Hunting IcedID and unpacking automation with Qiling

By Quentin Fois, Pavankumar Chaudhari

Published: 2021-07-26 · Archived: 2026-04-05 23:39:44 UTC

In our previous blog post “Detecting IcedID” we provided a global overview of the IcedID threat, its multiple stages and capabilities.

This new blog post is focused on how to be proactive and hunt for IcedID DLL components to extract network IOCs. It will involve a combination of Yara rules, the Qiling framework, and Python scripting.

Being a highly active threat, IcedID updates its packing technique regularly. This article focuses on what has been observed during the April – May 2021 timeframe. While the Yara introduced in this blog post may not be up to date for the latest samples at the time of publication, the overall hunting pipeline stays valid and can easily be tuned to tackle the latest threats.

This article is articulated in 4 parts:

- Building a Yara rule to find packed DLL samples
- Overview of the collected samples
- Automatic unpacking with Qiling
- IOC extraction

Hunting for packed samples with Yara

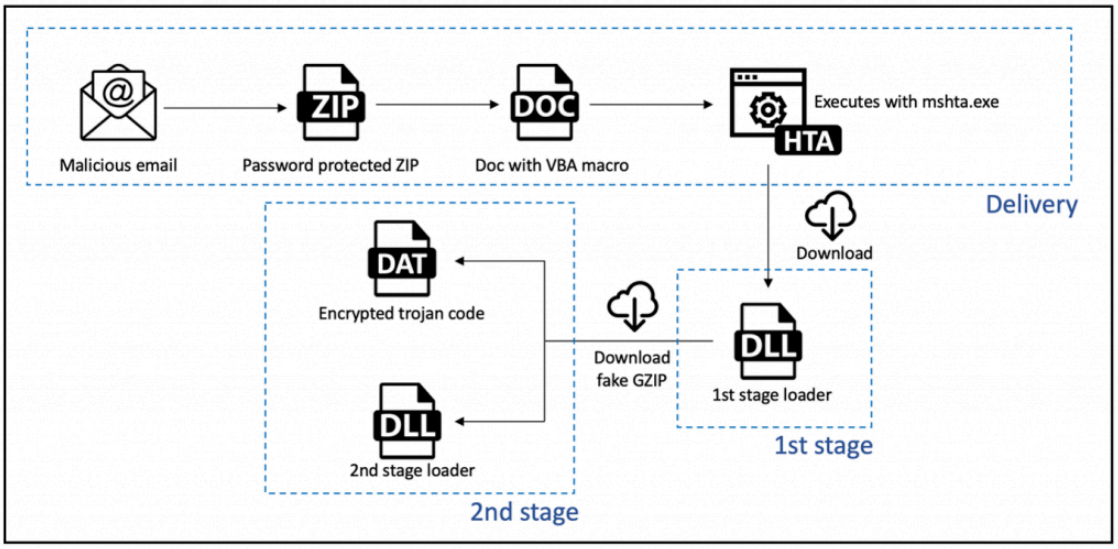

Figure 1 IcedID infection chain

Figure 1 above shows a diagram of the typical IcedID infection chain. The two DLLs mentioned above, 1st stage and 2nd stage loaders, are packed and obfuscated using the same packer. Thanks to VMware technology, detection

rules can be written on any layer of a packed sample. Meaning that we can write a detection rule based on the unpacked version of the DLL making our detection resilient to packer alteration.

However, for hunting purposes, being capable of detecting the packed layer of a sample is essential. This allows practitioners to find samples in an environment where only a static approach is available.

Writing a Yara rule is the standard way to detect samples in such an environment, and our research makes no exception. Designing a Yara rule is always a trade-off. One needs to find the fine balance between:

- Having a Yara so specific that it will only trigger on a couple of samples.
- Having a Yara so generic that it will trigger on unrelated samples, maybe even benign ones.

For this research, we focused on finding patterns in the code generated by the packer.

