Malware Analysis - Sikomode

▼ Tools Used:

▼ Basic Analysis

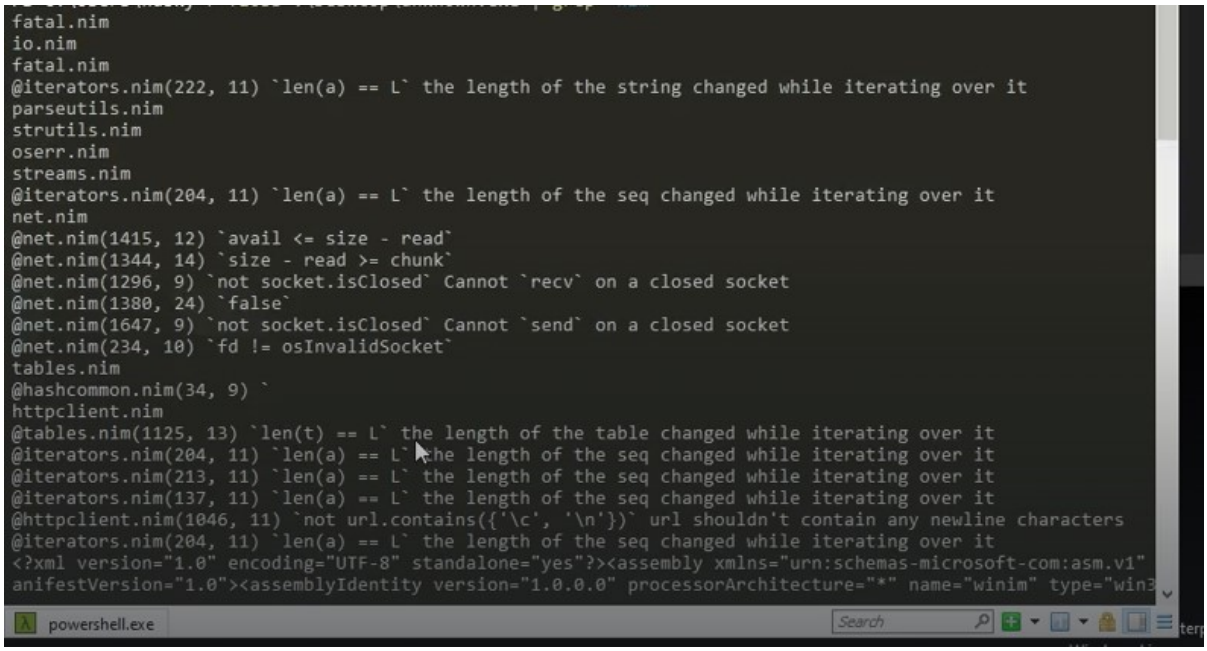
- File hashes
- VirusTotal
- FLOSS
- PEStudio
- PEView
- Wireshark
- Inetsim
- Netcat
- TCPView
- Procmon

▼ Advanced Analysis

- Cutter
- Debugger

▼ Finding the language in which binary is written

Bring the unknown.exe file to the desktop and perform normal static analysis. To find the language use **floss. \Desktop\unknown.exe | grep “nim”** command. We can see that all the method calls start with **nim** which is the language in which the binary is written.

