Статья "Максимальный" справочник по борьбе с отладкой под авторством Питера Ферри

Отладчик, пожалуй, наиболее часто используемый инструмент при реверс инжиниринге (следующий по распространенности инструмент после дизассемблера, такого как Interactive DisAssembler (IDA)). В результате, анти-отладочные трюки, вероятно, являются наиболее распространенной функцией кода, предназначенной для вмешательства в реверс-инжиниринг (а анти-дизассемблирующие конструкции являются следующими наиболее распространенными). Эти приемы могут просто обнаружить присутствие отладчика, отключить отладчик, вырваться из-под контроля отладчика или даже использовать уязвимость в отладчике. Присутствие отладчика может быть выведено косвенно, или может быть обнаружен определенный отладчик. Отключение или побег из-под контроля отладчика может быть достигнуто как общими, так и конкретными способами. Однако использование уязвимости достигается в отношении определенных отладчиков. Конечно, отладчик не обязательно должен присутствовать для того, чтобы попытаться использовать эксплоит.

Как правило, когда загружается отладчик, среда отладчика изменяется операционной системой, чтобы позволить отладчику взаимодействовать с программой (единственным исключением из этого является отладчик Obsidian). Некоторые из этих изменений более очевидны, чем другие, и по-разному влияют на работу отлаживаемого процесса. Среду также можно изменить по-разному, в зависимости от того, использовался ли отладчик для создания процесса или отладчик подключается к уже запущенному процессу.

Далее следует перечень известных методов, используемых для обнаружения присутствия отладчика, и в некоторых случаях защиты от них.

Примечание: Этот текст содержит несколько фрагментов кода в 32-битной и 64-битной версиях. Для простоты в 64-битных версиях предполагается, что все указатели стека и кучи и все дескрипторы соответствуют 32 битам. Они также полагаются на тот факт, что PEB всегда находится в нижней памяти.

1. NtGlobalFlag

Одно из самых простых изменений, которое вносит система, является также одним из наиболее неправильно понятых: поле NtGlobalFlag в Блоке Окружения Процесса. Поле NtGlobalFlag существует по смещению 0x68 в Блоке Окружения Процесса в 32-разрядных версиях Windows и по смещению 0xBC в 64-разрядных версиях Windows.

Значение в этом поле по умолчанию равно нулю. Значение не изменяется, когда отладчик подключается к процессу. Однако значение может быть изменено до некоторой степени находясь под контролем процесса. Есть также два ключа реестра, которые можно использовать для установки определенных значений. В их отсутствие процесс, созданный отладчиком, будет иметь фиксированное значение в этом поле по умолчанию, но это конкретное значение можно изменить с помощью определенной переменной среды. Поле состоит из набора флагов. Для процесса, созданного отладчиком, будут установлены следующие флаги:

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

Таким образом, способ обнаружения присутствия отладчика состоит в проверке комбинации этих флагов. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

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

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
push 60h
pop rsi
gs:lodsq ;Process Environment Block
mov al, [rsi*2+rax-14h] ;NtGlobalFlag
and al, 70h
cmp al, 70h
je being_debugged
```

Обратите внимание, что для 32-разрядного процесса в 64-разрядных версиях Windows существует отдельный Блок Окружения Процесса для 32-разрядной и 64-разрядной частей. Поля в 64-битной части затрагиваются так же, как и для 32-битной части.

Таким образом, существует еще одна проверка, которая использует этот 32-разрядный код для проверки 64-разрядной среды Windows:

```
mov eax, fs:[30h] ; Process Environment Block
;64-bit Process Environment Block
;follows 32-bit Process Environment Block
mov al, [eax+10bch] ;NtGlobalFlag
and al, 70h
cmp al, 70h
je being_debugged
```

Распространенная ошибка - использовать прямое сравнение, не маскируя сначала другие биты. В этом случае, если установлены другие биты, присутствие отладчика будет пропущено.

Способ победить эту технику для отладчика, заключается в том, чтобы изменить значение обратно на ноль, прежде чем возобновить процесс. Однако, как отмечено выше, начальное значение может быть изменено одним из четырех способов. Первый метод включает значение реестра, которое влияет на все процессы в системе. Это значение реестра является строковым значением "GlobalFlag" раздела реестра "HKLM\System\CurrentControlSet\Control\Session Manager". Здесь значение помещается в поле NtGlobalFlag, хотя оно может быть изменено позже в Windows. Изменение этого параметра реестра требует перезагрузки для вступления в силу. Это требование приводит к другому способу обнаружения присутствия отладчика, который также знает о значении реестра. Если отладчик копирует значение реестра в поле NtGlobalFlag, чтобы скрыть его присутствие, и если значение реестра изменяется, но система не перезагружается, тогда отладчик может быть обманут, используя это новое значение вместо истинного значения. Отладчик был бы обнаружен, если бы процесс знал, что истинное значение было чем-то отличным от того, что появляется в значении реестра. Один из способов определить истинное значение - запустить другой процесс, а затем запросить его значение NtGlobalFlag. Отладчик, который не знает о значении реестра, также раскрывается таким образом.

Второй метод также включает значение реестра, но оно влияет только на именованный процесс. Это значение реестра также является строковым значением "GlobalFlag", но в разделе реестра "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<имя файла>". "<Имя файла>" должно быть заменено именем исполняемого файла (не DLL), к которому будут применены флаги при выполнении файла. Как и выше, значение здесь помещается в поле NtGlobalFlag, хотя оно может быть изменено позже в Windows. Значение, установленное с помощью этого метода, объединяется со значением, которое применяется ко всем процессам, если таковые имеются.

Третий метод изменения значения основан на двух полях в Таблице Конфигурации Загрузки. В одном поле (GlobalFlagsClear) перечислены флаги для очистки, а в другом поле (GlobalFlagsSet) перечислены флаги, которые необходимо установить. Эти параметры применяются после того, как значения реестра GlobalFlag были применены, поэтому они могут переопределять значения, указанные в значениях реестра GlobalFlag. Однако они не могут переопределять значения, которые Windows устанавливает, когда определенные флаги остаются установленными (хотя они могут удалять флаги, которые устанавливаются, когда отладчик создает процесс). Например, установка флага FLG_USER_STACK_TRACE_DB (0x1000) заставляет Windows установить флаг FLG_HEAP_VALIDATE_PARAMETERS (0x40). Если флаг FLG_USER_STACK_TRACE_DB установлен в одном из значений реестра GlobalFlag, то даже если флаг FLG_HEAP_VALIDATE_PARAMETERS помечен для очистки в Таблице Конфигурации Загрузки, он все равно будет установлен Windows позже во время загрузки процесса.

Четвертый метод относится к изменениям, которые вносит Windows, когда отладчик создает процесс. При установке переменной среды "_NO_DEBUG_HEAP" три флага кучи не будут установлены в поле NtGlobalFlag из-за отладчика. Конечно, они все еще могут быть установлены значениями реестра GlobalFlag или полем GlobalFlagsSet в Таблице Конфигурации Загрузки.

2. Флаги кучи

Куча содержит два флага, которые инициализируются вместе с NtGlobalFlag. Значения в этих полях зависят от наличия отладчика, но также зависят от версии Windows. Расположение этих полей зависит от версии Windows. Два поля называются "Flags" и "ForceFlags". Поле "Flags" существует по смещению 0x0C в куче в 32-разрядных версиях Windows NT, Windows 2000 и Windows XP; и по смещению 0x40 в 32-разрядных версиях Windows Vista и более поздних. Поле "Flags" существует по смещению 0x14 в куче в 64-разрядных версиях Windows XP и по смещению 0x70 в куче в 64-разрядных версиях Windows Vista и более поздних версий. Поле ForceFlags существует по смещению 0x10 в куче в 32-разрядных версиях Windows NT, Windows 2000 и Windows XP; и по смещению 0x44 в 32-разрядных версиях Windows Vista и более поздних. Поле ForceFlags существует по смещению 0x18 в куче в 64-разрядных версиях Windows XP и по смещению 0x74 в куче в 64-разрядных версиях Windows Vista и более поздних версий.

Значение для поля Flags обычно устанавливается в HEAP_GROWABLE (2) во всех версиях Windows. Значение для поля ForceFlags обычно устанавливается равным нулю во всех версиях Windows. Однако оба эти значения зависят от версии подсистемы хост-процесса для 32-битного процесса (у 64-битного процесса такой зависимости нет). Значения полей соответствуют указанным, только если версия подсистемы равна 3.51

или выше. Если версия подсистемы 3.10-3.50, тогда флаг HEAP_CREATE_ALIGN_16 (0x10000) также будет установлен в обоих полях. Если версия подсистемы меньше 3.10, файл не будет работать вообще. Это особенно интересно, потому что общепринятым методом является размещение двух и нулевых значений в соответствующих полях, чтобы скрыть присутствие отладчика. Однако, если версия подсистемы не проверена, то это действие может выявить присутствие чего-либо, что пытается скрыть отладчик.

При наличии отладчика в поле Flags обычно устанавливается комбинация этих флагов в Windows NT, Windows 2000 и 32-разрядной Windows XP:

> HEAP_GROWABLE (2)
> HEAP_TAIL_CHECKING_ENABLED (0x20)
> HEAP_FREE_CHECKING_ENABLED (0x40)
> HEAP_SKIP_VALIDATION_CHECKS (0x10000000)
> HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)

В 64-разрядных версиях Windows XP и Windows Vista и более поздних версиях поле Flags обычно устанавливается в комбинацию следующих флагов:

> HEAP_GROWABLE (2)
> HEAP_TAIL_CHECKING_ENABLED (0x20)
> HEAP_FREE_CHECKING_ENABLED (0x40)
> HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)

Когда присутствует отладчик, поле ForceFlags обычно устанавливается в комбинацию этих флагов:

> HEAP_TAIL_CHECKING_ENABLED (0x20)
> HEAP_FREE_CHECKING_ENABLED (0x40)
> HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)

Флаг HEAP_TAIL_CHECKING_ENABLED устанавливается в полях кучи, если в поле NtGlobalFlag установлен флаг FLG_HEAP_ENABLE_TAIL_CHECK. Флаг HEAP_FREE_CHECKING_ENABLED устанавливается в полях кучи, если в поле NtGlobalFlag установлен флаг FLG_HEAP_ENABLE_FREE_CHECK. Флаг HEAP_VALIDATE_PARAMETERS_ENABLED (и флаг HEAP_CREATE_ALIGN_16 (0x10000) в Windows NT и Windows 2000) устанавливается в полях кучи, если в поле NtGlobalFlag установлен флаг FLG_HEAP_VALIDATE_PARAMETERS.

Это поведение можно предотвратить в Windows XP и более поздних версиях, в результате чего вместо этого будут использоваться значения по умолчанию, создав переменную среды "_NO_DEBUG_HEAP".

Флаги кучи также могут управляться для каждого отдельного процесса через строковое значение "PageHeapFlags" раздела реестра "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<filename>".

Расположение кучи можно получить несколькими способами. Одним из способов является использование функции Get32cessHeap() для kernel32. Это эквивалентно использованию этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
mov eax, fs:[30h] ;Process Environment Block
mov eax, [eax+18h] ;get process heap base
```

или используя этот 64-битный код для проверки 64-битной Среда Windows:

```
push 60h
pop rsi
gs:lodsq ;Process Environment Block
mov eax, [rax+30h] ;get process heap base
```

Как и в случае с Блоком Окружения, для 32-разрядного процесса в 64-разрядных версиях Windows существует отдельная куча для 32-разрядной и 64-разрядной частей. Поля в 64-битной части затрагиваются так же, как и для 32-битной части.

Таким образом, существует еще одна проверка, которая использует этот 32-разрядный код для проверки 64-разрядной среды Windows:

```
mov eax, fs:[30h] ;Process Environment Block
;64-bit Process Environment Block
;follows 32-bit Process Environment Block
mov eax, [eax+1030h] ;get process heap base
```

Другой способ заключается в использовании функции Get32cessHeaps() kernel32. Эта функция просто пересылается в функцию ntdll RtlGetProcessHeaps(). Функция возвращает массив куч процесса. Первая куча в списке такая же, как и та, которую

возвращает функция kernel32 GetProcessHeap(). Запрос также может быть выполнен с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push 30h
pop esi
fs:lodsd ;Process Environment Block
;get process heaps list base
mov esi, [esi+eax+5ch]
lodsd
```

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
push 60h
pop rsi
gs:lodsq ;Process Environment Block
;get process heaps list base
mov esi, [rsi*2+rax+20h]
lodsd
```

или используя этот 32-битный код для проверки 64-битной среды Windows:

```
mov eax, fs:[30h] ;Process Environment Block
;64-bit Process Environment Block
;follows 32-bit Process Environment Block
mov esi, [eax+10f0h] ;get process heaps list base
lodsd
```

Таким образом, способ обнаружения присутствия отладчика состоит в проверке специальной комбинации флагов в поле Flags. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows, если версия подсистемы находится (или может быть) в диапазоне 3.10-3.50:

```
call GetVersion
cmp al, 6
cmc
sbb ebx, ebx
and ebx, 34h
mov eax, fs:[30h] ;Process Environment Block
mov eax, [eax+18h] ;get process heap base
mov eax, [eax+ebx+0ch] ;Flags
;neither HEAP_CREATE_ALIGN_16
;nor HEAP_SKIP_VALIDATION_CHECKS
and eax, 0effeffffh
;HEAP_GROWABLE
;+ HEAP_TAIL_CHECKING_ENABLED
;+ HEAP_FREE_CHECKING_ENABLED
;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp eax, 40000062h
je being_debugged
```

Click to expand...

или этот 32-разрядный код для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows, если версия подсистемы составляет 3,51 или выше:

```
call GetVersion
cmp al, 6
cmc
sbb ebx, ebx
and ebx, 34h
mov eax, fs:[30h] ;Process Environment Block
mov eax, [eax+18h] ;get process heap base
mov eax, [eax+ebx+0ch] ;Flags
;not HEAP_SKIP_VALIDATION_CHECKS
bswap eax
and al, 0efh
;HEAP_GROWABLE
;+ HEAP_TAIL_CHECKING_ENABLED
;+ HEAP_FREE_CHECKING_ENABLED
;+ HEAP_VALIDATE_PARAMETERS_ENABLED
;reversed by bswap
cmp eax, 62000040h
je being_debugged
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
push 60h
pop rsi
gs:lodsq ;Process Environment Block
mov ebx, [rax+30h] ;get process heap base
call GetVersion
cmp al, 6
sbb rax, rax
and al, 0a4h
;HEAP_GROWABLE
;+ HEAP_TAIL_CHECKING_ENABLED
;+ HEAP_FREE_CHECKING_ENABLED
;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp d [rbx+rax+70h], 40000062h ;Flags
je being_debugged
```

Click to expand...

или используя этот 32-битный код для проверки 64-битной среды Windows:

```
push 30h
pop eax
mov ebx, fs:[eax] ;Process Environment Block
;64-bit Process Environment Block
;follows 32-bit Process Environment Block
mov ah, 10h
mov ebx, [ebx+eax] ;get process heap base
call GetVersion
cmp al, 6
sbb eax, eax
and al, 0a4h
;Flags
;HEAP_GROWABLE
;+ HEAP_TAIL_CHECKING_ENABLED
;+ HEAP_FREE_CHECKING_ENABLED
;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp [ebx+eax+70h], 40000062h
je being_debugged
```

Click to expand...

Вызов функции GetVersion() из kernel32 может быть еще более обфусцированн путем простого извлечения значения непосредственно из поля NtMajorVersion в структуре KUSER_SHARED_DATA со смещением 0x7ffe026c для конфигураций в 2 ГБ пространства пользователя. Это значение доступно во всех 32-разрядных и 64-разрядных версиях Windows.

Другой способ обнаружить присутствие отладчика - проверить наличие специальной комбинации флагов в поле ForceFlags. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows, если версия подсистемы находится (или может быть) в диапазоне 3.10-3.50:

```
call GetVersion
cmp al, 6
cmc
sbb ebx, ebx
and ebx, 34h
mov eax, fs:[30h] ;Process Environment Block
mov eax, [eax+18h] ;get process heap base
mov eax, [eax+ebx+10h] ;ForceFlags
;not HEAP_CREATE_ALIGN_16
btr eax, 10h
;HEAP_TAIL_CHECKING_ENABLED
;+ HEAP_FREE_CHECKING_ENABLED
;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp eax, 40000060h
je being_debugged
```

Click to expand...

или использовать этот 32-разрядный код для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows, если версия подсистемы составляет 3,51 или выше:

```
call GetVersion
cmp al, 6
cmc
sbb ebx, ebx
and ebx, 34h
mov eax, fs:[30h] ;Process Environment Block
mov eax, [eax+18h] ;get process heap base
;ForceFlags
;HEAP_TAIL_CHECKING_ENABLED
;+ HEAP_FREE_CHECKING_ENABLED
;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp [eax+ebx+10h], 40000060h
je being_debugged
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
push 60h
pop rsi
gs:lodsq ;Process Environment Block
mov ebx, [rax+30h] ;get process heap base
call GetVersion
cmp al, 6
sbb rax, rax
and al, 0a4h
;ForceFlags
;HEAP_TAIL_CHECKING_ENABLED
;+ HEAP_FREE_CHECKING_ENABLED
;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp d [rbx+rax+74h], 40000060h
je being_debugged
```

Click to expand...

или используя этот 32-битный код для проверки 64-битной среды Windows:

```
call GetVersion
cmp al, 6
push 30h
pop eax
mov ebx, fs:[eax] ;Process Environment Block
;64-bit Process Environment Block
;follows 32-bit Process Environment Block
mov ah, 10h
mov ebx, [ebx+eax] ;get process heap base
sbb eax, eax
and al, 0a4h
;ForceFlags
;HEAP_TAIL_CHECKING_ENABLED
;+ HEAP_FREE_CHECKING_ENABLED
;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp [ebx+eax+74h], 40000060h
je being_debugged
```

Click to expand...

3. Куча

Когда куча инициализируется, проверяются флаги кучи, и в зависимости от того, какие флаги установлены, в среду могут вноситься дополнительные изменения. Если установлен флаг EAP_TAIL_CHECKING_ENABLED, то последовательность 0xABABABAB будет добавлена дважды в 32-разрядной среде Windows (четыре раза в 64-разрядной среде Windows) в конце выделенного блока. Если установлен флаг HEAP_FREE_CHECKING_ENABLED, то последовательность 0xFEEEFEEE (или ее часть) будет добавлена, если для заполнения свободного пространства до следующего блока требуются дополнительные байты. Таким образом, способ обнаружить присутствие отладчика - проверить эти значения. Если указатель кучи известен, то проверку можно выполнить путем непосредственного изучения данных кучи. Однако Windows Vista и более поздние версии используют защиту кучи как на 32-разрядной, так и на 64-разрядной платформах, с введением ключа XOR для кодирования размера блока. Использование этого ключа необязательно, но по умолчанию оно используется. Расположение служебного поля также отличается в Windows NT / 2000/XP и Windows Vista и более поздних версиях. Следовательно, версия Windows должна быть принята во внимание. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor ebx, ebx
call GetVersion
cmp al, 6
sbb ebp, ebp
jb l1
;Process Environment Block
mov eax, fs:[ebx+30h]
mov eax, [eax+18h] ;get process heap base
mov ecx, [eax+24h] ;check for protected heap
jecxz l1
mov ecx, [ecx]
test [eax+4ch], ecx
cmovne ebx, [eax+50h] ;conditionally get heap key
l1: mov eax, <heap ptr>
movzx edx, w [eax-8] ;size
xor dx, bx
movzx ecx, b [eax+ebp-1] ;overhead
sub eax, ecx
lea edi, [edx*8+eax]
mov al, 0abh
mov cl, 8
repe scasb
je being_debugged
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor ebx, ebx
call GetVersion
cmp al, 6
sbb rbp, rbp
jb l1
;Process Environment Block
mov rax, gs:[rbx+60h]
mov eax, [rax+30h] ;get process heap base
mov ecx, [rax+40h] ;check for protected heap
jrcxz l1
mov ecx, [rcx+8]
test [rax+7ch], ecx
cmovne ebx, [rax+88h] ;conditionally get heap key
l1: mov eax, <heap ptr>
movzx edx, w [rax-8] ;size
xor dx, bx
add edx, edx
movzx ecx, b [rax+rbp-1] ;overhead
sub eax, ecx
lea edi, [rdx*8+rax]
mov al, 0abh
mov cl, 10h
repe scasb
je being_debugged
```

Click to expand...

Не существует эквивалента для 32-разрядного кода для проверки 64-разрядной среды Windows, поскольку 64-разрядная куча не может быть проанализирована 32-разрядной функцией кучи.

Если указатель неизвестен, его можно получить с помощью функции kernel32 HeapWalk() или функции ntdll RtlWalkHeap() (или даже функции kernel32 GetCommandLine()). Возвращенное значение размера блока декодируется автоматически, поэтому версия Windows больше не имеет значения в этом случае. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
mov ebx, offset l2
;get a pointer to a heap block
l1: push ebx
mov eax, fs:[30h] ;Process Environment Block
push d [eax+18h] ;save process heap base
call HeapWalk
cmp w [ebx+0ah], 4 ;find allocated block
jne l1
mov edi, [ebx] ;data pointer
add edi, [ebx+4] ;data size
mov al, 0abh
push 8
pop ecx
repe scasb
je being_debugged
...
l2: db 1ch dup (0) ;sizeof(PROCESS_HEAP_ENTRY)
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
mov rbx, offset l2
;get a pointer to a heap block
l1: push rbx
pop rdx
push 60h
pop rsi
gs:lodsq ;Process Environment Block
;get a pointer to process heap base
mov ecx, [rax+30h]
call HeapWalk
cmp w [rbx+0eh], 4 ;find allocated block
jne l1
mov edi, [rbx] ;data pointer
add edi, [rbx+8] ;data size
mov al, 0abh
push 10h
pop rcx
repe scasb
je being_debugged
...
l2: db 28h dup (0) ;sizeof(PROCESS_HEAP_ENTRY)
```

Click to expand...

