

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.

LoadLibraryA	KERNEL32
GetModuleFileNameA	KERNEL32
CreateMutexA	KERNEL32
GetLastError	KERNEL32
MessageBoxA	USER32
EsetCrackme2015	
Error	
Application already launched ...	
Missing DLL file	
KERNEL32.dll	
USER32.dll	

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 F
    call GetLastError ; Indirect Call Near F
    cmp eax, 0B7h ; Compare Two Operands
    jnz short loc_402375 ; Jump if Not Zerc

```

```

075341072 mov edi, edi
075341074 push ebp
075341075 mov ebp, esp
075341077 push 0
075341079 push [ebp+arg_24]
07534107C push [ebp+arg_20]
07534107F push [ebp+arg_1C]
075341082 push [ebp+arg_18]
075341085 push [ebp+arg_14]
075341088 push [ebp+arg_10]
07534108B push [ebp+arg_C]
07534108E push [ebp+arg_8]
075341091 push [ebp+arg_4]
075341094 push [ebp+arg_0]
075341097 push 0
075341099 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 .1P.....D$P.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 0000000006FF24D04 ↪

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 ^]Ä...ú$!Ä.j_2ÄW
00401EA0  33 FF 39 3D 30 10 40 00 74 2F 56 E8 F0 FC FF FF 3ÿ9=0.@.t/Vègüÿÿ

```

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

```

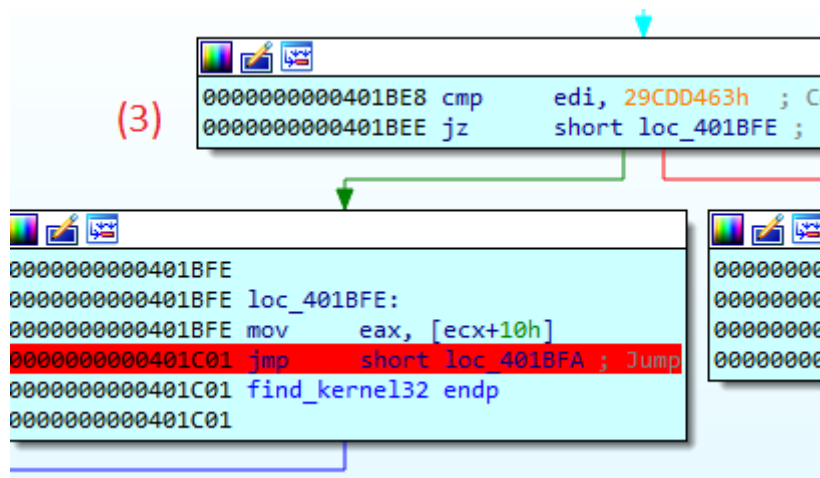
- The FNV-1a hash algorithm calculates its hash to determine searched library name. (2)

```

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 (2)
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

```

- Finally, it compares two hash value (3)



