Hi there,

During an ongoing investigation, one of our IR team members managed to locate an unknown sample on an infected machine belonging to one of our clients. We cannot pass that sample onto you currently as we are still analyzing it to determine what data was exfilatrated. However, one of our backend analysts developed a YARA rule based on the malware packer, and we were able to locate a similar binary that seemed to be an earlier version of the sample we're dealing with. Would you be able to take a look at it? We're all hands on deck here, dealing with this situation, and so we are unable to take a look at it ourselves.
We're not too sure how much the binary has changed, **though developing some automation tools might be a good idea**, in case the threat actors behind it start utilizing something like Cutwail to push their samples.
I have uploaded the sample alongside this email.

Thanks, and Good Luck!

Basic Static & Dynamic Anaylsis

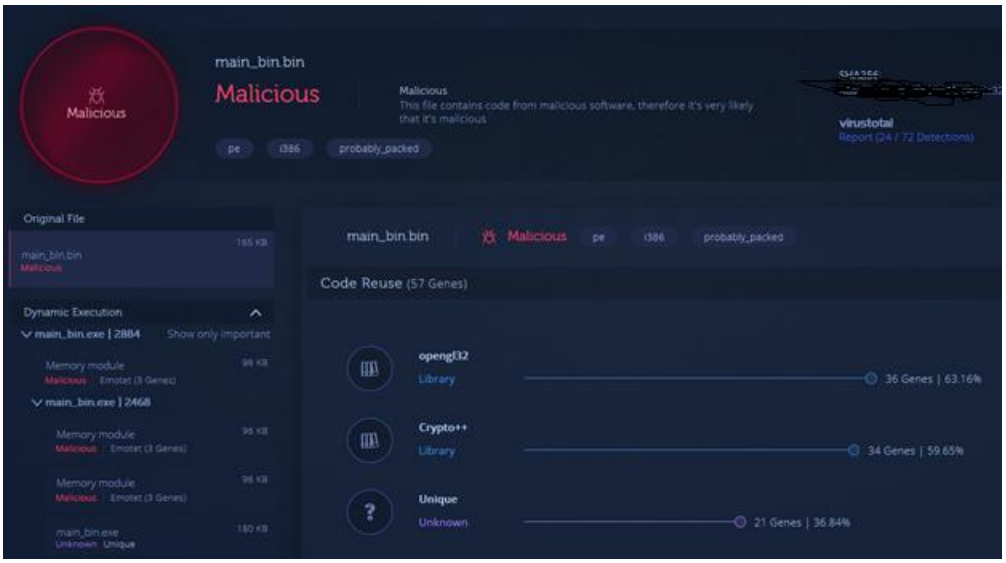
Sample information:
SHA-256: [Redacted]

Intezer: [Redacted]

Any.run:
[Redacted]

VirusTotal:
[Redacted]

We begin by running the sample in a sandboxed environment like **any.run**. We can immediately see that this process launches itself and **svchost** as a sub process. This can also be confirmed within **Intezer**, and in **Intezer** one can examine the strings contained within each sub process. We can assume there is process injection happening.



Regarding any connection made to outer servers, we can see that pastebin.com is being contacted.

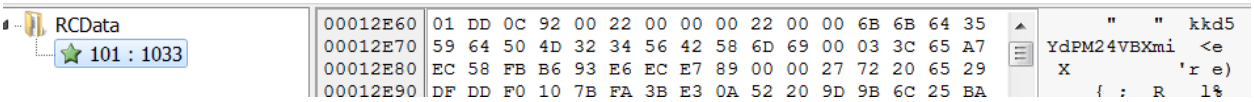
Network Communication

DNS Resolutions

+ pastebin.com

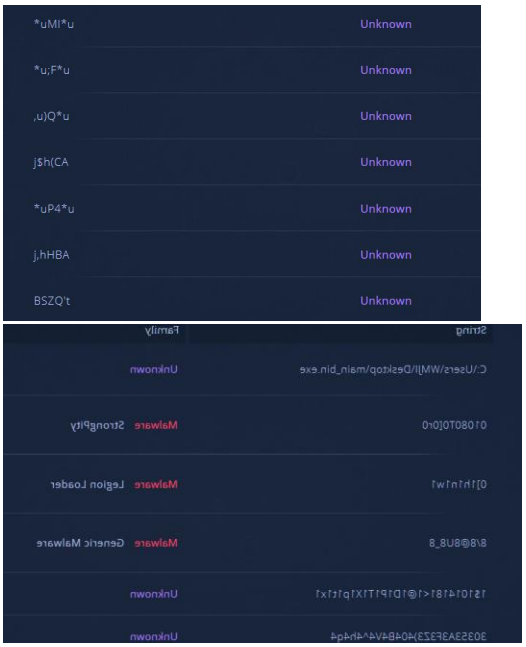
+ pastebin.com

There is a suspicious looking resource within the resource section:



Before the sandbox quit on Any.run – one can see a strange looking MessageBox string.

Finally, the strings found in all first 3 loaded process appear to encrypt, and this can be easily confirmed within Intezer:



Summary:

We're going to be looking for process injection, I suspect that the resource located within the resource section would be mapped into memory, unpacked or decrypted and then injected into the second sub process. There are 4 sub process in total so our final payload would be located on the fourth sub process. It is assumed that the payload would connect to pastebin and display a message box. There is definitely string encryption going on, so we'll have to deal with that as well.

Static and Dynamic Anaylsis

This binary is compiled with Visual Studio C++, the implications of that mean that the actual main function is located somewhere within the **start** function and I've located it by recursively traversing xrefs from one of the imported functions. The main function is located at **0x00401400**

```
loc_4017EF:
call    sub_404099
mov     edi, eax
call    sub_404433
mov     esi, [eax]
call    sub_40442D
push    edi
push    esi
push    dword ptr [eax]
call    func_main?
add     esp, 0Ch
mov     esi, eax
call    sub_401E45
test    al, al
jz      short loc_401884
```

When viewing the code of the main function one can see gibberish strings being pushed before function calls:

```
mov     ecx, offset a5ea5Qpy4_0 ; ".5ea5/QPY4//"
call    sub_401300
mov     ecx, offset aS9a4e5fbhe35n ; "s9a4E5fbhe35n"
call    sub_401300
mov     edi, ds:LoadLibraryA
push    offset a5ea5Qpy4_0 ; ".5ea5/QPY4//"
call    edi ; LoadLibraryA
mov     ebx, ds:GetProcAddress
push    offset aS9a4e5fbhe35n ; "s9a4E5fbhe35n"
push    eax             ; hModule
call    ebx ; GetProcAddress
mov     ecx, offset aYb14e5fbhe35 ; "yb14E5fbhe35"
mov     esi, eax
call    sub_401300
push    offset a5ea5Qpy4_0 ; ".5ea5/QPY4//"
call    edi ; LoadLibraryA
push    offset aYb14e5fbhe35 ; "yb14E5fbhe35"
push    eax             ; hModule
call    ebx ; GetProcAddress
mov     ecx, offset aF9m5b6e5fbhe35 ; "F9m5b6E5fbhe35"
mov     edi, eax
call    sub_401300
push    offset a5ea5Qpy4_0 ; ".5ea5/QPY4//"
call    ds:LoadLibraryA
push    offset aF9m5b6e5fbhe35 ; "F9m5b6E5fbhe35"
```

The function that is called after each push is the same, I instantly assume there is some string encryption going on. The extensive use of **LoadLibraryA** and **GetProcAddress** also makes me assume these strings are API strings that are resolved using **LoadLibraryA** and **GetProcAddress**.