Не существует эквивалента для 32-разрядного кода для проверки 64-разрядной среды Windows, поскольку 64-разрядная куча не может быть проанализирована 32-разрядной функцией кучи.

4. TLS

Локальное хранилище потоков является одним из самых интересных методов борьбы с отладкой, которые существуют, потому что, несмотря на то, что они известны уже более десяти лет, все еще открываются новые способы их использования (и злоупотребления). Локальное хранилище потока существует для инициализации данных, специфичных для потока, до запуска этого потока. Поскольку каждый процесс содержит хотя бы один поток, это поведение включает в себя возможность инициализировать данные до запуска основного потока. Инициализация может быть выполнена путем указания статического буфера, который копируется в динамически распределенную память, и/или посредством выполнения кода в массиве обратных вызовов для динамической инициализации содержимого памяти. Чаще всего злоупотребляют массивом обратного вызова.

Массив обратных вызовов TLS может быть изменен (более поздние записи могут быть изменены) и/или расширен (новые записи могут быть добавлены) во время выполнения. Вновь добавленные или измененные обратные вызовы будут вызываться с использованием новых адресов. Нет ограничений на количество обратных вызовов, которые могут быть размещены. Расширение может быть сделано с помощью этого кода (идентичного для 32-битной и 64-битной) в 32-битной или 64-битной версии Windows:

```
l1: mov d [offset cbEnd], offset l2
ret
l2: ...
```

Обратный вызов в l2 будет вызван, когда обратный вызов вернется в l1.

Адреса обратного вызова локального хранилища потоков могут указывать за пределы образа, например, на вновь загруженные библиотеки DLL. Это можно сделать косвенным путем, загрузив DLL и поместив возвращенный адрес в массив обратных вызовов TLS. Это также можно сделать напрямую, если известен адрес загрузки DLL. Значение imagebase может использоваться в качестве адреса обратного вызова, если DLL структурирована таким образом, чтобы победить DEP, если оно включено, или может быть получен и использован действительный адрес экспорта. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
l1: push offset l2
call LoadLibraryA
mov [offset cbEnd], eax
ret
l2: db "myfile", 0
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
mov rcx, offset l2
call LoadLibraryA
mov [offset cbEnd], rax
ret
l2: db "myfile", 0
```

В этом случае заголовок "MZ" файла с именем "tls2.dll" будет выполнен, когда обратный вызов вернется на l1. Кроме того, файл может ссылаться на себя, используя

этот 32-разрядный код в 32-разрядной или 64-разрядной версии Windows:

```
l1: push 0
call GetModuleHandleA
mov [offset cbEnd], eax
ret
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
l1: xor ecx, ecx
call GetModuleHandleA
mov [offset cbEnd], rax
ret
```

В этом случае заголовок "MZ" текущего процесса будет выполнен, когда обратный вызов вернется в l1 . Адреса обратного вызова локального хранилища потока могут содержать RVA импортированных адресов из других библиотек DLL, если таблица адресов импорта изменяется, чтобы указывать на массив обратного вызова. Импорт разрешается до вызова обратных вызовов, поэтому импортированные функции будут вызываться нормально при достижении записи массива обратных вызовов.

Обратные вызовы TLS получают три параметра стека, которые могут быть переданы непосредственно в функции. Первый параметр - это ImageBase хост-процесса. Он может быть использован функцией kernel32 LoadLibrary() или функцией kernel32 WinExec(), например. Параметр ImageBase будет интерпретироваться функциями ядра LoadLibrary() или ядра WinExec() как указатель на имя файла для загрузки или выполнения. Создав файл с именем "MZ [некоторая строка]", где "[некоторая строка]" соответствует содержимому заголовка файла хоста, обратный вызов TLS получит доступ к файлу без какой-либо явной ссылки. Конечно, часть строки "MZ" также может быть заменена вручную во время выполнения, но многие функции полагаются непредсказуемыми.

Обратные вызовы локального хранилища потоков вызываются всякий раз, когда поток создается или уничтожается (если только процесс не вызывает kernel32 DisableThreadLibraryCalls() или функции ntdll LdrDisableThreadCalloutsForDll()). Это включает в себя поток, который создается Windows, когда отладчик подключается к процессу. Поток отладчика является особенным в том смысле, что его точка входа не указывает внутри образа. Вместо этого он указывает внутри kernel32.dll. Таким образом, простой метод обнаружения отладчика заключается в использовании обратного вызова TLS для запроса начального адреса каждого потока, который

создается. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push eax
mov eax, esp
push 0
push 4
push eax
push eax
mov eax, esp
push 0
push 4
push eax
```

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor ebp, ebp
enter 20h, 0
push 4
pop r9
push rbp
pop r8
;ThreadQuerySetWin32StartAddress
push 9
pop rdx
push -2 ;GetCurrentThread()
pop rcx
call NtQueryInformationThread
leave
cmp rbp, offset l1
jnb being_debugged
...
l1: <code end>

Click to expand...
```

Поскольку обратные вызовы локального хранилища выполняются до того, как отладчик может получить контроль, обратный вызов может вносить другие

изменения, такие как удаление точки останова, которая обычно размещается в точке входа . Патч можно сделать с помощью этого кода (идентичного для 32-битной и 64-битной) в 32-битной или 64-битной версии Windows:

```
;<val> is byte at l1
mov b [offset l1], <val>
ret
l1: <host entrypoint>
```

Защита от этой техники очень проста и все более необходима. Это вопрос вставки точки останова в первый байт первого обратного вызова локального хранилища потока, а не в точку входа хоста. Это позволит отладчику получить контроль перед тем, как в процессе может выполняться любой код (конечно, за исключением загруженных DLL). Однако необходимо соблюдать осторожность в отношении адреса обратного вызова, поскольку, как отмечено выше, исходное значение по этому адресу может быть RVA импортируемой функции. Таким образом, адрес не может быть прочитан из файла. Это должно быть прочитано из памяти изображения.

Выполнение обратных вызовов TLS также зависит от платформы. Если исполняемый файл импортируется только из ntdll.dll или kernel32.dll, то обратные вызовы не будут вызываться во время события "on attach" при запуске в Windows XP и более поздних версиях. Когда процесс запускается, функция ntdll LdrInitializeThunk() обрабатывает список InLoadOrderModuleList. Список nLoadOrderModuleList содержит список библиотек DLL для обработки. Значение Flags в ссылочной структуре должно иметь бит LDRP_ENTRY_PROCESSED, очищенный хотя бы в одной DLL, для обратных вызовов TLS, вызываемых при присоединении.

Этот бит всегда устанавливается для ntdll.dll, поэтому при импорте файла из только ntdll.dll не будет выполняться обратный вызов локального хранилища потока при присоединении. В Windows 2000 и более ранних версиях была ошибка сбоя, если файл не импортировался из kernel32.dll, явно (то есть импортируется из kernel32.dll напрямую) или неявно (то есть импортировался из DLL, которая импортируется из kernel32.dll; или DLL, которая импортирует из DLL, которая импортирует из kernel32.dll, независимо от длины цепочки).

Эта ошибка была исправлена в Windows XP, заставляя ntdll.dll явно загружать kernel32.dll перед обработкой таблицы импорта хоста. Когда kernel32.dll загружен, он добавляется в InLoadOrderModuleList. Проблема в том, что это исправление внесло побочный эффект.

Побочный эффект возникает, когда ntdll.dll извлекает адрес экспортированной функции из kernel32.dll через функцию ntdll LdrGetProcedureAddressEx(). Побочный эффект может быть вызван в результате получения любой экспортируемой функции, но он запускается в данном конкретном случае, когда ntdll извлекает адрес одной из следующих функций: BaseProcessInitPostImport() (только для Windows XP и Windows Server 2003), BaseQueryModuleData () (Только для Windows XP и Windows Server 2003, если функция BaseProcessInitPostImport() не существует), BaseThreadInitThunk() (Windows Vista и более поздние версии) или BaseQueryModuleData() (Windows Vista и более поздние версии, если BaseThreadInitThunk() не существует).

Побочным эффектом является то, что функция ntdll LdrGetProcedureAddressEx() устанавливает флаг LDRP_ENTRY_PROCESSED для записи kernel32.dll в списке InLoadOrderModuleList. В результате при импорте файла, импортируемого только из kernel32.dll, обратные вызовы локального хранилища потоков не выполняются. Это можно считать ошибкой в Windows.

Существует простой обходной путь для этой проблемы: импортировать что-либо из другой библиотеки DLL и при условии, что у библиотеки DLL есть ненулевая точка входа. Затем обратные вызовы TLS будут выполняться при присоединении. Обходной путь работает, потому что значение поля Flags будет иметь бит LDRP_ENTRY_PROCESSED, очищенный для этой DLL.

В Windows Vista и более поздних версиях динамически загружаемые библиотеки DLL также поддерживают локальное хранилище потоков. Оно находится в прямом противоречии с существующей документацией формата Portable Executable, в которой говорится, что "Статически объявленные объекты данных TLS", то есть обратные вызовы TLS, могут использоваться только в статически загружаемых файлах образа. Этот факт делает ненадежным использование статических данных локального хранилища потоков в DLL, если вы не знаете, что DLL или что-либо статически связанное с ней никогда не будет загружаться динамически с помощью функции API LoadLibrary". Кроме того, будут вызываться обратные вызовы TLS независимо от того, что присутствует в таблице импорта. Таким образом, DLL может импортировать из ntdll.dll или kernel32.dll или даже вообще без DLL (в отличие от случая .exe, описанного выше), и обратные вызовы будут вызваны!

5.Anti-Step-Over

Большинство отладчиков поддерживают пошаговое выполнение определенных инструкций, таких как последовательности "call" и "rep". В таких случаях программная точка останова часто помещается в поток команд, и затем процессу разрешается возобновить выполнение. Обычно отладчик снова получает управление, когда достигается программная точка останова. Однако в случае последовательности "rep"

отладчик должен проверить, что инструкция, следующая за префиксом rep, действительно является инструкцией, к которой rep применяется по закону. Некоторые отладчики предполагают, что любой префикс rep предшествует строковой инструкции. Это создает уязвимость, когда инструкция, следующая за префиксом rep, является полностью другой инструкцией. В частности, проблема возникает, если эта инструкция удаляет программную точку останова, которая была бы помещена в поток, если команда перешагнула. В этом случае, когда инструкция перешагивает, а программная точка останова удаляется командой, выполнение возобновляется под полным контролем процесса и никогда не возвращается к отладчику. Пример кода выглядит так:

```
rep
l1: mov b [offset l1], 90h
l2: nop
```

Если попытка перешагнуть на l1, то выполнение будет возобновлено свободно с l2.

Более общий метод использует строковые инструкции для удаления точки останова. Патч можно сделать с помощью этого 32-битного кода в 32-битной или 64-битной версии Windows:

```
mov al, 90h
xor ecx, ecx
inc ecx
mov edi, offset l1
rep stosb
l1: nop
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
mov al, 90h
xor ecx, ecx
inc ecx
mov rdi, offset l1
rep stosb
l1: nop
```

Существуют варианты этой техники, например, использование "rep movs" вместо "rep stos". Флаг направления может использоваться для изменения направления записи в

память, так что перезапись может быть пропущена. Патч можно сделать с помощью этого 32-битного кода в 32-битной или 64-битной версии Windows:

```
mov al, 90h
push 2
pop ecx
mov edi, offset l1
std
rep stosb
nop
l1: nop
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
mov al, 90h
push 2
pop rcx
mov rdi, offset l1
std
rep stosb
nop
l1: nop
```

Решением этой проблемы является использование аппаратных точек останова во время перехода по строковым инструкциям. Это особенно важно, если учесть, что отладчик не может знать, является ли установленная им точка останова выполненной. Если процесс удаляет точку останова, он может впоследствии восстановить точку останова и затем выполнить точку останова, как обычно. Отладчик увидит ожидаемое исключение точки останова и будет вести себя как обычно. Это может быть выполнено с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
mov al, 90h
l1: xor ecx, ecx
inc ecx
mov edi, offset l3
l2: rep stosb
l3: nop
cmp al, 0cch
l4: mov al, 0cch
jne l1
l5: ...
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
mov al, 90h
l1: xor ecx, ecx
inc ecx
mov rdi, offset l3
l2: rep stosb
l3: nop
cmp al, 0cch
l4: mov al, 0cch
jne l1
l5: ...
```

В этом примере, перешагнув инструкцию на l2, код достигнет l4, а затем вернется к l1. Это приведет к замене точки останова на l2 на втором проходе и выполнению на l3. Затем отладчик восстановит управление. В то время единственное очевидное отличие будет состоять в том, что регистр AL будет содержать значение 0xCC вместо ожидаемого 0x90. Это позволит достичь уровня l5 за один проход вместо двух. Конечно, возможны гораздо более тонкие варианты, включая выполнение совершенно разных путей кода.

Разновидность техники может использоваться для простого обнаружения присутствия отладчика. Проверка может быть выполнена с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
xor ecx, ecx
inc ecx
mov esi, offset l1
lea edi, [esi + 1]
rep movsb
l1: mov al, 90h
l2: cmp al, 0cch
je being_debugged
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
xor ecx, ecx
inc ecx
mov rsi, offset l1
lea rdi, [esi + 1]
rep movsb
l1: mov al, 90h
l2: cmp al, 0cch
je being_debugged
```

Этот код обнаружит точку останова, которая находится на l1. Он работает путем копирования значения в l1 через 90h в l1 + 1. Затем значение сравнивается на l2.

6.Аппаратура

A. Аппаратные точки останова

Когда возникает исключение, Windows создает контекстную структуру для передачи обработчику исключения. Структура будет содержать значения общих регистров, селекторов, управляющих регистров и регистров отладки. Если отладчик присутствует и передает исключение отладчику с использованием аппаратных точек останова, то регистры отладки будут содержать значения, которые показывают наличие отладчика. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor eax, eax
push offset l1
push d fs:[eax]
mov fs:[eax], esp
int 3 ;force an exception to occur
...
l1: ;execution resumes here when exception occurs
mov eax, [esp+0ch] ;get ContextRecord
mov ecx, [eax+4] ;Dr0
or ecx, [eax+8] ;Dr1
or ecx, [eax+0ch] ;Dr2
or ecx, [eax+10h] ;Dr3
jne being_debugged
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
mov rdx, offset l1
xor ecx, ecx
inc ecx
call AddVectoredExceptionHandler
int 3 ;force an exception to occur
...
l1: ;execution resumes here when exception occurs
mov rax, [rcx+8] ;get ContextRecord
mov rcx, [rax+48h] ;Dr0
or rcx, [rax+50h] ;Dr1
or rcx, [rax+58h] ;Dr2
or rcx, [rax+60h] ;Dr3
jne being_debugged
```

Click to expand...

Значения для регистров отладки также могут быть изменены до возобновления выполнения в 32-разрядных версиях Windows, что может привести к неконтролируемому выполнению, если программная точка останова не установлена в соответствующем месте.

B.Подсчет инструкций

Подсчет команд можно выполнить, зарегистрировав обработчик исключений, а затем установив аппаратные точки останова на определенных адресах. При нажатии на соответствующий адрес будет сгенерировано исключение EXCEPTION_SINGLE_STEP (0x80000004). Это исключение будет передано в обработчик исключений. Обработчик исключений может выбрать настройку указателя инструкции для указания новой инструкции, опционально установить дополнительные аппаратные точки останова на определенных адресах и затем возобновить выполнение. Для установки точек останова необходим доступ к контекстной структуре. Копию контекстной структуры можно получить, вызвав функцию kernel32 GetThreadContext(), которая позволяет при необходимости установить начальные значения для аппаратных точек останова. Впоследствии, когда возникает исключение, обработчик исключений автоматически получает копию структуры контекста. Отладчик будет мешать пошаговому выполнению, что приведет к другому количеству команд по сравнению с отсутствием отладчика. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor eax, eax
push offset l5
push d fs:[eax]
mov fs:[eax], esp
int 3 ;force exception to occur
l1: nop
l2: nop
l3: nop
l4: nop
cmp al, 4
jne being_debugged
...
l5: push edi
mov eax, [esp+8] ;ExceptionRecord
mov edi, [esp+10h] ;ContextRecord
push 55h ;local-enable DR0, DR1, DR2, DR3
pop ecx
inc d [ecx*2+edi+0eh] ;Eip
mov eax, [eax] ;ExceptionCode
sub eax, 80000003h ;EXCEPTION_BREAKPOINT
jne l6
mov eax, offset l1
scasd
stosd ;Dr0
inc eax ;l2
stosd ;Dr1
inc eax ;l2
stosd ;Dr2
inc eax ;l4
stosd ;Dr3
;local-enable breakpoints
;for compatibility with old CPUs
mov ch, 1
xchg ecx, eax
scasd
stosd ;Dr7
xor eax, eax
pop edi
ret
l6: dec eax ;EXCEPTION_SINGLE_STEP
jne being_debugged
```

> inc b [ecx*2+edi+6] ;Eax
> pop edi
> ret
>
> Click to expand...

Поскольку в этом методе используется структурированный обработчик исключений, его нельзя использовать в 64-разрядных версиях Windows. Код можно легко переписать, чтобы вместо него использовать Vectored Exception Handler. Это требует создания потока и изменения его контекста, потому что регистры отладки не могут быть назначены внутри обработчика исключений с векторным управлением в 64-разрядных версиях Windows. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версии Windows XP или более поздней версии:

```
xor ebx, ebx
push eax
push esp
push 4 ;CREATE_SUSPENDED
push ebx
push offset l1
push ebx
push ebx
call CreateThread
mov esi, offset l7
push esi
push eax
xchg ebp, eax
call GetThreadContext
mov eax, offset l2
lea edi, [esi+4]
stosd ;Dr0
inc eax
stosd ;Dr1
inc eax
stosd ;Dr2
inc eax
stosd ;Dr3
scasd
push 55h ;local-enable DR0, DR1, DR2, DR3
pop eax
stosd ;Dr7
push esi
push ebp
call SetThreadContext
push offset l6
push 1
call AddVectoredExceptionHandler
push ebp
call ResumeThread
jmp $
l1: xor eax, eax
l2: nop
l3: nop
l4: nop
l5: nop
```

```
cmp al, 4
jne being_debugged
...
l6: mov eax, [esp+4]
mov ecx, [eax] ;ExceptionRecord
;ExceptionCode
cmp [ecx], 80000004h ;EXCEPTION_SINGLE_STEP
jne being_debugged
mov eax, [eax+4] ;ContextRecord
cdq
mov dh, 1
inc b [eax+edx-50h] ;Eax
inc d [eax+edx-48h] ;Eip
or eax, -1 ;EXCEPTION_CONTINUE_EXECUTION
ret
l7: dd 10002h ;CONTEXT_i486+CONTEXT_INTEGER
db 0b0h dup (?)
```

Click to expand...

или этот 64-битный код для проверки 64-битной среды Windows:

```
push rax
push rsp
push 4 ;CREATE_SUSPENDED
sub esp, 20h
xor r9d, r9d
mov r8, offset l1
xor edx, edx
xor ecx, ecx
call CreateThread
mov ebp, eax
mov rsi, offset l7-30h
push rsi
pop rdx
xchg ecx, eax
call GetThreadContext
mov rax, offset l2
lea rdi, [rsi+48h]
stosq ;Dr0
inc rax
stosq ;Dr1
inc rax
stosq ;Dr2
inc rax
stosq ;Dr3
scasq
push 55h ;local-enable DR0, DR1, DR2, DR3
pop rax
stosd ;Dr7
push rsi
pop rdx
mov ecx, ebp
call SetThreadContext
mov rdx, offset l6
xor ecx, ecx
inc ecx
call AddVectoredExceptionHandler
mov ecx, ebp
call ResumeThread
jmp $
l1: xor eax, eax
l2: nop
```

