ESET CrackMe Analysis

Table of contents

List of figures

Part 1
First Stage

Part 1: First Stage

The first stage of this challenge is 32-bit windows executable, that when running request a valid password (Figure1)

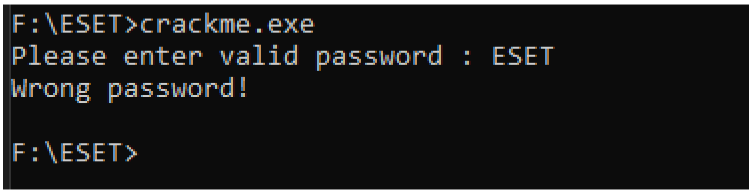


Figure 1: Request Password

using IDA Pro to disassemble the executable and see what is happening, it Do some Anti-Debugging Techniques using "IsDebuggerPresent", "GetTickCount", and check the "BeingDebugged" field in PEB Struct, and exit the program in case it detects debugger (Figure 2,3)

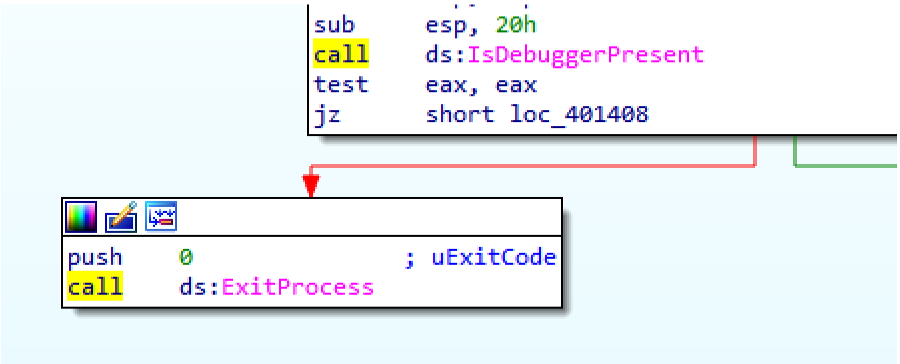


Figure 2: Anti-Debugging Technique using IsDebuggerPresent

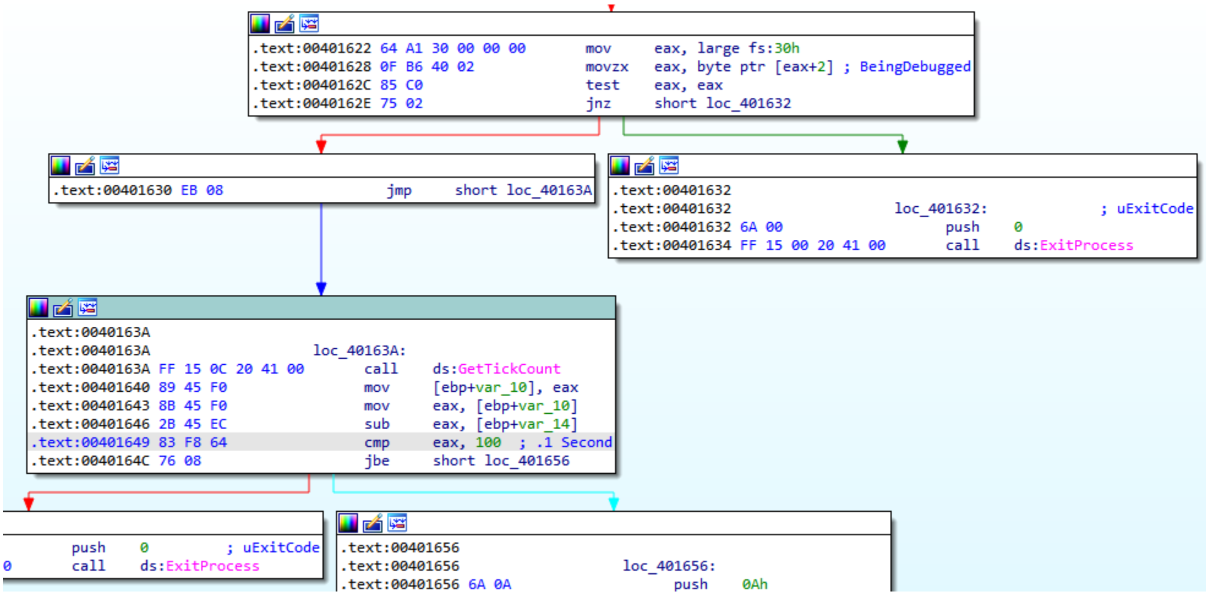


Figure 3:Anti-Debugging Techniques

it decrypts the buffer and output it to the console then it encrypts the buffer again (Figure 4), the Crypto function is simple XOR with byte key, the key is increment with a value on each loop iteration (Figure 5).

```
push    3                  ; Number to increment the Key With on each loop iteration
push    25h ; '%'          ; Key
push    1Fh                ; Buffer Len
push    offset Buffer      ; "uDNOBQ"
call    Decrypt_Encrypt_Buffer
```

Figure 4: Decrypting the Console Message

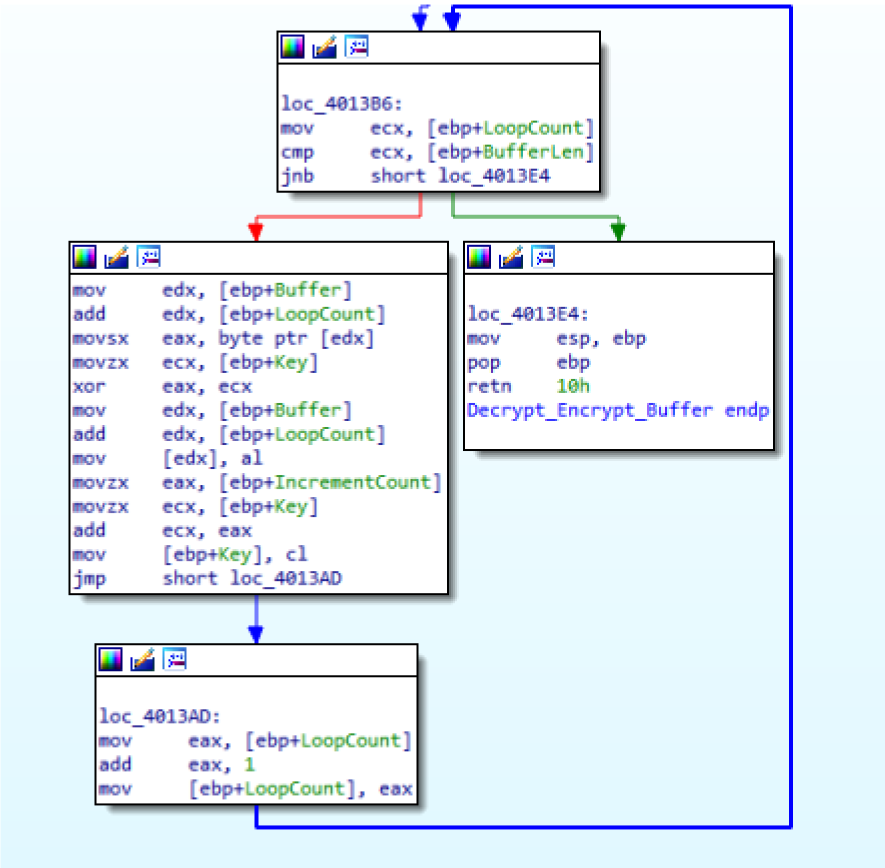


Figure 5: Decryption Function

Then it reads 10 characters from the console (password), After that it validates the password based on the following Equations

Password [7] + Password [6] = 0xCD

Password [8] + Password [5] = 0xC9

Password [7] + Password [6] + Password [3] = 0x13A

Password [9] + Password [4] + Password [8] + Password [5] = 0x16F

Password [1] + Password [0] = 0xC2

Password [0] + Password [1] + Password [2] + Password [3] + Password [4] + Password [5] + Password [6] + Password [7] + Password [8] + Password [9] = 0x39B

This is 10*10 Equation, but we have only 6 of them we need another 4 equations to be able to solve it, looking at the "WriteConsoleA" it is called 4 times one of them is call after we pass all the above password validation and hash validation of the password (figure 6), so using IDA Debugger and Set the Instruction pointer at the start of this block we get this message (figure 7)

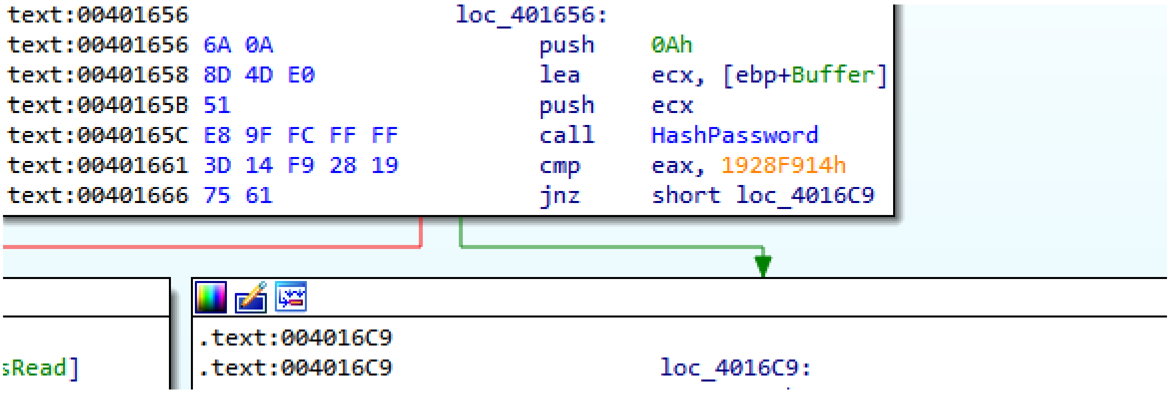


Figure 6: hash validation of the password

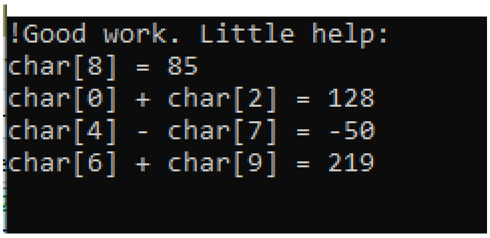


Figure 7: rest of Equations

Now we have the rest of the equation it can be solved using Octave, and the password is "Pr0m3theUs" running the challenge again and pass it the correct password gave us the message in figure 8

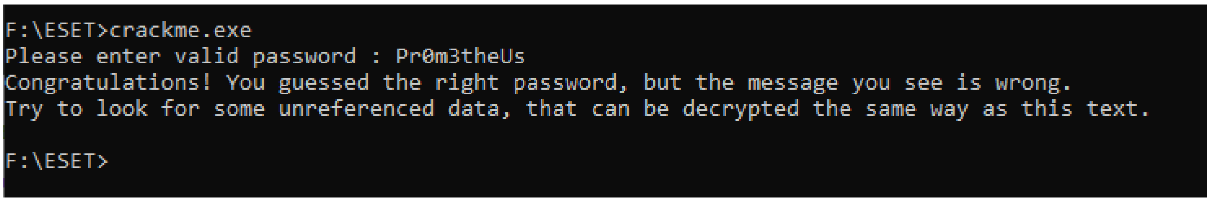


Figure 8: Correct password message

Searching for data that can be decrypted using the same method, opening debugger and set breakpoint at the function that is responsible decrypting the buffer, we see there is some data after the buffer that is not touched, changing the buffer that is being decrypted to point to that data, we see a URL to download the second stage (figure 9).

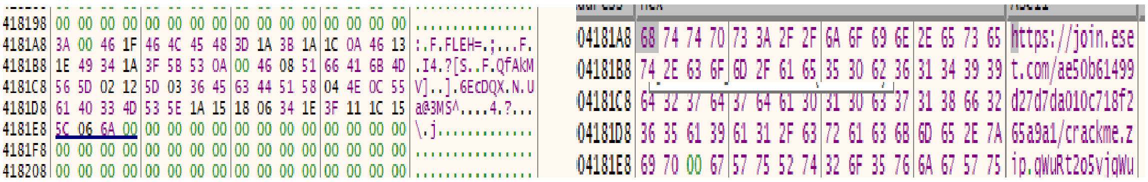


Figure 9: Before and After decrypting the URL

Part 2
Second Stage

Part 2: Second Stage

2.1 EsetCrackme2015

The Second stage is made of Executable and DLL, running the Executable a dialog box appears and request for 3 passwords (figure 10).

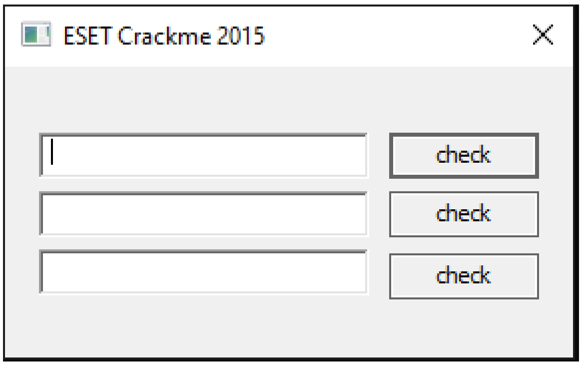


Figure 10: Dialog Box

Opening process hacker to see the process tree, it creates another process and there is RWX memory present which is indication of injection (figure 11,12).

