**COMP 4108 Assignment 2**

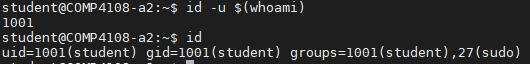
Shane Bishop

Student #: 101030053

# Part A

**Question 1**

My UID (user ID) is 1001. I can retrieve this in two ways, as shown in the screenshot below.



id -u $(whoami) prints my UID, and id without any arguments prints my UID, primary GID (group ID), and then a list of all of my groups and GIDs.

The $(command) syntax is a subshell. Subshells are used to run a command and feed that output to further commands. Subshells can be nested. In this instance, the bash shell runs whoami and then substitutes the output of whoami to generate the command id -u student, and then the shell runs that command.

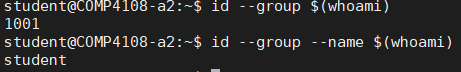
I can also find my UID by running cat /etc/passwd | grep $(whoami), in which case my UID is the number after the second colon (the number after the third colon is my primary GID).



Why did I run these commands? Because I knew id was the command to run to see a user’s UID. I also knew user data, including the user’s UID, is stored in /etc/passwd.

**Question 2**

My primary group name is student, and my primary GID is 1001, as shown in the screenshot below.



I can also get this information using id or /etc/passwd, as described in Question 1 above (although my primary group name would not be in /etc/passwd, it is in /etc/group).

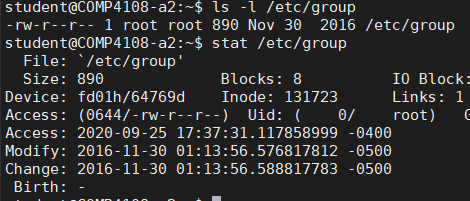
Why did I run these commands? Because I knew id was the command to run to see a user’s UID. I also knew user data, including the user’s GID, is stored in /etc/passwd.

**Question 3**

The filesystem path of the Linux group file is /etc/group.

**Question 4**

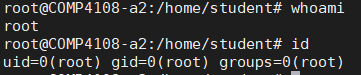
As seen in the screenshot below, the permissions of the Linux group file is -rw-r--r-- (this is shown in both the output of ls and of stat). The octal representation of these permissions is 0644, as shown on the first Access line in the output of stat.



Why did I run these commands? I already knew ls -l printed the permissions string. I am unfamiliar with the stat command, but I was told to use it by the assignment. I am familiar with the stat syscall (system call) though, which I know gives information about a file, including permissions information. Also, from looking at the stat man page, I known that it is used for displaying file status.

**Question 5**

The GID of the root group is 0. I was able to check this by switching to root, and then running id. The root user has a primary group named “root”, and as can be seen in the output below, the GID of the root group is 0.



I can also find this using /etc/group. The group man page says each line in /etc/group has the format group\_name:password:GID:user\_list. So in the output below, the number after the second colon is the root group’s GID.



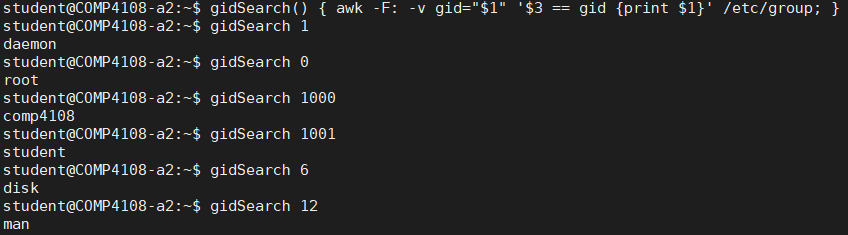
**Question 6**

Using my knowledge of /etc/group (described in question 5), I found the shadow group’s GID is 42.



**Question 7**

Here is my screenshot with 5 invocations:



Breakdown of gidSearch(): Everything can be done with a single line of awk.

-F: tells awk to use a colon as the field separator.

-v gid="$1" tells awk to pass the value of $1 (the single argument to gidSearch()) as a variable gid.

Then the actual awk script (the portion between single quotes) tells awk to print the first column (with the group name) if the third column (the group ID) matches the gid variable.

Finally, the /etc/group at the end tells awk to do its work on that file.

