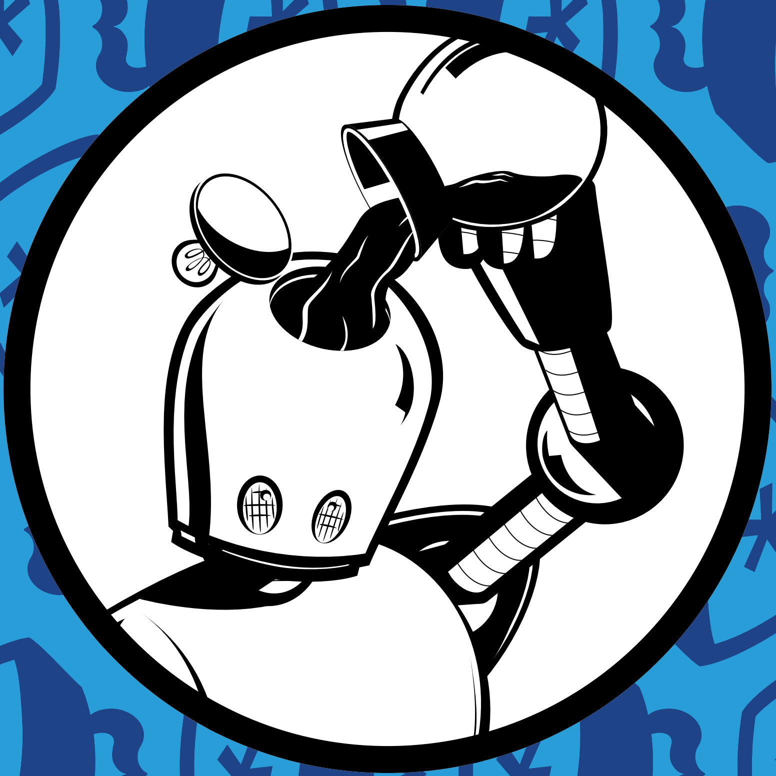


Malware Data Science

Attack Detection and Attribution

Joshua Saxe with Hillary Sanders

no starch press

6

UNDERSTANDING MACHINE LEARNING–BASED MALWARE DETECTORS

With the open source machine learning tools available today, you can build custom, machine learning–based malware detection tools, whether as your primary detection tool or to supplement commercial solutions, with relatively little effort.

But why build your own machine learning tools when commercial antivirus solutions are already available? When you have access to examples of particular threats, such as malware used by a certain group of attackers targeting your network, building your own machine learning–based detection technologies can allow you to catch new examples of these threats.

In contrast, commercial antivirus engines might miss these threats unless they already include signatures for them. Commercial tools are also "closed books"—that is, we don't necessarily know how they work and we have limited ability to tune them. When we build our own detection methods, we know how they work and can tune them to our liking to reduce false positives or false negatives. This is helpful because in some applications you might be willing to tolerate more false positives in exchange for fewer false negatives (for example, when you're searching your network for suspicious files so that you can hand-inspect them to determine if they are malicious), and in other applications you might be willing to tolerate more false negatives in exchange for fewer false positives (for example, if your application blocks programs from executing if it determines they are malicious, meaning that false positives are disruptive to users).

In this chapter, you learn the process of developing your own detection tools at a high level. I start by explaining the big ideas behind machine learning, including feature spaces, decision boundaries, training data, underfitting, and overfitting. Then I focus on four foundational approaches—logistic regression, k-nearest neighbors, decision trees, and random forest—and how these can be applied to perform detection.

You'll then use what you learned in this chapter to learn how to evaluate the accuracy of machine learning systems in Chapter 7 and implement machine learning systems in Python in Chapter 8. Let's get started.

Steps for Building a Machine Learning–Based Detector

There is a fundamental difference between machine learning and other kinds of computer algorithms. Whereas traditional algorithms tell the computer what to do, machine-learning systems learn how to solve a problem by example. For instance, rather than simply pulling from a set of preconfigured rules, machine learning security detection systems can be trained to determine whether a file is bad or good by learning from examples of good and bad files.

The promise of machine learning systems for computer security is that they automate the work of creating signatures, and they have the potential to perform more accurately than signature-based approaches to malware detection, especially on new, previously unseen malware.

Essentially, the workflow we follow to build any machine learning–based detector, including a decision tree, boils down to these steps:

1. **Collect** examples of malware and benignware. We will use these examples (called *training examples*) to train the machine learning system to recognize malware.
2. **Extract** features from each training example to represent the example as an array of numbers. This step also includes research to design good features that will help your machine learning system make accurate inferences.
3. **Train** the machine learning system to recognize malware using the features we have extracted.
4. **Test** the approach on some data not included in our training examples to see how well our detection system works.

Let's discuss each of these steps in more detail in the following sections.

Gathering Training Examples

Machine learning detectors live or die by the training data provided to them. Your malware detector's ability to recognize suspicious binaries depends heavily on the quantity and quality of training examples you provide. Be prepared to spend much of your time gathering training examples when building machine learning–based detectors, because the more examples you feed your system, the more accurate it's likely to be.

The quality of your training examples is also important. The malware and benignware you collect should mirror the kind of malware and benignware you expect your detector to see when you ask it to decide whether new files are malicious or benign.

For example, if you want to detect malware from a specific threat actor group, you must collect as much malware as possible from that group for use in training your system. If your goal is to detect a broad class of malware (such as ransomware), it's essential to collect as many representative samples of this class as possible.

By the same token, the benign training examples you feed your system should mirror the kinds of benign files you will ask your detector to analyze once you deploy it. For example, if you are working on detecting malware on a university network, you should train your system with a broad sampling of the benignware that students and university employees use, in order to avoid false positives. These benign examples would include computer games, document editors, custom software written by the university IT department, and other types of nonmalicious programs.

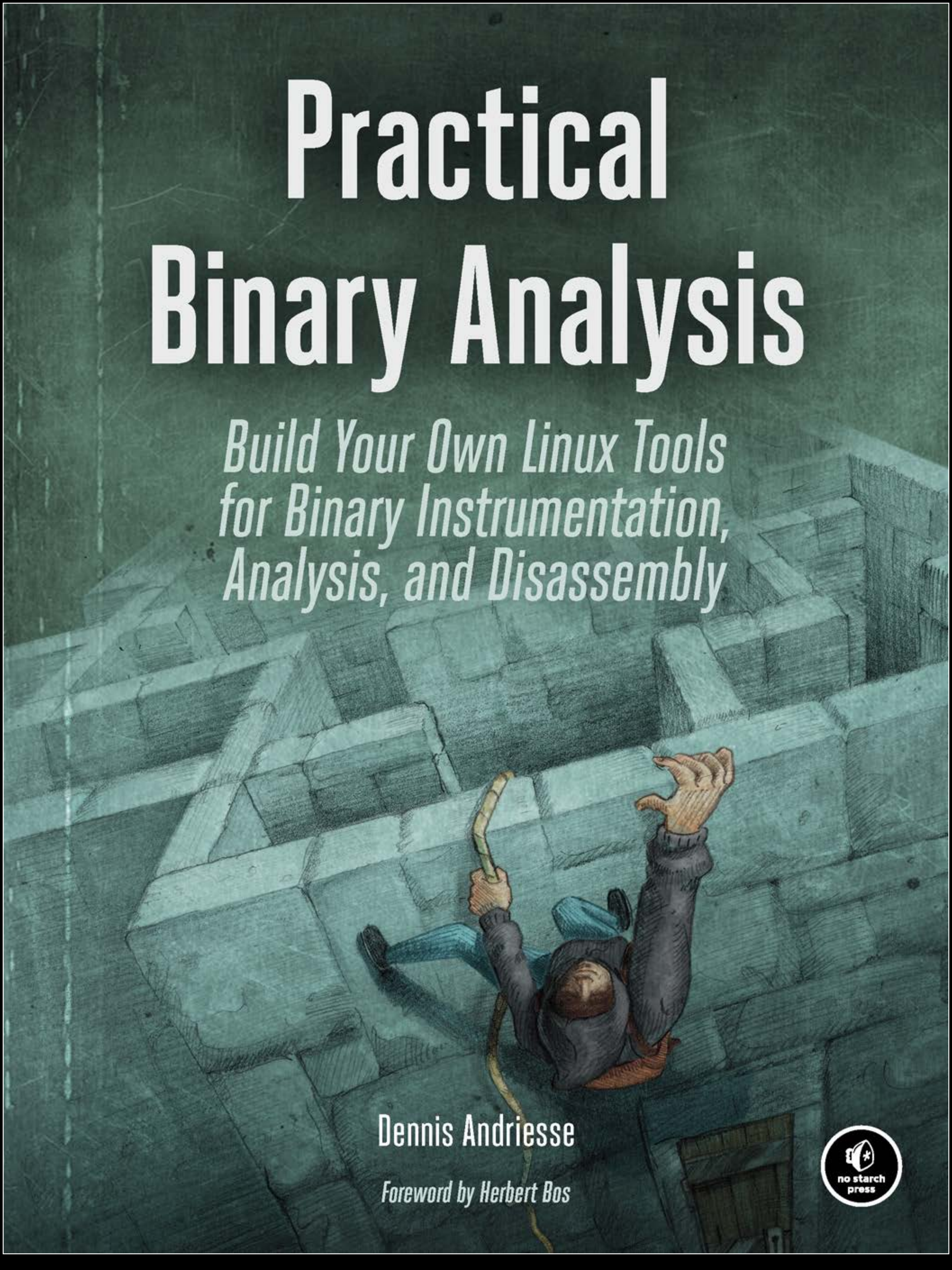
To give a real-world example, at my current day job, we built a detector that detects malicious Office documents. We spent about half the time on this project gathering training data, and this included collecting benign documents generated by more than a thousand of my company's employees. Using these examples to train our system significantly reduced our false positive rate.

