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

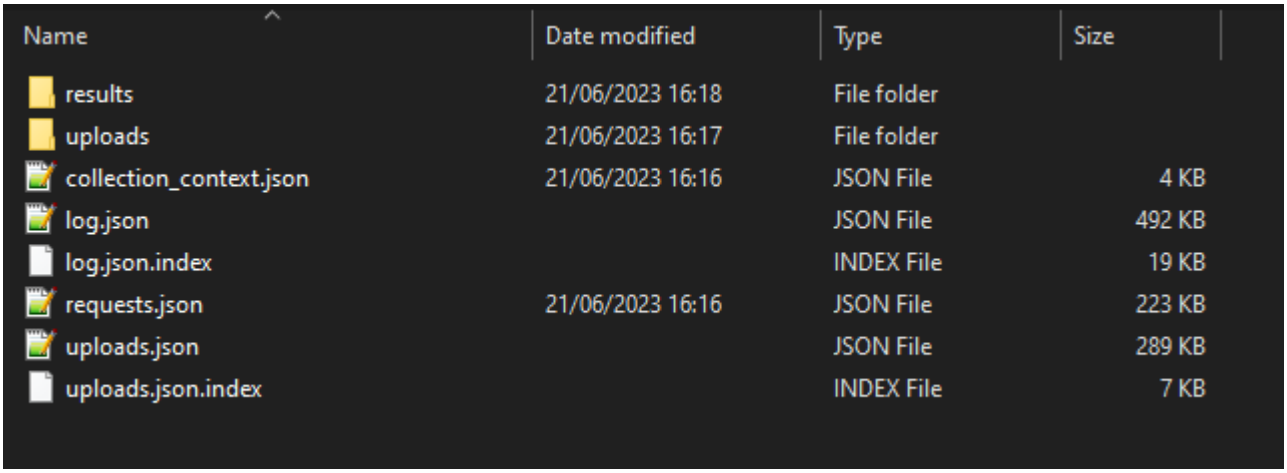
This is the "**insane**" level sherlock task with category "**Malware Analysis**" from "Hack The Box" cybersecurity platform.

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 ransomwared. 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.`

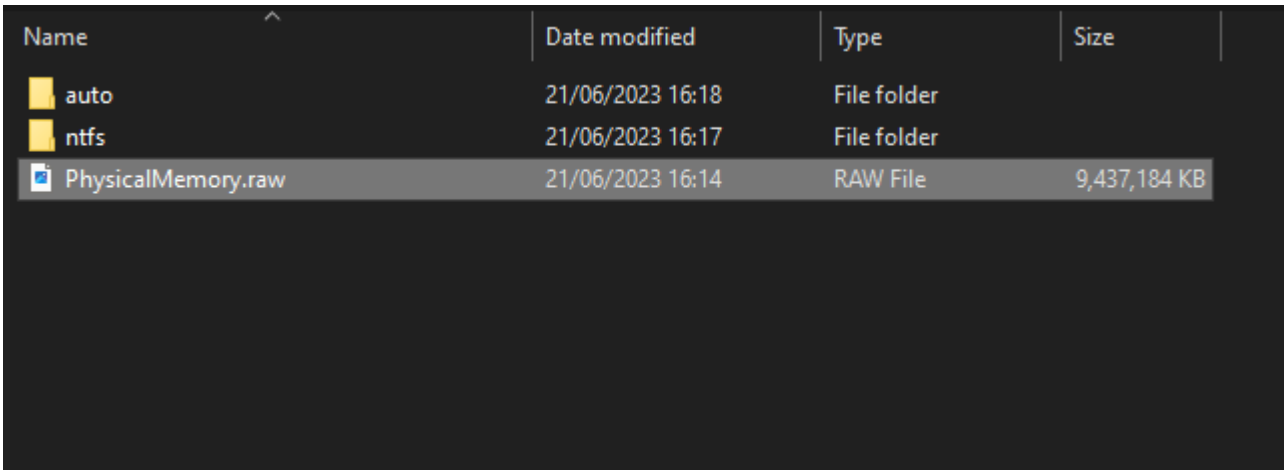
First look

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.

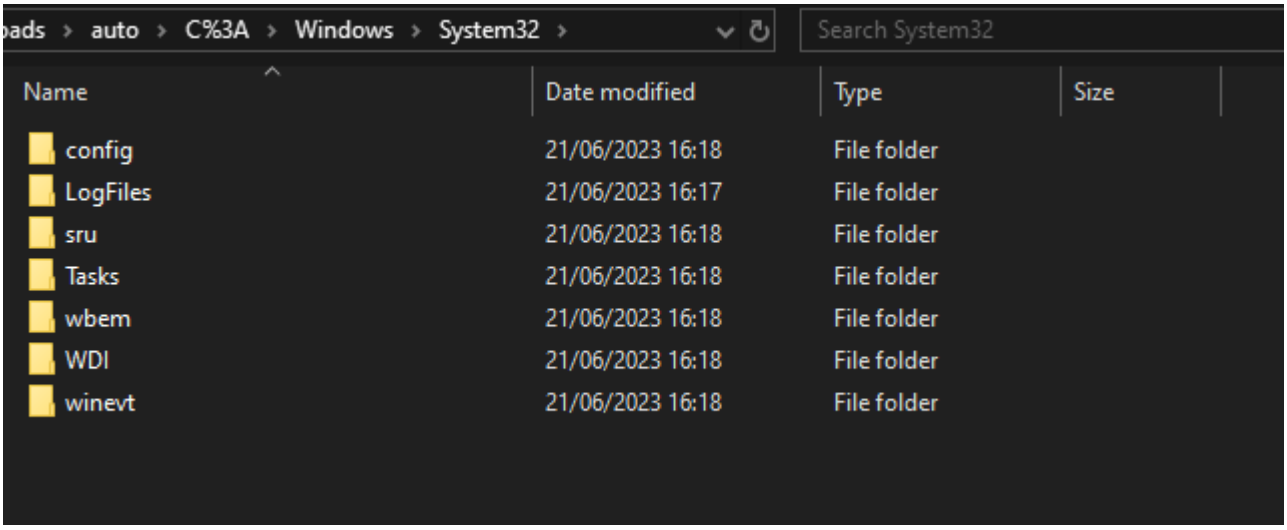
Let's download 1 GB size file in attachment, unarchive and look what's inside:



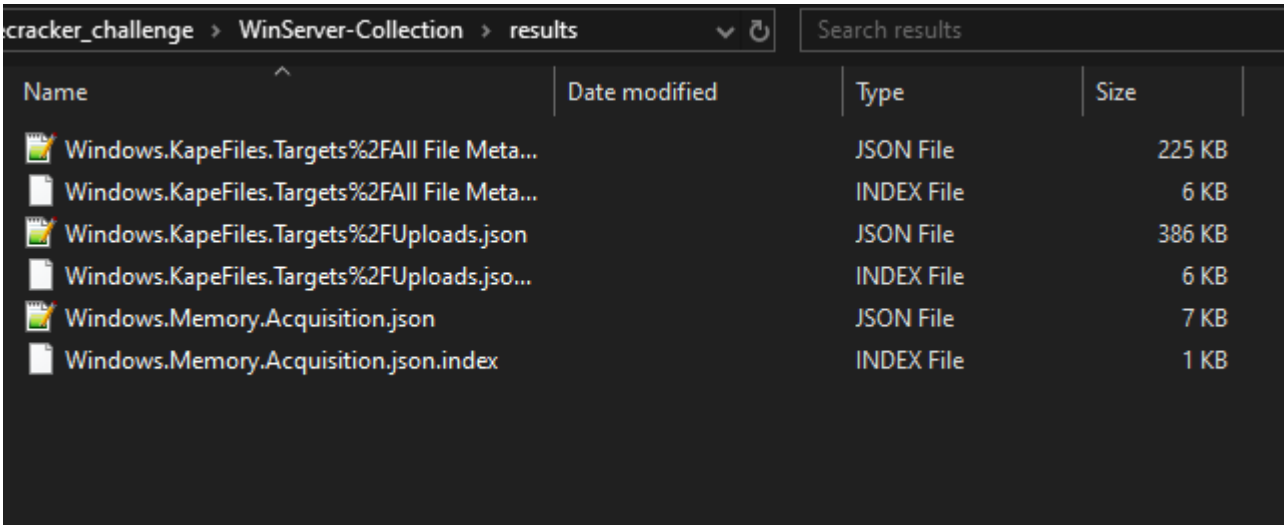
Okay, in "Uploads" directory is a dump of physical memory:



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



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

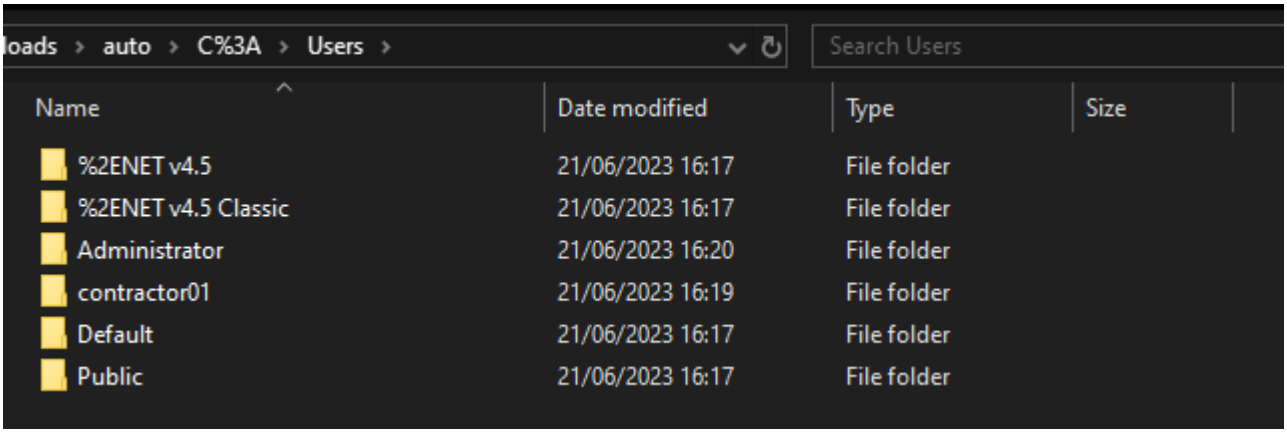


Time to investigate what has happened.