At **loc_401550** and **loc_401570** one can located something that resembles a RC4 KSA routine, it is easily recognized by the two loop procedures iterating 256 times.

```
loc_401550:
mov     [ebp+eax+var_108], al
inc     eax
cmp     eax, 256
jl      short loc_401550

mov     edi, [ebp+lpResourceAddrInSection]
xor     esi, esi
nop     word ptr [eax+eax+00000000h]

func_RC4DecryptResource:
mov     bl, [ebp+esi+var_108]
mov     eax, 88888889h
mul     esi
mov     eax, esi
shr     edx, 3
mov     ecx, edx
shl     ecx, 4
sub     ecx, edx
sub     eax, ecx
lea     ecx, [ebp+var_108]
movzx   eax, byte ptr [eax+edi+0Ch]
add     al, bl
add     bh, al
movzx   eax, bh
add     ecx, eax
movzx   eax, byte ptr [ecx]
mov     [ebp+esi+var_108], al
inc     esi
mov     [ecx], bl
cmp     esi, 256
```

These are just quick assumptions I've made by looking at the binary, the rest of it is filled with obfuscated code and otherwise an extensive use of registers and dynamic resolving so we'll have to resort to dynamic analysis. I've disabled ASLR for this binary with CFF explorer so it would be easier to debug it.

I've set relevant breakpoints within the debugger. So all the resource related functions to locate any resource loading, CreateProcessInternalW for process initialization and VirtualProtect, VirtualAllocEx and WriteProcessMemory for injection. In addition I've set breakpoints on OutputDebugString and IsDebuggerPresent to catch easy implemented anti analysis.

```
76BE432F <kernel32.dll.VirtualProtect>        Enabled  mov edi,edi                    0
76BE4A2D <kernel32.dll.IsDebuggerPresent>     Enabled  jmp <JMP.&IsDebuggerPresent>    0
76BE5929 <kernel32.dll.LockResource>          Enabled  mov edi,edi                    0
76BE5941 <kernel32.dll.FindResourceW>         Enabled  mov edi,edi                    0
76BF3BC3 <kernel32.dll.CreateProcessInternalW Enabled  push 624                       0
76BFD978 <kernel32.dll.VirtualAllocEx>        Enabled  mov edi,edi                    0
76BFD9A8 <kernel32.dll.WriteProcessMemory>    Enabled  mov edi,edi                    0
76BFE983 <kernel32.dll.FindResourceA>         Enabled  mov edi,edi                    0
76C0B27F <kernel32.dll.OutputDebugStringA>    Enabled  mov edi,edi                    0
76C0D1C4 <kernel32.dll.OutputDebugStringW>    Enabled  mov edi,edi                    0
```

First, as assumed the sub_401300 seems to be resolving strings:

```
Breakpoint Not Set
00401420  B9 40494100   mov ecx,main_bin.414940          414940:"FindResourceA"
00401425  E8 D6FEFFFF   call <main_bin.func_StringDecrypt>
0040142A  8B3D 04E04000 mov edi,dword ptr ds:[<&LoadLibraryA>]
00401430  68 20494100   push main_bin.414920             414920:"kernel32.dll"
00401435  FFD7          call edi
00401437  8B1D 08E04000 mov ebx,dword ptr ds:[<&GetProcAddress>]
0040143D  68 40494100   push main_bin.414940             414940:"FindResourceA"
00401442  50            push eax
00401443  FFD3          call ebx
00401445  B9 70494100   mov ecx,main_bin.414970          414970:"LoadResource"
0040144A  8BF0          mov esi,eax
0040144C  E8 AFFEFFFF   call <main_bin.func_StringDecrypt>
00401451  68 20494100   push main_bin.414920             414920:"kernel32.dll"
00401456  FFD7          call edi
00401458  68 70494100   push main_bin.414970             414970:"LoadResource"
0040145D  50            push eax
0040145E  FFD3          call ebx
00401460  B9 30494100   mov ecx,main_bin.414930          414930:"SizeofResource"
00401465  8BF8          mov edi,eax
00401467  E8 94FEFFFF   call <main_bin.func_StringDecrypt>
0040146C  68 20494100   push main_bin.414920             414920:"kernel32.dll"
00401471  FF15 04E04000 call dword ptr ds:[<&LoadLibraryA>]
00401477  68 30494100   push main_bin.414930             414930:"SizeofResource"
0040147C  50            push eax
0040147D  FFD3          call ebx
0040147F  B9 60494100   mov ecx,main_bin.414960          414960:"LockResource"
```

The function func_StringDecrypt seems to be decrypting strings using some kind of custom base64 decoding, The reason I suspect is – is because the use of the full alphanumeric string that is being passed within this function. This function would be studied extensively later on in this paper and we'll attempt to generate an automation script for it to decode all strings within this sample.

```
00401336  6666:0F1F8400 0000000 nop word ptr ds:[eax+eax],ax
00401340  0F1005 102C4100       movups xmm0,xmmword ptr ds:[412C10]   00412C10:"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU
00401347  66:A1 502C4100        mov ax,word ptr ds:[412C50]           00412C50:"/="
0040134D  8A1C31                mov bl,byte ptr ds:[ecx+esi]
00401350  0F1145 B8             movups xmmword ptr ss:[ebp-48],xmm0
00401354  66:8945 F8            mov word ptr ss:[ebp-8],ax
00401358  0F1005 202C4100       movups xmm0,xmmword ptr ds:[412C20]   00412C20:"qrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789(
0040135F  A0 522C4100           mov al,byte ptr ds:[412C52]
00401364  6A 01                 push 1
00401366  0F1145 C8             movups xmmword ptr ss:[ebp-38],xmm0
0040136A  8845 FA               mov byte ptr ss:[ebp-6],al
0040136D  0F1005 302C4100       movups xmm0,xmmword ptr ds:[412C30]   00412C30:"GHIJKLMNOPQRSTUVWXYZ01234567890./="
00401374  0F1145 D8             movups xmmword ptr ss:[ebp-28],xmm0
00401378  0F1005 402C4100       movups xmm0,xmmword ptr ds:[412C40]   00412C40:"WXYZ01234567890./="
0040137F  0F1145 E8             movups xmmword ptr ss:[ebp-18],xmm0
00401383  E8 6C250000           call main_bin.4038F4
```

Then the sample does something interesting:

```
; Allocate Memory for resource
;
call    func_HeapAlloc
add     esp, 4
push    edi
call    [ebp+LPVOID_LockResource]
mov     ebx, eax
mov     [ebp+lpResourceAddrInSection], ebx
mov     ecx, [ebx+8]
lea     edi, [ecx+ecx*4]
mov     ecx, offset VirtualAlloc ; "I9egh1/n//b3"
add     edi, edi
mov     [ebp+var_118], edi
call    func_StringDecryptCustomBase64
push    offset kernel32_dll ; ".5ea5/QPY4//"
call    ds:LoadLibraryA
push    offset VirtualAlloc ; "I9egh1/n//b3"
push    eax             ; hModule
call    ds:GetProcAddress
push    4
push    1000h
;
; Resource size IS 1541c but 15400 is passed
;
push    edi
push    0
call    eax
```

It allocates space for the resource but it skips the first 0x1C bytes within the resource.