Extracting Features

To classify files as good or bad, we train machine learning systems by showing them features of software binaries; these are file attributes that will help the system distinguish between good and bad files. For example, here are some features we might use to determine whether a file is good or bad:

- Whether it's digitally signed
- The presence of malformed headers
- The presence of encrypted data
- Whether it has been seen on more than 100 network workstations

To obtain these features, we need to extract them from files. For example, we might write code to determine whether a file is digitally signed, has malformed headers, contains encrypted data, and so on.



Practical Binary Analysis

Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly

Dennis Andriesse

Foreword by Herbert Bos

no starch press

5
BASIC BINARY ANALYSIS IN LINUX

Even in the most complex binary analysis, you can accomplish surprisingly advanced feats by combining a set of basic tools in the right way. This can save you hours of work implementing equivalent functionality on your own. In this chapter, you'll learn the fundamental tools you'll need to perform binary analysis on Linux.

Instead of simply showing you a list of tools and explaining what they do, I'll use a *Capture the Flag (CTF)* challenge to illustrate how they work. In computer security and hacking, CTF challenges are often played as contests, where the goal is typically to analyze or exploit a given binary (or a running process or server) until you manage to capture a flag hidden in the binary. The flag is usually a hexadecimal string, which you can use to prove that you completed the challenge as well as unlock new challenges.

In this CTF, you start with a mysterious file called *payload*, which you can find on the VM in the directory for this chapter. The goal is to figure out how to extract the hidden flag from *payload*. In the process of analyzing *payload* and looking for the flag, you'll learn to use a wide range of basic binary analysis tools that are available on virtually any Linux-based system (most of them as part of GNU `coreutils` or `binutils`). I encourage you to follow along.

Most of the tools you'll see have a number of useful options, but there are far too many to cover exhaustively in this chapter. Thus, it's a good idea to check out the man page for every tool using the command `man` *`tool`* on the VM. At the end of the chapter, you'll use the recovered flag to unlock a new challenge, which you can complete on your own!

5.1 Resolving Identity Crises Using file

Because you received absolutely no hints about the contents of *payload*, you have no idea what to do with this file. When this happens (for instance, in reverse engineering or forensics scenarios), a good first step is to figure out what you can about the file type and its contents. The `file` utility was designed for this purpose; it takes a number of files as input and then tells you what type each file is. You may remember it from Chapter 2, where I used `file` to find out the type of an ELF file.

The nice thing about `file` is that it isn't fooled by extensions. Instead, it searches for other telltale patterns in the file, such as magic bytes like the `0x7f ELF` sequence at the start of ELF files, to find out the file type. This is perfect here because the *payload* file doesn't have an extension. Here's what `file` tells you about *payload*:

```
$ file payload
payload: ASCII text
```

As you can see, *payload* contains ASCII text. To examine the text in detail, you can use the `head` utility, which dumps the first few lines (10 by default) of a text file to `stdout`. There's also an analogous utility called `tail`, which shows you the last few lines of a file. Here's what the `head` utility's output shows:

```
$ head payload
H4sIAKiT61gAA+xaD3RTVZq/Sf9TSKL8aflnn56ioNJJSiktDpqUlL5oOUpbYEVIOzRtI2naSV5K
YVOHTig21jqojH9mnRV35syZPWd35ZzZOOXHxWBHYJydXf4ckRldZRUxBRzxz2CFQvb77ru3ee81
AZdZZ92z+XrS733fu993v/v/vnt/bqmVfNNkBlqOcCFyy6KFZiUHKi1buMhMLAvMiOoXWSzlZYtA
v2hRWRkRzN94ZEChoOQKCAJp8fdcNt2V3v8fpe9X1y7T63Rjsp7cTlCKGq1UtjL9yPUJGyupIHnw
/zoym2SDnKVIZyVWFR9hrjnPZeky4JcJvwq9LFforSo+i6XjXKfgWaoSWFX8mclExQkRxuww1uOz
Ze3x2UOqfpDFcUyvttMzuxFmN8LScO54er26fJns18DODaxcnNtZOrsiPVLdh1ILPudey/xda1Xx
MpauTGN3L9hlk69PJsZXsPxS1YvA4uect8N3fN7m8rLv+Frm+7z+UM/8nory+eVlJcHOklIak4ml
rbm7kabn9SiwmKcQuQ/g+3n/OJj/byfuqjvO9uKVj888906TvxXM+G4qSbRbX1TQCZnWPNQVwG86
/F7+4IkHl1a/eebY91bPemngU8OpI58YNjrWD16u3P3wuzaJ3kh4i6vpuhT6g7rkfs6kODtS6P8l
hf6NFPocfXL9yRTpSOny+NtJ8vR3pOhfl8J/bgr9VynOb6bQkxTl+ixF+p+mON+qx743k+wWmlT6
```

That definitely doesn't look human-readable. Taking a closer look at the alphabet used in the file, you can see that it consists of only alphanumeric characters and the characters + and /, organized in neat rows. When you see a file that looks like this, it's usually safe to assume that it's a *Base64* file.

Base64 is a widely used method of encoding binary data as ASCII text. Among other things, it's commonly used in email and on the web to ensure that binary data transmitted over a network isn't accidentally malformed by services that can handle only text. Conveniently, Linux systems come with a tool called `base64` (typically as part of GNU `coreutils`) that can encode and decode Base64. By default, `base64` will encode any files or `stdin` input given to it. But you can use the `-d` flag to tell `base64` to decode instead. Let's decode *payload* to see what you get!

```
$ base64 -d payload > decoded_payload
```

This command decodes *payload* and then stores the decoded contents in a new file called `decoded_payload`. Now that you've decoded *payload*, let's use `file` again to check the type of the decoded file.

```
$ file decoded_payload
decoded_payload: gzip compressed data, last modified: Tue Oct 22 15:46:43 2019, from Unix
```

Now you're getting somewhere! It turns out that behind the layer of Base64 encoding, the mysterious file is actually just a compressed archive that uses `gzip` as the outer compression layer. This is an opportunity to introduce another handy feature of `file`: the ability to peek inside zipped files. You can pass the `-z` option to `file` to see what's inside the archive without extracting it. Here's what you should see:

```
$ file -z decoded_payload
decoded_payload: POSIX tar archive (GNU) (gzip compressed data, last modified:
                 Tue Oct 22 19:08:12 2019, from Unix)
```

You can see that you're dealing with multiple layers that you need to extract, because the outer layer is a `gzip` compression layer and inside that is a `tar` archive, which typically contains a bundle of files. To reveal the files stored inside, you use `tar` to unzip and extract `decoded_payload`, like this:

```
$ tar xvzf decoded_payload
ctf
67b8601
```

As shown in the `tar` log, there are two files extracted from the archive: *ctf* and *67b8601*. Let's use `file` again to see what kinds of files you're dealing with.

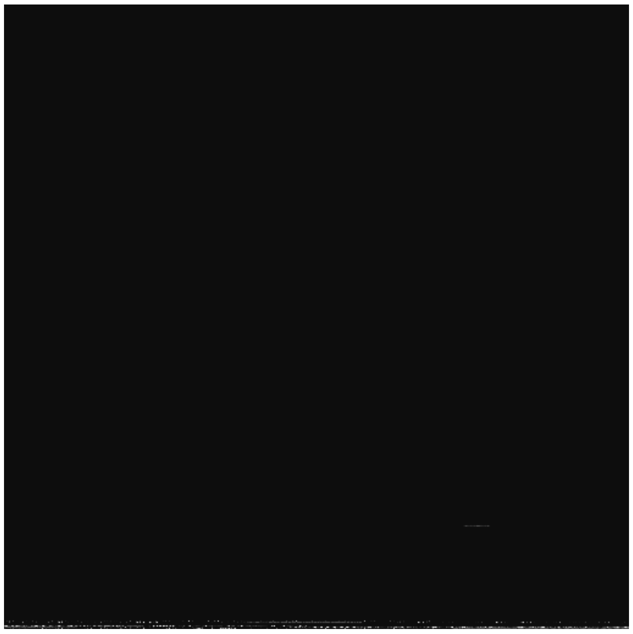
```
$ file ctf
ctf: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=29aeb60bcee44b50d1db3a56911bd1de93cd2030, stripped
```

The first file, *ctf*, is a dynamically linked 64-bit stripped ELF executable. The second file, called *67b8601*, is a bitmap (BMP) file of 512 × 512 pixels. Again, you can see this using `file` as follows:

```
$ file 67b8601
67b8601: PC bitmap, Windows 3.x format, 512 x 512 x 24
```

This BMP file depicts a black square, as you can see in Figure 5-1a. If you look carefully, you should see some irregularly colored pixels at the bottom of the figure. Figure 5-1b shows an enlarged snippet of these pixels.

