

CruLoader: Zero2Auto

 malwarebookreports.com/cruloder-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.

E8 53150000	call main_bin.10825B0	
83C4 0C	add esp,C	
B9 94480901	mov ecx,main_bin.1094894	1094894:".5ea5/QPY4//"
E8 96020000	call <main_bin.Decrypt_String>	
B9 A4480901	mov ecx,main_bin.10948A4	10948A4:"pe5lg5Ceb35ffn"
E8 8C020000	call <main_bin.Decrypt_String>	
8B35 04E00801	mov esi,dword ptr ds:[<&LoadLibraryA>]	
68 94480901	push main_bin.1094894	1094894:".5ea5/QPY4//"
FFD6	call esi	
68 A4480901	push main_bin.10948A4	10948A4:"pe5lg5Ceb35ffn"
50	push eax	
FF15 08E00801	call dword ptr ds:[<&GetProcAddress>]	
8D8D A8FBFFFF	lea ecx,dword ptr ss:[ebp-458]	

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?)

01081340	0F1005 102C0901	movups xmm0,xmmword ptr ds:[1092C10]	01092C10:"abcde fghi jklmnopqrstuvwxy zABCDEFGHIJKLMN O PQRSTU VWXY Z01234567890./="
01081340	66:A1 502C0901	mov ax,word ptr ds:[1092C50]	01092C50:"/="
01081340	8A1C31	mov bl,byte ptr ds:[ecx+esi]	ecx+esi*1:".5ea5/QPY4//"
01081354	0F1145 B8	movups xmmword ptr ss:[ebp-48],xmm0	
01081354	66:8945 F8	mov word ptr ss:[ebp-8],ax	
01081358	0F1005 202C0901	movups xmm0,xmmword ptr ds:[1092C20]	01092C20:"qrstuvwxyzABCDEFGHIJKLMN O PQRSTU VWXY Z01234567890./="
0108135F	A0 522C0901	mov al,byte ptr ds:[1092C52]	
01081364	6A 01	push 1	
01081366	0F1145 C8	movups xmmword ptr ss:[ebp-38],xmm0	
0108136A	8845 FA	mov byte ptr ss:[ebp-6],al	
0108136D	0F1005 302C0901	movups xmm0,xmmword ptr ds:[1092C30]	01092C30:"GHIJKLMN O PQRSTU VWXY Z01234567890./="
01081374	0F1145 D8	movups xmmword ptr ss:[ebp-28],xmm0	
01081378	0F1005 402C0901	movups xmm0,xmmword ptr ds:[1092C40]	01092C40:"wxyz01234567890./="
0108137F	0F1145 E8	movups xmmword ptr ss:[ebp-18],xmm0	
01081383	E8 6c250000	call main_bin.10838F4	
01081388	0F8ECB	movsx ecx,bl	ecx:".5ea5/QPY4//"
0108138B	8AF8	mov bh,al	
0108138D	51	push ecx	ecx:".5ea5/QPY4//"
0108138E	8D45 B8	lea eax,dword ptr ss:[ebp-48]	
01081391	50	push eax	
01081392	E8 F90D0000	call main_bin.1082190	
01081397	8BD0	mov edx,eax	
01081399	83C4 0C	add esp,C	
0108139C	85D2	test edx,edx	
0108139E	74 2D	je main_bin.10813CD	
010813A0	8D45 B8	lea eax,dword ptr ss:[ebp-48]	
010813A3	8BC8	mov ecx,eax	ecx:".5ea5/QPY4//"
010813A5	2BD0	sub edx,eax	
010813A7	8D79 01	lea edi,dword ptr ds:[ecx+1]	edi:"5ea5/QPY4//", ecx+1:"5ea5/QPY4//"
010813AA	66:0F1F4400 00	nop word ptr ds:[eax+eax],ax	
010813B0	8A01	mov al,byte ptr ds:[ecx]	ecx:".5ea5/QPY4//"
010813B2	41	inc ecx	ecx:".5ea5/QPY4//"
010813B3	84C0	test al,al	
010813B5	75 F9	jne main_bin.10813B0	
010813B7	2BCF	sub ecx,edi	ecx:".5ea5/QPY4//", edi:"5ea5/QPY4//"
010813B9	8D42 0D	lea eax,dword ptr ds:[edx+D]	
010813BC	3BC1	cmp eax,ecx	ecx:".5ea5/QPY4//"
010813BE	7C 07	jl main_bin.10813C7	
010813C0	2BD1	sub edx,ecx	ecx:".5ea5/QPY4//"
010813C2	83C2 0D	add edx,D	
010813C5	EB 02	jmp main_bin.10813C9	
010813C7	8BD0	mov edx,eax	
010813C9	8A7C15 B8	mov bh,byte ptr ss:[ebp+edx-48]	[ebp-4C]:".5ea5/QPY4//"
010813CD	8845 B4	mov eax,dword ptr ss:[ebp-4C]	
010813D0	883C30	mov byte ptr ds:[eax+esi],bh	
010813D3	8D50 01	lea edx,dword ptr ds:[eax+1]	
010813D6	46	inc esi	
010813D7	8A08	mov cl,byte ptr ds:[eax]	
010813D9	40	inc eax	
010813DA	84C9	test cl,cl	
010813DC	75 F9	jne main_bin.10813D7	
010813DE	8B4D B4	mov ecx,dword ptr ss:[ebp-4C]	[ebp-4C]:".5ea5/QPY4//"
010813E1	2BC2	sub eax,edx	
010813E3	3BF0	cmp esi,eax	
010813E5	0F8C 55FFFFFF	jl main_bin.1081340	