I didn’t bother including a script file with gidSearch() because I included it in my screenshot above, and I’m also including it inline right here:

gidSearch() {

awk -F: -v gid="$1" '$3 == gid {print $1}' /etc/group

}

**Question 8**

For all parts of this question, I used the find man page to learn the different options for find. To save space I have copy-pasted output rather than using screenshots.

Part A

student@COMP4108-a2:~$ sudo find /A2/Haystack -user comp4108

/A2/Haystack

/A2/Haystack/Bar

/A2/Haystack/Bar/Charlie

/A2/Haystack/Bar/Alpha

/A2/Haystack/Foo

/A2/Haystack/Foo/S/S

/A2/Haystack/Foo/P/P

/A2/Haystack/Foo/J/J

/A2/Haystack/Foo/D/D

/A2/Haystack/Foo/E/E

/A2/Haystack/Foo/F/F

/A2/Haystack/Foo/M/M

/A2/Haystack/Foo/B/B

/A2/Haystack/Foo/W/W

/A2/Haystack/Foo/Z/Z

/A2/Haystack/Foo/Q/Q

/A2/Haystack/Foo/O/O

/A2/Haystack/Foo/L/L

/A2/Haystack/Foo/X/X

/A2/Haystack/Foo/U/U

/A2/Haystack/Foo/A/A

/A2/Haystack/Foo/V/V

/A2/Haystack/Foo/I/I

/A2/Haystack/Foo/C/C

/A2/Haystack/Foo/H/H

/A2/Haystack/Foo/N/N

/A2/Haystack/Foo/K/K

/A2/Haystack/Foo/R/R

/A2/Haystack/Foo/G/G

/A2/Haystack/Baz

/A2/Haystack/Baz/Beta

/A2/Haystack/Baz/Charlie

Part B

student@COMP4108-a2:~$ sudo find /A2/Haystack -group root | wc -l

422

Part C

student@COMP4108-a2:~$ sudo find /A2/Haystack -user sshd

/A2/Haystack/Foo/S/D

/A2/Haystack/Foo/S/W

/A2/Haystack/Foo/S/X

/A2/Haystack/Foo/S/C

/A2/Haystack/Foo/S/R

/A2/Haystack/Foo/P/B

/A2/Haystack/Foo/P/W

/A2/Haystack/Foo/P/T

/A2/Haystack/Foo/P/I

/A2/Haystack/Foo/P/C

/A2/Haystack/Foo/P/N

/A2/Haystack/Foo/J/P

/A2/Haystack/Foo/J/B

/A2/Haystack/Foo/J/O

/A2/Haystack/Foo/D/J

/A2/Haystack/Foo/D/M

/A2/Haystack/Foo/D/T

/A2/Haystack/Foo/D/U

/A2/Haystack/Foo/D/A

/A2/Haystack/Foo/E/S

/A2/Haystack/Foo/E/D

/A2/Haystack/Foo/E/F

/A2/Haystack/Foo/E/W

/A2/Haystack/Foo/E/V

/A2/Haystack/Foo/F/Q

/A2/Haystack/Foo/M/J

/A2/Haystack/Foo/M/B

/A2/Haystack/Foo/M/I

/A2/Haystack/Foo/M/H

/A2/Haystack/Foo/M/N

/A2/Haystack/Foo/M/K

/A2/Haystack/Foo/B/P

/A2/Haystack/Foo/B/M

/A2/Haystack/Foo/B/Q

/A2/Haystack/Foo/B/U

/A2/Haystack/Foo/B/R

/A2/Haystack/Foo/W/E

/A2/Haystack/Foo/W/O

/A2/Haystack/Foo/W/U

/A2/Haystack/Foo/W/I

/A2/Haystack/Foo/Z/W

/A2/Haystack/Foo/Z/L

/A2/Haystack/Foo/Z/X

/A2/Haystack/Foo/Z/Y

/A2/Haystack/Foo/Q/E

/A2/Haystack/Foo/Q/H

/A2/Haystack/Foo/O/D

/A2/Haystack/Foo/O/B

/A2/Haystack/Foo/O/Q

/A2/Haystack/Foo/O/V

/A2/Haystack/Foo/O/C

/A2/Haystack/Foo/O/R

/A2/Haystack/Foo/L/J

/A2/Haystack/Foo/L/E

/A2/Haystack/Foo/L/F

/A2/Haystack/Foo/L/M

/A2/Haystack/Foo/L/U

/A2/Haystack/Foo/L/I

/A2/Haystack/Foo/L/C

/A2/Haystack/Foo/L/N

/A2/Haystack/Foo/L/G

/A2/Haystack/Foo/X/S

/A2/Haystack/Foo/X/L

/A2/Haystack/Foo/T/S

