Global Information Assurance Certification Paper

A Tour of TOCTTOUs

J. Craig Lowery

SANS GSEC practical v.1.4b (August 2002)

"Time of check to time of use" (TOCTTOU) vulnerabilities exist due to race conditions arising from an invalid assumption: That nothing affecting the validity of a security assertion changes between the time it is checked and the time an operation that depends on that assertion is performed. In fact, it is quite possible that the security of the environment changes with respect to the assertion during this interval. If these changes are cleverly timed and orchestrated, the operation may result in a security breach. This paper characterizes this particular category of security vulnerabilities, describes various types of TOCTTOUs and particular situations in which they have arisen historically, and presents a short set of guidelines for reducing or eliminating these flaws.

Problem Context

It is an accepted fact that poor software quality accounts for a vast majority of security vulnerabilities in modern computer systems[1]. Software quality can suffer for a number of reasons ranging from poor design to poor implementation. Making invalid assumptions is one of the primary mistakes. For example, the architect or programmer may assume that certain input combinations will never be presented because they are unlikely, resulting in a brittle implementation that fails or is compromised when these unlikely inputs occur.

Similarly, one may not fully comprehend the side effects and timing issues associated with integrated components, or one may make other invalid assumptions about their function. The operating system that provides the execution context for a program can be viewed as a "black-box" component. Operating systems purposefully present the face of a single-user virtual machine to client programs in multiprogrammed environments. What logically is a single thread of execution is, in reality, a distributed component in an encompassing execution thread comprising the operating system and the interleaved executions of hosted programs.

Characterizing TOCTTOUs

The illusion of a single-thread of execution may encourage invalid assumptions that do not account for the existence of external agents; some of

these external agents, typically other processes hosted by the same operating environment, may be malicious and capable of exploiting the context switching mechanism to wage an attack on other processes. Consider the typical algorithm for accessing a resource within a programming environment:

1. Obtain a reference to the resource
2. Determine resource characteristics by querying the reference
3. Analyze the query results to determine fitness of the resource
4. If the resource is fit, access it using the reference

Steps 2 and 4 of this algorithm are of particular importance, as they constitute the boundaries of what may be considered a critical code section. Step 2 is commonly known as the "check" step. Step 4 is commonly known as the "use" step. The resource reference is the object being checked and used. If the resolution of the resource reference changes between the check and use steps, then actions performed in step 4 may cause a security breach. This is the essence of the "time of check to time of use" vulnerability, commonly denoted by the acronym TOCTTOU.

References to TOCTTOU vulnerabilities date as far back as at least 1973. One particular paper[2] describes security measures taken in the IBM VM/370 environment to prevent against vulnerabilities due to "discrepancies between parameters at time-of-check and time-of-use". The acronym TOCTTOU is generally attributed to a research paper published in 1974[3]. Since then, flaws characterized by race conditions created by potential changes in a resource reference's resolution have been systematically labeled TOCTTOU.

A form of serialization flaw[4], TOCTTOUs occur due to the programmer's invalid assumption that the resolution of the resource reference will not change between the check step and the use step. A poor solution to this problem is to reduce the window of opportunity to an attacker by reducing the size and running time of the critical section; although this may make the vulnerability more difficult to exploit, it does not resolve it.

Three methods for mitigating TOCTTOU vulnerabilities are presented in the *Handbook of Information Security Management:*[5]

- **Increase the number of checks:** More frequent checks create multiple smaller windows of timing vulnerability, making it more difficult for the attacker to achieve the necessary synchrony to successfully penetrate the system. Although helpful, this method only reduces the window of exposure and does not eliminate it.

- **Check nearer to use:** By moving the time-of-check closer to the time-of-use, the window of vulnerability is proportionally reduced. Again, this merely reduces the window of exposure rather than eliminating it

completely.

- **Immutable bindings:** Making a reference such that its resolution is immutable (*i.e.*, cannot be changed) means that it will always resolve to the same object. This approach eliminates TOCTTOU vulnerabilities, but may not be possible in some situations.

To re-emphasize: Although reducing the window of vulnerability is helpful, it merely makes it harder for an attacker to penetrate, not impossible. Therefore, of the three solutions presented above in the *Handbook*, only the last is truly acceptable.

The assumption that the resolution of a reference is as static as the reference itself is valid in some circumstances. Therefore, one effective way to avoid TOCTTOU errors is to choose resource references that are guaranteed to resolve consistently at both the check and use steps. Immutable bindings can be achieved in several ways. First, the reference may be so closely bound to the object that it simply cannot be by an external agent or, if it were changed, would break all pre-existing references. An example commonly given is the file descriptor within a process table data structure, as opposed to a file name in a directory. File names in UNIX can be up to four levels of indirection away from the file itself: symbolic link resolves to a hard link resolves to an i-node index resolves to a file descriptor. Each of these indirections provides an opportunity for a TOCTTOU attack.

The second way to achieve immutable bindings is to insure that the reference's binding, normally mutable, cannot be changed between the check and use steps: a "temporary immutability." This can be accomplished by providing an atomic operation which both checks and uses the reference, similar to the atomic "test and set" operation commonly used to implement hardware-based semaphores.

Simulated atomic operations can be implemented by treating the code section from check to use as a classical critical section, guarded at entry and exit by a mutual exclusion primitive, or by using other existing locking mechanisms, such as those provided by database management systems to lock records or tables for update. More sophisticated systems that have been designed against TOCTTOU vulnerabilities may employ a *reference monitor* to track object references and detect when unsafe changes have occurred during an operation.

Prototypical Examples

What follows is a representative list of TOCTTOU vulnerabilities. Some of these examples are historical in nature, having actually occurred in production software. Others are contrived examples.

Filename Redirection

Bishop and Dilger[6] analyze an attack set forth by Tanenbaum[7] in which the resolution of a UNIX filename changes from the time a privileged process creates a file to the time that process attempts to change ownership of it.

The privileged program's intended function is to create a new, empty directory and assign ownership of it to the requesting user. It does this by first exercising *mkdir()*, then *chown()*. If the new directory is created within a directory to which the requesting user has full access, then that user can unlink the new directory entry and replace it with a link of the same name to some other file on the system, such as */etc/passwd*. If this is done before the privileged program exercises *chown()*, then when the *chown()* is executed, it will follow the new link and change the ownership of the targeted file to be the requesting user.

The resource reference in this case is the name of the new directory. The time of check is when the *mkdir()* is exercised: At that time it is certain that the name references the newly created directory. The time of use is when the *chown()* is exercised. However, the meaning of the name – that is, the i-node that it ultimately maps to – changes between the check and use.

Source code can be statically analyzed to help determine if filename redirection TOCTTOUs exist. Examples of available scanners include ITS4[8] and RATS[9]. Another oft-cited way to prevent filename redirection vulnerabilities is to make use of immutable references, such as a file descriptor within the kernel's data structure representing the process. For example, in POSIX environments, use of the fstat() system call is preferred over that of *stat() or lstat()* to obtain file status information[10], since the *fstat()* takes an immutable (by external methods) file descriptor as reference to the file to be checked.

Setuid Scripts

Tsyrklevich[10] explains how the functionality of early versions of the UNIX *exec()* system call can lead to security breaches when combined with setuid scripts.

When *exec()* is presented a file to execute, it first opens the file to examine the first two bytes. If it finds a magic number in this position, it proceeds to load and execute the file as a binary executable. If it finds the character bytes "#!" – indicating that the file is a script – it reads additional characters until encountering a line feed. This string is the pathname and optional parameters of the command interpreter which will execute the script. The interpreter is loaded and pointed to the script file by *exec()*.

Normally, *exec()* does not change the privileges of a process when its context is changed, as in the loading of a new binary. However, if the file to be

executed has the setuid bit set, then a *setuid()* system call is performed prior passing control to the new program, effectively changing the process's user ID to that of the owner of the file being executed. If the file to be loaded is a true binary, then it is certain that the subsequent execution is under the control of that file's contents. By contrast, if the file is a script with the setuid bit set, then the *associated command interpreter* is executed with the privileges of the script's owner. Because there is a period of time between when the script is inspected by *exec()* to when it is opened and read by the interpreter, the possibility exists of a TOCTTOU attack, as follows.

1. The attacker creates a symbolic link to an existing script owned by user "root" (the superuser).
2. The attacker attempts to execute the symbolic link.
3. The system follows the link to root's script, opens it, gets the interpreter pathname, and observes the setuid bit is "on."
4. In the meantime, the attacker redirects the symbolic link to a malicious script that the attacker owns. The status of this malicious script's setuid bit is immaterial.
5. The system executes the interpreter as root, and points it to the symbolic link.
6. The interpreter follows the symbolic link to the malicious script and executes it with root privileges.

In this scenario, the resource is the symbolic link. The time of check is when the *exec()* system call follows the symbolic link and finds a script file owned by root with setuid turned "on." The time of use is when the interpreter follows that same link, which now points to a malicious script. This problem can be resolved by having *exec()* pass an open file descriptor rather than a pathname to the command interpreter.

Relocated Subdirectory

Yet another specialized instance of filename redirection reported by Tsyrklevich[10] involves a false assumption made by implementers of the GNU file utilities package.

The *rm* command can remove a directory structure by recursing through the directory hierarchy in a depth-first fashion. After removing the contents of a child directory, the program returns to the parent directory by using the ".." directory entry of the child directory. If an attacker is able to move the child directory to a new location at the right moment, then *rm* will change to the new parent directory and delete its contents, which was not intended.

