Checking for Race Conditions in File Accesses

Matt Bishop and Michael Dilger

CSE-95-10

September 1995

Checking for Race Conditions in File Accesses

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

Abstract

We develop a theory of vulnerabilities and their signatures, and use this theory to categorize race conditions that occur when processes interact with files in the UNIX operating system and that present security vulnerabilities. We present a formal language for describing these vulnerabilities, and derive an underlying characteristic. Using this characteristic, we present a tool that analyzes programs for possible race conditions, and present the results of one such analysis in which five previously undiscovered potential race conditions were located in a very widely used program. We conclude that the basic theory and application is sound enough to aid in the detection of those flaws, and that the methodology appears to generalize well to other classes of vulnerabilities.

Introduction

Unlike ordinary bugs, which prevent applications or systems from functioning correctly, a security hole enables a user (called an *attacker*) to gain privileges, access to data, or the ability to interfere with others' work. Researchers have studied the nature of attacks in the context of intrusion detection [4][6]; in this context, the goal is to determine that an attack is occurring, as well as the nature of the attack, from logs. Each attack must have a unique signature in order to differentiate it from other attacks, even if they exploit the same underlying security hole, called a *vulnerability.*

This paper focuses on the characteristics of the underlying vulnerabilities that attackers exploit rather than the precise mechanism used to exploit them. In essence, we do not distinguish between two different attacks which take advantage of the same vulnerability because we are not interested in different techniques to exploit that vulnerability. We simply note the characteristics of the vulnerability, and from them extract general mechanisms for possible attack.

This approach has several benefits. First, it enables us to characterize the precise conditions under which an attack will effectively exploit a vulnerability. If the vulnerability is not present, it cannot be exploited; if it is, then all attacks exploiting it will take specific steps, in any of a number of specific environments, to do so. The central theme of this paper is that those steps and that environment can be characterized precisely, so to detect the attack, one need only spot the exploitation steps in a log, and then verify that the environment allowed exploitation of the flaw using the found steps. As an example, consider the attack described in [9] which allows a user to gain superuser (operator) privileges on some UNIX systems [13]. The steps are:

1. Copy a command interpreter to the superuser's mailbox, which is implemented as an ordinary file; this works when the mail spool directory, which contains the mailboxes, is world writable. Note this deletes any of the superuser's mail by overwriting the mailbox.
2. As the attacker owns the (newly-created) mailbox, make it executable and setuid to the superuser. Now, if it could be executed, it would run with the attacker's privileges; all that remains

is to change the owner of the superuser’s mailbox to the superuser.

3. Create an empty file and mail it to the superuser. When the mail program delivers the letter, it will append the letter to the superuser’s mailbox in the spool directory. It will also note that the superuser does not own the mailbox, and reset the mailbox’s owner to the superuser. The protection modes (such as setuid and excised permissions) are not affected.
4. The attacker can then execute the superuser’s mailbox as a program; it will provide a command interpreter (due to the copy in step 1), and run with superuser privileges (due to the setuid bit in step 2 and the ownership of the mailbox in step 3).

The primary vulnerability here is a failure to verify that the mailbox is not an executable program when changing the owner; specifically, the mail delivery program does not check that the protection modes are set to a safe state (readable and writable by the owner, and all other permissions disabled). This vulnerability can be characterized by an environment setting (the fact that the spool directory is world writable) and a sequence of actions (at the lowest level, a change of protection mode followed by a change of ownership). Note that this is not the only possible characterization; for example, a slightly different characterization would be that the mailbox be world writable and the sequence of actions be the setting of world execute permission on the mailbox followed by a change of (group) ownership. However, it does provide a way to fix the vulnerability (specifically, prevent the sequence of actions from occurring in the environment).

The second benefit of our approach is that it provides a way for an analyst to look for problems in existing code. For example, given a set of environments and related sequences of actions for the above vulnerability, one can determine whether it is present. Note that the analysis should not be confined to programs such as the mail delivery agent, but rather any program which can alter both the permissions and the owner of a file (or can perform all the steps in any sequence of actions).