/A2/Haystack/Foo/T/P

/A2/Haystack/Foo/T/E

/A2/Haystack/Foo/T/F

/A2/Haystack/Foo/T/Z

/A2/Haystack/Foo/T/X

/A2/Haystack/Foo/T/V

/A2/Haystack/Foo/U/J

/A2/Haystack/Foo/U/Q

/A2/Haystack/Foo/U/T

/A2/Haystack/Foo/U/G

/A2/Haystack/Foo/A/J

/A2/Haystack/Foo/A/X

/A2/Haystack/Foo/A/V

/A2/Haystack/Foo/A/C

/A2/Haystack/Foo/A/Y

/A2/Haystack/Foo/A/R

/A2/Haystack/Foo/V/S

/A2/Haystack/Foo/V/W

/A2/Haystack/Foo/V/N

/A2/Haystack/Foo/I/J

/A2/Haystack/Foo/I/D

/A2/Haystack/Foo/I/W

/A2/Haystack/Foo/I/Z

/A2/Haystack/Foo/I/G

/A2/Haystack/Foo/C/J

/A2/Haystack/Foo/C/D

/A2/Haystack/Foo/C/M

/A2/Haystack/Foo/C/B

/A2/Haystack/Foo/C/L

/A2/Haystack/Foo/C/H

/A2/Haystack/Foo/C/N

/A2/Haystack/Foo/H/S

/A2/Haystack/Foo/H/P

/A2/Haystack/Foo/H/M

/A2/Haystack/Foo/H/O

/A2/Haystack/Foo/H/V

/A2/Haystack/Foo/N/E

/A2/Haystack/Foo/N/M

/A2/Haystack/Foo/N/O

/A2/Haystack/Foo/N/T

/A2/Haystack/Foo/N/U

/A2/Haystack/Foo/N/I

/A2/Haystack/Foo/N/R

/A2/Haystack/Foo/N/G

/A2/Haystack/Foo/Y/S

/A2/Haystack/Foo/Y/P

/A2/Haystack/Foo/Y/J

/A2/Haystack/Foo/Y/M

/A2/Haystack/Foo/Y/T

/A2/Haystack/Foo/Y/U

/A2/Haystack/Foo/K/D

/A2/Haystack/Foo/K/F

/A2/Haystack/Foo/K/W

/A2/Haystack/Foo/K/L

/A2/Haystack/Foo/K/R

/A2/Haystack/Foo/R/F

/A2/Haystack/Foo/R/A

/A2/Haystack/Foo/R/H

/A2/Haystack/Foo/G/D

/A2/Haystack/Foo/G/W

/A2/Haystack/Foo/G/O

/A2/Haystack/Foo/G/L

/A2/Haystack/Foo/G/U

/A2/Haystack/Foo/G/A

Part D

student@COMP4108-a2:~$ sudo find /A2/Haystack -perm 777

/A2/Haystack/Bar/Beta

/A2/Haystack/Foo/F

/A2/Haystack/Foo/U/M

/A2/Haystack/Foo/U/Y

/A2/Haystack/Foo/H/P

/A2/Haystack/Foo/H/H

/A2/Haystack/Baz/Charlie

**Question 9**

The command would be

find /A2/Haystack -type d -perm 777 -exec sudo chmod 750 {} +

The first argument to find is the path to do the operations on.

The -type d indicates to only perform operations on directories (the question says to only change **directories** with permissions 777).

The -perm 777 finds “files” (in this case directories) with 777 permissions.

The -exec sudo chmod 750 {} + runs sudo chmod 750 on the find results. I could have used the more common -exec command ; style, but the -exec command {} + style is more efficient. Rather than running command once for each find result, it runs the command once with all the results.

**Question 10**

Setup

student@COMP4108-a2:~$ mkdir -p top/middle/bottom

student@COMP4108-a2:~$ mkdir -p top/middle\_two/bottom\_two/end\_of\_line

student@COMP4108-a2:~$ mkdir -p top/middle\_three

student@COMP4108-a2:~$ tree top

top

├── middle

│   └── bottom

├── middle\_three

└── middle\_two

└── bottom\_two

└── end\_of\_line

6 directories, 0 files

I used mkdir -p to create the directories, then tree to confirm directory structure.

