Malware Analysis Series (MAS): Article 10 | Linux

by Alexandre Borges
release date: JAN/15/2025 | rev: A.1

0. Quote

"No thinking - that comes later. You must write your first draft with your heart. You rewrite with your head. The first key to writing is... to write, not to think"
(William Forrester played by Sean Connery | "Finding Forrester" movie - 2000)

1. Introduction

Welcome to the **tenth (and last) article** of *the **Malware Analysis Series***, where we are reviewing concepts, techniques and practical steps used for analyzing **ELF malware binaries**. If readers have not read my previous articles yet, all of them are available on the following links:

- **ERS_02:** https://exploitreversing.com/2024/01/03/exploiting-reversing-er-series-article-02/
- **ERS_01:** https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/
- **MAS_9:** https://exploitreversing.com/2025/01/08/malware-analysis-series-mas-article-09/
- **MAS_8:** https://exploitreversing.com/2024/08/07/malware-analysis-series-mas-article-08/
- **MAS_7:** https://exploitreversing.com/2023/01/05/malware-analysis-series-mas-article-7/
- **MAS_6:** https://exploitreversing.com/2022/11/24/malware-analysis-series-mas-article-6/
- **MAS_5:** https://exploitreversing.com/2022/09/14/malware-analysis-series-mas-article-5/
- **MAS_4:** https://exploitreversing.com/2022/05/12/malware-analysis-series-mas-article-4/
- **MAS_3:** https://exploitreversing.com/2022/05/05/malware-analysis-series-mas-article-3/
- **MAS_2:** https://exploitreversing.com/2022/02/03/malware-analysis-series-mas-article-2/
- **MAS_1**: https://exploitreversing.com/2021/12/03/malware-analysis-series-mas-article-1/

This article offers an introductory analysis of ELF binaries on Linux, and we will be moving slowly and carefully to not getting in unnecessary details that, at first approach, do not contribute to building the knowledge. The main goal of this article is to keep it short, simple, and informative, avoiding touching on tons of details associated with ELF format, and only examining the most important aspects of the malware.

2. Acknowledgments

The year is 2025. Four years ago, I started drafting articles with the sole purpose of helping the hacking and information security community, and as I could already imagine at that time, it would be challenging to find time to continue producing content, and indeed this side effect has been confirmed over time.

As I have been using IDA Pro for a long time, I needed a license of my own, and that was when **Ilfak Guilfanov (@ilfak)** and **Hex-Rays SA (@HexRaysSA)** decided to help me, and since then they have provided continuous and decisive support to write this **Malware Analysis Series (MAS)**, which is focused on malware analysis, and the **Exploiting Reversing series (ERs)**, which is my **current and long-term series** on internals, vulnerability research and, eventually, exploitation in critical topics such as Windows, kernel drivers, macOS, browser and hypervisors.

Time flies, and companies around the world have become more demanding in terms of technical skills, but I still believe that one of the most effective ways to help these professionals is to write articles because such content can offer a solid method to learn details that would be a bit more difficult in live conferences or even other media. I still face serious time constraints to write, but I keep trying to do it because, in some way, I know that these series have been important for people's careers.

Life may be short, but every moment is worth it. Enjoy the journey and keep exploiting it!

3. Lab Setup

This article and next ones I will be using the following lab configuration:

- **Virtual machine running Ubuntu 24.04.x LTS:** https://ubuntu.com/download/desktop
- **IDA Pro or IDA Home version (@HexRaysSA):** https://hex-rays.com/ida-pro/
- **Malwoverview:** https://github.com/alexandreborges/malwoverview

4. Retrieving malware samples

We need to establish a starting point to initiate our article analyzing malware threats for Linux and, probably, one of the possible approaches is to search for samples and retrieve one of them to start our work. As readers already know, Linux binaries are represented in **ELF format**, so we can use Malwoverview tool to search such samples. One of the recommended sources to list and download samples is Malware Bazaar, which readers could do by executing the following commands (I specified "-o 0" because I have used a clear background for the terminal, but if you are using a dark background, so you should omit this option):

- **malwoverview -b 2 -B elf -o 0** **(list last fifty samples reported as ELF format)**
- **malwoverview -b 2 -B linux -o 0** **(list last fifty samples reported as Linux)**
- **malwoverview -b 5 -B <sha256 hash> -o 0** **(download sample represented by the given hash)**

Once you have downloaded the sample, you can retrieve reports from different services such as Virus Total, Triage, InQuest, URL Haus, Alien Vault, Polyswarm, Hybrid Analysis and :

- **malwoverview -v 2 -V <sha256 hash> -o 0** **(Virus Total)**

- **malwoverview -a 5 -A <sha256 hash> -o 0 (Hybrid-Analysis)**
- **malwoverview -j 2 -J <sha256 hash> -o 0 (URL Haus)**
- **malwoverview -p 1 -P <sha256 hash> -o 0 (Polyswarm)**
- **malwoverview -n 4 -N <sha256 hash> -o 0 (Alien Vault)**
- **malwoverview -i 2 -I <sha256 hash> -o 0 (InQuest)**
- **malwoverview -b 1 -B <sha256 hash> -o 0 (Malware Bazaar)**
- **malwoverview -x 1 -X <sha256 hash> -o 0 (Triage)**
- **malwoverview -x 2 -X <triage_id> -o 0 (Triage)**

Eventually, some samples have been written in Golang, and even though it is more interesting to examine binaries written in C/C++, readers can do a quick triage using Yara with a basic (and flawed) rule:

```
root@ubuntu01:~/malware/linux# cat yara_golang.yar

rule golang
{
  meta:
    desc = "Golang Rule"
    author = "Alexandre Borges"
    version = "1.0"
    last_modified = "2025-01-08"

    strings:
      $a1 = "golang" wide ascii
      $re1 = /(go-1\.[1-9]{2})/ ascii wide
      $re2 = /("?([a-zA-Z0-9_-]{20})\/([a-zA-Z0-9_-]{20})\/([a-zA-Z0-9_-]{20}\/([a-zA-Z0-9_-]){20}("?))/  ascii wide

    condition:
      (1 of them)
}
root@ubuntu01:~/malware/linux#
root@ubuntu01:~/malware/linux# yara -wr yara_golang.yar linux/unpacked/
golang linux/unpacked//713b699c04f21000fca981e698e1046d4595f423bd5741d712fd7e0bc358c771.elf
golang linux/unpacked//24b5cdfc8de10c99929b230f0dcbf7fcefe9de448eeb6c75675cfe6c44633073.elf
golang linux/unpacked//0c395715bfeb8f89959be721cd2f614d2edb260614d5a21e90cc4c142f5d83ad.elf
golang linux/unpacked//b50bdfa4dc778404fda39499f2627c4c510fb7c650daee5147e851090b3ab820.elf
golang linux/unpacked//6a0449a0b92dc1b17da219492487de824e86a25284f21e6e3af056fe3f4c4ec0.elf
golang linux/unpacked//bddc5a2605d4d8adff58e52f83000f536a64b84815f088e5ecce842a0ac8493c.elf
golang linux/unpacked//e29aa629bf492a087a17fa7ec0edb6be4b84c5c8b0798857939d8824fa91dbf9.elf
golang linux/unpacked//3d8d25e2204f25260c42a29ad2f6c5c21f18f90ce80cb338bc678e242fba68cd.elf
golang linux/unpacked//10d25dd902a46d9c50908390227d971ca2b9ddb782b88c60daed051e2f16c942.elf
```

[Figure 1]: Basic Yara rule

It is important to highlight that I am **not** interested in being precise here, and certainly this rule is far from being good (the condition clause is horrible, by the way). The idea is that we want to be able to distinguish between pure C/C++ and Golang binaries when it is necessary.

5. Gathering basic information

I will be using different tools to collect properties of our malicious binary, and I am going to focus on static analysis. As usual, quite a few concepts and fundamentals will be provided to support and improve the understanding of the topics exposed. I am going to use a simple example, whose hash is the following one:

- **SHA256: f864922f947a6bb7d894245b53795b54b9378c0f7633c521240488e86f60c2c5**

```
root@ubuntu01:~/malware/mas/mas_10# malwoverview -x 1 -X f864922f947a6bb7d894245b53795b54b9378c0f7633
c521240488e86f60c2c5 -o 0 | grep id: | nl | head -7
     1  id:        230524-q61ecacg54
     2  id:        230515-rmm6cafg49
     3  id:        230515-rma6saeb9s
     4  id:        230430-rqk56ahh86
     5  id:        211026-p8pvyshea6
root@ubuntu01:~/malware/mas/mas_10# tput init
root@ubuntu01:~/malware/mas/mas_10# malwoverview -x 2 -X 230524-q61ecacg54 -o 0

                                    TRIAGE SEARCH REPORT

-------------------------------------------------------------------------------------------------

score:     10

id:        230524-q61ecacg54
target:    Hacker.zip
size:      33410
md5:       042795e45254b01cd90c9b8cbb52d6e3
sha1:      f6ad4c0bc72be2f31c085cdeda3ac65b3876d4e7
sha256:    b390abe40de4fc488c98730339f483401af8a64a567019dd4875bf7935dd5b14
completed: 2023-05-24T13:56:41Z
signatures:
           Sodinokibi family
           Sodinokibi/Revil Elf

targets:
           md5:      c83df66c46bcbc05cd987661882ff061
           score:    1
           sha1:     48d1558fe3ac689b7eaac82738a023c13f4c0e7c
           sha256:   f864922f947a6bb7d894245b53795b54b9378c0f7633c521240488e86f60c2c5
           size:     109336bytes
           target:   Ransomware.elf
           tasks:    behavioral1

root@ubuntu01:~/malware/mas/mas_10# malwoverview -x 7 -X 230524-q61ecacg54 -o 0

                                    TRIAGE DYNAMIC REPORT

-------------------------------------------------------------------------------

id:        230524-q61ecacg54
target:    Hacker.zip
score:     1
submitted: 2023-05-24T13:53:08Z
size:      33410
md5:       042795e45254b01cd90c9b8cbb52d6e3
sha1:      f6ad4c0bc72be2f31c085cdeda3ac65b3876d4e7
sha256:    b390abe40de4fc488c98730339f483401af8a64a567019dd4875bf7935dd5b14
static_tags:

analysis:
           score:    1
           reported: 2023-05-24T13:56:41Z
           platform: ubuntu-18.04_amd64
           resource: ubuntu1804-amd64-20221125-en
           time_krn: 3282
           tags:
           ttps:
           features:
                     analog
                     overview

processes:
           procid:   1
           pid:      599
           ppid:     588
           cmd:      ['/tmp/Ransomware.elf']
           image:    /tmp/Ransomware.elf
```

[Figure 2]: Triage report

Download and unzip it: **malwoverview.py -x 5 -X 230524-q61ecacg54 -o 0 ; 7z e 230524-q61ecacg54.bin -pinfected**

```
root@ubuntu01:~/malware/mas/mas_10# malwoverview -v 2 -V f864922f947a6bb7d894245b53795b54b9378c0f7
633c521240488e86f60c2c5.bin -o 0

MD5 hash:            c83df66c46bcbc05cd987661882ff061
SHA1 hash:           48d1558fe3ac689b7eaac82738a023c13f4c0e7c
SHA256 hash:         f864922f947a6bb7d894245b53795b54b9378c0f7633c521240488e86f60c2c5

Malicious:           41
Undetected:          23

AV Report:
                     Avast:          ELF:Filecoder-BN [Trj]
                     Avira:          LINUX/Filecoder.hefhz
                     BitDefender:    Trojan.Linux.Ransom.221674
                     DrWeb:          Linux.Encoder.105
                     Emsisoft:       Trojan.Linux.Ransom.221674 (B)
                     ESET-NOD32:     a variant of Linux/Filecoder.Sodinokibi.A
                     F-Secure:       Malware.LINUX/Filecoder.hefhz
                     FireEye:        Trojan.Linux.Ransom.221674
                     Fortinet:       Linux/Filecoder_Sodinokibi.A!tr
                     Kaspersky:      HEUR:Trojan-Ransom.Linux.Sodin.a
                     McAfee:         GenericRXVY-XO!C83DF66C46BC
                     Microsoft:      Ransom:Linux/MoneyMessage.K!MTB
                     Panda:          CLEAN
                     Sophos:         CLEAN
                     Symantec:       Trojan Horse
                     TrendMicro:     TROJ_FRS.0NA103F122
```

[Figure 3]: Virus Total report

Apparently, the sample is **Sodinokibi ransomware**. We can run a brief sequence of commands to collect eventual evidence about the binary:

```
root@ubuntu01:~/malware/mas/mas_10# file mas_10.bin
mas_10.bin: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interp
reter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.24, BuildID[sha1]=98bab2fe2b19684dbb80
9db7287a70dfd54381ea, stripped
root@ubuntu01:~/malware/mas/mas_10#
root@ubuntu01:~/malware/mas/mas_10# readelf mas_10.bin -h
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401650
  Start of program headers:          64 (bytes into file)
  Start of section headers:          107544 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         28
  Section header string table index: 27
root@ubuntu01:~/malware/mas/mas 10#
```

[Figure 4]: Getting binary's information (1)

We can extract some information from these first two commands:

- The binary is **64-bit** and has **amd64 architecture**.
- It is **dynamically linked** then it depends on shared libraries.
- There is a unique identification (**BuildID**), which is normally useful when analyzing core files, for example.
- The binary is **stripped**, so there is not any symbol information inside.
- The **binary's entry point is 0x401650**, and usually has the following sequence of functions invoked: **_start_function → __libc_start_main → main.**
- The **program headers**, which is a **table that provides loaders with information to load a binary into memory and, as expected, it is more focused on memory mapping**, has **9 entries**.
- The **section headers**, which **provide us with information about the binary itself and it is naturally focused on binary sections**, are counted in **28 entries**.