Below (Figure 2) is a screenshot of the disassembled code from a packed 1st stage DLL.

```
v2 = dword_18000874;
v3 = dword_18000870;
v4 = (((_BYTE)dword_18000874 * ((_BYTE)dword_18000874 - 1)) & 1) == 0 | dword_18000870 < 10;
v5 = dword_1800087C;
v6 = dword_18000878;
if ( !v4 )
  goto LABEL_5;
while ( 1 )
{
  if ( dword_18000878 >= 10 && (((_BYTE)dword_1800087C * ((_BYTE)dword_1800087C - 1)) & 1) != 0 )
  {
    while ( 1 )
      ;
  }
  if ( v4 )
    break;
LABEL_5:
  if ( dword_18000878 >= 10 && (((_BYTE)dword_1800087C * ((_BYTE)dword_1800087C - 1)) & 1) != 0 )
  {
    while ( 1 )
      ;
  }
}
v7 = *(int *)(a1 + 60);
result = 1164;
v9 = (unsigned int *)(a1 + *(unsigned int *)(v7 + a1 + 144));
if ( v9 && *(_DWORD *)(a1 + v7 + 148) )
{
  if ( dword_18000870 >= 10 && (((_BYTE)dword_18000874 * ((_BYTE)dword_18000874 - 1)) & 1) != 0 )
  {
    while ( 1 )
LABEL_13:
      ;
  }
```

Figure 2 Screenshot of disassembled packed code

Squared in red is a condition pattern that can be found in many places over the packed code, and as such may be a good candidate for a Yara rule. Let's have a closer look at the conditions.

```
(((_BYTE)dword_1800087C * ((_BYTE)dword_1800087C - 1)) & 1) != 0
```

Figure 3 Decompiled condition

Abstracting the variable used in the operation by a generic byte called x, we get:

```
((x * (x-1)) & 1) != 0
```

Figure 4 Generic condition

Simplifying the equation by stating Y = (x * (x-1)) gives us:

```
Y & 1 != 0
```

Figure 5 Simplified condition

One interpretation of this condition is checking if Y is odd or even.

The trick here is that Y = (x * (x-1)) always returns an even number because:

- either x or (x-1) is even.
- Multiplying any number with an even number gives an even number.

Thus Y & 1 always return 0. Summing up:

```
((x * (x-1)) & 1) != 0 <-> Always False whatever x value

((x * (x-1)) & 1) == 0 <-> Always True whatever x value
```

Figure 6 condition truth table

These conditions are what is called an opaque predicate. Opaque predicates are fake conditions that always resolve to the same outcome whatever value is used. Their only purpose is to artificially complexify the execution flow.

From our experience, these opaque predicate constructions seemed specific to this packer and would make a good candidate for a Yara rule.

Since Yara rules operate on the byte level, the next step is to find a proper byte pattern generalizing the instructions making this condition.

Looking at the assembly level, this is what the instructions look like:

```
8D 5D FF    lea     ebx, [rbp-1]
0F AF DD    imul    ebx, ebp
83 E3 01    and     ebx, 1
0F 84 7D+   jz      loc_18000A065
```

```
8D 41 FF    lea     eax, [rcx-1]
0F AF C1    imul    eax, ecx
83 E0 01    and     eax, 1
75 17       jnz     short loc_1800070A2
```

Figure 7 Highlighted in green are the common bytes between the instruction opcodes

By abstracting the part of the opcodes corresponding to a register, it can be converted into a seemingly generic Yara rule:

```
rule icedid_opaque_predicate_merged {
    strings:
        $opaque_predicate = { 8D ?? FF                  // lea REG32, [REG32-1]
                              0F ?? ??                  // imul REG32, REG32
                              83 ?? 01                  // and REG32, 1
                              (74 | 75 | 0F 84 | 0F 85) // short/far jz/jnz
                            }
    condition:
        uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550 and #opaque_predicate > 10
}
```

Figure 8 IcedID packer opaque predicate Yara rule

Running this rule on different datasets returned great results and provided us with many packed samples and very limited false positives. A curated and enriched IoC CSV, as well as the Yara rule is available on our Github repository.

Packed samples observations

This section highlights a few observations made on the bulk of samples collected via the Yara rule described above. Hashes can be found in the annexes section and on Github.

- Most packed DLLs contain a valid timestamp both in the file header and in the debug directory. Timestamp does not appear to be edited and could be used for timeline building purpose.

