Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare *Redux*

Matt Bishop

CSE-95-8

September 1995

Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare *Redux*

Matt Bishop
Department of Computer Science
University of California at Davis
Davis, CA 95616-8562
phone: (916) 752-8060
email: bishop@cs.ucdavis.edu

Abstract

A pernicious type of security problem, race conditions create a timing interval in which the manipulation of resources allows an attacker to gain privileges, read and alter protected files, and violate the security policy of the site. The majority of these conditions found on applications and system programs in the UNIX operating system arise during file system manipulation and file accesses. This paper discusses why race conditions arise, presents some examples, and explores ways to alleviate the problem of their occurrence in programs, both through modification of system calls and careful programming. A tool to scan for potential race conditions, as well as a library function to analyze the environment dynamically at run time (and thus detect such conditions) are presented.

Introduction

Recently, many privileged UNIX programs have been shown to be seriously flawed, allowing users to modify privileged files or to create programs with *root* privileges. The specific flaw is a *race condition*, in which two actions that should be performed atomically are not. This flaw often arises in privileged programs.

For example, suppose a program that is setuid to *root* wishes to save data in a file owned by the user executing the program. The following is a typical fragment of code designed to do this:

```
if (access(filename, W_OK) == 0){
     if ((fd = open(filename, O_WRONLY)) == NULL){
          perror(filename);
          return(0);
     }
     /* now write to the file */
```

The program cannot simply open the target file (the name of which is in the variable *filename*), because as *root* that open will always succeed, even if the real user (the user running the program) would not be able to write to the file. Instead, the program must first check if the real user could write to the file, and *only if so* proceed to write to it. The idea is that no user should be able to use a privileged program to alter a file unless that user could have altered the file without the added privileges.

So, the first step is to use the system call *access*(2); this returns 0 if the real user has the desired access to the named file (in this case, write access to the file named in *filename*), and -1 if

not. Then, if the user does have that access, the *root* program opens the file for writing. If the user does not, the file is not opened.

The problem with this procedure lies in the implementation: the check, *access*, and the open, *open*(2), are not indivisible with respect to one another. As a result, if the object referred to by *filename* changes between the two function calls, the second object will be opened even though the access check involved only the first object. This is the race condition, and is the basis for numerous programming flaws in all systems which provide multiple levels of privilege.

In this paper we examine the causes of these flaws in privileged programs written for the UNIX operating system, and suggest several enhancements to the operating system to enable programmers to avoid these conditions. The next section provides two well-known and recent examples of race conditions, to demonstrate that there is a problem; a detailed discussion of the causes of race conditions in UNIX applications follows, and this leads to the proposed changes in the fourth section. We then discuss a very primitive race condition analyzer, and avoidance techniques in programming. We conclude with suggestions for future work, in the last section.

Some Race Conditions

The presentation of security flaws in extant programs is a touchy issue, because for those sites which have not yet patched the holes, such information presents a security risk. For this reason, in this section we will present two security holes the existence of which was trumpeted widely and for which fixes are available (see the Endnote). In this way we do not increase the dangers for those sites which have not yet patched the holes, because the holes are widely known.

The Password Program Race Condition [7]

On SunOS 4.1 (and other) systems, the program *passwd*(1) allows the user to specify the password file to be used. This enables a race condition allowing any user (*attacker*) to gain access to any other user's (*target's*) accounts. The attack depends upon the user specifying the file *passwd* is to use as the password file, which *passwd* updates as root.

Under normal conditions, the *passwd* program accesses files like this:

1. open and read the password file to get the entry for the user; then close password file
2. create and open a temporary file called "ptmp" in the directory of the password file
3. open the password file again, and copy the contents to "ptmp", updating the changed information
4. close the password file and "ptmp" and rename "ptmp" to be the password file

The attack hinges on the race condition arising when the password file is in a directory the user can write to. Basically, one creates a bogus password file named ".rhosts" with the following as the first entry and everything else a copy of the regular password file:

```
localhost attacker :::::
```

One then puts this in a directory *pwd_dir*, and builds a symbolic link *link* which resolves to *pwd_dir*. Then invoke *passwd* giving "*link*/.rhosts" as the name of the password file. Here's what happens; the steps A, B, *etc*. are done by the attacker at the points indicated:

1. The process opens and reads "*link*/.rhosts" to get the entry for the user; then it closes that password file

A. The attacker changes the symbolic link *link* to point to the *target*'s home directory *target_dir.*

2. The process creates and opens a temporary file called "ptmp" in the directory of the password file, which in this case is *link*, or (now) *target_dir.*

B. The attacker switches *link* back to *pwd_dir.*

3. The process opens "*link*/.rhosts" again (which is the password file named in the command line), and copies the contents to "ptmp", updating the changed information. Note that "ptmp" is still in *target_dir* as it was opened in step 2.