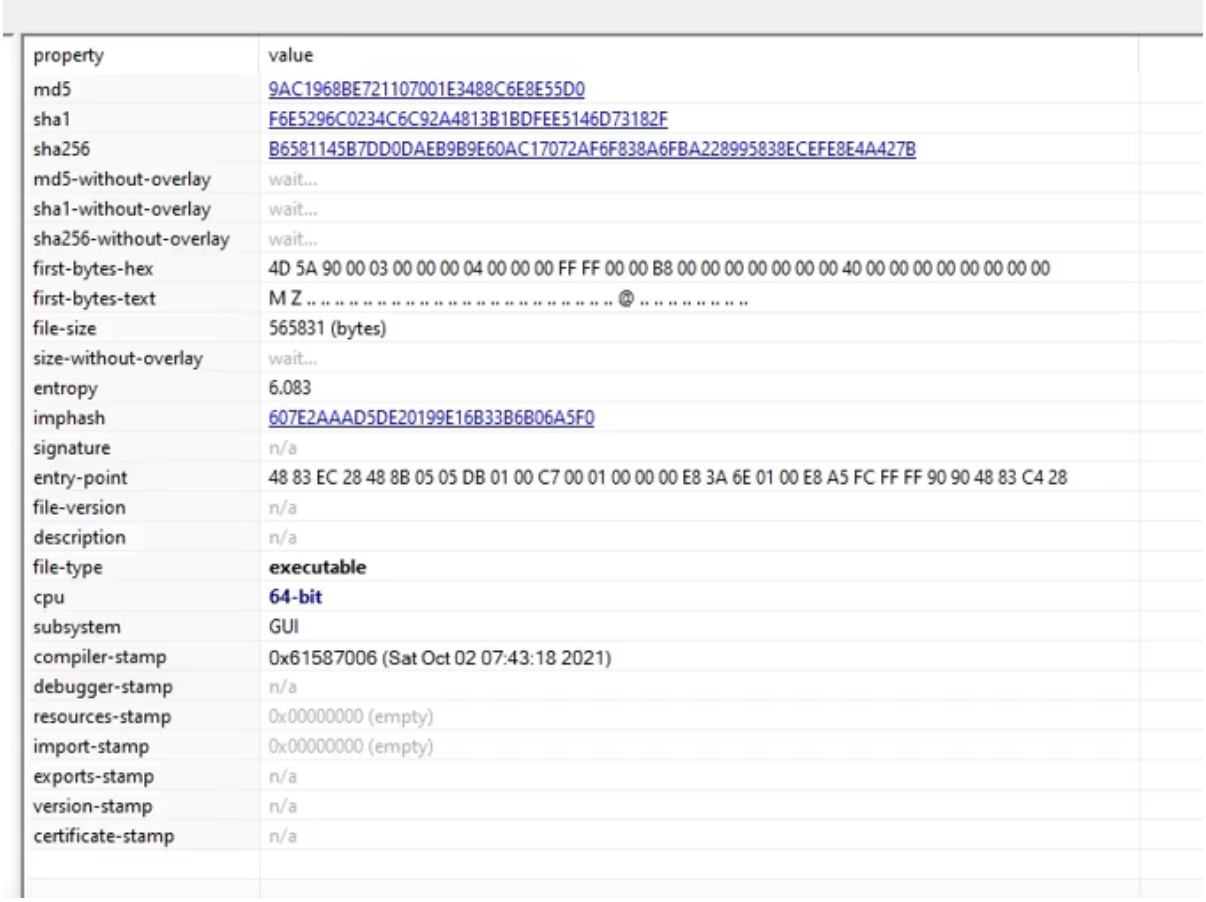


▼ Finding the File Hash

We will check for file hash using the Get-FileHash -Algorithms SHA256 “unknown.exe” command and submit to Virus-total to check for the results. The hash value will be displayed.

▼ Finding the architecture of the binary

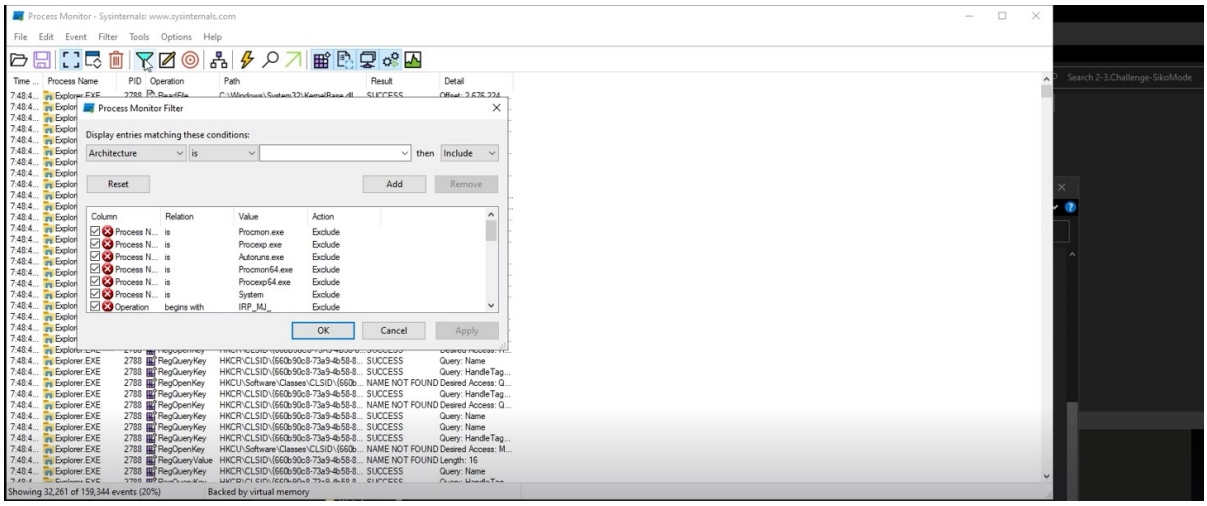
PEview is used here. When we load the unknown.exe file in it, it says 64-bit files are provided here. Finding it is 64-bit,we open it in **pestudio** which gives details that this is an x64-bit architecture binary for a 64-bit CPU.



property	value
md5	9AC1968BE721107001E3488C6E8E55D0
sha1	F6E5296C0234C6C92A4813B1BDFEE5146D73182F
sha256	B6581145B7DD0DAEB9B9E60AC17072AF6F838A6FBA228995838ECEFE8E4A427B
md5-without-overlay	wait...
sha1-without-overlay	wait...
sha256-without-overlay	wait...
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00
first-bytes-text	M Z .. .. @ .. ..
file-size	565831 (bytes)
size-without-overlay	wait...
entropy	6.083
imphash	607E2AAAD5DE20199E16B33B6B06A5F0
signature	n/a
entry-point	48 83 EC 28 48 8B 05 05 DB 01 00 C7 00 01 00 00 00 E8 3A 6E 01 00 E8 A5 FC FF FF 90 90 48 83 C4 28
file-version	n/a
description	n/a
file-type	**executable**
cpu	**64-bit**
subsystem	GUI
compiler-stamp	0x61587006 (Sat Oct 02 07:43:18 2021)
debugger-stamp	n/a
resources-stamp	0x00000000 (empty)
import-stamp	0x00000000 (empty)
exports-stamp	n/a
version-stamp	n/a
certificate-stamp	n/a

▼ Checking the Binary Persistency

Procmon (Process Monitor) tool can be used to find whether the binary is persistent or not. Malware achieves persistence by adding an entry to the run keys in the Windows Registry or the Startup folder. But when we run the exe file in Procmon and observe the filter, we can’t find any host-based indicators like registry keys or process IDs which means that this is a **non-persistent binary.**

