

# AgentTesla dropped via NSIS installer

I1v1ngc0d3.wordpress.com/2021/11/12/agenttesla-dropped-via-nsis-installer/

dominik

12 Nov 2021

Lately one of our customers received a suspicious file which was blocked by our sandbox solution but it was unclear if this was malicious and if so what malware it was so I did an analysis and want to share my results with you. The main goal of this article is to show how to extract the final payload.

The sample is now available on VT:

ce8a9bf908ce35bf0c034c61416109a44f015eabf058b12485450cd40af95fc3

## NSIS installer

If we do some static analysis via DiE (Detect it Easy) we see that the file is of type NSIS installer. One easy way to obtain the files is to just simply extract the files with 7-ZIP. Unfortunately I'm not aware of any way to reverse the NSIS script (any hint welcome ;-)).

installer	Nullsoft Scriptable Install System(2.40)[zlib,solid]	S	?
linker	Microsoft Linker(6.0*)[EXE32]	S	?
overlay	NSIS data(-)[-]	S	

After extracting the file we find a folder \$PLUGINDIR as expected and another file with a random name and some bytes in it.

📁 \$PLUGINDIR	11/11/2021 3:35 AM	File folder	
📄 1bcfef39f3a713c08f90e93311ca5b5a.ec73...	11/10/2021 12:55 ...	FILE_3D72404B706...	308 KB
📄 w66zlsqpnuyue6	11/9/2021 11:34 PM	File	286 KB

## First dll – swfmwfkkeh.dll

Inside of the \$PLUGINDIR directory we find one file named swfmwfkkeh.dll (SHA1: 56f3d68f10bde42216634f987b421feee696506e). Once again we open it up in DiE and find out that its written in C/C++ and some exports which look a little strange.

compiler	Microsoft Visual C/C++ (2015 v.14.0)[msvcrt]		S
linker	unknown(14.0)[32]		S
rdin ^	RVA	Name	
0000	00000000	00000000	
0001	00001310	00007c2e	zsnqqjqi

In the imports there are some false flags but the VirtualProtect seems to be reasonable

3	00008232		03f3	MultiByteToWideChar
4	00008248		05d0	VirtualProtect
5	0000825a		0602	WideCharToMultiByte
6	00008270		0630	lstrcatA
7	0000827c		0639	lstrcpyA
8	00008288		063c	lstrcpynA
9	00008294		063f	lstrlenA

Now we open up the file in IDA Pro and take a look at the export functions

Name	Address	Ordinal
zqnqqjqj	10001310	1
DriverEntry	100067A6	[main entry]

We assume that the NSIS installer will start the DLL and call the exported function „zqnqqjqj“ so we start our analysis there. After setting up the stack the function initializes „var\_14“ which then is compared if its above 4722 (in fact the function does a „jump not below“ with the main functionality in the false tree).

We will start with the right block first because there you can see that the memory address at 10009014 will receive RWX permissions and after that will be called. So we can assume that at this location there must be some assembly code. Now lets take a look what happens in the left block because this is where we land after the first comparison. At this point „var\_14“ is still below 4722. As you can see the variable (now in ecx) is used as a pointer in the marked area by utilizing „byte ptr loc\_10009014[ecx]“. So it grabs the first bytes of what ever is at this address. If

we take a look what is there we see some „strange“ assembly code – this doesn't look as valid assembly code at all. If we look further down in the left block we see some xor, sub, add and mov operations so we can assume that this code will be modified.

```

loc_10009014:                ; CODE XREF: zznqqjqji+2BB↑p
                                ; DATA XREF: zznqqjqji+29↑r ...
                                ; Store String
    stosd
    push    edx
    leave  ; High Level Procedure Exit
    mov     word ptr [esi-5Ch], ds
    cmc    ; Complement Carry Flag
    idiv   ebp                ; Signed Divide
    mov     edx, gs
    cmp     al, 0AAh ; 'a' ; Compare Two Operands
    or     esp, [eax+edi] ; Logical Inclusive OR
    sub     [edx], esi        ; Integer Subtraction
    rcl    edx, 30h          ; Rotate Through Carry Left
    jnz    short near ptr loc_1000905A+1 ; Jump if Not Zero (ZF=0)
    push   ds
    or     byte ptr [ebp-34h], 25h ; Logical Inclusive OR
    imul  ecx, esi, 3Fh ; '?' ; Signed Multiply
    scasd ; Compare String
    sub   eax, 18223C8Ah ; Integer Subtraction
    or   [eax-41323851h], esp ; Logical Inclusive OR
    pop  edx
    popf ; Pop Stack into Flags Register
    pop  eax
    cld  ; Clear Direction Flag
    push esp
    dec  ecx                ; Decrement by 1
    xor  ch, [edx]          ; Logical Exclusive OR
    add  ah, [esi+edx*8] ; Add
    fmul st(3), st         ; Multiply Real
    dec  esi                ; Decrement by 1
    cmp  dword ptr [eax-20751D43h], 0FFFFFFCFh ; Compare Two Operands
    cmp  ebx, [ebp+2C8763A5h] ; Compare Two Operands

```

After all the byte manipulations are done the byte is written back to the address 10009014. This time esi receives the pointer (mov esi, [ebp+var\_14]).

```

xor     eax, esi ; Logical Exclusive OR
mov     [ebp+var_15], al
mov     al, [ebp+var_15]
mov     esi, [ebp+var_14]
mov     byte ptr loc_10009014[esi], al
mov     eax, [ebp+var_14]
add     eax, 1 ; Add
mov     [ebp+var_14], eax
jmp     loc_10001327 ; Jump

```

Then we jump back to the comparison if var\_14 is already above 4722. If this is the case we change the page permissions to RWX and execute the now modified code.

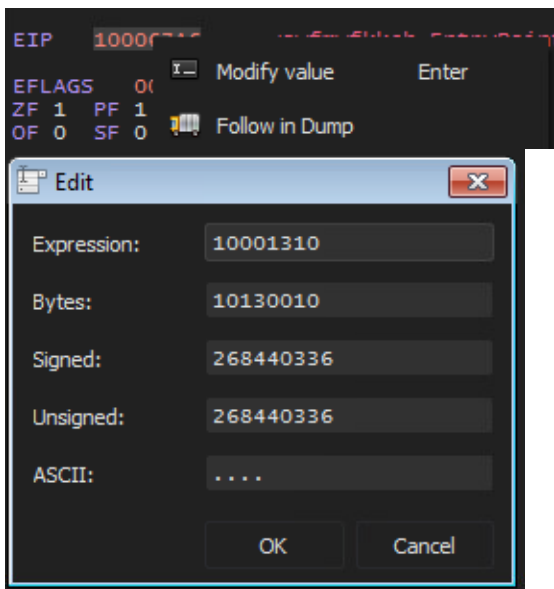
## Let's debug some code

The decryption routine is very very long and we don't want to go into reversing this algorithm. Below is a screenshot of the algorithm – hell no!

