

IDP 3.0 (User Mode) .

Захват разрушением указателей.

Эта продвинутая техника. Она позволяет выполнить захват кода без его модификации. Впервые данная техника мной использовалась для захвата **SST**, в последствии использовалась многократно, после чего был сформулирован концепт. Захват **SST** выполнялся следующим образом. В целевой **SST** ссылка на таблицу аргументов (**ArgumentTable**) делается не валидной. Элемент таблицы адресуется парой регистров, один из которых индексирует его, другой содержит базу таблицы, предварительно загруженную из ссылки в **SST**. При обращении к таблице для извлечения из неё числа аргументов сервиса для копирования их из пользовательского стека в ядерный возникает исключение. Диспетчер исключений получает адрес инструкции вызвавшей исключение, адрес к которому произошло обращение и контекст потока на момент исключения. Выполняется загрузка в регистр адресующий таблицу оригинального адреса её, взводится флажок **TF** и выполняется рестарт инструкции. Обращение происходит к валидному (оригинальному) адресу и после исполнения инструкции генерируется трассировочное исключение. С этого места выполняется трассировка кода до входа в обработчик сервиса, после чего трассировка прекращается и тред продолжает исполняться нормально. **В данном случае созданы условия для возникновения исключения (не валидный указатель), после которого код трассируется до целевого места.**

Эти три механизма лежат в основе данной техники. Изменить адрес к которому должно произойти обращение при рестарте инструкции можно двумя способами – изменить регистры (**RGP** если используются) или изменить сегмент, посредством изменения сегментных регистров. Относительно движка для первого способа необходим полноценный дизассемблер. Используется перезагрузка сегментов как простой и универсальный способ.

Процедурные ветвления при трассировке кода необходимо пропустить. Если процедурное ветвление находится в пределах процедуры с целевым кодом, то выполняется замена адреса возврата в стеке, сразу после ветвления (межсегментные ветвления довольно редки в юзермоде). Возврат из процедуры будет выполнен на новый адрес. При этом можно выполнить возврат на оригинальный адрес, либо прежде загрузив не валидный адрес обработать исключение, перезагрузив **Eip**. Если до целевого места имеется цепочка стековых фреймов (формируются последовательностью вложенных процедурных ветвлений), то необходимо выполнить их развёртку (бактрейс) и заменить адрес возврата из необходимой процедуры, как указано выше. Используется дизассемблирование каждой инструкции в процессе трассировки (см. **Op.asm, IsCallOpcode()**).

–

Например для захвата **LdrpGetProcedureAddress()** можно использовать разные методы, рассмотрим захват описываемой техникой. Для проверки валидности псевдохэнгла модуля используется процедура **LdrpCheckForLoadedDllHandle()**. Эта процедура выполняет сканирование бызы данных загрузчика для поиска базы модуля в нём. Предварительно проверяется значение переменной **LdrpLoadedDllHandleCache**, куда сохраняется дескриптор модуля **PLDR_DATA_TABLE_ENTRY**, к которому произошло последнее обращение (таким образом выполняется кеширование для ускорения поиска). Выполним разрушение указателя в переменной **LdrpLoadedDllHandleCache**. При извлечении базы модуля из дескриптора его по ссылке в этой переменной для последующего сравн

ения его с проверяемой базой возникнет исключение (см. `\win2k\private\ntos\dll\ldrsnap.c`):

```
mov eax,dword ptr ds:[LdrpLoadedDllHandleCache]
...
cmp dword ptr ds:[eax + 18],esi
```

Оно обрабатывается средствами движка и необходимо для генерации останова и последующей обработки. Диспетчер исключений может поступить двумя способами: подменить адрес возврата из этой процедуры (находится в стековом фрейме адресуемом регистром **Ebp**) на свой код (либо на не валидный адрес) или трассировать всю процедуру. После этого **Eip** в контексте потока при возврате из процедуры будет указывать на тело предыдущей процедуры, в данном случае это **LdrpGetProcedureAddress()**. Когда произойдёт возврат в эту процедуру взводим **TF** и начинаем трассировать код. На каждом шаге определяем тип инструкции. Если это процедурное ветвление, то трассируем его, после чего заменяем в стеке адрес возврата, предварительно сохранив текущий, затем прекращаем трассировку. Тогда исполнит процедуру, после чего последует возврат не из процедуры на заменённый адрес. Загружаем в **Eip** сохранённый адрес. Таким образом процедуры пропускается:

