CSE 406 Presentation Report : Malware analysis by YARA

[1] Alina Zaman : 1905099
[2] Saha Kuljit Shantanu : 1905119

March 9, 2024

1 Introduction

YARA is a powerful tool that has been fascinating malware analysts. The purpose of YARA encompasses the detection of presence of evil in malware by pattern or string matching. After complete analysis, the malware can be classified using the YARA tool.

2 Getting Started

The installation of YARA in Ubuntu Linux is quiet a short and convenient process. Here it is step by step

- Navigating to this link, the first instruction we get is to install the source code tarball file from https://github.com/VirusTotal/yara/releases. The latest version is **4.5.0**

- Now that the source-code.tar.gz is installed, it is time to compile it, hence we run the following commands in order, to unzip :

```
$ tar -zxf yara-4.5.0.tar.gz
$ cd yara-4.5.0
```

- Now we have to install all the dependencies before compilation :

```
$ sudo apt-get install automake libtool make gcc pkg-config
$ sudo apt-get install flex bison
```

- Now it is time to start the compilation of YARA tool :

```
$ ./bootstrap.sh
```

- Now it is necessary to install the openSSL library and configure the tool as the tool cannot be configured without openSSL library :

```
$ sudo apt-get install libssl-dev
$ ./configure
$ make
$ sudo make install
```

- Now that the tool is compiled and configured, we have to test if all packages are installed

```
$ make check
```

- Now we need to test the tool by executing our yara rule :

```
$ yara <yara-rule-file> <test-file-or-folder>
```

  If yara-rule is executed on a folder, it executes the rule for all file inside the directory recursively

  To print elaborately on every line of every file where the rules match on the conditions :

```
$ yara -s -r <yara-rule-file> <test-file-or-folder>
```

3 Opening Malware

The malware we were supplied for analysis by YARA tool was ”libxselinux.so” which is a library file that has powerful properties to make system calls when any other c program includes it and executes itself.

Executing the malware in local machine is a risk factor, hence it is not feasible to execute the file. However, the malware can be opened in hexadecimal format and assembly format.

We can open the malware ”libxselinux.so” into some file ”libxselinux.txt” in hexadecimal format through the following command:

```
$ xxd libxselinux.so > libxselinux.txt
```

We can open the malware ”libxselinux.so” into some file ”libxselinux.asm” in x86 assembly encoding format through the following command:

```
$ objdump -d -S libxselinux.so > libxselinux.asm
```

4 Identifying the file type of malware

In YARA, we can easily identify a file by a hidden specific property in the hex file. If the file is an **exe** file, it will have identifying tokens ’MZ’ at the starting of the file :

```

rule test_IS_EXE_FILE {

    meta :

        description = "Let's see if the file is exe"
        author = "Saha Kuljit Shantanu"
        date = "05-03-2024"

    strings :

        $exe_sign = { 4D 5A } //contains MZ


    condition:

        $exe_sign at 0

}

```

If the file is an **exe** file, it will have identifying tokens 'ELF' following any character at the starting of the file :

```

rule test_IS_OUT_FILE {

    meta :

        description = "Let's see if the file is out file"
        author = "Saha Kuljit Shantanu"
        date = "05-03-2024"

    strings :

        $out_sign = { ?? 45 4C 46 } //contains .ELF


    condition:

        $out_sign at 0

}

```

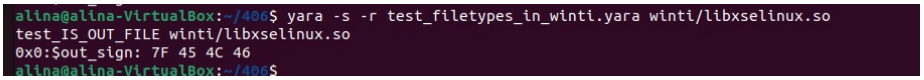
We can also verify if the file is a deb file by the following rule:

```

rule test_IS_DEB_FILE {

    meta :

        description = "Let's see if the file is deb file"
        author = "Saha Kuljit Shantanu"
        date = "05-03-2024"

    strings :

        $deb_sign = { 21 3C 61 72 63 68 3E }


    condition:

        $deb_sign at 0

}

```

The malware may be a zip file with an outfile inside it, that executes on unzipping. For verification we check the following rule :

```

rule test_IS_OUT_FILE_IN_ZIP {

    meta :

        description = "Let's see if the file is safe to unzip"
        author = "Saha Kuljit Shantanu"
        date = "05-03-2024"

    strings :

        $zip_sign = "PK\x03\x04"
        $out_sign = ".out"


    condition:

        $zip_sign at 0 and any of them

}

