

Link: <https://app.hackthebox.com/sherlocks/Safecracker/play>

This is the "insane" level sherlock task with category "**Malware Analysis**" from "Hack The Box" cybersecurity platform.

Description:

We recently hired some contractors to continue the development of our Backup services hosted on a Windows server. We have provided the contractors with accounts for our domain. When our system administrator recently logged on, we found some pretty critical files encrypted and a note left by the attackers. We suspect we have been ransomware. We want to understand how this attack happened via a full in-depth analysis of any malicious files out of our standard triage. A word of warning, our tooling didn't pick up any of the actions carried out - this could be advanced.

First look

According to the description we have the triage of Windows Server system after ransomware activity, which must be thoroughly analyzed and answers to the questions provided.

Let's download 1 GB size file in attachment, unarchive and look what's inside:

Name	Date modified	Type	Size
results	21/06/2023 16:18	File folder	
uploads	21/06/2023 16:17	File folder	
collection_context.json	21/06/2023 16:16	JSON File	4 KB
log.json		JSON File	492 KB
log.json.index		INDEX File	19 KB
requests.json	21/06/2023 16:16	JSON File	223 KB
uploads.json		JSON File	289 KB
uploads.json.index		INDEX File	7 KB

Okay, in "Uploads" directory is a dump of physical memory:

Name	Date modified	Type	Size
auto	21/06/2023 16:18	File folder	
ntfs	21/06/2023 16:17	File folder	
PhysicalMemory.raw	21/06/2023 16:14	RAW File	9,437,184 KB

Actually, there is a lot of artifacts to go through:

oads > auto > C%3A > Windows > System32 >				Search System32
Name	Date modified	Type	Size	
config	21/06/2023 16:18	File folder		
LogFiles	21/06/2023 16:17	File folder		
sru	21/06/2023 16:18	File folder		
Tasks	21/06/2023 16:18	File folder		
wbem	21/06/2023 16:18	File folder		
WDI	21/06/2023 16:18	File folder		
winevt	21/06/2023 16:18	File folder		

In "results" folder the traces of **Kape**, forensics artifact collection utility, are seen:

cracker_challenge > WinServer-Collection > results				Search results
Name	Date modified	Type	Size	
Windows.KapeFiles.Targets%2FAI File Meta...		JSON File	225 KB	
Windows.KapeFiles.Targets%2FAI File Meta...		INDEX File	6 KB	
Windows.KapeFiles.Targets%2FUploads.json		JSON File	386 KB	
Windows.KapeFiles.Targets%2FUploads.jso...		INDEX File	6 KB	
Windows.Memory.Acquisition.json		JSON File	7 KB	
Windows.Memory.Acquisition.json.index		INDEX File	1 KB	

Time to investigate what has happened.

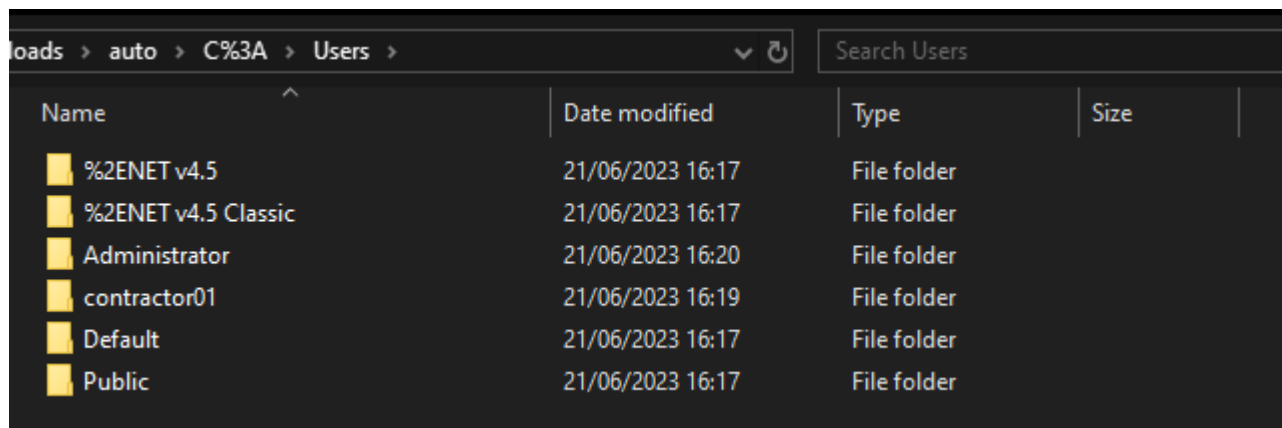
Filesystem investigation

First question is:

Which user account was utilized for initial access to our company server?

So, now I have to look at forensics artifacts, I guess.

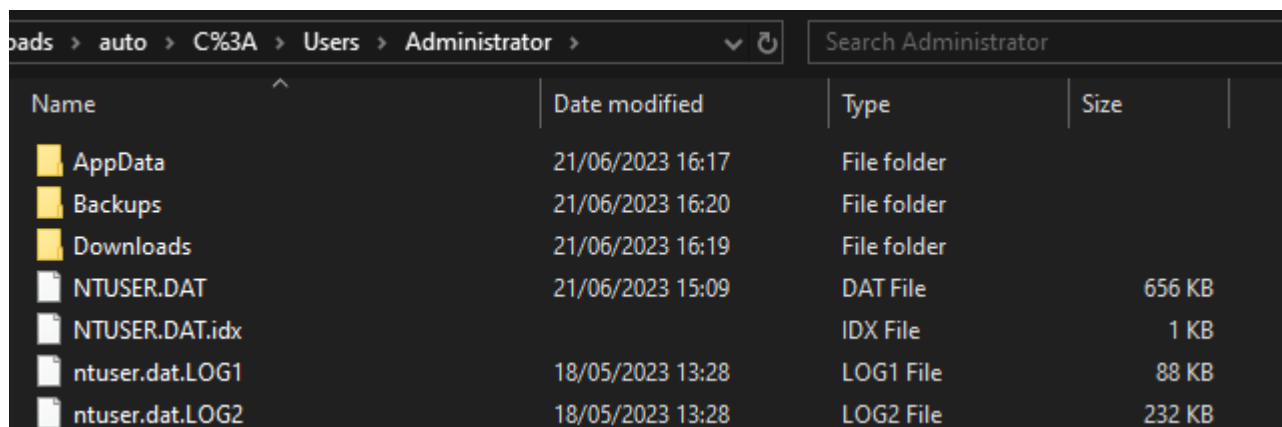
User account and initial access may be found in logs I think, but I will just start by casually walking through users directories.



Name	Date modified	Type	Size
%2ENET v4.5	21/06/2023 16:17	File folder	
%2ENET v4.5 Classic	21/06/2023 16:17	File folder	
Administrator	21/06/2023 16:20	File folder	
contractor01	21/06/2023 16:19	File folder	
Default	21/06/2023 16:17	File folder	
Public	21/06/2023 16:17	File folder	

.NET v4.5 user folders don't have any personal files, so let's leave them for now.

In the folder of user "Administrator" there are 2 interesting folders: "Backups" and "Downloads".



Name	Date modified	Type	Size
AppData	21/06/2023 16:17	File folder	
Backups	21/06/2023 16:20	File folder	
Downloads	21/06/2023 16:19	File folder	
NTUSER.DAT	21/06/2023 15:09	DAT File	656 KB
NTUSER.DAT.idx		IDX File	1 KB
ntuser.dat.LOG1	18/05/2023 13:28	LOG1 File	88 KB
ntuser.dat.LOG2	18/05/2023 13:28	LOG2 File	232 KB

In "Backups" there are suspicious files with .31337 and .note extensions:

C%3A > Users > Administrator > Backups					Search Backups
Name	Date modified	Type	Size		
iisstart.htm	07/06/2023 11:57	Firefox HTML Doc...	1 KB		
passbolt-recovery-kit.txt	07/06/2023 15:57	TXT File	6 KB		
sales-leads.json	07/06/2023 15:58	JSON File	268 KB		
sales-pitch.mp4.31337	21/06/2023 16:06	31337 File	51,776 KB		
sales-pitch.mp4.note	21/06/2023 16:06	NOTE File	1 KB		
splunk-add-on-for-microsoft-windows_...	21/06/2023 13:55	TGZ File	149 KB		
updates.zip.31337	21/06/2023 16:06	31337 File	23,937 KB		
updates.zip.note	21/06/2023 16:06	NOTE File	1 KB		

Contents of the first "note" file look like this:

```

You have been hacked by Cybergang31337

Please can you deposit $200,000 in BTC to the following address:
- 16ftSEQ4ctQFDtVZiUBusQUjRrGhM3JYwe

Once you have done so please email: decryption@cybergang31337.hacker
indicating your source BTC address and we will confirm and release
decryption keys.

Regards
-Cybergang31337

```

Thus, file extension `.31337` must be the encrypted version of original file.

This will help me answer the third and 17-th questions:

Q3: How many files have been encrypted by the the ransomware deployment?

Q17: What file extension does the ransomware rename files to?

Answer: .31337

There is a BTC address so Q18 can be answered too.

Answer to Q18: 16ftSEQ4ctQFDtVZiUBusQUjRrGhM3JYwe

Even more weird the contents of "Downloads" folder:

C%3A > Users > Administrator > Downloads					Search Downloads
Name	Date modified	Type	Size		
desktop.ini	21/06/2023 16:19	Configuration sett...	1 KB		
MsMpEng.exe	12/06/2023 13:43	Application	4,201 KB		
Sysmon.zip.31337	21/06/2023 16:06	31337 File	4,978 KB		
Sysmon.zip.note	21/06/2023 16:06	NOTE File	1 KB		

How Windows Defender executable ended up here?

I need to check it on VirusTotal or something similar.

This is undetected but has **ELF64** format and not **PE64**.

0

Community Score

6b091a4a56a66f858d4c75eb72e16e87169526873147fc78cc4dc261d4cac000

MsMpEng.exe

elf shared-lib 64bits

Size

4.10 MB

Last Analysis Date

5 hours ago

ELF

Reanalyze

Similar

More

DETECTION

DETAILS

RELATIONS

BEHAVIOR

COMMUNITY

Join our Community

and enjoy additional community insights and crowdsourced detections, plus an API key to [automate checks](#).

Security vendors' analysis

Do you want to automate checks?

Acronis (Static ML)	Undetected	AhnLab-V3	Undetected
AliCloud	Undetected	ALYac	Undetected
Antiy-AVL	Undetected	Arcabit	Undetected
Avast	Undetected	Avast-Mobile	Undetected
AVG	Undetected	Avira (no cloud)	Undetected

Moreover, no relation to the Windows Defender or Microsoft in "Details" section at all.

I suppose this is a malware, but I will check it a little bit later.

At the file path `..\WinServer-`

`Collection\uploads\auto\C%3A\Users\Administrator\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadLine\ConsoleHost_history.txt` is a powershell command line history file of "Administrator" user.