Before exploring what this all means, let's first take a closer look at *ctf*, the ELF file you just extracted.



(a) The complete figure

(b) Enlarged view of some of the colored pixels at the bottom

Figure 5-1: The extracted BMP file, 67b8601

5.2 Using ldd to Explore Dependencies

Although it's not wise to run unknown binaries, since you're working in a VM, let's try running the extracted *ctf* binary. When you try to run the file, you don't get far.

```
$ ./ctf
./ctf: error while loading shared libraries: lib5ae9b7f.so:
      cannot open shared object file: No such file or directory
```

Before any of the application code is even executed, the dynamic linker complains about a missing library called *lib5ae9b7f.so*. That doesn't sound like a library you normally find on any system. Before searching for this library, it makes sense to check whether *ctf* has any more unresolved dependencies.

Linux systems come with a program called `ldd`, which you can use to find out on which shared objects a binary depends and where (if anywhere) these dependencies are on your system. You can even use `ldd` along with the `-v` flag to find out which library versions the binary expects, which can be useful for debugging. As mentioned in the `ldd man` page, `ldd` may run the binary to figure out the dependencies, so it's not safe to use on untrusted binaries unless you're running it in a VM or another isolated environment. Here's the `ldd` output for the *ctf* binary:

```
$ ldd ctf
        linux-vdso.so.1 =>  (0x00007fff6edd4000)
        lib5ae9b7f.so => not found
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f67c2cbe000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f67c2aa7000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f67c26de000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f67c23d5000)
        /lib64/ld-linux-x86-64.so.2 (0x0000561e62fe5000)
```

Luckily, there are no unresolved dependencies besides the missing library identified earlier, *lib5ae9b7f.so*. Now you can focus on figuring out what this mysterious library is and how you can obtain it in order to capture the flag!

Because it's obvious from the library name that you won't find it in any standard repository, it must reside somewhere in the files you've been given so far. Recall from Chapter 2 that all ELF binaries and libraries begin with the magic sequence `0x7f ELF`. This is a handy string to look for in search of your missing library; as long as the library is not encrypted, you should be able to find the ELF header this way. Let's try a simple `grep` for the string `'ELF'`.

```
$ grep 'ELF' *
Binary file 67b8601 matches
Binary file ctf matches
```

Real-World Web Hacking

A Field Guide to Bug Hunting

Peter Yaworski

no starch press

2

OPEN REDIRECT VULNERABILITY

We'll begin our discussion with *open redirect vulnerabilities,* which occur when a target visits a website and that website sends their browser to a different URL, potentially on a separate domain. Open redirects exploit the trust of a given domain to lure victims to a malicious website. A phishing attack can also accompany a redirect to trick users into believing they're submitting information to a trusted site when their valuable information is being sent to a malicious site. When combined with other attacks, open redirects can also enable attackers to distribute malware from the malicious site or to steal OAuth tokens (a topic we explore in Chapter 17).

Because open redirects only redirect users, they're sometimes considered low impact and not deserving of a bounty. For example, the Google bug bounty program typically considers open redirects too low risk to reward. The Open Web Application Security Project (OWASP), which is a community that focuses on application security and curates a list of the most critical security flaws in web applications, has also removed open redirects from their 2017 list of top ten vulnerabilities.

Although these are low-impact vulnerabilities, they're great for learning how browsers handle redirects in general. In this chapter you'll learn how open redirects can be exploited and how to identify key parameters, using three bug reports as examples.

How Open Redirects Work

Intended redirects often work by accepting a destination URL as a parameter in a URL. This parameter tells a browser to send a `GET` request to the destination URL. A site is vulnerable to an open redirect when it doesn't check that the redirect address is a safe URL, or it redirects the user without notifying them that they're being redirected. When you're looking for these vulnerabilities, keep an eye out for URL parameters that include certain names, such as `url=`, `redirect=`, `next=`, and so on, which might denote URLs that users will be redirected to. For example, suppose Google had functionality to redirect users to Gmail by visiting the following URL:

```
https://www.google.com/?redirect_to=https://www.gmail.com
```

In this scenario, when you visit this URL, Google receives a `GET` HTTP request and uses the `redirect_to` parameter's value to determine where to redirect your browser. After doing so, Google servers return an HTTP response with a status code instructing the browser to redirect the user. Typically, the status code is 302, but in some cases it could be 301, 303, 307, or 308. These HTTP response codes tell your browser that a page has been found; however, the codes also specify that the browser make a `GET` request to the `redirect_to` parameter's value, *https://www.gmail.com/*, which is denoted in the 30*x* HTTP responses' `Location` header. The `Location` header specifies where to redirect `GET` requests.

Now, suppose we change the original URL to the following:

```
https://www.google.com/?redirect_to=https://www.<attacker>.com
```

If Google isn't validating that the `redirect_to` parameter is for one of its own legitimate sites where it intends to send visitors, an attacker could substitute the parameter with another URL. As a result, an HTTP response could instruct your browser to make a `GET` request to *https://www.*<attacker>*.com/*. After the attacker has you on their malicious site, they could carry out other attacks.

Although the parameter in this example was clearly labeled, keep in mind that redirect parameters might not always be obviously named: parameters will vary from site to site or even within a site. In some cases, parameters might be labeled with just single characters, such as `r=` or `u=`.

In addition to parameter-based attacks, HTML meta refresh tags and JavaScript can redirect browsers. HTML `<meta>` tags can tell browsers to refresh a web page and make a `GET` request to a URL defined in the tag's `content` attribute. Here is what one might look like:

```
<meta http-equiv="refresh" content="0; url=https://www.google.com/"
```

The content attribute defines how browsers make an HTTP request in two ways. First, the content attribute defines how long the browser waits before making the HTTP request to the URL; in this case, `0` seconds. Secondly, the content attribute also specifies the `url` parameter in the website the browser makes the `GET` request to; in this case, `https://www.google.com`.Attackers can use this redirect behavior in situations where they have the ability to control the content attribute of a `<meta>` tag or to inject their own tag via some other vulnerability.

An attacker can also use JavaScript to redirect users by modifying the window's `location` property through the *Document Object Model* (*DOM*). The DOM is an API for HTML and XML documents that allows developers to modify the structure, style, and content of a web page. Because the `Location` header denotes where a request should be redirected to, browsers will immediately interpret this JavaScript and redirect to the specified URL. An attacker can modify the window's `location` property by using any of the following JavaScript:

```
window.location = https://www.google.com/
window.location.href = https://www.google.com
window.location.replace(https://www.google.com)
```

Although the differences between these approaches to changing window properties aren't important for the purposes of this book, the key takeaway is finding opportunities where `window.location` can be set to an attacker-controlled value. Typically, this opportunity occurs only where an attacker can execute JavaScript, either via a cross-site scripting vulnerability or where the website intentionally allows users to define a URL to redirect to, as in the HackerOne interstitial redirect vulnerability detailed later in the chapter on page XX.

When you're searching for open redirect vulnerabilities, you'll usually be monitoring your proxy history for a `GET` request sent to the site you're testing which includes a parameter specifying a URL redirect.

Shopify Theme Install Open Redirect

Difficulty: Low

URL: *https://apps.shopify.com/services/google/themes/preview/supply--blue?domain_name=XX*

Source: *https://www.hackerone.com/reports/101962/*

Date reported: November 25, 2015

Bounty paid: $500

The first example of an open redirect you'll learn about was found on Shopify, which is a commerce platform that allows users to set up an online store to sell goods. Shopify allows administrators to customize the look and feel of their stores by changing their theme. As part of that functionality, Shopify offered a feature to provide a preview for the theme by redirecting the store owners to a URL. The redirect URL was formatted as such:

```
https://app.shopify.com/services/google/themes/preview/supply--blue?domain_name=<example>.com
```

The `domain_name` parameter at the end of the URL redirected to the user's store domain and added `/admin` to the end of the URL. Shopify was expecting that the `domain_name` would always be a user's store and wasn't validating its value as part of the Shopify domain. As a result, an attacker could exploit the parameter to redirect a victim to *http://<example>.com/admin/* where the malicious attacker could carry out other attacks.

Takeaways

Not all vulnerabilities are complex. For this open redirect, simply changing the `domain_name` parameter to an external site would result in the user being redirected offsite from Shopify.