The third benefit is to provide guidelines for fixing the flaw. Essentially, one obtains a complete characterization of the flaw, and simply ensures that the sequence of actions cannot occur in the corresponding environment. One can also determine if a particular program creates a mechanism for attack by looking for a sequence of actions and, if present, then checking to see if the environment under which the flaw arises can exist.

This paper discusses both the theory and application of these ideas to the problem of race conditions during file accesses. The next section presents some background, especially work done in the field of analysis of systems for vulnerabilities and some similar work (but with a very different twist) in intrusion detection. Characterizing vulnerabilities is the topic of the third section, and the fourth applies these ideas to race conditions, presenting both an analysis of them and a prototype race condition checker. The final section discusses the results of this work and suggests some avenues for future work.

Background

The analysis of systems for vulnerabilities has been a constant, ongoing effort since multiprocessing began; indeed, one motivation behind the use of memory protection was to prevent processes from writing to one another’s memory. Codification of techniques to examine systems began with the Flaw Hypothesis Methodology [12], and were extended by the Program Analysis (PA) project [3] and the Research Into Secure Operating Systems (RISOS) project [1], which provided taxonomies for classifying vulnerabilities. Both these projects focussed on operating systems, and argued that the taxonomies were generic rather than specific to individual machines. They backed this argument up by analyzing flaws in several operating systems, and suggesting common underlying features of members of each class of flaws. In some cases, they developed tools to aid in the analysis of operating systems.

Denning's seminal intrusion detection paper [4] suggested automating detection of attacks, and her anomaly detection model was quickly joined by a misuse model of intrusion detection. This model, predicated on the fact that each attack can be described as a sequence of actions, examines system logs looking for those sequences of actions. If found, a system security officer is notified that an attack may be (or may have been) occurring. This led to research into characterizing attacks on specific operating systems and environments, and into classifying attacks in general [10]

Our work is related to earlier work in vulnerability analysis, and is similar to (but distinct from) attack analysis. With respect to the earlier studies on system flaw analysis, we focus on application-level programs rather than the operating system (although many of our techniques could be adapted to work with an operating system). Our justification for this is that many operating systems have a built-in trapdoor, allowing some privileged user complete control over the system, and therefore analyzing the programs which might exploit that flaw will provide insight into vulnerabilities of systems. Further, in the author's experience, few attacks exploit specific operating system flaws; most exploit flaws in a privileged program or concurrent execution with a privileged and an unprivileged program. Given the large number of programs that are available free or for little cost on the World Wide Web, as well as commercially available programs, analysis of vulnerabilities they may introduce would be quite useful. Providing guidance on how to check for these vulnerabilities would be even more useful, and one goal of our work is to provide such guidance.

Rather than focusing on the characterization of attacks, we study the vulnerabilities underlying the attack, which is the cause of the problem rather than the symptom of it. More pithily, we seek to understand *what* is being exploited, not *how* it is being exploited. The key benefit is being able to minimize the descriptions of the characterizations of the vulnerabilities. Note that studying attacks requires a characterization sufficient to distinguish two different attacks exploiting the same vulnerability. From our point of view, the difference is irrelevant. The same vulnerability gets exploited.

Further, our focus is more on detection and prevention, rather than detection during and after exploitation. A good analogy is between this and securing reusable passwords; detecting attacks is similar to password cracking, in which the attack is detected after the system is vulnerable. Detecting vulnerabilities is similar to proactive password checking, in which the source of the vulnerability (programs or passwords) is analyzed before it is put into place. The latter approach minimizes risk because the instance of the known vulnerability is reported before being added to the system.

Our ultimate goal is to produce a "programmer's toolkit" containing (among other things) tools to analyze programs for potential problems when they are written. In some cases, vulnerabilities can be fixed, and in other cases, the specific environments in which the vulnerability arises can be noted so users (and installers) will know the risks and be forewarned about the dangers of using the program. This is closely related to the development of a "tester's toolkit" [5], which will provide tools and a library for assurance and testing in a more general realm.

A Theory For Characterizing Vulnerabilities

Characterization of vulnerabilities encompasses many levels of abstraction; for example, vulnerabilities may be introduced at the design level, the specification level, the implementation level, or the operational level. What follows is phrased in terms of the implementation level, because that is where it currently can make its greatest contribution. But generalization to higher levels of abstraction is quite possible and will be discussed later.

We base our characterization upon a taxonomy which is a descendant of the Program Analysis taxonomy; for the purposes of this paper, we group several axes of the taxonomy into a single class we shall call *environmental information*. As our characterization of vulnerabilities is closely related to characterizations of attacks, let us begin there.

We define a *signature* of an attack to be a minimal set of environmental information and a minimal sequence of actions which results in a breach of security. By *minimal* we mean that no unnecessary information is present; in other words, if any action in the sequence is omitted, or any of the given environmental information is omitted, the attack fails.

As an example, consider the attack described in the introduction. Its signature is:

((*allowed*({ *w* }, *spool_dir*, A) , *create*(*spool_dir*/Umailbox, A)) ,
(ν , *read*(*cmd_interpreter*, A, *buf*)),
(ν , *write*(*spool_dir*/Umailbox, A, *buf*)),
(ν , *close*(*spool_dir*/Umailbox, A)),
(ν , *ch_perms*(*setuid_A*+*execute*, *spool_dir*/Umailbox, A)) ,
(ν , *ch_owner*(U, *spool_dir*/Umailbox, *root*)))

This signature consists of a sequence of pairs. Each pair consists of an environmental condition that must hold, and a command that must be executed (we use ν to mean "no special conditions" or "no action"). In the example above, the commands are at the level of system calls. The commands may in fact be at any level of abstraction that is convenient for the analyst. For our purposes, this is the system call level. For the attack to succeed, each command must succeed. For the command to succeed, the environment condition must hold.

The attack requires the execution of six system calls by specific users in a given order. The first call is by user A (attacker) to create a mailbox for U, the victim. In order to do this, the spool directory in which the mailbox is to reside must be writable by A. A copies the command interpreter to the mailbox (the next three commands). Then A sets both execute and setuid permissions; as A owns the file, no environmental conditions are needed. Finally, the superuser, *root*, must change the ownership of the mailbox from A to U. Again, as the superuser can always do this, no environmental conditions are needed. Given this sequence, there now exists a setuid to U program that is executable by an user on the system, and hence anyone can work with U's privileges.

We should note that the first component of the pair is strictly environmental. In our example, given that the spool directory is writable by A, and that the system appropriately defines the semantics of the *ch_owner* system call, this attack will succeed. A second, related attack would exist if there were a program that could create a file owned by A in the spool directory. Its signature would, however, be different.

Note also that this characterization of an attack is at a lower level than other characterizations, since it ignores the expression of the execution of the system calls. That is, this sequence may be launched interactively by three distinct programs, by a single shell script, by a C program, by a command script, and so forth. All that is irrelevant. The relevant part is the initial state and the three system calls.

Given a specific vulnerability, we define its signature as the set of minimal attack signatures that exploit the vulnerability. For example, the above attack takes advantage of two distinct vulnerabilities, the first being the improper initial configuration of the system (that is, spool directories should not be world writable) and the second being the failure of the change owner command to check that A can "give away" the mailbox. Both must be present for this attack to work. Let us deal with these separately.

The first vulnerability has a very simple signature, since it is concerned with environmental (state) information and nothing more. Its signature, *in toto,* is:

(*allowed*({ *w* }, *spool_dir*, A) , ν)

so if this holds, the system is vulnerable. Other attacks exploit this also. That the sequence of commands is irrelevant here is typical of signatures of this class of flaws, which makes sense as they deal with improper initializations.

The second vulnerability has a more complex characterization. It is the changing of ownership without adequately validating the file being given away; in this case, the required validation would be to note the setuid bit and abort.[1] It may be characterized as follows:

((*allowed* ({ *suid* }, *file*), *ch_owner*(U, *file*)) ,
(*allowed* ({ *suid* }, *file*), ν))

Notice that one needs environmental information to characterize this vulnerability. This says that initially, the file has the setuid bit set. Its ownership is changed to U. After the change, the setuid bit remains set. Both pairs are needed; omitting the second leads to a characterization which can be met by systems which do not preserve the setuid bit over change of ownership; yet they do not have this vulnerability.