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

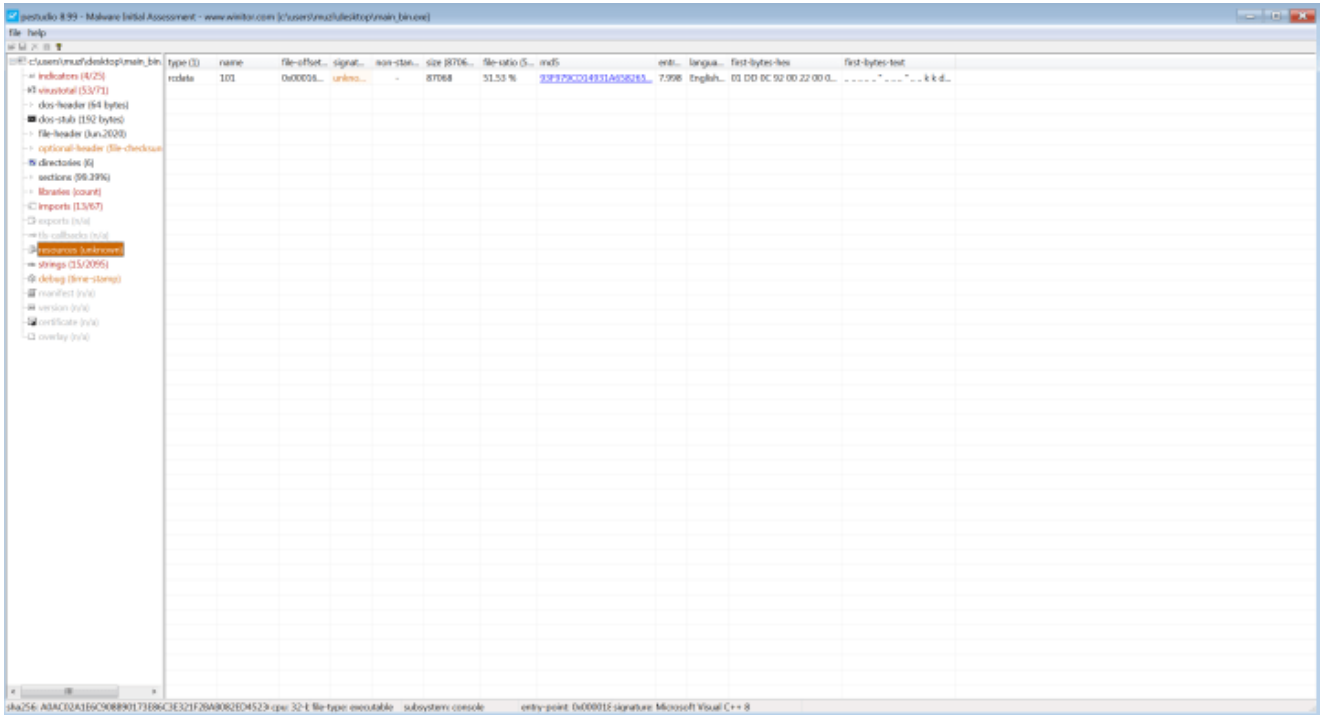


Figure 3: Resource (RCDATA)

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:"kerne132.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.