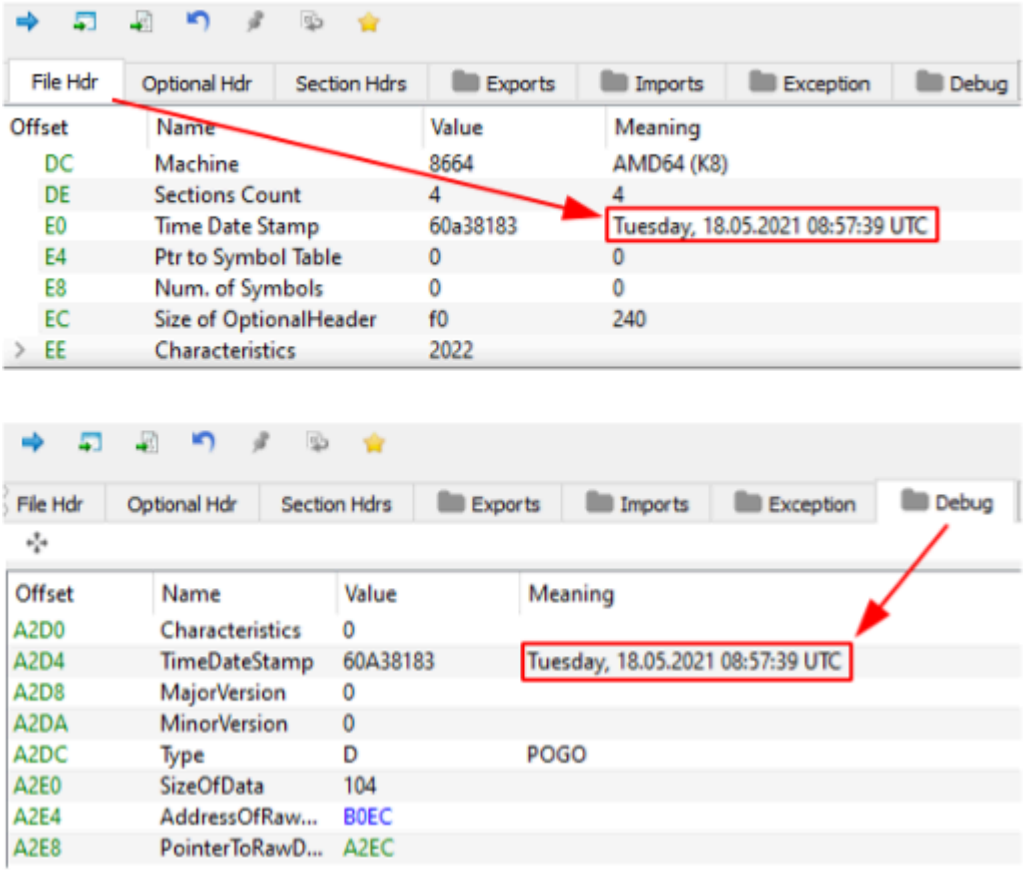

Figure 9 Timestamps found in the File Header and Debug directory

- Two subsystems were observed:

- **IMAGE_SUBSYSTEM_WINDOWS_GUI:** Commonly used by GUI application.
- **IMAGE_SUBSYSTEM_NATIVE:** This subsystem is used by drivers. However, in this case it is just here to confuse analysis systems as the DLL is invoked using rundll32 as a regular user space DLL.

