A Unified Approach to Detecting Binding Based Race Condition Attacks

Bharat Goyal, Sriranjani Sitaraman and S. Venkatesan

Digital Forensics and Emergency Preparedness Institute and
Distributed Systems Research Group
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083
{bharatg, ginss}@student.utdallas.edu, venky@utdallas.edu

Abstract

Synchronization flaws due to race conditions, in which the binding of a name to an object changes between repeated references, occur in many programs. A malicious program acts by exploiting a window of opportunity between the points of execution of another program. Such vulnerabilities can be exploited to gain unauthorized access to resources of a system. We propose a unified approach to detect and analyze these flaws in the Unix file system. We use our approach to identify binding based race condition attacks in an interleaved execution trace of a system utility and a malicious program. We apply our approach to numerous binding based race condition attacks. We also present an algorithm to flag these attacks when an interleaved execution trace of a system utility with a malicious program is available.

1. Introduction

Security holes or vulnerabilities allow an attacker to gain unauthorized privileges, gain unauthorized access to protected data or interfere with the work of others. The characteristics of such attacks have been well studied [4], [6], [7], [8], [9], [11], [16].

This work focuses on a particular class of attacks referred to as binding based race condition attacks. These are time-of-check-to-time-of-use (TOCTTOU) attacks arising as a result of synchronization flaws. TOCTTOU attacks [5] occur when a system utility (program) checks that a certain condition, say C, is true at time t_1 and performs certain actions subsequently, say at time t_2. At t_2, the utility assumes that C is true. A malicious program can act between t_1 and t_2 and ensure that condition C does not hold true at t_2. Clearly, the interval $[t_1, t_2]$ is a "vulnerable" period. The malicious program exploits the non-atomicity across the various statements of a system utility and acts in a window of opportunity referred to as the *programming interval* [4]. Hence, the malicious program can cause the system utility to perform an action that it originally did not intend to do.

Many system utilities are application-level programs that are intended to do a specific job. In many operating systems such as Unix [1], a trusted user (an administrator) is allowed complete control over the system. In order to provide the delegation of rights to other users when they execute the system utility, the system utility may be provided with higher privileges (through mechanisms such as *setuid*[1]) than the user who executes them. This enables the user executing the system utility to accomplish some useful task.

A common TOCTTOU flaw in the Unix environment occurs when a system utility is run with the effective userid (*euid*) root. The system utility may be a *setuid* utility if it needs higher privileges than that of the user (running the utility) to perform certain operations such as updating a protected file or using structures such as privileged ports. If this *setuid* system utility performs an update on the user's file, an attacker (who has privileges of the user) can use the interval between checking the access permissions (using the real userid) of the file, and updating (as root) the file to make the system utility perform an unintended action.

[1] A *setuid* program is an executable file that has the *setuid* bit set in its permission mode field. When a process executes a *setuid* program, the kernel sets the effective user id fields in the process table and user area (*u area)* to the owner of the file.

An example of this flaw using the *access(2)* and the *open(2)* system calls is shown below. Steps 1 and 4 are from the system utility (running with *setuid* bit set), and steps 2 and 3 belong to a malicious program.

```
1. access (filename)
                                          2. rm (filename)
                                          3. symlink (protectedfile, filename)
4. open (filename, O_CREAT | O_TRUNC | O_WRONLY)
      /* now write to the file */
```

In the intended execution, steps 1 and 4 must be executed with no break in-between. However, the object referred to by *filename* (which is a user owned file) in the *access(2)* system call in step 1 is changed by the malicious program before the subsequent *open(2)* call in step 4. Thus, the *open(2)* system call (in step 4) will open a file which is different from the file whose permissions were checked in the *access(2)* call (in step 1). This happens because the malicious program removes the file and creates a symbolic link to a protected file using the same name, after the execution of the *access(2)* call and before the execution of the *open(2)* call. Clearly, the *open(2)* call in step 4 opens the protected file, assuming that it has checked for the access permissions using the real userid. Since the utility is running with higher privileges (being *setuid*), the *open(2)* call succeeds in performing an unintended operation on the protected file.

This attack was detected in *xterm(1)*, a terminal emulation program [14] and is depicted in Figure 1.

