Applied Emulation - Analysis of MarsStealer

By map[name:Alessandro Strino]

Published: 2023-11-15 · Archived: 2026-04-05 16:32:31 UTC

Introduction

Emulation is a technique that could be very handy and effective when we have to deal with **malware triage**, **configuration extractor** and **deobfuscate part of the code without rewriting complex algorithms**. Even if it seems magic (and it's not unfortunately) it's still not possible to apply emulation on random code. However, if applied correctly this method could really speed up our malware analysis and triage. Through this blogpost I would like to give an overview about emulation usage and apply it in a real case scenario.

Recently, I followed a twitter warning about Vidar malware in the wild and eager to revalidate an IDA-python script to deobfuscate strings, I immediately jumped into it. However, extracting that sample it was pretty obvious that I was dealing with another stealer, called MarsStealer. Since I did have a lot of information about that sample, I thought that could be a good choice to experiment with emulation. The result was quite promising and because of that **I want to take this occasion to show a few basic emulations that could, hopefully, help someone else to speed up its analysis**.

Opening up in IDA the original sample, it was clear that a lot of strings were actually obfuscated and the code was partially packed, because of the references to jumps or calls towards registries and un-initialized DWORD.

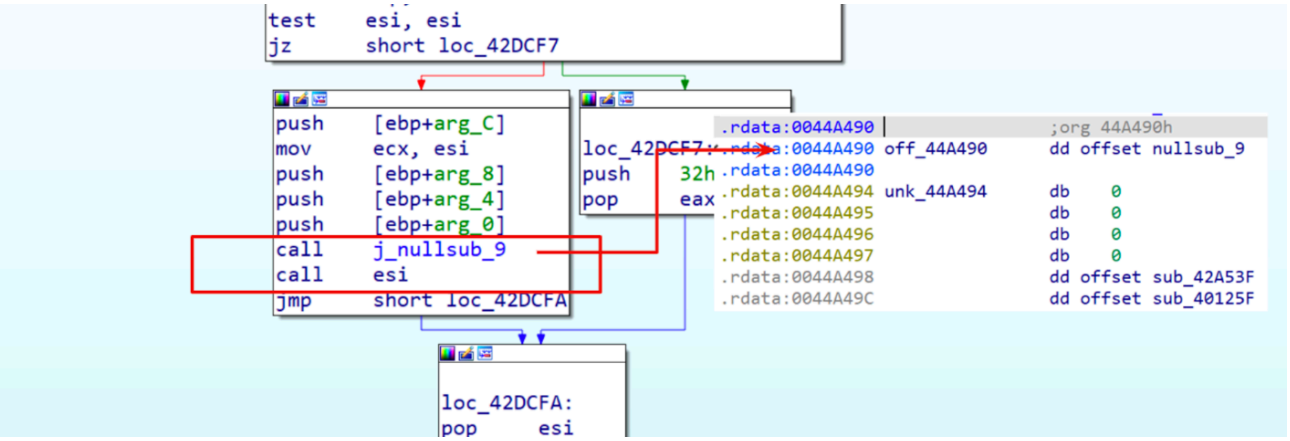

Figure 1: MarsStealer packed code

In order to extract the actual payload that will contain a deobfuscation string routine and additional code, it's necessary to go for dynamic analysis and speed up our extraction. As always one of the quickest methods to extract unpacked information is to look for **VirtualAlloc** and **VirtualProtect.**