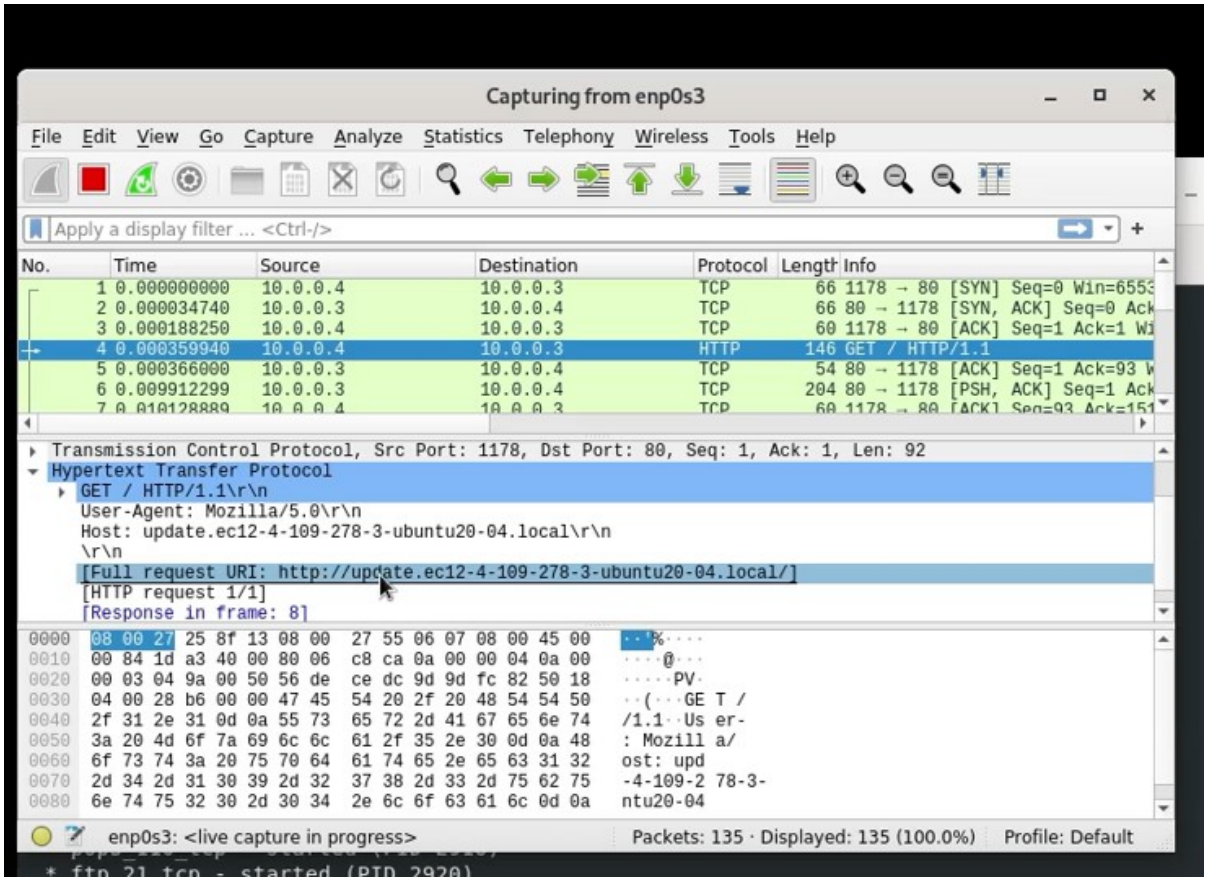


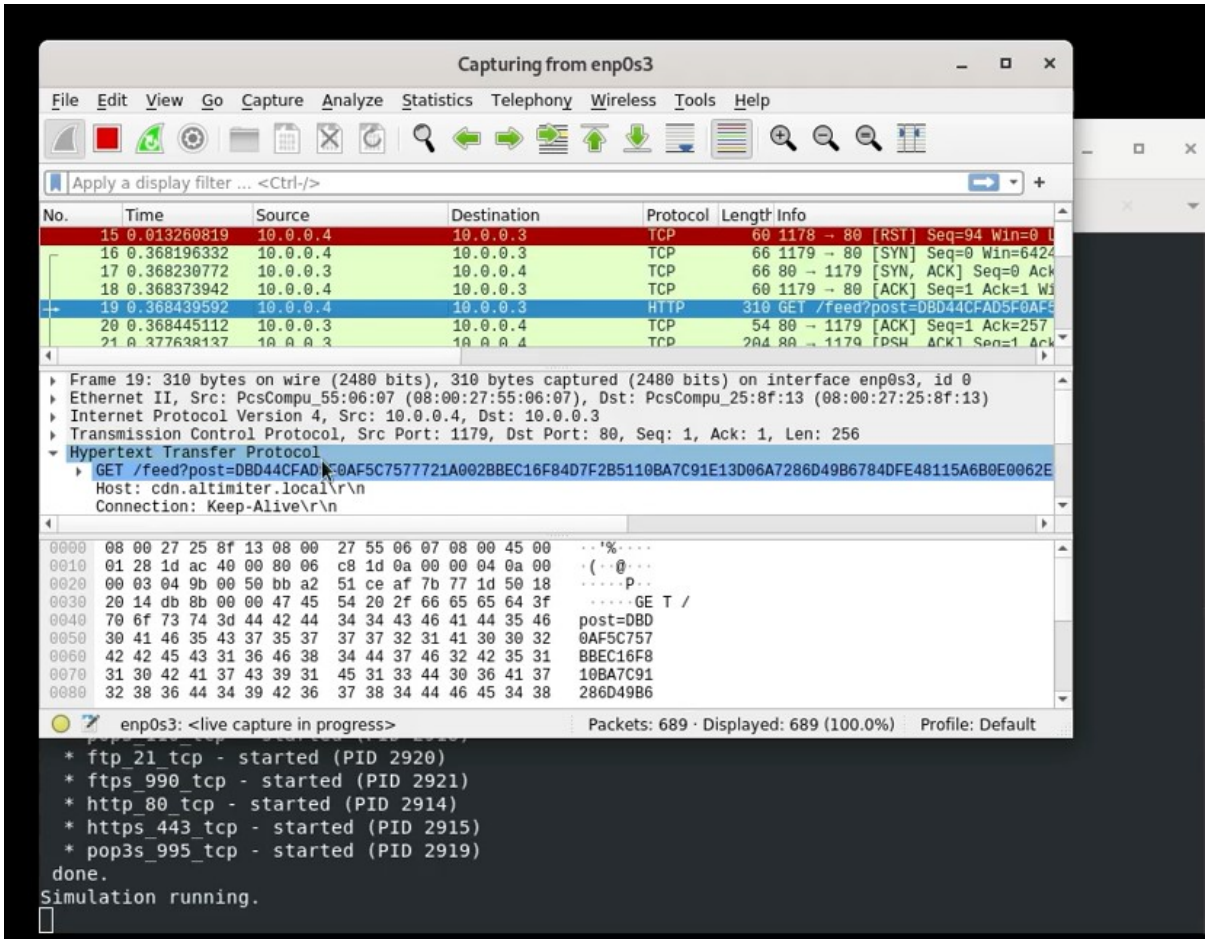
▼ Conditions that can delete the binary by itself

When the inetsim (internet Simulator) i.e., REMnux is running behind, then the exe file does not delete itself. When the inetsim is offed, then the file gets deleted when clicked after performing some malicious activity on the system. This is can be dynamically observed using the Wireshark tool. Also when we kill the inetsim during the executables routine, then it deletes itself.