Contents:

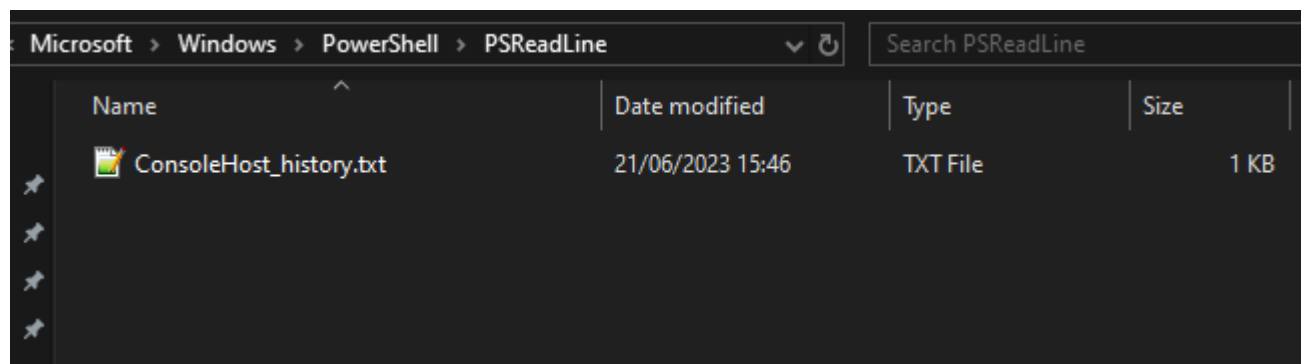
```
wsl --install
Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.SecurityProtocolType]::Tls12; iex ((New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1
'))
```

```
choco install firefox -y
choco install filezilla -y
choco install filezilla.server
netstat -nao
gpupdate /force
wsl --list -v
wsl --set-version Ubuntu-20.04 2
wsl --install
wsl --set-version Ubuntu-20.04 2
wsl --set-version Ubuntu 2
wsl -l -v
wsl --set-version Ubuntu-22.04 2
wsl -l 0v
wsl -l -v
wsl --set-default-version 2
wsl --list-online
wsl --list --online
wsl --install
wsl --set-default-version 2
wsl --set-version Ubuntu-22.04 2
wsl --set-version Ubuntu-20.04 2
wsl --list --online
wsl --set-version Ubuntu-20.04
dism.exe /online /enable-feature /featurename:Microsoft-Windows-
Subsystem-Linux /all /norestart
wsl --set-version Ubuntu-20.04
wsl -l -v
wslconfig.exe /u Ubuntu
wsl -l -v
wsl --install
wslconfig /l
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-
Subsystem-Linux
wsl --install
wsl
wsl --install
wsl --install -d Ubuntu-20.04
wslconfig /l
```

```
wsl -l -v
wsl --install -d Ubuntu-20.04 2
wsl -l -o
wslconfig.exe /u Ubuntu
wsl -l -v
wslconfig.exe /u Ubuntu-20.04
wsl -l -v
wsl --install -d Ubuntu 2
wsl --install -d Ubuntu
wsl -l -v
wslconfig.exe /u Ubuntu
wsl --set-default-version 2
ping 1.1.1.1
ipconfig
wsl --install#
wsl --install
wsl --install -d Ubuntu
wsreset.exe
net stop wuauserv
net start wuauserv
wsl --install -d Ubuntu
wsl --install -d Debian
reboot now
wsl --install
wsl --install -d Ubuntu
wsl --install -d Ubuntu-20.04
wsl
winget uninstall
wsl --list
wsl --install
wsl --install -d Ubuntu-22.04
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform
/all /norestart
wsl --install -d Ubuntu-22.04
wsl
wsl --install -d Ubuntu-22.04
wsl
wsl -l -v
```

WSL has been installed and configured with Ubuntu 22.04 inside. Nothing scary here.

In **contractor01** user directory I also stumbled upon powershell command line history file:



Here it is:

```
ubuntu
whoami
net user
net group
net groups
cd ../../
cd .\Users\contractor01\Contacts\
ls
cd .\PSTools\
ls
.\PsExec64.exe -s -i cmd.exe
```

Command `ubuntu` means, that the WSL has been started with Ubuntu Linux distribution previously configured by "Administrator".

Aside the reconnaissance commands, the `PsExec` with `-s` flag executing `cmd.exe` runs shell under SYSTEM account. This is not normal at all so user **contractor01** must be compromised.

Answer to the Q1 is: **contractor01**

Second question:

Which command did the TA utilize to escalate to SYSTEM after the initial compromise?

Answer: `.\PsExec64.exe -s -i cmd.exe`

Checked everything and there are no more executable files except **MsMpEng.exe**.
Let's count encrypted files by parsing **\$MFT** file with **MFTECmd** tool, then move on to analyzing the executable.

```
PS C:\Users\kaixeb_re\Downloads> .\MFTECmd.exe -f '.\$MFT' --csv mft_res
MFTECmd version 1.3.0.0

Author: Eric Zimmerman (saericzimmerman@gmail.com)
https://github.com/EricZimmerman/MFTECmd

Command line: -f .\$MFT --csv mft_res

Warning: Administrator privileges not found!

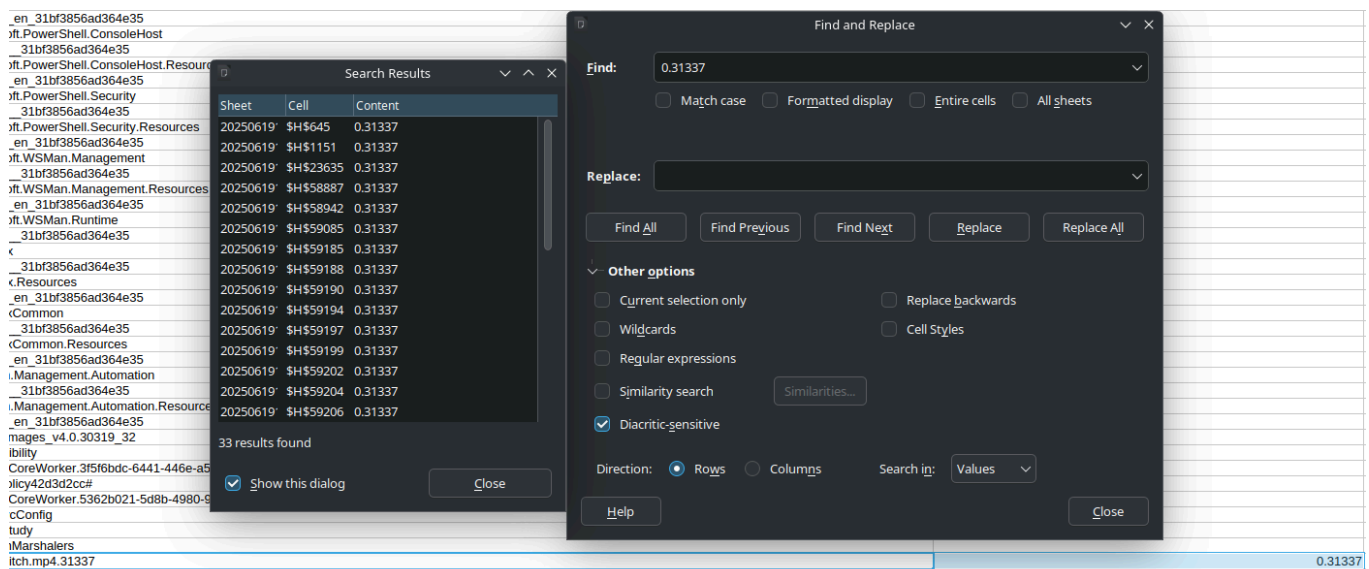
File type: Mft

Processed .\$MFT in 7.1074 seconds

.\$MFT: FILE records found: 219,410 (Free records: 1,614) File size: 216MB
Path to mft_res doesn't exist. Creating...
      CSV output will be saved to mft_res\20250619172419_MFTECmd_$MFT_Output.csv

PS C:\Users\kaixeb_re\Downloads>
```

Show all files with that extension:



The screenshot shows a Windows file explorer window with a search results table. The table has three columns: Sheet, Cell, and Content. It lists 33 results found. A 'Find and Replace' dialog box is open, showing the search results table. The 'Find' field contains '0.31337'. The 'Replace' field is empty. The 'Find and Replace' dialog box has buttons for 'Find All', 'Find Previous', 'Find Next', 'Replace', and 'Replace All'. It also has checkboxes for 'Match case', 'Formatted display', 'Entire cells', and 'All sheets'. The 'Other options' section includes checkboxes for 'Current selection only', 'Replace backwards', 'Wildcards', 'Cell Styles', 'Regular expressions', 'Similarity search', and 'Diacritic-sensitive'. The 'Direction' is set to 'Rows' and 'Search in' is set to 'Values'. The 'Help' button is at the bottom left of the dialog box.

Sheet	Cell	Content
20250619	\$H\$645	0.31337
20250619	\$H\$1151	0.31337
20250619	\$H\$23635	0.31337
20250619	\$H\$58887	0.31337
20250619	\$H\$58942	0.31337
20250619	\$H\$59085	0.31337
20250619	\$H\$59185	0.31337
20250619	\$H\$59188	0.31337
20250619	\$H\$59190	0.31337
20250619	\$H\$59194	0.31337
20250619	\$H\$59197	0.31337
20250619	\$H\$59199	0.31337
20250619	\$H\$59202	0.31337
20250619	\$H\$59204	0.31337
20250619	\$H\$59206	0.31337

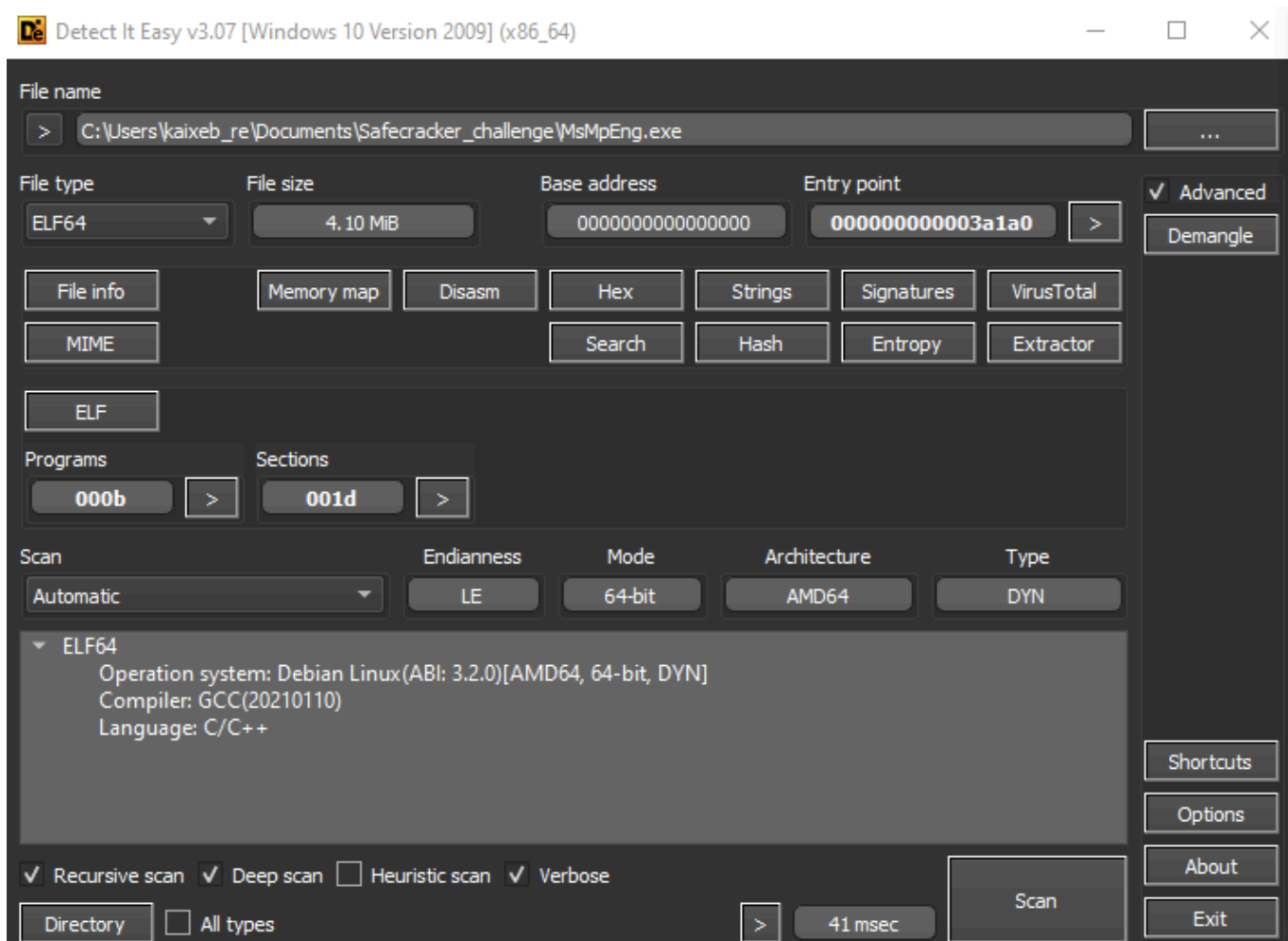
There are 33 of them.

