**TOCTOU Race Condition Attack Lab**

*50.005 Computer System Engineering*

***Due date: 25 March 08:30 AM (Week 9)***

[Outline](#_pckct1z1y98f)

[Getting Started](#_ayjir8e06gjo)

[Grading -- [50 pts]](#_knltmoc6mpjv)

[Submission Procedure](#_rrx7joq8cqav)

[Background](#_ybylh6bup881)

[Setup](#_wrnd5tsw0nkb)

[File Permission](#_trayj0h5kslh)

[The SUID bit](#_x8bvb3hw1ft8)

[The Vulnerable Program](#_5f80zzrl4w15)

[The TOCTOU Bug](#_675uvoaao3x3)

[The Symbolic Link Program](#_3w6xeir666fu)

[The Attack](#_vtjq3lalag96)

[The Fix](#_cypx7e86ufc)

[Summary](#_ccrmzmabo3po)

# Outline

In this lab, you are tasked to investigate a program with TOCTOU (Time of Check - Time of Use) race-condition vulnerability.

The lab is written entirely in C, and it is more of **an investigative lab** with fewer coding components as opposed to our previous lab. At the end of this lab, you should be able to:

* Understand what is a TOUTOU bug and why is it prone to attacks
* Detect race-condition caused by the TOCTOU bug
* Provide a fix to this TOCTOU vulnerability
* Examine file permissions and modify them
* Understand the concept of **‘privileged programs’:** user levelvs root level
* Compile programs and make documents with different privilege level
* Understand how **sudo** works
* Understand the difference between symbolic and hard links
* Write a .c program that creates a symbolic link to existing file

# Getting Started

Clone the files:

git clone <https://github.com/natalieagus/50005Lab3.git>

**Closely follow the instructions** given in this handout. **DO NOT create your own script for submission. DO NOT modify any of the makefile either.**

**WARNING: This assignment cannot be done in WSL because the access() system call does NOT work the same way as it does on the original Linux / UNIX kernel. You can either :**

1. **Borrow your friends’ mac to run the commands.**
2. **Install Ubuntu (dual boot), you can find the guide** [**here**](https://itsfoss.com/install-ubuntu-dual-boot-mode-windows/)**.**
3. **Use a VM such as VirtualBox. There’s plenty of guides on the internet. You can find them** [**here**](https://itsfoss.com/install-linux-in-virtualbox/)**.**
4. **Setup cloud services such as EC2**

## 

# Grading

The points awarded in this lab are written in each of the sections below. The number of points in total for this lab is **20 points.**

# Submission Procedure

1. **This is an individual assignment.**
2. Export this handout as a word document and write your answers for each question in blue.
3. Export as pdf and **ZIP** it (not rar, or any other compression algorithm)
4. **Upload** to @csesubmitbot telegram bot using the command /submitlab3
5. **CHECK** your submission by using the command /checksubmission

# Background

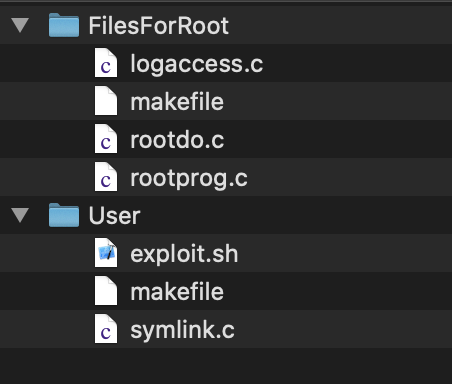
Recall that a **race condition** occurs when two or more threads (or processes) access and perform non-atomic operations on a **shared** variable (be it across threads or processes) value at the same time. Since we cannot control the *order of execution*, we can say that **the threads / processes race to modify the value of the shared variable**.

The final value of the shared variable therefore can be **non deterministic**, depending on the particular **order** in which the access takes place. In other words, the cause of race condition is due to the fact that the function performed on the shared variable is *non-atomic*.

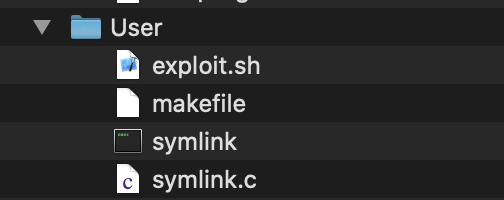
In this lab we are going to exploit a program that is **vulnerable to race-condition**. The program alone is **single-threaded**, so in the absence of an attacker, there’s nothing wrong with the program. The program however is *vulnerable* because an attacker can exploit the fact that the program can be subjected to race-condition.

### Setup

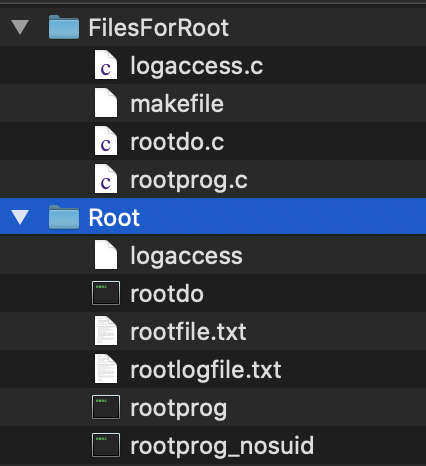
After downloading the files, you should find that the following files are given to you.



There are two folders, Root and User. Now go to the User folder and call make. You should have the following output:



**Now login as a root user (see the guide** [**below**](#_kb5fl480gwq4)**)**, go to the FilesForRoot folder, and call make. This should create a new Root folder with the following contents:



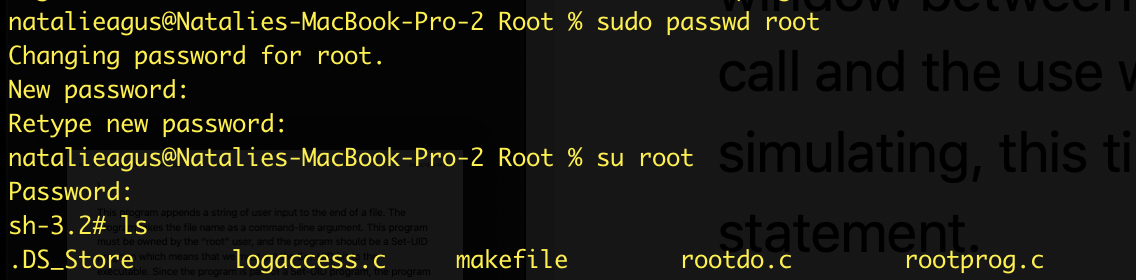
If you are already a **root** user, then you need to create a normal user account. Usually, you are logged in as a normal user.