C. The attacker switches *link* back to *target_dir.*

4. The process closes "*link*/.rhosts" (which involves no interaction with the file name *link* as only file descriptors are involved) and "ptmp" and renames "ptmp" to be "*link*/.rhosts"; as *link* is now *target_dir*, this makes the password file into the victim's ".rhosts" file. Given the first line of that file, the attacker can now *rlogin*(1) to the victim's account.

The Binmail Race Condition [8]

This attack, again usable on many systems, has as its goal being able to write to any file on the system. It is much more straightforward than the *passwd* race condition. The program *binmail*[1] is the program that delivers mail by writing it into the recipient's mailbox. Here are the steps used to do so:

1. Use the system call *lstat*2) to get information (file type, protection mode,. *etc*.) about the mailbox. (If the mailbox does not exist, this is not an error; it will be created at step 2.) Note the use of *lstat*, to detect symbolic links; *binmail* will not deliver mail to a symbolic link.

2. Assuming the file is a mailbox (*i.e.*, a regular file and not a symbolic link), append the letter to the mailbox, as *root*.

The race condition lies between these steps. If the attacker can either delete the mailbox (if it already exists) or simply create a symbolic link to the file to be written to, then at step 2, the process will append the letter to the target file. Note that the appending is done as root, so the file can be created if it does not exist, and will be altered if it does exist.

Detailed Discussion

These race conditions arise because of the ways in which files (or more generally, objects in the file tree) may be referred to. The first is by *name*; in this mode, the object is determined by walking the path name, and the binding occurs at the last component of the name; in other words, the binding is both late and transient (as it is done anew at each reference). The second is by *descriptor*; in this mode, the object is determined when the descriptor is assigned and from then until deallocation, the descriptor refers to that object. In other words, the binding is both early and permanent. (See [2], [6] for a detailed discussion of how the operating system handles names and descriptors.)

In both the *passwd* and *binmail* race conditions, all references to the files involved are made through names. By altering the meaning of the (fixed) names between references, the attacker

1. Actually, /bin/mail; see *mail*(1)

alters the objects to which those names refer. This is the cause of the problem, and the term "race" refers to altering the meaning before the names are bound to objects. The same is true for the example in the introduction; even though the second system call is *open* (which binds a descriptor to an object), the initial binding requires the object be located by its name; so the first call, *access*, uses the name and the second call also uses the name. Again, there is a window of vulnerability in which the meaning of the name can be changed.

Once bound, descriptors do not suffer this problem. If descriptor *d* refers to a particular file named "/tmp/foobar", and that file is deleted, it will be removed from the file system hierarchy, but will not be expunged until *d* is deallocated (closed). Should a new file be named "/tmp/foobar" after the deletion of the original, but before the descriptor is deallocated, any actions on *d* (such as writing or status requests) will refer to the original file, not the new one.

Solution #1: Do descriptor binding first

This leads to the idea that the object should be bound to the descriptor first, and then all references to the object will refer to the intended object. For example, the first two lines of the *xterm* code shown in the introduction would be replaced by:

```
if ((fd = open(filename, O_WRONLY)) == NULL){
     if (access(filename, W_OK) == 0){
```

This is appealing, because the first line appears to bind the object referred to by *filename* to the process. Hence the second line clearly refers to that object. But as with anything appealing in computer security, this won't work; if the meaning of the name can be switched between the two calls, the first will open the original object, and the second reference will check access permissions on the new object (which, presumably, allow the object to be written to).

The mistaken and seductive belief here is the object being bound to the process. It is really bound to the descriptor *fd*, so there is no assurance that an alternate identification of the object will be bound to the same object. Indeed, that is what happens here; the late binding of the name allows the meaning of the name to be altered.

Solution #2: Make versions of file manipulation system calls that use descriptors, not names

This leads to a second solution, one which (unfortunately) requires kernel modification. The idea is to eliminate as much dependence on the name of an object as is possible. For example, in the *xterm* hole, the two first lines would be replaced by:

```
if ((fd = open(filename, O_WRONLY)) == NULL){
     if (faccess(fileno(fp), W_OK) == 0){
```

Here, *faccess* is a system call which works like the *access* system call, except that it uses a descriptor rather than a name. Because it uses the descriptor associated with the object opened in the previous line, the access check refers to the object opened for writing, even if the meaning of the name in *filename* is changed between the open and the access.

This solution is quite appealing, because it allows elimination of a large class of race conditions; many system calls which use names have descriptor equivalents. Not all do, however, and the proposal is to augment those which do not have descriptor equivalents.