Answer to Q3: **33**

Malware analysis

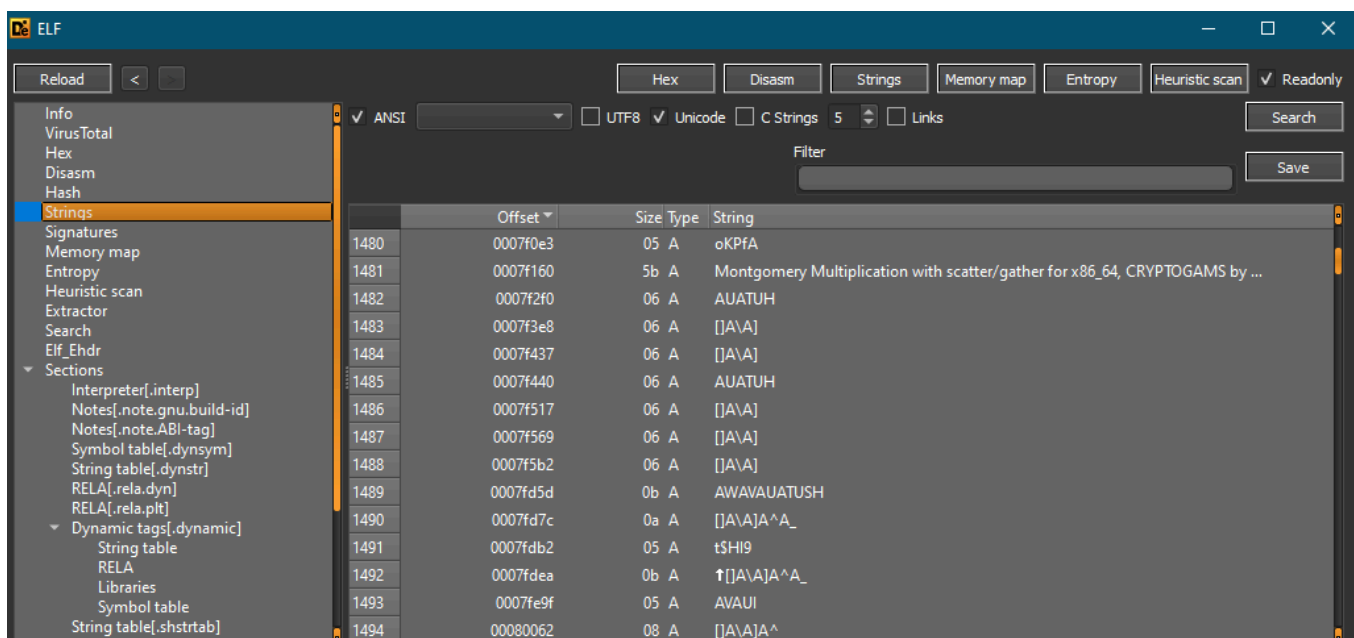
Basic static analysis is the first thing that I should do.

Let's open the binary in **DiE (Detect It Easy)** program:

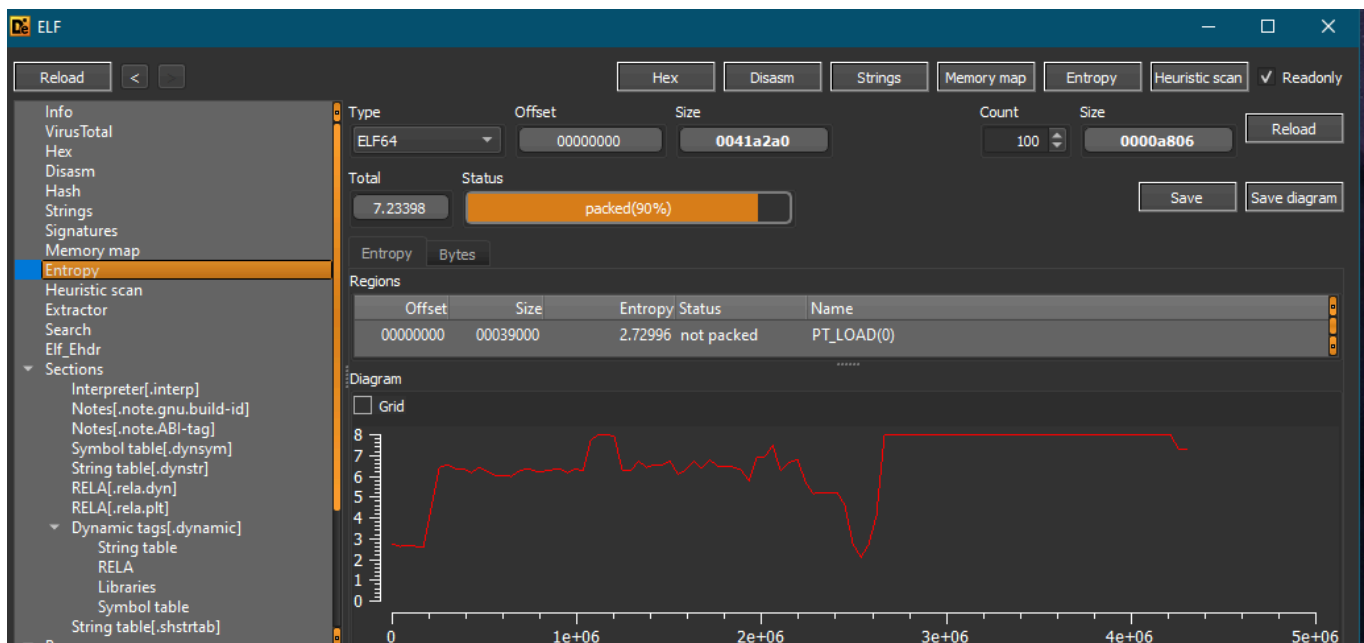


Going further, I need to check if binary is packed or not.

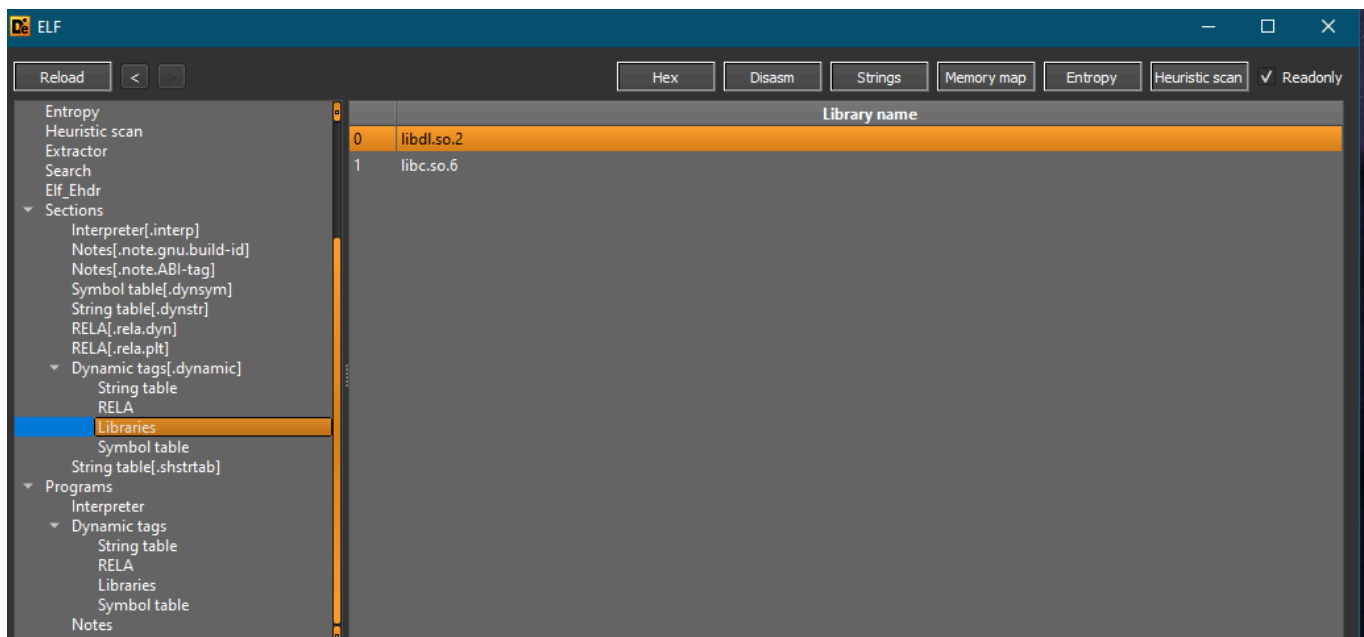
Strings mostly look like gibberish but there are some related to crypto:



Entropy level is very high:

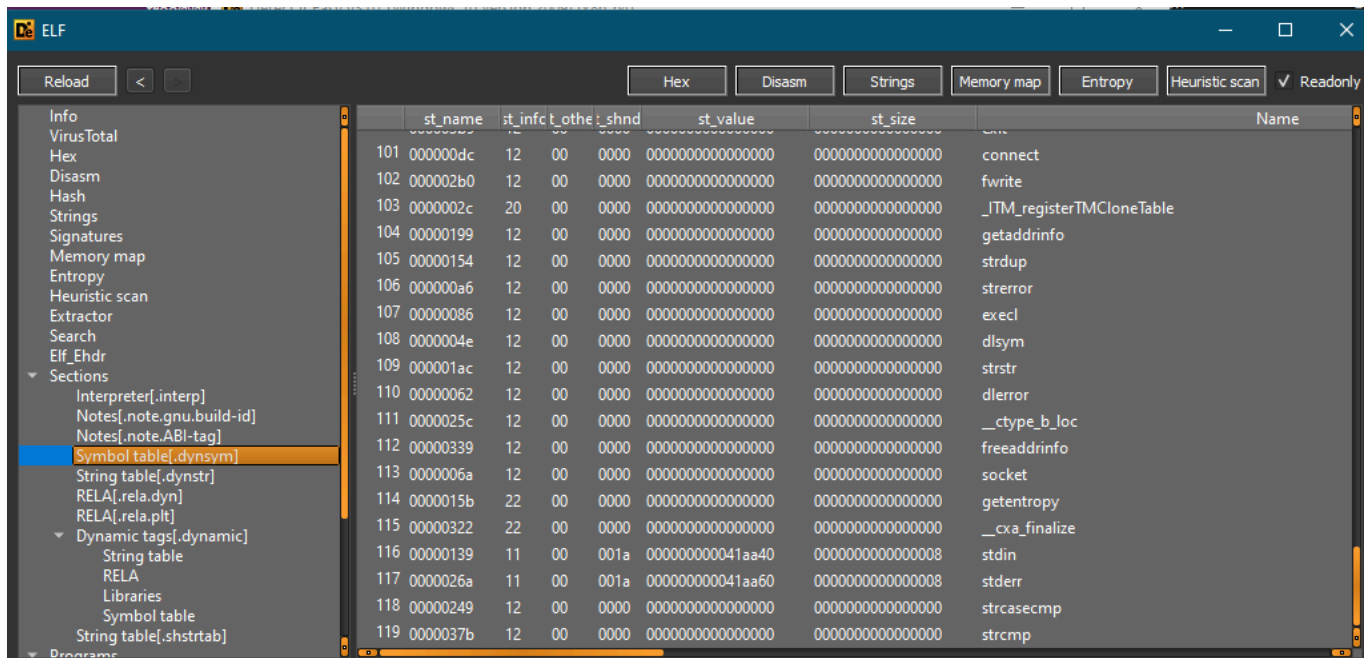


Imported libraries are only the basic ones:

