Community Experience Distilled

Learning Linux Binary Analysis

Uncover the secrets of Linux binary analysis with this handy guide

Ryan "elfmaster" O'Neill

[PACKT] open source
PUBLISHING
community experience distilled

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 **'The Linux Environment and Its Tools'**
- A synopsis of the book's content
- More information on **Learning Linux Binary Analysis**

About the Author

Ryan "elfmaster" O'Neill is a computer security researcher and software engineer with a background in reverse engineering, software exploitation, security defense, and forensics technologies. He grew up in the computer hacker subculture, the world of EFnet, BBS systems, and remote buffer overflows on systems with an executable stack. He was introduced to system security, exploitation, and virus writing at a young age. His great passion for computer hacking has evolved into a love for software development and professional security research. Ryan has spoken at various computer security conferences, including DEFCON and RuxCon, and also conducts a 2-day ELF binary hacking workshop.

He has an extremely fulfilling career and has worked at great companies such as Pikewerks, Leviathan Security Group, and more recently Backtrace as a software engineer.

Ryan has not published any other books, but he is well known for some of his papers published in online journals such as *Phrack* and *VXHeaven*. Many of his other publications can be found on his website at `http://www.bitlackeys.org`.

Preface

Software engineering is the act of creating an invention that exists, lives, and breathes on a microprocessor. We call it a program. Reverse engineering is the act of discovering how exactly that program lives and breathes, and furthermore it is how we can understand, dissect, or modify the behavior of that program using a combination of disassemblers and reversing tools and relying on our hacker instincts to master the target program which we are reverse engineering. We must understand the intricacies of binary formats, memory layout, and the instruction set of the given processor. We therefore become masters of the very life given to a program on a microprocessor. A reverse engineer is skilled in the art of binary mastery. This book is going to give you the proper lessons, insight, and tasks required to become a Linux binary hacker. When someone can call themselves a reverse engineer, they elevate themselves beyond the level of just engineering. A true hacker can not only write code but also dissect code, disassembling the binaries and memory segments in pursuit of modifying the inner workings of a software program; now that is power…

On both a professional and a hobbyist level, I use my reverse engineering skills in the computer security field, whether it is vulnerability analysis, malware analysis, antivirus software, rootkit detection, or virus design. Much of this book will be focused towards computer security. We will analyze memory dumps, reconstruct process images, and explore some of the more esoteric regions of binary analysis, including Linux virus infection and binary forensics. We will dissect malware-infected executables and infect running processes. This book is aimed at explaining the necessary components for reverse engineering in Linux, so we will be going deep into learning ELF (executable and linking format), which is the binary format used in Linux for executables, shared libraries, core dumps, and object files. One of the most significant aspects of this book is the deep insight it gives into the structural complexities of the ELF binary format. The ELF sections, segments, and dynamic linking concepts are vital and exciting chunks of knowledge. We will explore the depths of hacking ELF binaries and see how these skills can be applied to a broad spectrum of work.

The goal of this book is to teach you to be one of the few people with a strong foundation in Linux binary hacking, which will be revealed as a vast topic that opens the door to innovative research and puts you on the cutting edge of low-level hacking in the Linux operating system. You will walk away with valuable knowledge of Linux binary (and memory) patching, virus engineering/analysis, kernel forensics, and the ELF binary format as a whole. You will also gain more insights into program execution and dynamic linking and achieve a higher understanding of binary protection and debugging internals.

I am a computer security researcher, software engineer, and hacker. This book is merely an organized observation and documentation of the research I have done and the foundational knowledge that has manifested as a result.

This knowledge covers a wide span of information that can't be found in any one place on the Internet. This book tries to bring many interrelated topics together into one piece so that it may serve as an introductory manual and reference to the subject of Linux binary and memory hacking. It is by no means a complete reference but does contain a lot of core information to get started with.

What this book covers

Chapter 1, *The Linux Environment and Its Tools*, gives a brief description of the Linux environment and its tools, which we will be using throughout the book.

Chapter 2, *The ELF Binary Format*, helps you learn about every major component of the ELF binary format that is used across Linux and most Unix-flavored operating systems.

Chapter 3, *Linux Process Tracing*, teaches you to use the ptrace system call to read and write to process memory and inject code.

Chapter 4, *ELF Virus Technology – Linux/Unix Viruses*, is where you discover the past, present, and future of Linux viruses, how they are engineered, and all of the amazing research that surrounds them.

Chapter 5, *Linux Binary Protection*, explains the basic internals of ELF binary protection.

Chapter 6, *ELF Binary Forensics in Linux*, is where you learn to dissect ELF objects in search of viruses, backdoors, and suspicious code injection.

