“Funky malware format” found in Ocean Lotus sample

blog.malwarebytes.com/threat-analysis/2019/04/funky-malware-format-found-in-ocean-lotus-sample

Posted: April 19, 2019 by hasherezade April 19, 2019

Recently, at the SAS conference I talked about “Funky malware formats”—atypical executable formats used by malware that are only loaded by proprietary loaders. Malware authors use them in order to make static detection more difficult, because custom formats are not recognized as executable by AV scanners.

Using atypical formats may also slow down the analysis process because the file can’t be parsed out of the box by typical tools. Instead, we need to write custom loaders in order to analyze them freely.

Last year, we described one such format in a post about Hidden Bee. This time, we want to introduce you to another case that we discussed at the SAS Conference. It is a sample of Ocean Lotus, also known as APT 32, a threat group associated with Vietnam.

Sample

49a2505d54c83a65bb4d716a27438ed8f065c709 – the main executable

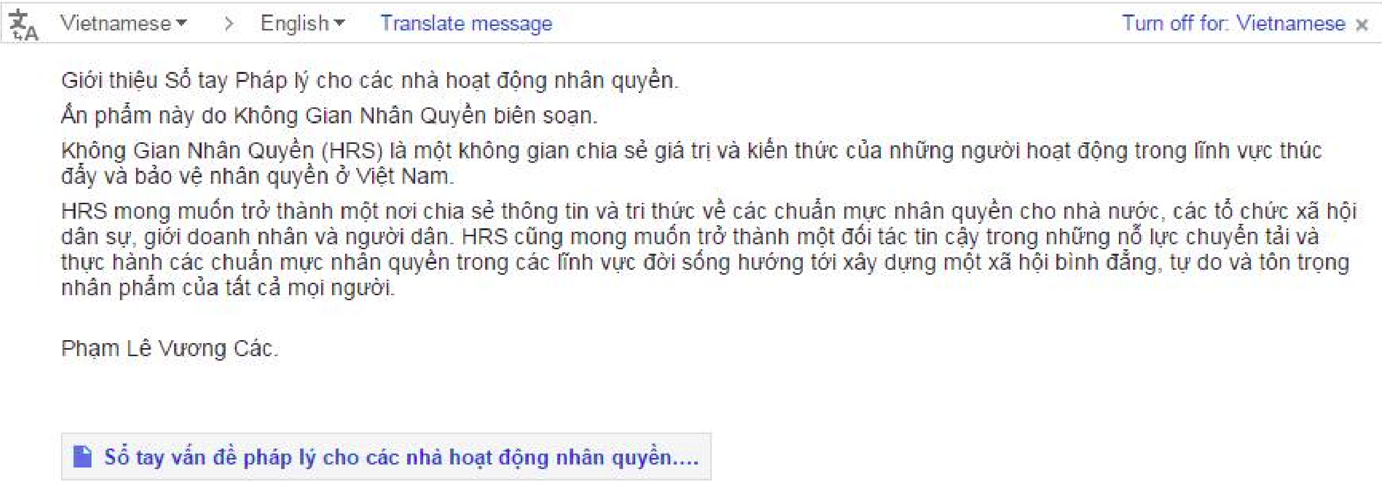
Special thanks to Minh-Triet Pham Tran for providing the material.

Overview

The sample comes with two elements—BLOB and CAB—that are both executables in the same unknown format. The custom format is achieved by conversion from PE format (we can guess it by observing some artifacts typical for PE files, i.e. the manifest) However, the header is fully custom, and the way of loading it has no resemblance with PE. Some of the information from a typical PE (for example, the layout of the sections) is not preserved: sections are shuffled.

Origin

This sample is from June 10, 2017, from the following email:



Content of the phishing email, along with its attachment

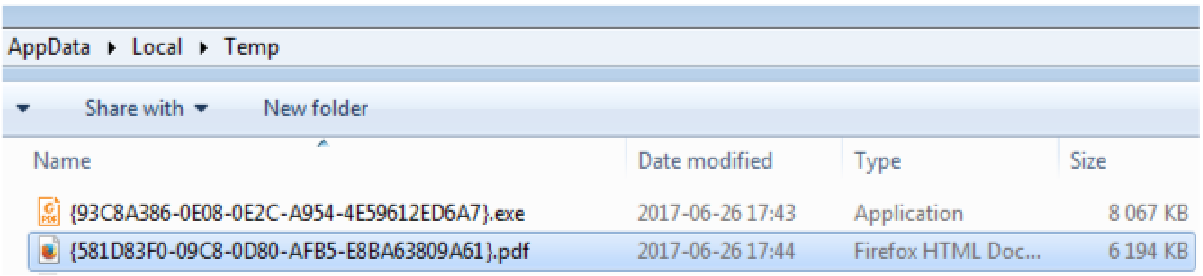
The title "Sổ tay vấn đề pháp lý cho các nhà hoạt động nhân quyền" translates to: "Handbook of legal issues for human rights activists." It's a subject line for a spear phishing campaign targeting Vietnamese activists.

The malicious sample was delivered as an attachment to the email: a zipped executable. The icon tried to imitate a PDF (FoxitPDF reader).

An executable with FoxitFDF icon

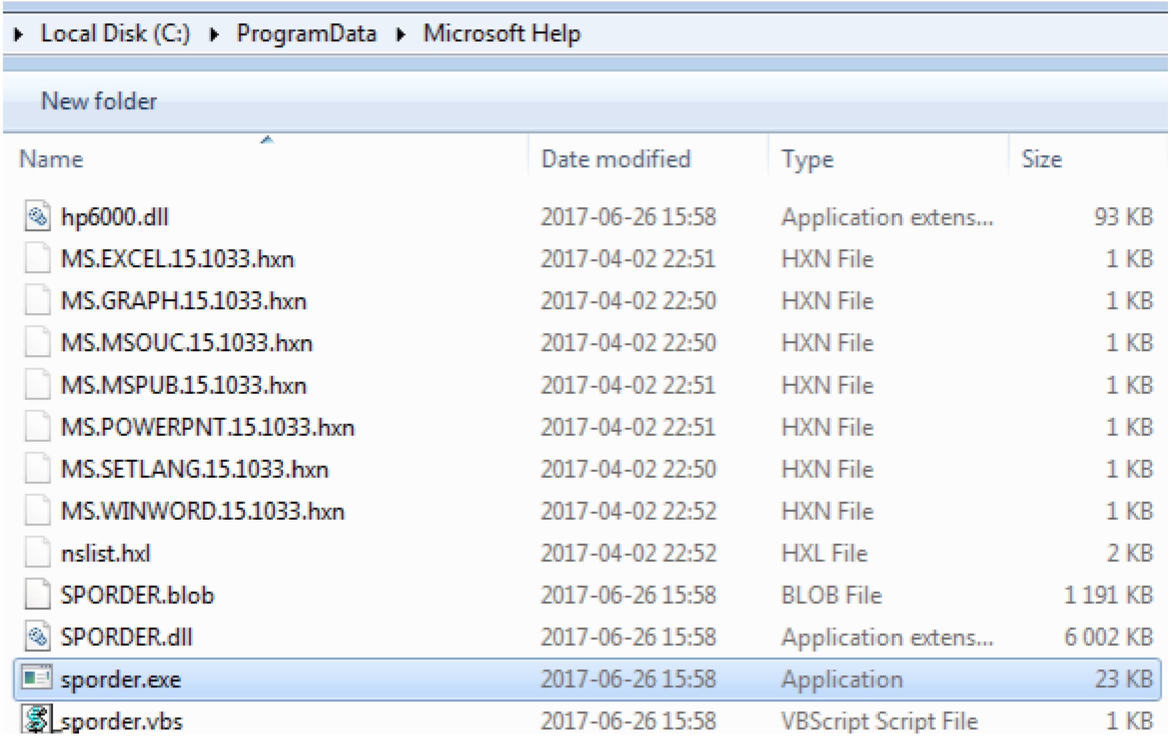
Behavioral analysis

After being run, the sample copies itself into %TEMP%, unpacks, and launches the decoy PDF.



The main executable and the decoy copied to the Temp folder

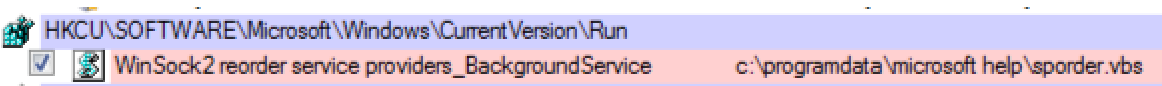
While the user is busy reading the launched document, the dropper unpacks the real payload. It is dropped into *C:\ProgramData\Microsoft Help*:



All the elements of the malware unpacked

The dropper executable is deleted afterwards.

The malware manages to bypass UAC at default level. We can see the application *sporder.exe* running with elevated privileges.
Persistence is provided by a simple Run key, leading to the dropped script:



Added run key (view from Sysinternals Autoruns)

The interesting factor is that the sample has an “expiry date” after which the installer no longer runs.

