CruLoader: Zero2Auto

malwarebookreports.com/cruloader-zero2auto/

muzi July 8, 2022

Taking a break from my normal blog posts to complete the practical analysis from the Zero2Automated course from Vitali Kremez and Daniel Bunce.

Assignment Background

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 exfiltrated. 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.

Stage 1 Exe

```
Filename: main_bin.exe
MD5: a84e1256111e4e235250a8e3bb11f903
SHA1: 1b76e5a645a0df61bb4569d54bd1183ab451c95e
SHA256: a0ac02a1e6c908b90173e86c3e321f2bab082ed45236503a21eb7d984de10611
```

Stepping into the main function, several obfuscated strings are moved into registers and a function call is made immediately afterwards. Next, LoadLibrary and GetProcAddress are called, indicating that these obfuscated strings are almost certainly obfuscated libraries to import.

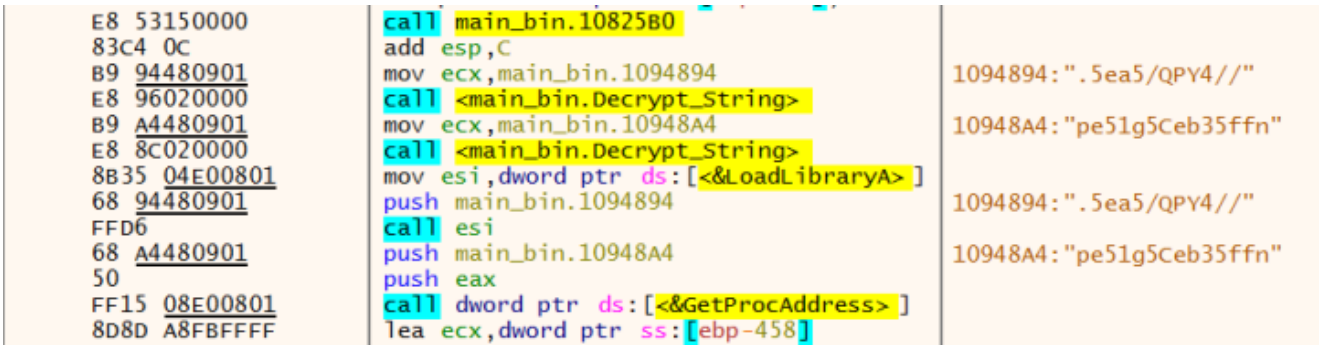


Figure 1: Obfuscated Library Imports

String Decryption/Decoding

Stepping into the function called right after moving the obfuscated strings, a few things stand out that indicate the purpose of it:

- The long string of characters that appear to be an extended/custom alphabet
- The `add edx, D` instruction (ROT-13 anyone?)