```

When this property is tested on our malware, we get the output as in the figure



```
alina@alina-VirtualBox:~/406$ yara -s -r test_filetypes_in_winti.yara winti/libxselinux.so
test_IS_OUT_FILE winti/libxselinux.so
0x0:$out_sign: 7F 45 4C 46
alina@alina-VirtualBox:~/406$
```

Figure 1: Classifying the file of malware

5 Matching pattern to detect evil on malware

5.1 Matching regular string

By the following string matching rule, we can manually track the presence of keywords **NOP**, **GNU** or **SOCKET** in the malware to estimate if the malware fills the memory with NOPs, or compiles a c code or, opens a socket connection respectively :

```

//Normal string matching

rule test_NOP_GNU_SOCKET {

    meta :

        description = "Let's see if the file has nops, or can compile with gnu or can open a socket"
        author = "Saha Kuljit Shantanu"
        date = "05-03-2024"

    strings :

        $nop = "NOP" nocase
        $gnu = "GNU" nocase
        $socket = "SOCKET" nocase


    condition:

        $nop or $gnu or $socket


}

```

5.2 Matching regular expressions

By the following regular expression matching rule, we can manually track the malware to estimate whether it can take input from a program and give output to console :

```

//Matching with regular expressions

rule test_SCAN_PRINT {

    meta :

        description = "Let's see if the file has scanning and printing properties"
        author = "Saha Kuljit Shantanu"
        date = "05-03-2024"

    strings :


        $scan = /[a-z]{0,}scanf[a-z]{0,}/
        $print = /[a-z]{0,}printf[a-z]{0,}/
        $get = /[a-z]{0,}get[a-z]{0,}/
        $put = /[a-z]{0,}put[a-z]{0,}/


    condition:

        $scan or $print or $get or $put


}


```

5.3 Matching Hexadecimal pattern

```

//Matching with hex strings

rule is_FILE_OUT {

    meta :

        description = "Let's see if the file is an out file"
        author = "Saha Kuljit Shantanu"
        date = "05-03-2024"

    strings :

        $is_out = { ?? 45 4C 46 }

    condition:

        $is_out


}

```

6 Checking conditions to analyse malware

6.1 Detecting system calls

By the following rule, we can detect if the malware is susceptible to make system calls :

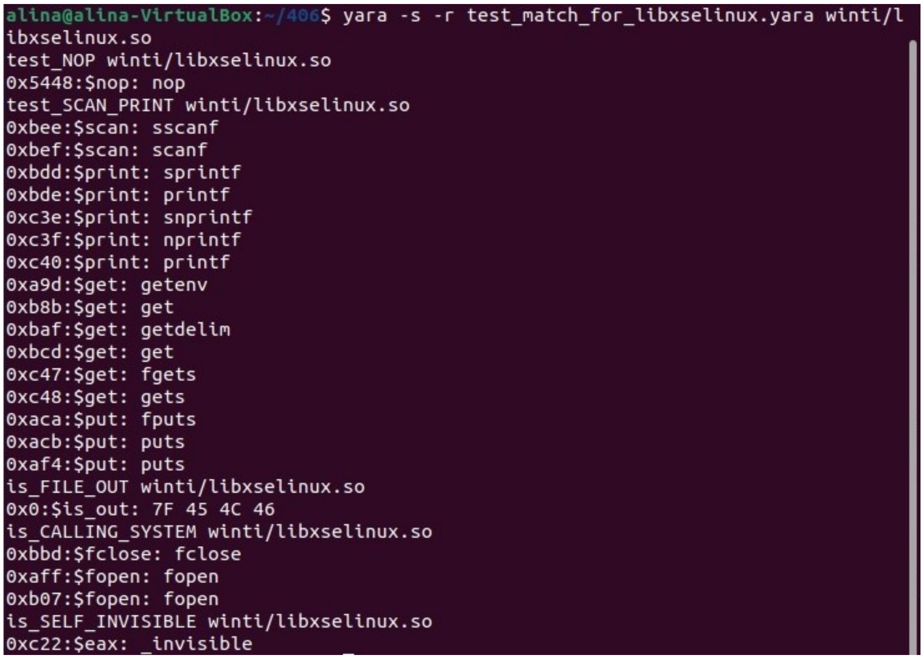
```

//Detecting system call

rule is_CALLING_SYSTEM {

    meta :

        description = "Let's see if the file is making system calls"
        author = "Saha Kuljit Shantanu"
        date = "05-03-2024"

    strings :

        $fclose = "fclose"
        $fopen = "fopen"
        $sleep = "sleep"

    condition:

        2 of them


}