LdrpGetProcedureAddress:

```
...
call LdrpCheckForLoadedDllHandle ; Break
...
call RtlImageDirectoryEntryToData
...
call LdrpSnapThunk
...
```

Когда при трассировке будет достигнуто целевое место, трассировка прекращается. Посредством этого можно контролировать код ниже и выше по цепочке стековых фреймов. Это не единственное решение, может применяться например разрушение структур в базе данных загрузчика для достижения подобных результатов. Может использоваться нотификация (выполняется в **LdrpRunInitializeRoutines()** если сброшен флажок **LDRP_ENTRY_PROCESSED** в описателе модуля и пр., но это не относится к сабжу.

—

В некоторых случаях эта техника позволяет избежать заикливания и рекурсивных вызовов. Например при захвате куков. В XP используется программная защита (**DEP**) стека от переполнения посредством куков. Следует рассмотреть механизм подробнее. В самом начале исполнения уязвимой процедуры в стеке сохраняется значение переменной **__security_cookie**:

RtlCreateUserThread:

```
push 324
push 7C92E140
call _SEH_prolog
mov eax,dword ptr ds:[__security_cookie]
mov dword ptr ss:[ebp-1C],eax
...
mov ecx,dword ptr ss:[ebp-1C]
call __security_check_cookie
call _SEH_epilog
ret 28
```

Перед возвратом из процедуры сохранённое в стеке значение сравнивается со значением переменной и если они отличаются, либо старшая часть сохранённого значения отлична от нуля генерируется исключение **STATUS_STACK_BUFFER_OVERRUN**:

```
__security_check_cookie:
    cmp ecx,dword ptr ds:[__security_cookie]
    jnz __report_gsfailure
    test ecx,FFFF0000
    jnz __report_gsfailure
    ret
```

__report_gsfailure() сохраняет контекст потока и вызывает финальный обработчик исключений посредством **RtlUnhandledExceptionFilter()**. В нём происходит нотификация отладчика посредством **DbgPrint()** и далее **vDbgPrintExWithPrefix()**. Последняя функция защищена куками. Если использовать захват кукков загрузкой в старшую часть переменной **__security_cookie** значения отличного от нуля (тогда при возврате из многих функций произойдёт вызов **__report_gsfailure()**), это приведёт к рекурсивным вызовам до исчерпания стека, после чего процесс будет завершён ядром. Для предотвращения этой ситуации можно использовать технику *IDP*. В начале **RtlUnhandledExceptionFilter2()** выполняется чтение командной строки (**TEB.Peb -> PEB.ProcessParameters -> RTL_USER_PROCESS_PARAMETERS.CommandLine.Buffer**):

```
RtlUnhandledExceptionFilter2:
    push 1C
    push 7C967288
    call _SEH_prolog
    mov eax,dword ptr fs:[18]
    mov eax,dword ptr ds:[eax+30]
    mov eax,dword ptr ds:[eax+10]
    mov ebx,dword ptr ds:[eax+44]
    ...
```

Посредством разрушения **PEB.ProcessParameters** этот код может быть захвачен. При этом произойдёт вызов диспетчера исключений, который получит контекст потока (в стековом фрейме, адресуемом регистром **Ebp**) достаточный для возврата на код, после вызова **__security_check_cookie()**. При возврате из многих функций управление будет передаваться диспетчеру исключений. Подобным образом можно было бы выполнить захват **_SEH_prolog()** и **_SEH_epilog()**. Обнулить сегментный регистр **Fs** и обрабатывать исключение при доступе к **TEB**. Проблема заключается в использовании быстрых системных вызовов, которые восстанавливают этот регистр в дефолтное значение. Решением может быть установка хардварных точек останова на **KiFastSystemCallRet()**.

Описание движка.