Chapter 7, *Process Memory Forensics*, shows you how to dissect a process address space in search of malware, backdoors, and suspicious code injection that live in the memory.

Chapter 8, *ECFS – Extended Core File Snapshot Technology*, is an introduction to ECFS, a new open source product for deep process memory forensics.

Chapter 9, *Linux /proc/kcore Analysis*, shows how to detect Linux kernel malware through memory analysis with /proc/kcore.

1
The Linux Environment and Its Tools

In this chapter, we will be focusing on the Linux environment as it pertains to our focus throughout this book. Since this book is focused about Linux binary analysis, it makes sense to utilize the native environment tools that come with Linux and to which everyone has access. Linux comes with the ubiquitous binutils already installed, but they can be found at `http://www.gnu.org/software/binutils/`. They contain a huge selection of tools that are handy for binary analysis and hacking. This is not another book on using IDA Pro. IDA is hands-down the best universal software for reverse engineering of binaries, and I would encourage its use as needed, but we will not be using it in this book. Instead, you will acquire the skills to hop onto virtually any Linux system and have an idea on how to begin hacking binaries with an environment that is already accessible. You can therefore learn to appreciate the beauty of Linux as a true hackers' environment for which there are many free tools available. Throughout the book, we will demonstrate the use of various tools and give a recap on how to use them as we progress through each chapter. Meanwhile, however, let this chapter serve as a primer or reference to these tools and tips within the Linux environment. If you are already very familiar with the Linux environment and its tools for disassembling, debugging, and parsing of ELF files, then you may simply skip this chapter.

Linux tools

Throughout this book, we will be using a variety of free tools that are accessible by anyone. This section will give a brief synopsis of some of these tools for you.

GDB

GNU Debugger (**GDB**) is not only good to debug buggy applications. It can also be used to learn about a program's control flow, change a program's control flow, and modify the code, registers, and data structures. These tasks are common for a hacker who is working to exploit a software vulnerability or is unraveling the inner workings of a sophisticated virus. GDB works on ELF binaries and Linux processes. It is an essential tool for Linux hackers and will be used in various examples throughout this book.

Objdump from GNU binutils

Object dump (**objdump**) is a simple and clean solution for a quick disassembly of code. It is great for disassembling simple and untampered binaries, but will show its limitations quickly when attempting to use it for any real challenging reverse engineering tasks, especially against hostile software. Its primary weakness is that it relies on the `ELF` section headers and doesn't perform control flow analysis, which are both limitations that greatly reduce its robustness. This results in not being able to correctly disassemble the code within a binary, or even open the binary at all if there are no section headers. For many conventional tasks, however, it should suffice, such as when disassembling common binaries that are not fortified, stripped, or obfuscated in any way. It can read all common `ELF` types. Here are some common examples of how to use `objdump`:

- View all data/code in every section of an `ELF` file:
  ```
  objdump -D <elf_object>
  ```
- View only program code in an `ELF` file:
  ```
  objdump -d <elf_object>
  ```
- View all symbols:
  ```
  objdump -tT <elf_object>
  ```

We will be exploring `objdump` and other tools in great depth during our introduction to the `ELF` format in *Chapter 2*, *The ELF Binary Format*.

Objcopy from GNU binutils

Object copy (**Objcopy**) is an incredibly powerful little tool that we cannot summarize with a simple synopsis. I recommend that you read the manual pages for a complete description. `Objcopy` can be used to analyze and modify `ELF` objects of any kind, although some of its features are specific to certain types of `ELF` objects. `Objcopy` is often times used to modify or copy an `ELF` section to or from an `ELF` binary.

To copy the `.data` section from an `ELF` object to a file, use this line:

```
objcopy -only-section=.data <infile> <outfile>
```

The `objcopy` tool will be demonstrated as needed throughout the rest of this book. Just remember that it exists and can be a very useful tool for the Linux binary hacker.

strace

System call trace (**strace**) is a tool that is based on the `ptrace(2)` system call, and it utilizes the `PTRACE_SYSCALL` request in a loop to show information about the system call (also known as `syscalls`) activity in a running program as well as signals that are caught during execution. This program can be highly useful for debugging, or just to collect information about what `syscalls` are being called during runtime.

