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

Posted: August 30, 2018 by hasherezade August 30, 2018

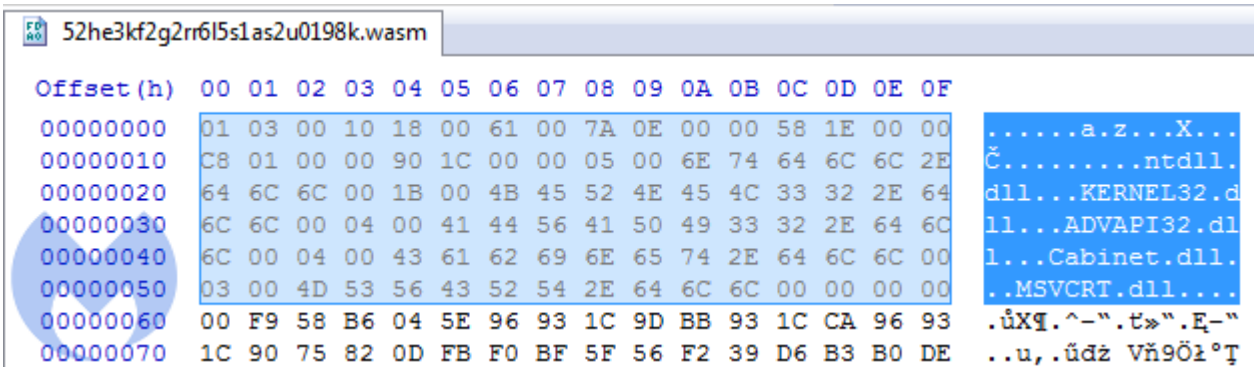
Malware can be made of many components. Often, we encounter macros and scripts that work as malicious downloaders. Some functionalities can also be achieved by position-independent code—so-called shellcode. But when it comes to more complex elements or core modules, we almost take it for granted that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

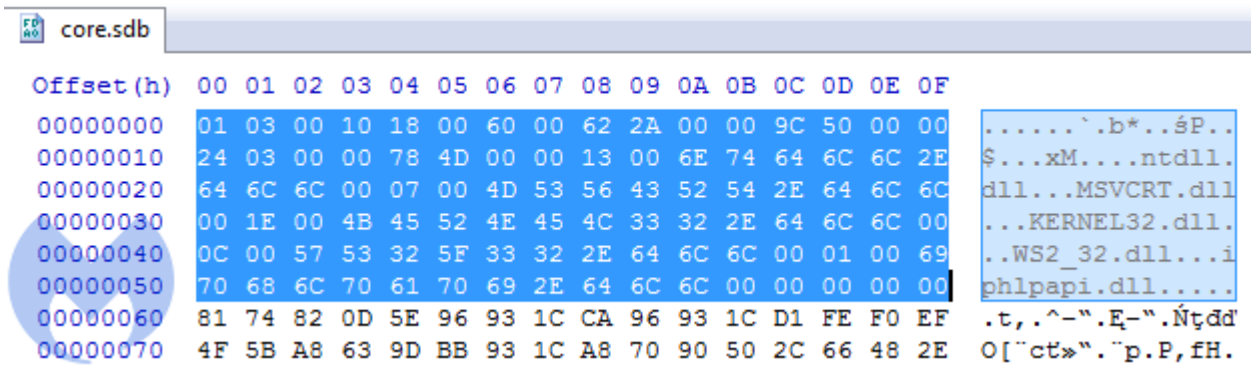
Overview

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


52he3kf2g2rr6l5s1as2u0198k.wasm

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...
00000010	C8	01	00	00	90	1C	00	00	05	00	6E	74	64	6C	6C	2E	Č.........ntdll.
00000020	64	6C	6C	00	1B	00	4B	45	52	4E	45	4C	33	32	2E	64	dll...KERNEL32.d
00000030	6C	6C	00	04	00	41	44	56	41	50	49	33	32	2E	64	6C	ll...ADVAPI32.dl
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....
00000060	00	F9	58	B6	04	5E	96	93	1C	9D	BB	93	1C	CA	96	93	.ůX¶.^-".ť».Ę-"
00000070	1C	90	75	82	0D	FB	F0	BF	5F	56	F2	39	D6	B3	B0	DE	..u,.űđż Vň9Ö³°Ţ