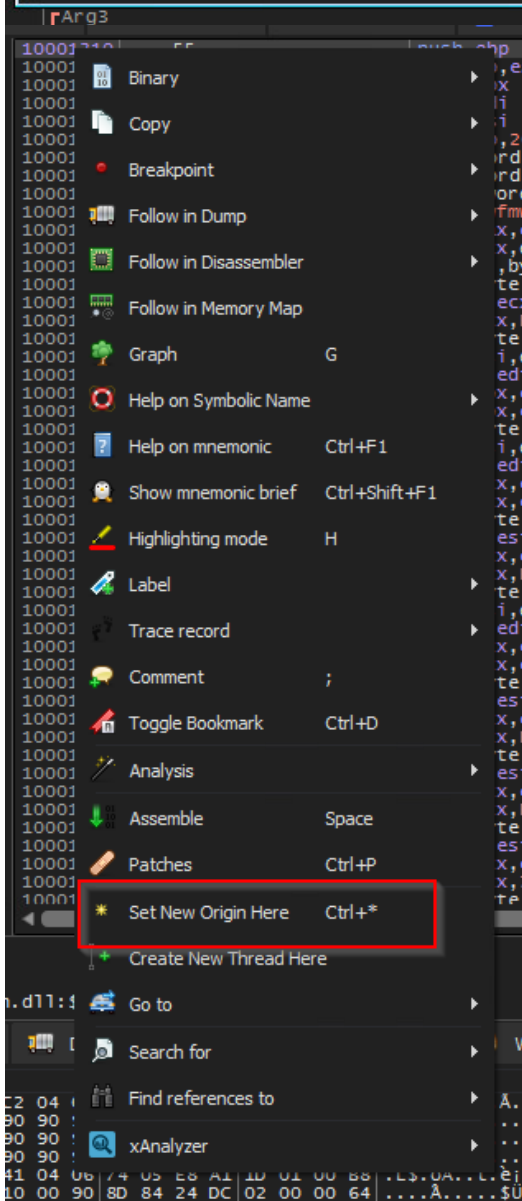
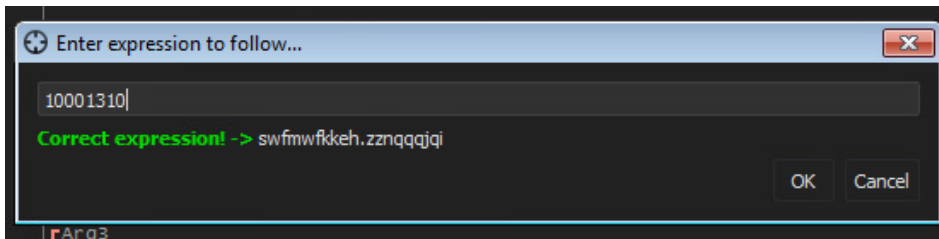




So let's open the file up in x64dbg and jump to user code but stop there. We want to start at the function „zsnqqqjqi“. If we switch to text mode in IDA we can see the address. We have two options to change the instruction pointer (EIP) to continue from this address. First we can right-click on the EIP register and modify the value.



Or we can jump to the address in the disassembly view (Ctrl + G) and then right-click „Set new origin here“.



After that we can start debugging.

EIP	Hex	OpCode	Comment	Symbol
10001310	55		push ebp	zznqqjqj
10001311	89E5		mov ebp,esp	
10001313	53		push ebx	
10001314	57		push edi	
10001315	56		push esi	
10001316	83EC 20		sub esp,20	
10001319	C745 EC 00000000		mov dword ptr ss:[ebp-14],0	
10001320	C745 EC 00000000		mov dword ptr ss:[ebp-14],0	
10001327	> 817D EC 72120000		cmp dword ptr ss:[ebp-14],1272	
1000132E	0F83 6B020000		jae swfmwfkkeh.1000159F	
10001334	31C0		xor eax,eax	
10001336	8B4D EC		mov ecx,dword ptr ss:[ebp-14]	
10001339	8A140D 14900010		mov dl,byte ptr ds:[ecx+10009014]	
10001340	8855 EB		mov byte ptr ss:[ebp-15],dl	
10001343	0FB64D EB		movzx ecx,byte ptr ss:[ebp-15]	
10001347	83F1 FF		xor ecx,FFFFFFFF	
1000134A	884D EB		mov byte ptr ss:[ebp-15],cl	
1000134D	8B75 EC		mov esi,dword ptr ss:[ebp-14]	
10001350	0FB67D EB		movzx edi,byte ptr ss:[ebp-15]	
10001354	89FB		mov ebx,edi	
10001356	29F3		sub ebx,esi	
10001358	885D EB		mov byte ptr ss:[ebp-15],bl	
1000135B	8B75 EC		mov esi,dword ptr ss:[ebp-14]	
1000135E	0FB67D EB		movzx edi,byte ptr ss:[ebp-15]	
10001362	89F9		mov ecx,edi	
10001364	31F1		xor ecx,esi	
10001366	884D EB		mov byte ptr ss:[ebp-15],cl	
10001369	0FB675 EB		movzx esi,byte ptr ss:[ebp-15]	
1000136D	89F1		mov ecx,esi	
1000136F	83F1 FF		xor ecx,FFFFFFFF	
10001372	884D EB		mov byte ptr ss:[ebp-15],cl	
10001375	8B75 EC		mov esi,dword ptr ss:[ebp-14]	
10001378	0FB67D EB		movzx edi,byte ptr ss:[ebp-15]	
1000137C	89F9		mov ecx,edi	
1000137E	01F1		add ecx,esi	
10001380	884D EB		mov byte ptr ss:[ebp-15],cl	
10001383	0FB675 EB		movzx esi,byte ptr ss:[ebp-15]	
10001387	89F1		mov ecx,esi	
10001389	81F1 B3000000		xor ecx,83	
1000138F	884D EB		mov byte ptr ss:[ebp-15],cl	
10001392	0FB675 EB		movzx esi,byte ptr ss:[ebp-15]	
10001396	89F1		mov ecx,esi	
10001398	81C1 E5000000		add ecx,E5	
1000139E	884D EB		mov byte ptr ss:[ebp-15],cl	
100013A1	0FB675 EB		movzx esi,byte ptr ss:[ebp-15]	
100013A5	89F1		mov ecx,esi	

We also see as we did in IDA Pro the initialization of the pointer and the comparison with 1272 (hex) = 4722 (dez). Here is on speciality about assembly. Instead of the jnb operation we saw in IDA Pro we see a jae (jump if above or equal). In fact the operations are interchangeable because both check if the ZERO flag is set.

We know from our static analysis that after the decryption loop we change the permission of the page so lets follow the jump and place a breakpoint at this point.

10001597	8945 EC		mov dword ptr ss:[ebp-14],eax	
1000159A	E9 88FDFFFF		jmp swfmwfkkeh.10001327	
• 1000159F	> 8D05 14900010		lea eax,dword ptr ds:[10009014]	
100015A5	890424		mov dword ptr ss:[esp],eax	[LPVOID lpAddress = [esp]:sub_77609250+14
100015A8	C74424 04 72120000		mov dword ptr ss:[esp+4],1272	SIZE_T dwSize = "MZx"
100015B0	C74424 08 40000000		mov dword ptr ss:[esp+8],40	DWORD flNewProtect = PAGE_EXECUTE_READWRITE
100015B8	8D45 F0		lea eax,dword ptr ss:[ebp-10]	
100015BB	894424 0C		mov dword ptr ss:[esp+C],eax	PDWORD lpflOldProtect
100015BF	FF15 747F0010		call dword ptr ds:[<&VirtualProtect>]	VirtualProtect
100015C5	83EC 10		sub esp,10	
100015C8	8945 E4		mov dword ptr ss:[ebp-1C],eax	[ebp-1C]:sub_77609250+14
100015CB	E8 447A0000		call swfmwfkkeh.10009014	
100015D0	31C0		xor eax,eax	

Now we want to observe how the code changes (without reversing the algorithm). Follow the address 10009014 in disassembler to see the code.



```

10009014 5 AB stosd
10009015 52 push edx
10009016 C9 leave
10009017 8C5E A4 mov word ptr ds:[esi-5C],ds
1000901A F5 cmc
1000901B F7FD idiv ebp
1000901D 8CEA mov edx,gs
1000901F 3C AA cmp al,AA
10009021 0B2438 or esp,dword ptr ds:[eax+edi]
10009024 2932 sub dword ptr ds:[edx],esi
10009026 C1D2 30 rcl edx,30
10009029 75 30 jne swfmwfkkeh.1000905B
1000902B 1E push ds
1000902C 824D CC 25 or byte ptr ss:[ebp-34],25
10009030 6BCE 3F imul ecx,esi,3F
10009033 AF scasd
10009034 2D 8A3C2218 sub eax,18223C8A
10009039 09A0 AFC7CDBE or dword ptr ds:[eax-41323851],esp
1000903F 5A pop edx
10009040 9D popfd
10009041 58 pop eax
10009042 FC cld
10009043 54 push esp
10009044 49 dec ecx
10009045 322A xor ch,byte ptr ds:[edx]
10009047 0224D6 add ah,byte ptr ds:[esi+edx*8]
1000904A DCCB fmul st(3),st(0)
1000904C 4E dec esi
1000904D 83B8 BDE28ADF CF cmp dword ptr ds:[eax-20751D43],FFFFFFC
10009054 3B9D A563872C cmp ebx,dword ptr ss:[ebp+2C8763A5]
1000905A D2EF shr bh,cl
1000905C AF scasd
1000905D 26:27 daa
1000905F D238 sar byte ptr ds:[eax],cl
10009061 FF07 inc dword ptr ds:[edi]

