Demonstration of Malware Analysis Tools

14-11-2022

Mohd Ayan Khan - 20BCE0644Sujay Kumar- 20BDS0294Gokul Raj- 20BCE2743

Submitted to: Prof Anand M





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

🔁 Kali 2022 x64 Customized by zSecurity 1.0.10 - VMware Workstation 16 Player (Non-commercial use only)	
Player - 📙 - 육 🗇 🛛	and the second se
Applications Places @ cutter Nov 14 10:59	1 💉 📢 🔿
Load Options	•
Program: //met/Downloads/dronchet.ava.vir	
ringum. noorborniousjureprincese.m	
✓ Analysis: Enabled	
None Auto Auto	Exp Advanced
Analyze all consecutive functions	A
Two and Arrument matching analysis	
Analyze rade after transleds	
Analyze function proludes	
Analyze imm tables in switch statements	
Continue analysis after each function	
Do not load bin information (-n)	
✓ Use virtual addressing	
✓ Import demangled symbols	
Advanced options	Carel

53 (c-1: 2022 - c.t. C	In the solution of the Albert Albert Albert Albert Albert Albert Albert								
All 2022 xo4 Customized									
Player 👻 📘 👻 🛄									
Applications Places	cutter		N	ov 14 10:59				1 =	O
			Cutter – /root/Do	wnloads/drog	oshot.exe.vir			0 6	
File Edit View Windows D	Debug Help								
	Ture for every considerations								
N. N	type hag name or address here								
Functions C 🗵				(Dashboard				08
Name		OVER	VIEW						
• () entry0									
→ G fcn.00401050		Info							
→ ④ fcn.00401088		File:	/root/Downloads/dropshot.ex	FD:	3	Architecture:	x86		
→ Ō fcn.004010a7		Format:	pe	Base addr:	0x00400000	Machine:	i386		
▹ ④ fcn.004010df		Bits:	32	Virtual addr:	True	OS:	windows		
Image: Image: Market Mar Market Market Market Market Market Ma Artik Market Ma Artik Market Mar Market Market		Class:	PE32	Canary:	False	Subsystem:	Windows GUI		
▹ ④ fcn.004011d7		Mode:	r-x	Crypto:	False	Stripped:	False		
• () tcn.004012a0		Size:	191 kB	NX bit:	True	Relocs:	False		
 (a) fch.00401310 (b) fch.00401310 		Type:	EXEC (Executable file)	PIC:	Irue	Endianness:	LE		
 (a) fcn.00401300 (b) fcn.00401400 		Language:	c	Static:	haise	Compilea:	Mon Feb 8 00:15:47 199:		
> @ fcn.00401440				Keiro.	NIA	compiler.	N/A		
▶ ⊛ fcn.00401480			Certifica	ites					
▶ 🙆 fcn.004014c0									
 G fcn.00401500 		Hashe	s			Librarie	S		
Image: Image: Market Mar Market Market Market Market Market Ma Market Market Ma Market Market M Market Market						kernel32.dll			
Image: Image		MD5:	0ccc9ec82f1d44c243329014	lb82d3125		user32.dll			
• (ig) fcn.004015f0		SHA1:	279ff728023eeaa1715403ed	823801bf3493	f5ca	gdi32.dll			
 (a) fcn.00401640 (b) fcn.00401660 		CHA356	62aabco7a6741a0270cddac	40cd1d715305	aldososos 20bbaaaceff0b	advapi32.dll			
 A fcn 00401650 C fcn 00401650 		JIIA2 50.	02880001857418527000080	490010715505	cidobobeozoobeaeconbc				
→ G fcn.00401750		CRC32:	238bf60c						
▶ 🙆 fcn.004017a0		ENTROPY:	7.107746						
▶ 🕭 fcn.004017e0									
▹ ④ fcn.00401860		Analys	sis info						
 Image: 6 fcn.004018c0 		Functions:	404						
▶ ④ fcn.00401900		X-Refs:	3313						
▹ ④ fcn.00401950		Calls:	2968						
• (Ja) fcn.004019a0		Strings:	2408						
 Op fcn 00401960 C fcn 00401940 		Symbols:	182						
→ Ø fcn.00401a80		Imports:	182						
→ G fcn.00401ad0		Analysis cov	rerage: 62789 bytes						
▶ 🖗 fcn.00401b40 🗸		Code size:	/3/28 bytes						
4 b		coverage pe	0370						
Quick Filter	Bullout Chines Investo Count Discountly Count (or		former Descention (anter 0)						
	Dasnooard Surings imports Search Disassembly Graph (er	illiyo) Hexi	ump Decompiler (entry0)						

