Secure programmer: Prevent race conditions

Resource contention can be used against you

David Wheeler (dwheelerNOSPAM@dwheeler.com), Research Staff Member, Institute for Defense Analyses

Summary: Learn what a race condition is and why it can cause security problems. This article shows you how to handle common race conditions on UNIX-like systems, including how to create lock files correctly, alternatives to lock files, how to handle the file system, and how to handle shared directories -- and, in particular, how to correctly create temporary files in the /tmp directory. You'll also learn a bit about signal handling.

Date: 07 Oct 2004
Level: Intermediate
Also available in: Japanese

Using a stolen password, "Mallory" managed to log into an important server running Linux. The account was limited, but Mallory knew how to cause trouble with it. Mallory installed and ran a trivial program with odd behavior: It quickly created and removed many different symbolic link files in the /tmp directory, using a multitude of processes. (When accessed, a symbolic link file, also called a symlink, redirects the requester to another file.) Mallory's program kept creating and removing many different symlinks pointing to the same special file: /etc/passwd, the password file.

A security precaution on this server was that every day, it ran the security program Tripwire -- specifically, the older version, 2.3.0. As Tripwire started up, it tried to create a temporary file, as many programs do. Tripwire looked and found that there was no file named /tmp/twtempa19212, so that appeared to be a good name for a temporary file. But after Tripwire checked, Mallory's program then created a symlink with that very name. This was no accident. Mallory's program was designed to create the very file names most likely to be used by Tripwire. Tripwire then opened up the file and starting writing temporary information. But instead of creating a new empty file, Tripwire was now overwriting the password file. From then on, no one -- not even the administrators -- could log into the system, because the password file had been corrupted. It could have been worse. Mallory's attack could have overwritten any file, including critical data stored on the server.

Introduction to race conditions

The story is hypothetical; Mallory is a conventional name for an attacker. But the attack, and the vulnerability it exploits, are all too common. The problem is that many programs are vulnerable to a security problem called a *race condition*.

A race condition occurs when a program doesn't work as it's supposed to because of an unexpected ordering of events that produces contention over the same resource. Notice that a race condition doesn't need to involve contention between two parts of the same program. Many security problems occur if an outside attacker can interfere with a program in unexpected ways. For example, Tripwire V2.3.0 determined that a certain file didn't exist, and then tried to create it, not taking into account that the file could have been created by an attacker between those two steps. Decades ago, race conditions were less of a problem. Then, a computer system would often run one simple sequential program at a time, and nothing could interrupt it or vie with it. But today's computers typically have a large number of processes and threads running at the same time, and often have multiple processors executing different programs simultaneously. This is more flexible, but there's a danger: If those processes and threads share any resources, they may be able to interfere with each other. In fact, a vulnerability to race conditions is one of the more common vulnerabilities in software, and on UNIX-like systems, the directories /tmp and /var/tmp are often misused in a way that opens up race conditions.

But first, we need to know some terminology. All UNIX-like systems support user processes; each process has its own separate memory area, normally untouchable by other processes. The underlying kernel tries to make it appear that the processes run simultaneously. On a multiprocessor system, they really can run simultaneously. A process notionally has one or more threads, which share memory. Threads can run simultaneously, as well. Since threads can share memory, there are usually many more opportunities for race conditions between threads than between processes. Multithreaded programs are much harder to debug for that very reason. The Linux kernel has an elegant basic design: There are only threads, and some threads happen to share memory with other threads (thus implementing traditional threads), while others don't (thus implementing separate processes).

To understand race conditions, let's first look at a trivial C statement:

Listing 1. Trivial C statement

```
b = b + 1;
```

Looks pretty simple, right? But let's pretend that there are two threads running this line of code, where `b` is a variable shared by the two threads, and `b` started with the value 5. Here's a possible order of execution:

Listing 2. Possible order of execution with shared "b"

```
(thread1) load b into some register in thread 1.
      (thread2) load b into some register in thread 2.
(thread1) add 1 to thread 1's register, computing 6.
      (thread2) add 1 to thread 2's register, computing 6.
(thread1) store the register value (6) to b.
      (thread2) store the register value (6) to b.
```

We started with 5, then two threads each added one, but the final result is 6 -- not the expected 7. The problem is that the two threads interfered with each other, causing a wrong final answer.