For now, we have not found anything really weird or that represents an issue, and the information shown so far was already expected.

Thus, we can list the **program header table** by running the following command:

```
root@ubuntu01:~/malware/mas/mas_10# readelf mas_10.bin -lW

Elf file type is EXEC (Executable file)
Entry point 0x401650
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  PHDR           0x000040 0x0000000000400040 0x0000000000400040 0x0001f8 0x0001f8 R E 0x8
  INTERP         0x000238 0x0000000000400238 0x0000000000400238 0x00001c 0x00001c R   0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000400000 0x0000000000400000 0x014fc4 0x014fc4 R E 0x200000
  LOAD           0x015df0 0x0000000000615df0 0x0000000000615df0 0x004504 0x105858 RW  0x200000
  DYNAMIC        0x015e08 0x0000000000615e08 0x0000000000615e08 0x0001f0 0x0001f0 RW  0x8
  NOTE           0x000254 0x0000000000400254 0x0000000000400254 0x000044 0x000044 R   0x4
  GNU_EH_FRAME   0x013ae8 0x0000000000413ae8 0x0000000000413ae8 0x00041c 0x00041c R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
  GNU_RELRO      0x015df0 0x0000000000615df0 0x0000000000615df0 0x000210 0x000210 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.
version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
root@ubuntu01:~/malware/mas/mas_10# _
```

[Figure 5]: Getting binary information (2)

It is also recommended to **retrieve the dynamic segment** by running the following command:

```
root@ubuntu01:~/malware/mas/mas_10# readelf mas_10.bin -d

Dynamic section at offset 0x15e08 contains 26 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libm.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libpthread.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x401268
 0x000000000000000d (FINI)               0x4102b0
 0x0000000000000019 (INIT_ARRAY)         0x615df0
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x615df8
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x400298
 0x0000000000000005 (STRTAB)             0x4008d8
 0x0000000000000006 (SYMTAB)             0x4002c0
 0x000000000000000a (STRSZ)              683 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x616000
 0x0000000000000002 (PLTRELSZ)           1416 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x400ce0
 0x0000000000000007 (RELA)               0x400c98
 0x0000000000000008 (RELASZ)             72 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x400c08
 0x000000006fffffff (VERNEEDNUM)         3
 0x000000006ffffff0 (VERSYM)             0x400b84
 0x0000000000000000 (NULL)               0x0
root@ubuntu01:~/malware/mas/mas_10#
```

[Figure 6]: Getting binary information (3)

According to the last two outputs, we can do the following considerations:

- This binary has 9 program headers, and readers can observe that each section is associated with one segment, which contains runtime execution.

- The main program headers (check Figure 5) have the following description:

    - **PHDR:** this header is a meta-segment, which contains the program header table and some metadata.
    - **INTERP:** this header an indication on the necessary system loader should be used to load this binary into memory. In this case, it is /lib64/ld-linux-x86-64.so.2.
    - **LOAD:** this headers provides information such as memory size, permission and alignment to the loader about how to load the binary into the memory.
    - **DYNAMIC:** this header provides information about the shared library dependencies and relocations.
    - **NOTE:** this header usually stores meta-information provided the vendor.
    - **GNU_PROPERTY:** this header is usually generated by the linker, and it is used to locate .note.gnu.property section.
    - **GNU_EH_FRAME:** this header provides the memory address of the stack unwind tables, which are used by exception handlers (throw or try/catch/finally).
    - **GNU_RELRO:** this header, named as Relocation Read-Only, is used for exploit mitigations such as -znow (full RELRO mitigation) or -zrelro (partial RELRO mitigation).

- **GNU_STACK:** this header, created by the linker, provides information whether stack is or not executable. Thus, the key information presented by this header is the memory protection used granted to the stack.

- The dynamic section has 28 entries, and a few of them are:

  - **NEEDED:** it indicates main libraries dependencies, which are used by the dynamic linker to load shared libraries when the program starts. In this case, the binary has a direct dependency on three shared libraries.
  - **STRTAB:** it holds a reference to the address of a string table, which contains strings used by the dynamic linker, as expected.
  - **SYMTAB:** it holds a reference to a symbol table, which contains information about symbols such as functions (FUNC), global data variable (OBJECT), sections (SECTION), thread-local data variable (TLS) and even symbols that do not have a specific type (NOTYPE). Additionally, symbols can be shared with external programs and libraries (GLOBAL), not accessible to other programs and libraries (LOCAL) and can even be used by function's implementation and overwritten in a later moment.
  - **STRSZ:** it represents the size of the string table.
  - **SYMENT:** it represents the size of the symbol table.

- There are many other **dynamic sections**, but they are not important for now.
- The **symbol table** referred to above can be checked by running the following command:

```
root@ubuntu01:~/malware/mas/mas_10# readelf -s mas_10.bin | head -20

Symbol table '.dynsym' contains 65 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND free@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (3)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.3.2 (4)
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
    11: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fread@GLIBC_2.2.5 (2)
    12: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.3.2 (4)
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
    15: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
    16: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
root@ubuntu01:~/malware/mas/mas 10#
```

[Figure 7]: Listing a symbol table (truncated)

Although we have a list of sections associated with segments using **readelf -lW**, they are not shown in an organized way, and a better approach would be using the following command:

```
root@ubuntu01:~/malware/mas/mas_10# objdump mas_10.bin -h

mas_10.bin:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000400238  0000000000400238  00000238  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  0000000000400254  0000000000400254  00000254  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000400274  0000000000400274  00000274  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .gnu.hash     00000028  0000000000400298  0000000000400298  00000298  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynsym       00000618  00000000004002c0  00000000004002c0  000002c0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynstr       000002ab  00000000004008d8  00000000004008d8  000008d8  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version  00000082  0000000000400b84  0000000000400b84  00000b84  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version_r 00000090  0000000000400c08  0000000000400c08  00000c08  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rela.dyn     00000048  0000000000400c98  0000000000400c98  00000c98  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rela.plt     00000588  0000000000400ce0  0000000000400ce0  00000ce0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .init         0000001a  0000000000401268  0000000000401268  00001268  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .plt          000003c0  0000000000401290  0000000000401290  00001290  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .text         0000ec60  0000000000401650  0000000000401650  00001650  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .fini         00000009  00000000004102b0  00000000004102b0  000102b0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .rodata       00003828  00000000004102c0  00000000004102c0  000102c0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 15 .eh_frame_hdr 0000041c  0000000000413ae8  0000000000413ae8  00013ae8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 16 .eh_frame     000010bc  0000000000413f08  0000000000413f08  00013f08  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 17 .init_array   00000008  0000000000615df0  0000000000615df0  00015df0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 18 .fini_array   00000008  0000000000615df8  0000000000615df8  00015df8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 19 .jcr          00000008  0000000000615e00  0000000000615e00  00015e00  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 20 .dynamic      000001f0  0000000000615e08  0000000000615e08  00015e08  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 21 .got          00000008  0000000000615ff8  0000000000615ff8  00015ff8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 22 .got.plt      000001f0  0000000000616000  0000000000616000  00016000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 23 .data         000040f4  0000000000616200  0000000000616200  00016200  2**5
                  CONTENTS, ALLOC, LOAD, DATA
 24 .bss          00101348  000000000061a300  000000000061a300  0001a2f4  2**5
                  ALLOC
 25 .comment      0000002b  0000000000000000  0000000000000000  0001a2f4  2**0
                  CONTENTS, READONLY
root@ubuntu01:~/malware/mas/mas 10#
```

[Figure 8]: Section list

- To the list of sections presented above, a concise description of the main ones follows below::

  - **.text section:** it contains instructions, is marked as read-only (as usually we see on PE binaries too) and code (executable).
  - **.data section:** it contains global and initialized variables.

- **.bss section**: it contains global and non-initialized variables.
- **.rodata section**: it contains global data that should not be changed (const variables).
- **.dynamic section**: it contains information used by the loader to link and, mainly, make the binary ready to be execute. Additionally, it also maintains an own string and symbol table.
- **.init section:** it stores the init function, which works as a constructor, which is called before the entry point similar to a constructor in a C++ program.
- **.fini section:** it stores the fini function, which works as a destructor, and it is called right before the program exit or the library is unloaded, and this behavior is also similar to a destructor in a C++ program.
- **.jcr section:** it stores information about Java classes that are required to be registered.
- **string table (.dynstr):** this section defines and contains all strings needed by the ELF binary, but it doesn't contain anything releated to literals used by the program. Thus, it is focused strictly on strings.
- **symbol table (.dynsym):** this section defines and contains a table of symbols (name, address and respective size of functions and variables, in general), which are usually a named location or even external library's reference used by the program.

If we are interested in examining potentially interesting sections such as **.rodata** of this binary, execute:

```
root@ubuntu01:~/malware/mas/mas_10# objdump -s -j .rodata mas_10.bin | head -32

mas_10.bin:     file format elf64-x86-64

Contents of section .rodata:
 4102c0 01000200 00000000 00000000 00000000  ................
 4102d0 65787061 6e642033 322d6279 7465206b  expand 32-byte k
 4102e0 65787061 6e642031 362d6279 7465206b  expand 16-byte k
 4102f0 766d782d 2a00706b 696c6c20 2d392025  vmx-*.pkill -9 %
 410300 73000000 00000000 65737863 6c69202d  s.......esxcli -
 410310 2d666f72 6d617474 65723d63 7376202d  -formatter=csv -
 410320 2d666f72 6d61742d 70617261 6d3d6669  -format-param=fi
 410330 656c6473 3d3d2257 6f726c64 49442c44  elds=="WorldID,D
 410340 6973706c 61794e61 6d652220 766d2070  isplayName" vm p
 410350 726f6365 7373206c 69737420 7c206177  rocess list | aw
 410360 6b202d46 20225c22 2a2c5c22 2a222027  k -F "\"*,\"*" '
 410370 7b737973 74656d28 22657378 636c6920  {system("esxcli
 410380 766d2070 726f6365 7373206b 696c6c20  vm process kill
 410390 2d2d7479 70653d66 6f726365 202d2d77  --type=force --w
 4103a0 6f726c64 2d69643d 22202431 297d2700  orld-id=" $1)}'.
 4103b0 72007700 4572726f 72206372 65617465  r.w.Error create
 4103c0 206e6f74 6520696e 20646972 2025730a   note in dir %s.
 4103d0 00457272 6f722070 61727365 20636667  .Error parse cfg
 4103e0 00666174 616c2065 72726f72 206d616c  .fatal error mal
 4103f0 6c6f6320 656e6300 25732573 0025732e  loc enc.%s%s.%s.
 410400 6c636b00 00000000 25732069 7320616c  lck.....%s is al
 410410 72656164 7920696e 2070726f 67726573  ready in progres
 410420 73206f66 2065636e 72797074 696f6e2e  s of ecnryption.
 410430 2e2e0a00 772b0072 622b0046 696c6520  ....w+.rb+.File
 410440 69732065 6e637279 70746564 20627920  is encrypted by
 410450 64617461 2025730a 0046696c 65206572  data %s..File er
 410460 726f7220 005b2573 5d20616c 72656164  ror .[%s] alread
 410470 7920656e 63727970 7465640a 00000000  y encrypted.....
```

[Figure 9]: Examining first bytes from .rodata section

Through the output we found strings related command to be executed (**esxclivm process kill --type=force**) and to encryption scripts, which tell us a bit about the malware/ransomware operation.