Part A

student@COMP4108-a2:~$ chmod 664 top/middle/bottom/ top/middle\_two/bottom\_two/

chmod modifies the permissions of files and directories. Here I set the permissions to 664.

Part B

student@COMP4108-a2:~$ touch top/middle\_three/foo.txt

student@COMP4108-a2:~$ chmod ug+x top/middle\_three/foo.txt

student@COMP4108-a2:~$ ls -l top/middle\_three/foo.txt

-rw**x**rw**x**r-- 1 student student 0 Sep 25 21:30 top/middle\_three/foo.txt

Here I created the empty foo.txt using touch, then I used ug+x to set (+) the executable (x) permission for user (u) and group (g). I then confirmed that the user execute bit was set (see red x above) and the group execute bit was set (see green x above)

Part C

student@COMP4108-a2:~$ sudo chown root top

student@COMP4108-a2:~$ ls -l | grep top | awk '{print $3}'

root

Here I changed top’s owner to root with chown (short for **ch**ange **own**er). I then confirmed this worked using awk (ls -l prints the owner at the third column).

Part D

student@COMP4108-a2:~$ sudo chgrp www-data top

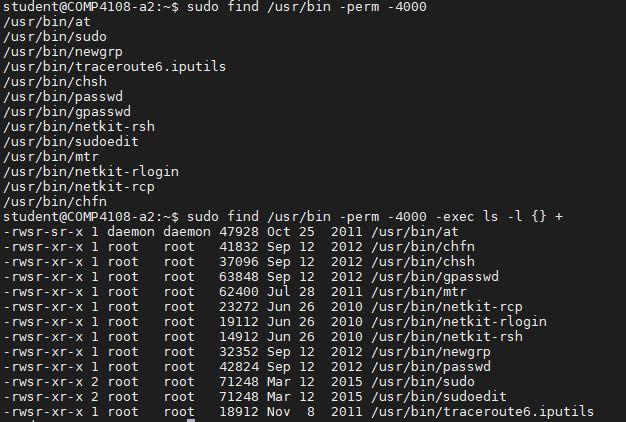
student@COMP4108-a2:~$ ls -l | grep top | awk '{print $4}'

www-data

Here I changed top’s group to www-data with chgrp (short for **ch**ange **gr**ou**p**). I then confirmed this worked using awk (ls -l prints the group at the fourth column).

**Question 11**

Part A



I ran two commands. The first command answers this question, the second command was just to sanity check.

The first command uses find to search /usr/bin. The argument to -perm is -4000 (with the leading dash) because the leading dash effectively tells find to search for files with 4\*\*\* permissions (this means find should accept all files regardless of ugo permissions on the file). I used sudo to ensure find wouldn’t hit any permissions issues in subdirectories.

The second command is just a sanity check, where every file found by find has its permissions string displayed to confirm they all have their setuid bit set. In this instance, all of the files have the same permissions except for /usr/bin/at.

Part B

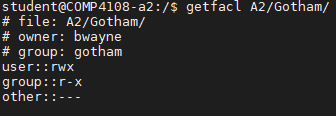
For my discussion, I am focussing on sudo (found at /usr/bin/sudo on the assignment VM). The setuid bit causes the program that the setuid bit is set on to always run as the owner of that file. In the case of sudo, sudo is used to run commands as a different user than the current user (the user must be a member of the sudo group to run the sudo command). Often sudo is used to run single commands as root. Therefore, sudo has its setuid bit set with root as its owner so that sudo always runs as root, so that sudo can run the command it’s given to run as the specified user. (sudo at least always starts running as root, and then the effective UID might be changed, after sudo has started, to the desired user).

sudo defaults to running the command it is given as the root user, but if a different username is given using either the -U or -u flag, the command will be run as the specified user.

# Part B

**Question 1**

The ACL (access control list) for /A2/Gotham is in the screenshot below.



I knew to use getfacl because the getfacl man page says getfacl gets a file’s ACL (this makes sense – presumably getfacl is short for **get f**ile **ACL**).

**Question 2**



The -R flag makes chmod make modifications recursively through the directory and its subdirectories.

o+rx means add (+) rx permissions for the other (o) category.

I used sudo so I wouldn’t get any permissions errors.

