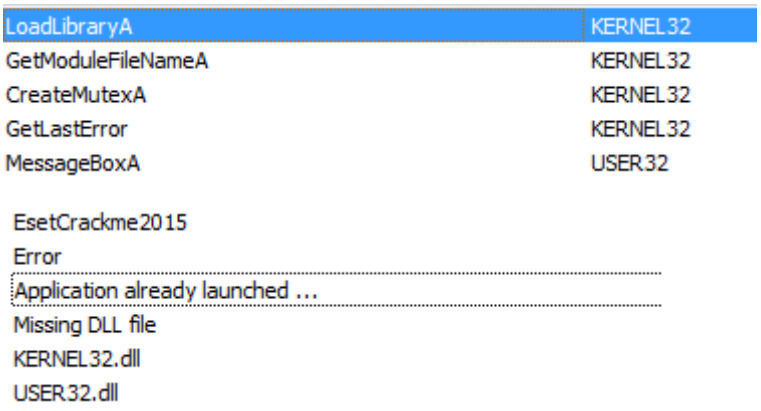
In this paper, we try to find first password which has expected us. When I downloaded the file which found from URL, it includes EsetCrackme2015.exe and EsetCrackme2015.dll files. Firstly, when I have static analysed for EsetCrackme2015.exe I found some interested artifacts.



There is no more imports or strings. The PE file is not packed with runtime-packer. Actually, the PE file proceed like a loader for DLL file. It uses CreateMutex to keep under control system resources and as an anti-debugging technique combined with CreateProcess.

```
push    offset Name ; "EsetCrackme2015"
push    1       ; bInitialOwner
push    0       ; lpMutexAttributes
call    CreateMutexA ; Indirect Call Near P
call    GetLastError ; Indirect Call Near P
cmp     eax, 0B7h ; Compare Two Operands
jnz     short loc_402375 ; Jump if Not Zerc
```

```
75341072 mov     edi, edi
75341074 push    ebp
75341075 mov     ebp, esp
75341077 push    0
75341079 push    [ebp+arg_24]
7534107C push    [ebp+arg_20]
7534107F push    [ebp+arg_1C]
75341082 push    [ebp+arg_18]
75341085 push    [ebp+arg_14]
75341088 push    [ebp+arg_10]
7534108B push    [ebp+arg_C]
7534108E push    [ebp+arg_8]
75341091 push    [ebp+arg_4]
75341094 push    [ebp+arg_0]
75341097 push    0
75341099 call    near ptr kernel32_CreateProcessInternalA
```

```
0207FFD0  001B15A6  debug033:001B15A6
0207FFD4  0050FD1C  debug028:0050FD1C
0207FFD8  00000000
0207FFDC  00000000
0207FFE0  00000000
0207FFE4  00000000
0207FFE8  00000004
```

CreateProcess arguments

```
0050FD10  1C FD 50 00 2E 03 00 10  44 FE 50 00 43 3A 5C 57  .ıP.....DşP.C:\W
0050FD20  69 6E 64 6F 77 73 5C 73  79 73 74 65 6D 33 32 5C  indows\system32\
0050FD30  73 76 63 68 6F 73 74 2E  65 78 65 00 00 00 00 00  svchost.exe.....
```

The second process(svchost.exe) will not be under the debugger's control.

Next, I have static analyzed to DLL file. According to DLL characteristics it must have include some exports more than standart PE file. But in this case, there is no any export or import or meaningful strings. Therefore, we should suspicious it performs dynamically resolves its library and functions.

```
:10000259 mov     edi, 811C9DC5h
:1000025E jz      short loc_1000028B
:10000260
:10000260 loc_10000260:
:10000260 mov     dl, [ecx]
:10000262 add     ecx, 2
:10000265 lea     ebx, [edx-61h]
:10000268 cmp     bl, 19h
:1000026B ja      short loc_10000270
:1000026D add     dl, 0E0h
:10000270
:10000270 loc_10000270:
:10000270 movsx   edx, dl
:10000273 xor     edx, edi
:10000275 imul    edx, 1000193h
:1000027B cmp     word ptr [ecx], 0
:1000027F mov     edi, edx
:10000281 jnz     short loc_10000260
:10000283 cmp     edi, 0FC706866h
```

When we run the program, it loads the DLL. The DLL's entry point is 0x10000226. It uses FNV-1a hash to calculate EsetCrackme2015.exe module name. The program searches some byte sequence. Thus, the execution will be continue new entry point.

```
mov     eax, [eax+10h]
test    eax, eax
jz      short loc_100002E0
mov     ecx, 1000h
mov     edx, 1010101h

loc_100002A6:
mov     esi, [ecx+eax]
add     esi, edx
cmp     esi, 0FB131506h
jnz     short loc_100002CF
mov     esi, [ecx+eax+4]
add     esi, edx
cmp     esi, 20C16ADFh
jnz     short loc_100002CF
mov     esi, [ecx+eax+8]
add     esi, edx
cmp     esi, 0C43360A2h
jz      short loc_100002DA
```

```
loc_100002DA:
lea     eax, [ecx+eax+0Ch]
call    eax
```

```
RAX 0000000000401E9F ↳ new_entrypoint
RBX 000000006FF24D04 ↳
```

We must search 0x01010101 minus for these bytes added to searched bytes.

```
00401E90  5E 5D C3 05 14 12 FA DE  69 C0 1F A1 5F 32 C3 57  ^]Ã...úŞiÀ.¡_2ÃW
00401EA0  33 FF 39 3D 30 10 40 00  74 2F 56 E8 F0 FC FF FF  3ÿ9=0.@.t/Vèğüÿÿ
```

The program uses FNV-1a hash as mentioned above to resolve library and functions as dynamically.

