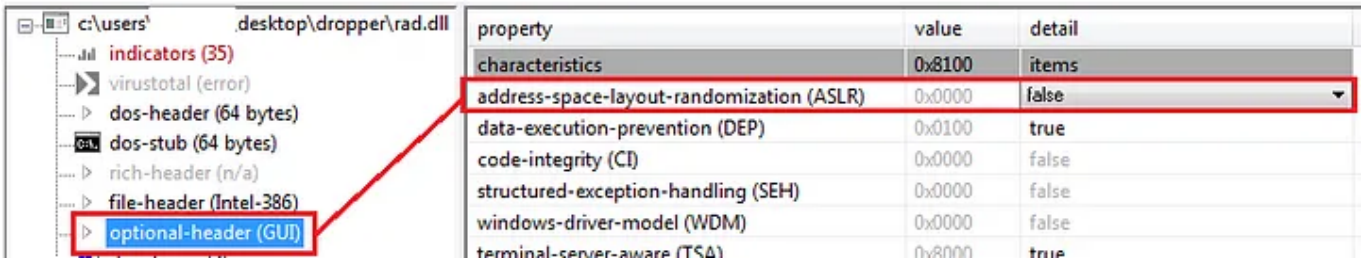
Dancing With Shellcodes: Analyzing Rhadamanthys Stealer

As for usage, in the dark web, the malware authors offer various deals for using the malware such as monthly or even lifetime payments.

1. *PART 1: The Dropper*
   - *Unpacking mechanism: getting to the first shellcode*
   - *Shellcode execution via Callback*
   - *Investigating the first shellcode*
   - *Fixing functions statically: Defining functions*
   - *Fixing functions statically: Defining code*
   - *Fixing the shellcode: Rebase the address*
   - *Shellcode functionality*
   - *Summarize the first shellcode*

---

property	value
md5	89EC4405E9B2CAB987F2E4F7E4B1666E
sha1	EC48082347136444540C9B8BA4EABCFDC526868C
sha256	AF04EE03D69A7962FA5350D0DF00FAFC4AE85A07DFF32F99F0D8D63900A47466
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00
first-bytes-text	M Z . . . . . @ . . . . .
file-size	189440 bytes
entropy	6.346
imphash	5231D45D27FAAB064697CD89D612E981
signature	Microsoft Visual C++ v6.0
tooling	n/a
entry-point	55 8B EC 6A FF 68 E8 A4 40 00 68 28 37 40 00 64 A1 00 00 00 00 50 64 89 25 00 00 00 00 83 EC 58 53
file-version	n/a
description	n/a
file-type	executable



Check ASLR

Unpacking mechanism: getting to the first shellcode

As we observe the dropper in IDA, we see a large embedded “blob” in the *.rdata* section. Usually, these kinds of blobs can potentially contain data that

Creating new heap

Then, the function *sub_408028* will be the core function that will deal with encrypting the blob. Inside *sub_408028*, there are two interesting functions:

Next, the same function will override the blob and will decrypt a shellcode.

The entire chain can also be seen in the following pseudo-code of IDA pro:

1. The function *sub_405728* is responsible to invoke the API call *ImmEnumInputContext*
2. *sub_405728* receives as a parameter function named *sub_407228* which is just a wrapper for another function that jumps to the shellcode address
3. The final result is that *ImmEnumInputContext* will get the address of the shellcode in its second argument “lpfn” and will execute it.

To investigate it statically, we obviously must dump the shellcode, to do it do the following:

1. Right click on the address of the shellcode and click “Follow in Memory Map”

According to the IDA website[6] blue means “Regular functions, i.e. functions not recognized by FLIRT or Lumina.”
And brown means “Instructions(code) not belonging to any functions. These could appear when IDA did not detect or misdetected function boundaries, or hint at code obfuscation being employed which could prevent proper function creation. It could also be data incorrectly being treated as code.”

And when we look at an area in the IDA view that contains both we see the following:

We can obviously see that the brown color is a legit code, however, IDA doesn't consider it as a code and therefore does not show it as a function.

To fix this, we can just scroll and observe statically from where this function starts and when it ends.
In our case, it starts at the address *000029E*, we also see the prologue:
push ebp
mov ebp, esp
And ends at the address *000036B* with the epilogue:
leave

Defining functions

Then, we can see that the brown code is now considered a function, and a

NOTE: When fixing functions do not assume that the first "*retrn*" is the end of a function, pay attention to the jumps that might bypass this return and might indicate a longer function.

Fixing the shellcode: Defining code

In addition to the convenient scenario of a code that looks like code and just doesn't interpret as a function, we have a more tricky scenario when we need to change the data itself.

