Demonstration of Malware Analysis Tools

14-11-2022

—

Mohd Ayan Khan - 20BCE0644
Sujay Kumar - 20BDS0294
Gokul Raj - 20BCE2743

Submitted to: Prof Anand M

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Overview

A malware requires to be in a spectrum from the attacks that's happening around the internet, These different types of malware should be analyzed and processed using some malware tool which gives us a traditional tool, cutter, a tool which uses reverse engineering as a concept of analyzing malware. We have used this tool and analyzed every malware which has a specific SHA code, Entropy of a file and no API calls that took place.

Introduction

Cutter is an advanced reverse engineering platform powered by Rizin. It has all the reverse engineering features like hexdump, graph view and so on...We are now using this cutter tool to predict which type of malware it is and what class it falls on...

Aims and Objectives

1. To find the Malware and what exactly does the malware does

2. To execute the malware by malware tool and see the functions and type of functions its calling

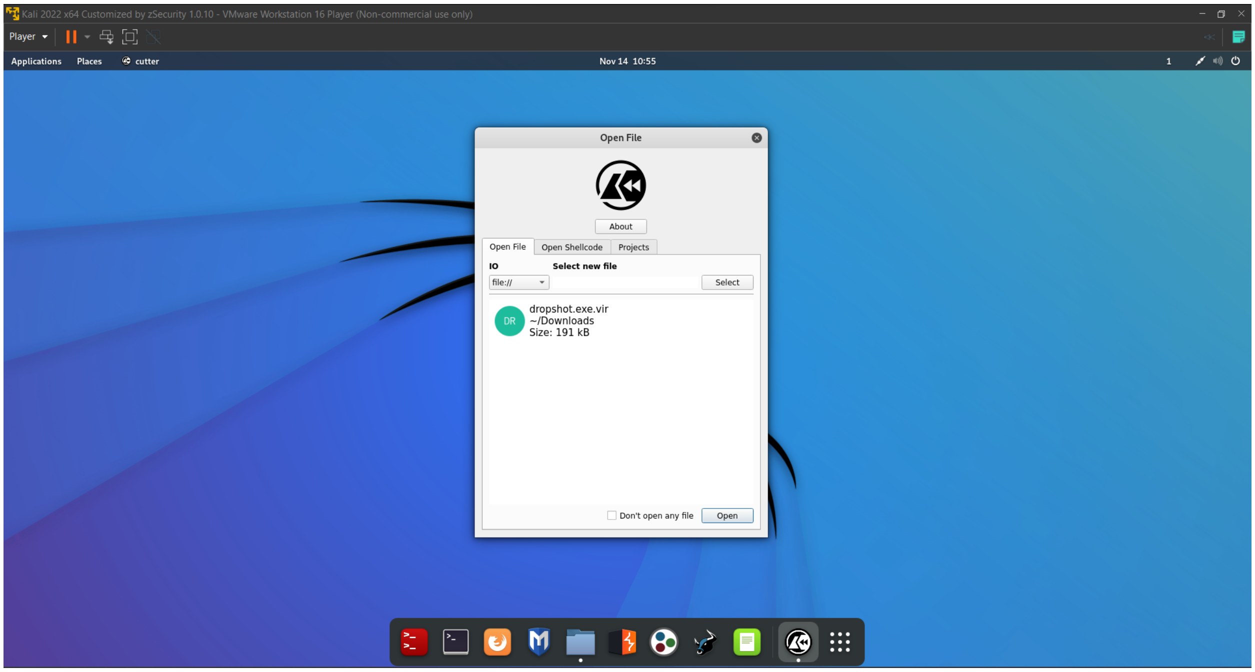
3. By using Machine Learning take all the data which we collected and start make a data-set with the raw data