00FC1550	888405 F8FEFFFF	mov byte ptr ss:[ebp+eax-108],al	S-Box Init
00FC1557	40	inc eax	
00FC1558	3D 00010000	cmp eax,100	Cmp Len S-Box (255 Decimal)
00FC1559	7C F1	j1 main_bin.FC1550	
00FC155F	88BD ECFEFFFF	mov edi,dword ptr ss:[ebp-114]	
00FC1563	33F6	xor esi,esi	
00FC1567	66:0F1F8400 00000000	nop word ptr ds:[eax+eax],ax	
00FC1570	8A9C35 F8FEFFFF	mov bl,byte ptr ds:[ebp+esi-108]	
00FC1577	B8 89888888	mov eax,88888889	
00FC157C	F7E6	mul esi	
00FC157E	8BC6	mov eax,esi	
00FC1580	C1EA 03	shr edx,3	
00FC1583	8BCA	mov ecx,edx	
00FC1585	C1E1 04	shl ecx,4	
00FC1588	2BCA	sub ecx,edx	
00FC158A	2BC1	sub eax,ecx	
00FC158C	8D8D F8FEFFFF	lea ecx,dword ptr ss:[ebp-108]	
00FC1592	0FB64438 0C	movzx eax,byte ptr ds:[eax+edi+c]	Key
00FC1597	02C3	add al,bl	
00FC1599	02F8	add bh,al	
00FC159B	0FB6C7	movzx eax,bh	
00FC159E	03C8	add ecx,eax	
00FC15A0	0FB601	movzx eax,byte ptr ds:[ecx]	
00FC15A3	888435 F8FEFFFF	mov byte ptr ss:[ebp+esi-108],al	Scramble S-Box (PRGA/S-Box Permutation)
00FC15AA	46	inc esi	
00FC15AB	8819	mov byte ptr ds:[ecx],bl	
00FC15AD	81FE 00010000	cmp esi,100	Cmp Len S-Box (255 Decimal)
00FC15B3	7C BB	j1 main_bin.FC1570	
00FC15B5	88BD E8FEFFFF	mov edi,dword ptr ss:[ebp-118]	
00FC15BB	33F6	xor esi,esi	
00FC15BD	8A7D F8	mov bh,byte ptr ss:[ebp-8]	
00FC15C0	8A4D F9	mov cl,byte ptr ss:[ebp-7]	
00FC15C3	85FF	test edi,edi	
00FC15C5	7E 56	jle main_bin.FC1610	
00FC15C7	66:0F1F8400 00000000	nop word ptr ds:[eax+eax],ax	
00FC15D0	FEC7	inc bh	
00FC15D2	8D95 F8FEFFFF	lea edx,dword ptr ss:[ebp-108]	
00FC15D8	0FB6C7	movzx eax,bh	
00FC15DB	03D0	add edx,eax	
00FC15DD	8A1A	mov bl,byte ptr ds:[edx]	mov bl, CT
00FC15DF	02C8	add cl,bl	
00FC15E1	0FB6C1	movzx eax,cl	
00FC15E4	888D F7FEFFFF	mov byte ptr ss:[ebp-109],cl	
00FC15EA	8D8D F8FEFFFF	lea ecx,dword ptr ss:[ebp-108]	
00FC15F0	03C8	add ecx,eax	
00FC15F2	0FB601	movzx eax,byte ptr ds:[ecx]	
00FC15F5	8802	mov byte ptr ds:[edx],al	
00FC15F7	8819	mov byte ptr ds:[ecx],bl	
00FC15F9	0FB602	movzx eax,byte ptr ds:[edx]	
00FC15FC	8B8D F0FEFFFF	mov ecx,dword ptr ss:[ebp-110]	
00FC1602	02C3	add al,bl	
00FC1604	0FB6C0	movzx eax,al	
00FC1607	0FB684D5 F8FEFFFF	movzx eax,byte ptr ss:[ebp+eax-108]	
00FC160F	30040E	xor byte ptr ds:[esi+ecx],al	XOR to Plaintext in Buffer
00FC1612	46	inc esi	
00FC1613	8A8D F7FEFFFF	mov cl,byte ptr ss:[ebp-109]	
00FC1619	3BF7	cmp esi,edi	
00FC161B	7C B3	j1 main_bin.FC1500	
00FC161D	8B8D F0FEFFFF	mov ecx,dword ptr ss:[ebp-110]	
00FC1623	E8 D8F9FFFF	call main_bin.FC1000	
00FC1628	8B4D FC	mov ecx,dword ptr ss:[ebp-4]	