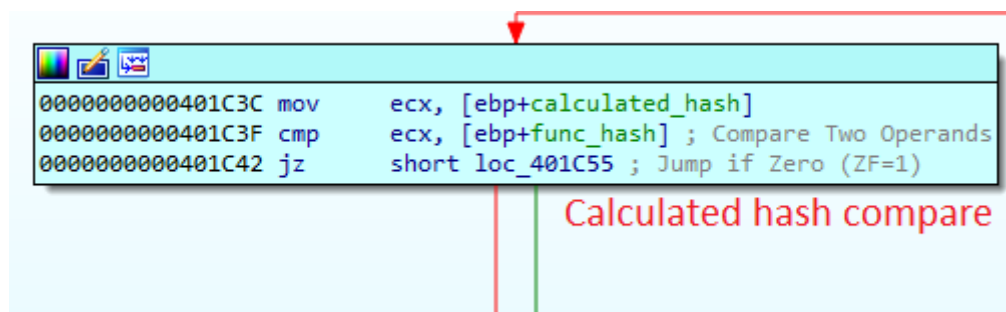
After completed library name hashing, the execution will be continue to resolve its function names.

```

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 mentioned 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
4021D8 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 ; Resource Identifier
402214 mov     edi, eax
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$j_ö..
10000330 00 0E 00 00 46 5C C8 05 4B 76 45 F9 B2 A4 AA 69 ....F\È.KvEu²H=i
10000340 92 2E 1A 1A F7 26 7C CE 98 8A 61 CA 50 16 8B AF '...÷&|Î~ŠaÊP.<~
10000350 D8 C3 88 70 BD 64 3D B1 B4 A0 3A 85 BA 1C DB 32 0Ä~p% d=±´.:...ô.Ú2
10000360 95 42 7A FC 32 3E 7C 0D AF C5 C6 68 B8 4D 42 40 •Bzü2>|.~Äeh,MB@
10000370 D7 BF 6E 26 8F EC A5 B3 EA 44 4B 99 34 D5 13 C2 x;n&.i¥³èDK™4Ö.Ä
10000380 C7 46 37 41 60 56 D5 D6 3D D3 A3 82 95 D0 24 CB ÇF7A`VÖÖ=Ó£,•ç$È
10000390 78 D4 11 D3 02 B2 0A 7A A5 67 9B FD 5D 49 16 00 xÔ.Ó.².z¥g>1]I..
100003A0 8B FD 41 A2 0D 45 6E 6B 04 54 B5 7B 9E FC A0 EC <1Aç.Enk.Tµ{.ü·i
100003B0 0D B8 C5 24 A1 5E 34 2A B3 DF 53 46 F1 BE 19 1D .,Ä$j_^4*³BSFñ%..

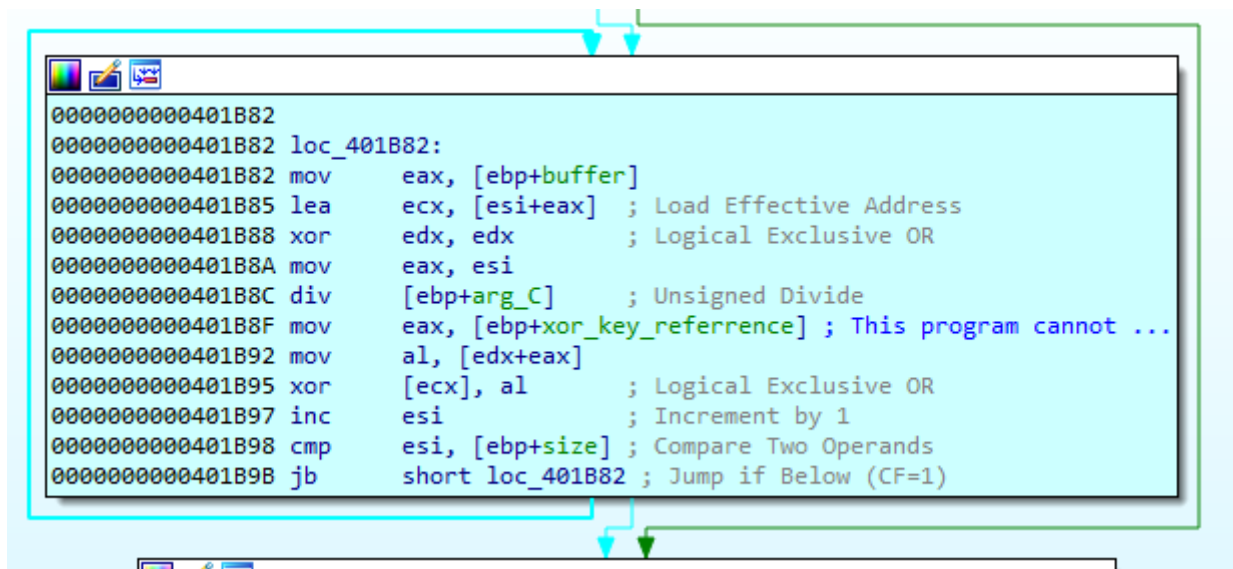
```