Hmm, perhaps this is the decryption key for the RC4 algorithm we saw before?

```
00416060 | 01 DD 0C 92 | 00 22 00 00 | 00 22 00 00 | 6B 6B 64 35 | .Ý..."..."..kkd5
00416070 | 59 64 50 4D | 32 34 56 42 | 58 6D 69 00 | 03 3C 65 A7 | YdPM24VBXmi..<e§
```

Then a call is performed to **sub_402DB0** which I renamed to func_CopyResourceWithoutKeyToAllocMem because that is exactly what it does, it just copies the resource without its key, so starting at offset 0x1C.

```
call    func_CopyResourceWithoutKeyToAllocMem
push    102h
lea     eax, [ebp+var_108]
xor     bh, bh
push    0
push    eax
call    sub_4025B0
```

because that is exactly what it does, it just copies the resource without its key, so starting at offset 0x1C. This function would seem very confusing at first, but if we set a hardware breakpoint on the allocated memory we'll break within this function we can confirm this. Because at **00403002** it performs this copying procedure and then it just exists the function:

```
00430000 03 3C 65 A7 EC 58 FB B6 93 E6 EC E7 89 00 00 27 .<e§ìXû¶.æìç...'
00430010 72 20 65 29 DF DD F0 10 7B FA 3B E3 0A 52 20 9D r e)ßÝð.{ú;ã.R .
00430020 9B 6C 25 BA 4A EF 5B 08 D4 0E 77 F1 50 E3 08 9C .l%ºJï[.Ô.wñPã..
00430030 11 36 E8 E0 9F 82 BD F5 89 B8 96 52 50 9C D3 2C .6èà..½õ.¸.RP.Ó,
00430040 6D 59 19 CE D4 82 54 DA 8A 93 19 99 1C 21 A9 12 mY.ÎÔ.TÚ.....!©.
00430050 C5 2A 1B 4A AD A6 14 3E CE 37 98 B1 AD 47 25 3D Å*.J.¦.>Î7.±.G%=
00430060 C9 A6 09 98 4C E1 4E 14 F9 13 06 2D 15 82 64 99 É¦..LáN.ù..-..d.
00430070 BD 28 0E 41 D1 CD F8 3C 9D 54 CA 60 BB E2 8D 91 ½(.AÑÍø<.TÊ`»â..
00430080 61 C8 B5 21 9B 0A A4 4F 08 08 EA 5A B7 A5 71 D6 aÈµ!..¤O..êZ·¥qÖ
00430090 7A AE 1D F2 3C 91 D0 E0 40 0A AB F0 65 3C E5 1D z®.ò<.Ðà@.«ðe<å.
004300A0 0E 79 FE 2C B2 5B 60 63 2A 72 1E 0D A9 53 B4 39 .yþ,²[`c*r..©S´9
004300B0 90 10 30 05 58 AF 0C 78 6D F6 AE D4 5B 1C 3E 1C ..0.X¯.xmö®Ô[.>.
004300C0 D3 96 C0 B6 A9 7D 28 A9 F4 F0 99 0D 92 02 38 F0 Ó.À¶©}(©ôð...8ð
```

```
00403000   F3:A5                 rep movsd
00403002   83E2 03               and edx,3
00403005   FF2495 14304000       jmp dword ptr ds:[edx*4+403014]
0040300C   FF248D 24304000       jmp dword ptr ds:[ecx*4+403024]
00403013   90                    nop
00403014   24 30                 and al,30
00403016   40                    inc eax
00403017   002C30                add byte ptr ds:[eax+esi],ch
00403018   40                    inc eax
0040301B   0038                  add byte ptr ds:[eax],bh
0040301D   3040 00               xor byte ptr ds:[eax],al
00403020   4C                    dec esp
00403021   3040 00               xor byte ptr ds:[eax],al
00403024   8B4424 0C             mov eax,dword ptr ss:[esp+C]
00403028   5E                    pop esi
00403029   5F                    pop edi
0040302A   C3                    ret
```

Breakpoint Not Set

Then sub_4025B0 is executed, it receives a stack address and we simply skip this function while looking at that address on dump we can see that it simply zeroes it out.

```
0018FE38 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FE48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FE58 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FE68 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FE78 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FE88 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FE98 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FEA8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FEB8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FEC8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FED8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FEE8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0018FEF8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

So thankfully I'm saving a lot of time by skipping these rabbit holes.

Then the assumed RC4 algorithm executes, what I want to do is locate the address to where the sample mapped the resource to memory as I suspect that it's going to be decrypted.

```
00430000 03 3C 65 A7 EC 58 FB B6 93 E6 EC E7 89 00 00 27 .<e§ìXû¶.æìç...'
00430010 72 20 65 29 DF DD F0 10 7B FA 3B E3 0A 52 20 9D r e)ßÝð.{ú;ã.R .
00430020 9B 6C 25 BA 4A EF 5B 08 D4 0E 77 F1 50 E3 08 9C .l%ºJï[.Ô.wñPã..
00430030 11 36 E8 E0 9F 82 BD F5 89 B8 96 52 50 9C D3 2C .6èà..½õ.¸.RP.Ó,
00430040 6D 59 19 CE D4 82 54 DA 8A 93 19 99 1C 21 A9 12 mY.ÎÔ.TÚ.....!©.
00430050 C5 2A 1B 4A AD A6 14 3E CE 37 98 B1 AD 47 25 3D Å*.J.¦.>Î7.±.G%=
00430060 C9 A6 09 98 4C E1 4E 14 F9 13 06 2D 15 82 64 99 É¦..LáN.ù..-..d.
00430070 BD 28 0E 41 D1 CD F8 3C 9D 54 CA 60 BB E2 8D 91 ½(.AÑÍø<.TÊ`»â..
00430080 61 C8 B5 21 9B 0A A4 4F 08 08 EA 5A B7 A5 71 D6 aÈµ!..¤O..êZ·¥qÖ
00430090 7A AE 1D F2 3C 91 D0 E0 40 0A AB F0 65 3C E5 1D z®.ò<.Ðà@.«ðe<å.
004300A0 0E 79 FE 2C B2 5B 60 63 2A 72 1E 0D A9 53 B4 39 .yþ,²[`c*r..©S´9
004300B0 90 10 30 05 58 AF 0C 78 6D F6 AE D4 5B 1C 3E 1C ..0.X¯.xmö®Ô[.>.
004300C0 D3 96 C0 B6 A9 7D 28 A9 E4 E0 99 0D 92 02 38 E0 Ó.À¶©}(©äà..'.8à
```

So I'm going to skip the RC4 decryption routine and jump straight to address **0x40161D**