Some system calls are designed to work with the file hierarchy; these quite naturally use the representation of the object name induced by that hierarchy. For example, the *mkdir*(1) system

call uses a directory name; so, if a privileged program needed to create a directory owned by the user, it would do so with:

```
if (mkdir("/tmp/bishop") >= 0){
     if (chown("/tmp/bishop", 1324, 25) < 0){
          perror("/tmp/bishop");
          /* proceed with error handling */
     }
}
```

and a race condition still arises. This suggests one more simple modification.

Solution #2A: Use solution #2 and define a new mode of *open*, O_ACCESS:

The goal is to associate an object and a file descriptor without constraining the mode of access. So, we propose augmenting the semantics of *open*(2), which currently does such binding but also either creates the object or constrains access.

When the second argument (which says how to open the object) is **O_ACCESS**, the following occurs:

1. If the object does not exist, the name is reserved and an inode is allocated, both in core and on the disk. The type of the object is "reserved", which is a type distinct from all other object types (directory, file, socket, *etc*.) This file type is transient, and may be altered by subsequent system calls (for example, to *mkdir*(2), *mknod*(2), and *open*(2)). When one of those system calls is issued, the "reserved" attribute changes to the appropriate file type.
2. If the object exists, this is like an *open* except that neither read nor write permission is granted. A subsequent *open* can add these if the semantics are appropriate.

The intent of these modifications is to bind the object to a file descriptor (creating the object, if necessary) and then making all accesses to the object through the descriptor. While it is tempting to use the existing modes of the *open* system call for this, it is not possible to "add" access modes, which means one would need to open the object for reading, writing and appending; further, if the object did not exist, this would create a regular file, whereas the object being bound might need to be a directory or a queue.

Then the above fragment becomes:

```
if ((fd = open("/tmp/bishop", O_ACCESS)) >= 0){
     if (fmkdir(fd) < 0 || fchown(fd, 1324, 25) < 0){
          perror("/tmp/bishop");
          /* proceed with error handling */
     }
}
```

Now the race condition does not arise, because only one binding occurs (at the open); all other operations are on that object and none other.

As was noted above, this requires modifications to the kernel. Simply building a new interface, or a new library, will not do, because that merely pushes the system calls that create the possibility for a race condition down to a lower level of abstraction. It does not eliminate the problem; in fact, it arguably makes the problem worse, as now the conditions can occur in the interface to a system call, eliminating the trust that can be reposed in the atomicity of those calls.

Checking for Race Conditions

Given the relationship between object names and the potential existence of race conditions, it is possible to write a simple program that scans the source code of programs looking for these conditions. For example, any access call followed by an open with the same object name is a potential race condition.

We emphasize potential. Whether or not the race condition can occur depends on environment [5]. One fix for the *binmail* hole on many UNIX systems is to set the sticky bit on the mail spool directory, and then ensure each user always has a mailbox (empty if need be). This prevents the attacker from placing a symbolic link into the directory with the same name as anyone's mailbox after the *lstat* and before the *open*, due to the semantics of the sticky bit. But the *binmail* program still has the security hole, and should a mailbox ever disappear, the attacker can use that hole to gain access to another's account.

Clearly, a fully-developed tool would build a call graph and locate pairs of suspicious system calls (that is, calls which might lead to a race condition). It would then analyze their arguments and determine if the arguments representing the object names are (or could be) the same; the latter will of necessity be imprecise as solving it completely would be equivalent to solving the halting problem. A list of such pairs of calls would be printed, and the human analyst would analyze them in light of the environment in which the program would run.

As a prototype, a very simple scanner was built; it was lexically based rather than syntactically based. It only reported cases where the relevant arguments were identical. So for example, it would report this sequence as suspicious:

```
char fname[] = "/tmp/xyzzy";
if (access(fname, R_OK) >= 0 && (fd = open(fname, O_RDONLY)) >= 0)
    ...
```

but not

```
char fname[] = "/tmp/xyzzy";
char *p = fname;
if (access(fname, R_OK) >= 0 && (fd = open(p, O_RDONLY)) >= 0)
    ...
```

because the arguments are lexically different. Also, the environment was ignored, on the theory that the developer would use to tool to spot potential problems, eliminate them when he or she could, and warn the installer (user) about the rest.

This prototype analyzer proved wildly successful, uncovering a serious problem in *sendmail* 8.6.10; it has been corrected in version 8.6.12. Given its success, we feel that development of the production-quality scanner described earlier would be quite worthwhile, and are working on a more complex (although probably not production quality!) scanner.

Prevention is a better strategy than detection; but how realistic is such a strategy?

Current Prevention: Programming and Environment

As we pointed out earlier, race conditions depend on the use of at least one file name. They also require the ability of the attacker to alter the referent of the name; in the two examples, had the attacker not been able to alter the referent, the action taken using the file name would have

affected the intended object and there would have been no window of vulnerability. This suggests two points of analysis, both focussing on environment as the critical factor.

