Dynamic Detection and Prevention of Race Conditions in File Accesses

Eugene Tsyrklevich
eugene@securityarchitects.com

Bennet Yee
bsy@cs.ucsd.edu

Department of Computer Science and Engineering
University of California, San Diego

Abstract

Race conditions in filesystem accesses occur when sequences of filesystem operations are not carried out in an isolated manner. Incorrect assumptions of filesystem namespace access isolation allow attackers to elevate their privileges without authorization by changing the namespace bindings. To address this security issue, we propose a mechanism for keeping track of all filesystem operations and possible interferences that might arise. If a filesystem operation is found to be interfering with another operation, it is temporarily suspended allowing the first process to access a file object to proceed, thereby reducing the size of the time window when a race condition exists. The above mechanism is shown to be effective at stopping all realistic filesystem race condition attacks known to us with minimal performance overhead.

1 Introduction

Dating back to the 1970's, race conditions, otherwise known as the Time Of Check To Time Of Use (TOCTTOU) bug, are one of the oldest known security flaws. They are a class of local vulnerabilities that can be used to elevate one's privileges without authorization and occur when operations are not carried out atomically as expected.

Race conditions have been found in a variety of scenarios ranging from tracing processes [9] to filesystem accesses [13]. This paper focuses on a subset of race conditions dealing with filesystem accesses. The threat model is that of an adversary with unprivileged access to a system who is trying to elevate his or her privileges by exploiting a filesystem race condition vulnerability.

Some security vulnerabilities, such as buffer overflows, receive a lot of attention and have a variety of solutions to prevent them. Other vulnerabilities, such as race conditions, receive far less attention and have no definite solutions to stop them. In addition, race conditions can be very hard to spot and eliminate at compile time, it is thus beneficial to develop a dynamic and automated protection against this type of flaws.

This paper presents an analysis of filesystem race condition vulnerabilities and ways to prevent them. It also describes a prototype system that detects and stops all realistic filesystem race condition attacks known to us. The proposed solution stops the attacks with minimum performance overhead, while retaining application compatibility.

The rest of this paper is organized as follows. Section 2 describes race conditions in filesystem accesses and presents several sample vulnerabilities. The design of the prototype system and the rationale behind it is covered in Section 3. Section 4 details the implementation of the prototype system, as well as some of the decisions made during the design phase incorporating the security of the system and code portability. Section 5 reviews the attack scenarios used for the security testing of the prototype system. Section 6 describes the compatibility testing and Section 7 covers the system performance. Section 8 reviews related work, Section 9 proposes future work and Section 10 provides the conclusions.

2 Race Conditions in File Accesses

Race conditions in filesystem accesses occur when applications incorrectly assume that a sequence of filesys-

```
if (access(fn, R_OK | W_OK) == 0) {
        fd = open(fn, O_RDWR, 0644);
        if (fd < 0)
        {
                err(1, fn);
        }

        /* use the file */
```

Figure 1: Race Condition in Access Operation

tem operations is atomic and the filesystem namespace is otherwise static. In reality, attackers can modify the filesystem namespace at any time by replacing files with symbolic links or by renaming files and directories.

2.1 Access Operation

The code fragment in Figure 1 is taken from an insecure setuid application that needs to check whether a user running the program has the right to write to a file (since the application is setuid root, it can write to any file).

The source code incorrectly assumes that a file object referenced by the filename "fn" does not change in between the `access` and `open` invocations. The assumption is invalid since an attacker can replace the file object "fn" with another file object (such as a symbolic link to another file), tricking the program into accessing arbitrary files. To avoid this race condition, an application should change its effective id to that of a desired user and then make the `open` system call directly or use `fstat` after the `open` instead of invoking `access`.

2.2 Directory Operations

Another subtle race condition was recently discovered in the GNU file utilities package [13]. During the recursive removal of a directory tree, an insecure `chdir("..")` operation is performed, after removing the content of a subdirectory, to get back to the upper directory. An attacker, who is able to exploit this race condition, can trick users into deleting arbitrary files and directories. The bug is triggered when an attacker moves a subdirectory, that the rm utility is processing, to a different directory.

Figure 2 contains a slightly abbreviated system call trace of a recursive removal of the `/tmp/a` directory tree that contains two nested subdirectories. Consider what happens when an attacker moves `/tmp/a/b/c` directory to `/tmp` before the first `chdir("..")` call is invoked. After removing the contents of the `/tmp/a/b/c` subdirectory (now `/tmp/c`) and executing two `chdir("..")` calls, instead of changing the directory to `/tmp/a`, the `rm` utility would find itself in the root directory that it would continue to delete.

```
chdir("/tmp/a")
chdir("b")
chdir("c")
unlink("*")
[attacker entrypoint: "mv /tmp/a/b/c /tmp"]
chdir("..")
rmdir("c")
unlink("*")
chdir("..")
rmdir("b")
unlink("*")
rmdir("/tmp/a")
```

Figure 2: Race Condition in Directory Operations

The race condition was fixed by adding additional checks to verify that the inode of the directories did not change before and after the `chdir` calls; in other words, the directory tree namespace did not change during the execution of the tool.