```
00430000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ..........yy..
00430010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ¸.......@.......
00430020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00430030 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 ................
00430040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..º..´.Í!¸.LÍ!Th
00430050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
00430060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS 
00430070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode....$.......
00430080 2A E9 99 31 6E 88 F7 62 6E 88 F7 62 6E 88 F7 62 *é.1n.÷bn.÷bn.÷b
00430090 7A E3 F4 63 64 88 F7 62 7A E3 F2 63 E1 88 F7 62 zãôcd.÷bzãòcá.÷b
004300A0 7A E3 F3 63 7C 88 F7 62 65 E7 F2 63 4B 88 F7 62 zãóc|.÷beçòcK.÷b
004300B0 65 E7 F3 63 7F 88 F7 62 65 E7 F4 63 7F 88 F7 62 eçóc..÷beçôc..÷b
004300C0 7A E3 F6 63 6D 88 F7 62 6E 88 F6 62 3B 88 F7 62 zãöcm.÷bn.öb;.÷b
```

Yay! I decide to dump the new PE file out to disk but we're not done yet. We have to see how it's going to be injected to memory.

We jump into **sub_401000**

```
mov     ebx, [edi+IMAGE_DOS_HEADER.e_lfanew]
add     ebx, edi
mov     [ebp+addr_NTHeaders], ebx
call    ds:GetModuleFileNameA
cmp     dword ptr [ebx], 'EP'
jnz     loc_4012E2
```

First the sample resolves the ImageBase of the current executing sample and then it attempts to confirm and locate the address of the NT_Headers of the decrypted payload.

Then the current process main.bin is created in suspended mode

```
004010A3   6A 00          push 0
004010A5   6A 00          push 0
004010A7   6A 00          push 0
004010A9   8D8D FCFBFFFF  lea ecx,dword ptr ss:[ebp-404]
004010AF   51             push ecx                    ecx:"C:\\Users\\Danus\\Desktop\\Malware Test\\main_bin.bi
004010B0   FFD0           call eax
```

Seems like there is going to be process injection involved.

```
push    40h
push    3000h
push    [ebx+IMAGE_NT_HEADERS.OptionalHeader.SizeOfImage]
lea     ecx, [ebx+IMAGE_NT_HEADERS.OptionalHeader.ImageBase]
push    dword ptr [ecx]
mov     [ebp+PayloadImageBase], ecx
push    dword ptr [ebp+NewProcHandle]
call    eax
```

First memory is allocated within the new process. The sample attempts to execute VirtualAllocEx on the new process. Attempting to allocate memory at the payloads PE default ImageBase and with the virtual Size of the image. This won't work though under our modified execution since we disabled ASLR. Why? Because both processes execute at the same ImageBase, the new process already has memory allocated within that region, so we must enable ASLR and start again. So, lets do that just that and we'll see that it would work.

Then the sample will copy the payloads section to their correct virtual addresses:

```
addr_MapSectionsPayLoad:
mov     ecx, [edi+IMAGE_DOS_HEADER.e_lfanew]
add     ecx, esi
push    0
push    dword ptr [ecx+edi+108h]
mov     eax, [ecx+edi+10Ch]
add     eax, edi
push    eax
mov     eax, [ecx+edi+104h]
add     eax, [ebp+var_46C]
push    eax
push    dword ptr [ebp+NewProcHandle]
call    [ebp+addr_WriteProcessMemory]
mov     eax, [ebp+addr_NTHeaders]
inc     ebx
add     esi, 28h
movzx   eax, word ptr [eax+6]
cmp     ebx, eax
jl      short addr_MapSectionsPayLoad
```

Then something very interesting happens:
The ImageBase of the payload is written to the PEB of the new process, specifically at offset 0x8.

```
1: [esp] 0000003C
2: [esp+4] 7EFDE008
3: [esp+8] 00430134
4: [esp+C] 00000004
5: [esp+10] 00000000
```

Syntax

```
struct _PEB {
    0x000 BYTE InheritedAddressSpace;
    0x001 BYTE ReadImageFileExecOptions;
    0x002 BYTE BeingDebugged;
    0x003 BYTE SpareBool;
    0x004 void* Mutant;
    0x008 void* ImageBaseAddress;
```

```
push    dword ptr [ebp+NewProcHandle]
;
; Write payload ImageBase into PEB in the new process
;
call    [ebp+addr_WriteProcessMemory]
```

We can assume that the payload would use this to resolve its own APIs.

Finally, **SetThreadContext** and **ResumeThread** are called, and the injected payload executes.

```
push    dword ptr [ebp+NewProcHandle+4]
call    edi
push    dword ptr [ebp+NewProcHandle+4]
call    esi
```

Second Payload Analysis

Sample information:
SHA-256: [Redacted]

Intezer: [Redacted]

Any.run: [Redacted]

VirusTotal: [Redacted]

Advanced Static and Dynamic Analysis

I'm going to assume process injection again and go straight into analysis, I'll disable ASLR for this execution as I would be executing the second stage payload independently and so the previous problem, we encountered due disabling ASLR shouldn't bother us.

It is observed that this second stage contains API hashing as the extensive use of CRC32 constants indicates that:

```
CRC32 [poly] :: 000004EA :: 004010EA
CRC32 [poly] :: 000004FB :: 004010FB
CRC32 [poly] :: 0000050D :: 0040110D
CRC32 [poly] :: 0000051F :: 0040111F
CRC32 [poly] :: 00000531 :: 00401131
CRC32 [poly] :: 00000543 :: 00401143
CRC32 [poly] :: 00000555 :: 00401155
CRC32 [poly] :: 00000566 :: 00401166
CRC32 [poly] :: 00000A81 :: 00401681
CRC32 [poly] :: 00000A92 :: 00401692
```

In addition, the function that was seen in the first stage loader that simply copies the payload into memory from the resource section can be seen:

```
func_CopyPayloadAsSeenBefore??? proc near

arg_0= dword ptr  4
arg_4= dword ptr  8
arg_8= dword ptr  0Ch

push    edi
push    esi
mov     esi, [esp+8+arg_4]
mov     ecx, [esp+8+arg_8]
mov     edi, [esp+8+arg_0]
mov     eax, ecx
mov     edx, ecx
add     eax, esi
cmp     edi, esi
jbe     short loc_4037D0
```

I've set relevant breakpoints within the debugger. So, all the resource related functions to locate any resource loading, CreateProcessInternalW for process initialization and VirtualProtect, VirtualAllocEx and WriteProcessMemory for injection. In addition, I've set breakpoints on OutputDebugString and IsDebuggerPresent to catch easy implemented anti analysis.

The malware takes the name of the file that its currently executing from and hashes it using a CRC32 hashing algorithm, it can be identified by the CRC32 hashing constants found within this function. The sample then compares the result against a constant value. I'm assuming its using this method as an anti-analysis method to check if the samples name is "sample" or "malware" its really hard to tell. If this check matches the sample quits execution.

