API Hashing in 64-bit Malware Shellcode

March 7 to 15 April 2022

By Melodi Joy Halim

What I've Been Up To...

Overview of the Past Weeks

Week 1 (7 - 13 Mar)	:	Windows Shellcode in General
Week 2 (14 - 20 Mar)	:	API Hashing and Finding Shellcodes
Week 3 (21 - 27 Mar)	:	Focus on x64 Shellcode and Differences
Week 4 (28 Mar - 3 Apr)	:	Metasploit / DoublePulsar / BendyBear Shellcode Hashing Algorithms
Week 5 (4 - 10 Apr)	:	Hashing Algorithms in Appendix
Week 6 (11 - 14 Apr)	:	BADHATCH Shellcode Hashing Algorithm

Overview of the Past Weeks

Week 1 (7 - 13 Mar)	:	How the heck does shellcode work in general? (PEB, etc.)
Week 2 (14 - 20 Mar)	:	What does API hashing really mean? Where do you use it?
Week 3 (21 - 27 Mar)	:	Realising I was supposed to focus on x64 and dying on the inside
Week 4 (28 Mar - 3 Apr)	:	Sad attempts to look at some malware shellcodes + trying to find more
Week 5 (4 - 10 Apr)	:	COVID week (did all the appendix stuff, mostly hashing algorithms)
Week 6 (11 - 14 Apr)	:	WHY DID I JUST REALISE BADHATCH EXISTED?!

Findings
(Mostly in Literature Survey)

1.

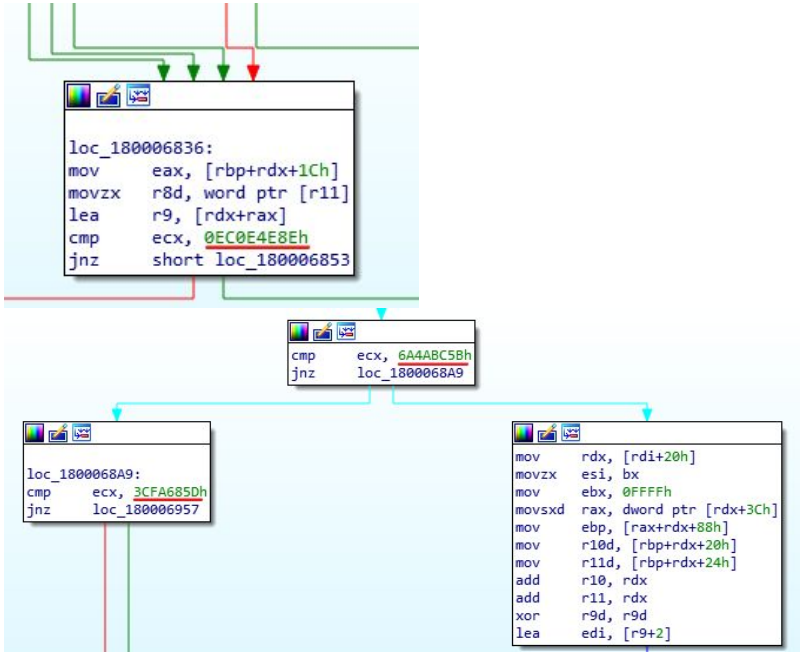
x64 shellcode can/tend to still use 32-bit hashes/hashing algorithms for API hashing

32-bit Hashes in x64 Shellcode

```
mov     rcx, r14
mov     r10d, 726774Ch
call    rbp
mov     rdx, r13
push    101h
pop     rcx
mov     r10d, 6B8029h
call    rbp
push    0Ah
pop     r14
push    rax
push    rax
xor     r9, r9
xor     r8, r8
inc     rax
mov     rdx, rax
inc     rax
mov     rcx, rax
mov     r10d, 0E0DF0FEAh
call    rbp
```

```
pop     r8
mov     rdx, r12
mov     rcx, rdi
mov     r10d, 6174A599h
call    rbp
test    eax, eax
jz      short loc_160
dec     r14
jnz     short loc_13E
push    56A2B5F0h
call    rbp

sub     rsp, 10h
mov     rdx, rsp
xor     r9, r9
push    4
pop     r8
mov     rcx, rdi
mov     r10d, 5FC8D902h
```



2.