Repeating the same command, but for **.data section,** we have:

```
root@ubuntu01:~/malware/mas/mas_10# objdump -s -j .data mas_10.bin | head -40

mas_10.bin:     file format elf64-x86-64

Contents of section .data:
 616200 00000000 00000000 00000000 00000000  ................
 616210 00000000 00000000 00000000 00000000  ................
 616220 32000000 00000000 f0024100 00000000  2.........A.....
 616230 f6024100 00000000 08034100 00000000  ..A.......A.....
 616240 bd084100 00000000 01000000 00000000  ..A.............
 616250 00000000 00000000 70000000 00000000  ........p.......
 616260 c2084100 00000000 01000000 00000000  ..A.............
 616270 00000000 00000000 74000000 00000000  ........t.......
 616280 ca084100 00000000 00000000 00000000  ..A.............
 616290 00000000 00000000 73000000 00000000  ........s.......
 6162a0 00000000 00000000 00000000 00000000  ................
 6162b0 00000000 00000000 00000000 00000000  ................
 6162c0 7b22706b 223a2234 6e4f4e75 34476d61  {"pk":"4nONu4Gma
 6162d0 48663430 52764268 48636c70 616d7063  Hf40RvBhHclpampc
 6162e0 734b795a 4d786653 656c674d 6d5a452f  sKyZMxfSelgMmZE/
 6162f0 6e493d22 2c227069 64223a22 24326124  nI=","pid":"$2a$
 616300 31322444 33576b34 642e6379 30653045  12$D3Wk4d.cy0e0E
 616310 69567144 504a6531 2e30364f 4d523364  iVqDPJe1.06OMR3d
 616320 756f4d52 49483738 69375846 5862536b  uoMRIH78i7XFXbSk
 616330 434c4875 4c6f4d47 222c2273 7562223a  CLHuLoMG","sub":
 616340 22383633 39222c22 64626722 3a66616c  "8639","dbg":fal
 616350 73652c22 6574223a 302c226e 626f6479  se,"et":0,"nbody
 616360 223a224c 53307450 54303949 46646c62  ":"LS0tPT09IFdlb
 616370 474e7662 57557549 45466e59 576c754c  GNvbWUuIEFnYWluL
 616380 69413950 5430744c 53304b43 6c737458  iA9PT0tLS0KClstX
 616390 53425861 47463063 79424959 5842775a  SBXaGF0cyBIYXBwZ
 6163a0 57342f49 46737458 516f4b57 57393163  W4/IFstXQoKWW91c
 6163b0 69426d61 57786c63 79426863 6d55675a  iBmaWxlcyBhcmUgZ
 6163c0 57356a63 6e6c7764 47566b4c 43426862  W5jcnlwdGVkLCBhb
 6163d0 6d516759 33567963 6d567564 47783549  mQgY3VycmVudGx5I
 6163e0 48567559 585a6861 57786859 6d786c4c  HVuYXZhaWxhYmxlL
 6163f0 69425a62 33556759 32467549 474e6f5a  iBZb3UgY2FuIGNoZ
 616400 574e7249 476c304f 69426862 4777675a  WNrIGl0OiBhbGwgZ
 616410 6d6c735a 584d6762 32346765 57393163  mlsZXMgb24geW91c
 616420 69427a65 584e305a 57306761 47467a49  iBzeXN0ZW0gaGFzI
 616430 47563464 47567563 326c7662 69423752  GV4dGVuc2lvbiB7R
root@ubuntu01:~/malware/mas/mas 10#
```

[Figure 10]: Checking first bytes from .data section

Of course, there is something within this section, but we do not have further details. Apparently it is an exceedingly long string and includes a series of keys and associated values like a dictionary. In this case, the obvious movement is extract it using simple commands like strings. Additionally, we can use the same strings command to search for anything else that could be useful for us:

- **strings -a -n200 mas_10.bin | jq**

```
root@ubuntu01:~/malware/mas/mas_10# strings -a -n200 mas_10.bin | jq
{
  "pk": "4nONu4GmaHf40RvBhHclpampcsKyZMxfSelgMmZE/nI=",
  "pid": "$2a$12$D3Wk4d.cy0e0EiVqDPJe1.06OMR3duoMRIH78i7XFXbSkCLHuLoMG",
  "sub": "8639",
  "dbg": false,
  "et": 0,
  "nbody": "LS0tPT09IFdlbGNvbWUuIEFnYWluLiA9PT0tLS0KClstXSBXaGF0cyBIYXBwZW4/IFstXQoKWW91c
iBmaWxlcyBhcmUgZW5jcnlwdGVkLCBhbmQgY3VycmVudGx5IHVuYXZhaWxhYmxlLiBZb3UgY2FuIGNoZWNrIGl0Oi
BhbGwgZmlsZXMgb24geW91ciBzeXN0ZW0gaGFzIGV4dGVuc2lvbiB7RVhUfS4KQnkgdGhlIHdheSwgZXZlcnl0aGl
uZyBpcyBwb3NzaWJsZSB0byByZWNvdmVyIChyZXN0b3JlKSwgYnV0IHlvdSBuZWVkIHRvIGZvbGxvdyBvdXIgaW5z
dHJ1Y3Rpb25zLiBPdGhlcndpc2UsIHlvdSBjYW50IHJldHVybiB5b3VyIGRhdGEgKE5FVkVSKS4KClstXSBXaGF0I
Gd1YXJhbnRlZXM/IFstXQoKSXRzIGp1c3QgYSBidXNpbmVzcy4gV2UgYWJzb2x1dGVseSBkbyBub3QgY2FyZSBhYm
91dCB5b3UgYW5kIHlvdXIgZGVhbHMsIGV4Y2VwdCBnZXR0aW5nIGJlbmVmaXRzLiBJZiB3ZSBkbyBub3QgZG8gb3V
yIHdvcmsgYW5kIGxpYWJpbGl0aWVzIC0gbm9ib2R5IHdpbGwgbm90IGNvb3BlcmF0ZSB3aXRoIHVzLiBJdHMgbm90
IGluIG91ciBpbnRlcmVzdHMuClRvIGNoZWNrIHRoZSBhYmlsaXR5IG9mIHJldHVybmluZyBmaWxlcywgeW91IHNob
3VsZCBnbyB0byBvdXIgd2Vic2l0ZS4gVGhlcmUgeW91IGNhbiBkZWNyeXB0IG9uZSBmaWxlIGZvciBmcmVlLiBUaG
F0IGlzIG91ciBndWFyYW50ZWUuCklmIHlvdSB3aWxsIG5vdCBjb29wZXJhdGUgd2l0aCBvdXIgc2VydmljZSAtIGZ
vciB1cywgaXRzIGRvZXMgbm90IG1hdHRlci4gQnV0IHlvdSB3aWxsIGxvc2UgeW91ciB0aW1lIGFuZCBkYXRhLCBj
YXVzZSBqdXN0IHdlIGhhdmUgdGhlIHByaXZhdGUga2V5LiBJbiBwcmFjdGljZSAtIHRpbWUgaXMgbXVjaCBtb3JlI
HZhbHVhYmxlIHRoYW4gbW9uZXkuCgpbK10gSG93IHRvIGdldCBhY2Nlc3Mgb24gd2Vic2l0ZT8gWytdCgpVc2luZy
BhIFRPUiBicm93c2VyIQogIDEpIERvd25sb2FkIGFuZCBpbnN0YWxsIFRPUiBicm93c2VyIGZyb20gdGhpcyBzaXR
lOiBodHRwczovL3RvcnByb2plY3Qub3JnLwogIDIpIE9wZW4gb3VyIHdlYnNpdGU6IGh0dHA6Ly9hcGxlYnp1NDd3
Z2F6YXBkcWtzNnZyY3Y2emNuanBwa2J4YnI2d2tldGY1Nm5mNmFxMm5teW95ZC5vbmlvbi97VUlEfQoKV2FybmluZ
zogc2Vjb25kYXJ5IHdlYnNpdGUgY2FuIGJlIGJsb2NrZWQsIHRoYXRzIHdoeSBmaXJzdCB2YXJpYW50IG11Y2ggYm
V0dGVyIGFuZCBtb3JlIGF2YWlsYWJsZS4KCldoZW4geW91IG9wZW4gb3VyIHdlYnNpdGUsIHB1dCB0aGUgZm9sbG9
3aW5nIGRhdGEgW4gdGhlIGlucHV0IGZvcm06CktleToKCgp7S0VZfQoKCi0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS
LS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tC
gohISEgREFOR0VSICEhIQpET04nVCB0cnkgdG8gY2hhbmdlIGZpbGVzIGJ5IHlvdXJzZWxmLCBET04nVCB1c2UgYW
55IHRoaXJkIHBhcnR5IHNvZnR3YXJlIGZvciByZXN0b3JpbmcgeW91ciBkYXRhIG9yIGFudGl2aXJ1cyBzb2x1dGl
vbnMgLSBpdHMgbWF5IGVudGFpbCBkYW1hZ2Ugb2YgdGhlIHByaXZhdGUga2V5IGFuZCwgYXMgcmVzdWx0LCBUaGUg
TG9zcyBhbGwgZGF0YS4KISEhICEhISAhISEKT05FIE1PUkUgVElNRTogSXRzIGluIHlvdXIgaW50ZXJlc3RzIHRvI
GdldCB5b3VyIGZpbGVzIGJhY2suIEZyb20gb3VyIHNpZGUsIHdlICh0aGUgYmVzdCBzcGVjaWFsaXN0cykgbWFrZS
BldmVyeXRoaW5nIGZvciBmcmVlIGJ1dCBwbGVhc2UgIG5vdCBwYXRhIG9uZSBmaWxlIGZvciBmcmVlLiBUaUciEhISAhISE
gISEhAA==",
  "nname": "{EXT}-readme.txt",
  "rdmcnt": 0,
  "ext": ".vemar"
}
```

[Figure 11]: Extracting the long string and formatting it

The **nbody** field can be decoded using **base64 -d <Base64 message>** command, the result is the following:

```
---=== Welcome. Again. ===---

[-] Whats Happen? [-]

Your files are encrypted, and currently unavailable. You can check it: all files on your system
 has extension {EXT}.
By the way, everything is possible to recover (restore), but you need to follow our instruction
s. Otherwise, you cant return your data (NEVER).

[-] What guarantees? [-]

Its just a business. We absolutely do not care about you and your deals, except getting benefit
s. If we do not do our work and liabilities - nobody will not cooperate with us. Its not in our
 interests.
To check the ability of returning files, You should go to our website. There you can decrypt on
e file for free. That is our guarantee.
If you will not cooperate with our service - for us, its does not matter. But you will lose you
r time and data, cause just we have the private key. In practice - time is much more valuable t
han money.
```

```
[-] What guarantees? [-]

Its just a business. We absolutely do not care about you and your deals, except getting benefit
s. If we do not do our work and liabilities - nobody will not cooperate with us. Its not in our
 interests.
To check the ability of returning files, You should go to our website. There you can decrypt on
e file for free. That is our guarantee.
If you will not cooperate with our service - for us, its does not matter. But you will lose you
r time and data, cause just we have the private key. In practice - time is much more valuable t
han money.

[+] How to get access on website? [+]

Using a TOR browser!
  1) Download and install TOR browser from this site: https://torproject.org/
  2) Open our website: http://aplebzu47wgazapdqks6vrcv6zcnjppkbxbr6wketf56nf6aq2nmyoyd.onion/{U
ID}

Warning: secondary website can be blocked, thats why first variant much better and more availab
le.

When you open our website, put the following data in the input form:
Key:


{KEY}


--------------------------------------------------------------------------------

!!! DANGER !!!
DON'T try to change files by yourself, DON'T use any third party software for restoring your da
ta or antivirus solutions - its may entail damage of the private key and, as result, The Loss a
ll data.
!!! !!! !!!
ONE MORE TIME: Its in your interests to get your files back. From our side, we (the best specia
lists) make everything for restoring, but please should not interfere.
```

[Figure 12]: Decoded ransomware message

Both extracted and decoded strings show that apparently the encrypted files have extension .vemar, the website is on the Darknet (as usual), there is a KEY (probably dynamically generated) to submit the form, and there is an associated UID.

```
root@ubuntu01:~/malware/mas/mas_10# strings -a -n50 mas_10.bin | head -12
esxcli --formatter=csv --format-param=fields=="WorldID,DisplayName" vm pro
cess list | awk -F "\"*,\"*" '{system("esxcli vm process kill --type=force
 --world-id=" $1)}'
[%s] semms to be protected by os but let's encrypt anyway...
iji iji iji iji iji iji tYiji iji iji ifi iji iji iji
iji iji iji iji ij.----------.ji iji ifi iji iji iji
iji iji iji iji ij| ENCRYPTED |ji iji ifi iji iji iji
iji iji iji iji ij|- - - - - -|ji iji ifi iji iji iji
iji iji iji iji ij|   FILES   |ji iji ifi iji iji iji
iji iji iji iji ij|           |ji iji ifi iji iji iji
iji iji iji iji ij|    MBs    |ji iji ifi iji iji iji
iji iji iji iji ij'__________'ji iji ifi iji iji iji
Key inizialization error ... something wrong with config
Using silent mode, if you on esxi - stop VMs manualy
root@ubuntu01:~/malware/mas/mas_10#
```

[Figure 13]: Further strings found in the binary

The only note is that the ransomware is looking for VMware ESXi installations and kill any associated process before executing any operation.

