# Reverse Engineering Snake Keylogger: Full .NET Malware Analysis Walkthrough

any.run/cybersecurity-blog/reverse-engineering-snake-keylogger/

Lena aka LambdaMamba

March 25, 2024

Reverse Engineering Snake Keylogger: Full .NET Malware Analysis Walkthrough

## Introduction

In order to understand malware comprehensively, it is essential to employ various analysis techniques and examine it from multiple perspectives. These techniques include behavioral, network, and process analysis during sandbox analysis, as well as static and dynamic analysis during reverse engineering.

I (Lena aka LambdaMamba), prefer to begin with sandbox analysis to understand the malware’s behavior. The insights from sandbox analysis provide a foundational understanding of what to anticipate and what specific aspects to investigate during the reverse engineering process. Recognizing what to look for is crucial in reverse engineering because malware authors often employ a myriad of tricks to mislead analysts, as will be demonstrated in this reverse engineering walkthrough. We will also be taking a look into how malware can be modded to make analysis easier.

Let’s dive right into it!

## Preparation for Reverse Engineering

This sample has 4 stages, dynamic code execution, code reassembly, obfuscation, steganography, junk code, and various other anti-analysis techniques. We’ll eventually get into fun things like modding the malware, so stay with me here!

The modded Snake Keylogger

The sandbox analysis of this Snake Keylogger was covered in my previous article Analyzing Snake Keylogger in ANY.RUN: a Full Walkthrough. From the sandbox analysis, I have a general understanding of what to look for during reverse engineering. These are some things I would look out for the Snake Keylogger during reverse engineering:

- Malware config: SMTP credentials used for data exfiltration
- Infostealing functionalities: Steals credentials from Browsers and Apps
- Defense evasion techniques: Move to Temp after execution, Checks and kills certain processes
- Anti-analysis techniques: Execution stops if not connected to internet, Self-Deletion after execution

For this Reverse Engineering walkthrough, I will be conducting static and dynamic analysis with the use of a decompiler and debugger. Thus, it is important to prepare an isolated environment that is specifically prepared for malware analysis.

Malware Analysis Environment used in this Reverse Engineering Walkthrough:

Recommended setups for safety:

- Disable Network Adapters to prevent accidental connection to the network
- Minimize resource sharing between guest and host, disable shared clipboard, drag and drop, etc.

Streamline Snake Keylogger analysis
with the ANY.RUN sandbox

Sign up for free

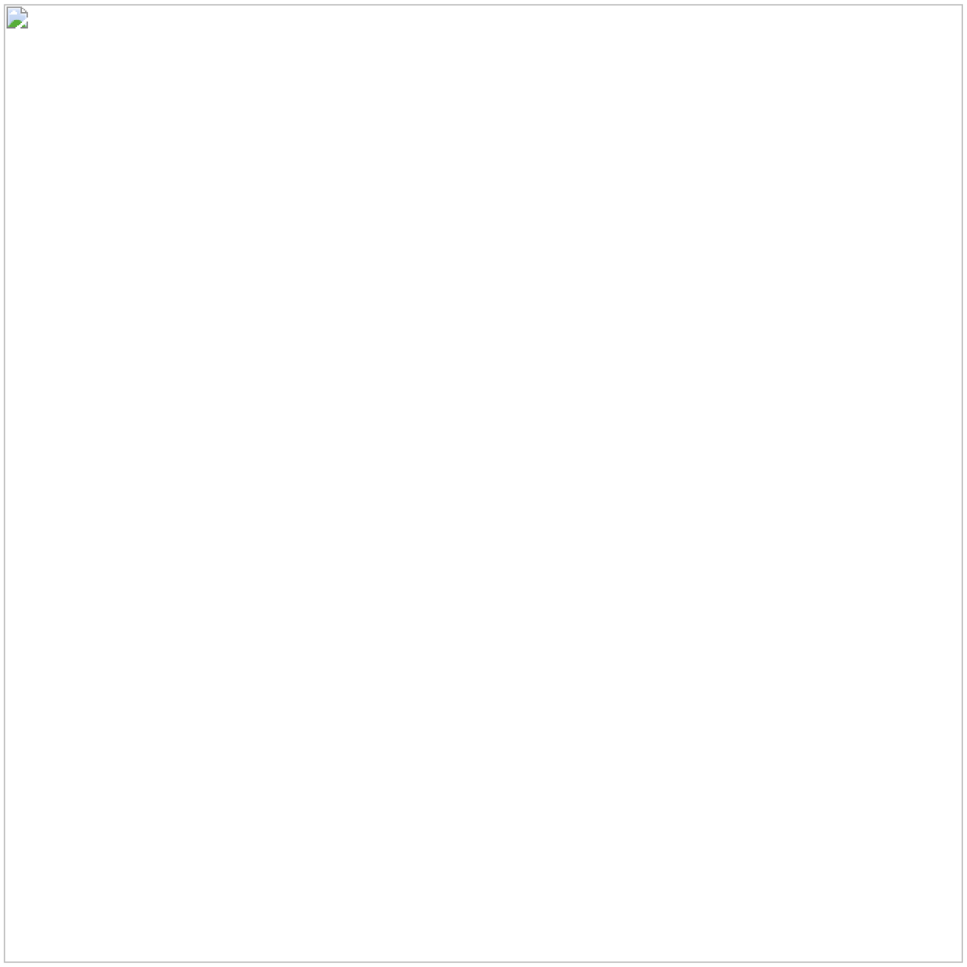
## Stage 1: pago 4094.exe

### Determining the File Attributes

The executable “pago 4094.exe” is identical to the one covered in my Analyzing Snake Keylogger in ANY.RUN: a Full Walkthrough, and has the following attributes:

- SHA1 hash of “A663C9ECF8F488D6E07B892165AE0A3712B0E91F”
- MIME type of “application/x-dosexec”
- PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows

Putting “pago 4094.exe” through DIE (Detect it Easy) showed that the Library is “.NET(v4.0.30319)[-]” and the Linker is the “Microsoft Linker(48.0)[GUI32]”.