The procedure works like this:
- An iterator searches module names via InMemoryOrderModuleList.(1)

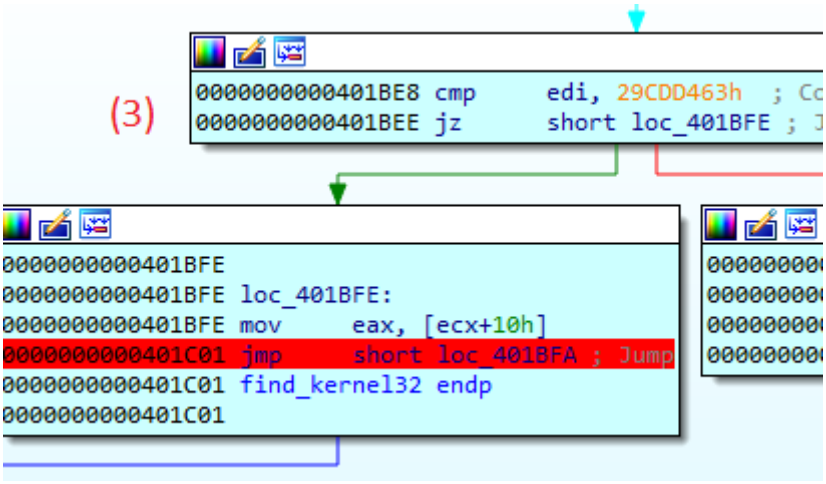
```
401BA0 ; _PEB
401BA0
401BA0 find_kernel32 proc near                    (1)
401BA0 mov     eax, large fs:30h
401BA6 mov     eax, [eax+0Ch]  ; _PEB_LDR_DATA
401BA9 mov     ecx, [eax+14h]  ; struct _LIST_ENTRY InMemoryOrderModuleList;
401BAC push    ebx
401BAD push    esi
401BAE push    edi
401BAF mov     esi, ecx
```

- The FNV-1a hash algorithm calculates it's hash to determine searched library name. (2)

```
00401BB3 loc_401BB3:
00401BB3 mov     eax, [ecx+28h]
00401BB6 test    eax, eax
00401BB8 jz      short loc_401BF8
00401BBA cmp     word ptr [eax], 0
00401BBE mov     edi, 811C9DC5h
00401BC3 jz      short loc_401BF0
00401BC5
00401BC5 loc_401BC5:
00401BC5 mov     dl, [eax]
00401BC7 add     eax, 2
00401BCA lea     ebx, [edx-61h]
00401BCD cmp     bl, 19h
00401BD0 ja      short loc_401BD5
00401BD2 add     dl, 0E0h
00401BD5
00401BD5 loc_401BD5:
00401BD5 movsx   edx, dl
00401BD8 xor     edx, edi
00401BDA imul    edx, 1000193h
00401BE0 cmp     word ptr [eax], 0
00401BE4 mov     edi, edx
00401BE6 jnz     short loc_401BC5
```

(2)

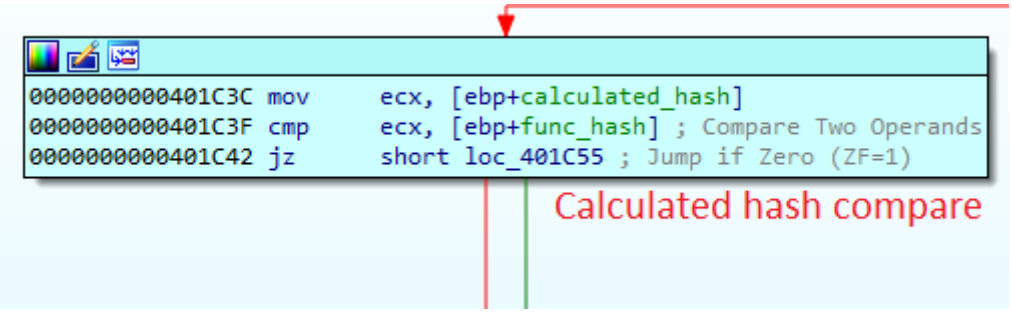
- Finally, it compares two hash value (3)



After completed library name hashing, the execution will be continue to resolve it's function names.

```
401EAA push    esi
401EAB call    find_kernel32   ; _PEB
401EB0 mov     esi, eax
401EB2 push    2FA62CA8h       ; Sleep
401EB7 call    resolve_exports ; Call Procedure
401EBC push    edi
401EBD push    edi
401EBE push    eax
401EBF push    offset Thread1_code
401EC4 push    edi
401EC5 push    edi
401EC6 push    60AC7E39h       ; CreateThread
401ECB mov     dword_401030, edi
401ED1 call    resolve_exports ; Call Procedure
401ED6 call    eax             ; Indirect Call
```

The "resolve_exports" routine works similar ways as mentionad above. Only the function's hash value transfers as argument to resolve_exports routine.