Figure 11:Process Tree

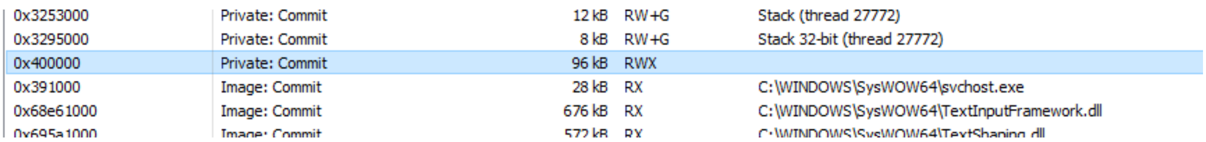


Figure 12: Allocated RWX memory

Open "EsetCrackme2015.exe" in IDA we see that it creates Mutex with the name "EsetCrackme2015" and exit if the mutex is already create, then it loads "EsetCrackme2015.dll" and return (figure 13)

```
HMODULE __stdcall start(int a1, int a2, int a3, int a4)
{
  HMODULE result; // eax
  unsigned int v5; // kr00_4
  char v6; // [esp+0h] [ebp-106h]
  char v7; // [esp+1h] [ebp-105h]
  CHAR EsetCrackme2015[260]; // [esp+2h] [ebp-104h]

  CreateMutexA(0, 1, Name);
  if ( GetLastError() == ERROR_ALREADY_EXISTS )
    return (HMODULE)MessageBoxA(0, aApplicationAlr, aError, 0x30u);
  GetModuleFileNameA(0, EsetCrackme2015, 0x104u);
  v5 = strlen(EsetCrackme2015);
  *(&v7 + v5) = 108;
  *(_WORD *)(&v6 + v5 - 1) = 27748;
  result = LoadLibraryA(EsetCrackme2015);
  dword_40102C = (int)result;
  if ( !result )
    result = (HMODULE)MessageBoxA(0, EsetCrackme2015, Caption, 0x30u);
  return result;
}
```

Figure 13: Main of EsetCrackme2015.exe

Moving our attention to the DLL, it has no export functions and the DllMain is getting the main module base address from LDR_DATA_TABLE structures inside PEB Struct (Figure 14) and resolve function inside the main module and call it (figure 15)

```
:10000231 64 A1 30 00 00 00       mov     eax, large fs:30h
:10000237 8B 40 0C                mov     eax, [eax+0Ch]
:1000023A 8B 40 14                mov     eax, [eax+14h]
:1000023D 56                      push    esi
:1000023E 8B F0                   mov     esi, eax
:10000240 85 C0                   test    eax, eax
:10000242 0F 84 9A 00 00 00       jz      loc_100002E2
```

Figure 14: Get main module address

```
text:100002DA
text:100002DA                   loc_100002DA:
text:100002DA 8D 44 01 0C          lea     eax, [ecx+eax+0Ch]
text:100002DE FF D0                call    eax
```

Figure 15:Call Function inside main module

Opening "EsetCrackme2015.exe" inside x32 debugger and we add breakpoint on LoadLibraryA then tell the debugger to stop at the DLLEntry, then add break point at the call instruction inside the DLL to see which function is being called, so the function which is called is at offset 1E9F, so going to that offset in IDA we see that it resolves Kernel32.dll address from PEB structure then it get the address of the function Sleep and CreateRemoteThreadEx (there is anti-Hooking and anti-breakpoint, when it resolve API address it skips the first instruction which is "mov edi,edi", this technique is used a lot during the challenge) then it Create thread and pass the address of Sleep as argument (Figure 16).

```
401E9F                 push    edi             ; Real EntryPoint
401EA0                 xor     edi, edi
401EA2                 cmp     Flag, edi       ; compare the flag so that it does not run twice
401EA8                 jz      short loc_401ED9
401EAA                 push    esi
401EAB                 call    GetKernel32BaseAddress
401EB0                 mov     esi, eax
401EB2                 push    2FA62CA8h       ; Sleep Hash, Sleep Address is Argument to Created Thread
401EB7                 call    GetProcAddress
401EBC                 push    edi
401EBD                 push    edi
401EBE                 push    eax
401EBF                 push    offset StartThreadAddress
401EC4                 push    edi
401EC5                 push    edi
401EC6                 push    60AC7E39h
401ECB                 mov     Flag, edi
401ED1                 call    GetProcAddress
401ED6                 call    eax             ; CreateRemoteThreadEx
401ED8                 pop     esi
401ED9
401ED9 loc_401ED9:                             ; CODE XREF: .text:00401EA8↑j
401ED9                 pop     edi
401EDA                 retn
```

Figure 16: Real Entry Point of the Executable

During the life of the "EsetCrackme2015.exe" process it uses important Data structure that will be referenced a lot during the execution here are some field that I was able to recover:

+0h	EsetCrackme2015.dll Base Address
+4h	Size of EsetCrackme2015.dll
+108h	if 0 the Pipe server is working, else exit the pipe and process
+109h	Flag to Choose operation in Decrypted Dll
+10Bh	Command Executed //help in serializing the operation
+10Dh	Event Handle //Sync Execution between threads
+111h	GetProcAddress from Hash

+115h Decrypt Buffer Function address

+119h Decrypt DLL Function address

+11Dh Decrypted Buffer (probably used to decrypt the DLL)

+121h Pipe Handle

+125h Thread Handle

+129h base value for hash of API

The real Executable entry point is just wrapper that calls another function this function Create Event that sync the execution between threads (in case the user exits the dialog box to terminate or entered the correct password to drop next part of challenge), then it creates a thread that will be working as PIPE Server (handle command received from the user mode and kernel mode clients) we will explain this thread details later (figure 17).

```
40                  inc     eax
68 E2 DD D2 F9      push    0F9D2DDE2h ; hModule
66 89 45 D3         mov     [ebp+var_2D], ax
E8 C5 FA FF FF      call    GetProcAddress_ ; CreateEvent
FF D0               call    eax
8B 0D 00 24 40 00   mov     ecx, ImportantStruct
53                  push    ebx
53                  push    ebx
53                  push    ebx
68 13 1F 40 00      push    offset Thread_Start_Addresss_Pipe_Server
53                  push    ebx
53                  push    ebx         ; lpProcName
68 39 7E AC 60      push    60AC7E39h ; hModule
89 81 0D 01 00 00   mov     [ecx+10Dh], eax
C7 81 19 01 00 00+  mov     dword ptr [ecx+119h], offset DecryptDLL
11 1B 40 00
E8 99 FA FF FF      call    GetProcAddress_
FF D0               call    eax
8B 35 00 24 40 00   mov     esi, ImportantStruct
68 01 01 00 00      push    101h        ; Buffer Signiture
89 86 25 01 00 00   mov     [esi+125h], eax
```

Figure 17: Create Event and Thread

Then it searches the Dll address Space for data (encrypted String and DLL) (figure 18,19), the data the executable is searching for is a structure that has the following definition.

Struct DataPattern{

BYTE Signature [2]; //this is the Signature that is being searched for

DWORD DataSize;

BYTE Data [DataSize];

}

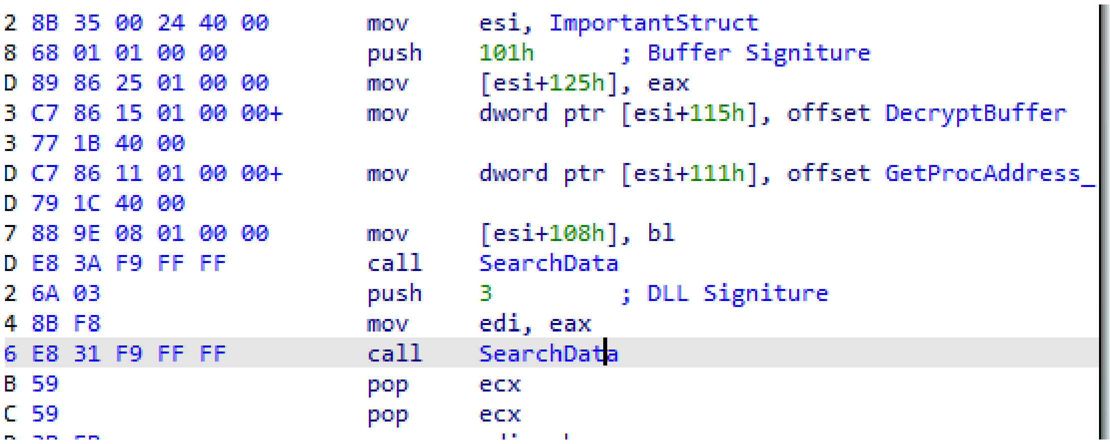


Figure 18: Search for Data and DLL Start address

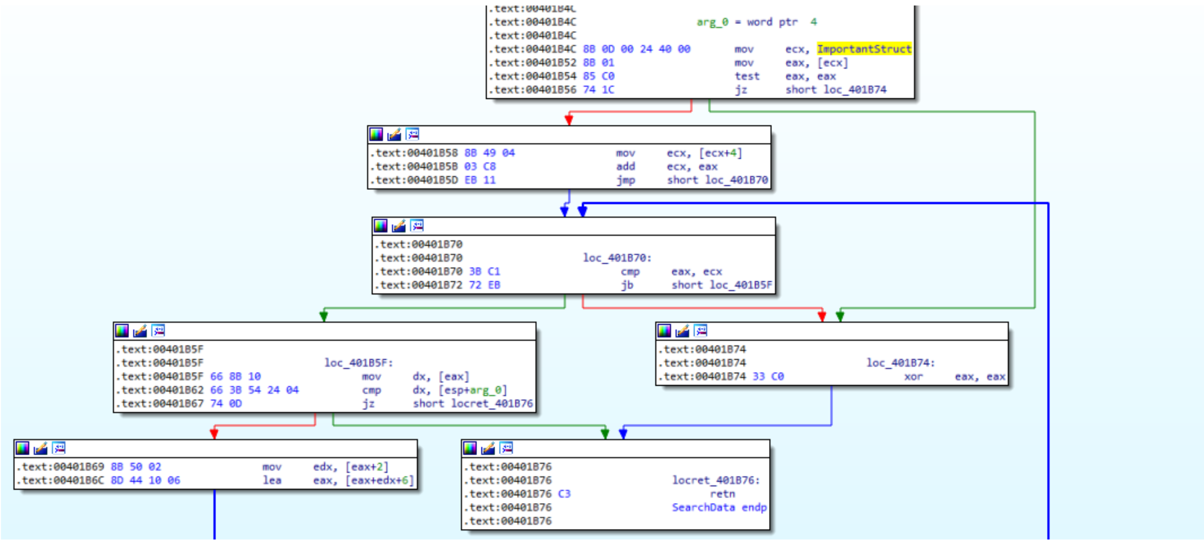


Figure 19: Search Data Function

Then it decrypts the string and Dll, dumping the Dll after the decryption (this Dll is responsible for injecting svchost.exe, dropping the driver and third stage challenge), then it calls the Decrypted DLL entry point to decrypt the PE file to be injected and to Start Svchost.exe and inject it (figure 20).

```
.text:745F0DDC BE 7C 07 5F 74          mov     esi, offset aSvchostExe ; "\\svchost.exe"
.text:745F0DE1 A5                      movsd
.text:745F0DE2 A5                      movsd
.text:745F0DE3 A5                      movsd
.text:745F0DE4 8D 85 FC FE FF FF       lea     eax, [ebp+var_104]
.text:745F0DEA 50                      push    eax
.text:745F0DEB A4                      movsb
.text:745F0DEC E8 FA FD FF FF          call    InjectProcess
.text:745F0DF1 8B 35 F4 0E 5F 74       mov     esi, ImportantStruct
.text:745F0DF7 59                      pop     ecx
.text:745F0DF8 6A 02                   push    2
.text:745F0DFA 58                      pop     eax
.text:745F0DFB 66 89 86 09 01 00+      mov     [esi+109h], ax
.text:745F0DFB 00
```

Figure 20:Inject Svchost.exe

we know the result will be injection, so instead of continue analysis this path I just add breakpoint at the APIs that is used for injection (CreateRemoteThread, SetThreadContext, QueueUserApc,…), and I let the execution continue (the break point is at the middle of the API because of anti-breakpoint), the API that gets executed is SetThreadontext (this is process injection using thread hijacking) so we attach windbg in noninvasive mode which allow us to benefit from the power of windbg without the need to detach x32dbg (figure 21)