Shopify Login Open Redirect

Difficulty: Low

URL: *http://mystore.myshopify.com/account/login/*

Source: *https://www.hackerone.com/reports/103772/*

Date reported: December 6, 2015

Bounty paid: $500

This second example of an open redirect is similar to the first Shopify example except in this case, Shopify's parameter isn't redirecting the user to the domain specified by the URL parameter; instead, the open redirect tacks the parameter's value onto the end of a Shopify subdomain. Normally, this functionality would be used to redirect a user to a specific page on a given store. However, the URL can still be manipulated into redirecting the browser away from Shopify's subdomain and to an attacker's website by adding characters to change the meaning of the URL.

In this bug, after the user logged into Shopify, Shopify used the parameter `checkout_url` to redirect the user. For example, let's say a victim visited this URL:

```
http://mystore.myshopify.com/account/login?checkout_url=.<attacker>.com
```

They would have been redirected to the URL *http://mystore.myshopify.com.<attacker>.com/*, which isn't a Shopify domain.

Because the URL ends in *.<attacker>.com* and DNS lookups use the rightmost domain label, the redirect goes to the *<attacker>.com* domain. So when *http://mystore.myshopify.com.<attacker>.com/* is submitted for DNS lookup, it will match on *<attacker>.com*, which Shopify doesn't own, and not *myshopify.com* as

Shopify would have intended. Although an attacker wouldn't be able to freely send a victim anywhere, they could send a user to another domain by adding special characters, such as a period, to the values they can manipulate.

Takeaways

If you can only control a portion of the final URL returned by a site, adding special URL characters might change the meaning of the URL and redirect a user to another domain. Let's say you can only control the `checkout_url` parameter value, and you also notice that the parameter is being combined with a hardcoded URL on the backend of the site, such as the store URL *http://mystore.myshopify.com/*. In this situation, try adding special URL characters, like a period or the @ symbol, to test the vulnerability of the parameter.

HackerOne Interstitial Redirect

Difficulty: Low

URL: N/A

Source: *https://www.hackerone.com/reports/111968/*

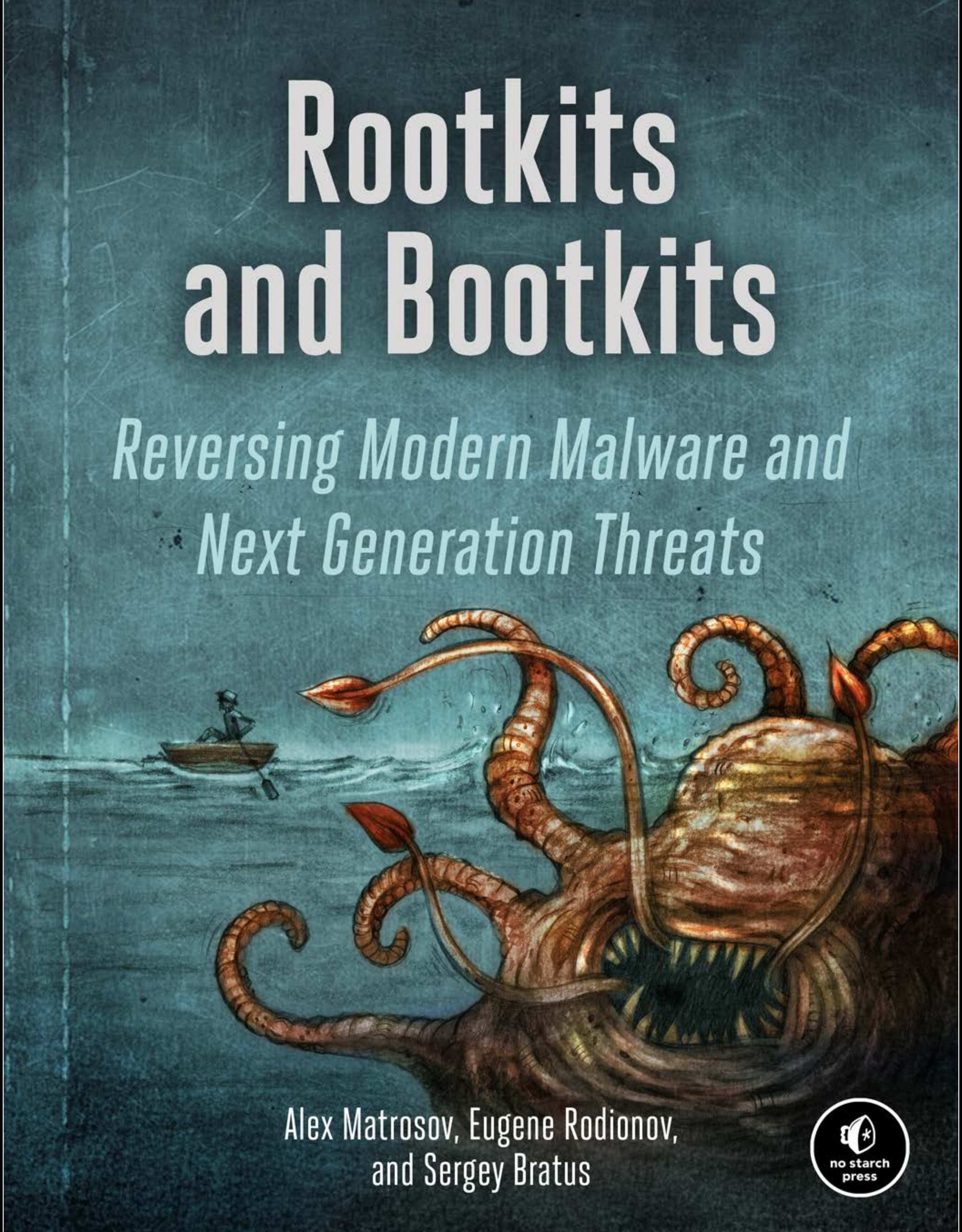
Date reported: January 20, 2016

Bounty paid: $500

An *interstitial web page* displays before expected content. Using one is a common method to protect against open redirect vulnerabilities. Any time you're redirecting a user to a URL, you can show an interstitial web page with a message explaining to the user that they're leaving the domain they're on. As a result, if the redirect page shows a fake log in or tries to pretend to be the trusted domain, the user will know that they're being redirected. This is the approach HackerOne takes when following most URLs off its site, for example, when following links in submitted reports.

Although you can use interstitial web pages to avoid redirect vulnerabilities, complications in the way sites interact with one another can still lead to compromised links. HackerOne uses Zendesk, a customer service support ticketing system, for its *https://support.hackerone.com/* subdomain. When *hackerone.com* is followed by */zendesk_session*, users are led from HackerOne's platform to HackerOne's Zendesk platform without an interstitial page because URLs containing the *hackerone.com* domain are trusted links. However, anyone can create custom Zendesk accounts and pass them to the `/redirect_to_account?state=` parameter. The custom Zendesk account can then redirect to another website not owned by Zendesk or HackerOne. Because Zendesk allows for redirecting between accounts without interstitial pages, the user can be taken to the untrusted site without warning. As a solution, HackerOne identified links containing `zendesk_session` as external links, which rendered an interstitial warning page when clicked.

In order to make this bug report, the hacker Mahmoud Jamal created an account on Zendesk with the subdomain *http://compayn.zendesk.*



Rootkits and Bootkits

Reversing Modern Malware and Next Generation Threats

Alex Matrosov, Eugene Rodionov, and Sergey Bratus

no starch press

FEST ROOTKIT: THE MOST ADVANCED SPAM AND DDOS BOT

This chapter is devoted to one of the most advanced spam and distributed denial of service (DDoS) botnets discovered—the Win32/Festi botnet, which we'll refer to simply as Festi from now on. Festi has powerful spam delivery and DDoS capabilities, as well as interesting rootkit functionality that allows it to stay under the radar by hooking into the filesystem and system registry. Festi also conceals its presence by actively counteracting dynamic analysis with debugger and sandbox evasion techniques.

From a high-level point of view, Festi has a well-designed modular architecture implemented entirely in the kernel-mode driver. Kernel-mode programming is, of course, fraught with danger: a single error in the code can cause the system to crash and render it unusable, potentially leading the user to reinstall the system afresh, wiping the malware. For this reason it's rare for spam-sending malware to rely heavily on kernel-mode programming. The fact that Festi was able to inflict so much damage is indicative of the solid technical skills of its developer(s) and their in-depth understanding of the Windows system. Indeed, they came up with several interesting architectural decisions, which we'll cover in this chapter.

The Case of Festi Botnet

The Festi botnet was first discovered in the fall of 2009, and by May 2012 it was one of the most powerful and active botnets for sending spam and performing DDoS attacks. The botnet was initially available to anyone for lease, but after early 2010 it was restricted to major spam partners only, like Pavel Vrublebsky, one of the actors that used Festi botnet for criminal activities, as detailed in the book *Spam Nation* by Brian Krebs (Sourcebooks, Inc., 2014).

According to statistics from M86 Security Labs (currently Trustwave) for 2011, shown in Figure 2-1, Festi was one of the three most active spam botnets in the world in the reported period.