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.

File Edit View Wine	dows Debug Help)		
$\leftrightarrow \rightarrow$	🕨 🔹 Type flag	g name or address h	iere	
Functions	0 X			
Name		fon 00401250	(in+32 +)	ang 9h int32 t ang ch):
► 🕼 entrv0		: var int32 t	var 8h @	ebp-0x8
▶		; var int32_t	var_4h @	ebp-0x4
 G fcn 0040105d 		; arg int32_t	arg_8h @	ebp+0x8
 (a) fcn 00401030 (b) fcn 00401088 		; arg int32_t	arg_ch @	ebp+0xc
 A fcn 00401088 A fcn 00401057 		0x004012a0	push	ebp obp
 (a) fcn.004010d7 (b) fcn.004010df 		0x004012a3	sub	esp. 8
• () icn.004010di		0x004012a6	push	4 ; 4
▶ ()) fcn.0040119b		0x004012a8	push	0x1000
) 🖉 fcn.004011d7		0x004012ad	movsx	eax, word [arg_ch]
▶ 🕭 fcn.004012a0		0x004012b1	add	eax, 1
🕨 🕼 fcn.00401310		0x004012b4	push	eax
Icn.004013b0		0x004012b5	push	0 dward [Mintur] Aller] - 004121-0 - 10
▶ 🔎 fcn.00401400		0X004012D7	call	dword [virtualAlloc] ; 0x415188 ; LP
▶ 🖗 fcn.00401440		0x004012c0	mov	dword [var_5h], eax
G fcn 00401480 G		- 0x004012c7	jmp	0x4012d2
 C fcn 00401400 C fcn 004014c0 		0x004012c9	mov	ecx, dword [var_4h]
• () (cn.004014c0		0x004012cc	add	ecx, 1
▶ () tcn.00401500		0x004012cf	mov	dword [var_4h], ecx
) 🖉 fcn.00401560		• 0x004012d2	movsx	edx, word [arg_ch]
) 🕼 fcn.004015a0		0x004012d6	cmp	dword Lvar_4hj, edx
 (k) fcn.004015f0 		0X00401209	Jge	ex dword [var 4b]
Icn.00401640		0x004012dp	mov	ecx dword [arg 8h]

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.