For sanity, I print the exit code of chmod with echo $? to be sure it succeeded (an exit code of 0 means success, a non-zero exit code usually means an error occurred).

**Question 3**

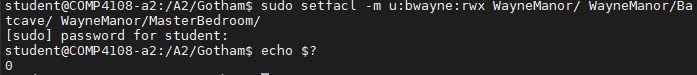


The -m flag is short for --modify, both of which modify the ACL of specified files

The u:jgordon:rw expression means, for the user with jgordon username, give that user rw permissions on all of the provided files.

Finally, setfacl can take a list of files to modify, so I provided the path to each directory I wanted to modify.

**Question 4**



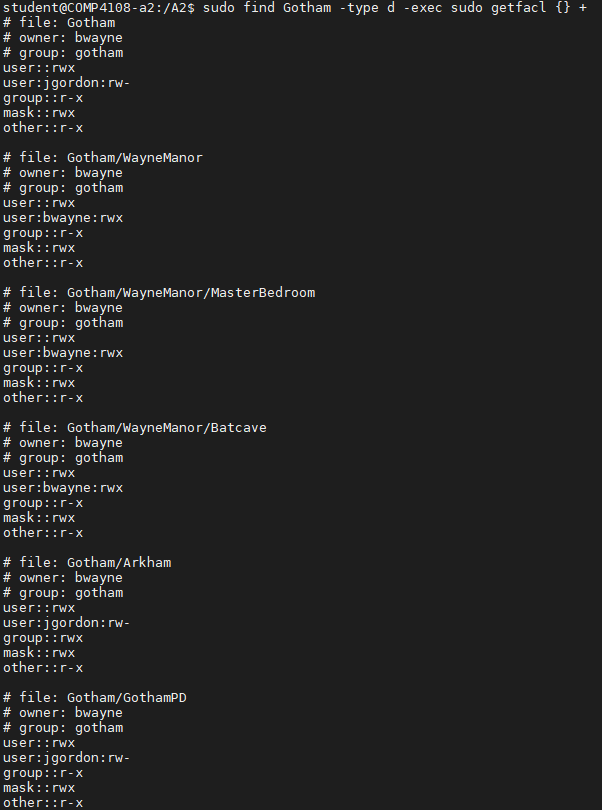
This is basically the same as for question 3. I use the -m flag to modify, and I use sudo so I don’t get any permissions errors. The u:bwayne:rwx string given as an argument to the -m flag means to give the bwayne user read, write, and execute permissions on all items. Then I provide a list of items to apply the ACL to. I finally check the exit code of setfacl for sanity.

**Question 5**



The setfacl -x option is used for removing ACLs. The rest should be clear from the description given for question 4 above.

**Question 6**



Here I use find with its -exec flag to run getfacl on every file in /A2/Gotham. I use the + syntax, which I describe in question 9 of part A. I use -type d with find to get back only directories, because the question only asks for the ACL of directories, and not of other types of “files”.

Why use sudo on find? Since the x bit on directories influences the ability to search directories, I used sudo to ensure I can search the entire directory. I similarly used sudo on the getfacl command to be sure I get the full ACL for each directory (which may not have been necessary).

# Part C

**Part A**

Using strace, I was able to determine where the debug file was (irrelevant output is omitted):

student@COMP4108-a2:~$ strace /A2/Racing/Slow/vuln\_slow 0 hello > /dev/null

execve("/A2/Racing/Slow/vuln\_slow", <omitted for space>) = 0

…

open("/home/student/.debug\_log", O\_RDWR|O\_CREAT|O\_APPEND, 0666) = 3

access("/home/student/.debug\_log", W\_OK) = 0

…

write(1, "Sleeping for 0 seconds.\n", 24) = 24

write(1, "Writing debug message.\n", 23) = 23

open("/home/student/.debug\_log", O\_RDWR|O\_CREAT|O\_APPEND, 0666) = 4

…

write(4, "hello\n", 6) = 6

close(4) = 0

…

write(1, "Goodbye!\n", 9) = 9

exit\_group(0) = ?

I redirect stdout to /dev/null so the output of vuln\_slow is not mixed with the output of strace.

