THOMAS
CHOPITEA

ADRIEN
GSELL

RACE ATTACKS!

NICE CODE, WE'LL TAKE IT!

First Part

Documentation of the program

The program we created for this lab consists of 5 files :

IPassword.java This file is the interface of the *Password* class. It contains the headers of the four functions that are used in this class.

Password.java This file is the vulnerable functional class of this program. It contains four functions. *check* sees if a user-name/password pair is correct, *updatePwd* allows to modify the password associated to a user-name, *deleteUser* allows to remove a user-name/password pair from the database, and *maxReqTime* gives the maximum time that was used to perform these actions.

The function *updatePwd* contained in this file is the one we will use to perform a race attack. We will modify a password without knowing the old one, therefore gaining access to the hijacked account.

PasswordCorrected.java This file is the corrected version of the file Password.java. The *updatePwd* function has been modified to make it resistant against the race attack used in the first part of this lab.

Principal.java This file contains the description of the thread which will be used to represent the attackers Kalle and Malin. This class just calls the *updatePwd* function and prints some information about the user and the information he sent.

PasswordApp.java This file contains the main function of our program. The database (consisting of two *String* arrays) is created here. Then, the two threads corresponding to the attackers Kalle and Malin are generated, initiated and launched. A few print commands are also used to make the output of the program human-readable.

The default version of the program used by this file is the corrected one. To access the one vulnerable to the race attack, the files **PasswordApp.java** and **Principal.java** have to be slightly modified. Some indications are given in the form of comments inside these two files.

Vulnerabilities other than race conditions

Since Java manages memory and array bound-checking automatically, the scenarios in which buffer overflow can occur are rare. Furthermore, this program does not require any user interaction, which means that most variable's values are completely deterministic.

The fact that the program does not take into account user interaction means that information flow is completely predictable: the output will globally be the same for each run of the same build (the only difference here being the execution time, which could give an indication on how much the processor is being used… but this is not really our concern here).

Assumptions about Bob

We only assume that Bob actually has valid credentials in our database (a valid username / password combination). This is what we want to change in order to hijack his account! Bob doesn’t even need to try a login attempt for our attack to work.

We also have no reason to assume Bob is a particularly bad guy.

The attack

The attack exploits a pretty basic *race condition* in the way the Password class is implemented. The race condition occurs precisely in this part of the code visible in **Fig. 1**.

When *Principal* tries to update his password, *Password* will first check if the variables *username* and *oldpassword* correspond to an existing pair in the database. The result of this check is stored in a global Boolean variable of the class *Password* (1). The program checks, then, the value of this variable in order to know if the user has the right to modify the password. If this verification is successful, the program looks for the right user-name and modify the password associated to it (3).

The sleep function (2) represents the point on which the attack will take place. To succeed in this attack, *Kalle* and *Malin* will have to connect at the same time.

Kalle is the first one. He will try to connect as Bob. He doesn't know the password so he will use a random word and the authentication should fail; this information is stored in the global variable *auth*. At this moment, *Kalle* is at the sleep point (2).

Malin tries now to connect as himself. As he is using a valid user-name / password pair, his trial of authentication should succeed. This action modifies the value of the *auth* global variable. *Malin* is now also sleeping.

After one second, *Kalle* wakes up. In order to know if *Kalle* can modify the password associated to *Bob*, the program will check the value of the *auth* variable. As long as it has been changed by *Malin*, *Kalle* will be allowed to modify *Bob*’s password.

