Checking for Race Conditions in File Accesses

Matt Bishop, *Michael Dilger*
Department of Computer Science
University of California at Davis
Davis, CA 95616-8562

Abstract

Flaws due to race conditions in which the binding of a name to an object changes between repeated references occur in many programs. We examine one type of this flaw in the UNIX operating system, and describe a semantic method for detecting possible instances of this problem. We present the results of one such analysis in which a previously undiscovered race condition flaw was found.

1. Introduction

Ordinary bugs and misconfigurations prevent applications or systems from functioning correctly. By contrast, security holes or vulnerabilities enable a user (called an *attacker*) to gain privileges, access to data, or the ability to interfere with others' work via by exploitation. Much research, especially in the field of intrusion detection [5][9], draws on characteristics of these attacks [12]. But many attacks can exploit a single vulnerability, implying that the characteristics of the flaws themselves are more fundemental and should be of interest.

This work focuses on a semantic characteristic of one class of the time-of-check-to-time-of-use (TOCTTOU) flaws. A TOCTTOU flaw occurs when a program checks for a particular characteristic of an object, and then takes some action that assumes the characteristic still holds when in fact it does not. This particular flaw has a distinguished lineage, being described by both the Program Analysis (PA) project [3] and the Research Into Secure Operating Systems (RISOS) project [1] as a subclass of the class of timing or synchronization flaws.

A subclass of TOCTTOU flaws, which we call *TOCTTOU binding flaws*, arise when object identifiers are fallaciously assumed to remain bound to an object. The results in this paper demonstrate the effectiveness of a semantic approach to detecting TOCTTOU binding flaws. The next section shows that this problem is severe enough to be worthy of examination. Section 3 describes two characteristics of TOCTTOU binding flaws, and section 4 presents a tool that detects some TOCTTOU binding flaws. Section 5 demonstrates the unsolvable nature of the general problem, and discusses the approximate relative power of different types of analyzers. The final section

offers some comments on checking for these flaws dynamically, and using logs.

2. Demonstration of the Problem

The analysis in this paper focuses on application-level programs rather than on the operating system. Many operating systems allow some trusted user complete control over the system. Although such a privileged user violates basic security design principles [15], it eases problems of administration. Access to these users requires either a password or use of a mechanism by which the privileged user delegates privilege to a set of utilities. The UNIX operating system [14] is one of the better-known, and most widely-used, systems to use this scheme.

As the delegation of rights creates potential security problems, analyzing these utilities to which rights have been delegated will provide insight into vulnerabilities on the systems where they appear. Few attacks exploit specific operating system kernel flaws; most exploit flaws in these utilities. A method of locating these flaws would enable these attacks to be detected or prevented. As privileged UNIX programs are available either commercially or on the World Wide Web, such a method would allow sites to verify software before installation.

The archetypal TOCTTOU binding flaw in a privileged program on the UNIX operating system arises when a setuid to *root* program is to save data in a file owned by the user executing the program. The program should not alter the file unless the user could alter the file without any special privileges. Code to do so typically looks like this:

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

If the program were to omit the *access*(2) system call, the *open*(2) system call would always succeed, because the effective UID of the process is *root*. If the user executing the program could not write to the file, the *access* system call would return -1 and the open would never be attempted. So this fragment allows the process to write to the file if, and only if, the user executing the program could do so.

If the object referred to by *filename* changes between the two system calls, though, the second object will be opened even though its access was never checked (access to the first object was

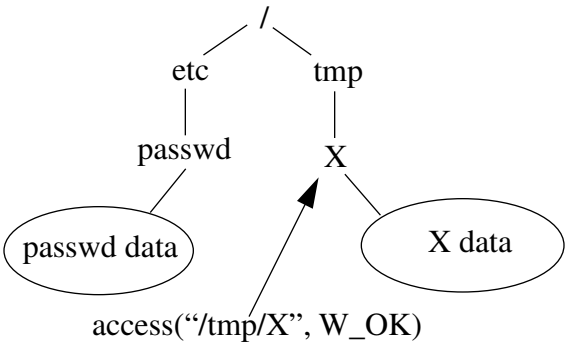

Figure 1a.

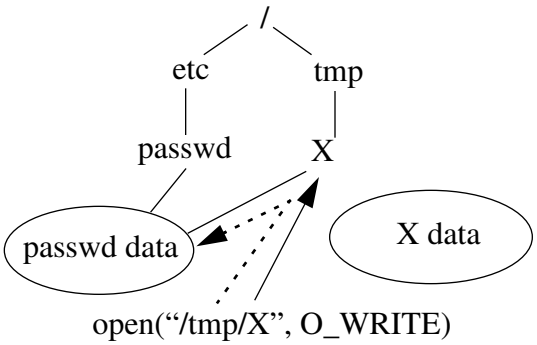

Figure 1b.

Figure 1. Example of the TOCTTOU binding flaw.

checked).

