Reversing malware in a custom format: Hidden Bee elements

blog.malwarebytes.com/threat-analysis/2018/08/reversing-malware-in-a-custom-format-hidden-bee-elements/

hasherezade

August 30, 2018

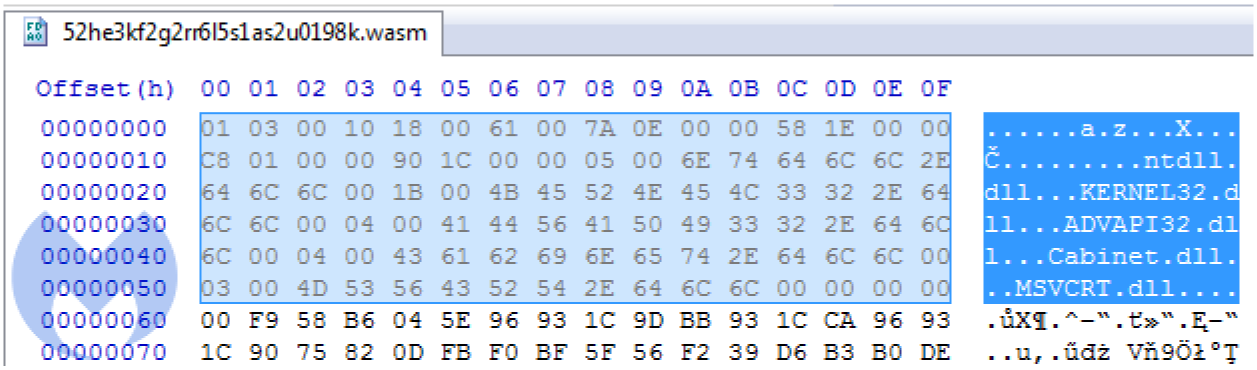
Malware can be made of many components. Often, we encounter macros and scripts that work as malicious downloaders. Some functionalities can also be achieved by position-independent code—so-called shellcode. But when it comes to more complex elements or core modules, we almost take it for granted that it will be a PE file that is a native Windows executable format.

The reason for this is simple: It is much easier to provide complex functionality within a PE file than within a shellcode. PE format has a well-defined structure, allowing for much more flexibility. We have certain headers that define what imports should be loaded and where, as well as how the relocations should be applied. This is a default format generated when we compile applications for Windows, and its structure is then used by Windows Loader to load and execute our application. Even when the malware authors write custom loaders, they are mostly for the PE format.

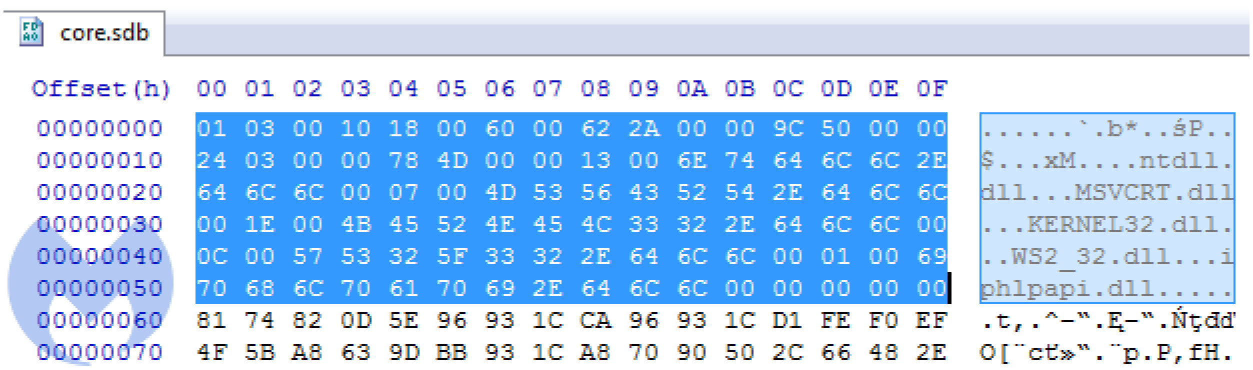
However, sometimes we find exceptions. Last time, when we analyzed payloads related to Hidden Bee (dropped by the Underminer exploit kit), we noticed something unusual. There were two payloads dropped that didn't follow the PE format. Yet, their structure looked well organized and more complex than we usually encounter dealing with pieces of shellcode. We decided to take a closer look and discovered that the authors of this malware actually created their own executable format, following a consistent structure.