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...f|$.....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\_reference 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.

```

10000330 00 0E 00 00 4D 5A 90 00 03 00 00 00 04 00 00 00 .....MZ.....
10000340 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 yy.....@...
10000350 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
10000360 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
10000370 C0 00 00 00 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C À.....º..´.Í!..L
10000380 CD 21 54 68 69 73 20 70 72 6F 67 72 61 6D 20 63 Í!This·program·c
10000390 61 6E 6E 6F 74 20 62 65 20 72 75 6E 20 69 6E 20 annot·be·run·in·
100003A0 44 4F 53 20 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 DOS·mode....$....
100003B0 00 00 00 00 C9 C9 87 D9 8D A8 E9 8A 8D A8 E9 8A ....ÉÉ±Ù.."éŠ.."éŠ
100003C0 8D A8 E9 8A 96 35 46 8A 8C A8 E9 8A 96 35 74 8A ."éŠ-5FŠœ"éŠ-5tŠ
100003D0 8C A8 E9 8A 52 69 63 68 8D A8 E9 8A 00 00 00 00 œ"éŠRich.."éŠ....
100003E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
100003F0 00 00 00 00 50 45 00 00 4C 01 02 00 02 5F E7 54 ....PE..L...._cT
10000400 00 00 00 00 00 00 00 00 E0 00 02 21 0B 01 0A 00 .....à..!....

```

(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).

```

002B0000 51 01 00 1E 01 00 4D 5A 90 00 03 00 00 00 04 00 Q.....MZ.....
002B0010 00 00 FF FF 00 00 B8 00 00 00 00 00 00 40 00 ..yy.....@.
002B0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
002B0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
002B0040 00 00 E8 00 00 00 0E 1F BA 0E 00 B4 09 CD 21 B8 ..è.....º..´.Í!..
002B0050 01 4C CD 21 54 68 69 73 20 70 72 6F 67 72 61 6D .LÍ!This·program
002B0060 20 63 61 6E 6E 6F 74 20 62 65 20 72 75 6E 20 69 ·cannot·be·run·i
002B0070 6E 20 44 4F 53 20 6D 6F 64 65 2E 0D 0D 0A 24 00 n·DOS·mode....$.

```

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.

```

100002E0 5F 5B 5E 33 C0 40 C2 0C 00 02 00 19 00 00 00 0C _[^3À@Â.....
100002F0 15 7E 19 20 20 20 20 0C 0C 23 20 24 0A 22 24 33 .~.....#$. "$3
10000300 22 3D 20 00 20 20 20 50 04 00 20 00 00 00 6E D8 "=.....P.....n0

```

This data is encrypted. The XORing\_routine will be decrypt encrypted data using "PIPE" key.

```

401F2D loc_401F2D:
401F2D mov     [ebp+PIPE], 223F043Eh
401F34 add     [ebp+PIPE], 23114512h ; Add
401F3B push     4
401F3D lea     edx, [ebp+PIPE] ; Load Effective Address
401F40 push     edx
401F41 push     dword ptr [eax+2]
401F44 lea     ecx, [eax+6] ; Load Effective Address
401F47 push     ecx
401F48 mov     [ebp+var_8], ecx
401F4B call    XORing_routine ; Call Procedure
401F50 mov     eax, ptr_EsetDLL
401F55 add     esp, 10h ; Add
401F58 cmp     byte ptr [eax+108h], 0 ; Compare Two

```



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.....n0
```

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 easily 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 Subtraction
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 Subtraction
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.

002A0000	51 01 00 1E	01 00 4D 5A	90 00 03 00	00 00 04 00	Q.....MZ.....
002A0010	00 00 FF FF	00 00 B8 00	00 00 00 00	00 00 40 00	..ÿÿ.....@.
002A0020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....
002A0030	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....
002A0040	00 00 E8 00	00 00 0E 1F	BA 0E 00 84	09 CD 21 B8	..è.....!.
002A0050	01 4C CD 21	54 68 69 73	20 70 72 6F	67 72 61 6D	.Li!This program
002A0060	20 63 61 6E	6E 6F 74 20	62 65 20 72	75 6E 20 69	cannot be run i
002A0070	6E 20 44 4F	53 20 6D 6F	64 65 2E 0D	0D 0A 24 00	n DOS mode....\$.
002A0080	00 00 00 00	00 00 53 E5	75 29 17 84	18 7A 17 84	.....Sâu)...z..
002A0090	18 7A 17 84	18 7A 0C 19	80 7A 37 84	18 7A 0C 19	.z...z...z7..z..
002A00A0	85 7A 07 84	18 7A 0C 19	B1 7A 4B 84	18 7A 1E FC	.z...z...zK..z..
002A00B0	88 7A 10 84	18 7A 17 84	1A 7A 4E 84	18 7A 0C 19	.z...z...zN..z..
002A00C0	84 7A 10 84	18 7A 0C 19	81 7A 16 84	18 7A 0C 19	.z...z...z..z..

Allocated memory and resource for 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.

BASE64 table :: 0000D558 :: 0040E158  
 ... Referenced at 00401182  
 ... Referenced at 0040129C  
 + Unfiltered references...  
 SHA1 [Compress] :: 00002833 :: 00403433  
 ... The reference is above.