From this output, we can see that first the program is loaded into memory with execve() (the process was spawned with fork(), and strace starts tracing after the fork(), but before any exec\*() syscalls). Then /home/student/.debug\_log is opened once with a file descriptor (fd) of 3, which is followed by an access() syscall. The process then writes some messages to stdout, and then opens /home/student/.debug\_log again, this time getting an fd of 4 back (the open() call is even given the same arguments). The program then writes “hello” to fd 4 and closes the file. Finally, the program prints “Goodbye!” and exits.

With this, I know now that the program is writing to /home/student/.debug\_log which I own, as shown in the screenshot below (ls -l prints the owner at column three):



If I run strace again, but with a sleep time greater than 0, I see the following additional syscall:

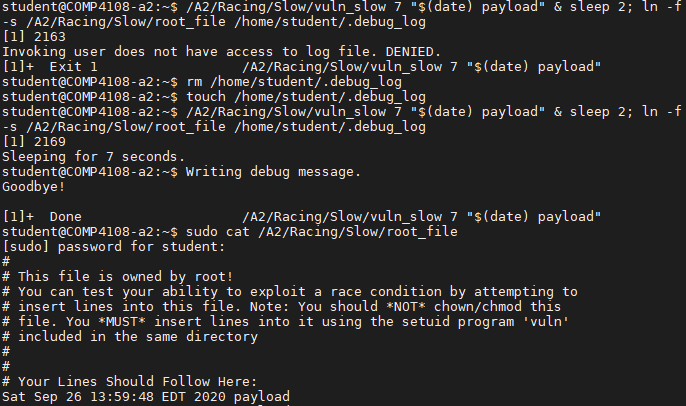
nanosleep({2, 0}, 0x7ffd1b109620) = 0

This call to nanosleep() is just before opening /home/student/.debug\_log. This must be when the program is sleeping (i.e., running idle). I want to exploit the program during this window.

Next, I found the path to root\_file:



This is a screenshot of how I exploited the program:



I ran vuln\_slow in the background (using the & shell syntax), telling it to sleep for 7 seconds. Immediately after that, I paused for 2 seconds (the idea was that would be long enough for vuln\_slow to have started its sleep), and then I used ln to create my symlink (symbolic link). The -f flag for ln is the “force” flag, which forces the link file to be overwritten; I used this to avoid needing to run both rm and ln. The -s flag for ln is to create a symlink.

Then the output looks a bit confusing. The bash shell returns a prompt, because all foreground processes have finished running. However, the background vuln\_slow process is still running, and it then prints its last two messages. I hit Enter, which causes bash to print that the vuln\_slow process has terminated (bash always does this when background programs finish running). I then printed the contents of the root\_file using cat (with sudo of course) to confirm my payload with the timestamp was there, and sure enough it was!

I now describe how this attack works.

Looking at the output of ls, we can see vuln\_slow is owned by root, and that its setuid bit (the s in the first rws indicates that the setuid bit is set) is set:



This means that this program always runs with an effective UID of 0 (the root user has a UID of 0).

When a setuid program is run, it runs with the owner’s UID as its effective UID (in this case, the effective UID is 0), and with the executing user’s UID as its real UID (in this case, 1001). So eUID=0 and rUID=1001.

From the man page for the access() syscall, it says that the check performed by access() is based on the process’s real UID and GID, rather than the effective IDs. From the output of strace above, we can see that the access() syscall is checking if the rUID has write access (the mode parameter is W\_OK).

So essentially what the program does is it checks if the rUID has write permissions on the file. If the rUID does have write permissions, it sleeps for the specified amount of time, and then does the write. If the rUID does not have write permissions, it terminates immediately with an error message:



So here is what happens with my attack, in chronological order:

1. vuln\_slow checks the file with access(), the check passes
2. vuln\_slow starts its sleep
3. I symlink the file vuln\_slow will open after its sleep to point to a file I do not have write access to
4. vuln\_slow finishes its sleep
5. vuln\_slow opens the file, which follows the symlink to the root-owned file
6. vuln\_slow, assuming I must have write access since the access() check passed earlier, proceeds to write to the file, inserting my payload

Part B

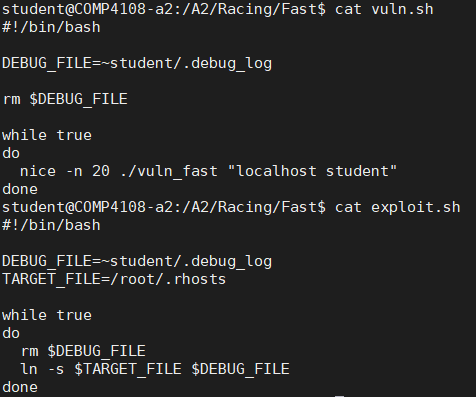
I already knew the home directory of the root user is usually /root, but I checked just in case:



The sixth column in /etc/passwd is always the user’s home directory, so I was correct that the root’s home directory is /root.