Движок выполняет захват ссылок разрушением указателей. Допустим имеется переменная, содержащая указатель на область данных известного размера. Из адреса этой области формируется смещение в пределах первых **16** страниц в адресном пространстве, которое загружается в переменную. Создаётся дескриптор в **LDT** с базой, которая в совокупности со смещением адресует целевую область памяти. До обращения к этой области памяти, через ссылку на неё в переменной, должен быть зарегистрирован векторный

обработчик исключений последним в цепочке (после диспетчера движка). Когда происходит обращение к области памяти, на которую ссылалась переменная возникнет исключение. Диспетчер исключений движка на основании адреса (области памяти) к которому произошло обращение определит соответствующий этому сегменту селектор, загрузит его в сегментные регистры, взведёт флажок **TF** и передаст управление следующему обработчику в цепочке. При этом обработчик получит код ошибки **IDP_BREAKPOINT**. Получив это сообщение обработчик должен завершить обработку, выполнив возврат на прерванный код. Произойдёт обращение к области памяти, формируемой смещением и базой сегмента. После чего возникнет трассировочное исключение, необходимое для восстановления сегментных регистров. Обработчик движка восстановит селекторы и регистр флагов, после чего передаст управление на следующий обработчик. Это второе событие, обработчик получит код его **IDP_SINGLE_STEP**. С данного места может быть начата дальнейшая обработка (трассировка и пр). Информацию об области памяти связанной с переменной диспетчер может извлечь из структуры **SEGMENT_ENTRY**, указатель на которую находится в **TEB** по смещению **0xFFC**. Смещение области памяти формируется таким образом, дабы два региона проецируемые на недоступные страницы не пересекались. Это позволяет регистрировать множество областей памяти, формируя для каждой из них свой дескриптор.

```

0x00000000 -> +----- SEG1 -----+ N
                |                     | O
                |                     |
0x00000SSS -> +-----+ A -----+
                |1         | C       | O
                |2         | C       | F
                |3         | E       | F
                +-----+ S       | S
                |         | S       | E
0x00010000 -> +-----+-----+ T
                |         | BASE   | V
0xB0000000 -> +----- SEG2 -----+ > +
                |         |         | LINE ADDR
0xB0000SSS -> +-----+ <-----+
&            |1         | R
0x00000FFF   |2         | W
                |3         | E
                +-----+-----+
                |         |

```

Таким образом движок отслеживает доступ к определённой области памяти, производя нотификацию стороннего кода при доступе к этой области. Целью его не является подмена данных. Для этого следует выделить память в начале адресного пространства (где-то ниже целевой области памяти) и создать дескриптор в **LDT** с базой, равной разности адреса целевой области и нижележащего буфера с подменяемыми данными. Далее при возникновении исключения следует загрузить селектор созданного дескриптора в сегментные регистры, обращение произойдёт к буферу.

Защита от трассировки диспетчера и дедлока.

Пользовательский обработчик не получит сообщение об трассировочном исключении в системном диспетчере ("Если возникает исключение **#DB (STATUS_SINGLE_STEP)**, то в контексте потока находящемся в стеке и текущем сбрасывается **TF (Trap Flag)**). Для иного исключения, отличного от **#DB (STATUS_SINGLE_STEP)** на момент возникновения которого был взведён **TF**, вход в диспетчер исключений (**KiUserExce**

ptionDispatcher() выполняется с взведённым **TF**. После чего генерируется трассировочное исключение (**#DB**) и **TF** сбрасывается. Если пользовательский обработчик не проверяет адреса останова, то трассировка диспетчера будет выполняться до захвата критической секции **RtlCalloutEntryLock** в **RtlCallVectoredExceptionHandlers()**. Происходят рекурсивные вызовы диспетчера исключений, трассировка **RtlEnterCriticalSection(RtlCalloutEntryLock)** сдвигается некорректным полем **RTL_CRITICAL_SECTION.LockCount** и при следующем входе в критическую секцию произойдёт deadlock, тред будет ждать сигнализацию эвента. Чтобы избежать подобной ситуации диспетчер исключений движка проверяет адрес трассировочного исключения и если он указывает на вторую инструкцию системного диспетчера (определяется динамически, если диспетчер захвачен сплайсингом, например джамп на обработчик, второй инструкцией является следующая за ветвлением), то трассировка прекращается и управление возвращается на диспетчер.

Использование пары сегментов.

Инструкция **Movs** копирует данные из сегмента **Ds:[Esi]** в сегмент **Es:[Edi]**. Использование префиксов переопределения сегмента изменит сегмент, из которого копируются данные. Эта инструкция требует специальной обработки. Так как для остальных инструкций перезагружаются регистры **Fs**, **Es**, **Ds** и **Gs** (обнуляется планировщиком), для **Movs** выполняется перезагрузка регистров в зависимости от направления пересылки данных. Если выполняется запись в захваченную область памяти, то перезагружается только регистр **Es**. Если выполняется чтение из захваченной области памяти, то определяется наличие префиксов переопределения сегментов. Если их нет, то перезагружается регистр **Ds**. Если определён префикс сегмента **Es**, перезагружается только **Es**. Иначе перезагружаются регистры **Fs**, **Ds** и **Gs**.

Сегменты кода (**Cs**) и стека (**Ss**) не переопределяются. Например инструкция **Push dword ptr Ds:[Eax]** использует два сегмента, стека (**Ss**) и данных (**Ds**), последний может быть переопределён. Захват стека не используется. Переопределение сегмента кода не возможно. Обычно данные и код разделены и необходимость захвата ссылки, адресуемой через регистр **Cs** редка. Для захвата ссылки на процедуру следует подменить указатель на неё в переменной.

Следующие ситуации не допустимы:

- Захват ссылки, которая используется в процессе обработки исключения. Например захват **TEB** и **PEB** в ссылках этих структур приведёт к рекурсивному вызову диспетчера до исчерпания стека, так как **RtlDispatchException()** обращается к этим ссылкам. Для обхода этого следует выполнить захват системного диспетчера исключений жёсткой модификацией его кода. Удобна атомарная замена смещения в инструкции **Call RtlDispatchException** на диспетчер движка. Если имеется возможность записи в ядро, можно изменить ссылку на системный диспетчер в переменной ядра **KeUserExceptionDispatcher**.

- Выполнять захват области памяти, адрес которой проверяется кодом. Например захват базы данных загрузчика в ссылке на него, которая находится в **Peb.Ldr** приведёт к зависанию. Поток будет выполнять бесконечный цикл.

LdrpCheckForLoadedDll() :

```
...
7C916A49:
    cmp edi,esi
```

```

je short 7C916A7C
mov ebx,edi
mov edi,dword ptr ds:[edi]
cmp dword ptr ds:[ebx+8],0
je short 7C916A49
push 1
lea eax,dword ptr ds:[ebx+24]
push eax
lea eax,dword ptr ss:[ebp-240]
push eax
call ntdll.RtlEqualUnicodeString
test al,al
je short 7C916A49
...

```

Esi - указывает на текущий элемент списка. **Esi** - указывает на начало списка (**PEB_LDR_DATA.InLoadOrderModuleList.Flink**), то есть **[Peb.Ldr] + 0xC**, соответственно захвачен, не валидный и на него нет ссылок в этом двусвязном списке. Указатели никогда не совпадут, так как один из них изменён при захвате ссылки.

- Ссылка не должна передаваться в ядро или использоваться им (напр. **PEB.LoaderLock** изменяется ядром при завершении треда). Иначе в ядре возникнет исключение и сервис вернёт ошибку.

Простой пример. В базе данных загрузчика (**Ldr**) имеется множество ссылок, например указатель на юникодовское имя модуля (**LDR_DATA_TABLE_ENTRY.BaseDllName.Buffer**). Выполнив захват этого указателя из описателя первого модуля в списке средствами движка, после чего пытаемся загрузить сторонний модуль с помощью **LoadLibraryA()**. Диспетчер исключений будет вызван многократно, это поле используется загрузчиком для поиска модуля по имени. Исключение возникает внутри **RtlEqualUnicodeString()** на инструкции:

```
mov cx,word ptr ds:[edi] ; UNICODE "ntdll.dll"
```

Из регистра **Ebp** в контексте можем получить исходный указатель для бактрейса и выполнив его получим последовательность вызовов:

```

ntdll.RtlEqualUnicodeString
ntdll._LdrpCheckForLoadedDll@20
ntdll._LdrpLoadImportModule@20
ntdll._LdrpHandleOneNewFormatImportDescriptor@20
ntdll._LdrpHandleNewFormatImportDescriptors@16
ntdll._LdrpWalkImportDescriptor@8
ntdll._LdrpLoadDll@24
kernel32._LdrLoadDll@16
kernel32.LoadLibraryExW
kernel32.LoadLibraryExA
kernel32.LoadLibraryA

```

При сравнении имён модулей для каждого символа имени происходит останов. Например для захвата кода после вызова **LoadLibraryA()** следует выполнить бактрейс, найти фрейм с указателем в эту функцию и подменить в нём адрес возврата. Если диспетчер исключений является заглушкой, то после отработки загрузка завершится успешно.

Модель вызова.

Обращение к движку выполняется посредством вызова нача

ла кода. В регистре **Eax** указывается номер вызываемого сервиса. Движок сам очищает стек от параметров. Используются следующие сервисы:

- Инициализация движка.

```
#define IDP_INITIALIZE_ENGINE 0x00000000
```

```
typedef NTSTATUS (*PENTRY) (  
    );
```

*При инициализации генерируется два исключения, первое **STATUS_PRIVILEGED_INSTRUCTION(#GP)**, после него трассировочное **STATUS_SINGLE_STEP(#DB)**. Это внутренние исключения, они обрабатываются движком.*

- Захват ссылки.

```
#define IDP_ADD_REFERENCE 0x00000001
```

```
typedef NTSTATUS (*PENTRY) (  
    IN OUT PVOID *Reference  
    IN ULONG SpaceSize  
    );
```

- Регистрация векторного обработчика исключений.

```
#define IDP_ADD_VEH 0x00000002
```

```
typedef PVOID (*PENTRY) (  
    IN ULONG First,          // Ноль.  
    IN PVECTORED_EXCEPTION_HANDLER Handler  
    );
```

```
typedef LONG (*PVECTORED_EXCEPTION_HANDLER) (  
    IN OUT PEXCEPTION_POINTERS *ExceptionInformation  
    );
```

Пользовательский векторный обработчик исключений должен быть установлен следующим в цепочке, после диспетчера исключений движка. Это достигается передачей первым параметром нуля.

- Отмена регистрации обработчика исключений.

```
#define IDP_REMOVE_VEH 0x00000003
```

```
typedef ULONG (*PENTRY) (  
    IN PVOID Handle  
    );
```

- Получение адреса экспорта по имени или его хэшу.

```
#define IDP_QUERY_ENTRY 0x00000004
```

```
typedef NTSTATUS (*PENTRY) (  
    IN PVOID ImageBase OPTIONAL,  
    IN PVOID HashOrFunctionName,  
    IN PCOMPUTE_HASH_ROUTINE HashRoutine OPTIONAL,  
    IN ULONG PartialCrc,  
    OUT *PVOID Entry  
    );
```

```
typedef ULONG (*PCOMPUTE_HASH_ROUTINE) (
    IN ULONG UserParameter,
    IN PVOID Buffer,
    IN ULONG Length
);
```

Если калбэк вычисляющий хэш не задан, то второй параметр рассматривается как указатель на имя экспорта. Если база модуля не задана, то используется **ntdll.dll**. Калбэк должен вернуть в регистре **Eax** хэш для строки.

- Получает адресов экспорта по **Crc32** имён.

```
#define IDP_QUERY_ENTRIES 0x00000005
```

```
typedef NTSTATUS (*PENTRY) (
    IN PVOID ImageBase OPTIONAL,
    IN ULONG PartialCrc,
    IN PULONG Crc32List,
    OUT *PVOID EntriesList
);
```

Маркером конца списка является ноль. Перечисляется экспорт модуля, для каждого имени вычисляется **Crc32** посредством **ntdll!RtlComputeCrc32**, выполняется сравнение со значением в массиве. Если значения равны, выполняется выборка следующего элемента из массива.

Структура **SEGMENT_ENTRY** описывает сегмент, к которому произошло обращение:

```
typedef struct _SEGMENT_ENTRY {
    PVOID SegmentBase;
    PVOID SegmentLimit;
    PVOID SegmentAddress;
    PVOID Reference;
} SEGMENT_ENTRY, *PSEGMENT_ENTRY;
```

Ссылка на **SEGMENT_ENTRY** находится в **TEB** по смещению:

```
#define IDP_SEGMENT_ENTRY_OFFSET 0x00000FFC
```

И является частью структуры, описывающей останов:

```
typedef struct _THREAD_STATE {
    ULONG reFlags;
    PSEGMENT_ENTRY Segment;
} THREAD_, *PTHREAD_STATE;
```

Смещение которой в **TEB**:

```
#define IDP_THREAD_STATE_OFFSET 0x00000FF8
```

Коды событий для **VEH**:

```
#define IDP_BREAKPOINT 0x80FE0001 // Останов.
#define IDP_SINGLE_STEP 0x80FE0002 // Трассировка.
```

По стилю написания движёк является микодом (**MI**), то есть кодом с изменяемым графом и может быть отморфлен.

Декабрь 2009, VirusTech.org