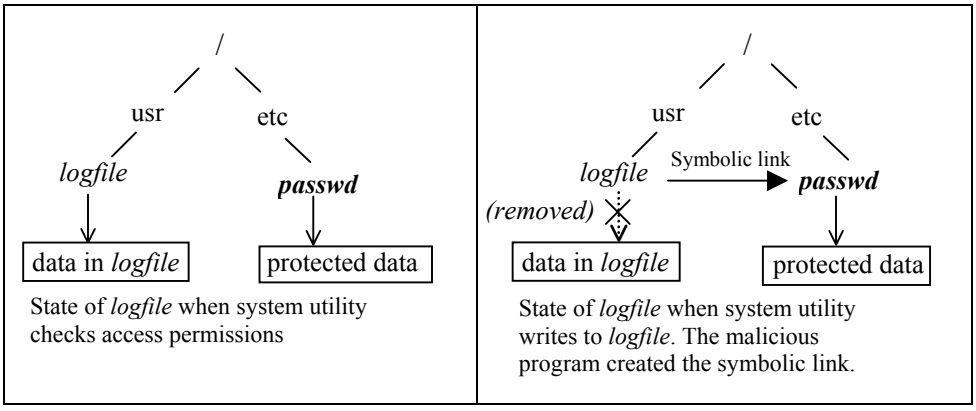

Figure 1: xterm attack

Note that an attacker cannot necessarily exploit all *programming intervals*. It is not always possible for an attacker to change the assumptions in the programming interval due to reasons such as lack of privileges. In order for the attacker to exploit the interval, another key condition referred to as the *environment condition* must also be true [4].

Race condition attacks exploit the lack of mutual exclusion on the objects across the various statements of the system utility. A trivial way to guard against these attacks is to lock the entire file system (or the set of objects accessed by the system utility) during the execution of the system utility. This guarantees that no other program can access the files accessed by the system utility during its execution. However, this approach severely restricts the availability of resources, enforces a sequential order of execution of all programs, which is impractical and undesirable, and prevents numerous safe concurrent executions

In this paper, we devise a unified approach to detect these attacks. Our approach consists of representing the systems calls by a sequence of actions on objects and developing an algorithm to detect the attacks. We show the effectiveness of our approach by examining several known attacks. Our approach may also be used to flag potential unknown attacks.

2. System Model

Consider a computer system running the Unix operating system. In order to access any resources of the system, a process in Unix must go through the system call interface. Since Unix allows concurrent executions, the execution

of a process may not be atomic and may be interleaved by the other processes. The interleaved execution trace of the programs/processes at the granularity of system calls can be obtained using utilities such as *truss(1)* in Unix systems and *strace* in Linux systems. Traces can also be obtained if the system administrator enforces strict logging mechanisms for various processes running on a system.

Unix guarantees indivisible or atomic execution of all system calls. Hence, we model each system call as a sequence of actions. We do not assume atomicity across system calls. Commands such as *mv* are also atomic in Unix, and are therefore considered in the same manner as system calls. Unix has a hierarchical file system structure where the location of a file is defined relative to the root directory (/).

3. The Proposed Solution

We identify a set of objects used in the interleaved execution trace of the system utility and the malicious program. Examples of objects are the filenames and file descriptors used in the Unix file system. Reading the object, writing to the object or changing an object's association are examples of actions that a system call may perform on the objects. Execution trace of the system utility and the malicious program is represented by a sequence of actions on objects in the order of their occurrence in the trace. We then define conflict rules to detect vulnerabilities or attacks.

Note that we analyze the trace of the concurrent execution of the system utility and the malicious program. Concurrent execution allows us to capture the *environment condition*, such as privileges of various files, directories, etc., that the system utility accesses. Hence, it is easy to determine at that time if the file referred to is a file or a directory, or a symbolic link to a file or to a directory. If the malicious program has sufficient privileges to access and modify those objects, the *environment condition* holds true. We assume that several valid interleavings can be obtained by executing the system utility and a malicious program concurrently. Note that we do not consider the interleaving in which all operations of the malicious program are performed before any operation of the system utility, as this situation does not belong to the class of binding based race condition attacks.

We next explain the various terms used in the remainder of the paper.

3.1 Nomenclature Used

Transaction T_i
Partially ordered set (S_i, $<_i$) where S_i is the set of actions of the transaction and $<_i$ is the order in which the actions are executed [17]. In our approach, a transaction refers to an *atomic command*[2] or a system call.

$RS(T_i)$ – Read Set of transaction T_i
For transaction T_i, the Read Set contains all objects read by T_i.

$WS(T_i)$ – Write Set of transaction T_i
For transaction T_i, the Write Set contains all objects written by T_i.

$AS(T_i)$ – Action Set of transaction T_i
For transaction T_i, the Action Set contains all the objects on which T_i performs any of the defined actions. Hence, the $AS(T_i)$ includes $RS(T_i)$, $WS(T_i)$ and all objects whose associations are changed or removed.
SY
SYstem utility

MP
Malicious Program or the attack program

3.2 Set of Objects

The objects that are used for determining the conflicts are:

[2] An atomic command consists of a sequence of statements executed in an indivisible manner. E.g. *mv* command

1. **File, a/b**: *b* refers to the name of the file object, such as a file (including hard links), a directory or a symbolic link; *a* refers to the absolute pathname of the directory that contains *b*.
2. **File descriptor, $fd_{a/b}$**: Refers to the file descriptor entry of the file object *a/b* (as defined above).
3. **Present working directory, pwd**: The pathname of the directory in which the program is executing at that instant of time.