This is the `strace` command used to trace a basic program:

```
strace /bin/ls -o ls.out
```

The `strace` command used to attach to an existing process is as follows:

```
strace -p <pid> -o daemon.out
```

The initial output will show you the file descriptor number of each system call that takes a file descriptor as an argument, such as this:

```
SYS_read(3, buf, sizeof(buf));
```

If you want to see all of the data that was being read into file descriptor 3, you can run the following command:

```
strace -e read=3 /bin/ls
```

You may also use `-e write=fd` to see written data. The `strace` tool is a great little tool, and you will undoubtedly find many reasons to use it.

ltrace

library trace (**ltrace**) is another neat little tool, and it is very similar to `strace`. It works similarly, but it actually parses the shared library-linking information of a program and prints the library functions being used.

Basic ltrace command

You may see system calls in addition to library function calls with the `-S` flag. The `ltrace` command is designed to give more granular information, since it parses the dynamic segment of the executable and prints actual symbols/functions from shared and static libraries:

```
ltrace <program> -o program.out
```

ftrace

Function trace (**ftrace**) is a tool designed by me. It is similar to `ltrace`, but it also shows calls to functions within the binary itself. There was no other tool I could find publicly available that could do this in Linux, so I decided to code one. This tool can be found at `https://github.com/elfmaster/ftrace`. A demonstration of this tool is given in the next chapter.

readelf

The `readelf` command is one of the most useful tools around for dissecting `ELF` binaries. It provides every bit of the data specific to `ELF` necessary for gathering information about an object before reverse engineering it. This tool will be used often throughout the book to gather information about symbols, segments, sections, relocation entries, dynamic linking of data, and more. The `readelf` command is the Swiss Army knife of `ELF`. We will be covering it in depth as needed, during *Chapter 2*, *The ELF Binary Format*, but here are a few of its most commonly used flags:

- To retrieve a section header table:

  ```
  readelf -S <object>
  ```

- To retrieve a program header table:

  ```
  readelf -l <object>
  ```

- To retrieve a symbol table:

  ```
  readelf -s <object>
  ```

- To retrieve the `ELF` file header data:

  ```
  readelf -e <object>
  ```

- To retrieve relocation entries:

  ```
  readelf -r <object>
  ```

- To retrieve a dynamic segment:

  ```
  readelf -d <object>
  ```

ERESI – The ELF reverse engineering system interface

ERESI project (`http://www.eresi-project.org`) contains a suite of many tools that are a Linux binary hacker's dream. Unfortunately, many of them are not kept up to date and aren't fully compatible with 64-bit Linux. They do exist for a variety of architectures, however, and are undoubtedly the most innovative single collection of tools for the purpose of hacking `ELF` binaries that exist today. Because I personally am not really familiar with using the ERESI project's tools, and because they are no longer kept up to date, I will not be exploring their capabilities within this book. However, be aware that there are two Phrack articles that demonstrate the innovation and powerful features of the ERESI tools:

- Cerberus ELF interface (`http://www.phrack.org/archives/issues/61/8.txt`)
- Embedded ELF debugging (`http://www.phrack.org/archives/issues/63/9.txt`)

Useful devices and files

Linux has many files, devices, and `/proc` entries that are very helpful for the avid hacker and reverse engineer. Throughout this book, we will be demonstrating the usefulness of many of these files. Here is a description of some of the commonly used ones throughout the book.

/proc/<pid>/maps

`/proc/<pid>/maps` file contains the layout of a process image by showing each memory mapping. This includes the executable, shared libraries, stack, heap, VDSO, and more. This file is critical for being able to quickly parse the layout of a process address space and is used more than once throughout this book.

/proc/kcore

The `/proc/kcore` is an entry in the `proc` filesystem that acts as a dynamic core file of the Linux kernel. That is, it is a raw dump of memory that is presented in the form of an `ELF` core file that can be used by GDB to debug and analyze the kernel. We will explore `/proc/kcore` in depth in *Chapter 9*, *Linux /proc/kcore Analysis*.

/boot/System.map

This file is available on almost all Linux distributions and is very useful for kernel hackers. It contains every symbol for the entire kernel.

/proc/kallsyms

The `kallsyms` is very similar to `System.map`, except that it is a `/proc` entry that means that it is maintained by the kernel and is dynamically updated. Therefore, if any new LKMs are installed, the symbols will be added to `/proc/kallsyms` on the fly. The `/proc/kallsyms` contains at least most of the symbols in the kernel and will contain all of them if specified in the `CONFIG_KALLSYMS_ALL` kernel config.