“pago 4094.exe” on DIE shows the Library and Linker

Since “pago 4094.exe” is a 32-bit .NET malware, 32-bit dnSpy will be used for Reverse Engineering. This executable was opened as “mKkHQ (1.0.0.0)” in dnSpy.

The executable is opened as“mKkHQ (1.0.0.0)” on dnSpy

## Static Analysis with the Decompiler

The entry point in a .NET executable is the method where execution starts upon launch, so I will start the analysis from the entry point. We can go to the Entry Point by right clicking “mKkHQ” in the Assembly Explorer, and selecting “Go to Entry Point” in the dropdown menu. The entry point is under “Program”, and *Main()* could be observed in the decompiled code.

The entry point that contains the Main function

This application was filled with code for a “Airplane Travelling Simulation” application. However, I know from the Snake Keylogger Sandbox Analysis that functionalities related to Airplane Travelling simulations were never observed.

A bunch of irrelevant code related to an “Airplane Travelling” application

This means that the payload for the Snake Keylogger is loaded somewhere before the Airplane Travelling simulation application starts. A suspicious block of code was observed, lines 91~96 and 100~104 in *Form1, InitializeComponent()*. It uses *GetObject()* to get the data from a resource named “Grab”, and will be decrypted using the key *ps* in the For loop.

A suspicious block of code before the “Airplane Travelling” application starts

“Grab” is under the Resources of “mKkHQ”, and the contents were a bunch of gibberish when viewed in the memory.

The contents of “Grab” in the memory

## Determining the Junk Code

I commented out the suspicious block of code, re-compiled, and saved as “pago 4094_mod.exe”.

Commenting out the suspicious block of code

“pago 4094_mod.exe” was executed in an ANY.RUN sandbox, which can be found here. It opens an application that asks for input text files, and will start an airplane simulation application complete with a GUI.

The “Airplane Travelling” application on the ANY.RUN Sandbox

This means that the suspicious block of code (lines 91~96 and 100~104) are responsible for the malicious activities.

## Dynamic Analysis with the Debugger

I set the breakpoints, and started the debugger. (IMPORTANT: Before starting the debugger, please make sure your malware analysis environment is isolated, and does not contain any important data as we will be executing malicious code.)

The byte array *data2* contains the contents of “Grab” from the *GetObject(“Grab”)* as expected when the program is run until Line 93.

The contents of byte array *data2* when run until Line 93

The contents of *data2* can be observed in the memory, which is identical to what we previously saw in “Grab” under Resources.

The contents of byte array *data2* in memory when run until Line 93

After the For loop responsible for decrypting the code, I could see that *data2[0]* contained 0x4D (“M” in ASCII) and *data2[1]* contained 0x5A (“Z” in ASCII). This indicates that it’s the start of the DOS header (“MZ..”).

The contents of byte array *data2* after decryption, which shows the start of the DOS header

Viewing *data2* in memory showed the DOS header (indicated by “MZ”), DOS Stub (indicated by “This program cannot be run in DOS mode”), and the PE header (indicated by “PE”).

The contents of byte array *data2* in memory after decryption

The binary data is in the byte array *data2,* and *Assembly.Load(data2)* loads the binary data as an assembly into the current application domain. *Activator.CreateInstance(type, args)* creates the instance and will start execution of the loaded assembly.

Use of *Assembly.Load* to load the assembly

The loaded module is called “Aads”, and can be seen under the dnSpy Module tab, which I saved as a DLL. This “Aads.dll” will be the next stage, namely stage 2.

The module “Aads.dll” can be observed under the Modules Tab

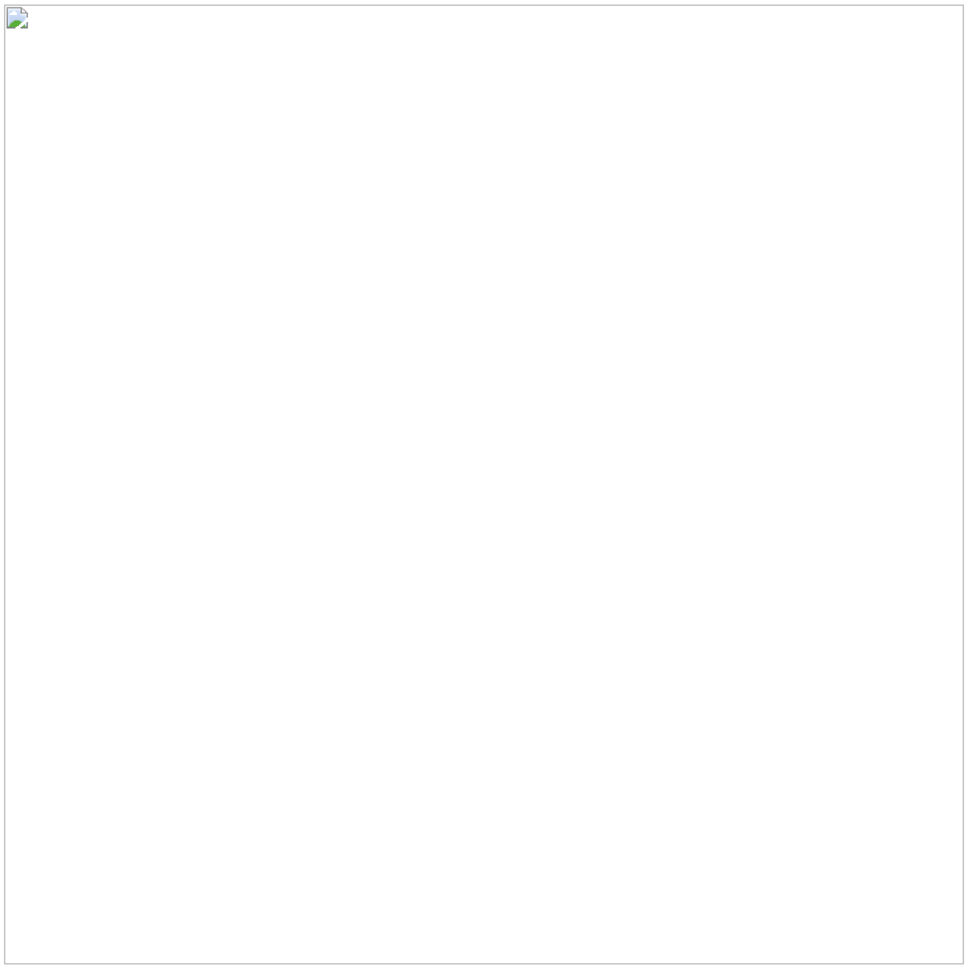
## Stage 2: Aads.dll

### Determining the File Attributes

The “Aads.dll” has the following attributes:

- SHA1 hash of “244000E9D84ABB5E0C78A2E01B36DDAD8958D943”
- MIME type of “application/x-dosexec”
- PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows

Putting “Aads.dll” through DIE (Detect it Easy) showed that it’s a DLL (Dynamic Link Library), where the Library is “.NET(v2.0.50727)[-]” and the Linker is the “Microsoft Linker(8.0)[DLL32]”.


“Aads.dll” on DIE shows the Library and Linker

### Static Analysis with the Decompiler

I opened “Aads.dll” in dnSpy 32-bit like the previous stage. There’s a lot of sorting and searching functions, but no code related to infostealing was observed. Based on the static analysis of “Aads.dll”, it seems like some data will be rearranged into data that is responsible for the next stage.

Decompiling the “Aads.dll” on dnSpy reveals a bunch of search and sort functions

Back in Stage 1, *GetObject()* was used to get data from the Resources that was decrypted into the Stage 2 DLL. Thus, I decided to look for *GetObject()*, in the hopes of finding the source data of the next stage.

After looking around, I found *GetObject(x10)*. Here, it uses the image data from a file, and this resource name is specified inside string variable *x10*.

Usage of *GetObject()*

## Dynamic Analysis with the Debugger

I set the breakpoints on Line 475 and 477, and ran the debugger. I could see that *x10* was “ivmSL” when run until Line 475.

Obtaining the filename used for *GetObject()* with the debugger

The “ivmsL” is under Airplane.Travelling’s resources. A noisy grainy image which looked like steganography could be observed.

The steganography image “ivmsL” that is used in *GetObject()*

This is the contents of the file “ivmsL” when viewed in the memory.

The contents of “ivmsL” in the memory

This image bitmap data will then go through various byte array searching and sorting algorithms. These include Heapsorts, Quicksorts.

Contents of the byte array *array* before the sorting is complete

Contents of the byte array *array* in memory before the sorting is complete

After various searching and sorting operations, I could see that *array[0]* contained 0x4D (“M” in ASCII) and *array[1]* contained 0x5A (“Z” in ASCII). This indicates that it’s the start of the DOS header (“MZ..”).

The contents of byte array *array* when sorting is complete, which shows the start of the DOS header

Viewing *array* in memory showed the DOS header (indicated by “MZ”), DOS Stub (indicated by “This program cannot be run in DOS mode”), and the PE header (indicated by “PE”).

The contents of byte array *array* in memory when sorting is complete

The loaded module is called “Tyrone”, and can be seen under the dnSpy Module tab, which I saved as a DLL. This “Tyrone.dll” will be the next stage, namely stage 3.

The module “Tyrone.dll” can be observed under the Modules Tab

See how ANY.RUN can help your security team

Schedule a demo

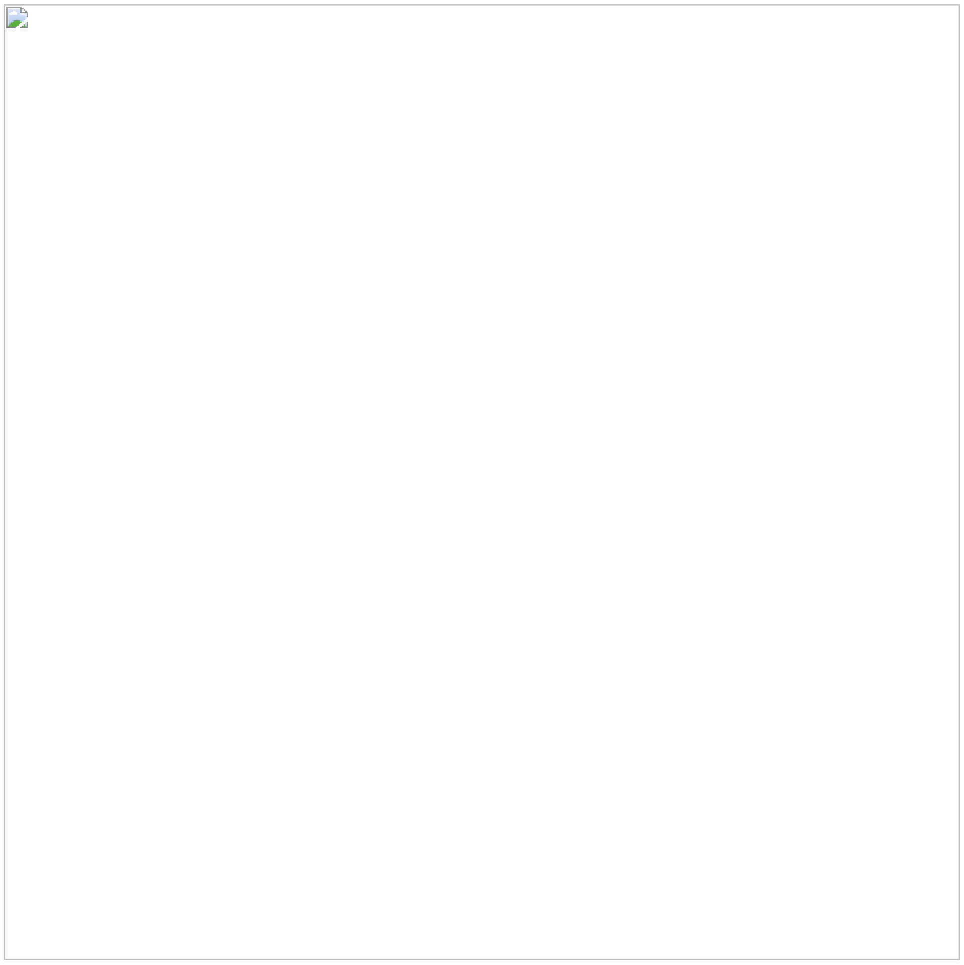
## Stage 3: Tyrone.dll

### Determining the File Attributes

The “Tyrone.dll” has the following attributes:

- SHA1 hash of “6523D31662B71A65533B11DA299240F0E8C1FF2C”
- MIME type of “application/x-dosexec”
- PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows

Putting “Tyrone.dll” through DIE (Detect it Easy) showed that it’s a DLL (Dynamic Link Library), where the Library is “.NET(v2.0.50727)[-]”, Compiler is “VB.NET(-)[-]” and the Linker is “Microsoft Linker(8.0)[DLL32]”.