▼ Callback Domains

To determine the first callback of this binary, we use **Wireshark** to listen to the file. We see the TCP handshake and capture the first HTTP packet that comes. In its header, we can find a large URL. When we grep the URL as a command there is no result shown. When we again scroll down in the filter and find an HTTP request which is an exfiltration domain. So there are two different domains(callback, and exfiltrate).


Capturing from enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
1 0.000000000 10.0.0.4 10.0.0.3 TCP 66 1178 → 80 [SYN] Seq=0 Win=6553
2 0.000034740 10.0.0.3 10.0.0.4 TCP 66 80 → 1178 [SYN, ACK] Seq=0 Ack
3 0.000188250 10.0.0.4 10.0.0.3 TCP 60 1178 → 80 [ACK] Seq=1 Ack=1 Wi
4 0.000359940 10.0.0.4 10.0.0.3 HTTP 146 GET / HTTP/1.1
5 0.000366000 10.0.0.3 10.0.0.4 TCP 54 80 → 1178 [ACK] Seq=1 Ack=93 W
6 0.009912299 10.0.0.3 10.0.0.4 TCP 204 80 → 1178 [PSH, ACK] Seq=1 Ack
Transmission Control Protocol, Src Port: 1178, Dst Port: 80, Seq: 1, Ack: 1, Len: 92
Hypertext Transfer Protocol
GET / HTTP/1.1\r\n
User-Agent: Mozilla/5.0\r\n
Host: update.ec12-4-109-278-3-ubuntu20-04.local\r\n
\r\n
[Full request URI: http://update.ec12-4-109-278-3-ubuntu20-04.local/]
[HTTP request 1/1]
[Response in frame: 8]
0000 08 00 27 25 8f 13 08 00 27 55 06 07 08 00 45 00
0010 00 84 1d a3 40 00 80 06 c8 ca 0a 00 00 04 0a 00
0020 00 03 04 9a 00 50 56 de ce dc 9d 9d fc 82 50 18
0030 04 00 28 b6 00 00 47 45 54 20 2f 20 48 54 54 50
0040 2f 31 2e 31 0d 0a 55 73 65 72 2d 41 67 65 6e 74
0050 3a 20 4d 6f 7a 69 6c 6c 61 2f 35 2e 30 0d 0a 48
0060 6f 73 74 3a 20 75 70 64 61 74 65 2e 65 63 31 32
0070 2d 34 2d 31 30 39 2d 32 37 38 2d 33 2d 75 62 75
0080 6e 74 75 32 30 2d 30 34 2e 6c 6f 63 61 6c 0d 0a
enp0s3: <live capture in progress>
Packets: 135 · Displayed: 135 (100.0%)
Profile: Default



▼ Conditions that the binary exfiltrate data

The binary must contact the initial domain which is the URL that we found out. If it makes a successful connection to it, then it will start to exfiltrate data. The data it is exfiltrating is **cosmo.jpeg.** But if it does not connect to the URL, then it will close itself from the desk.

▼ Exfiltration Domains

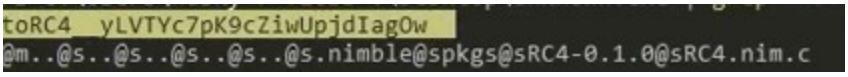
▼ Exfiltration process

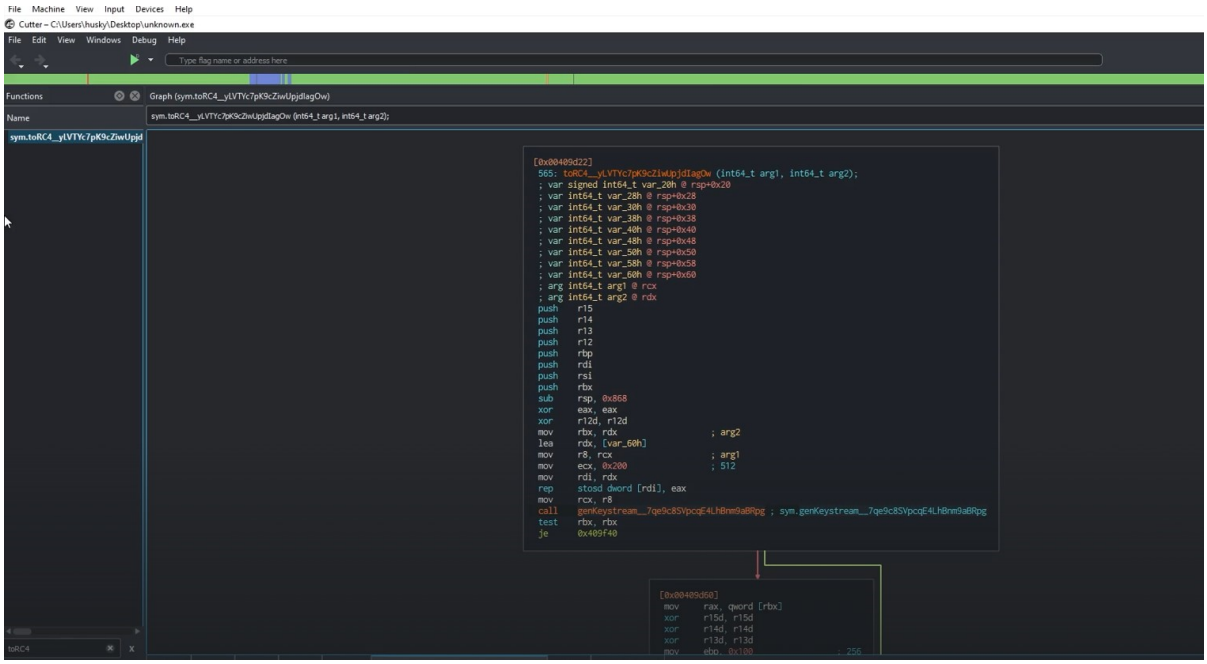
The file is cosmo.jpeg and the kind of data exfiltrated from it can be found using Wireshark. When we click on the **feed get request,** we find a large piece of data below in full URL(post=….). The data from the jpeg file is read by the malware in some way either encoded or encrypted and used with this get request.