3.3 Set of Actions

A minimal set of actions that act on the objects defined in Section 3.2 is presented in Table 1. The *read* action refers to reading the contents of the object. When the object is a file *a/b*, the *read* action is represented by `r(a/b)`. When file *a/b* is opened, a file descriptor $fd_{a/b}$ is associated with it. Action *map* is used to indicate an association between objects. Opening the file *a/b* is represented by `map(`$fd_{a/b}$`→ a/b)`. When a *read* action is performed on the file descriptor $fd_{a/b}$, it is represented by `r(`$fd_{a/b}$ `→ a/b)`. Similarly, the *write* action refers to creation of an object or modification of its contents. For example, `w(a/b)` denotes a *write* action on the file object *a/b* and `w(`$fd_{a/b}$ `→ a/b)` represents writing to the file *a/b* using its file descriptor $fd_{a/b}$. Creation of a symbolic link from a source object, *a/b*, to a target object, *c/d*, is represented by the action `map(a/b → c/d)`. Action *unmap* removes the association of an object. Closing a file removes the association between the file descriptor and the file and this is represented by the *unmap* action.

The parameter of each action differs based on the type of object it acts on. For example, consider a system call invocation on a symbolic link object *a/b*. If the system call involves a *read* action on the symbolic link, then we represent that action by `r(a/b)`. Checking the permissions of file *a/b*, which involves reading the inode of *a/b*, is represented by `r(a/b)`. If the system call involves a *read* action that follows the symbolic link from *a/b* to the target file *c/d*, we represent this action by `r(a/b → c/d)`. In this case, the inode contents of file *c/d* are read. Table 2 gives a complete list of the different types of actions.

ACTION SYMBOL	ACTION
r	Read
w	Write
map	Map – forms an association between objects
unmap	Unmap – removes the association between objects

Table 1: Set of Action/Operations

ACTION/OPERATION	**MEANING**
r(a/b) *or* w(a/b) *or* unmap(a/b)	Action on the object a/b. Symbolic links, if present, are not followed. Action is performed on the inode of a/b and not on contents of file a/b.
r(a/b → c/d) *or* w(a/b → c/d)	Action follows symbolic links where a/b is a link to c/d. Action is performed on the inode of c/d and not on the contents of the file c/d.
map(a/b → c/d)	Binding of the name a/b to object c/d.
map($fd_{a/b}$→ c/d)	Binding of the file descriptor of file a/b to object c/d where a/b is a symbolic link to file c/d

r($fd_{a/b}$ → a/b) *or* w($fd_{a/b}$ → a/b)	Action on the file descriptor of object a/b. Action (read or write) is performed on the file contents.
map(pwd → a/b)	pwd object is associated with a/b.
w(pwd)	Write action on the pwd object. For example, changing the current directory.

Table 2: Types of Actions/Operations

3.4 Conflict Rule

Consider a system utility performing a *read* or a *write* action by following symbolic links on an object. The system utility assumes that the object and its associations are not changed from the time it last accessed the object. If a malicious program performs a *map* action on that object in this interval, then the system utility may perform unintended actions.

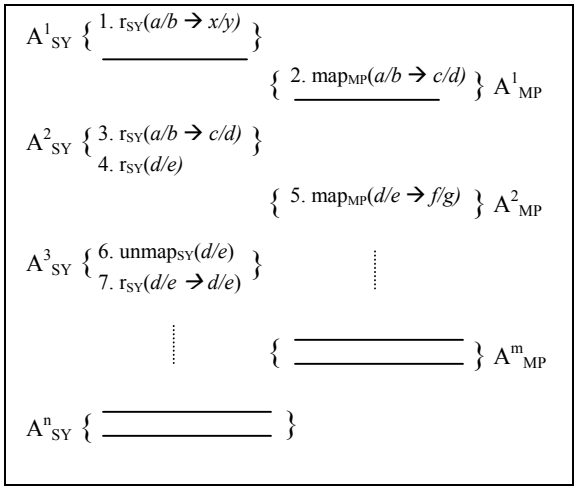

Figure 2: Interleaving of System Utility (SY) and malicious program (MP)

Let *n* sections of the system utility, SY, and *m* sections of the malicious program, MP, be interleaved as shown in Figure 2. Note that SY = [A^1_{SY} .. A^n_{SY}] and MP = [A^1_{MP} .. A^m_{MP}]. Two scenarios are worth noting:

Case 1: When the system utility references the object *a/b* in step 1, it may find that *a/b* is associated with some object *x/y*. Next, the attack program changes the association of *a/b* (from *x/y*) to *c/d* as shown in step 2. The system utility subsequently performs a *read* or *write* action on *a/b* by following symbolic link as shown in step 3. Thus, SY accesses *c/d* instead of *x/y* (that SY expected to access). This scenario represents an attack.

This can be represented more formally by:
$r_{SY}(a/b \rightarrow x/y) < map_{MP}(a/b \rightarrow c/d) < r_{SY}(a/b \rightarrow c/d)$

Several other attacks may exist based on the different actions performed by the system utility. For example, $w_{SY}(a/b) < map_{MP}(a/b \rightarrow c/d) < w_{SY}(a/b \rightarrow c/d)$ also represents an attack (not shown in Figure 2).

