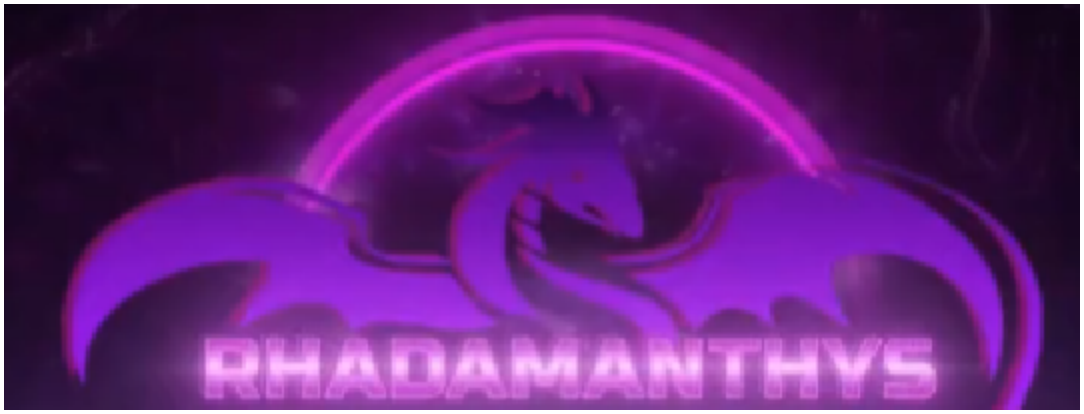
Dancing With Shellcodes: Analyzing Rhadamanthys Stealer

elis531989.medium.com/dancing-with-shellcodes-analyzing-rhadamanthys-stealer-3c4986966a88

Eli Salem January 16, 2023



Eli Salem

Jan 16

.

20 min read

Threat Background

Rhadamanthys is a newly emerged Information-Stealer that is written in C++. according to multiple reports[1] the malware has been active since late 2022.
In addition, the malware appears to masquerade itself as legitimate software such as AnyDesk installers[2], and Google Ads[3][13] to get the initial foothold.

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

Rhadamanthys
Also, the authors emphasize the malware's capabilities ranging from stealing digital coins, and system information collection, to execution of other processes such as Powershell.

In this article, I will investigate the Rhadamanthys stealer and reverse engineer the entire chain, from the first dropper to the malware itself.

As always, I will do it in a hybrid step-by-step tutorial and an actual presentation and will focus on the parts that I personally find more interesting(the ways the malware tries to evade detection).

1.
2.
3.

The Dropper

File hash: 89ec4405e9b2cab987f2e4f7e4b1666e

The dropper
The Rhadamanthys's dropper is a 32-bit file and similar to many droppers, it has relatively large entropy which indicates potentially packed content inside of it.

One of the relatively new features of PEstudio is the ability to check if the ASLR[4] feature is enabled. In my analysis, I always prefer to disable the ASLR so the addresses in IDA and in XDBG will be the same for tracking purposes.

In PEstudio, go to "optional-header" and then to the ASLR bar, then you can see under the "detail" column if it is false (disabled) or true (enabled).

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 will be decrypted during runtime.

Blob
The first activity the dropper do is to create a new heap

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:

1. - this function is responsible for returning an address containing the data to be written.
2. - a wrapper of memcpy

In the first iteration, the embedded blob will be written into the newly created heap

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

Decrypting the shellcode
Then, a call to *VirtualAlloc* will happen to create a newly allocated memory followed by memcpy to copy the shellcode from the heap to the new memory. Lastly, a *VirtualProtect* API call will be used to change the permission of the memory segment to *RWX*.

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

Decrypting the shellcode
The next thing we'll do is go to the address *004065A1* in the WinMain function (remember, ASLR is disabled so we can navigate easily in IDA and the debugger).

We could see that the value of the shellcode (that is dynamically located in the *EAX* register) is being transferred to another offset variable *42F6F0*.

Assign the shellcode address

Shellcode execution via Callback

After having a shellcode with EXECUTE permission, we need a way to execute it, in this case, the authors choose a cool trick in form of a Callback function.

The shellcode execution will go as the following:

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

ImmEnumInputcontext function in Microsoft documentation
The logic can be seen in the following pseudo-code

Shellcode execution
The reason for choosing this way is most likely to evade anti-virus products that rely on *CreateThread* \ *CreateRemoteThread* as a trigger point to scan addresses that may contain malicious content.

