Zero2Auto – Malware Book Reports

By muzi View all posts

Archived: 2026-04-10 03:18:39 UTC

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: VirtualAllo
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: t5gG8e514pbag5kg --> Decrypted: GetThrea
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: CreatePro
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: F5gG8e514pbag5kg --> Decrypted: SetThrea
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: ReadPro
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: s9a4E5fbhe35n --> Decrypted: FindResourc
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: Je9g5Ceb35ffz5=bel --> Decrypted: WriteP
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: I9egh1/n//b3rk --> Decrypted: VirtualAll
2022-05-30 22:06:05,423 - CruLoader Unpacker - CRITICAL [*] Encrypted: F9m5b6E5fbhe35 --> Decrypted: SizeofReso
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.

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.

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.

Figure 11: Resolve wininet APIs

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

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.

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.

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.

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: CreateProce
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted: F5gG8e514pbag5kg --> Decrypted: SetThread
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: WritePr
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: VirtualAllo
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted: F9m5b6E5fbhe35 --> Decrypted: SizeofResou
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: ReadProc
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: GetThread
2022-07-08 14:10:41,900 - CruLoader Unpacker - CRITICAL [*] Encrypted: s9a4E5fbhe35n --> Decrypted: FindResource
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
```
