64-bit Malware Shellcode

API Hashing Techniques

A Literature Survey

Melodi Joy Halim
Internship @ CSA (March 2022)
National Cyber Incident Response Centre (NCIRC)
Cyber Security Agency of Singapore (CSA)

March 2022

Project Title: Literature Survey of 64-bit Malware Shellcode API Hashing Technique

Project Description:

Windows API calls are a set of functions and data structures that any Windows program uses to ask the Windows OS to perform a task. Similarly, malware authors use Windows API as well instead of reinventing the wheel. In modern times, malware authors have improved their obfuscation techniques to hide the names of the API they wish to call. One of the popular techniques is hashing and it can be seen in shellcodes.

Side notes: A hash will be passed into a function that will hash each export of the DLL in question, until a matching pair is found
Existing: FireEye Precalculated String Hashes, OA Labs HashDB

Project Scope:

- Try out 64-bit shellcode
- Perform a literature survey on the type of hashing algorithms commonly used by malware (64-bit malware shellcode)

Table of Contents

Introduction

General Definition of API Hashing

API Hashing has been around for a relatively long time, being seen in malware from as early as 1998, and even till this day where it is still extremely common for both legitimate and obfuscation purposes.

The term refers to the use of the hashes of Windows API function names when calling them, such that the program resolves the function's virtual address by iterating through all exported function names of the specified module, and calculating each of their hashes to be compared to the given hash in order to identify the desired function.

As the API function is not directly called, its name is not reflected or left present in neither the code nor the import address table of the file.

API Hashing (B0032.001) is now also listed as a method under Anti-Static Analysis - Executable Code Obfuscation (B0032) in the Malware Behavior Catalog Matrix by the MITRE Malware Attribute Enumeration and Characterization (MAEC) Project.

MAEC™

ID	B0032
Objective(s)	Anti-Static Analysis
Related ATT&CK Technique	None

Executable Code Obfuscation

Executable code can be obfuscated to hinder disassembly and static code analysis. This behavior is specific to a malware sample's executable code (data and text sections).

For encryption and encoding characteristics of malware samples, as well as malware obfuscation behaviors related to non-malware-sample files and information, see Obfuscated Files or Information.

Methods

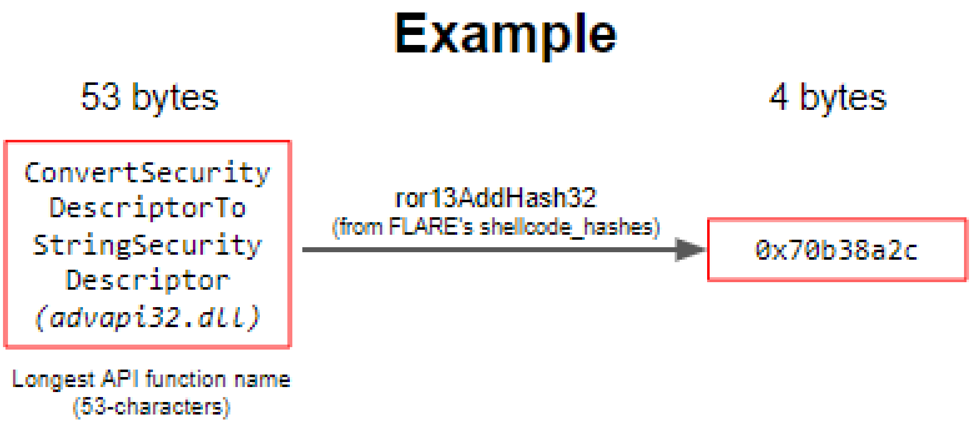
Name	ID	Description
API Hashing	B0032.001	Instead of storing function names in the Import Address Table (IAT) and calling GetProcAddress, a DLL is loaded and the name of each of its exports is hashed until it matches a specific hash. Manual symbol resolution is then used to access and execute the exported function. This method is often used by shellcode because it reduces the size of each import from a human-readable string to a sequence of four bytes. The Method is also known as "Imports by Hash" and "GET_APIS_WITH_CRC." [1]

As indicated in the description of such an obfuscation technique, shellcode in particular is where API hashing is often found given its compact nature.

Uses of API Hashing in Shellcode

Due to size constraints, shellcode authors may not hope to include the full-length name of each API function.

Instead, they may resort to computing equal-length hashes for each of these API functions, including the hashes in their shellcode for resolving imports instead.



This reduces the size of each API function name to a set number of bytes, effectively resolving the issue of size restrictions in shellcode.

Malicious Uses

API hashing also serves as a great step for obfuscation in malware.

As mentioned earlier, the usage of suspicious APIs would be masked from detection tools and analysts due to its presence being hidden in both code and the import table, thus delaying the identification of what the shellcode in question may be doing.

By hindering knowledge of what the malware does, the analyst also cannot proceed to look for other indicators produced from the use of APIs, such as looking for suspicious files when seeing the use of "CreateFileW".

API Hashing Techniques for 64-bit Shellcode

API Hash Resolution Process

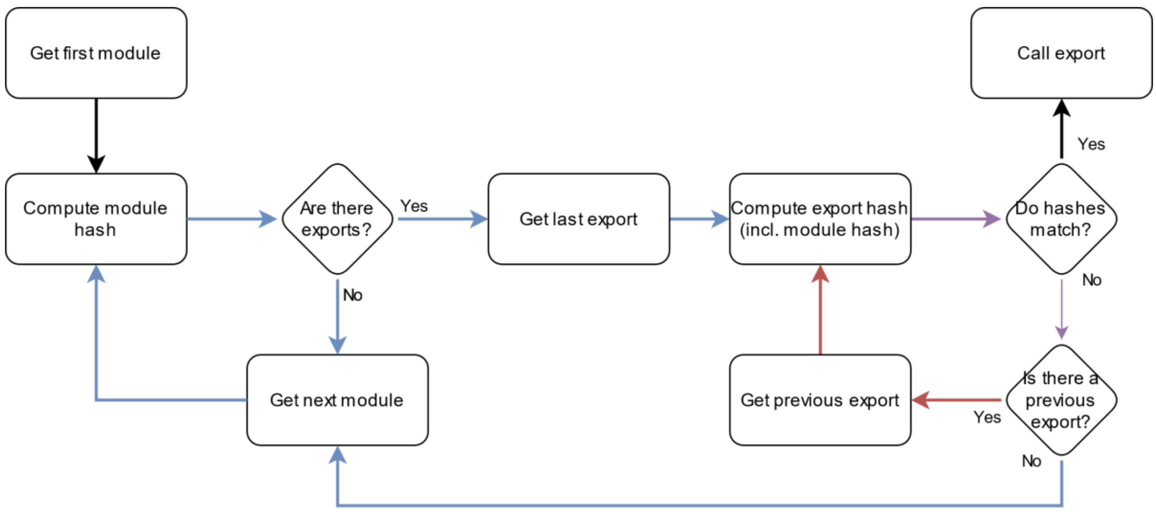
Hash resolution logic is usually similar to the following:

1. Locate the DLL specified
2. Loop through each export name of specified DLL
    a. Hash the export name
    b. Compare the hash with the specified hash
3. Call the export function with the address of the matching hashed name

For locating the DLL specified, there are a few fixed offsets from the Process Environment Block (PEB) that when followed would lead to LoadLibraryA and GetProcAddress *(as detailed in Appendix C)*.

These may then be used to load the required DLL modules and retrieve the address of a specific API function after identifying the required function name from the hash.

The iteration through the export address table of the DLL module for determining the right hash usually looks like the following:



Well-illustrated Import Resolution Flow Diagram by NVISO Labs

Usually, a custom version of "GetProcAddress" is created and made to first resolve the hash before calling the API right after as well, and this can be seen in examples such as:

```
;============================================================================
; Get function address in specific module
;
; Arguments: r15 = module pointer
;            edi = hash of target function name
; Return: eax = offset
;============================================================================
get_proc_addr:
    ; Save registers
    push rbx
    push rcx
    push rsi                ; for using calc_hash

    ; use rax to find EAT
    mov eax, dword [r15+60]  ; Get PE header e_lfanew
    mov eax, dword [r15+rax+136] ; Get export tables RVA

    add rax, r15
    push rax                 ; save EAT

    mov ecx, dword [rax+24]  ; NumberOfFunctions
    mov ebx, dword [rax+32]  ; FunctionNames
    add rbx, r15

_get_proc_addr_get_next_func:
    ; When we reach the start of the EAT (we search backwards), we hang or crash
    dec ecx                     ; decrement NumberOfFunctions
    mov esi, dword [rbx+rcx*4]  ; Get rva of next module name
    add rsi, r15                ; Add the modules base address

    call calc_hash

    cmp eax, edi                        ; Compare the hashes
    jnz _get_proc_addr_get_next_func    ; try the next function

_get_proc_addr_finish:
    pop rax                     ; restore EAT
    mov ebx, dword [rax+36]
    add rbx, r15                ; ordinate table virtual address
    mov cx, word [rbx+rcx*2]    ; desired functions ordinal
    mov ebx, dword [rax+28]     ; Get the function addresses table rva
    add rbx, r15                ; Add the modules base address
    mov eax, dword [rbx+rcx*4]  ; Get the desired functions RVA
```

Taken from eternalblue_kshellcode_x64.asm

```
;Hashing section to resolve a function address
GetProcessAddress:
        mov r13, rcx                     ;base address of dll loaded
        mov eax, [r13d + 0x3c]           ;skip DOS header and go to PE header
        mov r14d, [r13d + eax + 0x88]    ;0x88 offset from the PE header is the exp
ort table.

        add r14d, r13d                  ;make the export table an absolute base add
ress and put it in r14d.
        mov r10d, [r14d + 0x18]         ;go into the export table and get the numbe
rOfNames
        mov ebx, [r14d + 0x20]          ;get the AddressOfNames offset.
        add ebx, r13d                   ;AddressofNames base.

find_function_loop:
        jecxz find_function_finished   ;if ecx is zero, quit :( nothing found.
        dec r10d                       ;dec ECX by one for the loop until a match/n
one are found
        mov esi, [ebx + r10d * 4]      ;get a name to play with from the export tab
le.
        add esi, r13d                  ;esi is now the current name to search on.

find_hashes:
        xor edi, edi
        xor eax, eax
        cld

continue_hashing:
        lodsb                         ;get into al from esi
        test al, al                   ;is the end of string resarched?
        jz compute_hash_finished
        ror dword edi, 0xd            ;ROR13 for hash calculation!
        add edi, eax
        jmp continue_hashing
```

Taken from Topher Timzen's Windows x64 Shellcode Writing Tutorial

Variations of Hashing Algorithms

The following are the common methods used in hashing algorithms, but they are often mixed and matched as seen in the various implementations listed in tools such as HashDB and FLARE shellcode_hashes *(Appendix B)*.

RORXX

Most common method in hashing algorithms, and rotates the hash XX bits to the right.

Usually implemented as ROR13 in the shellcode of popular tools such as Msfvenom and Cobalt Strike.