```
00401F07   2BCA            sub ecx,edx
00401F09   8BD6            mov edx,esi                esi:"main_bin_dump.bin"
00401F0B   51              push ecx
00401F0C   E8 4FF7FFFF     call main_bin_dump.401660
00401F11   83C4 04         add esp,4
00401F14   3D 2DC425B9     cmp eax,B925C42D
00401F19   0F84 66010000   je main_bin_dump.402085
```

Then an API resolving routine called which utilizes the CRC32 hashing algorithm we seen earlier:

```
mov     edx, 8436F795h  ; hashId
xor     ecx, ecx        ; libarg
call    func_APIHash
```

The HashID(EDX) and the ID(ECX) which identifies the library to which to resolve the API from

```
push    ds:lpLibFileName[ecx*4] ; lpLibFileName
mov     [ebp+var_C], edx
call    ds:LoadLibraryA
```

```
lpLibFileName   dd offset aKernel32Dll_0
                                        ; DATA XREF: func_APIHash+9↑r
                                        ; "kernel32.dll"
                dd offset aNtdllDll     ; "ntdll.dll"
                dd offset aWininetDll   ; "wininet.dll"
aCruloader      db 'cruloader',0        ; DATA XREF: sub_401290+45↑o
```

I will not explain how the API Hashing and resolving in depth. Basically the export table of each loaded DLL is hashed to check which function name matches the hash passed into the function. If anyone wants to read about how that might be implemented they can read about it [on my github right here](#).

The sample loads IsDebuggerPresent to check if the malware is executing under a debugger, this can be easily circumvented.
The next anti analysis method located within **sub_401000** checks if any blacklisted process is running – the malware hashes each running process and then checks if the hash matches to a precomputed hash array. If they match the malware quits execution.

```
addr_check_if_match:
cmp     dword ptr [ebp+ecx*4+var_14], eax
jz      short loc_4011F7
```

Then the sample executes **sub_401D50** which resolves a lot of APIs that might indicate process injection

```
sub_401D50 proc near
mov     edx, 0A851D916h ; hashId
xor     ecx, ecx        ; libarg
call    func_APIHash
mov     edx, 4F58972Eh  ; hashId
mov     CreateProcessW, eax
xor     ecx, ecx        ; libarg
call    func_APIHash
mov     edx, 3872BEB9h  ; hashId
mov     WriteProcessMemory, eax
xor     ecx, ecx        ; libarg
call    func_APIHash
mov     edx, 0E62E824Dh ; hashId
xor     ecx, ecx        ; libarg
call    func_APIHash
mov     edx, 9CE0D4Ah   ; hashId
mov     VirtualAllocEx, eax
xor     ecx, ecx        ; libarg
call    func_APIHash
mov     edx, 0FF808C10h ; hashId
mov     addrVirtualAlloc, eax
xor     ecx, ecx        ; libarg
call    func_APIHash
mov     CreateRemoteThread, eax
xor     eax, eax
retn
sub_401D50 endp
```

1. Start **svchost** with suspended flags
2. Copy current PE into allocated memory
3. Allocate Memory in **svchost**
4. Rebuild current PE payload relocation table
5. Write payload into **svchost**
6. Use **CreateRemoteThread** to execute function **sub_401DC0**

To continue execution, we simply must attach to a second debugger instance and set a breakpoint on the functions location and after running **CreateRemoteThread** we should hit it.

```
00081DC0   53                 push ebx
00081DC1   8BDC               mov ebx,esp
00081DC3   83EC 08            sub esp,8
00081DC6   83E4 F0            and esp,FFFFFFF0
00081DC9   83C4 04            add esp,4
00081DCC   55                 push ebp
00081DCD   8B6B 04            mov ebp,dword ptr ds:[ebx+4]
00081DD0   896C24 04          mov dword ptr ss:[esp+4],ebp
00081DD4   8BEC               mov ebp,esp
00081DD6   83EC 40            sub esp,40
00081DD9   A1 04500900        mov eax,dword ptr ds:[95004]
00081DDE   33C5               xor eax,ebp
00081DE0   8945 FC            mov dword ptr ss:[ebp-4],eax
00081DE3   BA 3DA816DA        mov edx,DA16A83D
00081DE8   B9 02000000        mov ecx,2
00081DED   E8 1EF4FFFF        call 81210
00081DF2   BA E0056501        mov edx,16505E0
00081DF7   A3 B86A0900        mov dword ptr ds:[96AB8],eax
00081DFC   B9 02000000        mov ecx,2
00081E01   E8 0AF4FFFF        call 81210
00081E06   BA F598C06C        mov edx,6CC098F5
00081E0B   A3 D86A0900        mov dword ptr ds:[96AD8],eax
00081E10   B9 02000000        mov ecx,2
00081E15   E8 F6F3FFFF        call 81210
00081E1A   BA 241D19E5        mov edx,E5191D24
00081E1F   A3 BC6A0900        mov dword ptr ds:[96ABC],eax
00081E24   B9 02000000        mov ecx,2
00081E29   E8 E2F3FFFF        call 81210
00081E2E   0F1005 7C3C0900    movups xmm0,xmmword ptr ds:[93C7C]
00081E35   A3 D46A0900        mov dword ptr ds:[96AD4],eax
```

After setting up the breakpoint lets resume execution of the svchost instance and then skip CreateRemoteThread and see if anything happens.

```
EIP EDX   00081DC0   53                 push ebx
          00081DC1   8BDC               mov ebx,esp
          00081DC3   83EC 08            sub esp,8
          00081DC6   83E4 F0            and esp,FFFFFFF0
          00081DC9   83C4 04            add esp,4
          00081DCC   55                 push ebp
          00081DCD   8B6B 04            mov ebp,dword ptr ds:[ebx+4]
          00081DD0   896C24 04          mov dword ptr ss:[esp+4],ebp
          00081DD4   8BEC               mov ebp,esp
```

Success! We can continue analyzing this function within our documented IDA instance as this function is located in the second stage payload.

It's important to note that interestingly enough – this function is called previously at **loc_402085** when the func_CRC32Hash returns a hashed value for the current executing process name which matches a hardcoded hash.