In general, threads do not execute *atomically*, performing single operations all at once. Another thread may interrupt it between essentially any two instructions and manipulate some shared resource. If a secure program's thread is not prepared for these interruptions, another thread may be able to interfere with the secure program's thread. Any pair of operations in a secure program must still work correctly if arbitrary amounts of another thread's code is executed between them. The key is to determine, when your program is accessing any resource, if some other thread could interfere with it.

Back to top

Solving race conditions

The typical solution to a race condition is to ensure that your program has exclusive rights to something while it's manipulating it, such as a file, device, object, or variable. The process of gaining an exclusive right to something is called *locking*. Locks aren't easy to handle correctly. One common problem is a *deadlock* (a "deadly embrace"), in which programs get stuck waiting for each other to release a locked resource. Most deadlocks can be prevented by requiring all threads to obtain locks in the same order (for example, alphabetically, or "largest grain" to "smallest grain"). Another common problem is a *livelock*, where programs at least manage to gain and release a lock, but in such a way that they can't ever progress. And if a lock can get stuck, it can be very difficult to make it gracefully release. In short, it's often difficult to write a program that correctly, in all circumstances, locks and releases locks as needed.

A common mistake to avoid is creating lock files in a way that doesn't always lock things. You should learn to create them correctly or switch to a different locking mechanism. You'll also need to handle races in the file system correctly, including the ever-dangerous shared directories /tmp and /var/tmp, and how signals can be used safely. The sections below describe how to deal with them securely.

Back to top

Lock files

UNIX-like systems have traditionally implemented a lock shared between different processes by creating a file that indicates a lock. Using a separate file to indicate a lock is an example of an *advisory* lock, not a *mandatory* lock. In other words, the operating system doesn't enforce the resource sharing through the lock, so all processes that need the resource must cooperate to use the lock. This may appear primitive, but not all simple ideas are bad ones. Creating separate files makes it easy to see the state of the system, including what's locked. There are some standard tricks to simplify clean-up if you use this approach -- in particular, to remove stuck locks. For example, a parent process can set a lock, call a child to do the work (make sure only the parent can call the child in a way that it can work), and when the child returns, the parent releases the lock. Or, a cron job can look at the locks, which contain a process id. If the process isn't alive, it would erase the lock and restart the process. Finally, the lock file can

be erased as part of system start-up, so that if the system suddenly crashes, you won't have a stuck lock later.

If you're creating separate files to indicate a lock, there's a common mistake: calling `creat()` or its `open()` equivalent (the mode O_WRONLY | O_CREAT | O_TRUNC). The problem is that root can *always* create files this way, even if the lock file already exists -- which means that the lock won't work correctly for root. The simple solution is to use `open()` with the flags O_WRONLY | O_CREAT | O_EXCL (and permissions set to 0, so that other processes with the same owner won't get the lock). Note the use of O_EXCL, which is the official way to create "exclusive" files. This even works for root on a local file system. The simple solution won't work for NFS V1 or 2. If your lock files must work on remote systems connected using these old NFS versions, the solution is given in the Linux documentation: "create a unique file on the same filesystem (e.g., incorporating hostname and pid), use `link(2)` to make a link to the lockfile and use `stat(2)` on the unique file to check if its link count has increased to 2. Do not use the return value of the `link(2)` call."

If you use files to represent locks, make sure those lock files are placed where an attacker can't manipulate them (for example, can't remove them or add files that interfere with them). The typical solution is to use a directory for which the permissions don't allow unprivileged programs to add or remove files at all. Make sure that only programs you can trust can add and remove these lock files.

The Filesystem Hierarchy Standard (FHS) is widely used by Linux systems, and includes standard conventions for such locking files. If you just want to be sure that your server doesn't execute more than once on a given machine, you should usually create a process identifier with the name /var/run/NAME.pid, with the process id as the file content. In a similar vein, you should place lock files for things like device lock files in /var/lock.