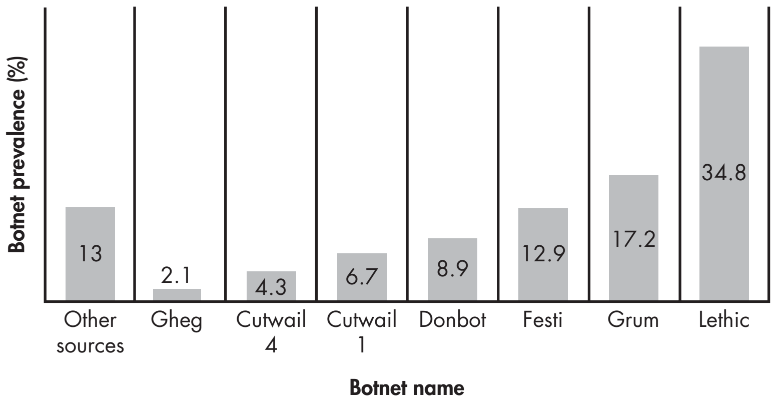


Figure 2-1: The most prevalent spam botnets according to M86 Security Labs

Festi's rise in popularity stemmed from a particular attack on Assist, a payment processing company.[1] Assist was one of the companies bidding for a contract with Aeroflot, Russia's largest airline, but a few weeks before Aeroflot was due to make its decision, cybercriminals used Festi to launch a massive DDoS attack against Assist. The attack rendered the processing system unusable for an extended period of time, eventually forcing Aeroflot to award another company the contract. This event is a prime example of how rootkits may be used in real-world crime.

1. Brian Krebs, "Financial Mogul Linked to DDoS Attacks," *Krebs on Security* blog, June 23, 2011, *http://krebsonsecurity.com/2011/06/financial-mogul-linked-to-ddos-attacks*

Dissecting the Rootkit Driver

The Festi rootkit is distributed mainly through a PPI (Pay-Per-Install) scheme similar to the TDL3 rootkit discussed in Chapter 1. The dropper's rather simple functionality installs into the system a kernel-mode driver that implements the main logic of the malware. The kernel-mode component is registered as a "system start" kernel-mode driver with a randomly generated name, meaning the malicious driver is loaded and executed at system bootup during initialization.

DROPPER INFECTOR

Dropper is a special type of infector. Droppers carry payload to the victim system within itself. Payload is frequently compressed and encrypted or obfuscated. Once executed, a dropper extracts the payload from its image and installs it on a victim system (i.e. drops it on the system – which explains the term used for this type of infector). Unlike droppers, downloaders – another type of infector – doesn't carry payload within itself but rather download it from a remote server.

The Festi botnet targets only the Microsoft Windows x86 platform and does not have a kernel-mode driver for 64-bit platforms. This was fine at the time of its distribution, as there were still many 32-bit operating systems being used, but means the rootkit has largely been rendered obsolete as 64-bit systems have outnumbered 32-bit systems.

The kernel-mode driver has two main duties: requesting configuration information from the command and control (C&C) server, and downloading and executing malicious modules in the form of plug-ins (illustrated in Figure 2-2). Each plug-in is dedicated to a certain job, such as performing DDoS attacks against a specified network resource, or sending spam to an email list provided by the C&C server.

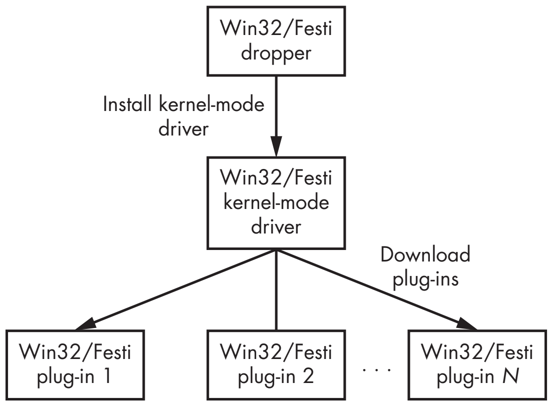


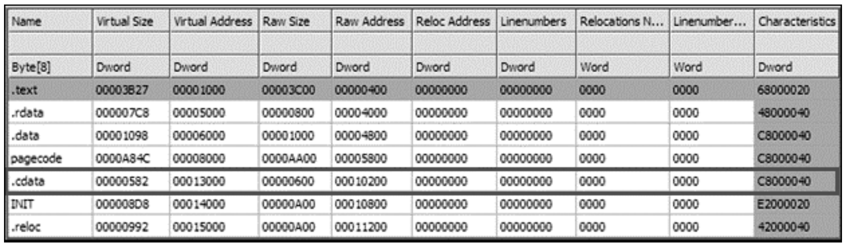
Figure 2-2: Operation of the Festi rootkit

Interestingly, the plug-ins aren't stored on the system hard drive but instead in volatile memory, meaning that when the infected computer is powered off or rebooted, the plug-ins vanish from system memory. This makes forensic analysis of the malware significantly harder since the only file stored on the hard drive is the main kernel-mode driver, which contains neither the payload nor any information on attack targets.

Festi Configuration Information for C&C Communication

To enable it to communicate with C&C server, Festi is distributed with three pieces of predefined configuration information: the domain names of C&C servers, the key to encrypt data transmitted between the bot and C&C, and the bot version information

This configuration information is hardcoded into the driver's binary. Figure 2-3 shows a section table of the kernel-mode driver with a writable section named `.cdata`, which stores the configuration data as well as strings that are used to perform the malicious activity.



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumber...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00003B27	00001000	00003C00	00000400	00000000	00000000	0000	0000	68000020
.rdata	000007C8	00005000	00000800	00004000	00000000	00000000	0000	0000	48000040
.data	00001098	00006000	00001000	00004800	00000000	00000000	0000	0000	C8000040
pagecode	0000A84C	00008000	0000AA00	00005800	00000000	00000000	0000	0000	C8000040
.cdata	00000582	00013000	00000600	00010200	00000000	00000000	0000	0000	C8000040
INIT	000008D8	00014000	00000A00	00010800	00000000	00000000	0000	0000	E2000020
.reloc	00000992	00015000	00000A00	00011200	00000000	00000000	0000	0000	42000040

Figure 2-3: Section table of Festi kernel-mode driver

The malware obfuscates the contents with a simple algorithm that XORs the data with a 4-byte key. The `.cdata` section in decrypted at the very beginning of the driver initialization.

The strings within the `.cdata` section, listed in Table 2-1, can garner the attention of security software, so obfuscating them helps the bot evade detection.

Table 2-1: Encrypted Strings in the Festi Configuration Data Section

String	Purpose
`\Device\Tcp` `\Device\Udp`	Names of device objects used by the malware to send and receive data over the network

`\REGISTRY\MACHINE\SYSTEM\`
`CurrentControlSet\Services\`
`SharedAccess\Parameters\FirewallPolicy\`
`StandardProfile\GloballyOpenPorts\List`
	Path to the registry key with the parameters of the Windows firewall, used by the malware to disable the local firewall

String	Purpose
`ZwDeleteFile, ZwQueryInformationFile, ZwLoadDriver, KdDebuggerEnabled, ZwDeleteValueKey, ZwLoadDriver`	Names of system services used by the malware

Festi's Object-Oriented Framework

Unlike many kernel-mode drivers, which are usually written in plain C using the procedural programming paradigm, the Festi driver has an object-oriented architecture. The main components (classes) of the architecture implemented by the malware are:

Memory manager Allocates and releases memory buffers

Network sockets Sends and receives data over the network

C&C protocol parser Parses C&C messages and executes received commands

Plug-in manager Manages downloaded plug-ins

The relationship between these components is illustrated in Figure 2-4.

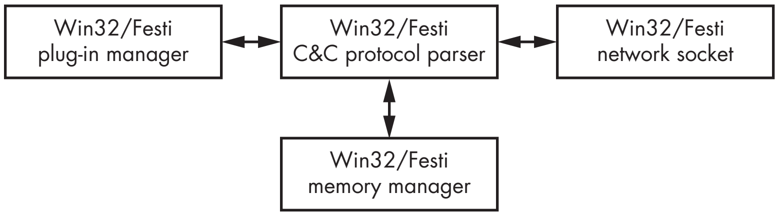

Win32/Festi plug-in manager
Win32/Festi C&C protocol parser
Win32/Festi network socket
Win32/Festi memory manager

Figure 2-4: Architecture of Festi kernel-mode driver

As you can see, the memory manager is the central component used throughout the bot by all other components.

This object-oriented approach allows the malware to be easily ported to other platforms, like Linux. To do so, an attacker would need to change only system-specific code (like the code that calls system services for memory management and network communication) that is isolated by the component's interface. Downloaded plug-ins, for instance, rely almost completely on the interfaces provided by the main module, and rarely use routines provided by the system to do system-specific operations.