The most common hashing algorithms still revolve around common operations

Common Operations + Variations

Operations:

1. ROR*XX* — MOST COMMON
2. ADD (each letter) — MOST COMMON
3. XOR — MOST COMMON
   a. Hardcoded hash — E.g. playWith**0xe8677835**Hash
   b. Changing key — E.g. rol3Xor**Eax**
   c. Next letter — E.g. rolNXorHash32
4. SHR*XX* / SHL*XX* — E.g. **shl7Shr**9XXXHash32 / **shr2Shl**5XorHash32
5. ROL — E.g. **rol**NXXXHash32
6. IMUL — E.g. **imul**83hAdd / **imul**21hAddHash32
7. OR — E.g. **or**21XorRor11Hash32 / **or**23XorRor17Hash32

Variations:

- Changing the number of bits (i.e. *XX*)
- Reversing / changing the order of the operations
- Converting UPPERCASE to lowercase / lowercase to UPPERCASE
- Adding final rounds / null character

Metasploit & Cobalt Strike

API Function Names:
ror13AddHash32,

DLL Names:
ror13AddHash32AddDll,

EternalBlue Exploit Payload:
ror13AddHash32Sub20h

ror13AddHash32Sub20h

```
cmp     al, 61h ; 'a'
jl      short loc_37
sub     al, 20h ; ' '

ror     r9d, 0Dh
add     r9d, eax
loop    loc_2D
push    rdx
mov     rdx, [rdx+20h]
mov     eax, [rdx+3Ch]
push    r9
add     rax, rdx
cmp     word ptr [rax+18h]
jnz     loc_CB
mov     eax, [rax+88h]
test    rax, rax
jz      short loc_CB
add     rax, rdx
mov     r8d, [rax+20h]
```

ror13AddHash32

```
xor     rax, rax
lodsb
ror     r9d, 0Dh
add     r9d, eax
cmp     al, ah
jnz     short loc_81
add     r9, [rsp+8]
cmp     r9d, r10d
jnz     short loc_72
pop     rax
mov     r8d, [rax+24h]
add     r8, rdx
mov     cx, [r8+rcx*2]
mov     r8d, [rax+1Ch]
add     r8, rdx
mov     eax, [r8+rcx*4]
add     rax, rdx
pop     r8
pop     r8
```

ror13AddHash32AddDll

DoublePulsar Backdoor Shellcode

DLL Name:
rol5XorHash32

```
xor     rbx, rbx
mov     rbx, gs:[rbx+30h] ; Linear self pointer to TEB
mov     rbx, [rbx+60h]  ; PEB pointer inside the TEB
mov     rbx, [rbx+18h]  ; PebLdr inside the TEB
mov     rbx, [rbx+10h]  ; Module list inside PebLdr

loc_E51:                ; BEGIN MODULE MATCHING LOOP
mov     rsi, [rbx+60h]  ; rsi -> module name
test    rsi, rsi
jz      return

mov     rbp, [rbx+30h]  ; rbp -> base of dll (MZ header)
test    rbp, rbp
jz      return

xor     rdx, rdx

loc_E6E:                ; Hash module name in RDX
rol     edx, 5
lodsw
or      al, 20h
xor     dl, al
cmp     word ptr [rsi], 0
jnz     short loc_E6E

mov     rbx, [rbx]      ; Advance to next loaded module
cmp     rdx, [rsp+38h+arg_module_name_hash] ; Check if hash matched
jnz     short loc_E51

rbp_module_base = rbp
mov     rdi, rbp_module_base
cmp     word ptr [rdi], 'ZM' ; Check for MZ header
jnz     short return
```

3.

