Applied Emulation - Decrypting Ursnif strings with Unicorn

By map[name:Alessandro Strino]

Published: 2023-12-20 · Archived: 2026-04-05 13:55:17 UTC

Introduction

Here we go with another chapter of the Applied Emulation series. In the previous blogpost we have talked about emulation and more specifically, the usage of *Dumpulator* to perform a quick triage. However, in this blogpost we are going to use a completely different approach, taking advantage of **static analysis to understand and emulate a string decryption algorithm through Unicorn**. As always, I would like to give a practical example on a real case scenario. Because of that, I have chosen to analyze an **Ursnif** sample.

It's worth mentioning that this sample is part of weekly challenges related to the Zero2Auto course. However, instead of automating the full code extraction, as requested by the challenge, I found even more interesting going into unexplored paths applying emulation instead.

Unicorn challenges

Using pure static analysis came with few limitations, in fact, Unicorn is a CPU emulator framework that does not have built in capabilities to invoke syscall directly. Because of that we need to overcome at least a couple of challenges before emulating our code:

- The first and probably the most challenging constraint is that we are going to emulate CPU instructions, instead of using direct calls to Windows API. Because of that, we need to choose our code carefully.
- After that we have identified a candidate algorithm, we still need to set up the emulation environment variables (e.g., stack memory, registers, sections, start and end code, algorithm parameters, etc…).

Speaking about "good candidate function", we could look for a custom and self-contained algorithm. Even if those requirements seem to be strict constraints, an experienced reverser knows that most of the malware out there have these components as a part of their decryption routine. However, let's take a step back. Someone may wonder why we don't use Dumpulator instead of Unicorn, considering all these limitations. The answer is straightforward:

Dumpulator may not be usable if we are unable to run the code and set up the user mode space. Nevertheless, I strongly believe that it is more **important to understand where and when a technique should be applied**, even if it adds a few layers of complexity, rather than rushing for the easiest or the first solution that comes to our mind.

Ursnif decryption routine

Decryption routine is pretty straightforward to locate, it just follows the campaignDate variable that is going to be used as a decryption key for the .bss segment. Key generation algorithm has been already described [here](#), however, I would like to focus on the decryption routine itself, in order to understand all requirements to emulate this code.

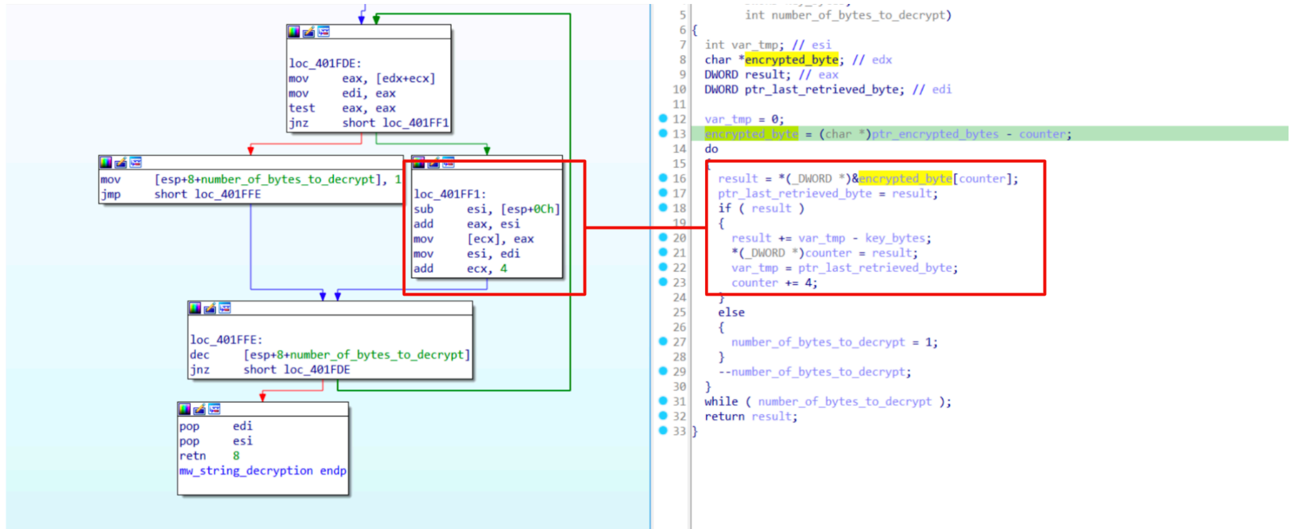

Figure 1: Decryption routine explained

At the first glance we could see that the function is self-contained and most of the operations are simple math (sub, add) that involves the key and the encrypted data. However, before proceeding it's important to have a look at the following instruction:

[ecx] is going to be the place to save the output of each decrypted byte. Nevertheless, ecx is also used as a counter to get the next byte from the **.bss segment**:

Because of that we need to handle that situation very carefully. In fact, **if we do not consider those instructions, we are going to have an access memory violation** with ecx pointing to an unallocated memory region. In order to solve this issue, there are two ways:

- Allocating memory at **0x000000**. In this case, ecx will be used as a counter and as a pointer without any issue;
- Otherwise, we could use a **hook** on our emulator that is going to analyze each instruction and when we are dealing with **[ecx]**, we could skip that instruction, redirecting the decrypted output towards another variable.

