Malware Analysis: ModiLoader - VinCSS Blog

By Yến Hứa

Published: 2020-09-11 · Archived: 2026-04-05 20:39:40 UTC

Table of Contents

1. Introduction

Recently, I have been investigating a malware loader which is **ModiLoader**. This loader is delivered through the Malspam services to lure end users to execute malicious code. Similar to other loaders, **ModiLoader** also has multi stages to download the final payload which is responsible for stealing the victim's information. After digged into some samples, I realized that this loader is quite simple and didn't apply anti-analysis techniques like **Anti-Debug**, **Anti-VM** that we have seen in **GuLoader/CloudEyE** samples (1;2). Instead, for avoiding antivirus detection, this loader uses digital signatures, decrypts payloads, Url, the inject code function at runtime and executes the payload directly from memory.

Currently, according to my observation, there are not many analysis documents about this loader in the world as well as in Vietnam. So, in this post, I will cover techniques are used by this loader as well as apply new released tool from FireEye is capa that helps to quickly find the loader's main code. During the analysis, I also try to simulate the malicious code in python script for automatic extracting and decoding payload, Url.

2. About the sample

SHA256: 9d71c01a2e63e041ca58886eba792d3fc0c0064198d225f2f0e2e70c6222365c

Results from PE Scanner tools show that this loader is written in **Delphi**, using **Digital Signatures** to bypass the AV programs running on the client:

3. Technical analysis

3.1. First stage analysis

At the first stage, the loader (*considered as the first payload*) performs the task of extracting data, decoding the second payload (*this payload can be **dll** or **exe***), and executing the payload from memory.

By using IDA, at the end of the automated analysis, IDA has identified up to **5,385** functions:

Code block at **start()** function of loader:

Although, much more functions were identified as above, most of them are Windows APIs as well as Delphi's library functions, so that finding out the main code related to decoding the second payload will take a long time. With the help of capa, I quickly found the code related to executing the second payload and then traced back to the code that responsible for decoding this payload.

The entire code at **sub_498CDC()** function is responsible for parsing the payload, mapping into the memory and executing it. Code in this function before and after applying the relevant struct:

Trace back will reach **sub_4994EC()**, this function performs tasks:

- Reads all data from the resource named "**T__7412N15D**" into memory.

- Finds "**OPPO**" string in resource binary data to retrieve the encrypted payload.

- Performs decoding to get the second payload. The key used in decoding process is a numeric value.
- Searches string in the second payload and replace it with the encoded URL string.

In the picture above, the decryption key is an integer converted from the string. In this sample, key value is **0x30**. The code is responsible for decoding the payload as shown below:

An implementation of this decoding operation can be written in Python as the below image:

Once the payload has been decoded, the loader will search for the placeholder in the decoded payload and replace the **168** “z” characters with the encoded URL string. Finally, once the payload is ready for execution, it

calls **sub_498CDC()** for executing the payload.

And from beginning until now, the above entire technical analysis can be done with a python script to obtain the second payload.

3.2. Second stage analysis

Check the payload retrieved in the above step, it is also written in Delphi:

With the similar method, I found **sub_45BE08()** which is responsible for allocating the region of memory, map the final payload after decoded into this region, and then execute it.

By tracing back, I found the code that starts at **TForm1_Timer1Timer** (*recognized by IDA by signature*) at the address is **0x45CC10**. Before calling **f_main_loader()** at address is **0x45C26C**, the code from here is responsible for decoding Url and checking the Internet connection by trying to connect to the decoded Url is **https://www.microsoft.com**.

Decoding algorithm at **f_decode_char_and_concat_str()** function is as simple as follows: **dec_char = (enc_char >> 4) | (0x10 * enc_char);**

At **f_main_loader()**, it also uses the same above function to decode and get the string is **“Yes”**. This string is later used as **xor_Key** for decoding the Url to download the last payload (*The encrypted Url is the string in the replacement step that was described above*) as well as decoding the downloaded payload. **f_decode_url_and_payload(void *enc_buf, LPSTR szKey, void *dec_buf)** function takes three parameters:

- The first parameter is **enc_buf**, used for store the encoded data.
- The second parameter is **szKey**. It is the **“Yes”** string used to decode the data.
- The third parameter is **dec_buf,** used for store the decoded data.

Diving into this decoding function, you will realize that it will loop through all data, each iteration takes 2 bytes, convert the string to an integer, then **xor** with the character extracted from the decryption key. Once decrypted, the byte is then concatenated to the third argument, which is the output buffer.

This entire decoding function is rewritten in python as follows:

Back to the **f_main_loader()**, first it will decode the Url for retrieving the last payload:

Perform decoding using the python code above, I obtain the Url as below image:

Next, it uses the **WinHTTP WinHttpRequest COM** object for downloading the encrypted payload from the above Url. Instead of using Internet APIs functions from **Wininet** library as in some other samples, the change to using COM object might be aimed at avoiding detection by AV programs.

Here, I use **wget** to download the payload. The payload's content is stored in hex strings similar to the encoded above Url.

Payload data will be reversed and decoded by the same **f_decode_url_and_payload** function with the same decoding key is **"Yes"**. Once decrypted, the sample will allocate a region of memory, map the payload into that region, and then execute it.

Along with the python code above, I can decode the downloaded payload and obtain the final payload. This payload is a dll file and also written in Delphi:

3.3. Third stage analysis

The above payload is quite complicated, it performs the following tasks:

- Reads data from a resource named "**DVCLAL**" into memory.
- Decrypts this resource, then based on the "***()%@5YT!@#G__T@#$%^&*()__#@$#57$#!@**" pattern to read the decrypted data into the corresponding variables.

- Retrieves the user's directory information through the %**USERPROFILE%** environment variable and set up the path to **%USERPROFILE%AppDataLocal** folder.
- Creates **Vwnt.url** and **Vwntnet.exe** (*copy of loader*) files in **%USERPROFILE%AppDataLocal** folder if that files not exist, then set the value is "**Vwnt**" that pointing to the **%USERPROFILE%AppDataLocalVwnt.url** file at "**HKCUSoftwareMicrosoftWindowsCurrentVersionRun**" key**.** Then write data to **Vwnt.url** with content that points to **Vwntnet.exe** file:

- Combines the decrypted data from the above resource for decrypting the new payload.

- Decrypts the function is responsible for injecting code. Check "**C:Program Files (x86)internet explorerieinstal.exe**" exists or not, if exists it will inject payload into **ieinstal.exe**.

- Based on the strings was dumped from the decrypted payload, I can confirm that it belongs to the **Warzone RAT**, a well-known RAT that is being offered online and promoted on various hacking forums.

4. References

- MalwareBazaarDatabase (ModiLoader)
- DBatLoader/ModiLoaderAnalysis – First Stage
- capa:Automatically Identify Malware Capabilities
- Warzone:Behind the enemy lines

Xem bài phiên bản tiếng Việt

Tran Trung Kien (aka m4n0w4r)

Malware Analysis Expert

R&D Center – VinCSS (a member of Vingroup)