Case 2: Steps 4, 5, 6 and 7 in Figure 2 represent a similar scenario on the object *d/e*. However, before SY performs the *read* action (in step 7), SY performs an *unmap* on *d/e* (in step 6). Clearly, *unmap* removes the association of *d/e* to *f/g* made by MP in step 5. Thus, the *read* action in step 7 does not result in an unintended operation.

This can be formally represented by:
$r_{SY}(d/e) < map_{MP}(d/e \rightarrow f/g) < unmap_{SY}(d/e) < r_{SY}(d/e \rightarrow d/e)$

Using the actions defined in Section 3.3, we propose the following conflict rule in order to detect an attack. A conflict occurs if:

i. $\{r_{SY}(a/b) \vee w_{SY}(a/b) \vee map_{SY}(a/b \rightarrow x/y) \vee r_{SY}(a/b \rightarrow x/y) \vee w_{SY}(a/b \rightarrow x/y)\} < map_{MP}(a/b \rightarrow c/d) < \{r_{SY}(a/b \rightarrow c/d) \vee w_{SY}(a/b \rightarrow c/d)\}$ and

ii. there is no $unmap_{SY}(a/b)$ s.t. $map_{MP}(a/b \rightarrow c/d) < unmap_{SY}(a/b) < \{r_{SY}(a/b \rightarrow c/d) \vee w_{SY}(a/b \rightarrow c/d)\}$

where $\vee$ represents the OR condition.

Condition (i) checks if the binding of the name to an object changes between repeated references by the system utility. The system utility may perform some operations based on certain assumptions that no longer hold true. Condition (ii) guarantees that the system utility does not detect false positives[3] if the utility nullifies the actions of the malicious program before performing any further operation on the same object.

We have used the action *map* used in condition (i) as part of $map_{MP}(a/b \rightarrow c/d)$ to denote the association between the name *a/b* and object *c/d*. It is a wrapper built using *read* and *write* operations performed by a system call on file system objects. It is possible that other such wrappers may be defined. The conflict rule presented above can then be extended to include those wrappers, without affecting the model on which the solution is based.

Note that we have used our conflict rule to simulate the concept of mutual exclusion. Mutual exclusion on an object maintains data consistency and provides atomic actions on that object. There are numerous ways to provide mutual exclusion - locks, monitors, semaphores, etc.

3.4.1 Proof of Conflict Rule

Consider the Unix file system to be a black box as shown in Figure 3. The user level processes interact with the file system through handles that are controlled by the system calls [3]. The file system is modeled as a set of objects on which some action is performed. Arrows shown in the figure represent the interaction of the programs (e.g., SY, MP, etc.) with the file system via the handles.

The inwards arrow going into the black box represent writing to an object in the file system. The outgoing arrows represent getting some output from the file system.

Clearly, all interaction with the file system can be captured using the model of the black box. This follows from the assumption that the only way in which user processes can interact with the file system is through system calls. The system calls consist of actions or operations that access the file system, either to read some value from it, or write a value to it.

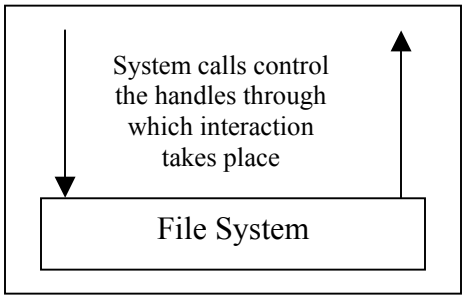

Figure 3: Black box model of file system

Hence, a system call can be represented by a sequence of operations on some object(s) inside the file system. Each of these operations is either a read or write on one or more objects. Thus the basic model captures all interaction with the file system.

Binding based race condition flaws in programs are synchronization flaws in which the binding of a name to an object changes between repeated references. A malicious program acts by exploiting a window of opportunity between the points of execution of a program.

Binding based race condition attacks can be represented by the above model. These attacks involve at least two processes interacting with each other in an inconsistent manner. A binding based race condition attack is of the form:

STEP 1. Operation (object *o*) by A^1_{SY}

STEP 2. Operation (object *o*) by A^1_{MP}

STEP 3. Operation (object *o*) by A^2_{SY}

where A^1_{SY} and A^2_{SY} are the two sections of a system utility and A^1_{MP} is the interleaved section of a malicious program.

[3] A false positive occurs when a "vulnerability" is flagged but it actually does not exist. Case 2 represents a false positive.

The system utility performs an action on the object *o* in step 3, assuming that the binding of the name to the object has not changed since the time it last made a reference to the same object *o* (step 1). However, in A^1_{MP}, the malicious program performs an action on the same object, (thus changing its binding in step 2) before the system utility executes Step 3. Since each of the operations requires interaction with the file system, each operation can be represented using either a read or a write action on some object(s).