It is time to install Radare2 to perform a quick triage:

- **git clone** https://github.com/radareorg/radare2
- **cd radare2**
- **git clone https://github.com/radareorg/radare2**

Executing a few basic commands, we have:

```
root@ubuntu01:~/malware/mas/mas_10# r2 -e emu.str=true -e bin.relocs.apply=true mas_10.bin
 -- Select your architecture with: 'e asm.arch=<arch>' or r2 -a from the shell
[0x00401650]> aaa
INFO: Analyze all flags starting with sym. and entry0 (aa)
INFO: Analyze imports (af@@@i)
INFO: Analyze entrypoint (af@ entry0)
INFO: Analyze symbols (af@@@s)
INFO: Analyze all functions arguments/locals (afva@@@F)
INFO: Analyze function calls (aac)
INFO: Analyze len bytes of instructions for references (aar)
INFO: Finding and parsing C++ vtables (avrr)
INFO: Analyzing methods (af @@ method.*)
INFO: Recovering local variables (afva@@@F)
INFO: Type matching analysis for all functions (aaft)
INFO: Propagate noreturn information (aanr)
INFO: Use -AA or aaaa to perform additional experimental analysis
[0x00401650]> iz~http
[0x00401650]> izz~http
[0x00401650]>
```

[Figure 14]: Radare2 | output 1

There is not any string related to http in data section (iz) or even the whole binary (izz), as expected. At this point, we can search for additional indicators such as:

- encrypt
- esxcli
- crypto
- download (or upload)

```
[0x00401650]> izz~encrypt

277 0x0001043b 0x0041043b 29   30   .rodata   ascii   File is encrypted by data %s\n
279 0x00010465 0x00410465 23   24   .rodata   ascii   [%s] already encrypted\n
280 0x00010480 0x00410480 61   62   .rodata   ascii   [%s] semms to be protected by os but le
t's encrypt anyway...\n
282 0x000104cf 0x004104cf 28   29   .rodata   ascii   File [%s] was NOT encrypted\n
283 0x000104ec 0x004104ec 24   25   .rodata   ascii   File [%s] was encrypted\n
[0x00401650]>

[0x00401650]> izz~esxcli

270 0x00010308 0x00410308 167  168  .rodata   ascii   esxcli --formatter=csv --format-param=f
ields=="WorldID,DisplayName" vm process list | awk -F "\"*,\"*" '{system("esxcli vm process k
ill --type=force --world-id=" $1)}'
[0x00401650]>
```

[Figure 15]: Radare2 | output 2

Get a compact list of imported functions, the number of imports and also the number of the functions:

```
[0x00401650]> ii | head -40
[Imports]
nth vaddr      bind   type   lib name
-------------------------------------
1   0x004012a0 GLOBAL FUNC       free
2   0x004012b0 GLOBAL FUNC       pthread_create
3   0x004012c0 GLOBAL FUNC       unlink
4   0x004012d0 GLOBAL FUNC       strncpy
5   ---------- WEAK   NOTYPE     _ITM_deregisterTMCloneTable
6   0x004012e0 GLOBAL FUNC       strcpy
7   0x004012f0 GLOBAL FUNC       ftello
8   0x00401300 GLOBAL FUNC       pthread_cond_broadcast
9   0x00401310 GLOBAL FUNC       toupper
10  0x00401320 GLOBAL FUNC       puts
11  0x00401330 GLOBAL FUNC       fread
12  0x00401340 GLOBAL FUNC       pthread_cond_wait
13  0x00401350 GLOBAL FUNC       fclose
14  0x00401360 GLOBAL FUNC       opendir
15  0x00401370 GLOBAL FUNC       strlen
16  0x00401380 GLOBAL FUNC       getopt_long
17  0x00401390 GLOBAL FUNC       strchr
18  0x004013a0 GLOBAL FUNC       printf
19  0x004013b0 GLOBAL FUNC       rewind
20  0x004013c0 GLOBAL FUNC       pclose
21  0x004013d0 GLOBAL FUNC       snprintf
22  0x004013e0 GLOBAL FUNC       memset
23  0x004013f0 GLOBAL FUNC       pow
24  0x00401400 GLOBAL FUNC       close
25  0x00401410 GLOBAL FUNC       closedir
26  0x00401420 GLOBAL FUNC       read
27  0x00401430 GLOBAL FUNC       __libc_start_main
28  0x00401440 GLOBAL FUNC       pthread_cond_signal
29  0x00401450 GLOBAL FUNC       calloc
30  0x00401460 GLOBAL FUNC       strcmp
31  0x00401470 GLOBAL FUNC       ftell
32  0x00401480 WEAK   NOTYPE     __gmon_start__
33  0x00401490 GLOBAL FUNC       pthread_timedjoin_np
34  0x004014a0 GLOBAL FUNC       time
35  0x004014b0 GLOBAL FUNC       __xstat
36  0x004014c0 GLOBAL FUNC       pthread_cond_init
37  0x004014d0 GLOBAL FUNC       readdir
[0x00401650]>
[0x00401650]> ii | wc -l
66
[0x00401650]> afl | wc -l
155
```

[Figure 16]: Radare2 | output 3

If compared to other binaries and mainly ransomware threats, there are not too many imported functions or defined functions, and this is good because it makes the analysis less complicated and extensive. Additionally, there is no evidence that there are C++ functions (mainly those using templates). For example, a first e very trivial method for searching for C++ functions would be to check for "::".

To expand the usage on Radare2, you can use its package manager and install plugins such as r2ghidra, r2dec and r2frida. However, my advice is that you use the stable release (5.9.4 or close version) because the version from the GitHub might be unstable with plugins. Therefore, the task set is reduced to collect visualize code and inter-connections:

- **curl -Ls https://github.com/radareorg/radare2/releases/download/5.9.4/radare2-5.9.4.tar.xz | tar xJv**
- **radare2-5.9.4/sys/install.sh**
- **apt install ninja**
- **apt install make**
- **ap install meson**
- **r2pm -Uci r2ghidra r2dec r2frida**

Files related to **radare2** are installed in **/root/.local/share/radare2** directory. For example, you can follow from this point using **r2dec plugin** for decompiling functions from binary:

```
[0x00401650]> s main

[0x0040687f]> pd 30

            ; DATA XREF from entry0 @ 0x40166d(r)
┌ 341: int main (signed int64_t argc, char **argv);
│ `- args(rdi, rsi) vars(6:sp[0x10..0x38])
│           0x0040687f      55              push rbp
│           0x00406880      4889e5          mov rbp, rsp
│           0x00406883      4883ec30        sub rsp, 0x30
│           0x00406887      897ddc          mov dword [var_24h], edi    ; argc
│           0x0040688a      488975d0        mov qword [var_30h], rsi    ; argv
│           0x0040688e      c745e80000..    mov dword [var_18h], 0
│           0x00406895      837ddc01        cmp dword [var_24h], 1
│       ┌─< 0x00406899      7f14            jg 0x4068af                 ; unlikely
│       │   0x0040689b      bf58074100      mov edi, str.Revix_1.2a__r_nUsage_example:_elf.exe
___path__vmfs____threads_5_r_n__silent___s__use_for_not_stoping_VMs_mode_r_n___BY_DEFAULT_THI
S_SOFTWARE_USES_50_THREADS____r ; 0x410758 ; "Revix 1.2a \r\nUsage example: elf.exe --path /v
mfs/ --threads 5\r\n--silent (-s) use for not stoping VMs mode\r\n!!!BY DEFAULT THIS SOFTWARE
 USES 50 THREADS!!!\r" ; const char *s
│       │   0x004068a0      e87baaffff      call sym.imp.puts           ; int puts(const char
*s)
│       │                                                               ; int puts("Revix 1.2a
 \r\nUsage example: elf.exe --path /vmfs/ --threads 5\r\n--silent (-s) use for not stoping VM
s mode\r\n!!!BY DEFAULT THIS SOFTWARE USES 50 THREADS!!!\r")
│       │   0x004068a5      b800000000      mov eax, 0
│       ┌─< 0x004068aa      e923010000      jmp 0x4069d2
│       │   ; CODE XREF from main @ 0x406899(x)
│       └─> 0x004068af      488b55d0        mov rdx, qword [var_30h]
│           0x004068b3      8b45dc          mov eax, dword [var_24h]
│           0x004068b6      4889d6          mov rsi, rdx                ; char **arg2
│           0x004068b9      89c7            mov edi, eax                ; int64_t arg1
│           0x004068bb      e82cffffff      call fcn.004067ec           ; fcn.004067ec(-1, -1)
│           0x004068c0      488945f8        mov qword [var_8h], rax
│           0x004068c4      488b45d0        mov rax, qword [var_30h]
│           0x004068c8      488b00          mov rax, qword [rax]
│           0x004068cb      488945f0        mov qword [var_10h], rax
│           0x004068cf      48837df800      cmp qword [var_8h], 0
│       ┌─< 0x004068d4      7514            jne 0x4068ea                ; likely
│       │   0x004068d6      bff3074100      mov edi, str.Unable_to_find_path_argument ; 0x4107
f3 ; "Unable to find path argument" ; const char *s
│       ||   0x004068db      e840aaffff      call sym.imp.puts           ; int puts(const char
*s)
│       ||                                                              ; int puts("Unable to
find path argument")
│       ||   0x004068e0      b800000000      mov eax, 0
│      ┌──< 0x004068e5      e9e8000000      jmp 0x4069d2
│      ||   ; CODE XREF from main @ 0x4068d4(x)
```

[Figure 17]: Radare2 | output 4

I am not going to proceed using **Radare2**, but certainly many readers might follow the analysis using it because messages and instructions there are truly clear.