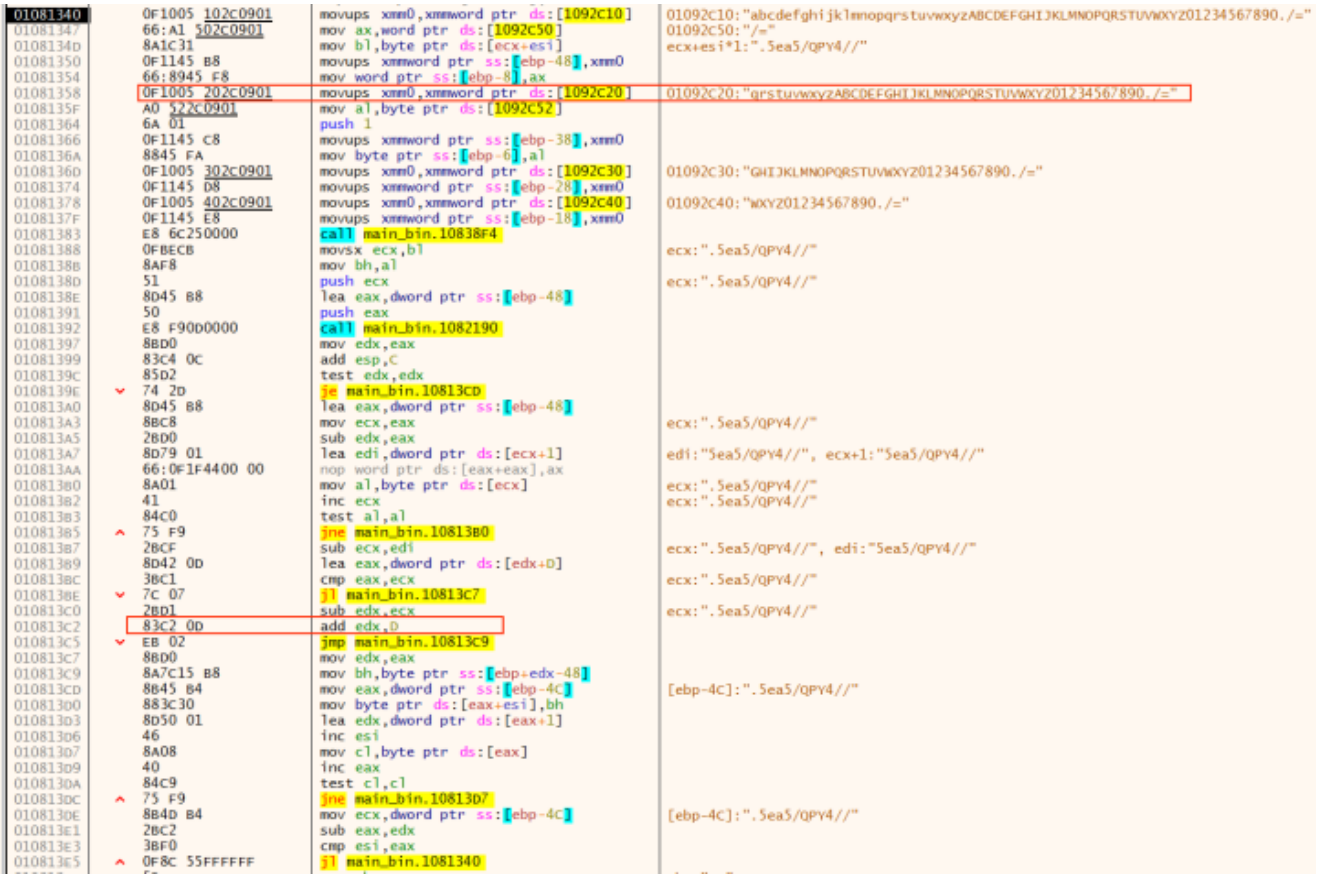


Figure 2: ROT-13 Decryption/Deobfuscation Routine

After returning from the decryption function, `kernel32.dll` is decoded. Now that the decryption/decode function has been identified, a string decoder can be written. String Decrypter/Decoder (and unpacker) can be found on my GitHub.

```
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: I9egh1/n//b3 -
-> Decrypted: VirtualAlloc
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted:
t5gG8e514pbag5kg --> Decrypted: GetThreadContext
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: yb3.E5fbhe35 -
-> Decrypted: LockResource
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: pe51g5Ceb35ffn
--> Decrypted: CreateProcessA
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted:
F5gG8e514pbag5kg --> Decrypted: SetThreadContext
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: yb14E5fbhe35 -
-> Decrypted: LoadResource
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted:
E514Ceb35ffz5=bel --> Decrypted: ReadProcessMemory
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: s9a4E5fbhe35n
--> Decrypted: FindResourceA
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted:
Je9g5Ceb35ffz5=bel --> Decrypted: WriteProcessMemory
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: I9egh1/n//b3rk
--> Decrypted: VirtualAllocEx
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: F9m5b6E5fbhe35
--> Decrypted: SizeofResource
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: .5ea5/QPY4// -
-> Decrypted: kernel32.dll
2022-05-30 22:06:05,424 - CruLoader Unpacker - CRITICAL [*] Encrypted: E5fh=5G8e514 -
-> Decrypted: ResumeThread
```

Now that the strings are decrypted, some additional functionality becomes apparent: packed code located in a resource and process injection (process hollowing). The executable being analyzed looks to be a crypter that will unpack the code contained in the RCDATA resource.

RC4 Decrypt Resource

Loading the executable into PE Studio, the `RCDATA` resource stands out – it is large in size at 87068 bytes and the 7.98 entropy indicates this is most likely encrypted data. In addition, the decrypted/deobfuscated strings above lend credibility to this theory, as the following libraries are used to access the resource:

- FindResourceA
- LoadResource
- SizeOfResource
- LockResource

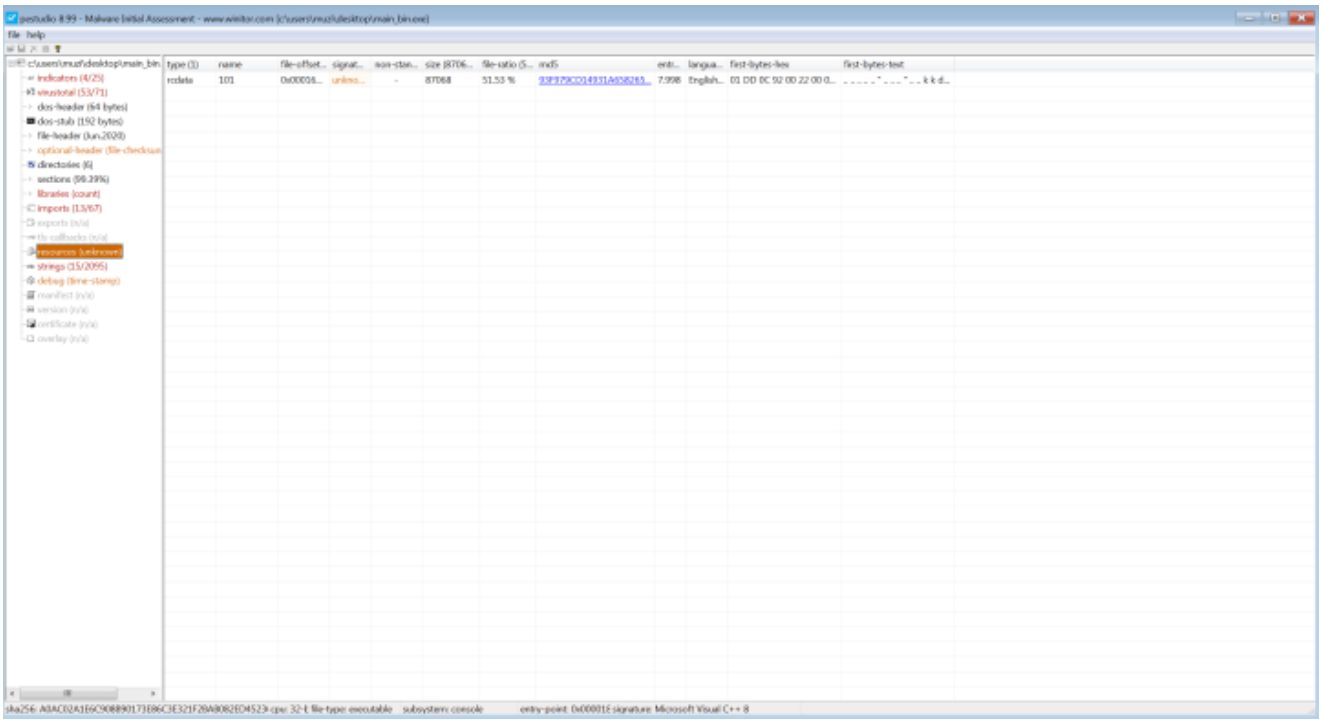
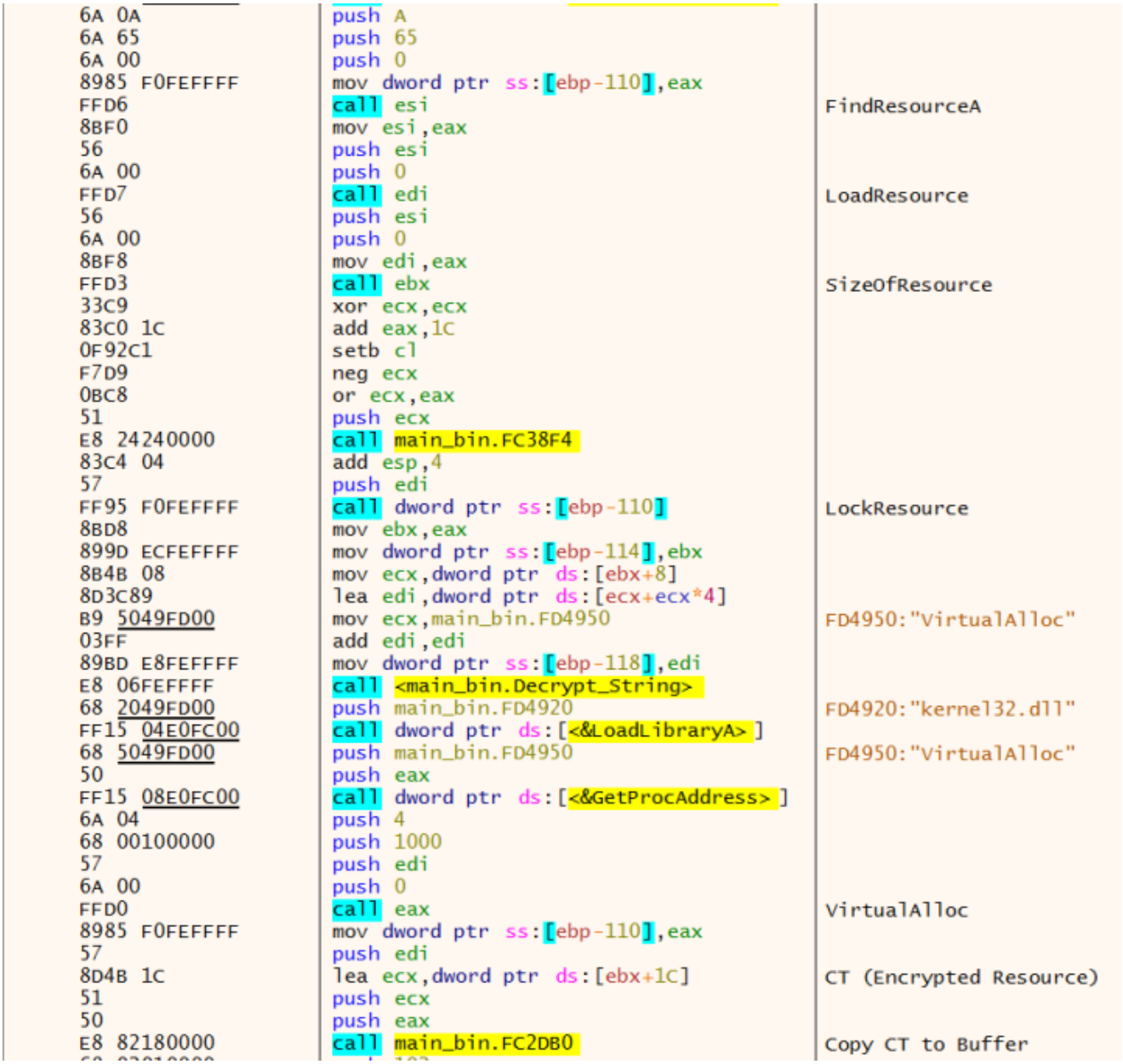