The EsetCrackme2015.exe creates two thread. I labeled it's execution code Thread1_code and Thread2_code. Thread1_code's most important part is which it calls sub_40213B.

```
4021BD push    ebx
4021BE push    ebx
4021BF push    offset Thread2_code
4021C4 push    ebx
4021C5 push    ebx
4021C6 push    60AC7E39h       ; CreateThread
4021CB mov     [ecx+10Dh], eax
4021D1 mov     dword ptr [ecx+119h], offset decrypt_buffer
4021DB call    resolve_function_hash ; Call Procedure
4021E0 call    eax             ; Indirect Call Near Procedure
4021E2 mov     esi, ptr_EsetDLL
4021E8 push    101h            Resource Identifier
4021ED mov     [esi+125h], eax
4021F3 mov     dword ptr [esi+115h], offset XORing_routine
4021FD mov     dword ptr [esi+111h], offset resolve_function_hash
402207 mov     [esi+108h], bl
40220D call    search_specific_bytes ; Call Procedure
402212 push    3
402214 mov     edi, eax        Resource Identifier
402216 call    search_specific_bytes ; Call Procedure
40221B pop     ecx
40221C pop     ecx
40221D cmp     edi, ebx        ; Compare Two Operands
```

The search_specific_bytes routine try to determine located some resource identifier. Before it was called, transferred an argument it which specified what resource searched.

The 101h resource header is located at 1000032E and it's size is 0E00h.

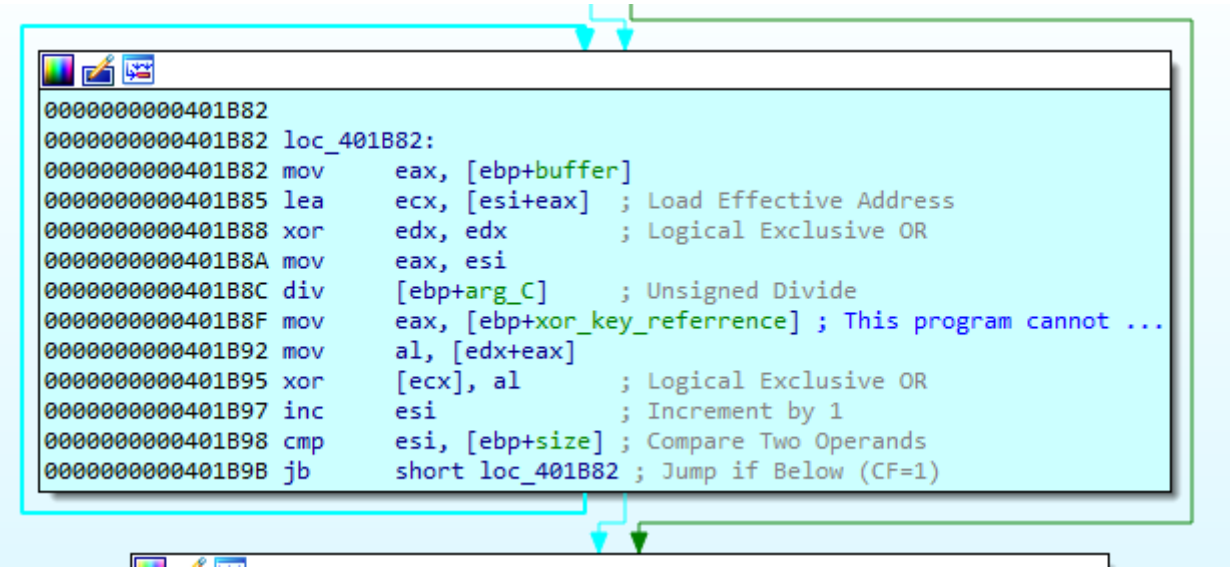
```
10000320  92 86 F6 1C 51 F1 16 68  B7 32 24 6A 5F F6 01 01  ’†ö.Qñ.h·2$j_ö..
10000330  00 0E 00 00 46 5C C8 05  4B 76 45 F9 B2 A4 AA 69  ....F\È.KvEù²¤ªi
10000340  92 2E 1A 1A F7 26 7C CE  98 8A 61 CA 50 16 8B AF  ’...÷&|Î˜ŠaÊP.‹¯
10000350  D8 C3 88 70 BD 64 3D B1  B4 A0 3A 85 BA 1C DB 32  ØÃˆp½d=±´ :…º.Û2
10000360  95 42 7A FC 32 3E 7C 0D  AF C5 C6 68 B8 4D 42 40  •Bzü2>|.¯ÅÆh¸MB@
10000370  D7 BF 6E 26 8F EC A5 B3  EA 44 4B 99 34 D5 13 C2  ×¿n&.ì¥³êDK™4Õ.Â
10000380  C7 46 37 41 60 56 D5 D6  3D D3 A3 82 95 D0 24 CB  ÇF7A`VÕÖ=Ó£‚•Ð$Ë
10000390  78 D4 11 D3 02 B2 0A 7A  A5 67 9B FD 5D 49 16 00  xÔ.Ó.².z¥g›ý]I..
100003A0  8B FD 41 A2 0D 45 6E 6B  04 54 B5 7B 9E FC A0 EC  ‹ýA¢.Enk.Tµ{.ü·ì
100003B0  0D B8 C5 24 A1 5E 34 2A  B3 DF 53 46 F1 BE 19 1D  .¸Å$¡^4*³ßSFñ¾..
```

The 3h resource header is located at 100001FA and it’s size is 20h.

```
100001F0  00 00 00 00 00 00 00 00  00 00 03 00 20 00 00 00  ............ ...
10000200  72 0C 22 10 29 77 44 1E  22 0D 30 11 0E 13 32 0D  r.".)wD.".0...2.
10000210  23 04 2D 00 7A 35 50 5A  2B 47 06 53 08 39 6E 2B  #.-.z5PZ+G.S.9n+
10000220  01 00 C3 00 00 00 83 7C  24 08 01 0F 85 B2 00 00  ..Ã...ƒ|$...…²..
```

Now, these resources is encrypted state. There is two decryption routine to decrypt these resources. I labeled these as XORing_routine(located at 401B77) and decrypt_buffer(located at 401B11). The XORing_routine decrypts resource located at 10000200.