### Switching to Root User

**To switch to the root, you must first set the password for the root account if you have not done so:**

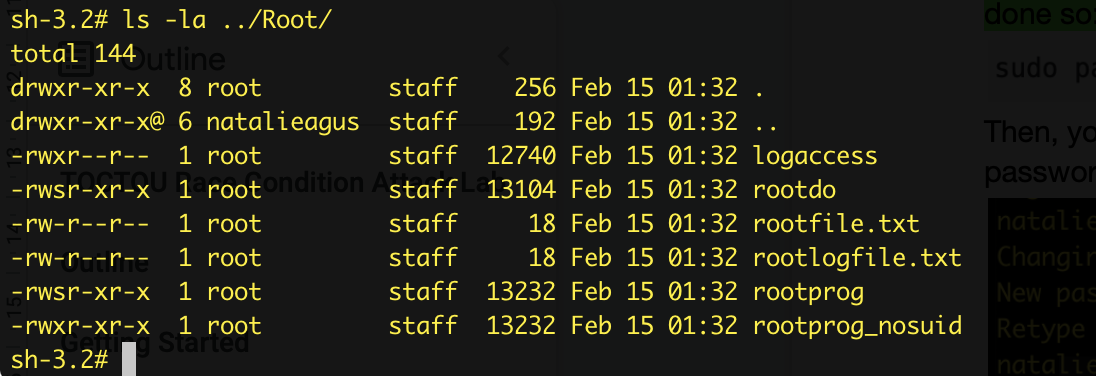


Then, you can switch to root using su root (note: you might be asked for your user password first if you haven’t called sudo recently):



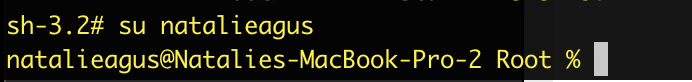
Notice the different prompt style, e.g: sh-3.2#. This means that you are logged in as a root user. Note that your own machine will look slightly different, but typically it has the # sign when you’re logged in as root. Root user is the user with the **highest (administrative) privilege**.

After ‘make’, type the command ls -la ../Root/ to see the file complete information:



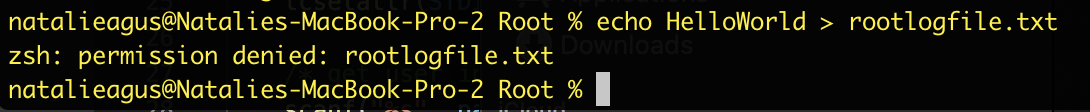
It is important to check that the newly created files belong to the user root as shown, and that the Root folder **also belongs to Root.**

You can switch back to normal user by using the same su command:



### File Permission

After switching back to normal user, try to **overwrite** one of the text files belonging to the root. You will face the **permission denied** message. This is because only root user has the write access, and other users can only read as indicated here:



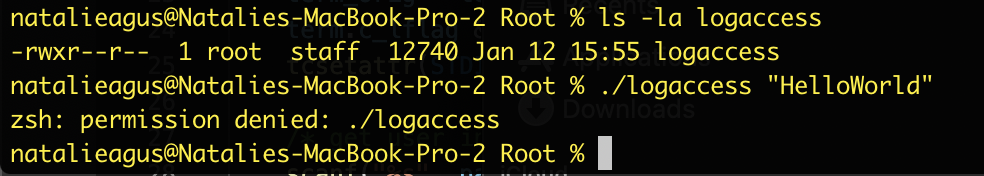
The file permission : - rw- r-- r-- means:

* **The first dash:** it is not a directory. If it is a directory, it will be written as ‘d’
* **The next three values:** rw- means that the file **owner** (root) can read, and write. The dash means that the file is NOT an executable
* **The second set of three values** in blue: r-- means that users in the same **group** can only read the file but cannot write any values into it. You can tell that the file’s group is called **staff**
* **The third set of three values** in red: r-- means that **others** (the rest of the users) can also only read the file but cannot write any values into it.

File permission can be set in C scripts using **octal** notation, e.g if this file can be executed, read, and write by owner but only read by the rest of the users, then the file permission becomes 0644, where the first 0 indicates octal notation for the permission.

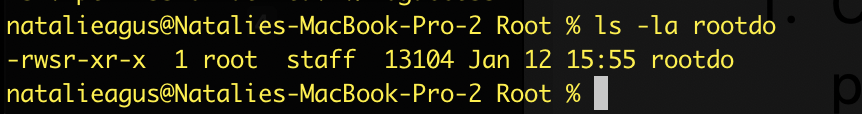
List the file permission for other file in the directory, e.g: the logaccess

This program, if executed successfully, can modify rootlogfile.txt with the message that we enter, so it should be run as ./logaccess <message> (read logaccess.c to see how it works)



Notice that we have an ‘x’. It means that the file is an **executable,** and that only **root** can execute it. If you are logged in as a normal user and tried to execute logaccess, then you will be also met with the **permission denied** message.

### The SUID bit

List the file permission for rootdo file:

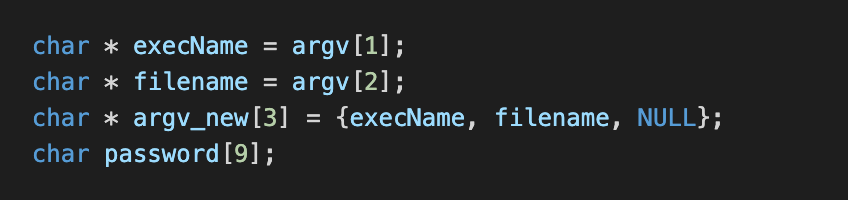
Unlike logaccess, other users are allowed to **execute** this file. More importantly, notice that instead of an ‘x’, we have an ‘s’ type of permission listed from the root.