“Tyrone.dll” on DIE shows the Library, Compiler, and Linker

## Deobfuscation

“Tyrone.dll” was opened in dnSpy 32-bit, and is heavily obfuscated. The Class and function names were not human-readable, and the code was difficult to analyze.

Decompiling the “Tyrone.dll” on dnSpy reveals obfuscated code

I deobfuscated “Tyrone.dll” using .NET Reactor Slayer, with all options selected.

Deobfuscating the “Tyrone.dll”

## Static Analysis with the Decompiler

After deobfuscating, the code was much easier to read. After looking around, a bunch of junk code related to a “pandemic simulation” was observed. I know they are junk code, because these functionalities were never observed back in my sandbox analysis.

A bunch of junk code were observed in “Tyrone.dll”

After looking around the deobfuscated code, there was no function that performed the infostealer activities. This means that there is likely a next stage.

Thus, I looked for *GetObject()* again. As expected, there was *GetObject(),* which gets the data from a resource whose name is specified by string variable *string_0*.

Usage of *GetObject()*

Based on static analysis of the deobfuscated code, a bunch of sorting takes place on the byte array data from *GetObject()*.

## Dynamic Analysis with the Debugger

*UmHYCAPJIp()* and *\u0020* in the obfuscated code corresponds to *method_0()* and *string_0* respectively in the deobfuscated code. Breakpoints were set on the obfuscated code, and after running until the breakpoint, *\u0020* was "wHzyWQnRZ".

Obtaining the filename used for *GetObject()* with the debugger

The “wHzyWQnRZ” is under the Resources of “Tyrone”, and the contents were a bunch of gibberish when viewed in the memory.

The contents of “wHzyWQnRZ” in the memory

I let the program run, and after a bunch of byte array rearranging, I could see that *\u0020[0]* contained 0x4D (“M” in ASCII) and *\u0020[1]* contained 0x5A (“Z” in ASCII). This indicates that it’s the start of the DOS header (“MZ..”).

The contents of byte array *\u0020* after the sorting is complete, which shows the start of the DOS header

Viewing *\u0020* in memory showed the DOS header (indicated by “MZ”), DOS Stub (indicated by “This program cannot be run in DOS mode”), and the PE header (indicated by “PE”). This is the next stage executable, namely stage 4, and I saved this as an EXE file.

The contents of byte array *\u0020* in memory when sorting is complete

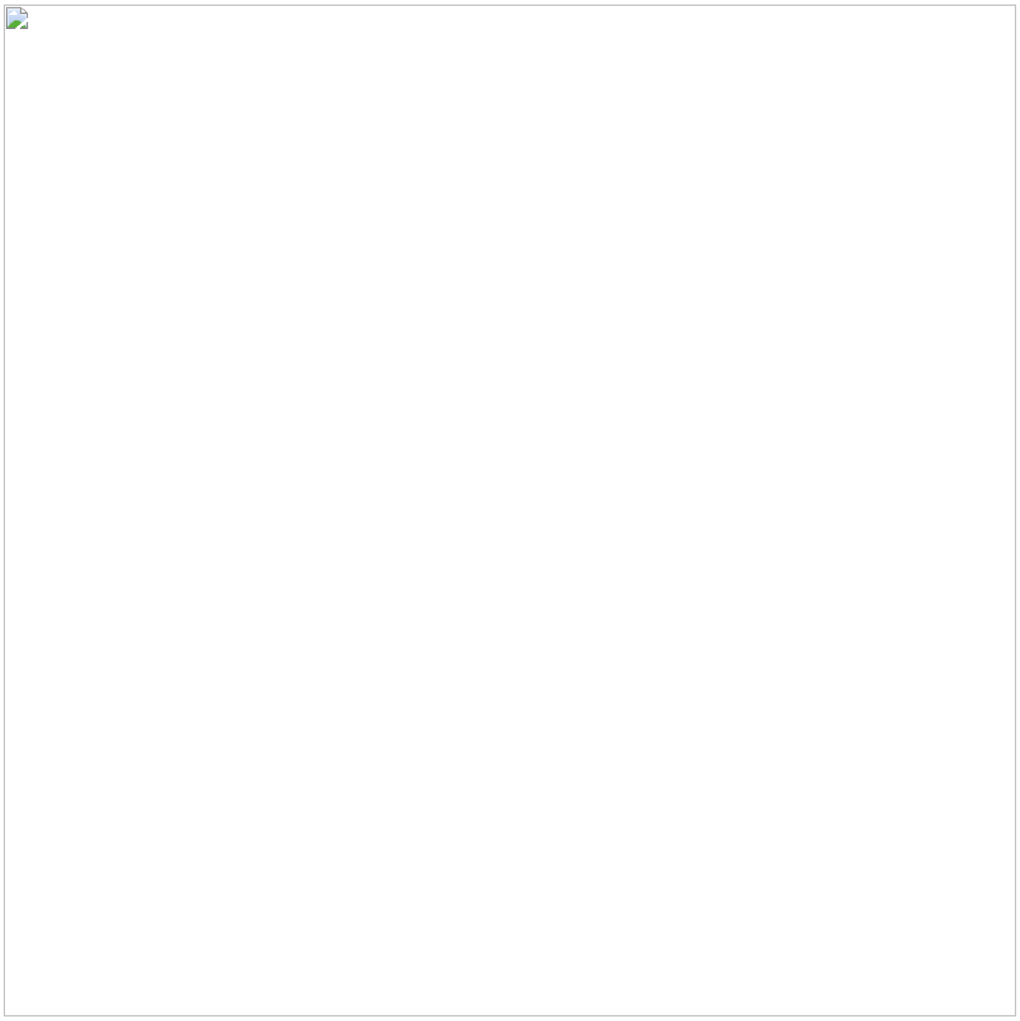
## Stage 4: lfwhUWZlmFnGhDYPudAJ.exe

### Determining the File Attributes

This stage's executable was called "lfwhUWZlmFnGhDYPudAJ.exe", and has the following attributes:

- SHA1 Hash of "86BE2A34EACBC0806DBD61D41B9D83A65AEF69C5"
- MIME type of "application/x-dosexec"
- PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows

Putting "lfwhUWZlmFnGhDYPudAJ.exe" through DIE (Detect it Easy) showed that the Library is ".NET(v4.0.30319)[-]", Compiler is "VB.NET(-)[-]", and Linker is "Microsoft Linker(80.0)[GUI32,admin]".


“lfwhUWZlmFnGhDYPudAJ.exe” on DIE shows the Library, Compiler and Linker

## Sandbox Analysis

Detonating “lfwhUWZlmFnGhDYPudAJ.exe” in an ANY.RUN sandbox showed that it was detected as a Snake Keylogger. The task can be found here.

The overview of “lfwhUWZlmFnGhDYPudAJ.exe” in an ANY.RUN sandbox

Analyze malware in the ANY.RUN sandbox for free

Sign up now

## Deobfuscation

“lfwhUWZlmFnGhDYPudAJ.exe” was opened in dnSpy 32-bit, and was heavily obfuscated. The class and function names were not human-readable, and the code was difficult to follow. I deobfuscated it using .NET Reactor Slayer again with all the options selected.

Decompiling the “lfwhUWZlmFnGhDYPudAJ.exe” on dnSpy reveals obfuscated code

Deobfuscating the “lfwhUWZlmFnGhDYPudAJ.exe”

## Renaming the Class and Functions

After deobfuscation, the code was much easier to read. After looking around, the infostealing functionalities were finally found. In order to better understand and follow the code, I did a lot of manual renaming of the class and functions. All the modified names have “lena_” prefix.

The left shows the deobfuscated code, and the right shows the deobfuscated and manually renamed code

## Analyzing the Snake Keylogger Code

### Extracting the Malware Config

In this section, I will be analyzing the Snake Keylogger code that is responsible for the malicious activities.

The malware config was observed in Class6, however it was encrypted with a hard coded encryption key.

The malware config that contains the encryption key, and the encrypted SMTP information

The config is decrypted using *lena_crypt(),* with the hardcoded key *lena_key*.

The algorithm used for decrypting the SMTP information

I converted this decryption code to Python. The first 8 bytes of the MD5 hash of "BsrOkyiChvpfhAkipZAxnnChkMGkLnAiZhGMyrnJfULiDGkfTkrTELinhfkLkJrkDExMvkEUCxUkUGr" is used for the decryption key, namely "6fc98cd68a1aab8b". It uses this key to decrypt the Base64 decoded config string with DES (ECB mode).

```
from Crypto.Cipher import DES
from Crypto.Hash import MD5
import base64



def lena_decrypt_snake(text, key_string):
  try:
      key = MD5.new(key_string.encode('ascii')).digest()[:8]
      cipher = DES.new(key, DES.MODE_ECB)
      decrypted_data = cipher.decrypt(base64.b64decode(text))
      decrypted_text = decrypted_data.decode('ascii', errors='ignore')
      padding_len = decrypted_data[-1]
      if padding_len < len(decrypted_data):
          return decrypted_text[:-padding_len]
      return decrypted_text


  except Exception as e:
      return str(e)


lena_key = "BsrOkyiChvpfhAkipZAxnnChkMGkLnAiZhGMyrnJfULiDGkfTkrTELinhfkLkJrkDExMvkEUCxUkUGr"
print("lena_sender_email_addr: ", lena_decrypt_snake("I22WW+qzjWDd9uzIPosYRadxnZcjebFO", lena_key))
print("lena_sender_email_pw: ", lena_decrypt_snake("MrZp4p9eSu2QFqjr3GQpbw==", lena_key))
print("lena_SMTP_server: ", lena_decrypt_snake("XHGvc06cCeeEGUtcErhxrCgs7X5wecJ1Yx74dJ0TP3M=", lena_key))
print("lena_receiver_email_addr: ", lena_decrypt_snake("I22WW+qzjWDd9uzIPosYRadxnZcjebFO", lena_key))
print("lena_SMTP_port: ", lena_decrypt_snake("oXrxxBiV5W8=", lena_key))
print("lena_padding: ", lena_decrypt_snake("Yx74dJ0TP3M=", lena_key))
```

*The Python code that decrypts the SMTP information*

When I ran the Python code with the strings in Class6, the malware config revealed itself. These were the SMTP information used for exfiltrating the data, which included the sender email address, password, SMTP server, recipient email address, and SMTP port. These are the same credentials observed in the Snake Keylogger Sandbox Analysis.

The decrypted SMTP information obtained from the Python code

## The Main Functionalities

Let’s take a look at what the actual payload of the Snake Keylogger does. It starts off by moving the executable to the Temp directory, and naming it after the current time.

The code responsible for moving the executable to Temp

After that, it will collect data from various places, including Browsers (e.g. Chrome, Comodo, Opera, Microsoft Edge, etc.), Applications (e.g. Outlook, Discord, etc.).

The calls from *Main()* that collects data from various places, mostly browsers

For example, this is the code segment responsible for collecting login data from Chrome. The login data file for Chrome is in "\Google\Chrome\User Data\Default\Login Data", and is a SQLite database file that contains saved login information.