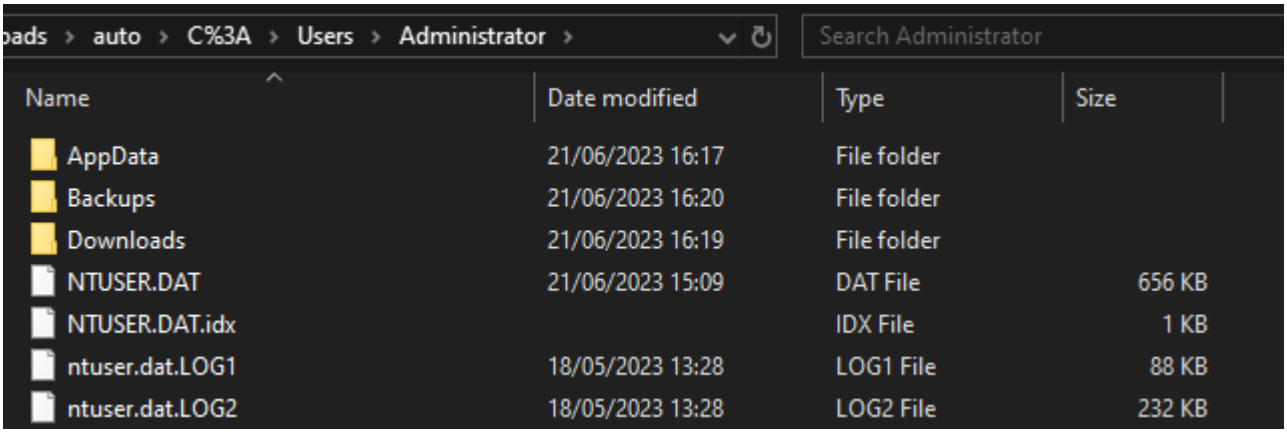
Filesystem investigation

First question is:
Which user account was utilized for initial access to our company server?

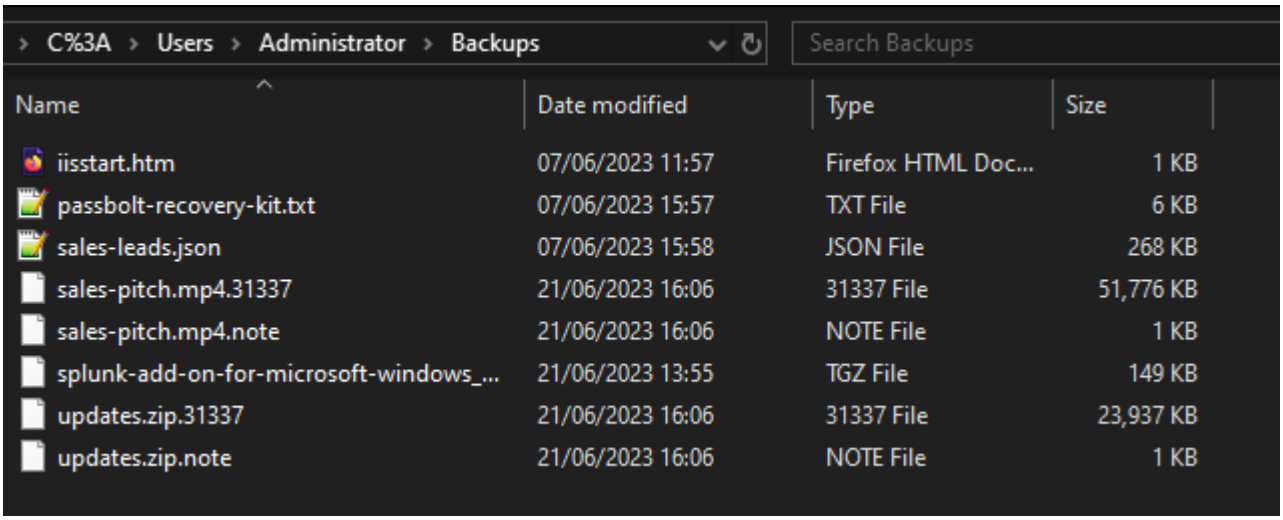
So, now I have to look at forensics artifacts, I guess.
User account and initial access may be found in logs I think, but I will just start by casually walking through users directories.



.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".



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



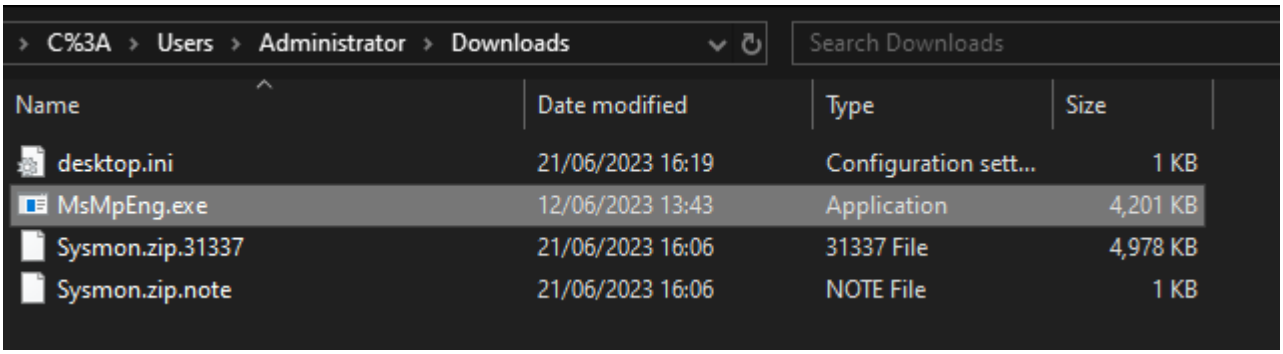
Contents of the first "note" file look like this:

```
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
```

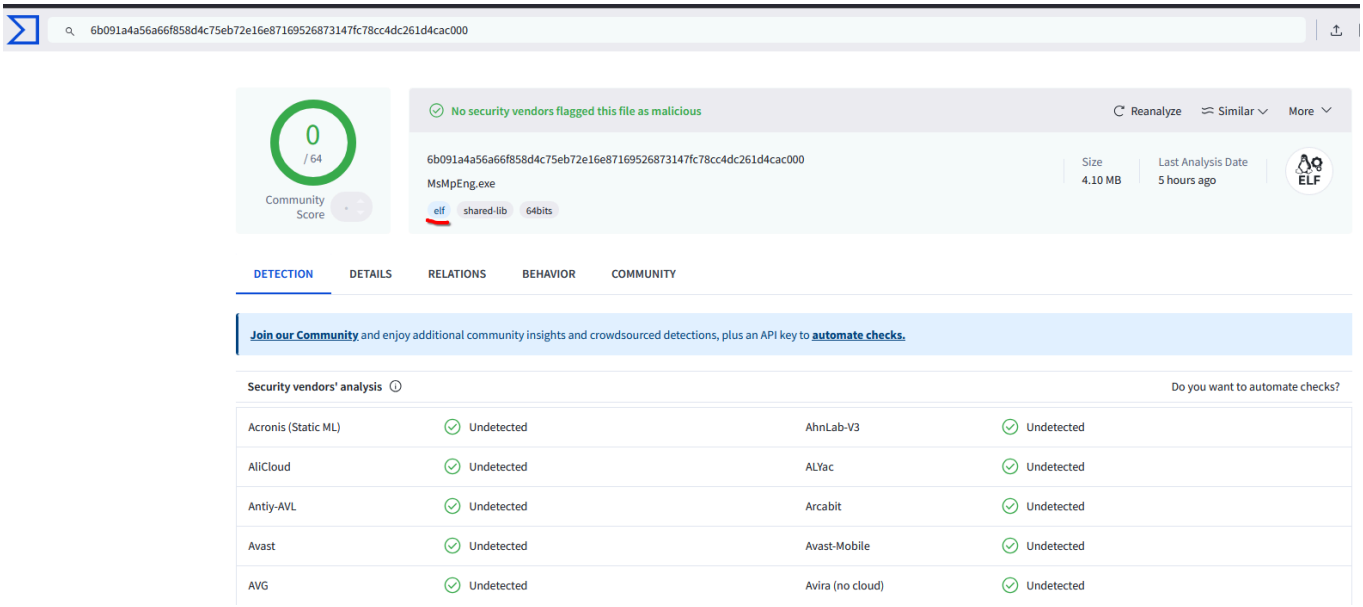
Thus, file extension `.31337` must be the encrypted version of original file.
This will help me answer the third and 17-th questions:
Q3: **How many files have been encrypted by the the ransomware deployment?**
Q17: **What file extension does the ransomware rename files to?**
Answer: **.31337**

There is a BTC address so Q18 can be answered too.
Answer to Q18: **16ftSEQ4ctQFDtVZiUBusQUjRrGhM3JYwe**

Even more weird the contents of "Downloads" folder:



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**.



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:

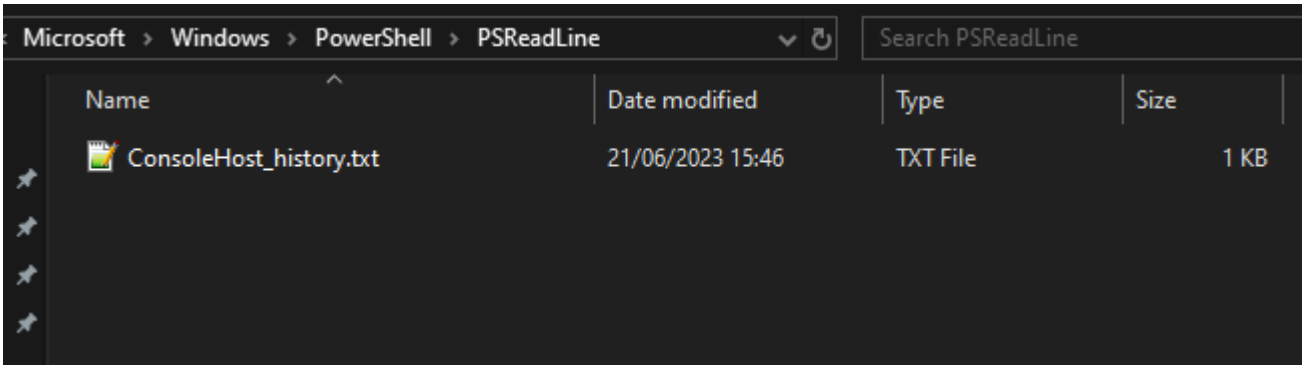
```
wsl --install
Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.SecurityProtocolType]::Tls12; iex ((New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1
'))
```

```
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
```

