0F8h ; °
00401960            db  72h ; r
00401961            db    1
00401962            db 0C3h ; +
00401963            db 0FFh
00401964 unk_401964 db  8Bh ; i             ; DATA XREF: text:0040195B
00401965            db  40h ; @
00401966            db  30h ; 0
00401967            db  0Fh
00401968             db 0B6h ;
00401969            db  40h ; @
0040196A            db    2
0040196B            db  85h ; a
0040196C            db 0C0h ; +
0040196D            db  75h ; u
```

Заметьте, как все три дизассемблера/отладчика попали в ловушку, которая является очень раздражающей и запутывающей анализ. Есть несколько других способов запутать дизассемблер, и это было только одним примером. Дополнительно, такой код может быть записан в макрос так, чтобы исходные тексты оставались более понятными.

VI. SEH и VEH

А теперь хотелось бы рассказать еще об одной интересной штуке – обработчике прерываний.

Почти все описанные приемы основаны на том, что приложение, нормально работающее в среде WINDOWS, валится под отладчиком, создавая какие-то ошибки. В принципе, этого достаточно, чтобы немного затруднить отладку и взлом программы, но можно поступить немного по-другому. И некоторые так и поступают. Существует способ, при котором программа вызвав исключительную ситуацию (ошибку) не просто валится, а пытается ее обработать и решить проблему. Либо другой вариант: программа, совершив ошибку, а, соответственно, обнаружив что она находится под отладчиком, передает управление в некоторый код, который может обработать и исправить ошибку, а может и сделать еще чего-нибудь. Так что вполне возможно, что Вам когда-нибудь встретится программа, которая после того как ее засунули в отладчик, отформатирует Вам жесткий диск ☺.

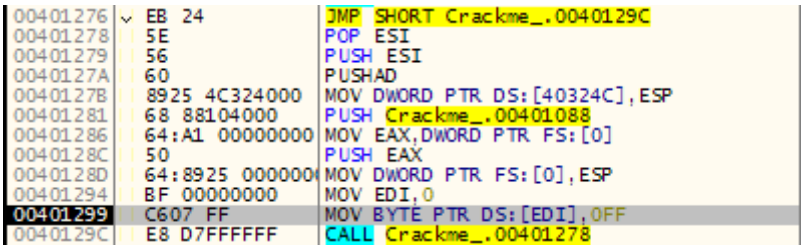
Так вот, попытаюсь объяснить как это делается.

Рассмотрим это на примере крякми, который прилагается к сему документу.

Ну что, начнем обследовать пациента? Запускаем крякми. Видим два эдита для имени и кода. После небольших манипуляций выясняем, что имя должно содержать больше четырех символов; что

крякми ругается, если не заполнено одно из полей; и что при вводе имени и кода (неправильного кода, или неправильного имени) крякми просто закрывается.

Для терапевтического осмотра достаточно. Приглашаем пациента не рентген. Грузим крякми в OllyDebugger (далее просто Оля). Запускаем (F9). Вводим имя и код. Я ввел Name = "Sturgeon" и Code = "1122334455". Нажимаем Try… и останавливаемся в таком месте.



Причем причина остановки прописана в нижней строчки отладчика.

Написано здесь то, что "доступ был запрещен при попытке записи по адресу [00000000]". Смотрим, почему это глупый крякмис лезет туда, куда ему не положено. Ну, собственно, тут все понятно. Вот эти строки кода:

```
00401294 MOV EDI,0                ;Помещаем в EDI 0, а потом
00401299 MOV BYTE PTR [EDI],0FF;пытаемся по адресу [EDI] что-то писать
```

Вот именно поэтому и происходит исключение. Для нас важно то, что это исключение происходит не в результате ошибки или неправильных данных, а сознательно заложено в программу. Следовательно, должна быть предусмотрена и обработка этого исключения.

И действительно, если мы посмотрим чуть выше, то прямо перед вызовом исключения мы видим установку обработчика исключений.