2.3 Setuid Shell Scripts

The secure execution of setuid shell scripts is a well-known problem in the earlier Unix systems. To determine whether a file is a script and should be processed by an interpreter, a Unix kernel checks the first two bytes of a file for a special "`#!`" character sequence. If the character sequence is found, the rest of the line is treated as the interpreter pathname (and its parameters) that should be executed to process the script. Unfortunately, in some kernels, the operation of opening and processing the first line of a file and the subsequent execution of the interpreter that reopens the script is not atomic. This allows attackers to trick the operating system into processing one file while executing the interpreter with root privileges on another file.

Various Unix systems deal with this problem in differ-

```
if (stat(fn, &sb) == 0) {
    fd = open(fn, O_CREAT | O_RDWR, 0);
    if (fd < 0)
    {
        err(1, fn);
    }

    /* use the file */
```

Figure 3: Race Condition in Temporary File Accesses

ent ways. Earlier Linux systems simply disregarded the setuid bits when executing the shell scripts. Some SysV systems as well as BSD 4.4 kernels pass to the interpreter a filename in a private, per-process filesystem namespace representing the already opened descriptor to the script (e.g. `/dev/fd/3`) in lieu of the original pathname. Passing such a pathname closes down the race condition vulnerability window since it is impossible for an attacker to change the private per-process filesystem namespace bindings.

2.4 Temporary File Accesses

Another type of race condition occurs in the handling of temporary files. An unsafe way to create a temporary file is to use the `stat` system call to check whether a file exists and if the file does not exist, invoke `open` to create one. The problem with this approach is that the sequence of `stat` followed by `open` is not atomic (similarly to the `access` and `open` system call sequence described above). This allows an attacker to create a symbolic link with the same filename after the `stat` reports that the file does not exist. When the `open` system call with the `O_CREAT` flag executes, it follows the attacker's link and overwrites the contents of the file to which the link points. Figure 3 shows the vulnerable code.

The correct way to do the same thing would be to invoke `open` with `O_CREAT` and `O_EXCL` flags set. The `O_EXCL` flag causes the `open` to fail when the file already exists.

3 System Design

The examples in the previous section illustrated that race conditions could occur under a wide variety of conditions. The theme that is present in all of the examples is that programmers erroneously assumed that their program is the only program accessing the filesystem. Instead, an attacker can change the filesystem namespace in the midst of a sequence of filesystem system calls and exploit the race condition in the code. We call the sequence of filesystem system calls that programmers assumed to be free from interference a *pseudo-transaction*. To address the problem of race conditions, it would be sufficient to provide the illusion of a filesystem namespace that behaves as if these pseudo-transactions were executed in isolation and free from interference. To achieve this, the system needs to be able to detect race condition attacks and stop them.

3.1 Transactions

The goal of detecting and preventing race conditions would be trivially achieved if executions of processes (or process groups) were transactions that enjoyed the ACID properties (Atomicity, Consistency, Isolation and Durability). ACID transactions were designed to permit concurrent database accesses while maintaining correctness. These ideas can be readily translated to filesystem accesses.

The atomicity property requires that a transaction commits all the involved data or none at all.

The consistency property guarantees that a transaction produces only consistent results; otherwise it aborts. Therefore, filesystem operations belonging to a transaction always produce coherent results without external interferences.

The isolation property requires that a running transaction remains isolated from all other running transactions. In the race protection system, this means that the filesystem operations in one transaction cannot effect the results of filesystem operations belonging to other transactions. This is the main characteristic that the race protection system tries to achieve.

Finally, durability requires that the results of the completed transactions must not be forgotten by the system.

The ideal operating system for the task would satisfy all of the above properties. QuickSilver is an example of one such system that provides built-in transaction primitives [14]. Unfortunately, the mainstream operating systems have no built-in support for transaction processing. In the absence of transaction processing, the race protection system has to at least guarantee the isolation property which requires the system to be able to detect the race condition attacks before they take place. Furthermore, the system must be able to enforce the property by preventing the race conditions from being exploited.

3.2 Detecting Race Conditions

Unfortunately, application programmers do not include code that informs the kernel when a pseudo-transaction begins or ends. We can, however, use heuristics to analyse filesystem-related activities and attempt to identify the pseudo-transactions. Below, we address how we identify the start of pseudo-transactions and policies to determine if a filesystem operation request would interfere with an in-progress pseudo-transaction.