```

6.2 Detecting visibility of malware

By the following rule, we can detect if the malware is susceptible to hide its existence :

```

//Matching malware invisibility

rule is_SELF_INVISIBLE {

    meta :

        description = "Let's see if the malware is hiding itself"
        author = "Saha Kuljit Shantanu"
        date = "05-03-2024"

    strings :

        $eax = "_invisible" nocase

    condition:

        $eax


}


```

The output to the evil detecting rules on our malware yields the following result



```
alina@alina-VirtualBox:~/406$ yara -s -r test_match_for_libxselinux.yara winti/l
ibxselinux.so
test_NOP winti/libxselinux.so
0x5448:$nop: nop
test_SCAN_PRINT winti/libxselinux.so
0xbee:$scan: sscanf
0xbef:$scan: scanf
0xbdd:$print: sprintf
0xbde:$print: printf
0xc3e:$print: snprintf
0xc3f:$print: nprintf
0xc40:$print: printf
0xa9d:$get: getenv
0xb8b:$get: get
0xbaf:$get: getdelim
0xbcd:$get: get
0xc47:$get: fgets
0xc48:$get: gets
0xaca:$put: fputs
0xacb:$put: puts
0xaf4:$put: puts
is_FILE_OUT winti/libxselinux.so
0x0:$is_out: 7F 45 4C 46
is_CALLING_SYSTEM winti/libxselinux.so
0xbbd:$fclose: fclose
0xaff:$fopen: fopen
0xb07:$fopen: fopen
is_SELF_INVISIBLE winti/libxselinux.so
0xc22:$eax: _invisible
```

Figure 2: Detecting evil from the malware

7 Modules

7.1 PE and ELF

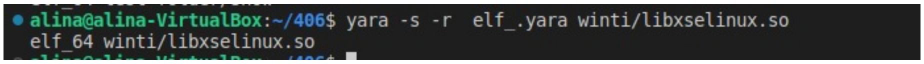
The PE module includes a **is_pe** variable that verifies whether the malware is an executable file.

Moreover, the PE module includes both **DLL** and **characteristics** variables that verify whether the malware is a dynamic library file. As a matter of fact, every executable file links to a dynamic library, hence a dynamic library file may not be executable but an executable file is always dynamic link library.

The ELF module includes a **machine** variable that keeps the properties of machine code and assembly code of the malware and verifies whether the malware is an output file. In our case, the assembly is X86 and the machine code is 64-bit. Hence we choose to verify **EM_X86_64** which means output file with X86 assembly coding and 64-bit machine language.

```
import "pe"
import "elf"


rule elf_64
{
    condition:
        elf.machine==elf.EM_X86_64
}

rule is_dll
{
    condition:
        pe.characteristics & pe.DLL
}

rule is_pe
{
    condition:
        pe.is_pe
}
```

The output when the rule is applied to our malware is like the following



```
alina@alina-VirtualBox:~/406$ yara -s -r  elf_.yara winti/libxselinux.so
elf_64 winti/libxselinux.so
```

Figure 3: Detecting type of the malware by PE and ELF module

This output indicates that the malware is neither executable, nor a dynamically linked library. Rather the malware is an output file, which we have verified earlier in the classic process as well, with output 1

7.2 Console, Math and Time

The math module includes the following functions:

- **entropy:** entropy is the measure of how random all the bytes are with respect to each other in a file. The actions of a malware with a large entropy is really hard to predict as different bytes are placed at random positions. On the other hand, actions of a malware with lesser entropy is easier to predict. In fact, it is comparatively easy to estimate the contents of a malware with lower entropy. A malware with entropy higher than 7 is considered to be a strong malware and a malware with entropy less than 7 is considered to be a weak malware.

  The entropy of libxselinux.so is **4.764744**. Hence this malware is weak.