**This is called the SUID bit.**

This **bit** allows normal user to gain **elevated privilege** when **executing** this program. If a normal user executes this program, this program runs in **root privileges (basically, the creator of the program)**. Let’s examine what rootdo does in the first place. Open rootdo.c and read what is it actually doing, especially this part of the code:

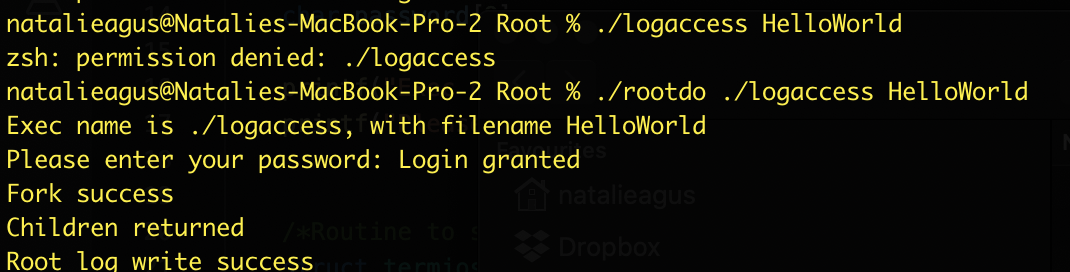


It scans for user input and store it in a buffer called password. Then, it compares whether the content of the buffer is the string *“password”.* If it is, it forks, and execute the program name that’s given as the third and fourth argument in the command line:



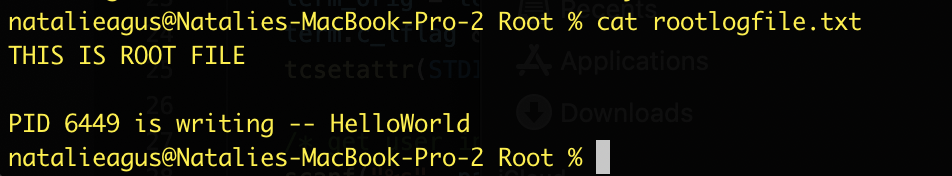
Recall that a process can create another process. This means that rootdo may fork() and call execvp() on the logaccess program successfully even though a normal user is the one who executes the rootdo in the first place and not the root user.

Let’s put this program into test:



At first, we tried to execute ./logaccess with argument “HelloWorld”, and met with the permission denied message because as we know it, logaccess can only be executed by root.

However, when we invoke rootdo and tells it to execute logaccess with “HelloWorld” as argument, we are prompted for the password (which we can just type “*password”* and enter it). Afterwhich, it seems like we can successfully write to the root log file:



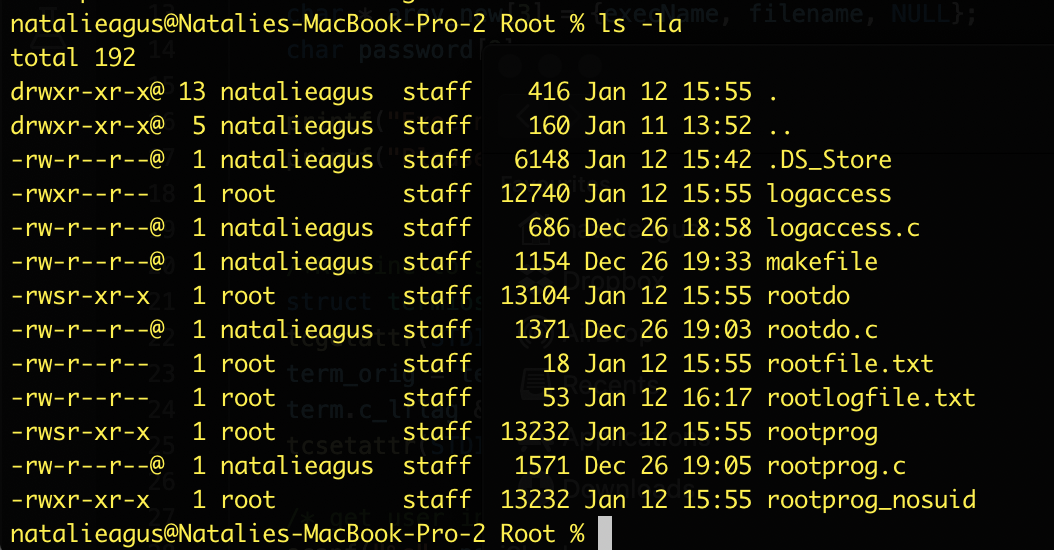
So how is it possible that we can execute the **logaccess** program while still logged in as a normal user? **This is thanks to the SUID bit being set for the rootdo program:**

* Upon successful password “verification”,
* It forks, and execute logaccess program **with root privileges**
* As the SUID bit of rootdo program is set, it **always runs with root privileges** regardless of which user executes the program

While rootdo seems like a **dangerous** program, don’t forget that the root itself was the one who made it and set the SUID bit in the first place, so yes it is indeed meant to run that way.

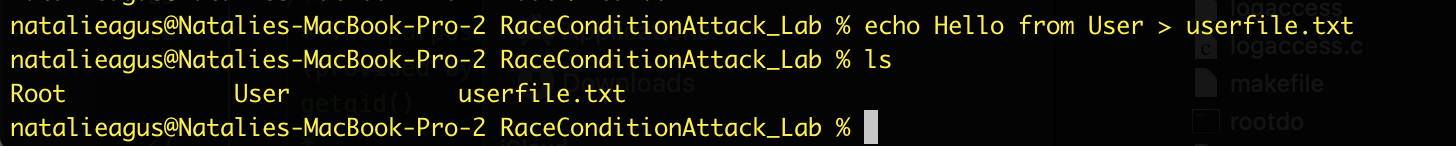
You did this when you logged in as root in the [earlier](#_wrnd5tsw0nkb) section and typed “make”. One of the tasks in the makefile is to set the SUID bit of the rootdo program.

This is in fact how your **sudo** program works. When you typo **sudo <command>,** it prompts you your password, then the program checks whether the user is verified, before executing with root privileges.

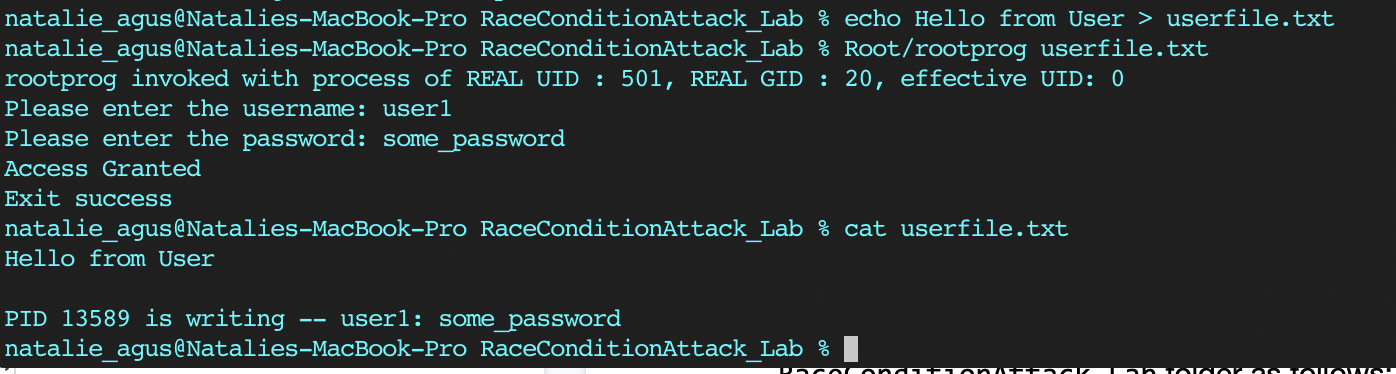
Now that you know what is the **SUID** bit, take note of the two versions of rootprog: one with SUID and one without. The one with the SUID is our **vulnerable program** that we will exploit in the next section. 