```
l3: nop
l4: nop
l5: nop
cmp al, 4
jne being_debugged
...
l6: mov rax, [rcx] ;ExceptionRecord
;ExceptionCode
cmp d [rax], 80000004h ;EXCEPTION_SINGLE_STEP
jne being_debugged
mov rax, [rcx+8] ;ContextRecord
inc b [rax+78h] ;Eax
inc q [rax+0f8h] ;Eip
or eax, -1 ;EXCEPTION_CONTINUE_EXECUTION
ret
l7: dd 10002h ;CONTEXT_i486+CONTEXT_INTEGER
db 0c4h dup (?)
```

Click to expand...

C.Прерывание #3

Всякий раз, когда возникает исключение программного прерывания, адрес исключения и значение регистра EIP будут указывать на инструкцию после той, которая вызвала исключение. Исключение точки останова рассматривается как особый случай. Когда возникает исключение EXCEPTION_BREAKPOINT (0x80000003), Windows предполагает, что оно было вызвано однобайтовым кодом операции "CC" (инструкция "INT 3"). Windows уменьшает адрес исключения, указывая на предполагаемый код операции «CC», а затем передает исключение обработчику исключений. На значение регистра EIP это не влияет. Таким образом, если используется код операции "CD 03" (длинная инструкция "INT 03"), адрес исключения будет указывать на "03", когда обработчик исключения получает управление.

D. Прерывание 0x2d

Прерывание 0x2D является частным случаем. Когда оно выполняется, Windows использует текущее значение регистра EIP в качестве адреса исключения, а затем оно увеличивается на единицу значения регистра EIP. Однако Windows также проверяет значение в регистре EAX, чтобы определить, как настроить адрес исключения. Если регистр EAX имеет значение 1, 3 или 4 во всех версиях Windows или значение 5 в Windows Vista и более поздних версиях, то Windows увеличивает его на один адрес исключения. Наконец, он выдает исключение EXCEPTION_BREAKPOINT

(0x80000003), если присутствует отладчик. Поведение прерывания 0x2D может вызвать проблемы для отладчиков. Проблема заключается в том, что некоторые отладчики могут использовать значение регистра EIP в качестве адреса для возобновления, в то время как другие отладчики могут использовать адрес исключения в качестве адреса для возобновления. Это может привести к пропуску однобайтовой инструкции или выполнению совершенно другой инструкции, поскольку первый байт отсутствует. Эти поведения могут использоваться, чтобы вывести присутствие отладчика. Проверка может быть выполнена с использованием этого кода (идентичного для 32-битной и 64-битной), чтобы проверить 32-битную или 64-битную среду Windows:

```
xor eax, eax ;set Z flag
int 2dh
inc eax ;debugger might skip
je being_debugged
```

E. Прерывание 0x41

Прерывание 0x41 может отображать другое поведение, если присутствует отладчик режима ядра или нет. Дескриптор прерывания 0x41 обычно имеет нулевой DPL, что означает, что прерывание не может быть успешно выполнено из кольца зазиты 3. Попытка выполнить это прерывание напрямую приведет к тому, что процессор выдает общую ошибку защиты (прерывание 0x0D), что в итоге приводит к исключению EXCEPTION_ACCESS_VIOLATION (0xC0000005). Однако некоторые отладчики перехватывают прерывание 0x41 и устанавливают его DPL равным 3, чтобы прерывание можно было успешно вызывать из пользовательского режима. Этот факт можно использовать для определения наличия отладчика режима ядра. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor eax, eax
push offset l1
push d fs:[eax]
mov fs:[eax], esp
mov al, 4fh
int 41h
jmp being_debugged
l1: ;execution resumes here if no debugger present
...
```

или использовать этот 64-битный код для проверки 64-битной среды Windows (хотя вряд ли он будет поддерживаться 64-битным отладчиком):

```
mov rdx, offset l1
xor ecx, ecx
inc ecx
call AddVectoredExceptionHandler
push 4fh
pop rax
int 41h
jmp being_debugged
l1: ;execution resumes here if no debugger present
```

F.MOV SS

Существует простой трюк для определения одноступенчатых операций, который работал с самых ранних процессоров Intel. Он использовался довольно часто во времена DOS, но он все еще работает во всех версиях Windows. Трюк основан на том факте, что определенные инструкции приводят к отключению всех прерываний при выполнении следующей инструкции. В частности, загрузка регистра SS очищает прерывания, чтобы позволить следующей инструкции загрузить регистр ESP без риска повреждения стека. Однако не требуется, чтобы следующая инструкция загружала что-либо в регистр ESP. Любая инструкция может следовать за загрузкой регистра SS. Если для пошагового выполнения кода используется отладчик, то в образе EFLAGS будет установлен флаг T. Обычно это не видно, поскольку флаг T будет очищен в образе EFLAGS после доставки каждого события отладчика. Однако, если флаги сохраняются в стеке до доставки события отладчика, тогда флаг T станет видимым. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push ss
pop ss
pushfd
test b [esp+1], 1
jne being_debugged
```

Интересная ситуация возникает в VirtualPC при работе с Windows 2000, которая заключается в том, что инструкция CPUID ведет себя таким же образом. Неизвестно,

почему это происходит.

Пример 64-битного кода отсутствует, поскольку в этой среде не поддерживается селектор SS.

7.API

Отладчик может быть обнаружен, отключен (в результате чего он теряет контроль над отладчиком), используя стандартные функции операционной системы.

A. Функции кучи

> BasepFreeActivationContextActivationBlock
> BasepFreeAppCompatData
> ConvertFiberToThread
> DeleteFiber
> FindVolumeClose
> FindVolumeMountPointClose
> HeapFree
> SortCloseHandle

Единственное, что объединяет все эти функции, это то, что они вызывают функцию ntdll RtlFreeHeap().

The kernel32

Функции BasepFreeActivationContextActivationBlock() и SortCloseHandle() kernel32 существуют только в Windows 7 и более поздних версиях. Функция kernel32 BasepFreeAppCompatData() существует только в Windows Vista и более поздних версиях. Ядро FindVolumeMountPointClose() вызывает функцию ntdll RtlFreeHeap() только как особый случай (в частности, когда параметр hFindVolumeMountPoint является допустимым указателем на дескриптор, который может быть успешно закрыт)

Однако дело в том, что функция ntdll RtlFreeHeap() содержит функцию, которая предназначена для использования вместе с отладчиком, - вызов функции ntdll DbgPrint(). Проблема заключается в том, что способ реализации функции ntdll DbgPrint() позволяет приложению обнаруживать присутствие отладчика при вызове функции.

Когда вызывается функция dtgPrint() ntdll, она вызывает исключение DBG_PRINTEXCEPTION_C (0x40010006), но исключение обрабатывается особым образом, поэтому зарегистрированный структурированный обработчик исключений

не увидит его. Причина в том, что Windows внутренне регистрирует свой собственный обработчик структурированных исключений, который использует исключение, если отладчик этого не делает. Однако в Windows XP и более поздних версиях любой зарегистрированный векторный обработчик исключений будет работать до структурированного обработчика исключений, который регистрирует Windows. Это может считаться ошибкой в Windows. Присутствие отладчика, который использует исключение, теперь может быть выведено из отсутствия исключения. Кроме того, другой обработчик доставляется в обработчик исключений если отладчик присутствует, но не использует исключение, или если отладчик вообще отсутствует. Если отладчик присутствует, но не использует исключение, Windows доставит исключение DBG_PRINTEXCEPTION_C (0x40010006). Если отладчик отсутствует, Windows доставит исключение EXCEPTION_ACCESS_VIOLATION (0xC0000005). Присутствие отладчика теперь может быть определено либо отсутствием исключения, либо значением исключения.

Существует еще один случай, который применяется, среди прочего, к функциям кучи и ресурсов, когда функции могут быть принудительно вызваны прерыванием отладки. Общим для них является проверка флага BeingDebugged в Блоке Окружения Процесса. Присутствие отладчика может быть подделано, чтобы вызвать исключение прерывания 3, и исключение должно быть видимым для отладчика. Таким образом, если исключение отсутствует (потому что отладчик использовал его), то присутствие отладчика раскрывается. Проверка может быть выполнена с использованием этого 32-разрядного кода, чтобы проверить 32-разрядную среду Windows в 32-разрядной версии для 64-разрядных версий Windows:

```
xor eax, eax
push offset l1
push d fs:[eax]
mov fs:[eax], esp
;Process Environment Block
mov eax, fs:[eax+30h]
inc b [eax+2] ;set BeingDebugged
push offset l2
call HeapDestroy
jmp being_debugged
l1: ;execution resumes here due to exception
...
l2: db 0ch dup (0)
dd 40000000h ;HEAP_VALIDATE_PARAMETERS_ENABLED
db 30h dup (0)
dd 40000000h ;HEAP_VALIDATE_PARAMETERS_ENABLED
db 24h dup (0)
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
mov rdx, offset l1
xor ecx, ecx
inc ecx
call AddVectoredExceptionHandler
push 60h
pop rsi
gs:lodsq ;Process Environment Block
inc b [rax+2] ;set BeingDebugged
mov rcx, offset l2
call HeapDestroy
jmp being_debugged
l1: ;execution resumes here due to exception
...
l2: db 14h dup (0)
dd 40000000h ;HEAP_VALIDATE_PARAMETERS_ENABLED
db 58h dup (0)
dd 40000000h ;HEAP_VALIDATE_PARAMETERS_ENABLED
db 30h dup (0)
```

Click to expand...

Значение флага появляется дважды в каждом случае, потому что оно размещается в обоих возможных местах для поля флага, в зависимости от версии Windows. Это исключает необходимость проверки версии.

Обратите внимание, что в Windows Vista и более поздних версиях поведение изменилось незначительно. Ранее прерывание отладки было вызвано вызовом функции ntdll DbgBreakPoint(). Теперь это вызвано инструкцией прерывания 3, которая сохраняется непосредственно в потоке кода. Результат в обоих случаях одинаков.

Обнаружение также может быть немного расширено. LastErrorValue в блоке среды потока может быть установлен в ноль до вызова функции, либо напрямую, либо путем вызова функции kernel32 SetLastError(). Если исключение не произошло, то при возврате из функции значение в этом поле (также возвращаемое функцией GetLastError() ядра32) будет установлено в ERROR_INVALID_HANDLE (6).

B. Дескрипторы

1.Функция OpenProcess

Иногда утверждается, что функция kernel32 OpenProcess() (или функция ntdll NtOpenProcess()) обнаруживает присутствие отладчика при использовании в процессе "csrss.exe". Это неверно. Хотя это правда, что вызов функции будет успешным в присутствии некоторых отладчиков, это происходит из-за побочного эффекта поведения отладчика (в частности, получения привилегии отладки), а не из-за самого отладчика (это должно быть очевидно поскольку вызов функции не удается при использовании с некоторыми отладчиками). Все, что он показывает, - то, что учетная запись пользователя для процесса является членом группы администраторов и имеет привилегию отладки. Причина в том, что успех или неудача вызова функции ограничена только уровнем привилегий процесса. Если учетная запись пользователя процесса является членом группы администраторов и имеет привилегию отладки, то вызов функции будет успешным; если нет, то нет. Обычному пользователю недостаточно приобрести привилегию отладки, и администратор не может успешно вызвать функцию без нее. Идентификатор процесса csrss.exe можно получить с помощью функции ntdll CsrGetProcessId() в Windows XP и более поздних версиях (другие способы существуют для более ранних версий Windows и показаны позже в контексте поиска процесса "Explorer.exe").). Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
call CsrGetProcessId
push eax
push 0
push 1f0fffh ;PROCESS_ALL_ACCESS
call OpenProcess
test eax, eax
jne admin_with_debug_priv
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
call CsrGetProcessId
push rax
pop r8
cdq
mov ecx, 1f0fffh ;PROCESS_ALL_ACCESS
call OpenProcess
test eax, eax
jne admin_with_debug_priv
```

Привилегию отладки можно получить с помощью этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
xor ebx, ebx
push 2 ;SE_PRIVILEGE_ENABLED
push ebx
push ebx
push esp
push offset l1
push ebx
call LookupPrivilegeValueA
push eax
push esp
push 20h ;TOKEN_ADJUST_PRIVILEGES
push -1 ;GetCurrentProcess()
call OpenProcessToken
pop ecx
push eax
mov eax, esp
push ebx
push ebx
push ebx
push eax
push ebx
push ecx
call AdjustTokenPrivileges
...
l1: db "SeDebugPrivilege", 0
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
xor ebx, ebx
push 2 ;SE_PRIVILEGE_ENABLED
push rbx
push rbx
mov r8d, esp
mov rdx, offset l1
xor ecx, ecx
call LookupPrivilegeValueA
push rax
mov r8d, esp
push 20h ;TOKEN_ADJUST_PRIVILEGES
pop rdx
or rcx, -1 ;GetCurrentProcess()
call OpenProcessToken
pop rcx
push rax
mov r8d, esp
push rbx
push rbx
sub esp, 20h
xor r9d, r9d
cdq
call AdjustTokenPrivileges
...
l1: db "SeDebugPrivilege", 0
```

Click to expand...

2.CloseHandle

Один из хорошо известных методов обнаружения отладчика включает функция kernel32 CloseHandle(). Если недопустимый дескриптор передается в функцию ядра 32 CloseHandle() (или непосредственно в функцию ntdll NtClose(), или функцию ядра 32 FindVolumeMountPointClose() в Windows 2000 и более поздних версиях (которая просто вызывает функцию ядра 32 CloseHandle ()), и присутствует отладчик, затем будет вызвано исключение EXCEPTION_INVALID_HANDLE (0xC0000008). Это исключение может быть перехвачено обработчиком исключения, и это указывает на то, что работает отладчик. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor eax, eax
push offset being_debugged
push d fs:[eax]
mov fs:[eax], esp
;any illegal value will do
;must be dword-aligned
;on Windows Vista and later
push esp
call CloseHandle
```

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
mov rdx, offset being_debugged
xor ecx, ecx
inc ecx
call AddVectoredExceptionHandler
;any illegal value will do
;must be dword-aligned
;on Windows Vista and later
mov ecx, esp
call CloseHandle
```

Однако существует второй случай, который предполагает использование защищенного дескриптора. Если защищенный дескриптор передается в функцию CloseHandle() (или непосредственно в функцию ntdll NtClose()) и присутствует отладчик, то будет вызвано исключение EXCEPTION_HANDLE_NOT_CLOSABLE (0xC0000235). Это исключение может быть перехвачено обработчиком исключения, и это указывает на то, что работает отладчик. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor eax, eax
push offset being_debugged
push d fs:[eax]
mov fs:[eax], esp
push eax
push eax
push 3 ;OPEN_EXISTING
push eax
push eax
push eax
push offset l1
call CreateFileA
push 2 ;HANDLE_FLAG_PROTECT_FROM_CLOSE
push -1
push eax
xchg ebx, eax
call SetHandleInformation
push ebx
call CloseHandle
...
l1: db "myfile", 0
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
mov rdx, offset being_debugged
xor ecx, ecx
inc ecx
call AddVectoredExceptionHandler
cdq
push rdx
push rdx
push 3 ;OPEN_EXISTING
sub esp, 20h
xor r9d, r9d
xor r8d, r8d
mov rcx, offset l1
call CreateFileA
mov ebx, eax
push 2 ;HANDLE_FLAG_PROTECT_FROM_CLOSE
pop r8
or rdx, -1
xchg ecx, eax
call SetHandleInformation
mov ecx, ebx
call CloseHandle
...
l1: db "myfile", 0
```

Click to expand...

Попытка победить любой из этих методов проще всего в Windows XP и более поздних версиях, где обработчик исключений FirstHandler Vectored может быть зарегистрирован отладчиком, чтобы скрыть исключение и молча возобновить выполнение. Однако существует проблема прозрачного перехвата функции AddVectoredExceptionHandler() из kernel32, чтобы предотвратить регистрацию другого обработчика в качестве первого обработчика. Недостаточно перехватить попытку зарегистрировать первый обработчик, а затем сделать его последним обработчиком. Причина в том, что это будет обнаружено путем регистрации двух обработчиков и последующего исключения, поскольку обработчики будут вызываться в неправильном порядке. Существует также потенциальная проблема, заключающаяся в том, чтобы перехватить функцию и обнаружить такой измененный запрос, а затем просто снова изменить его. Другой способ чтобы исправить это состоит в том, чтобы дизассемблировать функцию, чтобы найти базовый указатель, который содержит

заголовок списка, но это зависит от платформы. Конечно, поскольку функция возвращает указатель на структуру обработчика, список можно просмотреть, зарегистрировав обработчик и затем проанализировав структуру.

Эта ситуация все еще лучше, чем проблема прозрачного перехвата функции ntdll NtClose() в Windows NT и Windows 2000, чтобы зарегистрировать обработчик структурированных исключений, чтобы скрыть исключение.

Существует флаг, который может быть установлен для создания исключительного поведения, даже если нет отладчика. Устанавливая флаг FLG_ENABLE_CLOSE_EXCEPTIONS (0x400000) в значении реестра "HKLM\System\CurrentControlSet\Control\Session Manager\GlobalFlag", а затем перезагружая, функция CloseHandle() и функция ntdll NtClose() всегда будут вызывать исключение, если в функцию передан неверный или защищенный дескриптор. Эффект является общесистемным и поддерживается во всех версиях Windows для Windows NT, как 32-разрядных, так и 64-разрядных.

Есть другой флаг, который приводит к аналогичному поведению для других функций, которые принимают дескрипторы. Устанавливая флаг FLG_APPLICATION_VERIFIER (0x100) в значении реестра "HKLM\System\CurrentControlSet\Control\Session Manager\GlobalFlag" и затем перезагружая, функция ObtoferenceObjectByHandle() ntoskrnl всегда вызывает исключение, если недопустимый дескриптор передается в функцию (такую как функция SetEvent() kernel32), которая вызывает функцию ntoskrnl ObReferenceObjectByHandle(). Эффект также общесистемный.

Обратите внимание, что один из этих флагов в настоящее время задокументирован неправильно, а другой из этих флагов в настоящее время документирован не полностью. Flag "Enable close exception" задокументированокак вызывающее исключения для недопустимых дескрипторов, которые передаются функциям, отличным от функции ntoskrnl NtClose(), но это неверно; Flag "Enable bad handles detection" вызовет исключения для недопустимых дескрипторов, которые передаются в функции, отличные от функции ntoskrnl NtClose(), но это поведение не задокументировано.

3.Функция CreateFile

Немного ненадежный способ обнаружить присутствие отладчика - попытаться открыть исключительно файл текущего процесса. Когда присутствуют некоторые отладчики, это действие всегда будет неудачным. Причина в том, что при запуске процесса для отладки открывается дескриптор файла. Это позволяет отладчику читать отладочную информацию из файла (при условии, что она присутствует). Значение дескриптора сохраняется в структуре, которая заполняется, когда происходит событие

CREATE_PROCESS_DEBUG_EVENT. Если этот дескриптор не закрыт отладчиком, файл не может быть открыт для монопольного доступа. Поскольку отладчик не открыл файл, было бы легко забыть закрыть его.

Конечно, если какое-либо другое приложение (например, шестнадцатеричный редактор) проверяет файл, то открытие также завершится ошибкой по той же причине. Вот почему метод считается ненадежным, но в зависимости от намерения такие ложные срабатывания могут оказаться приемлемыми. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push 104h ;MAX_PATH
mov ebx, offset l1
push ebx
push 0 ;self filename
call GetModuleFileNameA
cdq
push edx
push edx
push 3 ;OPEN_EXISTING
push edx
push edx
inc edx
ror edx, 1
push edx ;GENERIC_READ
push ebx
call CreateFileA
inc eax
je being_debugged
...
l1: db 104h dup (?) ;MAX_PATH
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows (но метод не работает для 64-битных процессов):

```
mov r8d, 104h ;MAX_PATH
mov rbx, offset l1
push rbx
pop rdx
xor ecx, ecx ;self filename
call GetModuleFileNameA
cdq
push rdx
push rdx
push 3 ;OPEN_EXISTING
sub esp, 20h
xor r9d, r9d
xor r8d, r8d
inc edx
ror edx, 1 ;GENERIC_READ
push rbx
pop rcx
call CreateFileA
inc eax
je being_debugged
...
l1: db 104h dup (?) ;MAX_PATH
```

Click to expand...