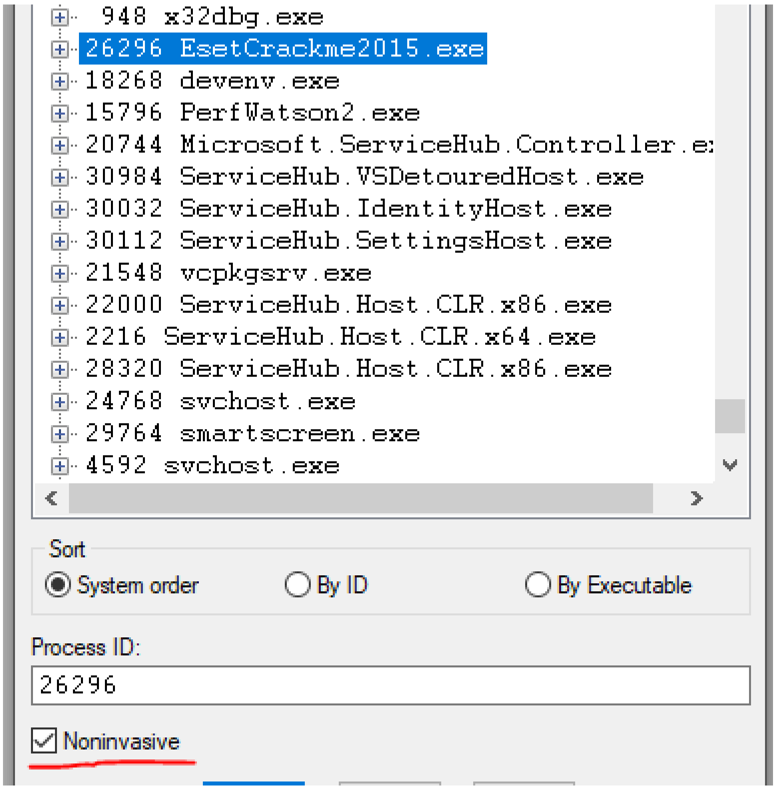


Figure 21: Attach windbg

After attaching windbg we can see the context struct that is being used (figure 22), the entry point will be in eax register (rcx in case 64-bit application)

```
0:000> dt ntdll!_CONTEXT 000c0000
   +0x000 ContextFlags     : 0x10007
   +0x004 Dr0              : 0
   +0x008 Dr1              : 0
   +0x00c Dr2              : 0
   +0x010 Dr3              : 0
   +0x014 Dr6              : 0
   +0x018 Dr7              : 0
   +0x01c FloatSave        : _FLOATING_SAVE_ARE,
   +0x08c SegGs            : 0x2b
   +0x090 SegFs            : 0x53
   +0x094 SegEs            : 0x2b
   +0x098 SegDs            : 0x2b
   +0x09c Edi              : 0
   +0x0a0 Esi              : 0
   +0x0a4 Ebx              : 0x2975000
   +0x0a8 Edx              : 0
   +0x0ac Ecx              : 0
   +0x0b0 Eax              : 0x403db3
   +0x0b4 Ebp              : 0
   +0x0b8 Eip              : 0x77174f70
   +0x0bc SegCs            : 0x23
   +0x0c0 EFlags           : 0x202
   +0x0c4 Esp              : 0x2abfc68
   +0x0c8 SegSs            : 0x2b
   +0x0cc ExtendedRegisters : [512]  ""
```

Figure 22: Context Struct

Adding break point on ResumeThread API then attach debugger to the injected Svchost.exe and add break point in 0x403db0 address to give us the chance to dump the memory before it starts execution.

Before we start analyzing the dumped executable, we will look at the thread that is created early before the injection, this thread first search for Encrypted PIPE name (Signature 0x0002) as it did with DLLs then decrypt the name (\\.\pipe\EsetCrackmePipe) after that it create named Pipe and wait for any client to connect (figure 23,24)

```
push    ebp
mov     ebp, esp
sub     esp, 14h
push    2
call    SearchData ; Get Encrypted pipe name
pop     ecx
test    eax, eax
jnz     short loc_401F2D
```

Figure 23: Get Encrypted pipe name

```
401F82 51                push    ecx
401F83 68 FF 00 00 00    push    0FFh
401F88 52                push    edx
401F89 6A 03             push    3
401F8B FF 75 F8          push    [ebp+PipeName] ; lpProcName
401F8E 2B F3             sub     esi, ebx
401F90 68 01 C4 15 A2    push    0A215C401h ; hModule
401F95 33 F7             xor     esi, edi
401F97 E8 67 FC FF FF    call    GetProcAddress
401F9C FF D0             call    eax         ; CreateNamedPipe
401F9E 8B 0D 00 24 40 00 mov     ecx, ImportantStruct
401FA4 8B B1 29 01 00 00 mov     esi, [ecx+129h]
401FAA 6A 00             push    0           ; lpProcName
401FAC 50                push    eax         ; lpProcName
401FAD 2B F3             sub     esi, ebx
401FAF 68 E6 D3 D5 58    push    58D5D3E6h ; hModule
401FB4 33 F7             xor     esi, edi
401FB6 89 81 21 01 00 00 mov     [ecx+121h], eax
401FBC E8 42 FC FF FF    call    GetProcAddress
401FC1 FF D0             call    eax         ; ConnectNamedPipe
401FC3 85 C0             test    eax, eax
401FC5 75 27             jnz     short loc_401FEE
```

Figure 24:Create and Wait named pipe

After a client connect it reads one byte (Command) then it reads two bytes (signature) (figure 25), then it writes those data back to the client before handling it, the communication cycle between the server and client is shown in (figure 26).

```
45 F0          lea     eax, [ebp+PipeData_2]
               push    eax
01             push    1          ; Number of Data to read
99 FD FF FF    call    ReadDataFromPipe
45 F4          lea     eax, [ebp+PipeData_1]
               push    eax
02             push    2          ; Number of Data to read
8E FD FF FF    call    ReadDataFromPipe
75 F4          push    [ebp+PipeData_1]
75 F0          push    [ebp+PipeData_2]
08 FE FF FF    call    HandleCommandFromClient
00 24 40 00    mov     eax, ImportantStruct
```

Figure 25:Read Data from Client

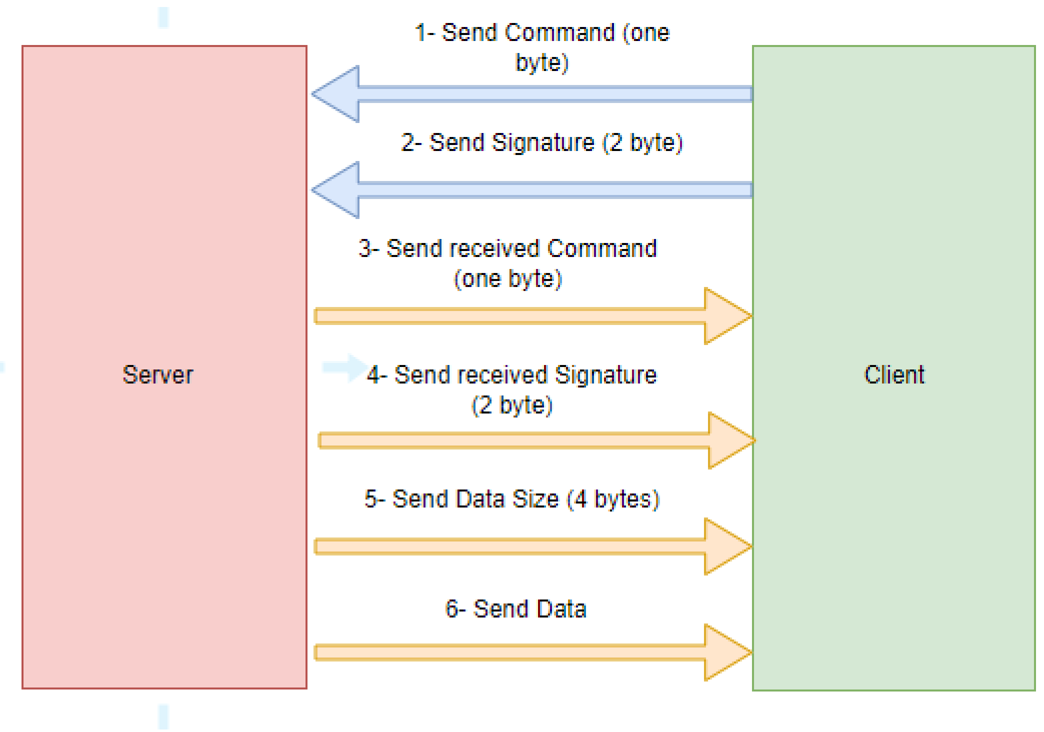


Figure 26: Server/client communication cycle

The command that is used in this client/server communication are:

1. Value "0x1" will use the two bytes that it receives as signature to search for data and send it to client.
2. Value "0x2" will use the two bytes that it receives as second command (exit pipe, drop files), this happens by moving the two bytes that are received (Second Command) to offset +10Bh in the important struct then Signal the event to break out of wait state, after that the main thread will call the decrypted DLL entry point to do the appropriate operation.

The below table shows the valid signatures and commands (note: all the data in encrypted).

Command	Signature	Description	Requestor
X	0x2	Pipe Name	EsetCrackme2015.exe
X	0x3	Dll Signature (self-injection)	EsetCrackme2015.exe
X	0x101	Buffer	EsetCrackme2015.exe
X	0x102		Self-injected Dll
X	0x103		Self-injected Dll
X	0x104		Self-injected Dll

1	0xAA02	Get registry key Name	Driver
1	0xAA06	Data for Virtual Machine	Driver
2	0xAA10	Drop the Dotnet component	Driver
1	0xBB01	Get Passwords Hashes	PE injected in Svchost.exe
1	0xBB02	Get encoded first password	PE injected in Svchost.exe
2	0xBB01	Drop the drv.zip file	PE injected in Svchost.exe
2	0xFFFF	Exit the pipe sever	PE injected in Svchost.exe
2	0xBB02	Drop the drv.zip file	PE injected in Svchost.exe
2	0xBB03	Drop the drv.zip file	PE injected in Svchost.exe
1	0xFF02	Image MD5	PuncherMachine
1	0xFF04	DLL	PuncherMachine
1	0xFF00	Array of Hashes	PuncherMachine
1	0xFF05	DLL	PunchCardReader

2.2 Injected PE File

Now let's move our attention the injected PE file inside Svchost.exe which is PIPE user client.

It first tries to validate that LoadLibraryA and a custom implementation of GetProcAddress is not hooked or has software breakpoint on it and if there a breakpoint it decrements the pointer by two to point to "int 0x3" which will make the application to crash later during execution (it is separating the detection from the action to make it hard to detect where the detection happens) (figure 26).

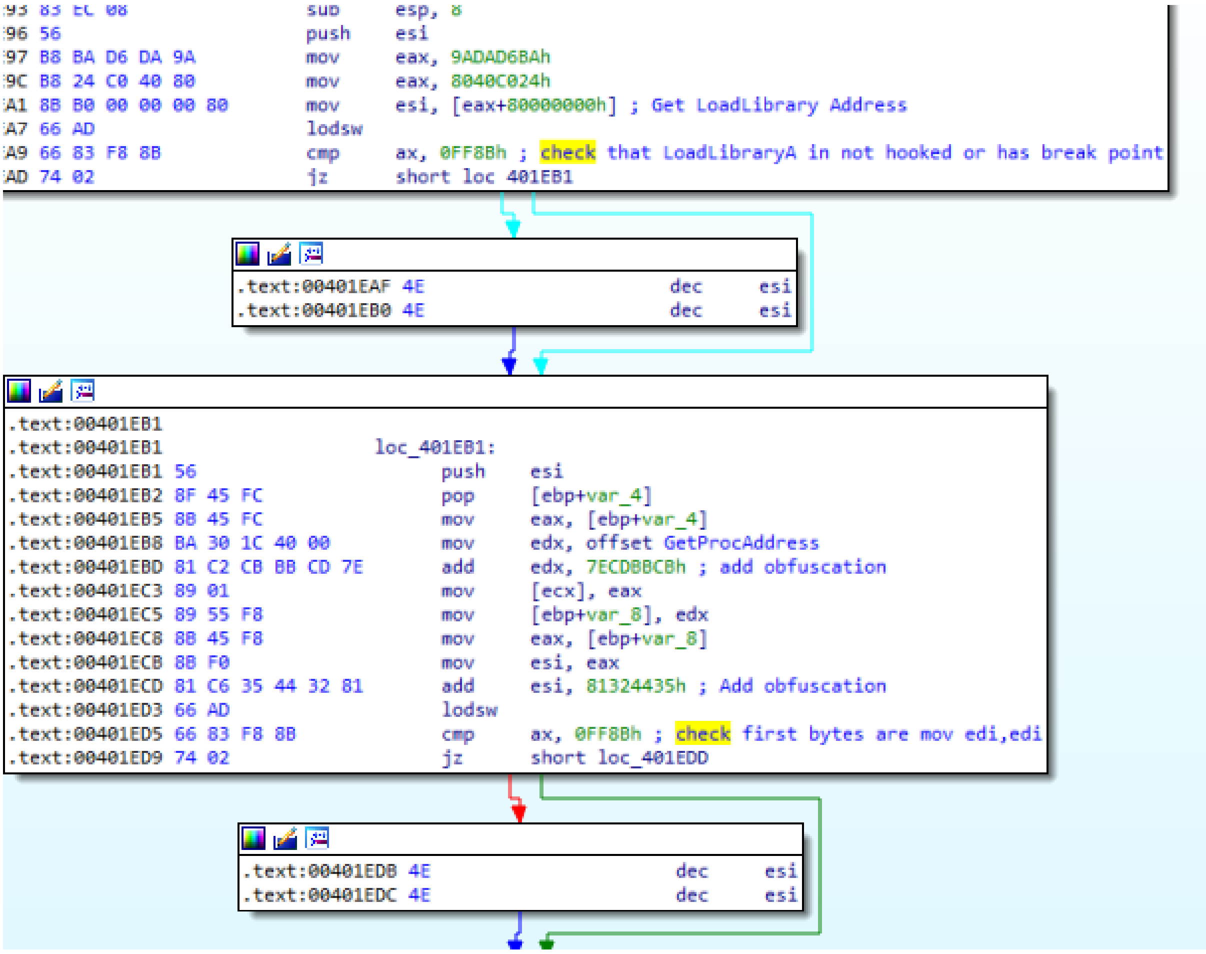


Figure 27: Anti-Debugging Technique

this executable too uses a structure that saves function pointer and data that it uses during execution.