```
sub     ecx, edx
mov     edx, esi
push    ecx
call    func_CRC32Hash
add     esp, 4
cmp     eax, 0B925C42Dh
jz      loc_402085

mov     edx, 8436F795h  ; hashId
xor     ecx, ecx        ; libarg
call    func_APIHash
call    eax
test    eax, eax
jnz     loc_402071

loc_402085:
call    sub_401DC0
mov     ecx, [ebp+var_4]
xor     eax, eax
xor     ecx, ebp
```

I quickly assumed that this is a method to detect if the binary is running from **svchost**, because as we seen in the malware's execution process tree, it executes **svchost** twice. I later confirmed this check by renaming the sample to **svchost.exe** and it worked:

```
00401F00  8A01          mov al,byte ptr ds:[ecx]
00401F02  41            inc ecx
00401F03  84C0          test al,al
00401F05  75 F9         jne svchost.401F00
00401F07  2BCA          sub ecx,edx
00401F09  8BD6          mov edx,esi                 esi:"svchost.exe"
00401F0B  51            push ecx
00401F0C  E8 4FF7FFFF   call svchost.401660
00401F11  83C4 04       add esp,4
00401F14  3D 2DC425B9   cmp eax,B925C42D
00401F19  0F84 66010000 je svchost.402085

EAX  B925C42D
EBX  7EFDE000
ECX  00000065
EDX  000000AF
EBP  0018FF40
ESP  0018FAF8
ESI  0018FB50
EDI  00562AC8
EIP  00401F19
```

Let's continue with the analysis.

This function first resolves a few Internet WINAPIs

```
00081DE8  B9 02000000      mov ecx,2
00081DED  E8 1EF4FFFF      call 81210
00081DF2  BA E0056501      mov edx,16505E0
00081DF7  A3 B86A0900      mov dword ptr ds:[<&InternetOpenA>],eax
00081DFC  B9 02000000      mov ecx,2
00081E01  E8 0AF4FFFF      call 81210
00081E06  BA F598C06C      mov edx,6CC098F5
00081E0B  A3 D86A0900      mov dword ptr ds:[<&InternetOpenUrlA>],
00081E10  B9 02000000      mov ecx,2
00081E15  E8 F6F3FFFF      call 81210
00081E1A  BA 241D19E5      mov edx,E5191D24
00081E1F  A3 BC6A0900      mov dword ptr ds:[<&InternetReadFile>],
00081E24  B9 02000000      mov ecx,2
00081E29  E8 E2F3FFFF      call 81210
00081E2E  0F1005 7C3C0900  movups xmm0,xmmword ptr ds:[93C7C]
00081E35  A3 D46A0900      mov dword ptr ds:[<&InternetCloseHandle
```

Then a strange string located at offset **0x0413C7C** is passed into a code block and decrypted by a very simple algorithm:

```
loc_401E60:
mov     dl, [ebp+ecx-40h]
rol     dl, 4
xor     dl, 0C5h
mov     [ebp+ecx-40h], dl
inc     ecx
cmp     ecx, eax
jl      short loc_401E60
```

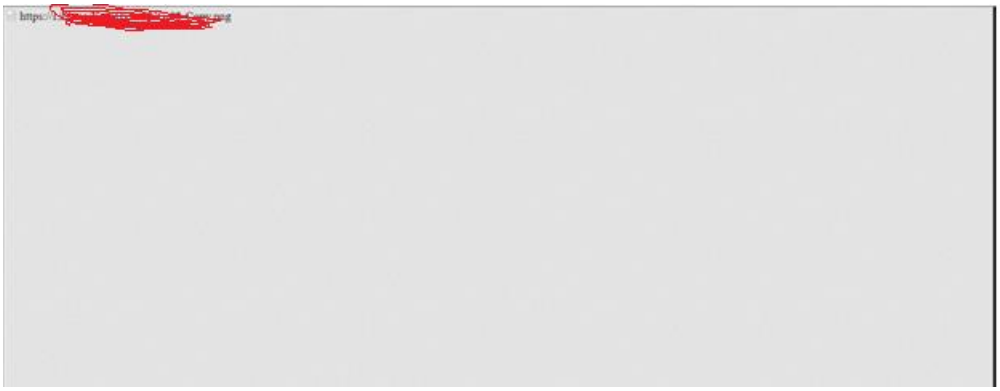
So, our encrypted byte sequence:

```
xmmword 413C7C  xmmword 0BACA7A0A1B6B4A5BAEAEFF6B5B1B1BDAh
```

Returns:

```
ESP    0049FA10    "https://pastebin.com/raw/[redacted]"
EST    00000000
```

Which resolves to **[Redacted]**



Hmm.. This might indicate stenography is involved. Let's continue with the analysis

This pastebin link is passed into **sub_401290** this function returns the image link contained within the pastebin and saves it within the memory.

This resolved link is passed into **sub_4013A0,** first the function reads the contents of the image file linked passed into it. Using **sub_401290** which I renamed to **func_ReadWebContents**

```
push    esi                     ; hashId
push    edi                     ; libarg
call    func_ReadWebContents
movq    xmm0, ds:qword_413CA4
mov     ebx, eax
mov     eax, dwSize
mov     [ebp+var_270], eax
mov     eax, ds:dword_413CAC
mov     [ebp+var_8], eax
lea     eax, [ebp+String]
push    eax                     ; lpString
mov     [ebp+var_268], 0
movq    qword ptr [ebp+String], xmm0
call    ds:lstrlenA
xor     edx, edx
```

Then the function decodes a string located with qword_413CA4

```
qword_413CA4    dq 13B6A6F6B6A60734h
```

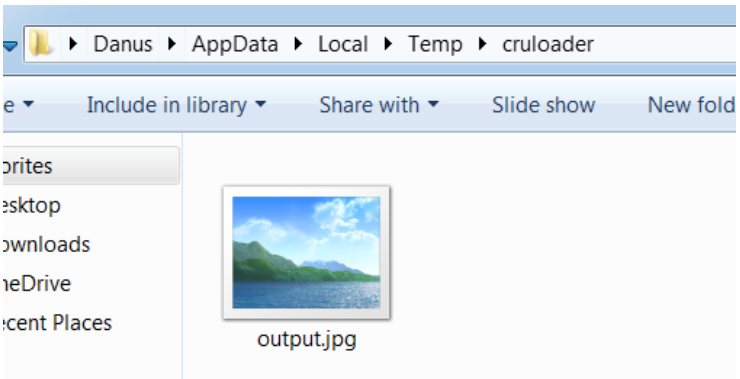
Which resolves to **output.jpg**, then it computes a path to the Temp directory and converts the it to a WCHAR type string, then specially picked bytes are extracted from the data section to compute this path:

```
"C:\\Users\\Danus\\AppData\\Local\\Temp\\cruloader"]=
```

Then CreateDirectory is invoked to create the **cruloader** folder, after which the output.jpg file string is append to this path to create the following string:

```
C:\\Users\\Danus\\AppData\\Local\\Temp\\cruloader\\output.jpg"
```

CreateFileW is then invoked to create this file



Then the PNG file extracted from the previous website is copied into this file using **WriteFile**