Back to top

Alternatives to lock files

Using a separate file to indicate a lock is an old way of doing things. Another approach is to use POSIX record locks, implemented through `fcntl(2)` as a discretionary lock. POSIX record locking is supported on nearly all UNIX-like platforms (it's mandated by POSIX.1), it can lock portions of a file (not just a whole file), and it can handle the difference between read locks and write locks. Also, if a process dies, its POSIX record locks are automatically removed.

Using separate files or an `fcntl(2)` discretionary lock only works if all programs cooperate. If you don't like that idea, you can use System V-style mandatory locks instead. Mandatory locks let you lock a file (or portion) so that every every `read(2)` and `write(2)` is checked for a lock, and anyone who doesn't hold the lock is held until the lock is released. That may be a little more convenient, but there are disadvantages. Processes with root privileges can be held up by a mandatory lock, too, and this often makes it easy to create a denial-of-service attack. In fact, the denial-of-service problem is so severe that mandatory locks are often avoided. Mandatory locks are widely available, but not universally. Linux and System V-based systems support them, but some other UNIX-like systems don't. On Linux, you have to specifically mount the file system to permit mandatory file locks, so many configurations may not support them by default.

Inside a process, threads may need to lock as well. There are many books that discuss those in great detail. The main issue here is to make sure that you carefully cover all cases; it's easy to forget a special case, or not handle things correctly. Basically, locks are hard to get correct, and attackers may be able to exploit errors in their handling. If you need many locks for threads inside a process, consider using a language and language constructs that automatically do a lot of lock maintenance. Many languages, such as Java and Ada95, have built-in language constructs that will automatically handle this -- and make the results more likely to be correct.

Where possible, it's often best to develop your program so you don't need a lock at all. A single server process that accepts client requests one at a time, processes that request to completion, and then gets the next request, in some sense automatically locks all the objects inside it. Such a simple design is immune to many nasty locking problems. If you need a lock, keeping it simple, such as using a single lock for nearly everything, has its advantages. This isn't always practical, since such designs can sometimes hurt performance. In particular, single-server systems need to make sure that no one operation takes too long. But it's worth considering. Systems using many locks are more likely to have defects, and there's a performance hit to maintaining many locks, too.

Back to top

Handling the file system

A secure program must be written to make sure that an attacker can't manipulate a shared resource in a way that causes trouble, and sometimes this isn't as easy as it seems. One of the most common shared resources is the file system. All programs share the file system, so it sometimes requires special effort to make sure an attacker can't manipulate the file system in a way that causes problems.

Many programs intended to be secure have had a vulnerability called a *time of check - time of use* (TOCTOU) race condition. This just means that the program checked if a situation was OK, then later used that information, but an attacker can change the situation between those two steps. This is particularly a problem with the file system. Attackers can often create an ordinary file or a symbolic link between the steps. For example, if a privileged program checks if there's no file of a given name, and then opens for writing that file, an attacker could create a symbolic link file of that name between those two steps (to /etc/passwd or some other sensitive file, for instance).

These problems can be avoided by obeying a few simple rules:

- Don't use `access(2)` to determine if you can do something. Often, attackers can change the situation after the `access(2)` call, so any data you get from calling `access(2)` may no longer be true. Instead, set your program's privileges to be equal to the privileges you intend (for example, set its effective id or file system id, effective gid, and use setgroups to clear out any unneeded groups), then make the `open(2)` call directly to open or create the file you want. On a UNIX-like system, the `open(2)` call is atomic (other than for old NFS systems versions 1 and 2).
- When creating a new file, open it using the modes `O_CREAT | O_EXCL` (the `O_EXCL` makes sure the call only succeeds if a new file is created). Grant only very narrow permissions at first -- at least forbidding arbitrary users from modifying it. Generally,

this means you need to use umask and/or open's parameters to limit initial access to just the user and maybe the user group. Don't try to reduce permissions after you create the file because of a related race condition. On most UNIX-like systems, permissions are only checked on open, so an attacker could open the file while the permission bits said it was OK, and keep the file open with those permissions indefinitely. You can later change the rights to be more expansive if you desire. You'll also need to prepare for having the open fail. If you absolutely must be able to open a new file, you'll need to create a loop that (1) creates a "random" file name, (2) open the file with O_CREAT | O_EXCL, and (3) stop repeating when the open succeeds.