However, by simply changing the number of bits in which each character is rotated from 13 to another number, it would render the shellcode to become undetectable by multiple antiviruses. (Hence the APIHashReplace tool by Huntress)

Due to how common ROR is as hashing algorithm in shellcode, some have even created complete spreadsheets of the hash values for each Windows API (varying no. of bits):

The Massive 2 MB Shellcode API Hash List - Compiled by Alexander Hanel

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	
1	.dll names start with #	Hash																	
2	API Name	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF	0x10	0x
3	#advapi32.dll																		
4	A_SHAFinal	0x00000	0xcd800	0x207b4	0x1176ff	0x94da5	0xe55e8	0x201a8	0xf5e52c	0xe3b6b	0xf1c52e	0xaab63	0xb31a9	0xe26ba	0xb52e8	0xe027c	0xb7aa7	0x019f01	0
5	A_SHAInit	0x00000	0xbb000	0x9ded0	0x0137f	0x02d5d	0x85163	0xf2f382	0xe8aa0	0xb1c1a	0x87dcd	0xd799e	0x542a8	0xe9d76	0xd7983	0xf42a7	0x3b9a5	0x01590	0
6	A_SHAUpdate	0x00000	0x8ec00	0x2aded	0x3dbce	0xd8b59	0x0f176	0xf87df1	0x9deb7	0x28e3a	0x8ff0e1	0xbd6e6	0x38c68	0x8ea52	0xdfae1	0x73378	0xf0a97	0x01d40	0
7	AbortSystemShutdownA	0x00000	0xe3362	0x1a66f	0x15aa6	0xf126a	0xc01c3	0x60fa6c	0x156df1	0x39080	0xc5bc3	0xd2bff9	0xd25bf4	0x1d860	0x4ca9a	0xa93dc	0xe2ebff	0x04370	0
8	AbortSystemShutdownW	0x00000	0xe3362	0x1a66f	0x15aa6	0xf126a	0xc01c3	0x60fa6e	0x156df1	0x39080	0xc5bc3	0xd2bff9	0xd25bf4	0x1d860	0x4ca9a	0xa93dc	0xe2ebff	0x04370	0
9	AccessCheck	0x00000	0xc2c00	0x1903d	0x2d365	0x0ccaa	0x383ca	0xdc0ed	0x488a2	0x3a19c	0xc864a	0x00463	0x34bed	0xffa9fe	0xf7eb6e	0x2c067	0x0ae78	0x02060	0
10	AccessCheckAndAuditAlarmA	0x00000	0x515f0	0xf9e867	0x392e0	0xaadcc	0xa9e2e	0x72378	0x72496	0x333a6	0x05559	0x9d5bf	0x48e85	0xd800f	0xba44f	0xf3313	0x4d067	0x04980	0
11	AccessCheckAndAuditAlarmW	0x00000	0x515f0	0xf9e867	0x392e0	0xaadcc	0xa9e2e	0x72378	0x72496	0x333a6	0x05559	0x9d5bf	0x48e85	0xd800f	0xba44f	0xf3313	0x4d067	0x04980	0
12	AccessCheckByType	0x00000	0x970b0	0x72419	0xac4b4	0x49674	0x7fe26	0xc5666	0x3e38c	0x808b8	0x5cd83	0x997b3	0x22c22	0x4c792	0x65df8e	0x8676a	0xc5a28	0x030c0	0
13	AccessCheckByTypeAndAuditAlarmA	0x00000	0x51725	0x8dc46	0xf28dcf	0x477e1	0xa3017	0xeb5ec	0x2fb50e	0xf420b	0x5723b	0x99a58	0xc9b8e	0x5b4dc	0x8ac46	0x38d83	0x0c015	0x059e0	0
14	AccessCheckByTypeAndAuditAlarmW	0x00000	0x51725	0x8dc46	0xf28dcf	0x477e1	0xa3017	0xeb5ec	0x2fb50e	0xf420b	0x5723b	0x99a58	0xc9b8e	0x5b4dc	0x8ac46	0x38d83	0x0c015	0x059e0	0
15	AccessCheckByTypeResultList	0x00000	0x5ca5c	0x31a1f7	0xe3667	0x3784a	0x1290d	0x96303	0xa6193	0xc0d74	0x1831a	0xb6de6	0x193f6c	0xbaf32	0x1bedd	0xc0a13	0x785df	0x04fc05	0
16	AccessCheckByTypeResultListAndAuditAlarmA	0x00000	0x5ca57	0x83cad	0xb59b9	0x64d6fc	0xd94c3	0x88891	0x49ad2	0xa601f	0x44897	0x86776	0x690c1	0xbdbc3	0xc3fe39	0x1c7ae	0xe9364	0x078e0	0
17	AccessCheckByTypeResultListAndAuditAlarmByHandleA	0x00001	0x3e5ca	0xf71d8	0x254d9	0x278b5	0x68a85	0x57f4f8	0xab17e	0x6cb6c	0x81e8b	0xcda18	0x235d9	0x4c824	0xb0130	0xc4281	0x82d60	0x09310	0

Taken from https://spreadsheets.google.com/spreadsheet/pub?key=0AsJdvp2NWuJDdHdiYmRUTzNJaHctZEVaZXZndjNCTlE&output=csv

(Small Diagram on the rorXXAddHash32 algorithm in Appendix B)

ADD

The part of the hashing algorithm that allows the resulting output to be considered a hash (since the sum of letters is not easily reversible into specific letters).

Adds the value of each character in the string to the hash value, resulting in a unique value for different combinations of letters.

If there is a NULL character added to the end of the string, it allows the rest of the hashing algorithm to be performed without the *add* portion of the hashing algorithm (as it would simply *add* null)

XOR

Another common method used in hashing algorithms, and may involve:

- a specific hardcoded hash in the algorithm (e.g. *playWith0xe8677835Hash*),
- a changing key (e.g. *rol3XorEax* or *chAddRol8Hash32*),
- or just the next letter that is part of the hash (e.g.*rol*N*XorHash32*).

OTHER COMMON OPERATIONS

- SHR and SHL (e.g. *shl7Shr19*XXX*Hash32*, *shr2Shl5XorHash32*)
- ROL (e.g. *rol*NXXX*Hash32*, chAddRol8Hash32)
- IMUL (e.g. *imul83hAdd*, *imul21hAddHash32*)
- OR (e.g. *or21hXorRor11Hash32*, *or23hXorRor17Hash32*)

MINOR VARIATIONS

UPPERCASE/LOWERCASE

Multiple methods can be used to convert uppercase characters to ***lowercase***, from ***adding 20h/32*** to performing ***OR 32***, but this of course causes the hash to change.

For converting lowercase characters to ***uppercase***, it may either ***subtract 20h/32*** or perform an ***AND 0xFFFFFFDF***.

FINAL ROUNDS

Final rounds of hashing without regard to any more characters may be performed (usually with position-relevant operations such as ror/rol/shr/shl).

OTHER NAMED HASHING ALGORITHMS

- **ZLIB**
  - CRC32/64
  - ADLER32
- **DJB2** (Nokoyawa ransomware, GuLoader Downloader Shellcode)
- **FNV** (Kazuar Backdoor Shellcode)
- **CARBANAK**
- **conti, conti_e9ff0077, conti_mm3** (Conti Ransomware)
- **poisonIvyHash** (POISON IVY RAT)
- **revil_010F** (REvil / Sodinokibi Ransomware)
- **zloader_bot** (zloader bot)
- **add_hiword_add_lowword** (Darkside ransomware)

Similar malware families tend to use extremely similar hashing logic to calculate and resolve API hashes.

For example, Cobalt Strike and Msfvenom will use the hash 0x726774c when resolving “LoadLibraryA” as both uses the ROR13 hashing algorithm by default, as Cobalt Strike’s payload shellcodes are based on Meterpreter shellcodes:

```
loop_funcname:
  xor rax, rax
  lodsb
  ror r9d, 0xd
  add r9d, eax
  cmp al, ah
  jne loop_funcname
  add r9, [rsp+0x8]
  cmp r9d, r10d
  jnz get_next_func
  ; If found, fix up stack, c
  pop rax
  mov r8d, dword [rax+0x24]
  add r8, rdx
  mov cx, [r8+0x2*rcx]
  mov r8d, dword [rax+0x1c]
  add r8, rdx
  mov eax, dword [r8+0x4*rcx]
  add rax, rdx
  ; We now fix up the stack a
```

Metasploit Framework x64 block_api.asm

```
def ROR(data, bits):
    return (data >> bits | data << (32 - bits)) & 0xFFFFFFFF

def hash_api(dll_name, api_name):
    # normalize api name
    api = bytes(api_name,'utf-8') + b'\x00'
    # normalize dll name
    dll = dll_name.upper().encode('utf-16')[2:] + b'\x00\x00'
    # compute api hash
    api_hash = 0
    for i in range(len(api)):
        api_hash = ROR(api_hash,0x0d) + api[i]
    # compute dll hash
    dll_hash = 0
    for i in range(len(dll)):
        dll_hash = ROR(dll_hash,0x0d) + dll[i]
    # compute final hash
    final_hash = (api_hash + dll_hash) & 0xFFFFFFFF
    print('0x%08x,%s_%s' % (final_hash, dll_name, api_name))
```

Cobalt Strike Hashing Algorithm implemented in Python

Difference in Hashing Algorithms for 64-bit Shellcode

Due to the specificity of the topic of this literature survey, I had wondered if there would be a difference in the hashing algorithms used in 32-bit shellcode and 64-bit shellcode.

So far, for most of the few x64 shellcodes I was able to see online, most of them had implemented existing hashing algorithms that generated 32-bit hashes for comparison and hash resolution.

I have only stumbled upon one x64 shellcode (by chance) that generated and compared against 64-bit hashes using a unique hashing algorithm that I had not encountered on either HashDB or Mandiant FLARE’s shellcode_hashes.

However, in general, 32-bit hashes are still commonly used, and 32-bit hashing algorithms can still be smoothly used in 64-bit shellcode.

Detection of API Hashing

Antivirus (AV) products often have detection capabilities for common artefacts left behind by default configurations in the API hashing of popular malware.

However, it was found that if a threat actor were to make even minor changes to such defaults in the API hashing, it may still pass through the AV undetected, resulting in a simple malware to begin being on par with even one of FUD status.

In the case of malware shellcode generated by Metasploit and Cobalt Strike, Huntress has found that while many AV vendors depended on YARA rules for ROR**13** API hashes in their shellcode detection capabilities.

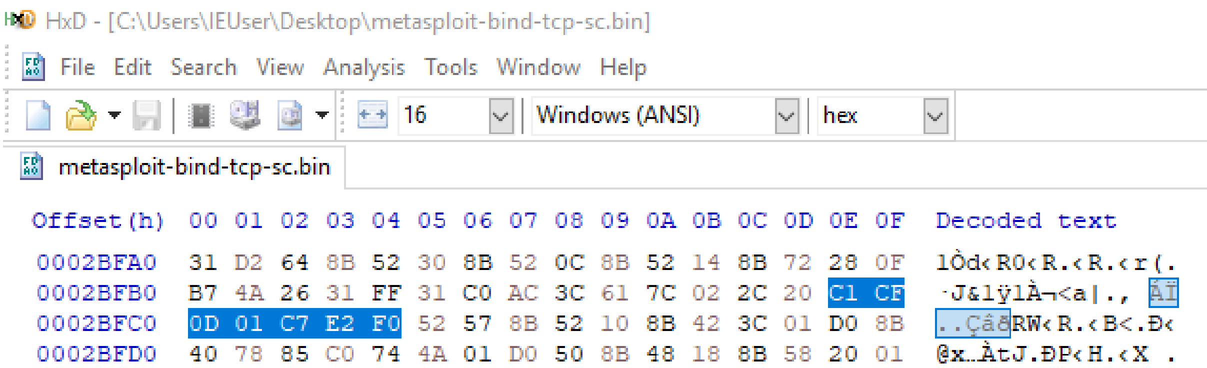
Any change in the number of bits rotated to the right from 0xd/13 to another number would cause a significant reduction in the number of vendor detections (from 17 vendors to only 2 vendors in the article after changing to 0xf/15).

However, changes can be made to such YARA rules to ensure a more comprehensive coverage of API hashing despite slight modifications to them.

This includes detecting the general patterns of a ror hashing sequence as opposed to specific ror13 hashing sequences:

64-bit rorXX sequence: `{ 41 c1 c9 ?? 41 01 c1 e2 ?? }`
32-bit rorXX sequence: `{ c1 cf ?? 01 c7 e2 ?? }`

Example of x64 Metasploit shellcode using 32-bit ROR13 hashing algorithm:



Examples for 64-bit Shellcode

Here were the few examples of x64 shellcodes using API Hashing found online. I used Mandiant FLARE's shellcode_hashes and OALabs HashDB to identify many of the hashes and their algorithms.