4. Make a machine learning model which detects malware and the class of malware.

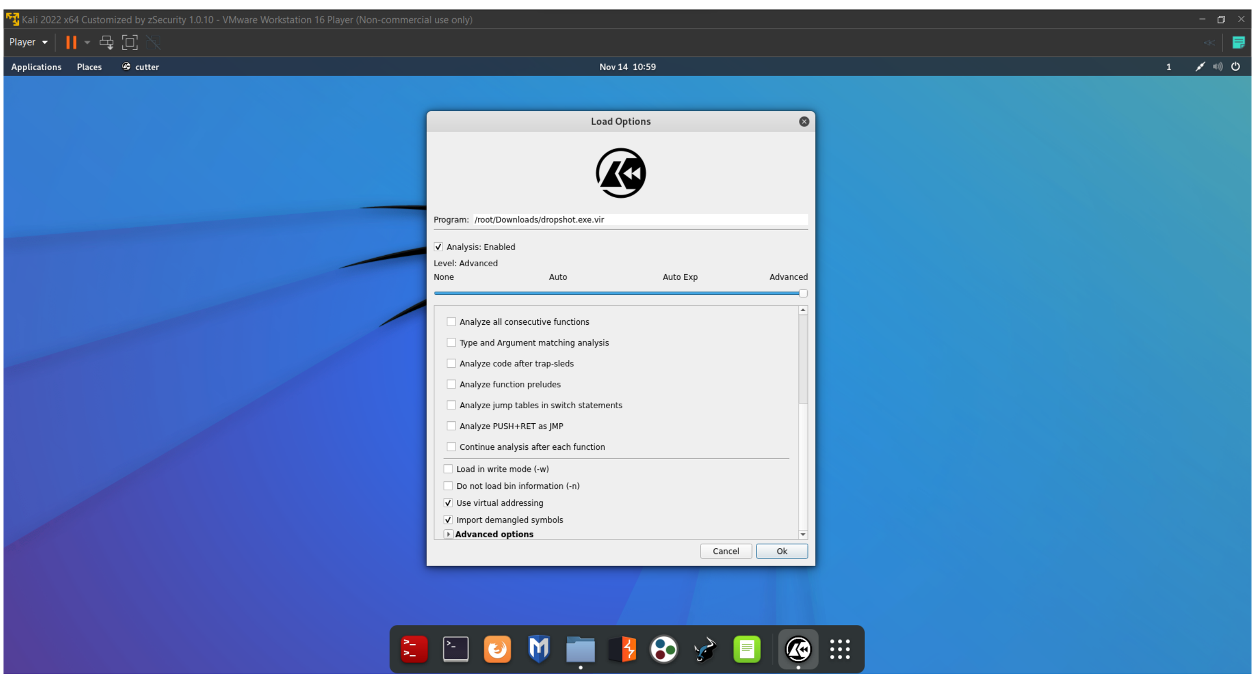
5. Deploy the Machine Learning model into the servers to detect malware.

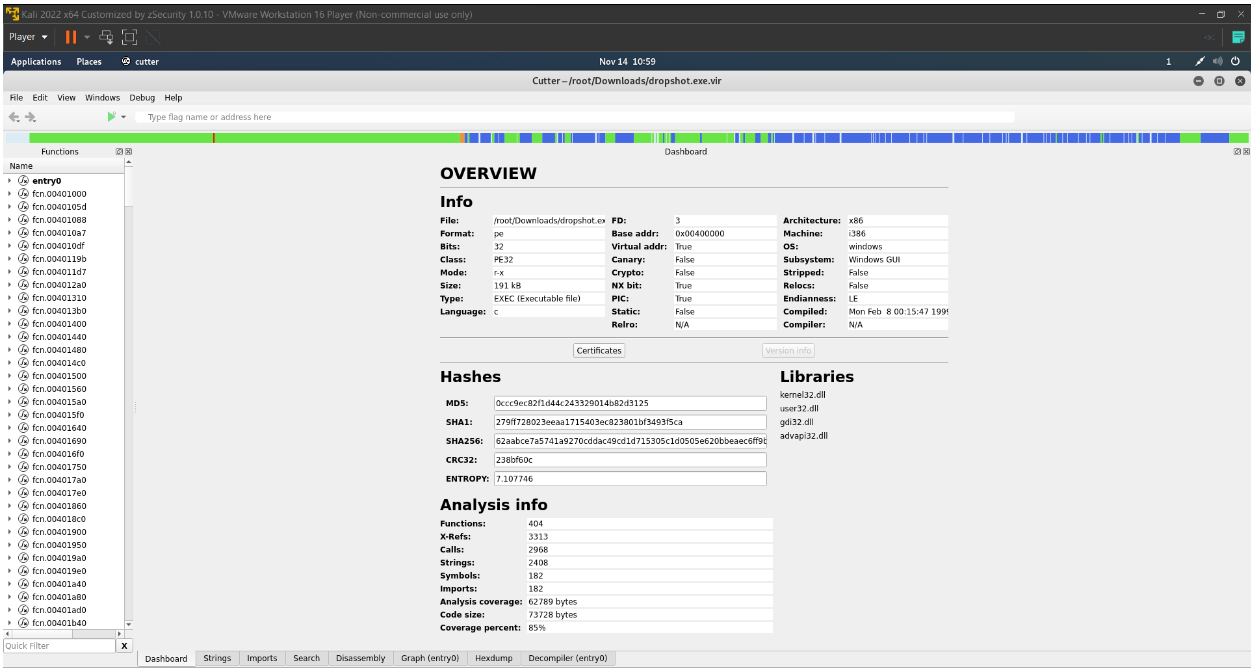
Proposed Methodology

As mentioned above, we will be using the Cutter tool based on the Radare2 framework.