The scenario in Figure 1 is an example of the TOCTTOU binding flaw. Figure 1a shows the state of the system at the time of the *access* system call; the solid arrow indicates the *access* refers to "/tmp/X". Both "/tmp/X" and "/etc/passwd" name distinct objects. However, before the process makes its *open* system call, "/tmp/X" is deleted and a direct alias (hard link) for "/etc/passwd" is created, and is named "/tmp/X". Then the *open* accesses the data associated with "/etc/passwd" when it opens "/tmp/X", since "/tmp/X" and "/etc/passwd" now refer to the same file. Figure 1b shows this, with the dashed arrow indicating which data is actually read and the solid arrow indicating the name given to *open*. The unprivileged process can then write to the protected password file. Several versions of the terminal emulation program *xterm*(1) [16] suffer from this flaw, which arises when logging sessions to a file.

Another instance of this flaw occurs on SunOS and HP/UX systems. The program *passwd*(1) allows the user to name the password file as a parameter. An attacker can gain access to any other user's accounts using a variant of the attack presented above [6]. Under normal conditions, the *passwd* program takes the following steps:

1. opens and reads the password file to get the entry for the user; then closes the password file;
2. creates and opens a temporary file called "ptmp" in the directory of the password file;
3. opens the password file again, and copies the contents to "ptmp", updating the changed information; and
4. closes the password file and "ptmp" and renames "ptmp" to be the password file.

The attack works as follows. Suppose the attacker can write to directory "pwd." The attacker creates a bogus password file named "pwd/.rhosts" with the following as the first entry:

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

and the remainder of the file a copy of the real password file. The attacker specifies this file to be the password file when calling *passwd*. During steps 1 and 3, the attacker wants the directory containing the password file to be "pwd"; during steps 2 and 4, the attacker wants the directory containing the password file to be the *target*'s home directory (belonging to the user being attacked). The following steps create a ".rhosts" that allows the attacker to log into the target's account without authentication. As the *passwd* program is setuid to *root*, lack of privileges over *target*'s home directory is irrelevant.

All references to the password file's directory will be made through an indirect alias (symbolic link) called "link" to enable the referent of that directory name to be changed. The sequence of events, augmented by the attacker's actions (A, B, and C), follows:

1. The *passwd* process opens and reads "link/.rhosts" to get the entry for the user; then it closes that password file (see Figure 2a).

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

2. The process creates and opens a temporary file called "ptmp" in the directory of the password file, which in this case is "link", and also "target" (see Figure 2b).

B. The attacker switches "link" back to "pwd".

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" as it was opened in step 2 (see Figure 2c).

C. The attacker switches "link" back to "target".

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", this makes the password file into the victim's ".rhosts" file (see Figure 2d).

At this point the attacker can *rlogin*(1) to the victim's account. Figure 2 summarizes this attack.

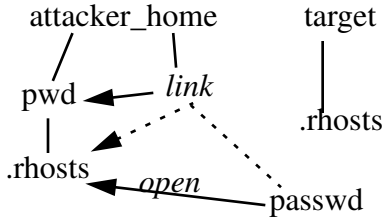

Figure 2a.

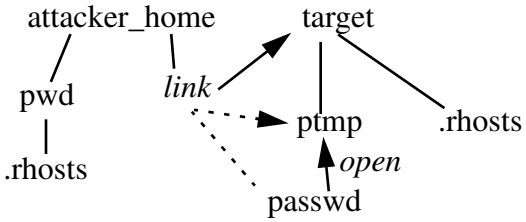

Figure 2b.

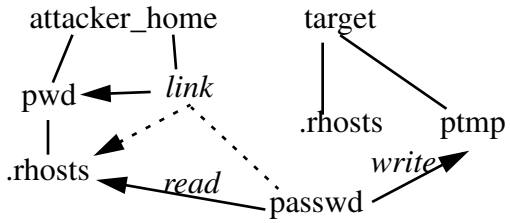

Figure 2c.

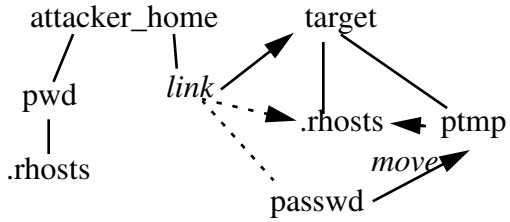

Figure 2d.

Figure 2. An example of the TOCTTOU binding flaw using *passwd*.

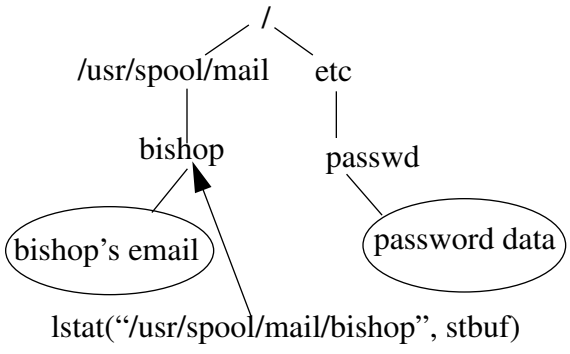

Figure 3a.