EternalBlue x64 Shellcode

Hashing Algorithms Used:
`ror13AddHash32` (MS17-010 GitHub) & **`ror13AddHash32AddDll`** (Metasploit Exploit)

In the official MS17-010 GitHub Repository detailing the vulnerability, a shellcode called 'eternalblue_kshellcode_x64.asm' is provided, and implements API hashing in order to find its desired function addresses in a custom 'GetProcAddress':

```
;================================================
; Get function address in specific module
;
; Arguments: r15 = module pointer
;            edi = hash of target function name
; Return: eax = offset
;================================================
get_proc_addr:
    ; Save registers
    push rbx
    push rcx
    push rsi                ; for using calc_hash

    ; use rax to find EAT
    mov eax, dword [r15+60]  ; Get PE header e_lf
    mov eax, dword [r15+rax+136] ; Get export tab

    add rax, r15
    push rax                 ; save EAT

    mov ecx, dword [rax+24]  ; NumberOfFunctions
    mov ebx, dword [rax+32]  ; FunctionNames
    add rbx, r15

_get_proc_addr_get_next_func:
    ; When we reach the start of the EAT (we sear
    dec ecx                     ; decrement Numbe
    mov esi, dword [rbx+rcx*4]  ; Get rva of next
    add rsi, r15                ; Add the modules

    call calc_hash

    cmp eax, edi                        ; Compare
    jnz _get_proc_addr_get_next_func    ; try the
```

It rotate bits to the right by 13 bits and adds the next character

`(ror13AddHash32)`

(Refer to Appendix B for more details on algorithm)

```
;==================================================================
; Calculate ASCII string hash. Useful for comparing ASCII string
;
; Argument: rsi = string to hash
; Clobber: rsi
; Return: eax = hash
;==================================================================
calc_hash:
    push rdx
    xor eax, eax
    cdq
_calc_hash_loop:
    lodsb                   ; Read in the next byte of the ASCII
    ror edx, 13             ; Rotate right our hash value
    add edx, eax            ; Add the next byte of the string
    test eax, eax           ; Stop when found NULL
    jne _calc_hash_loop
    xchg edx, eax
    pop rdx
    ret
```

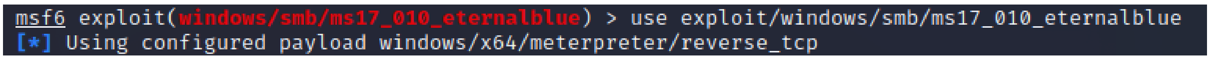
The WinAPI functions required by the shellcode include the following:

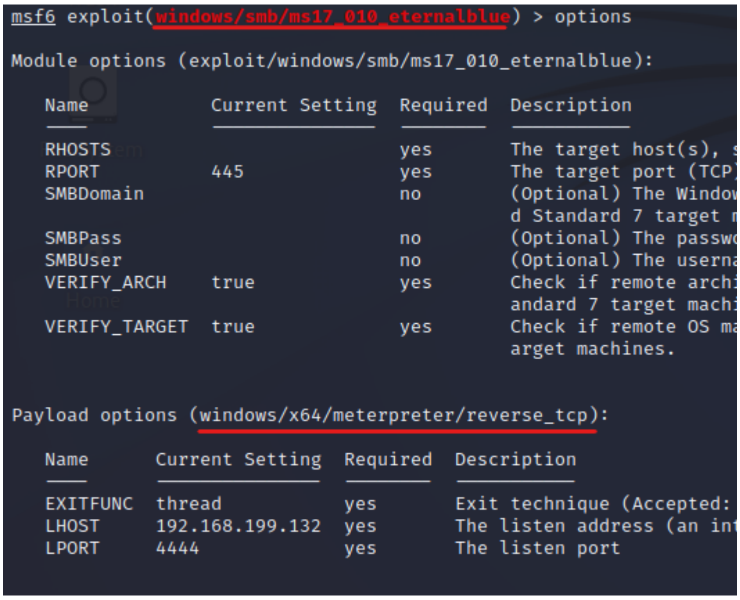
```
PSGETCURRENTPROCESS_HASH    EQU    0xdbf47c78
PSGETPROCESSID_HASH    EQU    0x170114e1
PSGETPROCESSIMAGEFILENAME_HASH    EQU    0x77645f3f
LSASS_EXE_HASH    EQU    0xc1fa6a5a
SPOOLSV_EXE_HASH    EQU    0x3ee083d8
ZWALLOCATEVIRTUALMEMORY_HASH    EQU    0x576e99ea
PSGETTHREADTEB_HASH    EQU    0xcef84c3e
KEINITIALIZEAPC_HASH    EQU    0x6d195cc4
KEINSERTQUEUEAPC_HASH    EQU    0xafcc4634
PSGETPROCESSPEB_HASH    EQU    0xb818b848
CREATETHREAD_HASH    EQU    0x835e515e
```

Shellcode GitHub Links:
https://github.com/worawit/MS17-010/blob/master/shellcode/eternalblue_kshellcode_x64.asm
https://gist.github.com/worawit/05105fce9e126ac9c85325f0b05d6501

However, in Metasploit's EternalBlue Exploit, a different hashing algorithm is used for API hashing in the shellcode generated.

When we use the payload `windows/smb/ms17_010_eternalblue` in Metasploit, we can see a message that says its payload will be defaulted to `windows/x64/meterpreter/reverse_tcp`.





This is also visible in the payload options of the exploit:

`windows/x64/meterpreter/reverse_tcp`

I loaded the shellcode into IDAFree to see the disassembly and found the following:

Before that I generated the shellcode using the 'generate' command and outputted it into a file before cleaning the file up and creating a bin file using HxD and dropping it into IDA.

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

                                ; C
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]
```

Two ROR instructions, one closer to the top and one further on.

```
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
```

It was not hard to find the first `ror` instruction performed before getting the base address of kernel32.dll `[rdx+20h]` , and the subsequent process of obtaining the PE Header `[rdx+3Ch]`, the export table `[rax+88h]` and function names `[rax+20h]`, meaning that this was for the hashing of the DLL name

Of course, there was the second `ror` function further on in the disassembly at the usual location where the function names are compared `(cmp r9d, r10d)`, and the ordinal/index of the function `[r8+rcx*2]` and the offset of the function address `[r8+rcx*4]` are both taken in.

However, before the comparison, there is an adding of a value on the stack `[rsp+8]` to the hash, which was odd for a typical `ror13AddHash32`.

I wondered if it was also adding the hash of the dll name as seen in one of the hashing algorithms prior to trying this out, so I found the hashes further on in the disassembly:

```
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, 0E0DF0FEA
call    rbp
```

They are all loaded into `r10d`, and `rbp` is called containing the hash-calculating function with `ror` above

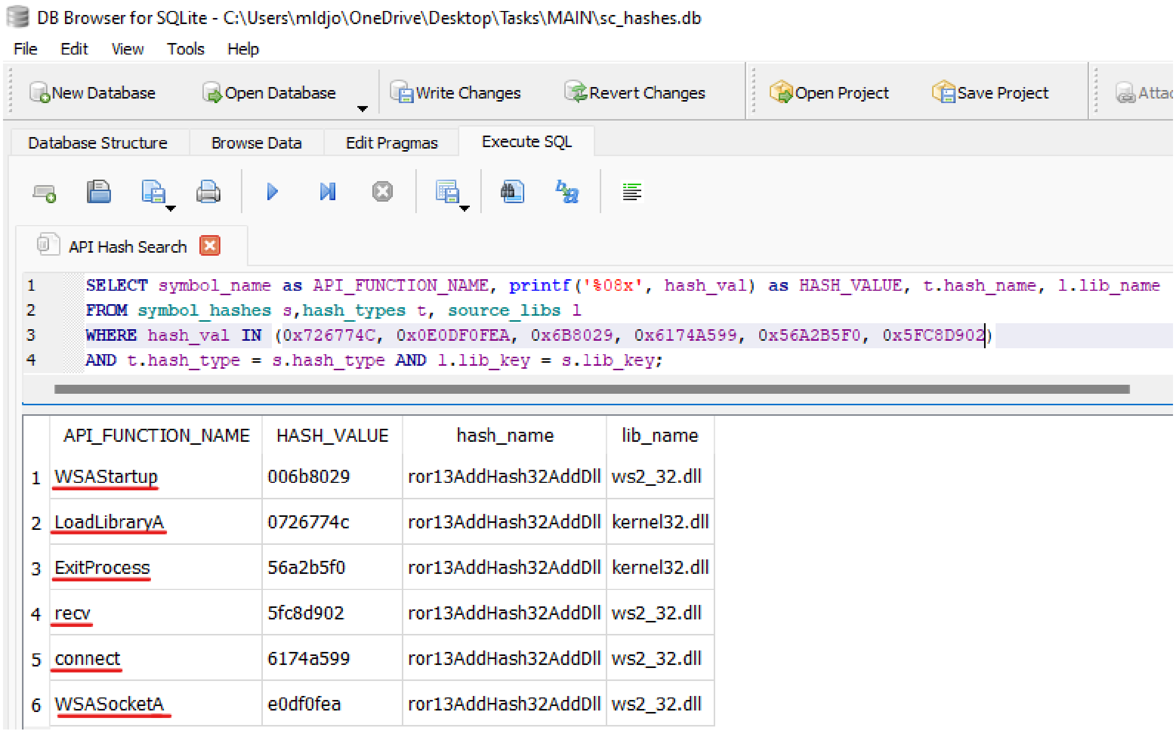
```
pop     r8
mov     rdx, r12
mov     rcx, rdi
mov     r10d, 6174A599h
call    rbp
test    eax, eax
jz      short loc_160
dec     r14
jnz     short loc_13E
push    56A2B5F0h
call    rbp

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

In order to confirm whether the hashing algorithm I had thought of was correct, I entered these hashes into DB Browser to see if FLARE's shellcode_hashes had them.

SQLite query I used to find the hashes:

```sql
SELECT symbol_name as API_FUNCTION_NAME, printf('%08x', hash_val) as
HASH_VALUE, t.hash_name, l.lib_name
FROM symbol_hashes s,hash_types t, source_libs l
WHERE hash_val IN (0x534C0AB8)
AND t.hash_type = s.hash_type AND l.lib_key = s.lib_key;
```

As it had turned out, the hashing algorithm used was indeed `ror13AddHash32`**`AddDll`**, which factored in the name of the DLL name when calculating the hash.



The following API functions were called:
`LoadLibraryA, ExitProcess, recv, connect, WSAStartup and WSASocketA`

Much later, I found that the shellcode generator could be found online, and had comments that confirmed the hashing algorithm (`ror13AddHash32`**`AddDll`**):

```
add r9, [rsp+0x8]          ; Add the current module hash to the function hash
cmp r9d, r10d              ; Compare the hash to the one we are searchnig for
```

Taken from:
https://github.com/rapid7/metasploit-framework/blob/04e8752b9b74cbaad7cb0ea6129c90e3172580a2/external/source/shellcode/windows/x64/src/block/block_api.asm

Interestingly, not all Metasploit-generated shellcodes share the same hashing algorithm, and although they all generally use `ror13AddHash32`, there is that slight variation in the one for `windows/x64/meterpreter/reverse_tcp` causing it to include the name of the DLL.

We will cover the hashing algorithm used in other shellcodes generated by Metasploit's Meterpreter in the next section.

Cobalt Strike / Meterpreter Shellcode

Hashing Algorithms Used:
ror13AddHash32 (Function Names) & **ror13AddHash32Sub20h** (DLL Names)

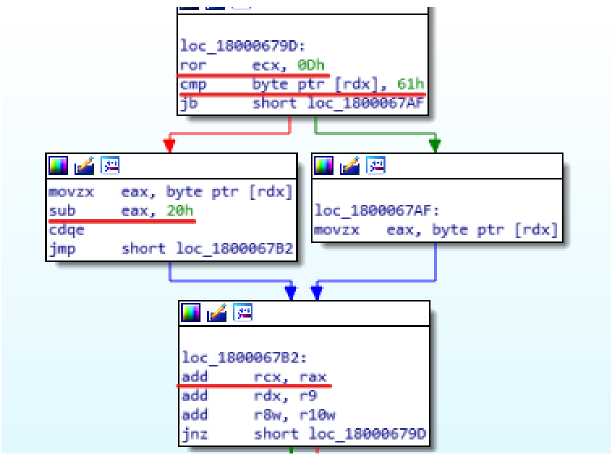
Although Cobalt Strike and Metasploit are independent of each other, Cobalt Strike's payloads are based on Meterpreter shellcodes, with one of its similarities including its API Hashing.