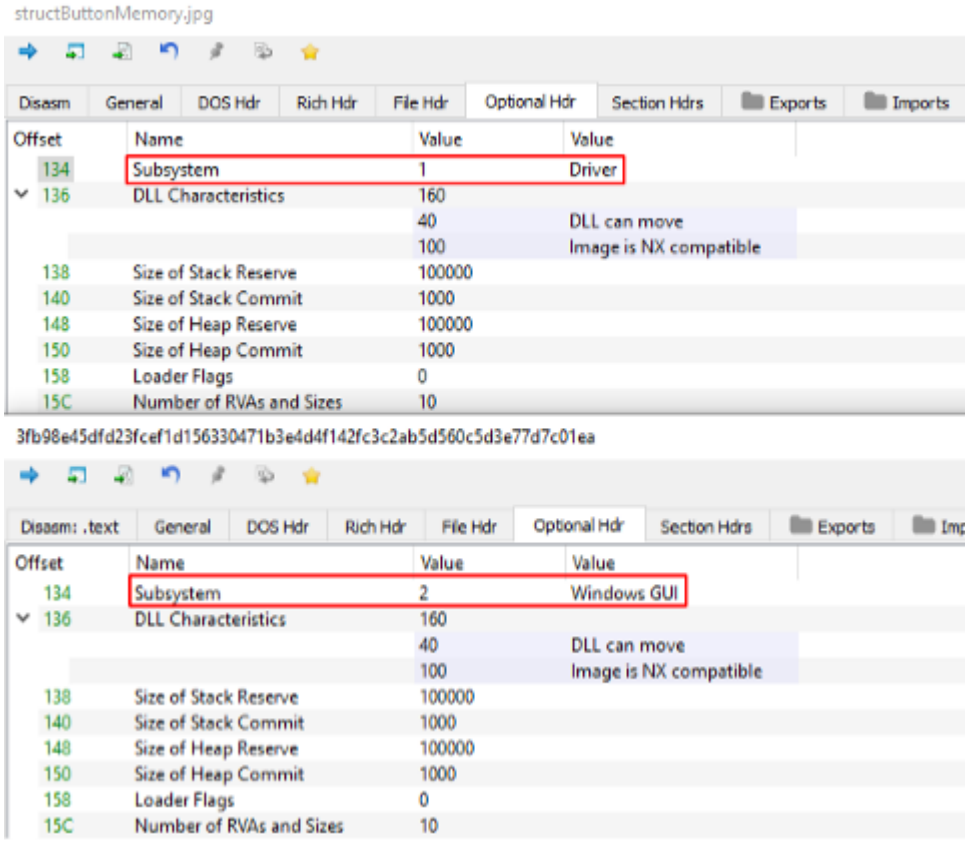

Figure 10 Packed DLL exposed different subsystems

- All samples analyzed have an exported function named either “PluginInit” or “update”. Usually “PluginInit” is used for 1st stage DLL, and “update” for 2nd stage DLL. Sometimes both are exported at the same time, and they both end up unpacking the sample the same way.