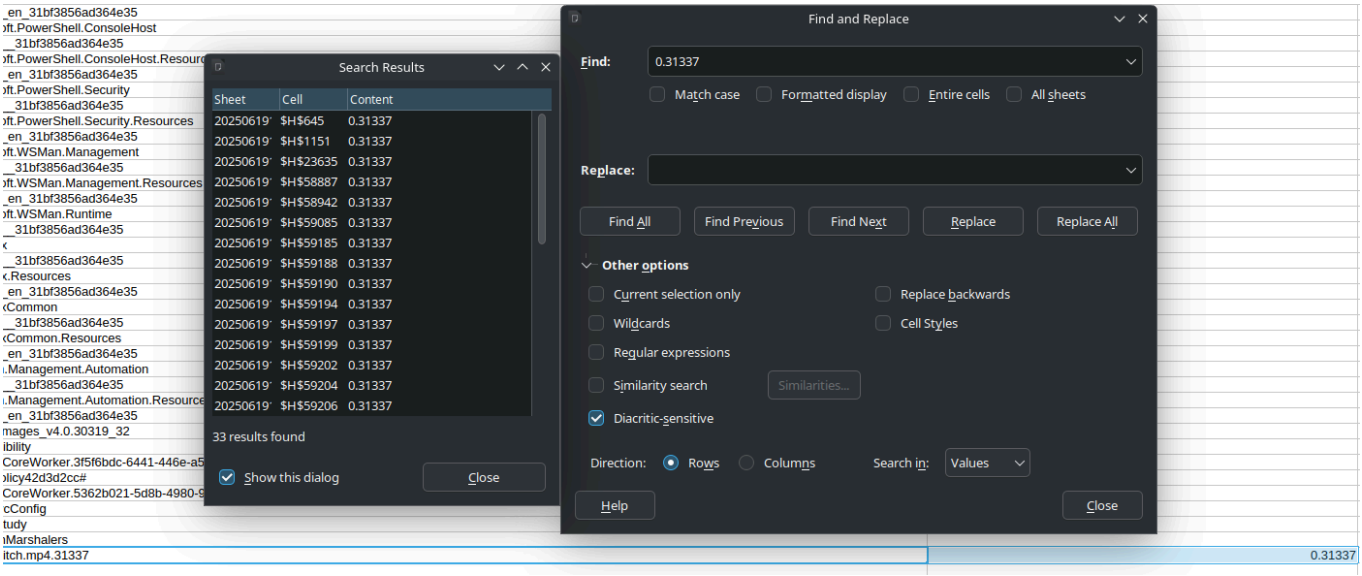
Command `ubuntu` means, that the WSL has been started with Ubuntu Linux distribution previously configured by "Administrator".
Aside the reconnaissance commands, the `PsExec` with `-s` flag executing `cmd.exe` runs shell under SYSTEM account. This is not normal at all so user **contractor01** must be 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>
```

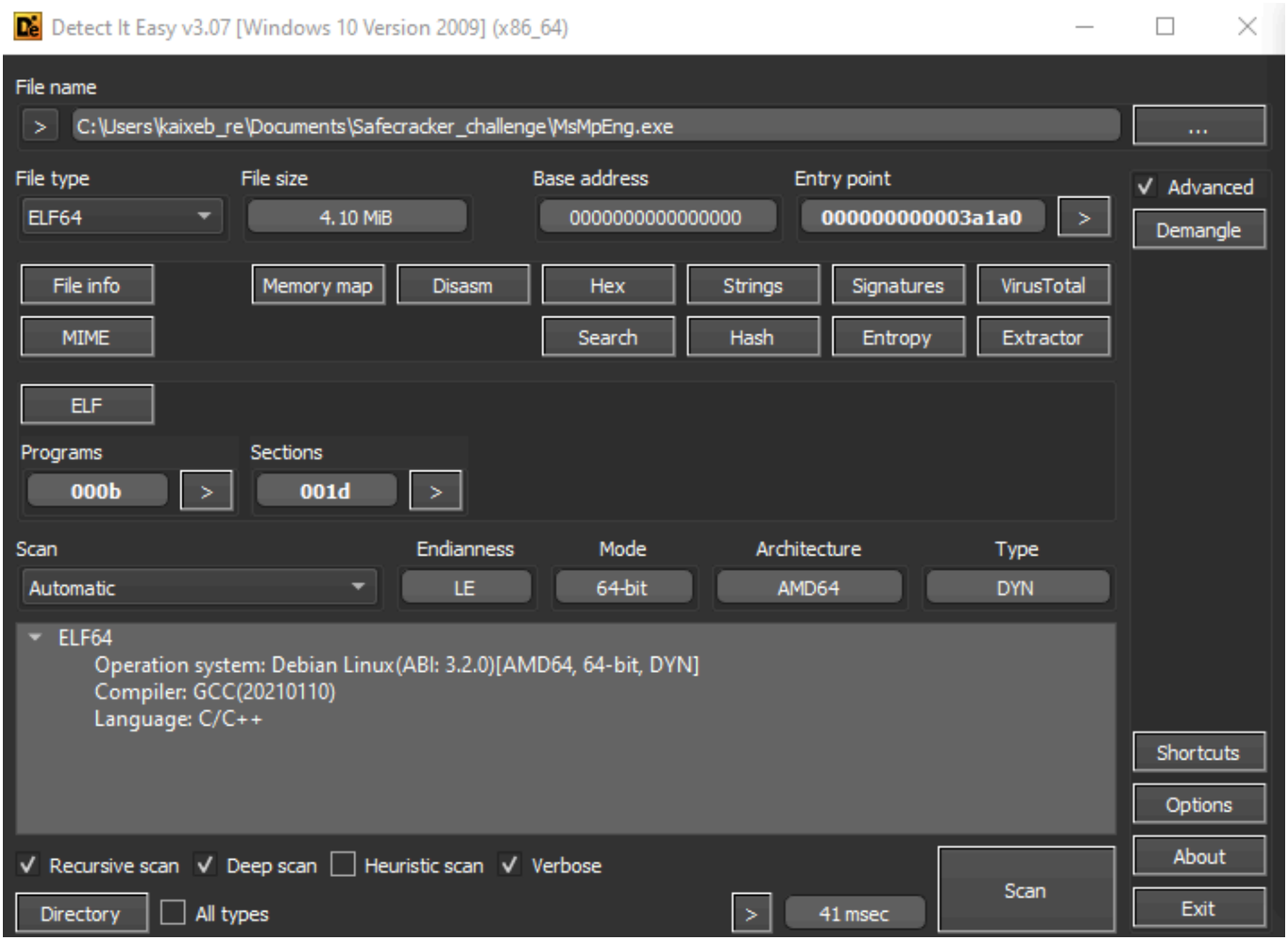
Show all files with that extension:



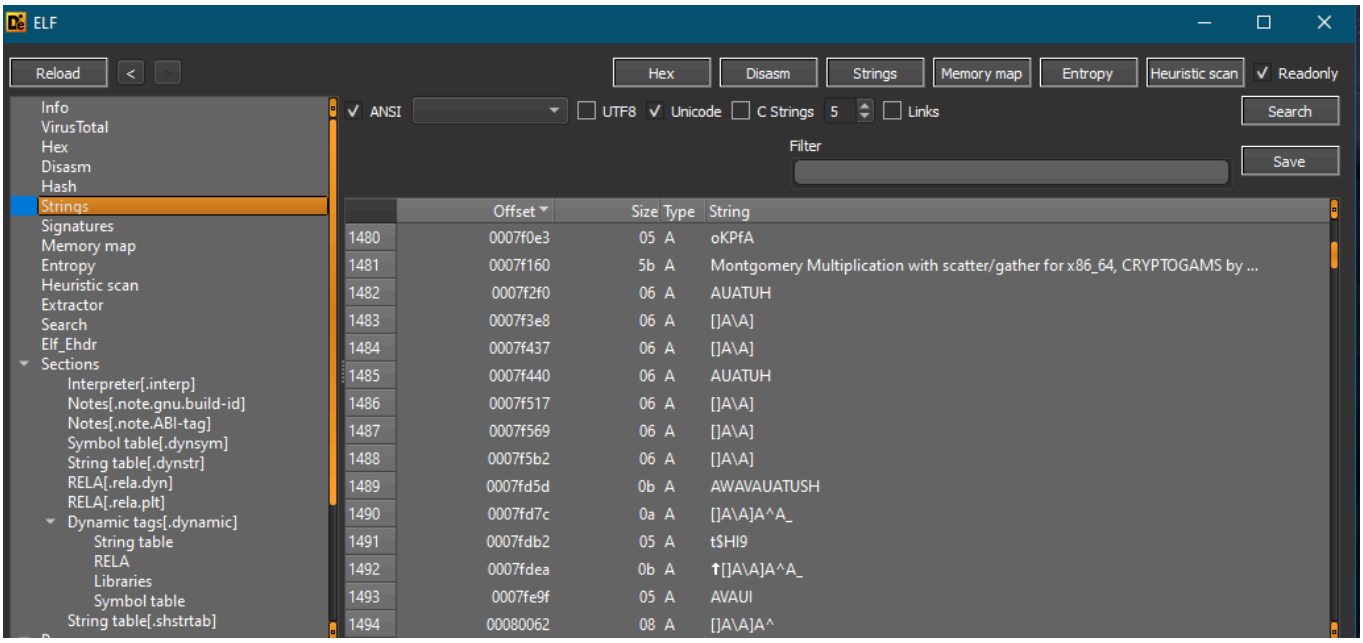
There are 33 of them.
Answer to Q3: **33**

Malware analysis

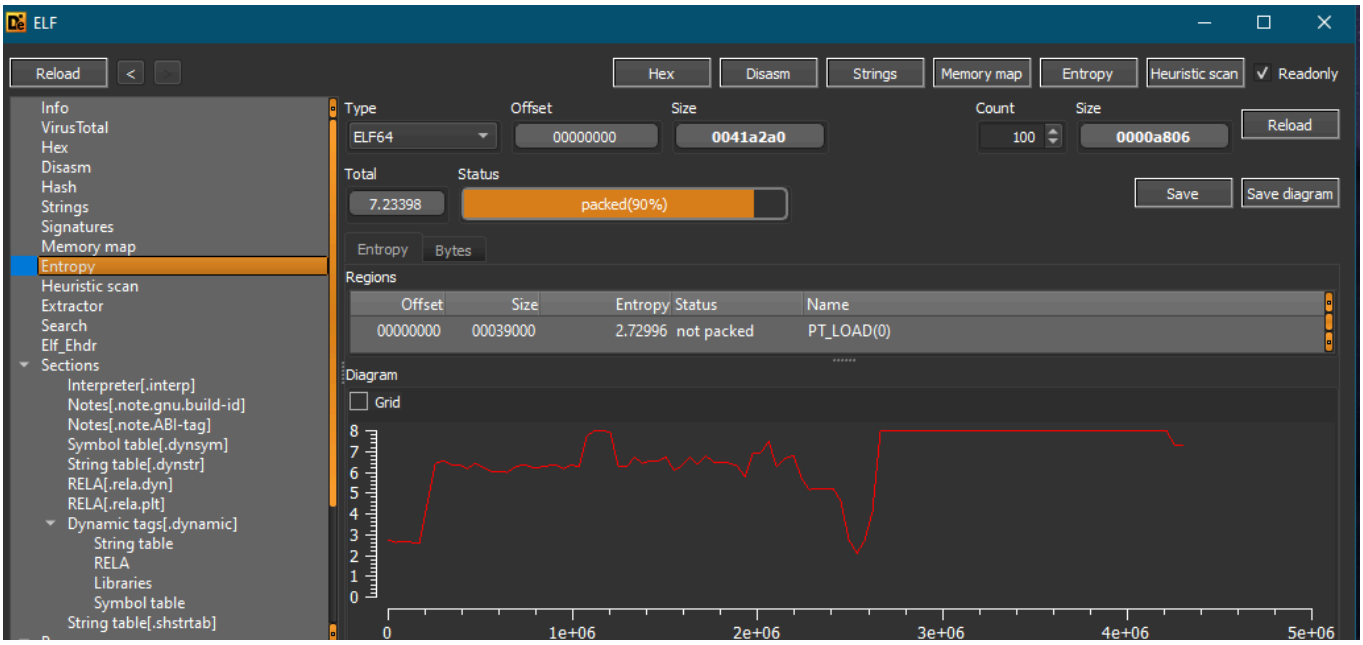
Basic static analysis is the first thing that I should do.
Let's open the binary in **DiE (Detect It Easy)** program:



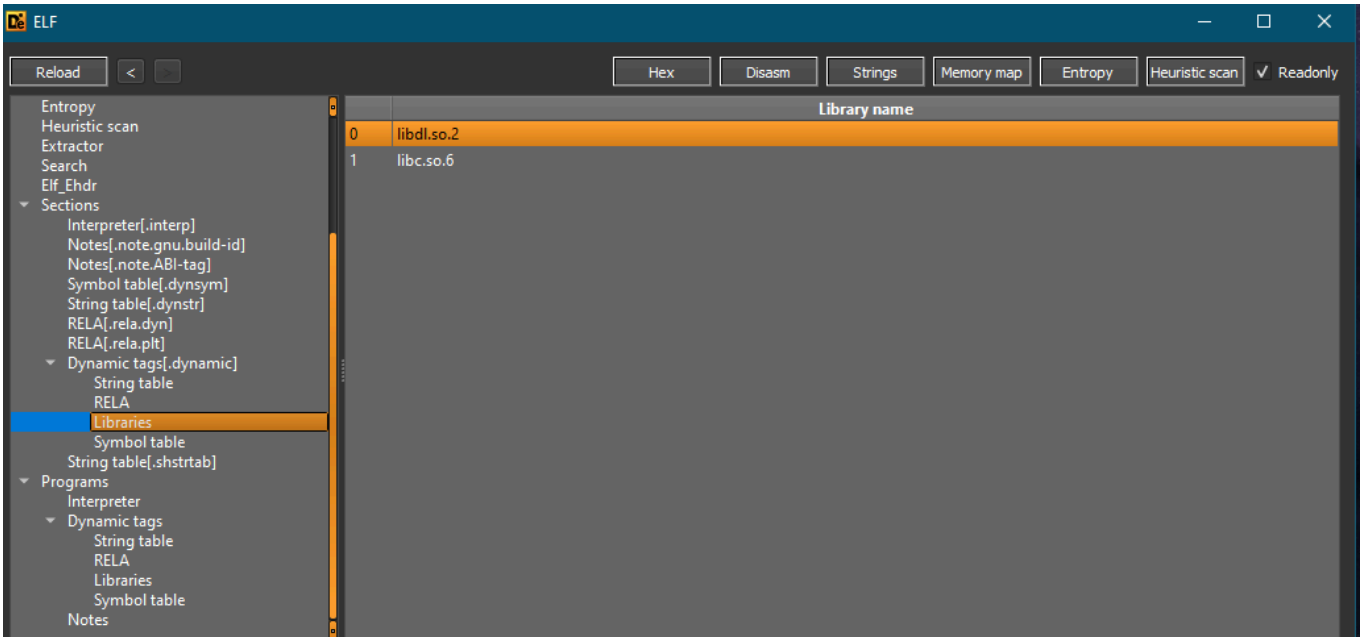
Going further, I need to check if binary is packed or not.
Strings mostly look like gibberish but there are some related to crypto:



Entropy level is very high:

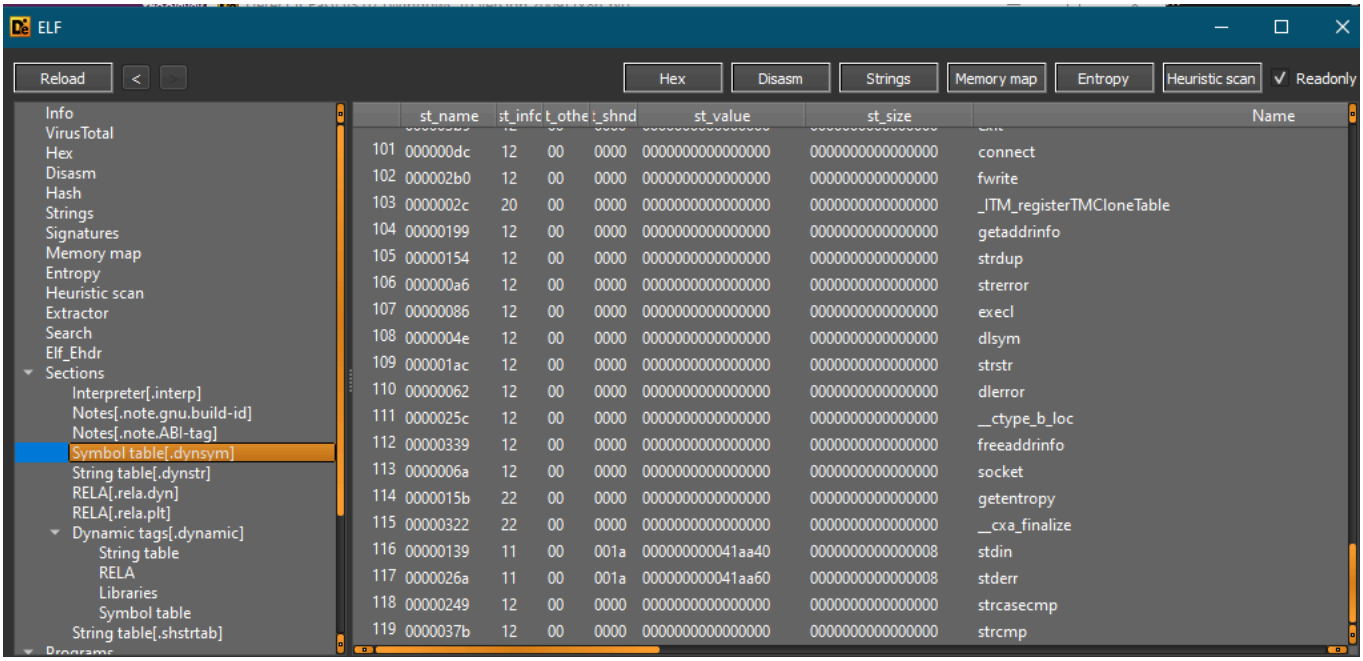


Imported libraries are only the basic ones:

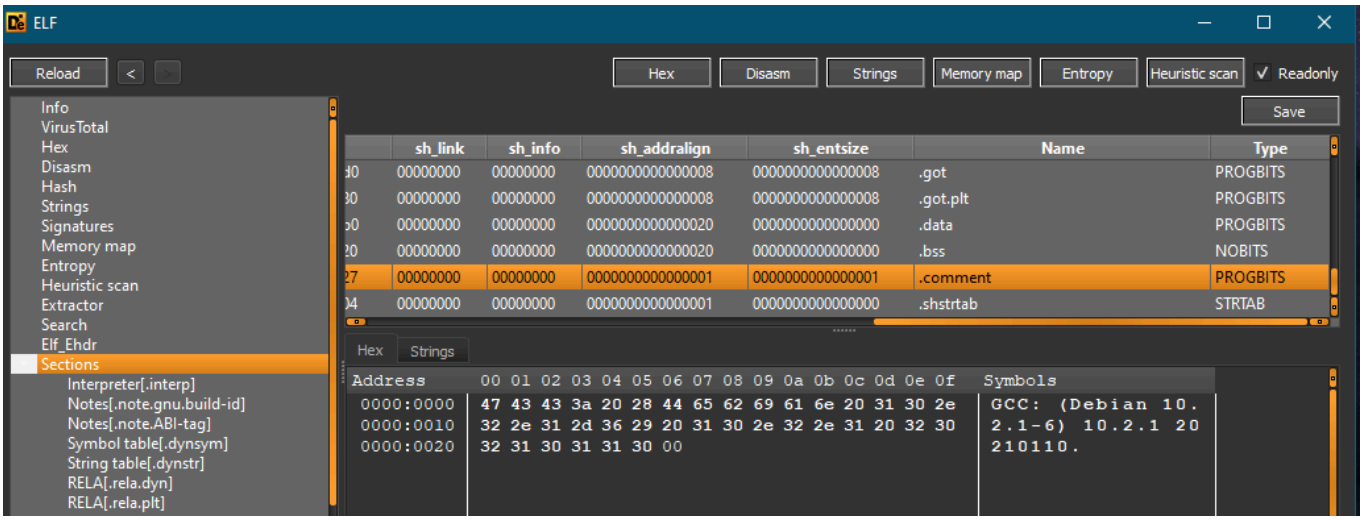


libdl.so.2 - library with functions, that provide dynamic linking facilities. **libc.so.6** - linux C library.

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.

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  size = (size_t)&unk_36D920;
  v41 = 1637173LL;
  ptr = malloc((unsigned int)::size);
  buf = malloc((size_t)&unk_36D920);
  sub_3A29B(byte_2893A0, (char *)ptr, ::size);
  errnum = sub_3A3CB((__int64)ptr, (__int64)buf, size, size);
  if ( errnum < 0 )
    sub_3A4AC(errnum);
  free(ptr);
  fd = memfd_create("test", 1LL);
  if ( fd <= 0 )
  {
    printf("ERROR FD:%i\n", fd);
    exit(-1);
  }
  errnum = write(fd, buf, errnum);
  if ( errnum <= 0 )
  {
    v3 = strerror(errnum);
    fprintf(stderr, "Error Writing: %s\n", v3);
    exit(-1);
  }
  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):

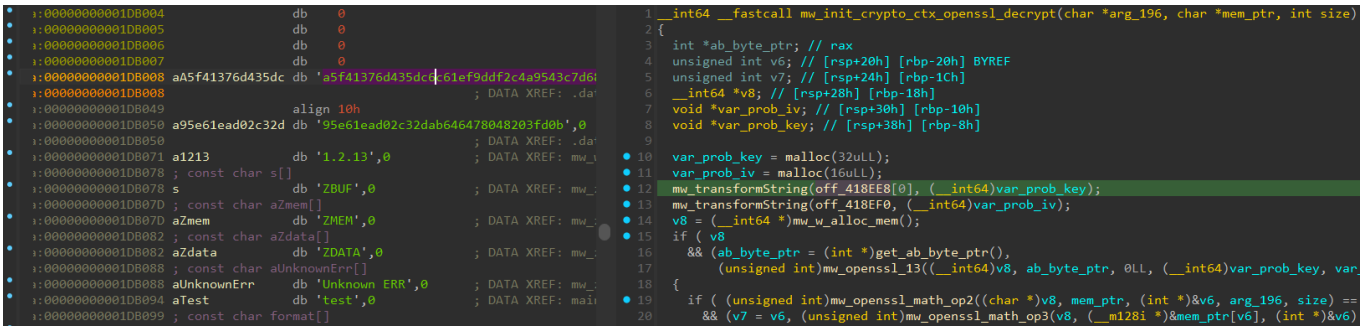
```
__int64 __fastcall sub_402D0(char *a1, char *a2, int *a3, char *a4, int a5)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  v5 = *((_DWORD *)a1 + 4);
  if ( v5 )
  {
    v5 = 0;
    mw_prob_transform_key(6, 166, 148, "crypto/evp/evp_enc.c", 474);
    return v5;
  }
  v6 = a2;
  v37 = a5;
  v36 = *(_DWORD *)(*(_QWORD *)a1 + 4LL);
  v11 = sub_41D50((__int64)a1, 0x2000u);
  v12 = v37;
  v13 = v11;
  v14 = (v37 + 7) >> 3;
  if ( !v13 )
    v14 = v37;
  v15 = v14;
  if ( a5 < 0 )
    goto LABEL_10;
  v16 = *(_QWORD *)a1;
  if ( a5 )
    goto LABEL_6;
  v40 = v37;
  v39 = v14;
  if ( (sub_41880(*(_QWORD *)a1) & 0xF0007) != 7 )
  {
000403AE sub_402D0:9 (403AE) (Synchronized with IDA View-A)
```

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

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