Internals

The main executable sporder.exe is packed with UPX. It imports the DLL SPORDER.dll:

Disasm: UPX1 | General | DOS Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports | Resources | Secu

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder
1AB0	KERNEL32.DLL	3	FALSE	0	0	0
1AC4	ADVAPI32.dll	1	FALSE	0	0	0
1AD8	COMCTL32.dll	1	FALSE	0	0	0
1AEC	MSVCRT.dll	1	FALSE	0	0	0
1B00	SPORDER.dll	1	FALSE	0	0	0
1B14	USER32.dll	1	FALSE	0	0	0
1B28	WS2_32.dll	1	FALSE	0	0	0

SPORDER.dll [1 entry]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
11F578	WSCWriteProviderOrder	-	-	11F624	-	0

Import table of SPORDER.exe (view from PE-bear)

SPORDER.dll imports another of the dropped DLLs, *hp6000.dll*:

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
9736F	hp6000.dll	1	FALSE	973A2	0	0	97397	973AA

hp6000.dll [1 entry]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
973AA	DllGetClassObject	-	973B2	973B2	-	75

Import table of SPORDER.exe (view from PE-bear)

The key malware functionality is, however, not provided by any of the dropped PE files. They are just used as loaders.

As it turns out, the core is hidden in two unknown files: BLOB and CAB.

Custom formats

The files with extensions BLOB and CAB are obfuscated with XOR. After decoding them, we notice some readable strings of code. However, none of them are valid PE files, and we cannot find any of the typical headers.

BLOB

The BLOB file is obfuscated by XOR. We can see the repeating pattern and use it as an XOR key:

SPORDER.blob

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  50 48 47 03 8B FE A8 E1 8A 99 0E 00 8B 4E B9 E1  PHG.‹ţ¨áŠ™..‹Nąá
00000010  8A 99 0E 00 8B FE A8 E1 8A 99 0E 00 8B FE A8 E1  Š™..‹ţ¨áŠ™..‹ţ¨á
00000020  8A 99 0E 00 8B FE A8 E1 8A 99 0E 00 8B FE A8 E1  Š™..‹ţ¨áŠ™..‹ţ¨á
00000030  8A 99 0E 00 8B FE A8 E1 8A 99 0E 00 8B FE A8 E1  Š™..‹ţ¨áŠ™..‹ţ¨á
00000040  8A 99 0E 00 8B FE A8 E1 8A 99 0E 00 8B FE A8 E1  Š™..‹ţ¨áŠ™..‹ţ¨á
00000050  8A 99 0E 00 8B FE A8 E1 8A 99 0E 00 8B FE A8 E1  Š™..‹ţ¨áŠ™..‹ţ¨á
```

SPORDER.blob (original version), the repeating pattern is selected

As a result, we get the following clear version: 2e68afae82c1c299e886ab0b6b185658

BLOB's header:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  0A D1 49 03 00 00 00 00 00 00 00 00 00 B0 11 00  .ŃI..........°..
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

The BLOB file looks like a processed PE file, however, its sections appear to be in swapped order. The first section seems to be .data, instead of .text.

We can see visible artifacts from the BZIP library and C++ standard library.

CAB

The CAB file is obfuscated with XOR in a similar way, but with a different key:

```
SPORDER.cab
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  D5 31 D8 08 36 49 7B F1 9F E8 01 00 36 39 7A F1  Ő1Ř.6I{ńźč..69zń
00000010  9F E8 01 00 36 49 7B F1 9F E8 01 00 36 49 7B F1  źč..6I{ńźč..6I{ń
00000020  9F E8 01 00 36 49 7B F1 9F E8 01 00 36 49 7B F1  źč..6I{ńźč..6I{ń
```

When we apply the key, we get an analogical clear version: b3f9a8adf0929b2a37db7b396d231110

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  4A D9 D9 08 00 00 00 00 00 00 00 00 00 70 01 00  JŮŮ..........p..
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

This sample also has a custom header, which does not resemble the PE header. However, we found sections inside that are typical for PE files, for example, a manifest.

```
00014040  00 00 00 00 00 00 00 00 04 00 00 00 00 00 01 00  ................
00014050  09 04 00 00 48 00 00 00 58 40 01 00 5A 01 00 00  ....H...X@..Z...
00014060  E4 04 00 00 00 00 00 00 3C 61 73 73 65 6D 62 6C  ä.......<assembl
00014070  79 20 78 6D 6C 6E 73 3D 22 75 72 6E 3A 73 63 68  y xmlns="urn:sch
00014080  65 6D 61 73 2D 6D 69 63 72 6F 73 6F 66 74 2D 63  emas-microsoft-c
00014090  6F 6D 3A 61 73 6D 2E 76 31 22 20 6D 61 6E 69 66  om:asm.v1" manif
000140A0  65 73 74 56 65 72 73 69 6F 6E 3D 22 31 2E 30 22  estVersion="1.0"
000140B0  3E 0D 0A 20 20 3C 74 72 75 73 74 49 6E 66 6F 20  >..  <trustInfo
000140C0  78 6D 6C 6E 73 3D 22 75 72 6E 3A 73 63 68 65 6D  xmlns="urn:schem
000140D0  61 73 2D 6D 69 63 72 6F 73 6F 66 74 2D 63 6F 6D  as-microsoft-com
000140E0  3A 61 73 6D 2E 76 33 22 3E 0D 0A 20 20 20 20 3C  :asm.v3">..    <
000140F0  73 65 63 75 72 69 74 79 3E 0D 0A 20 20 20 20 20  security>..
```

Loader

As it turned out, both files are loaded by hp6000.dll: 67b8d21e79018f1ab1b31e1aba16d201

The loading function is executed in an obfuscated way: when the DllMain is executed, it patches the main executable that loaded the DLL.

First, the file name of the current module is retrieved. Then, the file is read and the address of the entry point is fetched. Then, the analogical module that is loaded in the memory is set as an executable:

```
10001085 push    ebx
10001086 push    edi
10001087 mov     edi, [esi+3Ch]
1000108A mov     ebx, [edi+esi+50h]
1000108E add     edi, esi
10001090 lea     ecx, [esp+21Ch+flOldProtect]
10001094 push    ecx             ; lpflOldProtect
10001095 mov     [esp+220h+flOldProtect], 0
1000109D mov     edx, [edi+50h]
100010A0 push    40h             ; flNewProtect
100010A2 push    edx             ; dwSize
100010A3 push    esi             ; lpAddress
100010A4 add     ebx, esi
100010A6 call    ds:VirtualProtect ; EXECUTE_READ_WRITE
100010A6                         ; size=0x1C000
100010AC test    eax, eax
```

Using VirtualProtect to make the main module writable

Finally, the bytes are patched so that the entry point will redirect back to the appropriate function in the loading DLL:

```
10001130
10001130 loc_10001130:
10001130 mov     cl, 90h
10001132 mov     [eax+esi], cl
10001135 mov     [eax+esi+1], cl
10001139 mov     byte ptr [eax+esi+2], 0B8h
1000113E mov     dword ptr [eax+esi+3], offset to_execute_loader
10001146 mov     byte ptr [eax+esi+7], 0FFh
1000114B mov     byte ptr [eax+esi+8], 0E0h
10001150 mov     [eax+esi+9], cl
```

Patching the entry point of the main module, byte by byte

This is how the entry point of the main module looks after the patch is applied:

	Hex	Disasm	
2570	90	NOP	patch_8
2571	90	NOP	
2572	B81012E86D	MOV EAX, 0X6DE81210	
2577	FFE0	JMP EAX	
2579	90	NOP	

The Entry Point of the main module (sporder.exe) after patching

We see that the Virtual Address (RVA 0x1210 + DLL loading base) of the function within the DLL is moved to EAX, and then the EAX is used as a jump target.

The function that starts at RVA 0x1210 is a loader for BLOB and CAB:

```
10001210 to_execute_loader proc near
10001210
10001210 ms_exc= CPPEH_RECORD ptr -18h
10001210
10001210 ; __unwind { // __except_handler4
10001210 push    ebp
10001211 mov     ebp, esp
10001213 push    0FFFFFFFEh
10001215 push    offset stru_10015078
1000121A push    offset __except_handler4
1000121F mov     eax, large fs:0
10001225 push    eax
```

Beginning of the loading function

This redirection works, thanks to the fact that when the executable is loaded into the memory, before the Entry Point of the main module is hit, all the DLLs that are in its Import Table are loaded, and the DllMain of each is called. Just after the DLLs are loaded, the execution of the main executable starts. And in our case, the patched entry point redirects back to the DLL.

Inside the function loading BLOB and CAB:

```
Filename = 0;
memset(&v7, 0, 0x206u);
GetModuleFileNameW(0, &Filename, 0x104u);
lstrcpyW((LPWSTR)&String2, &Filename);
szLongPath = 0;
memset(&v3, 0, 0x206u);
if ( GetLongPathNameW(&String2, &szLongPath, 0x104u) )
  lstrcpyW((LPWSTR)&String2, &szLongPath);
lstrcpyW((LPWSTR)&pszPath, &String2);
PathStripPathW((LPWSTR)&pszPath);
lstrcpyW(&word_10017C18, &String2);
PathRemoveFileSpecW(&word_10017C18);
load_cab();
lstrcpyW(&szLongPath, &pszPath);
PathRemoveExtensionW(&szLongPath);
String1 = 0;
memset(&v5, 0, 0x206u);
lstrcpyW(&String1, L"Local\\{076B1DB0-2C01-45A5-BD0A-0CF5D6410DCB}");
lstrcatW(&String1, &word_10011AE0);
lstrcatW(&String1, &szLongPath);
if ( get_username(&String1) )
{
  v1 = 0;
  env_var = check_environment_var(&v1);       // set '@' if environment var is empty
  if ( !v1 || !create_process() )
  {
    switch ( env_var )
    {
      case 1:                                 // '@' -> '*'
        set_next_state_and_restart();
        break;
      case 2:
        store_info_set_next_state();          // '*' -> ':'
        break;
      case 3:
        create_mutex1();
        load_blob();
        break;
    }
  }
```

The function loading BLOB and CAB

As you can see, the CAB file is loaded first:

Executing the function loading CAB file (unconditional)

```
100013EB push    offset String2  ; lpString2
100013F0 push    offset word_10017C18 ; lpString1
100013F5 call    esi ; lstrcpyW
100013F7 push    offset word_10017C18 ; pszPath
100013FC call    ds:PathRemoveFileSpecW
10001402 call    load_cab
10001407 push    offset pszPath  ; lpString2
1000140C lea     edx, [esp+62Ch+szLongPath]
10001410 push    edx             ; lpString1
```

Further, we see this function retrieving some environmental variable. This variable is used to store the state of the application, and is shared between consecutive executions. Depending on this state, one of multiple execution paths can be taken.

The name of the variable is created by concatenating:

1. hardcoded string: L"Local\\{076B1DB0-2C01-45A5-BD0A-0CF5D6410DCB}"
2. the name of the executable
3. a local username

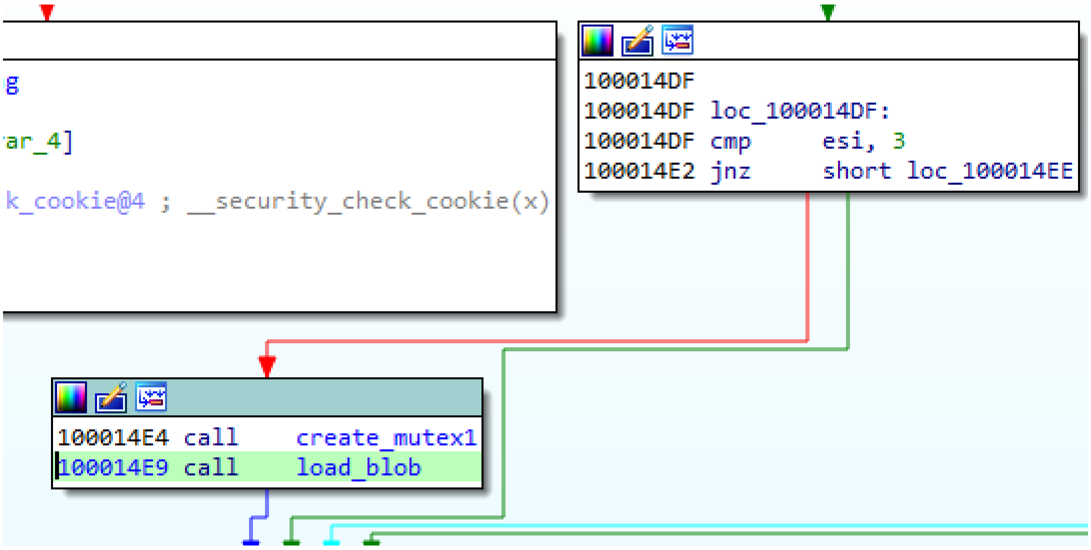
```
0F926FA8  push 0
0F926FAA  push eax
0F926FAB  call sporder.F9283B0
0F926FB0  add esp,C
0F926FB3  push 104
0F926FB8  lea ecx,dword ptr ss:[esp+8]
0F926FBC  push ecx
0F926FBD  push sporder.F938158                                F938158:L"Local\\{076B1DB0-2C01-45A5-BD0A-0CF5D6410DCB}_sporder_tester"
0F926FC2  call dword ptr ds:[<&GetEnvironmentVariableW>]
0F926FC8  mov esi,eax
0F926FCA  call dword ptr ds:[<&GetLastError>]
0F926FD0  test esi,esi
0F926FD2  jne sporder.F927027
0F926FD4  cmp eax,CB
0F926FD9  jne sporder.F92700E
0F926FDB  push sporder.F934218
0F926FE0  push sporder.F938158                                F938158:L"Local\\{076B1DB0-2C01-45A5-BD0A-0CF5D6410DCB}_sporder_tester"
0F926FE5  call dword ptr ds:[<&SetEnvironmentVariableW>]
0F926FEB  push 64
0F926FED  call dword ptr ds:[<&Sleep>]
0F926FF3  mov eax,1
0F926FF8  pop esi
0F926FF9  mov ecx,dword ptr ss:[esp+208]
0F927000  xor ecx,esp
0F927002  call sporder.F9276C9
0F927007  add esp,20C
0F92700D  ret
```

Setting the variable name

The content variable may be one of the following: '@', '*',':'. If it is empty, the first value '@' is set. Those variables are translated to particular states that control the flow.

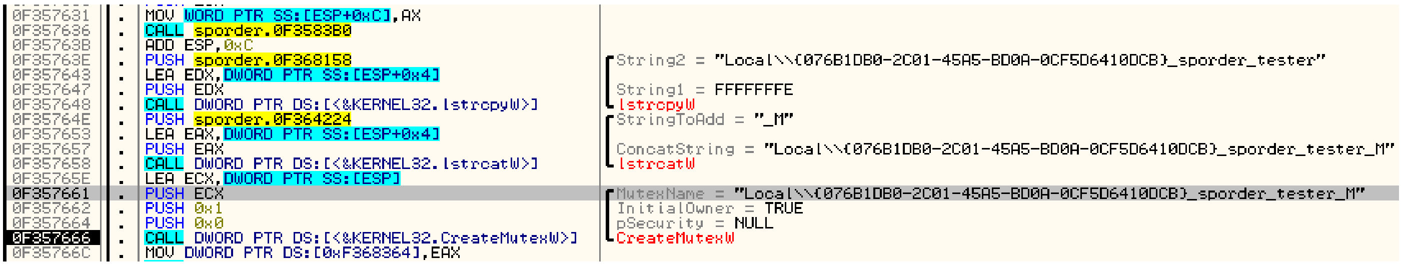
- '@' -> state 1
- '*' -> state 2
- ':' -> state 3

The main process is restarted on each state change. Finally, the state 3 creates mutex and loads the file with the BLOB extension.



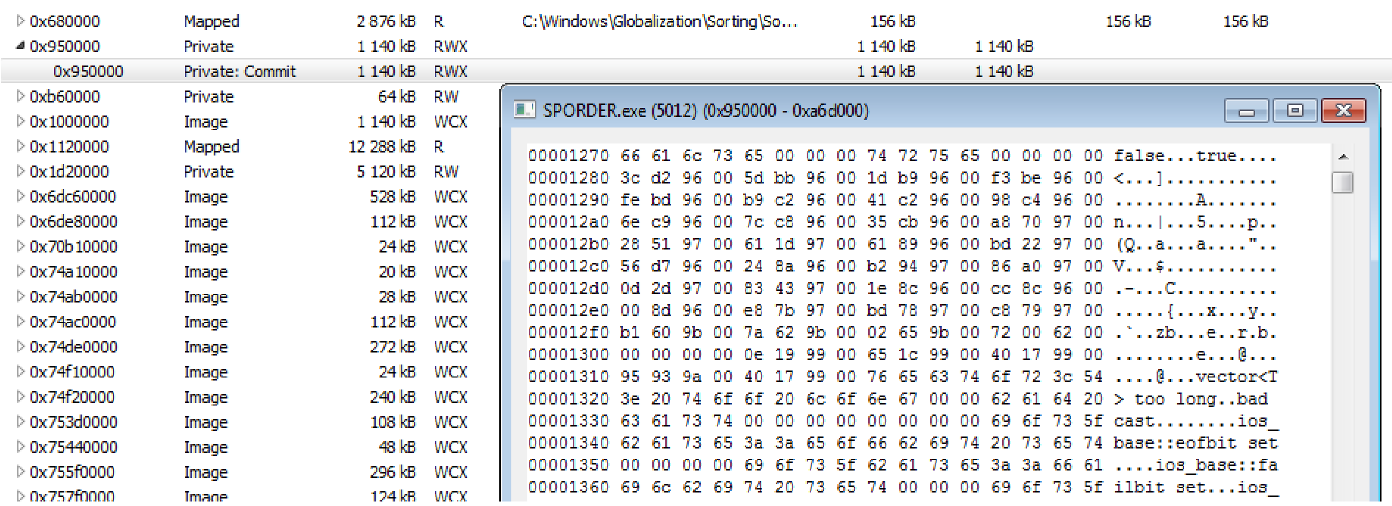
Final state: setting the mutex and loading the BLOB

The mutex name is the same as the variable name, but with a suffix "_M" added:



Setting the mutex

While the application runs, we can see the BLOB being loaded in executable form inside the main module's memory:



Memory of the sporder.exe, view from Process Hacker

By comparing the format that is loaded in the memory with the format that is stored on the disk, we can see that the beginning and the end of the BLOB is skipped in the loading process. So, we can guess that those parts are some headers that contains the information necessary for loading, but not for execution. The header at the beginning of the file will be referenced as Header1, and the one at the end (footer) will be referenced as Header2.

The Header2 file in the memory vs. its equivalent on the disk:

```
16FE0: 00 00 00 00 00 00 00 00 |........      16FE0: 00 00 00 00 00 00 00 00 |........
16FE8: 00 00 00 00 00 00 00 00 |........      16FE8: 00 00 00 00 00 00 00 00 |........
16FF0: 00 00 00 00 00 00 00 00 |........      16FF0: 00 00 00 00 00 00 00 00 |........
16FF8: 00 00 00 00 00 00 00 00 |........      16FF8: 00 00 00 00 00 00 00 00 |........
17000: 00 00 00 00 00 00 00 00 |........      17000: A0 21 00 00 3D 01 00 00 | !..=...
17008: 00 00 00 00 00 00 00 00 |........      17008: 01 00 00 00 EC 1D 01 00 |....ě...
17010: 00 00 00 00 00 00 00 00 |........      17010: 01 00 00 00 FC 58 00 00 |....üX..
17018: 00 00 00 00 00 00 00 00 |........      17018: 01 00 00 00 90 85 00 00 |........
17020: 00 00 00 00 00 00 00 00 |........      17020: 01 00 00 00 EC AA 00 00 |....ěŞ..
17028: 00 00 00 00 00 00 00 00 |........      17028: 01 00 00 00 EC 1A 01 00 |....ě...
17030: 00 00 00 00 00 00 00 00 |........      17030: 01 00 00 00 B1 B4 00 00 |....±´..
```

Comparing the memory dump with the raw file

We also found that some of the addresses were relocated (the new Image Base was added).

Reversing the reversed PE

The files with both extensions CAB and BLOB are loaded by the same function:

Start	End	Name	Type	Args	Is refered by
10002b60	10002edc	to_read_and_load_custom	int __cdecl	(LPCWSTR lpFileName, int blob_object)	2

int __cdecl **to_read_and_load_custom** (LPCWSTR lpFileName, int blob_object)

Is refered by 2: | Refers to 45:

Foreign Val.	From Address
load_cab	100017eb
load_blob	1000160c

View from IFL (Interactive Functions List)

The core of the loader is in the following function:

This is the function that we need to analyze in order to make sense out of the custom format.

```
10002E85 loc_10002E85:
10002E85 push    ebp
10002E86 push    ebx
10002E87 mov     ecx, edi
10002E89 mov     esi, edx
10002E8B call    load_custom_format
10002E90 mov     dword ptr [esi], 2E2E2E2Eh
10002E96 mov     edi, eax
10002E98 mov     eax, [esp+74h+var_34]
10002E9C xor     esi, esi
```

Let’s take a look at the loading process itself.

First, DWORD of the Header1 is skipped. Then, we have two DWORDs that are used as an XOR key. Once they are fetched, the rest of the header is decoded.

```
736026C0   LEA EDX,DWORD PTR DS:[ESI-0xC]
736026C3   MOV EDI,EDX
736026C5   SHR EDI,0x2
736026C8   XOR EAX,EAX
736026CA   MOV DWORD PTR SS:[ESP+0x2C],EDX
736026CE   TEST EDI,EDI
736026D0   JBE SHORT SPORDE_1.736026E4
736026D2   MOV ECX,EAX
736026D4   AND ECX,0x1
736026D7   MOV ECX,DWORD PTR DS:[EBX+ECX*4+0x4]
736026DB   XOR DWORD PTR DS:[EBX+EAX*4+0xC],ECX
736026DF   INC EAX
736026E0   CMP EAX,EDI
736026E2   JB SHORT SPORDE_1.736026D2
736026E4   XOR ESI,ESI
736026E6   XOR ECX,ECX
736026E8   TEST EDI,EDI
736026EA   JBE SHORT SPORDE_1.7360274A
736026EC   LEA ESP,DWORD PTR SS:[ESP]

Jump is NOT taken
736026D2=SPORDE_1.736026D2

Address   Hex dump                                          ASCII
00460054  00 70 01 00 00 00 00 00 00 00 00 00 9F E8 01 00  .p0.........čR0.
00460064  36 49 7B F1 9F E8 01 00 36 49 7B F1 9F E8 01 00  6I{˝čR0.6I{˝čR0.
00460074  36 49 7B F1 9F E8 01 00 36 49 7B F1 9F E8 01 00  6I{˝čR0.6I{˝čR0.
00460084  36 49 7B F1 9F E8 01 00 36 49 7B F1 9F E8 01 00  6I{˝čR0.6I{˝čR0.
```

After applying the key, we get the content of the file in its clear form. The next value from the headers is used in the formula calculating the size for loading the executable part of the module. In the currently analyzed case (the CAB file), it is 0x17000:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  4A D9 D9 08 00 00 00 00 00 00 00 00 00 70 01 00  JÙÙ.........p..
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Header 1 at the beginning of the CAB file, decoded

So, 0x17000 + 0x2000 is the size of the memory that will be allocated for the payload.

Example (from CAB file):

```
1000276C mov     edi, [ebx+0Ch]    ; saved_size
1000276F push    40h               ; flProtect
10002771 lea     eax, [edi+2000h]
10002777 push    1000h             ; flAllocationType
1000277C push    eax               ; dwSize
1000277D lea     esi, [ebx+10h]
10002780 push    0                 ; lpAddress
10002782 mov     [esp+8Ch+module_size], edi
10002786 lea     ebx, [edx-4]
10002789 mov     [esp+8Ch+dwSize], eax
1000278D call    ds:VirtualAlloc
10002793 mov     ebp, eax
```

```
73602762     POP EBX
73602763     ADD ESP,0x68
73602766     RETN
73602767     CMP EDX,0x4
7360276A  v  JB SHORT SPORDE_1.736027B6
7360276C     MOV EDI,DWORD PTR DS:[EBX+0xC]
7360276F     PUSH 0x40
73602771     LEA EAX,DWORD PTR DS:[EDI+0x2000]             saved_value+0x2000
73602777     PUSH 0x1000
7360277C     PUSH EAX
7360277D     LEA ESI,DWORD PTR DS:[EBX+0x10]
73602780     PUSH 0x0
73602782     MOV DWORD PTR SS:[ESP+0x40],EDI
73602786     LEA EBX,DWORD PTR DS:[EDX-0x4]
73602789     MOV DWORD PTR SS:[ESP+0x38],EAX
7360278D     CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>]  kernel32.VirtualAlloc
73602793     MOV EBP,EAX

DS:[73610070]=76B72FB6 (kernel32.VirtualAlloc)

0006F398   00000000  Address = NULL
0006F39C   00019000  Size = 19000 (102400.)
0006F3A0   00001000  AllocationType = MEM_COMMIT
0006F3A4   00000040  Protect = PAGE_EXECUTE_READWRITE
0006F3A8   2FC65A62
```

Then, 0x17000 bytes of the payload is copied, but the beginning containing the Header1 is skipped (the first 16 bytes).

After the module content is copied, Header2 is used to continue loading.

Looking at Header2, we can see some similarities with Header1. Again, the initial DWORD is skipped, and then we have a value that is used in a formula calculating the size of the memory to be allocated. The new memory region that is being allocated this time is used for the imports that are going to be loaded (the full process will be explained further).

Conceptually, we can divide Header 2 into two parts.

First comes a prolog that contains two DWORD values. Example from the currently-analyzed CAB file:

```
00016FF0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00017000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00017010  A0 21 00 00 3D 01 00 00 01 00 00 00 EC 1D 01 00   !..=.......ě...
00017020  01 00 00 00 FC 58 00 00 01 00 00 00 90 85 00 00  ....üX..........
00017030  01 00 00 00 EC AA 00 00 01 00 00 00 EC 1A 01 00  ....ěŞ......ě...
00017040  01 00 00 00 B1 B4 00 00 03 00 00 00 02 00 00 00  ....±´..........
00017050  DA 07 01 00 74 07 01 00 14 E0 00 00 01 00 00 00  Ú...t....ŕ......
```

Header2 (at the end of the CAB file) – prolog is hilighted

- val[0] = 0x21A0 -> skipped
- val[1] = 0x013D -> val[1]*8+0x400 -> size of the next area to allocate

Then there is a list of records of a custom type. Each record represents a different piece of information that is necessary for loading the module. They are identified by the type ID that is represented by a DWORD at the beginning of the record.

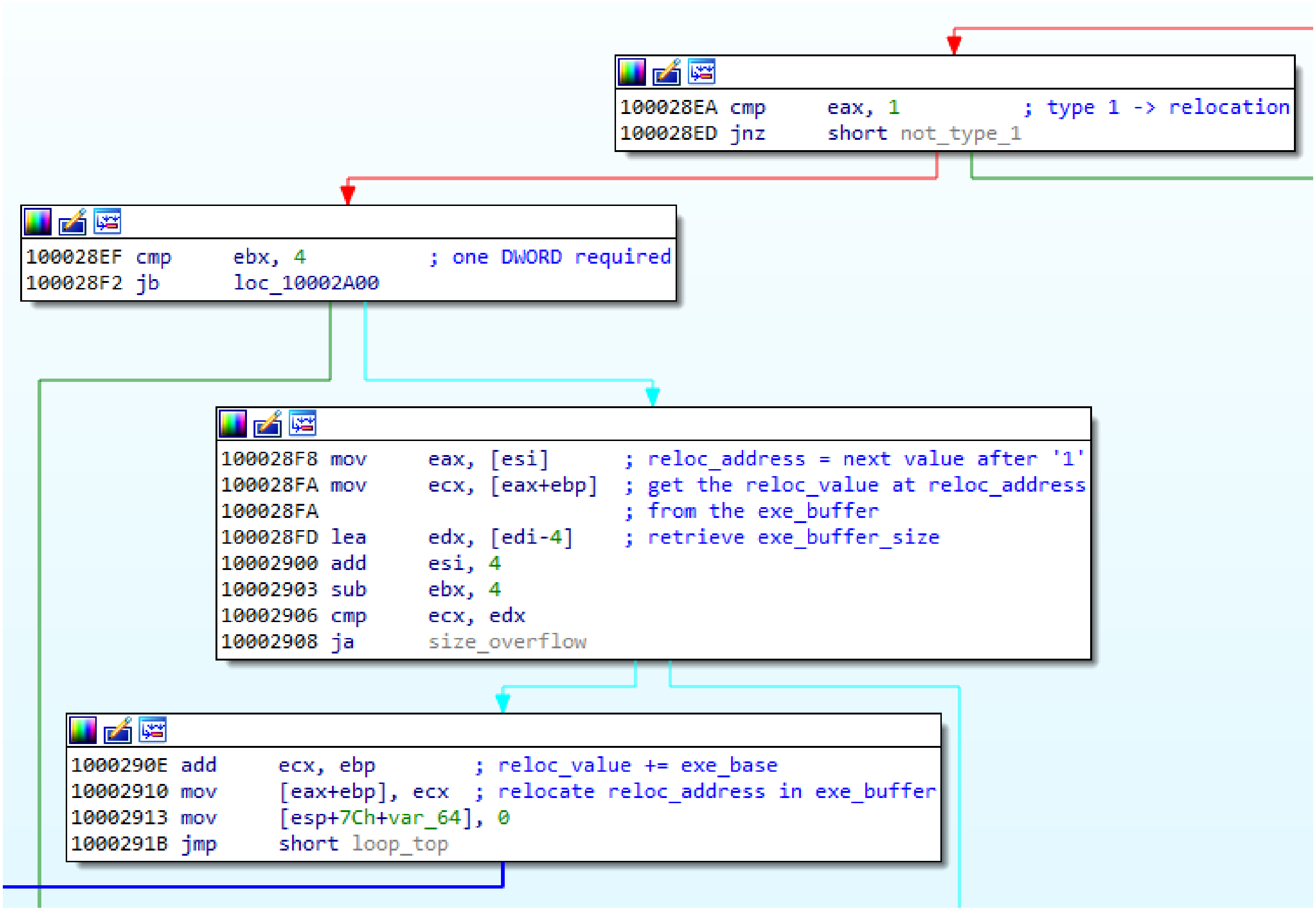
```
00017000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00017010  A0 21 00 00 3D 01 00 00 01 00 00 00 EC 1D 01 00   !..=.......ě...
00017020  01 00 00 00 FC 58 00 00 01 00 00 00 90 85 00 00  ....üX..........
00017030  01 00 00 00 EC AA 00 00 01 00 00 00 EC 1A 01 00  ....ěŞ......ě...
00017040  01 00 00 00 B1 B4 00 00 03 00 00 00 02 00 00 00  ....±´..........
00017050  DA 07 01 00 74 07 01 00 14 E0 00 00 01 00 00 00  Ú...t....ŕ......
00017060  24 11 00 00 01 00 00 00 C2 1B 00 00 02 00 00 00  $.......Â.......
00017070  01 00 00 00 30 10 00 00 00 00 00 00 01 00 00 00  ....0...........
```

Header2 (at the end of the CAB file) – records are hilighted

Relocations

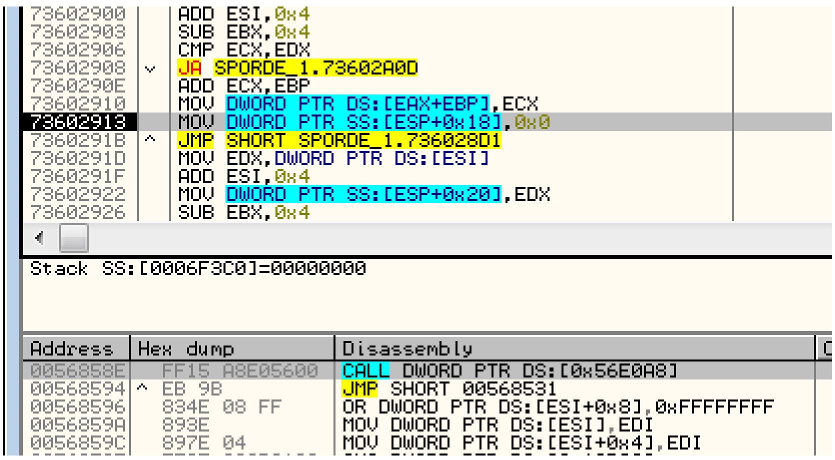
Type 1 stands for relocation. It has one DWORD as an argument. It is an address that needs to be relocated.

```
typedef struct {
        DWORD reloc_field;
} reloc_t;
```



Parsing of the type 1

We can see how the field is used to relocate the address. Example: filling the address at 0x8590:



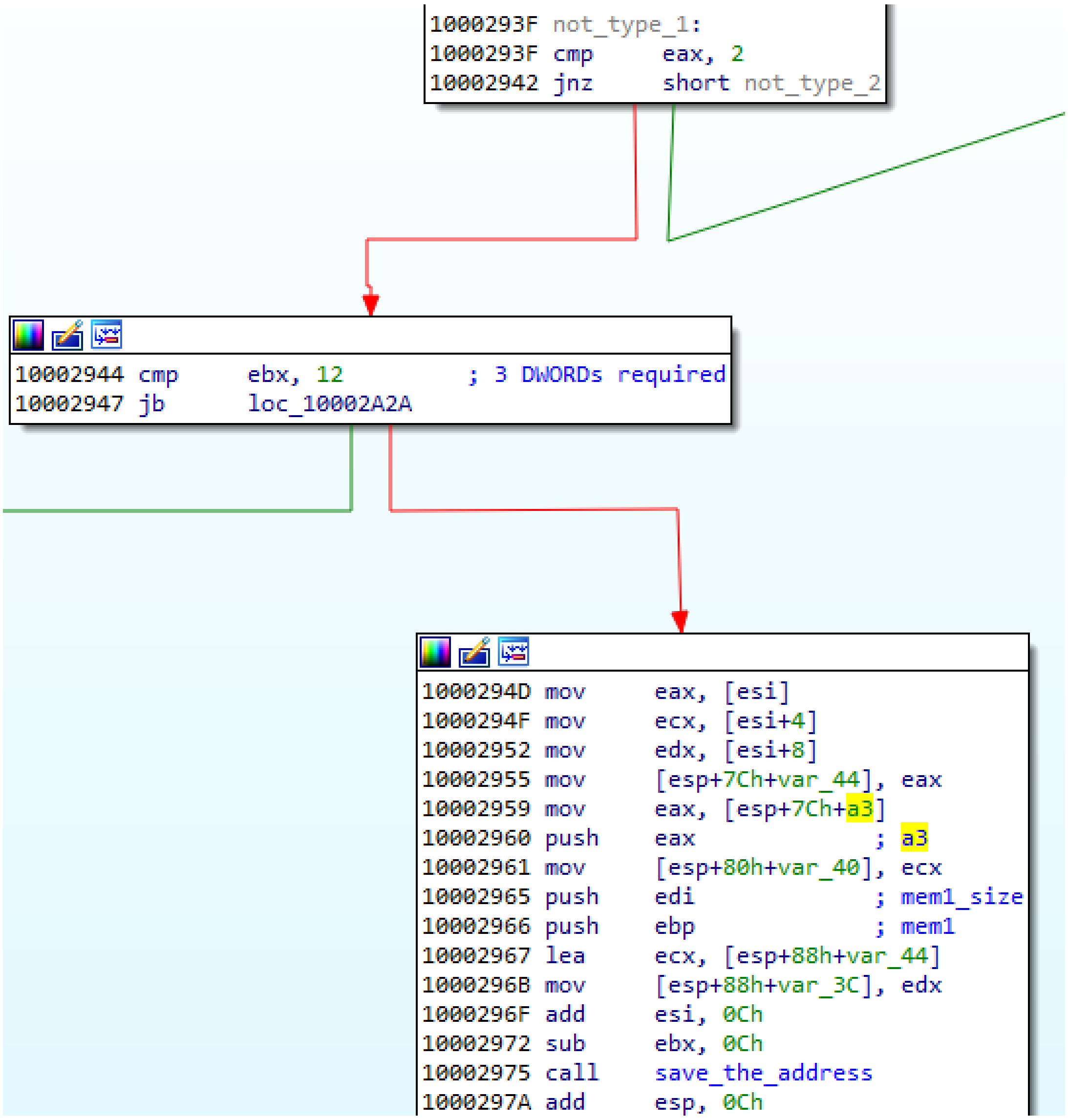
The address pointed by the relocation record is relocated to the base at which the module was loaded

Entry point

Type 2 stands for entry point or an exported function. The pointed address is stored on the list in order to be called later, after the loading finished. This record has three DWORD parameters.

