File-based race conditions in UNIX: TOCTOU

Razvan Raducu[1], Ricardo J. Rodríguez[1], Pedro Álvarez[1]

[1] Distributed Computing (DisCo)
Instituto de Investigación en Ingeniería de Aragón (I3A)
Universidad de Zaragoza, Mariano Esquillor s/n, 50018, Zaragoza, Spain.
e-mail: *{razvan, rjrodriguez, alvaper}@unizar.es*

Abstract

This work presents an introduction to the Time Of Check to Time Of Use (TOCTOU) vulnerability as well as the development of a user-space library that hooks vulnerable system calls and modifies their behavior.

Introduction

Race conditions may arise when performing operations that involve the file system because there is no guarantee of atomicity for I/O operations or consistency of the file system. These vulnerabilities are a true menace to integrity, availability and confidentiality of information systems since they may allow an attacker to escalate privileges. These vulnerabilities, known as Time Of Check to Time Of Use (TOCTOU), are a true challenge given their non-deterministic nature. Detecting and preventing them is an open research field. Their reproduction is equally difficult because their exploitability depends upon external factors such as system load or environment variables. This work presents a study of the vulnerability as well as our attempt at thwarting TOCTOU by implement a user-space library that replaces current file system calls with safer versions.

TOCTOU Overview

TOCTOU may arise when a given program performs several consecutive system calls that reference file system objects via their name, also known as pathname. UNIX-like operating systems are especially prone to such race conditions given their file system nature. These operations are neither atomic or isolated in the system. That is, an attacker could modify the underlying file system between any given consecutive system calls since the file system is shared among all running processes [1]. Since TOCTOU requires two or more consecutive non-atomic file-related operations, it is safe to say TOCTOU happens in two steps [2]:

1. The vulnerable program checks some condition about a given file, the *Time Of Check.*
2. The vulnerable program modifies the file assuming the previous condition still holds, the *Time Of Use.*

In order to be profitable by an attacker, the vulnerable program must be running with higher privileges than the attacker. That is, TOCTOU could be present but its exploitation could provide no benefit. If the vulnerable program is running with higher privileges, e.g. SUID to *root*, and an attacker is able to successfully exploit it, they will be able to potentially escalate that privileges. In UNIX universe, this translates to the attacker being a user with lesser privileges than the running program. It is also strictly necessary for the vulnerability to be exploitable that the attacker must have permissions to modify the referenced file. This implies that there are several directories immune to TOCTOU like */root* or */bin*, amongst many others [3].

TOCTOU is usually present in UNIX-like operating systems due to their referencing and path traversal mechanism. These OS allow users to reference file objects using either pathnames or file descriptors. When using pathnames, the application is susceptible to TOCTOU because the relation pathname-file object is not static and it can be modified by external processes. On the other hand, using file descriptors makes the application immune to TOCTOU since file descriptors refer to a single file object and cannot be modified by external agents[4].

Defending against TOCTOU

We developed a defense against TOCTOU that consists of a user-space library that intercepts vulnerable system calls. It changes the behavior of the original system call so it performs additional checks and verifies the integrity of the referenced file objects between system calls. Depending upon the result of the verification the library either calls the original system call (no race condition was detected)

or aborts the program's execution (race condition detected) thus thwarting the exploitation attempt.

The library is loaded dynamically in the memory space of every single process executed in the system whose real user ID (RUID) or real group user ID (RGID) differs from the process' effective user ID (EUID) or effective group ID (EGID). It is loaded by the dynamic linker before any other library so when a program performs a *access()* call, for example, it executes our version of the function instead. This behavior is achieved thanks to */etc/ld.so.preload* [5], [6].

Wei and Pu previously identified the set of vulnerable system calls that can result in TOCTOU [7]. Our library intercepts a total of 46 API functions that refer to file system objects via their name.

Our library maintains and manages an internal per-process list of referenced object and several metadata information like *inode*, *path*, *device id* or *file mode*. When an object is referenced for the first time, the information is retrieved and stored. Consecutive references to the same object are then checked against the cached information and if they do not match, and the process did not legitimately change the referenced object, it means an external process or agent modified the file. In this situation, the program is aborted and the logs are updated.

Limitations

When testing our library, we found several limitations that we consider as future improvements.

The most critical limitation we found is that our library could be a victim of TOCTOU as well. The reason behind this is that we are adding a *mediation* layer between the application and the current API, but the underneath system calls remain unchanged. TOCTOU happens within the API and it would require changes in the kernel to completely remove the vulnerability.

Another limitation is related to cooperating processes. Since our library maintains the metadata information on a per-process basis, whenever a process performs a *fork()* or *exec()* and the subsequent processes legitimately change the file object, our library would result in a false positive thus aborting legitimate programs.

Additionally, our library hooks standard C implementation of such functions (GLIBC) but there could be others. Furthermore, our library is limited to dynamically-linked programs. Statically-linked binaries are not protected by our library since their execution does not involve the dynamic linker.

Conclusions

We presented an introduction to TOCTOU vulnerability and how it can be exploited by any given attacker. Additionally, we presented our own approach when thwarting exploitation attempts. Our user-space library hooks vulnerable system calls and performs additional checks in order to detect external manipulations of the referenced file objects.

TOCTOU vulnerability is still a menace to UNIX-like operating systems. Its detection is difficult given its non-determinism and how subtle it is to detect and be aware-of. Many defenses have been previously proposed but they either are not enough or have been defeated. Given the current state, defending against TOCTOU falls on programmers and, from a security perspective, that is not viable. Security-aware programming requires a lot of experience and is both error-prone and a very difficult task.

REFERENCES

[1] E. Tsyrklevich and B. Yee, "Dynamic detection and prevention of race conditions in file accesses," *SSYM'03 Proc. 12th Conf. USENIX Secur. Symp.*, p. 17, 2003.

[2] M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses," *Comput. Syst.*, vol. 9, no. 2, pp. 131–152, 1996.

[3] J. Wei and C. Pu, "TOCTTOU Vulnerabilities in UNIX-Style File Systems : An Anatomical Study," *Security*, no. December, pp. 1–13, 2005.

[4] M. J. Bach and others, *The design of the UNIX operating system*, vol. 5. Prentice-Hall Englewood Cliffs, NJ, 1986.

[5] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann, 2000.

[6] H. Casanova, "Linking and Loading," 2010.

[7] J. Wei and C. Pu, "Modeling and preventing TOCTTOU vulnerabilities in Unix-style file systems," *Comput. Secur.*, vol. 29, no. 8, pp. 815–830, Nov. 2010.