Функция kernel32 CreateFile() может также использоваться для обнаружения присутствия драйверов режима ядра, которые могут принадлежать отладчику (или любому другому интересующему инструменту). Инструментам, использующим драйверы режима ядра, также необходим способ связи с этими драйверами. Очень распространенным методом является использование именованных устройств. Таким образом, при попытке открыть такое устройство любой успех указывает на присутствие драйвера. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
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, zero to end>
```

Click to expand...

Типичный список включает в себя следующие имена:

```
db "\\.\EXTREM", 0 ;Phant0m
db "\\.\FILEM", 0 ;FileMon
db "\\.\FILEVXG", 0 ;FileMon
db "\\.\ICEEXT", 0 ;SoftICE Extender
db "\\.\NDBGMSG.VXD", 0 ;SoftICE
db "\\.\NTICE", 0 ;SoftICE
db "\\.\REGSYS", 0 ;RegMon
db "\\.\REGVXG", 0 ;RegMon
db "\\.\RING0", 0 ;Olly Advanced
db "\\.\SICE", 0 ;SoftICE
db "\\.\SIWVID", 0 ;SoftICE
db "\\.\TRW", 0 ;TRW
db "\\.\SPCOMMAND", 0 ;Syser
db "\\.\SYSER", 0 ;Syser
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
mov rdi, offset l2
l1: xor edx, edx
push rdx
push rdx
push 3 ;OPEN_EXISTING
sub esp, 20h
xor r9d, r9d
xor r8d, r8d
push rdi
pop rcx
call CreateFileA
inc eax
jne being_debugged
or ecx, -1
repne scasb
cmp [rdi], al
jne l1
...
l2: <array of ASCIIZ strings, zero to end>
```

Click to expand...

Однако в настоящее время нет 64-битного списка из-за нехватки отладчиков режима ядра, которые предлагают сервисы режима пользователя.

Обратите внимание, что имя драйвера "\\.\ NTICE" действительно только для SoftICE до версии 4.0. SoftICE v4.x не создает устройства с таким именем на платформах под управлением Windows NT. Вместо этого имя устройства - "\\.\NTICExxxx"), где «xxxx» - это четыре шестнадцатеричных символа. Источником символов являются 9-й, 7-й, 5-й и 3-й символы из данных в значении реестра "Serial". Это значение появляется в нескольких местах в реестре. Драйвер SoftICE использует значение реестра "HKLM\System\CurrentContrlSet\Services\NTice\Serial". Функция DevIO_ConnectToSoftICE() использует значение реестра "HKLM\Software\NuMega\SoftIce\Serial". Алгоритм, который использует SoftICE, состоит в том, чтобы перевернуть строку, а затем, начиная с третьего символа, взять каждый второй символ для четырех символов. Конечно, для этого есть более простой

способ. Имя может быть сконструировано с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядных версиях Windows (SoftICE не работает в 64-разрядных версиях Windows):

```
xor ebx, ebx
push eax
push esp
push 1 ;KEY_QUERY_VALUE
push ebx
push offset l2
push 80000002h ;HKLM
call RegOpenKeyExA
pop ecx
push 0dh ;sizeof(l3)
push esp
mov esi, offset l3
push esi
push eax ;REG_NONE
push eax
push offset l4
push ecx
call RegQueryValueExA
push 4
pop ecx
mov edi, offset l6
l1: mov al, [ecx*2+esi+1]
stosb
loop l1
push ebx
push ebx
push 3 ;OPEN_EXISTING
push ebx
push ebx
push ebx
push offset l5
call CreateFileA
inc eax

Click to expand...
```

4. Функция LoadLibrary

Функция LoadLibrary() в kernel32 - удивительно простой и эффективный способ обнаружения отладчика. Когда файл загружается в присутствии отладчика, используя функцию LoadLibrary() kernel32 (или любой из ее вариантов - функцию LoadLibraryEx() kernel32 или функцию ntdll LdrLoadDll()), открывается дескриптор файла. Это позволяет отладчику читать отладочную информацию из файла (при условии, что она присутствует). Значение дескриптора сохраняется в структуре, которая заполняется, когда происходит событие LOAD_DLL_DEBUG_EVENT. Если этот дескриптор не закрыт отладчиком (выгрузка DLL не закроет его), файл не может быть открыт для монопольного доступа. Поскольку отладчик не открыл файл, было бы легко забыть закрыть его. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
mov ebx, offset l1
push ebx
call LoadLibraryA
cdq
push edx
push edx
push 3 ;OPEN_EXISTING
push edx
push edx
inc edx
ror edx, 1 ;GENERIC_READ
push edx
push ebx
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
mov rbx, offset l1
push rbx
pop rcx
call LoadLibraryA
xor edx, edx
push rdx
push rdx
push 3 ;OPEN_EXISTING
sub esp, 20h
xor r9d, r9d
xor r8d, r8d
inc edx
ror edx, 1 ;GENERIC_READ
push rbx
pop rcx
call CreateFileA
inc eax
je being_debugged
...
l1: db "myfile.", 0
```

Click to expand...

Альтернативный метод заключается в вызове другой функции, которая вызывает внутреннюю функцию CreateFile() из kernel32 (или любую ее разновидность) - функцию ntdll NtCreateFile() или функцию ntdll NtOpenFile(). Одним из примеров являются функции обновления ресурсов, такие как функция kernel32 EndUpdateResource(). Причина, по которой работает функция kernel32 EndUpdateResource(), заключается в том, что она в конечном итоге вызывает функцию kernel32 CreateFile() для записи новой таблицы ресурсов. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
mov ebx, offset l1
push ebx
call LoadLibraryA
push 0
push ebx
call BeginUpdateResourceA
push 0
push eax
call EndUpdateResourceA
test eax, eax
je being_debugged
...
l1: db "myfile.", 0
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows (но метод не работает для 64-битных процессов):

```
mov rbx, offset l1
push rbx
pop rcx
call LoadLibraryA
xor edx, edx
push rbx
pop rcx
call BeginUpdateResourceA
cdq
xchg ecx, eax
call EndUpdateResourceA
test eax, eax
je being_debugged
...
l1: db "myfile.", 0
```

Click to expand...

5. Функция ReadFile

Функция kernel32 ReadFile() может использоваться для автоматической модификации потока кода путем считывания содержимого файла в местоположение после вызова функции. Она также может использоваться для удаления программных точек останова, которые отладчик может поместить в поток кода, особенно сразу после вызова функции. Результатом в этом случае будет то, что код выполняется свободно. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
push 104h ;MAX_PATH
mov ebx, offset l2
push ebx
push 0 ;self filename
call GetModuleFileNameA
cdq
push edx
push edx
push 3 ;OPEN_EXISTING
push edx
;FILE_SHARE_READ
;because a debugger might prevent
;exclusive access to the running file
inc edx
push edx
ror edx, 1
push edx ;GENERIC_READ
push ebx
call CreateFileA
push 0
push esp
push 1 ;more bytes might be more useful
push offset l1
push eax
call ReadFile
l1: int 3 ;replaced by "M" from the MZ header
...
l2: db 104h dup (?) ;MAX_PATH

Click to expand...
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
mov r8d, 104h ;MAX_PATH
mov rbx, offset l2
push rbx
pop rdx
xor ecx, ecx ;self filename
call GetModuleFileNameA
cdq
push rdx
push rdx
push 3 ;OPEN_EXISTING
sub esp, 20h
xor r9d, r9d
;FILE_SHARE_READ
;because a debugger might prevent
;exclusive access to the running file
inc edx
push rdx
pop r8
ror edx, 1 ;GENERIC_READ
push rbx
pop rcx
call CreateFileA
push 0
mov r9d, esp
sub esp, 20h
push 1 ;more bytes might be more useful
pop r8
mov rdx, offset l1
xchg ecx, eax
call ReadFile
l1: int 3 ;replaced by "M" from the MZ header
...
l2: db 104h dup (?) ;MAX_PATH
```

Click to expand...

Один из способов преодолеть эту технику - использовать аппаратные контрольные точки вместо программных контрольных точек при переходе через вызовы функций.

C. Время выполнения

При наличии отладчика, который используется для пошагового выполнения кода, существует значительная задержка между выполнением отдельных инструкций по сравнению с собственным выполнением. Эта задержка может быть измерена с использованием одного из нескольких возможных источников времени. Эти источники включают инструкцию RDPMC (однако эта инструкция требует, чтобы в регистре CR4 был установлен флаг PCE, но это не настройка по умолчанию), инструкцию RDTSC , функцию kernel32 GetLocalTime(), функция GetSystemTime(), QueryPerformanceCounter(), GetTickCount(), функция ntoskrnl KiGetTickCount() (предоставляется через интерфейс прерываний 0x2A в 32-разрядных версиях Windows) и функция winmm timeGetTime(). Однако разрешение функции winmm timeGetTime() является переменным, в зависимости от того, является ли оно внутренним или внешнним по отношению к функции GetTickCount(), что делает очень ненадежным измерение небольших интервалов. Инструкция RDMSR также может использоваться в качестве источника времени, но ее нельзя использовать в пользовательском режиме. Проверка может быть выполнена для инструкции RDPMC с использованием этого кода для проверки 32-разрядной или 64-разрядной среды Windows:

```
xor ecx, ecx ;read 32-bit counter 0
rdpmc
xchg ebx, eax
rdpmc
sub eax, ebx
cmp eax, 500h
jnbe being_debugged
```

Проверка может быть выполнена для инструкции RDTSC с использованием этого кода (идентичного для 32-битной и 64-битной) для проверки 32-битной или 64-битной среды Windows:

```
rdtsc
xchg esi, eax
mov edi, edx
rdtsc
sub eax, esi
sbb edx, edi
jne being_debugged
cmp eax, 500h
jnbe being_debugged
```

Проверка может быть выполнена для функции GetLocalTime() kernel32, использующей этот 32-разрядный код для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
mov ebx, offset l1
push ebx
call GetLocalTime
mov ebp, offset l2
push ebp
call GetLocalTime
mov esi, offset l3
push esi
push ebx
call SystemTimeToFileTime
mov edi, offset l4
push edi
push ebp
call SystemTimeToFileTime
mov eax, [edi]
sub eax, [esi]
mov edx, [edi+4]
sbb edx, [esi+4]
jne being_debugged
cmp eax, 10h
jnbe being_debugged
...
l1: db 10h dup (?) ;sizeof(SYSTEMTIME)
l2: db 10h dup (?) ;sizeof(SYSTEMTIME)
l3: db 8 dup (?) ;sizeof(FILETIME)
l4: db 8 dup (?) ;sizeof(FILETIME)
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
mov rbx, offset l1
push rbx
pop rcx
call GetLocalTime
mov rbp, offset l2
push rbp
pop rcx
call GetLocalTime
mov rsi, offset l3
push rsi
pop rdx
push rbx
pop rcx
call SystemTimeToFileTime
mov rdi, offset l4
push rdi
pop rdx
push rbp
pop rcx
call SystemTimeToFileTime
mov rax, [rdi]
sub rax, [rsi]
cmp rax, 10h
jnbe being_debugged
...
l1: db 10h dup (?) ;sizeof(SYSTEMTIME)
l2: db 10h dup (?) ;sizeof(SYSTEMTIME)
l3: db 8 dup (?) ;sizeof(FILETIME)
l4: db 8 dup (?) ;sizeof(FILETIME)
```

Click to expand...

Проверка может быть выполнена для функции GetSystemTime() с использованием точно такого же кода, что и для функции GetLocalTime() , за исключением изменения имени функции.

Можно выполнить проверку для функции GetTickCount() kernel32, используя этот код (идентичный для 32-битной и 64-битной), чтобы проверить 32-битную или 64-битную среду Windows:

```
call GetTickCount
xchg ebx, eax
call GetTickCount
sub eax, ebx
cmp eax, 10h
jnbe being_debugged
```

Проверка может быть выполнена для функции ntoskrnl KiGetTickCount(), использующей этот 32-разрядный код для проверки 32-разрядной среды Windows в 32-разрядных версиях Windows (прерывание не поддерживается в 64-разрядных версиях Windows):

```
int 2ah
xchg ebx, eax
int 2ah
sub eax, ebx
cmp eax, 10h
jnbe being_debugged
```

Проверка может быть выполнена для функции kernel32 QueryPerformanceCounter(), использующей этот 32-разрядный код для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
mov esi, offset l1
push esi
call QueryPerformanceCounter
mov edi, offset l2
push edi
call QueryPerformanceCounter
mov eax, [edi]
sub eax, [esi]
mov edx, [edi+4]
sbb edx, [esi+4]
jne being_debugged
cmp eax, 10h
jnbe being_debugged
...
l1: db 8 dup (?) ;sizeof(LARGE_INTEGER)
l2: db 8 dup (?) ;sizeof(LARGE_INTEGER)
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
mov rsi, offset l1
push rsi
pop rcx
call QueryPerformanceCounter
mov rdi, offset l2
push rdi
pop rcx
call QueryPerformanceCounter
mov rax, [rdi]
sub rax, [rsi]
cmp rax, 10h
jnbe being_debugged
...
l1: db 8 dup (?) ;sizeof(LARGE_INTEGER)
l2: db 8 dup (?) ;sizeof(LARGE_INTEGER)
```

Click to expand...

Можно выполнить проверку для функции winmm timeGetTime (), используя этот код (идентичный для 32-разрядного и 64-разрядного), чтобы проверить 32-разрядную или 64-разрядную среду Windows:

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

D.Process-level

1.Функция CheckRemoteDebuggerPresent

Функция kernel32 CheckRemoteDebuggerPresent() была введена в Windows XP с пакетом обновления 1 (SP1) для запроса значения, существовавшего со времени Windows NT. Remote в этом смысле означает отдельный процесс на одной машине. Функция устанавливает в 0xffffffff значение, на которое указывает аргумент pbDebuggerPresent, если присутствует отладчик (то есть присоединенный к текущему процессу). Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

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

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
enter 20h, 0
mov edx, ebp
or rcx, -1 ;GetCurrentProcess()
call CheckRemoteDebuggerPresent
leave
test ebp, ebp
jne being_debugged
```

2.Родительский процесс

Пользователи обычно запускают приложения, щелкая значок, отображаемый процессом оболочки (Explorer.exe). В результате родительский процесс исполняемого процесса будет Explorer.exe. Конечно, если приложение выполняется из командной строки, то родительский процесс исполняемого процесса будет процессом командной строки. Выполнение приложения путем отладки приведет к тому, что родительский процесс исполняемого процесса станет процессом отладчика.

Выполнение приложений из командной строки может вызвать проблемы для некоторых приложений, поскольку они ожидают, что родительский процесс будет Explorer.exe. Некоторые приложения проверяют имя родительского процесса, ожидая, что оно будет "Explorer.exe". Некоторые приложения сравнивают идентификатор родительского процесса с идентификатором Explorer.exe. Несоответствие в любом случае может привести к тому, что приложение будет думать, что оно отлаживается.

На этом этапе мы сделаем небольшой обход и представим тему, которая логически должна появиться позже. Самый простой способ получить идентификатор процесса Explorer.exe - вызвать функции user32 GetShellWindow() и user32 GetWindowThreadProcessId(). Это оставляет идентификатор процесса и имя родительского процесса текущего процесса, что можно получить, вызвав функцию ntdll NtQueryInformationProcess() с классом ProcessBasicInformation. Вызовы могут быть сделаны с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

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

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
call GetShellWindow
enter 20h, 0
mov edx, ebp
xchg ecx, eax
call GetWindowThreadProcessId
leave
push 0
sub esp, 20h
push 30h ;sizeof(PROCESS_BASIC_INFORMATION)
pop r9
mov rbx, offset l1
push rbx
pop r8
cdq ;ProcessBasicInformation
or rcx, -1 ;GetCurrentProcess()
call NtQueryInformationProcess
;InheritedFromUniqueProcessId
cmp [rbx+20h], ebp
jne being_debugged
...
;sizeof(PROCESS_BASIC_INFORMATION)
l1: db 30h dup (?)
```

Click to expand...

Однако этот код имеет серьезную проблему, заключающуюся в том, что в одном сеансе может быть несколько экземпляров Explorer.exe, если значение реестра "HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced\SeparateProcess" (введено в Windows 2000) не равно нулю. Это приводит к запуску отдельной копии Explorer.exe для каждого открытого окна. В результате окно оболочки может не быть родительским процессом текущего процесса, и все же Explorer.exe является именем родительского процесса.

3. Функция CreateToolhelp32Snapshot

Идентификатор процесса Explorer.exe и родительского процесса текущего процесса, а также имя этого родительского процесса можно получить с помощью функции CreateToolhelp32Snapshot() и перечисления функции Process32Next (). Вызов может быть выполнен с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

Last edited: Aug 13, 2020

yashechka

Генератор контента.Фанат Ильфака и Рикардо Нарвахи

Эксперт

Joined
Nov 24, 2012

Messages
1,807

Reaction score
2,613

3. Функция CreateToolhelp32Snapshot

Идентификатор процесса Explorer.exe и родительского процесса текущего процесса, а также имя этого родительского процесса можно получить с помощью функции CreateToolhelp32Snapshot() и перечисления функции Process32Next (). Вызов может быть выполнен с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor esi, esi
xor edi, edi
push esi
push 2 ;TH32CS_SNAPPROCESS
call CreateToolhelp32Snapshot
mov ebx, offset l9
xchg ebp, eax
l1: push ebx
push ebp
call Process32First
l2: mov eax, fs:[eax+1fh] ;UniqueProcess
cmp [ebx+8], eax ;th32ProcessID
cmove edi, [ebx+18h] ;th32ParentProcessID
test edi, edi
je l3
cmp esi, edi
je l7
l3: lea ecx, [ebx+24h] ;szExeFile
push esi
mov esi, ecx
l4: lodsb
cmp al, "\"
cmove ecx, esi
or b [esi-1], " "
test al, al
jne l4
sub esi, ecx
xchg ecx, esi
push edi
mov edi, offset l8
repe cmpsb
pop edi
pop esi
jne l6
test esi, esi
je l5
mov esi, offset l10
cmp cl, [esi]
adc [esi], ecx
l5: mov esi, [ebx+8] ;th32ProcessID
l6: push ebx
```

> Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor esi, esi
xor edi, edi
xor edx, edx
push 2 ;TH32CS_SNAPPROCESS
pop rcx
call CreateToolhelp32Snapshot
mov rbx, offset l9
xchg ebp, eax
l1: push rbx
pop rdx
mov ecx, ebp
call Process32First
l2: mov eax, gs:[rax+3fh] ;UniqueProcess
cmp [rbx+8], eax ;th32ProcessID
cmove edi, [rbx+20h] ;th32ParentProcessID
test esi, esi
je l3
cmp esi, edi
je l7
l3: lea ecx, [rbx+2ch] ;szExeFile
push rsi
mov esi, ecx
l4: lodsb
cmp al, "\"
cmove ecx, esi
or b [rsi-1], " "
test al, al
jne l4
sub esi, ecx
xchg ecx, esi
push rdi
mov rdi, offset l8
repe cmpsb
pop rdi
pop rsi
jne l6
test esi, esi
je l5
mov rsi, offset l10
cmp cl, [rsi]
adc [rsi], ecx
```