+0h	loadlibraryA
+4h	Getprocaddress
+dh	Handle to previous window procedure to call it
+110h	callWindwProcAddressA
+114h	strcmp

Then it creates a thread that base64 decode some Strings to use (EDIT, user32.dll, Kernel32.dll), resolve some function address then it change the windows procedure handler (figure 28,29)

```
3 5C E2 40 00          mov     eax, 40E25Ch
3 DF 18 00 00          call    Base64Decode ; get Kernel32.dll name
3 02                   jmp     short loc_4022D5
```

Figure 28:Base64 Decode

```
<t:00402294 50                  push    eax
<t:00402295 6A 00               push    0
<t:00402297 51                  push    ecx
<t:00402298 FF D3               call    ebx         ; Get EDIT Windows
<t:0040229A 68 60 23 40 00      push    offset New_Window_Handler
<t:0040229F 6A FC               push    0FFFFFFFCh
<t:004022A1 50                  push    eax
<t:004022A2 89 3D E4 2E 41 00   mov     ds:412EE4h, edi
<t:004022A8 FF D6               call    esi         ; Change the handle function
<t:004022AA 89 47 0C            mov     [edi+0Ch], eax
```

Figure 29:change window handler

Then it tries to adjust privilege to "DebugPrivilige", after that it reads some data from the PIPE Server with the command "0x1" and signature "0xBB01" (Encrypted Passwords hashes), "0xBB02" (Encrypted first password).

And the received data is Xor decrypted with the string "PIPE" (figure 30)

Password Hashes:

- 0B6A1C6651D1EB5BD21DF5921261697AA1593B7E
- 0F30181CF3A9857360A313DB95D5A169BED7CC37
- 869B39E9F2DB16F2A771A3A38FF656E050BB1882

Encoded First Password:

- RFV1aV4fQ1FydFxk

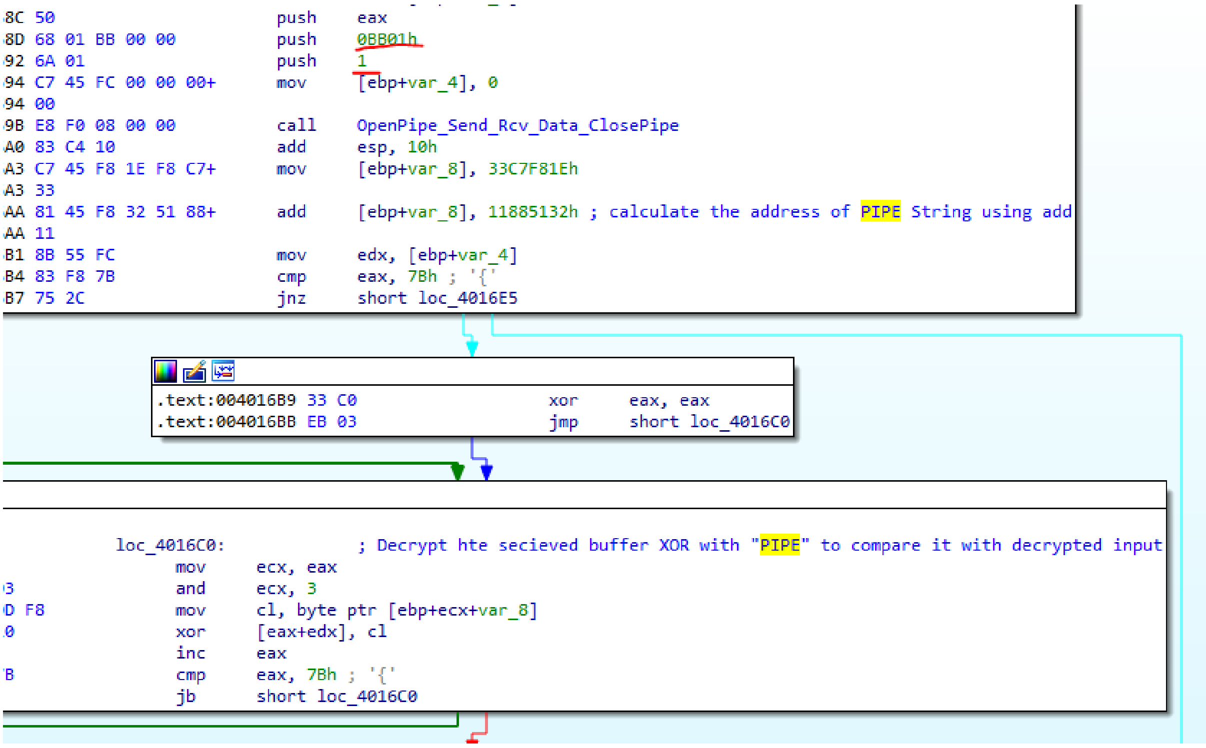


Figure 30:Xor Decrypt received Data

Now we look at the new window handler and see what it does.

It checks if the received message is WM_GETTEXT and if not return, then it Base64Encode the received message then Subtract “0x1” from the odd index character of the encoded data, after that it compares it with “RFV1aV4fQ1FydFxk” (figure 31)

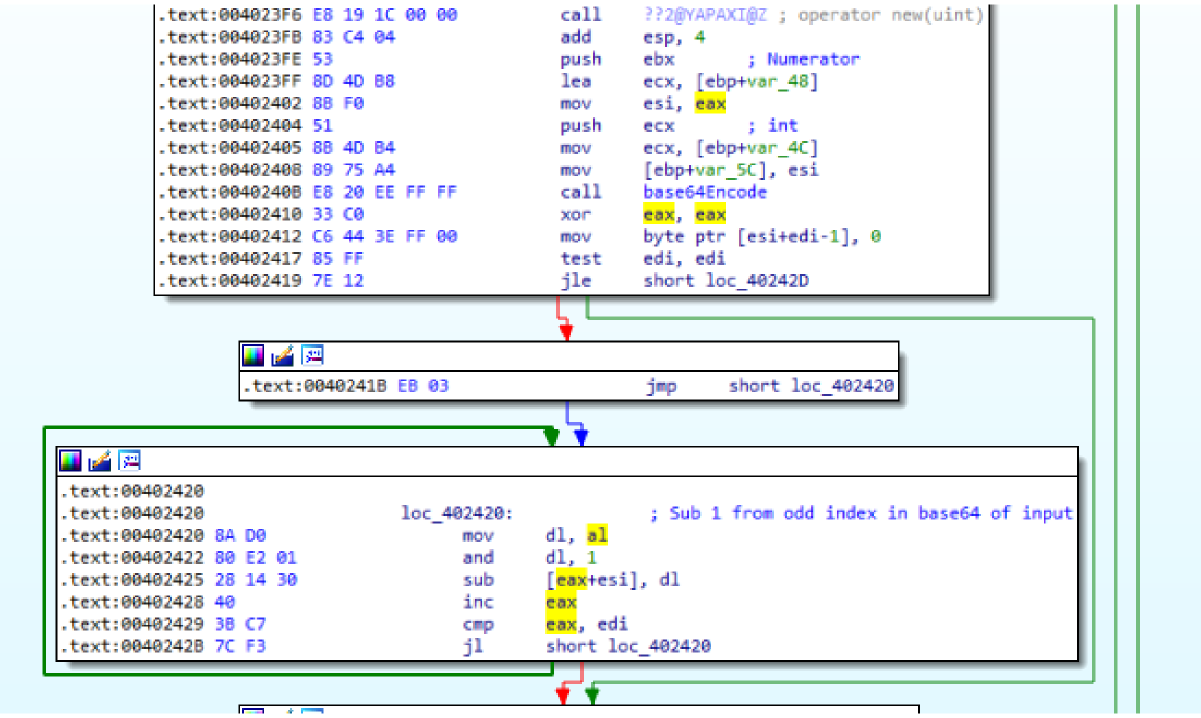


Figure 31: Encode Message

Doing the opposite operation to the string "RFV1aV4fQ1FydFxk" we get the string "Devin Castle", and it is the correct password, then the pipe user client (PE injected inside Svchost.exe) send message through the pipe with command "0x2" and signature "0xBB01" this will drop the drv.zip file (figure 32).

```
:745F0E10 6A 01              push    1
:745F0E12 68 8C 07 5F 74     push    offset aDrvZip ; "drv.zip"
:745F0E17 B8 52 01 00 00     mov     eax, 152h
:745F0E1C E8 6E FD FF FF     call    DropFile
:745F0E21 8B 35 F4 0E 5F 74  mov     esi, ImportantStruct
:745F0E27 59                 pop     ecx
:745F0E28 59                 pop     ecx
:745F0E29 6A 03              push    3
:745F0E2B 58                 pop     eax
:745F0E2C 66 89 86 09 01 00+ mov     [esi+109h], ax
:745F0E2C 00
```

Figure 32: Drop Driver

Part 3
Driver

Part 3: Driver

When we unzip the driver, we see that it's 32-bit legacy driver, and the INSTALLME file teel us to install this driver on windows 7.

3.1 Driver Entry

Open the crackme_drv.sys in IDA to see what major Dispatcher it registers (figure 33).

```
DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)Create_Close;
DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)Create_Close;
DriverObject->MajorFunction[3] = (PDRIVER_DISPATCH)Read_Write;
DriverObject->MajorFunction[4] = (PDRIVER_DISPATCH)Read_Write;
DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)DeviceControl;
DriverObject->MajorFunction[27] = (PDRIVER_DISPATCH)PNP;
DriverObject->MajorFunction[22] = (PDRIVER_DISPATCH)Power;
DriverObject->MajorFunction[23] = (PDRIVER_DISPATCH)SystemControl;
DriverObject->DriverExtension->AddDevice = AddDeviceFunction;
DriverObject->DriverUnload = (PDRIVER_UNLOAD)Unload;
return 0;
```

Figure 33: Driver Registered Functions

The two function that is of interest is the Read_Write Major Dispatcher (called when a user mode application Read or write to the device that the driver created), and AddDeviceFunction (this function is called when the Plug-and-Play manager detect a new device is attached), we will start with AddDeviceFunction function.

3.2 AddDeviceFunction

It will Creates Device "\\Device\\45736574" then create virtual hard disk then attach to PNP device (figure 34,35)

```
2
2                         loc_401182:
2 68 28 41 40 00                  push    offset SourceString ; "\\Device\\45736574"
7 8D 55 E0                        lea     edx, [ebp+DestinationString]
A 52                              push    edx             ; DestinationString
B FF 15 08 40 40 00               call    ds:RtlInitUnicodeString
1 8D 45 F8                        lea     eax, [ebp+DeviceObject]
4 50                              push    eax             ; DeviceObject
5 6A 00                           push    0               ; Exclusive
7 68 00 01 00 00                  push    FILE_DEVICE_SECURE_OPEN ; DeviceCharacteristics
C 6A 07                           push    FILE_DEVICE_DISK ; DeviceType
E 8D 4D E0                        lea     ecx, [ebp+DestinationString]
1 51                              push    ecx             ; DeviceName
2 68 98 00 00 00                  push    98h ; '˜'       ; DeviceExtensionSize
7 8B 55 08                        mov     edx, [ebp+DriverObject]
A 52                              push    edx             ; DriverObject
B FF 15 30 40 40 00               call    ds:IoCreateDevice
1 89 45 F4                        mov     [ebp+Status], eax
4 83 7D F4 00                     cmp     [ebp+Status], 0
8 7D 08                           jge     short loc_4011C2
```

Figure 34: Create Device

```
8B 4D F8                 mov     ecx, [ebp+DeviceObject]
51                       push    ecx
E8 20 6E 00 00           call    VirtualHardDisk
8B 55 FC                 mov     edx, [ebp+DeviceExtension]
8B 42 28                 mov     eax, [edx+28h]
83 C8 01                 or      eax, 1
8B 4D FC                 mov     ecx, [ebp+DeviceExtension]
89 41 28                 mov     [ecx+28h], eax
8B 55 0C                 mov     edx, [ebp+TargetDevice]
52                       push    edx             ; TargetDevice
8B 45 F8                 mov     eax, [ebp+DeviceObject]
50                       push    eax             ; SourceDevice
FF 15 24 40 40 00        call    ds:IoAttachDeviceToDeviceStack
8B 4D FC                 mov     ecx, [ebp+DeviceExtension]
89 41 04                 mov     [ecx+4], eax
8B 55 FC                 mov     edx, [ebp+DeviceExtension]
83 7A 04 00              cmp     dword ptr [edx+4], 0
```

Figure 35: Attach to Device

Then it creates some threads that will do the rest of the work.