Figure 3: Resource (RCDATA)

Following in the debugger, the `RCDATA` resource is loaded, then, starting at position 0x1C (28 decimal), the ciphertext is copied into an allocated buffer.



```
6A 0A           push A
6A 65           push 65
6A 00           push 0
8985 F0FEFFFF   mov dword ptr ss:[ebp-110],eax
FFD6            call esi                        FindResourceA
8BF0            mov esi,eax
56              push esi
6A 00           push 0
FFD7            call edi                        LoadResource
56              push esi
6A 00           push 0
8BF8            mov edi,eax
FFD3            call ebx                        SizeOfResource
33C9            xor ecx,ecx
83C0 1C         add eax,1C
0F92C1          setb cl
F7D9            neg ecx
0BC8            or ecx,eax
51              push ecx
E8 24240000     call main_bin.FC38F4
83C4 04         add esp,4
57              push edi
FF95 F0FEFFFF   call dword ptr ss:[ebp-110]     LockResource
8BD8            mov ebx,eax
899D ECFEFFFF   mov dword ptr ss:[ebp-114],ebx
8B4B 08         mov ecx,dword ptr ds:[ebx+8]
8D3C89          lea edi,dword ptr ds:[ecx+ecx*4]
B9 5049FD00     mov ecx,main_bin.FD4950         FD4950:"VirtualAlloc"
03FF            add edi,edi
89BD E8FEFFFF   mov dword ptr ss:[ebp-118],edi
E8 06FEFFFF     call <main_bin.Decrypt_String>
68 2049FD00     push main_bin.FD4920            FD4920:"kernel32.dll"
FF15 04E0FC00   call dword ptr ds:[<&LoadLibraryA>]
68 5049FD00     push main_bin.FD4950            FD4950:"VirtualAlloc"
50              push eax
FF15 08E0FC00   call dword ptr ds:[<&GetProcAddress>]
6A 04           push 4
68 00100000     push 1000
57              push edi
6A 00           push 0
FFD0            call eax                        VirtualAlloc
8985 F0FEFFFF   mov dword ptr ss:[ebp-110],eax
57              push edi
8D4B 1C         lea ecx,dword ptr ds:[ebx+1C]   CT (Encrypted Resource)
51              push ecx
50              push eax
E8 82180000     call main_bin.FC2DB0            Copy CT to Buffer
```

Figure 4: Resource Loaded and Copied into Allocated Memory

Next, the ciphertext decrypted using RC4. The key starts at offset 0xC (12) of the resource and is 16 bytes long.