/proc/iomem

The `iomem` is a useful proc entry as it is very similar to `/proc/<pid>/maps`, but for all of the system memory. If, for instance, you want to know where the kernel's text segment is mapped in the physical memory, you can search for the `Kernel` string and you will see the `code/text` segment, the data segment, and the `bss` segment:

```
$ grep Kernel /proc/iomem
01000000-016d9b27 : Kernel code
016d9b28-01ceeebf : Kernel data
01df0000-01f26fff : Kernel bss
```

ECFS

Extended core file snapshot (**ECFS**) is a special core dump technology that was specifically designed for advanced forensic analysis of a process image. The code for this software can be found at `https://github.com/elfmaster/ecfs`. Also, *Chapter 8*, *ECFS – Extended Core File Snapshot Technology*, is solely devoted to explaining what ECFS is and how to use it. For those of you who are into advanced memory forensics, you will want to pay close attention to this.

Linker-related environment points

The dynamic loader/linker and linking concepts are inescapable components involved in the process of program linking and execution. Throughout this book, you will learn a lot about these topics. In Linux, there are quite a few ways to alter the dynamic linker's behavior that can serve the binary hacker in many ways. As we move through the book, you will begin to understand the process of linking, relocations, and dynamic loading (program interpreter). Here are a few linker-related attributes that are useful and will be used throughout the book.

The LD_PRELOAD environment variable

The `LD_PRELOAD` environment variable can be set to specify a library path that should be dynamically linked before any other libraries. This has the effect of allowing functions and symbols from the preloaded library to override the ones from the other libraries that are linked afterwards. This essentially allows you to perform runtime patching by redirecting shared library functions. As we will see in later chapters, this technique can be used to bypass anti-debugging code and for userland rootkits.

The LD_SHOW_AUXV environment variable

This environment variable tells the program loader to display the program's auxiliary vector during runtime. The auxiliary vector is information that is placed on the program's stack (by the kernel's `ELF` loading routine), with information that is passed to the dynamic linker with certain information about the program. We will examine this much more closely in *Chapter 3*, *Linux Process Tracing*, but the information might be useful for reversing and debugging. If, for instance, you want to get the memory address of the VDSO page in the process image (which can also be obtained from the `maps` file, as shown earlier) you have to look for `AT_SYSINFO`.

Here is an example of the auxiliary vector with `LD_SHOW_AUXV`:

```
$ LD_SHOW_AUXV=1 whoami
AT_SYSINFO: 0xb7779414
AT_SYSINFO_EHDR: 0xb7779000
AT_HWCAP: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2
AT_PAGESZ: 4096
AT_CLKTCK: 100
AT_PHDR:  0x8048034
AT_PHENT: 32
AT_PHNUM: 9
AT_BASE:  0xb777a000
AT_FLAGS: 0x0
AT_ENTRY: 0x8048eb8
AT_UID:  1000
AT_EUID: 1000
AT_GID:  1000
AT_EGID: 1000
AT_SECURE: 0
```

```
AT_RANDOM: 0xbfb4ca2b
AT_EXECFN: /usr/bin/whoami
AT_PLATFORM: i686
elfmaster
```

The auxiliary vector will be covered in more depth in *Chapter 2*, *The ELF Binary Format*.

Linker scripts

Linker scripts are a point of interest to us because they are interpreted by the linker and help shape a program's layout with regard to sections, memory, and symbols. The default linker script can be viewed with `ld -verbose`.

The `ld` linker program has a complete language that it interprets when it is taking input files (such as relocatable object files, shared libraries, and header files), and it uses this language to determine how the output file, such as an executable program, will be organized. For instance, if the output is an `ELF` executable, the linker script will help determine what the layout will be and what sections will exist in which segments. Here is another instance: the `.bss` section is always at the end of the data segment; this is determined by the linker script. You might be wondering how this is interesting to us. Well! For one, it is important to have some insights into the linking process during compile time. The `gcc` relies on the linker and other programs to perform this task, and in some instances, it is important to be able to have control over the layout of the executable file. The `ld` command language is quite an in-depth language and is beyond the scope of this book, but it is worth checking out. And while reverse engineering executables, remember that common segment addresses may sometimes be modified, and so can other portions of the layout. This indicates that a custom linker script is involved. A linker script can be specified with `gcc` using the `-T` flag. We will look at a specific example of using a linker script in *Chapter 5*, *Linux Binary Protection*.

Summary

We just touched upon some fundamental aspects of the Linux environment and the tools that will be used most commonly in the demonstrations from each chapter. Binary analysis is largely about knowing the tools and resources that are available for you and how they all fit together. We only briefly covered the tools, but we will get an opportunity to emphasize the capabilities of each one as we explore the vast world of Linux binary hacking in the following chapters. In the next chapter, we will delve into the internals of the ELF binary format and cover many interesting topics, such as dynamic linking, relocations, symbols, sections, and more.

Where to buy this book

You can buy Learning Linux Binary Analysis from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.