```

Before decryption

```

10009014 5 E9 D8070000 jmp swfmwfkkeh.100097F1
10009019 55 push ebp
1000901A 8B5C mov ebp,esp
1000901C 83EC 40 sub esp,40
1000901F 53 push ebx
10009020 56 push esi
10009021 57 push edi
10009022 8365 F0 00 and dword ptr ss:[ebp-10],0
10009026 0F57C0 xorps xmm0,xmm0
10009029 66:0F1345 E0 movlpd qword ptr ss:[ebp-20],xmm0
1000902E 0F57C0 xorps xmm0,xmm0
10009031 66:0F1345 E8 movlpd qword ptr ss:[ebp-18],xmm0
10009036 8365 F8 00 and dword ptr ss:[ebp-8],0
1000903A C745 FC 28000000 movl qword ptr ss:[ebp-4],28
10009041 8365 F4 00 and dword ptr ss:[ebp-C],0
10009045 FF75 0C push dword ptr ss:[ebp+C]
10009048 FF75 10 push dword ptr ss:[ebp+10]
1000904B 8D45 F8 lea eax,dword ptr ss:[ebp-8]
1000904E 50 push eax
1000904F E8 FD000000 call swfmwfkkeh.10009151
10009054 8945 D8 mov dword ptr ss:[ebp-28],eax
10009057 8955 DC mov dword ptr ss:[ebp-24],edx
1000905A FF75 0C push dword ptr ss:[ebp+C]
1000905D FF75 10 push dword ptr ss:[ebp+10]
10009060 8D45 F8 lea eax,dword ptr ss:[ebp-8]
10009063 50 push eax
10009064 E8 E8000000 call swfmwfkkeh.10009151
10009069 8945 D0 mov dword ptr ss:[ebp-30],eax
1000906C 8955 D4 mov dword ptr ss:[ebp-2C],edx
1000906F FF75 0C push dword ptr ss:[ebp+C]
10009072 FF75 10 push dword ptr ss:[ebp+10]
10009075 8D45 F8 lea eax,dword ptr ss:[ebp-8]
10009078 50 push eax
10009079 E8 D3000000 call swfmwfkkeh.10009151
1000907E 8945 C8 mov dword ptr ss:[ebp-38],eax
10009081 8955 CC mov dword ptr ss:[ebp-34],edx
10009084 FF75 0C push dword ptr ss:[ebp+C]
10009087 FF75 10 push dword ptr ss:[ebp+10]
1000908A 8D45 F8 lea eax,dword ptr ss:[ebp-8]
1000908D 50 push eax
1000908E E8 BE000000 call swfmwfkkeh.10009151
10009093 8945 C0 mov dword ptr ss:[ebp-40],eax

```

After decryption

As you can see now the code changed fundamentally and it starts with a jmp to another offset in the code. Jump back to the current instruction by double-clicking the EIP register.

Lets single step over the VirtualProtect and then jump into the decrypted code (F7).

```

1000159F > 8D05 14900010 lea eax,dword ptr ds:[10009014]
100015A5 890424 mov dword ptr ss:[esp],eax
100015A8 C74424 04 72120000 mov dword ptr ss:[esp+4],1272
100015B0 C74424 08 40000000 mov dword ptr ss:[esp+8],40
100015B8 8D45 F0 lea eax,dword ptr ss:[ebp-10]
100015BB 894424 0C mov dword ptr ss:[esp+C],eax
100015BF FF15 747F0010 call dword ptr ds:[<&VirtualProtect>]
100015C5 83EC 10 sub esp,10
100015C8 8945 E4 mov dword ptr ss:[ebp-1C],eax
100015CB E8 447A0000 call swfmwfkkeh.10009014
100015D0 31C0 xor eax,eax

```

We follow the jump and land at this address where we can see values being push/popped and then moved into a local variable (ebp-28). These values are ASCII codes so we can convert them manually or step over until we reach 0x10009849 where the string is terminated by a 0 (xor eax,eax = 0; mov memory,ax).



```

100097F1 55 push ebp
100097F2 8BEC mov ebp,esp
100097F4 81EC 80040000 sub esp,480
100097FA 6A 53 push 53
100097FC 58 pop eax
100097FD 66:8945 D8 mov word ptr ss:[ebp-28],ax
10009801 6A 68 push 68
10009803 58 pop eax
10009804 66:8945 DA mov word ptr ss:[ebp-26],ax
10009808 6A 6C push 6C
1000980A 58 pop eax
1000980B 66:8945 DC mov word ptr ss:[ebp-24],ax
1000980F 6A 77 push 77
10009811 58 pop eax
10009812 66:8945 DE mov word ptr ss:[ebp-22],ax
10009816 6A 61 push 61
10009818 58 pop eax
10009819 66:8945 E0 mov word ptr ss:[ebp-20],ax
1000981D 6A 70 push 70
1000981F 58 pop eax
10009820 66:8945 E2 mov word ptr ss:[ebp-1E],ax
10009824 6A 69 push 69
10009826 58 pop eax
10009827 66:8945 E4 mov word ptr ss:[ebp-1C],ax
1000982B 6A 2E push 2E
1000982D 58 pop eax
1000982E 66:8945 E6 mov word ptr ss:[ebp-1A],ax
10009832 6A 64 push 64
10009834 58 pop eax
10009835 66:8945 E8 mov word ptr ss:[ebp-18],ax
10009839 6A 6C push 6C
1000983B 58 pop eax
1000983C 66:8945 EA mov word ptr ss:[ebp-16],ax
10009840 6A 6C push 6C
10009842 58 pop eax
10009843 66:8945 EC mov word ptr ss:[ebp-14],ax
10009847 33C0 xor eax,eax
10009849 66:8945 EE mov word ptr ss:[ebp-12],ax
1000984D 8365 F8 00 and dword ptr ss:[ebp-8],0

```

If we follow the address ebp-28 we can see the string

```

0014F8FE 00 00 7A 00 5F 77 3A F0 93 75 FF FF FF FF 30 F9 ..z._w:ð.uyyyy0u
0014F90E 14 00 34 F9 14 00 40 00 00 00 53 00 68 00 6C 00 ..4u..@...S.h.l
0014F91E 77 00 61 00 70 00 69 00 2E 00 64 00 6C 00 6C 00 w.a.p.i...d.l.l
0014F92E 00 00 00 90 00 10 00 20 00 00 40 00 00 00 64 F9 .....@...du
0014F93E 14 00 74 F9 14 00 D0 15 00 10 14 90 00 10 72 12 ..tu.D.....r

```

After that the next call will get us the Magic (4D 5A) of Kernel32.dll

```

100096DB 64:A1 30000000 mov eax,dword ptr fs:[30]
100096E1 8B40 0C mov eax,dword ptr ds:[eax+C]
100096E4 8B40 0C mov eax,dword ptr ds:[eax+C]
100096E7 8B00 mov eax,dword ptr ds:[eax]
100096E9 8B00 mov eax,dword ptr ds:[eax]
100096EB 8B40 18 mov eax,dword ptr ds:[eax+18]
100096EE C3 ret

```

After that we see a lot of calls to the same function and what we suspect to be API hashes. So lets dive into this function and try to figure out what algorithm is used to hash the API function names. For easier analysis I dumped the DLL again with Scylla and opened it up again in IDA Pro.

## API Hashing

We will go into the details of the function but what it mainly does is hashing the function names (exports) of the DLL and comparing it with the provided hash. If they match the function is found and a pointer to the function is stored.