This is the Cutter tool





We can see the whole information of the malware file being used (Dropshot Malware).

Basic static analysis

We started our analysis of a malware sample by statically inspecting the binary. A simple static analysis can occasionally determine whether a file is dangerous, reveal details about its behavior, and aid in our comprehension of the situation. Although simple and rapid, basic static analysis is often ineffective against complex malware.

Entropy

Entropy is a metric measuring how effectively information is stored. Entropy is the measurement of unpredictability in a set of values, to put it simply (data). Different programmes calculate a file's entropy in a similar way. Typically, it ranges from 0.0 to 8.0. Entropy is a trustworthy indicator of whether a file is packed, compressed, or contains packed or compressed data. A binary that is packed will likely have a high entropy value. A binary or some of its components are likely compressed or packed if a file has an entropy of 6.8 or higher.

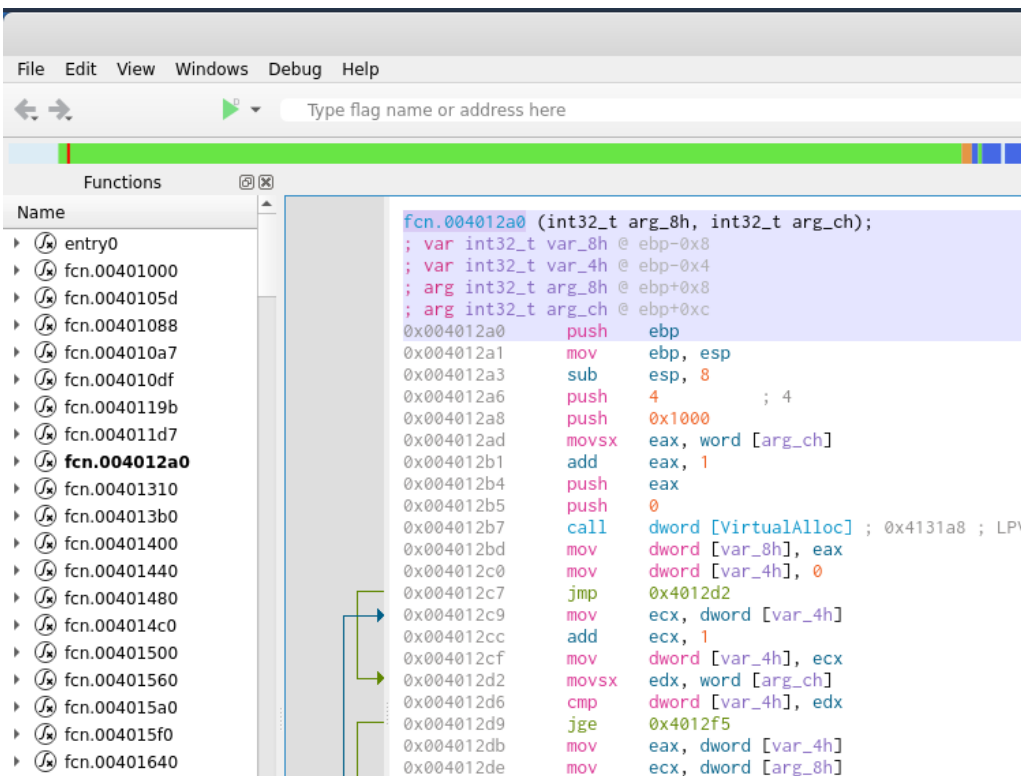
The file we are using has an entropy of 7.1, which is a very good example of compressed data.

Understanding the strings decryption process

We discovered that the file decrypts its embedded strings using a really simple technique. This function passed muster in our examination mainly because it was used before **“LoadLibraryA”** and **“GetProcAddress”** and was called frequently throughout the code. Therefore, it appeared to us as a method of dynamically loading libraries and functions in order to make analysis more difficult. a strategy that is very common among malware writers.

The decryption function, which can be found at 0x4012a0, seems to have two inputs.

This is the function that decrypts the strings.