XORing_routine get four parameters. These are buffer, size, xor_key_referrence and I cannot defined last argument. This routine uses PE header data like "This program cannot ..." for XOR key value.

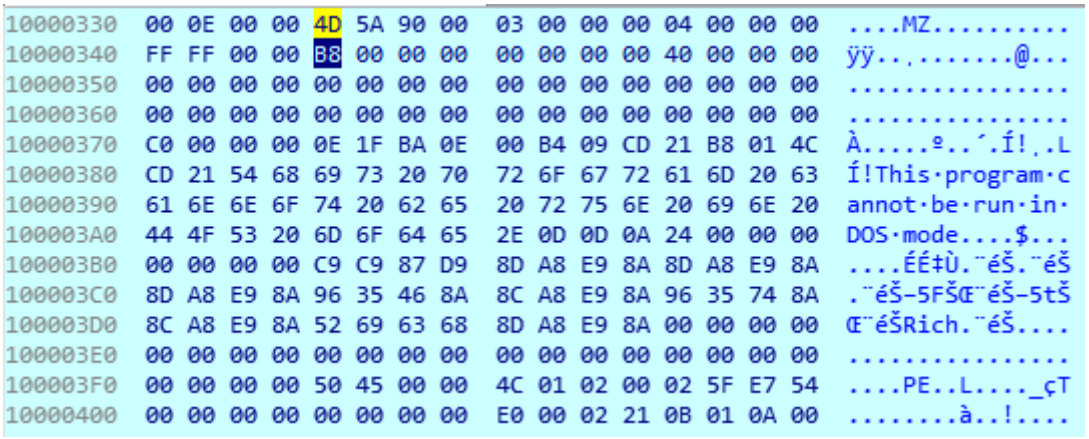


After XORing_routine running;

```
100001F0  00 00 00 00 00 00 00 00  00 00 03 00 20 00 00 00  ............ ...
10000200  53 58 4A 79 5A 57 34 6C  4D 6A 42 70 63 33 51 6C  SXJyZW4lMjBpc3Ql
10000210  4D 6A 42 74 5A 57 35 7A  59 32 68 73 61 57 4E 6F  MjBtZW5zY2hsaWNo
```

This data have used to XOR encryption/decryption operations as key value in decrypt_buffer routine.

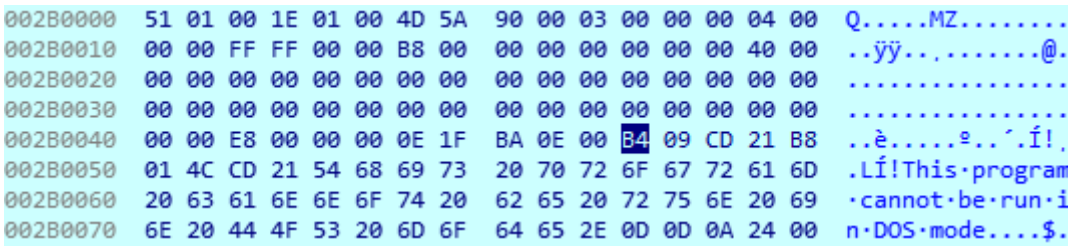
The decrypt_buffer routine gets three arguments as labeled buffer(encrypted), size(encrypted buffer size) and XORkey_buffer.



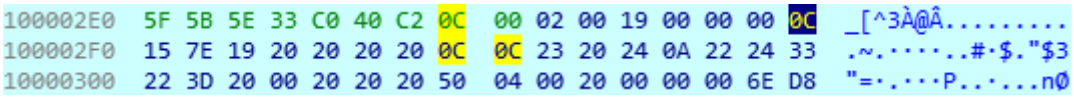
(Decrypted 101h resource)

When I was debugging, I had detected some VirtualAlloc function resolving.
The DLL contains embedded buffer started at 10001134. The buffer is accessible through finding header bytes like(start points with buffer) 102h(200Ch size), 103h(2AFh size), 104h(C5h size). Also allocated memory spaces fills with located buffers.

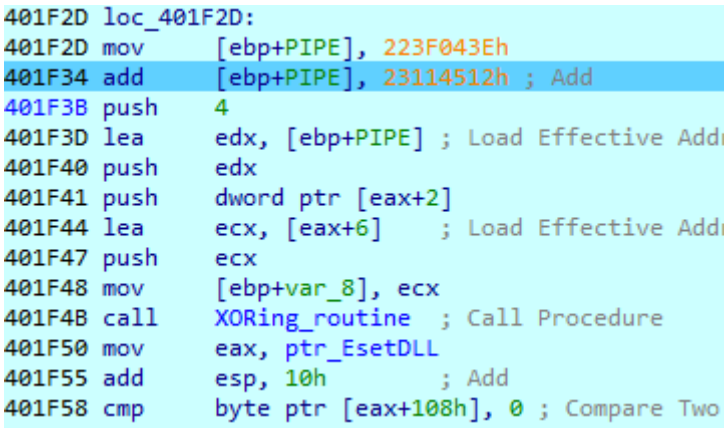
151h → Its buffer at 100034C0, allocated and filled new memory. Probably for svchost.exe(Allocated memory at 2B0000).