```
typedef struct {
    DWORD count;
    DWORD entry_rva;
    DWORD name_rva;
} entry_point_t;
```

Example of the record of type 2:



Parsing of the type 2

Address to be stored: params[1] = 0x00001030

```
00017030  01 00 00 00 B1 B4 00 00 03 00 00 00 02 00 00 00  ....±´..........
00017040  DA 07 01 00 74 07 01 00 14 E0 00 00 01 00 00 00  Ú...t....ŕ......
00017050  24 11 00 00 01 00 00 00 C2 1B 00 00 02 00 00 00  $.......Â.......
00017060  01 00 00 00 30 10 00 00 00 00 00 00 01 00 00 00  ....0...........
00017070  23 80 00 00 01 00 00 00 E3 AC 00 00 01 00 00 00  #€......ă¬......
00017080  BC 01 01 00 01 00 00 00 8D B4 00 00 01 00 00 00  Ľ.......Ť´......
00017090  68 2C 00 00 01 00 00 00 C3 A0 00 00 01 00 00 00  h,......Ă ......
```

Record of the type 2 in the original file

By observing the execution flow, we can confirm that indeed the stored entry point of the module is being called later:

```
100018A3 call    ds:lstrcpyW
100018A9 lea     edx, [esp+45Ch+String1]
100018B0 push    edx              ; main module path
100018B1 call    ebx              ; call entry point of .CAB module, RVA = 0x1030
100018B3 lea     ebx, [esp+45Ch+Buffer]
100018B7 call    delete_file_till_success
```

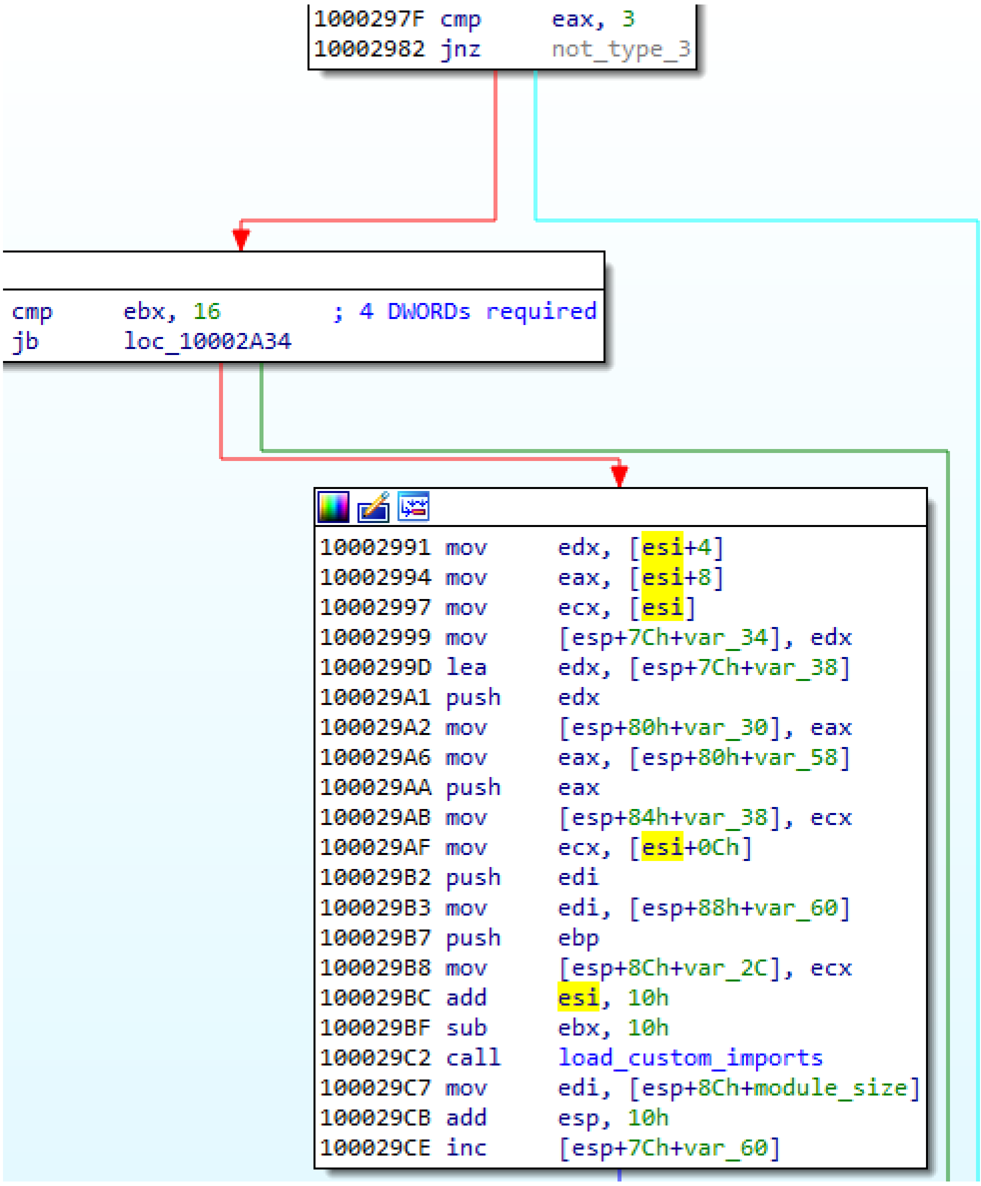
The address in the loader where the CAB module is called after being loaded

Exported functions are stored in the same way, along with their names.

Imports

Type 3 stands for imports. It has four DWORD parameters.

```
typedef struct {
    DWORD type;
    DWORD dll_rva;
    DWORD func_rva;
    DWORD iat_rva;
} import_t;
```



Parsing of the type 3

Example of a chunk responsible for encoding imports:

```
00017010  01 00 00 00 FC 58 00 00 01 00 00 00 90 85 00 00  ....üX..........
00017020  01 00 00 00 EC AA 00 00 01 00 00 00 EC 1A 01 00  ....ěŞ......ě...
00017030  01 00 00 00 B1 B4 00 00 03 00 00 00 02 00 00 00  ....±´..........
00017040  DA 07 01 00 74 07 01 00 14 E0 00 00 01 00 00 00  Ú...t....ŕ......
00017050  24 11 00 00 01 00 00 00 C2 1B 00 00 02 00 00 00  $.......Â.......
```

Record of the type 3 in the original file

Type: params[0] = 0x00000002 – means the function will be imported by name, meaning of all the possible types of this record.

Address of the DLL: params[1] = 0x0107DA

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

000107D0  63 65 48 61 6E 64 6C 65 00 00 41 44 56 41 50 49  ceHandle..ADVAPI
000107E0  33 32 2E 64 6C 6C 00 00 81 02 47 65 74 57 69 6E  32.dll....GetWin
000107F0  64 6F 77 73 44 69 72 65 63 74 6F 72 79 57 00 00  dowsDirectoryW..
```