We know that the function receives the address to the module base of kernel32.dll. The address is passed via the ECX register. Additionally the function receives the precomputed hash via the EDX register. The first steps in the program are to store the passed arguments into edi respectively in var\_4. Then the function goes over the memory region and reads in the PE

structure looking for the export directory. To better understand what's going on here and what offsets are used take a look at this (huge) diagram:

<https://raw.githubusercontent.com/corkami/pics/master/binary/pe102/pe102.svg>

```
; Attributes: bp-based frame

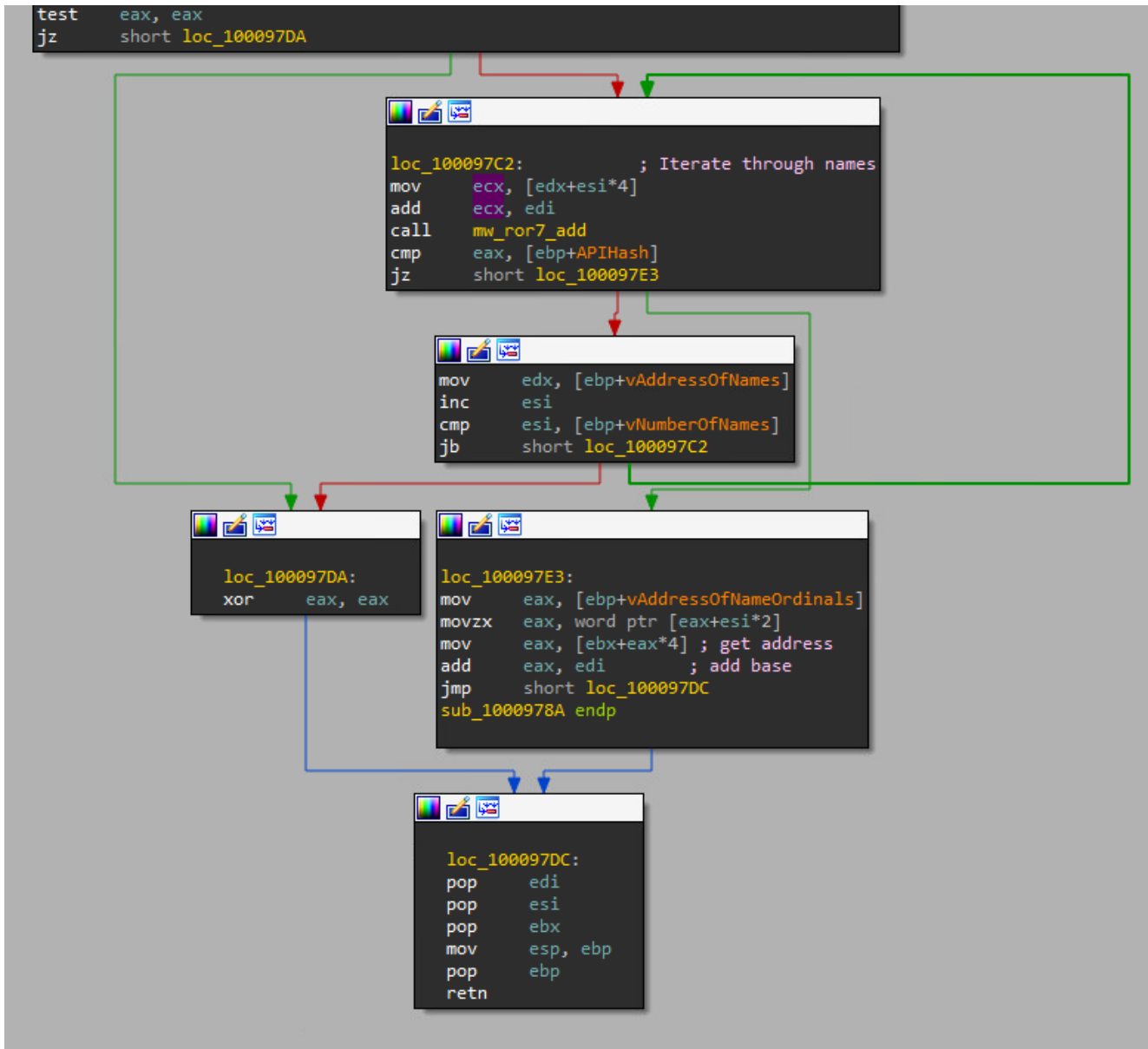
sub_1000978A proc near

vAddressOfNameOrdinals= dword ptr -10h
vAddressOfNames= dword ptr -0Ch
vNumberOfNames= dword ptr -8
APIHash= dword ptr -4

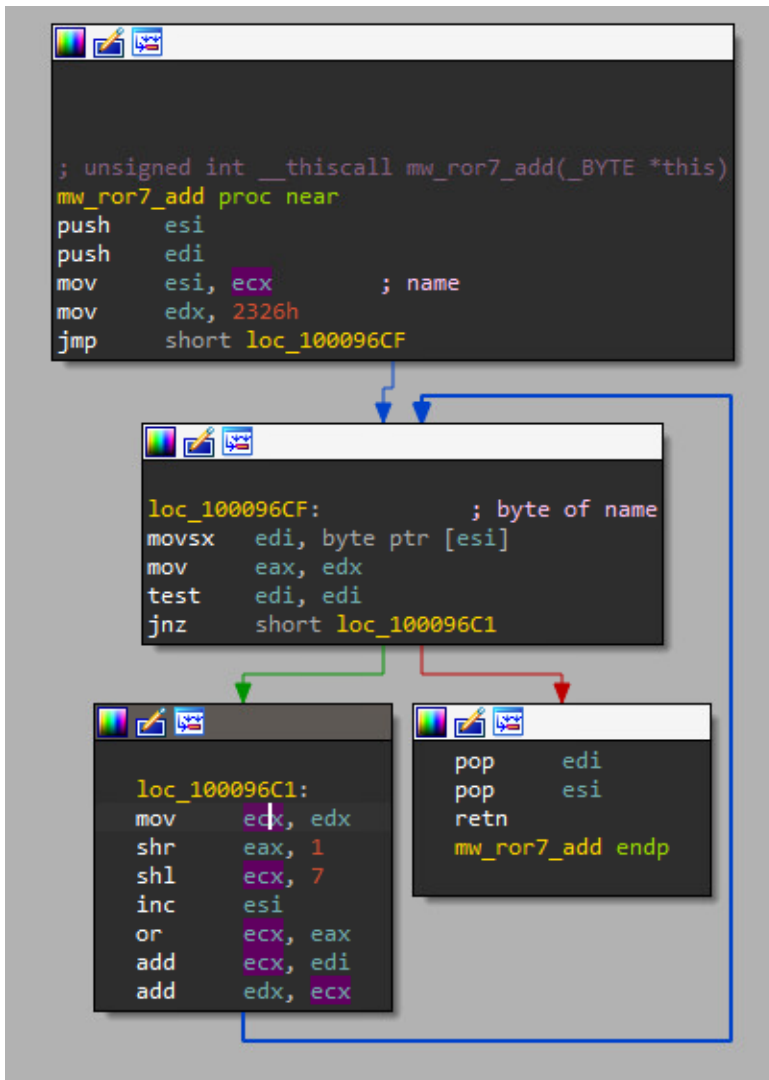
push    ebp
mov     ebp, esp
sub     esp, 10h
push    ebx
push    esi
push    edi
mov     edi, ecx          ; MZ header
mov     [ebp+APIHash], edx
xor     esi, esi
mov     eax, [edi+IMAGE_DOS_HEADER.e_lfanew]
mov     eax, [eax+edi+IMAGE_NT_HEADERS.OptionalHeader.DataDirectory.VirtualAddress]
add     eax, edi          ; MZ header + VA of DataDirectory
mov     edx, [eax+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
mov     ebx, [eax+IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
add     edx, edi          ; MZ header + AddressOfNames
mov     ecx, [eax+IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
add     ebx, edi          ; MZ header + AddressOfNameOrdinals
mov     eax, [eax+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
add     ecx, edi          ; MZ header + NumberOfNames
mov     [ebp+vAddressOfNames], edx
mov     [ebp+vAddressOfNameOrdinals], ecx
mov     [ebp+vNumberOfNames], eax
test    eax, eax
jz     short loc_100097DA

loc_100097C2:          ; Iterate through names
mov     ecx, [edx+esi*4]
```

Now the function gets the first name from the export directory ( $edx+esi*4$ ) and calls the function `sub_100096B6`. The function name (better: the address of the function name) is passed via the ECX register.