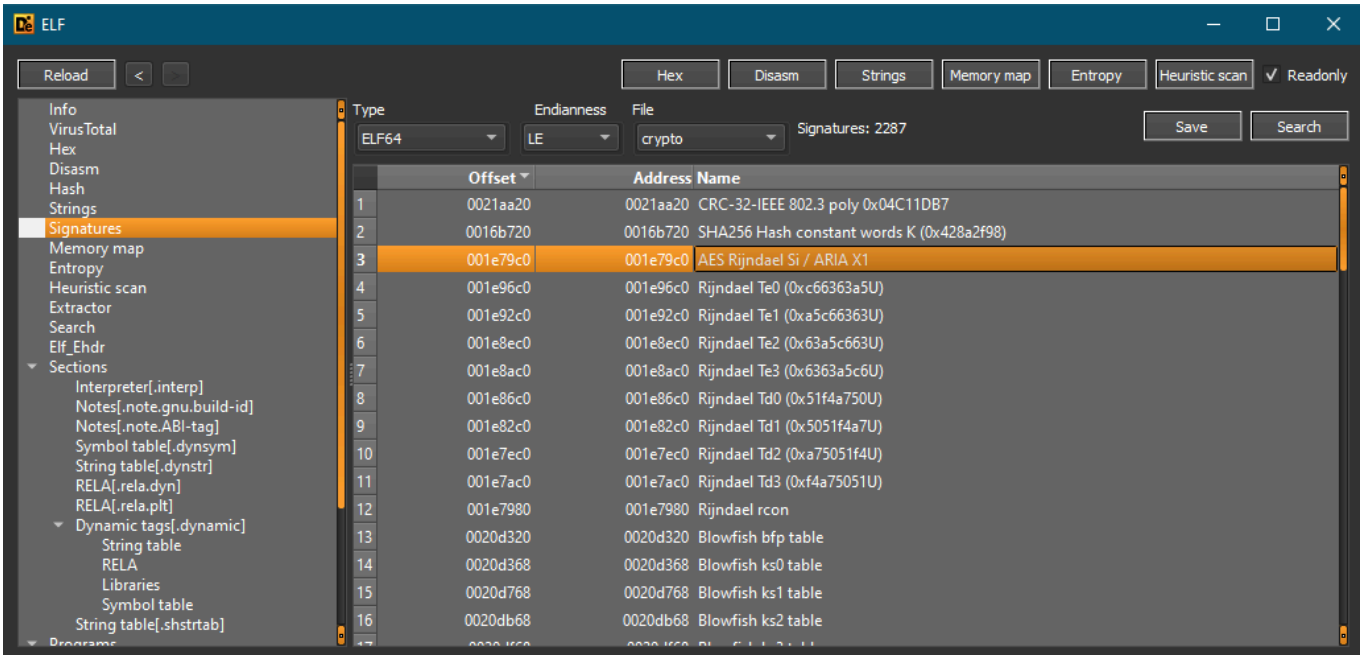
So, I guess the cryptography algorithms used in malware are from the OpenSSL library.

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?



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:

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  size = (size_t)&unk_36D920;                   // 197 bytes
  v41 = 1637173LL;
  ptr = malloc((unsigned int)::size);
  buf = malloc((size_t)&unk_36D920);
  mw_init_crypto_ctx_openssl_decrypt(byte_2893A0, (char *)ptr, ::size);
  errnum = mw_w_decompress_execute((__int64)ptr, (__int64)buf, size, size);
  if ( errnum < 0 )
    mw_zlib_errors(errnum);
```

```
__int64 __fastcall mw_w_decompress_execute(__int64 a1, __int64 a2, __int64 a3, __int64 a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  v11 = 0LL;
  v12 = 0LL;
  v13 = 0LL;
  v7 = a3;
  v6 = a3;
  v5 = a1;
  v10 = a4;
  v9 = a4;
  v8 = a2;
  v14 = mw_math_op6(&v5, 47, "1.2.13", 112);
  if ( !v14 )
  {
    v14 = mw_compression_stuff((__int64)&v5, 4);
    if ( v14 == 1 )
      return v10;
  }
  mw_execute_dynamic_funcs(&v5);
  return v14;
}
```

```
__int64 __fastcall mw_compression_stuff(__int64 a1, int a2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  if ( !a1 || !*(_QWORD *)(a1 + 64) || !*(_QWORD *)(a1 + 72) )
    return (unsigned int)-2;
  v2 = *(_QWORD *)(a1 + 56);
  v3 = -2;
  if ( v2 )
  {
    if ( a1 == *(_QWORD *)v2 )
    {
      v4 = *(_DWORD *)(v2 + 8);
      v5 = v4 - 16180;
      if ( (unsigned int)(v4 - 16180) <= 0x1F )
      {
        dest = *(_BYTE **)(a1 + 24);
        if ( dest )
        {
          v7 = *(unsigned __int8 **)a1;
          v253 = *(_DWORD *)(a1 + 8);
          if ( *(_QWORD *)a1 || !*(_DWORD *)(a1 + 8) )
          {
```

```
                    v7 = v116;
                    v3 = 0;
                    goto LABEL_78;
                  }
                  v11 -= 2;
                  v116 = v7 + 2;
                  v10 += 16;
                  v12 += (unsigned __int64)v7[1] << v117;
                }
LABEL_226:
                *(_DWORD *)(v2 + 24) = v12;
                if ( (_BYTE)v12 != 8 )
                {
LABEL_227:
                  v23 = v12;
                  v22 = a1;
                  v7 = v116;
                  *(_QWORD *)(a1 + 48) = "unknown compression method";
                  *(_DWORD *)(v2 + 8) = 16209;
                  goto LABEL_60;
                }
                if ( (v12 & 0xE000) != 0 )
                {
```

```
int __fastcall mw_zlib_errors(int a1)
{
  if ( a1 == -3 )
    return puts("ZDATA");
  if ( a1 > -3 )
    return puts("Unknown ERR");
  if ( a1 == -5 )
    return puts("ZBUF");
  if ( a1 == -4 )
    return puts("ZMEM");
  else
    return puts("Unknown ERR");
}
```