Overview

The first payload: b3eb576e02849218867caefaa0412ccd (with .wasm extension, imitating Web Assembly) is a loader, downloading and unpacking a Cabinet file:



The second payload: 11310b509f8bf86daa5577758e9d1eb5, unpacked from the Cabinet:



We can see at first that in contrast to most shellcodes, it does not start from a code, but from some headers. Comparing both modules, we can see that the header has the same structure in both cases.

Headers

We took a closer look to decipher the meaning of particular fields in the header.

```
core.sdb

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

00000000  01 03 00 10 18 00 60 00 62 2A 00 00 9C 50 00 00  ......`.b*..śP..
00000010  24 03 00 00 78 4D 00 00 13 00 6E 74 64 6C 6C 2E  $...xM....ntdll.
00000020  64 6C 6C 00 07 00 4D 53 56 43 52 54 2E 64 6C 6C  dll...MSVCRT.dll
00000030  00 1E 00 4B 45 52 4E 45 4C 33 32 2E 64 6C 6C 00  ...KERNEL32.dll.
00000040  0C 00 57 53 32 5F 33 32 2E 64 6C 6C 00 01 00 69  ..WS2_32.dll...i
00000050  70 68 6C 70 61 70 69 2E 64 6C 6C 00 00 00 00 00  phlpapi.dll.....
00000060  81 74 82 0D 5E 96 93 1C CA 96 93 1C D1 FE F0 EF  .t,.^–".Ę–".Ńţđď
00000070  4F 5B A8 63 9D BB 93 1C A8 70 90 50 2C 66 48 2E  O[¨cť»".¨p.P,fH.
00000080  F8 5C EF 6E 72 3C 94 7C 0B 0F B5 A5 D6 94 93 1C  ř\ďnr<"|..µĄÖ"".
00000090  77 E2 E1 F9 89 5F B7 29 8D AF D2 7D F5 26 BD 6B  wâáů‰_·)ŤŻŇ}ő&˝k
```

The first DWORD: 0x10000301 is the same in both. We didn't find this number corresponding to any of the pieces within the module. So, we assume it is a magic number that makes an identifier of this format.

Next, two WORDs are offsets to elements related to loading the imports. The first one (0x18) points to the list of DLLs. The second block (0x60) looks more mysterious at first. Its meaning can be understood when we load the module in IDA. We can see the cross-references to those fields:

```
00000004                dd 0
00000060 dword_60       dd 0D827481h            ; DATA XREF: sub_3F7F+A9↓r
```

xrefs to dword_60

Direction	Typ	Address	Text
Do...	r	sub_3F7F+A9	call ds:dword_60
Do...	r	sub_3F7F+BD	call ds:dword_60
Do...	r	sub_3F7F+D1	call ds:dword_60
Do...	r	sub_3F7F+17F	call ds:dword_60
Do...	r	sub_3F7F+1B8	call ds:dword_60
Do...	r	sub_3F7F+314	call ds:dword_60

We see that they are used as IAT—they are supposed to be filled with the addresses to the imported functions:

```
00004038                push    ebx
00004039                push    3Fh ; '?'
0000403B                push    edi
0000403C                call    ds:dword_60
00004042                add     esp, 0Ch
00004045                mov     [ebp+var_10], eax
00004048                test    eax, eax
0000404A                jnz     short loc_405C
0000404C                push    ebx
0000404D                push    23h ; '#'
0000404F                push    edi
00004050                call    ds:dword_60
```