And here we have find the hashing algorithm. First the address of the function name is stored in ESI. Then we move the constant 2326 (hex) into EDX register and jump down to loc\_100096CF where we get the first char by utilizing „movsx edi, byte ptr [esi]“. Then the constant 2326h is copied into EAX. With „test edi, edi“ the function checks if there are still characters in the function name left or if the string termination („0“) is reached.



## Pseudocode of the hashing algorithm

If we remove some of the optimizations by the compiler and do the shr / shl operations inline (these operations can only be done on registers therefore the compiler had to assemble it this way) then we get a very simple code. Basically its just starting with a constant (I called iv for Initialization Vector) and then shifts right and left and adds the current char. This way we go over all the chars in the string until we reach the end of the string. To simplify things I use a fixed string of „GetTempPathW“. To account for the EDX register (which is 32 bytes long) we have to make sure that we stay inside this range and therefore have to do an AND operation with 0xFFFFFFFF.

```

iv = 0x2326
name = "GetTempPathW"
hash = 0
for i in range(len(name)):
    iv = (iv + (iv >> 1 | iv << 7) + ord(name[i])) & 0xFFFFFFFF
    hash = iv

print(hex(hash))

```

## Resolve the called API hashes

If we jump over one of these function calls we see the pointer in the EAX being returned. So we can jump over all the function call and make notes of the resolved API hashes. You could write a IDAPython script if you want but there are not many hashes so I decided to do it manually.

Register	Value	Comment
EAX	770AD53C	<kernel32.GetTempPathW>
EBX	FFFFFFB8	
ECX	FFFF5F74	
EDX	7FC01DAE	
EBP	0014F940	
ESP	0014F4C0	
ESI	00001271	
EDI	00000071	'q'

Address	Disassembly	API Hash
10009856	8945 FC mov dword ptr ss:[ebp-4],eax	
10009859	BA AE1DC07F mov edx,7FC01DAE	GetTempPathW
1000985E	8B4D FC mov ecx,dword ptr ss:[ebp-4]	
10009861	E8 24FFFFFF call swfmwfkkeh.1000978A	
10009866	8945 B4 mov dword ptr ss:[ebp-4C],eax	
10009869	BA 1A727FFF mov edx,FF7F721A	GetModuleFileNameW
1000986E	8B4D FC mov ecx,dword ptr ss:[ebp-4]	
10009871	E8 14FFFFFF call swfmwfkkeh.1000978A	
10009876	8945 94 mov dword ptr ss:[ebp-6C],eax	
10009879	BA 66A3D67F mov edx,7FD6A366	LoadLibraryW
1000987E	8B4D FC mov ecx,dword ptr ss:[ebp-4]	
10009881	E8 04FFFFFF call swfmwfkkeh.1000978A	
10009886	8945 B8 mov dword ptr ss:[ebp-48],eax	
10009889	8D45 D8 lea eax,dword ptr ss:[ebp-28]	
1000988C	50 push eax	
1000988D	FF55 B8 call dword ptr ss:[ebp-48]	
10009890	BA 3A655A7F mov edx,7F5A653A	shlwapi.PathAppendW
10009895	8BC8 mov ecx,eax	
10009897	E8 EEFEEEEFF call swfmwfkkeh.1000978A	
1000989C	8945 B0 mov dword ptr ss:[ebp-50],eax	
1000989F	BA 78A0917F mov edx,7F91A078	ExitProcess
100098A4	8B4D FC mov ecx,dword ptr ss:[ebp-4]	
100098A7	E8 DEFEEFFFF call swfmwfkkeh.1000978A	
100098AC	8945 98 mov dword ptr ss:[ebp-68],eax	
100098AF	BA 2336E67F mov edx,7FE63623	CreateFileW
100098B4	8B4D FC mov ecx,dword ptr ss:[ebp-4]	
100098B7	E8 CEFEEFFFF call swfmwfkkeh.1000978A	
100098BC	8945 AC mov dword ptr ss:[ebp-54],eax	
100098BF	BA 7F72BD7F mov edx,7FBD727F	GetFileSize
100098C4	8B4D FC mov ecx,dword ptr ss:[ebp-4]	
100098C7	E8 BEFEFFFF call swfmwfkkeh.1000978A	
100098CC	8945 A8 mov dword ptr ss:[ebp-58],eax	
100098CF	BA DD7AB47F mov edx,7FB47ADD	VirtualAlloc
100098D4	8B4D FC mov ecx,dword ptr ss:[ebp-4]	
100098D7	E8 AEFEEFFFF call swfmwfkkeh.1000978A	
100098DC	8945 A4 mov dword ptr ss:[ebp-5C],eax	
100098DF	BA 40F8E77F mov edx,7FE7F840	ReadFile
100098E4	8B4D FC mov ecx,dword ptr ss:[ebp-4]	
100098E7	E8 9EFEFFFF call swfmwfkkeh.1000978A	
100098EC	8945 A0 mov dword ptr ss:[ebp-60],eax	
100098EF	BA FBF1E17F mov edx,7FE1F1FB	CloseHandle
100098F4	8B4D FC mov ecx,dword ptr ss:[ebp-4]	
100098F7	E8 8EFEFFFF call swfmwfkkeh.1000978A	
100098FC	8945 9C mov dword ptr ss:[ebp-64],eax	

From what we see we can expect some file operations.

After all necessary APIs are resolved we can see that the malware does the same push/pop trick as before – so we do the same and run to the end and jump to the address in the dump

```

00100098FF 6A 77          push 77
0010009901 58             pop eax
0010009902 66:8945 BC    mov word ptr ss:[ebp-44],ax
0010009906 6A 36          push 36
0010009908 58             pop eax
0010009909 66:8945 BE    mov word ptr ss:[ebp-42],ax
001000990D 6A 36          push 36
001000990F 58             pop eax
0010009910 66:8945 C0    mov word ptr ss:[ebp-40],ax
0010009914 6A 7A          push 7A
0010009916 58             pop eax
0010009917 66:8945 C2    mov word ptr ss:[ebp-3E],ax
001000991B 6A 6C          push 6C
001000991D 58             pop eax
001000991E 66:8945 C4    mov word ptr ss:[ebp-3C],ax
0010009922 6A 73          push 73
0010009924 58             pop eax
0010009925 66:8945 C6    mov word ptr ss:[ebp-3A],ax
0010009929 6A 71          push 71
001000992B 58             pop eax
001000992C 66:8945 C8    mov word ptr ss:[ebp-38],ax
0010009930 6A 70          push 70
0010009932 58             pop eax
0010009933 66:8945 CA    mov word ptr ss:[ebp-36],ax
0010009937 6A 6E          push 6E
0010009939 58             pop eax
001000993A 66:8945 CC    mov word ptr ss:[ebp-34],ax
001000993E 6A 79          push 79
0010009940 58             pop eax
0010009941 66:8945 CE    mov word ptr ss:[ebp-32],ax
0010009945 6A 75          push 75
0010009947 58             pop eax
0010009948 66:8945 D0    mov word ptr ss:[ebp-30],ax
001000994C 6A 65          push 65
001000994E 58             pop eax
001000994F 66:8945 D2    mov word ptr ss:[ebp-2E],ax
0010009953 6A 36          push 36
0010009955 58             pop eax
0010009956 66:8945 D4    mov word ptr ss:[ebp-2C],ax
001000995A 33C0          xor eax,eax
001000995C 66:8945 D6    mov word ptr ss:[ebp-2A],ax

```

There we see a strange string „w66zlsqpnue6“.

```

0014F8FC 77 00 36 00 | 36 00 7A 00 | 6C 00 73 00 | 71 00 70 00 | w.6.6.z.l.s.q.p.
0014F90C 6E 00 79 00 | 75 00 65 00 | 36 00 00 00 | 53 00 68 00 | n.y.u.e.6...S.h.
0014F91C 6C 00 77 00 | 61 00 70 00 | 69 00 2E 00 | 64 00 6C 00 | l.w.a.p.i...d.l.
0014F92C 65 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 |