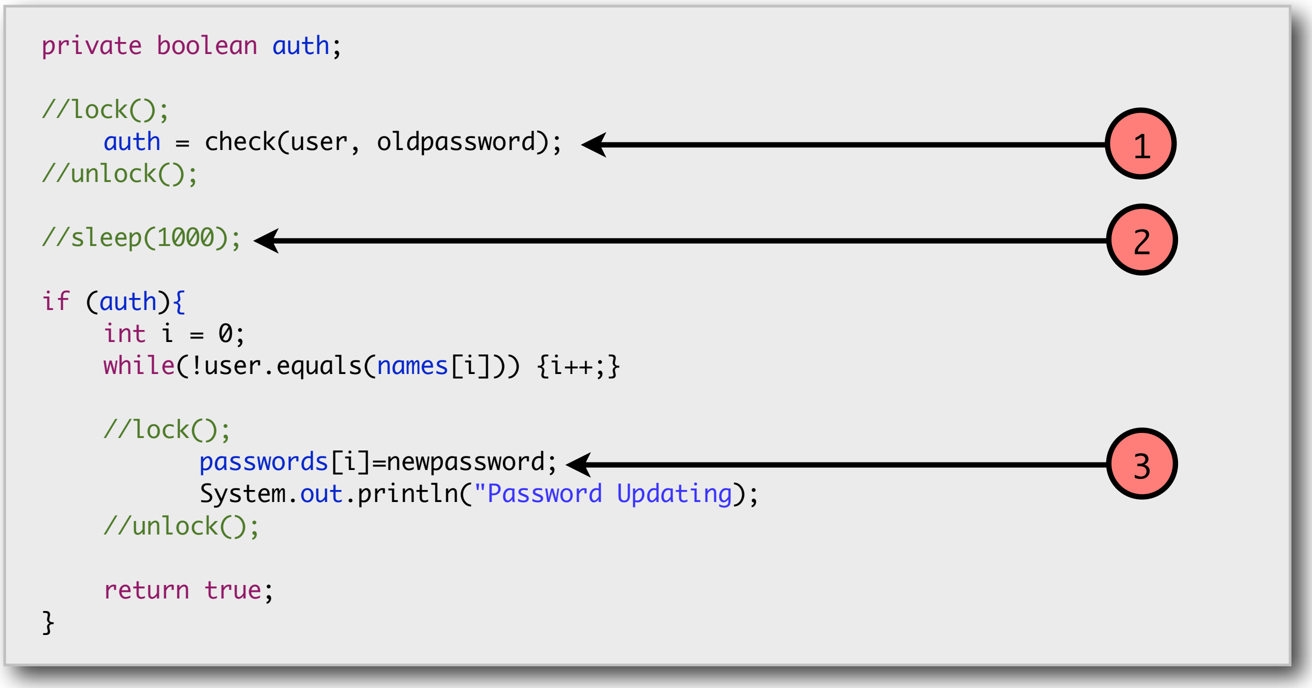
This mechanism is represented in **Fig. 2**.

The protection technique

The problem of the vulnerable program is the possibility of having two threads running the same function at the same time.

To protect our program, we locked the sensitive part of the password update function. By removing the global *auth* variable and by checking the identity of the user directly in the *if branch*, we removed the possibility for a thread to change the result of an authentication while another thread is updating its password.

Fig. 1 Sample code of the unsecured program



```
private boolean auth;

//lock();
    auth = check(user, oldpassword);
//unlock();

//sleep(1000);

if (auth){
    int i = 0;
    while(!user.equals(names[i])) {i++;}

    //lock();
          passwords[i]=newpassword;
          System.out.println("Password Updating);
    //unlock();

    return true;
}
```

Fig. 2 Program execution during the race attack

```
            Thread 1 : Kalle                                  Thread 2 : Malin

auth = check(BOB, WHATEVER);
// auth false
                                                  auth = check(MALIN, good_password);
                                                  //auth true

if (auth){

            passwords[i]=newpassword;

     return true;
}
//Kalle could change Bob's password
because auth was true when it was
checked by the programm.
                                                  if (auth){

                                                              passwords[i]=newpassword;

                                                       return true;
                                                  }
```

Second Part

In this part, we examine how to efficiently and securely prevent a "password input field" from displaying cleartext characters, and replacing them with asterisks (*) instead.

Simple solution

The simple solution relies on a thread constantly deleting the last character from *Standard.out* and replacing it with an asterisk.

Does it work ?

The solution itself does work (i.e. the typed characters are not seen on the screen and are replaced by asterisks). This being said, an additional asterisk is inserted at the beginning of the input line, which could confuse the user. This would also increase security since the numbers of asterisks don't match the number of characters in the password, but would be completely useless if the "snooper" also knows about this. We think it is more of a bug in the implementation than an actual feature.

Enumerate vulnerabilities

Although quite simple, this code still has some flaws:

· There is a TOCCTOU race condition on the private variable *stop*. If another instance of the thread is created from the same object (e.g. when another user requests the same service on the network), then since the "stop" variable is private, the threads will stop replacing text with asterisks as soon as the first user has finished using the first instance of the tread.

In a more general way, if the variable "stop" (which is copied on the stack for optimization, as we will see later) is modified by any other program so as to be evaluated as false, then the thread will stop and the characters will appear in clear text again.