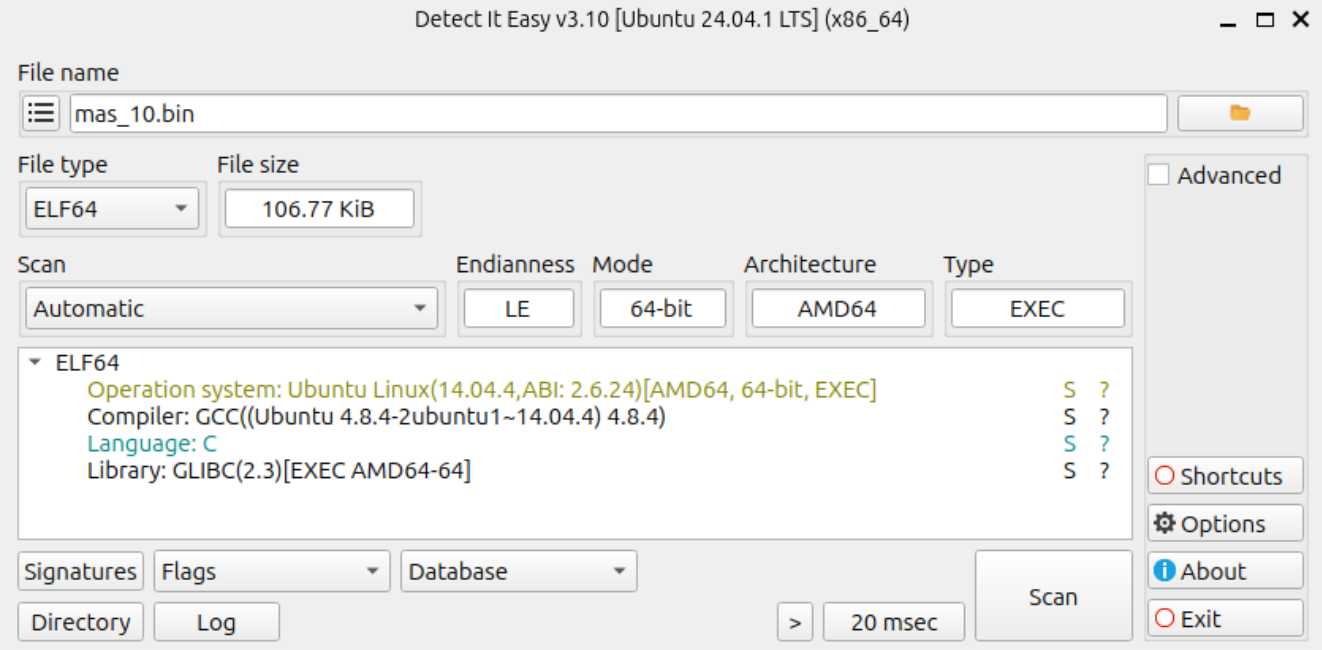
Check for libraries loaded dynamically at loading time:

```
root@ubuntu01:~/malware/mas/mas_10# ldd mas_10.bin
        linux-vdso.so.1 (0x00007ffed9586000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007b45ecf2b000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007b45ecf26000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007b45ecc00000)
        /lib64/ld-linux-x86-64.so.2 (0x00007b45ed035000)
root@ubuntu01:~/malware/mas/mas_10#
```

[Figure 18]: Shared libraries

This ransomware, like most of other ones, uses multi-threads and there is nothing new here.

If you want to install **Detect It Easy (DiE)** on your system, I recommend that you read the respective page: https://github.com/horsicq/Detect-It-Easy/blob/master/docs/BUILD.md



[Figure 19]:Checking further binary information and packers

We see that it is a typical binary:

- 64-bit for Ubuntu AMD64.
- Compiled using GCC.
- Using typica GLIBC.
- No further external libraries.

Readers could use **signsrch** (wget https://aluigi.altervista.org/mytoolz/signsrch.zip), which can be easily compiled as shown below, to detect eventual algorithms being used by the malware:

- **unzip signsrch.zip -d signsrch_dir**
- **cd signsrch_dir/src ; make**
- **cp ../signsrch.sig .**

```
root@ubuntu01:~/tools/signsrch_dir# ./signsrch -e /root/malware/mas/mas_10/mas_10.bin

Signsrch 0.2.4
by Luigi Auriemma
e-mail: aluigi@autistici.org
web:    aluigi.org
  optimized search function by Andrew http://www.team5150.com/~andrew/
  disassembler engine by Oleh Yuschuk

- open file "/root/malware/mas/mas_10/mas_10.bin"
- 109336 bytes allocated
- load signatures
- open file ./signsrch.sig
- 3075 signatures in the database
- start 2 threads
- start signatures scanning:

  offset   num  description [bits.endian.size]
  ----------------------------------------
  004102d0 2612 SALSA [32.le.24&]
  00411260 2005 B64EncodeTable [..64]
  00411260 1996 rfc3548 Base 64 Encoding with URL and Filename Safe Alphabet [..62]
  004112c0 896  Rijndael Te0 (0xc66363a5U) [32.le.1024]
  004116c0 898  Rijndael Te1 (0xa5c66363U) [32.le.1024]
  00411ac0 900  Rijndael Te2 (0x63a5c663U) [32.le.1024]
  00411ec0 902  Rijndael Te3 (0x6363a5c6U) [32.le.1024]
  004122c0 904  Rijndael Te4 (0x63636363U) [32.le.1024]
  004126c0 905  Rijndael Td0 (0x51f4a750U) [32.le.1024]
  00412ac0 907  Rijndael Td1 (0x5051f4a7U) [32.le.1024]
  00412ec0 909  Rijndael Td2 (0xa75051f4U) [32.le.1024]
  004132c0 911  Rijndael Td3 (0xf4a75051U) [32.le.1024]
  004136c0 913  Rijndael Td4 (0x52525252U) [32.le.1024]
  00413ac0 914  Rijndael rcon [32.le.40]

- 14 signatures found in the file in 0 seconds
- done
root@ubuntu01:~/tools/signsrch_dir#
```

[Figure 20]: Signsrch output

You can also check the file's entropy, and there are diverse ways to do it, where one of them is using the simple **ent** command (**apt -y install ent**), as shown below:

```
root@ubuntu01:~/malware/mas/mas_10# ent mas_10.bin
Entropy = 5.667002 bits per byte.

Optimum compression would reduce the size
of this 109336 byte file by 29 percent.

Chi square distribution for 109336 samples is 2519200.10, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 88.0705 (127.5 = random).
Monte Carlo value for Pi is 3.383602239 (error 7.70 percent).
Serial correlation coefficient is 0.416685 (totally uncorrelated = 0.0)
root@ubuntu01:~/malware/mas/mas_10#
```

[Figure 21]: ent command

Another tool that might produce some information to be used later is **binwalk** (**apt -y install binwalk**), but this time the information is not relevant, as shown below:

```
root@ubuntu01:~/malware/mas/mas_10# binwalk mas_10.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             ELF, 64-bit LSB executable, AMD x86-64, version 1 (SYSV)
70240         0x11260         Base64 standard index table
```

[Figure 22]: binwalk output

Based on indicators collected by **ldd, DiE, signsrch, ent** and **binwalk** commands, a list of comments follows below:

- The binary is **64-bit** and has **amd64 architecture**.
- It is **dynamically linked** then it depends on shared libraries.
- There is a series of cryptographic constants being used by ransomware, which suggests that it might be using algorithm **SALSA20** (stream cipher) and **AES** for symmetric encryptions. Both facts are not an absolute certainty, and they should be checked during the analysis.
- The binary is **stripped**, so there is not any symbol information inside.
- The binary was compiled using **GCC**, which could be useful information while reversing it.
- The **entropy (5.66)** is low, which can suggest a non-packed binary.
- The **program headers**, which is a **table that provides loaders with information to load a binary into memory and, as expected, it is more focused on memory mapping**, has 9 entries.
- The **section headers**, which **provide us with information about the binary itself and it is naturally focused on binary sections**, are counted in 28 entries.
- The shown **.init section** stores the **init function**, which works as a constructor, and it is called before the entry point. Likewise, the **.fini section** stores the **fini function**, which works as a destructor, and it is called right before the program quitting, or the library being unloaded.

The malicious binary is using a series of shared libraries, as expected in most cases, and then there are many external symbols coming from these libraries too. Once a symbol is found, the loader writes the symbol's address into a location (actually a slot within a table) indicated by the relocation entry (check **readelf** and **objdump** outputs).

Later, when such a symbol needs to be used by the malware, a call will be performed using the same slot of the table which contains the symbol's address, which remains constant for later occurrences. The mentioned table used for dynamic relocations is named **GOT (Global Offset Table)**, and readers can see a reference to it by looking for **.got section** in previous commands.

There is another table used by dynamic allocation that is called **PLT (Procedure Linkage Table)**, which is associated with the **.plt section** (as expected, of course). This table is typically associated with a lazy symbol binding concept, which means that not all symbols are used when a program launches, so there is a delay for the resolution procedure until such a symbol is really used.

Thus, functions always call the stub address stored in the **PLT** and, when a function is called for the first time, the **PLT** forwards the call to a resolver function that will find the real address of the symbol and update the **GOT** with the real address. Afterwards, it also updated the **PLT** with the real address of the symbol because subsequent calls will check the **PLT**.

These concepts are used by different threats and exploits, and a few examples are:

- Exploits can use **GOT** to find API functions' addresses such as system, setuid, memcpy, strcpy and many other ones. Eventually such an approach would only be feasible if **GOT** can be found in a static location (predictable), which happens only with non-PIE programs.

- Readers could remember **PIE (Position Independent Executable)** programs can be loaded in any address in the memory, which makes finding **GOT** harder. Obviously, it would not be a concern for malicious binaries.

- There are exploits that replace entries in **GOT** with addresses pointing to a malicious function or even a payload.

At the same way that we have seen on Windows operating system, malware threats have injected shared libraries into processes to read and extract information, escalate privileges, hook and even establishing persistence, and all of these mentioned attacks start by compromising the process structure (represented by **task_struct**), which hold memory maps, opened file descriptors, scheduling information and so on, and it is related through the slab allocator (and variants) to **kmem_cache** that caches kernel objects and related information such as respective pointers and meta-information.

Finally, we have the main tool for any malware triage that is **capa** (from Mandiant), which one of last versions can be downloaded from https://github.com/mandiant/capa/releases/tag/v8.0.1. Personally, I copy capa binary to **/usr/local/bin** directory, but readers can adopt any approach to put it in a valid path. Executing capa binary is direct, as shown below:

```
root@ubuntu01:~/malware/mas/mas_10# capa mas_10.bin
```

md5	c83df66c46bcbc05cd987661882ff061
sha1	48d1558fe3ac689b7eaac82738a023c13f4c0e7c
sha256	f864922f947a6bb7d894245b53795b54b9378c0f7633c521240488e86f60c2c5
analysis	static
os	linux
format	elf
arch	amd64
path	/root/malware/mas/mas_10/mas_10.bin

ATT&CK Tactic	ATT&CK Technique
DEFENSE EVASION	Obfuscated Files or Information [T1027]
DISCOVERY	File and Directory Discovery [T1083]

MBC Objective	MBC Behavior
CRYPTOGRAPHY	Encrypt Data::AES [C0027.001]
	Encrypt Data::RC4 [C0027.009]
	Generate Pseudo-random Sequence::RC4 PRGA [C0021.004]
DATA	Check String [C0019]
	Checksum::CRC32 [C0032.001]
	Encode Data::Base64 [C0026.001]
DEFENSE EVASION	Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]
	Obfuscated Files or Information::Encryption-Standard Algorithm [E1027.m05]
DISCOVERY	File and Directory Discovery [E1083]
FILE SYSTEM	Delete File [C0047]
	Move File [C0063]
	Read File [C0051]
	Set File Attributes [C0050]
	Writes File [C0052]
PROCESS	Create Process [C0017]
	Create Thread [C0038]

Capability	Namespace
hash data with CRC32	data-manipulation/checksum/crc32
encode data using Base64	data-manipulation/encoding/base64
reference Base64 string	data-manipulation/encoding/base64
encrypt data using AES (2 matches)	data-manipulation/encryption/aes
encrypt data using RC4 PRGA	data-manipulation/encryption/rc4
change file permission on Linux (2 matches)	host-interaction/file-system
delete file on Linux	host-interaction/file-system/delete
enumerate files recursively	host-interaction/file-system/files/list
move file	host-interaction/file-system/move
read file on Linux (5 matches)	host-interaction/file-system/read
write file on Linux (2 matches)	host-interaction/file-system/write
create process on Linux (4 matches)	host-interaction/process/create
create thread (2 matches)	host-interaction/thread/create

[Figure 23]: capa output

Based on information offered by **capa**, we have an initial direction to proceed with our analysis:

- The ransomware apparently uses RC4 and AES as symmetric cryptography.
- There are pieces of code in Base64 (we already know this fact).
- There is a file and directory discovery portion as performed by any ransomware.
- The ransomware removes (of course), moves, reads, and writes files (nothing new).
- Threads are created, as expected, due to performance issues.

It is a pretty standard ransomware.