```
l5: mov esi, [rbx+8] ;th32ProcessID
l6: push rbx
pop rdx
mov ecx, ebp
call Process32Next
test eax, eax
jne l2
dec b [offset l10+1]
jne l1
jmp being_debugged
l7: ...
;trailing zero is converted to space
l8: db "explorer.exe "
l9: dd 130h ;sizeof(PROCESSENTRY32)
db 12ch dup (?)
l10:db 0ffh, 1, ?, ?
```

Click to expand...

Поскольку эта информация поступает из ядра, для кода пользовательского режима нет простого способа предотвратить обнаружение отладчиком этого вызова. Обычный метод, который пытается победить его, состоит в том, чтобы принудительно заставить функцию Process32Next() вернуть FALSE, что приводит к преждевременному завершению цикла. Однако это должно быть подозрительным условием, если либо Explorer.exe, либо текущий процесс не был замечен.

Код будет выполняться за один проход, если существует только одна копия Explorer.exe. Если существует несколько копий, код выполнит второй проход. При первом проходе будет получен идентификатор родительского процесса. На втором этапе идентификатор родительского процесса будет сравниваться с идентификатором процесса каждого экземпляра Explorer.exe.

Однако в этом коде есть небольшая проблема, заключающаяся в том, что если одновременно вошли в систему несколько пользователей, их процессы также будут видны. По крайней мере один из них также будет Explorer.exe, и он не будет родительским процессом любого процесса в этом сеансе. Это заставит код выполнить два прохода, даже если будет достаточно только одного из них, поскольку в текущем сеансе существует только один экземпляр Explorer.exe.

Один из способов избежать этой проблемы - определить имя пользователя и доменное имя процесса и сопоставить его, прежде чем принимать процесс в том виде, как он найден. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor esi, esi
xor edi, edi
push esi
push 2 ;TH32CS_SNAPPROCESS
call CreateToolhelp32Snapshot
mov ebx, offset l16
xchg ebp, eax
l1: push ebx
push ebp
call Process32First
l2: mov eax, fs:[eax+1fh] ;UniqueProcess
cmp [ebx+8], eax ;th32ProcessID
cmove edi, [ebx+18h] ;th32ParentProcessID
test edi, edi
je l3
cmp esi, edi
je l9
l3: lea ecx, [ebx+24h] ;szExeFile
push esi
mov esi, ecx
l4: lodsb
cmp al, "\"
cmove ecx, esi
or b [esi-1], " "
test al, al
jne l4
sub esi, ecx
xchg ecx, esi
push edi
mov edi, offset l15
repe cmpsb
pop edi
pop esi
jne l8
mov eax, [ebx+8] ;th32ProcessID
push ebx
push ebp
push esi
push edi
call l10
dec ecx ;invert Z flag
```

```
jne l6
push ebx
push edi
dec ecx
call l11
pop esi
pop edx
mov cl, 2
;compare user names
;then domain names
l5: lodsb
scasb
jne l6
test al, al
jne l5
mov esi, ebx
mov edi, edx
loop l5
l6: pop edi
pop esi
pop ebp
pop ebx
jne l8
test esi, esi
je l7
mov esi, offset l17
cmp cl, [esi]
adc [esi], ecx
l7: mov esi, [ebx+8] ;th32ProcessID
l8: push ebx
push ebp
call Process32Next
test eax, eax
jne l2
dec b [offset l17+1]
jne l1
jmp being_debugged
l9: ...
l10:push eax
push 0
push 400h ;PROCESS_QUERY_INFORMATION
```

```
call OpenProcess
xchg ecx, eax
jecxz l14
l11:push eax
push esp
push 8 ;TOKEN_QUERY
push ecx
call OpenProcessToken
pop ebx
xor ebp, ebp
l12:push ebp
push 0 ;GMEM_FIXED
call GlobalAlloc
push eax
push esp
push ebp
push eax
push 1 ;TokenUser
push ebx
xchg esi, eax
call GetTokenInformation
pop ebp
xchg ecx, eax
jecxz l12
xor ebp, ebp
l13:push ebp
push 0 ;GMEM_FIXED
call GlobalAlloc
xchg ebx, eax
push ebp
push 0 ;GMEM_FIXED
call GlobalAlloc
xchg edi, eax
push eax
mov eax, esp
push ebp
mov ecx, esp
push ebp
mov edx, esp
push eax
push ecx
```

```
push ebx
push edx
push edi
push d [esi]
push 0
call LookupAccountSidA
pop ecx
pop ebp
pop edx
xchg ecx, eax
jecxz l13
l14:ret
;trailing zero is converted to space
l15:db "explorer.exe "
l16:dd 128h ;sizeof(PROCESSENTRY32)
db 124h dup (?)
l17:db 0ffh, 1, ?, ?
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor esi, esi
xor edi, edi
xor edx, edx
push 2 ;TH32CS_SNAPPROCESS
pop rcx
call CreateToolhelp32Snapshot
mov rbx, offset l16
xchg ebp, eax
l1: push rbx
pop rdx
mov ecx, ebp
call Process32First
l2: mov eax, gs:[rax+3fh] ;UniqueProcess
cmp [rbx+8], eax ;th32ProcessID
cmove edi, [rbx+20h] ;th32ParentProcessID
test esi, esi
je l3
cmp esi, edi
je l9
l3: lea ecx, [rbx+2ch] ;szExeFile
push rsi
mov esi, ecx
l4: lodsb
cmp al, "\"
cmove ecx, esi
or byte [rsi-1], " "
test al, al
jne l4
sub esi, ecx
xchg ecx, esi
push rdi
mov rdi, offset l15
repe cmpsb
pop rdi
pop rsi
jne l8
mov r8d, [rbx+8] ;th32ProcessID
push rbx
push rbp
push rsi
push rdi
```

```
call l10
dec ecx ;invert Z flag
jne l6
push rbx
push rdi
dec rcx
call l11
pop rsi
pop rdx
mov cl, 2
;compare user names
;then domain names
l5: lodsb
scasb
jne l6
test al, al
jne l5
mov esi, ebx
mov edi, edx
loop l5
l6: pop rdi
pop rsi
pop rbp
pop rbx
jne l8
test esi, esi
je l7
mov rsi, offset l17
cmp cl, [rsi]
adc [rsi], ecx
l7: mov esi, [rbx+8] ;th32ProcessID
l8: push rbx
pop rdx
mov ecx, ebp
call Process32Next
test eax, eax
jne l2
dec b [offset l17+1]
jne l1
jmp being_debugged
l9: ...
```

```
l10:cdq
xor ecx, ecx
mov ch, 4 ;PROCESS_QUERY_INFORMATION
enter 20h, 0
call OpenProcess
leave
xchg ecx, eax
jrcxz l14
l11:push rax
mov r8d, esp
push 8 ;TOKEN_QUERY
pop rdx
call OpenProcessToken
pop rbx
xor ebp, ebp
l12:mov edx, ebp
xor ecx, ecx ;GMEM_FIXED
enter 20h, 0
call GlobalAlloc
leave
push rbp
pop r9
push rax
pop r8
push rax ;simulate enter
mov ebp, esp
push rbp
sub esp, 20h
push 1 ;TokenUser
pop rdx
mov ecx, ebx
xchg esi, eax
call GetTokenInformation
leave
xchg ecx, eax
jrcxz l12
xor ebp, ebp
l13:mov ebx, ebp
mov edx, ebp
xor ecx, ecx ;GMEM_FIXED
enter 20h, 0
```

```
call GlobalAlloc
xchg ebx, eax
xchg edx, eax
xor ecx, ecx ;GMEM_FIXED
call GlobalAlloc
leave
xchg edi, eax
push rbp
mov ecx, esp
push rbp
mov r9d, esp
push rax
push rsp
push rcx
push rbx
sub esp, 20h
push rdi
pop r8
mov edx, [rsi]
xor ecx, ecx
call LookupAccountSidA
add esp, 40h
pop rcx
pop rbp
xchg ecx, eax
jrcxz l13
l14:ret
;trailing zero is converted to space
l15:db "explorer.exe "
l16:dd 130h ;sizeof(PROCESSENTRY32)
db 12ch dup (?)
l17:db 0ffh, 1, ?, ?
```

Click to expand...

Обратите внимание, что этот код предназначен только для демонстрационных целей. Он демонстрирует ряд плохих практик программирования, он позволяет сделать утечку как дескрипторов, так и памяти, и он не предназначен для использования в каком-либо виде продукта.

Разновидность этого метода ищет имена конкретных инструментов, которые могут указывать на наличие отладчика или аналогичного. В этом случае присутствие инструментов в сеансах других вошедших в систему пользователей может быть приемлемым. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push 0
push 2 ;TH32CS_SNAPPROCESS
call CreateToolhelp32Snapshot
mov ebx, offset l6
xchg ebp, eax
l1: push ebx
push ebp
call Process32First
l2: lea ecx, [ebx+24h] ;szExeFile
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
mov edi, offset l5
l4: push ecx
push esi
repe cmpsb
pop esi
pop ecx
je being_debugged
push ecx
or ecx, -1
repne scasb
pop ecx
cmp [edi], al
jne l4
push ebx
push ebp
call Process32Next
test eax, eax
jne l2
...
l5: <array of ASCII strings
space then zero to end each one
zero to end the list
```

>
> l6: dd 128h ;sizeof(PROCESSENTRY32)
> db 124h dup (?)
>
> Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor edx, edx
push 2 ;TH32CS_SNAPPROCESS
pop rcx
call CreateToolhelp32Snapshot
mov rbx, offset l6
xchg ebp, eax
l1: push rbx
pop rdx
mov ecx, ebp
call Process32First
l2: lea ecx, [rbx+2ch] ;szExeFile
mov esi, ecx
l3: lodsb
cmp al, "\"
cmove ecx, esi
or b [rsi-1], " "
test al, al
jne l3
sub esi, ecx
xchg ecx, esi
mov rdi, offset l5
l4: push rcx
push rsi
repe cmpsb
pop rsi
pop rcx
je being_debugged
push rcx
or ecx, -1
repne scasb
pop rcx
cmp [rdi], al
jne l4
push rbx
pop rdx
mov ecx, ebp
call Process32Next
test eax, eax
jne l2
...
l5: <array of ASCII strings
```

> space then zero to end each one
> zero to end the list
> >
> l6: dd 130h ;sizeof(PROCESSENTRY32)
> db 12ch dup (?)
>
> Click to expand...

Обратите внимание, что может возникнуть проблема с обнаружением процесса, имя которого действительно содержит пробел, поскольку не делается никаких попыток провести различие между реальным именем процесса и процессом, имя которого является подстрокой, которая заканчивается именно там, где появляется пробел. Однако, опять же, такая ситуация может быть приемлемой.

4. Функция DbgBreakPoint

Функция ntdll DbgBreakPoint() вызывается, когда отладчик подключается к уже запущенному процессу. Функция позволяет отладчику получить контроль, потому что возникает исключение, которое может перехватить отладчик. Однако для этого необходимо, чтобы функция оставалась нетронутой. Если функция изменена, ее можно использовать для выявления присутствия отладчика или для запрета подключения. Это присоединение может быть предотвращено простым стиранием точки останова. Патч можно сделать с помощью этого 32-битного кода в 32-битной или 64-битной версии Windows:

```
push offset l1
call GetModuleHandleA
push offset l2
push eax
call GetProcAddress
push eax
push esp
push 40h ;PAGE_EXECUTE_READWRITE
push 1
push eax
xchg ebx, eax
call VirtualProtect
mov b [ebx], 0c3h
...
l1: db "ntdll", 0
l2: db "DbgBreakPoint", 0
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
mov rcx, offset l1
call GetModuleHandleA
mov rdx, offset l2
xchg rcx, rax
call GetProcAddress
push rax
pop rbx
enter 20h, 0
push rbp
pop r9
push 40h ;PAGE_EXECUTE_READWRITE
pop r8
xor edx, edx
inc edx
xchg rcx, rax
call VirtualProtect
mov b [rbx], 0c3h
...
l1: db "ntdll", 0
l2: db "DbgBreakPoint", 0
```

Click to expand...

5. Функция DbgPrint

Обычно функция ntdll DbgPrint() вызывает исключение, но зарегистрированный обработчик структурированных исключений его не видит. Причина в том, что Windows регистрирует свой собственный структурированный обработчик исключений внутри. Этот обработчик будет использовать исключение, если отладчик этого не делает. ""Обычно" в данном случае относится к тому факту, что исключение может быть подавлено в Windows 2000 и более поздних версиях.

В Windows NT возникает проблема, если функция ntdll _vsnprintf () подключена, а также вызывается функцию ntdll DbgPrint(). Результатом в этом случае будет рекурсивный вызов, пока не произойдет переполнение стека. Эта ошибка была исправлена в Windows 2000 путем добавления флага "busy" со смещением 0xf74 в Блоке Окружения Потока. Функция немедленно завершится, если установлен флаг. Однако ошибка - была вновь введена в функцию ntdll DbgPrintReturnControlC(), которая была добавлена в Windows 2000, и она также присутствует в Windows XP. Эта

ошибка была исправлена в Windows Vista путем добавления флага "busy"» в бите 1 со смещением 0xfca в Блоке Окружения Потока (но также с удалением флага занятости со смещением 0xf74 в блоке среды потока).

Windows XP представила функцию ntdll DbgPrintEx() и функцию ntdll vDbgPrintEx(), что еще больше расширило поведение. Обе функции принимают уровень серьезности в качестве параметра. Если значение не равно 0xffffffff, то вызывается функция ntdll NtQueryDebugFilterState() с идентификатором компонента и уровнем серьезности. Если соответствующая запись равна нулю в таблице отладчика ядра, то функция немедленно возвращается.

В Windows XP существует более серьезная ошибка: функция ntdll vDbgPrintExWithPrefix() не проверяет длину префикса при копировании его в буфер стека. Если строка достаточно длинная, адрес возврата (и, необязательно, запись обработчика структурированных исключений) можно заменить, что может привести к выполнению произвольного кода. Однако основным смягчающим фактором здесь является то, что это внутренняя функция, которую не следует вызывать напрямую. Все открытые функции, которые вызывают функцию ntdll vDbgPrintExWithPrefix(), используют пустой префикс. До Windows XP выполнялась копия фиксированной длины. Поскольку в большинстве случаев это было неоправданно большим и могло привести к сбоям из-за недопустимых операций чтения, в Windows XP его заменили строковой копией. Однако длина строки не была проверена перед выполнением копирования, что привело к переполнению буфера. Известно, что такого рода ошибки существуют как минимум в одной другой DLL в Windows XP.

Windows Vista снова изменила поведение. Если присутствует отладчик и уровень серьезности равен 0x65, то функция ntdll NtQueryDebugFilterState() не вызывается. Если уровень серьезности не равен 0xffffffff, а отладчик отсутствует или уровень серьезности не равен 0x65, то будет вызываться функция ntdll NtQueryDebugFilterState() с идентификатором компонента и уровнем серьезности. Как и раньше, если соответствующая запись равна нулю в таблице отладчика ядра, то функция возвращает немедленно. Флаг занятости проверяется только в этой точке, и функция немедленно возвращается, если флаг установлен.

Если присутствует отладчик пользовательского режима (который определяется чтением флага BeingDebugged в Блоке Окружения Процесса), или отладчик режима ядра отсутствует в Windows Vista и более поздних версиях (что определяется чтением члена KdDebuggerEnabled элемента KUSER_SHARED_DATA структура, со смещением 0x7ffe02d4 для конфигураций пространства пользователя 2 ГБ), тогда Windows вызовет исключение DBG_PRINTEXCEPTION_C (0x40010006). Если отладчик в

пользовательском режиме отсутствует и если в Windows Vista и более поздних версиях присутствует отладчик в режиме ядра, Windows выполнит прерывание 0x2d (которое не приводит к исключению) и затем вернется.

В Windows XP и более поздних версиях, если возникает исключение, любой зарегистрированный векторный обработчик исключений будет работать до структурированного обработчика исключений, который регистрирует Windows. Это может считаться ошибкой в Windows.

6.DbgSetDebugFilterState

Функцию ntdll DbgSetDebugFilterState() (или функцию ntdll NtSetDebugFilterState(), обе представленные в Windows XP) нельзя использовать для обнаружения присутствия отладчика, несмотря на то, что предполагает его имя, поскольку она просто устанавливает флаг в таблице, который будет проверяется отладчиком режима ядра, если он присутствовал. Хотя это правда, что вызов функции будет успешным в присутствии некоторых отладчиков, это происходит из-за побочного эффекта поведения отладчика (в частности, получения привилегии отладки), а не из-за самого отладчика (это должно быть очевидно поскольку вызов функции не удается при использовании с некоторыми отладчиками). Все, что он показывает, это то, что учетная запись пользователя для процесса является членом группы администраторов и имеет привилегию отладки. Причина в том, что успех или неудача вызова функции ограничена только уровнем привилегий процесса. Если учетная запись пользователя процесса является членом группы администраторов и имеет привилегию отладки, то вызов функции будет успешным; если нет, то нет. Обычному пользователю недостаточно приобрести привилегию отладки, и администратор не может успешно вызвать функцию без нее. Вызов фильтра можно выполнить с помощью этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
push 1
push 0
push 0
call NtSetDebugFilterState
xchg ecx, eax
jecxz admin_with_debug_priv
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
push 1
pop r8
xor edx, edx
xor ecx, ecx
call NtSetDebugFilterState
xchg ecx, eax
jecxz admin_with_debug_priv
```

7.IsDebuggerPresent

Функция kernel32 IsDebuggerPresent() была представлена в Windows 95. Возвращает ненулевое значение, если присутствует отладчик. Проверка может быть выполнена с использованием этого кода (идентичного для 32-битной и 64-битной), чтобы проверить 32-битную или 64-битную среду Windows:

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

Внутри функция просто возвращает значение флага BeingDebugged. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
mov eax, fs:[30h] ;Process Environment Block
cmp b [eax+2], 0 ;check BeingDebugged
jne being_debugged
```

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
push 60h
pop rsi
gs:lodsq ;Process Environment Block
cmp b [rax+2], 0 ;check BeingDebugged
jne being_debugged
```

или используя этот 32-битный код для проверки 64-битной среды Windows:

```
mov eax, fs:[30h] ;Process Environment Block
;64-bit Process Environment Block
;follows 32-bit Process Environment Block
cmp b [eax+1002h], 0 ;check BeingDebugged
jne being_debugged
```

Чтобы победить эти методы, нужно только установить флаг BeingDebugged в ноль.

8.NtQueryInformationProcess

a.ProcessDebugPort

Функция ntdll NtQueryInformationProcess() принимает параметр, который является классом информации для запроса. Большинство классов не документированы. Однако одним из документированных классов является ProcessDebugPort(7). Можно запросить наличие (а не значение) порта. Возвращаемое значение равно 0xffffffff, если процесс отлаживается. Внутри функция запрашивает ненулевое состояние поля DebugPort в структуре EPROCESS. Внутренне, так работает функция ядра32 CheckRemoteDebuggerPresent(). Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

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

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor ebp, ebp
enter 20h, 0
push 8 ;ProcessInformationLength
pop r9
push rbp
pop r8
push 7 ;ProcessDebugPort
pop rdx
or rcx, -1 ;GetCurrentProcess()
call NtQueryInformationProcess
leave
test ebp, ebp
jne being_debugged
```

Click to expand...

Поскольку эта информация поступает из ядра, для кода пользовательского режима нет простого способа предотвратить обнаружение отладчиком этого вызова.

b.ProcessDebugObjectHandle

В Windows XP появился объект отладки. Когда начинается сеанс отладки, создается объект отладки, и с ним связывается дескриптор. Можно запросить значение этого дескриптора, используя недокументированный класс ProcessDebugObjectHandle (0x1e). Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

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

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor ebp, ebp
enter 20h, 0
push 8 ;ProcessInformationLength
pop r9
push rbp
pop r8
push 1eh ;ProcessDebugObjectHandle
pop rdx
or rcx, -1 ;GetCurrentProcess()
call NtQueryInformationProcess
leave
test ebp, ebp
jne being_debugged
```

Click to expand...

Поскольку эта информация поступает из ядра, для кода пользовательского режима нет простого способа предотвратить обнаружение отладчиком этого вызова. Обратите внимание, что запрос дескриптора объекта отладки в 64-битной системе при наличии отладчика может увеличить количество ссылок дескриптора, тем самым предотвращая завершение отладчика.

c.ProcessDebugFlags

Недокументированный класс ProcessDebugFlags (0x1f) возвращает обратное значение бита NoDebugInherit в структуре EPROCESS. То есть возвращаемое значение равно нулю, если присутствует отладчик. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
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
```

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor ebp, ebp
enter 20h, 0
push 4 ;ProcessInformationLength
pop r9
push rbp
pop r8
push 1fh ;ProcessDebugFlags
pop rdx
or rcx, -1 ;GetCurrentProcess()
call NtQueryInformationProcess
leave
test ebp, ebp
je being_debugged
```

Click to expand...

Поскольку эта информация поступает из ядра, для кода пользовательского режима нет простого способа предотвратить обнаружение отладчиком этого вызова.

9.OutputDebugString

Функция kernel32 OutputDebugString() может демонстрировать различное поведение в зависимости от версии Windows и наличия отладчика. Наиболее очевидное различие в поведении заключается в том, что функция kernel32 GetLastError() вернет ноль, если отладчик присутствует, и ненулевой, если отладчик отсутствует. Однако это относится только к Windows NT/ 2000/XP. В Windows Vista и более поздних версиях код ошибки не изменяется во всех случаях. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor ebp, ebp
push ebp
push esp
call OutputDebugStringA
cmp fs:[ebp+34h], ebp ;LastErrorValue
je being_debugged
```

или этого 64-битный кода для проверки 64-битной среды Windows:

```
xor ebp, ebp
enter 20h, 0
mov ecx, ebp
call OutputDebugStringA
cmp gs:[rbp+68h], ebp ;LastErrorValue
je being_debugged
```

Причина, по которой это сработало, заключалась в том, что Windows попыталась открыть сопоставление с объектом с именем DBWIN_BUFFER. Когда это не удалось, устанавливается код ошибки. После этого был вызов функции ntdll DbgPrint(). Как отмечено выше, если присутствует отладчик, исключение может быть использовано отладчиком, в результате чего код ошибки будет очищен. Если отладчик отсутствует, то это исключение будет использовано Windows, и код ошибки останется. Однако в Windows Vista и более поздних версиях код ошибки восстанавливается до значения, которое было до вызова функции OutputDebugString() из kernel32. Это не очищается явно, в результате чего этот метод обнаружения становится совершенно ненадежным.