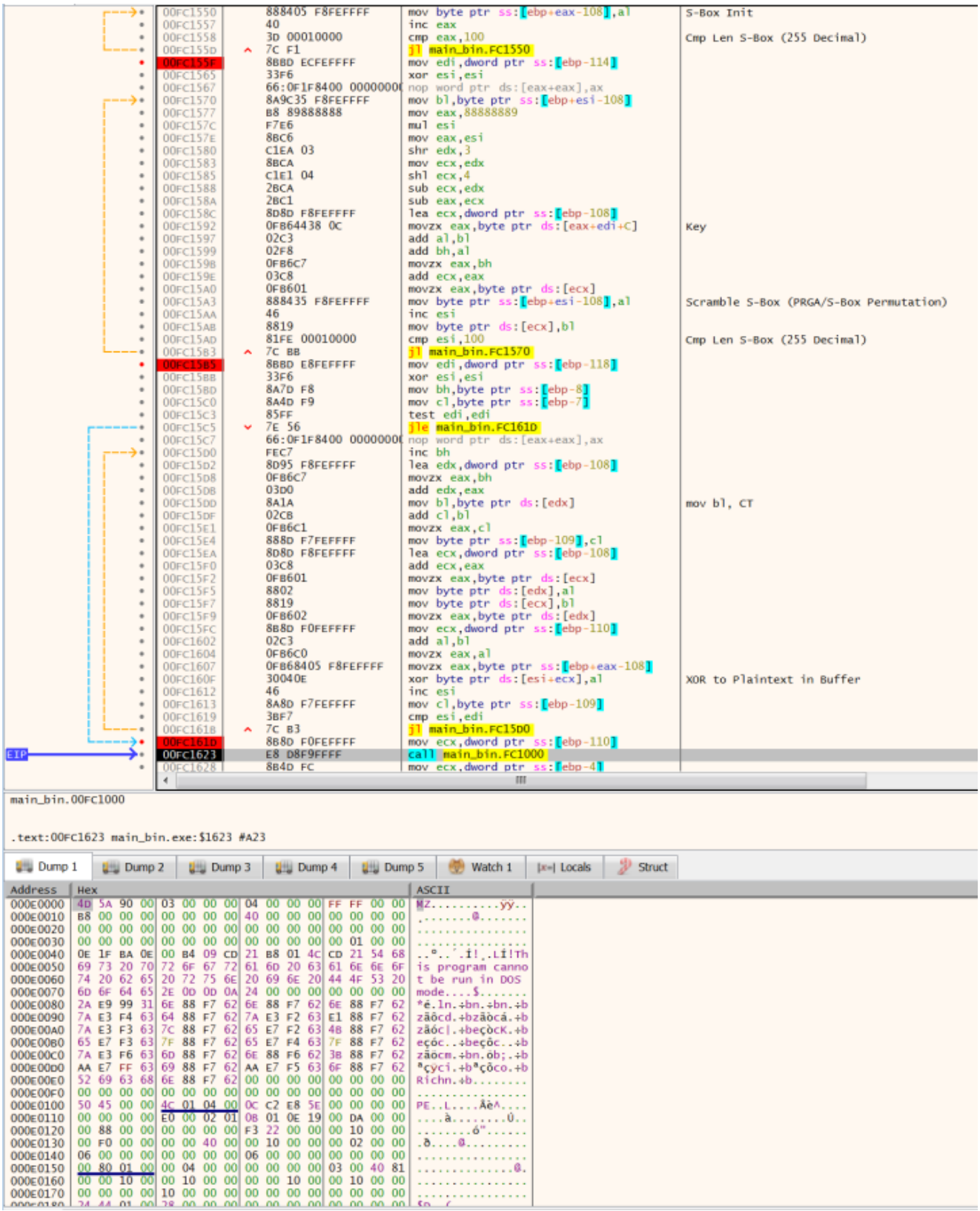


Figure 5: RC4 Decryption Routine Unveils Decrypted Executable

After decrypting the executable, it is loaded and executed via process injection (process hollowing) with the following calls:

- CreateProcessA
- VirtualAlloc

- GetThreadContext
- ReadProcessMemory
- VirtualAllocEx
- WriteProcessMemory
- SetThreadContext
- ResumeThread

Once ResumeThread is called, execution is transferred to the new, decrypted executable.

Stage 2 Exe

```
Filename: cruloader
MD5: f56a2fd3fd94be87f5c79e822734168d
SHA1: 191050c7ae62c5665938f7666519bcd94f784fd5
SHA256: a4997fbff9bf2ebfee03b9373655a45d4ec3b1bcee6a05784fe4e022e471e8e7
```

Jumping straight into main, the malware first computes the CRC32 of its filename, then checks it against the CRC32 hash `0xB925C42D`, which corresponds to `svchost.exe`. The CRC32 algorithm can be easily identified by `0xedb88320`, which is the so-called “reversed” representation of the CRC32 generator polynomial. Luckily for us, `0xB925C42D` aka “svchost.exe” is included in the following list of CRC32 API hashes. Lists of API hashes such as this are commonly found on GitHub and can be found simply by Googling for the specific API Hash/constant.

```
                    uint __fastcall CRC32_Hash(undefined4 param_1, byte * ...
     uint               EAX:4          <RETURN>
     undefined4         ECX:4          param_1
     byte *             EDX:4          param_2
     int                Stack[0x4]:4   param_3                          XR
                    CRC32_Hash                                      XREF[2]:

00401660 55             PUSH        EBP
00401661 8b ec          MOV         EBP,ESP
00401663 51             PUSH        param_1
00401664 83 3d 8c       CMP         dword ptr [DAT_0041628c],0x0
         62 41 00
         00
0040166b 56             PUSH        ESI
0040166c 57             PUSH        EDI
0040166d 8b f2          MOV         ESI,param_2
0040166f 0f 85 b0       JNZ         LAB_00401725
         00 00 00
00401675 33 ff          XOR         EDI,EDI

                    LAB_00401677                                    XREF[1]:
00401677 8b cf          MOV         param_1,EDI
00401679 8b c7          MOV         EAX,EDI
0040167b d1 e9          SHR         param_1,1
0040167d 8b d1          MOV         param_2,param_1
0040167f 81 f2 20       XOR         param_2,0xedb88320
         83 b8 ed
```

Figure 6: Compute CRC32 Hash of Filename

Depending on if the filename is `svchost.exe`, the code will take one of two branches. Branch one, where the filename is not `svchost.exe` will be examined first, followed by branch two.

Branch One: Filename != svchost.exe

If the filename is not svchost.exe, Cruloader begins an anti analysis routine to identify if it is being analyzed. First, it resolves the API `IsDebuggerPresent` to detect an attached debugger. Next, it resolves `CreateToolhelp32Snapshot`, `Process32FirstW`, and `Process32NextW` in order to compute the CRC32 checksum of every running process to compare against the following analysis tools:

- `7C6FFE70` (processhacker.exe)
- `47742A22` (wireshark.exe)
- `D2F05B7D` (x32dbg.exe)
- `659B537E` (x64dbg.exe)