There are only very few pure 64-bit hashing algorithms that are specifically meant for 64-bit hashes.

FIN8's BADHATCH x64 Shellcode

```
; DATA XREFS from fcn.0000055e @ +0x1e9c, +0x9e2a, +0xb5d4
0x0000003f      488b4f10        mov rcx, qword [rdi + 0x10] ; in
; XREFS: DATA 0x00004904  DATA 0x00004a0f  DATA 0x00008b4d  DAT
; XREFS: DATA 0x00009a9b  DATA 0x0000b467  DATA 0x0000b5a6
0x00000043      48ba58af711b.   movabs rdx, 0xb6233cd91b71af58 ;
; XREFS: DATA 0x00003600  DATA 0x0000487e  DATA 0x00007fb5  DAT
; XREFS: DATA 0x0000bedd
0x0000004d      e8fa010000      call fcn.0000024c
; XREFS: DATA 0x0000106e  DATA 0x0000200c  DATA 0x0000412a  DAT
; XREFS: DATA 0x00005e9f  DATA 0x0000681d  DATA 0x00007050  DAT
; XREFS: DATA 0x0000b7f0
0x00000052      488b4f10        mov rcx, qword [rdi + 0x10] ; in
; XREFS: DATA 0x000026c4  DATA 0x0000299e  DATA 0x00006661  DAT
; XREFS: DATA 0x000074d1  DATA 0x0000bbf8
0x00000056      48bac89fa925.   movabs rdx, 0xb279b3c825a99fc8 ;
; XREFS: DATA 0x00000678  DATA 0x00001684  DATA 0x00005d9b  DAT
; XREFS: DATA 0x0000a0e8  DATA 0x0000a127
0x00000060      488bf0          mov rsi, rax                ; in
; XREFS: DATA 0x00002845  DATA 0x00002a19  DATA 0x00004792  DAT
; XREFS: DATA 0x0000b329
0x00000063      e8e4010000      call fcn.0000024c
; XREFS: DATA 0x00001cf3  DATA 0x00006dac  DATA 0x000080ed  DAT
; XREFS: DATA 0x0000aa42  DATA 0x0000ac4e
```

64-BIT HASHES!

There always seemed to be a call for **fcn.0000024c** after loading the hashes into rdx, and afterwards it moves eax into rsi

FIN8's BADHATCH x64 Shellcode

Confirming **fcn.0000024c** was the function for calculating hashes:

Hash => rdx => r12 cmp rax

1. **Comparison to hashes** loaded into rdx

Start of Function

```
; arg int64_t arg2 @ rsi
; arg int64_t arg3 @ rdx
; arg int64_t arg4 @ rcx
mov rax, rsp
mov qword [rax + 8], rbx
mov qword [rax + 0x10], rbp
; arg2
mov qword [rax + 0x18], rsi
; arg1
mov qword [rax + 0x20], rdi
push r12
push r14
push r15
; arg4
movsxd rax, dword [rcx + 0x3c]
xor r11d, r11d
; arg3
mov r12, rdx
mov r9d, dword [rax + rcx + 0x88]
; arg4
mov r8, rcx
```

Near to End of Function

```
0x34f [on]
; CODE XREF from fcn.0000024c @ 0x30c
mov rcx, rdi
shr rcx, 0x17
xor rcx, rdi
imul rcx, r14
mov rax, rcx
shr rax, 0x2f
xor rax, rcx
cmp rax, r12
je 0x39a
t f
```

FIN8's BADHATCH x64 Shellcode

Confirming **fcn.0000024c** was the function for calculating hashes:

2. Retrieval of **ordinal** and **function relative address** at the end of function

```
0x00000393      415f          pop r15
0x00000395      415e          pop r14
0x00000397      415c          pop r12
0x00000399      c3            ret
; CODE XREF from fcn.0000024c @ 0x36a
> 0x0000039a      418b4124      mov eax, dword [r9 + 0x24]
0x0000039e      4903c0        add rax, r8
0x000003a1      420fb71458    movzx edx, word [rax + r11*2]
0x000003a6      418b411c      mov eax, dword [r9 + 0x1c]
0x000003aa      498d0c00      lea rcx, [r8 + rax]
0x000003ae      8b0491        mov eax, dword [rcx + rdx*4]
0x000003b1      4903c0        add rax, r8
< 0x000003b4      ebc9          jmp 0x37f
```

Retrieval of ordinal

Retrieval of RVA of function

FIN8's BADHATCH x64 Shellcode

Confirming **fcn.0000024c** was the function for calculating hashes:

3. **Calls to rsi register** after function

```
0x00000043      48ba58af711b.  movabs rdx, 0xb6233cd91b71af58
; XREFS: DATA 0x00003600  DATA 0x0000487e  DATA 0x00007fb5  DA
; XREFS: DATA 0x0000bedd
0x0000004d      e8fa010000     call fcn.0000024c
; XREFS: DATA 0x0000106e  DATA 0x0000200c  DATA 0x0000412a  DA
; XREFS: DATA 0x00005e9f  DATA 0x0000681d  DATA 0x00007050  DA
; XREFS: DATA 0x0000b7f0
0x00000052      488b4f10       mov rcx, qword [rdi + 0x10] ; i
; XREFS: DATA 0x000026c4  DATA 0x0000299e  DATA 0x00006661  DA
; XREFS: DATA 0x000074d1  DATA 0x0000bbf8
0x00000056      48bac89fa925.  movabs rdx, 0xb279b3c825a99fc8
; XREFS: DATA 0x00000678  DATA 0x00001684  DATA 0x00005d9b  DA
; XREFS: DATA 0x0000a0e8  DATA 0x0000a127
0x00000060      488bf0         mov rsi, rax                ; i
; XREFS: DATA 0x00002845  DATA 0x00002a19  DATA 0x00004792  DA
; XREFS: DATA 0x0000b329
0x00000063      e8e4010000     call fcn.0000024c
```

```
om fcn.0000055e @ +0x1a4d, +0x2685
498bd7          mov rdx, r15
x000008ef  DATA 0x000009ef  DATA 0x00002
x00006d48  DATA 0x00006e06  DATA 0x00007
ffd6            call rsi
om fcn.0000055e @ +0x1449, +0x1f21, +0x2
4d63553c        movsxd r10, dword [r13 +
om fcn.0000055e @ +0x8858, +0xa929
4533db          xor r11d, r11d
```

rax contained the function name address => moved into **rsi**

FIN8's BADHATCH x64 Shellcode

Hashing Algorithm in fcn.0000024c:

```
0x34f [on]
; CODE XREF from
mov rcx, rdi
shr rcx, 0x17
xor rcx, rdi
imul rcx, r14
mov rax, rcx
shr rax, 0x2f
xor rax, rcx
cmp rax, r12
je 0x39a
```

```
0x330 [om]
; CODE XREF fr
mov rcx, rbp
shr rcx, 0x17
xor rcx, rbp
imul rcx, r14
mov rax, rcx
shr rax, 0x2f
xor rax, rcx
xor rdi, rax
imul rdi, r15
```

```
0x2dc [og]
; CODE XREF from fcn.
mov rax, qword [rdx]
add rdx, 8
mov rcx, rax
shr rcx, 0x17
xor rcx, rax
imul rcx, r14
mov rax, rcx
shr rax, 0x2f
xor rax, rcx
xor rdi, rax
imul rdi, r15
```