```
00081514   FFB5 90FDFFFF   push dword ptr ss:[ebp-270]
0008151A   53              push ebx
0008151B   56              push esi
EIP 0008151C   FF95 8CFDFFFF   call dword ptr ss:[ebp-274]
00081522   56              push esi
00081523   FF15 24F00800   call dword ptr ds:[<&CloseHandle>]
00081529   66:A1 CC3C0900  mov ax,word ptr ds:[93CCC]
0008152F   8BF3            mov esi,ebx
00081531   F3:0F7E05 C43C0900   movq xmm0,qword ptr ds:[93CC4]
00081539   66:8945 F8      mov word ptr ss:[ebp-8],ax
0008153D   8D45 F0         lea eax,dword ptr ss:[ebp-10]
00081540   50              push eax

dword ptr [ebp-274]=[0049F794 <&WriteFile>]=<kernel32.WriteFile>

0008151C

Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | Locals | Struct

Address   Hex                                          ASCII
029B0000  89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52  .PNG........IHDR
029B0010  00 00 20 4B 00 00 12 2A 08 06 00 00 00 8D 3B A4  .. K...*......;¤
029B0020  25 00 00 00 09 70 48 59 73 00 00 2F AD 00 00 2F  %....pHYs../.../
029B0030  AD 01 5E AA 9E 98 00 00 00 19 74 45 58 74 53 6F  ..^ª......tEXtSo
029B0040  66 74 77 61 72 65 00 41 64 6F 62 65 20 49 6D 61  ftware.Adobe Ima
029B0050  67 65 52 65 61 64 79 71 C9 65 3C 00 04 35 36 49  geReadyqÉe<..56I
029B0060  44 41 54 78 DA EC D0 41 01 80 00 10 04 21 9D FE  DATxÚìÐA.....!.þ
029B0070  9D CF 0A EE 1F 22 F0 DE DD 03 00 00 00 00 00 00  .Ï.î."ðÞÝ.......
```

Afterwards the sample attempts to locate a string "**redaolurc**" (which is **cruloader** reversed) within the image data download

```
00041100  00 00 00 00 00 00 00 72 65 64 61 6F 6C 75 72 63  .......redaolurc
00041110  2C 3B F1 61 62 61 61 61 65 61 61 61 9E 9E 61 61  ,;ñabaaaeaaažžaa
00041120  D9 61 61 61 61 61 61 61 21 61 61 61 61 61 61 61  Ùaaaaaaa!aaaaaaa
00041130  61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
00041140  61 61 61 61 61 61 61 61 61 61 61 61 61 60 61 61  aaaaaaaaaaaaa`aa
00041150  6F 7E DB 6F 61 D5 68 AC 40 D9 60 2D AC 40 35 09  o~ÛoaÕh¬@Ù`-¬@5.
00041160  08 12 41 11 13 0E 06 13 00 0C 41 02 00 0F 0F 0E  ..A.......A.....
00041170  15 41 03 04 41 13 14 0F 41 08 0F 41 25 2E 32 41  .A..A...A..A%.2A
00041180  0C 0E 05 04 4F 6C 6C 6B 45 61 61 61 61 61 61 61  ....OllkEaaaaaaa
00041190  65 8B D8 31 21 EA B6 62 21 EA B6 62 21 EA B6 62  e‹Ø1!ê¶b!ê¶b!ê¶b
000411A0  28 92 25 62 2B EA B6 62 2A 85 B7 63 23 EA B6 62  ('%b+ê¶b*…·c#ê¶b
000411B0  2A 85 B3 63 32 EA B6 62 2A 85 B2 63 2D EA B6 62  *…³c2ê¶b*…²c-ê¶b
```

After locating the string within the image data, this offset is used to access encrypted data. The data is **xored** with **xmmword(**which is 128-bit) **40 times**. The xor key is **0x61**('a'). One could also notice that there are a lot of 'a' characters within the encoded payload. These 'a' characters are actually zeroes within the binary, because 0 ^ 0x61 = 0x61 so this payload isn't obfuscated with high class obfuscation as on could infer this pretty quickly.

```
xmmword_413CD0  xmmword 6161616161616161616161616161616161h
```

```
loc_4015B4:
movups  xmm0, xmmword ptr [eax-20h]
lea     eax, [eax+40h]
add     ecx, 40h
movaps  xmm1, xmm2
pxor    xmm1, xmm0
movups  xmmword ptr [eax-60h], xmm1
movups  xmm0, xmmword ptr [eax-50h]
pxor    xmm0, xmm2
movups  xmmword ptr [eax-50h], xmm0
movups  xmm0, xmmword ptr [eax-40h]
pxor    xmm0, xmm2
movups  xmmword ptr [eax-40h], xmm0
movups  xmm0, xmmword ptr [eax-30h]
pxor    xmm0, xmm2
movups  xmmword ptr [eax-30h], xmm0
cmp     ecx, esi
jb      short loc_4015B4
```

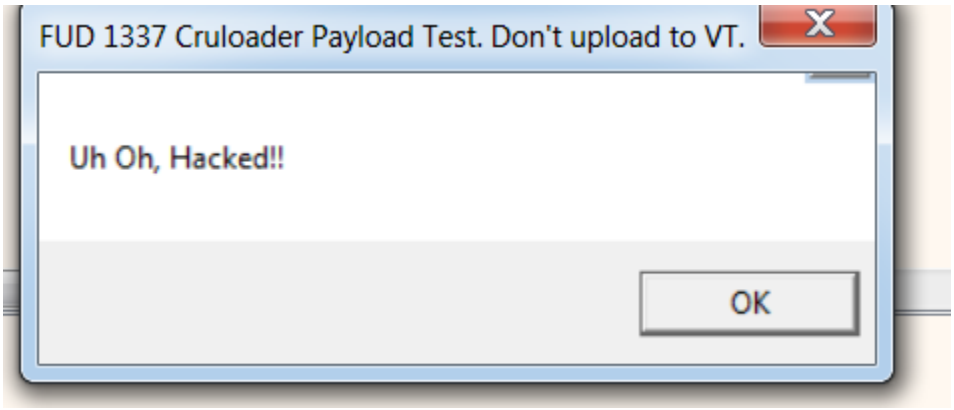
And the result is a valid PE file:

```
02E71110 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ..........ÿÿ..
02E71120 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ¸.......@.......
02E71130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
02E71140 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 ................
02E71150 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..º..´.Í!¸.LÍ!Th
02E71160 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
02E71170 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS 
02E71180 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode....$.......
02E71190 65 8B D8 31 21 EA B6 62 21 EA B6 62 21 EA B6 62 e.Ø1!ê¶b!ê¶b!ê¶b
02E711A0 28 92 25 62 2B EA B6 62 2A 85 B7 63 23 EA B6 62 (.%b+ê¶b*.·c#ê¶b
02E711B0 2A 85 B3 63 32 EA B6 62 2A 85 B2 63 2D EA B6 62 *.³c2ê¶b*.²c-ê¶b
02E711C0 2A 85 B5 63 20 EA B6 62 35 81 B7 63 24 EA B6 62 *.µc ê¶b5.·c$ê¶b
02E711D0 21 EA B7 62 0E EA B6 62 E5 85 BE 63 23 EA B6 62 !ê·b.ê¶bå.¾c#ê¶b
```

After dumping this PE I've observed it within IDA and it would appear as if this is the final payload!