The next value is a DWORD (0x2A62). If we follow it in IDA, we see that it leads to the beginning of a new function:

```
:00002A62 ; ---------------------------------------------------------------------------
:00002A62                 push    ebp
:00002A63                 mov     ebp, esp
:00002A65                 sub     esp, 22Ch
:00002A6B                 call    sub_2986
:00002A70                 test    al, al
:00002A72                 jnz     short loc_2A7D
:00002A74                 call    sub_29A3
:00002A79                 test    eax, eax
:00002A7B                 jz      short loc_2A84
:00002A7D
:00002A7D loc_2A7D:                               ; CODE XREF: seg000:00002A72↑j
:00002A7D                 xor     eax, eax
:00002A7F                 jmp     locret_2BBC
```

This function is not referenced by any other functions so we can suspect that it is the program’s Entry Point.

The meaning of the next value (0x509C) is easy to guess because it is the same as the size of the full module.

Then, we have the last two DWORDs of the header. The second DWORD (0x4D78) leads to the structure that is very similar to the PE’s relocations. We can guess that it must be a relocation table of the module, and the previous DWORD specifies its size.

```
00004D70  00 00 00 00 00 00 00 00 84 02 00 00 C2 02 00 00  ........„...Â...
00004D80  09 03 00 00 B0 28 00 00 59 29 00 00 67 29 00 00  ....°(..Y)..g)..
00004D90  92 29 00 00 AD 29 00 00 C8 29 00 00 F7 29 00 00  ’)...)..Č)..÷)..
00004DA0  FD 29 00 00 14 2A 00 00 2A 2A 00 00 56 2A 00 00  ý)...*..**..V*..
00004DB0  B3 2A 00 00 B8 2A 00 00 CF 2A 00 00 D5 2A 00 00  ł*..¸*..Ď*..Ő*..
00004DC0  DB 2A 00 00 EC 2A 00 00 FC 2A 00 00 0E 2B 00 00  Ű*..ě*..ü*...+..
00004DD0  3D 2B 00 00 4D 2B 00 00 5D 2B 00 00 75 2B 00 00  =+..M+..]+..u+..
00004DE0  7F 2B 00 00 84 2B 00 00 9D 2B 00 00 A9 2B 00 00  .+..„+..ť+..©+..
00004DF0  B2 2B 00 00 EF 2B 00 00 6B 2C 00 00 70 2C 00 00  ˛+..ď+..k,..p,..
```

This is how we were able to reconstruct the full header:

```
typedef struct {
        DWORD magic;

        WORD dll_list;
        WORD iat;
        DWORD ep;
        DWORD mod_size;

        DWORD relocs_size;
        DWORD relocs;
} t_bee_hdr;
```

Imports

As we know from the header, the list of the DLLs starts at the offset 0x18. We can see that each of the DLL’s names are prepended with a number:

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

00000000  01 03 00 10 18 00 60 00 62 2A 00 00 9C 50 00 00  ......`.b*..śP..
00000010  24 03 00 00 78 4D 00 00 13 00 6E 74 64 6C 6C 2E  $...xM....ntdll.
00000020  64 6C 6C 00 07 00 4D 53 56 43 52 54 2E 64 6C 6C  dll...MSVCRT.dll
00000030  00 1E 00 4B 45 52 4E 45 4C 33 32 2E 64 6C 6C 00  ...KERNEL32.dll.
00000040  0C 00 57 53 32 5F 33 32 2E 64 6C 6C 00 01 00 69  ..WS2_32.dll...i
00000050  70 68 6C 70 61 70 69 2E 64 6C 6C 00 00 00 00 00  phlpapi.dll.....
00000060  81 74 82 0D 5E 96 93 1C CA 96 93 1C D1 FE F0 EF  .t‚.^–“.Ę–“.Ńţđď
```

The numbers are not corresponding with a DLL name: In two different modules, the same DLL had different numbers assigned. But if we sum up all the numbers, we find that their total sum is the same as the number of DWORDs in the IAT. So, we can make an educated guess that those numbers are specifying how many functions will be imported from a particular DLL.

We can describe it as the following structure (where the name's length is not specified):