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:

```
__int64 __fastcall mw_w_decompress_execute(__int64 a1, __int64 a2, __int64 a3, __int64 a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  v11 = 0LL;
  v12 = 0LL;
  v13 = 0LL;
  v7 = a3;
  v6 = a3;
  v5 = a1;
  v10 = a4;
  v9 = a4;
  v8 = a2;
  v14 = mw_math_op6(&v5, 47, "1.2.13", 112);
  if ( !v14 )
  {
    v14 = mw_compression_stuff((__int64)&v5, 4);
    if ( v14 == 1 )
      return v10;
  }
  mw_execute_dynamic_funcs(&v5);
  return v14;
}
```

```
__int64 __fastcall mw_execute_dynamic_funcs(_QWORD *a1)
{
  void (__fastcall *v2)(__int64, __int64); // rax
  __int64 v3; // rsi
  __int64 v5; // rdi

  if ( !a1 )
    return 4294967294LL;
  if ( !a1[8] )
    return 0xFFFFFFFELL;
  v2 = (void (__fastcall *)(__int64, __int64))a1[9];
  if ( !v2 )
    return 0xFFFFFFFELL;
  v3 = a1[7];
  if ( !v3 || a1 != *(_QWORD **)v3 || (unsigned int)(*(_DWORD *)(v3 + 8) - 16180) > 0x1F )
    return 0xFFFFFFFELL;
  v5 = a1[10];
  if ( *(_QWORD *)(v3 + 72) )
  {
    v2(v5, *(_QWORD *)(v3 + 72));
    v2 = (void (__fastcall *)(__int64, __int64))a1[9];
    v3 = a1[7];
    v5 = a1[10];
  }
  v2(v5, v3);
  a1[7] = 0LL;
  return 0LL;
}
```