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

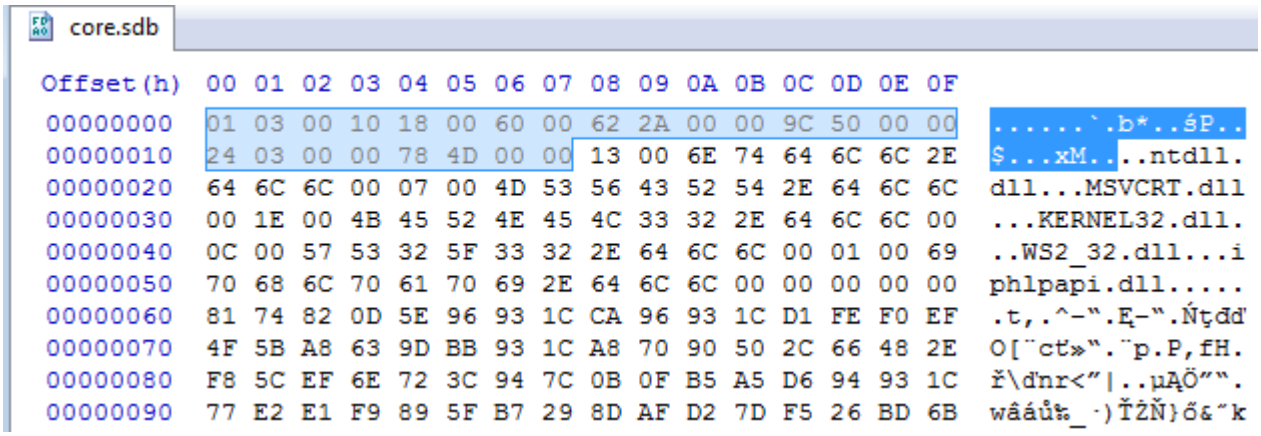

core.sdb

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.

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

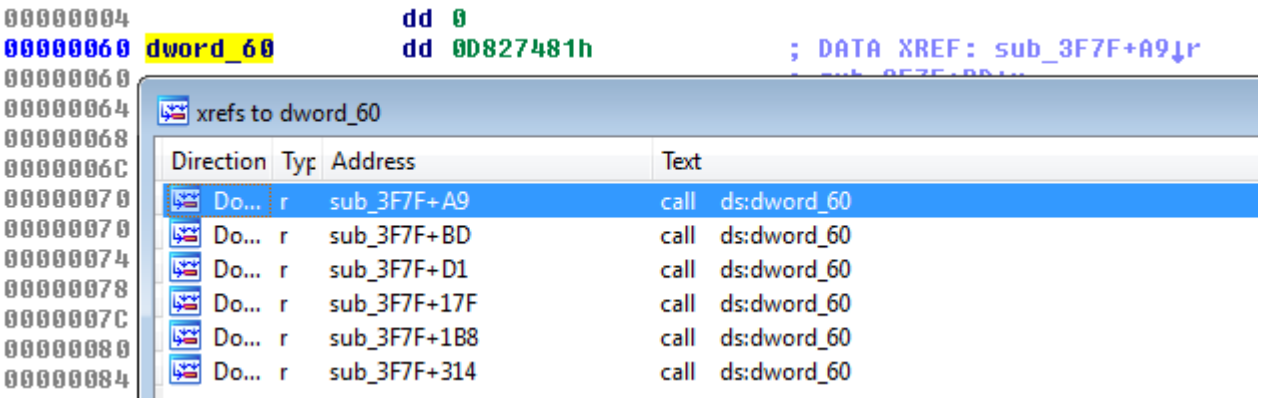
We took a closer look to decipher the meaning of particular fields in the header.


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:



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
```

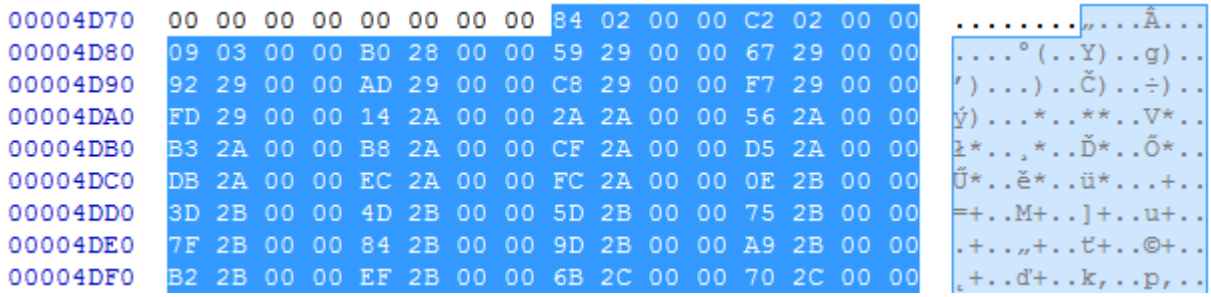
The next value is a DWORD (0x2A62). If we follow it in IDA, we see that it leads to the beginning of a new function:

```
: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.