- When performing operations on a file's meta-information, such as changing its owner, stat-ing the file, or changing its permission bits, first open the file and then use the operations on open files. Where you can, avoid the operations that take file names, and use the operations that take file descriptors instead. This means use the `fchown( )`, `fstat( )`, or `fchmod( )` system calls, instead of the functions taking file names, such as `chown()`, `chgrp()`, and `chmod()`. Doing so will prevent the file from being replaced while your program is running (a possible race condition). For example, if you close a file and then use `chmod()` to change its permissions, an attacker may be able to move or remove the file between those two steps and create a symbolic link to another file (say /etc/passwd).
- If your program walks the file system, recursively iterating through subdirectories, be careful if an attacker could ever manipulate the directory structure you're walking. A common example of this is an administrator, system program, or privileged server running your program while walking through parts of the file system controlled by ordinary users. The GNU file utilities (fileutils) can do recursive directory deletions and directory moves, but before V4.1, it simply followed the ".." special entry as it walked the directory structure. An attacker could move a low-level directory to a higher level while files were being deleted. Fileutils would then follow the ".." directory up much higher, possibly up to the root of the file system. By moving directories at the right time, an attacker could delete every file in the computer. You just can't trust ".." or "." if they're controlled by an attacker.

If you can, don't place files in directories that can be shared with untrusted users. Failing that, try to avoid using directories that are shared between users. Feel free to create directories that only a trusted special process can access.

Consider avoiding the traditional shared directories /tmp and /var/tmp. If you can just use a pipe to send data from one place to another, you'll simplify the program and eliminate a potential security problem. If you need to create a temporary file, consider storing temporary files somewhere else. This is particularly true if you're not writing a privileged program; if your program isn't privileged, it's safer to place temporary files inside that user's home directory, being careful to handle a root user who has "/" as their home directory. That way, even if you don't create the temporary file "correctly," an attacker usually won't be able to cause a problem -- because the attacker won't be able to manipulate the contents of the user's home directory.

But avoiding shared directories is not always possible, so we'll need to understand how to handle shared directories like /tmp. This is sufficiently complicated that it deserves a section of its own.

Back to top

Shared directories (like /tmp)

Basics of shared directories

Be extremely careful if your trusted program will share a directory with potentially untrusted users. The most common shared directories on UNIX-like systems are /tmp and /var/tmp, and many security vulnerabilities stem from misuse of these directories. The /tmp directory was originally created to be a convenient place to create temporary files, and normally, temporary files shouldn't be shared with anyone else. But it quickly found a second use: a standard place to create shared objects between users. Because these shared directories have multiple uses, it's hard for the operating system to enforce access control rules to prevent attacks. Instead, you actually have to use them correctly to avoid attack.

When you're using a shared directory, make sure the directory and file permissions are appropriate. You obviously need to limit who can read or write a file you create in a shared directory. But if multiple users can add files to a directory in a UNIX-like system, and you plan to add files to that directory from a privileged program, make sure that the *sticky* bit is set on that directory. In an ordinary directory (one without the sticky bit), anyone with write privileges to a directory -- including an attacker -- can delete or rename files, and cause all sorts of problems. For example, a trusted program could create a file in such a directory, and an untrusted user could delete and rename. On UNIX-like systems, shared directories need to have the sticky bit set. In sticky directories, files can be unlinked or renamed only by root or the file owner. The directories /tmp and /var/tmp are normally implemented as sticky, so that eliminates a few problems.