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

```
  fd = memfd_create("test", 1LL);
  if ( fd <= 0 )
  {
    printf("ERROR FD:%i\n", fd);
    exit(-1);
  }
  errnum = write(fd, buf, errnum);
  if ( errnum <= 0 )
  {
    v3 = strerror(errnum);
    fprintf(stderr, "Error Writing: %s\n", v3);
    exit(-1);
  }
  free(buf);
```

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

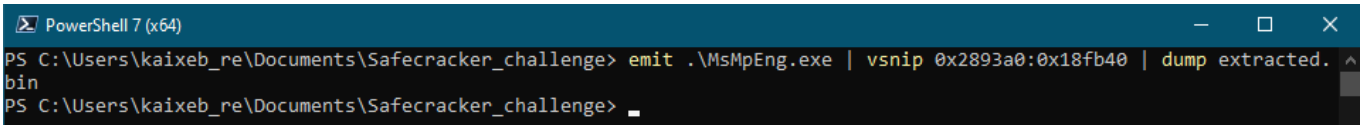
```
  sprintf(s, "/proc/self/fd/%i", fd);
  execl(s, "PROGRAM", 0LL);
  return 0LL;
}
```

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

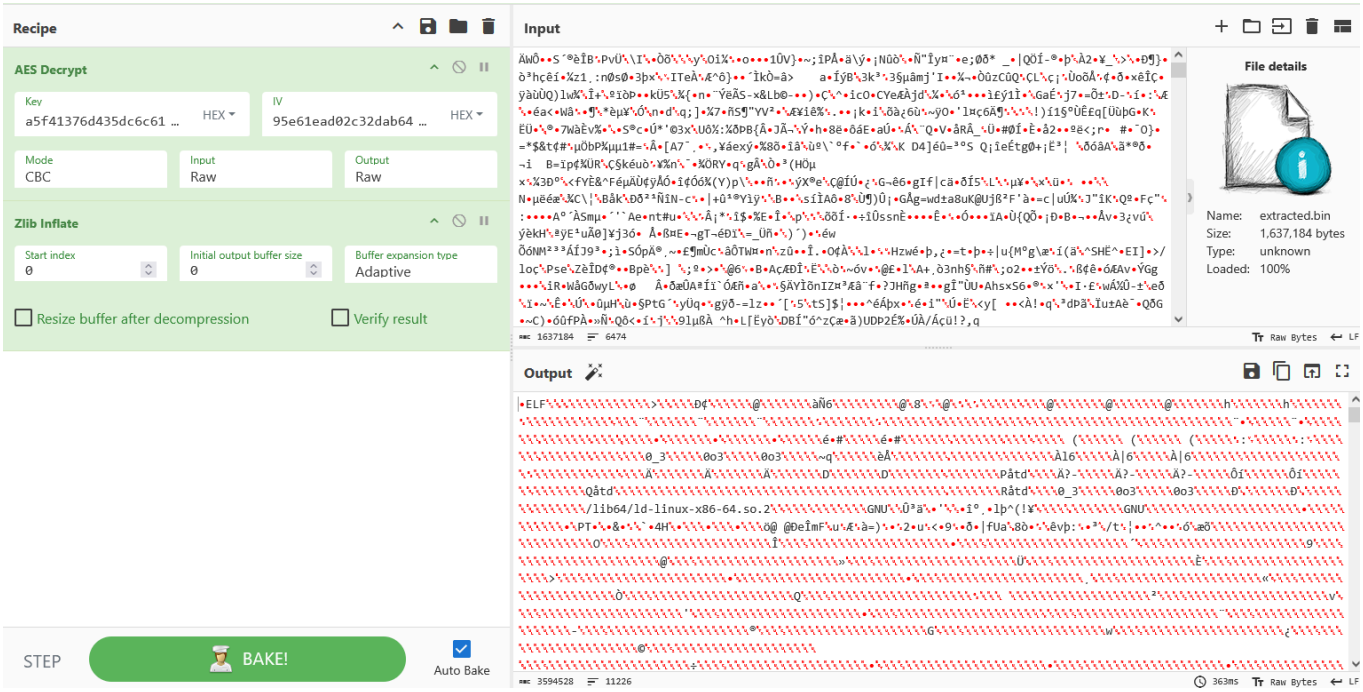
Now, let's get back to the AES and bruteforce the encryption modes.
First, extract the binary blob at address `0x2893a0` (`0x2883a0` physical offset in the file):

Size is 1637173 (0x18fb40).

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

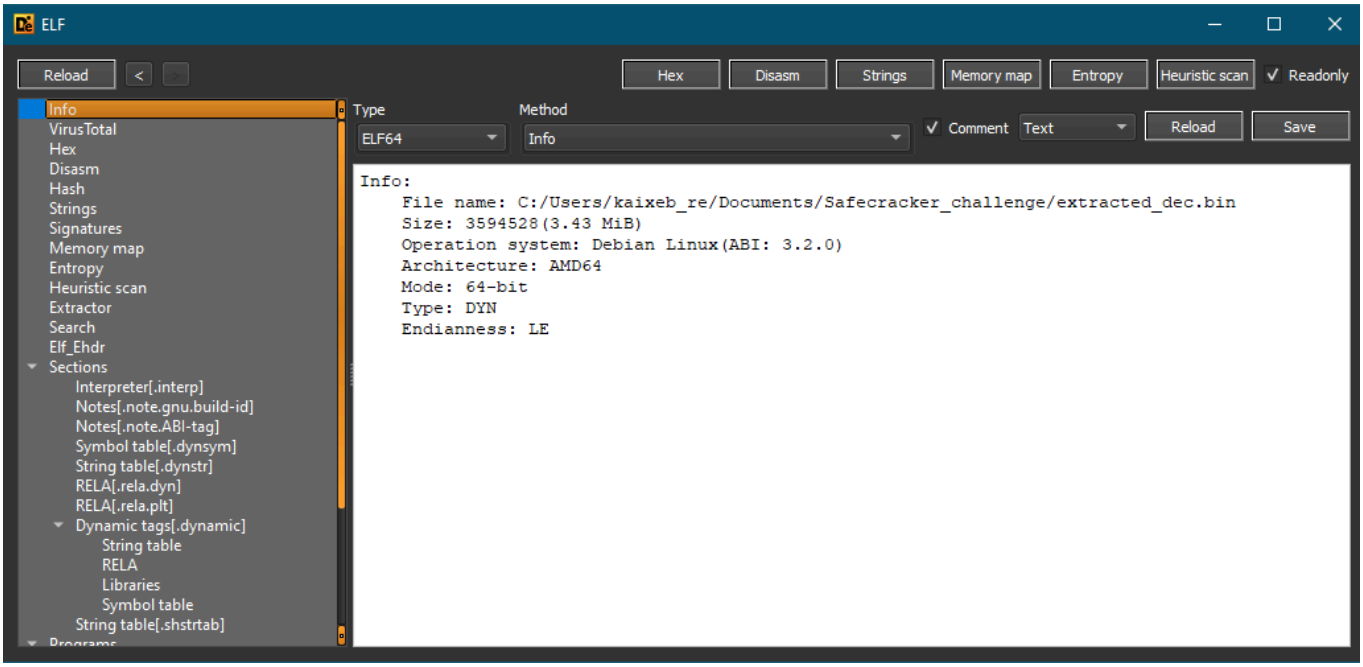


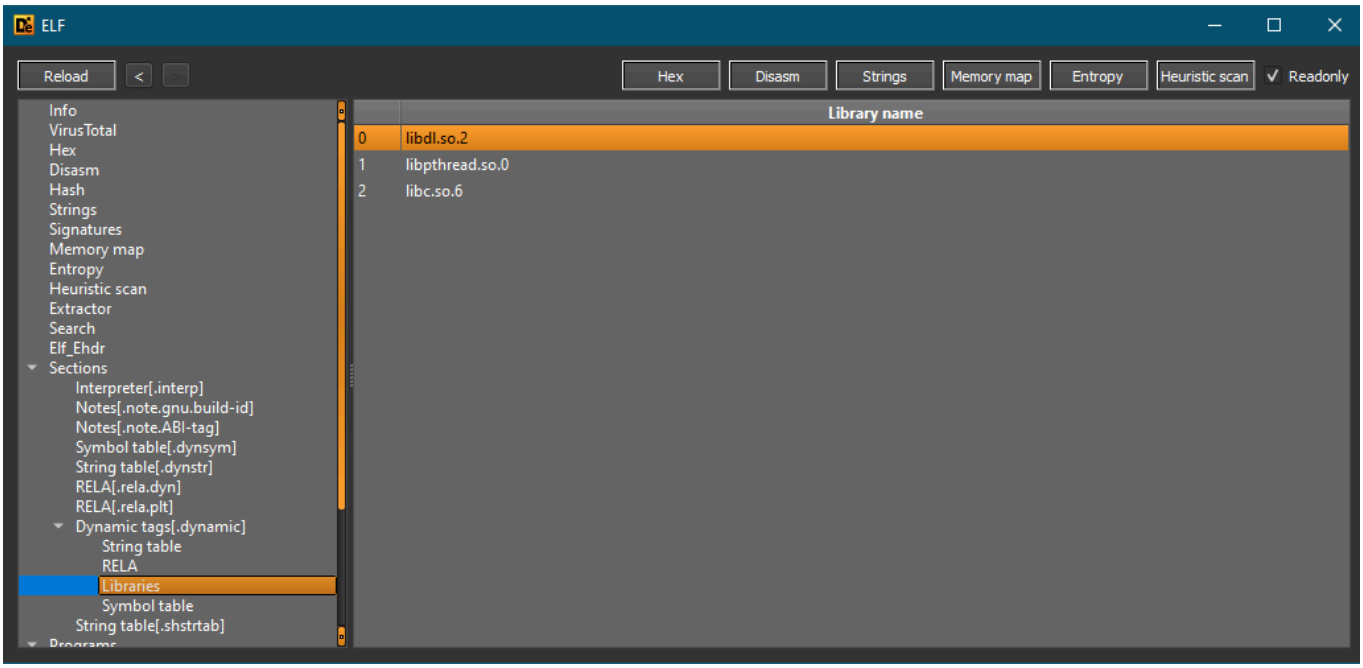
Then decrypt and inflate it in **CyberChef**:

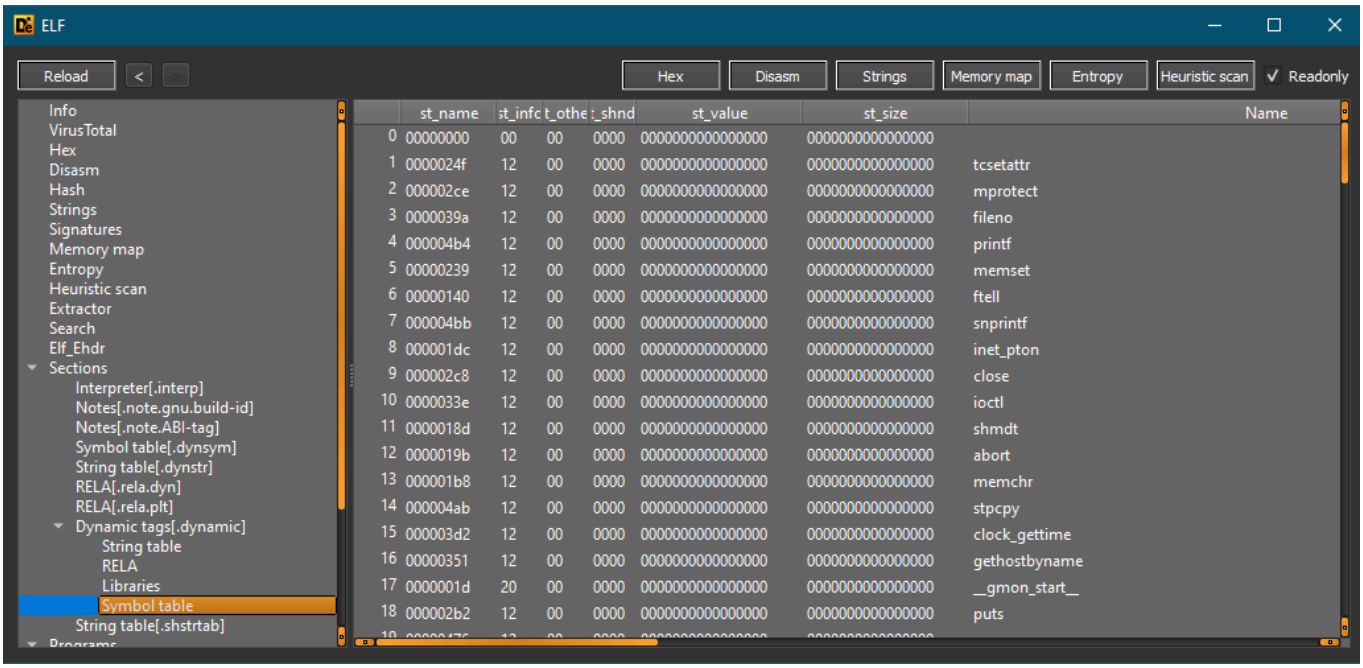


Here we can see the start of the ELF header.

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



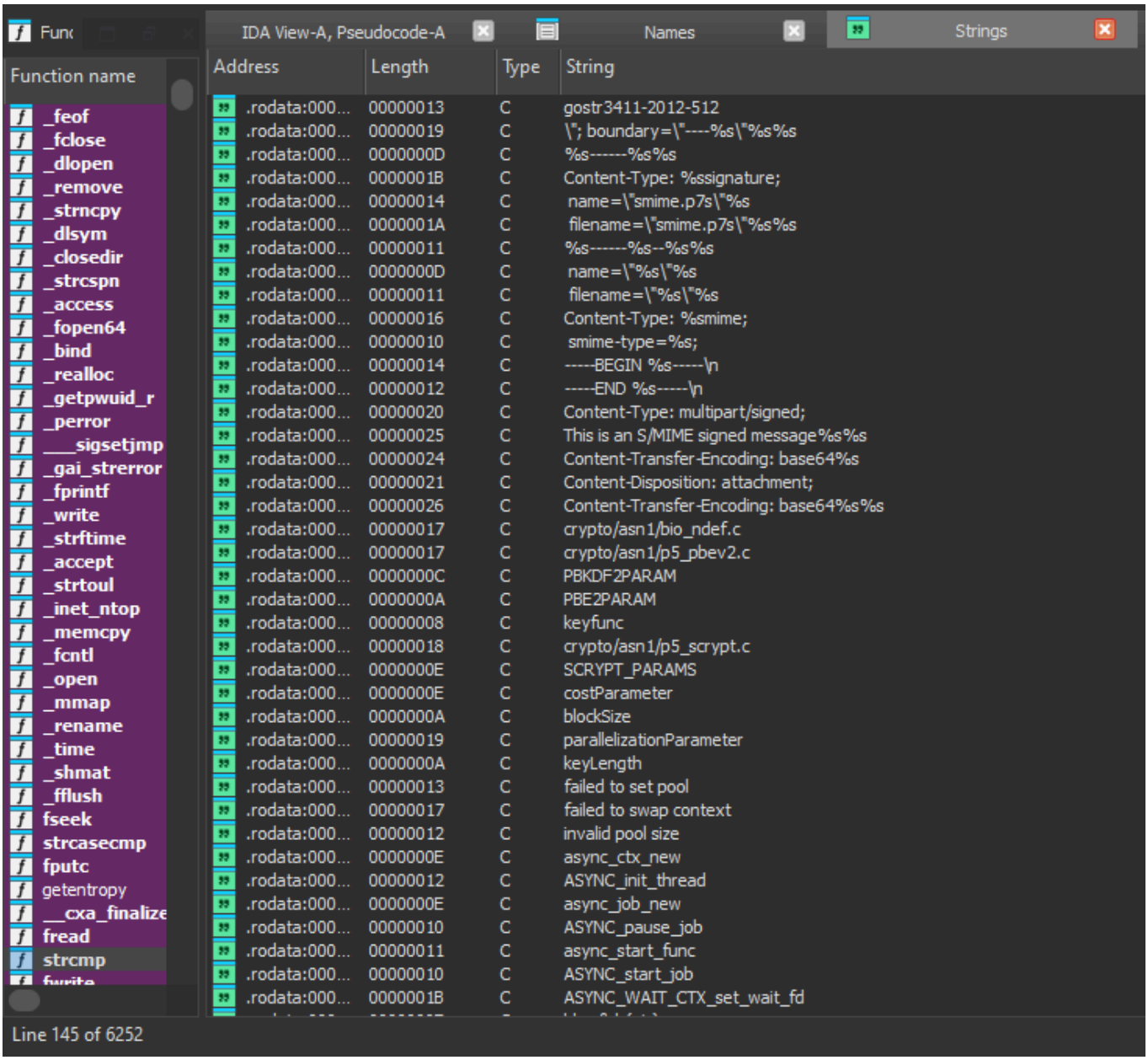




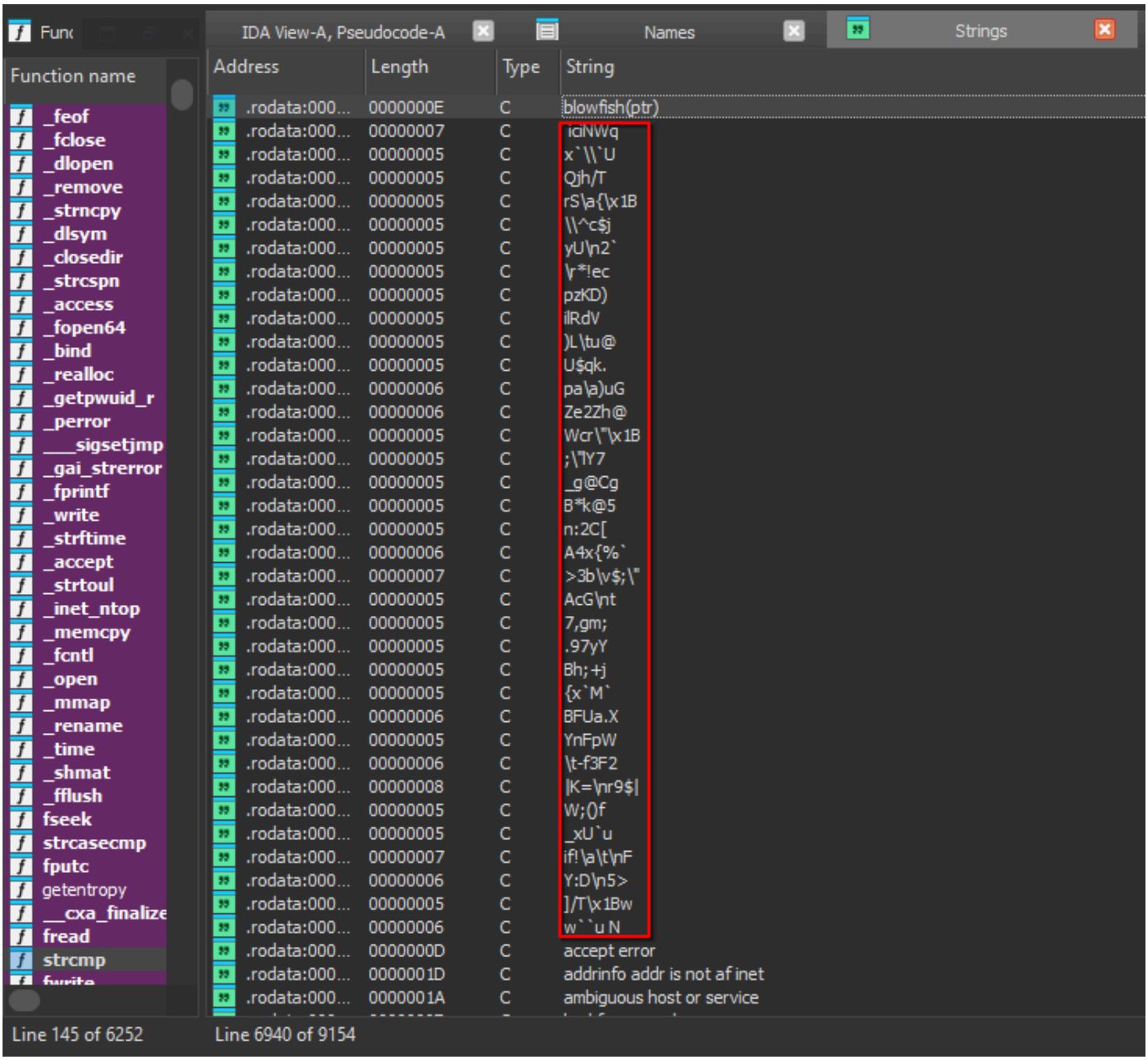
Q6: **What encryption was the packer using?**
Answer: **AES-256-CBC**

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.