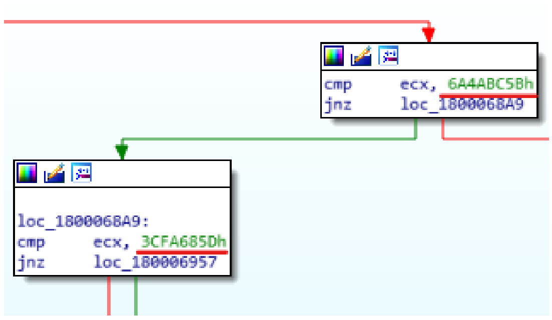
For the purpose of seeing what hashing algorithms are used, I attempted to disassemble 3 shellcodes from Metasploit belonging to Meterpreter:

```
payload/windows/x64/meterpreter_bind_tcp
payload/windows/x64/meterpreter_reverse_http
payload/windows/meterpreter_reverse_https
```

The hash resolution process for all 3 had followed the same procedures and used the same hashing algorithm detailed below:

Calculating the hash for the DLL name:



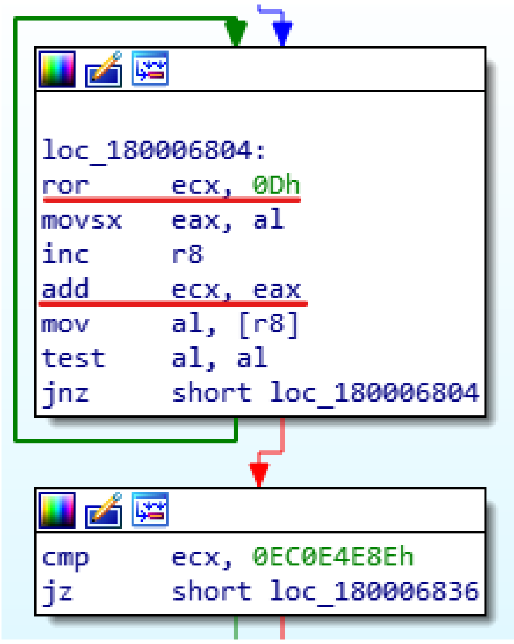


The DLL hashes
used in all 3 shellcodes

As seen above, the hashing algorithm is almost similar to the one we found for the DLLs in the EternalBlue shellcode (`ror13AddHash32`**`Sub20h`**).

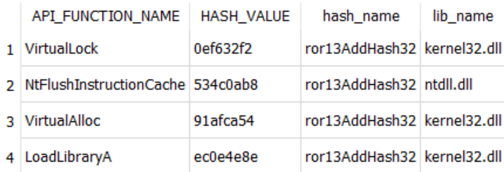
The only ultimate difference is that the DLL hash is NOT added to the function hash as it did previously for the functions in the EternalBlue Shellcode (`ror13AddHash32`**`AddDll`**)

Calculating the hash for the function name:



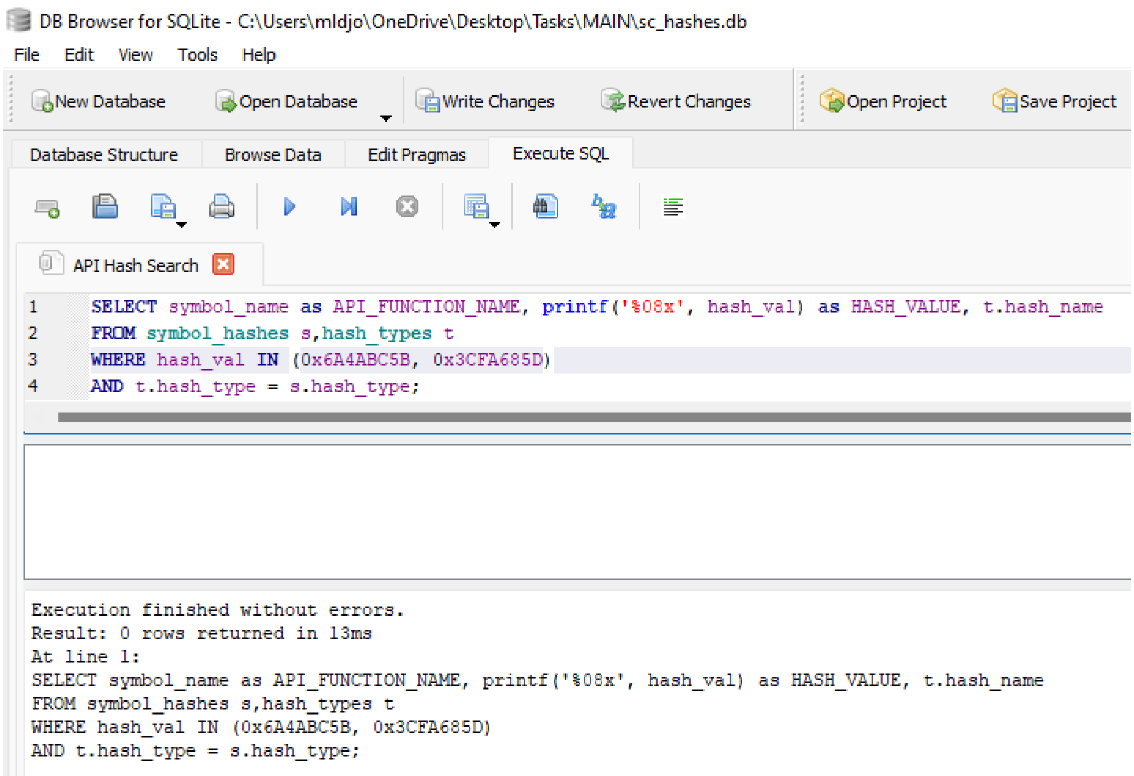
While the EternalBlue Shellcode used ror13AddHash32**AddDll** for its functions, the 3 other Meterpreter shellcodes used a simple **ror13AddHash32** hashing algorithm

Some of the function calls included:



	API_FUNCTION_NAME	HASH_VALUE	hash_name	lib_name
1	VirtualLock	0ef632f2	ror13AddHash32	kernel32.dll
2	NtFlushInstructionCache	534c0ab8	ror13AddHash32	ntdll.dll
3	VirtualAlloc	91afca54	ror13AddHash32	kernel32.dll
4	LoadLibraryA	ec0e4e8e	ror13AddHash32	kernel32.dll

When double-checking with the DLL name hashes, I found that searching through FLARE's shellcode_hashes yielded no results:

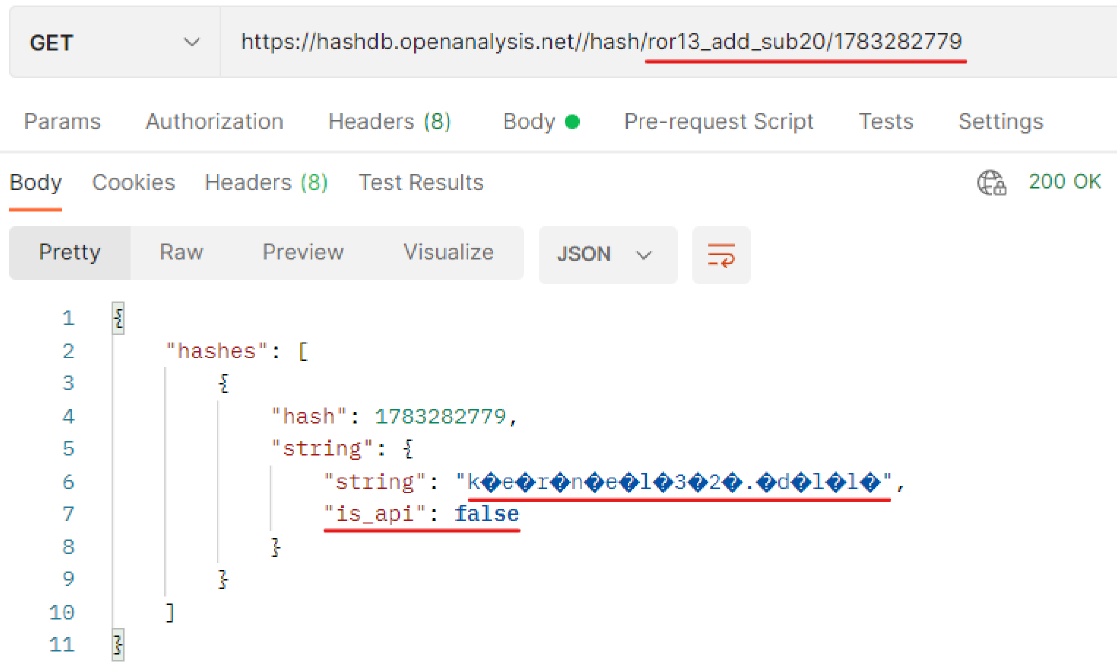


It would appear that the **ror13AddHash32Sub20h** hashing algorithm used in metasploit does not match the one used in FLARE's shellcode_hashes.

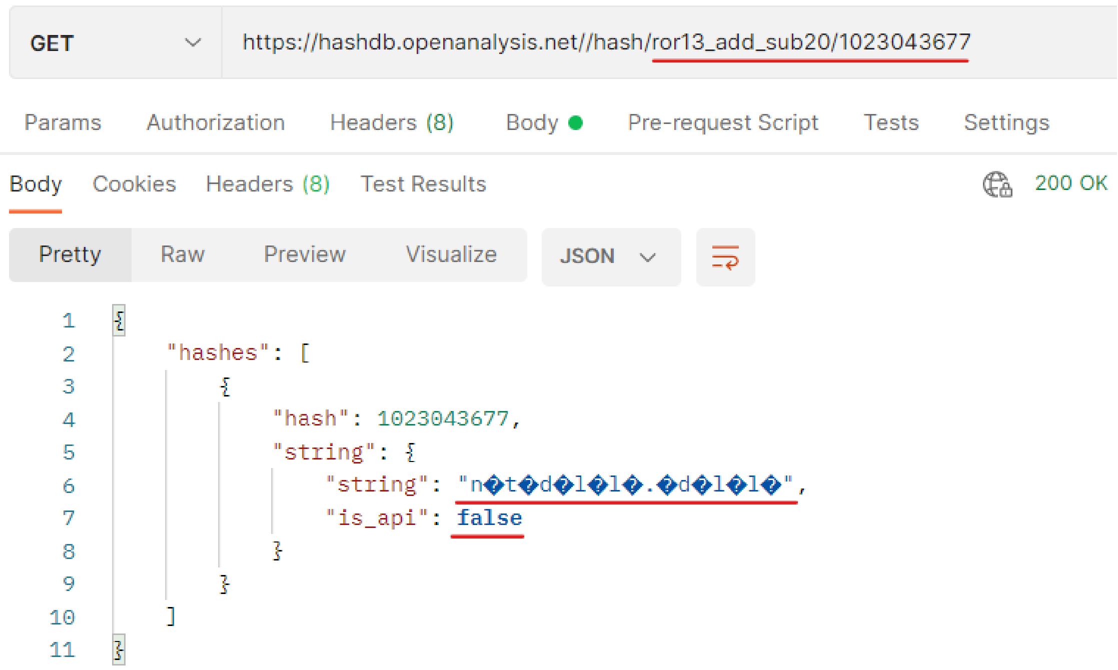
This is when I tried using OALabs HashDB, which required me to know the hashing algorithm and have the decimal version of the hash, but helped yield better results:

```
0x6A4ABC5B => 1783282779
0x3CFA685D => 1023043677
```

Found kernel32.dll:



Found ntdll.dll:



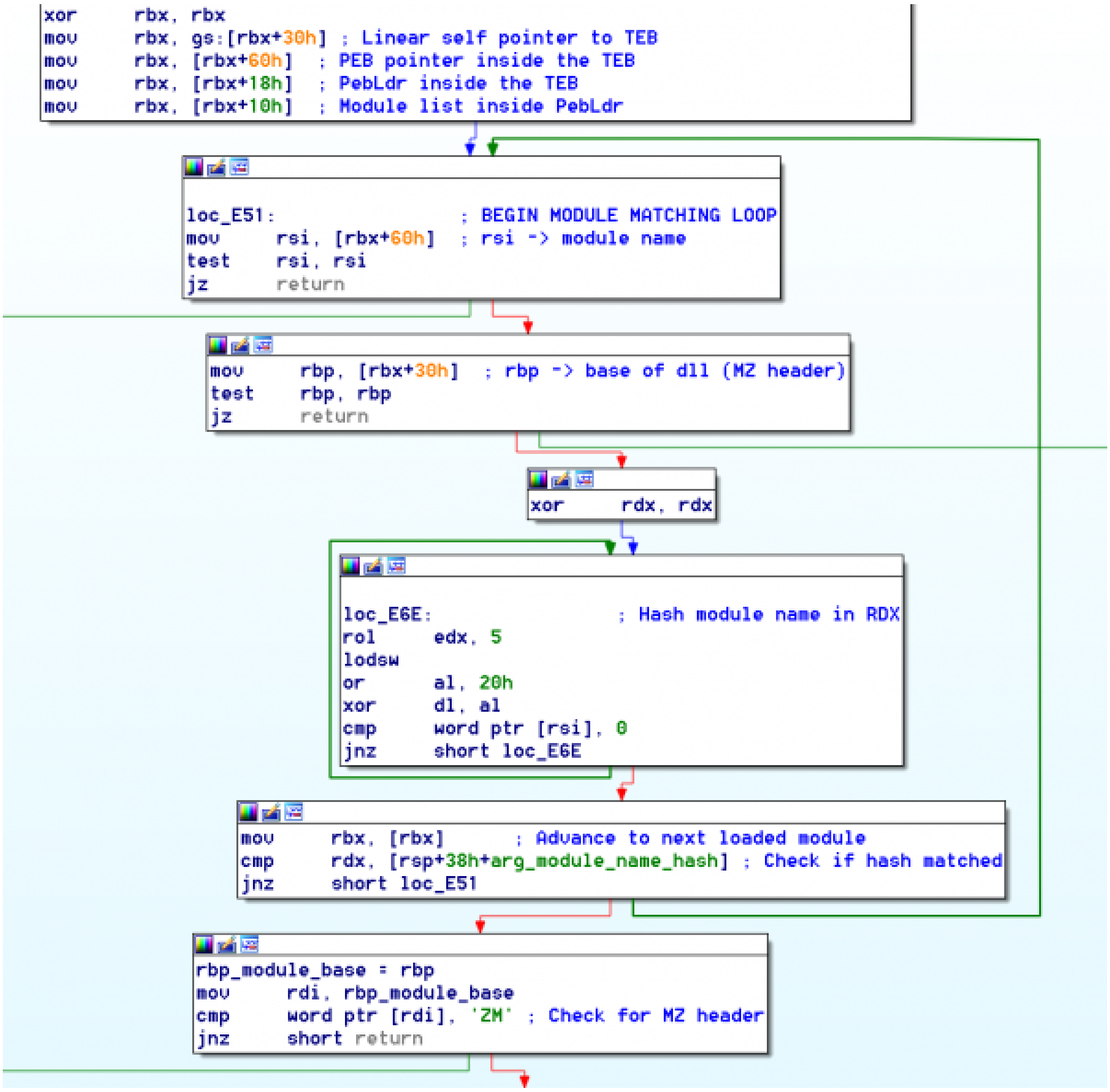
HashDB can be more effective when used as a plugin in IDA Pro but since I didn't have IDA Pro and it would not work with IDA Free, I had settled with this method for now.

DoublePulsar Backdoor Shellcode

Hashing Algorithm Used: `rol5XorHash32`

For the user mode DoublePulsar Backdoor Shellcode, there had been an article detailing its analysis and had included a screenshot of the usual Window shellcode procedure finding the PEB all the way to the API hashing algorithm it uses.

As we can see in the screenshot below, the hash is rotated by 5 bits to the left and XOR-ed with the uppercase version of each letter to produce the final hash, thus implementing the `rol5XorHash32` hashing algorithm.



FIN8's BADHATCH Shellcode

Hashing Algorithm Used:
`shl7Shr19XorHash32` (Gigamon Article) AND **`unknown algorithm`** (Malware Bazaar)

Looking through this shellcode was the first time I had seen a 64-bit hashing algorithm that generated 64-bit hashes. Other x64 shellcodes had all still implemented 32-bit hashing algorithms and used 32-bit hashes as well.

Even for HashDB, hashing algorithms made specifically for 64-bit shellcode was a rare sight, let alone Mandiant FLARE's shellcode_hashes that seemed to have *only* 32-bit hashing algorithms.

However, I started first with finding the article for it:
https://blog.gigamon.com/2019/07/23/abadbabe-8badf00d-discovering-badhatch-and-a-detailed-look-at-fin8s-tooling/

The article had stated that the "*64-bit shellcode*" of the first stage of the malware downloader uses "*the **Carberp function hash resolution routine** to hide the names of API functions being used*", which was an API hashing algorithm I had not heard of yet.

Upon searching it up, I found a few sources that would show that the Carberp Trojan uses the `shl7Shr19XorHash32` hashing algorithm, as seen in the following sources:

Virus Bulletin: Carberp

```
DWORD HashString(char *pszApiName)
{
  DWORD retval = 0;
  char byte;
  char *copy = pszApiName;
  if(pszApiName == NULL)
  {
     return -1;
  }
  byte = pszApiName[0];
  while(byte != NULL)
  {
     retval = (retval << 7) | (retval >> 0x19);
     unsigned char key = copy[0];
     retval = key ^ retval;
     copy++;
     byte = copy[0];
  }
  return retval;
}
```

Carberp Source on GitHub

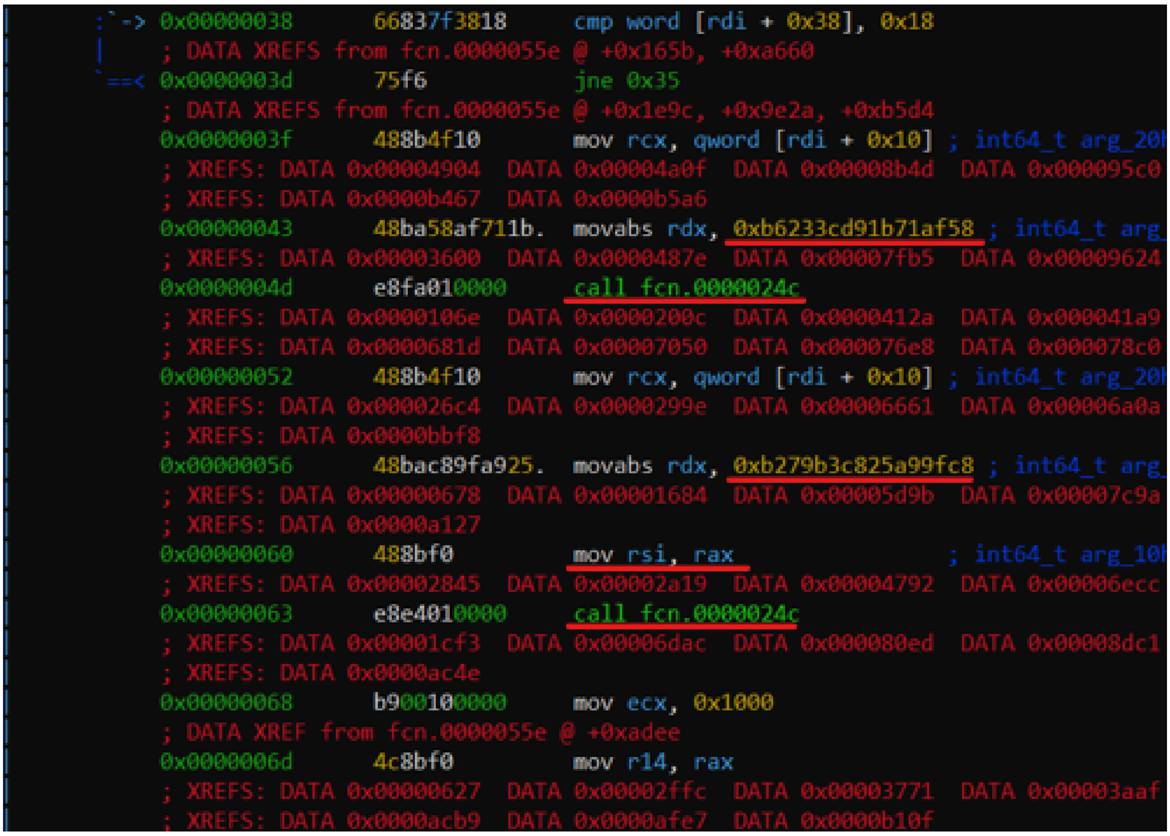
```
DWORD CalcHashW( PWSTR str )
{
	if ( !str )
	{
		return 0;
	}

	DWORD hash = 0;
	PWSTR s = str;

	while (*s)
	{
		hash = ((hash << 7) & (DWORD)-1) | (hash >> (32 - 7));
		hash = hash ^ *s;
		s++;
	}

	return hash;
}
```

However, I had later found that the x64 shellcode was also available on Malware Bazaar, but realised it was of a different hash:

Malware Bazaar SHA256 Hash (BADHATCH):
`77f5e4a86ba682c490a53ee2b170e74692d6539bde00971f4d6dd2b90e557340`

Gigamon Blog SHA256 Hash (BADHATCH):
`c5642641064afc79402614cb916a1e3bd5ddd4932779709e38db64d6cc561cd5`

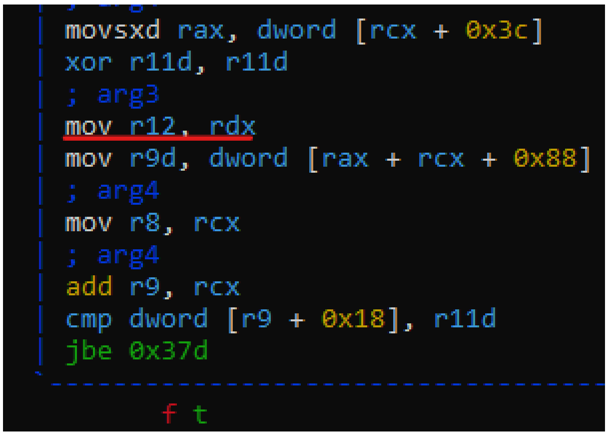
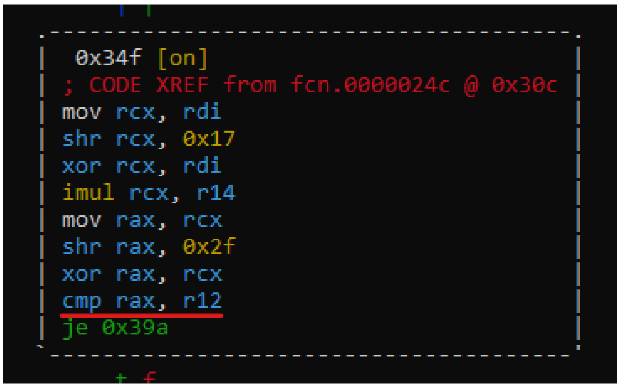
Upon opening it up in radare2, printing the disassembly, and looking through it:



The first most noticeable features after the usual shellcode procedures (finding the number of function names and their addresses) were the 64-bit hashes that were being loaded before a function was called.

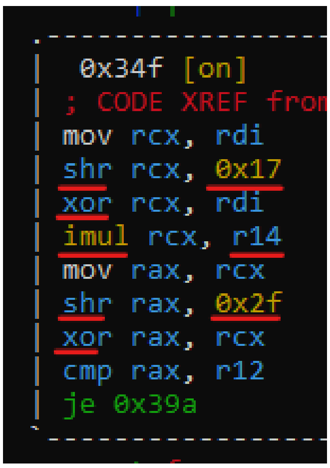
The function **fcn.0000024c** seemed to be the function calculating the hash and verifying them against the loaded hashes in rdx, and it seemed even more so because rsi (where the output of fcn.0000024c is moved into) is later called:

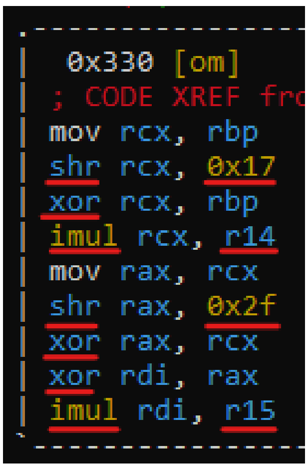
So I printed the disassembled fcn.0000024c, and found the following that helped confirmed it was the hash calculation function:

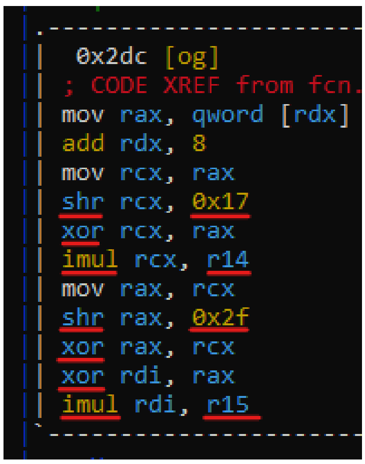
	rdx, where the hash was loaded into before the calling of the function, was loaded into r12
	There is a final comparison with r12 before the function ends or loops back to the hashing algorithm.