Now, We can analyze Thread2_code. The Thread2_code located at 401F13. Thread2_code also calls earlier mentioned routines with different resource identifier. Firstly, It searches 2h resource identifier and was found at 100002E9.



This data is encrypted. The XORing_routine will be decrypt encrypted data using "PIPE" key.



After decryption;

```
100002E0  5F 5B 5E 33 C0 40 C2 0C  00 02 00 19 00 00 00 5C  _[^3À@Â........\
100002F0  5C 2E 5C 70 69 70 65 5C  45 73 65 74 43 72 61 63  \.\pipe\EsetCrac
10000300  6B 6D 65 50 69 70 65 00  04 00 20 00 00 00 6E D8  kmePipe...·...nØ
```

The "2h" resource header specifies a named pipe like "\\.\pipe\EsetCrackmePipe".

In this point, I want to guess something about these artifacts.

1. If we have detected a named pipe, must be a client pipe and server pipe which communicate via named pipe.

2. For the above be true, must be additional process creation. This theory can validate easyly with Process Monitor. Also, another way set a breakpoint CreateProcessA and analyze the arguments. Svchost.exe process created as CREATE_SUSPENDED state and does not run until the ResumeThread function is called. (Process Hollowing – Process Replacement)

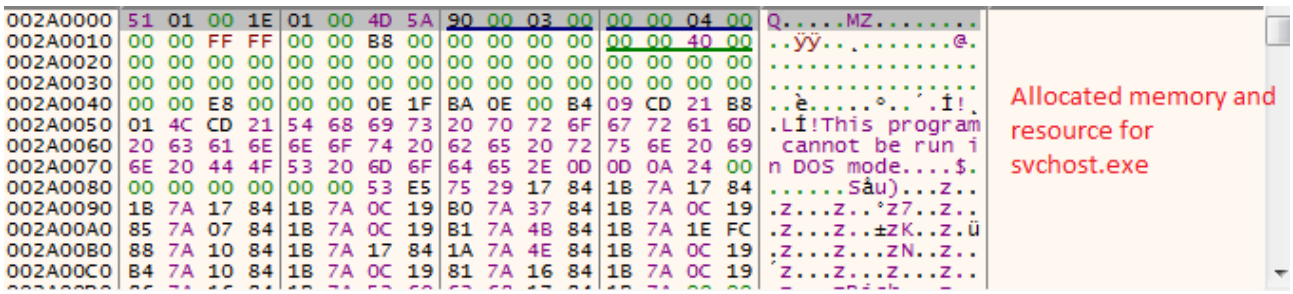
3. Probably, there are some embedded buffers in earlier decrypted PE file(at 10000334). This guess require to determine memory allocation, writing data to allocated memory.

The Thread2_code calls CreateNamedPipe and ConnectNamedPipe functions. Therefore, we can decide to EsetCrackme2015.exe is server pipe.

```
401F88 push    edx
401F89 push    3
401F8B push    [ebp+var_8]
401F8E sub     esi, ebx        ; Integer Subtractic
401F90 push    0A215C401h      ; CreateNamedPipe
401F95 xor     esi, edi        ; Logical Exclusive
401F97 call    resolve_exports ; Call Procedure
401F9C call    eax             ; Indirect Call Near
401F9E mov     ecx, ptr_EsetDLL
401FA4 mov     esi, [ecx+129h]
401FAA push    0
401FAC push    eax
401FAD sub     esi, ebx        ; Integer Subtractic
401FAF push    58D5D3E6h       ; ConnectNamedPipe
401FB4 xor     esi, edi        ; Logical Exclusive
401FB6 mov     [ecx+121h], eax
401FBC call    resolve_exports ; Call Procedure
401FC1 call    eax             ; Indirect Call Near
401FC3 test    eax, eax        ; Logical Compare
401FC5 jnz     short loc_401FEE ; Jump if Not Zero
```

After this stage, the execution will be continue via named pipes. To understand this stage, we must understand how named pipes works. Also I will be validate my guess above mentioned. Now, I break out from IDA and I will debug in OllyDbg.

Firstly, to extract svchost.exe resource before process injection, I set a breakpoint on VirtualAlloc. The reason of this, I want to show the whether client pipe is svchost.exe.



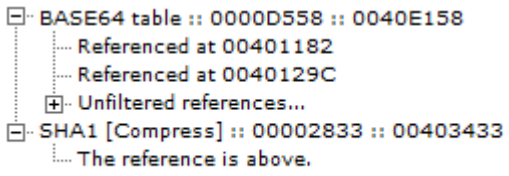
We can dump of memory with Scylla to examine PE dump. Don't remember to remove garbage bytes in PE header.