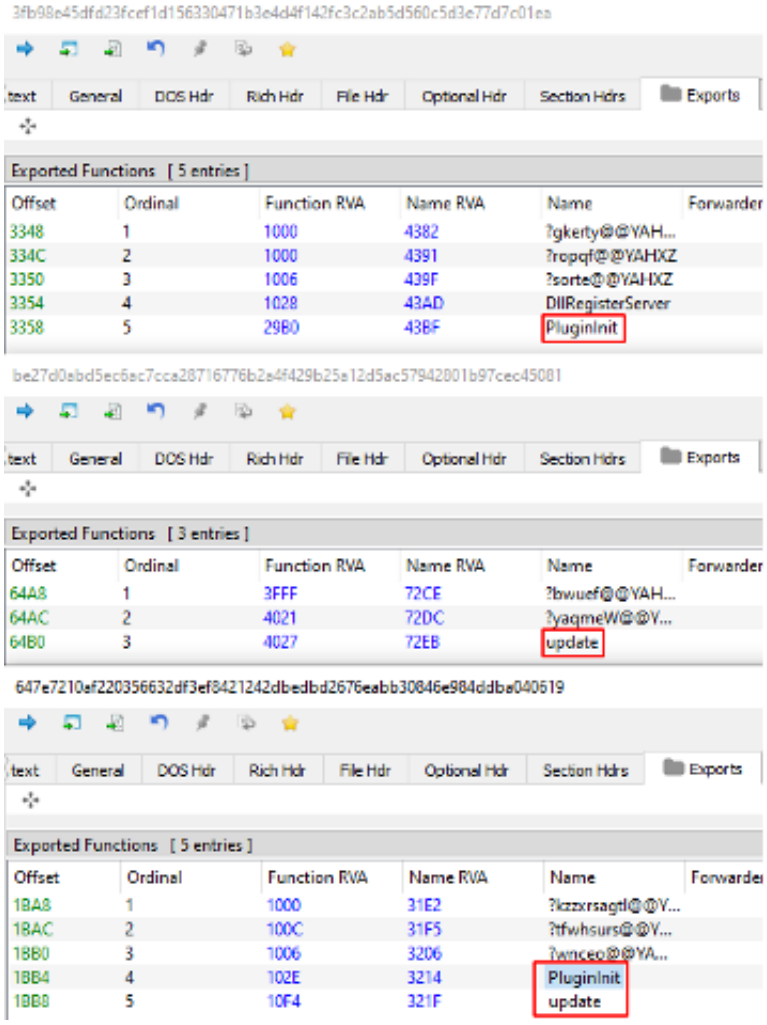

Figure 11 Export function names highlighted from 3 samples

- Some DLLs were identical except for 4 bytes in the overlay section at the very end of the binary. This data did not appear to be used at all during the unpacking routine and thus suggest that is only a trick to change the file hash. Not sure if samples are distributed as such or if another external actor just does so to mix up leads.

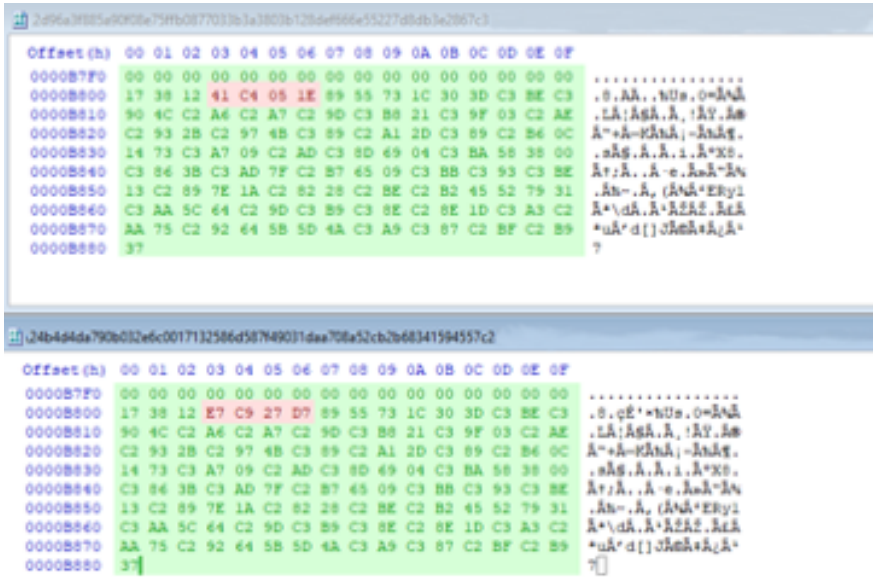

Figure 12 Two samples only differing by 4 bytes in the overlay part of the binary

- One sample was signed with a valid certificate chain.

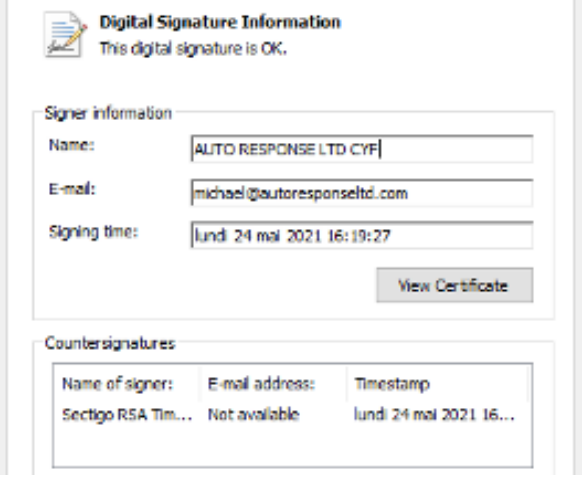

Figure 13 Signed sample

Overall, all these DLL samples are a mix of 1st stage and 2nd stage DLLs. While we already gathered interesting information just by looking at the available metadata, 1st stage DLLs contain something even more valuable: the domain name contacted to download the 2nd stage DLL.

However, to extract it one would first have to unpack the DLL and then decrypt the configuration blob.