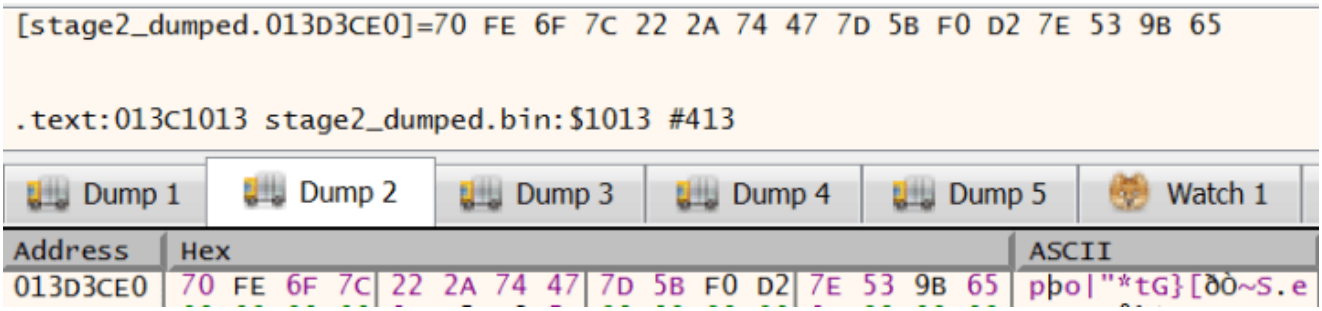


Figure 7: Pre-calculated Hashes to Check Running Processes Against

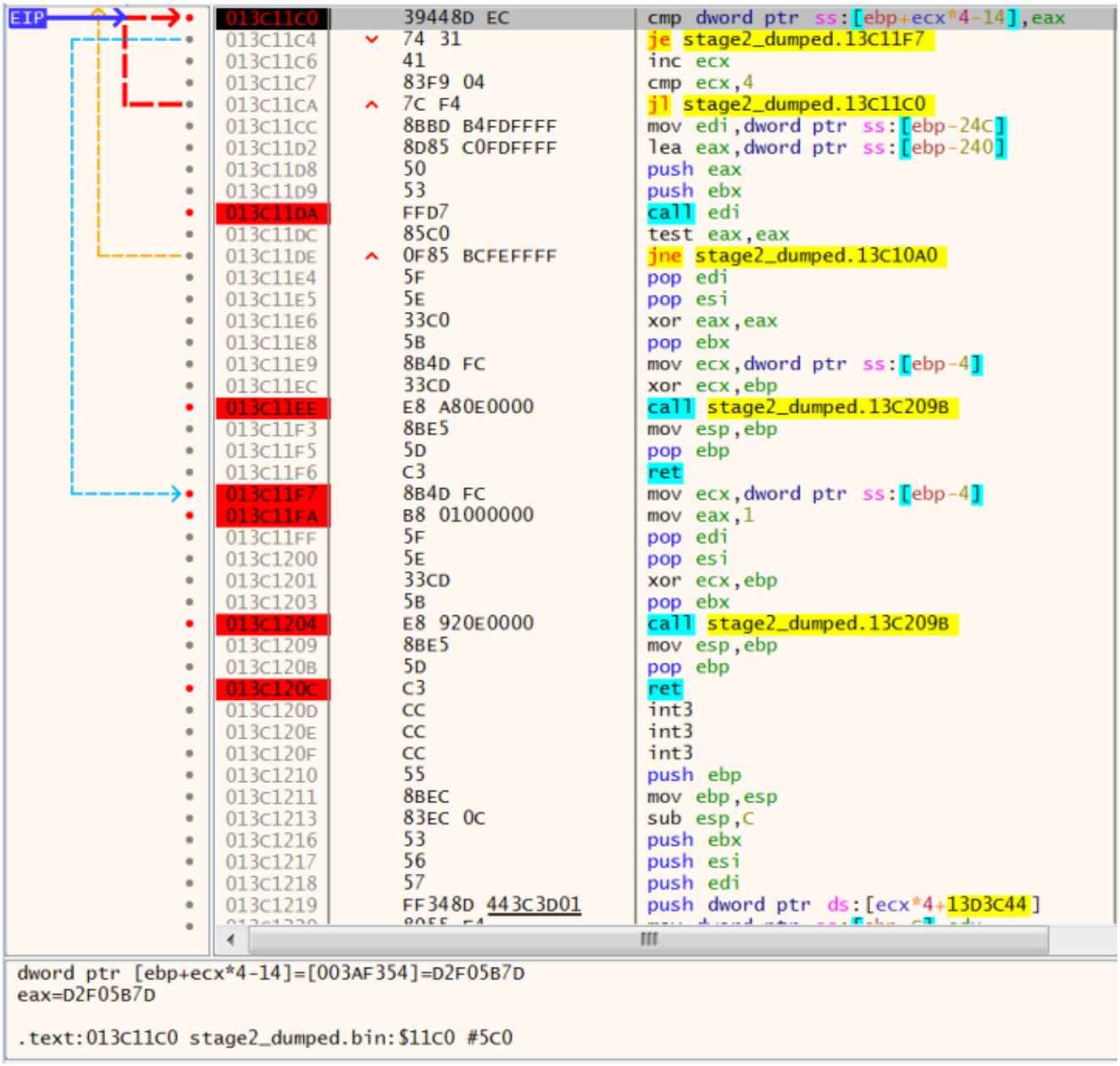


Figure 8: Example Match for x32dbg.exe

If any of the tools listed above are discovered running, the malware exits immediately. After determining that it is not being analyzed/debugged, the malware next resolves the same APIs used previously for process injection.

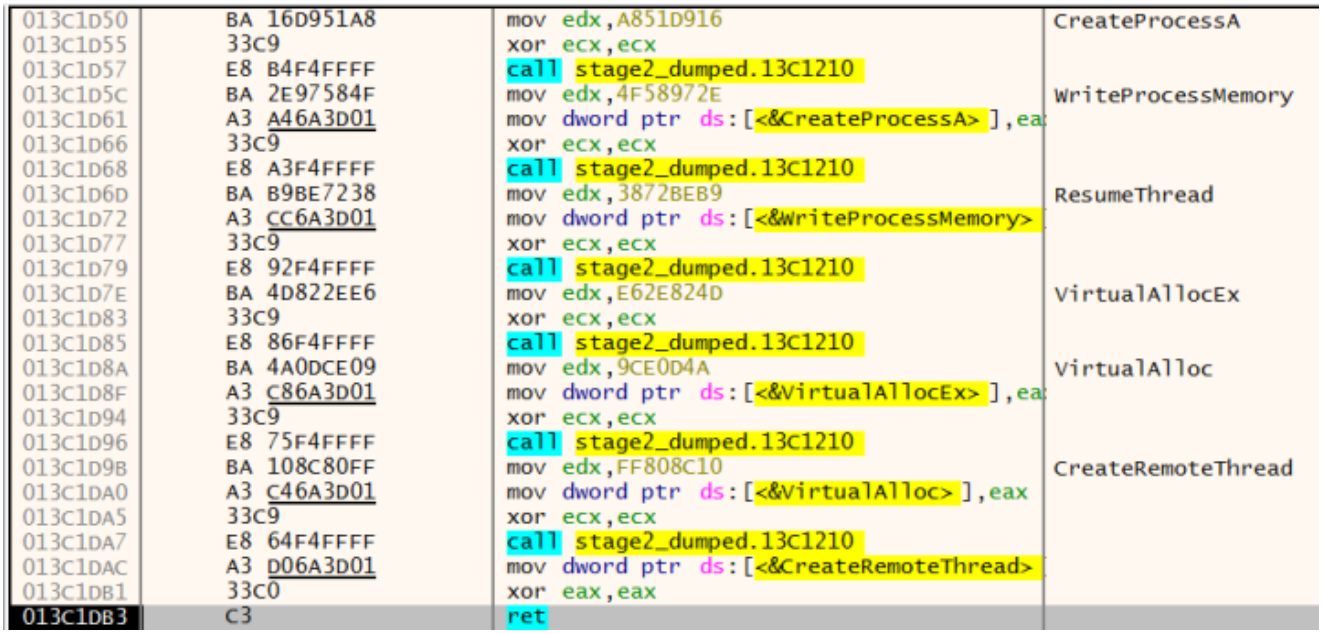


Figure 9: Resolving APIs for Process Injection

After resolving the APIs, the malware decrypts the string `C:\Windows\System32\svchost.exe`, then creates a suspended `svchost.exe` process.