Plug-in Management

Plug-ins downloaded from the C&C server are loaded and executed by the malware. To manage the downloaded plug-ins efficiently, Festi maintains an array of pointers to a specially defined `PLUGIN_INTERFACE` structure. Each structure corresponds to a particular plug-in in memory and provides the bot with specific entry points—routines responsible for handling data received from C&C, as shown in Figure 2-5. This way, Festi keeps track of all the malicious plug-ins loaded in memory.

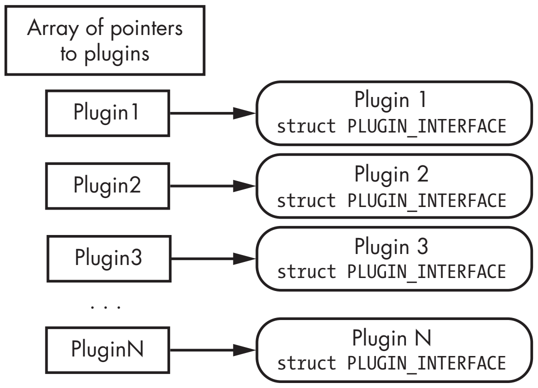

Array of pointers to plugins
Plugin1
Plugin2
Plugin3
. . .
PluginN
Plugin 1 struct PLUGIN_INTERFACE
Plugin 2 struct PLUGIN_INTERFACE
Plugin 3 struct PLUGIN_INTERFACE
Plugin N struct PLUGIN_INTERFACE

Figure 2-5: Layout of the array of pointers to PLUGIN_INTERFACE structures

Listing 2-1 shows the layout of the PLUGIN_INTERFACE structure.

```
struct PLUGIN_INTERFACE
{
  // Initialize plug-in
  PVOID Initialize;
  // Release plug-in, perform cleanup operations
  PVOID Release;
  // Get plug-in version information
  PVOID GetVersionInfo_1;
  // Get plug-in version information
  PVOID GetVersionInfo_2;
  // Write plug-in-specific information into tcp stream
  PVOID WriteIntoTcpStream;
  // Read plug-in specific information from tcp stream and parse data
  PVOID ReadFromTcpStream;
  // Reserved fields
  PVOID Reserved_1;
  PVOID Reserved_2;
};
```

Listing 2-1: Defining the PLUGIN_INTERFACE structure

The first two routines, `Initialize` and `Release`, are intended for plug-in initialization and termination, respectively. The following two routines, `GetVersionInfo_1` and `GetVersionInfo_2`, are used to obtain version information for the plug-in in question.

The routines `WriteIntoTcpStream` and `ReadFromTcpStream` are used to exchange data between the plug-in and the C&C server. When Festi transmits data to the C&C server, it runs through the array of pointers to the plug-in interfaces and executes the `WriteIntoTcpStream` routine of each registered plug-in, passing a pointer to a *TCP stream* object as a parameter. The TCP stream object implements the functionality of the network communication interface.

On receiving data from the C&C server, the bot executes the plug-ins' `ReadFromTcpStream` routine, so that the registered plug-ins can get parameters and plug-in-specific configuration information from the network stream. As a result, every loaded plug-in can communicate with the C&C server independently of all other plug-ins, which means plug-ins can be developed independently of one another, increasing the efficiency of their development and the stability of the architecture.

Built-in Plug-ins

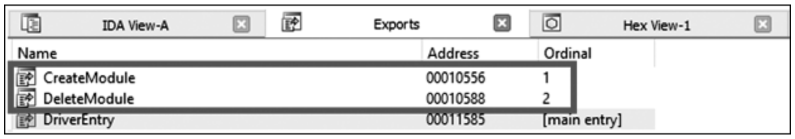
Upon installation, the main malicious kernel-mode driver implements two built-in plug-ins: the *configuration information manager* and the *bot plug-in manager*.

Configuration Manager

The configuration manager plug-in is responsible for requesting configuration information and downloading plug-ins from the C&C server. This simple plug-in periodically connects to the C&C server to download the data. The delay between two consecutive requests is specified by the C&C server itself, likely to avoid static patterns that security software can use to detect infections. We describe the network communication protocol between the bot and the C&C server in the section "Festi Network Communication Protocol" on page XX.

Plug-in Manager

The plug-in manager is responsible for maintaining the array of downloaded plug-ins. It receives remote commands from the C&C server and loads and unloads specific plug-ins, delivered in compressed form, onto the system. Each plug-in has a default entry point—`DriverEntry`—and also exports the two routines `CreateModule` and `DeleteModule`, as shown in Figure 2-6.



Name	Address	Ordinal
CreateModule	00010556	1
DeleteModule	00010588	2
DriverEntry	00011585	[main entry]

Figure 2-6: Export Address table of a Festi plug-in

LINUX BASICS FOR HACKERS

GETTING STARTED WITH *NETWORKING*, *SCRIPTING*, AND *SECURITY* IN KALI

OCCUPYTHEWEB

no starch press

8
BASH SCRIPTING

Any self-respecting hacker must be able to write scripts. For that matter, any self-respecting Linux administrator must be able to script. Hackers often need to automate commands, sometimes from multiple tools, and this is most efficiently done through short programs they write themselves.

In this chapter, we build a few simple bash shell scripts to start you off with scripting. We'll add capabilities and features as we progress, eventually building a script capable of finding potential attack targets over a range of IP addresses.

To become an *elite* hacker, you also need the ability to script in one of the widely used scripting languages, such as Ruby (Metasploit exploits are written in Ruby), Python (many hacking tools are Python scripts), or Perl (Perl is the best text-manipulation scripting language). I give a brief introduction to Python scripting in Chapter 17.

A Crash Course in Bash

A *shell* is an interface between the user and the operating system that enables you to manipulate files and run commands, utilities, programs, and much more. The advantage of a shell is that you perform these tasks immediately from the computer and not through an abstraction, like a GUI, which allows you to customize your task to your needs. A number of different shells are available for Linux, including the Korn shell, the Z shell, the C shell, and the *B*ourne-*a*gain *sh*ell, more widely known as bash.

Because the bash shell is available on nearly all Linux and UNIX distributions (including macOS and Kali), we'll be using the bash shell, exclusively.

The bash shell can run any system commands, utilities, or applications your usual command line can run, but it also includes some of its own built-in commands. Table 8-1 later in the chapter gives you a reference to some useful commands that reside within the bash shell.

In earlier chapters, you used the `cd`, `pwd`, `set`, and `umask` commands. In this section, you will be using two more commands: the `echo` command, first used in Chapter 7, which displays messages to the screen, and the `read` command, which reads in data and stores it somewhere else. Just learning these two commands alone will enable you to build a simple but powerful tool.

You'll need a text editor to create shell scripts. You can use whichever Linux text editor you like best, including vi, vim, emacs, gedit, kate, and so on. I'll be using Leafpad in these tutorials, as I have in previous chapters. Using a different editor should *not* make any difference in your script or its functionality.

Your First Script: "Hello, Hackers-Arise!"

For your first script, we will start with a simple program that returns a message to the screen that says `"Hello, Hackers-Arise!"` Open your text editor, and let's go.

To start, you need to tell your operating system which interpreter you want to use for the script. To do this, enter a *shebang*, which is a combination of a hash mark and an exclamation mark, like so:

```
#!
```

You then follow the shebang (`#!`) with `/bin/bash` to indicate that you want the operating system to use the bash shell interpreter. As you'll see in later chapters, you could also use the shebang to use other interpreters, such as Perl or Python. Here, you want to use the bash interpreter, so enter the following:

```
#! /bin/bash
```

Next, enter the echo command, which tells the system to simply repeat (or *echo*) back to your monitor whatever follows the command.

In this case, we want the system to echo back to us "Hello, Hackers-Arise!", as done in Listing 8-1. Note that the text or message we want to echo back must be in double quotation marks.

```
#! /bin/bash

# This is my first bash script. Wish me luck.

echo "Hello, Hackers-Arise!"
```

Listing 8-1: Your "Hello, Hackers-Arise!" script

Here, you also see a line that's preceded by a hash mark (#). This is a *comment*, which is a note you leave to yourself or anyone else reading the code to explain what you're doing in the script. Programmers use comments in every coding language. These comments are not read or executed by the interpreter, so you don't need to worry about messing up your code. They are visible only to humans. The bash shell knows a line is a comment if it starts with the # character.

Now, save this file as *HelloHackersArise* with no extension and exit your text editor.

Setting Execute Permissions

By default, a newly created bash script is not executable even by you, the owner. Let's look at the permissions on our new file in the command line by using cd to move into the directory and then entering ls -l. It should look something like this:

```
kali >ls -l
--snip--
-rw-r--r-- 1 root root 42 Oct 22 14:32 HelloHackersArise
--snip--
```

As you can see, our new file has rw-r--r-- (644) permissions. As you learned in Chapter 5, this means the owner of this file only has read (r) and write (w) permissions, but no execute (x) permissions. The group and all other users have only read permissions. We need to give ourselves execute permissions in order to run this script. We change the permissions with the chmod command, as you saw in Chapter 5. To give the owner, the group, and all others execute permissions, enter the following:

```
kali >chmod 755 HelloHackersArise
```

Now when we do a long listing on the file, like so, we can see that we have execute permissions:

```
kali >ls -l
--snip--
-rwx r-x r-x 1 root root 42 Oct 22 14:32 HelloHackersArise
--snip--
```

The script is now ready to execute!