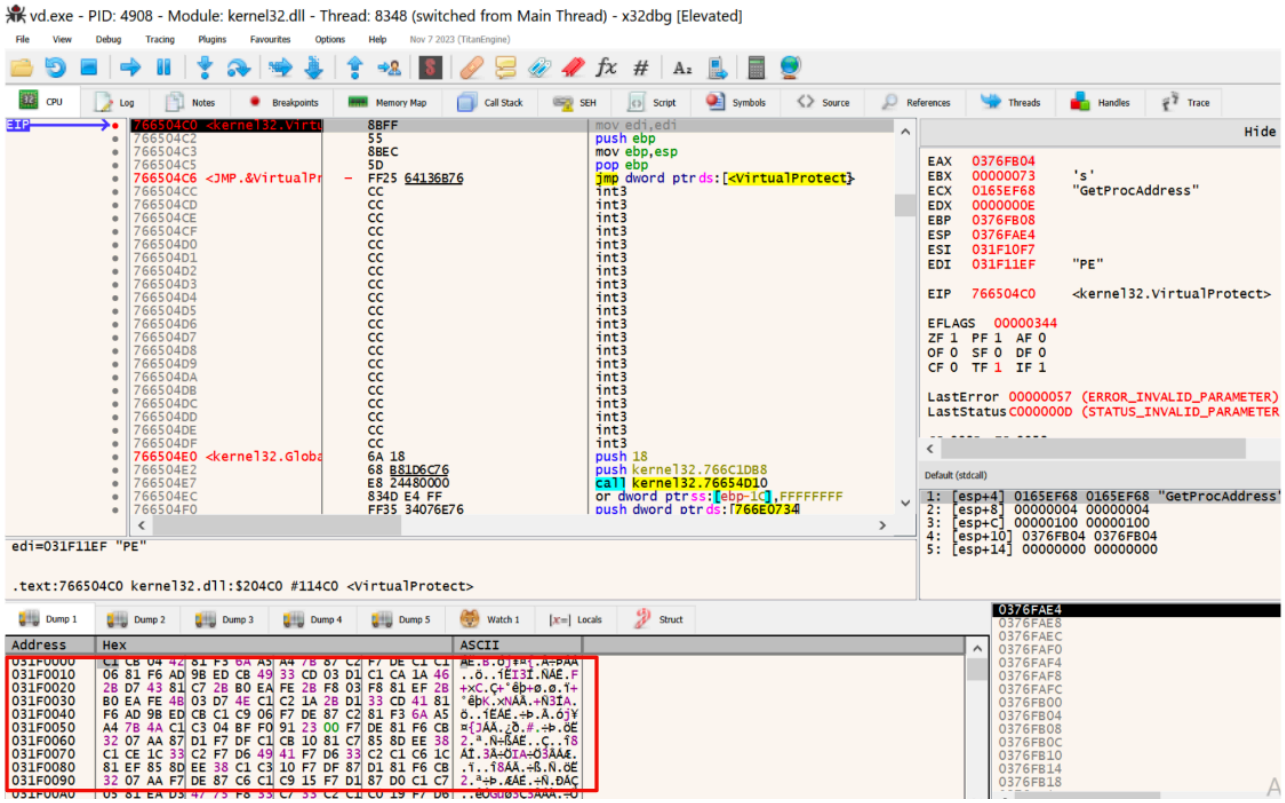

Figure 2: Unpacked payload retrieved

Deobfuscation routine analysis

Since that our purpose is to find out code to emulate, we could look for a deobfuscation routine within the payload extracted. Our search won't last long because, almost **immediately after the VirtualProtect** call the malware jumps directly to the allocated memory, **starting the name resolving routine.**