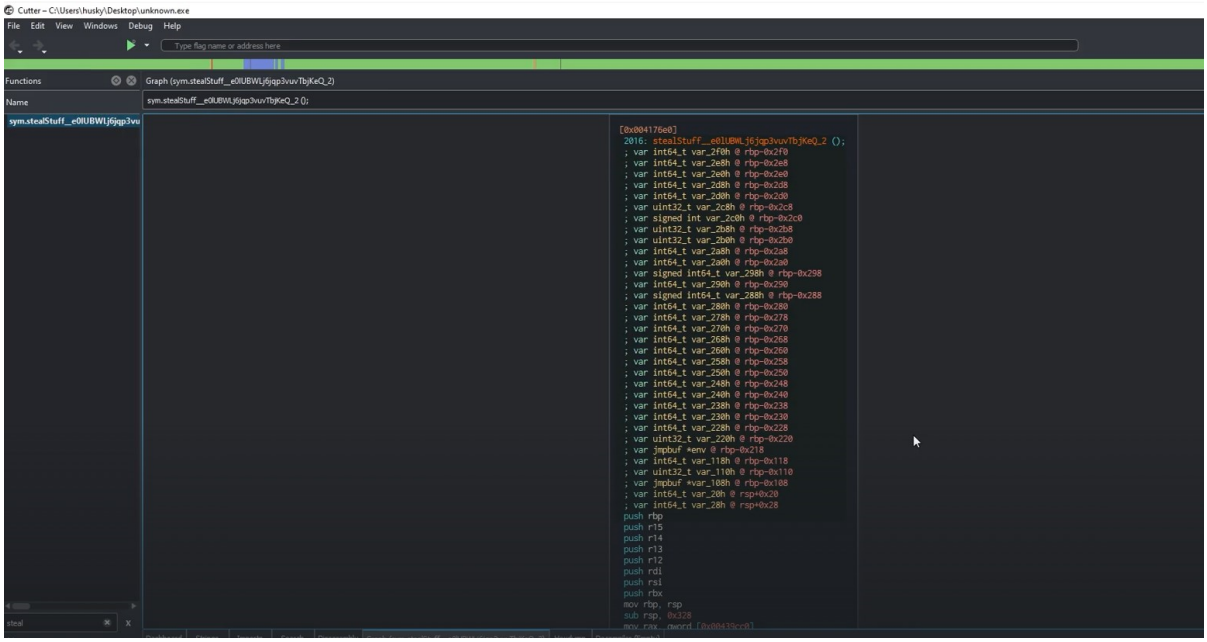
▼ Encryptions Used

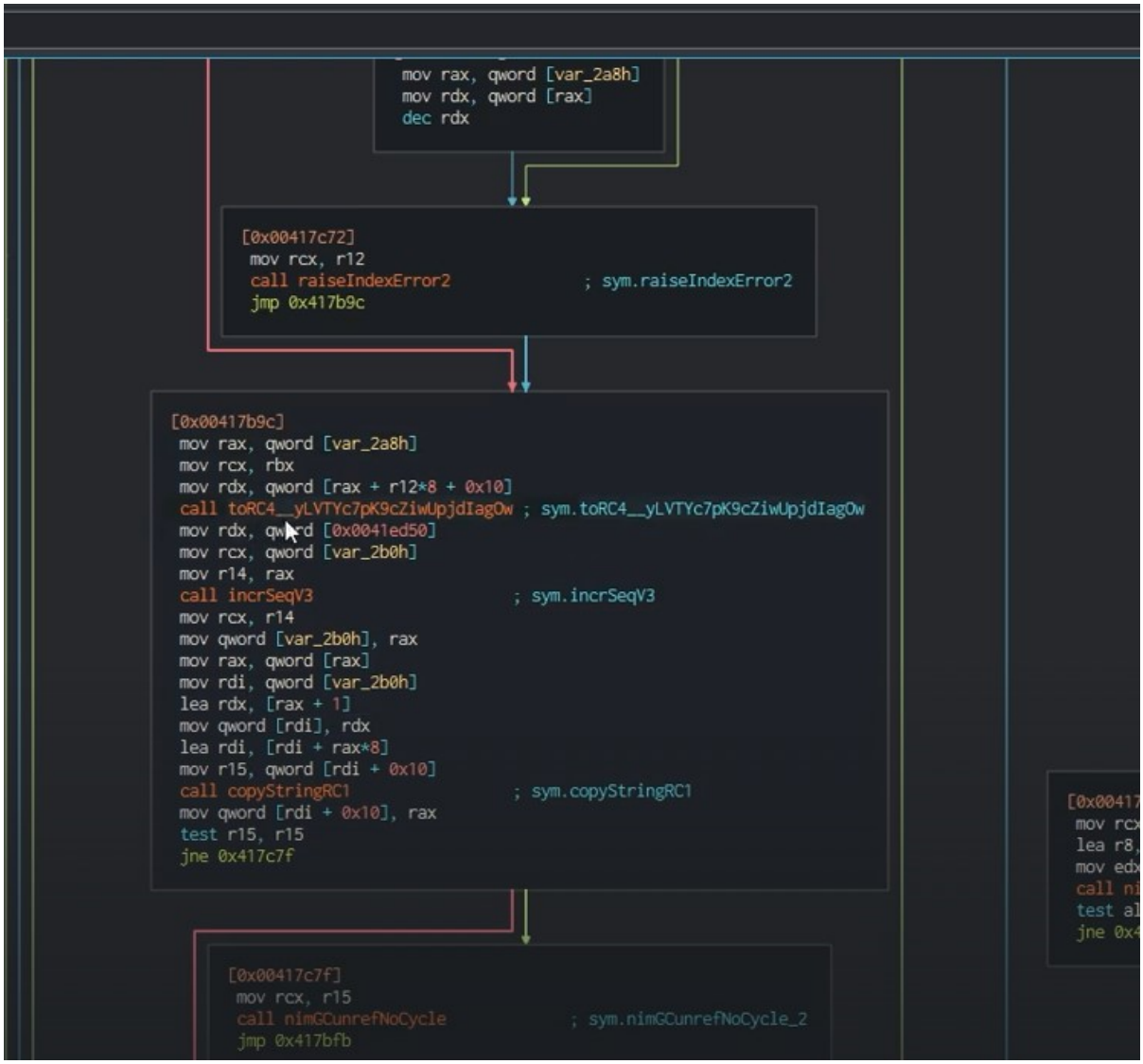
Advanced analysis can be done to find this. One way to find this is to look at the string references in the imported libraries. We go back to the commander and check for **floss .unknown.exe | grep “RC4”**, it might be the RC4 algorithm used by looking at the stream of strings. something related to 2RC4 occurs. To learn more, we use the **Cutter too**l to analyze and load in it. We search for 2rc4 and select graph view for better info in which we can find the 2rc4 method call. After discovering we can find that **stealstuff** is making a call to 2rc4.