This also emphasizes the difference between an attack signature and a vulnerability signature. An attack signature should *not* have post-conditions to describe whether the attack has succeeded. The reason lies in the use of such signatures; one wants to find attacks even if they fail, so that the attack fails is in some sense irrelevant. (Our example attack signature was written with this in mind.) The vulnerability signature is binary in purpose; whether the vulnerability exists, or it does not. The post-condition (second pair) describes what must hold if the vulnerability exists and hence is an integral part of that signature.

We can use this type of characterization to look for vulnerabilities in programs. Rather than consider attack signatures, which describe exploitations, we focus on the vulnerabilities, which characterize the minimal conditions and sequences of system calls that must exist for exploitation to occur. The next section applies this notion to the class of flaws called *race conditions* by developing some signatures for that class of vulnerability, and then looking for them in programs.

Derivation of the General Vulnerability Signature for Race Conditions Involv-

1. Most systems, in fact, simply turn this bit off.

ing Files under UNIX

Consider now a slightly different attack, the *mkdir* attack from UNIX version 7 [15]. The attack is characterized with the following signature[1]:

```
( (                   ν                  , createdirfile("./temp", root) ) ,
  ( allowed ( { w }, "." )    , removedirfile("./temp", A) ) ,
  (                   ν                  , direct_alias(password_file, "./temp") ),
  (                   ν                  , ch_owner(A, "./temp", root) )  )
```

The command *mkdir* creates a directory. It works by first creating a directory file as the omnipotent user (necessary as the appropriate system call may be executed only by that user) and then changes ownership of the directory to the executor of the *mkdir* command. However, if between the creation and the change of ownership, the directory file is deleted (as the semantics of the UNIX operating system allow) and a pointer to another file substituted, the file being pointed to will have its ownership changed to A. That is, the change owner call within the *mkdir* command does not check that the object affected is the same as the one created in the first step.

This takes advantage of a vulnerability called a *race condition*, which we shall exploit in the next section. The specific vulnerability here is the non-atomicity of the *createdirfile* and *ch_owner* calls. If those are executed without the other intervening calls, there is no problem; if the intervening calls are executed, A will own *password_file* (and be able to alter it at will). The characterization of the vulnerability is:

```
( (               ν                  , createdirfile("zzz") ) ,
  ( allowed ( { w }, "." )     , removedirfile("zzz", A) ) ,
  ( not_owner("xxx", A)    , direct_alias("xxx", "zzz") )
  (               ν                  , ch_owner(A, "zzz", root) ) ,
  (  is_owner("xxx", A)     , ν) )
```

A second race condition flaw occurs in the program *xterm*, a terminal emulator for the X Window System [14]. The attack is:

```
( ( allowed( { w }, "/tmp", A ) & not_exist("/tmp/logfile")
                                                  , createfile("/tmp/logfile", root) ),
  (                              ν                , removefile("/tmp/logfile", A ) ),
  (                              ν                , direct_alias(password_file, "/tmp/logfile") ),
  (                              ν                , ch_owner(A, "/tmp/logfile", root) ) )
```

The program *xterm* runs as the omnipotent user on many systems. When a user asks it to log all input and output to a specific file, the program checks to see if the file exists. If not, it creates it (as the omnipotent user) and changes ownership to the executor of the program. This is the case the above signature deals with.[2] The program creates the log file and changes its ownership (steps 1 and 4); but between those two steps, the attacker deletes the file and substitutes a pointer to a privileged file. The fourth step then changes ownership of the privileged file.

1. Here, "." refers to the current directory. The names of the system calls are not those used by the UNIX operating system, but are more descriptive; the mapping is that *createdirfile* is *mknod*, *removedirfile* is *unlink*, *direct_alias* is *link*, and *ch_owner* is *chown*.
2. In the other case (that the log file exists), it checks that the executor of the program can write to the named file, and then opens the file. This also creates a race condition which can be modelled the same way (the precise signature is left as an exercise to the reader).

Again, the problem is that the creation and change of ownership must be atomic. As they are not, the file whose ownership is in question can be switched (deleted and linked to another file).

Notice that both these race conditions arise because the design calls for the change of ownership and the creation of the object (directory or file) to be atomic; yet the programs do not enforce the atomicity. This is a feature of the UNIX operating system; it is not possible to guarantee that two successive system calls will not have system calls from other processes interleaved, due to the way the UNIX system schedules its processes. In short, neither the programs nor the operating system provide adequate atomicity of system calls; the programs cannot do so of themselves, and the kernel does not provide adequate support for critical sections.