Theorem 1: *The proposed conflict rule captures all binding based race condition attacks.*

Proof: The conflict rule flags an attack if the following two conditions hold true across the parts of the system utility and the malicious program:

i. $\{r_{SY}(a/b) \vee w_{SY}(a/b) \vee map_{SY}(a/b \rightarrow x/y) \vee r_{SY}(a/b \rightarrow x/y) \vee w_{SY}(a/b \rightarrow x/y)\} < map_{MP}(a/b \rightarrow c/d) < \{r_{SY}(a/b \rightarrow c/d) \vee w_{SY}(a/b \rightarrow c/d)\}$ and

ii. there is no $umap_{SY}(a/b)$ s.t. $map_{MP}(a/b \rightarrow c/d) < umap_{SY}(a/b) < \{r_{SY}(a/b \rightarrow c/d) \vee w_{SY}(a/b \rightarrow c/d)\}$

A binding based race condition attack occurs when the system utility makes repeated references to an object(s), and falsely acts under certain assumptions that it expects to hold true in the references. A malicious program may exploit the programming interval to change these assumptions, so that the system utility later acts thinking that the assumptions still hold true.

For a race condition attack, there needs to be at least one common object between the references of the system utility and the interleaved reference(s) of the malicious program. Let this object be *b*.

There are three parts to the attack:

Part 1: Initial reference to an object *o* by SY
This operation has to be either a read or write operation on *b*. A wrapper such as "map" also involves read and write actions.

Part 2: The intervening operation by MP on *o*
This operation is also performed on *b*. This operation consists of *read* or *write* action(s). In case of binding based race condition attack, the malicious program changes the binding of a name to an object. Hence, the *map* action (wrapper) of the form map($b \rightarrow c$) involves a sequence of read and write operations such that the binding of the name *b* is changed to *c*.

Part 3: The repeated reference by SY on *o*, assuming that the binding of *b* has not changed
The operation(s) is a read or write operation on *b*. However, since the malicious program changed the binding of the name *b* to *c*, the operation is performed on *c*.

The three parts correspond to condition (i) of the conflict rule. Condition (ii) of the conflict rule exists so that false positives are not detected. This is because the effects of the change in binding of *b* are undone by the system utility before making the subsequent reference to *b*. ■

3.5 Race Condition Attack Detection (RCAD) Algorithm

We propose an algorithm utilizing the conflict rule of Section 3.4. This algorithm can be used to develop a tool that flags race condition attacks in a system utility when an interleaved execution trace of the system utility and a malicious program is available.

RCAD algorithm, shown in Figure 4, proceeds as follows. All *m* sections of the malicious program ($[A^1_{MP} .. A^m_{MP}]$ as shown in Figure 2) are examined sequentially for *map* actions (step 1). For each object *o* involved in a *map* action in any section *i* (A^i_{MP}) of the malicious program (step 2), the algorithm checks if *o* was referenced in a preceding section of the system utility $[A^i_{SY} .. A^1_{SY}]$ (step 3). If *o* is not involved in any action in $[A^1_{SY} .. A^i_{SY}]$, there is no *programming interval* with respect to *o*, and hence, the object *o* is removed from our consideration. If *o* is involved in an action in $[A^1_{SY} .. A^i_{SY}]$, the algorithm finds the first action on *o* performed by the system utility in a subsequent

part [A^{i+1}_{SY} .. A^{n}_{SY}]. If the action is *read* or *write* that follows symbolic links, then the algorithm flags an attack (step 6). If the action found is *unmap* action, then *o* is removed from consideration since the system utility removes the association made by the malicious program before performing a subsequent *read* or *write* action on it (step 5).

The pseudocode of the algorithm follows.

```
STEP 1  for i = 1 to m { // for all the sections of the malicious program
                O = [all objects involved in a map action in A^i_MP ]
                // Multiple occurrences of the same object exist if it is involved in a map action more than once.
STEP 2          for each object o in O {
                        // check if o was referenced in [A^i_SY .. A^1_SY] (the preceding parts of the system utility)
STEP 3                  if there exists a j, 1 ≤ j ≤ i such that o ∈ AS(A^j_SY) then{
                                // AS(A^j_SY) is the Action Set containing all the objects on which a system
                                // utility SY performs an action in the part A^j_SY
                                // check in the subsequent parts of the system utility if a read or a write
                                // action (that follows symbolic links) on  o exists or if unmap action on o exists
STEP 4                          for k = i+1 to n {
                                        examine each action C in A^k_SY (sequentially)
STEP 5                                  if (C is unmap action on o) {
                                                // remove object o from consideration
                                                delete object o from O
                                                goto step 2
                                        } // end of if statement (step 5)
STEP 6                                  else if (C is a read or write action on o performed by SY by
                                        following symbolic links) {
                                                flag conflict;
                                        } //end of else if statement (step 6)
                                } // end of for statement (step 4)
                                // first action has been located so no need to check in the previous parts of SY
                                delete object o from O
                                goto step 2
STEP 7                  } else { // no conflict involving o in A^i_MP exists
                                delete object o from O
                                goto step 2
                        } //end of else for statement (step 7)
                } // end of  for statement (step 2)
        } // end of  for statement (step 1)
```

Figure 4: RCAD Algorithm

In our approach, mutual exclusion is violated if there is a *map* action on an object *o* by the malicious program between repeated references to *o* such that the system utility subsequently acts on *o* under certain assumptions (based on the earlier reference by the system utility) that no longer hold true. Thus, the critical section for the system utility is defined for each object used by the system utility.

Complexity Analysis:
The algorithm has a running time of $O(X * Y)$ where X is the total number of map actions in the malicious program and Y is the total number of actions in the system utility. Detailed analysis is presented in [9].

Theorem 2: RCAD algorithm detects all binding based race condition attacks without false positives.

Proof: Please see [9].

4. Representation of System Calls/Commands With Their Sequence Of Actions:

We consider each system call or atomic command as a transaction and represent it by a sequence of actions. Table 3 lists the various system calls and commands and their respective sequences of actions. We next explain three entries of Table 3.

Example 1: *stat(2)* is a system call that returns detailed information about a file *a/b* where *a/b* is the absolute pathname of the file *b* (located in the directory *a*). Hence, the actions involved are checking the entire hierarchy *a* for search permissions, and *b* for read permissions. Both of these involve *read* actions on *a* and *a/b*. Furthermore, *stat(2)* follows symbolic links. Hence, if *a/b* is a symbolic link to a file *c/d*, then the corresponding sequence of actions would be {r(a),r(a/b → c/d)}[4]. If any component of *a* includes a symbolic link, the link is traversed.

Example 2: Consider the system call chdir(a/b) (*a* is the pathname and *b* is a directory). Let *pwd* be an object that refers to the present working directory as defined in the Unix file system. *chdir(2)* first performs r(a) to check for search permissions of the entire directory path in which the target directory *a/b* is located. r(a/b) refers to the checking of permissions for reading/accessing the target directory. map(pwd → a/b) denotes the fact that the present working directory is changed to *a/b* as a result of *chdir(2)*. Using the same approach, if *a/b* above is a symbolic link to a target directory *c/d*, the sequence of actions is {r(a), r(a/b → c/d), map(pwd → c/d)}.

Example 3: Creation of a symbolic link *a/b* to *c/d* where *c/d* can be either a file or a directory uses the *symlink(2)* system call. r(a) involves checking for search permissions in the path list and write permissions in the directory in which the link is to be created. w(a/b) indicates the creation of a new object *a/b*. map(a/b → c/d) captures the association between *a/b* and *c/d*. The corresponding sequence of actions for *symlink(2)* is hence {r(a) w(a) w(a/b) map(a/b → c/d)}. If a subsequent system call that follows the symbolic links involves reading *a/b* and the association has not been changed since the previous association by the *symlink(2)* system call, the corresponding action will be r(a/b → c/d).

SYSTEM CALLS/ COMMANDS	**RELATED ACTIONS**
access(a/b) uses real userid b = file[5]	{r(a), r(a/b)}
chdir(a/b) [pwd → a/b] b = dir	{r(a), r(a/b), map(pwd → a/b)}
chdir(a/b) [pwd → a/b] b = symlink to c/d where d = dir	{r(a), r(a/b → c/d), map(pwd → c/d)
chmod(a/b) b = existent symlink to c/d d = file	{r(a), r(a/b → c/d), w(a/b → c/d)}
new usage of **chown**(a/b) b = existent symlink to c/d d = file	{r(a), r(a/b → c/d), w(a/b → c/d)}
new usage of **lchown**(a/b) (or) old usage of **chown**(a/b) b = existent symlink to c/d d = file	{r(a), r(a/b), w(a/b)}
close($fd_{a/b}$) b = file	{unmap($fd_{a/b}$)}
fstat($fd_{a/b}$)	{r($fd_{a/b}$ → a/b)}
mv(a/b, c/d) b = existent dir d = existent dir	{r(a), r(a/b), r(c), r(c/d), w(a), unmap(a/b) w(c/d), w(c/d/b), w(c/d/b/..), map(c/d/b/.. → c/d)}

[4] {} are used to denote a sequence of actions.
[5] File refers to a regular file, and directory refers to a directory file as defined in the Unix file system.

open(a/b, O_TRUNC, O_WRONLY) b = symlink to c/d where d = file	{r(a), r(a/b → c/d), w(a/b → c/d), map($fd_{a/b}$ → c/d)}
open(a/b, O_CREAT, O_WRONLY) or **creat**(a/b) b = non-existent	{r(a), w(a), w(a/b), map($fd_{a/b}$ → a/b)}
open(a/b, O_RDONLY) b = file	{r(a), r(a/b), map($fd_{a/b}$ → a/b)}
read($fd_{a/b}$, buf, ctr) b = file	{r($fd_{a/b}$ → a/b)}
rename(a/b, c/d) b = existing file d = file	{r(a), r(a/b), r(c), r(c/d), w(a), unmap(a/b), w(c), w(c/d)}
rm(a/b) b = file	{r(a), r(a/b), w(a), unmap(a/b)}
rmdir(a/b) b = empty dir	{r(a), r(a/b), w(a), unmap(a/b)}
stat(a/b) b = file	{r(a), r(a/b)}
stat(a/b) b = symbolic link to c/d	{r(a), r(a/b → c/d)}
symlink(c/d, a/b) b = non-existent file d = file / directory	{r(a), w(a), w(a/b), map(a/b → c/d)}
unlink(a/b) b = file	{r(a), r(a/b), w(a), unmap(a/b)}
write($fd_{a/b}$, buf) b = file	{r($fd_{a/b}$ → a/b), w($fd_{a/b}$ → a/b)}

Table 3: Representing system calls / atomic commands with their respective sequences of actions. Atomic commands are marked in bold italics.

Table 3 lists the scenarios where the object involved in an action is a symbolic link to a file, symbolic link to a directory, a file or a directory. These scenarios are essential to preserve the semantics of the various system calls in the Unix file system.

5. Detection of Known Attacks Using Our Algorithm

We next describe how RCAD algorithm detects several known binding based race condition attacks in the Unix system.

5.1 Binmail attack:

Binmail is a *setuid* program that is used to deliver mail by writing it to the user's mailbox. The system utility first creates a temporary file (unlinking the file if it is already present) in a world writeable directory, and then opens the temporary file to write the user's mail to it. It uses the temporary file to update the mailbox file of the user. If the user can guess the name of the temporary file, the user can remove the temporary file and point it to a protected file before the system utility writes the mail to it. An attacker can exploit this vulnerability window to write to any protected file in the system [2].

As shown in Table 4, binmail creates a temporary file *a/s* to append the user's mail and later renames *a/s* as the user's mailbox file. Between steps 3 and 6 of binmail, an attacker can create a symbolic link by the same name *a/s*, pointing the link to a protected file such as the password file */etc/passwd*. Since the *open(2*) call follows symbolic links, it will open the password file and write to it. Note that since *mktemp(3C)* returns a unique file name and does not have any useful action on the file system, *mktemp(3C)* is not represented by a transaction or a sequence of actions.

BINMAIL SYSTEM UTILITY	MALICIOUS PROGRAM
1. s = mktemp() 2. stat(a/s) 3. if exist(a/s) unlink(a/s)	
	4. rm(a/s) 5. symlink("/etc/passwd", a/s)
6. $fd_{a/s}$ = open(a/s, O_CREAT | O_TRUNC | O_WRONLY) 7. write($fd_{a/s}$, …)	

Table 4: Binmail attack description

BINMAIL SYSTEM UTILITY	MALICIOUS PROGRAM
1. no action 2. {r(a), r(a/s)} 3. {r(a), **r(a/s)**, w(a), unmap(a/s)} A^1_{sy}	
	4. no actions, a/s already deleted 5. {r(a), w(a), w(a/s), **map(a/s → /etc/passwd)**} A^1_{MP}
6. {r(a), **r(a/s → /etc/passwd)**, w(a/s → /etc/passwd), map($fd_{a/s}$ → /etc/passwd)} 7. {r($fd_{a/s}$ → /etc/passwd), w($fd_{a/s}$ → /etc/passwd)} A^2_{sy}	

Table 5: Sequence of actions for binmail attack

Using our approach, the various system calls in Table 4 are represented by sequences of actions and the result is shown in Table 5. Initially, since *a/s* is a file, there is `r(a/s)` at step 2 for obtaining information about *a/s* using *stat(2)* call. Between steps 3 and 6, the attacker creates a symbolic link from *a/s* to a protected file. The *open(2)* call in step 6 follows the symbolic link, opens the protected file, and writes to the protected file in step 7.

We next show how RCAD Algorithm detects the above attack. The algorithm sequentially examines each segment of the malicious program (A^1_{MP}) in the trace. A^1_{MP} consists of the sequence of actions in step 5 in Table 5. It finds a *map* action involving object *a/s* in the malicious program. The algorithm then searches for a reference to *a/s* in preceding segment (A^1_{sy}) of the system utility. A^1_{sy} consists of sequences of actions in steps 2 and 3. The first reference to *a/s* is found in step 3 in the *read* action `r(a/s)`. Next, the algorithm searches the subsequent sections of the system utility (A^2_{SY}) for either a *read/write* action on *a/s* that follows symbolic links or an *unmap* action on *a/s*. It finds the *read* action `r(a/s → /etc/passwd)` in step 6 and flags an attack. The actions that match the conflict rule are marked in bold in Table 5 in lines 3, 5, and 6.

It is possible to obtain another interleaving of the same system calls by re-arranging the various calls. We next show another interleaving in which there is no binding based race condition attack.