Programs sometimes leave behind junk temporary files, so most UNIX-like systems automatically delete old temporary files in the special directories /tmp and /var/tmp (the program "tmpwatch" does this), and some programs automatically remove files from special temporary file directories. That sounds convenient -- except that attackers may be able to keep a system so busy that *active* files become old. The result: The system may automatically remove a file name in active use. What happens then? An attacker may try to create his own file of the same name, or at least get the system to create another process and reuse the same file name. The result: chaos. This is called the *tmpwatch* problem. To resolve this, once you create a temporary file atomically, you must always use the file descriptor or file stream you got when you opened the file. Never reopen the file, or use any operations that use the file name as a parameter. Always use the file descriptor or associated stream -- or the tmpwatch race issues will cause problems. You can't create the file, close it, and reopen it -- even if the permissions limit who can open it.

Sticky directories and limited permissions on the files you create are only the first step. Attackers may try to slip in actions before or between actions of the secure program. A common attack is to create and uncreate symbolic links in the shared directory to some other file while your program is running -- files like /etc/passwd or /dev/zero are common destinations. The attacker's goal is to create a situation in which the secure program determines that a given file name doesn't exist. The attacker then creates the symbolic link to another file, and then the secure program performs some operation, but now it actually opened an unintended file. Often, important files can be clobbered or modified this way. Another variation is creating and uncreating normal files the attacker is allowed to write, so that the privileged program creates an "internal" file that can sometimes be controlled by the attacker.

The general problem when creating files in these shared directories is that you must guarantee that the file name you plan to use doesn't already exist at the time of creation, and atomically create the file. Checking before you create the file doesn't work because after the check occurs, but before creation, another process can create that file with that file name. Using an unpredictable or unique file name doesn't work by itself because an attacker can repeatedly guess until success. So, you'll need to perform an operation that creates a new file or fails -- and does nothing else. UNIX-like systems can do this, but you need to know to ask for it.

Solutions for shared directories

There are many nonsolutions, unfortunately. Some programs just call `mktemp(3)` or `tmpnam(3)` directly to create a temporary file name, and then simply open it under the assumption that it'll be OK. Bad plan. In fact, `tmpnam(3)` isn't reliable with threads and doesn't handle loops reliably, so it should never be used. The 1997 "Single UNIX Specification" recommends using `tmpfile(3)`, but its implementation on some old systems is, unfortunately, unsafe.

In C, to safely create a temporary file in a shared (sticky) directory, the usual solution is to set your `umask()` value to a restrictive value, and then repetitively: (1) create a random file name, (2) open(2) it using `O_CREAT | O_EXCL` (which atomically creates the file and fails if it's not created), and (3) stop repeating when the open succeeds.

C programmers don't need to actually do this directly; simply call the library function `mkstemp(3)`, which does this. Some implementations of `mkstemp(3)` don't set the `umask(2)` to a limited value, so it's wise to call `umask(2)` first to force the file to a restrictive value. A minor nuisance is that `mkstemp(3)` doesn't directly support the environment variables TMP or TMPDIR, so you have to do more work if that's important to you.

The GNOME programming guidelines recommend the following C code when creating file system objects in shared (temporary) directories to securely open temporary files:

Listing 3. Recommended C code for temporary files

```
char *;
int fd;
do {
  filename = tempnam (NULL, "foo");
  fd = open (filename, O_CREAT | O_EXCL | O_TRUNC | O_RDWR, 0600);
  free (filename);
} while (fd == -1);
```

Note that, although the insecure function `tempnam(3)` is being used, it is wrapped inside a loop using `O_CREAT and O_EXCL` to counteract its security weaknesses, so this use is OK. A nifty side-effect is that `tempnam(3)` usually uses TMPDIR, so it lets you redirect temporary files if needed. Note that you need to `free()` the file name. You should `close()` and `unlink()` the file after you are done. One minor disadvantage to this approach is that, since tempnam can be used insecurely, various compilers and security scanners may give you spurious warnings about its use. This isn't a problem with `mkstemp(3)`.