Address of the import: params[2] = 0x010774

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00010770  00 00 7D 00 43 72 65 61 74 65 53 65 72 76 69 63   ..}.CreateServic
00010780  65 57 00 00 49 00 43 68 61 6E 67 65 53 65 72 76   eW..I.ChangeServ
00010790  69 63 65 43 6F 6E 66 69 67 32 57 00 4B 00 43 68   iceConfig2W.K.Ch
000107A0  61 6E 67 65 53 65 72 76 69 63 65 43 6F 6E 66 69   angeServiceConfi
```

In contrast to PE format, the address of the imported function is not loaded into the main module. Instead, it is written into the separate executable area (in the given example it is written at VA: 0x00240001):

```
Address   Hex dump      Disassembly
00240000    90          NOP
00240001    B8 2C717976 MOV EAX,advapi32.CreateServiceW
00240006  - FFE0        JMP EAX
00240008    0000        ADD BYTE PTR DS:[EAX],AL
0024000A    0000        ADD BYTE PTR DS:[EAX],AL
```

And then, the address where the import was filled is filled back in the main module. The address in the main module that needs to be filled is specified by the last parameter of this record. In the given example, chunk[3] = 0x0000E014 is being filled by 0x00240001:

```
0E000: 00 00 00 00 00 00 00 00 |........    0E000: 00 00 00 00 00 00 00 00 |........
0E008: 00 00 00 00 00 00 00 00 |........    0E008: 00 00 00 00 00 00 00 00 |........
0E010: 00 00 00 00 00 00 00 00 |........    0E010: 00 00 00 00 01 00 24 00 |......$.
0E018: 00 00 00 00 00 00 00 00 |........    0E018: 00 00 00 00 00 00 00 00 |........
```

Atypical IAT

The functions from the embedded list are for a loader, however, as mentioned earlier, the addresses are not filled in a normal IAT, typical for PE format. Rather, all are filled as a list of jumps stored in a newly-allocated memory page.

```
736024A2    PUSH EBX
736024A3    CALL DWORD PTR DS:[<&KERNEL32.GetProcAddress>]   kernel32.GetProcAddress
736024A9    TEST EAX,EAX                                     kernel32.GetEnvironmentStrings
736024AB  v JE SHORT SPORDE_1.73602516
736024AD    MOV EDX,DWORD PTR SS:[ESP+0xC]
736024B1    MOV ECX,DWORD PTR DS:[EDX+0x4]
736024B4    TEST ECX,ECX
736024B6  v JE SHORT SPORDE_1.73602522
736024B8    MOV EDX,DWORD PTR SS:[ESP+0x1C]
736024BC    ADD EDX,-0x4
736024BF    CMP ECX,EDX
736024C1  v JA SHORT SPORDE_1.73602522
736024C3    MOV EDX,DWORD PTR SS:[ESP+0x18]
736024C7    MOV EBX,DWORD PTR SS:[ESP+0xC]
736024CB    MOV DWORD PTR SS:[EBP+EDI*8+0x2],EAX             kernel32.GetEnvironmentStrings
736024CF    LEA EAX,DWORD PTR SS:[EBP+EDI*8+0x1]
736024D3    MOV BYTE PTR SS:[EBP+EDI*8],0x90
736024D8    MOV BYTE PTR DS:[EAX],0xB8
736024DB    MOV BYTE PTR SS:[EBP+EDI*8+0x6],0xFF
736024E0    MOV BYTE PTR SS:[EBP+EDI*8+0x7],0xE0
736024E5    MOV DWORD PTR DS:[ECX+EDX],EAX                   kernel32.GetEnvironmentStrings

Address=00240041
EAX=76B72F99 (kernel32.GetEnvironmentStrings)

Address   Hex dump     Disassembly                              Comment
00240000    90         NOP
00240001    B8 2C717976 MOV EAX,advapi32.CreateServiceW
00240006  - FFE0       JMP EAX                                  kernel32.GetEnvironmentStrings
00240008    90         NOP
00240009    B8 64CA7776 MOV EAX,advapi32.OpenSCManagerW
0024000E  - FFE0       JMP EAX                                  kernel32.GetEnvironmentStrings
00240010    90         NOP
00240011    B8 D62CB776 MOV EAX,kernel32.WriteConsoleA
00240016  - FFE0       JMP EAX                                  kernel32.GetEnvironmentStrings
00240018    90         NOP
00240019    B8 D8307B76 MOV EAX,advapi32.ChangeServiceConfig2W
0024001E  - FFE0       JMP EAX                                  kernel32.GetEnvironmentStrings
00240020    90         NOP
00240021    B8 A135B776 MOV EAX,kernel32.TlsAlloc
00240026  - FFE0       JMP EAX                                  kernel32.GetEnvironmentStrings
00240028    90         NOP
00240029    B8 8012B776 MOV EAX,kernel32.GetProcessHeap
0024002E  - FFE0       JMP EAX                                  kernel32.GetEnvironmentStrings
00240030    90         NOP
00240031    B8 511BAE74 MOV EAX,version.VerQueryValueW
00240036  - FFE0       JMP EAX                                  kernel32.GetEnvironmentStrings
00240038    90         NOP
00240039    B8 BF455C76 MOV EAX,shlwapi.PathFileExistsW
0024003E  - FFE0       JMP EAX                                  kernel32.GetEnvironmentStrings
00240040    0000       ADD BYTE PTR DS:[EAX],AL
00240042    0000       ADD BYTE PTR DS:[EAX],AL
```

The import loading function not only fills the address, but also emits the necessary code for the jump:

```
  imported_func = GetProcAddress(v11, v12);
  if ( !imported_func )
    return 1000405;
  v14 = *(_DWORD *)(v18 + 4);
  if ( !v14 || v14 > a3 - 4 )
  {
    lstrcpyA(
      byte_10017E40,
      "bua6i EzhEOF meus u0Upa ObIEPO 1aE5 GoEK Ka4 ipUuri yhub MhaF VhoW BeH EwIT 8it awIv otIg Nh");
    return 1000406;
  }
  v6 = (_DWORD *)v18;
  *(_DWORD *)(a4 + 8 * a1 + 2) = imported_func;
  v15 = (_BYTE *)(a4 + 8 * a1 + 1);
  *(_BYTE *)(a4 + 8 * a1) = 0x90u;
  *v15 = 0xB8u;
  *(_BYTE *)(a4 + 8 * a1 + 6) = 0xFFu;
  *(_BYTE *)(a4 + 8 * a1 + 7) = 0xE0u;
  *(_DWORD *)(v14 + a2) = v15;
}
```

Address of the imported function is retrieved and written into the emitted jump

Meaning of the type field

The import record has a field type, that can have one of the following values: 1,2,3,4.

The 1 and 2 are the most important: They are used for loading the imports. 1 stands for loading by ordinals, 2 for loading by name. The remaining 3 and 4 are used for cleanup of the fields that are no longer needed. 3 erases import name, 4 erases DLL name.

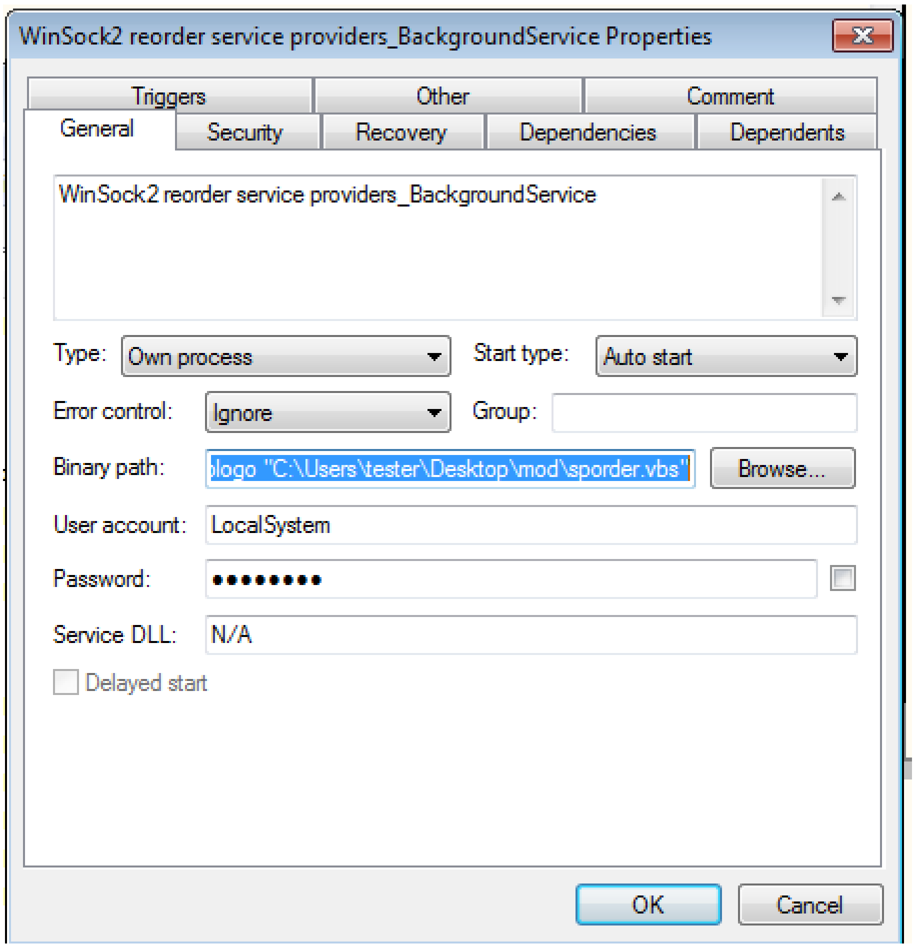
```
    else if ( func_type == 4 )              // erase library name
    {
      if ( lib_name && *lib_name )
      {
        do
          *lib_name++ = 0;
        while ( *lib_name );
      }
      lstrcpyW(&String1, L"IghOWO ZhoUV akhIab bhi8 Th");
    }
    else
    {
      lib = LoadLibraryA(lib_name);
      if ( !lib )
        return 1000403;                     // skip
      if ( func_type == 1 )
      {
        func_name = *(const CHAR **)func_field;// by ordinal
        lstrcpyW(
          &String1,
          L"ecEob nho6i OlIWO alAce 0az bol pi9 RoHO 0huawo wiy 6euw PaP cic WeG EpUOS EbhUK e0Iar j");
      }
      else
      {
        if ( func_type != 2 )
          return 1000404;
        func_name = (const CHAR *)(buffer + *(_DWORD *)func_field);// by name
      }
      imported_func = GetProcAddress(lib, func_name);
      if ( !imported_func )
        return 1000405;
      v14 = *(_DWORD *)(func_field + 4);
      if ( !v14 || v14 > buffer_size - 4 )
      {
        lstrcpyA(
          byte_10017E40,
          "bua6i EzhEOF meus u0Upa ObIEPO 1aE5 GoEK Ka4 ipUuri yhub MhaF VhoW BeH EwIT 8it awIv otIg Nh");
        return 1000406;
      }
```

When the record of the type 3 or 4 occurs, the pointer in the IAT area is still incremented, so as a result we can see some gaps between the functions records:

```
00240199  B8 46BAB676  MOV EAX,kernel32.Sleep
0024019E  FFE0         JMP EAX
002401A0  90           NOP
002401A1  B8 C31DB776  MOV EAX,kernel32.FreeEnvironmentStringsW
002401A6  FFE0         JMP EAX
002401A8  90           NOP
002401A9  B8 70DAB676  MOV EAX,kernel32.TlsGetValue
002401AE  FFE0         JMP EAX
002401B0  90           NOP
002401B1  B8 9F68B576  MOV EAX,kernel32.GetStringTypeExA
002401B6  FFE0         JMP EAX
002401B8  90           NOP
002401B9  B8 88DAB676  MOV EAX,kernel32.TlsSetValue
002401BE  FFE0         JMP EAX
002401C0  90           NOP
002401C1  B8 B813B776  MOV EAX,kernel32.TlsFree
002401C6  FFE0         JMP EAX
002401C8  90           NOP
002401C9  B8 D62D3F77  MOV EAX,ntdll.RtlAllocateHeap
002401CE  FFE0         JMP EAX
002401D0  90           NOP
002401D1  B8 36DBB676  MOV EAX,kernel32.SetFilePointer
002401D6  FFE0         JMP EAX
002401D8  90           NOP
002401D9  B8 9C367876  MOV EAX,advapi32.CloseServiceHandle
002401DE  FFE0         JMP EAX
002401E0  90           NOP
002401E1  B8 A83EB676  MOV EAX,kernel32.IsDebuggerPresent
002401E6  FFE0         JMP EAX
002401E8  90           NOP
002401E9  B8 BC0BB676  MOV EAX,kernel32.GetShortPathNameW
002401EE  FFE0         JMP EAX
002401F0  0000         ADD BYTE PTR DS:[EAX],AL
002401F2  0000         ADD BYTE PTR DS:[EAX],AL
002401F4  0000         ADD BYTE PTR DS:[EAX],AL
002401F6  0000         ADD BYTE PTR DS:[EAX],AL
002401F8  90           NOP
002401F9  B8 531C5C76  MOV EAX,shlwapi.PathRemoveExtensionW
002401FE  FFE0         JMP EAX
00240200  90           NOP
00240201  B8 2B45B776  MOV EAX,kernel32.MultiByteToWideChar
00240206  FFE0         JMP EAX
00240208  90           NOP
00240209  B8 F633B776  MOV EAX,kernel32.GetModuleFileNameA
0024020E  FFE0         JMP EAX
00240210  0000         ADD BYTE PTR DS:[EAX],AL
00240212  0000         ADD BYTE PTR DS:[EAX],AL
00240214  0000         ADD BYTE PTR DS:[EAX],AL
00240216  0000         ADD BYTE PTR DS:[EAX],AL
00240218  90           NOP
00240219  B8 74797776  MOV EAX,advapi32.StartServiceW
0024021E  FFE0         JMP EAX
00240220  0000         ADD BYTE PTR DS:[EAX],AL
00240222  0000         ADD BYTE PTR DS:[EAX],AL
00240224  0000         ADD BYTE PTR DS:[EAX],AL
00240226  0000         ADD BYTE PTR DS:[EAX],AL
00240228  90           NOP
00240229  B8 00BFB676  MOV EAX,kernel32.GetLastError
0024022E  FFE0         JMP EAX
00240230  90           NOP
00240231  B8 D919AE74  MOV EAX,version.GetFileVersionInfoSizeW
00240236  FFE0         JMP EAX
00240238  90           NOP
00240239  B8 60EBB576  MOV EAX,kernel32.RaiseException
0024023E  FFE0         JMP EAX
00240240  0000         ADD BYTE PTR DS:[EAX],AL
00240242  0000         ADD BYTE PTR DS:[EAX],AL
00240244  0000         ADD BYTE PTR DS:[EAX],AL
00240246  0000         ADD BYTE PTR DS:[EAX],AL
00240248  90           NOP
00240249  B8 6913B776  MOV EAX,kernel32.GetConsoleOutputCP
0024024E  FFE0         JMP EAX
00240250  90           NOP
00240251  B8 9FBBB676  MOV EAX,kernel32.QueryPerformanceCounter
```

Functionality of the custom files

The CAB file is another installer that provides persistence to the whole package by creating a service:



"C:\Windows\system32\wscript.exe" /B /nologo "C:\Users\tester\Desktop\mod\sporder.vbs"

I also generate the VBS script that is dropped:

sporder.vbs

```
on error resume next
sub F1()
p = "C:\Users\tester\Desktop\mod\"
l = array("SPORDER.dll")
set o = CreateObject("Scripting.FileSystemObject")
for i = 0 to 0
if not o.FileExists(p + l(i)) then exit sub
next
set o = CreateObject ("WScript.Shell")
o.CurrentDirectory = p
o.run "sporder.exe", 0, 0
end sub
F1
```

The CAB file is loaded first, just to install the malware, and then deleted.

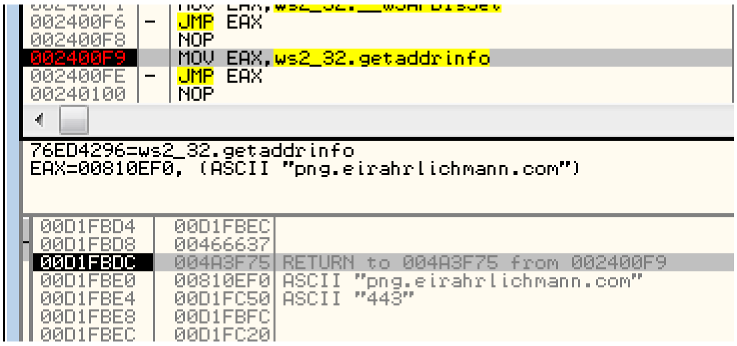
All the espionage-related features are performed by the BLOB that is loaded later and kept persistent in the memory of the loader.

In addition to being in a custom format, BLOB is also heavily obfuscated.

We can observe its attempts to connect to one of the CnCs:

```
png.eirahrlichmann.com : 443
engine.lanaurmi.com :3389
movies.onaldest.com : 44818
images.andychroeder.com : 80
png.eirahrlichmann.com : 44818
engine.lanaurmi.com : 44818
movies.onaldest.com : 9091
images.andychroeder.com : 9091
png.eirahrlichmann.com : 3389
```

Some of those domains are known from previous reports on Ocean Lotus, i.e. [the Cyclance white paper].



Ocean Lotus: a creative APT

Ocean Lotus often surprises researchers with its creative obfuscation techniques. Recently, a different sample of Ocean Lotus was found using steganography to hide their executables (you can read more about it in the report of ThreatVector). The format that we described is just one of many unusual forms that their implants can take.

Appendix

Parser for the described format:
https://github.com/hasherezade/funky_malware_formats/tree/master/lotus_parser
Presentation from the SAS conference: