CWE-367:
Time-of-Check Time-of-Use Race Condition

CS463 - Network Security

Submitted to: Dr. Mahendra Pratap Singh

Abhishek Kumar

181CO201
abhishek.181co201@nitk.edu.in

Background

A **Race Condition** occurs the successful execution of a program depends on the order or sequence in which uncoordinated processes and threads are executed. The classic example is two threads that simultaneously increment the value of a global variable as shown below:

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Such an execution incorrectly sets the value of the global variable as 1, instead of the expected value of 2.

Race conditions are hard to identify because they are non-deterministic and cannot be reliably reproduced. However, race conditions can be avoided by careful software design and the use of appropriate concurrency safeguards in form of locks, mutexes, and transactions.

Multitasking

Most modern OSs share CPU time between the processes and threads in order to improve CPU utilization. For example - the OS might execute process A, then switch to process B, then switch to process C, and then complete process A. Therefore, even simple,

non-concurrent applications are vulnerable to race conditions and therefore require careful design if they utilize shared resources like files, sockets and devices.

Time-of-Check Time-of-Use Race Condition

Time-of-Check Time-of-Use (also referred to as TOCTOU or TOCTTOU) is a race condition in which the state of a resource (typically a file) is changed after the check, invalidating the check itself.

While TOCTOU vulnerabilities are normally associated with filesystem-based attacks, any application that uses shared resources without appropriate synchronization mechanisms is vulnerable. This includes web-based applications which share network resources or store their state in an external database.

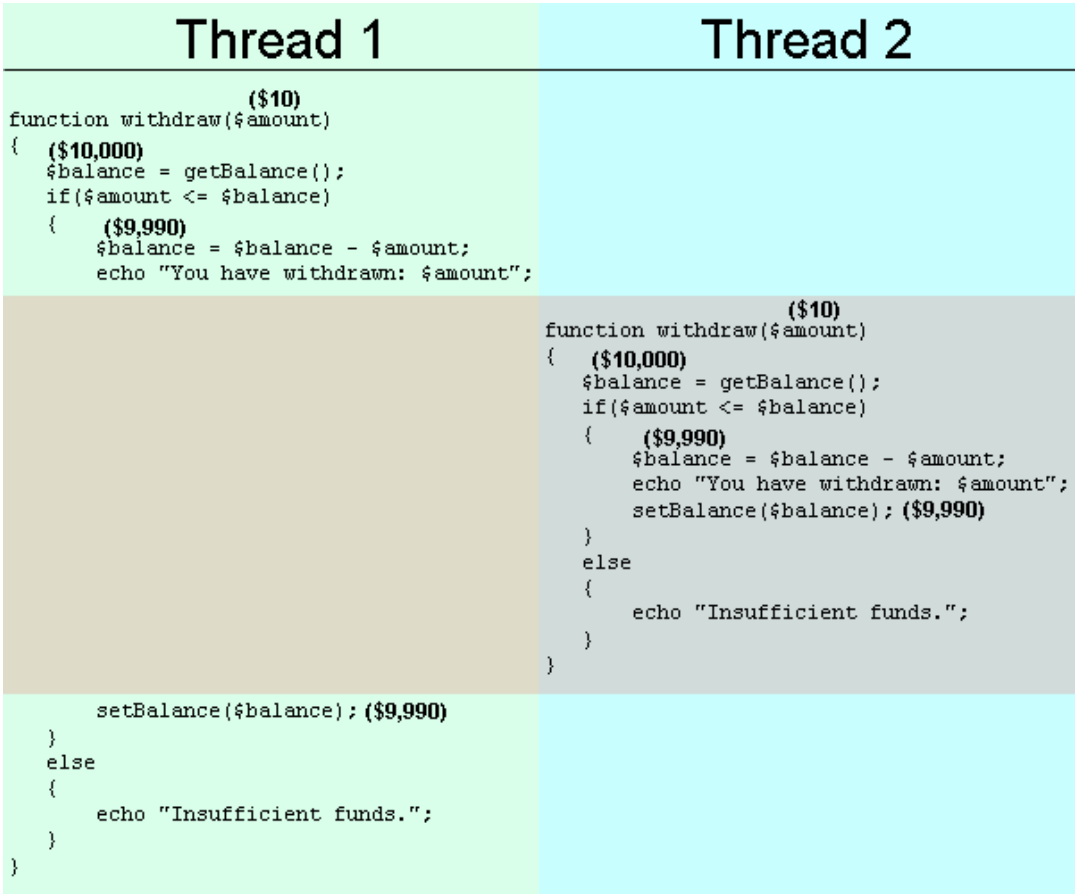
Web-based Example

Consider a PHP-based application to withdraw money from an online bank account. The application stores the current balance of a user in a SQL database and can read/write the balance using the functions *getBalance()* and *setBalance()*.

The logic for withdrawing money is straightforward, and is:

1. Read the current balance of the user using *getBalance()*.
2. Check whether the current balance is greater than withdrawn amount.
3. if it is, update the new balance using *setBalance()*.
4. Otherwise, display an error to the user.

Assume an attacker has $10,000 in their account and makes multiple requests simultaneously to withdraw $10. It is possible that the execution of the different requests will be interleaved as shown in the following figure, such that the amount of money withdrawn from the account is less than the amount actually received.



Significance

TOCTOU vulnerabilities are a very significant problem. Between 2000 and 2004, CERT issued 20 advisories on TOCTOU vulnerabilities. They cover a wide range of applications from system management tools (/bin/sh, shar) to user level applications (openSSL, Openoffice). More recently, TOCTOU vulnerabilities were discovered in Log4J 2.15 and Docker.

Implementation

I have implemented a filesystem-based TOCTOU vulnerability as follows: While running the program as root, we check whether we can edit a file using *access()* function. After passing the check, we can replace the file with a symbolic link to another file to which the current user does not have access and overwrite its contents. The source code is available on abhishekkumar2718/Time-of-Check-Time-of-Use.

Scenario

There are two Linux users - *abhishek* and *sachin* and two files *temporary-file*, owned by *abhishek,* and *privileged-file*, owned by *sachin*. *abhishek* does not have read, write or execute permissions to the *privileged-file.*

The expected outcome of the program is that it writes "*Hello, World"* to *temporary-file* when run as root through *abhishek*. The program is careful to check whether *abhishek* in fact has write permission to *temporary-file*.

Insecure Implementation

The insecure implementation works as follows:

1. Check whether *abhishek* has write permission to *temporary-file* through *access()*.
2. If yes, open the file using *fopen().* Write to the file using *fprintf().*
3. If not, display an error message and terminate.

In the gap between steps 1 and 2, the attacker deletes *temporary-file* and creates a symbolic link from *temporary-file* to *privileged-file*. Then in the second step, ***fopen()* resolves the symbolic link and opens *privileged-file***. This is incorrect because the *access()* called for *temporary-file.* Thus, the attacker is able to overwrite the contents of any protected file that current user might not have access to.

Secure Implementation

The secure implementation works as follows:

1. l*stat()* the file before opening.
2. *open()* the file, returning a file descriptor.
3. *fstat()* the file descriptor returned in second step.
4. Compare the file type and mode, inode number and ID of device containing file between stat structures returned in first and third steps.
5. If the stat structures are same, return FILE pointer by opening the file descriptor.

This is safer because:

- Uses *lsat()* instead of *stat()* or *access()* and does not resolve symbolic links.
- Compares the stat structures before and after opening the file and verifies that they are same.
- Relies on file descriptor and inode numbers which are immutable, instead of file names which can point to different files.

Possible Solutions

Use Immutable Handles

A key observation in TOCTOU vulnerabilities is they usually involve a file system that takes a file name for input instead of a file handle or a file descriptor. By using file descriptors and pointers, we can ensure that the file does not change. Similarly, use immutable handles for other contexts instead of mutable handles.

Delegate Checks To Resource

A TOCTOU vulnerability relies on the result of the check invalidated (due to the attacker's activities) and thus leads to incorrect assumptions in the rest of the application. If possible, avoid explicitly checking conditions and let the resource (or operating system) itself handle checks. In practice, this means avoiding function calls like *access()* and letting the operating system reject *fwrite()* operation.

Use Transactions, Synchronization Primitives

During the attack, the resource is changed or altered *after* the check but before the *use* of the resource. One way of avoiding such problems is to use locks, mutexes, and semaphores before checking the resource and releasing them after use. This ensures that no other process can modify the resource in the gap between *check* and *use*.

If the resource (like a relational database that is ACID-compliant) provides transactions, prefer transactions over locks and other primitives.

Identify Vulnerable System Calls Through Static Analysis

Jinpeng Wei and Calton Pu in their paper, “TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study”, identified certain pair of system calls that can be possibly exploited for a TOCTOU attack. Based on their model, they have implemented static analysis tools which can be used to verify and warn about possible attacks.

References

1. Race Condition | Wikipedia
2. Practical Race Condition Vulnerabilities in Web Applications
3. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study
4. Docker Bug Allows Root Access to Host File System | Decipher
5. Log4J 2.15 TOCTOU Vulnerability Illustrated by GoSecure Researchers - GoSecure