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

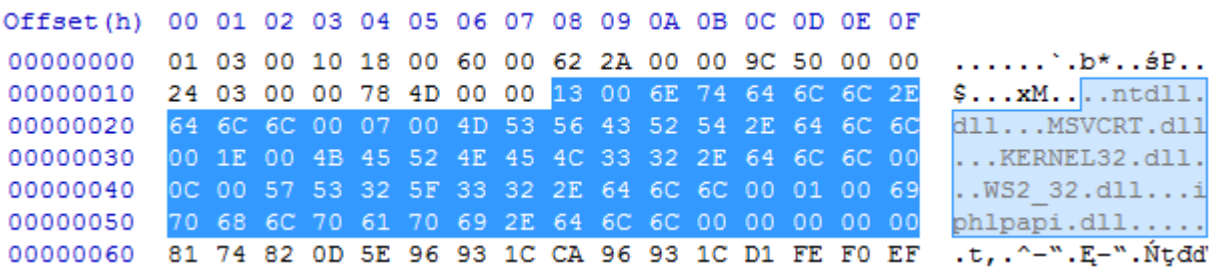
```
typedef struct {
        DWORD magic;

        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:

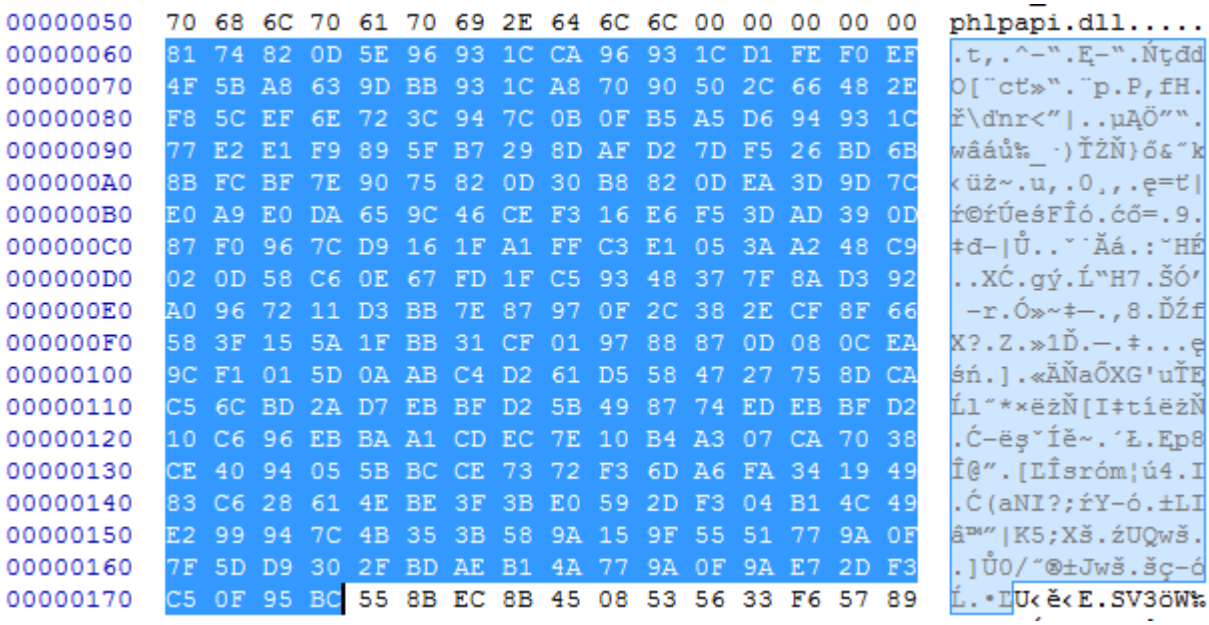


The numbers are not corresponding with a DLL name: In two different modules, the same DLL had different numbers assigned. But if we sum up all the numbers, we find that their total sum is the same as the number of DWORDs in the IAT. So, we can make an educated guess that those numbers are specifying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

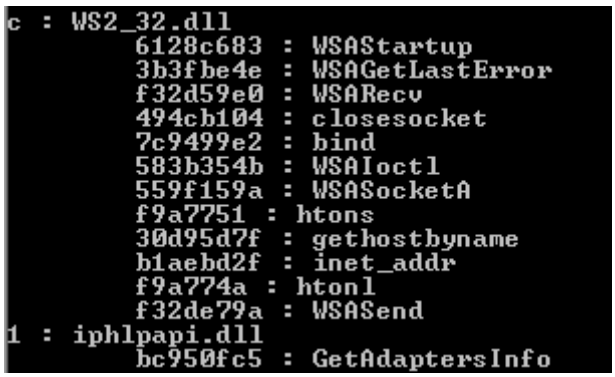


It is common in malware that when the function's names are not given as an explicit string, they are imported by checksum. The same is done in this case. Guessing the appropriate function that was used for calculating the checksum can be more difficult. Fortunately, we found it in the loader component:

```
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:

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).