Now all that was left was to attempt to understand the hashing algorithm. Unfortunately, other than seeing a general pattern of what the hashing algorithm did, I was unable to fully confirm what it was doing due to the lack of remaining time.

The following operations were repeated multiple times, but it seemed that different conditions were checked before going through each of them:







r14 and r15 were other long hardcoded hashes that were XOR-ed with the hash, and there had been another hardcoded hash that was factored into the hashing algorithm:

```
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
```

I have tried to search for a similar hashing algorithm on both HashDB and FLARE's shellcode_hashes, but it anything similar to the following did not exist:

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

x64 BADHATCH Shellcode Taken From:
https://bazaar.abuse.ch/sample/77f5e4a86ba682c490a53ee2b170e74692d6539bde00971f4d6dd2b90e557340/

BlackTech's BendyBear Shellcode

Hashing Algorithms Used: **`unknown algorithm`**

BendyBear is another x64 shellcode that uses API hashing, and the article detailing its analysis states that "*the shellcode loads its dependency modules and resolves any necessary Windows Application Programming Interface (API) calls using standard shellcode API hashing*"

However, all sources talking about the shellcode did not detail what hashing algorithm was "*standard shellcode API hashing*", but only remarked on the DLLs loaded:

• Advapi32.dll • Kernel32.dll	• Msvcrt.dll • User32.dll	• Ws2_32.dll

The shellcodes available on Malware Bazaar only included the x86 versions of the shellcode, or its WaterBear Loader counterparts.

All in all, it was still the few malware that used purely x64 shellcode (as opposed to x86 shellcode using Heaven's Gate), and so I have included it in this literature survey.

Conclusion

In conclusion, though it is rare to find shellcode written purely for 64-bit Windows systems in the wild, the hashing algorithms that it uses for hashing its Windows API calls can still be shared with that of its 32-bit Windows system counterpart.

As of now, it would appear that most malware shellcodes would still rely on the Heaven's Gate Technique to be able to execute x86 shellcodes on 64-bit systems by abusing Wow64 *(as detailed in Appendix D)*, thus ensuring backwards compatibility for malware on legacy systems that are 32-bit.

There are not many shellcodes written in 64-bit nor articles detailing its difference in API Hashing, and it is still unlikely that Wow64 will be discontinued in the near future due to the strong prevalence of 32-bit applications and Microsoft's penchant for backwards compatibility.

However, it was worthwhile exploring what is currently available, and I hope that the literature survey has covered enough of the available literature online regarding API Hashing for 64-bit malware shellcode.

References

Babkin, D., 2020. *Deep Dive Into Assembly Language - Windows Shellcode - GetProcAddress*. [online] www.dennisbabkin.com. Available at: <https://dennisbabkin.com/blog/?t=how-to-implement-getprocaddress-in-shellcode> [Accessed 10 March 2022].

Brennan, M., 2022. *Hackers No Hashing: Randomizing API Hashes to Evade Cobalt Strike Shellcode Detection*. [online] Huntress.com. Available at: <https://www.huntress.com/blog/hackers-no-hashing-randomizing-api-hashes-to-evade-cobalt-strike-shellcode-detection> [Accessed 1 April 2022].

En.wikipedia.org. 2022. *Win32 Thread Information Block - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Win32_Thread_Information_Block> [Accessed 12 March 2022].

Fernandez, S., 2010. *General notes about exploiting Windows x64*. [online] Immunityinc.com. Available at: <https://www.immunityinc.com/downloads/win64_confidence2010.pdf> [Accessed 10 March 2022].

Harbison, M., 2021. *BendyBear: Novel Chinese Shellcode Linked With Cyber Espionage Group BlackTech*. [online] Unit42. Available at: <https://unit42.paloaltonetworks.com/bendybear-shellcode-blacktech/> [Accessed 5 April 2022].

Herzog, C., 2021. *Using Nt and Zw Versions of the Native System Services Routines - Windows drivers*. [online] Docs.microsoft.com. Available at: <https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/using-nt-and-zw-versions-of-the-native-system-services-routines> [Accessed 10 March 2022].

Hyvärinen, N., 2017. *Analyzing the DOUBLEPULSAR Kernel DLL Injection Technique - F-Secure Blog*. [online] F-Secure Blog. Available at: <https://blog.f-secure.com/analyzing-the-doublepulsar-kernel-dll-injection-technique/> [Accessed 12 March 2022].

Hyvärinen, N., 2018. *DOUBLEPULSAR Usermode Analysis: Generic Reflective DLL Loader - F-Secure Blog*. [online] F-Secure Blog. Available at: <https://blog.f-secure.com/doublepulsar-usermode-analysis-generic-reflective-dll-loader/> [Accessed 23 March 2022].

Ired.team. 2021. *Windows API Hashing in Malware - Red Teaming Experiments*. [online] Available at: <https://www.ired.team/offensive-security/defense-evasion/windows-api-hashing-in-malware> [Accessed 22 March 2022].

Koivunen, T., 2011. *Carberp, a new bag of tricks*. [online] Virusbulletin.com. Available at: <https://www.virusbulletin.com/virusbulletin/2011/01/carberp-new-bag-tricks> [Accessed 2 April 2022].

Offensive-security.com. n.d. *Generating Payloads in Metasploit*. [online] Available at: <https://www.offensive-security.com/metasploit-unleashed/generating-payloads/> [Accessed 1 April 2022].

O'Meara, K., 2019. *Behavioral Matrix and Tool Analysis of Energetic Bear and GreyEnergy Actor*. [online] Apps.dtic.mil. Available at: <https://apps.dtic.mil/sti/pdfs/AD1090355.pdf> [Accessed 18 March 2022].

Popescu, I., 2019. *Writing shellcodes for Windows x64*. [online] Nytro Security. Available at: <https://nytrosecurity.com/2019/06/30/writing-shellcodes-for-windows-x64/> [Accessed 10 April 2022].

Popescu, I., 2016. *Introduction to Windows shellcode development – Part 3*. [online] Security Café. Available at: <https://securitycafe.ro/2016/02/15/introduction-to-windows-shellcode-development-part-3/> [Accessed 10 April 2022].

reddit. 2022. *Writing shellcodes for Windows x64*. [online] Available at: <https://www.reddit.com/r/netsec/comments/c7hiwc/writing_shellcodes_for_windows_x64/> [Accessed 12 March 2022].

Savelesky, K., Miles, E. and Warner, J., 2019. *ABADBABE 8BADF00D: Discovering BADHATCH and a Detailed Look at FIN8's Tooling*. [online] Gigamon. Available at: <https://blog.gigamon.com/2019/07/23/abadbabe-8badf00d-discovering-badhatch-and-a-detailed-look-at-fin8s-tooling/> [Accessed 10 April 2022].

Threat Intelligence Team, 2021. *Decoding Cobalt Strike: Understanding Payloads - Avast Threat Labs*. [online] Avast Threat Labs. Available at: <https://decoded.avast.io/threatintel/decoding-cobalt-strike-understanding-payloads/ > [Accessed 29 March 2022].
Timzen, T., 2014. *Windows x64 Shellcode – Topher Timzen –*. [online] Tophertimzen.com. Available at: <https://www.tophertimzen.com/blog/windowsx64Shellcode/> [Accessed 2 April 2022].

Wanve, U., 2020. *GuLoader: Peering Into a Shellcode-based Downloader | CrowdStrike*. [online] crowdstrike.com. Available at: <https://www.crowdstrike.com/blog/guloader-malware-analysis/> [Accessed 12 March 2022].

Zerosum0x0.blogspot.com. 2017. *DoublePulsar Initial SMB Backdoor Ring 0 Shellcode Analysis*. [online] Available at: <https://zerosum0x0.blogspot.com/2017/04/doublepulsar-initial-smb-backdoor-ring.html> [Accessed 12 March 2022].

Appendix

A Sightings of API Hashing in 64-bit Shellcode Found Online

Exploit DB Shellcodes

EDB-ID	Name	Hashing Algorithm
49819	*Windows/x64* - **Dynamic Null-Free WinExec PopCalc Shellcode** (2021)	NOT <api name to resolve addresses> E.g. `0x9A879AD19C939E9C => calc.exe` `NOT => 0x6578652e636c6163`

49820	*Windows/x64* - **Dynamic NoNull Add RDP Admin (BOKU:SP3C1ALM0V3) Shellcode**
	
	(2021)
		NOT <api name to resolve addresses>
		SHR 0x8
		
		E.g. `0x9C9A87BA9196A80F => WinExec`
		`NOT => 0x636578456e6957F0`
		`SHR 0x8 => 0x00636578456e6957`

13533	*Windows/x64* - **URLDownloadToFileA(http://localhost/trojan.exe) + Execute Shellcode**	`hash_export:`
		`lodsb`
		`add edx,eax`
		`rol edx, 5`
		`dec eax`
		`jns hash_export`
		`ror edx, 5`
		`cmp dx,word [r15]`
		`; found api?`
		`jne load_index`
		
		`movzx edx,word [r11+2*r12-2]`
		`mov eax,[r9+4*rdx]`
		`add rax,rbx`
		`add r15,2`
		`; skip hash`

It seemed that only 3 used obfuscation of their API calls *(though the use of NOT seems more like simple obfuscation than hashing)*. All other shellcodes found on Exploit DB did not seem to use hashing/obfuscation of their API calls of any sort, and either:

1. Pushed the **exact string** in hexadecimal directly to the stack, allowing the plain string to be seen in static analysis of the shellcode
2. Used the **relative address/ordinal** of the desired function from the base address of the DLL and added it to the base address

E.g. (EDB-ID 48229) Windows\x64 - Dynamic MessageBoxA or MessageBoxW PEB & Import Table Method Shellcode

The strings being pushed to the stack are visible in the shellcode and can be seen using the 'strings' utility in static analysis

```
; Create string 'MessageBoxA'
 mov ecx, 0x41786f6f      ; Axoo : 41786f6f
 shr ecx, 8
 push ecx                 ; "oxA,0x00"
 push 0x42656761          ; Bega : 42656761
 push 0x7373654d          ; sseM : 7373654d

 jmp Counter
```

```
char code[] = \
"\x55\x89\xe5\x83\xec\x10\x31\xc9\xf7\xe1\x89\xc3\x64\x8b\x5b\x30\x8b\x5b"
"\x08\x53\x58\x03\x43\x3c\xb2\x80\x66\x01\xd0\x8b\x10\x01\xda\x80\xc2\x0c"
"\x66\xb9\x33\x32\x51\x68\x55\x53\x45\x52\x89\x65\xfc\x31\xc9\x52\x31\xc0"
"\xb0\x14\xf6\xe1\x01\x04\x24\x58\x50\x8b\x75\xfc\x8b\x38\x01\xdf\x51\x31"
"\xc9\xfc\xb1\x06\xf3\xa6\x59\x74\x04\x58\x41\xeb\xde\x58\x89\x45\xf8\x2c"
"\x0c\x8b\x00\x01\xd8\x89\x45\xf4\xb9\x6f\x6f\x78\x41\xc1\xe9\x08\x51\x68"
"\x61\x67\x65\x42\x68\x4d\x65\x73\x73\xeb\x08\xc6\x44\x24\x0a\x57\x8b\x45"
"\xf4\x31\xc9\x89\xe6\x31\xd2\x8b\x38\x39\xd7\x74\xec\x01\xdf\x47\x47\x51"
"\x31\xc9\xfc\xb1\x0b\xf3\xa6\x59\x74\x07\xb2\x04\x01\xd0\x41\xeb\xe0\x8b"
"\x45\xf8\x04\x04\x8b\x38\x01\xdf\x31\xc0\xb0\x04\x66\xf7\xe1\x01\xf8\x8b"
"\x00\x8a\x5c\x24\x0a\x31\xc9\x51\x80\xfb\x41\x74\x18\x68\x4b\x2d\x55\x2d"
"\x68\x42\x2d\x4f\x2d\x89\xe2\x42\x88\x2a\x42\x41\x80\xf9\x04\x74\x07\xeb"
"\xf4\x68\x42\x4f\x4b\x55\x31\xc9\x89\xe3\x51\x53\x53\x51\xff\xd0";
```