Although it is not important to a malicious binary, you should remember that Linux provides a series of standard binary protections such as:

- **NX (no-execute bit):** there are regions of memory marked as non-executable.
- **Stack Canaries:** a protection to detect stack overflow.
- **ASLR:** system, application processes and libraries are loaded at random addresses.
- **PIE:** executable binaries are loaded at random addresses every time they are launched.
- **Fortity-Source:** additional checks added to detect buffer overflow and potential runtime exploitation.
- **CFI (Control Flow Integrity):** it checks that the control flow of the program has not been changed, which makes the usage of exploitation techniques such as ROP and JOP harder.
- **Relocation Read-Only (RELRO):** GOT is changed read-only after the initial linker resolution, which prevents GOT from being overwritten by attackers, who aim to detour the execution flow to a malicious payload.
- **RTLD_NOW:** this flag, used by function such as **dlopen( )**, forces the symbol resolution and related relocations for shared libraries to happen at load time.

Of course, there are other protections (most of the time, variations from these ones presented above).

Additionally, there are multiple tools to check some of these binary protections, and such tools can be installed using the following commands:

- **binary-security-check:** cargo install binary-security-check (of course, the system must have Rust and associated packages installed).

- **checksec:**
    - git clone https://github.com/slimm609/checksec
    - go build

- **harnening-check:** apt -y install devscripts

Using **checksec and binary-security-check** tools, we have the following:

```
root@ubuntu01:~/github/checksec# ./checksec file ../../malware/mas/mas_10/mas_10.bin

  _____ _    _ ______ _____ _  __ _____ ______ _____
 / ____| |  | |  ____/ ____| |/ // ____|  ____/ ____|
| |    | |__| | |__ | |    | ' /| (___ | |__ | |
| |    |  __  |  __|| |    |  <  \___ \|  __|| |
| |____| |  | | |___| |____| . \ ____) | |____| |____
 \_____|_|  |_|______\_____|_|\_\_____/|______\_____|

RELRO           Stack Canary      NX            PIE             RPATH      RUNPATH      Symbols
      FORTIFY    Fortified   Fortifiable      Name
Partial RELRO   No Canary Found   NX enabled    PIE Disabled    No RPATH   No RUNPATH   No Symbols
      No          0            7              ../../malware/mas/mas_10/mas_10.bin
root@ubuntu01:~/github/checksec#
root@ubuntu01:~/github/checksec# binary-security-check ../../malware/mas/mas_10/mas_10.bin | sed "
s/\!/\\n[+] /g"
../../malware/mas/mas_10/mas_10.bin:
[+] ASLR
[+] STACK-PROT +READ-ONLY-RELOC
[+] IMMEDIATE-BIND
[+] FORTIFY-SOURCE(
[+] strcpy,
[+] strncpy,
[+] printf,
[+] snprintf,
[+] strcat,
[+] fread,
[+] memset,
[+] read,
[+] sprintf)
```

[Figure 23]: Checking protections using multiple tools

As expected, the purpose of malware or ransomware is not binary security, but the information shown above highlights a probability that the code is not well-done.

6. Reversing

Open our sample on **IDA Pro** (my current version is 8.4 SP2). If we check for signatures (**View → Open Subviews → Signatures or SHIFT+F5**), we will not see anything there. If you want, you can right-click on the background and pick up **Apply new signature** to add signatures such as **elf64, libc** or even **go_std_abi0** (if it was a Golang file), but in this case you will not have success and no function signature will be added, unfortunately. Using the same approach, we can add **Type Libraries** (**SHIFT+F11**). There is already an added **Type Library (gnulnx_x64)**, and we do not have further useful libraries to include. Finally, and as usual, we must decompile the whole binary by going to **File → Produce File → Create C File**, where the IDA will suggest **mas_10.bin.c** as filename, and we can accept it.

The disassembly of the **main function** (not **start** function) is shown below:

```
.text:000000000040687F ; int __fastcall main(int, char **, char **)
.text:000000000040687F main            proc near               ; DATA XREF: start+1D↑o
.text:000000000040687F
.text:000000000040687F var_30          = qword ptr -30h
.text:000000000040687F var_24          = dword ptr -24h
.text:000000000040687F newthread       = qword ptr -18h
.text:000000000040687F var_10          = qword ptr -10h
.text:000000000040687F var_8           = qword ptr -8
.text:000000000040687F
.text:000000000040687F ; __unwind {
.text:000000000040687F                 push    rbp
.text:0000000000406880                 mov     rbp, rsp
.text:0000000000406883                 sub     rsp, 30h
.text:0000000000406887                 mov     [rbp+var_24], edi
.text:000000000040688A                 mov     [rbp+var_30], rsi
.text:000000000040688E                 mov     dword ptr [rbp+newthread], 0
.text:0000000000406895                 cmp     [rbp+var_24], 1
.text:0000000000406899                 jg      short loc_4068AF
.text:000000000040689B                 mov     edi, offset aRevix12aUsageE ; "Revix 1.2a \r\nUsage example: elf.exe -"...
.text:00000000004068A0                 call    _puts
.text:00000000004068A5                 mov     eax, 0
.text:00000000004068AA                 jmp     locret_4069D2
.text:00000000004068AF ; ---------------------------------------------------------------------------
.text:00000000004068AF
.text:00000000004068AF loc_4068AF:                             ; CODE XREF: main+1A↑j
.text:00000000004068AF                 mov     rdx, [rbp+var_30]
.text:00000000004068B3                 mov     eax, [rbp+var_24]
.text:00000000004068B6                 mov     rsi, rdx
.text:00000000004068B9                 mov     edi, eax
.text:00000000004068BB                 call    sub_4067EC
.text:00000000004068C0                 mov     [rbp+var_8], rax
.text:00000000004068C4                 mov     rax, [rbp+var_30]
.text:00000000004068C8                 mov     rax, [rax]
.text:00000000004068CB                 mov     [rbp+var_10], rax
.text:00000000004068CF                 cmp     [rbp+var_8], 0
.text:00000000004068D4                 jnz     short loc_4068EA
.text:00000000004068D6                 mov     edi, offset aUnableToFindPa ; "Unable to find path argument"
.text:00000000004068DB                 call    _puts
.text:00000000004068E0                 mov     eax, 0
.text:00000000004068E5                 jmp     locret_4069D2
.text:00000000004068EA ; ---------------------------------------------------------------------------
```

[Figure 24]: IDA disassembly window | main function

Once we have the binary entirely decompiled, when can use the pseudo-code representation of the binary and, when it is necessary, we check against the Assembly code to clear eventual doubts or misunderstandings. Anyway, one of most frequently questions about any malware analysis is how to initiate the analysis and proceed from the choosen piece of code. There are a few possibilities here:

- starting from beginning (**main** function).
- getting orientation by strings.
- getting orientation by functions.
- starting from important and known functions.
- checking for called cryptographic functions.
- checking for random parts of the code and, from there, investigating cross references.
- checking for functions that manipulate data from sessions such as **.rodata** or **.data** (check sections by using **CTRL+S** on IDA)

Before taking decisions, you should remember that questions that can be answered (all or part of them) are almost the same of any other malicious binary, with rare variantion and even purposes as, for example, a ransomware:

- communication method and C2
- cryptography algorithms
- persistence
- algorithm used
- targeted file types and/or folders.

In this article, our focus will be **cryptography algorithms**. A method to find important functions in the disassembly code is to use **Capa Explorer**. To install it on Linux, you have to follow the steps below:

- **python -m pip install -U flare-capa**
- **Download capa rules from** https://github.com/mandiant/capa-rules/releases
- **wget https://raw.githubusercontent.com/mandiant/capa/master/capa/ida/plugin/capa_explorer.py**
- **From the home directory, execute: mkdir .idapro/plugins**
- **cp capa_explorer.py .idapro/plugins/**

Close and open the IDA Pro again, load the respective **mas_10.idb file**, go to **Edit → Plugins → FLARE capa explorer** (**ALT+F5**) and indicate the extracted capa-rules directory. Once the analysis has been concluded, you will see something like:

[Figure 25]: FLARE Capa Explorer

Some obvious and few observations that can be made about the code:

- As signsrch tool, **Capa Explorer** have also identified several functions involved with cryptography.
- **AES** and **RC4** are the main symmetric algorithms used by this ransomware.
- **Base64 encoding** is also present, and it was expected according to what we have seen previously.
- For ransomware, operations such as reading, writing, moving, and deleting are completely usual.

However, the information offered by **Capa Explorer** is limited, and there are multiple parts of the code involved with cryptography that, eventually, deserves some light on.

The **sub_40E2AA routine** performs **Base64 encoding** and decoding, and there are some indications such as the explicit alphabet and constants. The piece of code below is the decoding part:

```
while ( 1 )
{
  ptr_char = arg_2_ptr_char1--;
  if ( !ptr_char || *(_BYTE *)(counter + a1) == '=' )
    break;
  if ( !ab_check_valid_Base64_char(*(_BYTE *)(counter + a1)) )
  {
    ++counter;
  }
  else
  {
    index = i++;
    counter_update = counter++;
    *(&target_buffer + index) = *(_BYTE *)(counter_update + a1);
    if ( i == 4 )
    {
      for ( i = 0; i <= 3; ++i )
      {
        ptr_base64 = (unsigned int)strchr(
                                     "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/",
                                     (unsigned __int8)*(&target_buffer + i));
        *(&target_buffer + i) = ptr_base64
                              - (_DWORD)"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
      }
      v13 = 4 * target_buffer + ((v17 & 0x30) >> 4);
      v14 = 16 * v17 + ((v18 & 0x3C) >> 2);
      v15 = (v18 << 6) + v19;
      for ( i = 0; i <= 2; ++i )
      {
        v6 = counter_2++;
        ptr_buffer[v6] = *(&v13 + i);
      }
      i = 0;
```

[Figure 25]: sub_40E2AA routine: it performs Base64 encoding and decoding

Another indication is the **sub_40E001 routine,** which presents the following very characteristic code related to Base64 character verification:

```
_BOOL8 __fastcall sub_40E001(unsigned __int8 a1)
{
  return ((*__ctype_b_loc())[a1] & 8) != 0 || a1 == '+' || a1 == '/';
}
```

[Figure 26]: sub_40E001 routine: it performs Base64 character verification

The **sub_401ED8 routine** is the setup of Salsa20's initial state (known as **load/store little-endian** in the **Daniel Bernstein implementation** and other ones -- the paper is here: https://cr.yp.to/snuffle/spec.pdf), which has, in very imprecise words, initialization, quarter round, column and row rounds operations, addition of the original state into the processed one, and keystream generation phases.

```
__int64 __fastcall sub_401ED8(__int64 a1, unsigned __int16 *a2)
{
  __int64 result; // rax

  *(_DWORD *)(a1 + 24) = (*((unsigned __int8 *)a2 + 3) << 24) | (*((unsigned __int8 *)a2 + 2) << 16) | *a2;
  *(_QWORD *)(a1 + 28) = (*((unsigned __int8 *)a2 + 7) << 24) | (*((unsigned __int8 *)a2 + 6) << 16) | (unsigned int)a2[2];
  result = a1;
  *(_DWORD *)(a1 + 36) = 0;
  return result;
}
```

[Figure 27]: Setup of Salsa20

The **sub_40173D routine** (shown below) performs the **Salsa20 byte manipulation** because it seems to be a **Quarter-Round function** due to the constants and, at the end of routine, there is the **double round part** (columns and rows). The constants are not equal to the reference (https://cr.yp.to/snuffle/salsafamily-20071225.pdf) because the code is using **__ROR4__ macro**, which is equivalent to return **__ROL__((uint32)value, -count)**. Additionally, readers can use this reference (https://hex-rays.com/blog/igors-tip-of-the-week-67-decompiler-helpers) from Hex-Rays about decompile helpers:

```
__int64 __fastcall sub_40173D(__int64 a1, __int64 a2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND

  for ( i = 0; i <= 15; ++i )
  {
    result = i;
    *(&v4 + i) = *(_DWORD *)(4LL * i + a2);
  }
  for ( i = 20; i > 0; i -= 2 )
  {
    v8 ^= __ROR4__(v4 + v16, 25);
    v12 ^= __ROR4__(v8 + v4, 23);
    v16 ^= __ROR4__(v12 + v8, 19);
    v4 ^= __ROR4__(v16 + v12, 14);
    v13 ^= __ROR4__(v9 + v5, 25);
    v17 ^= __ROR4__(v13 + v9, 23);
    v5 ^= __ROR4__(v17 + v13, 19);
    v9 ^= __ROR4__(v5 + v17, 14);
    v18 ^= __ROR4__(v14 + v10, 25);
    v6 ^= __ROR4__(v18 + v14, 23);
    v10 ^= __ROR4__(v6 + v18, 19);
    v14 ^= __ROR4__(v10 + v6, 14);
    v7 ^= __ROR4__(v19 + v15, 25);
    v11 ^= __ROR4__(v7 + v19, 23);
    v15 ^= __ROR4__(v11 + v7, 19);
    v19 ^= __ROR4__(v15 + v11, 14);
    v5 ^= __ROR4__(v4 + v7, 25);
    v6 ^= __ROR4__(v5 + v4, 23);
    v7 ^= __ROR4__(v6 + v5, 19);
    v4 ^= __ROR4__(v7 + v6, 14);
    v10 ^= __ROR4__(v9 + v8, 25);
    v11 ^= __ROR4__(v10 + v9, 23);
    v8 ^= __ROR4__(v11 + v10, 19);
    v9 ^= __ROR4__(v8 + v11, 14);
    v15 ^= __ROR4__(v14 + v13, 25);
```