At the beginning of the shellcode, we can see dynamically the assembly code

To fix it, we need to tell IDA that some specific addresses are actual code.
For example: in the dynamic view, we can see that the first 5 bytes are:
Call 450028

Therefore, we should tell IDA that the first 5 bytes are code, then, we can tell IDA to look at it as a function.
To do it, do the following:

1. Mark the data

Defining as code

As we see, the function jumps to the address at "*loc_28*" (IDA) or "*450028*" (debugger), however in IDA this content also needs to be fixed. Combining the two approaches of defining as code and defining as function can fix will do the trick.

After doing that, we now have 8 functions in the function name bar.

The first thing we can see is that the actual code in shellcode is very small, there are 8\9 functions, and the rest is a big chunk of data. From this, we can assume that the shellcode will potentially use that data.

So let's “go with the flow” and understand this shellcode

1. *sub_450000* just jumps to *sub_450028*
2. *sub_450028* jump jumps to *sub_45029E*

Hashing function

Next, in the address *00450314,* we can see the call for *VirtualAlloc*, don't forget to observe the allocated memory using follow in dump of the *EAX* register (in my case it's *00470000*).

sub_45003A will get the decrypted content and our newly allocated memory as arguments and will copy the data to it.

Summarize the first shellcode

To summarize the entire shellcode activity, we can look at it from a code point of view

At this point, it will be useless to continue writing the anti-analysis capabilities, so for those who want to see all, please visit the al-khaser project GitHub page.

Evasion Technique: Manipulate Exception Handling

One of the most interesting capabilities of the Rhadamanthys loader is exception-handling manipulation.

What is Exception handling?

In *sub_5978*, the loader gets the address of *KiUserExceptionDispatcher* and starts to iterate on it to search for a specific location where *ZwQueryInformationProcess* is called.

Patch KiUserExceptionDispatcher

1. *SEM_NOOPENFILEERRORBOX* - The system does not display the critical-error-handler message box. Instead, the system sends the error to the calling process.
2. *SEM_NOGPFAULTERRORBOX* — The system does not display the Windows Error Reporting dialog.
3. *SEM_FAILCRITICALERRORS* — The OpenFile function does not display a message box when it fails to find a file. Instead, the error is returned to the caller.

Note that mutexes with this name are already found in the OS and are created by *MSCTF.dll*, and more info can be found in this[10] article.

After creating the Mutex, we moved to a function named *sub_2B92* which holds the core activity and the main purpose of the loader.

Evasion Technique: Disabling hooks

In the function named *sub_8060,* we see one of the cool tricks of malware to protect themselves against user mode hooking.

and will use *memcpy* to copy the data from the fake to the real one. In this way, the malware verifies that no hooks are set.

Check for hooks in other DLLs

Config Decryption

The config decryption occurs in a function named *sub_3DD4*, which is a function that will do various activities that the main loader activity requires.

Collect information about the machine

Getting ConnectEx

Send & Recieve data

If we want to observe dynamically the data that is sent to the C2, do the following:

1. Set a breakpoint at the address where *WSASend* is being executed.
2. Follow in dump the address of the second parameter aka *lpBuffers*
3. This buffer is a *WSABUF* structure, and its second parameter is a pointer to the actual buffer that is sent to the C2.

Loader's goal

After performing its various capabilities and tricks, the loader will execute its main goal.

1. The loader will download a DLL from the C2
2. Write it to the disk with the name of *nsis_uns[xxxxxx].dll*
3. Spawn *Rundll32* to execute the DLL with the export function "*PrintUIEntry*" which is a name of a legitimate export function of the printui.dll

NSIS Loader

The loader is executed via a very long command that changed in every iteration

Also, for this part, I will only focus on the stealing capabilities and its targets.

Stealing KeePass passwords

The malware appears to be able to use the DLL KeePassHax[12], an open-source tool used to decrypt the password database.

Keepass

2. Pale Moon
3. Sleipnir5
4. Opera
5. Chrome
6. Twinkstar
7. Firefox
8. Edge

The malware appears to get the profile, username, and password of OpenVPN.

6. Bytecoin
7. Binance
8. Electron
9. Solar waller
10. Zap
11. WalletWasabi
12. Zcash

Querying registry keys for digital coming entities from Joe[

Resolving APIs dynamically

The stealer is resolving dynamically his APIs using the GetModuleHandle

5. EtwEventWrite

At this stage, I decided to stop my analysis

For everyone's convenience, I also uploaded all the files from my analysis including the shellcodes to VirusTotal.

Rhadamanthys files

https://www.virustotal.com/gui/file/8384322d609d7f26c6dc243422ecec3d40b3

[9] https://learn.microsoft.com/en-us/cpp/cpp/structured-exception-handling-c-cpp?view=msvc-170

[10] https://www.hexacorn.com/blog/2018/12/25/enter-sandbox-part-22-ctf-capturing-the-false-positive-artifacts/

[11] https://tria.ge/221227-vprhbsae8t/behavioral2#report

[12] https://github.com/HoLLy-HaCKeR/KeePassHax

Written by Eli Salem

Follow

173 Followers

Malware Researcher & Threat Hunter