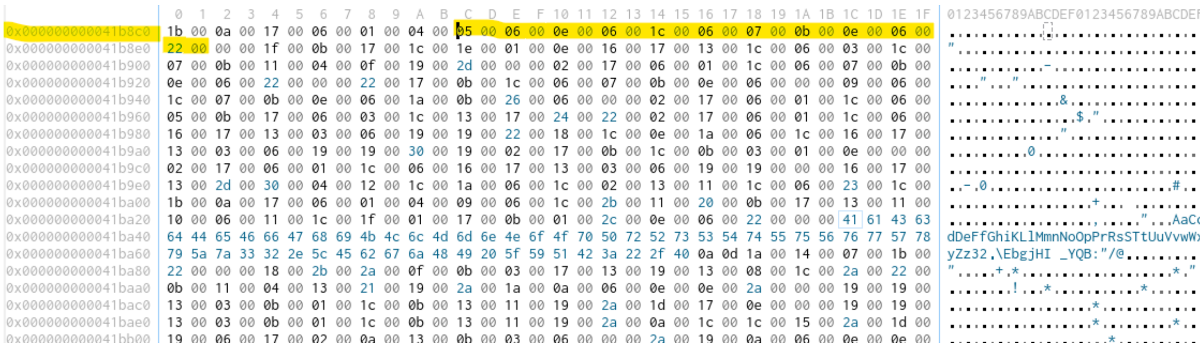
We can see that the output of strings decrypter (eax) is being passed along with an additional parameter, 1, to a different function at 0x4013b0.

This is the graph of the strings decrypter function.

Analyzing the decryption function

We already know that there are two arguments given to this function. An address is the first, followed by a number. The integer parameter is kept in the variable arg ch, and the address argument is kept in arg 8h. We can observe that VirtualAlloc allocates a buffer with the size of arg ch+1 in the first block, starting at 0x4012a0. The allocated buffer's address is then assigned to local 8h.

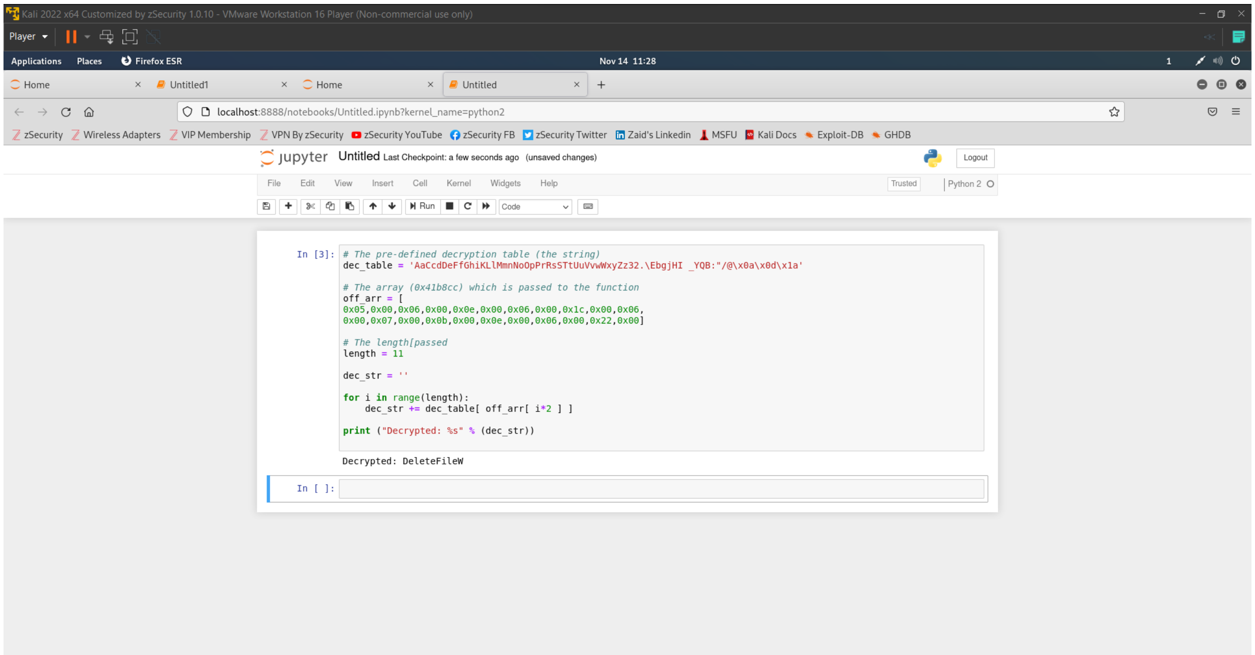
After that, we can see that local 4h has been given the value zero. The beginning of a loop is the following block. We can see that edx is given the integer stored at arg ch, and that edx then compares the integer to local 4h. Now we know that local 4h is a loop index and arg ch is some form of length or size. Now that we are aware of the functions of both the two local variables and one of the two arguments, we must comprehend the contents of the address that is supplied via arg 8h. Our string decryption method was being fed the value 0x41b8cc, as we could see. Now let's search for this address using the Hexdump widget. To look for a flag or an address, simply type this address into the textbox in the upper area. This half-word array of numbers, which begins at 0x41b8cc and ends at 0x0041b8e1, can be seen.