```
00401281    PUSH Crackme_.00401088
00401286    MOV EAX,DWORD PTR FS:[0]
0040128C    PUSH EAX
0040128D    MOV DWORD PTR FS:[0],ESP
```

Вот этот код основа SEH, поэтому я попробую прокомментировать его подробнее. В Win32 регистр FS указывает на структуру, где хранятся адреса обработчиков исключений. И для установки собственного обработчика используется следующая конструкция:

```
00401281 PUSH Crackme_.00401088      ;Установка нового обработчика исключений!!!
00401286 MOV EAX,DWORD PTR FS:[0]    ;Следующие три строчки необходимы для
0040128C PUSH EAX                    ;сохранения старого обработчика
0040128D MOV DWORD PTR FS:[0],ESP    ; исключений
```

То есть, команда PUSH xxxxxxxx перед строкой MOV EAX,DWORD PTR FS:[0] (между ними может быть еще какой-либо код) помещает в стек адрес нового обработчика исключений. Вот это-то нам и важно. Говоря простым языком, это означает, что при возникновении исключения выполнение программы передастся установленному новому обработчику исключений, который будет находиться по адресу Crackme_.00401088.

Теперь, когда мы знаем, где находится новый SEH можно поставить на нем бряк. Что мы и сделаем. Ставим бряк (F2) на адрес Crackme_.00401088. И запускаем программу по Shift+F9. Здесь тоже кое-какая интересная информация. Когда программа работает сама по себе, она обрабатывает свои исключения своими же обработчиками, и не спрашивает ни у кого разрешения. При запуске под отладчиком ситуация немного меняется. Оля (умница) понимает, что произошло исключение и

тормозит программу, выводя сообщение о причине, вызвавшей исключение. Далее по выбору можно позволить обрабатывать исключения либо отладчику (F9), либо самой программе (Shift+F9). Так как Оля обрабатывает исключения SEHами операционной системы, которые не знают про существование хитрых антикрякерских приемов, то ни к чему хорошему это не приводит. Поэтому, нажимая Shift+F9, мы позволяем самому крякми обрабатывать исключения своими SEHами.