All this opening shows an oddity of the standard C IO library: There's no standard way to use `fopen()` with `O_EXCL`, so you can't open files the normal C way and safely create a temporary file. If you want to use the Standard C IO library, use `open()`, and you can then use `fdopen()` with mode "w+b" to transform the file descriptor into a FILE *.

Perl programmers should use File::Temp, which tries to provide a cross-platform means of securely creating temporary files. However, first read the documentation carefully on how to use it properly; it includes interfaces to the unsafe functions as well. I suggest explicitly setting its safe_level to HIGH because this will invoke additional security checks. This is actually true for most programming libraries; most libraries include interfaces to the insecure and secure operations, so you need to look at its documentation and make sure you're choosing the secure version.

Note that using `O_EXCL` doesn't work on directories on older versions of NFS (version 1 or 2) because these old versions don't correctly support `O_EXCL`. If you use `O_EXCL` yourself, your program will be insecure if the shared directory is implemented using these old NFS versions. There's actually a complicated workaround for old versions of NFS involving the use of `link(2)` and `stat(2)`. You can read about it in the Linux `open(2)` manual page among other places, if it's critical that your programs work in such circumstances. I'm not discussing it here because even if *your* program will work with old NFS versions, many other programs you use will not include this workaround. You're unlikely to have a secure system running NFS V1 or 2 for temporary directories anyway because of the other programs not using the workaround, so it's much wiser to simply require the use of NFS V3 or higher if you use remotely mounted temporary directories.

You could try to use `mkdtemp(3)`, but this is generally a bad idea because temp cleaners may decide to erase them.

If you're writing a shell script, use pipes or the user's home directory for temporary files. Don't use the /tmp or /var/tmp directories at all; ordinary shells usually don't support file descriptors, so tmpfile cleaners will eventually cause them to fail. If there's no tmpfile cleaner, and you simply must create temporary files in /tmp, at least use `mktemp(1)` to counter the more obvious attacks because `mktemp(1)` (not `mktemp(3)`) will use the `O_EXCL` mode to counter typical race condition attacks. The worst thing you could do is something that's disturbingly common: pretending that "$$" can't be guessed by an attacker, and simply redirecting information to such files. This won't use the `O_EXCL` mode of creation as required. An attacker could simply pre-create likely files, or repeatedly create and uncreate them, and eventually take over the program. Thus, shell scripts like this are almost certainly a serious vulnerability:

Listing 4. Vulnerable shell script

```
echo "This is a test" > /tmp/test$$  # DON'T DO THIS.
```

Don't reuse a temporary file name (remove and recreate it), no matter how you obtained the "secure" temporary file name in the first place. An attacker can observe the original file name and hijack it before you recreate it the second time. And of course, always use appropriate file permissions.

Clean up after yourself -- either by using an exit handler, or making use of UNIX file system semantics and `unlink()`ing the file immediately after creation so the directory entry goes away, but the file itself remains accessible until the last file descriptor pointing to it is closed. You can then continue to access it within your program by passing around the file descriptor. Unlinking the file has a lot of advantages for code maintenance. The file is automatically deleted, no matter how your program crashes. It also decreases the likelihood that a maintainer will insecurely use the file name (they need to use the file descriptor instead). The one minor problem with immediate unlinking is that it makes it slightly harder for administrators to see how disk space is being used.

There have been some successes in building countermeasures into operating systems, though they're not widely deployed at this time. See the resource list for links to Solar Designer's Linux kernel patch from the OpenWall project, RaceGuard, and Eugene Tsyrklevich and Bennet Yee's work for more information.

Back to top

Signal handling

Race conditions can also occur in signal handling. Programs can register to handle various kinds of signals, but signals can happen at the most inopportune times, including while you're already handling another signal. The only thing you should normally do inside a signal handler is set a global flag that will be processed later. There are a few more operations that can be done securely in signals, but not many, and you must have a deep understanding of signal handling before doing so. That's because only a few system calls can be safely called inside signals. Only calls that are re-entrant or not interruptible by signals can be called safely. You can call library functions, but only a very few can be safely called. Calling most functions inside a signal handler, such as `free()` or `syslog()`, is a major sign of trouble. For more information, see the paper by Michal Zalewski called "Delivering Signals for Fun and Profit." But you're better off setting flags -- and nothing else -- in a signal handler, as opposed to trying to create sophisticated handlers.