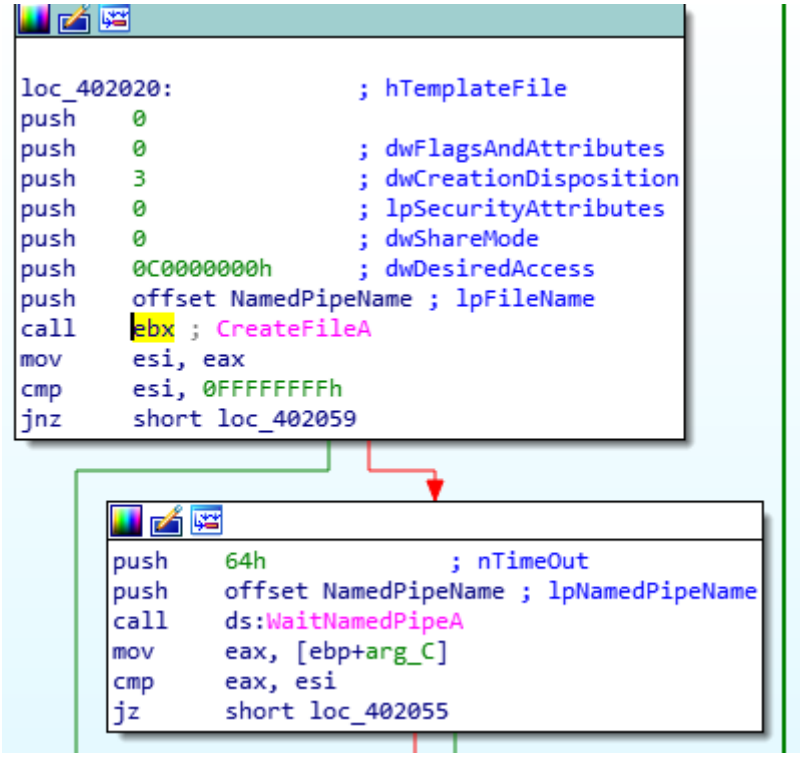
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. o
.text	400	B000	1000	AF45	60000020	0	0
.rdata	B400	3200	C000	3094	40000040	0	0
.data	E600	2000	10000	41A0	C0000040	0	0
.rsrc	10600	200	15000	1A8	40000040	0	0
.reloc	10800	1600	16000	1452	42000040	0	0

That's good! It's like unmapped file. There is no problem in section alignment. We continue. If my guess is true, I should find some artifacts in dumped PE file. By reading the manual documentation about named pipes I decided what must find there.

- If svchost.exe is a client pipe, it must call CreateFile function with pipe name argument.
- It must call SetNamedPipeHandleState to change read/write mode from/to pipe.
- Also another good indicators are ReadFile, WriteFile and WaitNamedPipeA functions used to named pipe operations.

In additional, it includes Base64 and SHA1 algorithms. I will be explain later.





Also there is another way to dump PE resource. We have already know used process injection technique earlier. You can set a break point on ResumeThread function before program is executing, and dump it's resource. I have used this way to continue for analyze.

While I was analyze both programs I synchronize it. That means, there are debugger session for both server and client pipe. To do this, I set a breakpoint at ResumeThread before client is executing. Thus, I have attached the debugger to suspended process in another session. Next, to examine pipe operations both server and client pipe set breakpoints on CreateFileA, ReadFile, WriteFile and FlushFileBuffers functions. Reason of this, both server and client pipes write data to pipe and other read from pipe.

Firstly, the client pipe resolves which it's will use pipe name with "PIPE" key.

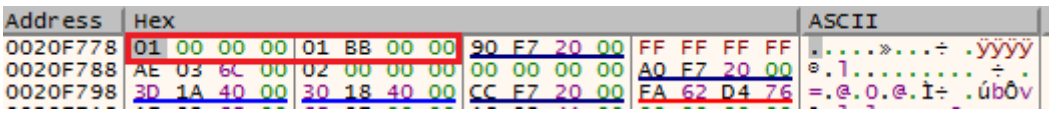
\\\\.\\pipe\\EsetCrackmePipe

```
000401F98 57                        push    edi
000401F99 C7 45 FC 3E C4 37+        mov     [ebp+PIPE], 2237C43Eh
000401FA0 81 45 FC 12 85 18+        add     [ebp+PIPE], 23188512h ; Add
000401FA7 BF 01 00 00 00            mov     edi, 1
000401FAC 33 C0                     xor     eax, eax ; Logical Exclusive OR
000401FAE 83 C9 FF                  or      ecx, 0FFFFFFFFh ; Logical Inclusive OR
000401FB1 81 EF 60 0E 41 00         sub     edi, offset NamedPipeName ; Integer Sub
000401FB7 EB 07                     jmp     short loc_401FC0 ; Jump
```

```
3401FC0
3401FC0                     loc_401FC0:
3401FC0 8D B4 07 60 0E 41+          lea     esi, NamedPipeName[edi+eax] ; Load Effecti
3401FC7 83 E6 03                    and     esi, 3  ; Logical AND
3401FCA 0F B6 5C 35 FC              movzx   ebx, byte ptr [ebp+esi+PIPE] ; Move with Z
3401FCF 30 98 61 0E 41 00           xor     byte_410E61[eax], bl ; Logical Exclusive C
3401FD5 8D 71 FF                    lea     esi, [ecx-1] ; Load Effective Address
3401FD8 83 E6 03                    and     esi, 3  ; Logical AND
3401FDB 0F B6 5C 35 FC              movzx   ebx, byte ptr [ebp+esi+PIPE] ; Move with Z
3401FE0 30 98 62 0E 41 00           xor     byte_410E62[eax], bl ; Logical Exclusive C
3401FE6 8B F1                       mov     esi, ecx
3401FE8 8B D0                       mov     edx, eax
3401FEA 83 E6 03                    and     esi, 3  ; Logical AND
3401FED 0F B6 5C 35 FC              movzx   ebx, byte ptr [ebp+esi+PIPE] ; Move with Z
3401FF2 30 98 63 0E 41 00           xor     byte_410E63[eax], bl ; Logical Exclusive C
3401FF8 83 E2 03                    and     edx, 3  ; Logical AND
3401FFB 8A 54 15 FC                 mov     dl, byte ptr [ebp+edx+PIPE]
3401FFF 30 90 60 0E 41 00           xor     NamedPipeName[eax], dl ; Logical Exclusive
```