[Figure 28]: sub_40173D: Salsa20 quarter-round and double-round operations

The **sub_401F94 routine** performs the remaining **Salsa20** operations:

```
__int64 __fastcall sub_401F94(__int64 a1, __int64 a2, __int64 a3, unsigned int a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  v5 = a4;
  if ( a4 )
  {
    while ( 1 )
    {
      sub_40173D((__int64)v8, a1);
      if ( !++*(_DWORD *)(a1 + 32) )
        ++*(_DWORD *)(a1 + 36);
      if ( v5 <= 0x40 )
        break;
      for ( i = 0; i <= 0x3F; ++i )
        *(_BYTE *)(a3 + i) = *(_BYTE *)(i + a2) ^ v8[i];
      v5 -= 64;
      a3 += 64LL;
      a2 += 64LL;
    }
    for ( i = 0; ; ++i )
    {
      result = i;
      if ( i >= v5 )
        break;
      *(_BYTE *)(a3 + i) = *(_BYTE *)(i + a2) ^ v8[i];
    }
  }
  return result;
}
```

[Figure 29]: Finalizing Salsa20 phases

The **sub_40CC2B routine** is responsible for hashing string using **CRC32**, where we can see polynomial **0xEDB88320** (https://en.wikipedia.org/wiki/Cyclic_redundancy_check#table). If reads want to read about **CRC32**, a good write up is https://github.com/Michaelangel007/crc32. The routine code is shown below:

```
__int64 __fastcall ab_crc32(int a1, unsigned __int8 *a2, __int64 a3)
{
  unsigned __int8 *v3; // rax
  unsigned __int64 i; // [rsp+20h] [rbp-18h]
  unsigned int v9; // [rsp+34h] [rbp-4h]

  v9 = ~a1;
  while ( a3-- )
  {
    v3 = a2++;
    v9 ^= *v3;
    for ( i = 0LL; i <= 7; ++i )
      v9 = (v9 >> 1) ^ -(v9 & 1) & 0xEDB88320;
  }
  return ~v9;
}
```

[Figure 30]: CRC32 hashing

The **sub_40E7F5 routine** prepares for the **AES** encryption, and the code is related to the initialization phase (setup key), according to the key size, which represents the setup phase:

```
13  v4 = a1;
14  v9 = 0;
15  *a1 = a2[3] ^ (a2[2] << 8) ^ (a2[1] << 16) ^ (*a2 << 24);
16  a1[1] = (a2[6] << 8) ^ (a2[5] << 16) ^ (a2[4] << 24) ^ a2[7];
17  a1[2] = (a2[10] << 8) ^ (a2[9] << 16) ^ (a2[8] << 24) ^ a2[11];
18  a1[3] = (a2[14] << 8) ^ (a2[13] << 16) ^ (a2[12] << 24) ^ a2[15];
19  if ( a3 == 128 )
20  {
21    while ( 1 )
22    {
23      v5 = v4[3];
24      v4[4] = (unsigned __int8)dword_4122C0[HIBYTE(v5)] ^ dword_4122C0[(un
25      v4[5] = v4[1] ^ v4[4];
26      v4[6] = v4[2] ^ v4[5];
27      v4[7] = v4[3] ^ v4[6];
28      if ( ++v9 == 10 )
29        break;
30      v4 += 4;
31    }
32    return 10LL;
33  }
34  else
35  {
36    a1[4] = (a2[18] << 8) ^ (a2[17] << 16) ^ (a2[16] << 24) ^ a2[19];
37    a1[5] = (a2[22] << 8) ^ (a2[21] << 16) ^ (a2[20] << 24) ^ a2[23];
38    if ( a3 == 192 )
```

[Figure 31]: AES setup (cypher key expansion)

The **sub_40F370** performs AES encryption itself through key expansion and table-base implementations, which tables are **dword_411EC0**, **dword_411AC0**, **dword_4116C0** and **dword_4112C0**, as shown below:

```
1  __int64 __fastcall sub_40F370(_DWORD *a1, int a2, unsigned __int8 *a3, _BYTE *a4)
2  {
3    // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5    v5 = a1;
6    v17 = a3[3] ^ (a3[2] << 8) ^ (a3[1] << 16) ^ (*a3 << 24) ^ *a1;
7    v15 = a3[7] ^ (a3[6] << 8) ^ (a3[5] << 16) ^ (a3[4] << 24) ^ a1[1];
8    v13 = a3[11] ^ (a3[10] << 8) ^ (a3[9] << 16) ^ (a3[8] << 24) ^ a1[2];
9    v11 = a3[15] ^ (a3[14] << 8) ^ (a3[13] << 16) ^ (a3[12] << 24) ^ a1[3];
10   v10 = a2 >> 1;
11   while ( 1 )
12   {
13     v9 = dword_411EC0[(unsigned __int8)v11] ^ dword_411AC0[BYTE1(v13)] ^ dword_4116C0[BYTE2(v15)] ^ dword_4112C0[HIBYTE(v17)] ^ v5[4];
14     v8 = dword_411EC0[(unsigned __int8)v17] ^ dword_411AC0[BYTE1(v11)] ^ dword_4116C0[BYTE2(v13)] ^ dword_4112C0[HIBYTE(v15)] ^ v5[5];
15     v7 = dword_411EC0[(unsigned __int8)v15] ^ dword_411AC0[BYTE1(v17)] ^ dword_4116C0[BYTE2(v11)] ^ dword_4112C0[HIBYTE(v13)] ^ v5[6];
16     v6 = dword_411EC0[(unsigned __int8)v13] ^ dword_411AC0[BYTE1(v15)] ^ dword_4116C0[BYTE2(v17)] ^ dword_4112C0[HIBYTE(v11)] ^ v5[7];
17     v5 += 8;
18     if ( !--v10 )
19       break;
20     v17 = dword_411EC0[(unsigned __int8)v6] ^ dword_411AC0[BYTE1(v7)] ^ dword_4116C0[BYTE2(v8)] ^ dword_4112C0[HIBYTE(v9)] ^ *v5;
21     v15 = dword_411EC0[(unsigned __int8)v9] ^ dword_411AC0[BYTE1(v6)] ^ dword_4116C0[BYTE2(v7)] ^ dword_4112C0[HIBYTE(v8)] ^ v5[1];
22     v13 = dword_411EC0[(unsigned __int8)v8] ^ dword_411AC0[BYTE1(v9)] ^ dword_4116C0[BYTE2(v6)] ^ dword_4112C0[HIBYTE(v7)] ^ v5[2];
23     v11 = dword_411EC0[(unsigned __int8)v7] ^ dword_411AC0[BYTE1(v8)] ^ dword_4116C0[BYTE2(v9)] ^ dword_4112C0[HIBYTE(v6)] ^ v5[3];
24   }
```

[Figure 32]: AES encryption (table-base implementations)

The **sub_40EEE9** routine prepares and sets the AES key for decryption. At the same form, the **sub_40FA4C** routine is responsible for AES decryption itself. As the entire process occurs in an equivalent way to the encryption phase, I am not going to show it here.

The **sub_405801 routine** seems to be related to **curve2519-donna**. The **curve25519** is an elliptic curve (used with **ECDH -- Elliptic-curve Diffie-Hellman**) developed by **Dan Bernstein**, and **Adam Langley (@agl__)** wrote the **donna version**. Further information can be found here:

- **Original code by Adam Langley:** https://github.com/agl/curve25519-donna

- **File used to identify the algorithm:** https://github.com/agl/curve25519-donna/blob/master/curve25519-donna.c
- **Wikipedia:** https://en.wikipedia.org/wiki/Curve25519

Within **sub_405801 routine**, I have found **sub_403792**, which is responsible for the reduced coefficient form for the input, and that was my hint on the elliptic curve mentioned:

```
_QWORD *__fastcall ab_ec25519_reduced_coefficient_form(_QWORD *a1, _QWORD *a2)
{
  _QWORD *result; // rax

  *a1 = (int)*a2 * (__int64)(int)*a2;
  a1[1] = 2LL * (int)*a2 * (int)a2[1];
  a1[2] = 2
        * ((int)a2[1] * (__int64)(int)a2[1] + (int)*a2 * (__int64)(int)a2[2]);
  a1[3] = 2
        * ((int)a2[2] * (__int64)(int)a2[1] + (int)*a2 * (__int64)(int)a2[3]);
  a1[4] = 4LL * (int)a2[1] * (int)a2[3]
        + (int)a2[2] * (__int64)(int)a2[2]
        + 2LL * (int)*a2 * (int)a2[4];
  a1[5] = 2
        * ((int)a2[3] * (__int64)(int)a2[2]
         + (int)a2[1] * (__int64)(int)a2[4]
         + (int)*a2 * (__int64)(int)a2[5]);
  a1[6] = 2
        * ((int)a2[3] * (__int64)(int)a2[3]
         + (int)a2[2] * (__int64)(int)a2[4]
         + (int)*a2 * (__int64)(int)a2[6]
         + 2LL * (int)a2[1] * (int)a2[5]);
  a1[7] = 2
        * ((int)a2[1] * (__int64)(int)a2[6]
         + (int)a2[4] * (__int64)(int)a2[3]
         + (int)a2[2] * (__int64)(int)a2[5]
         + (int)*a2 * (__int64)(int)a2[7]);
  a1[8] = (int)a2[4] * (__int64)(int)a2[4]
        + 2
        * ((int)a2[6] * (__int64)(int)a2[2]
         + (int)*a2 * (__int64)(int)a2[8]
         + 2
         * ((int)a2[7] * (__int64)(int)a2[1] + (int)a2[3] * (__int64)(int)a2[5]));
```

[Figure 33]: curve25519-donna: coefficient reduction

The **sub_40C6E4 routine** does not seem very meaningful, as shown below:

```
__int64 __fastcall sub_40C6E4(_BYTE *a1)
{
  if ( !(unsigned int)sub_40C62D(a1, 0x20u) )
    return 0LL;
  *a1 &= 0xF8u;
  a1[31] &= ~0x80u;
  a1[31] |= 0x40u;
  return 1LL;
}
```

[Figure 34]: sub_40C6E4 routine

However, the **sub_40C62D routine**, which is called by **sub_40C6E4 routine**, shows the there is a generation of a random value, which helps the **sub_40C6E4 routine** to produce a 32 bytes random value:

```
__int64 __fastcall sub_40C62D(void *a1, unsigned __int16 a2)
{
  int fd; // [rsp+14h] [rbp-Ch]
  unsigned __int64 v4; // [rsp+18h] [rbp-8h]

  fd = open("/dev/urandom", 0);
  if ( fd == -1 )
  {
    printf("Error open urandm line %d!\n", 30LL);
  }
  else
  {
    v4 = read(fd, a1, a2);
    if ( v4 == -1LL )
    {
      printf("Error read urandm line %d!\n", 38LL);
    }
    else if ( a2 > v4 )
    {
      printf("rand: try to read %hu but get %lu bytes\n", a2, v4);
    }
  }
  close(fd);
  return 1LL;
}
```