Эта функция, пожалуй, наиболее известна из-за ошибки в OllyDbg v1.10, возникшей в результате ее использования. OllyDbg передает пользовательские данные непосредственно в функцию msvcrt _vsprintf(). Эти данные могут содержать токены форматирования строки. Определенный токен в определенной позиции заставит функцию пытаться получить доступ к памяти, используя один из переданных параметров. Существует целый ряд вариантов атаки, все из которых представляют собой произвольно выбранные комбинации токенов, которые работают. Однако все, что требуется, это три токена. Первые два токена полностью произвольны. Третий токен должен быть "% s". Это связано с тем, что функция _vsprintf() вызывает функцию __vprinter() и передает ноль в качестве четвертого параметра. Четвертый параметр доступен третьим токеном, если там используется "% s". Результатом является доступ с нулевым указателем и сбой. Ошибка не может быть использована для выполнения произвольного кода.

10.RtlQueryProcessHeapInformation

Функция ntdll RtlQueryProcessHeapInformation() может использоваться для чтения флагов кучи из памяти процесса текущего процесса. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows, если версия подсистемы находится (или может быть) в диапазоне 3.10-3.50:

```
push 0
push 0
call RtlCreateQueryDebugBuffer
push eax
xchg ebx, eax
call RtlQueryProcessHeapInformation
mov eax, [ebx+38h] ;HeapInformation
mov eax, [eax+8] ;Flags
;neither CREATE_ALIGN_16
;nor HEAP_SKIP_VALIDATION_CHECKS
and eax, 0effeffffh
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
cmp eax, 40000062h
je being_debugged
```

Click to expand...

или использую этот 32-разрядный код для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows, если версия подсистемы составляет 3,51 или выше:

```
push 0
push 0
call RtlCreateQueryDebugBuffer
push eax
xchg ebx, eax
call RtlQueryProcessHeapInformation
mov eax, [ebx+38h] ;HeapInformation
mov eax, [eax+8] ;Flags
bswap eax
;not HEAP_SKIP_VALIDATION_CHECKS
and al, 0efh
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
cmp eax, 62000040h
je being_debugged
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor edx, edx
xor ecx, ecx
call RtlCreateQueryDebugBuffer
mov ebx, eax
xchg ecx, eax
call RtlQueryProcessHeapInformation
mov eax, [rbx+70h] ;HeapInformation
;Flags
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
cmp d [rax+10h], 40000062h
je being_debugged
```

Click to expand...

11.RtlQueryProcessDebugInformation

Функция ntdll RtlQueryProcessDebugInformation() может использоваться для чтения определенных полей из памяти процесса запрошенного процесса, включая флаги кучи. Функция делает это для флагов кучи, вызывая внутреннюю функцию ntdll RtlQueryProcessHeapInformation(). Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows, если версия подсистемы находится (или может быть) в диапазоне 3.10-3.50:

```
xor ebx, ebx
push ebx
push ebx
call RtlCreateQueryDebugBuffer
push eax
xchg ebx, eax
push 14h ;PDI_HEAPS + PDI_HEAP_BLOCKS
push d fs:[eax+20h] ;UniqueProcess
call RtlQueryProcessDebugInformation
mov eax, [ebx+38h] ;HeapInformation
mov eax, [eax+8] ;Flags
;neither CREATE_ALIGN_16
;nor HEAP_SKIP_VALIDATION_CHECKS
and eax, 0effeffffh
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
cmp eax, 40000062h
je being_debugged
```

Click to expand...

или использую этот 32-разрядный код для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows, если версия подсистемы составляет 3,51 или выше:

```
xor ebx, ebx
push ebx
push ebx
call RtlCreateQueryDebugBuffer
push eax
push 14h ;PDI_HEAPS + PDI_HEAP_BLOCKS
xchg ebx, eax
push d fs:[eax+20h] ;UniqueProcess
call RtlQueryProcessDebugInformation
mov eax, [ebx+38h] ;HeapInformation
mov eax, [eax+8] ;Flags
bswap eax
;not HEAP_SKIP_VALIDATION_CHECKS
and al, 0efh
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
;reversed by bswap
cmp eax, 62000040h
je being_debugged
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor edx, edx
xor ecx, ecx
call RtlCreateQueryDebugBuffer
push rax
pop r8
push 14h ;PDI_HEAPS + PDI_HEAP_BLOCKS
pop rdx
mov ecx, gs:[rdx+2ch] ;UniqueProcess
xchg ebx, eax
call RtlQueryProcessDebugInformation
mov eax, [rbx+70h] ;HeapInformation
;Flags
;GROWABLE
;+ TAIL_CHECKING_ENABLED
;+ FREE_CHECKING_ENABLED
;+ VALIDATE_PARAMETERS_ENABLED
cmp d [rax+10h], 40000062h
je being_debugged
```

Click to expand...

12.SwitchToThread

Функция SwitchToThread() в kernel32 (или функция ntdll NtYieldExecution()) позволяет текущему потоку предложить оставить оставшуюся часть своего временного интервала и разрешить выполнение следующего запланированного потока. Если ни один из потоков не запланирован для выполнения (или когда система занята определенными способами и не позволяет переключению происходить), то функция ntdll NtYieldExecution() возвращает состояние STATUS_NO_YIELD_PERFORMED (0x40000024), что вызывает функцию kernel32 SwitchToThread() вернуть ноль. Когда приложение отлаживается, одношаговое выполнение кода вызывает события отладки и часто приводит к невозможности выхода. Однако это безнадежно ненадежный метод обнаружения отладчика, поскольку он также обнаруживает наличие потока, который выполняется с высоким приоритетом. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push 20h
pop ebp
l1: push 0fh
call Sleep
call SwitchToThread
cmp al, 1
adc ebx, ebx
dec ebp
jne l1
inc ebx ;detect 32 non-yields
je being_debugged
```

или используя этот 64-битный код для проверки 64-битной среды Windows:

```
push 20h
pop rbp
l1: push 0fh
pop rcx
call Sleep
call SwitchToThread
cmp al, 1
adc ebx, ebx
dec ebp
jne l1
inc ebx ;detect 32 non-yields
je being_debugged
```

Click to expand...

13.Toolhelp32ReadProcessMemory

Функция kernel32 Toolhelp32ReadProcessMemory() (представленная в Windows 2000) позволяет одному процессу открывать и читать память другого процесса. Она сочетает в себе функцию kernel32 OpenProcess() с функцией kernel32 ReadProcessMemory() (или функцию kernel32 CloseHandle()). Обратите внимание, что функция в настоящее время неправильно задокументирована относительно того, как копировать данные из текущего процесса. Функция задокументирована как принятие нуля для ID процесса для чтения памяти текущего процесса, но это неверно. Идентификатор процесса всегда должен быть действительным. Для чтения из текущего процесса идентификатор процесса должен быть значением, которое возвращает функция

kernel32 GetCurrentProcessId(). Эта функция может использоваться в качестве другого способа обнаружения условия перехода, путем проверки точки останова после вызова функции. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
push eax
mov eax, esp
xor ebx, ebx
push ebx
inc ebx
push ebx
push eax
push offset l1
push d fs:[ebx+1fh] ;UniqueProcess
call Toolhelp32ReadProcessMemory
l1: pop eax
cmp al, 0cch
je being_debugged
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
xor ebp, ebp
enter 20h, 0
push 1
pop r9
push rbp
pop r8
mov rdx, offset l1
mov ecx, gs:[rbp+40h] ;UniqueProcess
call Toolhelp32ReadProcessMemory
l1: leave
cmp bpl, 0cch
je being_debugged
```

Click to expand...

14.UnhandledExceptionFilter

Когда возникает исключение и не существует зарегистрированных обработчиков исключений (ни структурированных, ни векторных), или если ни один из зарегистрированных обработчиков не обрабатывает исключение, то функция kernel32 UnhandledExceptionFilter() будет вызываться в качестве последнего средства. Если отладчик отсутствует (что определяется вызовом функции ntdll NtQueryInformationProcess() с классом ProcessDebugPort), то будет вызван обработчик, зарегистрированный функцией kernel32 SetUnhandledExceptionFilter(). Если присутствует отладчик, этот вызов не будет достигнут. Вместо этого исключение будет передано отладчику. Функция определяет наличие отладчика, вызывая функцию ntdll NtQueryInformationProcess с классом ProcessDebugPort. Отсутствующее исключение можно использовать для определения наличия отладчика. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
push offset l1
call SetUnhandledExceptionFilter
;force an exception to occur
int 3
jmp being_debugged
l1: ;execution resumes here if exception occurs
...
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows (но этот метод не работает аналогичным образом для 64-разрядных процессов):

```
mov rcx, offset l1
call SetUnhandledExceptionFilter
;force an exception to occur
int 3
jmp being_debugged
l1: ;execution resumes here if exception occurs
...
```

15.VirtualProtect

Функция kernel32 VirtualProtect() (или функция kernel32 VirtualProtectEx(), или затем функция ntdll NtProtectVirtualMemory()) могут использоваться для выделения защитных страниц. Защитные страницы - это страницы, которые вызывают исключение при первом обращении к ним. Они обычно размещаются в нижней части стека, чтобы перехватить потенциальную проблему, прежде чем она станет

неисправимой. Защитные страницы также могут быть использованы для обнаружения отладчика. Два предварительных шага - зарегистрировать обработчик исключений и выделить защитную страницу. Порядок этих шагов не важен. Как правило, страница изначально выделяется как доступная для записи и выполнения, чтобы в нее можно было поместить некоторый контент, хотя это совершенно необязательно. После заполнения страницы защита страницы изменяется, чтобы преобразовать страницу в защитную страницу. Следующий шаг - попытаться выполнить что-то с защищенной страницы . Это должно привести к тому, что обработчик исключений получит исключение EXCEPTION_GUARD_PAGE (0x80000001). Однако, если присутствует отладчик, он может перехватить исключение и позволить продолжить выполнение. Такое поведение, как известно, происходит в OllyDbg. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
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
push 140h ;PAGE_EXECUTE_READWRITE+PAGE_GUARD
push 1
push eax
xchg ebp, eax
call VirtualProtect
push offset l1
push d fs:[ebx]
mov fs:[ebx], esp
push offset being_debugged
;execution resumes at being_debugged
;if ret instruction is executed
jmp ebp
l1: ;execution resumes here if exception occurs
...
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows (хотя это применимо к отладчикам, отличным от OllyDbg, поскольку OllyDbg не работает в 64-разрядных версиях Windows):

```
push 40h ;PAGE_EXECUTE_READWRITE
pop r9
mov r8d, 1000h ;MEM_COMMIT
push 1
pop rdx
xor ecx, ecx
call VirtualAlloc
mov b [rax], 0c3h
push rax
pop rbx
enter 20h, 0
push rbp
pop r9
;PAGE_EXECUTE_READWRITE+PAGE_GUARD
mov r8d, 140h
push 1
pop rdx
xchg rcx, rax
call VirtualProtect
mov rdx, offset l1
xchg ecx, eax
call AddVectoredExceptionHandler
push offset being_debugged
;execution resumes at being_debugged
;if ret instruction is executed
jmp rbx
l1: ;execution resumes here if exception occurs
...

Click to expand...
```

E.System-level

1.FindWindow

Функция user32 FindWindow() может использоваться для поиска окон по имени или классу. Это простой способ обнаружить присутствие отладчика, если отладчик имеет графический интерфейс пользователя. Например, можно найти OllyDbg, передав OLLYDBG в качестве имени класса для поиска. WinDbg можно найти, передав WinDbgFrameClass в качестве имени класса для поиска. Наличие этих инструментов можно проверить с помощью этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
mov edi, offset l2
l1: push 0
push edi
call FindWindowA
test eax, eax
jne being_debugged
or ecx, -1
repne scasb
cmp [edi], al
jne l1
...
l2: <array of ASCIIZ strings, zero to end>
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
mov rdi, offset l2
l1: xor edx, edx
push rdi
pop rcx
call FindWindowA
test eax, eax
jne being_debugged
or ecx, -1
repne scasb
cmp [rdi], al
jne l1
...
l2: <array of ASCIIZ strings, zero to end>
```

Click to expand...

Типичный список включает в себя следующие имена:

```
db "OLLYDBG", 0
db "WinDbgFrameClass", 0 ;WinDbg
db "ID", 0 ;Immunity Debugger
db "Zeta Debugger", 0
db "Rock Debugger", 0
db "ObsidianGUI", 0
db 0
```

2.NtQueryObject

В Windows XP появился объект отладки. Когда начинается сеанс отладки, создается объект отладки, и с ним связывается дескриптор. Используя функцию ntdll NtQueryObject(), можно запросить список существующих объектов и проверить количество дескрипторов, связанных с любым существующим объектом отладки (функция может быть вызвана на любой платформе на базе Windows NT, но только в Windows XP и более поздние версии будут иметь объект отладки в списке). Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor ebx, ebx
xor ebp, ebp
jmp l2
l1: push 8000h ;MEM_RELEASE
push ebp
push esi
call VirtualFree
l2: xor eax, eax
mov ah, 10h ;MEM_COMMIT
add ebx, eax ;4kb increments
push 4 ;PAGE_READWRITE
push eax
push ebx
push ebp
call VirtualAlloc
;function does not return required length
;for this class in Windows Vista and later
push ebp
;must calculate by brute-force
push ebx
push eax
push 3 ;ObjectAllTypesInformation
push ebp
xchg esi, eax
call NtQueryObject
;presumably STATUS_INFO_LENGTH_MISMATCH
test eax, eax
jl l1
lodsd ;handle count
xchg ecx, eax
l3: lodsd ;string lengths
movzx edx, ax ;length
lodsd ;pointer to TypeName
xchg esi, eax
;sizeof(L"DebugObject")
;avoids superstrings
;like "DebugObjective"
cmp edx, 16h
jne l4
xchg ecx, edx
mov edi, offset l5
```

```
repe cmpsb
xchg ecx, edx
jne l4
;checking TotalNumberOfHandles
;works only on Windows XP
;cmp [eax], edx ;TotalNumberOfHandles
;check TotalNumberOfObjects instead
cmp [eax+4], edx ;TotalNumberOfObjects
jne being_debugged
l4: lea esi, [esi+edx+4] ;skip null and align
and esi, -4 ;round down to dword
loop l3
...
l5: dw "D","e","b","u","g"
dw "O","b","j","e","c","t"
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor ebx, ebx
xor ebp, ebp
jmp l2
l1: mov r8d, 8000h ;MEM_RELEASE
xor edx, edx
mov ecx, esi
call VirtualFree
l2: xor eax, eax
mov ah, 10h ;MEM_COMMIT
add ebx, eax ;4kb increments
push 4 ;PAGE_READWRITE
pop r9
push rax
pop r8
mov edx, ebx
xor ecx, ecx
call VirtualAlloc
;function does not return required length
;for this class in Windows Vista and later
enter 20h, 0
;must calculate by brute-force
push rbx
pop r9
push rax
pop r8
push 3 ;ObjectAllTypesInformation
pop rdx
xor ecx, ecx
xchg esi, eax
call NtQueryObject
leave
;presumably STATUS_INFO_LENGTH_MISMATCH
test eax, eax
jl l1
lodsq ;handle count
xchg ecx, eax
l3: lodsq ;string lengths
movzx edx, ax ;length
lodsq ;pointer to TypeName
xchg esi, eax
;sizeof(L"DebugObject")
```

```
;avoids superstrings
;like "DebugObjective"
cmp edx, 16h
jne l4
xchg ecx, edx
mov rdi, offset l5
repe cmpsb
xchg ecx, edx
jne l4
;checking TotalNumberOfHandles
;works only on Windows XP
;cmp [rax], edx ;TotalNumberOfHandles
;check TotalNumberOfObjects instead
cmp [rax+4], edx ;TotalNumberOfObjects
jne being_debugged
l4: lea esi, [rsi+rdx+8] ;skip null and align
and esi, -8 ;round down to dword
loop l3
...
l5: dw "D","e","b","u","g"
dw "O","b","j","e","c","t"
```

Click to expand...

Поскольку эта информация поступает из ядра, для кода пользовательского режима нет простого способа предотвратить обнаружение отладчиком этого вызова.

3.NtQuerySystemInformation

a. SystemKernelDebuggerInformation

Функция ntdll NtQuerySystemInformation() принимает параметр, который является классом информации для запроса. Большинство классов не документированы. Это включает в себя класс SystemKernelDebuggerInformation (0x23), который существует со времен Windows NT. Класс SystemKernelDebuggerInformation возвращает значение двух флагов: KdDebuggerEnabled в al и KdDebuggerNotPresent в ah. Таким образом, возвращаемое значение в ax равно нулю, если присутствует отладчик. Вызов может быть выполнен с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push eax
mov eax, esp
push 0
push 2 ;SystemInformationLength
push eax
push 23h ;SystemKernelDebuggerInformation
call NtQuerySystemInformation
pop eax
test ah, ah
je being_debugged
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
push rax
mov edx, esp
xor r9, r9
push 2 ;SystemInformationLength
pop r8
push 23h ;SystemKernelDebuggerInformation
pop rcx
call NtQuerySystemInformation
pop rax
test ah, ah
je being_debugged
```

Поскольку эта информация поступает из ядра, не существует простого способа предотвратить этот отладчик для обнаружения присутствия отладчика. Функция записывает только два байта, независимо от размера ввода. Этот небольшой размер необычен и может показать наличие некоторых скрывающих инструментов, потому что такие инструменты обычно записывают четыре байта в место назначения.

Вызов функции может быть дополнительно запутан путем простого извлечения значения непосредственно из поля KdDebuggerEnabled в структуре KUSER_SHARED_DATA со смещением 0x7ffe02d4 для конфигураций пространства пользователя 2 ГБ. Это значение доступно во всех 32-разрядных и 64-разрядных версиях Windows. Интересно, что значение, возвращаемое вызовом функции, происходит из отдельного места, поэтому любой инструмент, который хочет скрыть отладчик, должен будет исправить значение в обоих местах.

b. SystemProcessInformation

Идентификатор процесса как Explorer.exe и родительского процесса текущего процесса, так и имя этого родительского процесса можно получить с помощью функции ntdll NtQuerySystemInformation() с классом SystemProcessInformation (5). Один вызов функции возвращает весь список запущенных процессов, который затем необходимо проанализировать вручную. Это функция, которую внутренне вызывает функция kernel32 CreateToolhelp32Snapshot(). Как и в случае с алгоритмом функции CreateToolhelp32Snapshot() следует проверять имя пользователя и имя домена, чтобы избежать ложного совпадения и потенциально сократить количество проходов подпрограммы. Вызов может быть выполнен с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor ebp, ebp
xor esi, esi
xor edi, edi
jmp l2
l1: push 8000h ;MEM_RELEASE
push ebp
push ebx
call VirtualFree
l2: xor eax, eax
mov ah, 10h ;MEM_COMMIT
add esi, eax ;4kb increments
push 4 ;PAGE_READWRITE
push eax
push esi
push ebp
call VirtualAlloc
;function does not return
;required length for this class
push ebp
;must calculate by brute-force
push esi
push eax
push 5 ;SystemProcessInformation
xchg ebx, eax
call NtQuerySystemInformation
;presumably STATUS_INFO_LENGTH_MISMATCH
test eax, eax
jl l1
push ebx
push ebx
l3: push ebx
mov eax, fs:[20h] ;UniqueProcess
cmp [ebx+44h], eax ;UniqueProcessId
;InheritedFromUniqueProcessId
cmove edi, [ebx+48h]
test edi, edi
je l4
cmp ebp, edi
je l11
l4: mov ecx, [ebx+3ch] ;ImageName
jecxz l9
```

```
xor eax, eax
mov esi, ecx
l5: lodsw
cmp eax, "\"
cmove ecx, esi
push ecx
push eax
call CharLowerW
mov [esi-2], ax
pop ecx
test eax, eax
jne l5
sub esi, ecx
xchg ecx, esi
push edi
mov edi, offset l17
repe cmpsb
pop edi
jne l9
mov eax, [ebx+44h] ;UniqueProcessId
push ebx
push ebp
push edi
call l12
dec ecx ;invert Z flag
jne l7
push ebx
push edi
dec ecx
call l13
pop esi
pop edx
mov cl, 2
;compare user names
;then domain names
l6: lodsb
scasb
jne l7
test al, al
jne l6
mov esi, ebx
```

```
mov edi, edx
loop l6
l7: pop edi
pop ebp
pop ebx
jne l9
test ebp, ebp
je l8
mov esi, offset l18
cmp cl, [esi]
adc [esi], ecx
l8: mov ebp, [ebx+44h] ;UniqueProcessId
l9: pop ebx
mov ecx, [ebx] ;NextEntryOffset
add ebx, ecx
inc ecx
loop l10
pop ebx
dec b [offset l18+1]
l10:jne l3
;and possibly one pointer left on stack
;add esp, -b [offset l18]*4
jmp being_debugged
;and at least one pointer left on stack
;add esp, (b [offset l18+1]-b [offset l18]+1)*4
l11:...
l12:push eax
push 0
push 400h ;PROCESS_QUERY_INFORMATION
call OpenProcess
xchg ecx, eax
jecxz l16
l13:push eax
push esp
push 8 ;TOKEN_QUERY
push ecx
call OpenProcessToken
pop ebx
xor ebp, ebp
l14:push ebp
push 0 ;GMEM_FIXED
```

```
call GlobalAlloc
push eax
push esp
push ebp
push eax
push 1 ;TokenUser
push ebx
xchg esi, eax
call GetTokenInformation
pop ebp
xchg ecx, eax
jecxz l14
xor ebp, ebp
l15:push ebp
push 0 ;GMEM_FIXED
call GlobalAlloc
xchg ebx, eax
push ebp
push 0 ;GMEM_FIXED
call GlobalAlloc
xchg edi, eax
push eax
mov eax, esp
push ebp
mov ecx, esp
push ebp
mov edx, esp
push eax
push ecx
push ebx
push edx
push edi
push d [esi]
push 0
call LookupAccountSidA
pop ebp
pop ecx
xchg ebp, ecx
pop edx
xchg ecx, eax
jecxz l15
```