Shellcode entry point

Investigating the first shellcode

To investigate the shellcode we can choose one of the two:

1. Dump the entire allocated buffer and run it in Blobrunner[5]

2. Continue with the code dynamically (because why not?)

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”

Going to the memory map
2. Then, in the memory map, right click on the shellcode address and then “Dump Memory to File”

Dumping the shellcode
Then, drag and drop the dumped file in IDA.

To summarize the steps until now see the following graph

Fixing the shellcode: Defining functions

After the shellcode was loaded, we can see 5 functions that appear in the Function name bar. In addition, in the navigation bar, we can see the colors blue and brown.

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:

Defining functions
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:
leaveretrn

Defining functions
Now that we know the function boundaries, we can mark it all, and click “P”

Defining functions

Then, we can see that the brown code is now considered a function, and a new function *sub_29E* was added to the function name bar.

Defining functions
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 "*call 450028*" that suppose to take us to the address in *450028* which starts with "pop eax" and eventually calls to the function in the address *45029E* which in our case called *sub_29E*.
However, as we can see, statically we just see jibberish and it does not look like the dynamic view.

Defining as 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
2. Right click
3. Click on "Undefine"

Defining as code
Then, mark the 5 bytes and tell IDA to look at it as Code.

Defining as code
Defining as code
After doing it, we can see that the same data looks like the code from the debugger view

Defining as code
And as said, we can always turn it into a function of its own (because why not?)

Defining as function

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.

Defining as code and defining as function
After doing that, we now have 8 functions in the function name bar.

Function bar

Fixing the shellcode: Rebase the address

The last thing we need to do if we want to properly analyze the shellcode alongside the debugger is to match the addresses. To do it do the following:

1. Go to Edit
2. Segments
3. Rebase program

Rebase
4. Change the value to the value of the actual entry point of the shellcode in the debugger

5. Click OK

Rebase
And now we can see that the addresses statically and dynamically the same

Rebase
Finally, we can start and actually analyze the shellcode

Shellcode functionality

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. just jumps to
2. jump jumps to

sub_45029E is a larger function that contains multiple functions.

Shellcode functionality
sub_450249 This function access the Process Environment Block to get the address of *Kernel32.dll*. This behavior is traditional and happens in many shellcodes.

Get kernel32 address
sub_45036E This function gets 3 arguments

1. Kernel32 address
2. Hashes
3. An array that holds 4 functions

It then iterates through the kernel32 export functions and sends the names of the functions to another function named *sub_45040C***.** The only job of *sub_45040C* is to hash the function name it receives and return the hash.

Hashing function
Then, *sub_45036E* checks if the hashed function name matches the hash it got as an argument, if yes, it puts it in the array and sends it back to*sub_45029E*.
Overall the functions will be "*VirtualAlloc*, *LocalFree*, *LocalAlloc*, *VirtualFree"*

sub_450077
This function will decrypt the large data that is stored in our shellcode, and write it to the LocalAlloc we saw. This beginning of the decrypted data will look like this

Decrypting data
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*).

shellcode functionality
sub_45003A
This function will happen several times and it is basically a memcpy that copies data from one variable to the other.

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

copied data
And finally, in the address *00450365,* we have a "*call ebx*" that will take us into this our allocated memory in the offset *5BAB*, and as we can see, it's also another shellcode.

Jump to another shellcode

Summarize the first shellcode

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

Shellcode functionality
And from the following graph's point of view

Second shellcode decryption

The second shellcode aka Rhadamanthys loader

The main objective of this shellcode is to be the actual loader of the Rhadamanthys stealer. This shellcode has multiple evasion capabilities and we will observe some of them.

Note- In a similar way to the first shellcode, some fixes are needed.

Evasion Technique: Multiple Anti-Analysis

The Rhadamanthys loader contains large anti-analysis checks stolen from the al-khaser project[7]. This project was also used in the Bumblebee malware.

Some of the checks are checking for a virtual environment

Anti-analysis checks
Anti-analysis checks
Checks for specific users that could hint about a lab environment

Anti-analysis checks
Check for security-related DLLs

Anti-analysis checks
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?

According to Microsoft's documentation[9]: "Structured exception handling (SEH) is a Microsoft extension to C and C++ to handle certain exceptional code situations, such as hardware faults, gracefully."