main_bin.00FC1000

.text:00FC1623 main_bin.exe:\$1623 #A23

Address	Hex	ASCII
000E0000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	#Z.....yy..
000E0010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
000E0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000E0030	00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 0001.....
000E0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	...".! .L!Th
000E0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
000E0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
000E0070	6D 6F 64 65 2E 0D 0B 0A 24 00 00 00 00 00 00 00	mode....\$.
000E0080	2A E9 99 31 6E 88 F7 62 6E 88 F7 62 6E 88 F7 62	*e.ln.+bn.+bn.+b
000E0090	7A E3 F4 63 64 88 F7 62 7A E3 F2 63 E1 88 F7 62	zãöcd.+bzãöcã.+b
000E00A0	7A E3 F3 63 7C 88 F7 62 65 E7 F2 63 48 88 F7 62	zãöcl.+beçöck.+b
000E00B0	65 E7 F3 63 7F 88 F7 62 65 E7 F4 63 7F 88 F7 62	eçöc.+beçöc.+b
000E00C0	7A E3 F6 63 6D 88 F7 62 6E 88 F6 62 3B 88 F7 62	zãöcm.+bn.ob;.+b
000E00D0	AA E7 FF 63 69 88 F7 62 AA E7 F5 63 6F 88 F7 62	*çyci.+b*çöco.+b
000E00E0	52 69 63 68 6E 88 F7 62 00 00 00 00 00 00 00 00	Richn.+b.....
000E00F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000E0100	50 45 00 00 4C 01 04 00 0C C2 E8 5E 00 00 00 00	PE..L.....ÄeA....
000E0110	00 00 00 00 ED 00 02 01 0B 01 0E 19 0D DA 00 00ä.....ü..
000E0120	00 88 00 00 00 00 00 00 F3 22 00 00 00 10 00 00ö".....
000E0130	00 F0 00 00 00 00 40 00 00 10 00 00 00 02 00 00	..b.....ö.....
000E0140	06 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00
000E0150	00 80 01 00 00 04 00 00 00 00 00 00 03 00 40 81ö.....
000E0160	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00
000E0170	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00
000E0180	24 4A 01 00 78 00 00 00 00 00 00 00 00 00 00 00	en r

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
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`, Cruloder 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)


```
[stage2_dumped.013D3CE0]=70 FE 6F 7C 22 2A 74 47 7D 5B F0 D2 7E 53 9B 65

.text:013C1013 stage2_dumped.bin:$1013 #413
```

Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1

Address	Hex	ASCII
013D3CE0	70 FE 6F 7C 22 2A 74 47 7D 5B F0 D2 7E 53 9B 65	pbol"*tG}[ðð~s.e

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

EIP	Address	Hex	Disassembly
	013C11C0	39448D EC	cmp dword ptr ss:[ebp+ecx*4-14],eax
	013C11C4	74 31	je stage2_dumped.13C11F7
	013C11C6	41	inc ecx
	013C11C7	83F9 04	cmp ecx,4
	013C11CA	7C F4	j1 stage2_dumped.13C11C0
	013C11CC	8BBD B4FDFFFF	mov edi,dword ptr ss:[ebp-24C]
	013C11D2	8D85 C0FDFFFF	lea eax,dword ptr ss:[ebp-240]
	013C11D8	50	push eax
	013C11D9	53	push ebx
	013C11DA	FFD7	call edi
	013C11DC	85C0	test eax,eax
	013C11DE	0F85 BCFEFFFF	jne stage2_dumped.13C10A0
	013C11E4	5F	pop edi
	013C11E5	5E	pop esi
	013C11E6	33C0	xor eax,eax
	013C11E8	5B	pop ebx
	013C11E9	8B4D FC	mov ecx,dword ptr ss:[ebp-4]
	013C11EC	33CD	xor ecx,ebp
	013C11EE	E8 A80E0000	call stage2_dumped.13C2098
	013C11F3	8BE5	mov esp,ebp
	013C11F5	5D	pop ebp
	013C11F6	C3	ret
	013C11F7	8B4D FC	mov ecx,dword ptr ss:[ebp-4]
	013C11FA	B8 01000000	mov eax,1
	013C11FF	5F	pop edi
	013C1200	5E	pop esi
	013C1201	33CD	xor ecx,ebp
	013C1203	5B	pop ebx
	013C1204	E8 920E0000	call stage2_dumped.13C2098
	013C1209	8BE5	mov esp,ebp
	013C120B	5D	pop ebp
	013C120C	C3	ret
	013C120D	CC	int3
	013C120E	CC	int3
	013C120F	CC	int3
	013C1210	55	push ebp
	013C1211	8BEC	mov ebp,esp
	013C1213	83EC 0C	sub esp,C
	013C1216	53	push ebx
	013C1217	56	push esi
	013C1218	57	push edi
	013C1219	FF348D 443C3D01	push dword ptr ds:[ecx*4+13D3C44]