The code responsible for collecting Chrome login data

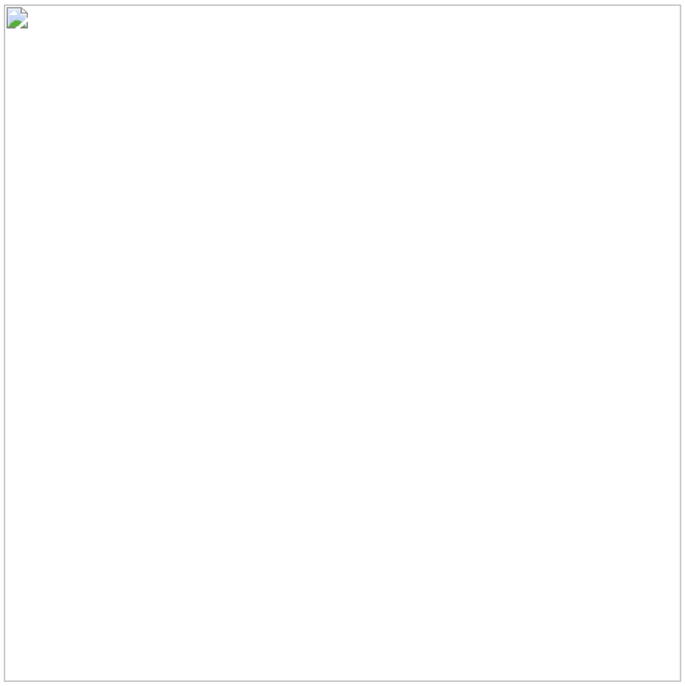
This is the code segment responsible for collecting saved login data from Microsoft Edge. The login data file for Microsoft Edge is in "\Microsoft\Edge\User Data\Default\Login Data", and is a SQLite database file that contains saved login information.

The code responsible for collecting Microsoft Edge login data

It also collects authentication tokens from Discord, and this is the code segment responsible for it. Discord stores its LevelDB database files in “\discord\Local Storage\leveldb”, and may contain the user’s authentication token. With the authentication token, the attacker can gain unauthorized access to the victim’s Discord account without a password.

The code responsible for collecting Discord tokens

Finally, it will exfiltrate all this collected information, via FTP, SMTP, or Telegram.


The calls from *Main()* that exfiltrates the collected data

This is the code responsible for exfiltrating with FTP. If the Snake Keylogger is configured to use FTP, it creates a FTP request. The FTP credentials are hard-coded into the Snake Keylogger code (“lena_padding_1” and “lena_padding_2” in this case), however, this Snake Keylogger sample is configured to use SMTP, so the code does not include the FTP credentials. Once the stolen data is prepared, it uploads it to the server with FTP.

The code responsible for FTP exfiltration

This Snake Keylogger sample is configured to use SMTP by default, and this is the code responsible for exfiltrating with SMTP. If the Snake Keylogger is configured to use SMTP, it constructs an email with *MailMessage*, and prepares the email sender address, receiver address, subject, body, and the stolen data as a text attachment. It then uses the SMTP credentials hardcoded in the malware configuration to authenticate and exfiltrate via SMTP with *smtpClient.Send().*

The code responsible for SMTP exfiltration

This is the code responsible for exfiltrating with Telegram. If the Snake Keylogger is configured to use Telegram, it creates the Telegram API request URL, with the bot token (“lena_padding_4” in this case) and chat ID (“lena_padding_5” in this case) where the data will be sent. However, this Snake Keylogger sample is configured to use SMTP, so the code does not include the Telegram bot token and chat ID.

The code responsible for Telegram exfiltration

## Other Interesting Functionalities

Here are some other interesting code segments in the Snake Keylogger. This code segment searches and kills processes related to security and monitoring. These processes include antiviruses (Norton, F-Prot, Avira, Kaspersky aka Avp, etc.), network monitoring tools (Wireshark, Snort, etc.), debuggers (OllyDbg, etc.), firewalls (ZoneAlarm, Outpost, BlackIce, etc.).

The code responsible for searching and killing processes related to security and monitoring

This code is responsible for taking screenshots. It uses a *Graphics* object to capture the entire screen, saves this as a PNG in the "SnakeKeylogger" folder, and exfiltrates it.

The code responsible for taking screenshots

This code is responsible for stealing and exfiltrating clipboard data.