Так вот. Ставим бряк на Crackme_.00401088. Shift+F9 и брякаемся на бряке (прошу прощения за тавтологию). Ну что, сработало. Пускаем программу дальше по F9 и… опять тормозим на исключении.

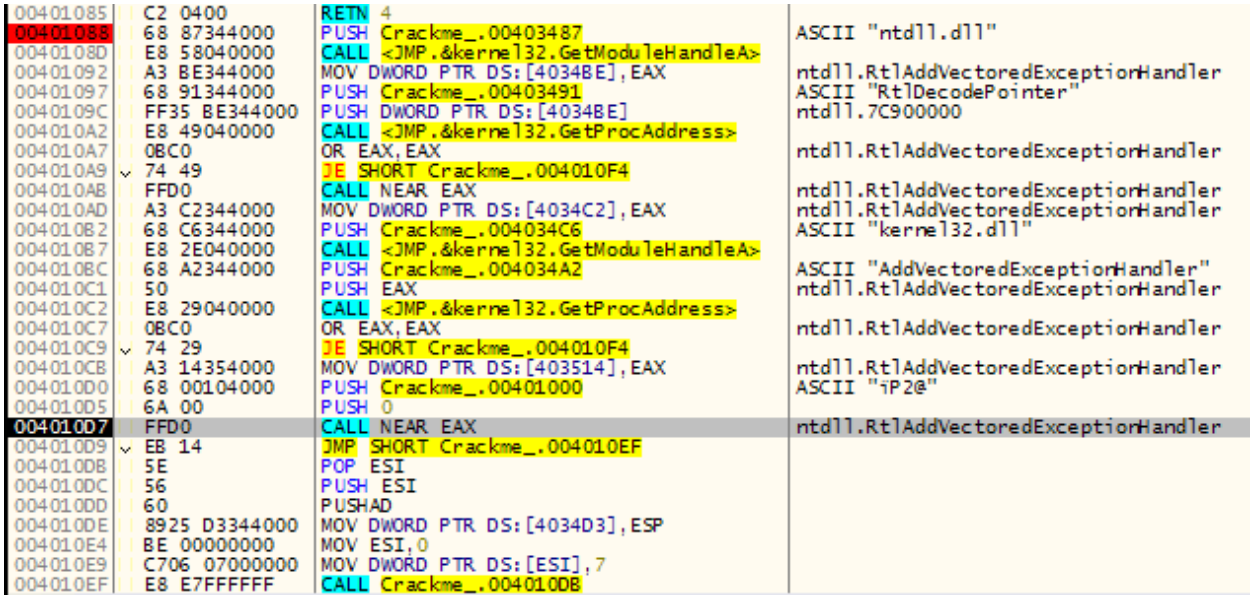
```
00401085   C2 0400          RETN 4
00401088   68 87344000      PUSH Crackme_.00403487                ASCII "ntdll.dll"
0040108D   E8 58040000      CALL <JMP.&kernel32.GetModuleHandleA>
00401092   A3 BE344000      MOV DWORD PTR DS:[4034BE],EAX
00401097   68 91344000      PUSH Crackme_.00403491                ASCII "RtlDecodePointer"
0040109C   FF35 BE344000    PUSH DWORD PTR DS:[4034BE]            ntdll.7C900000
004010A2   E8 49040000      CALL <JMP.&kernel32.GetProcAddress>
004010A7   0BC0             OR EAX,EAX
004010A9   74 49            JE SHORT Crackme_.004010F4
004010AB   FFD0             CALL NEAR EAX
004010AD   A3 C2344000      MOV DWORD PTR DS:[4034C2],EAX
004010B2   68 C6344000      PUSH Crackme_.004034C6                ASCII "kernel32.dll"
004010B7   E8 2E040000      CALL <JMP.&kernel32.GetModuleHandleA>
004010BC   68 A2344000      PUSH Crackme_.004034A2                ASCII "AddVectoredExceptionHandler"
004010C1   50               PUSH EAX
004010C2   E8 29040000      CALL <JMP.&kernel32.GetProcAddress>
004010C7   0BC0             OR EAX,EAX
004010C9   74 29            JE SHORT Crackme_.004010F4
004010CB   A3 14354000      MOV DWORD PTR DS:[403514],EAX
004010D0   68 00104000      PUSH Crackme_.00401000                ASCII "ïP2@"
004010D5   6A 00            PUSH 0
004010D7   FFD0             CALL NEAR EAX
004010D9   EB 14            JMP SHORT Crackme_.004010EF
004010DB   5E               POP ESI
004010DC   56               PUSH ESI
004010DD   60               PUSHAD
004010DE   8925 D3344000    MOV DWORD PTR DS:[4034D3],ESP
004010E4   BE 00000000      MOV ESI,0
004010E9   C706 07000000    MOV DWORD PTR DS:[ESI],7
004010EF   E8 E7FFFFFF      CALL Crackme_.004010DB
```

Причина исключения такая же, как и в первом случае: попытка записи по адресу [00000000].