Code for the strings decryption:

```
# The pre-defined decryption table (the string)

dec_table = 'AaCcdDeFfGhiKLlMmnNoOpPrRsSTtUuVvwWxyZz32.\EbgjHI
_YQB:"/@\x0a\x0d\x1a'


# The array (0x41b8cc) which is passed to the function

off_arr = [

0x05,0x00,0x06,0x00,0x0e,0x00,0x06,0x00,0x1c,0x00,0x06,

0x00,0x07,0x00,0x0b,0x00,0x0e,0x00,0x06,0x00,0x22,0x00]


# The length[passed

length = 11


dec_str = ''


for i in range(length):

    decr_str += dec_table[ off_arr[ i*2 ] ]


print ("Decrypted: %s" % (dec_str))
```

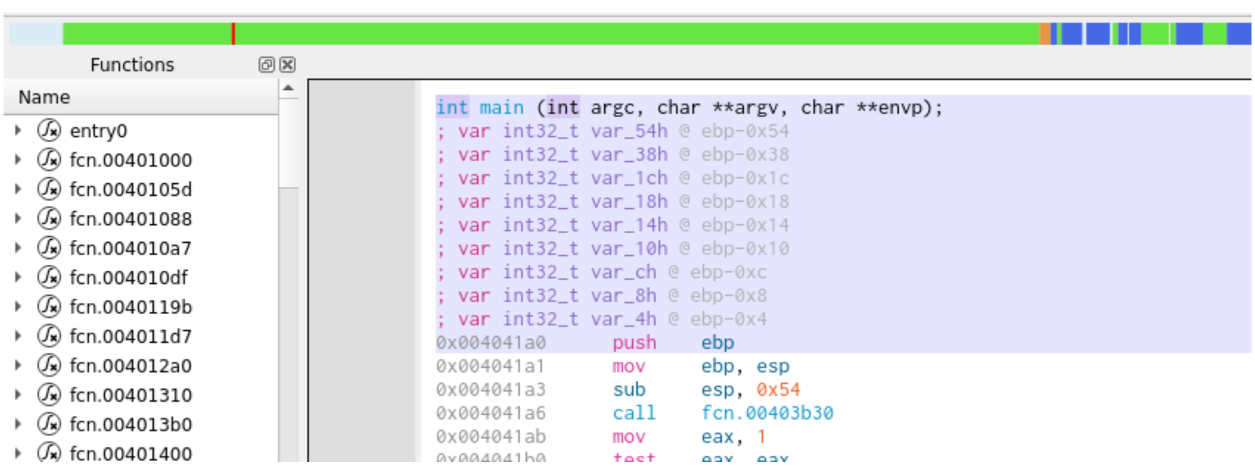
Output:



We can see that the string was successfully decrypted, yielding the API function "DeleteFileW."

The main() function:

Since the main() function is one of the essential ideas in programming, we are familiar with the function's role in programmes. We'll use the Graph mode to navigate the main flow in search of the resource decryption routine. We can observe that the main function's opening block calls a function located at 0x403b30.