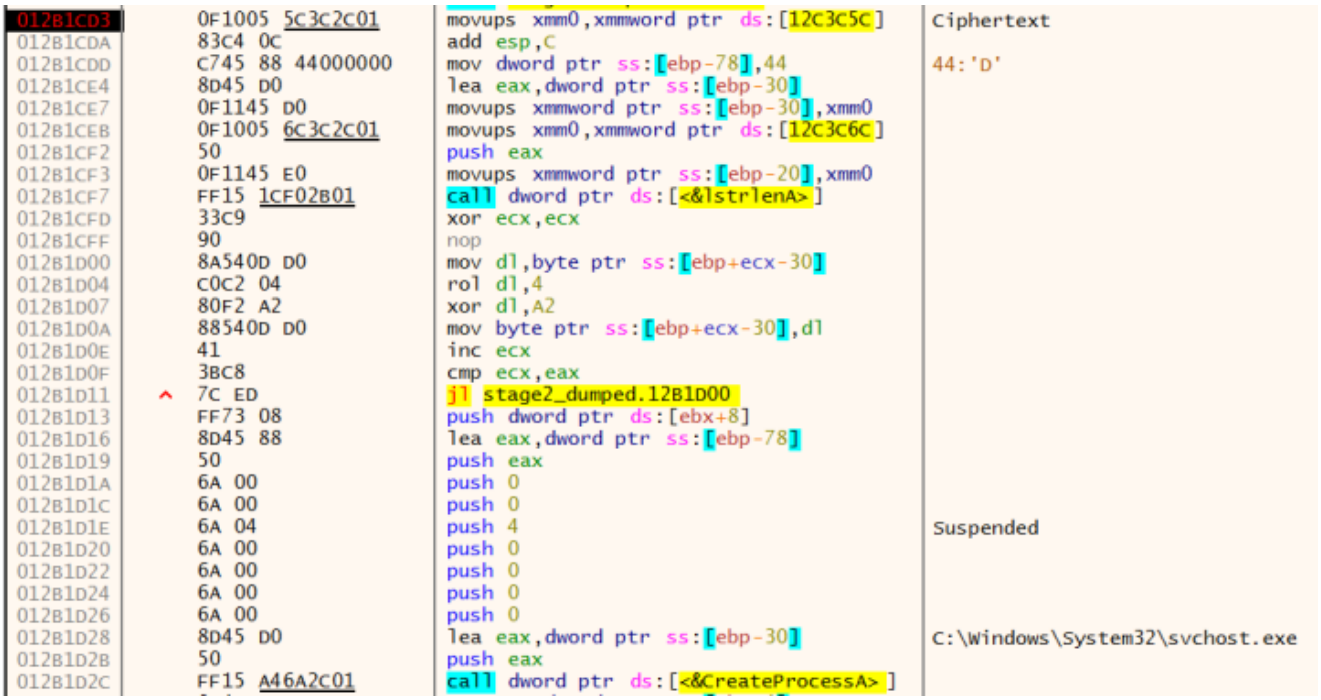


Figure 10: Decrypt String and Create Suspended Svchost Process

Next, the malware calls VirtualAlloc to allocate memory inside the running process to copy itself into. It then calls VirtualAllocEx to allocate a RWX region inside the new suspended process and uses WriteProcessMemory to write the copy of itself into the new process. Finally, it calls CreateRemoteThread to execute the code injected into `svchost.exe`.

Branch One: Filename == Svchost.exe

Now that CruLoader is running under `svchost.exe` (whether by changing the name or letting it inject into `svchost.exe`), the malware will take the other branch. This branch begins by resolving a few APIs for internet activity.

```
BA 3DA816DA      mov edx,DA16A83D
B9 02000000      mov ecx,2
E8 1EF4FFFF      call svchost.1371210
BA E0056501      mov edx,16505E0
A3 B86A3801      mov dword ptr ds:[<&InternetOpenA>],eax
B9 02000000      mov ecx,2
E8 0AF4FFFF      call svchost.1371210
BA F598C06C      mov edx,6CC098F5
A3 D86A3801      mov dword ptr ds:[<&InternetOpenUrlA>],
B9 02000000      mov ecx,2
E8 F6F3FFFF      call svchost.1371210
BA 241D19E5      mov edx,E5191D24
A3 BC6A3801      mov dword ptr ds:[<&InternetReadFile>],
B9 02000000      mov ecx,2
E8 E2F3FFFF      call svchost.1371210
```