> l16:ret
> l17:dw "e", "x", "p", "l", "o", "r", "e", "r"
> dw ".", "e", "x", "e", 0
> l18:db 0ffh, 1, ?, ?
>
> Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor ebp, ebp
xor esi, esi
xor edi, edi
jmp l2
l1: mov r8d, 8000h ;MEM_RELEASE
xor edx, edx
mov ecx, ebx
call VirtualFree
l2: xor eax, eax
mov ah, 10h ;MEM_COMMIT
add esi, eax ;4kb increments
push 4 ;PAGE_READWRITE
pop r9
push rax
pop r8
mov edx, esi
xor ecx, ecx
call VirtualAlloc
;function does not return
;required length for this class
xor r9d, r9d
;must calculate by brute-force
push rsi
pop r8
mov edx, eax
push 5 ;SystemProcessInformation
pop rcx
xchg ebx, eax
call NtQuerySystemInformation
;presumably STATUS_INFO_LENGTH_MISMATCH
test eax, eax
jl l1
push rbx
push rbx
l3: push rbx
call GetCurrentProcessId
cmp [rbx+50h], eax ;UniqueProcessId
;InheritedFromUniqueProcessId
cmove edi, [rbx+58h]
test edi, edi
je l4
```

```
cmp ebp, edi
je l11
l4: mov ecx, [rbx+40h] ;ImageName
jrcxz l9
xor eax, eax
mov esi, ecx
l5: lodsw
cmp eax, "\"
cmove ecx, esi
push rcx
xchg ecx, eax
enter 20h, 0
call CharLowerW
leave
mov [rsi-2], ax
pop rcx
test eax, eax
jne l5
sub esi, ecx
xchg ecx, esi
push rdi
mov rdi, offset l17
repe cmpsb
pop rdi
jne l9
mov r8d, [rbx+50h] ;UniqueProcessId
push rbx
push rbp
push rdi
call l12
dec ecx ;invert Z flag
jne l7
push rbx
push rdi
dec rcx
call l13
pop rsi
pop rdx
inc ecx
;compare user names
;then domain names
```

```
l6: lodsb
scasb
jne l7
test al, al
jne l6
mov esi, ebx
mov edi, edx
loop l6
l7: pop rdi
pop rbp
pop rbx
jne l9
test ebp, ebp
je l8
mov rsi, offset l18
cmp cl, [rsi]
adc [rsi], ecx
l8: mov ebp, [rbx+50h] ;UniqueProcessId
l9: pop rbx
mov ecx, [rbx] ;NextEntryOffset
add ebx, ecx
inc ecx
loop l10
pop rbx
dec byte [offset l18+1]
l10:jne l3
;and possibly one pointer left on stack
;add esp, -b [offset l18]*4
jmp being_debugged
;and at least one pointer left on stack
;add esp, (b [offset l18+1]-b [offset l18]+1)*4
l11:...
l12:cdq
xor ecx, ecx
mov ch, 4 ;PROCESS_QUERY_INFORMATION
enter 20h, 0
call OpenProcess
leave
xchg ecx, eax
jrcxz l16
l13:push rax
```

```
mov r8d, esp
push 8 ;TOKEN_QUERY
pop rdx
call OpenProcessToken
pop rbx
xor ebp, ebp
l14:mov edx, ebp
xor ecx, ecx ;GMEM_FIXED
enter 20h, 0
call GlobalAlloc
leave
push rbp
pop r9
push rax
pop r8
push rax ;simulate enter
mov ebp, esp
push rbp
sub esp, 20h
push 1 ;TokenUser
pop rdx
mov ecx, ebx
xchg esi, eax
call GetTokenInformation
leave
xchg ecx, eax
jrcxz l14
xor ebp, ebp
l15:mov ebx, ebp
mov edx, ebp
xor ecx, ecx ;GMEM_FIXED
enter 20h, 0
call GlobalAlloc
xchg ebx, eax
xchg edx, eax
xor ecx, ecx ;GMEM_FIXED
call GlobalAlloc
leave
xchg edi, eax
push rbp
mov ecx, esp
```

```
push rbp
mov r9d, esp
push rax
push rsp
push rcx
push rbx
sub esp, 20h
push rdi
pop r8
mov edx, [rsi]
xor ecx, ecx
call LookupAccountSidA
add esp, 40h
pop rcx
pop rbp
xchg ecx, eax
jrcxz l15
l16:ret
l17:dw "e","x","p","l","o","r","e","r"
dw ".","e","x","e",0
l18:db 0ffh, 1, ?, ?
```

Click to expand...

4.Селекторы

Значения селектора могут показаться стабильными, но на самом деле они нестабильны в определенных обстоятельствах, а также в зависимости от версии Windows. Например, значение селектора может быть установлено в потоке, но оно может не содержать это значение очень долго. Некоторые события могут привести к тому, что значение селектора вернется к значению по умолчанию. Одним из таких событий является исключение. В контексте отладчика одношаговое исключение по-прежнему является исключением, которое может вызвать непредвиденное поведение. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor eax, eax
push fs
pop ds
l1: xchg [eax], cl
xchg [eax], cl
```

Пример 64-битного кода отсутствует, поскольку селектор DS не поддерживается в этой среде.

В 64-разрядных версиях Windows однократный переход по этому коду вызовет исключение нарушения прав доступа в l1, поскольку селектор DS будет восстановлен в значение по умолчанию даже до достижения l1. В 32-разрядных версиях Windows селектор DS не будет восстанавливать свое значение, если только не возникнет исключение без отладки. Различия в поведении в зависимости от версии расширяются еще больше, если используется селектор SS. В 64-разрядных версиях Windows селектор SS будет восстановлен до значения по умолчанию, как в случае селектора DS. Однако в 32-разрядных версиях Windows значение селектора SS не будет восстановлено, даже если произойдет исключение. Таким образом, если мы изменим код, чтобы он выглядел так:

```
xor eax, eax
push offset l2
push d fs:[eax]
mov fs:[eax], esp
push fs
pop ss
xchg [eax], cl
xchg [eax], cl
l1: int 3 ;force exception to occur
l2: ;looks like it would be reached
;if an exception occurs
...

Click to expand...
```

затем, когда инструкция int 3 достигается в l1 и возникает исключение точки останова, обработчик исключения в l2 не вызывается, как ожидалось. Вместо этого процесс просто прекращается.

Вариант этого метода обнаруживает одношаговое событие, просто проверяя, было ли назначение успешным. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push 3
pop gs
mov ax, gs
cmp al, 3
jne being_debugged
```

Селекторы FS и GS являются особыми случаями. Для определенных значений на них будет влиять одношаговое событие даже в 32-разрядных версиях Windows. Однако в случае селектора FS (и, технически, селектора GS) он не будет восстановлен до значения по умолчанию в 32-разрядных версиях Windows, если для него было установлено значение от нуля до трех. Вместо этого он будет установлен на ноль (на селектор GS влияет то же самое, но значение по умолчанию для селектора GS равно нулю). В 64-разрядных версиях Windows он (они) будет восстановлен до значения (по умолчанию).

Этот код также уязвим к состоянию гонки, вызванному событием переключения потоков. Когда происходит событие переключения потока, оно ведет себя как исключение и приведет к изменению значений селектора, что в случае селектора FS означает, что оно будет установлено на ноль.

Разновидность этого метода решает эту проблему путем преднамеренного ожидания события переключения потока. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push 3
pop gs
l1: mov ax, gs
cmp al, 3
je l1
```

Однако этот код уязвим для проблемы, которую он пытался обнаружить в первую очередь, потому что он не проверяет, было ли исходное назначение успешным. Конечно, два фрагмента кода можно объединить для получения желаемого эффекта, ожидая, пока не произойдет событие переключения потока, а затем выполнив присваивание в пределах интервала времени, который должен существовать, пока не

произойдет следующий. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push 3
pop gs
l1: mov ax, gs
cmp al, 3
je l1
push 3
pop gs
mov ax, gs
cmp al, 3
jne being_debugged
```

F.User-interface

1.BlockInput

Функция user32 BlockInput() может блокировать или разблокировать все события мыши и клавиатуры (кроме последовательности клавиш ctrl-alt-delete). Эффект сохраняется до тех пор, пока процесс не завершится или функция не будет вызвана снова с противоположным параметром. Это очень эффективный способ отключить отладчики. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
push 1
call BlockInput
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
xor ecx, ecx
inc ecx
call BlockInput
```

Вызов требует, чтобы вызывающий поток имел привилегию DESKTOP_JOURNALPLAYBACK (0x0020) (которая установлена по умолчанию). В Windows Vista и более поздних версиях также требуется, чтобы процесс выполнялся с высоким уровнем целостности (то есть процесс требует повышения прав, если он

выполнялся с использованием стандартной учетной записи или учетной записи пользователя с низким уровнем прав), если повышение прав включено и либо " Параметр реестра HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\System\EnableUIPI "либо не существует (в этом случае используется значение по умолчанию "присутствует"и "установлен"), либо имеет ненулевое значение (и для этого требуются права администратора)

Функция не позволяет блокировать ввод дважды подряд, а также не позволяет разблокировать ввод дважды подряд. Таким образом, если один и тот же запрос сделан дважды для функции, то возвращаемое значение должно быть другим. Этот факт может использоваться для обнаружения присутствия ряда инструментов, которые перехватывают вызов, потому что большинство из них просто возвращают успех, независимо от ввода. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push 1
call BlockInput
xchg ebx, eax
push 1
call BlockInput
xor ebx, eax
je being_debugged
```

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor ecx, ecx
inc ecx
call BlockInput
xchg ebx, eax
xor ecx, ecx
inc ecx
call BlockInput
xor ebx, eax
```

2.FLD

В анализаторе команд с плавающей запятой в OllyDbg есть проблема, поскольку OllyDbg не отключает ошибки во время операций с плавающей запятой. Это позволяет двум значениям вызывать ошибки с плавающей точкой (и, следовательно, делать сбой

OllyDbg) при преобразовании из двойной расширенной точности в целое число. Проблемный код находится в функции __fuistq():

```
mov eax, [esp+04]
mov edx, [esp+08]
...
fld t [edx]
fistp q [eax]
wait
retn
```

Два значения +/- 9,223372036854775807e18. Проблема может быть продемонстрирована с помощью этих 32-битных кодов в 32-битной или 64-битной версии Windows:

```
fld t [offset l1]
...
l1: dq -1
dw 403dh

and

fld t [offset l1]
...
l1: dq -1
dw 0c03dh
```

Есть несколько способов, с помощью которых люди пытались решить проблему, например, полностью пропустить операцию или использовать альтернативную инструкцию (которая существует только на современном процессоре, тем самым подвергая OllyDbg другому сбою, если инструкция не поддерживается), все из которых в разной степени неверны. Правильное решение - просто изменить маску исключений с плавающей точкой, чтобы игнорировать такие ошибки. Это может быть достигнуто путем загрузки значения 0x1333 (из текущего значения 0x1332) в управляющее слово FPU.

3.NtSetInformationThread

В Windows 2000 появилось расширение функций, которое, на первый взгляд, может показаться существующим только в целях устранения неполадок. Это член ThreadHideFromDebugger (0x11) класса ThreadInformationClass. Его можно установить

для каждого потока, вызвав функцию ntdll NtSetInformationThread(). Он предназначен для использования внешним процессом, но любой поток может использовать его сам по себе. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

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

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
xor r9d, r9d
xor r8d, r8d
push 11h ;ThreadHideFromDebugger
pop rdx
push -2 ;GetCurrentThread()
pop rcx
call NtSetInformationThread
```

Когда вызывается функция, поток продолжает работать, но отладчик больше не получает никаких событий, связанных с этим потоком. Среди пропущенных событий - завершение процесса, если основной поток является скрытым. Причина существования этой функции заключается в том, чтобы избежать непредвиденного прерывания, когда внешний процесс использует функцию ntdll RtlQueryProcessDebugInformation() для запроса информации об отладчике. Функция ntdll RtlQueryProcessDebugInformation() внедряет поток в отладчик для сбора информации о процессе. Если внедренный поток не скрыт от отладчика, тогда отладчик получит управление при запуске потока, и отладчик прекратит выполнение.

4.SuspendThread

Функция kernel32 SuspendThread() (или функция ntdll NtSuspendThread()) может быть еще одним очень эффективным способом отключения отладчиков пользовательского режима. Это может быть достигнуто путем перечисления потоков данного процесса или поиска именованного окна и потом открыв ветку владельца, а затем приостановив ее. Вызов может быть выполнен с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows (на основе более раннего кода, который обнаружил родительский процесс и поместил идентификатор процесса в регистр EDI):

```
push edi
push 4 ;TH32CS_SNAPTHREAD
call CreateToolhelp32Snapshot
push 1ch ;sizeof(THREADENTRY32)
push esp
push eax
xchg ebx, eax
call Thread32First
l1: push esp
push ebx
call Thread32Next
cmp [esp+0ch], edi ;th32OwnerProcessID
jne l1
push d [esp+8] ;th32ThreadID
push 0
push 2 ;THREAD_SUSPEND_RESUME
call OpenThread
push eax
call SuspendThread
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows (на основе более раннего кода, который обнаружил родительский процесс и поместил идентификатор процесса в регистр EDI):

```
mov edx, edi
push 4 ;TH32CS_SNAPTHREAD
pop rcx
call CreateToolhelp32Snapshot
mov ebx, eax
push 1ch ;sizeof(THREADENTRY32)
pop rbp
enter 20h, 0
mov edx, ebp
xchg ecx, eax
call Thread32First
l1: mov edx, ebp
mov ecx, ebx
call Thread32Next
cmp [rbp+0ch], edi ;th32OwnerProcessID
jne l1
mov r8, [rbp+8] ;th32ThreadID
cdq
push 2 ;THREAD_SUSPEND_RESUME
pop rcx
call OpenThread
xchg ecx, eax
call SuspendThread
```

Click to expand...

5.SwitchDesktop

Платформы на базе Windows NT поддерживают несколько рабочих столов за сеанс. Можно выбрать другой активный рабочий стол, который будет скрывать окна ранее активного рабочего стола, и без очевидного способа переключиться обратно на старый рабочий стол (последовательность клавиш ctrl-alt-delete этого не сделает) , Кроме того, события мыши и клавиатуры с рабочего стола отладчика больше не будут доставляться отладчику, поскольку их источник больше не передается. Очевидно, что отладка невозможна. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
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
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
xor edx, edx
push rdx
;DESKTOP_CREATEWINDOW
;+ DESKTOP_WRITEOBJECTS
;+ DESKTOP_SWITCHDESKTOP
push 182h
sub esp, 20h
xor r9d, r9d
xor r8d, r8d
mov rcx, offset l1
call CreateDesktopA
xchg ecx, eax
call SwitchDesktop
...
l1: db "mydesktop", 0
```

Click to expand...

G.Uncontrolled execution

1.CreateProcess

Один из самых простых способов избежать контроля отладчика - заставить процесс выполнить еще одну копию. Как правило, процесс будет использовать объект синхронизации, такой как мьютекс, чтобы предотвратить повторение бесконечно. Первый процесс создаст мьютекс, а затем выполнит копирование процесса. Второй процесс не будет находиться под контролем отладчика, даже если первый процесс был. Второй процесс также будет знать, что это копия, поскольку мьютекс будет существовать. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
xor ebx, ebx
push offset l2
push ebx
push ebx
call CreateMutexA
call GetLastError
cmp eax, 0b7h ;ERROR_ALREADY_EXISTS
je l1
mov ebp, offset l3
push ebp
call GetStartupInfoA
call GetCommandLineA
sub esp, 10h ;sizeof(PROCESS_INFORMATION)
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 "mymutex", 0
l3: db 44h dup (?) ;sizeof(STARTUPINFO)
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
mov r8, offset l2
xor edx, edx
xor ecx, ecx
call CreateMutexA
call GetLastError
cmp eax, 0b7h ;ERROR_ALREADY_EXISTS
je l1
mov rbp, offset l3
push rbp
pop rcx
call GetStartupInfoA
call GetCommandLineA
mov rsi, offset l4
push rsi
push rbp
xor ecx, ecx
push rcx
push rcx
push rcx
push rcx
sub esp, 20h
xor r9d, r9d
xor r8d, r8d
xchg edx, eax
call CreateProcessA
or rdx, -1 ;INFINITE
mov ecx, [rsi]
call WaitForSingleObject
call ExitProcess
l1: ...
l2: db "mymutex", 0
l3: db 68h dup (?) ;sizeof(STARTUPINFO)
l4: db 18h dup (?) ;sizeof(PROCESS_INFORMATION)
```

Click to expand...

Довольно часто можно увидеть использование функции ядра Sleep() вместо функции kernel32 WaitForSingleObject(), но это приводит к состоянию гонки. Проблема возникает, когда во время выполнения происходит интенсивная загрузка процессора. Это может быть из-за достаточно сложной защиты (или преднамеренных задержек) во

втором процессе; а также действия, которые пользователь может выполнять во время выполнения, такие как просмотр сети или извлечение файлов из архива. В результате второй процесс может не достичь проверки мьютекса до истечения задержки; заставляя его думать, что это первый процесс. Если это произойдет, то процесс выполнит еще одну свою копию. Такое поведение может повторяться до тех пор, пока один из процессов не завершит проверку мьютекса успешно.

Также обратите внимание, что первый процесс должен ждать, пока второй процесс не завершится (ожиданием дескриптора процесса) или, по крайней мере, сигнализирует о его успешном запуске (например, ожиданием дескриптора события вместо использования мьютекса). В противном случае первый процесс может завершиться до того, как второй процесс выполнит проверку состояния, а затем второй процесс подумает, что это первый процесс, и цикл повторится.

Расширение метода самостоятельного выполнения - это отладка. Самостоятельная отладка, как следует из названия, - это запуск копии самого себя, а затем присоединение к ней в качестве отладчика. Это не означает отладку одного процесса, потому что это невозможно. Поскольку к процессу одновременно может быть подключен только один отладчик, второй процесс становится "не подлежащим отладке" обычными средствами. Первому процессу даже не нужно делать что-либо, связанное с отладчиком, хотя он, безусловно, может сделать это. В простейшем случае первый процесс может просто игнорировать любые события, связанные с отладчиком (например, загрузка DLL), за исключением события завершения процесса. В более сложном случае второй процесс может содержать типичные приемы антиотладки, такие как жестко запрограммированные точки останова, но теперь они могут иметь особое значение для первого процесса, что делает очень трудным моделирование среды отладки, используя только один процесс. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
xor ebx, ebx
push offset l4
push ebx
push ebx
call CreateMutexA
call GetLastError
cmp eax, 0b7h ;ERROR_ALREADY_EXISTS
je l3
mov ebp, offset l5
push ebp
call GetStartupInfoA
call GetCommandLineA
mov esi, offset l6
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 l7
jmp l2
l1: push 10002h ;DBG_CONTINUE
push d [esi+0ch] ;dwThreadId
push d [esi+8] ;dwProcessId
call ContinueDebugEvent
l2: push -1 ;INFINITE
push ebx
call WaitForDebugEvent
cmp b [ebx], 5 ;EXIT_PROCESS_DEBUG_EVENT
jne l1
call ExitProcess
l3: ;execution resumes here in second process
...
l4: db "mymutex", 0
```

> l5: db 44h dup (?) ;sizeof(STARTUPINFO)
> l6: db 10h dup (?) ;sizeof(PROCESS_INFORMATION)
> l7: db 60h dup (?) ;sizeof(DEBUG_EVENT)
>
> Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
mov r8, offset l4
xor edx, edx
xor ecx, ecx
call CreateMutexA
call GetLastError
cmp eax, 0b7h ;ERROR_ALREADY_EXISTS
je l3
mov rbp, offset l5
push rbp
pop rcx
call GetStartupInfoA
call GetCommandLineA
mov rsi, offset l6
push rsi
push rbp
xor ecx, ecx
push rcx
push rcx
push 1 ;DEBUG_PROCESS
push rcx
sub esp, 20h
xor r9d, r9d
xor r8d, r8d
xchg edx, eax
call CreateProcessA
mov rbx, offset l7
jmp l2
l1: mov r8d, 10002h ;DBG_CONTINUE
mov edx, [rsi+14h] ;dwThreadId
mov ecx, [rsi+10h] ;dwProcessId
call ContinueDebugEvent
l2: or rdx, -1 ;INFINITE
push rbx
pop rcx
call WaitForDebugEvent
cmp b [rbx], 5 ;EXIT_PROCESS_DEBUG_EVENT
jne l1
call ExitProcess
l3: ;execution resumes here in second process
...
l4: db "mymutex", 0
```

> l5: db 68h dup (?) ;sizeof(STARTUPINFO)
> l6: db 18h dup (?) ;sizeof(PROCESS_INFORMATION)
> l7: db 0ach dup (?) ;sizeof(DEBUG_EVENT)
>
> Click to expand...