Automatic unpacking with Qiling

As shown in the Yara creation section, the outer packed layer of the 1st stage DLL is full of opaque predicates and other obfuscation artifacts. While one could reverse the packer code and come up with a 100% static approach to unpack the inner layer of the DLL, this would take a non-negligeable amount of time and it would not be resilient to small packer change.

Previous manual experimentation showed that we were able to extract a clean version of the unpacked sample by using a debugger and placing a breakpoint on CreateThread. When the executed DLL triggers the breakpoint, we could then use PeSieve tool to extract the unpacked PE from memory.

However, this required too much manual interaction and did not scale well, so we found a middle ground by using Qiling.

Qiling is a lightweight emulator framework, easy to instrument, with a ready-to-use PE loader and Python bindings. It has already been introduced through multiple talks and blogpost.

Running a packed sample through Qiling is as easy as a few lines of python:

```
From qiling import *

# initialize emulator (x86_64 windows)
ql = Qiling(["./structButtonMemory.jpg"], "/qiling/examples/rootfs/x8664_windows")

# start emulation
ql.run(begin=0x180007FDF)
```

Figure 14 Executing IcedID 1st stage DLL via Qiling

Qiling output is rather verbose by default. One can see multiple calls to interesting windows API functions that can be used to have a better understanding of the execution flow and how the sample unpacks itself.

The output can be split into 4 parts:

1). Loader initialization

```
>>> from qiling import *
>>> ql = Qiling(["./structButtonMemory.jpg"], "/qiling/examples/rootfs/x8664_windows")
[=] Initiate stack address at 0x7fffffde000
[=] Loading ./structButtonMemory.jpg to 0x180000000
[=] PE entry point at 0x180008c37
[=] TEB addr is 0x6000030
[=] PEB addr is 0x60000b8
[=] Loading /qiling/examples/rootfs/x8664_windows/Windows/System32/ntdll.dll to 0x7ffff0000000
[!] Warnings while loading /qiling/examples/rootfs/x8664_windows/Windows/System32/ntdll.dll:
[!]   - SizeOfHeaders is smaller than AddressOfEntryPoint: this file cannot run under Windows 8.
[!]   - AddressOfEntryPoint lies outside the sections' boundaries. AddressOfEntryPoint: 0x0
[=] Done with loading /qiling/examples/rootfs/x8664_windows/Windows/System32/ntdll.dll
[=] Loading /qiling/examples/rootfs/x8664_windows/Windows/System32/kernel32.dll to 0x7ffff01e1000
[=] Done with loading /qiling/examples/rootfs/x8664_windows/Windows/System32/kernel32.dll
[=] Loading /qiling/examples/rootfs/x8664_windows/Windows/System32/shlwapi.dll to 0x7ffff0292000
[=] Done with loading /qiling/examples/rootfs/x8664_windows/Windows/System32/shlwapi.dll
[=] Loading /qiling/examples/rootfs/x8664_windows/Windows/System32/user32.dll to 0x7ffff02e4000
[=] Done with loading /qiling/examples/rootfs/x8664_windows/Windows/System32/user32.dll
[=] Loading /qiling/examples/rootfs/x8664_windows/Windows/System32/gdi32.dll to 0x7ffff0475000
[=] Done with loading /qiling/examples/rootfs/x8664_windows/Windows/System32/gdi32.dll
```

Figure 15 Qiling sample initialization logs

This is completely handled by Qiling framework. It loads the binary in a way Windows loader would do, creating PEB, TEB, then loading the IAT.

2). Unpacking routine

```
>>> ql.run(begin=0x180007FDF)
[=] LoadLibraryA(lpLibFileName = "kernel32.dll") = 0x7ffff01e1000
[=] GetProcAddress(hModule = 0x7ffff01e1000, lpProcName = "VirtualAlloc") = 0x7ffff01fa630
[=] GetProcAddress(hModule = 0x7ffff01e1000, lpProcName = "VirtualFree") = 0x7ffff01fa640
[=] VirtualAlloc(lpAddress = 0, dwSize = 0x3000, flAllocationType = 0x3000, flProtect = 0x4) = 0x5000008bc
[=] VirtualAlloc(lpAddress = 0, dwSize = 0x3000, flAllocationType = 0x3000, flProtect = 0x4) = 0x5000038bc
[=] VirtualAlloc(lpAddress = 0, dwSize = 0x5b000, flAllocationType = 0x3000, flProtect = 0x40) = 0x5000068bc
```

Figure 16 Unpacking routine logs

Here Qiling is instructed to start executing code from the address 0x180007FDF which was identified earlier as the address of the exported function ”PluginInit”. One can see the packer layer starting its unpacking job by allocating multiple memory region that will ultimately contain the unpacked binary.