- Thread 1:

Communicate with pipe server to get some data (Registry Key) using command "0x1" and signature "0xAA02" then decrypt the data using RC4 (the decrypted registry key is "ESETConst").

- Thread 2:

Communicate with pipe server with command "0x2" and signature "0xAA10" this will drop the "PunchCardReader.exe" and "PuncherMachine.exe" for the next stage and write the file "PunchCard.bmp" to the virtual hard disk (we will come back to this operation in Read_Write Dispatcher).

- Thread 3:

Will check the presence of the registry key "\\EsetCrackme" under the driver install registry key, then read the REG_SZ value with name "ESETConst", then will copy the key value to global buffer (will be used latter in the virtual machine) (figure 36)

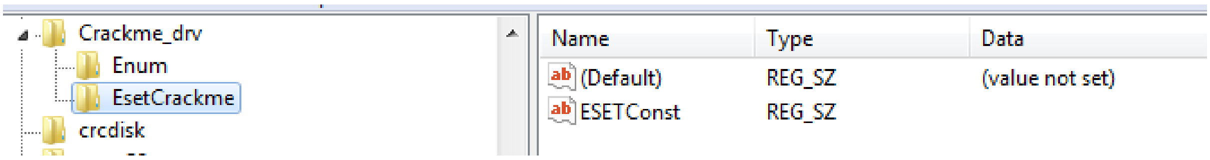


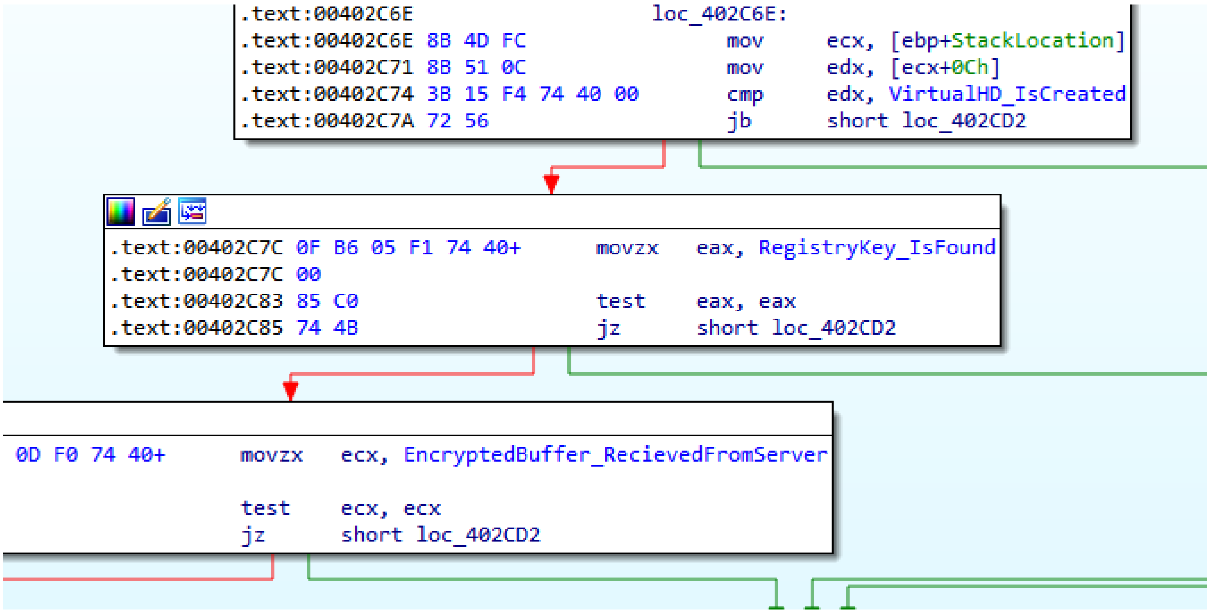
Figure 36: Registry Key

- Thread 4:

Will decrypt shellcode (Virtual machine) and a buffer that will be passed to the shellcode as argument using RC4, then it will communicate with pipe server with command "0x1" and signature "0xAA06" to get some RC4 encrypted data and copy that data (after decryption) to memory location after the decrypted buffer that will passed as argument to shellcode.

3.3 Read/Write Dispatcher

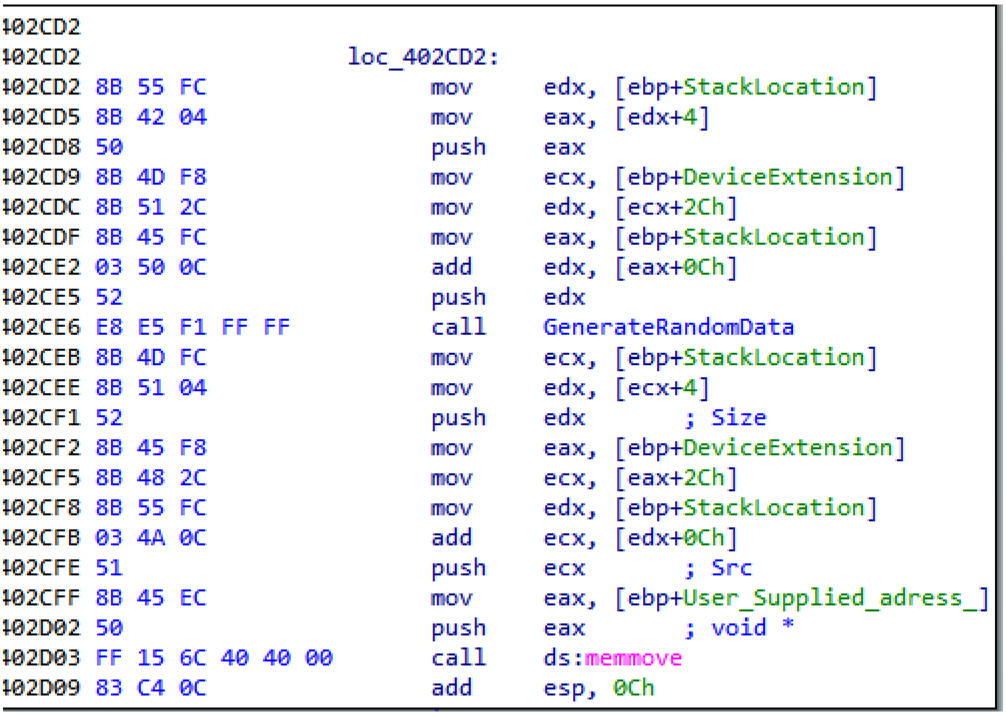
Back to Read_Write Dispatcher, if a write request is being issued to the virtual hard disk the operation will pass normally, but if a read operation it will check that the virtual hard disk is created and the "ESETConst" registry key is found, and the encrypted data is received from the server (figure 37)



```
.text:00402C6E                          loc_402C6E:
.text:00402C6E 8B 4D FC                         mov     ecx, [ebp+StackLocation]
.text:00402C71 8B 51 0C                         mov     edx, [ecx+0Ch]
.text:00402C74 3B 15 F4 74 40 00                cmp     edx, VirtualHD_IsCreated
.text:00402C7A 72 56                            jb      short loc_402CD2

.text:00402C7C 0F B6 05 F1 74 40+       movzx   eax, RegistryKey_IsFound
.text:00402C7C 00
.text:00402C83 85 C0                    test    eax, eax
.text:00402C85 74 4B                    jz      short loc_402CD2

0D F0 74 40+       movzx   ecx, EncryptedBuffer_RecievedFromServer

                   test    ecx, ecx
                   jz      short loc_402CD2
```

Figure 37: check that all threads run normally

If any of the checks fails it will generate random data and return it (figure 38).



```
402CD2
402CD2                     loc_402CD2:
402CD2 8B 55 FC                    mov     edx, [ebp+StackLocation]
402CD5 8B 42 04                    mov     eax, [edx+4]
402CD8 50                          push    eax
402CD9 8B 4D F8                    mov     ecx, [ebp+DeviceExtension]
402CDC 8B 51 2C                    mov     edx, [ecx+2Ch]
402CDF 8B 45 FC                    mov     eax, [ebp+StackLocation]
402CE2 03 50 0C                    add     edx, [eax+0Ch]
402CE5 52                          push    edx
402CE6 E8 E5 F1 FF FF              call    GenerateRandomData
402CEB 8B 4D FC                    mov     ecx, [ebp+StackLocation]
402CEE 8B 51 04                    mov     edx, [ecx+4]
402CF1 52                          push    edx             ; Size
402CF2 8B 45 F8                    mov     eax, [ebp+DeviceExtension]
402CF5 8B 48 2C                    mov     ecx, [eax+2Ch]
402CF8 8B 55 FC                    mov     edx, [ebp+StackLocation]
402CFB 03 4A 0C                    add     ecx, [edx+0Ch]
402CFE 51                          push    ecx             ; Src
402CFF 8B 45 EC                    mov     eax, [ebp+User_Supplied_adress_]
402D02 50                          push    eax             ; void *
402D03 FF 15 6C 40 40 00           call    ds:memmove
402D09 83 C4 0C                    add     esp, 0Ch
```

Figure 38: Generate Random data for requester

And if all the check passed it will execute the shellcode (virtual machine) on the data (0x200 bytes each time) (figure 39)

```
03146 8B 45 E8          mov     eax, [ebp+Function_]
03149 50                push    eax
0314A 8B 4D E4          mov     ecx, [ebp+KernelBase]
0314D 51                push    ecx
0314E 8B 55 F4          mov     edx, [ebp+DeviceExtention]
03151 8B 42 70          mov     eax, [edx+70h] ; pointer to byte_4071D0 data
03154 50                push    eax
03155 8B 4D E0          mov     ecx, [ebp+var_20] ; 200h value
03158 51                push    ecx
03159 8B 55 FC          mov     edx, [ebp+OffsetToData]
0315C 52                push    edx
0315D E8 1E FF FF FF    call    CallShellCode
03162 8B 45 FC          mov     eax, [ebp+OffsetToData]
03165 05 00 02 00 00    add     eax, 200h ; Get Next Chunck of data
0316A 89 45 FC          mov     [ebp+OffsetToData], eax
0316D EB C6             jmp     short loc_403135
```

Figure 39: call the shellcode on the data

So, adding a break point on the call to shellcode to dump it to the disk and open it in IDA, we notice the loop which interpret a bytecode sequence and execute handler based on the first byte (figure 40)

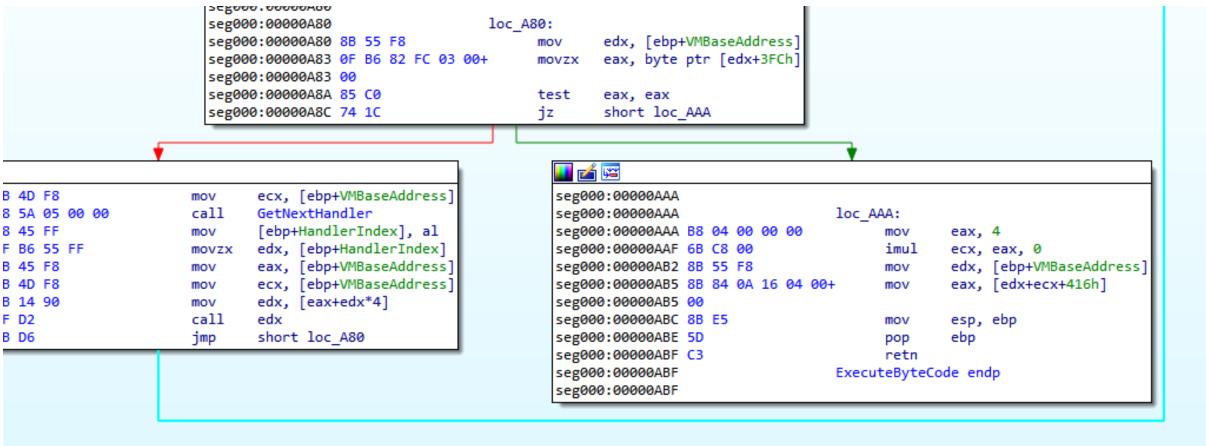


Figure 40: VM Loop

So, we know its virtual machine, so i start reverse engineering each handler, and this what I learned about the VM:

1. It's a variable length instruction
2. The Code Base is found at offset + 456h
3. The IP is at offset + 45Eh
4. The CMP instruction save its compare result at offset + 415h
5. The stack is at offset + 462h
6. The registers (16 register) are at offset +462h
7. Global data is at offset + 45Ah
8. The initial value for registers is at offset + 411h

Opcode value	Description
0x0	Ret
0x1	Mov reg, reg/imm
0x2	Call operand
0x3	Mov reg, [reg]
0x4	Push
0x5	Pop reg
0x6	Cmp (JZ, JNZ, JNB)
0x7	JUMP (conditional/unconditional)
0x8	Call (push IP, change IP with operand)
0x9	POP IP
0xA	Arithmetic (add, sub, xor ,..)
0xB	Malloc(operand)->R0
0xC	Call Kernel API
0xD	Call (New VM Code)
0xE	Ret

So, with the knowledge I have now on the VM I was able to write a disassembler for it (it is not very accurate, but with the help of debugger I was able to understand the logic of the VM).

it first checks the registry key that it read, and if it is zero it will jump to 8A instruction, but if the registry key is null the shellcode itself will not get executed

```
00000000: CMP     R12 , 0      //R12 has pointer to Register Key Value
00000004: JUMP    Conditional 8a (JZ)
0000000A: CMP     R13 , 0      //R13 has the length of Register Key Value
0000000E: JUMP    Conditional 11e (JZ)
```

```
00000014: MOV    R2    R12
00000017: MOV    R3    DATA_164        //encrypted BYTES
0000001E: MOV    R4    DATA_137        //Decryption Key “ETSE” HardCoded
00000025: MOV    R5    R2
00000028: ADD    R5    R13             //R5 Has reg Key end pointer
0000002B: MOV    R14   R3
0000002E: MOV    R0    DATA_160        //size of DATA_164
00000035: ADD    R14   R0
00000038: MOV    R15   R4
0000003B: ADD    R15   0x4             //R15 Has hardcoded KEY end pointer
0000003F: CMP    R2    R5
00000042: JUMP   Conditional 4b (JNZ)
00000048: MOV    R2    R12
0000004B: CMP    R4    R15
0000004E: JUMP   Conditional 5b (JNZ)
00000054: MOV    R4    DATA_137
0000005B: CMP    R3    R14
0000005E: JUMP   Conditional 100 (JZ)
00000064: MOV    R0    R2
00000067: XOR    R0    R3
0000006A: ADD    R0    0x1
0000006E: ROL    R0    x1
00000072: XOR    R0    R4
00000075: MOV    R3    R0
00000078: ADD    R2    0x1
0000007C: ADD    R3    0x1
00000080: ADD    R4    0x1
00000084: JUMP   3f
```

```
0000008A: MOV    R3     DATA_140    //Encrypted BYTE Data, Second Password
00000091: MOV    R4     DATA_137    //Decryption Key "ETSE"
00000098: MOV    R14    R3
0000009B: MOV    R0     DATA_13c
000000A2: ADD    R14    R0
000000A5: MOV    R15    R4
000000A8: ADD    R15    0x4
000000AC: CMP    R4     R15
000000AF: JUMP   Conditional bc (JNZ)
000000B5: MOV    R4     DATA_137
000000BC: CMP    R3     R14
000000BF: JUMP   Conditional e8 (JZ)
000000C5: MOV    R0     DATA_0
000000C9: XOR    R0     R3
000000CC: ADD    R0     0x1
000000D0: ROL    R0     0x1
000000D4: XOR    R0     R4
000000D7: MOV    R3     DATA_0
000000DA: ADD    R3     0x1
000000DE: ADD    R4     0x1
000000E2: JUMP   ac
000000E8: MOV    R0     DATA_13c
000000EF: MOV    R0     DATA_0
000000F2: PUSH   R0
000000F4: PUSH   DATA_140
000000FA: JUMP   112
```

as we see from the above code that the two loops will decrypt some data (will be user as decryption key for the image), so i decrypt the buffer using the hardcode key "ETSE" (by adding breakpoint at the first compare instruction and change the compare result and force it enter the second loop) we see that the data is "Barbakan Krakowski" which is the second password (figure 41)

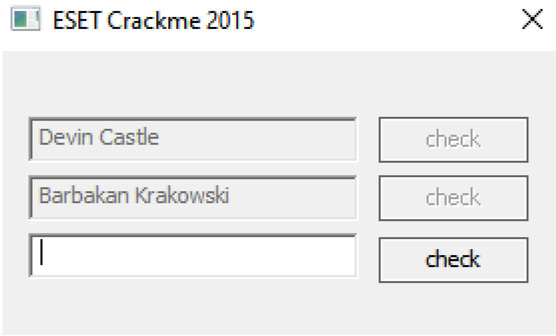


Figure 41: password

So, i continue our analysis to see where the call instruction at 0x116 is going, it is another VM code so disassembling it, we see that it is just decrypting the second stage VM

```
00000000: MOV   R0    R7
00000003: MOV   R1    R0
00000006: ADD      R0    0x1
0000000A: MOV   R2    R0
0000000D: ADD     R0    0x5
00000011: MOV   R3    R0
00000014: MOV   R0    R8
00000017: ADD      R0    R7
0000001A: ADD     R3    R7
0000001D: MOV   R4    R3
00000020: XOR      R4    R1
00000023: ADD      R1    R2
00000026: MOV   R3    R4
00000029: ADD      R3    0x1
0000002D: CMP   R3    R0
00000030: JUMP  Conditional 1d(JNB)
00000036: MOV   R0    R7
00000039: MOV   R0    DATA_37
0000003D: ADD     R0    0x1
00000041: MOV   R0    DATA_13
              : Ret
```

So, I dump the second stage VM and it's RC4 decryption function that decrypt the image data using the decrypted key from the first stage VM

RC4 Table initialize and randomize

```
00000007:    MOV   R1   DATA_0      //loop to initialize the table
0000000B:    MOV   R0   R1
0000000E:    ADD   R0   0x1
00000012:    ADD   R1   0x1
00000016:    CMP   R1   0x100
0000001B:    JUMP  Conditional JNB b
00000021:    MOV   R0   DATA_0
00000025:    MOV   R1   DATA_0
00000029:    MOV   R4   R12         //R12 has the Key (Generated from the first VM code)
0000002C:    MOV   R14  R4
0000002F:    ADD   R14  R13         //R13 has the key size (0x12)
```

Decrypt the image data

```
00000088:  CMP    R3   R4
0000008B:         JUMP   Conditional JZ dc
00000091:         ADD       R0    0x1
00000095:         DIV       R0    0x100
0000009A:         MOV       R2    R5
0000009D:         ADD       R2    R0
000000A0:         ADD       R1    R2
000000A3:         DIV       R1    0x100
000000A8:         MOV       R14   R5
000000AB:         ADD       R14   R1
000000AE:         MOV       R15   R2
000000B1:         MOV       R2    R14
000000B4:         MOV       R14   R15
000000B7:         MOV       R15   DATA_0
000000BB:         ADD       R15   R2
000000BE:         ADD       R15   R14
000000C1:         DIV       R15   0x100
000000C6:         ADD       R15   R5
000000C9:         MOV       R15   R15
000000CC:         XOR       R15   R3
000000CF:         MOV       R3    R15
000000D2:         ADD       R3    0x1
000000D6:         JUMP    88
```

So, now we need to see what the image is, we can do so using one of the following:

1. We can get the correct registry key by Brute force (we know that the result should be "Barbakan Krakowski").
2. Or we can add breakpoint at CreateFileW in the "EsetCrackme2015.exe" and dump the image when it is written to the virtual hard disk then decrypting it using RC4 with Key "Barbakan Krakowski" (note: that the decryption should happen on each 0x200 byte of data)

So doing so we get the image, and we can move to the next stage of the challenge (figure 42)

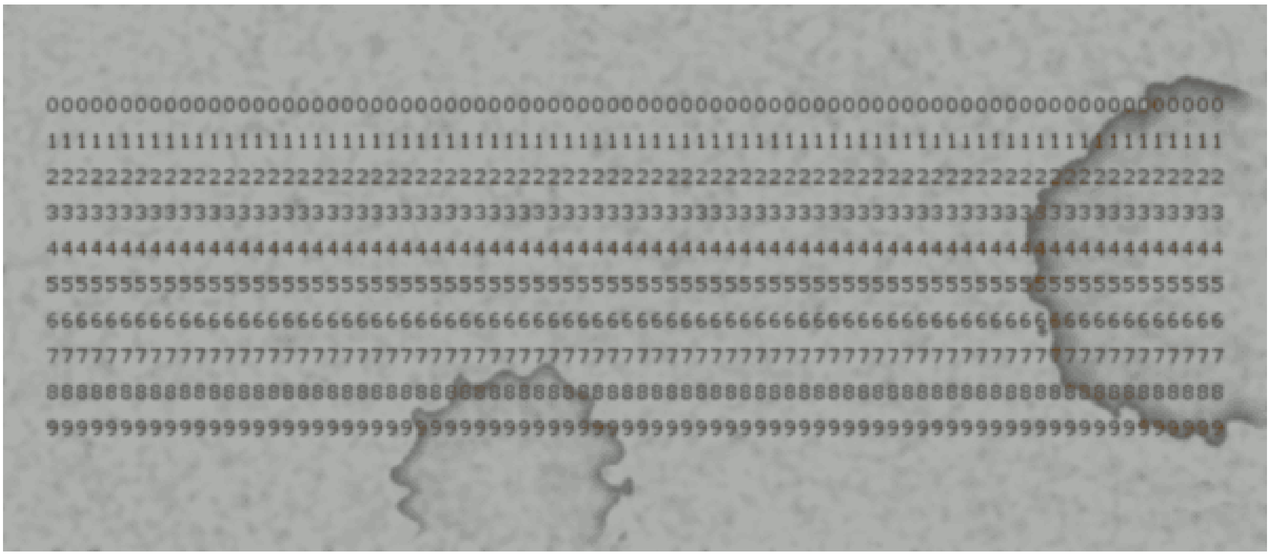


Figure 42: PunchCard image

Part 4
DotNet

Part 4: DotNet

4.1 PuncherMachine

After decrypting the image, we move our attention to the two dropped .Net files, we will start with PuncherMachine.exe.

When we start it requests a bmp image, supplying the PunchCard.bmp it requests that we input two calibration (figure 43)

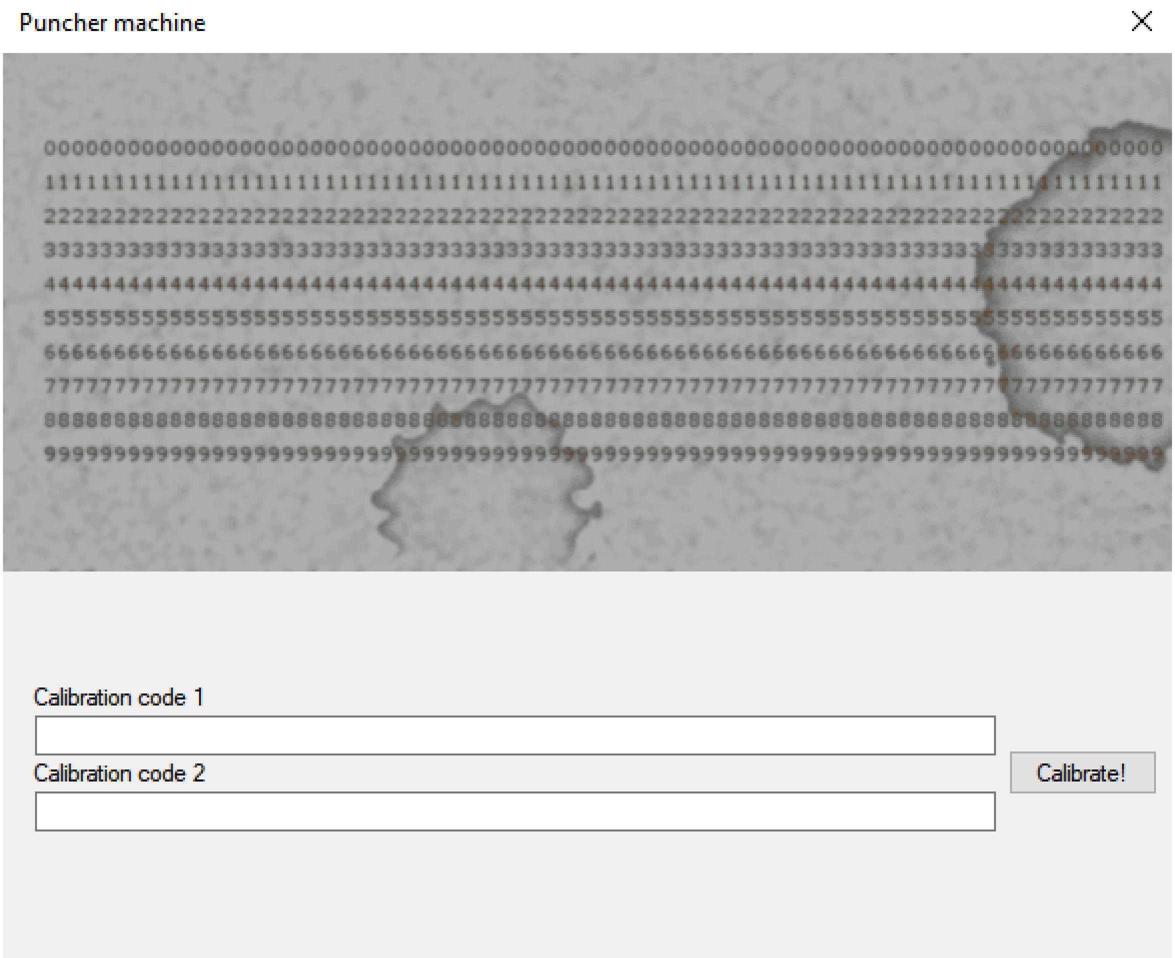


Figure 43: calibration request

Open the file dnspy tool, the file is obfuscated (string and control flow) and there is anti-debugging (figure 44).

```
{
    if (!Debugger.IsAttached)
    {
        for (;;)
        {
            IL_07:
            int num = 1879049379;
            for (;;)
```

```
case 4:
    num = ((!Debugger.IsLogging()) ? 1879049381 : 1879049383);
    continue;
case 5:
{
```