3.2.1 Scope of Pseudo-Transactions

By monitoring filesystem operations, the system tries to distinguish between normal and potentially malicious filesystem activities. First, the simple heuristics used for this recognize system calls that potentially start pseudo-transactions. Then, they prevent subsequent system calls from interfering with the pseudo-transaction. Filesystem calls that do not interfere are executed normally, while those that do are handled as described in Section 3.3. In our analysis of filesystem activities, we found that pseudo-transactions that involve sequences of more than two operations are not common. As a result, they are not addressed by our implementation. This allows us to end a potential pseudo-transaction at the next allowed system call, which may in turn start a new pseudo-transaction. We also allow a pseudo-transaction to end via a heuristically determined time out value (see Section 4.3); this ensures that misidentified pseudo-transactions will not persist in the system for too long.

We assume that a process does not maliciously interfere with its own filesystem accesses. Furthermore, `fork`'ed child processes run the same process image and are assumed to be trustworthy until an `exec` replaces their process image. This means that the granularity of detection is at the process level—all filesystem operations originating from a process will be treated as part of a single pseudo-transaction, even if the process performs work on behalf of multiple clients (e.g., a web server).

REMOVE = UNLINK | RMDIR | RENAME

DENY(ACCESS, REMOVE)
DENY(CHDIR, REMOVE)
DENY(EXEC, REMOVE)

Figure 4: Default Allow Policy

3.2.2 Policies

In order to detect potential race conditions, we propose two different policies. One policy disallows operation sequences that are prone to filesystem races, while the second policy only allows the operation sequences that are known to be safe. Both policies take into consideration the current filesystem operation and the last operation that dealt with the same file. If the operation sequence is found to be legal by the chosen policy, the current operation is allowed to proceed. Otherwise, the policy returns an error that indicates that a potential race condition is in progress.

Default Allow Policy As the name suggests, a default allow policy permits all filesystem operations to proceed unless they match one of the deny rules. Deny rules (listed in Figure 4) are constructed to address and stop specific race condition attacks. For example, the `DENY(ACCESS,REMOVE)` rule does not allow the `access` operation to be followed by any of the remove operations if they involve the same file object. All other filesystem operations, however, are allowed to proceed.

The above mentioned rule prevents the attack described in Section 2.1, the second deny rule addresses the race condition described in Section 2.2 while the third rule deals with the attack described in Section 2.3. Race conditions in temporary file accesses are treated separately as described in Section 4.2.

A default allow policy explicitly forbids sequences of filesystem operations that are prone to race conditions. As a result, the policy should detect all listed race condition attacks without causing any false positives. However, it may miss new attacks if they involve operation sequences not explicitly stated in the policy.

Default Deny Policy A default deny policy disallows all filesystem operations to proceed unless they match one of the permit rules. Permit rules (listed in Figure 5) are constructed to describe all known

```
OPEN_RW = OPEN_READ | OPEN_WRITE
RENAME = RENAME_TO | RENAME_FROM

PERMIT(OPEN_RW,            OPEN_RW | ACCESS | UTIMES | CHDIR | EXEC |
                           UNLINK | READLINK | CHMOD | CHOWN | RENAME)
PERMIT(OPEN_CREAT,         OPEN_RW | ACCESS | UTIMES | CHDIR | EXEC | RENAME_FROM)
PERMIT(ACCESS,             OPEN_RW | ACCESS | UTIMES | CHDIR |EXEC)
PERMIT(EXEC,               OPEN_READ | EXEC)
PERMIT(CHDIR,              OPEN_READ | CHDIR | ACCESS | READLINK)
PERMIT(RENAME_FROM,        OPEN_RW | ACCESS | UNLINK | RENAME_FROM)
PERMIT(RENAME_TO,          OPEN_RW)
PERMIT(CHMOD | CHOWN,      OPEN_RW | ACCESS |  CHMOD | CHOWN)
PERMIT(UTIMES,             OPEN_RW | ACCESS | CHMOD | CHOWN)
PERMIT(READLINK,           READLINK)
```

Figure 5: Default Deny Policy

valid filesystem operation sequences that we have observed in a normal system. For example, the `PERMIT(EXEC,OPEN_READ|EXEC)` rule allows an `exec` operation to be followed by another `exec` or by an `open` operation. All other operations involving the same file object will be flagged as race conditions.

The effectiveness and robustness of the race protection system depends on the policy in place and the quality of the rules in the chosen policy. A default deny approach should prevent any existing race condition attacks and possibly some of the new future attacks.

Future attacks are addressed by the existing rules that deal with all filesystem operations and not just the ones that are known to be susceptible to race conditions. Since each rule is a simple C macro that expands to an if statement, even a large number of rules should have negligible effect on the overall system performance.

3.3 Preventing Race Conditions

Once a race condition is identified, an error message is logged and a preventative measure must be taken. An error message should contain enough information to track down who caused the race condition and how. The approach implemented in our system is described in Section 3.3.5.

3.3.1 Transaction Rollback

One of the most sophisticated way to handle a race condition is to restart or abort the involved transactions (provided a transaction-enabled system such as QuickSilver [14] is used). Aborting the transactions should rollout any possible changes that the processes made up to that point. This solution stops race conditions without leaving the system in an inconsistent state.