The code responsible for stealing and exfiltrating clipboard data

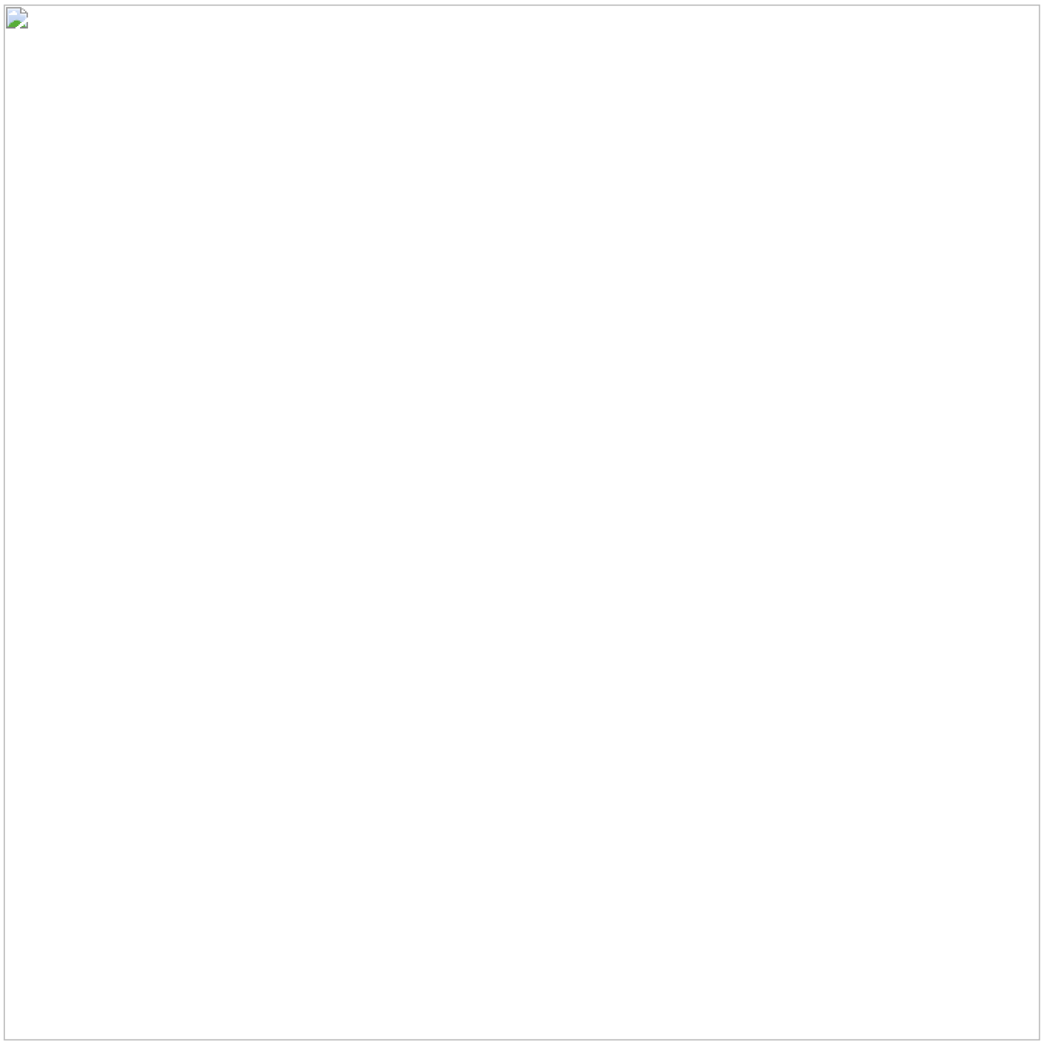
There is a Keylogger class that is responsible for the keylogging activities.

The Keylogger class with Keylogging functions

It monitors keystrokes with the event handler for the *KeyDown* and *KeyUp* event.

Code segment related to keystroke monitoring

It also identifies the keystrokes, and checks for special keys like Backspace, Tab, Enter, Space, End, Delete, etc.


Code segment related to keystroke identification

## Modding the Malware

Before we get into this section, please understand that we are only going to mod the malware to make analysis easier. Please do not abuse this knowledge!

### Modding Anti-Analysis Functionalities

If I tried to run the Stage 4 payload in an environment not connected to the internet, an exception will be thrown and will exit.

An exception is thrown

I edited the code, so that the Snake Keylogger will not terminate execution depending on internet connectivity, and not check the IP with *checkip.dyndns.org* as observed in the Snake Keylogger Sandbox Analysis.

The code responsible for connectivity checks are commented out

Upon execution, it will delete itself as observed in the Snake Keylogger Sandbox Analysis. Thus, I’ve modded it so it doesn’t delete itself to make debugging easier.

The code responsible for self-deletion is commented out

Upon execution, it will move itself to Temp as also observed in Snake Keylogger Sandbox Analysis. Thus, I also modded it so it does not move itself to Temp.

The code responsible for moving to Temp is commented out

It will now continue execution without being connected to the internet, not delete itself or move itself to Temp after this modification. This will make dynamic analysis in the isolated malware analysis environment easier.

Conduct sandbox analysis of Snake and other malware
in ANY.RUN

Sign up for free

## Modding the SMTP credentials

I wrote a script in Python that encrypts the malware config. I made a throwaway Outlook account, where the SMTP server is *smtp-mail.outlook.com.*

The first 8 bytes of the MD5 hash of “BsrOkyiChvpfhAkipZAxnnChkMGkLnAiZhGMyrnJfULiDGkfTkrTELinhfkLkJrkDExMvkEUCxUkUGr” is used for the decryption key, namely “6fc98cd68a1aab8b”. It then uses this key to encrypt the string with DES (ECB mode), and Base64 encoding is applied to the encrypted string.

```
from Crypto.Cipher import DES
from Crypto.Hash import MD5
from Crypto.Util.Padding import pad
import base64


def lena_encrypt_snake(plaintext, key_string):
   try:
       key = MD5.new(key_string.encode('ascii')).digest()[:8]
       cipher = DES.new(key, DES.MODE_ECB)
       padded_text = pad(plaintext.encode('ascii'), DES.block_size)
       encrypted_data = cipher.encrypt(padded_text)
       encrypted_text = base64.b64encode(encrypted_data).decode('ascii')
       return encrypted_text
   except Exception as e:
       return str(e)


lena_key = "BsrOkyiChvpfhAkipZAxnnChkMGkLnAiZhGMyrnJfULiDGkfTkrTELinhfkLkJrkDExMvkEUCxUkUGr"
print("lena_sender_email_addr: ", lena_encrypt_snake("<REDACTED>@outlook.com", lena_key))
print("lena_sender_email_pw: ", lena_encrypt_snake("<REDACTED>", lena_key))
print("lena_SMTP_server: ", lena_encrypt_snake("smtp-mail.outlook.com", lena_key))
print("lena_receiver_email_addr: ", lena_encrypt_snake("<REDACTED>@proton.me", lena_key))
print("lena_SMTP_port: ", lena_encrypt_snake("587", lena_key))
```

*The Python code that encrypts the SMTP information*

I ran the SMTP information through the encryption code.

The encrypted SMTP information obtained from the Python code

I added that into the malware config, and changed TLS to “True” as Outlook SMTP requires STARTTLS/TLS on port 587.

The modified and encrypted SMTP information is added to the config

## Modding for Customization

I changed the executable icon to my profile picture, and the file details using Resource Hacker.

Using Resource Hacker to customize the File details

I also added some functionality that changes the background picture, so I know when the Snake Keylogger has executed. I added my signature digital white snake wallpaper under the Resources.

Lena’s Digital White Snake Wallpaper

I added a new function *lena_snek()* that gets the image from Resources, temporarily saves it as a PNG in /Temp, and sets that as the background picture when the executable starts.