dword ptr [ebp+ecx*4-14]=[003AF354]=D2F05B7D
 eax=D2F05B7D

.text:013C11C0 stage2_dumped.bin:\$11C0 #5C0

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.

013C1D50	BA 16D951A8	mov edx,A851D916	CreateProcessA
013C1D55	33C9	xor ecx,ecx	
013C1D57	E8 B4F4FFFF	call stage2_dumped.13C1210	
013C1D5C	BA 2E97584F	mov edx,4F58972E	WriteProcessMemory
013C1D61	A3 A46A3D01	mov dword ptr ds:[<&CreateProcessA>],eax	
013C1D66	33C9	xor ecx,ecx	
013C1D68	E8 A3F4FFFF	call stage2_dumped.13C1210	
013C1D6D	BA B9BE7238	mov edx,3872BEB9	ResumeThread
013C1D72	A3 CC6A3D01	mov dword ptr ds:[<&WriteProcessMemory>],eax	
013C1D77	33C9	xor ecx,ecx	
013C1D79	E8 92F4FFFF	call stage2_dumped.13C1210	
013C1D7E	BA 4D822EE6	mov edx,E62E824D	VirtualAllocEx
013C1D83	33C9	xor ecx,ecx	
013C1D85	E8 86F4FFFF	call stage2_dumped.13C1210	
013C1D8A	BA 4A0DCE09	mov edx,9CE0D4A	VirtualAlloc
013C1D8F	A3 C86A3D01	mov dword ptr ds:[<&VirtualAllocEx>],eax	
013C1D94	33C9	xor ecx,ecx	
013C1D96	E8 75F4FFFF	call stage2_dumped.13C1210	
013C1D9B	BA 108C80FF	mov edx,FF808C10	CreateRemoteThread
013C1DA0	A3 C46A3D01	mov dword ptr ds:[<&VirtualAlloc>],eax	
013C1DA5	33C9	xor ecx,ecx	
013C1DA7	E8 64F4FFFF	call stage2_dumped.13C1210	
013C1DAC	A3 D06A3D01	mov dword ptr ds:[<&CreateRemoteThread>],eax	
013C1DB1	33C0	xor eax,eax	
013C1DB3	C3	ret	

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.

012B1C03	0F1005 5c3c2c01	movups xmm0,xmmword ptr ds:[12C3C5C]	Ciphertext
012B1CDA	83C4 0C	add esp,C	
012B1CDD	C745 88 44000000	mov dword ptr ss:[ebp-78],44	44: 'd'
012B1CE4	8045 D0	lea eax,dword ptr ss:[ebp-30]	
012B1CE7	0F1145 D0	movups xmmword ptr ss:[ebp-30],xmm0	
012B1CEB	0F1005 6c3c2c01	movups xmm0,xmmword ptr ds:[12C3C6C]	
012B1CF2	50	push eax	
012B1CF3	0F1145 E0	movups xmmword ptr ss:[ebp-20],xmm0	
012B1CF7	FF15 1cf02b01	call dword ptr ds:[<&1strlenA>]	
012B1CFD	33C9	xor ecx,ecx	
012B1CFF	90	nop	
012B1D00	8A540D D0	mov dl,byte ptr ss:[ebp+ecx-30]	
012B1D04	C0C2 04	rol dl,4	
012B1D07	80F2 A2	xor dl,A2	
012B1D0A	88540D D0	mov byte ptr ss:[ebp+ecx-30],dl	
012B1D0E	41	inc ecx	
012B1D0F	3BC8	cmp ecx,eax	
012B1D11	^ 7C ED	j1 stage2_dumped.12B1D00	
012B1D13	FF73 08	push dword ptr ds:[ebx+8]	
012B1D16	8045 88	lea eax,dword ptr ss:[ebp-78]	
012B1D19	50	push eax	
012B1D1A	6A 00	push 0	
012B1D1C	6A 00	push 0	
012B1D1E	6A 04	push 4	
012B1D20	6A 00	push 0	Suspended
012B1D22	6A 00	push 0	
012B1D24	6A 00	push 0	
012B1D26	6A 00	push 0	
012B1D28	8D45 D0	lea eax,dword ptr ss:[ebp-30]	
012B1D2B	50	push eax	
012B1D2C	FF15 A46A2C01	call dword ptr ds:[<&CreateProcessA>]	C:\Windows\System32\svchost.exe