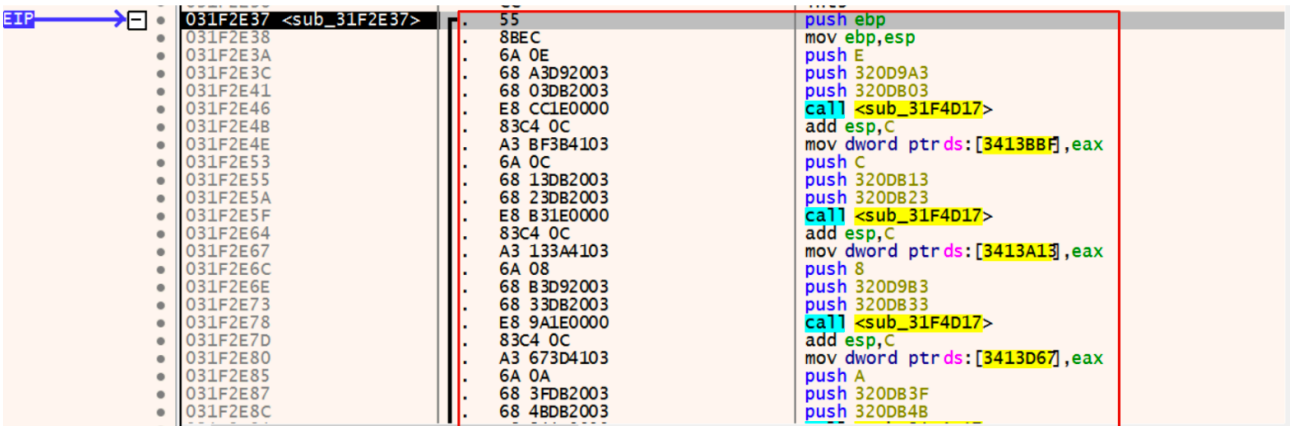

Figure 3: Deobfuscation wrapper

The highlighted function is a wrapper that contains the actual routine. The code is quite easy to spot because of the **three push instructions**. Opening the payload in IDA, it's possible to go a little bit deeper and explore the deobfuscation routine, reconstructing its signature and from that point, understanding the function flow.

Pseudocode-A

```
1  int __cdecl mw_stringResolver(char *ciphertext, char *key, unsigned int key_length)
2  {
3    char chr_ciphertext; // bl
4    unsigned int i; // [esp+4h] [ebp-Ch]
5    int v6; // [esp+8h] [ebp-8h]
6    int v7; // [esp+Ch] [ebp-4h] BYREF
7
8    v6 = MEMORY[0x47FA11B](64, key_length + 1);
9    *(_BYTE *)(key_length + v6) = 0;
10   for ( i = 0; i < key_length; ++i )
11   {
12     MEMORY[0x47FA1D3](75483849);
13     MEMORY[0x47FA1D3](75483850);
14     MEMORY[0x47FA1D3](75483856);
15     chr_ciphertext = ciphertext[i];
16     *(_BYTE *)(i + v6) = key[i % MEMORY[0x47FA117](key)] ^ chr_ciphertext;
17     MEMORY[0x47FA117](75483857);
18     MEMORY[0x47FA117](75483858);
19     MEMORY[0x47FA117](75483861);
20   }
21   v7 = 0;
22   MEMORY[0x47FA11F](v6, 4, 256, &v7);
23   return v6;
```

Figure 4: Deobfuscation routine

Regardless of the red box related memory errors, it's very easy to understand its core functionality and control flow. However, even if it seems quite easy to reconstruct its logic, I'm going to take this code as a use case to try a completely different approach, using **emulation to resolve all the strings**.

Emulation requirements

Before proceeding with emulation, there are few things to settle. The first one is that for emulating this code, we need to emulate the user mode because we are dealing with instructions that are going to make additional **calls to WindowsAPI**. For that reason, we are going to use dumpulator, implementing if needed some API calls. The second thing to talk about are **the requirements for dumpulator**. To make it effective, it's necessary to take a **minidump** of the process that we are analyzing and understand the parameters for **starting** and **stopping emulation**.

- In order to **take a minidump**, its possible to use **x32dbg/x64dbg** that include it as a command (e.g., minidump mstealer.dump);
- then to take the **starting point**, it's possible to take references to deobfuscation calls and save those addresses for later.