Figure 11: Resolve wininet APIs

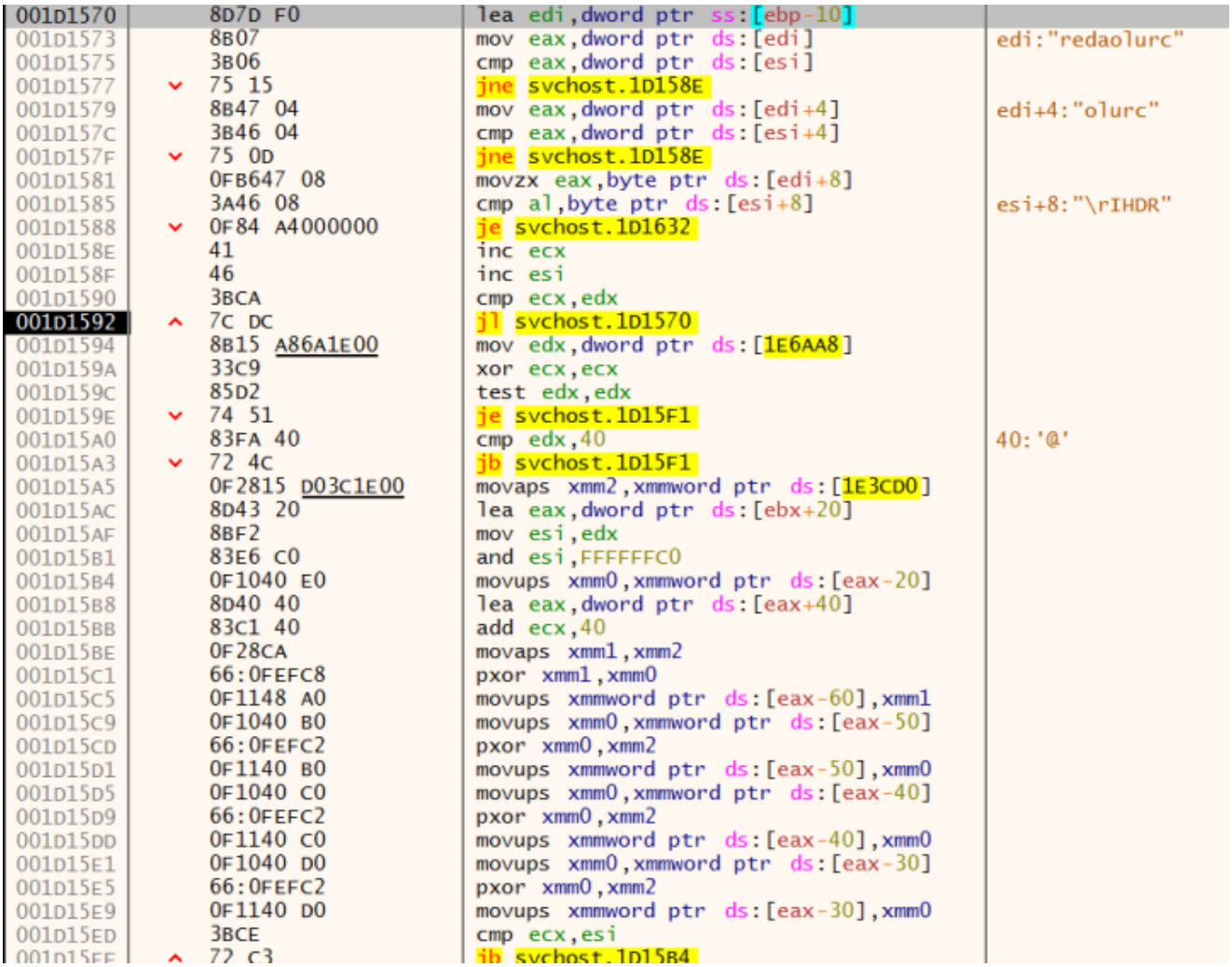
Next, the URL configuration is decrypted with a simple `rol` and `xor`.

```
01371E3A   66:A1 9C3C3801     mov ax,word ptr ds:[1383C9C]
01371E40   0F1145 C0          movups xmmword ptr ss:[ebp-40],xmm0
01371E44   66:8945 E0         mov word ptr ss:[ebp-20],ax
01371E48   8D45 C0            lea eax,dword ptr ss:[ebp-40]
01371E4B   0F1005 8C3C3801    movups xmm0,xmmword ptr ds:[1383C8C]
01371E52   50                 push eax
01371E53   0F1145 D0          movups xmmword ptr ss:[ebp-30],xmm0
01371E57   FF15 1CF03701      call dword ptr ds:[<&lstrlenA>]
01371E5D   33C9               xor ecx,ecx
01371E5F   90                 nop
01371E60   8A540D C0          mov dl,byte ptr ss:[ebp+ecx-40]
01371E64   C0C2 04            rol dl,4
01371E67   80F2 C5            xor dl,C5
01371E6A   88540D C0          mov byte ptr ss:[ebp+ecx-40],dl
01371E6E   41                 inc ecx
01371E6F   3BC8               cmp ecx,eax
01371E71   7C ED              jl svchost.1371E60
```

Figure 12: Decrypt URL Config

After decrypting the Pastebin URL, the malware makes a connection to the URL and receives a second URL back. It then makes another request to download a PNG. (Note: User-Agent of `CruLoader` could be used for detection. This kind of thing used to be more popular in the early 2010s, but Bumblebee Malware did this just last year.) The PNG is then written to the following path `C:\Users\USER\AppData\Local\Temp\cruloader\output.jpg`.

Next, CruLoader decrypts another string `redaolurc`. It then searches for this payload marker inside the PNG file.



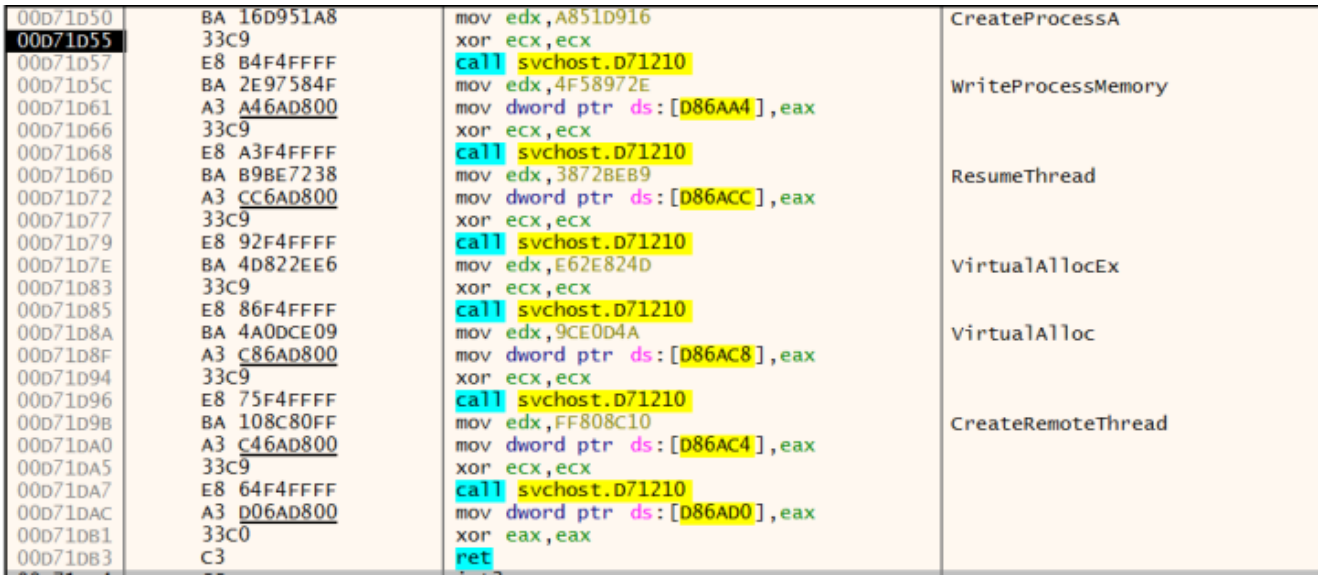
```
001D1570    8D7D F0          lea edi,dword ptr ss:[ebp-10]
001D1573    8B07             mov eax,dword ptr ds:[edi]          edi:"redaolurc"
001D1575    3B06             cmp eax,dword ptr ds:[esi]
001D1577    75 15            jne svchost.1D158E
001D1579    8B47 04          mov eax,dword ptr ds:[edi+4]        edi+4:"olurc"
001D157C    3B46 04          cmp eax,dword ptr ds:[esi+4]
001D157F    75 0D            jne svchost.1D158E
001D1581    0FB647 08        movzx eax,byte ptr ds:[edi+8]
001D1585    3A46 08          cmp al,byte ptr ds:[esi+8]          esi+8:"\rIHDR"
001D1588    0F84 A4000000    je svchost.1D1632
001D158E    41               inc ecx
001D158F    46               inc esi
001D1590    3BCA             cmp ecx,edx
001D1592    7C DC            jl svchost.1D1570
001D1594    8B15 A86A1E00    mov edx,dword ptr ds:[1E6AA8]
001D159A    33C9             xor ecx,ecx
001D159C    85D2             test edx,edx
001D159E    74 51            je svchost.1D15F1
001D15A0    83FA 40          cmp edx,40                          40:'@'
001D15A3    72 4C            jb svchost.1D15F1
001D15A5    0F2815 D03C1E00  movaps xmm2,xmmword ptr ds:[1E3CD0]
001D15AC    8D43 20          lea eax,dword ptr ds:[ebx+20]
001D15AF    8BF2             mov esi,edx
001D15B1    83E6 C0          and esi,FFFFFFC0
001D15B4    0F1040 E0        movups xmm0,xmmword ptr ds:[eax-20]
001D15B8    8D40 40          lea eax,dword ptr ds:[eax+40]
001D15BB    83C1 40          add ecx,40
001D15BE    0F28CA           movaps xmm1,xmm2
001D15C1    66:0FEFC8        pxor xmm1,xmm0
001D15C5    0F1148 A0        movups xmmword ptr ds:[eax-60],xmm1
001D15C9    0F1040 B0        movups xmm0,xmmword ptr ds:[eax-50]
001D15CD    66:0FEFC2        pxor xmm0,xmm2
001D15D1    0F1140 B0        movups xmmword ptr ds:[eax-50],xmm0
001D15D5    0F1040 C0        movups xmm0,xmmword ptr ds:[eax-40]
001D15D9    66:0FEFC2        pxor xmm0,xmm2
001D15DD    0F1140 C0        movups xmmword ptr ds:[eax-40],xmm0
001D15E1    0F1040 D0        movups xmm0,xmmword ptr ds:[eax-30]
001D15E5    66:0FEFC2        pxor xmm0,xmm2
001D15E9    0F1140 D0        movups xmmword ptr ds:[eax-30],xmm0
001D15ED    3BCE             cmp ecx,esi
001D15EF    72 C3            jb svchost.1D15B4
```