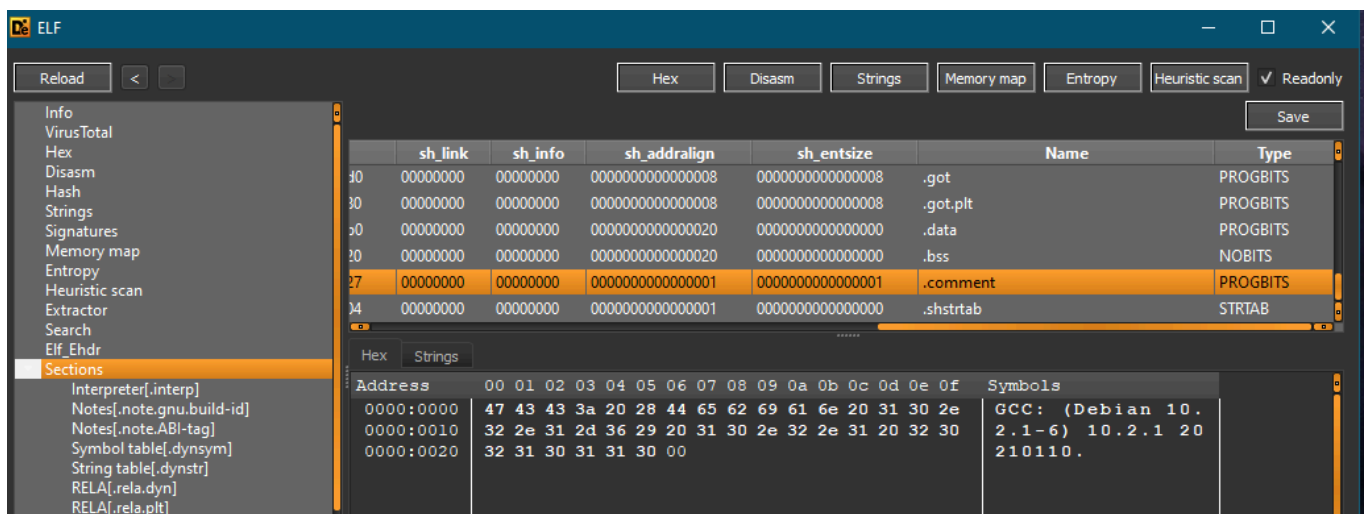


libdl.so.2 - library with functions, that provide dynamic linking facilities. **libc.so.6** - linux C library.

But there are function names (.dynsym section) related to network interaction such as `connect` , `getaddrinfo` , `socket` and etc.



By the way, question #16 requires to look at `.comment` section, so here we go:



Q16: What is the contents of the `.comment` section?

Answer: **GCC: (Debian 10.2.1-6) 10.2.1 20210110**

GCC is the compiler, so answer to Q14 is **gcc**.

Talking about packer, I am leaning towards the option, that It was actually used, but still need to check the binary in IDA.

Reverse engineering

Starting with the `main()` function, I stumble upon the `memfd_create()` call, from where the name of `memoryfd` is asked in Q8.

```

1 __int64 __fastcall main(int a1, char **a2, char **a3)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5     size = (size_t)&unk_36D920;
6     v41 = 1637173LL;
7     ptr = malloc((unsigned int)::size);
8     buf = malloc((size_t)&unk_36D920);
9     sub_3A29B(byte_2893A0, (char *)ptr, ::size);
10    errnum = sub_3A3CB((__int64)ptr, (__int64)buf, size, size);
11    if ( errnum < 0 )
12        sub_3A4AC(errnum);
13    free(ptr);
14    fd = memfd_create("test", 1LL);
15    if ( fd <= 0 )
16    {
17        printf("ERROR FD:%i\n", fd);
18        exit(-1);
19    }
20    errnum = write(fd, buf, errnum);
21    if ( errnum <= 0 )
22    {
23        v3 = strerror(errnum);
24        fprintf(stderr, "Error Writing: %s\n", v3);
25        exit(-1);
26    }
27    free(buf);

```

Answer to Q8: **test**

By the way, function `memfd_create` creates an anonymous file and returns a file descriptor which can be used to create memory mappings using the `mmap` function. The file behaves like a regular file, and so can be modified, truncated, memory-mapped, and so on. However, unlike a regular file, it lives in RAM and has a volatile backing storage.

Definition: `int memfd_create (const char *name, unsigned int flags)`

Then, I have found one function, that uses `crypto/evp/evp_enc.c` string as an argument, and by searching on the internet, it is related to OpenSSL

(https://github.com/openssl/openssl/blob/master/crypto/evp/evp_enc.c):

```

1  __int64 __fastcall sub_402D0(char *a1, char *a2, int *a3, char *a4, int a5)
2  {
3      // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5      v5 = *((_DWORD *)a1 + 4);
6      if ( v5 )
7      {
8          v5 = 0;
9          mw_prob_transform_key(6, 166, 148, "crypto/evp/evp_enc.c", 474);
10         return v5;
11     }
12     v6 = a2;
13     v37 = a5;
14     v36 = *((_DWORD *)((_QWORD *)a1 + 4LL));
15     v11 = sub_41D50((__int64)a1, 0x2000u);
16     v12 = v37;
17     v13 = v11;
18     v14 = (v37 + 7) >> 3;
19     if ( !v13 )
20         v14 = v37;
21     v15 = v14;
22     if ( a5 < 0 )
23         goto LABEL_10;
24     v16 = *((_QWORD *)a1);
25     if ( a5 )
26         goto LABEL_6;
27     v40 = v37;
28     v39 = v14;
29     if ( (sub_41880((__QWORD *)a1) & 0xF0007) != 7 )
30     {

```

000403AE sub_402D0:9 (403AE) (Synchronized with IDA View-A)

More to that, there is an error message in some function about OpenSSL:

```

1  void __fastcall __noreturn sub_80670(const char *a1, const char *a2, int a3)
2  {
3      sub_805B0("%s:%d: OpenSSL internal error: %s\n", a2, a3, a1);
4      abort();
5  }

```

So, I guess the cryptography algorithms used in malware are from the OpenSSL library.

The first function in `main()` after `malloc()` 's is full of OpenSSL functions. I named it respectively, but not precisely, because it requires too much time to go through and understand what's happening. One thing for sure is the data decryption phase. There are 32 and 16 bytes strings transformed and then sent into the abyss of mathematical operations.

32 bytes fits the key size and 16 bytes is IV. It's gotta be AES-256... but what mode?

```

1:0000000001D8004 db 0
1:0000000001D8005 db 0
1:0000000001D8006 db 0
1:0000000001D8007 db 0
1:0000000001D8008 aa5f41376d435dc db 'a5f41376d435dc6c61ef9ddf2c4a9543c7d68ec746e690fe391bf1604362742f:95e61ead02c32dab646478048203fd0b',0
1:0000000001D8008 ; DATA XREF: .data
1:0000000001D8049 align 10h
1:0000000001D8050 a95e61ead02c32d db '95e61ead02c32dab646478048203fd0b',0
1:0000000001D8050 ; DATA XREF: .data
1:0000000001D8050 a1213 db '1.2.13',0
1:0000000001D8071 ; const char s[] ; DATA XREF: mw_
1:0000000001D8078 s db 'ZBUF',0
1:0000000001D807D ; const char aZmem[] ; DATA XREF: mw_
1:0000000001D807D aZmem db 'ZMEM',0
1:0000000001D807D ; const char aZdata[] ; DATA XREF: mw_
1:0000000001D8082 aZdata db 'ZDATA',0
1:0000000001D8082 ; const char aUnknownErr[] ; DATA XREF: mw_
1:0000000001D8088 aUnknownErr db 'Unknown ERR',0
1:0000000001D8094 aTest db 'test',0
1:0000000001D8094 ; DATA XREF: main
1:0000000001D8099 ; const char format[]

1 __int64 __fastcall mw_init_crypto_ctx_openssl_decrypt(char *arg_196, char *mem_ptr, int size)
2 {
3     int *ab_byte_ptr; // rax
4     unsigned int v6; // [rsp+20h] [rbp-20h] BYREF
5     unsigned int v7; // [rsp+24h] [rbp-1Ch]
6     __int64 *v8; // [rsp+28h] [rbp-18h]
7     void *var_prob_iv; // [rsp+30h] [rbp-10h]
8     void *var_prob_key; // [rsp+38h] [rbp-8h]
9
10    var_prob_key = malloc(32uLL);
11    var_prob_iv = malloc(16uLL);
12    mw_transformString(off_418EE8[0], (__int64)var_prob_key);
13    mw_transformString(off_418EF0, (__int64)var_prob_iv);
14    v8 = (__int64 *)mw_w_alloc_mem();
15    if ( v8
16        && (ab_byte_ptr = (int *)get_ab_byte_ptr(),
17            (unsigned int)mw_openssl_13((__int64)v8, ab_byte_ptr, 0LL, (__int64)var_prob_key, var
18        {
19        if ( (unsigned int)mw_openssl_math_op2((char *)v8, mem_ptr, (int *)&v6, arg_196, size) ==
20            && (v7 = v6, (unsigned int)mw_openssl_math_op3(v8, (__int64 *)&mem_ptr[v6], (int *)&v6)

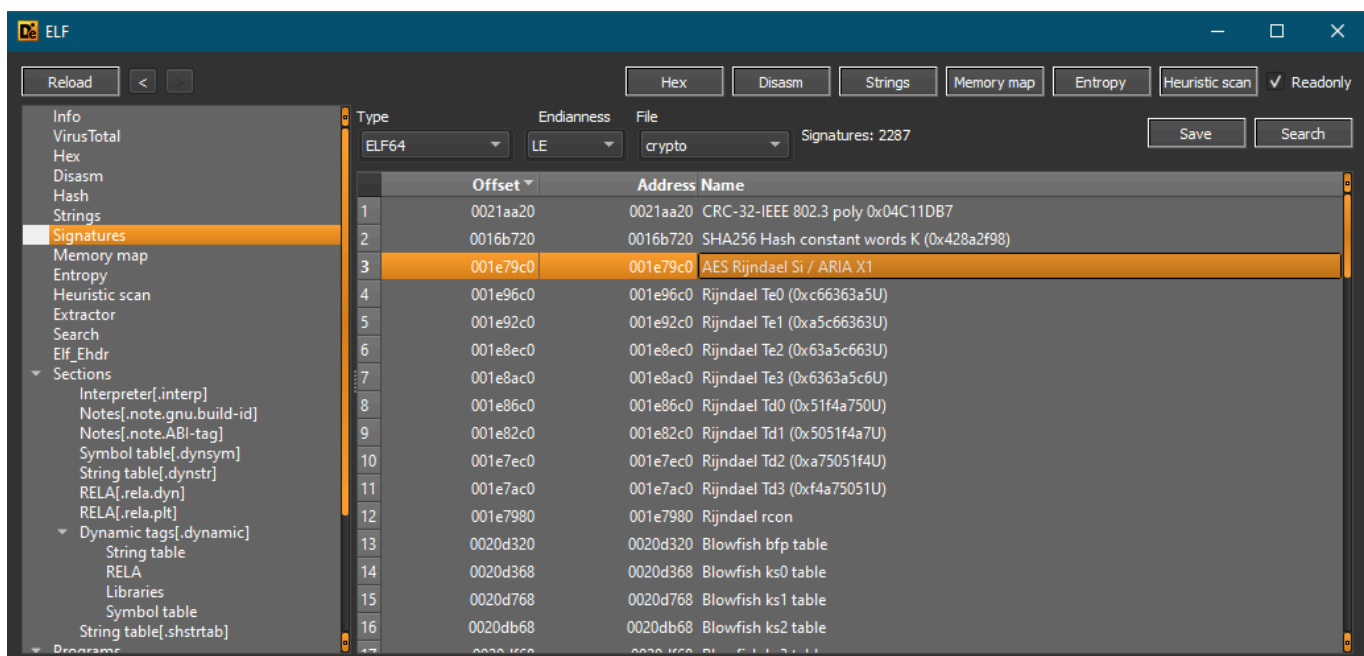
```

Q7: What was the encryption key and IV for the packer?