3.3.2 User Confirmation

In the absence of a transaction-enabled system, the system might ask a user what to do. Similar to a systrace GUI [12], a pop-up dialog might query a user about a possible action to be taken. This approach has the advantage of not being intrusive as it is up to the user to decide whether to kill a process, suspend a process or take some other action. However, interacting with users is more suitable for desktop machines rather than servers which might not have live personnel in front of them 24/7. In addition, an attacker might overwhelm a system with a high number of alerts causing a user to ignore the numerous dialog boxes.

3.3.3 Locking Out Processes

Another option that, at first glance, would seem to work well is to preclude any other processes from being scheduled during a sequence of critical (race-prone) accesses. Programs can be “wrapped” with a script which tells the scheduler to protect them from interfering accesses. This approach is analogous to having the process place a single lock on the entire filesystem.

While this would work in theory, there are many drawbacks that make this approach impractical. It is not clear whether it is possible to determine all the vulnerable critical sections correctly. As a data point, after kernel support for setuid script execution was added to the BSD kernel, about a decade passed before the race condition was noticed. It is also not clear whether this approach might cause deadlocks or invite denial-of-service attacks. Finally, since this locking system is very coarse grained, the system performance would suffer greatly.

3.3.4 Killing Processes

Another way way to prevent a race condition is to kill the involved processes. An alternative might be to silently fail all the consecutive system calls belonging to the involved processes and hope that the applications can cope with this and gracefully shutdown. While these solutions definitely prevent any possible abuse, they are too crude and are subject to denial-of-service attacks where an attacker might try to trick the system into killing "innocent" victim processes.

3.3.5 Suspending Processes

A better solution would minimize the effect of false positives—without killing the involved processes or otherwise completely locking files and preventing forward progress—and still prevent filesystem race attacks. Our system achieves this by allowing only one process to go on while *temporarily* suspending the other. In an attack scenario, this approach allows the victim process (the first to access a file object) to safely continue using the file while suspending the activity of an attacker process. Since a race condition is defined to exist only for brief amounts of time, delaying an attacker process is an effective way to prevent race conditions. In the scenario where two processes just happened to execute an unsafe combination of filesystem operations, the proposed solution of suspending a process allows one process to continue while preventing the second process from causing any unintentional damage. As both processes are eventually allowed to proceed, this scenario is equivalent to that of a highly-loaded machine where processes can be suspended or swapped out for extended periods of time before being allowed to run.

When a process invokes a system call that initiates a pseudo-transaction, it is marked as a "delay-lock owner" for the file. Processes attempting to access delay-locked files with interfering system calls are not completely locked out, but are instead temporarily suspended to try to prevent interference with the pseudo-transaction filesystem sequence. Delay-lock owners of files are never misidentified as interfering processes and thus delayed. Delay-lock ownership is inherited across `fork`'s and is given up at `exec`'s. Therefore, a delay lock may have many owners.

Note that we do not distinguish between parent and child processes in determining which is the attacker and which is the victim. Pseudo-transaction recognition is based solely on the global stream of system calls. Although the `fork` system call (see Section 4.4) is mediated, we only use it to allow child processes to inherit delay-lock ownership. Because we do not use per-process filename caches nor the asymmetric cache clearing scheme used by RaceGuard [6], an attack due to Casper Dik [7] is impossible.

Suspending a process is an effective technique to prevent race conditions, but it is not completely foolproof. If the attacker process wakes up before the victim process gets a chance to complete its operations, the race condition will still be present. To address this issue, the delay is calculated at runtime by taking into account the load average of a system. It would also be possible to allow a user application to control the delay by providing a new system call, but we did not implement this. While we have chosen the delay to be long enough for a process to be able to "close" its vulnerability time window, the system does not in any way detect or enforce that this actually holds.

4 Implementation

The prototype system was developed for the OpenBSD operating system [10], which was chosen for its focus on security. The system consists of a kernel module that intercepts a number of system calls. Even though modifying the operating system kernel code directly would provide a slightly faster solution, using kernel modules saves time writing and debugging the kernel code. It should be noted that mediating system calls introduces a race condition whereby an attacker changes the filename after it is processed by the mediated call but before it is processed by the original system call [8]; an integrated implementation that implements our technique directly in the kernel code would not leave this (small) timing window open.

4.1 File System Calls Mediation

The system calls that the kernel module intercepts are mainly the filesystem calls that accept pathnames as their parameters. The mediated system calls perform processing on the pathnames before returning the control to the original system calls. The pathname processing starts with the conversion of a name to its inode by means of the `namei` kernel function. The reason for dealing with inodes rather than names is to properly handle different pathnames that refer to the same file (e.g. `/etc/passwd` vs `/etc/../etc/passwd`).

Once a pathname is resolved to an inode, the system checks if any of the preceding operations affected the same inode. If this is the case, further checks need to be carried out to make sure no race conditions exist. Otherwise, all the related operation information is saved and the operation is allowed to proceed.