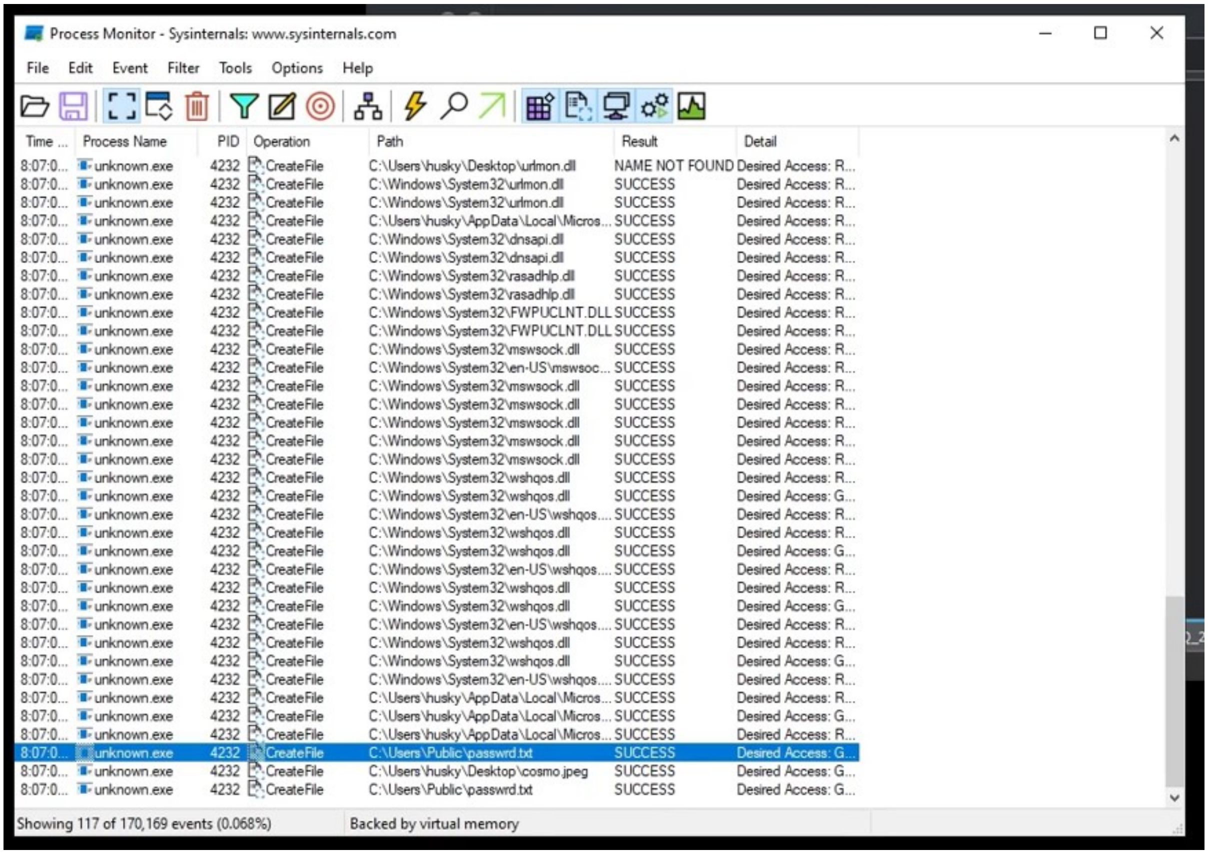
After knowing it is done with rc2 we can discover the key for encryption. Using the Procmon, we can filter with **Process Contains unknown** and **Operation is CreateFile.** We refresh by deleting and running the .exe file again. We can find that a new file named **\Public\passwd.txt** is been created. We open the passwd file and see the text **sikomode** which is the passwd for the encryption.