3 different parts of the function for different conditions / iterations; Problem is, I didn't have enough time to figure out the small conditions required to trigger each / how the process would go for iterating through each character in the string

```
0x287 [ob]
mov r10d, dword [r9 + 0x20]
movabs r15, 0x880355f21e6d1965
movabs r14, 0x2127599bf4325c37
; arg4
add r10, rcx
```

```
mov ebx, ecx
shr rax, 3
imul rdi, r15
; arg3
lea rsi, [rdx + rax*8]
movabs rax, 0xab00d73069525d99
xor rdi, rax
jmp 0x302
```

- shr17 / shr23
- xor
- imul 0x2127599bf4325c37
- shr2f / shr47
- xor
- imul 0x880355f21e6d1965

Challenges Faced

Biggest Challenge:

Finding relevant articles/shellcodes

"Literature Survey of 64-bit Malware Shellcode API Hashing Technique"

When looking at an article:

1. Is it talking about **shellcode**?
    a. Alot talked about API hashing in general malware
2. Is it talking about shellcode using **API hashing**?
    a. Alot talked about shellcode that didn't use API hashing (just pushed exact API function name string onto stack)
3. Is it talking about **x64** shellcode using API hashing?
    a. Even if it talked about the above 2, this is where most articles were scrapped because most were talking about x86 shellcode
4. Does it show the **hashing algorithm** of the x64 shellcode using API hashing?
    a. Only few articles detailed the hashing algorithm used by the shellcode, the rest only mentioned the DLLs called and that's it

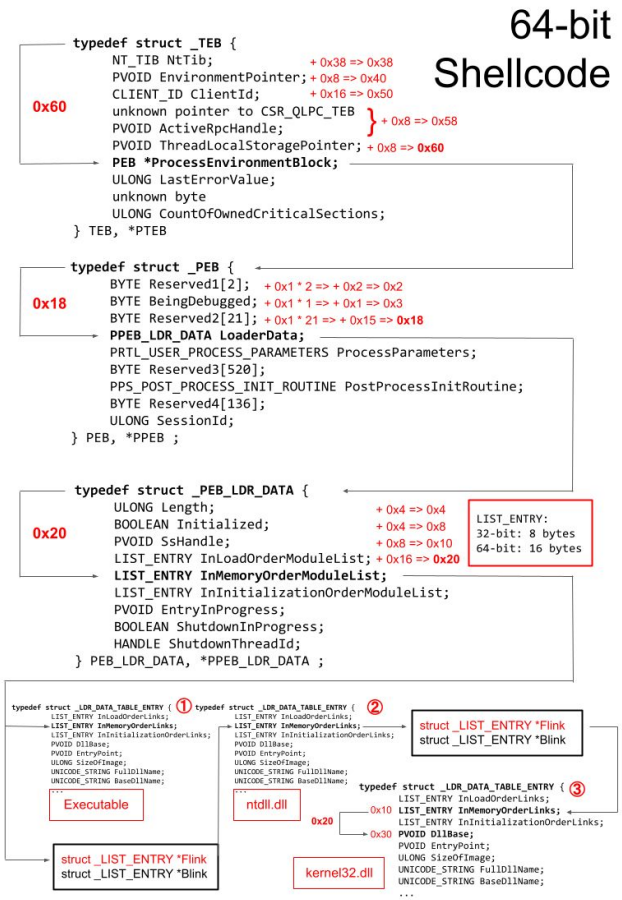
Things I Have Learnt

Shellcode Common Procedure

- A little bit of Linux on the first week
- Mostly Windows x64 shellcode

- Got the time to properly sort out my understanding and create diagrams to better reinforce

- This helped later on when looking at shellcode



What is Common in Hashing Algorithms for API Hashing

- I had assumed it was a lot more complicated, but it was just creative implementations of the same few methods
- It doesn't take a lot of operations for hashing in shellcodes
- Learnt about weirder algorithms like PoisonIvyHash, Carbanak, FNV, etc.