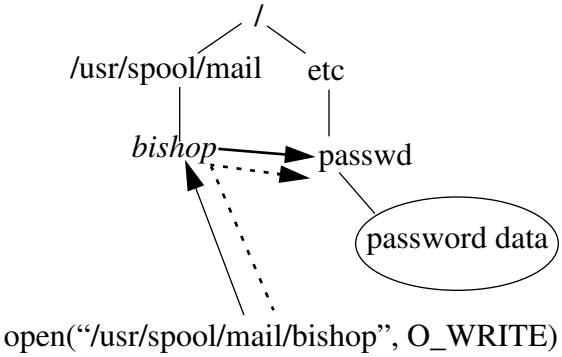

Figure 3b.

Figure 3. The *binmail* race condition attack.

Another attack, called the *binmail race condition* [7], lets the attacker write to any file on the system. The *binmail* program delivers mail by writing it into the recipient's mailbox. As a security check, *binmail* requires the mailbox to be a regular file, and not a symbolic link. But in this check lies a TOCTTOU binding flaw. The following two steps consitiute the check:

1. Use the *lstat(*2) system call to get information (file type, protection mode, *etc*.) about the mailbox.

2. If the mailbox is a regular file, append the letter to the mailbox, as *root*.

The TOCTTOU binding flaw lies between these steps. Figure 3a shows the state of the system at the time of the *lstat* system call. The mailbox file "/usr/spool/mail/bishop" is a regular file, so *binmail* continues. But before *binmail* can open the mailbox file, that file is deleted (by the

Figure 4. Graphical diagram of programming interval. *Syscall-0* begins the interval, and *syscall-1* ends it.

attacker) and a new file with the same name is created. This file is a link to the system password file. Figure 3b shows the state of the system after *binmail* opens the mailbox file; it is actually opening the link, and hence the password file. Now the letter will be appended to the password file; if it contains the appropriate contents, the attacker can now log in as the superuser without any password. Note that *binmail* appends as *root*, so the file can be created if it does not exist, and is altered if it does exist.

3. A Semantic Characterization of TOCTTOU Binding Flaws

A TOCTTOU flaw occurs when two events occur and the second depends upon the first. During the interval between the two events (see Figure 4), certain assumptions from the results of the first system call influence the second. If some action during that interval invalidates those assumptions, the results of the second action may not be what was intended. (Exploiting this situation requires an attacker to act during the interval. The more general term "race condition" captures the race between the attacker's trying to invalidate assumptions before the second action occurs.)

Call the existence of such an interval the *programming condition* and the interval itself the *programming interval*. Having found this condition holds, the attacker must be able to affect the assumptions created by the program's first action. That condition is the *environmental condition*. Both conditions must hold for there to be an exploitable TOCTTOU binding flaw.

In the initial example, the *access* system call creates the assumption that the user is authorized to alter the file "/tmp/X". The *open* acts upon that assumption. So the programming condition holds. If the attacker can alter the referent of the name "/tmp/X", then the environmental condition also holds and an exploitable TOCTTOU binding flaw exists (and given the semantics of "/tmp", Figure 1 shows it does indeed hold). Were the file in a directory that the attacker could not alter, the environmental condition would not hold and no exploitable TOCTTOU binding flaw would exist. The other two examples have similar conditions.

3.1. Programming Condition

The UNIX system provides two different forms of naming, with different semantics [2][13]. The first form is a file path name. The UNIX file system is conceptually a tree, with interior nodes being directories and leaf nodes being files, devices, or other entities. The path name specifies the path through the tree from the root to the target node. To access the object from a path name, the kernel begins at the beginning of the path name, and accesses each component named in the path. Each interior node contains the location (or address) of the next node in the path. The penultimate node in the path contains the location of the object; from this the object may be retrieved.[1] Conceptually, no caching of names to addresses is done; the name is mapped into the object each time.

The second form of file naming is the file descriptor. File descriptors are assigned to a file on a per-process basis, and bind directly to the object. When a process requests that a file descriptor be assigned to an object, it provides the file path name of the object. The system maps this address to an object, and returns a reference (the file descriptor) to the object. References using the file descriptor do not involve the system-wide object name (path name) but instead, the kernel uses a file descriptor local to the process to access the object directly.

Notice the difference in the way the addresses resolve to objects. File path names are resolved by indirection, requiring the naming and accessing of at least one object other than the file being addressed. File descriptors are resolved by accessing the file being addressed. The former correspond to (multiply) indirect pointers to the object, the latter to pointers to the object.

The difference in binding determines which pairs of file system calls can bind the interval in the programming condition. If the calls refer to files through descriptors, the binding of the file descriptor to the file cannot be changed by a second process. But if either refers to the file by a path name, then another process can alter the binding between name and file if the environment allows it. This observation defines pairs of system calls that allow TOCTTOU binding flaws to occur.