Running HelloHackersArise

To run our simple script, enter the following:

```
kali >./HelloHackersArise
```

The `./` before the filename tells the system that we want to execute this script in the file *HelloHackersArise* from the current directory. It also tells the system that if there is another file in another directory named *HelloHackersArise*, please ignore it and only run *HelloHackersArise* in the current directory. It may seem unlikely that there's another file with this name on your system, but it's good practice to use the `./` when executing files, as this localizes the file execution to the current directory and many directories will have duplicate filenames, such as *start* and *setup*.

When we press ENTER, our very simple script returns our message to the monitor:

```
Hello, Hackers-Arise!
```

Success! You just completed your first shell script!

Adding Functionality with Variables and User Input

So, now we have a simple script. All it does is echo back a message to standard output. If we want to create more advanced scripts, we will likely need to add some variables.

A *variable* is an area of storage that can hold something in memory. That "something" might be some letters or words (strings) or numbers. It's known as a variable because the values held within it are changeable; this is an extremely useful feature for adding functionality to a script.

In our next script, we will add functionality to prompt the user for their name, place whatever they input into a variable, then prompt the user for the chapter they're at in this book, and place that keyboard input into a variable. After that, we'll echo a welcome message that includes their name and the chapter back to the user.

Open a new file in your text editor and enter the script shown in Listing 8-2.

```
❶ #! /bin/bash

❷ # This is your second bash script. In this one, you prompt /
  # the user for input, place the input in a variable, and /
  # display the variable contents in a string.

❸ echo "What is your name?"

  read name

❹ echo "What chapter are you on in Linux Basics for Hackers?"

  read chapter

❺ echo "Welcome" $name "to Chapter" $chapter "of Linux Basics for Hackers!"
```

Listing 8-2: A simple script making use of variables

We open with `#! /bin/bash` to tell the system we want to use the bash interpreter for this script ❶. We then add a comment that describes the script and its functionality ❷. After that, we prompt the user for their name and ask the interpreter to read the input and place it into a variable we call `name` ❸. Then we prompt the user to enter the chapter they are currently working through in this book, and we again read the keyboard input into a variable, this time called `chapter` ❹.

In the final line, we construct a line of output that welcomes the reader by their name to the chapter they are on ❺. We use the `echo` command and provide the text we want to display on the screen in double quotes. Then, to fill in the name and chapter number the user entered, we add the variables where they should appear in the message. As noted in Chapter 7, to use the values contained in the variables, you must precede the variable name with the `$` symbol.

Save this file as *WelcomeScript.sh*. The *.sh* extension is the convention for script files. You might have noticed we didn't include the extension earlier; it's not strictly required, and if you leave the extension off, the file will save as a shell script file by default.

Now, let's run this script. Don't forget to give yourself execute permission with `chmod` first; otherwise, the operating system will scold you with a `Permission denied` message.

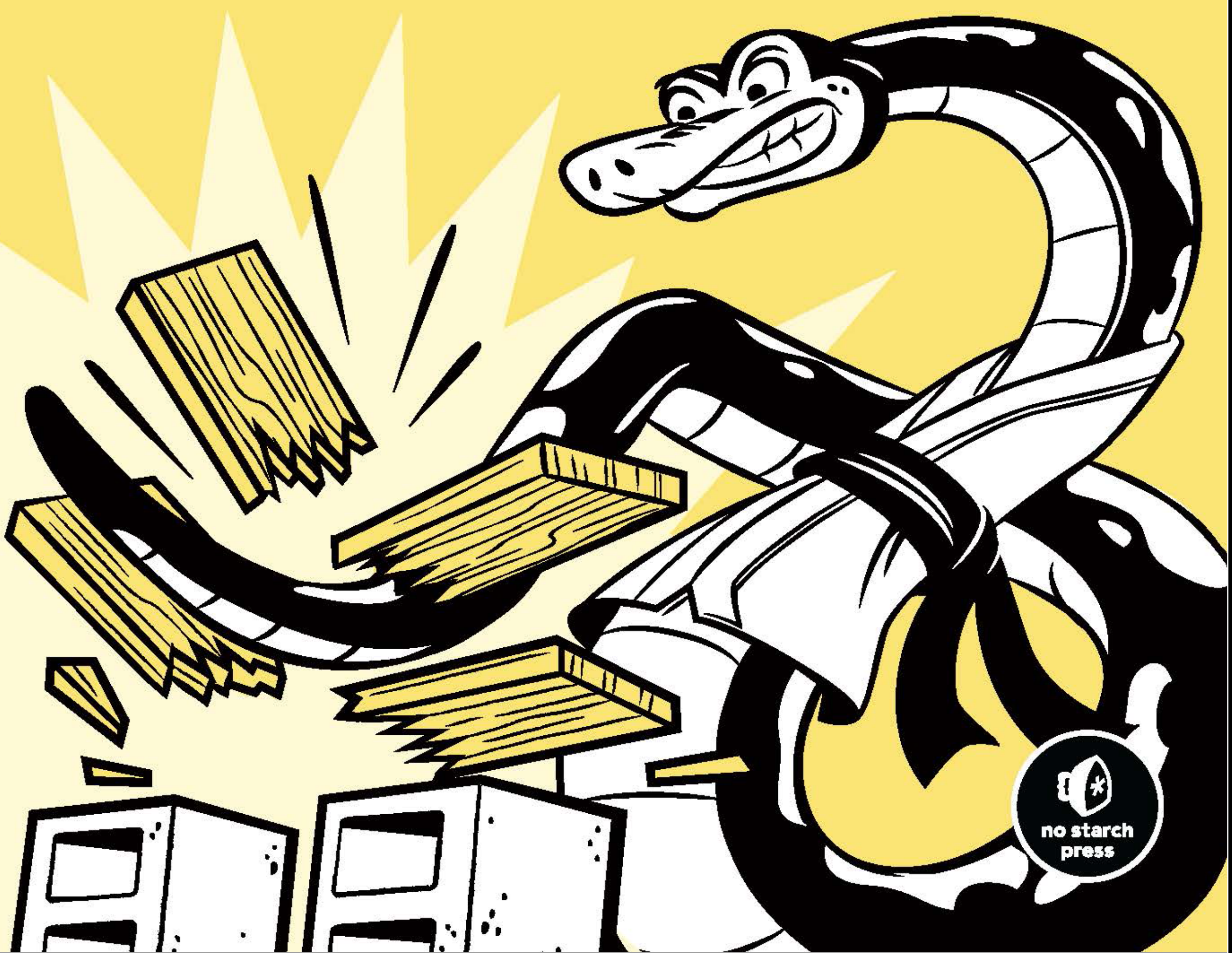
```
kali >./WelcomeScript.sh
What is your name?
OccupytheWeb
What chapter are you on in Linux Basics for Hackers?
8
Welcome OccupytheWeb to Chapter 8 of Linux Basics for Hackers!
```

As you can see, your script took input from the user, placed it into variables, and then used those inputs to make a greeting for the user.

SERIOUS PYTHON

BLACK-BELT ADVICE ON DEPLOYMENT, SCALABILITY, TESTING, AND MORE

JULIEN DANJOU



no starch press

3

DOCUMENTATION AND GOOD API PRACTICE

In this chapter, we'll discuss documentation; specifically, how to automate the trickier and more tedious aspects of documenting your project with *Sphinx*. While you will still have to write the documentation yourself, Sphinx will simplify your task. As it is common to provide features using a Python library, we'll also look at how to manage and document your public API changes. Because your API will have to evolve as you make changes to its features, it's rare to get everything built perfectly from the outset, but I'll show you a few things you can do to ensure your API is as user-friendly as possible.

We'll end this chapter with an interview with Christophe de Vienne, author of the Web Services Made Easy framework, in which he discusses best practices for developing and maintaining APIs.

Documenting with Sphinx

Documentation is one of the most important parts of writing software. Unfortunately, a lot of projects don't provide proper documentation. Writing documentation is seen as complicated and daunting, but it doesn't have to be: with the tools available to Python programmers, documenting your code can be just as easy as writing it.

One of the biggest reasons for sparse or nonexistent documentation is that many people assume the only way to document code is by hand. Even with multiple people on a project, this means one or more of your team will end up having to juggle contributing code with maintaining documentation—and if you ask any developer which job they'd prefer, you can be sure they'll say they'd rather write software than write *about* software.

Sometimes the documentation process is completely separate from the development process, meaning that the documentation is written by people who did not write the actual code. Furthermore, any documentation produced this way is likely to be out-of-date: it's almost impossible for manual documentation to keep up with the pace of development, regardless of who handles it.

Here's the bottom line: the more degrees of separation between your code and your documentation, the harder it will be to keep the latter properly maintained. So why keep them separate at all? It's not only possible to put your documentation directly in the code itself, but it's also simple to convert that documentation into easy-to-read HTML and PDF files.