```

## Reading and decrypting main payload

Lets continue debugging. And find out what this string is used for. Next we have a call to a memory region and as you might have expected its an API call. We call GetTempPathW.

```

EIP → 10009967 68 03010000  push 103
1000996C FF55 B4       call dword ptr ss:[ebp-4C]
1000996F 8D45 BC       lea eax,dword ptr ss:[ebp-44]
10009972 50           push eax
dword ptr ss:[ebp-4C]=[0014F8F4 "<0\nw+H\tww"]=<kernel32.GetTempPathW>

```

With the next call we are able to figure out what the previous strange string means. The string is appended to the result of GetTempPathW and therefore it must be a file. If you recall ebp-44 is the string and ebp-480 contains the string of the temp folder.

```

EIP → 1000996C FF55 B4       call dword ptr ss:[ebp-4C]
1000996F 8D45 BC       lea eax,dword ptr ss:[ebp-44]
10009972 50           push eax
10009973 8D85 80FBFFF  lea eax,dword ptr ss:[ebp-480]
10009979 50           push eax
1000997A FF55 B0       call dword ptr ss:[ebp-50]
1000997D 6A 00        push 0
dword ptr ss:[ebp-50]=[0014F8F0 <&PathAppendW>]=<shlwapi.PathAppendW>

```

After the call the newly formed string is stored at ebp-480 and we can follow this memory in the dump and see the final result



0014F4C0	43 00 3A 00	5C 00 55 00	73 00 65 00	72 00 73 00	C.:.\U.s.e.r.s.
0014F4D0	5C 00 50 00	65 00 74 00	65 00 72 00	5C 00 41 00	\.P.e.t.e.r.\.A.
0014F4E0	70 00 70 00	44 00 61 00	74 00 61 00	5C 00 4C 00	p.p.D.a.t.a.\.L.
0014F4F0	6F 00 63 00	61 00 6C 00	5C 00 54 00	65 00 6D 00	o.c.a.l.\.T.e.m.
0014F500	70 00 5C 00	77 00 36 00	36 00 7A 00	6C 00 73 00	p.\.w.6.6.z.l.s.
0014F510	71 00 70 00	6E 00 79 00	75 00 65 00	36 00 00 00	q.p.n.y.u.e.6...

Next we have a call to CreateFileW. Remember that the parameters are pushed to the stack in reverse order. One interesting fact about this call is the value 80000000 which is a constant for GENERIC\_READ which means that the file must already exist or we will get an error. At this point we can assume that the NSIS installer will copy the file over to the temp directory. To proceed with our debugging we have to copy the file ourselves. You can find the file in the „root“ folder.

1000997D	6A 00	push 0
1000997F	68 80000000	push 80
10009984	6A 03	push 3
10009986	6A 00	push 0
10009988	6A 07	push 7
1000998A	68 00000080	push 80000000
1000998F	8D85 80FBFFFF	lea eax,dword ptr ss:[ebp-480]
10009995	50	push eax
10009996	FF55 AC	call dword ptr ss:[ebp-54]
10009999	8945 F0	mov dword ptr ss:[ebp-10],eax
1000999C	837D F0 FF	cmp dword ptr ss:[ebp-10],FFFFFFFF
100099A0	75 02	jne swfmwfkkeh.100099A4

dword ptr ss:[ebp-54]=[0014F8EC &CreateFileW]=<kernel32.CreateFileW>

If the call succeeds EAX will contain the handle to the file. The result of the call is stored and checked in the next line.

10009996	FF55 AC	call dword ptr ss:[ebp-54]
10009999	8945 F0	mov dword ptr ss:[ebp-10],eax
1000999C	837D F0 FF	cmp dword ptr ss:[ebp-10],FFFFFFFF
100099A0	75 02	jne swfmwfkkeh.100099A4
100099A2	EB 66	jmp swfmwfkkeh.10009A0A
100099A4	6A 00	push 0
100099A6	FF75 F0	push dword ptr ss:[ebp-10]
100099A9	FF55 A8	call dword ptr ss:[ebp-58]
100099AC	8945 F8	mov dword ptr ss:[ebp-8],eax
100099AF	837D F8 FF	cmp dword ptr ss:[ebp-8],FFFFFFFF
100099B3	75 02	jne swfmwfkkeh.100099B7
100099B5	EB 53	jmp swfmwfkkeh.10009A0A
100099B7	6A 04	push 4
100099B9	68 00300000	push 3000
100099BE	FF75 F8	push dword ptr ss:[ebp-8]
100099C1	6A 00	push 0
100099C3	FF55 A4	call dword ptr ss:[ebp-5C]
100099C6	8945 F4	mov dword ptr ss:[ebp-C],eax
100099C9	837D F4 00	cmp dword ptr ss:[ebp-C],0
100099CD	75 02	jne swfmwfkkeh.100099D1
100099CF	EB 39	jmp swfmwfkkeh.10009A0A

If this is successful the malware gets the size of the file with a call to GetFileSize

100099A6	FF75 F0	push dword ptr ss:[ebp-10]
100099A9	FF55 A8	call dword ptr ss:[ebp-58]
100099AC	8945 F8	mov dword ptr ss:[ebp-8],eax
100099AF	837D F8 FF	cmp dword ptr ss:[ebp-8],FFFFFFFF
100099B3	75 02	jne swfmwfkkeh.100099B7
100099B5	EB 53	jmp swfmwfkkeh.10009A0A
100099B7	6A 04	push 4
100099B9	68 00300000	push 3000
100099BE	FF75 F8	push dword ptr ss:[ebp-8]
100099C1	6A 00	push 0
100099C3	FF55 A4	call dword ptr ss:[ebp-5C]
100099C6	8945 F4	mov dword ptr ss:[ebp-C],eax
100099C9	837D F4 00	cmp dword ptr ss:[ebp-C],0

dword ptr ss:[ebp-58]=[0014F8E8 &GetFileSize]=<kernel32.GetFileSize>

The same logic as above is used to check if the file size is not equal 0. Next we allocate virtual memory with a call to VirtualAlloc and pass the size (stored at ebp-8) to the function. The allocated memory is stored in ebp-C.



```

100099B5 EB 53 jmp swfmwfkkeh.10009A0A
100099B7 6A 04 push 4
100099B9 68 00300000 push 3000
100099BE FF75 F8 push dword ptr ss:[ebp-8]
100099C1 6A 00 push 0
100099C3 FF55 A4 call dword ptr ss:[ebp-5C]
100099C6 8945 F4 mov dword ptr ss:[ebp-C],eax
100099C9 837D F4 00 cmp dword ptr ss:[ebp-C],0
100099CD 75 02 jne swfmwfkkeh.100099D1
100099CF EB 39 jmp swfmwfkkeh.10009A0A
100099D1 6A 00 push 0

```

dword ptr ss:[ebp-5C]=[0014F8E4 <&VirtualAlloc>]=<kernel32.VirtualAlloc>

We do another check if the function succeeded and continue with a call to read file. The destination buffer is the just allocated memory region. In my case its 0x750000.

```

100099D1 6A 00 push 0
100099D3 8D45 90 lea eax,dword ptr ss:[ebp-70]
100099D6 50 push eax
100099D7 FF75 F8 push dword ptr ss:[ebp-8]
100099DA FF75 F4 push dword ptr ss:[ebp-C]
100099DD FF75 F0 push dword ptr ss:[ebp-10]
100099E0 FF55 A0 call dword ptr ss:[ebp-60]
100099E3 85C0 test eax,eax

```

[ebp-70]: ".M"  
eax: "&".M"