[Figure 35]: sub_40C6E4 routine

The **sub_408C7A routine** represents the **Keccak** algorithm, which is the base for SHA-3.

```
    v7 += 5;
    v10[v7 + v8] ^= *(&v2 + (v8 + 4) % 5) ^ __ROR8__(*(&v2 + (v8 + 1) % 5), 63);
    v7 += 5;
    v10[v7 + v8] ^= *(&v2 + (v8 + 4) % 5) ^ __ROR8__(*(&v2 + (v8 + 1) % 5), 63);
    v7 += 5;
    v10[v7 + v8] ^= *(&v2 + (v8 + 4) % 5) ^ __ROR8__(*(&v2 + (v8 + 1) % 5), 63);
    v7 += 5;
    v10[v7 + v8] ^= *(&v2 + (v8 + 4) % 5) ^ __ROR8__(*(&v2 + (v8 + 1) % 5), 63);
    ++v8;
    v7 = 0;
    v10[v8] ^= *(&v2 + (v8 + 4) % 5) ^ __ROR8__(*(&v2 + (v8 + 1) % 5), 63);
    v7 += 5;
    v10[v7 + v8] ^= *(&v2 + (v8 + 4) % 5) ^ __ROR8__(*(&v2 + (v8 + 1) % 5), 63);
    v7 += 5;
    v10[v7 + v8] ^= *(&v2 + (v8 + 4) % 5) ^ __ROR8__(*(&v2 + (v8 + 1) % 5), 63);
    v7 += 5;
    v10[v7 + v8] ^= *(&v2 + (v8 + 4) % 5) ^ __ROR8__(*(&v2 + (v8 + 1) % 5), 63);
    v7 += 5;
    v10[v7 + v8] ^= *(&v2 + (v8 + 4) % 5) ^ __ROR8__(*(&v2 + (v8 + 1) % 5), 63);
    ++v8;
    v7 = 0;
    v10[v8] ^= *(&v2 + (v8 + 4) % 5) ^ __ROR8__(*(&v2 + (v8 + 1) % 5), 63);
```

[Figure 36]: sub_408C7A routine: Keccak

The **sub_40E7F5 routine** sets the **AES** key for encryption (128, 192 and 256 bits), and the prototype and respective arguments follow:

```
__int64 __fastcall sub_40E7F5(
      int *a1,
      unsigned __int8 *a2,
      int a3)
{
```

- **a1: rk** | round key (obviously, it depends on the number of rounds | 4 * (Nr + 1)
- **a2: cipherKey** | input cypher key
- **a3: the length of the key** (128, 192, 256)

The routine is given as:

```
14  if ( a3 == 128 )
15  {
16    while ( 1 )
17    {
18      v5 = v4[3];
19      v4[4] = (unsigned __int8)dword_4122C0[HIBYTE(v5)] ^ dword_4122C0[(unsigned __int8)v5] & 0xFF00 ^ d
20      v4[5] = v4[1] ^ v4[4];
21      v4[6] = v4[2] ^ v4[5];
22      v4[7] = v4[3] ^ v4[6];
23      if ( ++v9 == 10 )
24        break;
25      v4 += 4;
26    }
27    return 10LL;
28  }
29  else
30  {
31    a1[4] = (a2[18] << 8) ^ (a2[17] << 16) ^ (a2[16] << 24) ^ a2[19];
32    a1[5] = (a2[22] << 8) ^ (a2[21] << 16) ^ (a2[20] << 24) ^ a2[23];
33    if ( a3 == 192 )
34    {
35      while ( 1 )
36      {
37        v6 = v4[5];
38        v4[6] = (unsigned __int8)dword_4122C0[HIBYTE(v6)] ^ dword_4122C0[(unsigned __int8)v6] & 0xFF00 ^
39        v4[7] = v4[1] ^ v4[6];
40        v4[8] = v4[2] ^ v4[7];
41        v4[9] = v4[3] ^ v4[8];
42        if ( ++v9 == 8 )
43          break;
44        v4[10] = v4[4] ^ v4[9];
45        v4[11] = v4[5] ^ v4[10];
46        v4 += 6;
47      }
```

[Figure 37]: sub_40E7F5 routine

The **sub_40C810 routine** comes up after we have quickly (actually, very quickly) analyzed previous routines, and, in general words, we see that it:

- Check if any data is provided to encrypt.
- **Curve25519 donna** and Keccak provides a better, efficient, and more secure method for key exchange. Thus, in this case, the asymmetric algorithm (**curve25519 donna**) is encrypting the symmetric key (AES) for exchange.
- Given the session key, the data is encrypted

- A **CRC32** hash of encrypted data is calculated and appended to the final encrypted data.

The piece of code putting all routines together is shown below:

```
unsigned __int8 *__fastcall sub_40C810(
        unsigned __int16 *input_data,
        __int64 cypher_key,
        unsigned int data_len,
        unsigned int *a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  *a4 = 0;
  if ( !data_len )
    return 0LL;
  v12 = data_len + 56;
  encrypted_data = (unsigned __int8 *)ab_memset_zero(data_len + 56);
  if ( !encrypted_data )
    return 0LL;
  *(_DWORD *)encrypted_data = 0;
  ab_copy_bytes((__int64)(encrypted_data + 4), cypher_key, data_len);
  ab_ww_curve2519_donna(v8, (__int64)v9);
  ab_curve2519_donna_keccak((__int64)v8, input_data, (__int64)cypher_key_1);
  ab_zeroed_array((__int64)v8, 0x20uLL);
  ab_generate_random(crc32_hash, 0x10u);
  ab_AES_data_encryption(
    cypher_key_1,
    256,
    (__int64)crc32_hash,
    (__int64)encrypted_data,
    data_len + 4);
  ab_zeroed_array((__int64)cypher_key_1, 0x20uLL);
  crc32_hash[4] = ab_CRC32_calculation(0, encrypted_data, data_len + 4LL);
  ab_copy_bytes((__int64)&encrypted_data[data_len + 4], (__int64)v9, 0x34u);
  *a4 = v12;
  return encrypted_data;
}
```

[Figure 38]: sub_40C810 routine

From this point onward all routines are simpler than those ones involving cryptographic operations. The first one is **sub_405913 routine**, but first we need to learn the value associated to **off_616228, format** and **off_616238**, which are shown below:

```
aExpand32ByteKe db 'expand 32-byte kexpand 16-byte kvmx-*',0
                                        ; DATA XREF: sub_401AD3+154↑o
aPkill9S db 'pkill -9 %s',0             ; DATA XREF: .data:format↓o
    align 8
aEsxcliFormatte db 'esxcli --formatter=csv --format-param=fields=="WorldID,DisplayNam'
                                        ; DATA XREF: .data:off_616238↓o
    db 'e" vm process list | awk -F "\"*,\"*" ',27h,'{system("esxcli vm p'
    db 'rocess kill --type=force --world-id=" $1)}',27h,0
```

[Figure 39]: sub_40C810 routine

Even though that the representation is not so clear, readers can realize that the line aims to kill any instance of ESXi. As expected, the **sub_405913 routine** use all of these elements to accomplish the objective of killing instances of the ESXi:

```
unsigned __int64 __fastcall sub_405913(void *a1)
{
  unsigned __int64 result; // rax
  FILE *v2; // [rsp+18h] [rbp-18h]
  FILE *stream; // [rsp+20h] [rbp-10h]
  int i; // [rsp+2Ch] [rbp-4h]

  do
  {
    for ( i = 0; !i; i = 1 )
    {
      sprintf(command, format, off_616228);
      stream = popen(command, "r");
      pclose(stream);
    }
    v2 = popen(off_616238, "r");
    pclose(v2);
    sleep(1u);
    result = (unsigned int)dword_61A320;
  }
  while ( !dword_61A320 );
  return result;
}
```

[Figure 40]: sub_405913 routine | killing ESXi instances

The **sub_405ED8 routine** starts the operation of file encryption , as shown below:

```
__int64 __fastcall sub_405ED8(const char *a1)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  v14 = (int *)sub_405BDC();
  v21 = 0;
  v1 = strlen(a1);
  newa = (char *)ab_memset_zero(v1 + 138);
  sprintf(newa, "%s%s", a1, needle);
  v2 = strlen(a1);
  filename = (char *)ab_memset_zero(v2 + 138);
  sprintf(filename, "%s.lck", a1);
  if ( sub_405E9E(filename) )
  {
    printf("%s is already in progress of ecnryption...\n", a1);
    return 0LL;
  }
  else
  {
    stream = fopen(filename, "w+");
    fclose(stream);
    v8 = fopen(a1, "rb+");
    if ( v8 )
    {
      v14[74] = 0x100000;
      ptr = ab_memset_zero(v14[74]);
      v6 = ab_memset_zero(v14[74]);
      if ( sub_405D96(v8) )
      {
        printf("File is encrypted by data %s\n", a1);
      }
```

[Figure 41]: sub_405ED8 routine | file encryption

The **sub_40CF44 routine** generates a JSON file, which probably contains information about the target system:

```
1  __int64 sub_40CF44()
2  {
3    // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5    s = (char *)ab_memset_zero(8196);
6    sprintf(
7      s,
8      "{\"ver\":%d,\"pid\":\"%s\",\"sub\":\"%s\",\"pk\":\"%s\",\"uid\":\"%s\",\"sk\""
9      ":\"%s\",\"os\":\"%s\",\"ext\":\"%s\"}",
10     512LL,
11     qword_71B590,
12     qword_71B598,
13     (const char *)qword_71B5B8,
14     (const char *)qword_71B5C0,
15     (const char *)qword_71B5C8,
16     (const char *)qword_71B5D8,
17     needle + 1);
18   v6 = strlen(s);
19   ptr = sub_40C810(word_61A2C0, (__int64)s, v6, &v3);
20   free(s);
21   v4 = sub_40E04A((char *)ptr, v3, &v2);
22   free(ptr);
23   v1[0] = (__int64)"{UID}";
24   v1[1] = qword_71B5C0;
25   v1[2] = (__int64)"{KEY}";
26   v1[3] = (__int64)v4;
27   v1[4] = (__int64)"{EXT}";
28   v1[5] = (__int64)(needle + 1);
29   ::ptr = sub_40E704(::ptr, (__int64)v1, 3u);
30   LODWORD(size) = strlen(::ptr);
31   free(v4);
32   return 1LL;
33 }
```

[Figure 42]: sub_40CF44 routine | JSON file generation

The **sub_40D53F routine** loads the JSON file:

```
1  void *sub_40D53F()
2  {
3    __int64 n; // [rsp+8h] [rbp-18h]
4    FILE *stream; // [rsp+10h] [rbp-10h]
5    void *ptr; // [rsp+18h] [rbp-8h]
6
7    stream = fopen("json.txt", "rb");
8    if ( !stream )
9    {
10     puts("Error no json file!");
11     exit(0);
12   }
```

[Figure 43]: sub_40D53F routine | loads the JSON file

The **sub_4064FD routine** parses each directory, encrypt its respective files and drops a ransomware note (**sub_4059B3 routine**). Additionally, note that the routine is recursive (line 30):

```
1  void __fastcall sub_4064FD(char *a1)
2  {
3    // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
4
5    dirp = opendir(a1);
6    if ( dirp )
7    {
8      memset(s1, 0, sizeof(s1));
9      while ( 1 )
10     {
11       v2 = readdir(dirp);
12       if ( !v2 )
13         break;
14       snprintf(s1, 0x800uLL, "%s/%s", a1, v2->d_name);
15       if ( strcmp(v2->d_name, ".")
16         && strcmp(v2->d_name, "..")
17         && strcmp(v2->d_name, src)
18         && v2->d_name[0] )
19       {
20         if ( v2->d_type == 8 )
21         {
22           ab_mutex_operations(qword_71B480, (__int64)ab_encrypt_file, s1);
23         }
24         else if ( v2->d_type == 4 && strcmp(s1, "//dev") )
25         {
26           if ( strcmp(s1, "/dev") )
27           {
28             strcpy(dest, s1);
29             v1 = sub_40DC0F(dest);
30             sub_4064FD(v1);
31           }
32         }
33       }
34     }
35     sub_4059B3(a1);
```

[Figure 44]: sub_4064FD routine | encrypt files within a directory

7. Conclusion

In this article we have reviewed a few basic concepts of analyzing an ELF binary as also we perform a quick and superficial analysis of a ransomware sample.

Finally, I have accomplished my promise in producing a ten-articles series! This was the **last article** of the **Malware Analysis Series (MAS)**, and I hope you have enjoyed reading all articles over the years. As you already know, I moved to another area (vulnerability research) a bit more than a couple of years ago, and now **I really do something I have passion to do**. Therefore, that is my advice. **Follow your heart**.

Just in case you want to stay connected:

- **Twitter: @ale_sp_brazil**
- **Blog:** https://exploitreversing.com

Keep learning, reversing, and exploiting everything, and I will see you next time!

Alexandre Borges