From the rhosts man page, I knew the target file must be /root/.rhosts, and that the .rhosts file uses a hostname [username] format, so my payload would be localhost student.

I then modified vuln.sh and exploit.sh as instructed, with this being the result:



In both scripts for DEBUG\_FILE I used an aspect of ~ that is less well-known. The expression ~username is expanded by some Unix shells, including bash, to the full path (without the ~) of that user’s home directory. I can prove this:



As both of these scripts are only 10 lines long, and they are displayed in the screenshot above, I am not including the two scripts as separate files. Also, their source is included inline below.

Contents of vuln.sh (blank lines removed):

#!/bin/bash

DEBUG\_FILE=~student/.debug\_log

rm $DEBUG\_FILE

while true

do

nice -n 20 ./vuln\_fast "localhost student"

done

Contents of exploit.sh (blank lines removed):

#!/bin/bash

DEBUG\_FILE=~student/.debug\_log

TARGET\_FILE=/root/.rhosts

while true

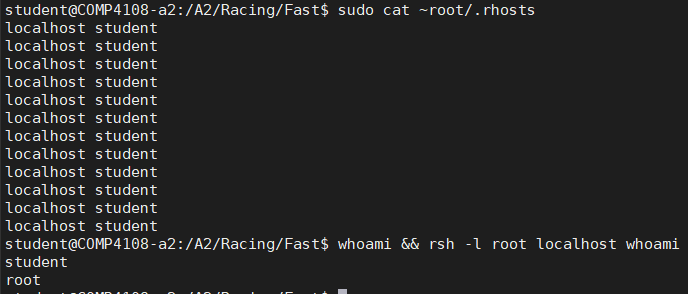
do

rm $DEBUG\_FILE

ln -s $TARGET\_FILE $DEBUG\_FILE

done

I then ran vuln.sh in one terminal and exploit.sh in another terminal, as instructed. I waited quite a bit longer than a minute, to be sure it would work. I then terminated both scripts and checked to see if I had succeeded (as we can see, I didn’t need to wait as long as I did, since the .rhosts file was modified multiple times):



The two whoamis show that, as the student user, I could run rsh, which then gave me root access.

I already explained how setuid works in part A above. We can confirm that vuln\_fast is also owned by root and has its setuid bit set:



We needed to use the nice program to slow the vuln\_fast program down. On Unix, every process has its own nice value. This nice value affects the CPU priority. The nice value can be set either using the nice() syscall, or specified directly from the command line with the nice program, as done here. The vuln\_fast program was deliberately set to have a low CPU priority to increase the chance of success with the exploit. I didn’t need the nice program for part A because I could specify the sleep time, but now I need it because vuln\_fast does not pause its execution.

If vuln\_fast does not pause its execution at all, then how could we ever expect a race condition? Well, every program only has its instructions executed on the CPU for a certain period of time, and then the CPU runs other programs. When the other programs are running, the program that does not have the CPU is idle. So the exploit succeeds when the vuln\_fast program does not have the CPU between the access() system call and the write() system call, if the exploit.sh script succeeded with the symlink operation during that small window. This window is increased in size by using the nice program.

The two scripts involved tight loops so the exploit could be attempted as many times as possible before the scripts were terminated. If the vuln.sh script paused execution, that would be wasted time where the exploit has no chance of succeeding (since the vulnerable program isn’t running). Likewise, if the exploit.sh script paused execution, then it would miss executions of the vulnerable program that it might succeed on.

We expected to successfully exploit the vulnerability after roughly a minute of execution on the VM because the assignment VMs aren’t very powerful. In more powerful machines, in an attempt to slow a vulnerable program down, tasks with high CPU usage could be run.

How is rsh being abused? I explained the symlinking process in my answer to part A above. This time the target file is one used by the rsh program. With rsh, entries in the .rhosts file allow listed users to execute commands over rsh without the target user’s password. The goal, knowing this, was to abuse the vuln\_fast program in order to write to the root user’s .rhosts file, so that we could gain root access using rsh.