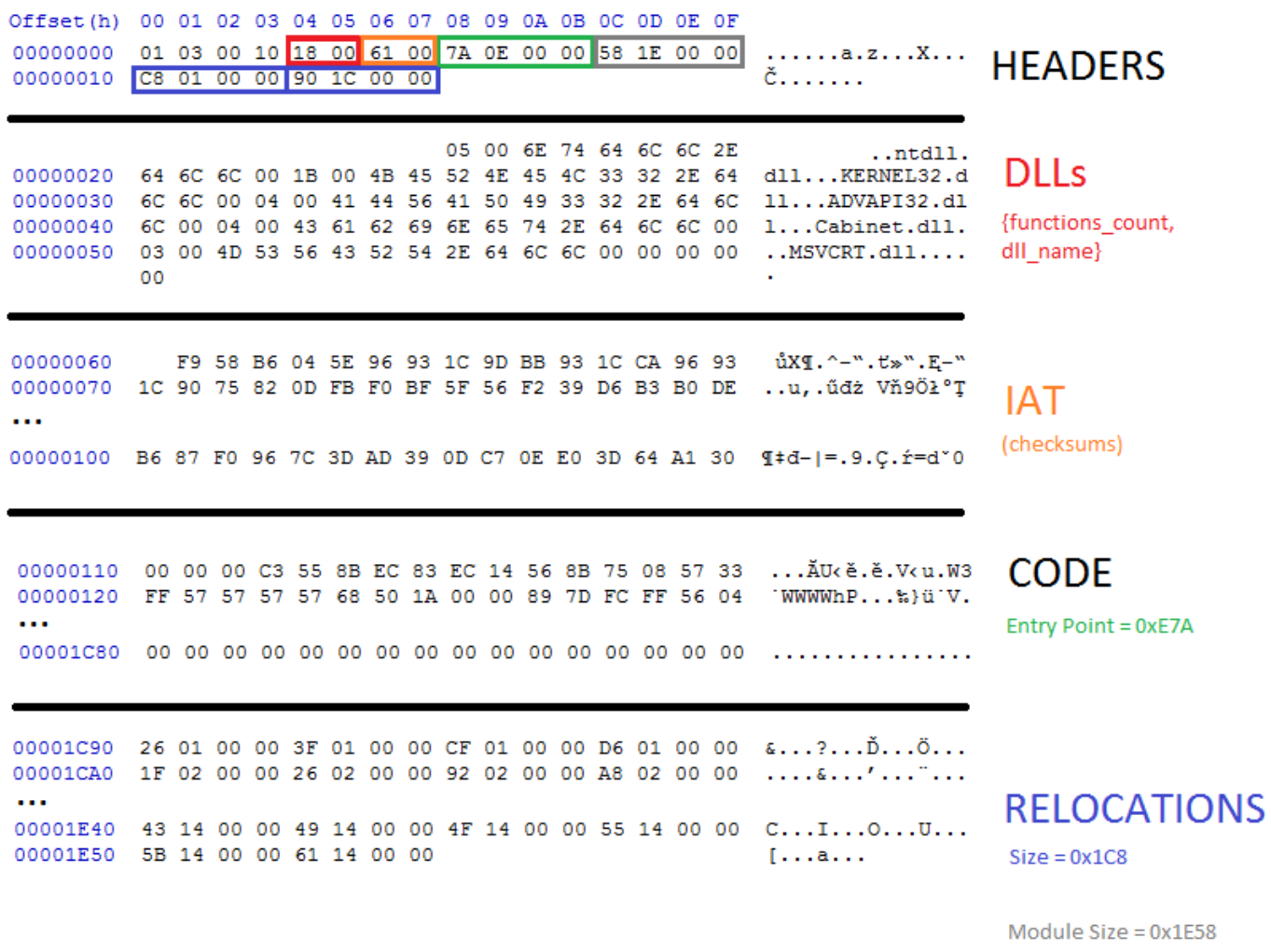
```
--RELOCS--
284 : c0
2c2 : bc
309 : c0
28b0 : 4be4
2959 : bc
2967 : c0
2992 : 13c
29ad : 90
29c8 : 134
29f7 : 48a0
29fd : 48a0
2a14 : 94
2a2a : 98
2a56 : 138
2ab3 : 4a40
2ab8 : 4c24
2acf : 4a34
```

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:



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.

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

core.sdb.tag

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

```
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.