The SEH is basically a linked list that has two pointers:

1. A pointer to the next SEH record
2. A pointer to the function that contains the code to deal with the error

Examples of errors are division by 0, and excessive string length.

Microsoft allows programmers to create their own exception handlers in order to manage errors by themselves.

How the loader uses it?

First, the loader gets the address of *ZwQueryInformationProcess*, then it saves it on another variable. Eventually, we enter the function named *sub_5978*.

Getting ZwQueryInformationProcess
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.

Iterating in
In *sub_5A5C* the loader set the hook in the desired location of the call to *ZwQueryInformationProcess*

Patch
So how the change looks like?

In the following image, we can see the call to *ZwQueryInformationProcess* that happens inside *KiUserExceptionDispatcher* from Ntdll as part of *KiUserExceptionDispatcher's* legitimate behavior.

After the change, we can see that the call was replaced to jump to a function in the loader that will perform the *ZwQueryInformationProcess* and will modify the *ProcessInformation* flag to be *6D* or *MEM_EXECUTE_OPTION_IMAGE_DISPATCH_ENABLE*.

Why does this flag matters?

This flag determines whether to allow execution outside the memory space of the loaded module. In other words, it enables exception handling to be performed on shellcode.

So how the exception handling will be managed?

Without being noticed, the initial dropper has registered an SEH record in the process memory with the name *_except_handler3*. Therefore, every exception that will be triggered by the shellcode will go there and will be managed by whatever logic the author decided.

This activity is most likely done to avoid raising suspicions if errors or exceptions anomalies will trigger.

The entire activity can be seen in the following graph

Manipulating the SEH

Evasion Technique: Avoiding error message

After controlling the exceptions, the loader will use the API call *SetErrorMode* with *0x8003* as an argument, this argument consists of the following three:

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

In other words, the loader doesn't want the system to display any error on the screen, and wants to handle them by himself.

Similar to controlling the exception handling, this is another maneuver of the loader to not raise any suspicions.

setErrorMode

Evasion Technique: Creating Mutex and impersonating a legitimate

The loader continues with creating a Mutex with the name that starts with "*Global\MSCTF.Asm.{digits}*".

Creating Mutex
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.

It first gets a handle to *ntdll.dll* and loads it to virtual memory, then, the loader gets the handle of the real *ntdll.dll* that is already loaded.

Check for hooks
It will then copy the bytes of the *SYSCALL* of *ZwProtectVirtualMemory* into another virtual memory in order to use it without explicitly using the *ZwProtectVirtualMemory* in ntdll address space.

Then, it will get the export table of both real and fake modules and will iterate on them. They will be compared using *memcmp*, and if they will found different, the loader will change the protection of the real function of ntdll 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
If we inspect it dynamically, this is a normal state when two functions are compared. We can see that the virtual address is different but the bytes are the same

Check for hooks
For learning purposes, I changed the first byte of the real function to start with E9. Then, the loader took us to the *memcpy* function that copied the data from the fake to the real to correct the change I made.

Disable hooks
Except for *ntdll.dll*, the loader will check the following DLLs:

1.
2.
3.

Check for hooks in other DLLs
The entire activity can be seen in the following graph (Was lazy so I just copy paste this from my previous blob)

Check for hooks logic

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.

In *sub_3DD4* we have two functions that will deal with the config decryption: *sub_28AA* and *sub_2911*.

sub_28AA

This function is basically just an RC4 algorithm

Config decryption
sub_2911

This function is also part of the decryption algorithm

Config decryption

When we step over *sub_2911* dynamically, we can see the data that hold the encrypted config at the third argument (address *42F6F8* in my case*).*

Config decryption
In our case, we can see that the C2 will be http://185[.]209.160.99/blob/top.mp4

Network

To start the network activity, the loader first collects two key pieces of information from the machine:

1. The default language using
2. The Locale using

Then, the same function will start to set the user-agent to send the data to the C2 which is the decrypted config we saw.

Collect information about the machine
Set the User-Agent
To communicate, the loader dynamically resolves multiple functions such as *socket*, *WSAIotcl*, and *CreateCompletionPort* to use the IOCP socket model.

Network activity
The loader uses *WSAIoctl* to invoke a handler for *LPFN_CONNECTEX* to use the *ConnectEx* function.

Getting ConnectEx
Eventually, the loader communicates with the APIs *WSARecv* & *WSASend*.