After the file is read we can take a look at the memory region.

```

00750000 87 45 C2 F2 51 3D 46 96 21 14 48 6F 36 2F B5 27 .EÄbQ=F.!..Ho6/u'
00750010 B3 86 E6 33 23 6C 01 EF 9C 15 7F B8 C8 E3 A6 D2 *.æ3#!.i...Eä!0
00750020 58 30 CB 84 14 22 6E B6 E3 90 29 FB C1 D0 72 48 [0E.."ñã.úÄprH
00750030 92 0C C4 1F 53 52 7F 6C BE 4E E5 DA FC 71 9C 3D ..A.SR.l%NäÜüq.=
00750040 DA 1E A1 DD 87 80 C7 77 AD 5E 87 51 23 D5 77 AA Ú.iÿ..Çw.À.Q#0wª
00750050 F8 F2 9B 06 B4 48 42 27 E3 F0 7E D1 1F F1 6B 5F øb..HB'äö~N.ñk
00750060 C4 25 F0 09 24 04 91 F1 12 07 AD 83 D7 C0 F1 17 Å%ð.$..ñ...xÅñ.
00750070 E5 78 96 F4 0E 90 82 40 3A 8B 47 B4 6C BD B0 F1 äx.ö...@:.G'1½°ñ
00750080 A8 CA CC 6A 0B 1F 55 84 40 D9 5F 1E 4E C9 F8 E8 Èij..U.@U..NÉøè
00750090 C4 1F 98 CC 6C 85 7F 9D 70 EC 9D 5B 03 EC 1B 34 Ä..Ij...pì.[.i.4
007500A0 8A 98 C5 88 44 41 33 CA 04 8A 1C 31 86 B8 6E 2A ..Ä.DA3È...i.»nª
007500B0 97 A8 19 28 0C 09 01 13 55 2D F8 3C 2E 80 63 3F .(.U-ø<..c?
007500C0 0A AD A2 5E 75 EF 1C 71 33 BA EA 24 87 B4 F5 45 ..c^uî.q3°è$.øE
007500D0 AE AB 11 F6 70 8E 8C B8 EE D7 B8 2D 17 F2 20 28 è«..öp..ix-.ò(
007500E0 55 08 C3 C1 E3 40 2E F3 40 3F D6 09 63 BF 7A 66 U.AAäe.ø@?0.cjzF
007500F0 93 1A 92 F9 2D 45 4E 0E B2 32 3D 98 82 89 09 E9 ...ù-EN.ª2=...é
00750100 13 23 B8 F2 4D 3D 46 96 70 4A 48 6F E9 88 27 27 .#øM=F.pJHoé.'
00750110 C2 20 67 2A 23 6C 01 EF 70 15 7F B8 CA E3 76 42 Å g*#!.ip..EävB

```

This does look like encrypted data but lets continue our analysis. After reading the file we close the handle to the file and jump into another function at 10009A0E. Notice that the functions receives two parameters. First the file size (ebp-8) and the allocated region of memory (ebp-C).

```

100099E0 FF55 A0 call dword ptr ss:[ebp-60]
100099E3 85C0 test eax,eax
100099E5 75 02 jne swfmwfkkeh.100099E9
100099E7 EB 21 jmp swfmwfkkeh.10009A0A
100099E9 FF75 F0 push dword ptr ss:[ebp-10]
100099EC FF55 9C call dword ptr ss:[ebp-64]
100099EF FF75 F8 push dword ptr ss:[ebp-8]
100099F2 FF75 F4 push dword ptr ss:[ebp-C]
100099F5 E8 14000000 call swfmwfkkeh.10009A0E
100099FA 8945 F4 mov dword ptr ss:[ebp-C],eax
100099FD FF75 F4 push dword ptr ss:[ebp-C]
10009A00 E8 8F020000 call swfmwfkkeh.10009C94
10009A05 6A 00 push 0

```

Close Handle

In the function we see a familiar code structure – it has great similarities with the first decryption loop. First there is a pointer initialized with zero stored as local variable at ebp-8. Then the pointer is incremented by one and check against the argument at ebp+C which is the file size. If the pointer value is lower than the size of data (jae) the encryption will continue otherwise we jump to 10009C8B. After the jump is NOT taken we see that the value at ebp+8 (the data itself) is moved into eax and then the pointer (counter) gets added.

```

10009A0E 55          push ebp
10009A0F 8BEC       mov ebp,esp
10009A11 51        push ecx
10009A12 51        push ecx
10009A13 8365 F8 00 and dword ptr ss:[ebp-8],0
10009A17 8365 F8 00 and dword ptr ss:[ebp-8],0
10009A1B EB 07     jmp swfmwfkkeh.10009A24
10009A1D 8B45 F8    mov eax,dword ptr ss:[ebp-8]
10009A20 40        inc eax
10009A21 8945 F8    mov dword ptr ss:[ebp-8],eax
10009A24 8B45 F8    mov eax,dword ptr ss:[ebp-8]
10009A27 3B45 0C    cmp eax,dword ptr ss:[ebp+C]
10009A2A 74 07     jae swfmwfkkeh.10009C8B
10009A30 8B45 08    mov eax,dword ptr ss:[ebp+8]
10009A33 0345 F8    add eax,dword ptr ss:[ebp-8]
10009A36 8A00     mov al,byte ptr ds:[eax]
10009A38 8B45 FF    mov byte ptr ss:[ebp-1],al
10009A3B 0FB645 FF movzx eax,byte ptr ss:[ebp-1]
10009A3F 05 B2000000 add eax,B2
10009A44 8B45 FF    mov byte ptr ss:[ebp-1],al
10009A47 0FB645 FF movzx eax,byte ptr ss:[ebp-1]
10009A4B D1F8     sar eax,1
10009A4D 0FB64D FF movzx ecx,byte ptr ss:[ebp-1]
10009A51 C1E1 07    shl ecx,7
10009A54 0BC1     or eax,ecx
10009A56 8B45 FF    mov byte ptr ss:[ebp-1],al
10009A59 0FB645 FF movzx eax,byte ptr ss:[ebp-1]
10009A5D 2D A4000000 sub eax,A4
10009A62 8B45 FF    mov byte ptr ss:[ebp-1],al
10009A65 0FB645 FF movzx eax,byte ptr ss:[ebp-1]
10009A69 3345 F8    xor eax,dword ptr ss:[ebp-8]
10009A6C 8B45 FF    mov byte ptr ss:[ebp-1],al
10009A6F 0FB645 FF movzx eax,byte ptr ss:[ebp-1]
10009A73 05 9E000000 add eax,9E
10009A78 8B45 FF    mov byte ptr ss:[ebp-1],al
10009A7B 0FB645 FF movzx eax,byte ptr ss:[ebp-1]
10009A7F 35 E5000000 xor eax,E5

```

Pointer

Size of data

Huge encryption loop

As we did with the previous encryption loop we don't want to dig into the algorithm and just see what's happening. We know that this code will manipulate the read file so go to address 10009C8B and set a breakpoint. After the decryption is complete we can see a MZ header in the dump.

```

Address  REX  ASCII
00750000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....ÿÿ..
00750010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .....@.....
00750020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00750030 00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00 .....
00750040 0E 1F BA 0E 00 84 09 CD 21 B8 01 4C CD 21 54 68 ..°.!.!..LI!Th
00750050 69 73 20 70 72 6F 67 72 61 60 20 63 61 6E 6E 6F is program canno
00750060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
00750070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode...$.
00750080 09 F0 B1 DC 4D 91 DF 8F 4D 91 DF 8F 4D 91 DF 8F .ð±ÜM.ß.M.ß.M.ß.
00750090 F9 0D 2E 8F 44 91 DF 8F F9 0D 2C 8F 35 91 DF 8F ù...D.ß.ù...5.ß.
007500A0 F9 0D 2D 8F 55 91 DF 8F 76 CF DC 8E 5C 91 DF 8F ù.-.U.ß.vIU.\.ß.
007500B0 76 CF DB 8E 5C 91 DF 8F 76 CF DA 8E 6F 91 DF 8F vIÖ.\.ß.vIÜ.o.ß.
007500C0 53 C3 4C 8F 4F 91 DF 8F 44 E9 4C 8F 44 91 DF 8F SÄL.O.ß.DéL.D.ß.
007500D0 4D 91 DE 8F 21 91 DF 8F DA CF D6 8E 4C 91 DF 8F M.p.! .ß.ÜIÖ.L.ß.
007500E0 DF CF 20 8F 4C 91 DF 8F DA CF DD 8E 4C 91 DF 8F ßI .L.ß.ÜIÿ.L.ß.
007500F0 52 69 63 68 4D 91 DF 8F 00 00 00 00 00 00 00 00 RichM.ß.....
00750100 00 00 00 00 00 00 00 00 50 45 00 00 4C 01 05 00 .....PE..L...
00750110 06 76 8B 61 00 00 00 00 00 00 00 00 E0 00 03 01 .v.a.....ä...