# The Vulnerable Program

Our vulnerable program is called rootprog. Let’s try and see what it does by executing it, but before that we need to create a normal user text file in the RaceConditionAttack\_Lab folder as follows:

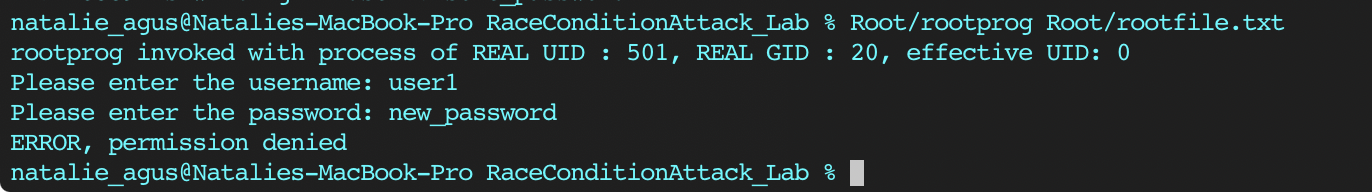


Then, we execute rootprog using userfile.txt as its argument. **From this point onwards, keep your current directory at the RaceConditionAttack\_Lab (not Root, and not User folder).**



**It will prompt you to type in *username* and then *password*, simulating some kind of program that allows us to create user / modify user with password very simply.**

But if we invoke rootprog again with a text file belonging to root to modify, we are faced with a “permission denied” message.



**WARNING: This assignment cannot be done in WSL because the access() system call does NOT work the same way as it does on the original Linux / UNIX kernel. You can either:**

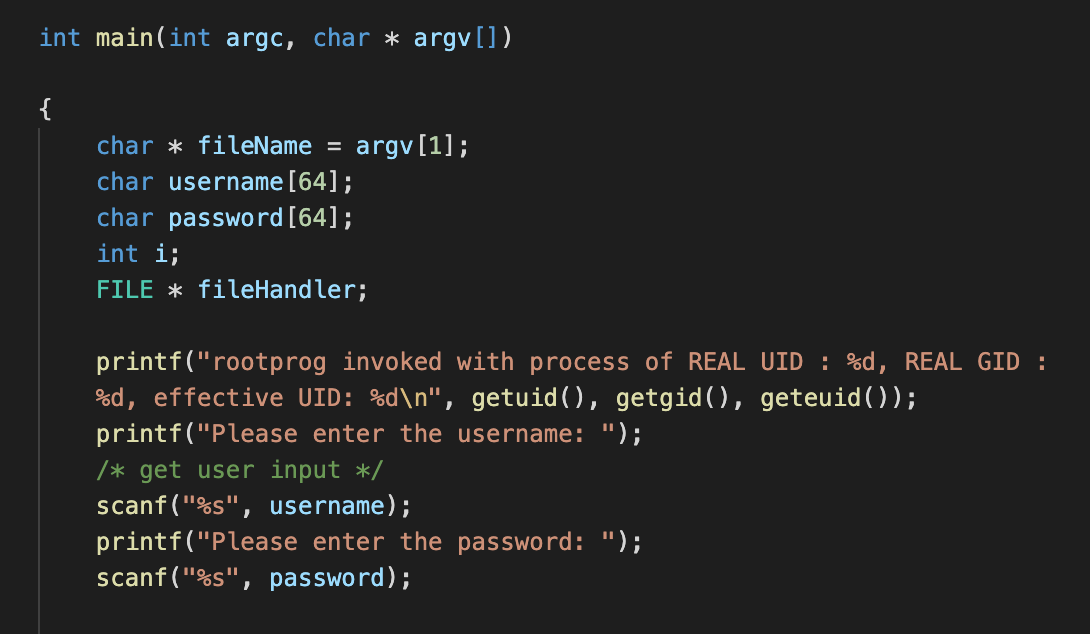
1. **Borrow your friends’ mac to run the commands**
2. **Install Ubuntu (dual boot), you can find the guide** [**here**](https://itsfoss.com/install-ubuntu-dual-boot-mode-windows/)**.**
3. **Use VM such as VirtualBox. There’s plenty of guides on the internet. You can find them** [**here**](https://itsfoss.com/install-linux-in-virtualbox/)**.**

This is because rootprog:

1. **Checks if the calling user has permission to the file requested**
2. **If yes, write to file, else print “permission denied”**

Open rootprog.c and read what its main() function does:

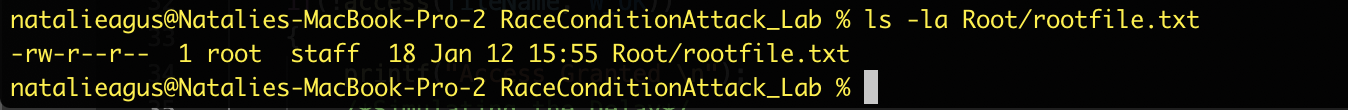
1. Stores argv[1] as fileName
2. Scans user input twice, once for username, the other for password and store it inside username and password buffers.

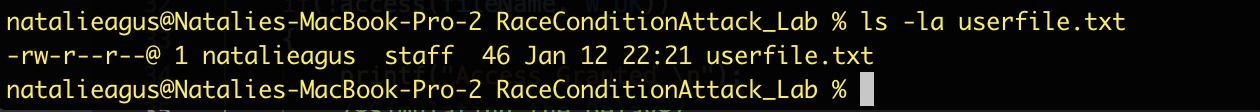


1. It uses the access system call to check if the **REAL** **calling user (not the *effective* user)** has access permission to the file being requested.



Obviously rootfile.txt can only modified by root user only,



While userfile.txt can be overwritten by the normal user: 

The access system call determines whether or not we have the **actual permission** to open the file later on using fopen and write into it using fwrite.