By double-clicking this line, we may access the graph of the sizable function fcn.00403b30. As we work our way through this method, we'll encounter some absurd Windows API calls with wrong arguments. Dropshot utilizes anti-emulation; for instance, this function engages in anti-emulation.

```
fcn.00403b30 ();
; var int32_t var_38h @ ebp-0x38
; var int32_t var_30h @ ebp-0x30
; var int32_t var_2ch @ ebp-0x2c
; var int32_t var_28h @ ebp-0x28
; var int32_t var_24h @ ebp-0x24
; var int32_t var_20h @ ebp-0x20
; var int32_t var_1ch @ ebp-0x1c
; var int32_t var_18h @ ebp-0x18
; var int32_t var_14h @ ebp-0x14
; var int32_t var_10h @ ebp-0x10
; var int32_t var_ch @ ebp-0xc
; var int32_t var_8h @ ebp-0x8
; var int32_t var_1h @ ebp-0x1
0x00403b30      push    ebp
0x00403b31      mov     ebp, esp
0x00403b33      sub     esp, 0x38
0x00403b36      mov     byte [var_1h], 0
0x00403b3a      push    0
0x00403b3c      push    0
0x00403b3e      push    0
0x00403b40      push    0
0x00403b42      push    0
0x00403b44      push    0
0x00403b46      push    0
0x00403b48      push    0
0x00403b4a      push    0
0x00403b4c      push    0
0x00403b4e      push    0
0x00403b50      push    0
0x00403b52      push    0
0x00403b54      push    0
0x00403b56      call    dword [CreateFontA] ; 0x413014 ; HFONT CreateFontA(int nHeight, int nWidth, int nEscapement, int nOrientation, int fnWeight, DWORD fdwItalic, DWORD fdwUnderline, DWORD fdwStrikeOut, DWORD fdwCh
0x00403b5c      mov     dword [var_24h], eax
0x00403b5f      cmp     dword [var_24h], 0
0x00403b63      je      0x403e7f
0x00403b69      push    0
0x00403b6b      push    0
0x00403b6d      push    0
0x00403b6f      push    0
0x00403b71      push    0
0x00403b73      call    dword [CallWindowProcA] ; 0x413254 ; LRESULT CallWindowProcA(FARPROC lpPrevWndFunc, HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
0x00403b79      mov     dword [var_10h], eax
0x00403b7c      cmp     dword [var_10h], 0
```

Anti-Emulation

The emulators of anti-malware programmes are tricked using anti-emulation techniques. One of the most crucial elements of many security products are the emulators. They are employed, among other things, in the analysis of shellcode and the behavior of malware. By imitating the target architecture's instruction set, the running environment, and dozens or even hundreds of well-known API functions, it simulates the program's workflow. All of this is done to trick malware into "thinking" a target user actually executed it in a genuine environment.

The aim of many anti-emulation strategies used by malware developers is to trick a general or even a particular emulator. The most popular method, which Dropshot's fcn.00403b30 also uses, is the use of unusual or undocumented API calls. This method can be strengthened by passing invalid inputs (such as NULL) to an API function that, in a real context, should result in an Access Violation exception.

We are facing a branch as main calls the fcn.00403b30. Taking inspiration from Cutter's Disassembly widget, here is the assembly:

```
0x004041a6    call    fcn.00403b30
0x004041ab    mov     eax, 1
0x004041b0    test    eax, eax
0x004041b2    je      0x40429d
0x004041b8    push    4           ; 4
0x004041ba    push    0x1000
0x004041bf    push    0x208       ; 520
0x004041c4    push    0
```

As you can see, the test eax, eax followed by je... is essentially verifying whether eax equals 0. As a result, the code would never branch to 0x40429d. The software transferred the value 1 to eax one instruction earlier, therefore 0x40429d will never be invoked.

Decrypting the resource

Code:

```
import cutter
import zlib

# Rotating lambda to the right
def rot_right(val, r_bits, max_bits): return \
  ((val & (2**max_bits-1)) >> r_bits % max_bits) | \
  (val << (max_bits-(r_bits % max_bits)) & (2**max_bits-1))


def decode_strings(verbose=True):
  if verbose:
    print("\n%s\n\tStarting the decode of the encrypted strings\n%s\n\n" %
       ('~'*60, '~'*60))

  # Declaration of decryption-table related variables
  decryption_table = 0x41BA3C
  decryption_table_end = 0x41BA77
  decryption_table_len = decryption_table_end - decryption_table
  decryption_function = 0x4012A0

  # Analyze the binary to better detect functions and x-refs
  cutter.cmd('aa')

  # Rename the decryption function
  cutter.cmd('afn decryption_function %d' % decryption_function)
```

```
    # Dump the decryption table to a variable
    decryption_table_content = cutter.cmdj(
        "pxj %d @ %d" % (decryption_table_len, decryption_table))

    # Iterate x-refs to the decryption function
    for xref in cutter.cmdj('axtj %d' % decryption_function):
        # Get the arguments passed to the decryption function: length and encrypted string
        length_arg, offsets_arg = cutter.cmdj('pdj -2 @ %d' % (xref['from']))

        # String variable to store the decrypted string
        decrypted_string = ""

        # Guard rail to avoid exception
        if (not 'val' in length_arg):
            continue

        # Manually decrypt the encrypted string
        for i in range(0, length_arg['val']):
            decrypted_string += chr(decryption_table_content[cutter.cmdj(
                'pxj 1 @ %d' % (offsets_arg['val'] + (i*2)))[0]])

        # Print the decrypted and the address it was referenced to the console
        if verbose:
            print(decrypted_string + " @ " + hex(xref['from']))

        # Add comments to each call of the decryption function
        cutter.cmd('CC Decrypted: %s @ %d' % (decrypted_string, xref['from']))

# This function was added in the 2nd part of the series about dropshot
def decrypt_resource(verbose=True):
    if verbose:
        print("\n%s\n\tStarting the decryption of the resource\n%s\n" %
              ('~'*60, '~'*60))
    # Get information on all resources in JSON format
    rsrcs = cutter.cmdj('iRj')
    rsrc_101 = {}

    # Locate resource 101 and dump it to an array
    for rsrc in rsrcs:
        if rsrc['name'] == 101:
            rsrc_101 = cutter.cmdj("pxj %d @ %d" %
                                   (rsrc['size'], rsrc['vaddr']))

    # Decompress the zlibbed array
    decompressed_data = zlib.decompress(bytes(rsrc_101))

    decrypted_payload = []
```

```
    # Decrypt the payload
    for b in decompressed_data:
        decrypted_payload.append((ror(b, 3, 8)))

    # Write the payload (a PE binary) to a file
    open(r'./decrypted_rsrc.bin', 'wb').write(bytearray(decrypted_payload))

    if verbose:
        print("Saved the PE to ./decrypted_rsrc.bin")


decode_strings()
decrypt_resource()

# Refresh the interface to load changes
cutter.refresh()
```