Thus, one class of race conditions involving file accesses may be characterized by the following pattern:

((*condition-1*, *call-1*), (*condition-2*, *call-2*))
where *call-1* and *call-2* are system calls and *condition-1* and *condition-2* are environmental conditions needed for the calls to produce a race condition. Not all pairs of system calls will cause race conditions, of course; so, the next step is to decide which pairs of system calls need to be indivisible (atomic). The way the system represents files to processes will determine this.

The only way a process may access files under the UNIX operating system is through system calls. To access an object (we shall use the more generic word object, since "file" includes directories, devices, and other entities), the process must be able to name the object. The UNIX system provides two different forms of addressing, with different semantics [2][11].

The first address 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 taken through the tree to reach the (leaf or interior) node corresponding to the address. To access the object from a path name, the kernel begins at the beginning of the path name, and accesses each interior node named in the path. Each interior node contains the address of the next node in the path. When the next-to-last final node in the path is accessed, the address of the final node is obtained, and 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 address is a file descriptor. File descriptors are assigned to a file on a per-process basis, and bind the address directly to the object. For example, 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 pointer (the file descriptor) to the object. When addressed using the file descriptor, the file system is not consulted; instead, the kernel uses the file descriptor to access the object directly.

Notice the subtle difference in the way the addresses are resolved to objects. File path names are resolved by indirection, requiring the accessing of at least one object other than the file being addressed. File descriptors are resolved by simply accessing the object in question. The former are akin to (multiply) indirect pointers to the object; the latter are akin to simple pointers to the object.

This is the key to determining which pairs of file system calls can be considered atomic and

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.

which ones cannot. If the two calls refer to files through descriptors, they are effectively atomic, as the binding of the file descriptor to the file cannot be changed by a second process. But if either refers to the file by name, then another process can alter the binding between name and file (assuming the environment allows it). We use this to define pairs of system calls that allow race conditions to occur.

If two sequential system calls refer to the same object using a file path name, the possibility of a race condition arises. If one uses a name and the second a file descriptor, and the first is *not* a call which maps a file path name to a descriptor, a race condition may arise. If both use file descriptors, or one maps a name to a file descriptor which the second uses, the possibility of a race condition does not arise. The astute reader will immediately recognize the reason for these classes; that as file path names are indirect pointers, one of the intermediate pointers (nodes on the tree) may be switched, whereas the file descriptors are direct pointers and hence not subject to such fiddling.

The next section presents an example of using this analysis to classify specific system calls, and presents the results of applying this theory to one program.

Application of the Theory

The specific goal of this section is to apply the above theory to analyze a program and locate any race conditions that may exist in the program. To simplify the problem, we make several assumptions:

5. The analysis program should ignore environment. Our reasoning here was that different computer systems have different environments, and if the analysis program based its analysis upon the current system's environment, that result would be invalid on a different system. To avoid the attendant confusion that might cause introduction of a flawed program onto a system, the analysis tool will simply check for the possibility of a race condition; the (human) analyst must evaluate the environmental conditions to determine if the current system configuration allows for the condition to occur in practise.
6. The prototype analysis tool will be based upon pattern matching. This was a pragmatic decision, done simply because we wanted to prove the applicability of the theory. In the conclusion, we shall outline how a production-quality tool would need to work; suffice it to say that problems such as pointer aliasing are quite difficult.
7. Although the analysis above was done with system calls, many programs interact with the operating system not through system calls but through library functions that invoke system calls (the Standard I/O Library is perhaps the best example) [16][7]. So we include those in the same way we included system calls. In what follows, when we refer to "system calls," we are including these library functions.

Given these assumptions, the tool would work as follows. Look for a pair of system calls with the same file name as argument. This raises the possibility of a race condition. The results are then manually checked. The relevant system calls are given in Table 1.