Read the documentation (<http://man7.org/linux/man-pages/man2/access.2.html>) of access() carefully, especially this part:

*“access() checks whether the* ***calling process can access*** *the file pathname. If pathname is a symbolic link, it is dereferenced.*

*The check is done using the calling process's real UID and GID, rather than the effective IDs as is done when actually attempting an operation (e.g.,* [*open*](http://man7.org/linux/man-pages/man2/open.2.html)*, fopen, execvp, etc) on the file. Similarly, for the root user, the check uses the set of permitted capabilities rather than the set of effective capabilities; and for non-root users, the check uses an empty set of capabilities.*

*This allows set-user-ID programs and capability-endowed programs to easily determine the invoking user's authority. In other words, access() does not answer the "can I read/write/execute this file?" question. It answers a slightly different question: "(assuming I'm a setuid binary)* ***can the user who invoked me read/write/execute this file****?", which gives set-user-ID programs the possibility to* ***prevent*** *malicious users from causing them to read files which users shouldn't be able to read.”*

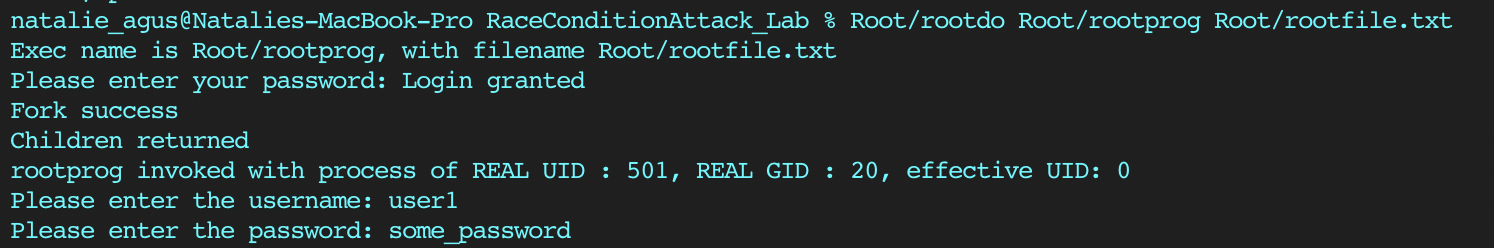
This is why in the previous section, we can use rootdo (with SUID being set), to execute (using execvp) logaccess program. This successfully allow the *elevated user* to write to rootlogfile.txt using logaccess program.

This is because in logaccess, we simply perform open to rootlogfile.txt. The open, execvp, etc system call, unlike the access system call, only checks the ***effective ID*** *of the calling process*, and **not the REAL ID**.

rootdo runs with *effective* **root privileges (effective, not real, since the caller to rootdo is only normal user),** and that’s enough to run logaccess program since it doesn’t utilise access to check for the calling process.

On the other hand, rootprog **tries** to be more secure by using the access system call to **prevent** users with elevated privileges to modify files that do not belong to them. However, it ends up being **susceptible to a particular race condition attack due this weakness called TOCTOU (time-of-check time-of-update).**

Consider calling rootprog using rootdo like how we called logaccess before.



***[2pt]*** *Can we try to run rootprog from rootdo and attempt to write something onto rootfile.txt? Do you think the message “helloFromUser” can be written onto rootfile.txt? rootdo is the one that calls execvp to execute rootprog, and rootprog of course runs with root privileges since its SUID bit is set.*

**[Q1] Your answer: No. As the rootprog checks for the real ID instead of the effective ID, it denies the permission of rootdo.**

***[1pt]*** *If writing is successful, will the entire file rootfile.txt be overwritten with the new sentence from buffer or is the content of buffer appended onto the end of the file?*

**[Q2] Your answer: The content of the buffer will be appended onto the end of the file while the original content of the file will remain unchanged.**

***[1pt]*** *Can root user overwrite this userfile.txt, although it belongs to normal user and not root?*

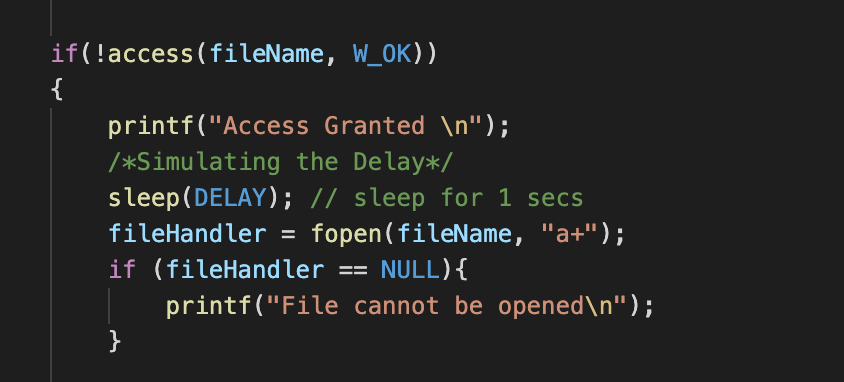
**[Q3] Your answer: Yes.**

# The TOCTOU Bug

The time-of-check to time-of-use (TOCTOU, TOCTTOU or TOC/TOU) is a class of software bug **caused by a race condition** involving:

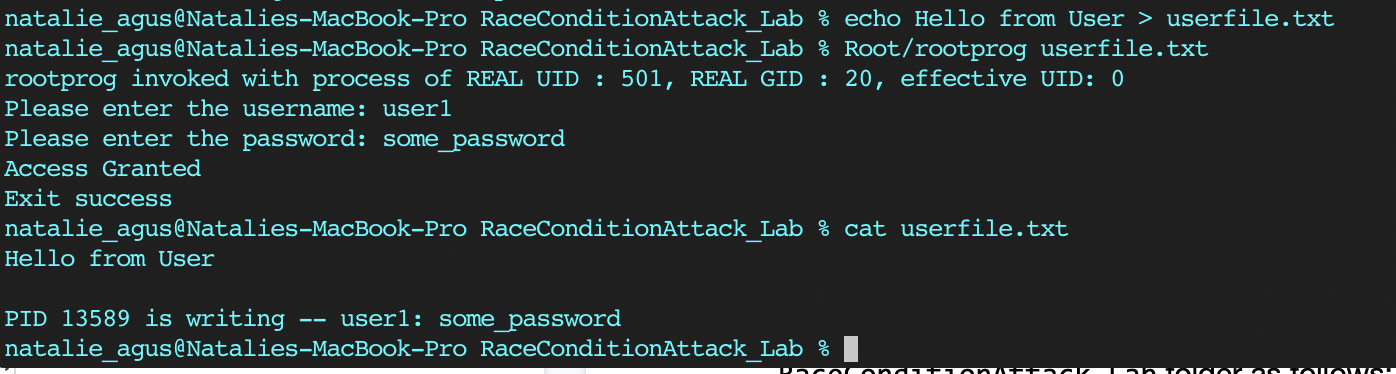
1. **The checking of the state of a part of a system** (such as this check in rootprog using access),
2. And the actual **use** of the results of that check

**In particular, the vulnerability lies here:**



We *exaggerate* the **delay** between **check using access** and **actual usage using fopen** by setting sleep(DELAY) in between, where DELAY is specified as 1 to simulate 1 second delay.