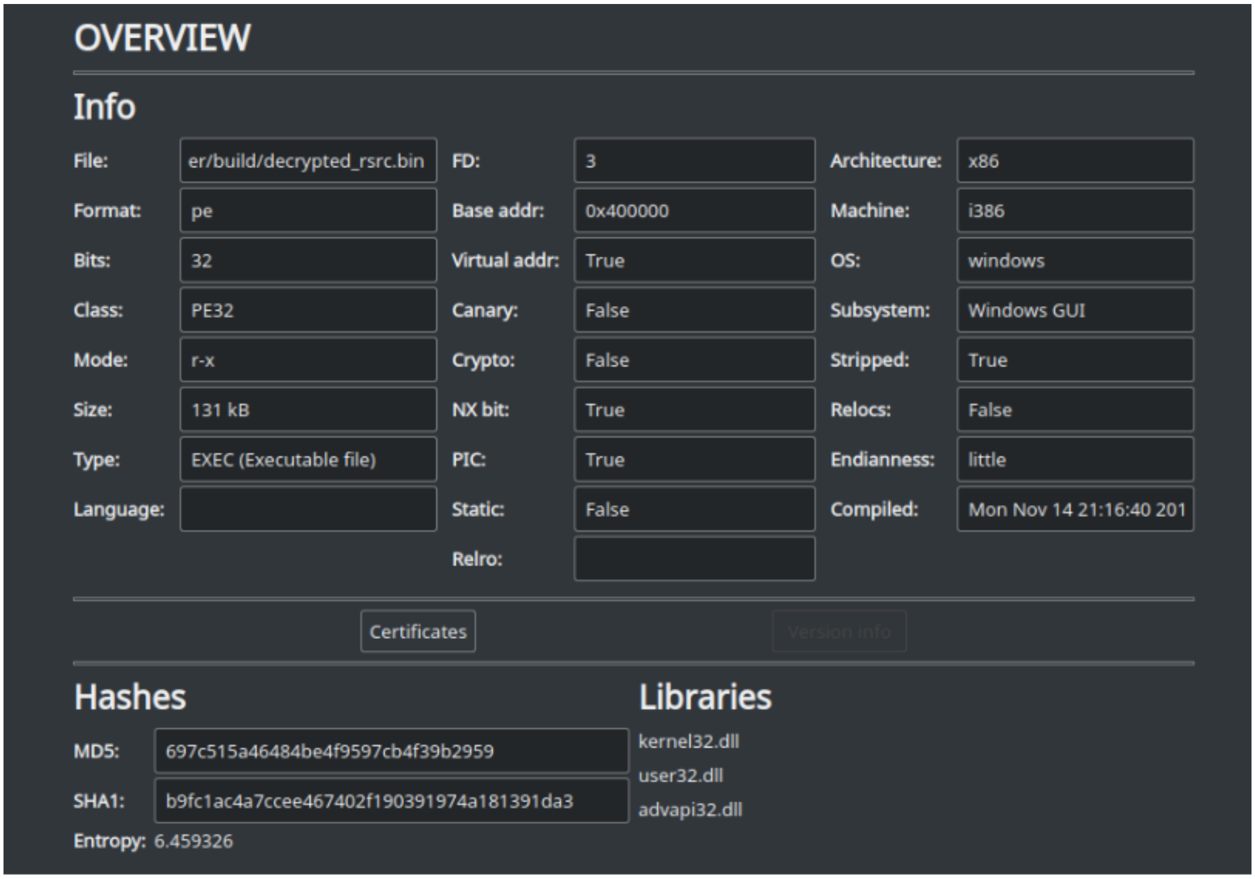
Output:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     Starting the decode of the encrypted strings
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