If two sequential system calls refer to the same object using a file path name, the possibility of a TOCTTOU binding flaw arises. If one uses a name and the second a file descriptor, and the first is *not* a call that maps a file path name to a descriptor, a TOCTTOU binding flaw may arise. If

1. If the file path name has exactly one component, the parent node is implicitly added to the path as the first component. The single exception is the root node, which is its own parent.

both use file descriptors, or one maps a name to a file descriptor that the second uses, the possibility of a TOCTTOU binding flaw does not arise. Path names are indirect pointers, so one of the interior pointers may be switched. File descriptors are direct pointers and hence not subject to such fiddling.

3.2. Environmental Condition

The goal of analyzing the environmental condition is to present an algorithmic technique to determine if the assumptions implied by the first call will hold during the interval created by the programming condition. If so, the race condition cannot be exploited. If not, it can. The object may change in one of two ways: alteration of the binding between the name and the object, or alteration of the object itself.

Consider a file F that occurs in two system calls causing the programming condition to hold. At the first system call, F refers to object O_1 and at the second, F refers to object O_2. Partition the set of all users into two subsets: T is the set of *trusted* users who will not alter the binding of F in the interval (that is, $O_1 = O_2$), and U is all other, *untrusted* users. The binding of the file F to the object O_1 is *trustworthy* if and only if no member of U can change the binding of F within the interval.

Define the Boolean function $w(I, o)$:

$$w(I,o) = \begin{cases} true & \text{if some } u \in U \text{ can alter the binding of object } o \text{ in the interval } I \\ false & \text{otherwise} \end{cases}$$

In what follows, d_i, l, and f refer to path components.

Lemma 1. Let d_i be a directory object and f an arbitrary object. Then

$$w(I, d_1/d_2/.../d_n/f) = w(I, d_1) \vee w(I, d_2) \vee ... \vee w(I, d_n) \vee w(I, f)$$

Proof: By induction on n.

BASIS: $n = 1$. If the binding of the name d_1 to the directory object is altered, then d_1/f refers to a new object and so the binding of the name d_1/f is also altered. If the binding of the name f to the object is altered, then the binding of d_1/f is also altered. Hence $w(I, d_1/f) = w(I, d_1) \vee w(I, f)$.

HYPOTHESIS: For $k = 1, ..., m–1$, $w(I, d_1/d_2/.../d_k/f) = w(I, d_1) \vee w(I, d_2) \vee ... \vee w(I, d_k) \vee w(I, f)$.

INDUCTION STEP. Consider $w(I, d_1/d_2/.../d_m/f)$. If the binding of d_1 changes, so will the binding for the object $d_2/.../d_m/f$. Hence $w(I, d_1/d_2/.../d_m/f) = w(I, d_1) \vee w(I, d_2/.../d_m/f)$ and so by the induction hypothesis, $w(I, d_1/d_2/.../d_m/f) = w(I, d_1) \vee w(I, d_2) \vee ... \vee w(I, d_k) \vee w(I, f)$, proving the claim. ■

Lemma 2. Let l be an indirect alias of the path $d_1/.../d_a$. Then

$$w(I, l) = w(I, d_1) \vee ... \vee w(I, d_a).$$

Proof: An indirect alias is semantically equivalent to the path it contains. The result follows immediately from this observation and lemma 1. ■

The significance of these lemmata is the implications for the binding over an interval. Let I be the programming interval, and f the name of the object referenced by the two system calls delimiting I. Then if $w(I, f)$ is true, an exploitable TOCTTOU binding flaw exists.

The lemmata also suggest how to determine the value of w for a given object o. Under the UNIX model of files, the owner of the object must be trusted, and the object must not be world writable. Further, if the group contains any members who are not trusted, the object must not be group writable either.[2] So, to test the trustworthiness of a binding, simply check those conditions for each component in the path name of the file.

If the object is being written, then the current contents of the object are irrelevant as they will be deleted. In this case, the trailing component of the object need not be checked[3]. However, if the object is being read, then altering the current contents of the file is sufficient to exploit a race condition. In this case, the object itself must be trustworthy.

4. A Prototype Implementation of the Analysis

A static analysis tool scans a given program's source code looking for potential TOCTTOU binding flaws. Because different computer systems have different environments, if the analysis program used one system's environment, that result might be invalid on a different system. The

2. System specific semantics may modify this rule; for example, on a SunOS, Solaris, IRIX, or HP/UX system, if the object is a directory, the sticky bit is set, and the next component of the path name exists, then the directory may be world writable. The semantics of the sticky bit in this context is that only the owner of an object may delete it from the directory.
3. Again, system constraints may require some checking. If the parent directory has the sticky bit set, and the sticky bit semantics are as described in the previous footnote, ownership (but not permissions) must be checked.

static tool should report only intervals (by line numbers) on which the programming condition holds. The (human) analyst would then check whether the environmental condition hold during that interval, for each specific system upon which the software is installed.