	0	1	2	3	4	5	6	7	8	9	A	В	С	D	Е	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1E	0123456789ABCDEF0123456789ABCDEF
0x000000000041b8c0	1b	00	0a	00	17	00	06	00	01	00	04	00	05	00	06	00	0e	00	06	00	1c	00	06	00	07	00	0b	00	0e	00	06	00	
0x000000000041b8e0	22	00	00	00	1f	00	0b	00	17	00	1c	00	1e	00	01	00	0e	00	16	00	17	00	13	00	1c	00	06	00	03	00	1c	00	"
0x000000000041b900	07	00	0b	00	11	00	04	00	0f	00	19	00	2d	00	00	00	02	00	17	00	06	00	01	00	1c	00	06	00	07	00	0b	00	
0x000000000041b920	0e	00	06	00	22	00	00	00	22	00	17	00	0b	00	1c	00	06	00	07	00	0b	00	0e	00	06	00	00	00	09	00	06	00	<i>n n</i>
0x000000000041b940	1c	00	07	00	Øb	00	0e	00	06	00	1a	00	Øb	00	26	00	06	00	00	00	02	00	17	00	06	00	01	00	1c	00	06	00	
0x000000000041b960	05	00	Øb	00	17	00	06	00	03	00	1c	00	13	00	17	00	24	00	22	00	02	00	17	00	06	00	01	00	1c	00	06	00	.\$."
0x000000000041b980	16	00	17	00	13	00	03	00	06	00	19	00	19	00	22	00	18	00	1c	00	0e	00	1a	00	06	00	1c	00	16	00	17	00	"
0x000000000041b9a0	13	00	03	00	06	00	19	00	19	00	30	00	19	00	02	00	17	00	0b	00	1c	00	0b	00	03	00	01	00	0e	00	00	00	
0x000000000041b9c0	02	00	17	00	06	00	01	00	1c	00	06	00	16	00	17	00	13	00	03	00	06	00	19	00	19	00	00	00	16	00	17	00	
0x000000000041b9e0	13	00	2d	00	30	00	04	00	12	00	1c	00	1a	00	06	00	1c	00	02	00	13	00	11	00	1c	00	06	00	23	00	1c	00	0
0x000000000041ba00	1b	00	0a	00	17	00	06	00	01	00	04	00	09	00	06	00	1c	00	2b	00	11	00	20	00	Øb	00	17	00	13	00	11	00	+
0x000000000041ba20	10	00	06	00	11	00	1c	00	1f	00	01	00	17	00	Øb	00	01	00	2c	00	0e	00	06	00	22	00	00	00	41	61	43	63	
0x000000000041ba40	64	44	65	46	66	47	68	69	4b	4c	6c	4d	6d	6e	4e	6f	4f	70	50	72	52	73	53	54	74	55	75	56	76	77	57	78	dDeFfGhiKL1MmnNoOpPrRsSTtUuVvwWx
0x000000000041ba60	79	5a	7a	33	32	2e	5c	45	62	67	6a	48	49	20	5f	59	51	42	3a	22	2f	40	0a	Ød	1a	00	14	00	07	00	1b	00	vZz32.\EbgiHI YOB:"/@
0x000000000041ba80	22	00	00	00	18	00	2b	00	2a	00	Øf	00	Øb	00	03	00	17	00	13	00	19	00	13	00	08	00	1c	00	2a	00	22	00	" + * * "
0x000000000041baa0	Øb	00	11	00	04	00	13	00	21	00	19	00	2a	00	1a	00	0a	00	06	00	0e	00	0e	00	2a	00	00	00	19	00	19	00	
0x000000000041bac0	13	00	03	00	0b	00	01	00	1c	00	Øb	00	13	00	11	00	19	00	2a	00	1d	00	17	00	0e	00	00	00	19	00	19	00	*
0x000000000041bae0	13	00	03	00	0b	00	01	00	1c	00	Øb	00	13	00	11	00	19	00	2a	00	0a	00	1c	00	1c	00	15	00	2a	00	1d	00	* * *
0x000000000041bb00	19	00	06	00	17	00	02	00	0a	00	13	00	Øb	00	03	00	06	00	00	00	2a	00	19	00	0a	00	06	00	0e	00	0e	00	*

Code for the strings decryption:

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

```
dec_table = 'AaCcdDeFfGhiKLlMmnNoOpPrRsSTtUuVvwWxyZz32.\EbgjHl
_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:

🔁 Kali 2022 x64 Customize	ed by zSecurity 1.0.10 - VN	ware Workstation 16 P	layer (Non-comm	ercial use only)						- @ ×
Player 👻 📕 👻 🛱										
Applications Places	Firefox ESR				Nov 14 11:	28			1	💉 🕬 O
C Home	× 🧧 Untitled1	× 📿 Hon	ne	× 🧧 Untitled	× +					•••
$\leftarrow \rightarrow \mathbf{C}$	🗘 🗅 loca	lhost:8888/notebooks	/Untitled.ipynb?k	ernel_name=python2					☆	⊚ ≡
ZzSecurity ZWireless	Adapters 🛛 🛛 VIP Member	ship 🔰 VPN By zSecur	ity 🛛 zSecurity Y	ouTube (zSecurity FB	🗾 zSecurity Twitter 🛛 🔝 Zaio	's Linkedin 💄 MSFU 🧧 Kali Docs 🔦 Exp	ploit-DB 🔦 GHDB			
		💭 jupyter	Untitled Last CI	neckpoint: a few seconds ago	(unsaved changes)			Nogout Logout		
		File Edit	View Insert	Cell Kernel Widgets	s Help		Trusted	Python 2 O		
		B + × 2	1 B 🛧 🔸	N Run 🔳 C 🕨 Code	v 🖽					
		In ();	# The pre-de dec_table = # The array off.arr = [0x05,0x00,0x7 0x00,0x07,0x7 # The length = 11 dec_str = for i in ran dec_str print ("Decr Decrypted: D	<pre>fined decryption tab AscCobe finkt.Newn (0x4108cc) which is 06,0x00,0x0e,0x00,0x00,0x0 (passed gellength): == dec_table[off_ar ypted; %s^% (dec_st eleteFileW</pre>	<pre>le (the string) oppresstUwwyz232. passed to the function 06,0x00,0x1c,0x00,0x26,0x00 09,0x00,0x1c,0x00,0x26,0x00 rf [1*2]] rf)</pre>	\EbgjHI _Y08:"/@\X0a\X0d\X1a'				

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.

Functions	o ×		
Name		int main (int argo char **argy char **envp):	
▶ 🕼 entry0		; var int32_t var_54h @ ebp-0x54	
▶ 🕭 fcn.00401000		; var int32_t var_38h @ ebp-0x38	
▶ 🕭 fcn.0040105d	_	; var int32_t var_1ch @ ebp-0x1c	
▶ 🕭 fcn.00401088		; var int32_t var_18n @ ebp-0x18 : var int32 t var 14h @ ebp-0x14	
▶ 🕭 fcn.004010a7		; var int32_t var_10h @ ebp-0x10	
▶ 🕭 fcn.004010df		; var int32_t var_ch @ ebp=0xc	
▶ 🕭 fcn.0040119b		; var int32_t var_8h @ ebp-0x8	
▶ 🕭 fcn.004011d7		0x004041a0 push ebp	
 Generation fcn.004012a0 		0x004041a1 mov ebp, esp	
▶ 🕭 fcn.00401310		0x004041a3 sub esp, 0x54	
▶ 🛞 fcn.004013b0		0x004041a6 call fcn.00403b30	
▶ () fcn.00401400			

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.



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.00403	030	
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))</pre>

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.cmdi('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 ISON format rsrcs = cutter.cmdj('iRj') rsrc 101 = {} # Locate resource 101 and dump it to an array for rsrc in rsrcs:

if rsrc['name'] == 101:

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

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.

Info					
File:	er/build/decrypted_rsrc.bin	FD:	3	Architecture:	x86
Format:	ре	Base addr:	0x400000	Machine:	i386
Bits:	32	Virtual addr:	True	OS:	windows
Class:	PE32	Canary:	False	Subsystem:	Windows GUI
Mode:	r-x	Crypto:	False	Stripped:	True
Size:	131 kB	NX bit:	True	Relocs:	False
Туре:	EXEC (Executable file)	PIC:	True	Endianness:	little
Language:		Static:	False	Compiled:	Mon Nov 14 21:16:40 20
		Relro:			
	Certifica	ates			
Hashe	s		Libraries		
MD5:	597c515a46484be4f9597cb4f3)b2959	kernel32.dll		
SHA1:	b9fc1ac4a7ccee467402f190391	974a181391da3	user32.dll		

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).