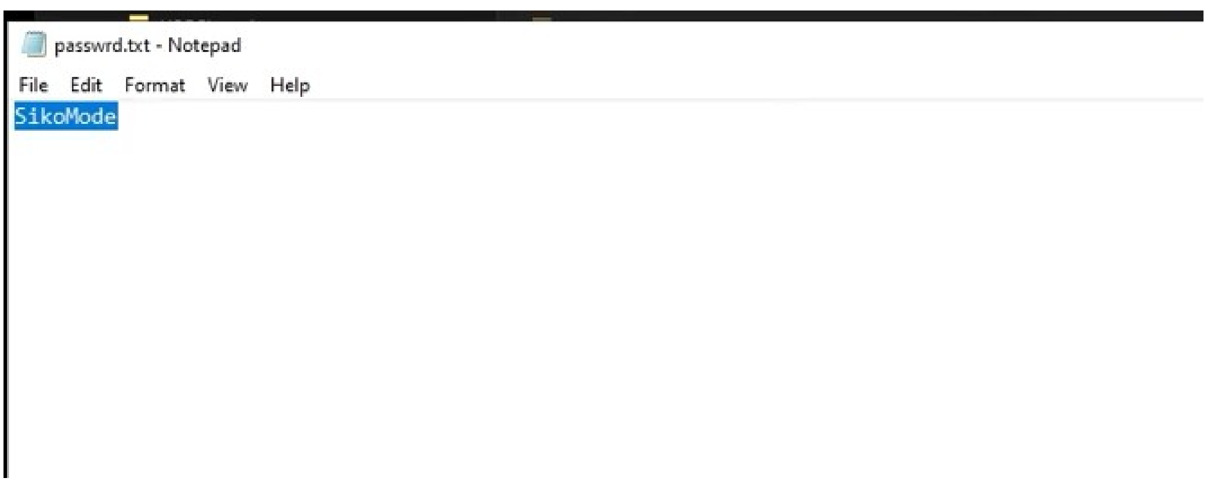





Cutter – C:\Users\husky\Desktop\unknown.exe
File Edit View Windows Debug Help
Type flag name or address here
Functions
Graph (sym.stealStuff__e0lUBWLj6jqp3vuvTbjKeQ_2)
Name
sym.stealStuff__e0lUBWLj6jqp3vuvTbjKeQ_2 ();
sym.stealStuff__e0lUBWLj6jqp3vu
[0x004176e0]
2016: stealStuff__e0lUBWLj6jqp3vuvTbjKeQ_2 ();
; var int64_t var_2f0h @ rbp-0x2f0
; var int64_t var_2e8h @ rbp-0x2e8
; var int64_t var_2e0h @ rbp-0x2e0
; var int64_t var_2d8h @ rbp-0x2d8
; var int64_t var_2d0h @ rbp-0x2d0
; var uint32_t var_2c8h @ rbp-0x2c8
; var signed int var_2c0h @ rbp-0x2c0
; var uint32_t var_2b8h @ rbp-0x2b8
; var uint32_t var_2b0h @ rbp-0x2b0
; var int64_t var_2a8h @ rbp-0x2a8
; var int64_t var_2a0h @ rbp-0x2a0
; var signed int64_t var_298h @ rbp-0x298
; var int64_t var_290h @ rbp-0x290
; var signed int64_t var_288h @ rbp-0x288
; var int64_t var_280h @ rbp-0x280
; var int64_t var_278h @ rbp-0x278
; var int64_t var_270h @ rbp-0x270
; var int64_t var_268h @ rbp-0x268
; var int64_t var_260h @ rbp-0x260
; var int64_t var_258h @ rbp-0x258
; var int64_t var_250h @ rbp-0x250
; var int64_t var_248h @ rbp-0x248
; var int64_t var_240h @ rbp-0x240
; var int64_t var_238h @ rbp-0x238
; var int64_t var_230h @ rbp-0x230
; var int64_t var_228h @ rbp-0x228
; var uint32_t var_220h @ rbp-0x220
; var jmpbuf *env @ rbp-0x218
; var int64_t var_118h @ rbp-0x118
; var uint32_t var_110h @ rbp-0x110
; var jmpbuf *var_108h @ rbp-0x108
; var int64_t var_20h @ rsp+0x20
; var int64_t var_28h @ rsp+0x28
push rbp
push r15
push r14
push r13
push r12
push rdi
push rsi
push rbx
mov rbp, rsp
sub rsp, 0x328
mov rax, qword [0x0043cc01]


mov rax, qword [var_2a8h]
mov rdx, qword [rax]
dec rdx
[0x00417c72]
mov rcx, r12
call raiseIndexError2 ; sym.raiseIndexError2
jmp 0x417b9c
[0x00417b9c]
mov rax, qword [var_2a8h]
mov rcx, rbx
mov rdx, qword [rax + r12*8 + 0x10]
call toRC4__yLVTYc7pK9cZiwUpjdIagOw ; sym.toRC4__yLVTYc7pK9cZiwUpjdIagOw
mov rdx, qword [0x0041ed50]
mov rcx, qword [var_2b0h]
mov r14, rax
call incrSeqV3 ; sym.incrSeqV3
mov rcx, r14
mov qword [var_2b0h], rax
mov rax, qword [rax]
mov rdi, qword [var_2b0h]
lea rdx, [rax + 1]
mov qword [rdi], rdx
lea rdi, [rdi + rax*8]
mov r15, qword [rdi + 0x10]
call copyStringRC1 ; sym.copyStringRC1
mov qword [rdi + 0x10], rax
test r15, r15
jne 0x417c7f
[0x00417c7f]
mov rcx, r15
call nimGCunrefNoCycle ; sym.nimGCunrefNoCycle_2
jmp 0x417bfb