The most common format for Python documentation is *reStructuredText*, or *reST* for short. It's a lightweight markup language (like Markdown) that's as easy to read and write for humans as it is for computers. Sphinx is the most commonly used tool for working with this format; Sphinx can read reST-formatted content and output documentation in a variety of other formats.

I recommend that your project documentation always include the following:

- The problem your project is intended to solve, in one or two sentences.
- The license your project is distributed under. If your software is open source, you should also include this information in a header in each code file; just because you've uploaded your code to the Internet doesn't mean that people will know what they're allowed to do with it.
- A small example of how your code works.
- Installation instructions.
- Links to community support, mailing list, IRC, forums, and so on.
- A link to your bug tracker system.
- A link to your source code so that developers can download and start delving into it right away.

You should also include a *README.rst* file that explains what your project does. This README should be displayed on your GitHub or PyPI project page; both sites know how to handle reST formatting.

NOTE *If you're using GitHub, you can also add a* CONTRIBUTING.rst *file that will be displayed when someone submits a pull request. It should provide a checklist for users to follow before they submit the request, including things like whether your code follows PEP 8 and reminders to run the unit tests. Read the Docs* (http://readthedocs.org/) *allows you to build and publish your documentation online automatically. Signing up and configuring a project is straightforward. Then Read the Docs searches for your Sphinx configuration file, builds your documentation, and makes it available for your users to access. It's a great companion to code-hosting sites.*

Getting Started with Sphinx and reST

You can get Sphinx from *http://www.sphinx-doc.org/*. There are installation instructions on the site, but the easiest method is to install with `pip install sphinx`.

Once Sphinx is installed, run `sphinx-quickstart` in your project's top-level directory. This will create the directory structure that Sphinx expects to find, along with two files in the *doc/source* folder: *conf.py*, which contains Sphinx's configuration settings (and is absolutely required for Sphinx to work), and *index.rst*, which serves as the front page of your documentation. Once you run the quick-start command, you'll be taken through a series of steps to designate naming conventions, version conventions, and options for other useful tools and standards.

The *conf.py* file contains a few documented variables, such as the project name, the author, and the theme to use for HTML output. Feel free to edit this file at your convenience.

Once you've built your structure and set your defaults, you can build your documentation in HTML by calling `sphinx-build` with your source directory and output directory as arguments, as shown in Listing 3-1. The command `sphinx-build` reads the *conf.py* file from the source directory and parses all the *.rst* files from this directory. It renders them in HTML in the output directory.

```
$ sphinx-build doc/source doc/build
  import pkg_resources
Running Sphinx v1.2b1
loading pickled environment... done
No builder selected, using default: html
building [html]: targets for 1 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
preparing documents... done
writing output... [100%] index
writing additional files... genindex search
```

```
copying static files... done
dumping search index... done
dumping object inventory... done
build succeeded.
```

Listing 3-1: Building a basic Sphinx HTML document

Now you can open *doc/build/index.html* in your favorite browser and read your documentation.

NOTE *If you're using* `setuptools` *or* `pbr` *(see Chapter 5) for packaging, Sphinx extends them to support the command* `setup.py build_sphinx`*, which will run* `sphinx-build` *automatically. The* `pbr` *integration of Sphinx has some saner defaults, such as outputting the documentation in the* /doc *subdirectory.*

Your documentation begins with the *index.rst* file, but it doesn't have to end there: reST supports `include` directives to include reST files from other reST files, so there's nothing stopping you from dividing your documentation into multiple files. Don't worry too much about syntax and semantics to start; reST offers a lot of formatting possibilities, but you'll have plenty of time to dive into the reference later. The complete reference (*http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html*) explains how to create titles, bulleted lists, tables, and more.

Sphinx Modules

Sphinx is highly extensible: its basic functionality supports only manual documentation, but it comes with a number of useful modules that enable automatic documentation and other features. For example, `sphinx.ext.autodoc` extracts reST-formatted docstrings from your modules and generates *.rst* files for inclusion. This is one of the options `sphinx-quickstart` will ask if you want to activate. If you didn't select that option, however, you can still edit your *conf.py* file and add it as an extension like so:

```
extensions = ['sphinx.ext.autodoc']
```

Note that `autodoc` will *not* automatically recognize and include your modules. You need to explicitly indicate which modules you want documented by adding something like Listing 3-2 to one of your *.rst* files.

```
.. automodule:: foobar
❶    :members:
❷    :undoc-members:
❸    :show-inheritance:
```

Listing 3-2: Indicating the modules for `autodoc` *to document*

In Listing 3-2, we make three requests, all of which are optional: that all documented members be printed ❶, that all undocumented members be printed ❷, and that inheritance be shown ❸. Also note the following:

- If you don't include any directives, Sphinx won't generate any output.
- If you only specify `:members:`, undocumented nodes on your module, class, or method tree will be skipped, even if all their members are documented. For example, if you document the methods of a class but not the class itself, `:members:` will exclude both the class and its methods. To keep this from happening, you'd have to write a docstring for the class or specify `:undoc-members:` as well.
- Your module needs to be where Python can import it. Adding `.`, `..`, and/or `../..` to `sys.path` can help.

The `autodoc` extension gives you the power to include most of your documentation in your source code. You can even pick and choose which modules and methods to document—it's not an "all-or-nothing" solution. By maintaining your documentation directly alongside your source code, you can easily ensure it stays up to date.

Automating the Table of Contents with autosummary

If you're writing a Python library, you'll usually want to format your API documentation with a table of contents containing links to individual pages for each module.

The `sphinx.ext.autosummary` module was created specifically to handle this common use case. First, you need to enable it in your *conf.py* by adding the following line:

```
extensions = ['sphinx.ext.autosummary']
```

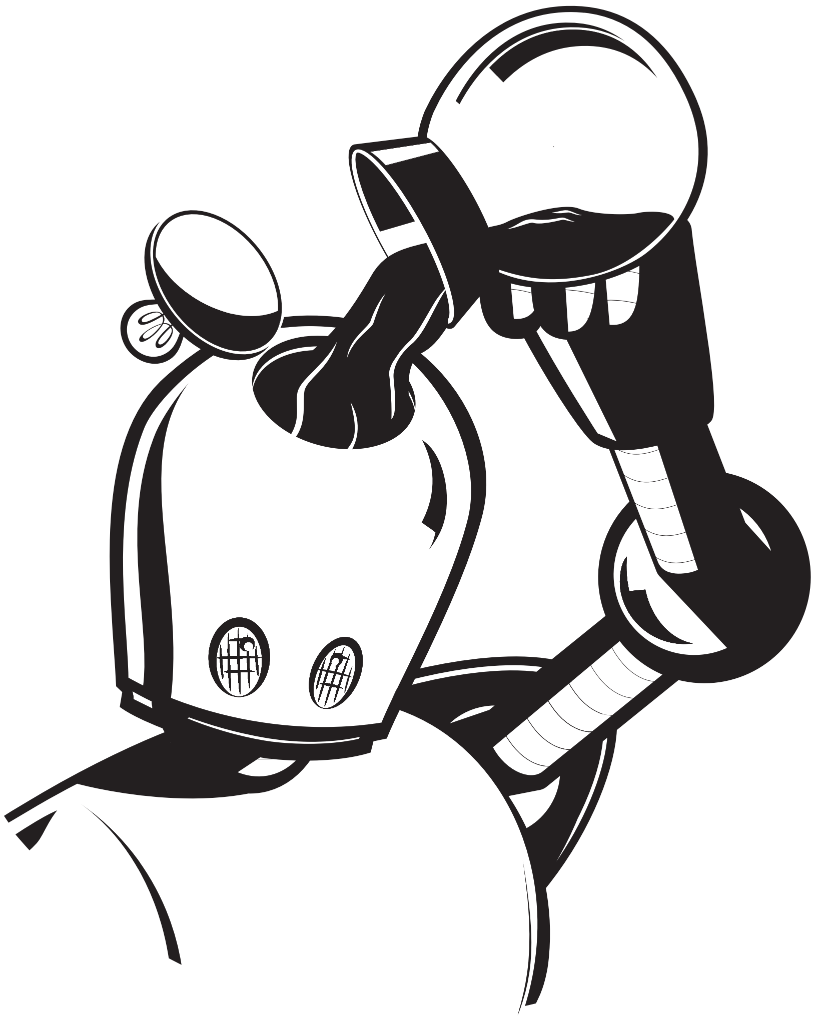
Then, you can add something like the following to an *.rst* file to automatically generate a table of contents for the specified modules:

```
.. autosummary::

   mymodule
   mymodule.submodule
```

This will create files called *generated/mymodule.rst* and *generated/mymodule.submodule.rst* containing the autodoc directives described earlier. Using this same format, you can specify which parts of your module API you want included in your documentation.

NOTE *The `sphinx-apidoc` command can automatically create these files for you; check out the Sphinx documentation to find out more.*



no starch press