Code segment for background change

I also added a new function *lena_save_txt()* that saves the collected information on the Desktop as “Passwords.txt” and “User.txt”, so I can observe what was stolen without viewing the PCAP or have access to the exfiltration email.

Code segment for saving collected information as a text file

## Executing the Modded Malware in a Sandbox

I detonated the modded Snake Keylogger in the ANY.RUN Sandbox, which can be found here.

Before detonating the modded Snake Keylogger on an ANY.RUN Sandbox

Upon execution, the background changed to my digital white snake wallpaper. This indicates that the modded Snake Keylogger has successfully executed.

Detonating the modded Snake Keylogger on an ANY.RUN Sandbox changes the background

Shortly after, “Passwords.txt” and “User.txt” showed up on the Desktop.

The collected information is saved as a text file on the Desktop

The contents of “Passwords.txt” and “User.txt” on the Desktop can be seen below. These are credentials I saved onto Google Chrome before detonating the Snake Keylogger.

“Passwords.txt” on the left, “User.txt” on the right in an ANY.RUN sandbox

After executing the Snake Keylogger, I received the “Passwords.txt” and “User.txt” from the throwaway Outlook email.

The email I received from the throwaway email account

This is consistent with the text file saved onto the Desktop, as well as what was observed back in the PCAP of the sandbox analysis. However, as Outlook's SMTP uses TLS/STARTTLS, the contents of these text files are encrypted and cannot be viewed in the PCAP.

“Passwords.txt” on the left, “User.txt” on the right

The email can be seen under “Sent Items” in my throwaway Outlook account which was used to send the email.

The email from the sender’s perspective

The malware config can be found in ANY.RUN’s *Malware Configuration.*

The Malware Configuration for the Snake Keylogger on ANY.RUN

## Conclusion

This walkthrough demonstrated the process of reverse engineering a .NET malware, specifically the Snake Keylogger. It also highlighted the importance of employing multiple analysis techniques.

Starting with sandbox analysis to gain a general understanding of the malware’s expected behavior is crucial for reverse engineering, as it guides us on what to look for especially when faced with various anti-analysis techniques employed by malware authors to deter analysts.

My prior sandbox analysis, Analyzing Snake Keylogger in ANY.RUN: a Full Walkthrough, provided me with insights into what to expect. Thus, despite encountering challenges such as junk code, obfuscation, multiple stages, steganography, dynamic code execution, and code reassembly, I knew what to look for during the reverse engineering process.

Finally, I also demonstrated how malware can be modded to make analysis easier.

## About ANY.RUN

ANY.RUN is a cloud-based malware analysis platform designed to support the work of security teams. It boasts a user base of 400,000 professionals who utilize the platform for threat analysis on Windows and Linux cloud virtual machines.

With ANY.RUN, you security team can enjoy:

- **Instant detection:** ANY.RUN can detect malware and identify various malware families using YARA and Suricata rules within approximately 40 seconds of file upload.
- **Hands-on analysis:** In contrast to many automated tools, ANY.RUN offers interactive capabilities, allowing users to engage directly with the virtual machine through their browser. This feature helps prevent zero-day exploits and advanced malware that can bypass signature-based detection.
- **Low cost:** ANY.RUN's cloud-based nature makes it a budget-friendly solution for businesses, eliminating the need for setup or maintenance efforts from the DevOps team.
- **Training functionality:** ANY.RUN's user-friendly interface enables even junior SOC analysts to quickly learn how to analyze malware and extract indicators of compromise (IOCs).

Get a personalized demo of ANY.RUN for your team.

Schedule a call with us →

## Appendix 1: IOC

| Name | pago 4094.exe | Aads.dll |
|---|---|---|
| MD5 | 1A0F4CC0513F1B56FEF01C815410C6EA | 60A14FE18925243851E7B89859065C24 |
| SHA1 | A663C9ECF8F488D6E07B892165AE0A3712B0E91F | 244000E9D84ABB5E0C78A2E01B36DD |
| SHA256 | D483D48C15F797C92C89D2EAFCC9FC7CBE0C02CABE1D9130BB9069E8C897C94C | 6CDEE30BA3189DF070B6A11A2F80E84 |
| SSDEEP | 12288:PXPZDbCo/k+n70P4uR87fD0iBTJj1ijFDTwA:hOz+IPz6/PF1ihDTwA | 1536:kEMoTcQA2YULtLNvpQy59F/ok19 |

## Appendix 2: Snake Keylogger Config Decryption Python Code

```
from Crypto.Cipher import DES
from Crypto.Hash import MD5
import base64



def lena_decrypt_snake(text, key_string):
  try:
      key = MD5.new(key_string.encode('ascii')).digest()[:8]
      cipher = DES.new(key, DES.MODE_ECB)
      decrypted_data = cipher.decrypt(base64.b64decode(text))
      decrypted_text = decrypted_data.decode('ascii', errors='ignore')
      padding_len = decrypted_data[-1]
      if padding_len < len(decrypted_data):
          return decrypted_text[:-padding_len]
      return decrypted_text


  except Exception as e:
      return str(e)
```

Lena aka LambdaMamba

I am a Cybersecurity Analyst, Researcher, and ANY.RUN Ambassador. My passions include investigations, experimentations, gaming, writing, and drawing. I also like playing around with hardware, operating systems, and FPGAs. I enjoy assembling things as well as disassembling things! In my spare time, I do CTFs, threat hunting, and write about them. I am fascinated by snakes, which includes the Snake Malware!

Check out:

lena-aka-lambdamamba

Lena aka LambdaMamba
Cybersecurity analyst and researcher
I am a Cybersecurity Analyst, Researcher, and ANY.RUN Ambassador. My passions include investigations, experimentations, gaming, writing, and drawing. I also like playing around with hardware, operating systems, and FPGAs. I enjoy assembling things as well as disassembling things! In my spare time, I do CTFs, threat hunting, and write about them. I am fascinated by snakes, which includes the Snake Malware!
Check out:
What do you think about this post?

4 answers

- Awful
- Average
- Great

No votes so far! Be the first to rate this post.

0 comments