Answer:

a5f41376d435dc6c61ef9ddf2c4a9543c7d68ec746e690fe391bf1604362742f:95e61ead02c32dab646478048203fd0b

Moreover, the **DiE** tool with signature search points at it:



Before I get to the encryption modes, let's finish superficially inspecting the `main()` function.

After OpenSSL initialization and decryption, the decompression follows, and it looks like **zlib** is used:

```

1 __int64 __fastcall main(int a1, char **a2, char **a3)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5     size = (size_t)&unk_36D920;                                // 197 bytes
6     v41 = 1637173LL;
7     ptr = malloc((unsigned int)::size);
8     buf = malloc((size_t)&unk_36D920);
9     mw_init_crypto_ctx_openssl_decrypt(byte_2893A0, (char *)ptr, ::size);
10    errnum = mw_w_decompress_execute((__int64)ptr, (__int64)buf, size, size);
11    if ( errnum < 0 )
12        mw_zlib_errors(errnum);

```

```

1 __int64 __fastcall mw_w_decompress_execute(__int64 a1, __int64 a2, __int64 a3, __int64 a4)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5     v11 = 0LL;
6     v12 = 0LL;
7     v13 = 0LL;
8     v7 = a3;
9     v6 = a3;
10    v5 = a1;
11    v10 = a4;
12    v9 = a4;
13    v8 = a2;
14    v14 = mw_math_op6(&v5, 47, "1.2.13", 112);
15    if ( !v14 )
16    {
17        v14 = mw_compression_stuff((__int64)&v5, 4);
18        if ( v14 == 1 )
19            return v10;
20    }
21    mw_execute_dynamic_funcs(&v5);
22    return v14;
23 }

```



```

1  __int64 __fastcall mw_compression_stuff(__int64 a1, int a2)
2  {
3      // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5      if ( !a1 || !*(_QWORD *)(a1 + 64) || !*(_QWORD *)(a1 + 72) )
6          return (unsigned int)-2;
7      v2 = *(_QWORD *)(a1 + 56);
8      v3 = -2;
9      if ( v2 )
10     {
11         if ( a1 == *(_QWORD *)v2 )
12         {
13             v4 = *(_DWORD *)(v2 + 8);
14             v5 = v4 - 16180;
15             if ( (unsigned int)(v4 - 16180) <= 0x1F )
16             {
17                 dest = *(_BYTE **)(a1 + 24);
18                 if ( dest )
19                 {
20                     v7 = *(unsigned __int8 **)a1;
21                     v253 = *(_DWORD *)(a1 + 8);
22                     if ( *(_QWORD *)a1 || !*(_DWORD *)(a1 + 8) )
23                     {

```

```

109         v7 = v116;
110         v3 = 0;
111         goto LABEL_78;
112     }
113     v11 -= 2;
114     v116 = v7 + 2;
115     v10 += 16;
116     v12 += (unsigned __int64)v7[1] << v117;
117 }
118 LABEL_226:
119     *(_DWORD *)(v2 + 24) = v12;
120     if ( (_BYTE)v12 != 8 )
121     {
122 LABEL_227:
123         v23 = v12;
124         v22 = a1;
125         v7 = v116;
126         *(_QWORD *)(a1 + 48) = "unknown compression method";
127         *(_DWORD *)(v2 + 8) = 16209;
128         goto LABEL_60;
129     }
130     if ( (v12 & 0xE000) != 0 )
131     {

```

```

1 int __fastcall mw_zlib_errors(int a1)
2 {
3     if ( a1 == -3 )
4         return puts("ZDATA");
5     if ( a1 > -3 )
6         return puts("Unknown ERR");
7     if ( a1 == -5 )
8         return puts("ZBUF");
9     if ( a1 == -4 )
10        return puts("ZMEM");
11    else
12        return puts("Unknown ERR");
13 }

```

Q10: What compression library was used to compress the packed binary?

Answer: **zlib**

Past function with compression stuff there is a function that executes some functions from the array, which comes from compression function:

```

1 __int64 __fastcall mw_w_decompress_execute(__int64 a1, __int64 a2, __int64 a3, __int64 a4)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5     v11 = 0LL;
6     v12 = 0LL;
7     v13 = 0LL;
8     v7 = a3;
9     v6 = a3;
10    v5 = a1;
11    v10 = a4;
12    v9 = a4;
13    v8 = a2;
14    v14 = mw_math_op6(&v5, 47, "1.2.13", 112);
15    if ( !v14 )
16    {
17        v14 = mw_compression_stuff((__int64)&v5, 4);
18        if ( v14 == 1 )
19            return v10;
20    }
21    mw_execute_dynamic_funcs(&v5);
22    return v14;
23 }

```

```

1 __int64 __fastcall mw_execute_dynamic_funcs(_QWORD *a1)
2 {
3     void (__fastcall *v2)(__int64, __int64); // rax
4     __int64 v3; // rsi
5     __int64 v5; // rdi
6
7     if ( !a1 )
8         return 4294967294LL;
9     if ( !a1[8] )
10        return 0xFFFFFFFFELL;
11    v2 = (void (__fastcall *))(__int64, __int64))a1[9];
12    if ( !v2 )
13        return 0xFFFFFFFFELL;
14    v3 = a1[7];
15    if ( !v3 || a1 != *(_QWORD **)v3 || (unsigned int)(*(__DWORD *) (v3 + 8) - 16180) > 0x1F )
16        return 0xFFFFFFFFELL;
17    v5 = a1[10];
18    if ( *(_QWORD *) (v3 + 72) )
19    {
20        v2(v5, *(_QWORD *) (v3 + 72));
21        v2 = (void (__fastcall *))(__int64, __int64))a1[9];
22        v3 = a1[7];
23        v5 = a1[10];
24    }
25    v2(v5, v3);
26    a1[7] = 0LL;
27    return 0LL;
28 }

```

Further, plain binary contents are written to the anonymous file 'test', then buffer with decompressed contents is freed:

```

14 fd = memfd_create("test", 1LL);
15 if ( fd <= 0 )
16 {
17     printf("ERROR FD:%i\n", fd);
18     exit(-1);
19 }
20 errnum = write(fd, buf, errnum);
21 if ( errnum <= 0 )
22 {
23     v3 = strerror(errnum);
24     fprintf(stderr, "Error Writing: %s\n", v3);
25     exit(-1);
26 }
27 free(buf);

```

At last, the malware starts the extracted binary from the memory (by referring to the /proc filesystem) with process name as PROGRAM :

```

60 sprintf(s, "/proc/self/fd/%i", fd);
61 execl(s, "PROGRAM", 0LL);
62 return 0LL;
63 }

```

Q4: What is the name of the process that the unpacked executable runs as?

Answer: **PROGRAM**

Now, let's get back to the AES and brute force the encryption modes.

First, extract the binary blob at address 0x2893a0 (0x2883a0 physical offset in the file):

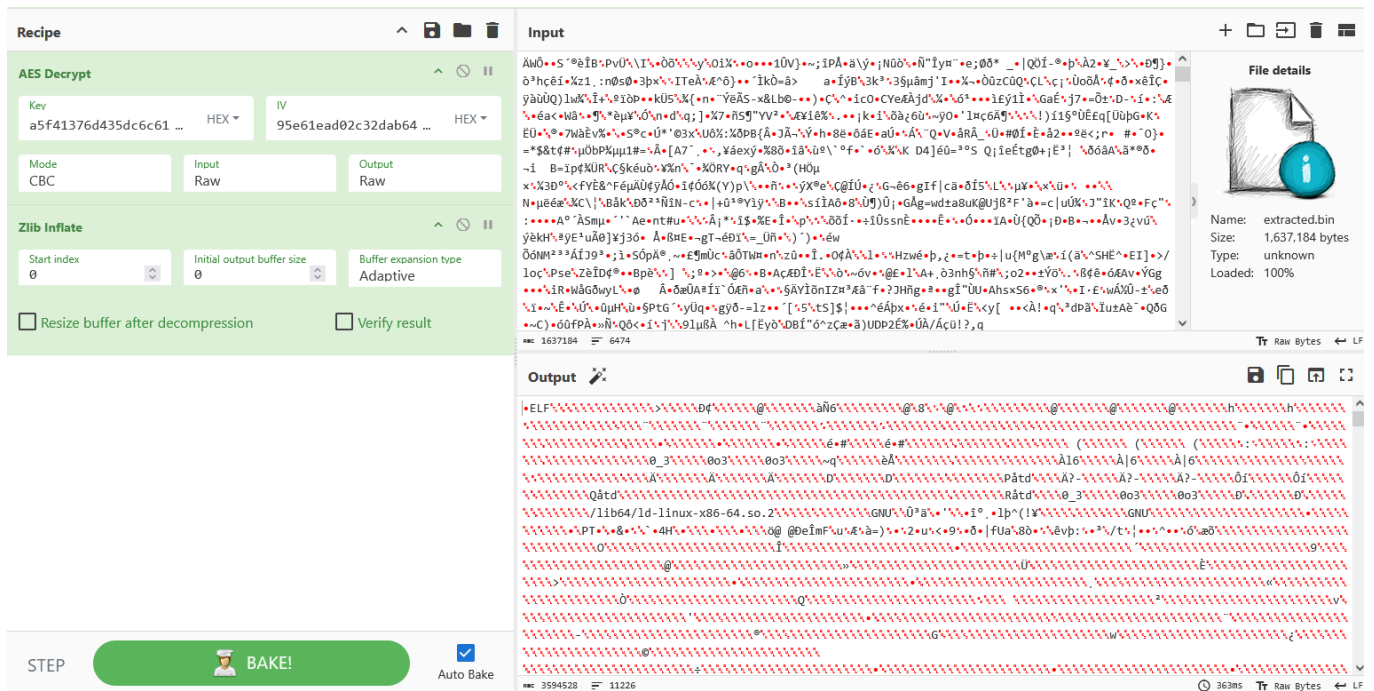
```
.data:00000000002893A0 ; char byte_2893A0[935296]
.byte_2893A0 db 0C4h, 'W', 0D4h, 95h, 8Ah, 'S', e
; DATA XREF:
.db 'B', 1Ch, 'P', 'v', 0DCh, 7, '\',
.db 0F5h, 1Bh, 12h, 4, 'y', 12h, 'O',
.db 'o', 0ADh, 88h, 84h, 'i', 0D8h, '
.db ';, 0EEh, 'P', 0C5h, 88h, 0E4h,
.db 'N', 0FBh, 0F2h, 12h, 99h, 0D1h,
.db 0A8h, 98h, 'e', ';', 0D8h, 0F0h,
.db '!', 'Q', 0D6h, 0CDh, '-', 0AEh,
.db '2', 88h, 0A5h, '-', 3, '>', 15h,
.db 87h, 0F2h, 0B3h, 'h', 0E7h, 0EAh,
.db 'I', 0B8h, ':', 'n', 0D8h, 's', e
.db 0D7h, 4, 0Bh, 'I', 'T', 'e', 0C0h
.db '!', 8Ah, 8Bh, 0B4h, 0CCh, 'k', e
.db 9, 'a', 94h, 0CDh, 0FDh, 'B', 5,
.db '3', 0A7h, 0B5h, 0E2h, 'm', 'j',
.db 0BCh, 0ACh, 83h, 0D2h, 0FBh, 'z',
.db 0C7h, 'L', 15h, 0E7h, 0A1h, 1Ch,
.db 0Ch, 0A2h, 94h, 0F0h, 99h, 0D7h,
.db 0Fh, 0E0h, 0F9h, 0D9h, 'Q', ')',
.db 0CEh, '+', 10h, 0BAh, 0EFh, 0F2h,
.db 0DCh, '5', 13h, 0BEh, '{', 8Ch, '
.db 0EBh, 0C3h, 'S', '-', 0D7h, '8', '
.db 2 dup(9Eh, ')', 97h, 0C7h, 1Ah,
.db 'O', 85h, 'C', 'Y', 'e', 0C6h, 0C
.db 0BCh, 90h, 1, 0F3h, 0B9h, 2 dup(8
.db 0C0h, '!', 0CCh, 8Ah, 6, 'C', 'S'
```

Size is 1637173 (0x18fb40).