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:

```
void __fastcall __noreturn start(__int64 a1, __int64 a2, void (*a3)(void))
{
  __int64 v3; // rax
  int v4; // esi
  __int64 v5; // [rsp-8h] [rbp-8h] BYREF
  char *retaddr; // [rsp+0h] [rbp+0h] BYREF

  v4 = v5;
  v5 = v3;
  _libc_start_main(main, v4, &retaddr, init, fini, a3, &v5);
  __halt();
}
```

```
__int64 __fastcall main(void **a1, char **a2, char **a3, __int64 a4, __int64 a5, __int64 a6)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  v12 = &unk_36DFC0;
  v14 = "daV324982S3bh2";
  v13 = 0LL;
  v15 = 14LL;
  v16 = 0LL;
  v17 = 0LL;
  if ( (unsigned int)mw_check_debug_handle_sigsegv() )
    goto LABEL_2;
  if ( (unsigned int)sub_4AA3D(a1, a2, v6, v7, v8, v9, v12, v13, v14, v15, v16, v17) )
    goto LABEL_2;
  a1 = (void **)(byte_9 + 2);
  raise(11);
  if ( (unsigned int)sub_4A3B5() )
    goto LABEL_2;
  a1 = &v12;
  if ( (unsigned int)sub_4A3F6(&v12) )
    goto LABEL_2;
  raise(11);
  puts("Running update, testing update endpoints");
  a1 = &v12;
  if ( (unsigned int)sub_4AB00((__int64)&v12)
    || (a2 = (char **)&v12, a1 = (void **)"/mnt/c/Users", (unsigned int)sub_4AC39("/mnt/c/Us
    || (raise(11), sub_4A8F6((__int64)&v12, (__int64)&v12, v10), a1 = &v12, (unsigned int)su
  {
LABEL_2:
    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:

```
__int64 mw_check_debug_handle_sigsegv()
{
  __sigset_t *p_sa_mask; // rdi
  __int64 i; // rcx
  struct sigaction act; // [rsp+8h] [rbp-A0h] BYREF

  if ( mw_get_tracer_pid() )
  {
    puts("*******DEBUGGED********");
  }
  else
  {
    p_sa_mask = &act.sa_mask;
    for ( i = 36LL; i; --i )                    // emptying blocked signals set
    {
      LODWORD(p_sa_mask->__val[0]) = 0;
      p_sa_mask = (__sigset_t *)((char *)p_sa_mask + 4);
    }
    act.sa_flags = SA_SIGINFO;                  // send additional info with signal to the handler
    act.sa_handler = (__sighandler_t)mw_nulify_sigsegv_signal;
    if ( sigaction(SIGSEGV, &act, 0uLL) == -1 ) // -1 is returned on error
      _exit(1);
  }
  return 0LL;
}
```

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

```
int mw_get_tracer_pid()
{
  FILE *v0; // rax
  FILE *v1; // rbx
  char *v2; // rdi
  char *v4; // rdi
  char *v5; // [rsp+0h] [rbp-408h] BYREF
  char s[1024]; // [rsp+8h] [rbp-400h] BYREF

  v5 = 0LL;
  v0 = fopen("/proc/self/status", "r");
  if ( v0 )
  {
    v1 = v0;
    while ( fgets(s, 990, v1) )
    {
      v2 = strstr(s, "TracerPid");
      if ( v2 )
      {
        if ( strtok_r(v2, ":", &v5) )
        {
          v4 = strtok_r(0LL, ":", &v5);
          if ( v4 )
            return atoi(v4);
        }
        return -1;
      }
    }
  }
}
```

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:

```
  else
  {
    p_sa_mask = &act.sa_mask;
    for ( i = 36LL; i; --i )                    // emptying blocked signals set
    {
      LODWORD(p_sa_mask->__val[0]) = 0;
      p_sa_mask = (__sigset_t *)((char *)p_sa_mask + 4);
    }
    act.sa_flags = SA_SIGINFO;                  // send additional info with signal to the handler
    act.sa_handler = (__sighandler_t)mw_nulify_sigsegv_signal;
    if ( sigaction(SIGSEGV, &act, 0uLL) == -1 ) // -1 is returned on error
      _exit(1);
  }
  return 0LL;
}
```

```
__int64 mw_nulify_sigsegv_signal()
{
  __int64 result; // rax
  sigaction oact; // [rsp+8h] [rbp-A0h] BYREF

  result = (unsigned int)(sigaction(SIGSEGV, 0LL, &oact) + 1);
  if ( !(_DWORD)result )                        // -1 is returned on sigaction() error,
                                                // so if result is 0, then error happened
                                                // and program exits
    _exit(1);
  return result;
}
```

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

```
int __fastcall mw_raise_sigsegv(__int64 a1)
{
  __int64 i; // rbx
  int result; // eax

  for ( i = *(_QWORD *)(a1 + 8); i; i = *(_QWORD *)(i + 40) )
    result = raise(SIGSEGV);
  return result;
}
```

Q12: **What exception does the binary raise?**
Answer: **SIGSEGV**

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

```
__int64 __fastcall main(void **a1, char **a2, char **a3, __int64 a4, __int64 a5, __int64 a6)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  v12 = &unk_36DFC0;
  v14 = "daV324982S3bh2";
  v13 = 0LL;
  v15 = 14LL;
  v16 = 0LL;
  v17 = 0LL;
  if ( (unsigned int)mw_check_debug_handle_sigsegv() )
    goto LABEL_2;
  if ( (unsigned int)sub_4AA3D(a1, a2, v6, v7, v8, v9, v12, v13, v14, v15, v16, v17) )
    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:

```
unk_36DFC0      db  0Ch             ; DATA XREF: mair
                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
```

```
  v12 = &unk_36DFC0;
  v14 = "daV324982S3bh2";
  v13 = 0LL;
  v15 = 14LL;
  v16 = 0LL;
  v17 = 0LL;
  if ( (unsigned int)mw_check_debug_handle_
    goto LABEL_2;
  if ( (unsigned int)mw_mem_manip(a1, a2, v
    goto LABEL_2;
  a1 = (void **)(byte_9 + 2);
  raise(11);
  if ( (unsigned int)ret_zero() )
    goto LABEL_2;
  a1 = &v12;
  if ( (unsigned int)sub_4A3F6(&v12) )
    goto LABEL_2;
```

And inside is a function that XORes data:

```
__int64 __fastcall xor_decrypt_strings(__int64 a1, __int64 a2, __int64 a3, __int64 a4, unsigned __int64 a5)
{
  unsigned __int64 i; // rcx

  for ( i = 0LL; i != a4; ++i )
    *(_BYTE *)(a3 + i) = *(_BYTE *)(a1 + i) ^ *(_BYTE *)(a2 + i % a5);
  *(_BYTE *)(a3 + i) = 0;
  return 0LL;
}
```

Marked up version:

```
__int64 __fastcall mw_xor_decrypt_data(
        __int64 enc_data_ptr,
        __int64 var_22,
        __int64 dec_data_ptr,
        __int64 size,
        unsigned __int64 var_41)
{
  unsigned __int64 i; // rcx

  for ( i = 0LL; i != size; ++i )
    *(_BYTE *)(dec_data_ptr + i) = *(_BYTE *)(enc_data_ptr + i) ^ *(_BYTE *)(var_22 + i % var_41);
  *(_BYTE *)(dec_data_ptr + i) = 0;
  return 0LL;
}
```

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
```

```
__int64 __fastcall mw_string_decryption(char *haystack)
{
  const char *v1; // r15
  int v2; // ebx
  int v3; // r13d
  size_t v4; // rax
  unsigned __int64 v5; // rcx

  v1 = a0;
  v2 = 0;
  v3 = dword_368484;
  while ( 1 )
  {
    if ( v3 <= v2 )
      return 0LL;
    v4 = strlen(v1);
    v5 = 0LL;
    while ( v4 != v5 )
    {
      needle[v5] = v1[v5] ^ aDav324982s3bh2[v5 % 0xE];
      if ( v4 < ++v5 )
        goto LABEL_7;
    }
    needle[v4] = 0;
LABEL_7:
    v1 += 8;
    if ( strstr(haystack, needle) )
      return 1LL;
    ++v2;
  }
```

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

Marked up version:

```
__int64 __fastcall mw_string_decryption(char *string_array)
{
  const char *encrypted_strings; // r15
  int current_strings_num; // ebx
  int total_strings_num; // r13d
  size_t current_enc_string_len; // rax
  unsigned __int64 curr_enc_string_byte_index; // rcx

  encrypted_strings = byte_3684A0;
  current_strings_num = 0;
  total_strings_num = dword_368484;
  while ( 1 )
  {
    if ( total_strings_num <= current_strings_num )
      return 0LL;
    current_enc_string_len = strlen(encrypted_strings);
    curr_enc_string_byte_index = 0LL;
    while ( current_enc_string_len != curr_enc_string_byte_index )// iterate through each byte of the encry
    {
      decrypted_strings_arr[curr_enc_string_byte_index] = encrypted_strings[curr_enc_string_byte_index] ^ 
      if ( current_enc_string_len < ++curr_enc_string_byte_index )
        goto LABEL_7;
    }
    decrypted_strings_arr[current_enc_string_len] = 0;
LABEL_7:
    encrypted_strings += 8;
    if ( strstr(string_array, decrypted_strings_arr) )
      return 1LL;
    ++current_strings_num;
  }
```

Q5: **What is the XOR key used for the encrypted strings?**
Answer: **daV324982S3bh2**

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

```
__int64 __fastcall main(void **a1, char **a2, char **a3, __int64 a4, __int64 a5, __int64 a6)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  encrypted_data = &unk_36DFC0;
  key = "daV324982S3bh2";
  v13 = 0LL;
  v15 = 14LL;
  v16 = 0LL;
  v17 = 0LL;
  if ( (unsigned int)mw_check_debug_handle_sigsegv()
    || (unsigned int)mw_mem_manip(a1, a2, v6, v7, v8, v9, encrypted_data, v13, key, v15, v16, v17)
    || (raise(SIGSEGV), (unsigned int)ret_zero())
    || (unsigned int)sub_4A3F6(&encrypted_data)
    || (raise(SIGSEGV),
        puts("Running update, testing update endpoints"),
        (unsigned int)mw_uri_check((__int64)&encrypted_data))// here strings are unencrypted
    || (unsigned int)mw_compare_filenames_to_decrypted_strings("/mnt/c/Users", (__int64)&encrypted_data)
    || (raise(SIGSEGV),
        sub_4A8F6((__int64)&encrypted_data, (__int64)&encrypted_data, v10),
        (unsigned int)sub_4A5C1((__int64)&encrypted_data)) )
  {
    sub_28164A();
  }
  raise(11);
  puts("----------------------------------------");
  puts("Configuration Successful\nYou can now connect to the Corporate VPN");
  return 0LL;
}
```

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.

```
DIR *v2; // rbp
struct dirent *v3; // rbx
unsigned __int8 d_type; // al
char filenames_array[4152]; // [rsp+0h] [rbp-1038h] BYREF

v2 = opendir(a1);
if ( !v2 )
  return 1LL;
while ( 1 )
{
  v3 = readdir(v2);
  if ( !v3 )
    break;
  raise(SIGSEGV);
  snprintf(filenames_array, 4096uLL, "%s/%s", a1, v3->d_name);
  d_type = v3->d_type;
  if ( d_type == DT_DIR )
  {
    if ( !mw_check_specifict_dir(v3->d_name) )
      mw_compare_filenames_to_decrypted_strings(filenames_array, a2);
  }
  else if ( d_type == DT_REG )
  {
    if ( (unsigned int)mw_string_decrypt_search(filenames_array) )
      mw_collect_decrypted_strings(a2, filenames_array);
  }
}
closedir(v2);
return 0LL;
}
```

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":

```
.rodata:000...  00000020  C  OpenSSL 1.1.1u-dev  xx XXX xxxx
.rodata:000...  00000044  C  OPENSSLDIR: \"/home/blitztide/Projects/Payloads/LockPick3.0/lib/ssl\"
.rodata:000...  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**