Back to top

Conclusions

This article has discussed what a race condition is, and why it can cause security problems. We've examined how to create lock files correctly and alternatives. We've looked at how to handle the file system and, in particular, how to handle shared directories to accomplish common tasks, such as creating temporary files in the /tmp directory. We've even briefly looked at signal handling, at least enough to know a safe way to use them.

Clearly, it's important for your program to protect against race conditions. But most of today's programs can't do everything by themselves. They must make requests to other libraries and programs, such as command interpreters and SQL servers. And one of the most common ways to attack programs is to exploit how they make requests to other programs. Next we'll examine how to call out to other programs without exposing a vulnerability.

Resources

- Read David's other *Secure programmer* columns on developerWorks.
- David's book *Secure Programming for Linux and Unix HOWTO* (Wheeler, 3 Mar 2003) gives a detailed account on how to develop secure software.
- Jarno Huuskonen's Bugtraq e-mail "Tripwire temporary files" on 9 Jul 2001 gives information about the Tripwire V1.3.1, 2.2.1 and 2.3.0 symlink race condition problem. This is CVE-2001-0774.
- CVE-2002-0435 notes the race condition in the GNU file utilities.
- The Perl 5.8 documentation of File::Temp is available online.
- Kris Kennaway's posting to Bugtraq about temporary files (15 Dec 2000) discusses more about temporary files.
- Michal Zalewski's Delivering Signals for Fun and Profit: Understanding, Exploiting and Preventing Signal-Handling Related Vulnerabilities (16-17 May 2001) describes signal handling, yet another way to introduce race conditions.
- Michal Zalewski's Problems with mkstemp() discusses the security problems arising from automatic cleaning of temporary directories.
- The security section of the GNOME Programming Guidelines provides a reasonable suggestion for how to create temporary files.
- The Single UNIX Specification, Version 3, 2004 edition (also known as IEEE Std 1003.1, 2004 Edition), is a common specification of what a "UNIX-like" system must do.
- Solar Designer's Linux kernel patch from the OpenWall project includes several interesting security countermeasures, including additional access limits that prevent a limited set of filesystem race condition attacks. More specifically, it limits users from following untrusted symbolic links created in certain directories and limits users from creating hard links to files they don't have read and write access to.
- Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman's paper "RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities" from the USENIX Association's 2001 Usenix Security Symposium describes RaceGuard, a Linux kernel modification to counter some race condition attacks by detecting certain attacks at run time.
- Eugene Tsyrklevich and Bennet Yee's Dynamic Detection and Prevention of Race Conditions in File Accesses from the 12th USENIX Security Symposium (August 2003) describes an approach to countering race conditions in the file system. They modify the OpenBSD kernel so that 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.
- Find more resources for Linux developers in the developerWorks Linux zone.

- Download no-charge trial versions of IBM middleware products that run on Linux, including WebSphere® Studio Application Developer, WebSphere Application Server, DB2® Universal Database, Tivoli® Access Manager, and Tivoli Directory Server, and explore how-to articles and tech support, in the Speed-start your Linux app section of developerWorks.

- Get involved in the developerWorks community by participating in developerWorks blogs.

- Browse for books on these and other technical topics.

About the author

David A. Wheeler is an expert in computer security and has long worked in improving development techniques for large and high-risk software systems. He is the author of the book *Secure Programming for Linux and Unix HOWTO* and is a validator for the Common Criteria. He also wrote the article "Why Open SourceSoftware/Free Software? Look at the Numbers!" and the Springer-Verlag book *Ada95: The Lovelace Tutorial*, and is the co-author and lead editor of the IEEE book *Software Inspection: An Industry Best Practice*. This article presents the opinions of the author and does not necessarily represent the position of the Institute for Defense Analyses. Contact David at dwheelerNOSPAM@dwheeler.com (after removing "NOSPAM").