3). Reflective loading

```
[=] Loading /qiling/examples/rootfs/x8664_windows/Windows/System32/advapi32.dll to 0x7ffff05fc000
[=] Done with loading /qiling/examples/rootfs/x8664_windows/Windows/System32/advapi32.dll
[=] LoadLibraryA(lpLibFileName = "ADVAPI32.dll") = 0x7ffff05fc000
[=] GetProcAddress(hModule = 0x7ffff05fc000, lpProcName = "GetUserNameA") = 0x7ffff0639f90
[=] GetProcAddress(hModule = 0x7ffff05fc000, lpProcName = "LookupAccountNameW") = 0x7ffff060ec60
[=] Loading /qiling/examples/rootfs/x8664_windows/Windows/System32/shell32.dll to 0x7ffff069d000
[=] Done with loading /qiling/examples/rootfs/x8664_windows/Windows/System32/shell32.dll
[=] LoadLibraryA(lpLibFileName = "SHELL32.dll") = 0x7ffff069d000
[=] GetProcAddress(hModule = 0x7ffff069d000, lpProcName = "SHGetFolderPathA") = 0x7ffff08823f0
[=] Loading /qiling/examples/rootfs/x8664_windows/Windows/System32/winhttp.dll to 0x7ffff1ae2000
[=] Done with loading /qiling/examples/rootfs/x8664_windows/Windows/System32/winhttp.dll
[=] LoadLibraryA(lpLibFileName = "WINHTTP.dll") = 0x7ffff1ae2000
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpOpen") = 0x7ffff1aed050
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpQueryHeaders") = 0x7ffff1aff960
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpReadData") = 0x7ffff1b01f50
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpReceiveResponse") = 0x7ffff1b119c0
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpSetOption") = 0x7ffff1aff5e0
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpCloseHandle") = 0x7ffff1b00e20
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpSendRequest") = 0x7ffff1b0ebe0
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpSetStatusCallback") = 0x7ffff1aff000
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpConnect") = 0x7ffff1b10ed0
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpQueryDataAvailable") = 0x7ffff1b01730
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpQueryOption") = 0x7ffff1aeffe0
[=] GetProcAddress(hModule = 0x7ffff1ae2000, lpProcName = "WinHttpOpenRequest") = 0x7ffff1aecaa0
[=] LoadLibraryA(lpLibFileName = "USER32.dll") = 0x7ffff02e4000
[=] GetProcAddress(hModule = 0x7ffff02e4000, lpProcName = "wsprintfA") = 0x7ffff02e5130
[=] GetProcAddress(hModule = 0x7ffff02e4000, lpProcName = "wsprintfW") = 0x7ffff030f300
[=] LoadLibraryA(lpLibFileName = "KERNEL32.dll") = 0x7ffff01e1000
[=] GetProcAddress(hModule = 0x7ffff01e1000, lpProcName = "GetComputerNameExA") = 0x7ffff0201760
[...]
```

Figure 17 Unpacked sample IAT parsing logs

After completing the unpacking logic, the sample starts loading the unpacked DLL in memory. We can observe the IAT parsing in the logs, including the winhttp.dll portion that will handle the C2 communication.

4). Unpacked sample execution

```
[=] CreateThread(lpThreadAttributes = 0, dwStackSize = 0, lpStartAddress = 0x5000078bc, lpParameter = 0, dwCreationFlags = 0, lpThreadId
= 0) = 0xa0000001
[=] Sleep(dwMilliseconds = 0xd)
[=] Sleep(dwMilliseconds = 0xb)
[=] Sleep(dwMilliseconds = 0x4)
[=] Sleep(dwMilliseconds = 0xd)
```

Figure 18 Unpacked sample execution logs

Finally, the unpacked DLL starts executing and one of the first APIs that is executed is CreateThread followed by a Sleep loop.