The static analyzer parses the input C program, and builds a control dependency graph and a data flow graph. From the control dependency graph, the analyzer determines potential programming intervals; from the data flow graph, the analyzer determines if the arguments to the system calls create such an interval. Specifically, if both system calls use file names, the static analyzer determines if the argument are the same; if one uses a file name and the other a file descriptor, the analyzer determines if the file name were bound to the descriptor when the descriptor is created.

Pointer aliasing complicates the data flow analysis. If the pointers are well-behaved, then the initialization identifies the variable to which the pointer refers; but if the pointers are ill-behaved, determining the referent requires complete knowledge of memory as well as specific data values. In essence, it requires the pointer to be evaluated when the program executes.

An even more complex problem is how the program will interact with the environment. For example, suppose one system call access the file "/tmp/X" and a second refers to "../tmp/X". If these refer to the same object, a programming interval exists. However, that cannot be determined without knowledge of the process' current working directory. Adding direct and indirect aliases complicates matters even more.

A prototype tool checks programs written for the SunOS and Solaris versions of the UNIX operating system. The availability of those systems in our environment dictated this choice. Several simplifying assumptions sped the development of the prototype (which is a proof-of-concept program only).

The bounds of the programming intervals constitute the first simplifying assumption. The analysis in Section 3.1 show three types of bounds: both system calls use file names; the initial system call uses a file name and the terminal one a descriptor; and the initial system call uses a file descriptor and the terminal one a name. Because of the complexities of tracking the path names associated with objects assigned file descriptors, the analyzer assumes both system calls bounding the programming interval involve path names.

Selecting only the most common library functions is the second simplification. The use of

library functions conceals the underlying system calls bounding the programming intervals. Since most interaction with file oriented system calls uses the standard I/O library, the list of functions includes the functions in that library which take a path name as an argument and invoke file-oriented system calls[4].

The prototype analysis tool is a Perl script which understands function boundaries but not local blocks, C language dependencies, nor interprocedural analysis. The prototype analyzer uses pattern matching over the source code to approximate generating and scanning a call dependency graph. It does no data flow analysis, but assumes that the file path name arguments are lexically identical in the system calls. That is, the prototype detects:

```
char tempfile[1024];
 ...
creat(tempfile, 0600);
chown(tempfile, 0, 0);
```

but not:

```
char tempfile[1024], *newfile = tempfile;
 ...
creat(tempfile, 0600);
chown(newfile, 0, 0);
```

as in the latter, the arguments are lexically different.

This analyzer was run on *sendmail* version 8.6.10, because *sendmail* has been successfully attacked in the past [16–21]. The output is in Appendix 1. The analyzer reported 24 possible programming intervals; after manual analysis, 5 met the programming condition. Given appropriate environmental conditions and appropriate security policy elements, all 5 allow unauthorized actions (see Appendix 2). Of the 5, one in particular allows users to violate a common element of most site security policies by adding permission, allowing the attacker to read other users' confidential files or mail. Appendix 3 shows the typescript of a sample attack.

5. Analysis Limits

Given an arbitrary program, consider the existence of exploitable TOCTTOU binding flaws in a program to be a property. Then this property holds for at least one computable program. By Rice's theorem [10], the set of programs for which this property holds is undecidable, so no

4. The command *nm*(1) lists the system calls in object files, among other externally defined labels.

generic decision procedure exists to determine if all programs have this type of exploitable TOCTTOU binding flaw.

Consider those UNIX programs which exhibit the programming condition and the environmental condition. Let E be the set of exploitable TOCTTOU binding flaws in one such program, and let R be the set of exploitable TOCTTOU binding flaws that the analyzer reports for that program. Let $E' = E \cap R$. If $E' = E = R$, then the analyzer is *precise* with respect to the program. If $E' = E \neq R$, then some exploitable TOCTTOU binding flaws are not reported, so the analyzer is *deficient* with respect to the program. If $E' = R \neq E$, then all exploitable TOCTTOU binding flaws are reported, as well as some conditions which are not really exploitable race conditions; the analyzer is *excessive* with respect to the program. Finally, if $E' \neq R$ and $E' \neq E$, then the analyzer is *incomplete*.

Deficient and excessive analyzers exist (trivial examples are the analyzer which always reports no exploitable TOCTTOU binding flaws and the analyzer which reports that every pair of system calls causes an exploitable TOCTTOU binding flaw). Determining whether an analyzer is precise or incomplete requires examining each of the two conditions in detail.

The programming condition requires detection of sequential system calls, the first of which must check for some property and the second of which must act on that property. In fact, the first system call may simply gather information which is then checked. The precise nature of the check depends upon the needs of the action and the programmer; for example, access permission may be checked using *access* (which performs the check) or *stat* (which obtains file information that can then be checked). Further, the distinction between system calls which "check" and system calls which "use" is a product of the program; for example, a program which lists file attributes might call *stat* to obtain the information, whereas another program might call *stat* to check authorization to access. Thus a precise analyzer would require some means of determining which calls were "checks" and which were "uses."