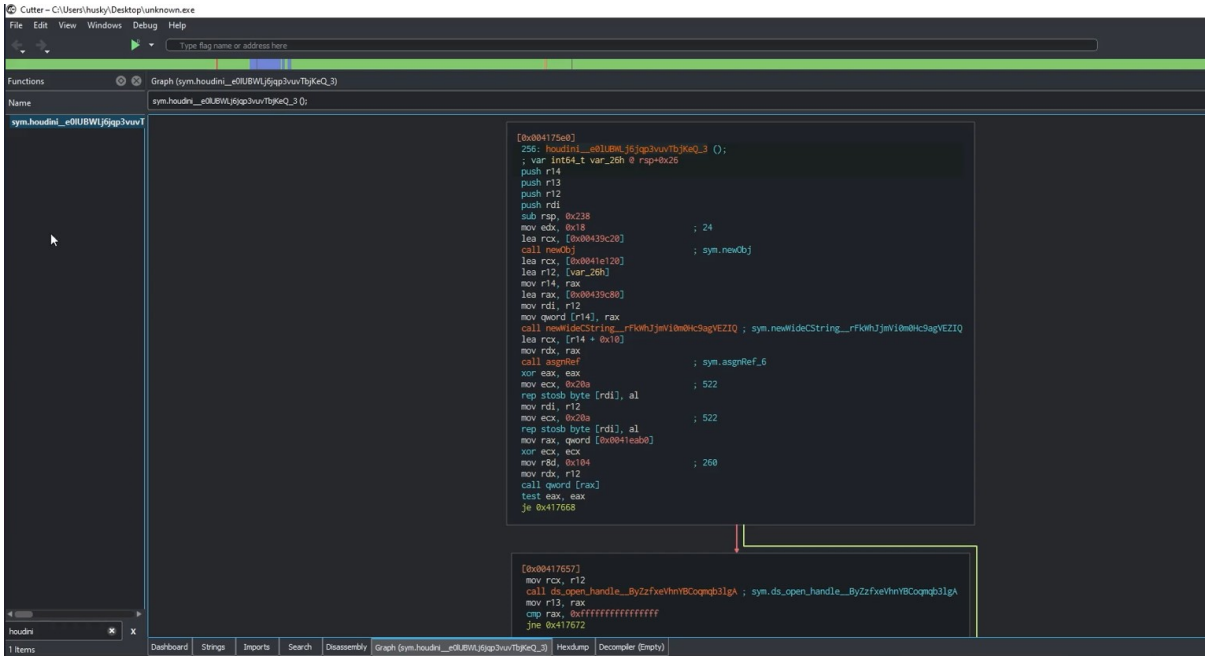


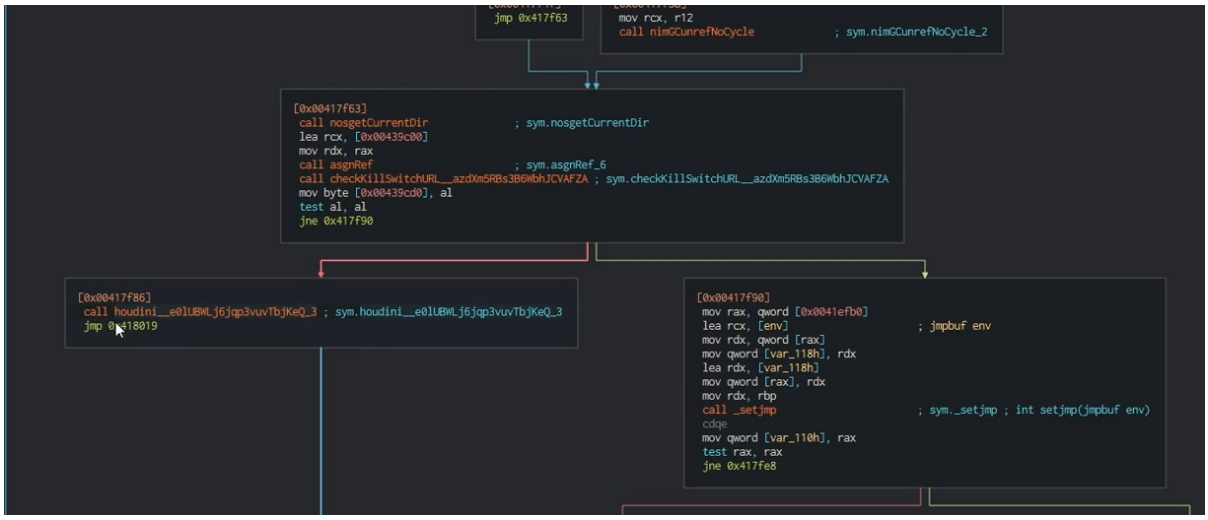


▼ 'Houdini'

Houdini is the method call used by the exe to delete itself from the desk. We can discover the significance of Houdini using the **Cutter tool.** Check in main method(.**NimMainModule**) to see where this method call is used. After the **kill**

function returns false, Houdini is used which directly jumps to the end of the program. Another instance is that, if the **kill function returns true** then **steal operations** are performed. If it is interrupted (as seen before), Houdini is called and also at the end of task completion, Houdini is called finally.






7f90
m.houdini__e0lUBWLj6jqp3vuvTbjKeQ_3
[0x00417f90]
mov rax, qword [0x0041efb0]
lea rcx, [env] ; jmpbuf env
mov rdx, qword [rax]
mov qword [var_118h], rdx
lea rdx, [var_118h]
mov qword [rax], rdx
mov rdx, rbp
call _setjmp ; sym._setjmp ; int setjmp(jmpbuf env)
cdqe
mov qword [var_110h], rax
test rax, rax
jne 0x417fe8
0x00417fc8]
call unpackResources__e0lUBWLj6jqp3vuvTbjKeQ ; sym.unpackResources__e0lUBWLj6jqp3vuvTbjKeQ
call stealStuff__e0lUBWLj6jqp3vuvTbjKeQ_2 ; sym.stealStuff__e0lUBWLj6jqp3vuvTbjKeQ_2
call popSafePoint ; sym.popSafePoint
[0x00417fe8]
call popSafePoint ; sym.popSafePoint
mov qword [var_110h], 0
call houdini__e0lUBWLj6jqp3vuvTbjKeQ_3 ; sym.houdini__e0lUBWLj6jqp3vuvTbjKeQ_3
call popCurrentException ; sym.popCurrentException
jmp 0x417fd7
[0x00417fd7]
call houdini__e0lUBWLj6jqp3vuvTbjKeQ_3 ; sym.houdini__e0lUBWLj6jqp3vuvTbjKeQ_3
cmp qword [var_110h], 0
je 0x418019