As mentioned above, there are two standpoints that can be adopted to determine whether a race condition exists between the two filesystem operations involving the same inode: default allow and default deny policies. With either policy in place, the system first compares the process ids of the involved processes. If they are the same, a race condition cannot exist since both operations originated from the same process. If not, the two operations are checked against the chosen ruleset. If the operation sequence is found to be legal, the information about the current operation is saved for later use. Otherwise, an error code is returned and the appropriate action to prevent the race condition is carried out (e.g., a process is temporarily suspended). It should be noted that, in the case of a false positive, the applications continue to make forward progress, albeit at a slower rate.

4.2 Temporary File Race Condition Processing

Temporary file race condition processing differs from the above described procedure. When the `stat` system call is invoked on a non-existent file, there is no inode value to process (since the file does not exist) and therefore the inode processing code fails to work. To address this problem, a separate linked list is used to keep track of all `stat` operations on the non-existent files (the same technique is used by RaceGuard [6]). When an `open` system call with an `O_CREAT` flag set (and no `O_EXCL` flag) is invoked on an existing file, the filename passed to the `open` is checked against the list of the non-existent filenames on the `stat` list. If there is a match between the names, an attack might be in progress and the system aborts the `open` system call with the file-already-exists error code (this is the `O_CREAT|O_EXCL` behavior).

As mentioned above, this approach deals with the filenames rather than the inodes because no inode value exists initially. Dealing with filenames in this case is not a problem since the filenames that are compared originate from the same process and are trusted to be consistent.

4.3 Data Management

To keep track of all the filesystem operations, the operation related information (process id, current time, file operation, inode and the corresponding pathname) is saved in a hash indexed by the inode value. As mentioned before, a new entry is added whenever a filesystem operation executes. The operation entries are removed by a cleaner routine that traverses the hash every second and removes all stale operation entries. A stale entry is defined to be an entry that originated more than 2 seconds ago (or 15 seconds for a directory entry) plus the current load average. For example, a load average of 1.58 causes the timeout to be 2+1.58=3.58 seconds or 15+1.58=16.58 seconds for a directory entry.

4.4 Other System Calls

Besides all the filesystem calls that are intercepted, there are several non-filesystem related system calls that need to be handled—`fork`, `exec`, and `exit`.

The `fork` system call creates a new process, and it needs to be intercepted to allow for duplication of all the entries associated with the current process. Duplicating the entries with the new process id permits file sharing between the parent and the child process. For example, a parent process might `fork` a child process that deletes some temporary files created by the parent. Without duplication of the parent entries, the child process would be found to be interfering with the parent process.

While `fork` spawns off processes that the race protection system deems to be trusted by the parent process, the `exec` system call replaces the current process image; this new process image should not, in general, be trusted. The mediated exec system call thus removes all the entries associated with the exec'ing process. In addition, to be able to handle the setuid shell script attack described in Section 2.3, the pathnames passed to the exec system call are also inserted into the hash table.

Finally, the exit system call causes the process to terminate and this allows us to remove all the hash entries associated with the terminating process.

4.5 Portability

The code was carefully designed for maximum portability. The system dependent calls are mostly hidden behind macros that greatly simplify the porting process. The parts of the code that are OS dependent are kernel module details, system call hooking, pathname to inode resolution and locking.

To simplify the porting process even further, the code for the majority of the mediated file system calls is automatically generated by a perl script. The script automatically generates all the function declarations, all the necessary data structures as well as the function code itself. Thus the majority of the OS dependent code can be ported by modifying a small perl script.

Finally, there is nothing inherent in the design of the race protection system that prevents it from being ported to non-UNIX operating systems such as Windows.

5 Security Testing

To test the effectiveness of the race protection system, four attack scenarios were developed and carried out with the race protection being disabled and then enabled. All the attacks were successfully detected and stopped (with both default allow and default deny policies).

All of the attacks, except for the temporary file vulnerability, involve the attacker process being suspended whilst the victim process continues to run and closes the vulnerability window. The temporary file race vulnerability involves failing the open system call and forcing the victim process to deal with the consequences.

This section illustrates the attack against a race condition in using the access system call. The rest of the attacks are described in [16].

Figure 6 illustrates a successful race condition attack against a sample victim program which contains the vulnerable access code introduced in Section 2.1. The figure presents a time line with events on the left as seen by the victim and events on the right as seen by the attacker.

In step 1, the victim program is executed and it prints out the information message after calling the access system call. The victim process then sleeps for several seconds to simplify the timing attack. Step 2 illustrates the attack taking place in between the access and open system calls. Some time after the attack takes place, a victim process wakes up, opens the /tmp/race file and prints out its content which happens to be the /etc/passwd file setup by the attacker. If the victim program wrote to the file instead of reading it, the result might have been a complete system compromise. The same attack is effective against real world programs that exhibit this flaw even without the artificial presence of sleep calls.

Figure 7 illustrates the same attack taking place with the race protection system in place and the attack being thwarted.

Similarly to the previous example, the same vulnerable victim process is executed. This time though, the race protection system does not allow the /tmp/race file to be removed in step 2 since it would violate the filesystem namespace integrity (in other words, the ruleset dictates that an access operation is not allowed to be followed by an unlink). Instead, the attacker initiated process is suspended while a victim process gets a chance to finish its file operations without the attacker's interference. After five seconds (delay = base 2 seconds + the system load average of 3 = 5), the attacker process is allowed to proceed and the /tmp/race file is replaced with a symbolic link without any consecutive security side-effects.

6 Compatibility Testing

In addition to the security tests described above, the race protection system (with both default allow and default deny policies) was used on several desktop and server machines for a period of several weeks. After initial adjustments to the default deny policy, no false positives or false negatives were experienced under realistic work loads.

The race protection system was also stress tested by running multiple OpenBSD kernel compile processes in parallel with a locate.updatedb application that in-

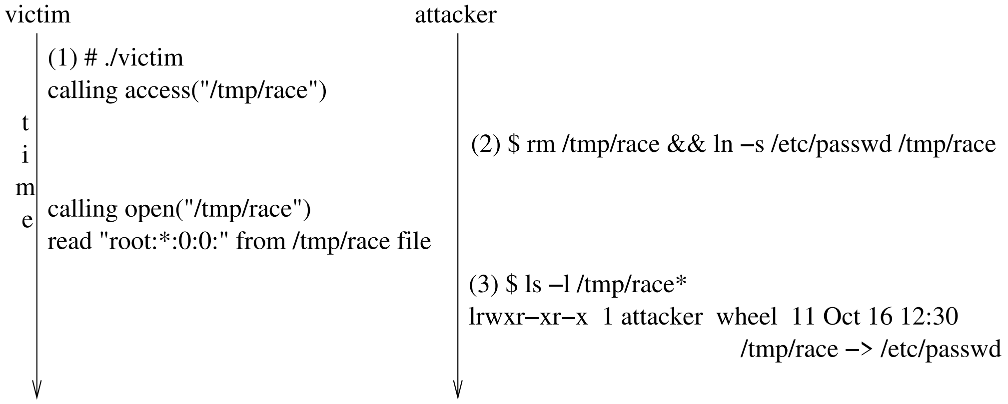

Figure 6: Successful Access Operation Attack

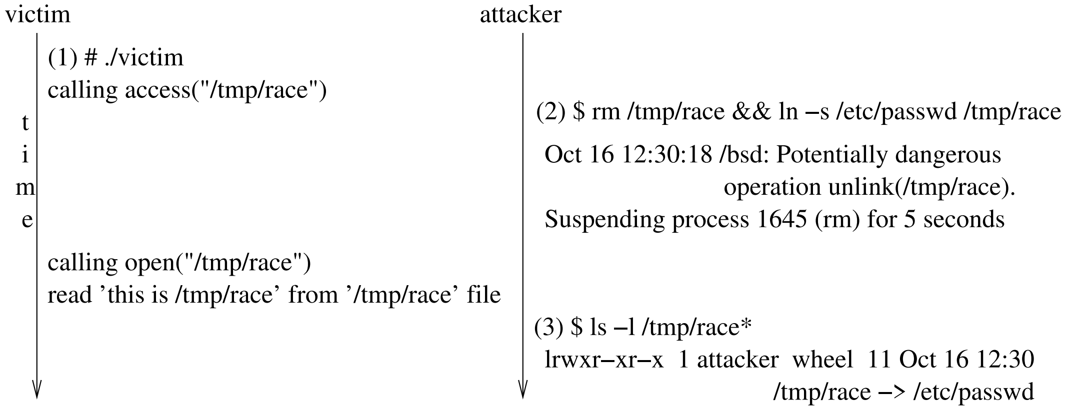

Figure 7: Unsuccessful Access Operation Attack

vokes `stat` on all the existing files. We observed no false negatives under these high loads.

This compatibility testing showed that there are no ill effects since false positives do not usually arise. Note, however, that even if false positives did arise, the only impact would be a delay in the execution of the process that was erroneously identified as an attacker.

7 Performance

To measure the overhead of the race protection system, several benchmarks were carried out on a system where the race protection was switched on and off. The tests were carried out on an OpenBSD 3.2 system running on a 1 GHz AMD Athlon computer with 256 megabytes of RAM and a 60Gig Western Digital hard drive with 8.9 ms access time.

7.1 Microbenchmarks

The first measured overhead is that of individual mediated system calls. To measure the load, each system call was called 10,000 times in a tight loop. Each test was carried out 10 times and the numbers were averaged. Table 1 shows the performance results.

As expected, the mediated system calls that need to perform path processing and check for any possible race conditions have a high overhead. For the `open` system call, the overhead of that processing amounts to over 100%. The `stat` system call does not perform any path processing and has an overhead of only 3%. The `fork` system call takes a long time to execute and needs to do little race condition processing; it exhibits relatively little overhead.

While 100% overhead of some mediated system calls might seem like an unacceptable solution, the next section illustrates that the overall application overhead is

System Call	open	stat	fork
Race Protection Off, microseconds	2.55	3.28	86.17
Race Protection On, microseconds	5.69	3.38	86.21
Total CPU Overhead (%)	123	3	0

Table 1: Microbenchmark Performance Results

	Real Time	User Time	System Time
Race Protection Off (sec)	427	363	37
Race Protection On (sec)	436	363	43
Total Overhead (%)	2	0	16

Table 2: Macrobenchmark Performance Results

much lower.

7.2 Macrobenchmarks

To measure the overall overhead, the OpenBSD kernel compile process was used as the benchmark. The compile process is computationally expensive but it also involves processing several thousand source and header files as well as creating and destroying several thousand processes. Therefore the benchmark represents a realistic workload of a loaded server.

Table 2 presents the performance numbers. As expected, the code executing in user mode shows no overhead. The code executing in kernel mode incurs a 16 percent overhead induced by the race protection system. The kernel overhead, however, is amortized over the overall benchmark execution leading to a much lower total overhead of 2 percent.

The total dynamic memory consumption initially peaks at 400Kb when the make process touches a large number of files at the same time. After the initial peak, the average memory consumption stays around 3 Kb for the rest of the benchmark process.

8 Related Work

A great deal of TOCTTOU research to date has focused on developing static tools and models for detecting race conditions at compile time. While several runtime solutions were proposed to resolve some of the problems associated with static tools, they have their own limitations.

8.1 Static Analysis

One of the first known static tools for detecting security flaws at compile time was developed as a result of the Protection Analysis project initiated at ISI by ARPA IPTO in the mid-1970's [3]. The goal of the project was to understand application and operating system security vulnerabilities and to identify "automatable pattern-directed techniques" for detecting security vulnerabilities. One of the flaws identified by the project was the "improper change" (TOCTTOU) flaw which occurred as a result of a parameter changing unexpectedly.

Another security project undertaken in the 1970's was RISOS (Research Into Secure Operating Systems) [1]. The RISOS project was a study of computer vulnerabilities in operating systems. One of the identified flaws was "inadequate serialization" which corresponds to a race condition between the `access` and `open` system calls as outlined in Figure 1.

Bishop and Dilger at UC Davis were one of the first to focus on race conditions in filesystem accesses [5]. They developed a static analysis tool for finding race conditions in C code. Their tool does not perform alias analysis nor is it control-flow aware. As a result of this, the tool can produce false negative and false positive results.

Another similar tool developed by Anguiano is based on ASTLOG [2]. Other static tools for detecting security vulnerabilities, among them race conditions, are ITS4 [17], Flawfinder [18] and RATS [15].

While the static source code analysis of TOCTTOU bugs is a valid approach, it suffers from several drawbacks. First, static tools do not have access to the runtime information needed to resolve any possible ambiguities. Filenames and environment information are not always known at compile time and even alias analysis and other proposed compiler techniques cannot solve this problem. Second, static tools are known to produce false positive as well as some false negative results [5, 2]. The reason for this being imprecise code analysis techniques as well as the absence of runtime information. Finally, the existing tools require access to the source code, which is not always available. Even if the source code is available, the time needed to run the code through one of the static tools and manually analyze the results, makes the static analysis approach less than satisfactory when trying to secure modern day systems.

8.2 Dynamic Analysis

One of the first runtime solutions was proposed by Bishop [4]. It requires applications to be modified to call a new library function that determines if a file object is safe to use. This approach is not transparent to applications and cannot be easily used to secure systems.

Another runtime solution, proposed by Solar Designer, addresses the problem by limiting users from following untrusted symbolic links created in certain directories and by limiting users from creating hard links to files they don't have read and write access to [11]. While this approach might be effective in stopping some race condition attacks, it is too limited to stop other variations of attacks. In addition, this approach breaks the applications that depend on the behavior restricted by the solution.

Similarly, RaceGuard focuses on race conditions that occur during the creation of temporary files [6]. While this is a valid approach, it is limited to temporary file race vulnerabilities.

9 Future Work

While we tried to address all known practical race conditions, there are still some questions left unanswered.

First, our implementation does not strictly enforce the correct behavior of allowing a victim process to proceed first and close its race condition window. The heuristics of dynamically calculating the delay performed well in our lab tests but this is no guarantee that they will work in all scenarios. Our delay formula and hash table cleaning times were experimentally created using ad-hoc methods. A proper justification of these, via an analysis of programmer isolation expectations and process runtimes, remains to be provided. Future implementations should address this issue more carefully.

Second, our implementation does not address the issue of subdirectory locking. As described in Section 2.2, attackers can move directories around thereby tricking processes into accessing arbitrary files. Future implementations should analyze this threat better for all race condition cases and not just the case described in Section 2.2.