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 is being executed
2. Follow in dump the address of the second parameter aka
3. This buffer is a structure, and its second parameter is a pointer to the actual buffer that is sent to the C2.
4. To see it, just follow in dump

Observing data send to the C2
Observing data send 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
3. Spawn to execute the DLL with the export function “” which is a name of a legitimate export function of the printui.dll.

Loader goal

NSIS Module: The Rhadamanthys stealer

The Nsis module consists of two parts:

1. A loader (the Nsis module before unpacking)
2. The actual stealer

NSIS Loader

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

Nsis module command
The interesting thing about the NSIS loader is that there are many loaders out there, but their detection rate is very low!

Nsis loader low detection rate
For the loader behavior, the NSIS loader just allocates data using *LocalAlloc* and copies it to mapped memory using *MapViewOfFile* and *memmove*. Eventually, it will jump to the shellcode address.

Loader main goal
Due to time constraints, I will not display this shellcode, however, it is just a small shellcode that unpacks and inject into the memory the Rhadamanthys stealer itself.

Rhadamanthys stealer capabilities

Finally, we arrived at the stealer himself!!!

Disclaimer: because of not abling to dynamically analyze the sample when the C2 was on, I only got the stealer from the following tria.ge sandbox link[11].

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

Usage of SQLite

The malware can collect and extract data using SQLite

Sqlite

Target multiple browsers

The malware target the following browsers in their info-stealing activity:

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

Browsers

Target OpenVPN

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

OpenVPN

Target steam accounts

The malware appears to aim at Steam's config\loginusers.vdf which contains information about Steam's users.

Valve

Target FileZilla passwords

The malware search for FileZila's specific files:

1. recentservers.xml
2. sitemanager.xml

These two files contain the passwords and other data of the FTP accounts.

FileZilla

Target CoreFTP

CoreFTP

Target Discord

The malware collects information from the discord directories, possibly to extract further data.

Discord

Collecting Telegram data

The malware targets Telegram desktop data which is located in encrypted files (such as *D877F783D5D3EF8*) in the "*tdata*" directory.

Telegram

Collecting information from various email

The malware target the following email clients:

1. Foxmail
2. Outlook
3. The BAT

Emails

Extracting web credentials using Vaultcli functions

Vault activity

Target WinSCP

The malware target sensitive registry keys of the WinSCP in order to collect information.

WinSCP

Target CryptoCurrency entities

The malware target the following cryptocurrencies entities and wallets:

1. Dogecoin

2. Litecoin
3. Monero
4. Qtum
5. Armory
6. Bytecoin
7. Binance
8. Electron
9. Solar waller
10. Zap
11. WalletWasabi
12. Zcash
13. Ronin
14. Avana
15. OKX

Crypto
Querying registry keys for digital coming entities from Joe[

Resolving APIs dynamically

The stealer is resolving dynamically his APIs using the GetModuleHandle and GetProcAddress API calls.

Dynamic resolving

Evasion technique: Modify and possibly manipulate AVAST modules

The stealer uses the same code that was used in the loader to verify and unhook functions and the same function appears to aim for the AVAST-related modules *aswhook.dll* & *aswAMSI.dll.*

Check AVAST’s AMSI-related DLLs
More amsi-related functions and DLLs that are being targeted by the stealer are:

1. avamsicli.dll
2. amsi.dll
3. AmsiScanString
4. AmsiScanBuffer
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/8384322d609d7f26c6dc243422ecec3d40b30f29421210e7fba448e375a134f6

References

[1] https://threatmon.io/rhadamanthys-stealer-analysis-threatmon/

[2] https://mobile.twitter.com/JAMESWT_MHT/status/1610620178441568261

[3] https://mobile.twitter.com/1ZRR4H/status/1610590795278712832

[4] https://en.wikipedia.org/wiki/Address_space_layout_randomization

[5] https://github.com/OALabs/BlobRunner

[6] https://hex-rays.com/blog/igors-tip-of-the-week-49-navigation-band/

[7] https://github.com/LordNoteworthy/al-khaser

[8] https://elis531989.medium.com/the-chronicles-of-bumblebee-the-hook-the-bee-and-the-trickbot-connection-686379311056

[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

[13] https://twitter.com/1ZRR4H/status/1614728368334716932

[14] https://www.joesandbox.com/analysis/783578/0/html#

[15] https://blog.cyble.com/2023/01/12/rhadamanthys-new-stealer-spreading-through-google-ads/