```
typedef struct {
        WORD func_count;
        char name;
} t_dll_name;
```

Then, the IAT comes as a list of DWORDs:

```
00000050  70 68 6C 70 61 70 69 2E 64 6C 6C 00 00 00 00 00  phlpapi.dll.....
00000060  81 74 82 0D 5E 96 93 1C CA 96 93 1C D1 FE F0 EF  .t‚.^–“.Ę–“.Ńţđď
00000070  4F 5B A8 63 9D BB 93 1C A8 70 90 50 2C 66 48 2E  O[¨cť»“.¨p.P,fH.
00000080  F8 5C EF 6E 72 3C 94 7C 0B 0F B5 A5 D6 94 93 1C  ř\ďnr<”|..µĄÖ”“.
00000090  77 E2 E1 F9 89 5F B7 29 8D AF D2 7D F5 26 BD 6B  wâáů‰_·)ŤŻŇ}ő&˝k
000000A0  8B FC BF 7E 90 75 82 0D 30 B8 82 0D EA 3D 9D 7C  ‹üż~.u‚.0¸‚.ę=ť|
000000B0  E0 A9 E0 DA 65 9C 46 CE F3 16 E6 F5 3D AD 39 0D  ŕ©ŕÚeśFÎó.ćő=.9.
000000C0  87 F0 96 7C D9 16 1F A1 FF C3 E1 05 3A A2 48 C9  ‡đ–|Ů..ˇ˙Ăá.:˘HÉ
000000D0  02 0D 58 C6 0E 67 FD 1F C5 93 48 37 7F 8A D3 92  ..XĆ.gý.Ĺ“H7.ŠÓ’
000000E0  A0 96 72 11 D3 BB 7E 87 97 0F 2C 38 2E CF 8F 66   –r.Ó»~‡—.,8.ĎŹf
000000F0  58 3F 15 5A 1F BB 31 CF 01 97 88 87 0D 08 0C EA  X?.Z.»1Ď.—.‡...ę
00000100  9C F1 01 5D 0A AB C4 D2 61 D5 58 47 27 75 8D CA  śń.].«ÄŇaŐXG'uŤĘ
00000110  C5 6C BD 2A D7 EB BF D2 5B 49 87 74 ED EB BF D2  Ĺl˝*×ëżŇ[I‡tíëżŇ
00000120  10 C6 96 EB BA A1 CD EC 7E 10 B4 A3 07 CA 70 38  .Ć–ëşˇÍě~.´Ł.Ęp8
00000130  CE 40 94 05 5B BC CE 73 72 F3 6D A6 FA 34 19 49  Î@”.[ĽÎsróm¦ú4.I
00000140  83 C6 28 61 4E BE 3F 3B E0 59 2D F3 04 B1 4C 49  .Ć(aNI?;ŕY-ó.±LI
00000150  E2 99 94 7C 4B 35 3B 58 9A 15 9F 55 51 77 9A 0F  â™”|K5;Xš.źUQwš.
00000160  7F 5D D9 30 2F BD AE B1 4A 77 9A 0F 9A E7 2D F3  .]Ů0/˝®±Jwš.šç-ó
00000170  C5 0F 95 BC 55 8B EC 8B 45 08 53 56 33 F6 57 89  Ĺ.•ĽU‹ě‹E.SV3öW‰
```

It is common in malware that when the function's names are not given as an explicit string, they are imported by checksum. The same is done in this case. Guessing the appropriate function that was used for calculating the checksum can be more difficult. Fortunately, we found it in the loader component:

```
DWORD checksum(char *func_name)
{
  DWORD result = 0x1505;
  while ( *func_name )
    result = *func_name++ + 33 * result;
  return result;
}
```

Knowing that we paired appropriate checksums with the function's names:

```
c : WS2_32.dll
        6128c683 : WSAStartup
        3b3fbe4e : WSAGetLastError
        f32d59e0 : WSARecv
        494cb104 : closesocket
        7c9499e2 : bind
        583b354b : WSAIoctl
        559f159a : WSASocketA
        f9a7751 : htons
        30d95d7f : gethostbyname
        b1aebd2f : inet_addr
        f9a774a : htonl
        f32de79a : WSASend
1 : iphlpapi.dll
        bc950fc5 : GetAdaptersInfo
```

Once the address of the function is retrieved, it is stored in the IAT in place of the checksum.

Relocations

Creating a relocation table is simple. It consists of the list of DWORDs that are identifying the offsets of the places in the code to which we should add the base where the module has been loaded. Without relocations applied, the module will crash (so, it is not position-independent like a typical shellcode).

```
--RELOCS--
284 : c0
2c2 : bc
309 : c0
28b0 : 4be4
2959 : bc
2967 : c0
2992 : 13c
29ad : 90
29c8 : 134
29f7 : 48a0
29fd : 48a0
2a14 : 94
2a2a : 98
2a56 : 138
2ab3 : 4a40
2ab8 : 4c24
2acf : 4a34
```

Comparison to PE format

While the PE format is complex, with a variety of headers, this one contains only essentials. Most of the information that is usually stored in a PE header is completely omitted here.

You can see a PE format visualized by Ange Albertini here.

Compare it with the visualization of the currently analyzed format:

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

00000000  01 03 00 10 18 00 61 00 7A 0E 00 00 58 1E 00 00  ......a.z...X...   HEADERS
00000010  C8 01 00 00 90 1C 00 00                          Č.......
-----------------------------------------------------------------------------
                                05 00 6E 74 64 6C 6C 2E          ..ntdll.   DLLs
00000020  64 6C 6C 00 1B 00 4B 45 52 4E 45 4C 33 32 2E 64  dll...KERNEL32.d   {functions_count,
00000030  6C 6C 00 04 00 41 44 56 41 50 49 33 32 2E 64 6C  ll...ADVAPI32.dl   dll_name}
00000040  6C 00 04 00 43 61 62 69 6E 65 74 2E 64 6C 6C 00  l...Cabinet.dll.
00000050  03 00 4D 53 56 43 52 54 2E 64 6C 6C 00 00 00 00  ..MSVCRT.dll....
          00                                               .
-----------------------------------------------------------------------------
00000060     F9 58 B6 04 5E 96 93 1C 9D BB 93 1C CA 96 93   ůX¶.^–".ť»".Ę–"    IAT
00000070  1C 90 75 82 0D FB F0 BF 5F 56 F2 39 D6 B3 B0 DE  ..u,.űđż_Vň9Öł°Ţ   (checksums)
...
00000100  B6 87 F0 96 7C 3D AD 39 0D C7 0E E0 3D 64 A1 30  ¶‡đ–|=.9.Ç.ŕ=dˇ0
-----------------------------------------------------------------------------
00000110  00 00 00 C3 55 8B EC 83 EC 14 56 8B 75 08 57 33  ...ĂU‹ě.ě.V‹u.W3   CODE
00000120  FF 57 57 57 57 68 50 1A 00 00 89 7D FC FF 56 04  ˙WWWWhP...‰}ü˙V.   Entry Point = 0xE7A
...
00001C80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
-----------------------------------------------------------------------------
00001C90  26 01 00 00 3F 01 00 00 CF 01 00 00 D6 01 00 00  &...?...Ď...Ö...   RELOCATIONS
00001CA0  1F 02 00 00 26 02 00 00 92 02 00 00 A8 02 00 00  ....&...'...¨...   Size = 0x1C8
...
00001E40  43 14 00 00 49 14 00 00 4F 14 00 00 55 14 00 00  C...I...O...U...
00001E50  5B 14 00 00 61 14 00 00                          [...a...

                                                                              Module Size = 0x1E58
```