x64 is Different and Still Quite Rare

- The moment I realised that the topic SPECIFICALLY said “64-bit” was when I panicked
- Not many shellcodes online written for 64-bit (other than 32-bit shellcodes with Heaven’s Gate), and even less for those that have API hashing in them

32-bit Windows Shellcode on Exploit DB:

FIRST PREVIOUS **1** 2 3 4 5 6 7 NEXT LAST

64-bit Windows Shellcode on Exploit DB:

FIRST PREVIOUS **1** 2 NEXT LAST

(✖ ∩ ✖)
(×_×;)

shell-storm.org:

Windows

- Windows/64 - Obfuscated Shellcode x86/x64 Download And Execute [Use PowerShell] - Generator *by Ali Razmjoo*
- Windows/64 - Add Admin, enable RDP, stop firewall and start terminal service - 1218 bytes *by Ali Razmjoo*
- Windows/64 - (URLDownloadToFileA) download and execute - 218+ bytes *by Weiss*
- Windows/64 - Windows Seven x64 (cmd) - 61 bytes *by agix*
- Windows - Add Admin, enable RDP, stop firewall and start terminal service - 1218 bytes *by Ali Razmjoo*
- Windows - Add Admin User Shellcode - 194 bytes *by Giuseppe D'Amore*
- Windows - Safari JS JITed shellcode - exec calc (ASLR/DEP bypass) *by Alexey Sintsov*
- Windows - Vista/7/2008 - download and execute file via reverse DNS channel *by Alexey Sintsov*
- Windows - sp2 (En + Ar) cmd.exe - 23 bytes *by AnTi SeCuRe*
- Windows - add new local administrator - 326 bytes *by Anastasios Monachos*
- Windows - pro sp3 (EN) - add new local administrator 113 bytes *by Anastasios Monachos*
- Windows - xp sp2 PEB IsBeingdebugged shellcode - 56 bytes *by Anonymous*
- Windows - XP Pro Sp2 English Message-Box Shellcode - 16 Bytes *by Aodrulez*
- Windows - XP Pro Sp2 English Wordpad Shellcode - 15 bytes *by Aodrulez*
- Windows - Write-to-file Shellcode *by Brett Gervasoni*
- Windows - telnetbind by winexec - 111 bytes *by DATA_SNIPER*
- Windows - useradd shellcode for russian systems - 318 bytes *by Darkeagle*
- Windows - XP SP3 English MessageBoxA - 87 bytes *by Glafkos Charalambous*
- Windows - SP2 english (calc.exe) - 37 bytes *by Hazem mofeed*
- Windows - SP3 english (calc.exe) - 37 bytes *by Hazem mofeed*
- Windows - Shellcode (cmd.exe) for XP SP2 Turkish - 26 Bytes *by Hellcode*
- Windows - Shellcode (cmd.exe) for XP SP3 English - 26 Bytes *by Hellcode*
- Windows - XP SP3 EN Calc Shellcode - 16 Bytes *by John Leitch*
- Windows - win32/PerfectXp-pc1/sp3 (Tr) Add Admin Shellcode - 112 bytes *by KaHPeSeSe*
- Windows - PEB Kernel32.dll ImageBase Finder - 49 Bytes *by Koshi*

32

E.g. BendyBear x64 Shellcode

The BendyBear sample was determined to be x64 shellcode for a stage-zero implant whose sole function is to download a more robust implant from a command and control (C2) server. Shellcode, despite its name, is used to describe the small piece of code loaded onto the target immediately following exploitation, regardless of whether or not it actually spawns a command shell. At 10,000+ bytes, BendyBear is noticeably larger than most, and uses its size to implement advanced features and anti-analysis techniques, such as modified RC4 encryption, signature block verification, and polymorphic code.

BendyBear: Only x86 shellcode and Waterbear loaders

MALWARE bazaar by ABUSE.ch

Browse | Upload | Hunting

Malware Samples

The table below shows all malware samples that are associated with this particulare tag (**max 400**).

Show 50 entries

Firstseen (UTC)	SHA256 hash	Tags
2021-02-10 08:39:24	2a09ec2d6edadd06e18c...	BendyBear
2021-02-10 08:39:21	5d1414b47d88e95ae661...	BendyBear
2021-02-10 08:39:18	76ef704d21fbaaceca8a1...	BendyBear
2021-02-10 08:39:15	9880ba4f93cade2f6bbb4...	BendyBear
2021-02-10 08:39:11	682122f34027e3f802592...	BendyBear
2021-02-10 08:39:08	49901034216a16cfd05c6...	BendyBear

Found! x86

Showing 1 to 6 of 6 entries

Shellcode Samples

* can't find x-x

x64 - (version 0.24)
64CC899EC85F612270FCFB120A4C80D52D78E68B05CAF1014D2FE06522F1E2D0
wg1.inkeslive[.]com

x86 - (version 0.1)
49901034216a16cfd05c613f438eccee4a7bf6079a7988b3e7094d9498379558
web2008.rutentw[.]com

x86 WaterBear Loaders

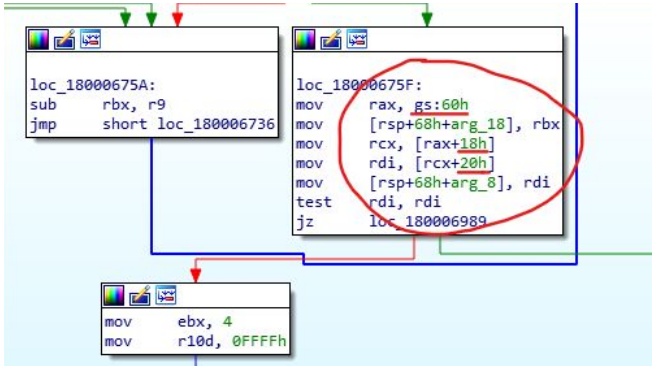
The following executables have been identified as loaders/injectors that contain older WaterBear x86 shellcode. The shellcode code is identical to the x86 sample 49901034216.... (version 0.1) listed above.

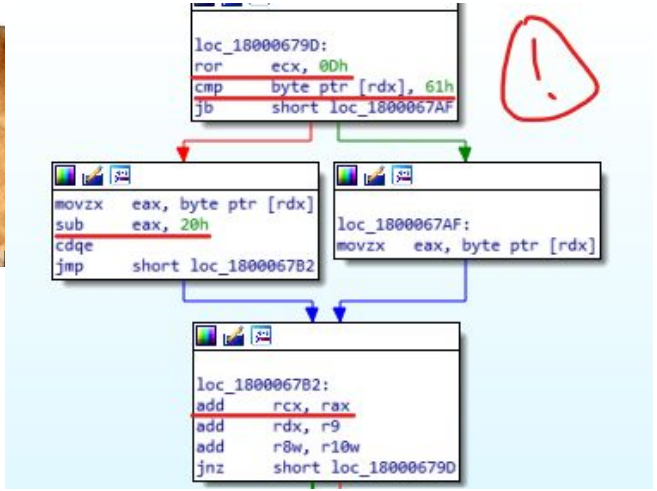
5d1414b47d88e95ae6612d3fc211c29b35cc5db4a8a992f5e27cff5203ebf44b
9880ba4f93cade2f6bbb4cc8efdcf087e8ac51b5c209ee32ad8134eb87ef70e1
682122f34027e3f8025928d446989b02952449f5e5930c2670f8f789f41573ff
2a09ec2d6edadd06e18c841e0ed794ba3eeb21818476f75ccc0e5d40e08eac80
76ef704d21fbaaceca8a131429ccfb9f5de3d8f43a160ddd281ffeafc391eb98

How to Somewhat **Look At Shellcode**

- Got the chance to try looking at some shellcode
- but relied on already disassembled ones online for filling in significant parts of the literature survey (to be safe)