2.CreateThread

Потоки - простой способ для отладчика передать управление в область памяти, где выполнение может свободно возобновиться, если точка останова не помещена в соответствующее место. Они также могут использоваться, чтобы мешать выполнению других потоков (включая основной поток), и таким образом мешать отладке. Одним из примеров является периодическая проверка программных точек останова или других изменений памяти, которые отладчик может вызвать в потоке кода. Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
xor eax, eax
push eax
push esp
push eax
push eax
push offset l2
push eax
push eax
call CreateThread
l1: ;here could be obfuscated code
;with lots of dummy function calls
;that would invite step-over
;and thus breakpoint insertion
...
l2: xor eax, eax
cdq
mov ecx, offset l2 - offset l1
mov esi, offset l1
l3: lodsb
add edx, eax ;simple sum to detect breakpoints
loop l3
cmp edx, <checksum>
jne being_debugged
mov ch, 1;small delay then restart
push ecx
call Sleep
jmp l2
```

Click to expand...

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor edx, edx
push rdx
push rsp
push rdx
sub esp, 20h
xor r9d, r9d
mov r8, offset l2
xor ecx, ecx
call CreateThread
l1: ;here could be obfuscated code
;with lots of dummy function calls
;that would invite step-over
;and thus breakpoint insertion
...
l2: xor eax, eax
cdq
mov ecx, offset l2 - offset l1
mov rsi, offset l1
l3: lodsb
add edx, eax ;simple sum to detect breakpoints
loop l3
cmp edx, <checksum>
jne being_debugged
mov ch, 1 ;small delay then restart
call Sleep
jmp l2
```

Click to expand...

3.DebugActiveProcess

Функция kernel32 DebugActiveProcess() (или функция ntdll DbgUiDebugActiveProcess() или функция ntdll NtDebugActiveProcess()) могут использоваться для подключения в качестве отладчика к уже запущенному процессу. Поскольку к процессу одновременно может быть подключен только один отладчик, сбой подключения к процессу может указывать на присутствие другого отладчика (хотя могут быть и другие причины сбоя, например ограничения дескриптора безопасности). Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
mov ebp, offset l4
push ebp
call GetStartupInfoA
xor ebx, ebx
mov esi, offset l5
push esi
push ebp
push ebx
push ebx
push ebx
push ebx
push ebx
push ebx
push offset l3
push ebx
call CreateProcessA
push d [esi+8] ;dwProcessId
call DebugActiveProcess
test eax, eax
je being_debugged
mov ebx, offset l6
jmp l2
l1: push 10002h ;DBG_CONTINUE
push d [esi+0ch] ;dwThreadId
push d [esi+8] ;dwProcessId
call ContinueDebugEvent
l2: push -1 ;INFINITE
push ebx
call WaitForDebugEvent
cmp b [ebx], 5 ;EXIT_PROCESS_DEBUG_EVENT
jne l1
call ExitProcess
l3: db "myfile", 0
l4: db 44h dup (?) ;sizeof(STARTUPINFO)
l5: db 10h dup (?) ;sizeof(PROCESS_INFORMATION)
l6: db 60h dup (?) ;sizeof(DEBUG_EVENT)
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
mov rbp, offset l4
push rbp
pop rcx
call GetStartupInfoA
mov rsi, offset l5
push rsi
push rbp
xor ecx, ecx
push rcx
push rcx
push rcx
push rcx
sub esp, 20h
xor r9d, r9d
xor r8d, r8d
mov rdx, offset l3
call CreateProcessA
mov ecx, [rsi+10h] ;dwProcessId
call DebugActiveProcess
test eax, eax
je being_debugged
mov rbx, offset l6
jmp l2
l1: mov r8d, 10002h ;DBG_CONTINUE
mov edx, [rsi+14h] ;dwThreadId
mov ecx, [rsi+10h] ;dwProcessId
call ContinueDebugEvent
l2: or rdx, -1 ;INFINITE
push rbx
pop rcx
call WaitForDebugEvent
cmp b [rbx], 5 ;EXIT_PROCESS_DEBUG_EVENT
jne l1
call ExitProcess
l3: db "myfile", 0
l4: db 68h dup (?) ;sizeof(STARTUPINFO)
l5: db 18h dup (?) ;sizeof(PROCESS_INFORMATION)
l6: db 0ach dup (?) ;sizeof(DEBUG_EVENT)
```

Click to expand...

4. Перечисления

Есть много функций перечисления, и некоторые из них находятся в DLL, отличных от kernel32.dll. Любой из них может использоваться для передачи управления в область памяти, где выполнение может свободно возобновляться, если точка останова не размещена в соответствующем месте. Вызов может быть выполнен с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows в 32-разрядной или 64-разрядной версиях Windows:

```
push 0
push 0
push offset l1
call EnumDateFormatsA
jmp being_debugged
l1: ;execution resumes here during enumeration
...
```

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
xor r8d, r8d
xor edx, edx
mov rcx, offset l1
call EnumDateFormatsA
jmp being_debugged
l1: ;execution resumes here during enumeration
...
```

5.GenerateConsoleCtrlEvent

Когда пользователь нажимает комбинацию клавиш Ctrl+C или Ctrl+Break, когда окно консоли находится в фокусе, Windows проверяет, должно ли событие обрабатываться или игнорироваться. Если событие должно быть обработано, то Windows вызывает зарегистрированный обработчик управления консоли или функцию kernel32 CtrlRoutine(). Функция kernel32 CtrlRoutine() проверяет наличие отладчика (который определяется чтением флага BeingDebugged в блоке среды процесса), а затем выдает исключение DBG_CONTROL_C (0x40010005) или исключение DBG_CONTROL_BREAK (0x40010008), если оно есть. Это исключение может быть перехвачено обработчиком исключений или обработчиком событий, но вместо этого исключение может использоваться отладчиком. В результате отсутствующее исключение можно использовать для определения наличия отладчика. Приложение

может зарегистрировать обработчик структурированных исключений или зарегистрировать обработчик событий, вызвав функцию kernel32 SetConsoleCtrlHandler(). Исключение можно вызвать, вызвав функцию kernel32 GenerateConsoleCtrlEvent(). Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
xor eax, eax
push offset l1
push d fs:[eax]
mov fs:[eax], esp
;Process Environment Block
mov ecx, fs:[eax+30h]
inc b [ecx+2]
push eax
push eax ;CTRL_C_EVENT
call GenerateConsoleCtrlEvent
jmp $
l1: ;execution resumes here if exception occurs
...
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
mov rdx, offset l1
xor ecx, ecx
inc ecx
call AddVectoredExceptionHandler
push 60h
pop rsi
gs:lodsq ;Process Environment Block
inc b [eax+2]
push 0
push 0 ;CTRL_C_EVENT
call GenerateConsoleCtrlEvent
jmp $
l1: ;execution resumes here if exception occurs
...
```

Click to expand...

6.NtSetInformationProcess

Возможно, из-за того, что значение регистра локальной таблицы дескрипторов (LDT) равно нулю в пользовательском режиме в физической (в отличие от виртуальной) среде Windows, он обычно не поддерживается должным образом (или вообще не поддерживается) отладчиками. В результате его можно использовать как простую технику антиотладки. В частности, может быть создана новая запись в таблице локальных дескрипторов, которая отображается на некоторый код. Это можно сделать для каждого процесса, вызвав функцию ntdll NtSetInformationProcess() и передав ProcessLdtInformation(0x0a) член класса ProcessInformationClass. Затем передача управляющей команды (call, jump, ret и так далее) В новую запись таблицы локальных дескрипторов может привести к тому, что отладчик запутается в новой области памяти, или даже откажется отлаживать дальше. Вызов может быть выполнен с использованием этого 32-разрядного кода для проверки 32-разрядных версий Windows (вызов не поддерживается в 64-разрядных версиях Windows):

```
;base must be <= PE->ImageBase
;but no need for 64kb align
base equ 12345678h
;sel must have bit 2 set
;CPU will set bits 0 and 1
;even if we don't do it
sel equ 777h

;4k granular, 32-bit, present
;DPL3, exec-only code
;limit must not touch kernel mem
;calculate carefully to use functions
push (base and 0ff000000h) \
+ 0c1f800h \
+ ((base shr 10h) and 0ffh)
push (base shl 10h) + 0ffffh
push 8
push sel and -8 ;bits 0-2 must be clear here
mov eax, esp
push 10h
push eax
push 0ah ;ProcessLdtInformation
push -1 ;GetCurrentProcess()
call NtSetInformationProcess
;jmp far sel:l1
db 0eah
dd offset l1 – base
dw sel
l1: ;execution continues here but using LDT selector
...
Click to expand...
```

7.NtSetLdtEntries

Функция ntdll NtSetLdtEntries() также позволяет устанавливать значения таблицы локальных дескрипторов напрямую, но только для текущего процесса. Это совершенно отдельная функция с немного другим форматом параметров по сравнению с методом ProcessLdtInformation выше, но результат тот же. Вызов может быть выполнен с использованием этого 32-разрядного кода для проверки 32-разрядных версий Windows (вызов не поддерживается в 64-разрядных версиях Windows):

```
;base must be <= PE->ImageBase
;but no need for 64kb align
base equ 12345678h
;sel must have bit 2 set
;CPU will set bits 0 and 1
;even if we don't do it
sel equ 777h
xor eax, eax
push eax
push eax
push eax
;4k granular, 32-bit, present
;DPL3, exec-only code
;limit must not touch kernel mem
;calculate carefully to use functions
push (base and 0ff000000h) \
+ 0c1f800h \
+ ((base shr 10h) and 0ffh)
push (base shl 10h) + 0ffffh
push sel
call NtSetLdtEntries
;jmp far sel:l1
db 0eah
dd offset l1 – base
dw sel
l1: ;execution continues here but using LDT selector
...
```

Click to expand...

8.QueueUserAPC

Функция kernel32 QueueUserAPC () (или функция ntdll NtQueueApcThread()) может использоваться для регистрации асинхронных вызовов процедур (APC). Вызовы асинхронных процедур - это функции, которые вызываются, когда связанный поток входит в состояние "оповещения", например, путем вызова функции kernel32 Sleep(). Она также вызываются перед точкой входа потока, если они были зарегистрированы до того, как поток начал работать. Они представляют собой интересный способ для отладчика передать управление в область памяти, где выполнение может свободно возобновиться, если точка останова не размещена в соответствующем месте. Их можно использовать вместо вызова функции CreateRemoteThread() kernel32, если известно,

что целевой поток вызывает одну из функций оповещения. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
xor eax, eax
push eax
push esp
push eax
push eax
push eax ;thread entrypoint is irrelevant
push eax
push eax
call CreateThread
push eax
push eax
push offset l1
call QueueUserAPC
jmp $
l1: ;execution resumes here when thread starts
...
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
xor edx, edx
push rdx
push rsp
push rdx
sub esp, 20h
xor r9d, r9d
xor r8d, r8d ;thread entrypoint is irrelevant
xor ecx, ecx
call CreateThread
xchg edx, eax
mov rcx, offset l1
call QueueUserAPC
jmp $
l1: ;execution resumes here when thread starts
...
```

Click to expand...

9.RaiseException

Функция kernel32 RaiseException() (или функция ntdll RtlRaiseException() и функция ntdll NtRaiseException()) могут использоваться для принудительного выполнения определенных исключений, включая исключения, которые обычно использует отладчик. Различные отладчики используют разные наборы исключений. В результате любые исключения из соответствующего набора, возникающие в присутствии конкретного отладчика, будут доставляться отладчику вместо отладчика. Отсутствующее исключение можно использовать для определения наличия определенного отладчика. Некоторые отладчики позволяют при определенных обстоятельствах доставлять отладчику определенные (или все) исключения. Однако это может привести к тому, что отладчик потеряет контроль над отладчиком. Чаще всего используется исключение DBG_RIPEVENT (0x40010007). Проверка может быть выполнена с использованием этого 32-разрядного кода для проверки 32-разрядной среды Windows на 32-разрядной или 64-разрядной версии Windows:

```
xor eax, eax
push offset l1
push d fs:[eax]
mov fs:[eax], esp
push eax
push eax
push eax
push 40010007h ;DBG_RIPEVENT
call RaiseException
jmp being_debugged
l1: ;execution resumes here due to execption
...

Click to expand...
```

или использовать этот 64-битный код для проверки 64-битной среды Windows:

```
mov rdx, offset l1
xor ecx, ecx
inc ecx
call AddVectoredExceptionHandler
xor r9d, r9d
xor r8d, r8d
cdq
mov ecx, 40010007h ;DBG_RIPEVENT
call RaiseException
jmp being_debugged
l1: ;execution resumes here due to exception
...

Click to expand...
```

Известно, что один отладчик позволяет направлять выполнение определенных действий на основе параметров, предоставляемых вместе с исключением, таких как замена программной точки останова произвольным байтом в любой части памяти процесса.

10.RtlProcessFlsData

Функция ntdll RtlProcessFlsData () является недокументированной функцией, которая была представлена в Windows Vista. Она вызывается функциями kernel32 FlsSetValue() и kernel32 DeleteFiber(). При вызове с соответствующими параметрами и значениями памяти функция будет выполнять код по указателю, указанному пользователем в памяти. Если отладчик не знает об этом, управление выполнением может быть потеряно. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
push 30h
pop eax
mov ecx, fs:[eax]
mov ah, 2
inc d [ecx+eax-4] ;must be at least 1
mov esi, offset l2-4
mov [ecx+eax-24h], esi
lodsd
push esi
call RtlProcessFlsData
jmp being_debugged
l1: ;execution resumes here during processing
...
l2: dd 0, offset l1, 0, 1
```

Click to expand...

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
push 60h
pop rax
mov rcx, gs:[rax]
mov ah, 3
inc d [rcx+rax-10h] ;must be at least 1
mov rsi, offset l2-8
mov [rcx+rax-40h], rsi
lodsq
push rsi
pop rcx
call RtlProcessFlsData
jmp being_debugged
l1: ;execution resumes here during processing
...
l2: dq 0, offset l1, 0, 1
```

Click to expand...

9.WriteProcessMemory

Функция kernel32 WriteProcessMemory() (или функция ntdll NtWriteVirtualMemory()) может использоваться практически так же, как функция kernel32 ReadFile(), как описано выше, за исключением того, что источником данных является пространство памяти процесса, а не файл на диске. Это может быть использовано в текущем процессе. Вызов может быть выполнен с использованием этого 32-разрядного кода в 32-разрядной или 64-разрядной версии Windows:

```
push 0
;replace byte at l1 with byte at l2
;step-over will also result
;in uncontrolled execution
push 1
push offset l2
push offset l1
push -1 ;GetCurrentProcess()
call WriteProcessMemory
l1: int 3
l2: nop
```

или используя этот 64-разрядный код в 64-разрядных версиях Windows:

```
push 0
sub esp, 20h
;replace byte at l1 with byte at l2
;step-over will also result
;in uncontrolled execution
push 1
pop r9
mov r8, offset l2
mov rdx, offset l1
or rcx, -1 ;GetCurrentProcess()
call WriteProcessMemory
l1: int 3
l2: nop
```

Click to expand...

Один из способов преодолеть эту технику - использовать аппаратные контрольные точки вместо программных контрольных точек при переходе через вызовы функций.

12.Intentional exceptions

Исключение, которое не обрабатывается отладчиком, - это простой способ для отладчика перенести управление в область памяти, где выполнение может свободно возобновляться, если точка останова не размещена в соответствующем месте. Намек на то, что такая попытка будет предпринята, часто выглядит так: 32-битный код:

```
xor eax, eax
push offset handler ;step 1
push d fs:[eax] ;step 2
mov fs:[eax], esp ;step 3
[code to force exception]
```

Это метод структурированной обработки исключений. Он существует только в 32-битных средах Windows. В 64-разрядных средах Windows вместо динамической регистрации обработчика исключений используется векторный обработчик исключений. В какой-то момент после обработчика исключений, генерируется исключение. Существует много способов создать исключение, например, используя недопустимые или привилегированные инструкции или недопустимые обращения к памяти.

Есть способы запутать некоторые инструкции в последовательности. Первый - удалить ссылки на селектор FS. Значение в FS:[18h] является указателем на область FS, но доступно с использованием любого селектора, кроме GS, следующим образом:

```
mov eax, fs:[18h]
push d [eax] ;step 2
mov [eax], esp ;step 3
```

Тогда есть способ сделать push косвенным:

```
mov ebp, [eax]
enter 0, 0 ;step 2
mov [eax], ebp ;step 3
```

Есть другие способы получить базовый адрес для селектора FS, например, с помощью функции kernel32 GetThreadSelectorEntry(), но общая последовательность шагов остается той же. Однако есть один опубликованный метод, который делает запись в память даже косвенной. Это выглядит так:

```
push 8
pop eax
call l2
push -1 ;push fs:[0] ;step 2
mov ecx, fs
mov ss, ecx
xchg esp, eax
call l1 ;change stack limit
l1: push eax ;mov fs:[0], esp ;step 3
mov esp, ebp ;point esp somewhere legal
l2: pop esp
call esp ;push offset handler ;step 1
;handler code here

Click to expand...
```

В этом коде предполагается, что до запуска нет доступной для записи памяти. Если бы они были, то код можно было бы сократить на два байта. Это также предполагает, что стек в памяти ниже, чем код. Если это не так, то при возникновении исключения Windows прекратит процесс. Интересно, что эта версия работает только в 32-битной среде на 64-битных версиях Windows, из-за предположения о поведении селекторов,

когда происходит исключение. В частности, местоположение l2 достигается дважды, во второй раз со значением селектора SS, равным значению селектора FS. Предполагается, что команда "pop esp" в l2 вызовет исключение в это время (потому что значение регистра ESP выйдет за пределы сегмента FS), и что значение селектора SS будет восстановлено до его правильного значения. Однако это восстановление происходит только в 64-разрядных версиях Windows. Это не происходит в 32-разрядных версиях Windows. Для достижения совместимости с 32-разрядными версиями Windows и одновременного исправления всех предположений код должен выглядеть следующим образом:

```
;writable page before this point
push 0ch
pop eax
call l2
push -1 ;push fs:[0] ;step 2
push fs
pop ss
xchg esp, eax
push esp ;change stack base
call l1 ;change stack limit
l1: push eax ;mov fs:[0], esp ;step 3
mov ecx, ds
mov ss, ecx ;restore ss
xchg esp, eax ;point esp somewhere legal
push esp ;point to exception instruction
l2: pop esp ;get address so call will fault
call esp ;push offset handler ;step 1
;handler code here
```

Click to expand...

a. Наномиты

Наномиты обычно работают, заменяя инструкции ветвления инструкцией "int 3", а затем используя таблицы в коде распаковки, чтобы определить, является ли "int 3" наномитом или вызовом отладки. Если "int 3" является наномитом, то таблицы также будут использоваться для определения, следует ли использовать ветвление, адрес назначения, если ветвление используется, и насколько велика инструкция, если ветвь не используется.

Процесс, защищенный наномитами, обычно требует самостоятельной отладки. Это позволяет отладчику обрабатывать исключения, которые генерируются отладчиком при "ударе" по наномиту.

Тем не менее, нет необходимости в самостоятельной отладке, кроме как в качестве антиотладочных целях. Один процесс может зарегистрировать свой собственный обработчик исключений и обрабатывать исключения самостоятельно. Существует по крайней мере один вирус, который, как известно, демонстрирует наиболее экстремальную версию этого поведения. Вирус упорядочивает каждую из своих инструкций в случайном порядке и связывает их все с помощью наномитов, которые несут информацию о местонахождении следующей инструкции.

H.Заключение

Как мы видим, антиотладочных приемов очень много. Некоторые из них известны, а некоторые новы, и не все из них имеют хорошую защиту. Нам нужен максимальный отладчик, который знает их все.

Источник: https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf
Автор перевода: yashechka
Переведено специально для портала XSS.is (c)
Last edited: Aug 13, 2020