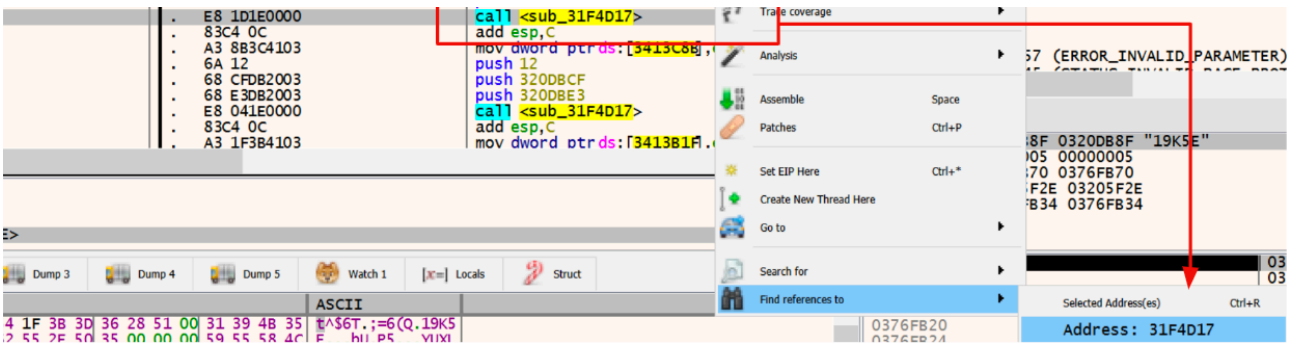

Figure 5: References to deobfuscation function

Now that we have two out of three requirements, it's necessary to focus on the **emulation ending point**. Of course, this one is the most important requirement and could impact your result in terms of efficiency (emulation is tremendously slow) and code writing (could be required to write more code that fits your needs).

For the purpose of this analysis/tutorial about emulation I'm going straight to the point using **hints collected from IDA** and doing some dynamic code analysis.

Observing carefully the Figure 4, it's easy to spot that the plaintext string is settled **after the for loop** and **before the VirtualProtect call**. Looking at the assembly with the information acquired, it's easy to understand that emulation should stop at the instruction **push ecx**. In fact, **ecx register is going to be a pointer for the plaintext string**.

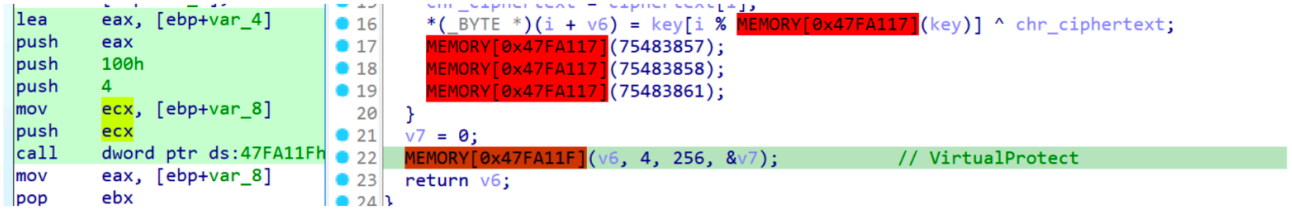

Figure 6: Focus on plaintext resolution

With all this information, the **emulation end variable** could be easily retrieved within the debugger at the address **0x031f4De5**.

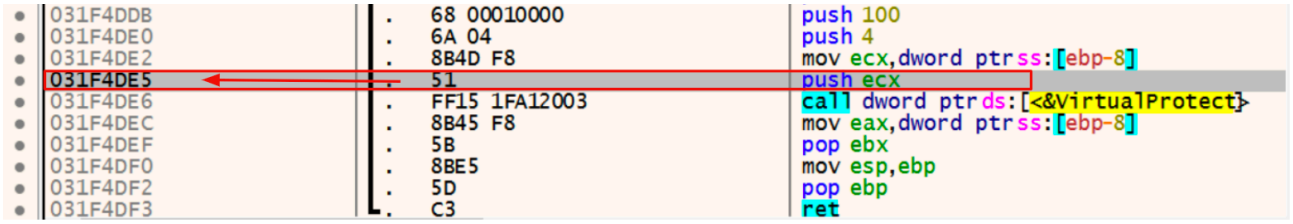

Figure 7: Emulation stop address

String Resolving Automation

Since that we have collected all the requirements for the emulation, we are ready to setup our code as follow:

Launching this script we are able to extract a lot of information on Mars Stealer, starting our triage without even reversing the whole malware. In fact, from the resolved string we have something related to common stealer targets such as: credit cards, browser, crypto wallet, etc..