- **mode:** mode method returns the byte that is present in a file maximum number of times. The mode byte of libxselinux.so is 00

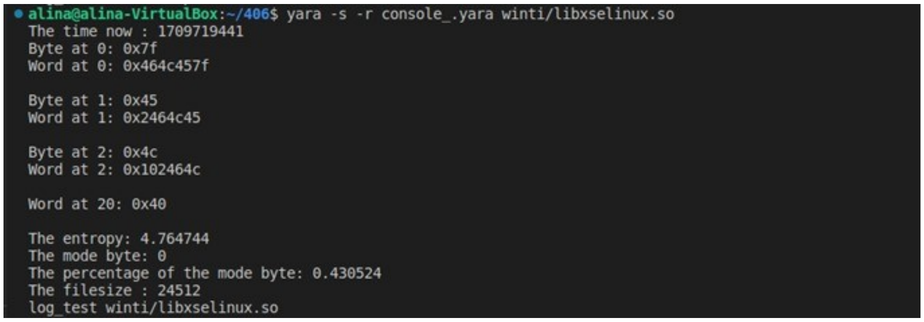
- **percentage:** percentage of a byte is the ratio of the number of that byte within the content to the number of byte in the constant times 100. The percentage of the mode byte(00) in libxselinux.so is 43

The time module includes a **now** function that prints the point in current timestamp in seconds.

The console module can print in console :

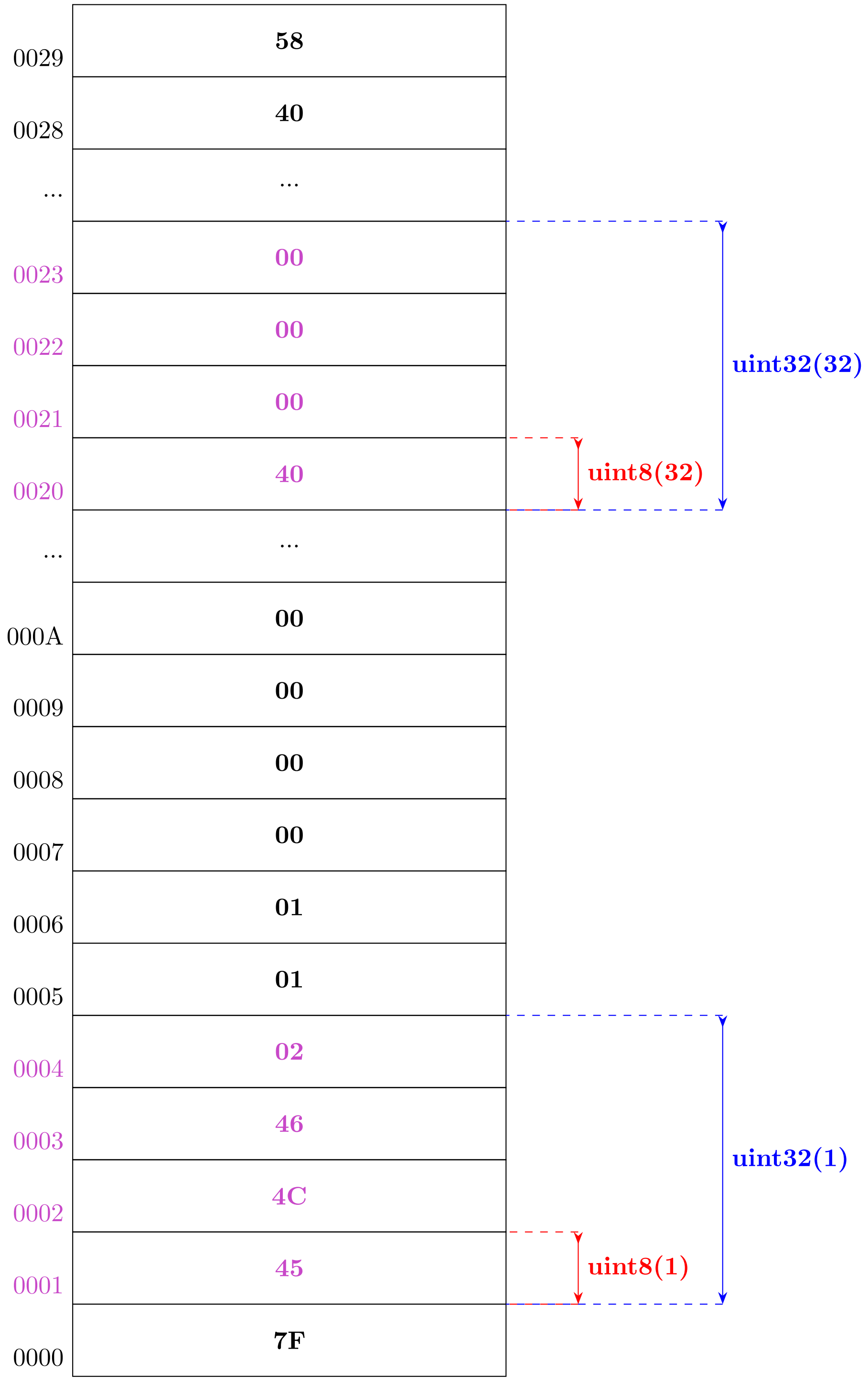
```

import "math"
import "console"
import "time"

rule log_test{
    // output strings !!!
    condition:

        console.log("The time now : ", time.now())
        and console.hex("Byte at 0: ", uint8(0))
        and console.hex("Word at 0: ", uint32(0))
        and console.log("")
        and console.hex("Byte at 1: ", uint8(1))
        and console.hex("Word at 1: ", uint32(1))
        and console.log("")
        and console.hex("Byte at 2: ", uint8(2))
        and console.hex("Word at 2: ", uint32(2))
        and console.log("")
        and console.hex("Word at 20: ", uint8(32))
        and console.log("")
        and console.log("The entropy: ", math.entropy( 0, filesize ))
        and console.log("The mode byte: ", math.mode( 0, filesize ))
        and console.log("The percentage of the mode byte: ",
        ↪  math.percentage( math.mode( 0, filesize), 0, filesize ))
        and console.log("The filesize : ", filesize)

}