Figure 44: Anti-Debugging

During the initialization it calculate the MD5 hash of some functions (any deobfuscation that change those function will crash the application) cross-reference where this hash is used, it is only used as key in AES decryption (for data received from pipe server) (figure 45,46,47), then it creates mutex with the name "3023912A-E3F8-4026-B6E1-3950992FAFE8" to make sure only one instance is running (figure 48)

```
byte[] hash;
using (MD5 md = MD5.Create())
{
    IEnumerable<Type> types = Assembly.GetExecutingAssembly().GetTypes();
    if (global::A.A.A == null)
    {
        global::A.A.A = new Func<Type, IEnumerable<MethodInfo>>(global::A.A.A);
    }
    IEnumerable<MethodInfo> source = types.SelectMany(global::A.A.A);
    if (global::A.A.A == null)
    {
        global::A.A.A = new Func<MethodInfo, h<MethodInfo, object[]>>(global::A.A.A);
    }
    IEnumerable<h<MethodInfo, object[]>> source2 = source.Select(global::A.A.A);
    if (global::A.A.A == null)
    {
        global::A.A.A = new Func<h<MethodInfo, object[]>, bool>(global::A.A.A);
    }
    IEnumerable<h<MethodInfo, object[]>> source3 = source2.Where(global::A.A.A);
    if (global::A.A.A == null)
    {
        global::A.A.A = new Func<h<MethodInfo, object[]>, I<MethodInfo, D>>(global::A.A.a);
    }
    IEnumerable<I<MethodInfo, D>> enumerable = source3.Select(global::A.A.A);
    IEnumerator<I<MethodInfo, D>> enumerator = enumerable.GetEnumerator();
    try
    {
        for (;;)
        {
            IL_DF:
            int num = enumerator.MoveNext() ? -1278104833 : -1278104834;
            for (;;)
```

Figure 45: Md5 functions

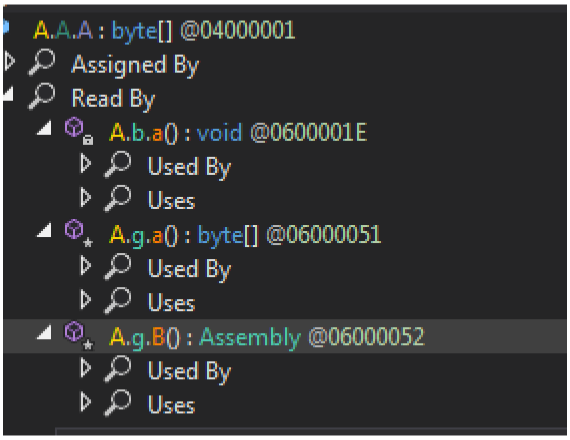


Figure 46: cross reference MD5 hash

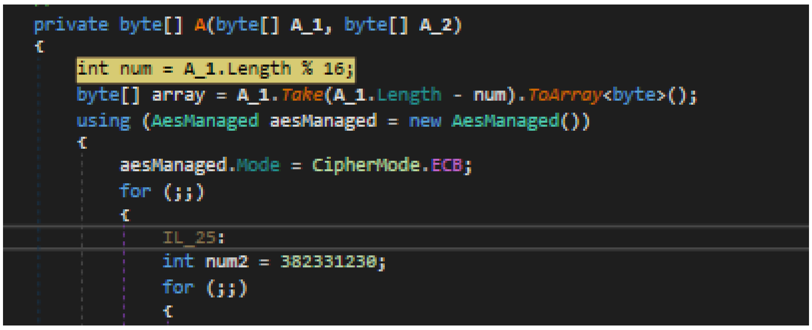


Figure 47: MD5 hash used in AES Decryption

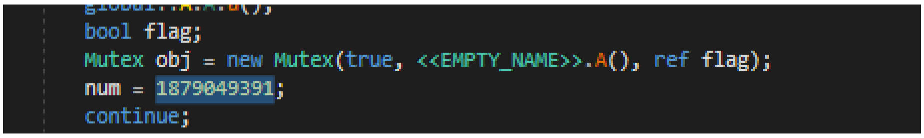


Figure 48: Create mutex

It also communicates with the pipe server to receive some data:

- MD5 hash of the expected PunchCard.bmp
- Dll (used for input hashing)
- Array of hashes to validate input (input validation)

It MD5 the input image and compare with the MD5 it receives from pipe server, and if they don't match it request another image (figure 49, 50)

```
public static byte[] A(string A_0)
{
    FileStream fileStream = File.OpenRead(A_0);
    MD5 md = MD5.Create();
    byte[] result;
    try
    {
        result = md.ComputeHash(fileStream);
    }
    finally
    {
        if (md != null)
        {
            for (;;)
            {
                IL_1A:
                int num = -1692838931;
                for (;;)
                {
                    switch (num ^ -1692838932)
                    {
                    case 1:
                        ((IDisposable)md).Dispose();
                        num = -1692838932;
                        continue;
                    case 2:
                        goto IL_1A;
                    }
                    goto Block_4;
                }
            }
            Block_4:;
        }
    }
    fileStream.Close();
    return result;
}
```

Figure 49: Md5 input image

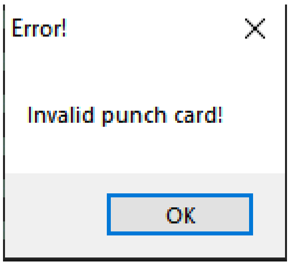


Figure 50" wrong image

After decrypting the received DLL, it loads it to memory (figure 51), cross reference the loaded assembly variable, it is only used in one location (will be executed when calibrate! Button is pressed (figure 52,53)), adding breakpoint at that line and dumping the DLL from memory

```
case 1:
{
    byte[] rawAssembly;
    Assembly assembly = Assembly.Load(rawAssembly);
    result = assembly;
    num = -969281078;
    continue;
}
case 2:
{
    byte[] rawAssembly = e.A(65284, global::A.A.A);
    num = -969281077;
    continue;
}
```

Figure 51: load DLL

```
    case 26:
        this.a = new TextBox();
        this.B = new TextBox();
        num = -1086550243;
        continue;
    case 27:
        this.B.Click += this.B;
        num = -1086550246;
        continue;
    case 28:
        this.A.Location = new Point(840, 8);
        num = -1086550171;
```

Figure 52: function related to calibrate! Button

```
        num = -1086550153;
        continue;
    case 62:
        this.a.Click += this.a;
        this.A.Controls.Add(this.B);
        num = -1086550241;
        continue;
    case 63:
```

Figure 53: Function related to Punch it! Button

When the enter the calibrate data, it takes the first input and divide it to two 8 bytes list (will treat them as hex values), then it connects to the pipe and get the array of hashes from the pipe server, load the received dll and call the createMethod function, this function will take the created list from the first input as argument, then it will create a function on the fly (figure 54).

```
{
    typeof(string)
};
DynamicMethod dynamicMethod = new DynamicMethod("", typeof(ulong), parameterTypes);
ILGenerator ilgenerator = dynamicMethod.GetILGenerator();
ilgenerator.DeclareLocal(typeof(ulong), true);
ilgenerator.DeclareLocal(typeof(int), true);
ilgenerator.DeclareLocal(typeof(ulong), true);
ilgenerator.DeclareLocal(typeof(bool), true);
Label label = ilgenerator.DefineLabel();
Label label2 = ilgenerator.DefineLabel();
Label label3 = ilgenerator.DefineLabel();
IlParticlesEmitor ilParticlesEmitor = new IlParticlesEmitor(ilgenerator);
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Nop, null, "IL_00000"));
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Ldc_I8, 3074457345618258791L, "IL_00010"));
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Stloc_0, null, "IL_000a0"));
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Ldc_I4_0, null, "IL_000b0"));
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Stloc_1, null, "IL_000c0"));
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Br_S, label, "IL_000d0"));
ilParticlesEmitor.addILParticle(new ILEmitParticle(label2, null, "IL_000d9"));
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Nop, null, "IL_000f0"));
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Ldloc_0, null, "IL_00100"));
try
{
    ilParticlesEmitor.addILParticle(new ILEmitParticle(hashtable[instructionHashes[0]], null, "IL_00110"));
```

Figure 54: createMethod function

It creates the on-the-fly method with the help of our first input, as we see it pushes the first 8 bytes of the first input in the middle of two instruction, the get_Chars method take two arguments so the instruction should be a push, the on-the-fly function is called with the second input as argument, so it must be ldarg.0 (0x0364ABE7) as it is not referenced any where (figure 55)

```
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Ldloc_0, null, "IL_00100"));
try
{
    ilParticlesEmitor.addILParticle(new ILEmitParticle(hashtable[instructionHashes[0]], null, "IL_00110"));
}
catch (Exception)
{
}
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Ldloc_1, null, "IL_00120"));
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Callvirt, typeof(string).GetMethod("get_Chars"), "IL_00130"));
ilParticlesEmitor.addILParticle(new ILEmitParticle(OpCodes.Conv_U8, null, "IL_00180"));
```

Figure 55: first 8 bytes of the first input

The next 8 bytes of the first input must be a call or an arithmetic operation that takes two arguments, since the two calls get_Chars and get_Length has no meaning in this location, so it must be an arithmetic operation (add, sub, div, mul).

the on-the-fly code is doing simple hashing

Result = (`0x2AAAAAAAAAAAAB67 + UTF16(input)`) `* 0x2AAAAAAAAAAAAB6F;`

After creating this function it will be called on each character of the second input after it is concatenated with a character (same index) in the string “0123456789ABCDEFGHIJKLMNOPQR/STUVWXYZabcdefghijklmnopqrstuvwxyz:#@'=\".<(+|$*);,%_>? -&” after hashing compare it with the array of hashes it gets from the pipe server, using brute force to get the type of operation and the second input, so the second part of the first input should be a multiplication (0x2D29C96C) and the second input is “Infant Jesus of Prague”, trying it as third password it was accepted (figure 56)

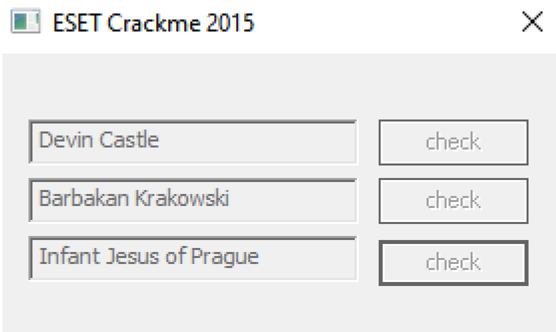


Figure 56:third password

Input the 0364ABE72D29C96C as first input and “Infant Jesus of Prague” (figure 57)