I will use **Binary Refinery** (<https://github.com/binref/refinery/>) tool to extract data from the binary:

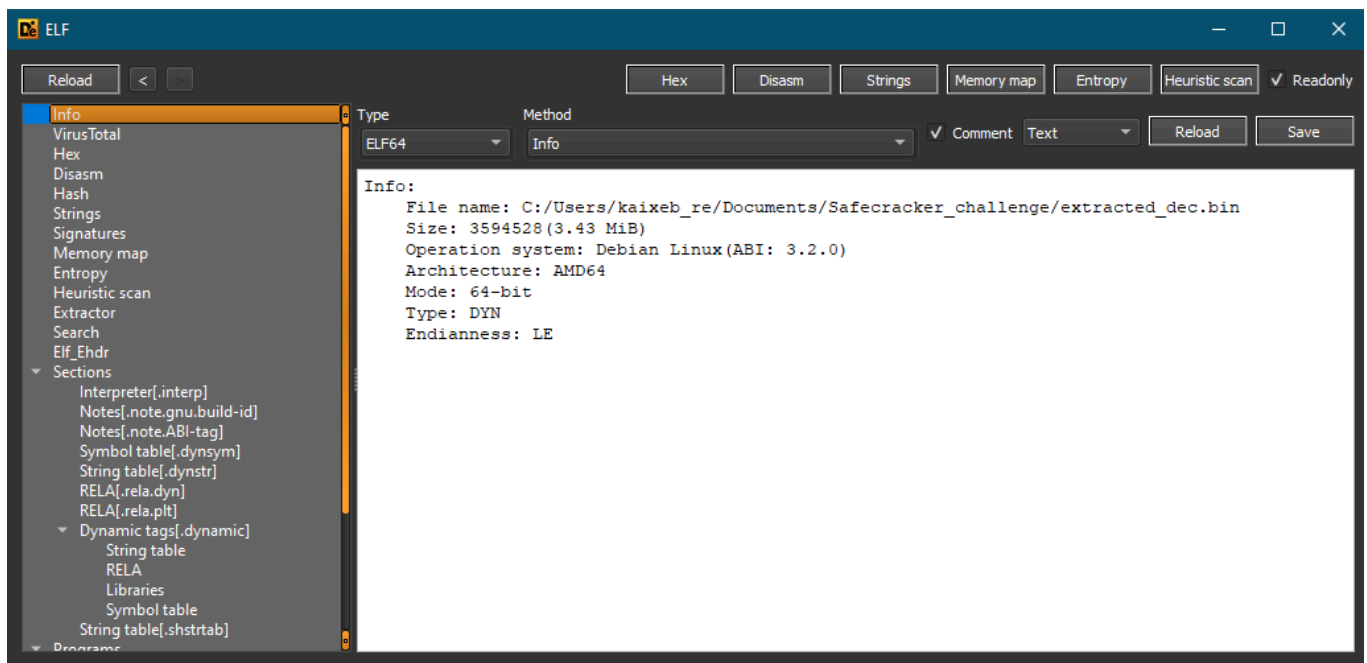
```
PowerShell 7 (x64)
PS C:\Users\kaixeb_re\Documents\Safecracker_challenge> emit .\MsMpEng.exe | vsnip 0x2893a0:0x18fb40 | dump extracted.
bin
PS C:\Users\kaixeb_re\Documents\Safecracker_challenge> _
```

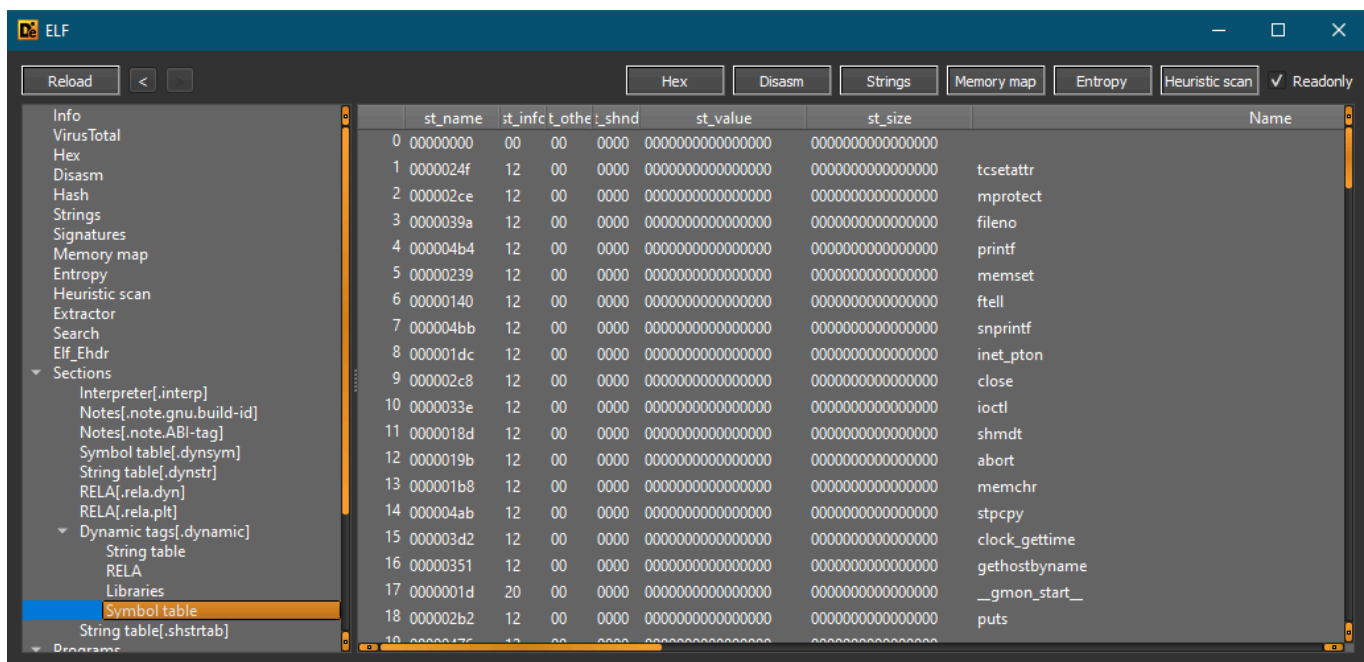
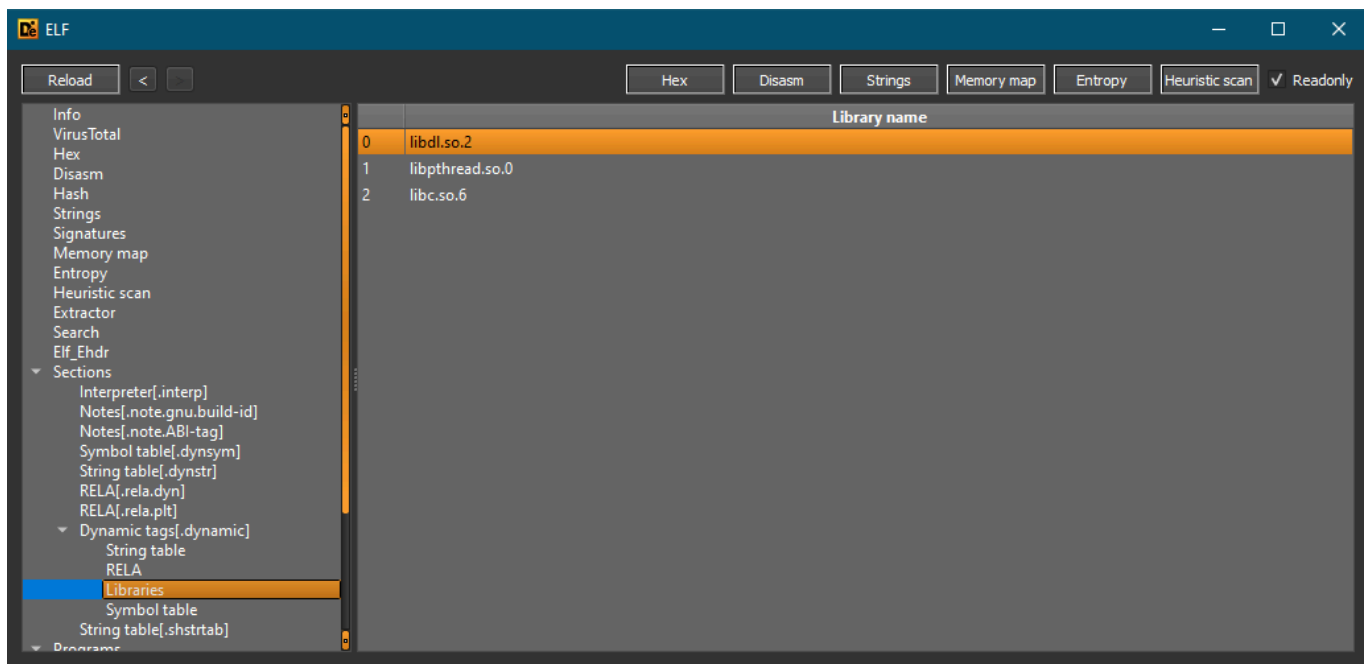
Then decrypt and inflate it in **CyberChef**:



Here we can see the start of the ELF header.

Looks like like the CBC mode is the right one and everything decrypted and decompressed correctly:





Q6: What encryption was the packer using?