```

The output when the rule is applied to our malware is like the following



```
alina@alina-VirtualBox:~/406$ yara -s -r console_.yara winti/libxselinux.so
The time now : 1709719441
Byte at 0: 0x7f
Word at 0: 0x464c457f

Byte at 1: 0x45
Word at 1: 0x2464c45

Byte at 2: 0x4c
Word at 2: 0x102464c

Word at 20: 0x40

The entropy: 4.764744
The mode byte: 0
The percentage of the mode byte: 0.430524
The filesize : 24512
log_test winti/libxselinux.so
```

Figure 4: Printing in console by Console module

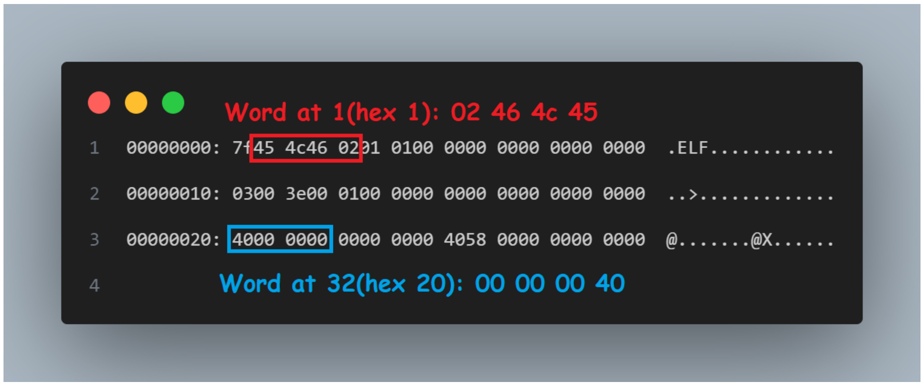


0029	**58**
0028	**40**
...	...
0023	**00**
0022	**00**
0021	**00**
0020	**40**
...	...
000A	**00**
0009	**00**
0008	**00**
0007	**00**
0006	**01**
0005	**01**
0004	**02**
0003	**46**
0002	**4C**
0001	**45**
0000	**7F**

- **uint8(x)** prints the byte that is located at the decimal position x, counted through a zero based indexing. The first byte from the the starting is at 0, second one is at 1 and so on.

- **uint32(x)** prints the word that is located at the start of decimal position x, counted through a zero based indexing. The word is located in order $x + 3 \rightarrow x + 2 \rightarrow x + 1 \rightarrow x$ The word from the the starting is at order $3 \rightarrow 2 \rightarrow 1 \rightarrow 0$, second one is at $4 \rightarrow 3 \rightarrow 2 \rightarrow 1$ and so on.

- **filesize** prints the size of the file in bytes. The libxselinux.so file is of 24512 bytes.



Word at 1(hex 1): 02 46 4c 45

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0300 3e00 0100 0000 0000 0000 0000 0000  ..>.............
00000020: 4000 0000 0000 0000 4058 0000 0000 0000  @.......@X......
```

Word at 32(hex 20): 00 00 00 40

Figure 5: Detecting position from hex file

8 Conclusion

YARA rules are syntactically flexible and the architecture is also modular with a robust framework. With its extensive community support, YARA continues to evolve as a vital asset in the arsenal of cybersecurity professionals, enabling proactive defense measures against emerging threats and facilitating collaborative efforts in the fight against cybercrime.