The environmental condition complicates this. Given a set of system calls which could bound the programming interval, the analyzer can report all possible programming intervals in a program. The interaction of the environment with those potential intervals creates problems beyond the trustworthiness of the file being accessed. The environment controls the interpretation of the name of the file used in the system calls. File aliases (both direct and indirect) and the low-level

representation of secondary storage add more complexity. The analyzer must have this information available.

In short, a precise analyzer requires a complete representation of the environment induced by the file system, and knowledge of the pairs of system calls required for checks and uses. All of this information is unlikely to be available in practise.

An analyzer is incomplete when it fails to report exploitable TOCTTOU binding flaws, and exploitable TOCTTOU binding flaws are reported erroneously. The manner in which the former can occur is clear; the latter occurs when (for example) data flows are not adequately traced. The prototype analyzer described in the previous section is an example of an incomplete analyzer.

6. Conclusion

As static analyzers cannot be precise, can dynamic (run-time) analyzers be precise? A dynamic analyzer tests the environment during execution, and warns when an exploitable TOCTTOU binding flaw occurs. Basically, the system call interfaces are modified to track the arguments and the association of file descriptors and names. Two successive system calls meeting the programming condition constitute a programming interval, and the trustworthiness of the object is tested at both system calls. If the object is untrustworthy at either point, either an exploitable TOCTTOU binding flaw exists or a trusted user has made an error. Further, the test does not introduce any new TOCTTOU binding flaws.

To elaborate, four combinations of trustworthiness are possible:

1. The object is trustworthy at both system calls. Then the object could not be changed during the interval, and no exploitable TOCTTOU binding flaw occurs.
2. The object is untrustworthy at both system calls. An exploitable TOCTTOU binding flaw exists.
3. The object is untrustworthy at the initial system call but trustworthy at the terminal system call. Then a trusted user changed those components of the object's name that were untrustworthy at some time *T* in the interval. But from the initial system call to time *T*, an exploitable TOCTTOU binding flaw existed.
4. The object is trustworthy at the initial system call but untrustworthy at the terminal system

call. Then a trusted user altered a component in the object's path to make it untrustworthy. By assumption (specifically, the definition of "trusted user") no trusted user will alter a component to make a trustworthy object untrustworthy; if such a user does, that should not have been trusted.

Dynamic analysis takes run-time environment into account and so provides a more precise testing of the program. It may not be precise, since references to disk block numbers will bypass virtually all reasonable checks. It could be made precise by having the modified system calls emulate the actions of the kernel in resolving file names, but only at considerable expense. Further, dynamic analysis does not prevent the TOCTTOU binding flaws from being exploited.

Many systems provide detailed audit capabilities. Assuming the log includes entries for the expansion of every indirect alias, an analysis of the log entries for file accesses would detect programming intervals and, given an initial environment, could also check that the environmental condition holds. From this, exploitable race conditions can be detected. Further, as the name of the object is known, analysis of other logs could indicate if the condition was in fact exploited.

The detection of security problems arising from race conditions is amenable to testing based on desired properties [8][11]. One such property is that the programming and environmental conditions not exist simultaneously; the precise statement of this property will vary from program to program, but if both conditions hold, a race condition may be exploited. Conversely, if the program contains portions of code for which the programming condition holds, the analyst can determine under what conditions an exploitable race condition will arise. The precise specifications needed to detect these problems varies from program to program, but a generalized template would ameliorate the difficulty of writing such a property for each program tested. This area is under active research.

This work studied TOCTTOU binding flaws arising from file system accesses. Processes interact, as do network objects, and their representation is often as objects other than files. Whether a similar technique will work in that case, and if so what the programming and environmental conditions should be, is an area for future work.

Acknowledgment: This work was supported through a contractual arrangement with the United States Air Force. Special thanks to Scot Templeton, who helped classify many security holes; to Rebecca Bace, Tim Grance, Toney Jennings, Karl Levitt, Ron Olsson, Dan Teal, and Kevin Zeise

for encouragement and useful discussions throughout; and to the anonymous referees whose diligence and thoughtful comments greatly improved this paper.

Availability of Tools. A library function implementing the tests described in section 3.2 is available at ftp://nob.cs.ucdavis.edu/pub/sec-tools/trustfile.tar.gz. Release of the prototype race condition analyzer has not yet been approved.

References

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

[2] Bach, M. J., *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ (1987).

[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] Bishop, M. and Klein, D., "Improving System Security via Proactive Password Checking," *Computers & Security* **14**(3) pp. 233–249 (1995).

[5] Denning, D., "An Intrusion Detection Model," *IEEE Transactions on Software Engineering* **SE-13**(2) pp. 222–232 (Feb. 1987).