```

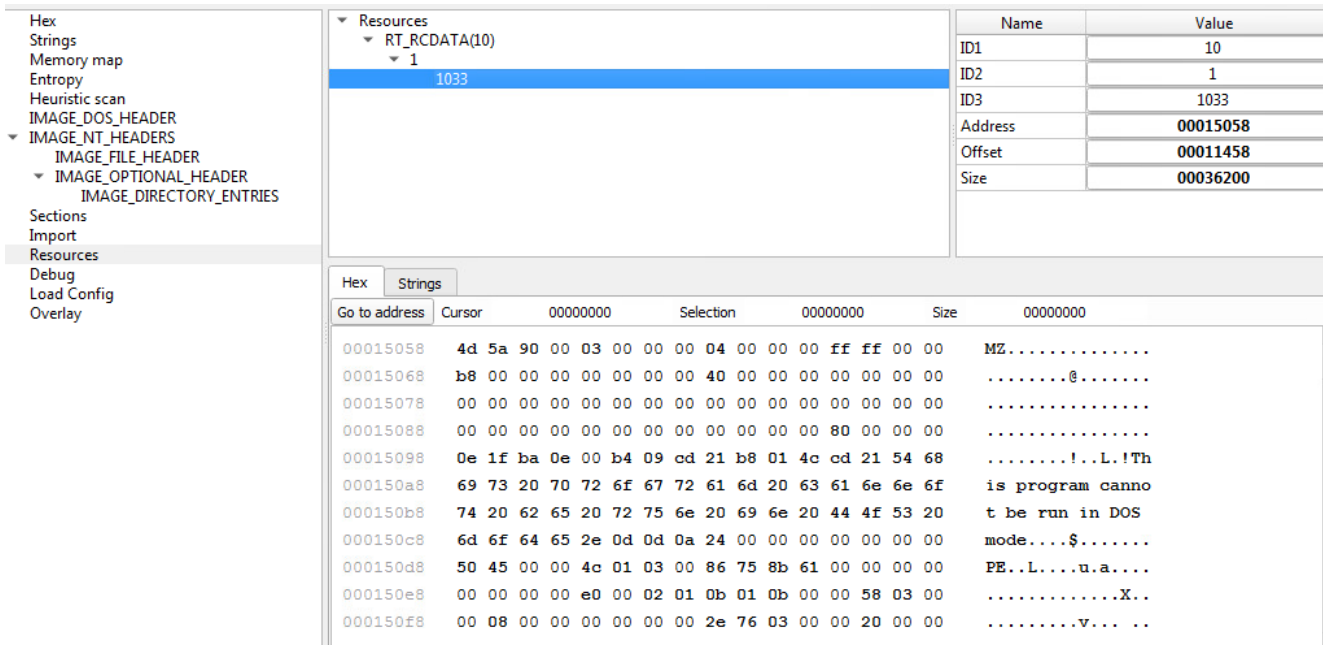
So right-click on the address 750000 and follow in memory map. Then right-click the address again and choose „Dump memory to file“. We want to stop our analysis of the initial sample here and continue with the dumped PE.

### Analyzing the dumped PE

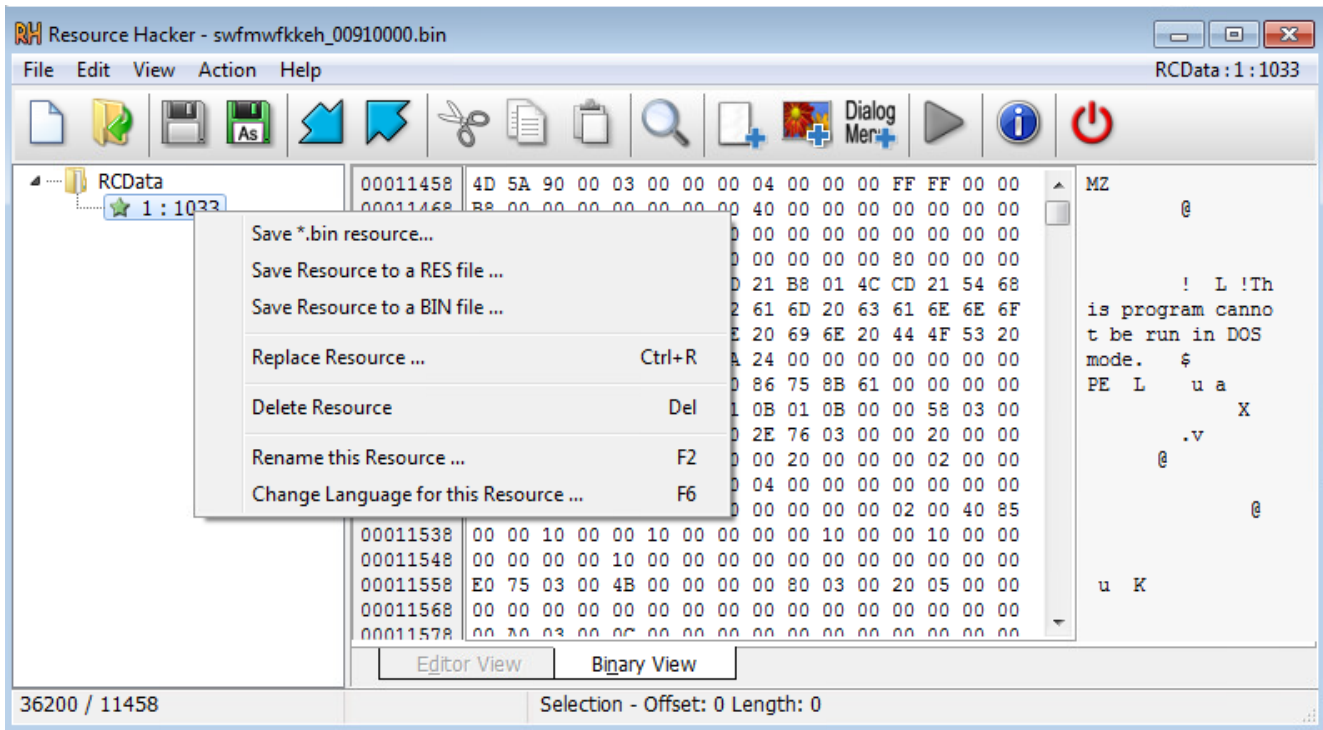
We once again start with some basic static analysis and open the file in DiE.

compiler	Microsoft Visual C/C++ (2015 v.14.0)[-]	S	
linker	Microsoft Linker(14.0, Visual Studio 2015 14.0*)[EXE32]	S	?

Immediately we can see that the PE imports APIs to handle resources so lets check if there is something interesting.



And wow ... there is an unencrypted PE inside of the resource section. To dump this PE we will utilize Resource Hacker and dump the file via „Save Resource to a BIN file...“



## Analyzing the dumped resource

Once again we open the dumped PE in DiE and see that its written in .NET and obfuscated.

protector	Obfuscator(1.0)[-]	S	
library	.NET(v4.0.30319)[-]	S	
compiler	VB.NET(-)[-]	S	
linker	Microsoft Linker(11.0)[EXE32]	S	?

From this point on I just dropped this sample into the CAPE and got a hit on AgentTeslaV3 YARA signatures.

Detections	Analysis	
Yara:	Category	Package
<b>AgentTeslaV3</b>	FILE	exe

Mission complete!

## IoCs

<b>NSIS Installer</b>	ce8a9bf908ce35bf0c034c61416109a44f015eabf058b12485450cd40af95fc3
<b>swfmwfkkeh.dll</b>	6d8bc73c6f2ef4ee700fc8bc4088f73a14dab355a2dd4e3e9aa3ddf52f7e946e
<b>Encrypted resource (inside of NSIS data) w66zlsqpnvue6</b>	c02ff5253bf3930f1ee14e088f50c827bf2209f3a7e9f00ed3994fd417d790b2
<b>Dumped PE</b>	9a72e5859b5564cecff5d5a4a929e81595d68aca1972ea2cf0fcf71c518d2cb9
<b>AgentTesla V3</b>	5459e87eb0a39243a35405866b2dca1d57c2c1ee02d24052635fcc48de5d397c