We wrote an analyzer to scan C programs for these sequences. The analyzer worked by pattern matching, which meant (among other things) that file path name arguments had to be lexically identical in the system calls. That is, it would pick up the following sequence:

```
char tempfile[1024];
 ...
```

access	acct	au_to_path	basename	catopen
chdir	chmod	chown	chroot	copylist
creat	db_initialize	dbm_open	dbminit	dirname
dlopen	execl	execle	execlp	execv
execve	execvp	fattach	fdetach	fopen
freopen	ftok	ftw	getattr	krb_recvauth
krb_set_tkt_string	kvm_open	lchown	link	lstat
mkdir	mkdirp	mknod	mount	nftw
nis_getservlist	nis_mkdir	nis_ping	nis_rmdir	nlist
open	opendir	pathconf	pathfind	readlink
realpath	remove	rename	rmdir	rmdirp
scandir	stat	statvfs	symlink	system
t_open	tempnam	tmpnam	tmpnam_r	truncate
umount	unlink	utime	utimes	utmpname
utmpxname				

Table 1. Some UNIX system calls that may lead to race conditions involving process/file interactions [16].

```
create(tempfile, 0600);
chown(tempfile, 0, 0);
```

but not this:

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

as in the latter, the arguments are lexically different. We felt that resolving these problems would require much work, and we wanted to analyze the feasibility of applying the theory before developing sophisticated tools to do so. We shall discuss this further in the next section.

This analyzer was run on *sendmail*, a program notorious for its past problems with security [17][18][19][20][21]; the version analyzed was version 8.6.10. The output is in Appendix 2. The analyzer reported 24 possible problems; after analysis (which in some cases simply consisted of glancing at the source code and grumbling about the primitiveness of the analyzer), 5 serious problems remained. These are summarized in Appendix 3. Of the five, one will work on all systems and allows the attacker to obtain privileges reserved to the system[1], another will work on all systems but is unlikely to produce valuable information, and the other three allow a user to reconfigure the mail system if certain environmental constraints are met.

We concluded the experiment was a success, and the analyzer worth extending. As a (humorous) aside, after these results, we decided to report the problems to the maintainer of sendmail. Before doing this, we obtained the latest copy of sendmail, only to discover that version 8.6.11 had been released hours earlier – and it contained a fix to the first, most serious, of the five bugs. We referred to this as the sixth race condition.

1. The exact nature of the privileges (*root* or *daemon*) depends on how the system is configured.

```
char tempfile[1024];
if (access(tempfile, R_OK) == 0)
     do_open(tempfile);
...
void do_open(char *filename)
{
     if ((fd = open(filename, O_READ)) < 0)
...
```

Figure 2. A program with a potential race condition that the prototype analyzer will not spot.

Conclusion

Given that an analyzer as primitive as our prototype was able to find four race conditions that could pose a potential threat to system security, one can clearly conclude that a more sophisticated checker could spot more possible race conditions, and eliminate many of the false negatives. For example, currently the prototype ignores pairs of system calls with syntactically different arguments even though the arguments may be semantically equivalent. Also, the analyzer does not use a call flow graph to determine the order of invocation of system calls, so the analyzer would not detect the potential race condition in the code fragment in Figure 2. Further, the analyzer does not take any of the environmental information into consideration. As was stated earlier, doing so is not appropriate unless the tool is *only* used on the system on which the target of the analysis will execute. Otherwise, one runs the risk of a false negative, and the accompanying false sense of security.

Solving these problems would require an analyzer based upon the semantic and syntactic content of the program. Specifically, the analyzer would parse the input C program, and build a call dependency graph. From that graph it could determine suspect sequences of calls, and from the symbol table determine the arguments used. The analyzer would need to handle pointer aliasing (see the example in the earlier section) and understand enough of the environment to determine when two file object names referred to the same entity. The latter issue is again one which will require the testing to be redone whenever the program is installed, or a warning for the installer to check for the possible security threat.

However, the analysis tool is not the main contribution of this work; the main contribution is that such a tool can be derived from an appropriate characterization of the vulnerability. This augments the argument that [5], [8], and others have made: that a realistic approach for locating security holes is to work from specifications and characterizations of potential errors or flaws. Interestingly, the description of the vulnerability here drives the specification, which is low level; the specification is that the specific signature of the vulnerability should not occur, and the analyst uses the automated tool to check for violations of the specification (specifically, for those cases where the vulnerability signature does occur). Contrast this to most specification-oriented work in which the specification determines the condition to be checked for; here, the derivation is reversed.