The first is to try to eliminate the use of the file name entirely, except when binding the file descriptor to the object. An earlier section discussed how this could be done; but currently, UNIX systems do not provide this flexibility.

The second is to try to control the environment sufficiently so the attacker cannot substitute a new object for the one referenced by name. Let *T* be a set of users whom the owner of the privileged program trusts, and let *U* be all other users. An object is *trustworthy* if the object bound to the name at the first reference of the name remains bound throughout the lifetime of the process. This means an object *O* is trustworthy if, and only if no member of *U* can replace *O* with a new object *O'*. So, in the *xterm* example, the log file name given to *xterm* would be the object *O*, and the password file would be the object *O'*. In the given scenario, as *O* is created in the user's home directory, the user can replace it with a symbolic link; so, if the user is not in *U*, the original log file is not trustworthy. Note that *root* must always be in *T*, or no file will be trustworthy.

When an object is referred to by its name, it must be checked to see if it is trustworthy. To do this, each of its ancestor directories must be unwriteable by any member of *U*; further, if any of those ancestors are symbolic links, the object to which the symbolic link refers, and all *its* ancestor directories, must also be unwriteable by any member of *U*. Finally, the object itself must not be a symbolic link, or if it is, all *the* ancestor directories of the object to which the symbolic link refers must also be unwriteable by any member of *U*.

A library function, *can_trust(file, T, U)*, which implements the above and returns 1 if the object *file* is trustworthy and 0 if not, is available; see the Endnote.

A third approach is to consider privilege as an element of environment and examine that further. The race condition, actually, is not the problem; the *combination* of race condition *and privilege* is. Specifically, if *passwd* did not run with *root* privileges, it would not have been able to write the ".rhosts" file into the target's directory. Similarly, if *binmail* were run with the privileges of the attacker rather than the privileges of *root*, it would be unable to open the target file for writing. So, when writing code that requires accessing files (or creating files) based on the real UID and/or GID of the process, make the access functions subprocesses which reset the effective UID and GID to the real UID and GID, and *in those subprocesses* create or access the files. This way, the race dcondition still exists, but because it occurs only when no extra privileges are involved, it is quite harmless. A sample library for writing to files is discussed and presented in [4]. Note this procedure eliminates the need for any *acces* system call.

Conclusion

Race conditions are not unique to the UNIX operating system; indeed, the Program Analysis study [3] and the RISOS study [1] both identified them as extant in a large number of operating systems and programs. The classic form of this condition has been named the time-of-check to time-of-use flaw (TOCTTOU flaw) and was first identified in 1974. This august lineage suggests that race condition flaws will continue to plague systems.

In this paper, we focussed on those race conditions arising due to file system accesses under the UNIX operating system. We informally examined why these conditions occur, and looked at various ways to fix the problem. Absent kernel modification, one cannot eliminate the problem;

but the problem of race conditions occurring can be greatly ameliorated, and we looked at both static (program analysis) and dynamic (run-time analysis) techniques to do this.

Endnote

The *passwd* and *binmail* flaws, and their fixes, were publicized by 8LGM. To obtain information about them, send a letter to 8lgm-fileserv@8lgm.org with the word "help" as the body of the message.

The scanner for race conditions is not yet freely available. If and when we release it, it will reside on *nob.cs.ucdavis.edu* in */pub/sec-tools*. The function *can_trust* is in that directory (as are several other goodies).

Acknowledgements. Thanks to Dorothy Denning and Lawrence Snyder, who started me on this path years ago; to Peter Neumann, who encouraged me and whose sparkling puns enliven any discussion (and occasionally clear rooms); to Kevin Ziese, Toney Jennings, Dan Teal, and Tim Grance for their enthusiasm and support, and to Karl Levitt and Bob Abbott for their words of wisdom and helpful discussions. Special thanks to Michael Dilger, who wrote the prototype race condition scanner in Perl. This work was supported by grant TDS 94-140 from Trident Data Systems, Inc. to the University of California at Davis.

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 (Apr. 1976).

[2] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood CLiffs, NJ (1986).

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

[4] M. Bishop, "How to Write a Setuid Program," Technical Report 85.6, Research Institute for Advanced Computer Science, Moffett Field, CA (May 1985).

[5] M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses," *submitted to the Third ACM Conference on Computer and Communication Security.*

[6] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Company, Reading, MA (1989).

[7] 8LGM, "[8lgm]-Advisory-7.UNIX.passwd.11-May-1994," available from *fileserv@bagpuss.demon.co.uk* (May 1994)

[8] 8LGM, "[8lgm]-Advisory-5.UNIX.mail.24-Jan-1992," available from *fileserv@bagpuss.demon.co.uk* (Jan 1992)