[6] 8LGM, "[8lgm]-Advisory-7.UNIX.passwd.11-May-1994," available from *fileserv@bag-puss.demon.co.uk* (May 1994)

[7] 8LGM, "[8lgm]-Advisory-5.UNIX.mail.24-Jan-1992," available from *fileserv@bag-puss.demon.co.uk* (Jan 1992)

[8] Fink, G. and Levitt, K., "Property-Based Testing of Privileged Programs," *Proceedings of the Tenth Annual Computer Security Applications Conference*, pp. 154–163 (Dec. 1994).

[9] Garvey, T. D. and Lunt, T. F., "Model-Based Intrusion Detection," *Proceedings of the Fourteenth National Computer Security Conference*, pp. 372–385 (Oct. 1991).

[10] Hennie, F., *Introduction to Computability*, Addison-Wesley, Reading, MA (1977).

[11] Ko, C., Fink, G., and Levitt, K., "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring," *Proceedings of the Tenth Annual Computer Security Applications Conference*, pp. 134–144 (Dec. 1994).

[12] Landwehr, C. E., Bull, A. R., McDermott, J. P., and Choi, W. S., "A Taxonomy of Computer Program Security Flaws," *Computing Surveys* **26**(3) pp. 211–255 (Sep. 1994).

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

[14] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," *Communications of the ACM* **17**(7) pp. 365–375 (July 1974).

[15] Saltzer and Schroeder, "Protection in Computer Systems," *Proceedings of the IEEE* **63**(9) (1975).

[16] Scheifler, R. W. and Gettys, J., "The X Window System," *ACM Transactions on Graphics* **5**(2) pp. 79–109 (Apr. 1987).

[17] *Sun Sendmail Vulnerability*, CERT Advisory CA-90:01 (Jan. 1990), available from *cert.org* via anonymous *ftp*.

[18] *Sendmail Vulnerability*, CERT Advisory CA-93:16 (Nov. 1993), available from *cert.org* via anonymous *ftp*.

[19] *Sendmail Vulnerability (Supplement)*, CERT Advisory CA-93:16a (Jan. 1994), available from *cert.org* via anonymous *ftp*.

[20] *Sendmail Vulnerabilities*, CERT Advisory CA-94:12 (July 1994), available from *cert.org* via anonymous *ftp*.

[21] *Sendmail Vulnerabilities*, CERT Advisory CA-95:05 (Feb. 1995), available from *cert.org* via anonymous *ftp*.

Appendix 1. Analyzer Output

This appendix shows the output of the analyzer run on the source code to *sendmail* version 8.6.10. Only those files with possible problems are shown; the analyzer actually prints the name of each file it analyzes, whether or not the file contains a potential problem. The lines beginning with numbers list the potential race conditions; each lists the line number and system (library) call that may cause the condition, and the common argument follows both.

```
alias.c:
429:fopen, 432:fopen, map->map_file
conf.c:
714:nlist, 721:nlist, %s
deliver.c:
2186:stat, 2262:chmod, filename
main.c:
708:stat, 784:chdir, QueueDir
1325:freopen, 1336:open, "/dev/null"
queue.c:
118:open, 144:rename, tf
118:open, 364:rename, tf
144:rename, 364:rename, tf
694:rename, 702:fopen, d->d_name
977:fopen, 1028:rename, qf
977:fopen, 1149:rename, qf
1028:rename, 1149:rename, qf
1036:unlink, 1149:rename, qf
readcf.c:
612:stat, 625:access, filename
612:stat, 630:fopen, filename
625:access, 630:fopen, filename
```

```
recipient.c:
645:lstat, 646:stat, filename
645:lstat, 648:stat, filename
646:stat, 648:stat, filename
util.c:
462:stat, 505:stat, fn
504:lstat, 505:stat, fn
462:stat, 507:stat, fn
504:lstat, 507:stat, fn
505:stat, 507:stat, fn
```

Appendix 2. Analysis of Output

This appendix describes the analysis of the race conditions identified by the human analyst after looking at the causes of the output in Appendix 1. In the following, the analyst assumes that the security policy of the site includes the following elements:

1. A user can change the protection modes of the files he or she owns, and no others.
2. If a user cannot read or search a directory, he or she should not be able to discover anything about the files or file names in that directory.
3. If a regular file is to be read, a terminal (or other non-regular file) may not be substituted for that regular file.
4. The definition of a *sendmail* configuration file macro class is set by the system administrators and not by unprivileged users.

The analyst also assumes that *sendmail* runs with system privileges, usually *root*. The race conditions are listed first; the analysis of the other reported intervals follows.

`deliver.c, 2186:stat, 2262:chmod, filename`

The routine *mailfile* sends mail to a named file. If the mail were to be written to a file in a directory writable by the attacker (the environmental condition), a race condition exists. The attacker links to that file any object with the desired permissions (which must allow the sender to write to the object). Then, between the *stat* (2186) and the *chmod* (2262), the attacker changes the file name to be the target file name. At line 2262 the protection mode of the target file is set to the protection mode of the original file. This action violates policy element 1.

`main.c, 708:stat, 784:chdir, QueueDir`

The *stat* to determine ownership of the queue directory occurs before the *chdir* to prevent the

user from running *sendmail* and switching into a protected directory. But if an attacker can switch the referent of the name of the directory to a protected (unreadable and unsearchable) directory between the *stat* and the *chdir*, *sendmail* will list the names of files beginning with *qf* in that directory, violating policy element 2. The environmental condition is that the configured mail queue directory (usually "/usr/spool/mqueue") be untrustworthy.

```
readcf.c, 612:stat, 625:access, filename
readcf.c, 612:stat, 630:fopen, filename
```

Given a line in *sendmail.cf* that defines a class from the contents of a file, *stat* checks that the file is a regular file. If the object named in *stat* is a symbolic link, *stat* reports on the object to which the link refers. If that link, or the object to which it is linked, is untrustworthy (the environmental condition), then immediately after the *stat* the link or object can be replaced by a link to a non-regular file, such as a terminal. The file is read using *fgets*(3), which also accepts input from a terminal. This action violates policy element 3.

```
readcf.c, 625:access, 630:fopen, filename
```

This race condition is a modification of the previous two, the only change being that the attacker changes the untrustworthy object after access is checked but before the object is opened. This action violates policy condition 4.

The following reports appeared to the scanner to be programming intervals but upon further analysis were not:

```
alias.c, 429:fopen, 432:fopen, map->map_file
```

The second function is in a conditional entered only when the first function fails

```
conf.c, 714:nlist, 721:nlist, %s
```

They are in a string argument to *printf*

```
main.c, 1325:freopen, 1336:open, "/dev/null"
```

The first function opens to read from it, and the second function to write to it, so the second function would make any reads return **EOF** — which is exactly what reads from */dev/null* do.

```
queue.c, 118:open, 144:rename, tf
queue.c, 118:open, 364:rename, tf
queue.c, 144:rename, 364:rename, tf
```

These functions open temporary files, and if the *open* fails or the file is locked, the temporary file is renamed so a new one can be tried

```
queue.c, 694:rename, 702:fopen, d->d_name
```

If the *rename* is reached, the next statement moves the flow of control to the top of the loop and a new file name is read; so the two functions will never be executed sequentially

```
queue.c, 977:fopen, 1028:rename, qf
queue.c, 977:fopen, 1149:rename, qf
queue.c, 1028:rename, 1149:rename, qf
```

The *rename* is executed only when the effective UID and the owner of the file are different or the file contains an invalid line. In both cases, the name is reset to a constrained queue file name which will be different than any other file name.

```
queue.c, 1036:unlink, 1149:rename, qf
```

The routine returns after the *unlink*, so at most one of these functions will be executed.

```
recipient.c, 645:lstat, 646:stat, filename
recipient.c, 645:lstat, 648:stat, filename
recipient.c, 646:stat, 648:stat, filename
```

Only one of the functions is ever executed.

```
util.c, 462:stat, 505:stat, fn
util.c, 504:lstat, 505:stat, fn
util.c, 462:stat, 507:stat, fn
util.c, 504:lstat, 507:stat, fn
util.c, 505:stat, 507:stat, fn
```

The functions have different arguments in the same variable on each call.

Appendix 3. Sample Exploitation of the Vulnerability

What would an exploitation of the first vulnerability in Appendix 2 look like? This shows the hypothetical result of one such exploitation.

```
1 % cat /.forward
sysadmins,/usr/spool/rootlog
2 % ls -ld /usr/spool
drwxrwxrwx   1  root    512 Dec  5 21:13 /usr/spool
3 % ls -ld /usr/spool/rootlog
-rw-r--r--   1  root    526 Dec 10 11:34 /usr/spool/rootlog
4% ls -sail /etc/pwd/shadow
-r--------   1  root   1329 Nov 16 16:58 /etc/pwd/shadow
5 % runrace /etc/pwd/shadow
won: /etc/pwd/shadow protection modes changed
-rw-r--r--   1  root   1329 Nov 16 16:58 /etc/pwd/shadow
```

1. The letter will be appended to the named file. An alternative is to look for bounced mail, which is appended to the file “dead.letter” using the same delivery mechanisms.

2. The mail file "/usr/spool/rootlog" is in a world writable directory and so can be deleted.

3. The mail file is world readable.

4. The shadow password file (which holds hashed passwords) is protected to prevent users from copying the hashed passwords and launching dictionary attacks [4].

5. The attack tool "runrace" sends mail to root, and as that mail is being delivered tries to replace "/usr/spool/rootlog" with "/etc/pwd/shadow". If the replacement succeeds, the protection modes of "/etc/pwd/shadow" are reset to those of "/usr/spool/rootlog".

We reported this security problem to the author of *sendmail*, and the *sendmail* 8.7 base distribution fixes the problem on all systems with a *fchmod*(2) system call.