E.g. (EDB-ID 45743) Windows/x64 - Remote (Bind TCP) Keylogger Shellcode

```
 6a:   41 5d                    pop    r13
 6c:   48 31 c0                 xor    rax,rax
 6f:   50                       push   rax
 70:   50                       push   rax
 71:   48 b8 77 73 32 5f 33     movabs rax,0x642e32335f327377
 78:   32 2e 64
 7b:   48 89 04 24              mov    QWORD PTR [rsp],rax
```

'd.23_2sw' from calling ws2_32.dll is plainly visible

E.g. (EDB-ID 42992) Windows/x64 - API Hooking Shellcode and (EDB-ID 40549) Windows/x64 - WinExec(cmd.exe) Shellcode

Use of Relative Address

```
xor rdx,rdx
mov rax,[gs:rdx+0x60] ;PPEB
mov rax,[rax+24] ;PPEB->Ldr
mov rsi,[rax+32] ;Ldr->InMemOrderModuleList.Flink
mov rax,[rsi]
mov rsi,[rax]

mov rdi,[rsi+32] ;rdi=kernel32.dll base Address

;-----------------------------------------------------------------
xor rsi,rsi
mov si,0x29f0
add rsi,rdi ;rsi=VirtualProtect()

;---------------------------------
;This Part is Important

xor r12,r12
mov r12w,0xa2b0  ;0x0000a2b0 is Relative Address of DeleteFileW()
add r12,rdi ;r12=DeleteFileW()
```

Use of Ordinal

```
mov rdi,[rsi+0x30] ;kernel32.dll base address


xor rbx,rbx
xor rsi,rsi


mov ebx,[rdi+0x3c] ;elf_anew
add rbx,rdi ;PE HEADER
mov dl,0x88
mov ebx,[rbx+rdx] ;DataDirectory->VirtualAddress
add rbx,rdi ;IMAGE_EXPORT_DIRECTORY

mov esi,[rbx+0x1c] ;AddressOfFunctions
add rsi,rdi


cdq

mov dx,1319 ;Ordinal of WinExec()




mov eax,[rsi+rdx*4]
add rax,rdi ;rax=WinExec()
```

Reference: https://www.exploit-db.com/shellcodes?platform=windows_x86-64

Forrest-orr.net Shellcodes

Name	Hashing Algorithm

Win64 WinExec
 	
Win64 EggHunter
 	
Win64 Message Box
	.HashNextByte:
	• Checks if NULL terminator has been reached
	• **sll1AddHash32** hashing algorithm (from FLARE's shellcode_hashes)
	    ○ *or* of 0x60
	    ○ *add*
	    ○ *shift logical left (shl)* by 1

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                          GetStringHash64                               ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

GetStringHash64:

Push            Rdi
Mov             Rdi, Rdx
Xor             Rbx, Rbx

.HashNextByte:

Cmp             Byte [Rcx], 0
Je              GetStringHash64.HashGenerated
Movzx           Eax, Byte [Rcx]
Or              Al, 60h
Movzx           Edx, Al
Add             Ebx, Edx
Shl             Rbx, 1
Inc             Rcx
Test            Rdi, Rdi
Jz              GetStringHash64.HashNextByte
Inc             Rcx
Jmp             GetStringHash64.HashNextByte

.HashGenerated:

Mov             Rax, Rbx
Pop             Rdi
Ret

End:
```

All 3 shellcodes seem to have been written by the same author and hence have the same hashing algorithm (sllAddHash32).

Reference: https://www.forrest-orr.net/shellcodes

Shell-storm.org Shellcodes

None of the 4 Win64 shellcodes in the repository seemed to use API hashing.

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*

Reference: https://www.forrest-orr.net/shellcodes

B List of Hashing Algorithms: In-depth

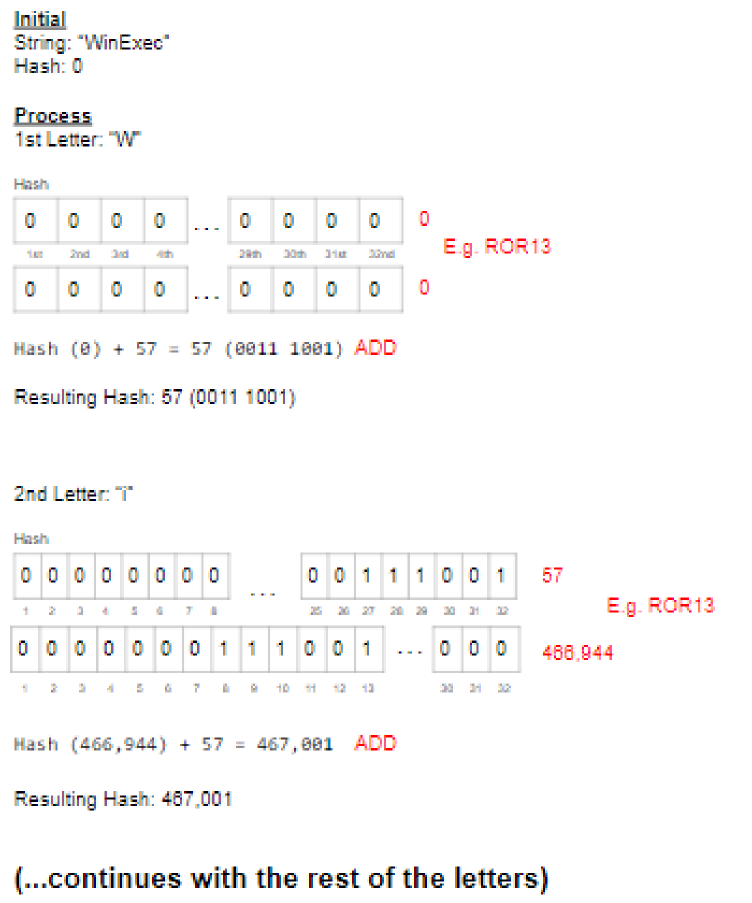
HashDB (as of March 2022)

add_rol5_hash_again	*refer to **addRol5HashOncemore32** in shellcode_hashes
ch_add_rol8	*refer to **chAddRol8Hash32** in shellcode_hashes
add_ror13	*refer to **addRor13Hash32** in shellcode_hashes
ror11_add	*refer to **ror11AddHash32** in shellcode_hashes
rol5_xor	*refer to **rol5XorHash32** in shellcode_hashes
rol3_xor_eax	*refer to **rol3XorEax** in shellcode_hashes
add_ror13_hash_again	*refer to **addRor13HashOncemore32** in shellcode_hashes
rol7_xor	*refer to **rol7XorHash32** in shellcode_hashes
ror9_add	*refer to **ror9AddHash32** in shellcode_hashes
ror13_add_sub20	*refer to **ror13AddHash32Sub20h** in shellcode_hashes *(hashes generated appear to be different for some reason)*
or23_xor_rol17	*refer to **or23hXorRor17Hash32** in shellcode_hashes
rol3_xor	*refer to **rol3XorHash32** in shellcode_hashes
rol7_add_xor2	*refer to **rol7AddXor2Hash32** in shellcode_hashes
rol5_add	*refer to **rol5XorHash32** in shellcode_hashes
ror13_add_sub1	*refer to **ror13AddHash32Sub1** in shellcode_hashes
xor_rol9	*refer to **xorRol9Hash32** in shellcode_hashes
permutations_e8677835	*refer to **playWith0xe8677835Hash** in shellcode_hashes
shl7_shr19_xor	*refer to **shl7Shr19XorHash32** in shellcode_hashes
ror13_add	*refer to **ror13AddHash32** in shellcode_hashes
fnv1a_64 *(64-bit hashing algorithm)*	Hash = **0xcbf29ce484222325** For each letter: • Hash XOR letter • **100000001b3** * Hash Same as **fnv1_64** but reversing the steps.

rol9_xor	*refer to **rol9XorHash32** in shellcode_hashes
mul83_add	*refer to **imul83hAdd** in shellcode_hashes
xor_shr8	*refer to **xorShr8Hash32** in shellcode_hashes
rol9_add	*refer to **rol9AddHash32** in shellcode_hashes
add1501_shl5	**Hash = 0x1505** For each letter: ● bit shift to the left ***(shl)* hash by 5 bits** ● ***add*** letter
ror7_add	*refer to **rol7AddHash32** in shellcode_hashes
fnv1_64 *(64-bit hashing algorithm)*	**Hash = 0xcbf29ce484222325** For each letter: ● **0x100000001b3** * Hash ● Hash XOR letter Same as **fnv1** from shellcode_hashes but 64-bit.
rol8_xor_b0d4d06	*refer to **rol8Xor0xB0D4D06Hash32** in shellcode_hashes
rol7_add	*refer to **rol7AddHash32** in shellcode_hashes
shl1_add	*refer to **sll1AddHash32** in shellcode_hashes *(given the order, it is more like add_Shl1 / addSll1Hash32)*
shr2_shl5_xor_init_c4d5a97a_stealbit	**Hash = 0xc4d5a97a** For each letter: ● Hash1 = bit shift to the right ***(shr)* Hash by 2 bits** ● Hash2 = bit shift to the left ***(shl)* Hash by 5 bits** ● Hash = Hash1 *add* Hash2 *add* letter
fnv1a	*refer to **fnv1** in shellcode_hashes
fnv1	**Hash = 0x811c9dc5** For each letter: ● **0x1000193** * Hash ● Hash XOR letter Same as **fnv1** from shellcode_hashes but reversing the steps.
carbanak	*refer to **hash_Carbanak** in shellcode_hashes

or21_xor_rol11	*refer to **or21hXorRor11Hash32** in shellcode_hashes
mult21_add	*refer to **imul21hAddHash32** in shellcode_hashes
fnv1_xor67f	*refer to **fnv1Xor67f** in shellcode_hashes
add_ror4	For each letter: ● Hash ***add*** Hash (zero out last 8 bits, & 0xffffff00) ***add*** character ● *rotate right **(ror)*** hash by **4 bits** Same as **addRor4WithNullHash32** but without adding the null hash
djb2_nokoyawa	**Hash = 5381/0x1505** For each letter: ● Hash = (Hash *** 33**) + (letter convert to uppercase by -20h)
guloader_3C389ABC *(DJB2 variant in x86 shellcode)*	**Hash = 5381/0x1505** For each letter: ● Hash = ((Hash *** 33**) + letter) **XOR 0x3C389ABC**
revil_010F	**Hash = x2b** For each letter: ● Hash = Hash *** 0x010F** + letter Final Hash = Hash ***AND*** **0x1fffff**

Mandiant FLARE's shellcode_hashes (as of March 2022)

ror7AddHash32	For each letter: ● *rotate right* ***(ror)*** hash by **7 bits** ● ***add*** letter
ror9AddHash32	● *rotate right* ***(ror)*** hash by **9 bits** ● ***add*** letter
ror11AddHash32	● *rotate right* ***(ror)*** hash by **11 bits** ● ***add*** letter
ror13AddHash32	● *rotate right* ***(ror)*** hash by **13 bits** ● ***add*** letter
ror13AddHash32Sub20h	● *rotate right* ***(ror)*** hash by **13 bits** ● ***add (converts to uppercase if lowercase by -20h)*** Same as ror13AddHash32 but standardises all letters to uppercase
ror13AddWithNullHash32	Append **b"\x00"** to end of string before doing **ror13AddHash32** Adding null to the end results in one final *ror13* without the *add* at the end.
ror13AddHash32Sub1	Same as **ror13AddHash32**, but **-1 from resulting hash**
rorXXAddHash32 **hashing algorithm process example**	

ror13AddHash32AddDll	Hash of **DLL Name**: For each letter: ● *rotate right* ***(ror)*** by **13 bits** ● ***add (converts to uppercase if lowercase by -32)*** ● *rotate right* ***(ror)*** by **13 bits** Does <u>2 final rounds</u> of ror13 after all letters have been added Hash of Function Name: For each letter: ● *rotate right* ***(ror)*** by **13 bits** ● ***add (case-insensitive)*** Does <u>1 final round</u> of ror13 after all letters have been added Final Hash = **DLL Name Hash + Function Name Hash** *(If exceeding 32 bits, it takes the overflow by subtracting 4294967296 (1111 1111 1111 1111 1111 1111 1111 1111))*
addRor4WithNullHash32	Append **b"\x00"** to end of string For each letter: ● Hash ***add*** Hash (zero out last 8 bits, & 0xffffff00) ***add*** character ● *rotate right* ***(ror)*** hash by **4 bits**
addRor13Hash32	Same as **ror13AddHash32**, but **reversing the steps (add first then ror13)**
addRor13HashOncemore32	Same as **addRor13Hash32**, but does **1 final round of ror13** after all letters have been added
addRol5HashOncemore32	For each letter: ● ***add*** letter ● *rotate left* ***(rol)*** hash by **5 bits** Does <u>1 final round</u> of rol5 after all letters have been added
rol5AddHash32	For each letter: ● *rotate left* ***(rol)*** hash by **5 bits** ● ***add*** letter
rol7AddHash32	● *rotate left* ***(rol)*** hash by **7 bits** ● ***add*** letter
rol9AddHash32	● *rotate left* ***(rol)*** hash by **9 bits** ● ***add*** letter

xorRol9Hash32	Same as **rol9XorHash32**, but **reversing the steps (rol9 first then xor letter)**
rol3XorEax	xor_key = 0 (this is called 'eax' in the python script) For each letter: ● add letter to last 8 bits of xor_key (using *or*) ● Hash = Hash **XOR** xor_key ● *rotate left **(rol)*** hash by **3 bits** ● Hash *add* 1 ● xor_key *bit shift to the left (shl)* by 8 bits
rol3XorHash32	For each letter: ● *rotate left **(rol)*** hash by **3 bits** ● Hash = Hash **XOR** letter
rol5XorHash32	● *rotate left **(rol)*** hash by **5 bits** ● Hash = Hash **XOR** (letter in lowercase, *OR* 32)
rol7XorHash32	● *rotate left **(rol)*** hash by **7 bits** ● Hash = Hash **XOR** letter
rol9XorHash32	● *rotate left **(rol)*** hash by **9 bits** ● Hash = Hash **XOR** letter
rol7AddXor2Hash32	For each letter: ● *rotate left **(rol)*** hash by **7 bits** ● ***add*** (letter **XOR 2**)
chAddRol8Hash32 *(oddly specific)*	For each letter: ● Hash = Hash **XOR** (letter * 256) ● *rotate left **(rol)*** hash by **8 bits** ● middle_part_of_hash = 4th and 5th bits of Hash in hexadecimal ● Hash = Hash **XOR** middle_part_of_hash
rol8Xor0xB0D4D06Hash32	For each letter: ● Hash = Hash **XOR** (uppercase letter, *AND* 0xDF) ● *rotate left **(rol)*** hash by **8 bits** ● Hash = Hash ***add*** (uppercase letter, *AND* 0xDF) Final Hash = Hash **XOR 0xB0D4D06**
shl7Shr19XorHash32	For each letter: ● Hash1 = *bit shift to the left **(shl)*** hash by **7 bits** ● Hash2 = *bit shift to the right **(shr)*** hash by **25 bits (0x19)** ● Final Hash of Iteration = (Hash1 OR Hash2) **XOR** (letter **XOR** 0xF4)

shl7Shr19AddHash32	For each letter: • Hash1 = *bit shift to the left **(shl)*** hash by **7 bits** • Hash2 = *bit shift to the right **(shr)*** hash by **25 bits (0x19)** • Final Hash of Iteration = (Hash1 OR Hash2) ***add*** letter
shl7SubHash32DoublePulser	For each letter: • *bit shift to the left **(shl)*** key by **7 bits** • Hash = **(key - Hash)** + letter Does 1 final iteration after all letters, but without adding the letter (i.e. Hash = key - Hash)

sll1AddHash32
 	
Seen often, used by Trojan Heriplor / Energetic Bear API Hashing Tool
	• *or* of 0x60
	• *add*
	• *shift logical left (shl)* by 1
	
	`def sll1AddHash32(inString,fName):`
	`    if inString is None:`
	`        return 0`
	`    val = 0`
	`    for i in inString:`
	`        b = i`
	`        b = 0xff & (b | 0x60)`
	`        val = val + b`
	`        val = val << 1`
	`        val = 0xffffffff & val`
	`    return val`

shr2Shl5XorHash32	Removes “Nt” or “Zw” from the beginning of the string For each letter: • Hash1 = *bit shift to the right **(shr)*** hash by **2 bits** • Hash2 = *bit shift to the left **(shl)*** hash by **5 bits** • Final Hash of Iteration = Hash **XOR** (letter + Hash1 + Hash2)
xorShr8Hash32	Hash = 0xFFFFFFFF For each letter: • Hash1 = *bit shift to the right **(shr)*** hash by **8 bits** • Hash2 = Hash **XOR** letter • Final Hash of Iteration = Hash1 **XOR** Hash * Hash2
imul83hAdd	For each letter: • Hash = Hash *** 0x83** (131) • ***add* character**

imul21hAddHash32	For each letter: ● Hash = Hash *** 0x21** (33) ● ***add*** (**uppercase character**, & 0xFFFFFFDF)
or21hXorRor11Hash32	For each letter: ● Hash = Hash **XOR** (letter *OR* 33 **(21h)**) (I'm not sure if this was meant to be converting it to lowercase since OR 33 does not always result in the lowercase of the same letter) ● *rotate left **(rol)*** hash by **11 bits** (0xb)
or23hXorRor17Hash32	For each letter: ● Hash = Hash **XOR** (letter *OR* 35 **(23h)**) (I'm not sure if this was meant to be converting it to lowercase since OR 35 does not always result in the lowercase of the same letter) ● *rotate left **(rol)*** hash by **17 bits** (0x11)
playWith0xe8677835Hash *(oddly specific)*	For each letter: ● Hash = Hash XOR letter ● Repeat the following 8 times: ○ Hash = Hash **XOR 0xe8677835** (only if Hash not 0x00000000) ○ *bit shift to the right (shr)* by 1
poisonIvyHash	Append **b"\x00"** to end of string if not already present For each letter: ●
mult21AddHash32	For each letter: ● Hash *** 0x21** ● ***add*** letter
add1505Shl5Hash32	**Hash = 0x1505** For each letter: ● Hash = Hash ***add*** (bit shift to the left ***(shl)*** **hash by 5**) ● ***add*** letter
dualaccModFFF1Hash	**Hash0 = 0, Hash1 = 1** (dual accumulators) For each letter: ● Hash1 = (letter *add* Hash1) **MOD 0x0FF1** ● Hash0 = (Hash0 + Hash1) **MOD 0x0FF1** Hash = bit shift to the left *(shl)* Hash0 by 0x10/16

<table>
<tr><td>hash_Carbanak</td><td>For each letter:
Hash
= letter add (Hash bit shift to the left (shl) by 4)
If Hash > 4,026,531,840 (0xF0000000):
((Hash AND 0xF0000000) shr 24) XOR Hash</td></tr>
<tr><td>hash_ror13AddUpperDllname Hash32</td><td>Hash of DLL Name:
For each letter:
rotate right (ror) by 13 bits
add (converts to uppercase if lowercase by -32)
rotate right (ror) by 13 bits
Does <u>2 final rounds</u> of ror13 after all letters have been added

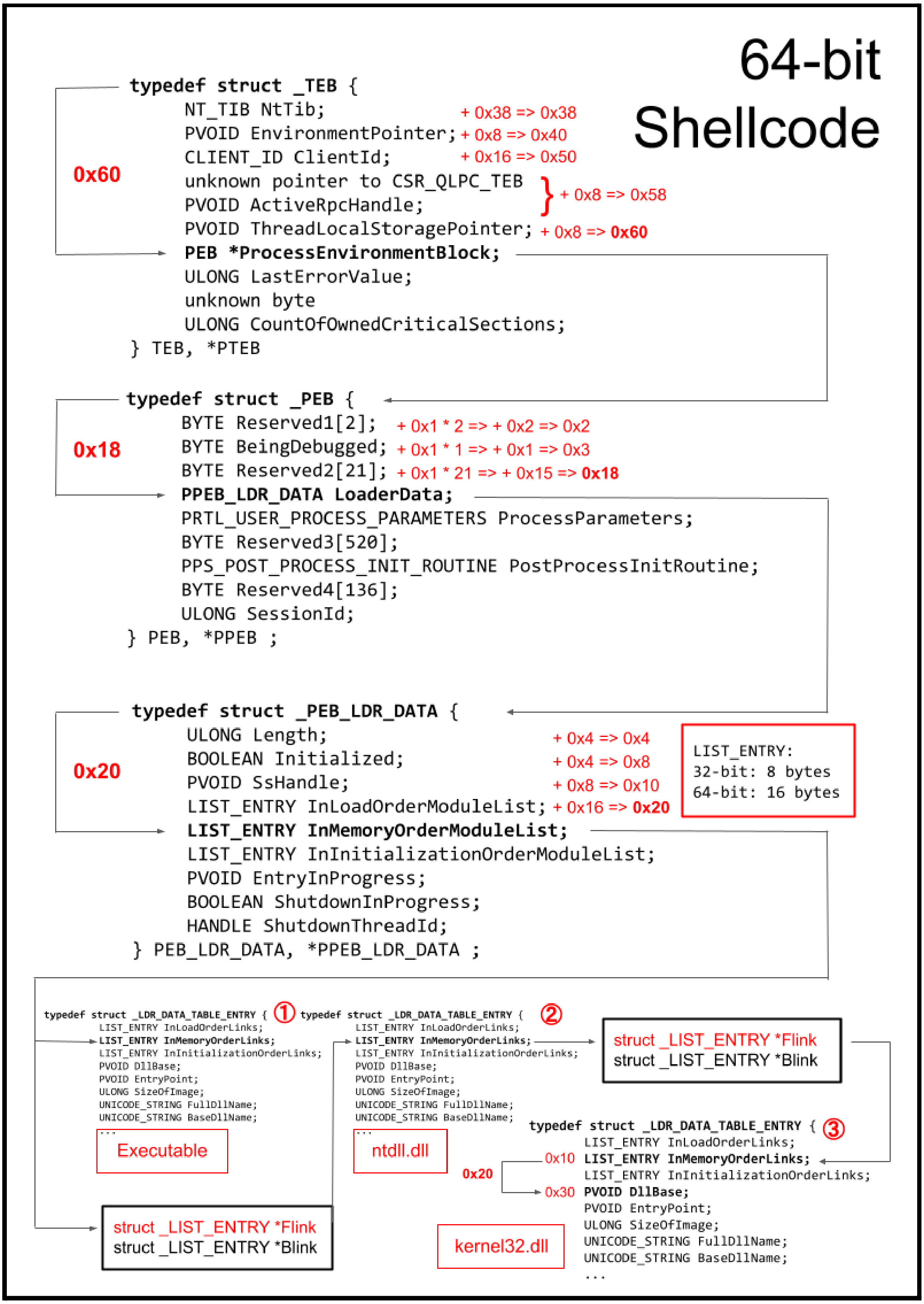
Hash of Function Name:
For each letter:
rotate right (ror) by 13 bits
add (case-insensitive)
Does <u>1 final round</u> of ror13 after all letters have been added

Final Hash = DLL Name Hash + Function Name Hash</td></tr>
<tr><td>fnv1Xor67f</td><td>Hash = 0x811c9dc5

For each letter:
Hash XOR letter
0x1000193 * Hash
Final_hash = Hash XOR 0x67f</td></tr>
<tr><td>fnv1</td><td>Hash = 0x811c9dc5

For each letter:
Hash XOR letter
0x1000193 * Hash</td></tr>
</table>

C 64-Bit Shellcode Diagrams (Loading Kernel32.dll)



NOTE: The diagram depicts following InMemoryOrderModuleList, but this is not necessarily the case as the shellcode may choose to follow InInitializationOrderModuleList or InLoadOrderModuleList as well.