Seeing everything constantly discussed in articles but now in real life feels like:

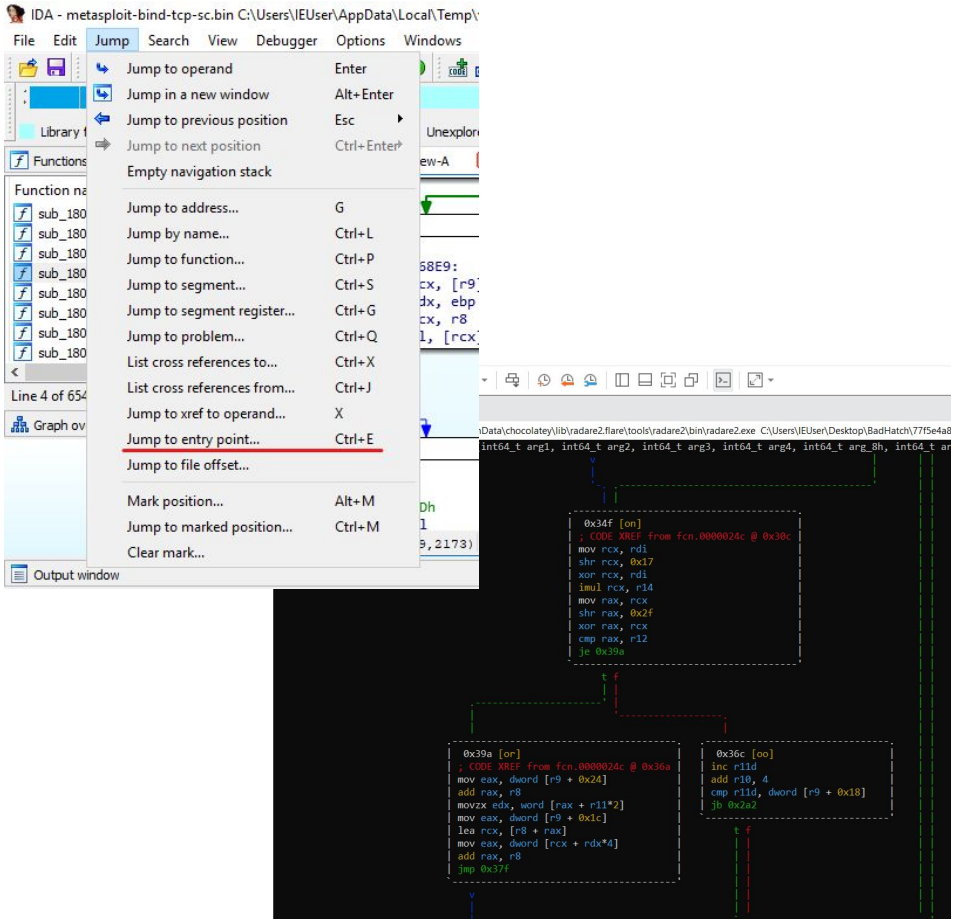




Getting Slightly **Familiar with IDA/radare2**

- Preparation for next semester where we actually use it in Polytechnic

- Converting the shellcode into exe to try to get it to work on debuggers was taking a bit too much time



Shellcode2exe! But has a limit to shellcode size…

README.md

X64 Shellcode2Exe Package

This tool is a compiled 64 bit executable for the tool shellcode2exe. It allows to convert a shellcode in an executable file.

```
shellcode2exe.exe: error: Payloads over 4k for EXE files are not supported
```

Usage

```
Shellcode to executable converter
by Mario Vilas (mvilas at gmail dot com)

Usage:
        shellcode2exe.exe payload.bin [payload.exe]
```

Things to Improve…

I should have started looking for shellcodes earlier.

Thank You!

I now have a brief understanding of x64 Windows Shellcode.

Contact Number For Scolding:

(+65) 9227 1813