For this blogpost we are going to use the first solution. Considering this post as an introduction of Unicorn, I don't want to overwhelm readers with a lot of content (setting up the environment will be quite challenging to follow). It's always good to master some basic concept before proceeding towards more advanced techniques

Setting up emulation environment

First of all we need to import the Unicorn module and also unicorn constants that we are going to use to interact with registers. Then we could use the pefile module to load our bin file and take notes of **.bss segment data.**.

Then, we need to take the whole code that needs to be emulated. Before proceeding, it's important to keep in mind that emulation is quite slow and we need to use only the code needed for our purpose. For instance, prolog and epilogue instructions are not very useful for us, since we are emulating a single function without knowing any information about previous ESP and EBP values. Because of that we could skip them.

To get the proper bytes, with IDA, we could highlight the code that is going to be emulated and, using **shift+e**, spawn a popup with all selected bytes, ready to be pasted in our code.

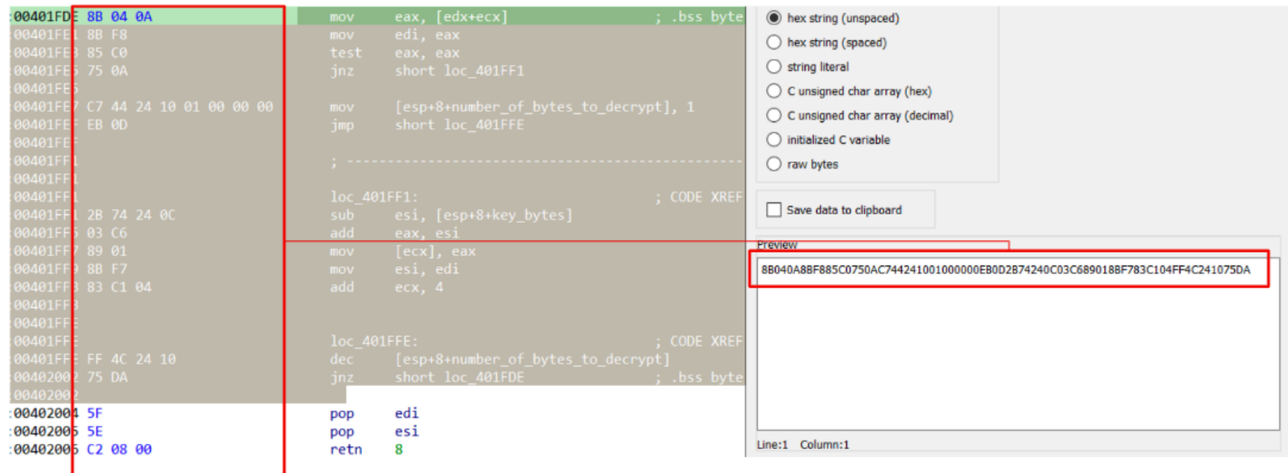

Figure 2: Select code to emulate

Now things are starting to heat up, trying to settle our emulation. First of all, we need to instantiate Unicorn, then we need to set up stack memory, ESP, data memory (using .bss content) and then fill the code section with our bytes. To make it easier to read, I have split the whole script in multiple parts, as follow:

Through this snippet of code, we have prepared everything related to the stack and data section. More precisely, with **uc.mem_map** we have mapped a memory address with its size and protections (rwx permissions have been chosen to avoid exceptions).

It's worth mentioning that, what really makes those memories different is the ESP register that represents a pointer onto the stack. However, under the hood, there are no differences between the allocated memories.

In order to get the key, we have called a gen_key routine (omitted here, to focus on emulation setup) that is going to return a string containing the decryption key. Then, once we get the key, we have to store it on the stack, respecting the little endian format. Along with key parameters we have also settled all other arguments needed for the function, such as: encrypted data (edx), counter (ecx) as well as updated the esp according to the instruction [esp+0xC] that is going to take the key bytes.

As mentioned above, since that ecx will be either a pointer and a counter, we are going to create a memory space located to **0x00000000**. In this case, when we are going to deal with **mov [ecx] ,eax** instruction, we will avoid any exception.

To conclude we need to setup memory for our code. This is a pretty straightforward task if have followed the previous steps. Map the code address, fill it with null bytes and then write the bytes stored in the code variable.

Now we have satisfied all setting requirements for our emulation and its time to test it out.