```

loc_402020:                ; hTemplateFile
push    0
push    0                  ; dwFlagsAndAttributes
push    3                  ; dwCreationDisposition
push    0                  ; lpSecurityAttributes
push    0                  ; dwShareMode
push    0C000000h          ; dwDesiredAccess
push    offset NamedPipeName ; lpFileName
call    ebx ; CreateFileA
mov     esi, eax
cmp     esi, 0FFFFFFFFh
jnz     short loc_402059

push    64h                ; nTimeout
push    offset NamedPipeName ; lpNamedPipeName
call    ds:WaitNamedPipeA
mov     eax, [ebp+arg_C]
cmp     eax, esi
jz      short loc_402055

```



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, 0FFFFFFFh ; 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 Effective Address
3401FC7 83 E6 03          and     esi, 3 ; Logical AND
3401FCA 0F B6 5C 35 FC      movzx   ebx, byte ptr [ebp+esi+PIPE] ; Move with 2
3401FCF 30 98 61 0E 41 00  xor     byte_410E61[eax], bl ; Logical Exclusive OR
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 2
3401FE0 30 98 62 0E 41 00  xor     byte_410E62[eax], bl ; Logical Exclusive OR
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 2
3401FF2 30 98 63 0E 41 00  xor     byte_410E63[eax], bl ; Logical Exclusive OR
3401FF8 83 E2 03          and     edx, 3 ; Logical AND
3401FFB 8A 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

```

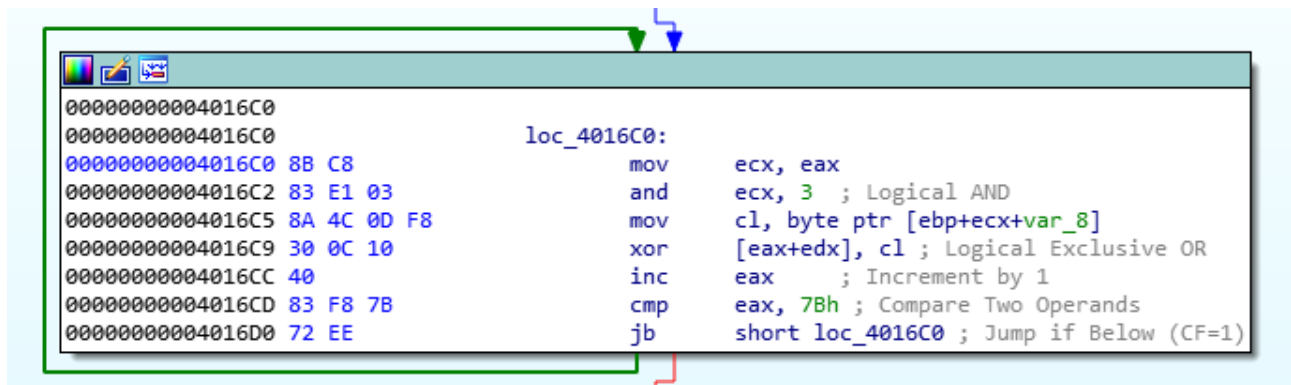
Address	Hex	ASCII
0020F778	01 00 00 00 01 BB 00 00 90 F7 20 00 FF FF FF FF	.....»...÷.yyyy
0020F788	AE 03 6C 00 02 00 00 00 00 00 00 00 A0 F7 20 00	®.l.....÷.
0020F798	3D 1A 40 00 30 18 40 00 CC F7 20 00 FA 62 D4 76	=.@.0.@.İ÷.úbôv

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:

00101298	68 7F 69 07	63 70 15 7C	16 7B 14 07	61 7F 16 77	h.i.cp. .{.a..w
001012A8	11 7E 67 74	11 7A 11 76	68 0F 16 73	65 7F 15 75	.~gt.z.vh..se..u
001012B8	65 79 12 07	61 71 68 77	50 79 16 76	60 78 68 74	ey..aqhwPy.v`xht
001012C8	13 0F 63 04	69 71 65 72	63 7F 60 04	63 78 63 01	..c.iqerc..cxc.
001012D8	12 70 65 01	65 08 61 73	69 08 15 01	67 0A 13 76	.pe.e.asi...g..v
001012E8	67 49 60 07	66 08 61 06	66 7F 65 74	14 78 15 07	gI`.f.a.f.et.x..
001012F8	65 08 14 77	61 0D 16 70	69 7B 61 77	66 78 66 7C	e..wa..pi{awfxf
00101308	67 08 11 74	65 70 63 07	67 0C 50 AB	AB AB AB AB	g..tëpç.g.P«««««

This branch is decryption loop for BB01h resource.



Decrypted form:

00101298	38 36 39 42	33 39 45 39	46 32 44 42	31 36 46 32	869B39E9F2DB16F2
001012A8	41 37 37 31	41 33 41 33	38 46 46 36	35 36 45 30	A771A3A38FF656E0
001012B8	35 30 42 42	31 38 38 32	00 30 46 33	30 31 38 31	50BB1882.0F30181
001012C8	43 46 33 41	39 38 35 37	33 36 30 41	33 31 33 44	CF3A9857360A313D
001012D8	42 39 35 44	35 41 31 36	39 42 45 44	37 43 43 33	B95D5A169BED7CC3
001012E8	37 00 30 42	36 41 31 43	36 36 35 31	44 31 45 42	7.0B6A1C6651D1EB
001012F8	35 42 44 32	31 44 46 35	39 32 31 32	36 31 36 39	5BD21DF592126169
00101308	37 41 41 31	35 39 33 42	37 45 00 AB	AB AB AB AB	7AA1593B7E.«««««

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:

```

0000000000401723
0000000000401723          loc_401723:
0000000000401723 8B D1          mov     edx, ecx
0000000000401725 83 E2 03      and     edx, 3 ; Logical AND
0000000000401728 8A 54 15 F8    mov     dl, byte ptr [ebp+edx+var_8]
000000000040172C 30 14 31      xor     [ecx+esi], dl ; Logical Exclusive OR
000000000040172F 41            inc     ecx ; Increment by 1
0000000000401730 3B C8          cmp     ecx, eax ; Compare Two Operands
0000000000401732 72 EF          jnb     short loc_401723 ; Jump if Below (CF=1)

```

00104F90	45 44 49 54	00 AB AB AB	AB AB AB AB	AB FE EE FE	ED 11	««««««««p1p
00104FA0	00 00 00 00	00 00 00 00	A1 79 79 44	F6 73 00 27	.....	yyDös.'
00104FB0	52 46 56 31	61 56 34 66	51 31 46 79	64 46 78 68	RFV1aV4fQ1FydF	Fxk

BB02h resource decrypted form: RFV1aV4fQ1FydFxx

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.

```

0000000000402420
0000000000402420          loc_402420:
0000000000402420 8A D0          mov     dl, al
0000000000402422 80 E2 01      and     dl, 1 ; Logical AND
0000000000402425 28 14 30      sub     [eax+esi], dl ; Integer Subtraction
0000000000402428 40            inc     eax ; Increment by 1
0000000000402429 3B C7          cmp     eax, edi ; Compare Two Operands
000000000040242B 7C F3          jl      short loc_402420 ; Jump if Less (SF!=OF)

```

00104F90	45 44 49 54	00 AB AB AB	AB AB AB AB	AB FE EE FE	ED 11	««««««««p1p
00104FA0	00 00 00 00	00 00 00 00	A1 79 79 44	F6 73 00 27	.....	yyDös.'
00104FB0	52 46 56 31	61 56 34 66	51 31 46 79	64 46 78 68	RFV1aV4fQ1FydF	Fxk

This small encoding routine convert base64 output to "dFVyd@=<" and compares with RFV1aV4fQ1FydFxx. 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 subtract 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.

RFV1aV4fQ1FydFxx → 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.