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 <u>B86A3801</u>	mov dword ptr ds:[&InternetOpenA],eax
B9 02000000	mov ecx,2
E8 0AF4FFFF	call svchost.1371210
BA F598C06C	mov edx,6CC098F5
A3 <u>D86A3801</u>	mov dword ptr ds:[&InternetOpenUrlA],
B9 02000000	mov ecx,2
E8 F6F3FFFF	call svchost.1371210
BA 241D19E5	mov edx,E5191D24
A3 <u>BC6A3801</u>	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 <u>9C3C3801</u>	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 <u>8C3C3801</u>	movups xmm0,xmmword ptr ds:[1383C8C]
01371E52	50	push eax
01371E53	0F1145 D0	movups xmmword ptr ss:[ebp-30],xmm0
01371E57	FF15 <u>1CF03701</u>	call dword ptr ds:[&!strlenA]
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\cru-loader\output.jpg`.

Next, `CruLoader` decrypts another string `redao1urc`. 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	
001D15FF	^ 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'.

00 00 00 00	00 00 00 72	65 64 61 6F	6C 75 72 63redaolurc
<u>4D 5A 90 00</u>	03 00 00 00	04 00 00 00	FF FF 00 00	MZ.....ÿÿ..
B8 00 00 00	00 00 00 00	40 00 00 00	00 00 00 00@.....
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00 00 00 00	00 00 00 00	00 00 00 00	00 01 00 00
0E 1F BA 0E	00 B4 09 CD	21 B8 01 4C	CD 21 54 68	..°..'.í!..Lí!Th
69 73 20 70	72 6F 67 72	61 6D 20 63	61 6E 6E 6F	is program cannot
74 20 62 65	20 72 75 6E	20 69 6E 20	44 4F 53 20	be run in DOS
6D 6F 64 65	2E 0D 0D 0A	24 00 00 00	00 00 00 00	mode....\$......
04 EA B9 50	40 8B D7 03	40 8B D7 03	40 8B D7 03	.ê¹P@.x.@.x.@.x.
49 F3 44 03	4A 8B D7 03	<u>4B E4 D6 02</u>	42 8B D7 03	IóD.J.x.KäÖ.B.x.
4B E4 D2 02	53 8B D7 03	4B E4 D3 02	4C 8B D7 03	KäÖ.S.x.KäÖ.L.x.
<u>4B E4 D4 02</u>	41 8B D7 03	<u>54 E0 D6 02</u>	45 8B D7 03	KäÖ.A.x.TàÖ.E.x.
40 8B D6 03	6E 8B D7 03	84 E4 DF 02	42 8B D7 03	@.Ö.n.x..äß.B.x.
84 E4 28 03	41 8B D7 03	84 E4 D5 02	41 8B D7 03	.ä(A.x..äö.A.x.
52 69 63 68	40 8B D7 03	00 00 00 00	00 00 00 00	Rich@.x.....
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
50 45 00 00	<u>4C 01 05 00</u>	19 AA E8 5E	00 00 00 00	PE..L.....aè^..
00 00 00 00	E0 00 02 01	0B 01 0E 19	00 0E 00 00à.....
00 14 00 00	00 00 00 00	6F 12 00 00	00 10 00 00o.....
00 20 00 00	00 00 40 00	00 10 00 00	00 02 00 00@.....
06 00 00 00	00 00 00 00	06 00 00 00	00 00 00 00
00 60 00 00	00 04 00 00	00 00 00 00	03 00 40 81@.....
00 00 10 00	00 10 00 00	00 00 10 00	00 10 00 00
00 00 00 00	10 00 00 00	00 00 00 00	00 00 00 00
8C 25 00 00	B4 00 00 00	00 40 00 00	E0 01 00 00	..%.´.....@..à..
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00 50 00 00	54 01 00 00	48 21 00 00	70 00 00 00	..P..T..H!..p..
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00 00 00 00	00 00 00 00	B8 21 00 00	40 00 00 00!@.....
00 00 00 00	00 00 00 00	00 20 00 00	04 00 00 00

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,38728EB9	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

```