There is a similar race condition on System.out. What if another thread just "adds" random characters to System.out just as often as Eraser replaces them with asterisks? Then all or part of the password input by the user would become visible

· As it is stated in the documentation, this implementation makes a really heavy use of threads containing a busy-wait loop (the last character is deleted and replaced by an asterisk even if it has already been masked). When under heavy load, the system may experience some delay when carrying out these instructions, which could result in a full or partial disclosure of input data, until all characters are masked again.

This also applies when a client is suffering from network congestion in the case the "*010**" string is being constantly received through the network (although it would become evident that such an implementation of password masking is ridiculously inefficient).

If yes : Protection techniques
Our opinion about this code is that the implementation needs to be rethought from scratch. Having a busy-wait loop constantly replacing the characters in the standard output is not a very efficient solution. It is resource consuming, and the Boolean variable can lead to a race condition if the variable's value is changed before the thread starts masking characters.

What could be changed here is to have the thread "react" on each keystroke of the user, much like the "getch()" function in C. In this way, the ASCII code sent through the Standard input is stored in memory (and subsequently dealt with, for example appending it to a string which will end up being the password once the character corresponding to "\n" is received) and not "reflected back" to Standard output. This solves both the race condition and the resource consumption issues we had found before.

If we want to keep the current implementation, then it would become necessary to lock the stop variable (so as to prevent external modifications) and the System.out variable (so as to prevent another thread of interfering with the Eraser thread). This would, of course, make it impossible tu run two "login interface" concurrently (since the other thread would need to wait until stop and System.out are unlocked).

Making the Code Secure and Reliable
In this part, two enhancements are detailed in the documentation:

· Declare the variable stop as volatile. This indicates that stop might be modified by other threads, thus preventing the compiler to do any code-optimization based around the use of this variable.

· Increase the priority of the calling thread to the maximum during the duration of the call so as to avoid slow character masking due to intense resource use.

· Storing the password into a char array rather than a String object. This would increase security since Strings are immutable (they cannot be modified after use: this means their values will still be around somewhere in memory until Java's garbage collector unallocates them). Ironically, this leads to the appearance of another race condition that we will discuss later on.

Does it work ?
The program works as it is intended to – the extra asterisk is still there, but characters are being masked as we type them.

Enumerate vulnerabilities
The information disclosure vulnerability caused by the extensive use of system resources has been dealt with by using priorities (which is effective despite still being quite inefficient).

That being said, the thread still depends on the variable "stop" to be executed. Now that it has been declared as volatile, the references to stop will always point to the memory and not to the stack, which protects it from eventual unintended modifications from outside programs. In any case, it still alterable by programs sharing the same memory space, or by other objects having access to the same thread instance.

As mentioned before, the fact that we are now using character arrays instead of String objects introduces a new race condition that was not present before.

String objects are immutable: this means that once they are created, then cannot be modified. To "actually" modify a String object, Java creates a new string object and updates the old reference. The other data is still left in memory. This of course makes String objects a risky place to store passwords or other sensitive information: a memory dump could easily reveal them, since the data is unreferenced to, but still there.

The proposed solution is to use a character array instead of String objects, so as to be able to clear it when it is no longer used (with *Arrays.fill(buf," ");*). Clearly, the memory section containing the password will be wiped out and no trace of it will remain in memory.

The "benefits" of such solution is precisely what creates the race condition: now, the contents of the variable can be changed. In the case of Strings, when two different threads have two references pointing towards the same String, if one of them tries to modify it then a new string will be created and the reference of that thread changed (because it is immutable). Now that we are working with a char array, the value of the array itself will be changed, instead of the reference. This means that the array content can change without the first thread expecting it, which can lead to TOCTTOU race conditions. This could probably be exploited in a similar way than the vulnerability in Part One of this Lab.

Protection techniques

To avoid this new race condition, we could implement a lock which would prevent anybody else from accessing the array containing the password before it is checked against the database and cleared from its contents. This would indeed impair any two users to log in simultaneously. But since the login process is not very long (the "critical section" of our locks being rather short), this will hardly be noticed by the end-user.

Conclusions that can be drawn from the use of multithreading

Multithreading is of course an essential part of any modern application. Nevertheless, the security principle by which more functionality (and therefore more complexity) causes more bugs (and therefore, more risk) still stands, stronger than ever.

Multithreading has really many advantages, but implementation has to be done very carefully – the smallest negligence could lead to an entire system compromise. Shared variables have to be accounted for and locks need to be put in the correct "security critical" sections.

"Data races. Ha ha. Funny looking little bugs, aren't they?"