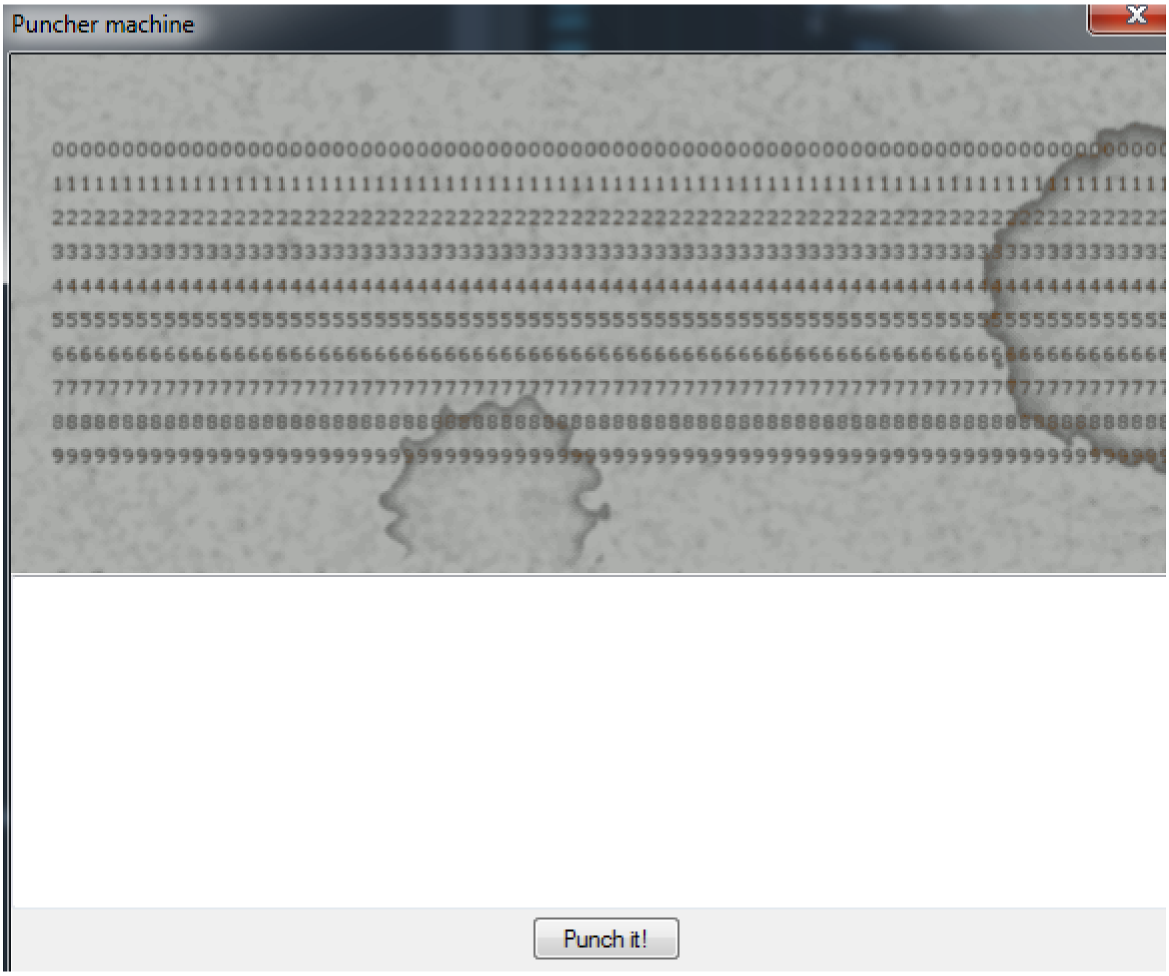


Figure 57: next input form

When providing and input it split it base on new line, then create an image with the name "punch_card_xxx.bmp" for each line, the dropped image is the PunchCard.bmp image but after embedding an encoded message in it (figure 58)

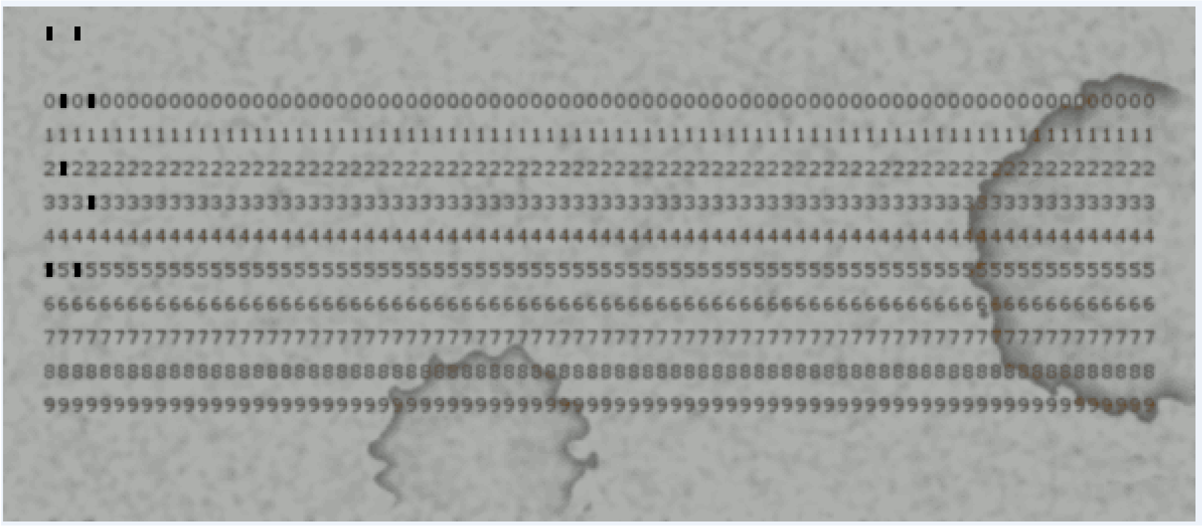


Figure 58: encoded image

4.2 PunchCardReader

Running the PunchCardReader.exe shows a dialog box that the verification is failed (figure 59)

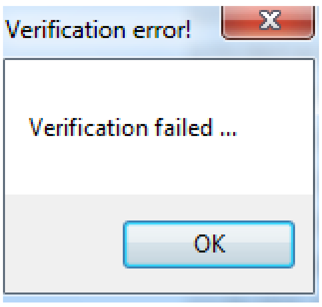


Figure 59:Error message

Open it in dnspy, it has the same initialization as "PuncherMachine.exe":

- MD5 some functions (used for decrypting message from pipe server).
- Communicate with pipe server to receive DLL

Register a function that will be called when the button "Read Punch Card" is pressed, it will read all the images with the format "punch_card_xxx.bmp" decode the message from it (our input from “PuncherMachine.exe”), invoke the createmethod function from the received DLL with our input as argument,

This function looks the same as the createMethod function in CalibrationDynMethod.dll (loaded in “PuncherMachine.exe”), so it’s creating a function on-the-fly using or input (it uses only the first 3 index, which mean that it wants three inputs only) (figure 60)

```
ilgenerator.Emit(OpCodes.Stloc_0);
ilgenerator.Emit(OpCodes.Ldc_I4, 48879);
ilgenerator.Emit(OpCodes.Stloc_1);
ilgenerator.Emit(OpCodes.Ldc_I4, 51966);
ilgenerator.Emit(OpCodes.Stloc_2);
ilgenerator.Emit(OpCodes.Ldc_I4, 47806);
ilgenerator.Emit(OpCodes.Stloc_3);
ilgenerator.Emit(OpCodes.Ldc_I4, 64206);
ilgenerator.Emit(OpCodes.Stloc_S, 4);
ilgenerator.Emit(OpCodes.Ldloc_0);
ilgenerator.Emit(OpCodes.Ldloc_1);
try
{
    ilgenerator.Emit((OpCode)hashtable[instructions[0]]);
}
catch (Exception)
{
}
ilgenerator.Emit(OpCodes.Ldloc_2);
ilgenerator.Emit(OpCodes.Ldloc_3);
try
{
    ilgenerator.Emit((OpCode)hashtable[instructions[1]]);
}
catch (Exception)
{
}
ilgenerator.Emit(OpCodes.Xor);
ilgenerator.Emit(OpCodes.Ldloc_S, 4);
```

Figure 60: Create function on-the-fly

The first two inputs should be an arithmetic operations and the result of both operations will be xored.

The input is validated based on the following equation

(0xDEAD ? 0xBEEF) ^ (0xCAFE ? 0xBABE) ^ 0xFACE ^ 0xf25065b4 == ToUInt32(GetBytes("ESET"))

So, the first two input must be multiplication (mul) and addition (add), ant that input is used as last instruction, so it must be return (ret) (figure 61)

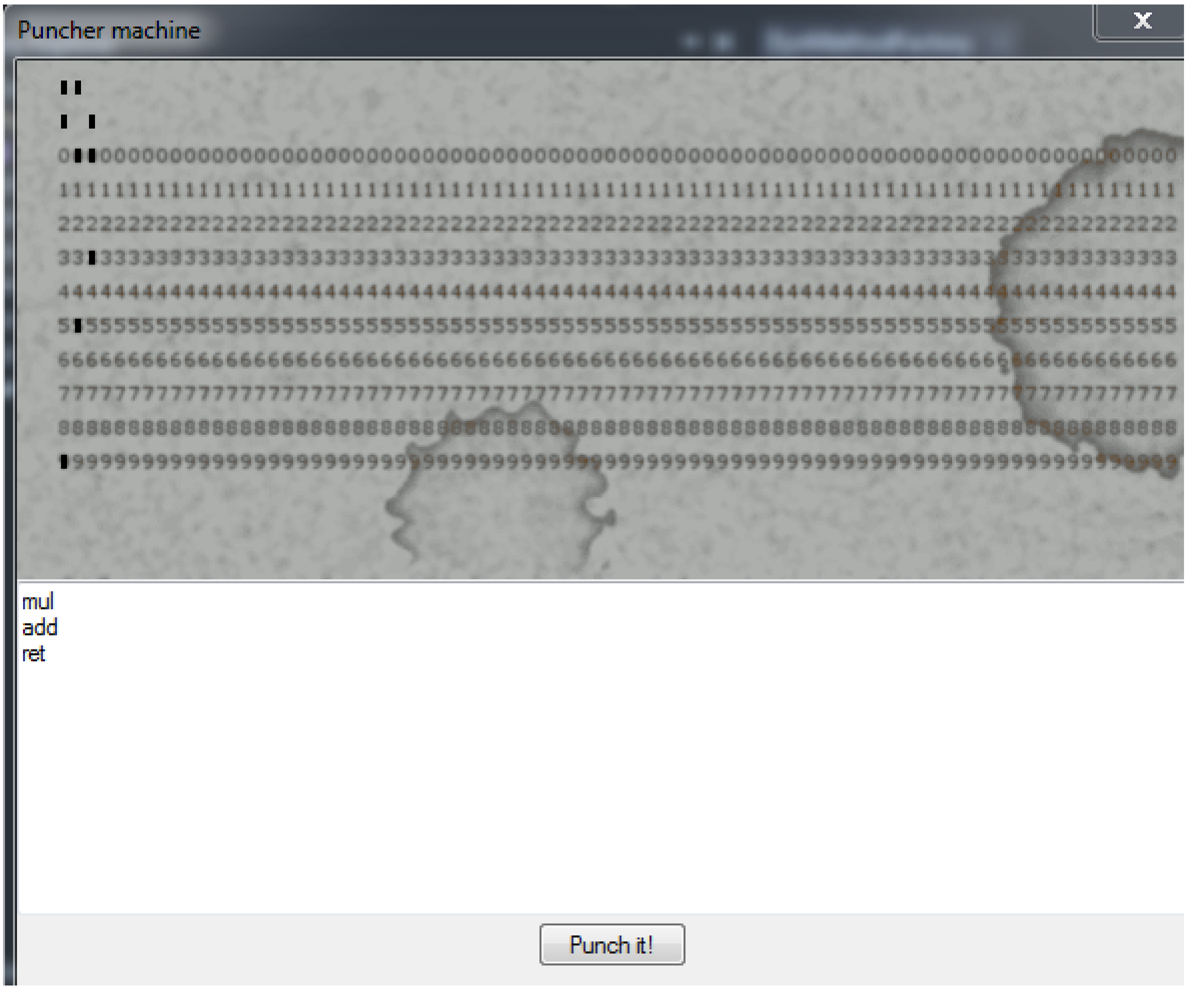


Figure 61: Punch data

After running the "PunchCardReader.exe" we are finally over (figure 62)

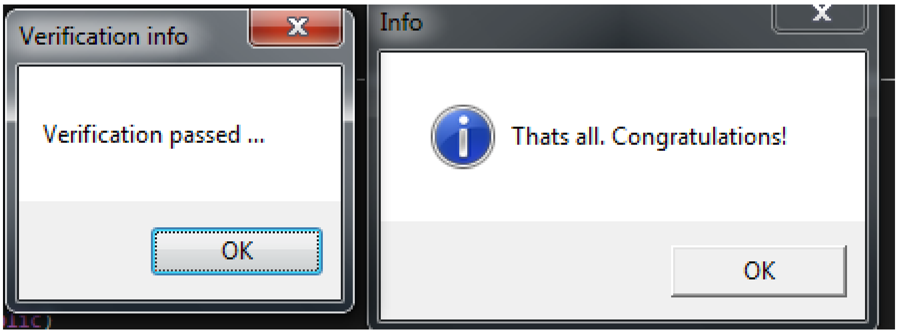


Figure 62: Final dialog box

The three passwords are:

1. Devin Castle
2. Barbakan Krakowski
3. Infant Jesus of Prague