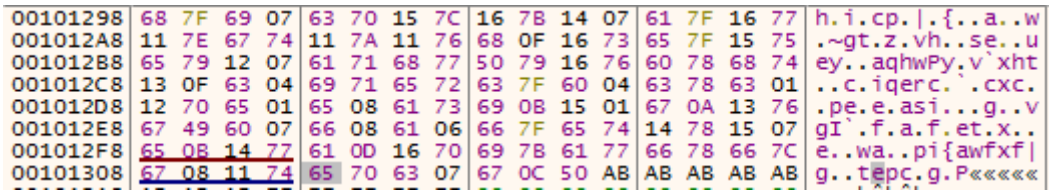
Client pipe writes 0x1 and BB01h bytes to pipe, respectively.

```
4020DF    8B1D 34C04000     mov ebx,dword ptr ds:[<&WriteFile>]
4020E5    57                push edi
4020E6    8D55 F8           lea edx,dword ptr ss:[ebp-8]
4020E9    52                push edx
4020EA    6A 01             push 1
4020EC    8D45 08           lea eax,dword ptr ss:[ebp+8]
4020EF    50                push eax
4020F0    56                push esi
4020F1    897D F8           mov dword ptr ss:[ebp-8],edi
4020F4    897D F4           mov dword ptr ss:[ebp-C],edi
4020F7    FFD3              call ebx
4020F9    57                push edi
4020FA    8D4D F8           lea ecx,dword ptr ss:[ebp-8]
4020FD    51                push ecx
4020FE    6A 02             push 2
402100    8D55 0C           lea edx,dword ptr ss:[ebp+C]
402103    52                push edx
402104    56                push esi
402105    FFD3              call ebx
402107    8B1D 38C04000     mov ebx,dword ptr ds:[<&ReadFile>]
40210D    57                push edi
```

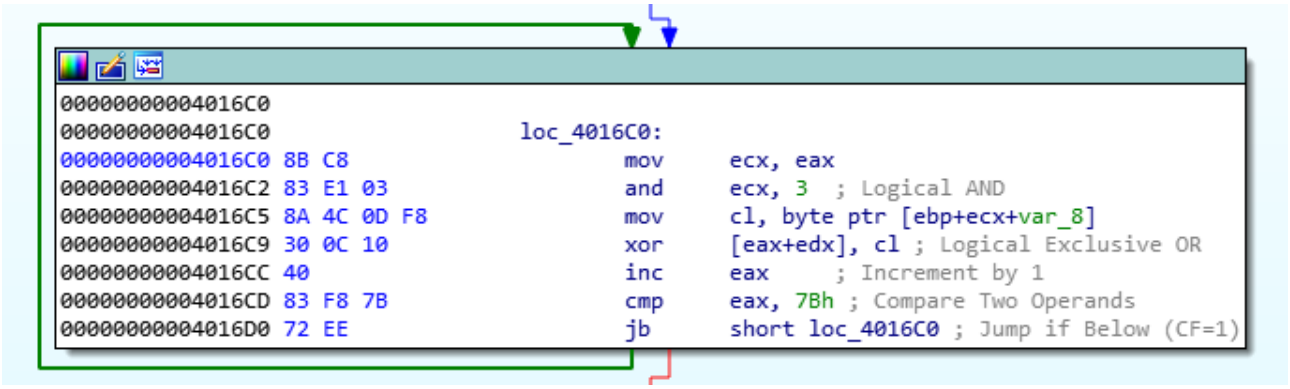


And next, read it's data from pipe. BB01(7B size) located at 10098997, BB02(11h size) located at 10098A18 in EsetCrackme2015.dll.

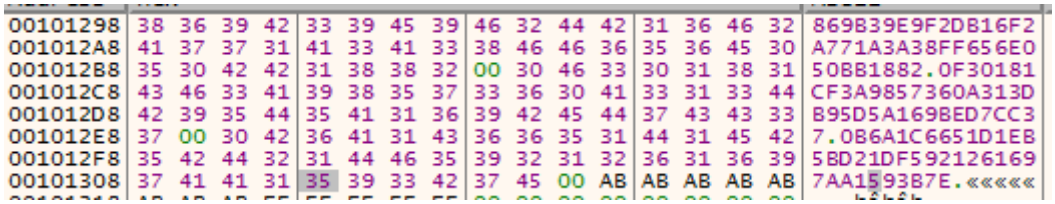
BB01h encrypted resource:



This branch is decryption loop for BB01h resource.