It should be noted that the value of the lpStartAddress parameter is listed as 0x5000078bc (Figure 14) which is inside the range returned by the last VirtualAlloc called in the 2nd step (Figure 12).

Knowing that, we can make the following assumptions:

- The embedded PE is being unpacked in one of the memory regions allocated with VirtualAlloc
- The sample is fully unpacked now that CreateThread is being called

From there, one can make use of the hooking functionalities exposed by Qiling to design the following unpacking procedure:

1. Hook VirtualAlloc
2. Hook CreateThread
3. Save each allocated memory address in an array
4. When we reach CreateThread, go through the allocated memory blob, and check if they contain a PE header. If yes, dump the content to disk

Running this properly unpacks the binary.

IcedID Stages and IOC extraction

As mentioned above, IcedID is a multi-stage threat. The same packer is used to pack both 1st stage and 2nd stage DLLs. However, only the 1st stage reaches out to a CnC subsequently containing a network domain IOC. For this reason, figuring out heuristics enabling one to identify what stage is being unpacked is important (see table)

Layer	Heuristic	Description
Packed DLL	Export function	1st stage DLL tend to have only the `PluginInit` function exported while 2nd stage DLL usually have `update` function exported.
Unpacked DLL	Imported functions	The 1st stage DLL import functions from the winhttp.dll DLL
Unpacked DLL	File size	The 1st stage DLL is about twice the size of the 2nd stage DLL

Table 1 DLL identification heuristics

Out of these, we found the presence of the “winhttp” DLL import to be the easiest heuristic to implement.

The 1st stage DLL contains a small configuration blob, storing an ID and a domain name from which 2nd stage will be downloaded. The “encryption” algorithm is straightforward: two data blobs of lengths 0x20 are xored between them.

```
1  do {
2    decrypted_conf[i] = xored_blob[i] ^ xored_blob[i + 0x40];
3    ++i;
4  } while ( i < 0x20 );
```

Figure 19 Configuration decryption routine

As far as we’ve observed, this xored data blob is always stored at the start of the .data section, making it trivial to locate and write a few lines python script to extract and decode the domain name.

Everything is stitched together in the Python script provided on our Github repository:

- Sample unpacking
- Stage identification
- Network IoC extraction

Conclusion

Through this article, we went through the different essential stages of sample hunting, from Yara designing to unpacking automation and IoC extraction. While the Yara provided may not generate hits anymore at the time of

reading due to the fast pace of IcedID packer update, the whole thought process stays relevant.

IOCs:

Annex 1: Figures SHA256

Figure 2:

- F75EFD25362D8EED3C2B190E3BA178B2CBBCF7C7DA0B991F4FE1D85072E8DCC6

Figure 5:

- D2511FDA63F3F4A1ED6A85765B0FBCDC4A965C933D77A2C1EB49D025B0E2609D

Figure 6:

- A15AB7254721CD06C225E4FBBEC127DE2AC987D6AA508C7D4803ACA9430BF94B
- 3FB98E45DFD23FCEF1D156330471B3E4D4F142FC3C2AB5D560C5D3E77D7C01EA

Figure 7:

- 3FB98E45DFD23FCEF1D156330471B3E4D4F142FC3C2AB5D560C5D3E77D7C01EA
- BE27D0ABD5EC6AC7CCA28716776B2A4F429B25A12D5AC57942801B97CEC45081
- 647E7210AF220356632DF3EF8421242DBEDBD2676EABB30846E984DDBA040619

Figure 8:

- 2D96A3F885A90F08E75FFB0877033B3A3803B128DEF666E55227D8DB3E2867C3
- 24B4D4DA790B032E6C0017132586D587F49031DAA708A52CB2B68341594557C2

Figure 9:

- 908381E81C59EC12908BA7929D38FAF542297E550DB628E330F3652CD6C99F91

Annex 2: Hunted IOCs

Available on https://github.com/Lastline-Inc/iocs-tools