An appropriate taxonomy is also helpful, as it enables one to delimit precisely the classes of vulnerabilities being searched for. Without the taxonomy, one would be unable to classify several vulnerabilities as being of the same type, and not recognize the commonality of their expression. This philosophy underlies much of the work done in misuse modelling for intrusion detection, specifically in the generation of attack databases. Here the focus of the taxonomy is different, as is the application: prevention rather than detection.

Acknowledgement: This work was supported by award TDS 94-140 from Trident Data Systems , Inc. to the University of California at Davis. Special thanks to Scot Templeton, who helped classify many security holes; and to Karl Levitt, Toney Jennings, Dan Teal, and Kevin Zeise for encouragement and useful discussions throughout.

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

[5] 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).

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

[7] Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ (1978).

[8] 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).

[9] Kumar, S. and Spafford, E., "A Pattern Matching Model for Misuse Intrusion Detection," *Proceedings of the Seventeenth National Computer Security Conference*, pp. 11–21 (Oct. 1994).

[10] 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).

[11] 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).

[12] Linde, R. R., "Operating System Penetration," *1975 National Computer Conference Proceedings* (*AFIPS Conference Proceedings* **44**), pp. 361–368 (May 1975).

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

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

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

(1987).

[16] *SunOS 5.4 UNIX User's Manual*, Sun Microsystems Inc. (Feb. 1993).

[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 is 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 names of all files analyzed, whether or not there is 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
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

These are the race conditions identified by the human analyst after looking at the causes of the output in Appendix 2.

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

on inspection, harmless; the second is in a conditional entered only when the first fails

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

on inspection, harmless; they are in a string argument to *printf*

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

The routine *mailfile* sends mail to a named file. This can be used to change the protection mode of one file to that of another. To do this, ensure the mail will be written to a file in a directory writable by the attacker (cancelling the delivery so it goes into *dead.letter* is a good way to do this). First, link to that file anything you like with the right protection modes. Between the *stat* (2186) and the *dopen* (2235) when the file is actually opened, change the file name to be something else. Your letter is appended, and then at line 2262 the original modes (initial ones) of the first file are restored. Note this may include setuid and setgid bits...

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

The *stat* to determine ownership of the queue directory is done before the *chdir*. The goal is to keep the user from running *sendmail* and switching into a protected directory. But this can be used to get into a protected (unsearchable) directory by switching the directory name between the *stat* and the *chdir*. Note that the routine *printqueue* repeats the *stat* and so is safe unless you are in the same group as the queue directory; read and search permission are **not** checked. The result is that the *runqueue* and *printqueue* routines both will list names of files beginning with *qf* in the directory.

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

on inspection, harmless; the first opens to read from it, and the second to write to it, so the second would make any reads return **EOF** -- which is exactly what reads from */dev/null* do anyway

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

On inspection, harmless; these 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
```

On inspection, harmless; if the *rename* is hit, the next statement moves the flow of control to the top of the loop and a new file name is read; so the two 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
```

On inspection, harmless; 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
```

On inspection, harmless; after the *unlink*, the routine returns, so at most one of these functions will be executed.

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

The routine *fileclass* reads the file name, given in the first argument. Suppose you want to make it read the file name from your terminal (that is, you will type it in). The *stat* checks to be sure the file being read is a regular file. You then delete it and put a link to the device file corresponding to your terminal. The *access* check will report that the real user of *sendmail* (you) can read it. It will then read it using *fgets*. So type in your own definitions. This only works if the file with the class definition has an ancestor directory you can write to.

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

The routine *fileclass* reads the file name, given in the first argument. Suppose you want to make it read any file you like. The *stat* checks to be sure the file being read is a regular file. The *access* check will report the real user of *sendmail* (you) can read it. You then delete it and put a link to another file. It will open the file using *fopen* and the effective user privileges of *sendmail* (often *root*). So make your own class definitions. This only works if the file with the class has an ancestor directory you can write to.

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

On inspection, harmless; 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
```

On inspection, harmless; the functions have different arguments in the same variable on each call