```
sub_401000 proc near
push    0               ; uType
push    offset Caption  ; "FUD 1337 Cruloader Payload Test. Don't "...
push    offset Text     ; "Uh Oh, Hacked!!"
push    0               ; hWnd
call    ds:MessageBoxA
xor     eax, eax
retn
sub_401000 endp
```

This PE Is then mapped, relocated and then fixed with **VirtualProtect** and finally injected into svchost.exe again within the function **sub_401750** this executing the final payload.



And that’s pretty much it!

Automation

Alright, Let's begin attempting to automate the process of extracting all payloads and dumping them on disk.

We begin with the resource section; this one is pretty easy.

```
00012E60 01 DD 0C 92 00 22 00 00 00 22 00 00 6B 6B 64 35 59 64 50 4D 32 34 56 42 58 6D 69 00 03 3C 65 A7
00012E80 EC 58 FB B6 93 E6 EC E7 89 00 00 27 72 20 65 29 DF DD F0 10 7B FA 3B E3 0A 52 20 9D 9B 6C 25 BA
00012EA0 4A EF 5B 08 D4 0E 77 F1 50 E3 08 9C 11 36 E8 E0 9F 82 BD F5 89 B8 96 52 50 9C D3 2C 6D 59 19 CE
00012EC0 D4 82 54 DA 8A 93 19 99 1C 21 A9 12 C5 2A 1B 4A AD A6 14 3E CE 37 98 B1 AD 47 25 3D C9 A6 09 98
```

First we begin looking for a string at offset 0x60 + 0xc from the begging of the resource section, and load the string which is 16 bytes in length.

```
##########################CONSTANTS##########################

#First Payload Settings
RESOURCE_START = 0x60
KEY_START = RESOURCE_START + 0xc
FIRST_KEY_LEN = 0xf
```

Then, we load the rest of the payload into another variable

```
def resource_load():


    #Load the PE file into pe
    pe = pefile.PE(r"C:\\Users\\user\\Desktop\\Analysis\\Zero2Auto\\Lesson 3\\Final Test\\First stage\\main_bin.bin")

    section_addr = None

    #Find the resource section
    for section in pe.sections:
        if(".rsrc" in section.Name.decode()):
            section_addr = section
            break


    data = section_addr.get_data()
    print("Physical Location:", hex(section_addr.PointerToRawData))

    #Locate the RC4 Key
    key = data[KEY_START:KEY_START+FIRST_KEY_LEN]

    #Load the cipher text
    cipher_text = data[KEY_START+FIRST_KEY_LEN+1:-1]

    return key, cipher_text
```

Then we use the ARC4 python module to decrypt this data and dump it on disk

```python
def main():
    key, cipher_text = resource_load()

    print("Decrypting RC4 Encrypted first stage...")

    arc4_key = ARC4(key)
    string_decrypt = arc4_key.decrypt(cipher_text)

    print("Loading Second Stage PE")
    with open('second_stage.bin', 'wb') as file_out:
        file_out.write(string_decrypt)
```

Now as we posses the second payload, we must locate the pastebin URL inside the PE file.

```
.rdata:00413C50 aCruloader      db 'cruloader',0        ; DATA XREF: func_ReadWebContents+45↑o
.rdata:00413C5A                 db 0
.rdata:00413C5B                 db    0
.rdata:00413C5C xmmword_413C5C  xmmword 7C6D1DBD1FEF1D5DDC6CCCBC5FEF891Eh
.rdata:00413C5C                                         ; DATA XREF: func_StartSvchost+33↑r
.rdata:00413C6C xmmword_413C6C  xmmword 7CAD7CC86D1DDCAC1C4D1DEF0919FCh
.rdata:00413C6C                                         ; DATA XREF: func_StartSvchost+4B↑r
.rdata:00413C7C xmm_string      xmmword 0BACA7A0A1B6B4A5BAEAEFF6B5B1B1BDAh
.rdata:00413C7C                                         ; DATA XREF: sub_401DC0+6E↑r
.rdata:00413C8C xmm_site        xmmword 2818CF8A0A988AAE2B4A7BAE8AAA6ABEh
```

We know our URL is located **two XMM_WORDS** (32 bytes) in size after the offset of the string "**cruloader**" so let's set this up:

```python
#Second Payload Settings
CRU_STRING_BIN = b"cruloader"
XMM_WORD_SIZE = 16
XOR_KEY_CHAR = 0xc5
```

```python
#Load the PE file into pe
pe = pefile.PE(r"C:\\Users\\user\\Desktop\\Analysis\\Zero2Auto\\Lesson 3\\Final Test\\Automation\\second_stage.bin")
section_addr = None

#Locate the rdata section
for section in pe.sections:
    if(".rdata" in section.Name.decode()):
        section_addr = section
        break

print("Physical Location:", hex(section_addr.PointerToRawData))

data = section_addr.get_data()

#Locate encrypted website offset by finding "CRULOADER" in string inside PE
#CRU_STRING_BIN_offset + len(CRU_STRING_BIN) + 3 + XMM_WORD_SIZE * 2 = encrypted string
string_offset = (data.find(CRU_STRING_BIN)) + len(CRU_STRING_BIN) + 3 + XMM_WORD_SIZE*2

#Load encrypted website string
string_encrypted = data[string_offset:string_offset+XMM_WORD_SIZE*2+1]
string_decrypted = ""
```

We calculate the offset by locating the "**cruloader**" string, adding 3 bytes to skip the null bytes and then jumping after both irrelevant XMM_WORDS. We extract our data and we set the XOR key to **0xC5**

Then for each byte extracted we perform a four ROL and then a xor to match the decryption algorithm

```
loc_401E60:
mov     dl, [ebp+ecx-40h]
rol     dl, 4
xor     dl, 0C5h
mov     [ebp+ecx-40h], dl
inc     ecx
cmp     ecx, eax
jl      short loc_401E60
```

```
#For each byte within the encryted string perform decryption
for byte in string_encrypted:
    byte = swap_int(byte) ^ XOR_KEY_CHAR
    string_decrypted = string_decrypted + chr(byte)
```

We then use URLLIB to extract the **pastebin** URL and then we use that same lib to extract the contents of the payload image

```
#Get the URL of the payload PNG file
with urllib.request.urlopen(string_decrypted) as web_content:
    file_webpage_png = web_content.read().decode('utf-8')

#Get the contents of the PNG file
with urllib.request.urlopen(file_webpage_png) as web_content:
    file_content_png_payload = web_content.read()
```

Finally, to locate the payload within the PNG payload file we locate the reverse "**cruloader**" string, extract the payload and then xor it with 0x61.

```
#Locate the payload within the PNG file
payload_offset = file_content_png_payload.find(FINAL_PAYLOAD_STRING_LOC) + len(FINAL_PAYLOAD_STRING_LOC)
payload_data = file_content_png_payload[payload_offset:-1]

payload_data_decrypted = b''

#Decrypt the payload
for byte in payload_data:
    byte = byte ^ PAYLOAD_XOR_KEY
    payload_data_decrypted = payload_data_decrypted + bytes([byte])

return payload_data_decrypted
```

And that's it! Easy as that!

I really enjoyed this challenge and I'm looking forward to continuing the course 😊
Hope you enjoyed reading this!