Static analysis

We can load this code into IDA as a blob of raw code. However, we will be missing important information. Due to the fact that the file doesn't follow a PE structure, and its import table is non-standard, we will have a hard time understanding which API calls are being made at which offset. To solve this problem, I made a tool that resolves hashes into function names and generates a TAG file to mark the offsets where each function's address is going to be filled.

core.sdb.tag

```
60;memchr
64;strcmp
68;strcpy
6c;tolower
70;_snwprintf
74;strlen
78;ZwOpenFile
7c;ZwClose
80;_snprintf
84;atoi
```

Those tags can be loaded into IDA using an IFL plugin:

```
seg000:00000060 dword_60        dd 0D827481h            ; DATA XREF: sub_3F7F+A9↓r
seg000:00000060                                         ; sub_3F7F+BD↓r ...
seg000:00000060                                         ; memchr
seg000:00000064 dword_64        dd 1C93965Eh            ; DATA XREF: sub_4758↓r
seg000:00000064                                         ; strcmp
seg000:00000068 dword_68        dd 1C9396CAh            ; DATA XREF: sub_4752↓r
seg000:00000068                                         ; strcpy
seg000:0000006C dword_6C        dd 0EFF0FED1h           ; DATA XREF: sub_365F:loc_369E↓r
seg000:0000006C                                         ; tolower
seg000:00000070 dword_70        dd 63A85B4Fh            ; DATA XREF: sub_30CF+25↓r
seg000:00000070                                         ; sub_338D+25↓r ...
seg000:00000070                                         ; _snwprintf
seg000:00000074 dword_74        dd 1C93BB9Dh            ; DATA XREF: sub_474C↓r
seg000:00000074                                         ; strlen
seg000:00000078 dword_78        dd 509070A8h            ; DATA XREF: sub_2CC2+49↓r
seg000:00000078                                         ; ZwOpenFile
```

Having all the API functions tagged, it is much easier to understand which actions are performed by the module. Here, for example, we can see that it will be establishing the connection with the C2 server:

```
000042E6 arg_0= dword ptr  4
000042E6 arg_4= dword ptr  8
000042E6
000042E6 push    esi
000042E7 push    [esp+4+arg_0]
000042EB
000042EB loc_42EB:                  ; inet_addr
000042EB call    ds:dword_164
000042F1 mov     esi, [esp+4+arg_4]
000042F5 cmp     eax, 0FFFFFFFFh
000042F8 mov     [esi+4], eax
000042FB jnz     short loc_4320

000042FD push    [esp+4+arg_0]
00004301 call    ds:dword_160     ; gethostbyname
00004307 test    eax, eax
```

Dynamic analysis

This format is custom, so it is not supported by the typical tools for analysis. However, after understanding it, we can write our own tools, such as the parser for the headers and loader that will help to run this format and analyze it dynamically.

In contrast to PE, the module doesn't have any sections. So, we need to load it in a continuous memory region with RWX (read-write-execute) access. Walking through the relocations list, we will add the value of the base at which the module was loaded to the listed addresses. Then, we have to resolve the imported functions by their hashes and fill the addresses in the thunks. After preparing the stage, it just needs to jump at the Entry Point of the module. We will load the prepared loader under the debugger and follow to the entry point of the loaded module.

Simple but rare

The elements described here are pretty simple—they serve as a first stage of the full malware package, downloading other pieces and injecting them into processes. However, what makes them interesting is the fact that their authors have shown some creativity and decided to invent a custom format that is less complex than a full-fledged PE, but goes a step further than a typical piece of shellcode.

Such module, in contrast to independent shellcode, is not self-sufficient and cannot be loaded in a trivial way, but must be parsed first. Given the fact that the format is custom, it is not supported by existing tools. This is where programming skills come in handy for a malware analyst.

Fortunately, fully custom formats are rather uncommon in the malware world; usually, authors rely heavily on existing formats, from time to time corrupting or customizing selected parts of PE headers.