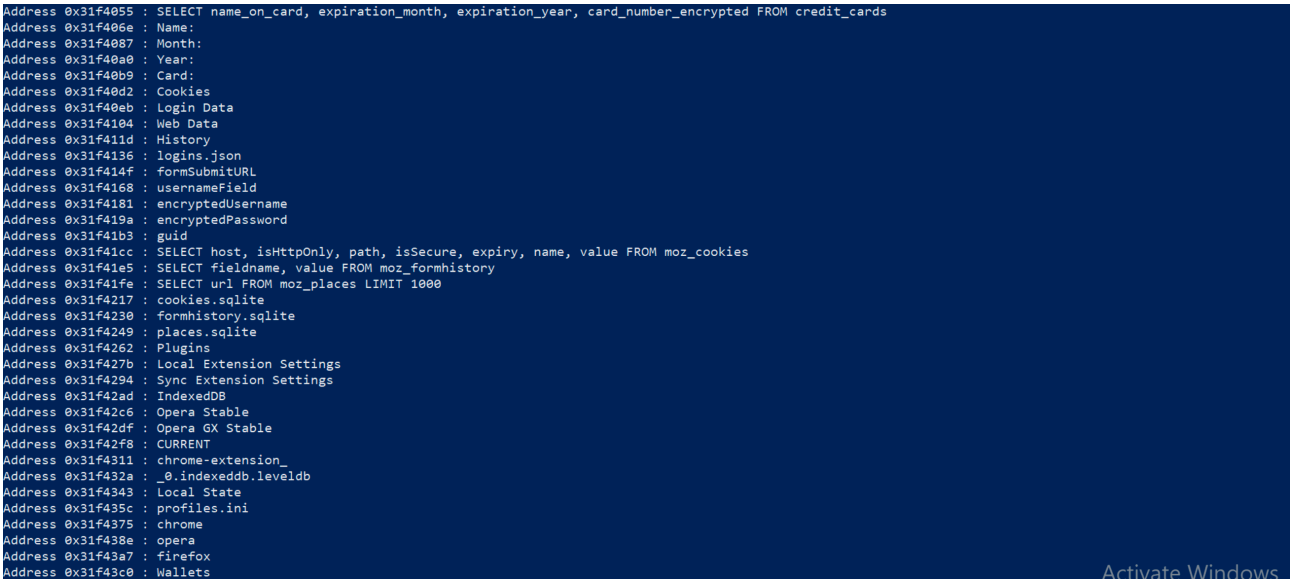

Figure 8: Retrieved strings

Additionally we also have a chance to get a few insights about anti-analysis or reversing-aware functions such as: **IsDebuggerPresent** or **CreateToolhelp32Snapshot**. Additionally we have also some indications about anti-sandbox techniques with **HAL9TH**, that should be the Microsoft sandbox computer name. All deobfuscated strings could be found within the **Reference section**.

Conclusion and next chapter

Emulation represents the state of the art for analyzing malware functions or triaging sample without losing yourself in complex and heavily obfuscated routine. It was pretty fun to analyze Mars Stealer through this technique. I'm thinking of creating additional and probably more structured content (maybe a Whitepaper) about malware emulation.

The script above could be used as a reference for further analysis, it's quite simple (and not perfect) but very effective and I used that as a "*soft*" introduction to this topic and also to give an idea of emulation capabilities.

Hope you enjoyed reading this post as much as I had reversing this malware and writing this article!

References

Sample analyzed:

- MalwareBazaar Sample

Minidump:

- mars_stealer_minidump.7z

String resolver:

- MarsStealer_stringResolver.py

Extracted strings:

- strings.txt

Dumpulator:

- Reference