```
004010E4    MOV ESI,0
004010E9    MOV BYTE PTR DS:[ESI],7
```

А вот установки нового обработчика исключений нигде не видно. И на старый программа тоже не попадает. Вот тут, честно говоря, пришлось покумекать, потому что я раньше этого не знал. Короче, перезапускаем программу, опять тормозим на обработчике по адресу Crackme_.00401088, а дальше пойдем пошагово (F8), и будем смотреть, чего вообще в этом крякми творится.

Пока мы будем проходить этот код можно увидеть, как вызываются API-функции GetModuleHandleA, GetProcAddress, и видно подозрительные строки "AddVectoredExceptionHandler" и "RtlAddVectoredExceptionHandler". Я человек неискушенный в программировании, но, насколько мне известно, такой (или примерно такой) код используется для нахождения адреса нативных функций. А, судя по тому, что присутствуют строки "AddVectoredExceptionHandler" и "RtlAddVectoredExceptionHandler", можно предположить, что их то адреса программа и ищет.



И вот стоим мы на строке

```
004010D7    CALL NEAR EAX     ; ntdll.RtlAddVectoredExceptionHandler
```

Название этих функций говорит само за себя "RtlAddVectoredExceptionHandler" = "Добавить векторный обработчик исключений". Когда-то я читал на BACMe статью про обработку прерываний. Статью, по правде говоря, я не осилил, но где-то в мозгу отложилось, что это одна из новых фишек WindowsXP - векторный обработчик исключений (VEH).

Раньше я не встречал такой функции, поэтому пришлось лезть в MSDN. Скажу сразу, такой функции я не нашел, но немного погуглив, узнал, что RtlAddVectoredExceptionHandler является переходником к AddVectoredExceptionHandler (или наоборот, я так и не понял). То есть возвращаемся к MSDN и ищем AddVectoredExceptionHandler

AddVectoredExceptionHandler перевод мой

Функция **AddVectoredExceptionHandler** регистрирует вектор обработки исключений.

```
PVOID AddVectoredExceptionHandler(
  ULONG FirstHandler,
  PVECTORED_EXCEPTION_HANDLER VectoredHandler
);
```

Параметры

FirstHandler

[in] Порядок, в котором будет вызываться новый обработчик. Если параметр не равен нулю, обработчик будет вызван первым. Если параметр равен нулю, то обработчик будет вызван последним.

VectoredHandler

[in] Указатель на обработчик исключений. Для более подробной информации смотрите VectoredHandler.

Замечательно! То есть, возвращаясь к нашему коду, видим:

```
004010D0   PUSH Crackme_.00401000      ; ASCII "ïP2@"
004010D5   PUSH 0
004010D7   CALL NEAR EAX                ; ntdll.RtlAddVectoredExceptionHandler
```

Порядок, в котором будет вызываться обработчик нам до лампочки. Лишь бы вызывался. А вот вызываться он будет по адресу Crackme_.00401000, как видно из первого PUSH. Ставим бряк (F2) на адрес 00401000, отпускаем прогу по Shift+F9, и брякаемся на 00401000.

Трассируем код, и, судя по действиям, которые выполняются над введенными именем и серийником, мы попали в проверочную часть крякми!!!

Разбирать проверку дальше не входит в мои задачи, тем более, что это совсем просто, а материал, здесь представленный он немного о другом.

VII. Типа заключительные слова

Ну вот после перевода кучи информации по антиотладке, я разобрался в некоторых штуках, о существовании которых раньше даже не подозревал. Я надеюсь, что начинающие реверсеры, узнают что-то новое из этой писульки и чуть-чуть поднимут свой уровень. А опытные реверсеры, прочитав мой труд, отметят мои ошибки, укажут на неточности перевода, расскажут что-нибудь новое и чуть-чуть поднимут мой уровень.

Вот несколько ссылочек, которые полезно посетить:

http://www.cracklab.ru – без комментариев

http://www.securityfocus.com/infocus/1893 - Nicolas Falliere, Windows Anti-Debug Reference

http://www.openrce.org/articles/full_view/25 и http://www.openrce.org/articles/full_view/26 - подробное описание Debug Objects *by* Alex Ionescu

http://www.piotrbania.com/all/articles/antid.txt - Piotr Bania описывает применение функции OpenProcess применительно к антиотладке

http://piotrbania.com/all/articles/bypassing_the_breakpoints.txt

http://vx.eof-project.net/viewtopic.php?id=142 - Подробное описание методов обнаружения виртуальных машин.

http://pferrie.tripod.com/papers/attacks2.pdf

xii http://pferrie.tripod.com/papers/attacks2.pdf

http://www.symantec.com/enterprise/security_response/weblog/2007/02/x86_fetchdecode_anomalies.html - список недавно введенных команд процессора, которые не эмулируются в виртуальных машинах

http://www.wasm.ru/article.php?article=tls – описание TLS и примеры его использования

http://gl00my.chat.ru/nt/mem.txt - Описание менеджера памяти

Sturgeon, 07/2008