In this example the resource reference is the ".." parent link. The time of check is when *rm* enters the child directory; at that time, the child's ".." link is not explicitly checked, but it is known to point to the parent directory in a consistent

file system (which we usually must assume). The time of use, however, is the return to the parent directory, at which time we are no longer guaranteed that ".." references the original parent directory.

The exposure can be mitigated in several ways. One is to fork a child that deletes the subdirectory while the parent waits, its current working directory unaltered; this solution has obvious performance drawbacks. Another is to remember the i-node number of the parent directory and re-check it upon return; although not foolproof, this method drastically reduces the exposure without the overhead of creating child processes 1-to-1 for each deleted subdirectory.

SQL SELECT Before INSERT

The dbi-users mailing list archive contains an entry[11] describing a vulnerability introduced by using an SQL SELECT statement to determine that a proposed key value does not currently exist in an opt-in database of e-mail addresses. If the SELECT returns no rows, then an INSERT follows to populate the table with the new row. One way to exploit this scheme is for an attacker to observe the proposed new key value in transit and request the same key value in a nearly concurrent request. If this second request completes before the first, it results in denial of service to the legitimate requestor, because the second INSERT from the earlier request will fail on a primary key constraint. If the attacker can influence processor workload or process priority, then the attack can be sustained indefinitely across repeated subsequent requests.

The resource reference in this case is the primary key value of the requesting subscriber: an e-mail address. When the SELECT is executed (time of check) it returns an empty result set, indicating the e-mail address is not yet registered. This assertion is assumed to be intact when the subsequent INSERT (time of use) is performed.

One way to resolve this particular vulnerability is to make use of table locking within the database management system. Prior to testing for the existence of a particular record, the table can be locked for writing, guaranteeing mutual exclusion of the critical section. If the SELECT query returns an empty result set, then the new user record can be inserted with a guarantee that another insert request has completed.

Java Class Loader

Dean[12] describes the situation encountered in early, poorly implemented, Java virtual machines in which the Java classloader does not enforce compile time type checking at run time. Most of these problems arise from runtimes that impose insufficient restrictions on classloader behavior, such as Sun's Java Development Kit 1.0.2. Classloaders were only required to validate a byte stream as being properly formatted Java byte code, and were not required to

enforce type compatibility between the loaded class and those classes to which it dynamically interfaced.

From a client perspective, it was thus possible for a malicious applet to provide its own classloader that permitted improper runtime typecasting, resulting in increased privileges. From an author's perspective, it was possible for a program, running in a different execution context than that in which it was tested, to unwittingly interface with rogue code in libraries that found their way onto the target system by other means.

The resource reference in this case is the Java byte code. The time of check is the compilation time, at which point the Java source is compiled, verified, and the byte code is created. The time of use is run time, in which the assumption is made that checks verified at compile time are still in force, when they may not be.

System Call Interposition

LOMAC[13], a loadable mandatory access control module, is a security enhancement for Linux kernels. It gains supervisory control over kernel operations by intercepting system calls, inspecting them, and comparing them against an access control policy. It does this by interposing itself between user processes and the kernel. Essentially, user processes make calls to the LOMAC "wrapper," which in turn makes an access decision (known as *mediation*) prior to forwarding the request to the kernel for processing.

Ironically, LOMAC – a supposed security enhancement – introduced many new TOCTTOU vulnerabilities into the kernel due to the way it handled parameter checking. During the mediation phase, LOMAC copied a parameter, such as a path name, into its own buffer space. This copy was checked by LOMAC for adherence to the security policy. Upon a successful check, however, the parameter was *re*-copied from user space to the kernel. The intervening time between the first and second copy provided the opportunity for attack as the parameter could be overwritten with a value that would not have passed mediation, yet is then passed to the kernel because the original value did pass mediation.

Later versions of LOMAC corrected this flaw by first copying the user parameters to kernel space, performing the mediation, then calling the kernel on the same parameter copies, thus avoiding opportunities for an attacker to modify them.

Replay Attacks

Computer networks introduce even more possibilities for TOCTTOU vulnerabilities[14]. Consider a naive implementation of a Web application: A user accesses the remote Web server by providing credentials in an authentication

dialog, which grants access to a restricted portion of the Web site. Subsequent communication can be copied by a network sniffer, or may remain in the client's cache for an indefinite period. In either case, HTML forms submitted originally by the legitimate user can be resubmitted at a later time (replayed) by an unauthorized user.

The assumption that enables the replay attack is that the HTML form would not be available to the client, and therefore could not be transmitted from that client, unless a legitimate user had not first authenticated with the server, permitting access to the form. In this case, the resource is the HTML form, the time of check is the legitimate user's authentication and subsequent initial transmission of the form, and the time of use is when the form is presented (whether by the legitimate user or an attacker).

Communication in a session context[15] is usually employed to thwart replay attacks. A session is a period of time bounded on the left by a successful user authentication and bounded on the right by a logoff or timeout. Upon session instantiation, a unique session identifier is created, and all messages exchanged during the session carry this identifier. The Web server will only honor incoming communications if the identified session is still active (*i.e.*, has not been discontinued by logoff or timeout). Further strengthening session semantics with message sequence numbers, relying on server-side state preservation, and adding encryption can reduce or eliminate replay attacks in Web applications.

Summary

TOCTTOU vulnerabilities arise from software design errors that assume a checked condition to be invariant when, in reality, it is not. Attackers exploit the failure of the software to account for asynchronous, external agents that can alter the checked condition in such a way as to take advantage of it. Although static code scanners can identify common TOCTTOU errors, they are not a completely sufficient solution to guarding against them. Software designers and implementers should take the following steps to avoid introducing TOCTTOUs into their products:

- Any conditional test that resolves an external reference should be flagged as a possible TOCTTOU introduction.
- References with mutable resolutions, such as filenames, should be replaced with references having immutable resolutions, such as file descriptors.
- Atomic operations, such as those similar to "test-and-set," should be used whenever available.

- If an immutable reference resolution or atomic operation is not possible, then the code spanning from the check step to the use step should be placed in a critical section that guarantees mutually exclusive control of the reference, so that alteration of the reference's resolution cannot occur while the critical section is executing.

Although minimizing the window of time during which the vulnerability can be exploited may reduce the exposure, it is not an acceptable solution as it merely reduces the still non-zero probability that a single attack will succeed, rather than eliminating the possibility entirely.

Identified now for more than a quarter of a century, TOCTTOU vulnerabilities are well understood. They can be subtle, and have often been overlooked even in products purporting to have a security focus. However, designers and programmers who watch for the TOCTTOU warning signs and consistently follow good programming practices should have no difficulty avoiding them.

---

[1] Labbate, Evelyn. "Vulnerability as a Function of Software Quality." SANS Institute Reading Room, March, 2001. URL: http://www.sans.org/rr/code/quality.php (25 Mar 2003)

[2] Attansasio, C. R. "Virtual Machines and Data Security." *ACM Proceedings of the Workshop on Virtual Computer Systems*, 1973.

[3] McPhee, W. S. Operating system integrity in OS/VS2. IBM Syst. J. 3, 3 (1974), 230–252.

[4] Landwehr, Carl E. *et al*. "A Taxonomy of Computer Program Security Flaws." ACM Computing Surveys, Vol. 26, No. 3., September 1994, pp. 211-254.

[5] Krause, Micki and Tipton, Harold F. editors. *Handbook of Information Security Management.* CRC Press LLC, 1998.

[6] Bishop ,Matt and Dilger, Michael. "Checking for Race Conditions in File Accesses." URL: http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/ucd-ecs-95-09.pdf (7 Feb. 2003)

[7]Tanenbaum, A. S. *Operating Systems Design and Implementation*. Prentice-Hall, Inc. 1987.

[8] John Viega , et al. "Token-based scanning of source code for security problems." ACM Transactions on Information and System Security (TISSEC) VOl. 5, No. 3, August 2002.

[9] Nazario, Jose. "Source Code Scanners for Better Code." LinuxJournal.com, 26 Jan 2002. URL: http://www.linuxjournal.com/article.php?sid=5673 (21 Mar 2003).

[10] Tsyrklevich, Eugene. "Dynamic Detection and Prevention of Race Conditions in File Accesses." http://www.cs.ucsd.edu/users/etsyrkle/thesis.ps.gz (Mar 20 2003)

[11] Leffler, J.*et al*. "Re: Re: Error Handling." dbi-users mail list archive. URL: http://opensource.nailabs.com/lomac/docs/lomac-freenix01.pdf (6 Feb. 2003)

[12] Dean, Drew. “The Security of Static Typing with Dynamic Linking.” URL: http://www.cs.princeton.edu/sip/pub/ccs4-preprint.pdf (20 Mar. 2003)

[13] Fraser, Timothy. “LOMAC: MAC You Can Live With.” URL: http://www.usenix.org/events/usenix01/freenix01/full_papers/fraser/fraser_html/node4.html (21 Mar 2003)

[14] Stanley, Richard A. “CS 697 Computer Security.” URL: http://www.cs.umb.edu/cs697/CS697A-Class 1.ppt (21 Mar 2003)

[15] Lowery, J. Craig. “A Checklist for Network Security.” *Dell Power Solutions*, May, 2002. URL: http://www.dell.com/us/en/esg/topics/power_ps2q02-lowery.htm (21 Mar 2003)