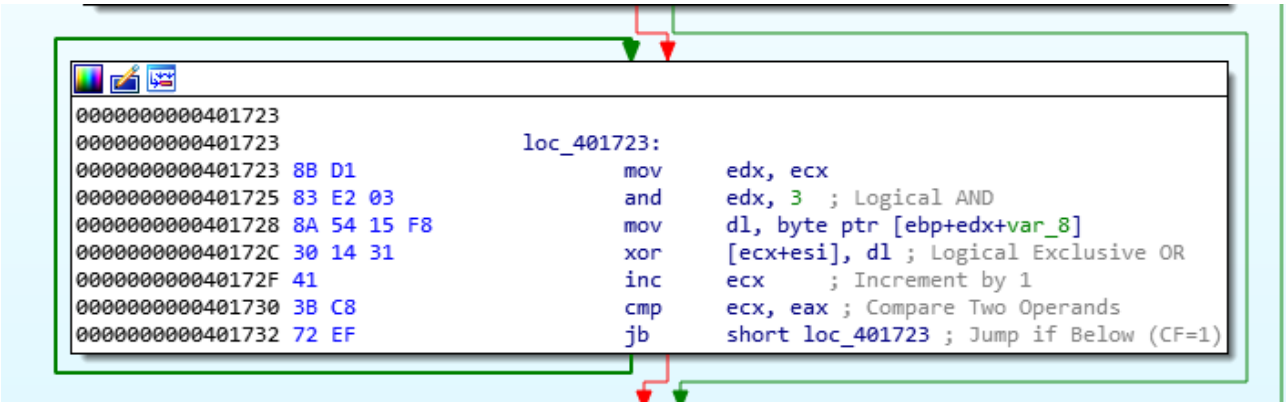
Decrypted form:

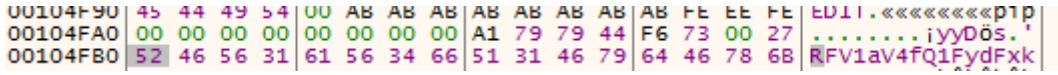


It includes three data but I have no idea what for it is use for now.

869B39E9F2DB16F2A771A3A38FF656E050BB1882
0F30181CF3A9857360A313DB95D5A169BED7CC37
0B6A1C6651D1EB5BD21DF5921261697AA1593B7E

All of operations mentioned above for BB01h resource similar for BB02h resource. This branch is decryption loop for BB02h resource:

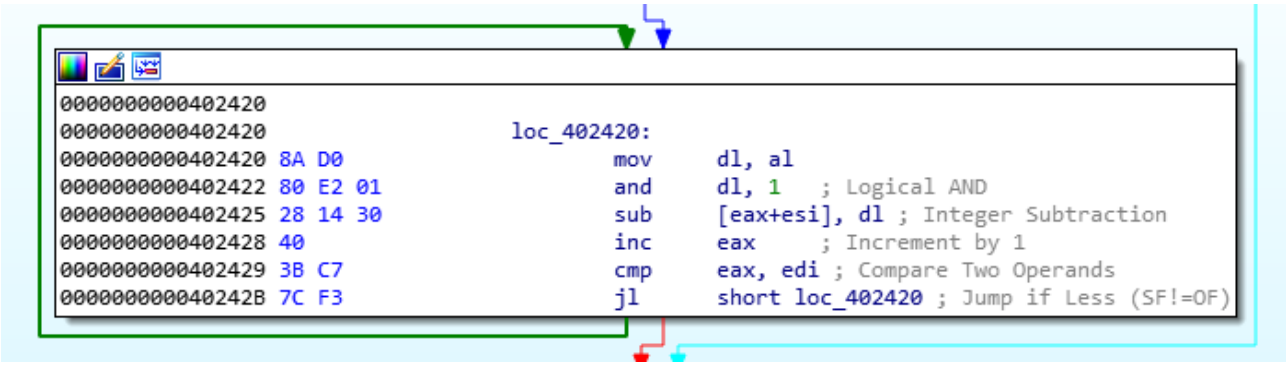




BB02h resource decrypted form: RFV1aV4fQ1FydFxk

I have determined Base64 and SHA1 algorithms earlier. Therefore, I set a breakpoint Base64(located at 401140) and SHA1(located at 402510).

When I entered first password, EIP hit the Base64 breakpoint. I have entered “test” password and Base64 encoded form is dGVzdA==. But in addition to base64, it performs simple logical and(&&) operation over base64 output.



This small encoding routine convert base64 output to "dFVyd@=<" and compares with RFV1aV4fQ1FydFxk. If it’s equal return zero and extract drv.zip resource to disk.

The routine basically run over indexes of base64 string's. Firstly, the logical and(&&)operation performs between string's index and 0x1 value. The result of this operation always either 0 or 1. According to result, the result substract from index value. It is written instead of the corresponding letter in the ASCII table.

0	1	2	3	4	5	6	7
d	G	V	z	d	A	=	=
0&&1	1&&1	2&&1	3&&1	4&&1	5&&1	6&&1	7&&1
0	1	0	1	0	1	0	1
d-0	G-1	V-0	z-1	d-0	A-1	=-0	=-1
d	F	V	y	d	@	=	<

This routine can be revert. If we can find the raw base64 encoded password, simply we can decode base64 encoded password.

RFV1aV4fQ1FydFxk → Base64 encoded + changed a little bit

According to above algorithm, the raw base64 encoded password is RGV2aW4gQ2FzdGxl

Any more only we need to do decoding base64 this password using any base64 decoder.

The first password is **Devin Castle**.

After entered this password, drv.zip resources was written to pipe and client reads it. Next writes to disk.