Figure 13: Search for Payload Marker ‘redaolurc’ in PNG

Once the payload marker is found, it XOR decrypts with the key 0x61 ‘a’.

Figure 14: Decrypted Payload

After decrypting the payload, the malware decrypts the string `C:\Windows\System32\svchost.exe` and resolves APIs to once again inject the final payload into svchost.exe.



```
00D71D50    BA 16D951A8     mov edx,A851D916                CreateProcessA
00D71D55    33C9            xor ecx,ecx
00D71D57    E8 B4F4FFFF     call svchost.D71210
00D71D5C    BA 2E97584F     mov edx,4F58972E                WriteProcessMemory
00D71D61    A3 A46AD800     mov dword ptr ds:[D86AA4],eax
00D71D66    33C9            xor ecx,ecx
00D71D68    E8 A3F4FFFF     call svchost.D71210
00D71D6D    BA B9BE7238     mov edx,3872BEB9                ResumeThread
00D71D72    A3 CC6AD800     mov dword ptr ds:[D86ACC],eax
00D71D77    33C9            xor ecx,ecx
00D71D79    E8 92F4FFFF     call svchost.D71210
00D71D7E    BA 4D822EE6     mov edx,E62E824D                VirtualAllocEx
00D71D83    33C9            xor ecx,ecx
00D71D85    E8 86F4FFFF     call svchost.D71210
00D71D8A    BA 4A0DCE09     mov edx,9CE0D4A                 VirtualAlloc
00D71D8F    A3 C86AD800     mov dword ptr ds:[D86AC8],eax
00D71D94    33C9            xor ecx,ecx
00D71D96    E8 75F4FFFF     call svchost.D71210
00D71D9B    BA 108C80FF     mov edx,FF808C10                CreateRemoteThread
00D71DA0    A3 C46AD800     mov dword ptr ds:[D86AC4],eax
00D71DA5    33C9            xor ecx,ecx
00D71DA7    E8 64F4FFFF     call svchost.D71210
00D71DAC    A3 D06AD800     mov dword ptr ds:[D86AD0],eax
00D71DB1    33C0            xor eax,eax
00D71DB3    C3              ret
```

Figure 15: Time to Inject into svchost.exe Again

Opening up the newly decrypted payload in Ghidra shows us that we have completed the challenge.

```
undefined4 FUN_00401000(void)

{
  MessageBoxA((HWND)0x0,"Uh Oh, Hacked!!","FUD 1337 Cruloader Payload Test. Don\'t upload to
  VT.",0)
  ;
  return 0;
}
```

Figure 16: Challenge Complete

Automation

Earlier I showcased the string decrypter and unpacker for stage one. Let's finish automating the config extraction and string decryption for stage two, along with the decryption of the final payload embedded in the PNG. Code available on GitHub. I got a bit lazy with my coding here at the end, but it does the job.

```
❯ python3 unpack.py -d -f main_bin.exe -o output.bin
2022-07-08 14:10:41,895 - CruLoader Unpacker - CRITICAL [*] Key is:
b'6b6b64355964504d32345642586d69'
2022-07-08 14:10:41,896 - CruLoader Unpacker - CRITICAL [*] Unpacking payload
2022-07-08 14:10:41,896 - CruLoader Unpacker - CRITICAL [*] Payload written to
output.bin
2022-07-08 14:10:41,899 - CruLoader Unpacker - CRITICAL [*] Encrypted: pe51g5Ceb35ffn
--> Decrypted: CreateProcessA
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted:
F5gG8e514pbag5kg --> Decrypted: SetThreadContext
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted: yb14E5fbhe35 -
-> Decrypted: LoadResource
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted:
Je9g5Ceb35ffz5=bel --> Decrypted: WriteProcessMemory
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted: .5ea5/QPY4// -
-> Decrypted: kernel32.dll
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted: I9egh1/n//b3rk
--> Decrypted: VirtualAllocEx
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted: F9m5b6E5fbhe35
--> Decrypted: SizeofResource
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted: yb3.E5fbhe35 -
-> Decrypted: LockResource
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted:
E514Ceb35ffz5=bel --> Decrypted: ReadProcessMemory
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted: I9egh1/n//b3 -
-> Decrypted: VirtualAlloc
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted:
t5gG8e514pbag5kg --> Decrypted: GetThreadContext
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted: s9a4E5fbhe35n
--> Decrypted: FindResourceA
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted: E5fh=5G8e514 -
-> Decrypted: ResumeThread

❯ python3 extract_config_decrypt_strings.py -f stage2_bin.exe
[*] Config URL(s): ['hxxps://pastebin[.]com/raw/mLem9DGk']
[*] Decrypted strings: ['\\output.jpg', 'redaolurc',
'C:\\Windows\\System32\\svchost.exe']

❯ python3 extract_payload_from_png.py -f cruloaderpng.png -o final_extracted.bin
2022-07-08 14:04:57,794 - CruLoader Unpacker - CRITICAL [*] Payload written to
final_extracted.bin
```