Testing

Running our emulator against few Ursnif samples reveal that we have correctly settled all the environment variables. As you can see from the image below, it has been possible to retrieve all encrypted strings. Some of them could be very useful in order to do a step further in our analysis, since that we have references to infrastructure, powershell script as well as network requests template.

```
%D,3&[illegible]FoverridelinkX%08X-%04X-%04X-%04X-%08X%04XKERNEL32.DLLP[illegible]uValueNTDLL.DLLGetStringValueZwWriteVirt
ualMemoryLoadLibraryAGetDWORDValueCreateKeyhDefKeyReturnValueroot\defaultZwWow64QueryInformationProcess64.binStdRegProvLdrUnregisterDllNotificationformSetSt
ringValue/images/version=%u&soft=%u&user=%08x%08x%08x%08x&server=%u&id=%u&type=%u&name=%sSetDWORDValueGetBinaryValue&action=%08xSetBinaryValueContent-Dispos
ition: form-data; name="upload_file"; filename="%s"DeleteKeysSubKeyNamesValueName

--%s
%s

sValue--%s--
https://__ProviderArchitecturePOST%systemroot%\system32\control.exe -h[illegible]F[illegible]FrY[illegible]B
[illegible]{%08X-%04X-%04X-%04X-%08X%04X}ZwSetContextThreadRtlNtStatusToDosErrorropenZwWow64ReadVirtualMemory6464ZwProtectVirtualMemory%02u-%02u-
%02u %02u:%02u:%02u
ZwGetContextThreadMozilla/4.0 (compatible; MSIE 8.0; Windows NT %u.%u%s)kernelbase%S%xNTDSAPI.DLLLdrRegisterDllNotificationS:(ML;;NW;;;LW)D:(A;;0x1fffff;;;W
D)(A;;0x1fffff;;;S-1-15-2-1)(A;;0x1fffff;;;S-1-15-3-1)%c%02X%s=%s&soft=%u&version=%u&user=%08x%08x%08x%08x&server=%u&id=%u&crc=%x&uptime=%u&%ssize=%u&hash=0
x%08xhttp://%u%u%uContent-Type: multipart/form-data; boundary=%sContent-Disposition: form-data; name="upload_file"; filename="%.4u.%lu"Content-Type: applica
tion/octet-streamGETCreateProcessAZwMapViewOfSectionZwCreateSectionZwUnmapViewOfSectionZwClose.jpeg.gif.bmp.aviSoftware\AppDataLow\Software\Microsoft\© 2020
Microsoft Corporation. All rights reserved.%systemroot%\system32\c_1252.nls\*.dll0123456789ABCDEF.exeSoftware\Microsoft\Windows\CurrentVersion\Run.dllLocal
\Global\&ip=%srundll32 "%s",%S&os=%s%u.%u_%u_%u_x%u&tor=1&dns=%s&whoami=%s%08x%08x%08x%08xrunascmd.exe/C "copy "%s" "%s" /y && rundll32 "%s",%S"/C "copy "%
s" "%s" /y && "%s" "%s""Low\MicrosoftWow64EnableWow64FsRedirectionIsWow64ProcessD:(D;OICI;GA;;;BG)(D;OICI;GA;;;AN)(A;OICI;GA;;;AU)(A;OICI;GA;;;BA)@CODE@HKCU
HKLMIE10RunOnceLastShown_TIMESTAMP/C ping localhost -n %u && del "%s"SOFTWARE\Microsoft\Windows NT\CurrentVersionInstallDate%S=new ActiveXObject('WScript.Sh
ell');%S.Run('powershell new-alias -name %S -value gp; new-alias -name %S -value iex; %S ([System.Text.Encoding]::ASCII.GetString((%S "%S:\%S").%s))',0,0);m
shta "about:<hta:application><script>%S='wscript.shell';resizeTo(0,2);eval(new ActiveXObject(%S).regread('%S\\\%S\\\%s'));if(!window.flag)close()</script>"I
E8RunOnceLastShown_TIMESTAMPSOFTWARE\Microsoft\Internet Explorer\MainVersionCheck_AssociationsnoHost:%APPDATA%avast
```

Figure 3: Decrypted strings

Conclusion & Further Development

Emulation is a powerful tool, like a swiss knife, and it needs to be considered anytime we are dealing with malware automation. I hope that some of you had fun reading this post and learnt something new, to the point that you can't wait to apply this technique to your own sample.

I'm still planning to write some structured content about this topic. I don't have a deadline for that but it is in my new year resolution list! In the meantime, keep reversing!

References

Sample analyzed:

- Unpacme
- MalwareBazaar

Unicorn:

- Unicorn engine

Emulator:

- ursnif_string_decryptor.py

Source: https://viuleeenz.github.io/posts/2023/12/applied-emulation-decrypting-ursnif-strings-with-unicorn/