Third, our implementation does not address the following race condition attack. Suppose an old style shell program that does not implement `test` functionality as a built-in function is used to interpret a script. (Alternatively, a script could be written to explicitly use `/bin/test` rather than the built-in function.) The shell `fork`'s a subprocess which `exec`'s the `test` program. The subprocess then invokes the system call `stat` and communicates its result to the shell via the process exit status. Since the subprocess has exited, the hash table entries pertaining to the file will be deleted by the `exit` system call mediation code. No hash entry would correspond to the parent process, since the shell only handles the file as a string. Based on the result of the `stat`, e.g., in a command such as `/bin/test -r $file || foo > $file`, another subprocess uses the file. This is a scenario that should be rare, but is nevertheless not detected by our scheme. Note that the RaceGuard scheme [6] is unable to protect against this as well. Better analysis and improved heuristics are required to detect this kind of pseudo-transaction.

Finally, our implementation only deals with timing races—i.e., race conditions that exist for short periods of time in between two file system call invocations. We do not handle other types of race conditions such as storage races. An example of a storage race condition is a predictable `/tmp` filename attack which involves an attacker planting a malicious link in the `/tmp` directory. At a later time, a victim process follows the link and clobbers an arbitrary file. Future implementations should try to classify all existing race condition types and ways to deal with them.

10 Conclusions

Race conditions in filesystem accesses occur when applications incorrectly assume that a sequence of filesystem operations is isolated and the filesystem namespace is otherwise static. Even though the existence of the TOCTTOU bugs dates back to almost 30 years, new race conditions are still being discovered today.

To prevent race conditions, a system should provide an illusion of an immutable namespace existing without external interferences. This goal resembles that of a transaction enabled system where the properties of atomicity, consistency, isolation and durability have to be satisfied. Since mainstream operating systems are not designed to satisfy any of the ACID properties, the race protection system has to emulate certain features to achieve the desired effect. A system that creates an illusion of an immutable namespace should satisfy the isolation property and should be effective in stopping race conditions. An immutable namespace effect is achieved by detecting and stopping race conditions. Race conditions are detected by tracking all file-related system activity and identifying any filesystem operation sequences that could possibly result in a TOCTTOU condition. When two filesystem operations are found to be interfering with each other, the one to access a file object last is suspended. This allows the first operation to proceed without any interferences. This solution stops all file race conditions attacks known to us and is also designed to prevent future attacks. It does this with minimum performance overhead while retaining application compatibility.

11 Acknowledgments

We would like to thank Christina Martinez for proofreading the paper, Peter Bartoli and Marshall Beddoe for providing us with hardware testbeds, Crispin Cowan for shepherding this paper and Keith Marzullo, Stefan Savage and the anonymous reviewers for their comments and suggestions.

This work is supported in part by the Office of Naval Research, Award N00014-01-1-0981. The opinions expressed in this paper are those of the authors.

References

[1] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security Analysis and Enhancements of Computer Operating Systems. NBSIR 76–1041, Institute for Computer Sciences and Technology, National Bureau of Standards, April 1976.

[2] R. Anguiano. A Static Analysis Technique for the Detection of TOCTTOU Vulnerabilities. Master Thesis, University of California Davis, 2001.

[3] R. Bisbey and D. Hollingsworth. Protection Analysis Project Final Report. ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute, May 1978.

[4] M. Bishop. Race Conditions, Files, and Security Flaws: or, The Tortoise and the Hare Redux. Technical Report 95-8, Department of Computer Science, University of California at Davis, September 1995.

[5] M. Bishop and M. Dilger. Checking for Race Conditions in File Accesses. Computing systems, pages 131–152, Spring 1996.

[6] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman. RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities. In Proceedings of the 10th USENIX Security Symposium, August 2001.

[7] C. Cowan. Personal communication.

[8] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In Proceedings of the Network and Distributed Systems Security Symposium, February 2003.

[9] G. Guninski. OpenBSD 2.9,2.8 local root compromise. Bugtraq mailing list, June 2001.

[10] The OpenBSD Project. `http://www.openbsd.org/`

[11] Openwall Project. Linux kernel patch. `http://www.openwall.com/linux/`

[12] N. Provos. Improving Host Security with System Call Policies. CITI Technical Report 02-3, November 2002.

[13] W. Purczynski. rm - recursive directory removal race condition. Bug-fileutils mailing list, March 2002.

[14] F. Schmuck and J. Wyllie. Experience with transactions in QuickSilver. In Proceedings of the 13th ACM Symposium on Operating System Principles, pages 239–53, October 1991.

[15] Secure Software. Rough Auditing Tool for Security (RATS). http://www.securesoftware.com/rats.php

[16] E. Tsyrklevich. Dynamic Detection and Prevention of Race Conditions in File Accesses. Master Thesis, University of California San Diego, 2002.

[17] J. Viega, J. Bloch, T. Kohno, and G. McGraw. ITS4: a static vulnerability scanner for C and C++ code. In proceedings of 16th Annual Computer Security Applications Conference, December 2000.

[18] D. Wheeler. Flawfinder. http://www.dwheeler.com/flawfinder