Answer: **AES-256-CBC**

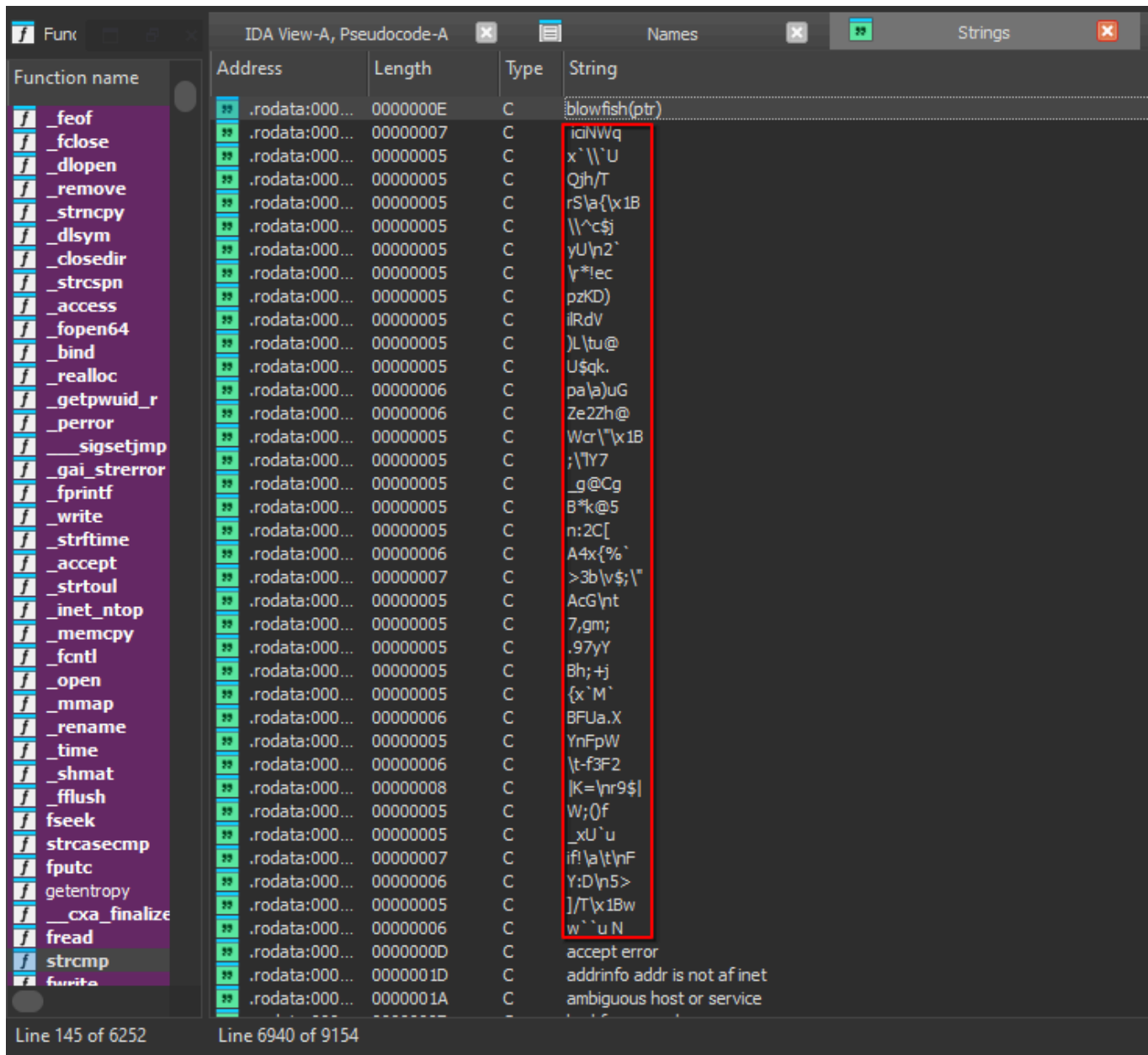
There is a **libpthread** library used, which provides means to manage threads, and multithreading is usual for filesystem encryption programs.

Another interesting thing here is **inet_pton** function in the symbol table. It's utilized for converting IP addresses.

Looking at strings, there are a lot of them and in clear text, so this is the final executable file, I suppose.

Function name	Address	Length	Type	String
_feof	.rodata:000...	00000013	C	gostr3411-2012-512
_fclose	.rodata:000...	00000019	C	\"; boundary=\"----%s\"%s%s
_dlopen	.rodata:000...	0000000D	C	%s-----%s%s
_remove	.rodata:000...	0000001B	C	Content-Type: %ssignature;
_strncpy	.rodata:000...	00000014	C	name=\"smime.p7s\"%s
_dlsym	.rodata:000...	0000001A	C	filename=\"smime.p7s\"%s%s
_closedir	.rodata:000...	00000011	C	%s-----%s--%s%s
_strcspn	.rodata:000...	0000000D	C	name=\"%s\"%s
_access	.rodata:000...	00000011	C	filename=\"%s\"%s
_fopen64	.rodata:000...	00000016	C	Content-Type: %ssmime;
_bind	.rodata:000...	00000010	C	smime-type=%s;
_realloc	.rodata:000...	00000014	C	-----BEGIN %s-----\n
_getpwuid_r	.rodata:000...	00000012	C	-----END %s-----\n
_perror	.rodata:000...	00000020	C	Content-Type: multipart/signed;
__sigsetjmp	.rodata:000...	00000025	C	This is an S/MIME signed message%s%s
_gai_strerror	.rodata:000...	00000024	C	Content-Transfer-Encoding: base64%s
_fprintf	.rodata:000...	00000021	C	Content-Disposition: attachment;
_write	.rodata:000...	00000026	C	Content-Transfer-Encoding: base64%s%s
_strftime	.rodata:000...	00000017	C	crypto/asn1/bio_ndef.c
_accept	.rodata:000...	00000017	C	crypto/asn1/p5_pbev2.c
_strtol	.rodata:000...	0000000C	C	PBKDF2PARAM
_inet_ntop	.rodata:000...	0000000A	C	PBE2PARAM
_memcpy	.rodata:000...	00000008	C	keyfunc
_fcntl	.rodata:000...	00000018	C	crypto/asn1/p5_scrypt.c
_open	.rodata:000...	0000000E	C	SCRIPT_PARAMS
_mmap	.rodata:000...	0000000E	C	costParameter
_rename	.rodata:000...	0000000A	C	blockSize
_time	.rodata:000...	00000019	C	parallelizationParameter
_shmat	.rodata:000...	0000000A	C	keyLength
_fflush	.rodata:000...	00000013	C	failed to set pool
_fseek	.rodata:000...	00000017	C	failed to swap context
_strcasecmp	.rodata:000...	00000012	C	invalid pool size
_fputc	.rodata:000...	0000000E	C	async_ctx_new
_getentropy	.rodata:000...	00000012	C	ASYNC_init_thread
__cxa_finalize	.rodata:000...	0000000E	C	ASYNC_job_new
_fread	.rodata:000...	00000010	C	ASYNC_pause_job
_strcmp	.rodata:000...	00000011	C	async_start_func
_fwrite	.rodata:000...	00000010	C	ASYNC_start_job
	.rodata:000...	0000001B	C	ASYNC_WAIT_CTX_set_wait_fd

But I managed to find some of them in unreadable form. Maybe be there is some decoding/decryption technique will be used, even blowfish, judging by string above.



As usual, let's start inspecting and marking up the binary from the entry point (function start), which leads straight to the main() function:

```

2 void __fastcall __noreturn start(__int64 a1, __int64 a2, void (*a3)(void))
3 {
4     __int64 v3; // rax
5     int v4; // esi
6     __int64 v5; // [rsp-8h] [rbp-8h] BYREF
7     char *retaddr; // [rsp+0h] [rbp+0h] BYREF
8
9     v4 = v5;
10    v5 = v3;
11    _libc_start_main(main, v4, &retaddr, init, fini, a3, &v5);
12    __halt();
13 }

```



```

1 __int64 __fastcall main(void **a1, char **a2, char **a3, __int64 a4, __int64 a5, __int64 a6)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5     v12 = &unk_36DFC0;
6     v14 = "daV324982S3bh2";
7     v13 = 0LL;
8     v15 = 14LL;
9     v16 = 0LL;
10    v17 = 0LL;
11    if ( (unsigned int)mw_check_debug_handle_sigsegv() )
12        goto LABEL_2;
13    if ( (unsigned int)sub_4AA3D(a1, a2, v6, v7, v8, v9, v12, v13, v14, v15, v16, v17) )
14        goto LABEL_2;
15    a1 = (void **)(byte_9 + 2);
16    raise(11);
17    if ( (unsigned int)sub_4A3B5() )
18        goto LABEL_2;
19    a1 = &v12;
20    if ( (unsigned int)sub_4A3F6(&v12) )
21        goto LABEL_2;
22    raise(11);
23    puts("Running update, testing update endpoints");
24    a1 = &v12;
25    if ( (unsigned int)sub_4AB00((__int64)&v12)
26        || (a2 = (char **)&v12, a1 = (void **)"/mnt/c/Users", (unsigned int)sub_4AC39("/mnt/c/Us
27        || (raise(11), sub_4A8F6((__int64)&v12, (__int64)&v12, v10), a1 = &v12, (unsigned int)su
28    {
29 LABEL_2:
30    sub_28164A(a1, a2);

```

First function refers to debugging the program and exception handling. I have marked it a little bit but it is not really needed, because binary is not stripped at all:

```

1 __int64 mw_check_debug_handle_sigsegv()
2 {
3     __sigset_t *p_sa_mask; // rdi
4     __int64 i; // rcx
5     struct sigaction act; // [rsp+8h] [rbp-A0h] BYREF
6
7     if ( mw_get_tracer_pid() )
8     {
9         puts("*****DEBUGGED*****");
10    }
11    else
12    {
13        p_sa_mask = &act.sa_mask;
14        for ( i = 36LL; i; --i ) // emptying blocked signals set
15        {
16            LODWORD(p_sa_mask->__val[0]) = 0;
17            p_sa_mask = (__sigset_t *)((char *)p_sa_mask + 4);
18        }
19        act.sa_flags = SA_SIGINFO; // send additional info with signal to the handler
20        act.sa_handler = (__sighandler_t)mw_nulify_sigsegv_signal;
21        if ( sigaction(SIGSEGV, &act, 0uLL) == -1 ) // -1 is returned on error
22            _exit(1);
23    }
24    return 0LL;
25 }

```

Going inside one function which I called `mw_get_tracer_pid()`, we see that `TracerPid` value of the current process is being read:

```

1 int mw_get_tracer_pid()
2 {
3     FILE *v0; // rax
4     FILE *v1; // rbx
5     char *v2; // rdi
6     char *v4; // rdi
7     char *v5; // [rsp+0h] [rbp-408h] BYREF
8     char s[1024]; // [rsp+8h] [rbp-400h] BYREF
9
10    v5 = 0LL;
11    v0 = fopen("/proc/self/status", "r");
12    if ( v0 )
13    {
14        v1 = v0;
15        while ( fgets(s, 990, v1) )
16        {
17            v2 = strstr(s, "TracerPid");
18            if ( v2 )
19            {
20                if ( strtok_r(v2, ":", &v5) )
21                {
22                    v4 = strtok_r(0LL, ":", &v5);
23                    if ( v4 )
24                        return atoi(v4);
25                }
26                return -1;
27            }
28        }
29    }

```

If the process is being debugged, then that value is not 0 and string "*****DEBUGGED*****" is printed.

Thus, we can answer 11-th, 15-th, 19-th questions:

Q11: The binary appears to check for a debugger, what file does it check to achieve this?

Answer: **/proc/self/status**

Q15: If the malware detects a debugger, what string is printed to the screen?

Answer: *******DEBUGGED*******

Q19: What string does the binary look for when looking for a debugger?