Kernel32.dll @ 0x4013c3
ntdll.dll @ 0x4013de
ZwResumeThread @ 0x40140a
ZwClose @ 0x40144a
ZwGetContextThread @ 0x40148a
NtSetContextThread @ 0x4014ca
CreateProcessW @ 0x40150a
GetModuleFileNameW @ 0x40156a
CreateFileW @ 0x4015aa
ReadFile @ 0x4015fa
WriteProcessMemory @ 0x40164a
Shell32.dll @ 0x40169b
SHGetSpecialFolderPathW @ 0x4016b4
Advapi32.dll @ 0x4016fb
RegOpenKeyW @ 0x401714
Advapi32.dll @ 0x40175b
RegCloseKey @ 0x401774
DeleteFileW @ 0x4017aa
Advapi32.dll @ 0x4017eb
RegQueryInfoKeyW @ 0x401804
Advapi32.dll @ 0x40186b
RegQueryValueExW @ 0x401884
GetTempPathW @ 0x4018ca
NtWriteVirtualMemory @ 0x40190a
WriteFile @ 0x40195a
RtlSetProcessIsCritical @ 0x4019aa
Psapi.dll @ 0x4019eb
GetModuleBaseNameA @ 0x401a04
OK @ 0x4039c7

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     Starting the decryption of the resource
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Saved the PE to ./decrypted_rsrc.bin
```

In [ ]:

After successfully running, our script "Saved the PE to./decrypted rsrc.bin."

The last step is to open decrypted rsrc.bin in a fresh instance of Cutter to confirm that it is, in fact, a PE file and that we did not somehow corrupt it.



The file was identified as PE by Cutter, and it appears that the code was correctly interpreted. The Wiper module of Dropshot is this binary that we just decrypted and saved; on its own, this particular piece of malware is quite intriguing.

Results and discussion:

Here comes to an end about decrypting Dropshot with Cutter and r2pipe. We got familiar with Cutter, radare2 GUI, and static analysis .We analyzed the decryption function and wrote a decryption script in r2pipe's Python binding. We came to know how main function code plays a vital role and some interesting things about Anti-Emulation.We also analyzed some components of APT33's Dropshot, an advanced malware.

Conclusion and Future Works:

Hence after decrypting the Dropshot,we are willing to extend our work by analyzing different kinds of malware and make a result of comparison of each malware so that we can understand more about the cutter tool .After testing on different malware we can analyze how efficient is cutter tool and we can come to know some new features of it.In future we are willing to publish our work.

References:

[1]Alrammal M, Naveed M, Sallam S, Tsaramirsis G. Malware analysis: Reverse engineering tools using santuko linux. Materials Today: Proceedings. 2022 Jan 1;60:1367-78.

[2]Waliulu RF, Alam TH. Reverse Engineering Analysis Forensic Malware WEBC2-DIV. Sinkron: jurnal dan penelitian teknik informatika. 2018 Sep 22;3(1):113-9.

[3]Le, D.T., Dinh, D.T., Nguyen, Q.L.T. and Tran, L.T., 2022. A Basic Malware Analysis Process Based on FireEye Ecosystem. *Webology (ISSN: 1735-188X)*, *19*(2).

[4]Cappers BC, Meessen PN, Etalle S, Van Wijk JJ. Eventpad: Rapid malware analysis and reverse engineering using visual analytics. In2018 IEEE Symposium on Visualization for Cyber Security (VizSec) 2018 Oct 22 (pp. 1-8). IEEE.

[5]Monnappa KA. Learning Malware Analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows malware. Packt Publishing Ltd; 2018 Jun 29.

[6]Zimba A, Simukonda L, Chishimba M. Demystifying ransomware attacks: reverse engineering and dynamic malware analysis of wannacry for network and information security. Zambia ICT Journal. 2017 Dec 11;1(1):35-40.

[7]Riyana A, Santoso B, Hartono R. Trojan malware analysis using reverse engineering method in Windows 7. Technium Soc. Sci. J.. 2022;30:775.

[8]Waliulu RF. Reverse Engineering Reverse Engineering Analysis Forensic Malware WEBC2-Div. INISTA: Journal of Informatics, Information System, Software Engineering and Applications. 2018 Sep 26;1(1).