Consider the rootprog being called by a user to modify a user text file as such:



The access() check of course grants the normal user caller to modify userfile.txt because indeed it belongs to the normal user.

However, during this ***delay***between checking (with access) and usage (with fopen), what can happen is:

1. A malicious attacker can ***replace***the actual file userfile.txt into a **symbolic link** pointing to the protected file, e.g: rootfile.txt
2. Since fopen only checks *effective user ID,* and rootprog has its SUID bit set (runs effectively as root despite being called by only normal user), the “supposedly secure” rootprog can end up allowing normal user to **modify** the *supposedly* protected file rootfile.txt

The malicious attacker has to attack and can only successfully launch the attack (modifying rootfile.txt) during that time window between time-of-check and time-of-use, hence the term “race condition vulnerability attack” or “a bug caused by race condition” -- as the attacker has to **race** with the rootprog program to *quickly change the* ***userfile.txt*** *into a* ***symbolic link*** *pointing to* ***rootfile.txt* ONLY on this very specific time window** of AFTER the access() check and BEFORE the fopen().

***[2pt]*** *What is a symbolic link? What is the difference between a symbolic link and the actual file?*

**[Q4] Your answer: Also known as shortcut, it is simply a file whose content is a text string that is automatically interpreted and followed by the operating system as a path to another file or directory. A symbolic link is a reference to the actual file so the symbolic link will not increase reference count unlike the actual file.**

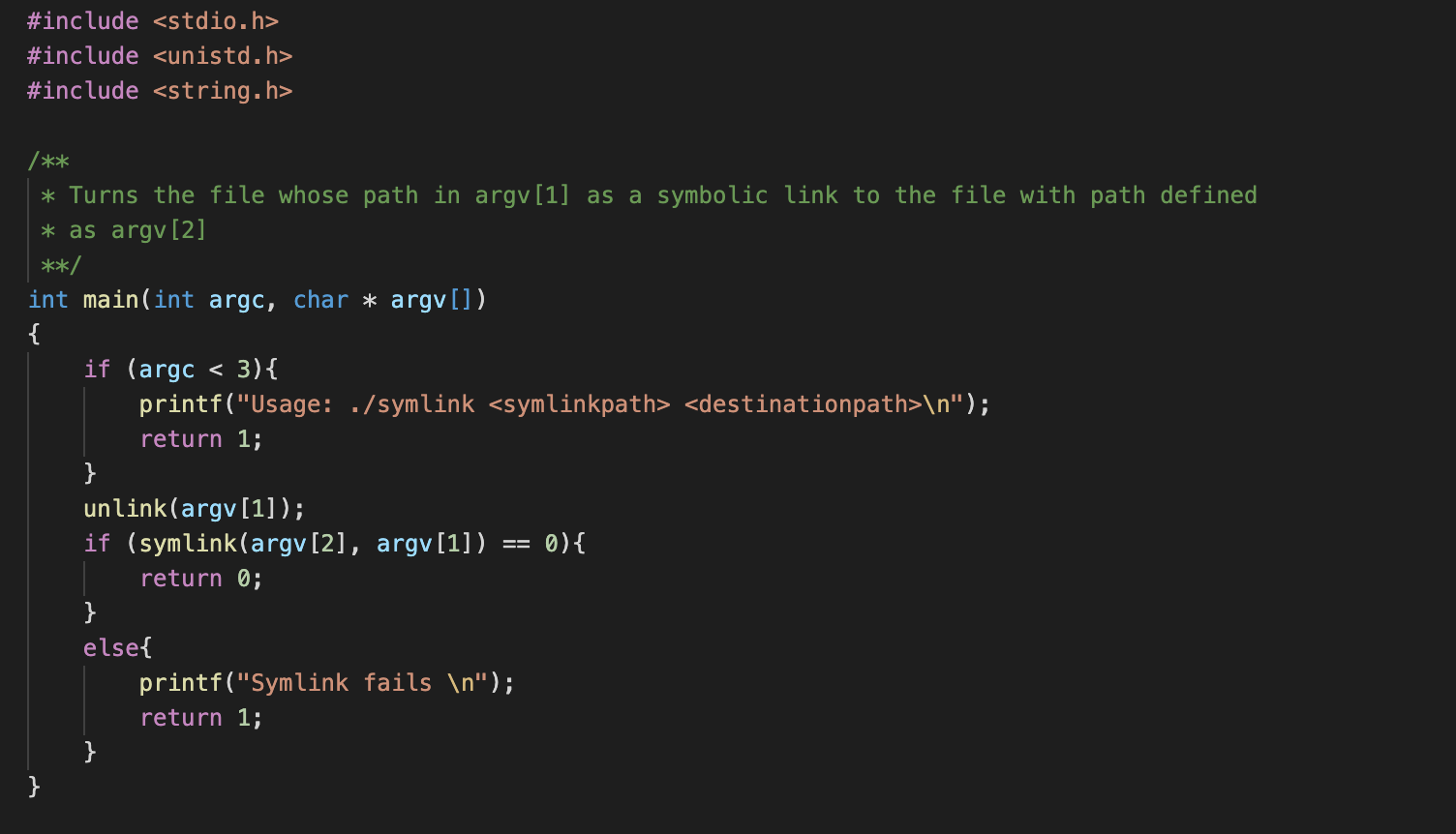
***[2pt]*** *Can you (a normal user) delete a file like rootfile.txt belonging to the root? Why yes or why not? Note: these files are located inside the Root directory that belongs to the root.*

**[Q5] Your answer: No. As the file permission format for *rootfile.txt* will be “-rw-r--r--” :   
- where the first dash means that it is a file,   
- then the next three characters “rw-” means that the file can be read and write by the root (owner)  
- then the second group of three characters “r--” means the file can be read by the users in the same group,   
- and the third group of three characters “r--” means the file can be read by the rest of the users.   
Since only the root can write the file, only the root can delete the file.**

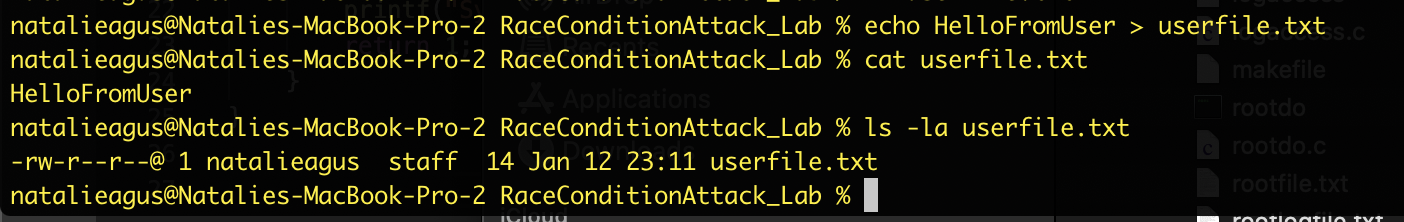
## 

# The Symbolic Link Program

Open symlink.c and read its content:



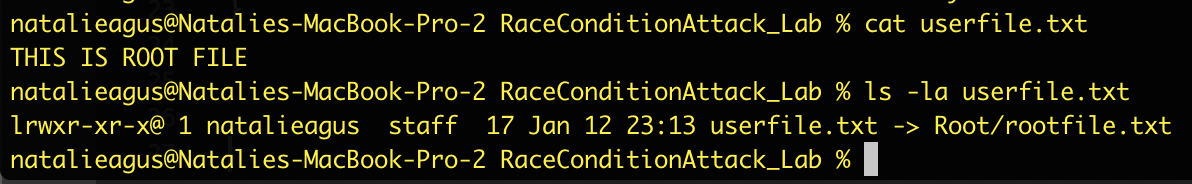
This program changes the target filename as specified as the first argument to the program as a symbolic link to the file whose path defined as the second argument to the program.

Consider the normal user text file userfile.txt that we created before. This text file belongs to a normal user with content “HelloFromUser”. 

Then we can call the program with these arguments (assuming we are currently inside the RaceConditionAttack\_Lab folder). Please change the commands accordingly if you are not calling symlink in this path.



And now notice how the content of userfile.txt is identical to rootfile.txt, and that userfile.txt is now a symbolic link, pointing to rootfile.txt inside the Root folder.



***[2pt]*** *What does lrwxr-xr-x mean?*

**[Q6] Your answer: The “l” stands for symbolic link, the root (owner) can read, write and execute, the users in the same group can read and execute, and the rest of the users can read and execute.**

Depending on your system, you might notice a difference in its icon as well (now with the little arrow to signify symbolic link). If you double click to open it, you will be redirected to open rootfile.txt instead.



## 

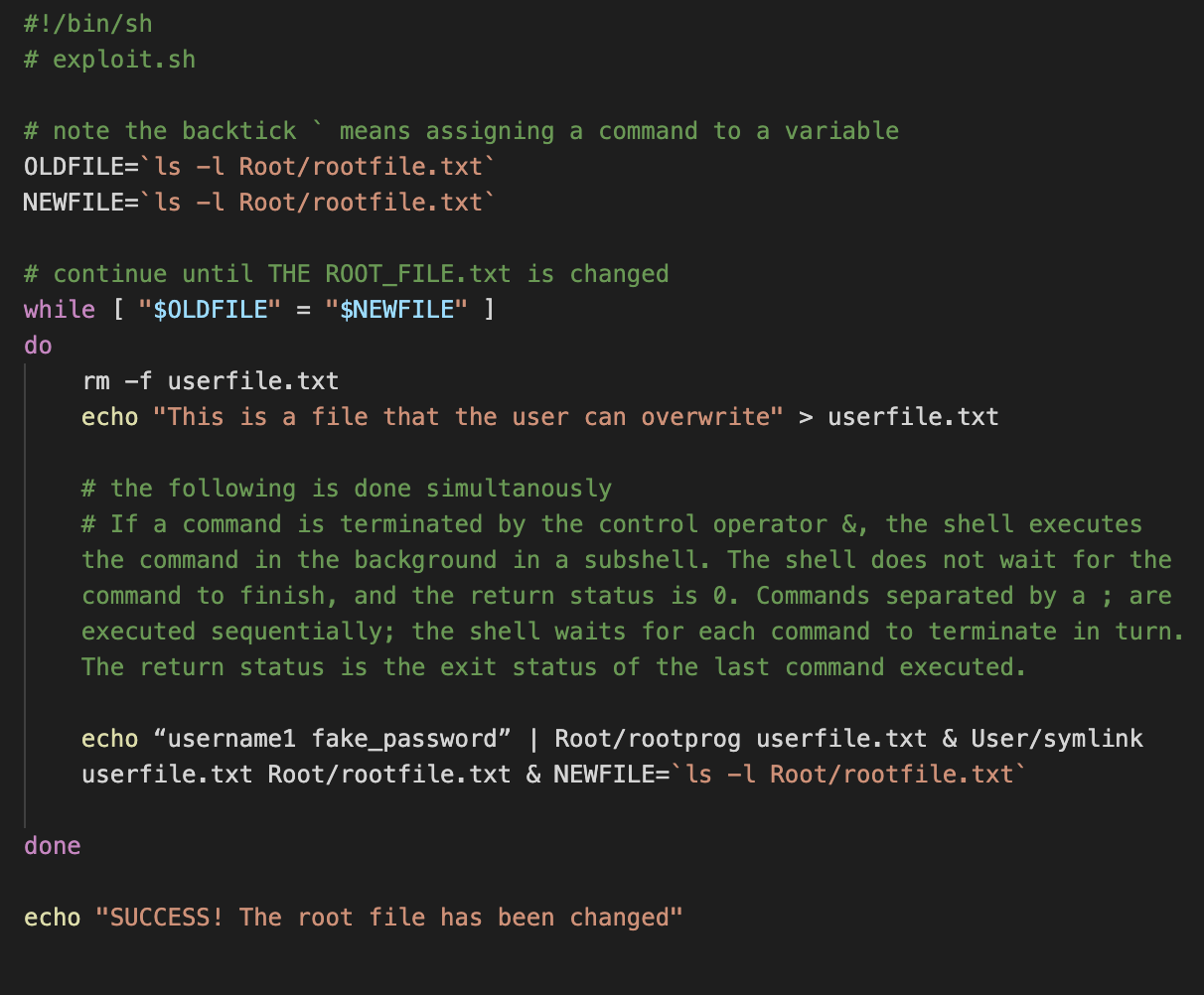
# The Attack

As a malicious attacker, you know that you need to launch your *attack*, which is to **replace** the userfile.txt **as a symbolic link** to the rootfile.txt, in the exact time delay AFTER the access() check but BEFORE the fopen() usage. Let’s define the *attack* **goal** itself as a really simple goal for this lab: **successfully write a new line (attack message) into a supposedly protected rootfile.txt as a normal user.**

Here’s what the attacker should do in essence:

1. Create a userfile: echo “This is a userfile” > userfile.txt
2. Simultaneously launch:
   * The execution of rootprog with userfile.txt as its argument and the *attack message*
   * The execution of symlink with userfile.txt and rootfile.txt as its arguments
3. Check if rootfile.txt has been injected by the attack message. If yes, attack is successful and we can stop the attack attempt. Else, remove userfile.txt and redo step (1).

It is impossible to do all these steps manually and rapidly (because we need to *race* with rootprog), and therefore we can write a bash script to do it. Open exploit.sh and in there you can find the following code. **Read it carefully to understand how it works.**



***[2pt]*** *How does the script exploit.sh check if the attack is successful and stop the loop?*

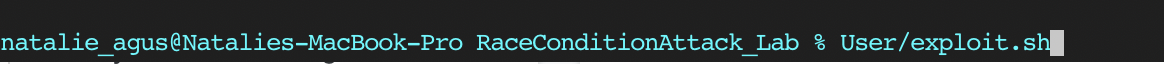
**[Q7] Your answer: The initial content of the file is “$OLDFILE” while the read content of the file is “$NEWFILE”. The script constantly compares the two contents. When they are the same, the file fails. However, when the contents are different, the attack is successful and eventually exits the while loop.**

***[1pt]*** *What “attack message” is injected into rootfile.txt when the attack is successful?*

**[Q8] Your answer: PID {pid} is writing -- "username1: fake\_password"**

In practice, the “attack message” can be modifying system password so that outsider can remotely log into your computer later on, or modifying some root files to mess up your computer, etc.

You can launch the script as follows:



***[2pt]*** *Comment on your output. Is your attack successful? If yes, how long does it take for the attack to be successful. If not, why not?*

**[Q9] Your answer: Yes. There were many attempts of execution with both permission denied and access granted, eventually the attack is successful as it ends with “SUCCESS! The root file has been changed”. It took about 1 second.**

***[1pt]*** *If we only sleep for 1 ms instead of 1s, what impact does it have to the attack?*

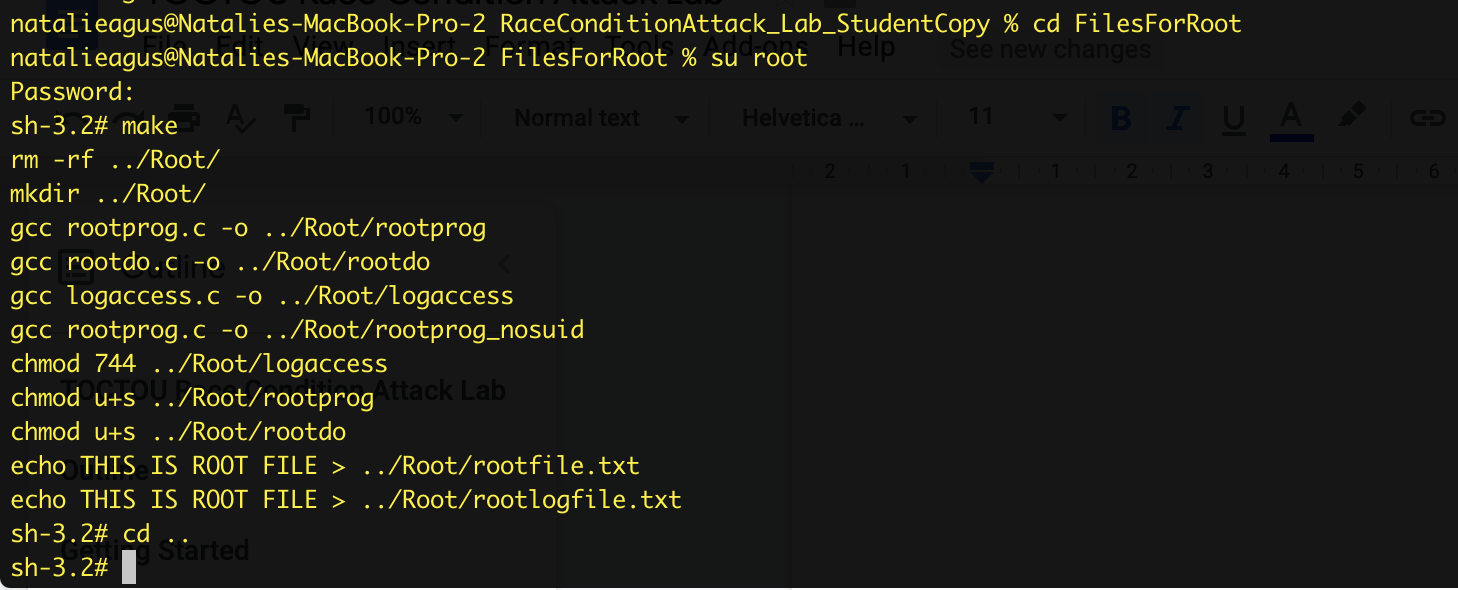
**[Q10] Your answer: No impact.**

# The Fix

One of the ways to fix this bug is to **manually set the effective UID** of the process as the **actual UID of the process** just *after* access is granted and *before* fopen() is called. You can do this using the following system call:

seteuid(getuid());

Modify rootprog.c and recompile it while logged in as root:



***[2pt]*** *Relaunch the attack script using User/exploit.sh again and comment on your output. Why do you think the output with this modification is different from the output by the original rootprog.c code?*

**[Q11] Your answer: The attack cannot be executed successfully. It is stuck in the while loop, alternating between with permission denied and access granted, and the file cannot be opened. This is because it is unable to get root access into the protected file using the TOCTOU vulnerability. As the “seteuid()” set the permissions level to “getuid()”, so real user id value is required.**

Of course another way is to disable the SUID bit of the rootprog altogether, however in practice sometimes this might not be ideal since there might be other parts of the program that requires execution with **elevated privilege**, temporarily. Open exploit.sh and replace rootprog with rootprog\_nosuid, and run the script again.

***[2pt]*** *After editing the shell script, relaunch the attack script using User/exploit.sh again and comment on your output. Why do you think the output with this modification is different from the output by the original rootprog.c code?*

**[Q12] Your answer: The attack cannot be executed successfully. The new executable does not have a SUID bit so the file cannot be opened. The SUID bit is required to allow normal users to gain elevated privilege when executing the program.**

# Summary

Ensure that you have answered all the questions in edimension. You may save your answer and resume later. No other separate code submission is needed.

By the end of this lab, we hope that you have learned:

1. What SUID bit does, and how can it be utilised to gain elevated privileges to access protected files
2. The differences between root and normal user
3. The meaning of file permission. Although we do not go through explicitly on how it is set, you can read about it here: <https://kb.iu.edu/d/abdb> and experiment how to do it using the chmod command.
4. How race condition happens and how it can be used as an attack
5. How to fix the TOCTOU bug