Answer: **TracerPid**

Returning back, if debugger is not detected, then **SIGSEGV** handle action is changed to nothing:

```

11 else
12 {
13     p_sa_mask = &act.sa_mask;
14     for ( i = 36LL; i; --i )           // emptying blocked signals set
15     {
16         LODWORD(p_sa_mask->__val[0]) = 0;
17         p_sa_mask = (__sigset_t *)((char *)p_sa_mask + 4);
18     }
19     act.sa_flags = SA_SIGINFO;           // send additional info with signal to the handler
20     act.sa_handler = (__sighandler_t)mw_nulify_sigsegv_signal;
21     if ( sigaction(SIGSEGV, &act, 0uLL) == -1 ) // -1 is returned on error
22         _exit(1);
23 }
24 return 0LL;
25 }

```

```

1 __int64 mw_nulify_sigsegv_signal()
2 {
3     __int64 result; // rax
4     sigaction oact; // [rsp+8h] [rbp-A0h] BYREF
5
6     result = (unsigned int)(sigaction(SIGSEGV, 0LL, &oact) + 1);
7     if ( !(_DWORD)result )           // -1 is returned on sigaction() error,
8                                     // so if result is 0, then error happened
9                                     // and program exits
10         _exit(1);
11     return result;
12 }

```

And later there is a function that actually raises this exception:

```

1 int __fastcall mw_raise_sigsegv(__int64 a1)
2 {
3     __int64 i; // rbx
4     int result; // eax
5
6     for ( i = *(_QWORD *)(a1 + 8); i; i = *(_QWORD *)(i + 40) )
7         result = raise(SIGSEGV);
8     return result;
9 }

```

Q12: What exception does the binary raise?

Answer: **SIGSEGV**

Further on, second function takes some string looking like a key - "daV324982S3bh2".

```

1 __int64 __fastcall main(void **a1, char **a2, char **a3, __int64 a4, __int64 a5, __int64 a6)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5     v12 = &unk_36DFC0;
6     v14 = "daV324982S3bh2";
7     v13 = 0LL;
8     v15 = 14LL;
9     v16 = 0LL;
10    v17 = 0LL;
11    if ( (unsigned int)mw_check_debug_handle_sigsegv() )
12        goto LABEL_2;
13    if ( (unsigned int)sub_4AA3D(a1, a2, v6, v7, v8, v9, v12, v13, v14, v15, v16, v17) )
14        goto LABEL_2;

```

But inside is only memory manipulation stuff that doesn't look meaningful.

Moreover, in disassembler listing it doesn't take those args in registers so then to use in function:

```

|004A209                xor     eax, eax
|004A20B                call    sub_28164A
|-----|
|004A210 ; -----|
|004A210                |
|004A210 loc_4A210:                ; CODE XREF: main+49↑j
|004A210                xor     eax, eax
|004A212                call    mw_mem_manip
|004A217                test    eax, eax
|004A219                jnz     short loc_4A209

```

After that one function takes encrypted-like data as an argument:

<pre> unk_36DFC0 db 0Ch db 15h db 22h ; " db 43h ; C db 41h ; A db 0Eh db 16h db 17h db 42h ; B db 32h ; 2 db 40h ; @ db 16h db 0Dh db 50h ; P db 0Dh db 0Fh db 78h ; x </pre>	<pre> 5 v12 = &unk_36DFC0; 6 v14 = "daV324982S3bh2"; 7 v13 = 0LL; 8 v15 = 14LL; 9 v16 = 0LL; 10 v17 = 0LL; 11 if ((unsigned int)mw_check_debug_handle_ 12 goto LABEL_2; 13 if ((unsigned int)mw_mem_manip(a1, a2, v 14 goto LABEL_2; 15 a1 = (void **)(byte_9 + 2); 16 raise(11); 17 if ((unsigned int)ret_zero()) 18 goto LABEL_2; 19 a1 = &v12; 20 if ((unsigned int)sub_4A3F6(&v12)) 21 goto LABEL_2; </pre>
--	---

And inside is a function that XORes data:

```

1 __int64 __fastcall xor_decrypt_strings(__int64 a1, __int64 a2, __int64 a3, __int64 a4, unsigned __int64 a5)
2 {
3     unsigned __int64 i; // rcx
4
5     for ( i = 0LL; i != a4; ++i )
6         *(_BYTE *)(a3 + i) = *(_BYTE *)(a1 + i) ^ *(_BYTE *)(a2 + i % a5);
7     *(_BYTE *)(a3 + i) = 0;
8     return 0LL;
9 }

```

Marked up version:

```
1 __int64 __fastcall mw_xor_decrypt_data(  
2     __int64 enc_data_ptr,  
3     __int64 var_22,  
4     __int64 dec_data_ptr,  
5     __int64 size,  
6     unsigned __int64 var_41)  
7 {  
8     unsigned __int64 i; // rcx  
9  
10    for ( i = 0LL; i != size; ++i )  
11        *(_BYTE *)(dec_data_ptr + i) = *(_BYTE *)(enc_data_ptr + i) ^ *(_BYTE *)(var_22 + i % var_41);  
12    *(_BYTE *)(dec_data_ptr + i) = 0;  
13    return 0LL;  
14 }
```

Then goes another XOR function:

```
byte_3684A0    db  4Ah                ; DATA XREF: mw_string_decryptio  
              db  11h  
              db  32h ; 2  
              db  55h ; U  
              db   0  
              db   0  
              db   0  
              db   0  
              db  4Ah ; J  
              db   5  
              db  39h ; 9  
              db  50h ; P  
              db  4Ah ; J  
              db   0  
              db   0  
              db   0  
              db  4Ah ; J  
              db  11h  
              db  26h ; &  
              db  47h ; G  
              db   0  
              db   0  
              db   0  
              db   0  
              db  4Ah ; J  
              db  11h
```

```

1 __int64 __fastcall mw_string_decryption(char *haystack)
2 {
3     const char *v1; // r15
4     int v2; // ebx
5     int v3; // r13d
6     size_t v4; // rax
7     unsigned __int64 v5; // rcx
8
9     v1 = a1;
10    v2 = 0;
11    v3 = dword_368484;
12    while ( 1 )
13    {
14        if ( v3 <= v2 )
15            return 0LL;
16        v4 = strlen(v1);
17        v5 = 0LL;
18        while ( v4 != v5 )
19        {
20            needle[v5] = v1[v5] ^ aDav324982s3bh2[v5 % 0xE];
21            if ( v4 < ++v5 )
22                goto LABEL_7;
23        }
24        needle[v4] = 0;
25    LABEL_7:
26        v1 += 8;
27        if ( strstr(haystack, needle) )
28            return 1LL;
29        ++v2;
30    }

```

Finally it uses the key defined at the start and not bytes from encrypted data itself like in the first case.

Marked up version:

```

1 __int64 __fastcall mw_string_decryption(char *string_array)
2 {
3     const char *encrypted_strings; // r15
4     int current_strings_num; // ebx
5     int total_strings_num; // r13d
6     size_t current_enc_string_len; // rax
7     unsigned __int64 curr_enc_string_byte_index; // rcx
8
9     encrypted_strings = byte_3684A0;
10    current_strings_num = 0;
11    total_strings_num = dword_368484;
12    while ( 1 )
13    {
14        if ( total_strings_num <= current_strings_num )
15            return 0LL;
16        current_enc_string_len = strlen(encrypted_strings);
17        curr_enc_string_byte_index = 0LL;
18        while ( current_enc_string_len != curr_enc_string_byte_index )// iterate through each byte of the encr
19        {
20            decrypted_strings_arr[curr_enc_string_byte_index] = encrypted_strings[curr_enc_string_byte_index] ^ ;
21            if ( current_enc_string_len < ++curr_enc_string_byte_index )
22                goto LABEL_7;
23        }
24        decrypted_strings_arr[current_enc_string_len] = 0;
25    LABEL_7:
26        encrypted_strings += 8;
27        if ( strstr(string_array, decrypted_strings_arr) )
28            return 1LL;
29        ++current_strings_num;
30    }
31 }

```

Q5: What is the XOR key used for the encrypted strings?

Answer: **daV324982S3bh2**

Next function takes directory path `/mnt/c/Users` as an argument:

```

1 __int64 __fastcall main(void **a1, char **a2, char **a3, __int64 a4, __int64 a5, __int64 a6)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5     encrypted_data = &unk_36DFC0;
6     key = "daV324982S3bh2";
7     v13 = 0LL;
8     v15 = 14LL;
9     v16 = 0LL;
10    v17 = 0LL;
11    if ( (unsigned int)mw_check_debug_handle_sigsegu()
12        || (unsigned int)mw_mem_manip(a1, a2, v6, v7, v8, v9, encrypted_data, v13, key, v15, v16, v17)
13        || (raise(SIGSEGV), (unsigned int)ret_zero())
14        || (unsigned int)sub_4A3F6(&encrypted_data)
15        || (raise(SIGSEGV),
16            puts("Running update, testing update endpoints"),
17            (unsigned int)mw_uri_check((__int64)&encrypted_data))// here strings are unencrypted
18        || (unsigned int)mw_compare_filenames_to_decrypted_strings("/mnt/c/Users", (__int64)&encrypted_data)
19        || (raise(SIGSEGV),
20            sub_4A8F6((__int64)&encrypted_data, (__int64)&encrypted_data, v10),
21            (unsigned int)sub_4A5C1((__int64)&encrypted_data)) )
22    {
23        sub_28164A();
24    }
25    raise(11);
26    puts("-----");
27    puts("Configuration Successful\nYou can now connect to the Corporate VPN");
28    return 0LL;
29 }

```

It recursively reads every file inside that directory, decrypts string array and tries to match decrypted string and filename. Matches are placed into other array.

```
3 DIR *v2; // rbp
4 struct dirent *v3; // rbx
5 unsigned __int8 d_type; // a1
6 char filenames_array[4152]; // [rsp+0h] [rbp-1038h] BYREF
7
8 v2 = opendir(a1);
9 if ( !v2 )
10     return 1LL;
11 while ( 1 )
12 {
13     v3 = readdir(v2);
14     if ( !v3 )
15         break;
16     raise(SIGSEGV);
17     snprintf(filenames_array, 4096uLL, "%s/%s", a1, v3->d_name);
18     d_type = v3->d_type;
19     if ( d_type == DT_DIR )
20     {
21         if ( !mw_check_specifict_dir(v3->d_name) )
22             mw_compare_filenames_to_decrypted_strings(filenames_array, a2);
23     }
24     else if ( d_type == DT_REG )
25     {
26         if ( (unsigned int)mw_string_decrypt_search(filenames_array) )
27             mw_collect_decrypted_strings(a2, filenames_array);
28     }
29 }
30 closedir(v2);
31 return 0LL;
32 }
```

Q9: What was the target directory for the ransomware?

Answer: /mnt/c/Users

For Q21 - What system call is utilized by the binary to list the files within the targeted directories? - the system call is needed for the function `readdir`, which actually lists files from the directory.

By searching on the net: "On Linux (and many other Unix-like systems), the primary system call utilized by `readdir()` to list directory entries is `getdents` (or its 64-bit variant, `getdents64`)".

So, the answer is **getdents64**.

Last function to inspect is gotta be a function with file encryption and ransomware functionality.

At the end of it is the snippet to delete original files:

```
67     if ( remove(*arr_filenames) )
68         fputs("Failed to delete original file", stderr);
69     arr_filenames = (const char **)arr_filenames[1];
```

And from Linux manual page we get the answer to the 22nd question:

DESCRIPTION [top](#)

`remove()` deletes a name from the filesystem. It calls `unlink(2)` for files, and `rmdir(2)` for directories.

Q22: Which system call is used to delete the original files?

Answer: **unlink**

Searching through the strings I find home directory of some user named "blitztide":

```
00000020 C OpenSSL 1.1.1u-dev xx XXX xxxx
00000044 C OPENSSLDIR: "/home/blitztide/Projects/Payloads/LockPick3.0/lib/ssl/"
00000174 C compiler: gcc -pthread -m64 -Wa,--noexecstack -Wall -O3 -static -DOPENSSL_U
```

Q20: It appears that the attacker has bought the malware strain from another hacker, what is their handle?

Answer: **blitztide**

Malware is for linux and linux doesn't use PE files, so it must be **.exe** extension, which is not targeted by the malware.

Q13: Out of this list, what extension is not targeted by the malware?

.pptx, .pdf, .tar.gz, .tar, .zip, .exe, .mp4, .mp3

Answer: **.exe**