5.2 Binmail non-attack:

BINMAIL SYSTEM UTILITY	MALICIOUS PROGRAM
1. s = mktemp() 2. stat(a/s)	
	3. rm(a/s) 4. symlink("/etc/passwd", a/s)
5. if exist(a/s) unlink(a/s) 6. $fd_{a/s}$ = open(a/s, O_CREAT | O_TRUNC | O_WRONLY) 7. write($fd_{a/s}$, …)	

Table 6: A non-attack interleaving description

BINMAIL SYSTEM UTILITY	MALICIOUS PROGRAM
1. no action 2. {r(a), r(a/s)} A^1_{sy}	
	3. {r(a), r(a/s), w(a), unmap(a/s)} 4. {r(a), w(a), w(a/s), map(a/s → /etc/passwd)} A^1_{MP}
A^2_{sy} 5. {r(a), r(a/s), w(a), unmap(a/s)} 6. {r(a), w(a), w(a/s), map($fd_{a/s}$ → a/s)} 7. {r($fd_{a/s}$ → a/s), w($fd_{a/s}$ → a/s)}	

Table 7: Sequence of actions for binmail non-attack of Table 6

Table 6 gives an alternate interleaving. The corresponding sequences of actions are shown in Table 7. In this case, there is no attack since *unlink(2)* in step 5 removes the symbolic link before creating the temporary file. RCAD algorithm finds a *map* action on object *a/s* in the only section (A^1_{MP}) of the malicious program. It searches A^1_{SY} (of the system utility) for a reference to *a/s* and finds a *read* action `r(a/s )` in step 2. In the subsequent part (A^2_{SY}) of the system utility, the `unmap(a/s)` action appears first. Hence the algorithm does not flag an attack and does not detect this false positive.

5.3 Rdist attack:

Rdist(1) (Remote File Distribution Program) [15] is a utility used in Unix to maintain identical copies of files over multiple hosts. Updates on the master copies are propagated to all the clients using the *rcmd(3)* interface. Since privileged ports are used for authentication in the *rcmd(3)* routine, *rdist(1)* runs with root privileges [9].

Table 8 summarizes the sequence of system calls that *rdist(1)* makes to update the file */user/data*. In step 2, *rdist(1)* creates a temporary file */user/rdista768* in the directory where */user/data* resides. It writes the new contents of the file to the temporary file and closes it in steps 3 and 4. In steps 7 and 8, it uses *chown(2)* and *chmod(2)* to change the owner and the mode of the temporary file. In the last step, *rdist(1)* renames the temporary file to */user/data*.

Rdist(1), in 4.3 BSD Unix and other variants of Unix, has a race condition flaw that relates to the way it updates the file as well as to the semantics of the *chown(2)* and *chmod(2)* system calls. The flaw enables an attacker to illegally acquire root privileges. Old usage of *chown(2)* does not follow symbolic links while *chmod(2)* follows symbolic

links. If the malicious program creates a symbolic link from the temporary file to */bin/sh* between steps 4 and 7 of the system utility, then *chmod(2)* changes the mode of the system shell to set its *setuid* bit, as it follows symbolic links from */user/rdista768*. Therefore the attacker can obtain root privileges by invoking */bin/sh*.

RDIST SYSTEM UTILITY	**MALICIOUS PROGRAM**
	1. execve("/usr/ucb/rdist");
2. $fd_{/user/rdista768}$ = creat("/user/rdista768"); 3. write($fd_{/user/rdista768}$, …); 4. close($fd_{/user/rdista768}$);	
	5. rename("/user/rdista768", "/user/tmp"); 6. symlink("/bin/sh", "/user/rdista768");
7. chown("/user/rdista768", owner); 8. chmod("/user/rdista768", pmode); 9. rename("/user/rdista768", "/user/data");	

Table 8: rdist attack description

Assuming that the temporary file */user/rdista768* does not exist, the attack described above is represented by a sequence of actions as shown in Table 9. Initially the temporary file */user/rdista768* is created by the system utility. In step 5, the malicious program renames the temporary file. In step 6, the malicious program creates a symbolic link with the name */user/rdista768* to the system shell */bin/sh*. In step 8, *chmod(2)* of the system utility follows symbolic links and the mode of */bin/sh* is changed. RCAD Algorithm flags the attack using the actions marked in bold in steps 2, 6, and 8.

RDIST SYSTEM UTILITY	**MALICIOUS PROGRAM**
	1. no actions
2. {r(/user), w(/user), **w(/user/rdista768),** map($fd_{/user/rdista768}$ → /user/rdista768)} 3. {r($fd_{/user/rdista768}$ → /user/rdista768), w($fd_{/user/rdista768}$ → /user/rdista768)} 4. {unmap($fd_{/user/rdista768}$)}	
	5. {r(/user), r(/user/rdista768), r(/user), r(/user/tmp), w(/user), unmap(/user/rdista768), w(/user), w(/user/tmp)} 6. {r(/user), w(/user), w(/user/rdista768), **map(/user/rdista768 → /bin/sh)**}
7. {r(/user), r(/user/rdista768), w(/user/rdista768)} 8. {r(/user), **r(/user/rdista768 → /bin/sh)**, w(/user/rdista768 → /bin/sh)} 9. {r(/user), r(/user/rdista768), r(/user), r(/user/data), w(/user), unmap(/user/rdista768), w(/user), w(/user/data)}	

Table 9: Sequence of actions for rdist attack

5.4 xterm attack:

The *xterm(1)* attack is explained in Section 1. We now show the representation of the attack in Table 11 and flag the attack using our algorithm.

XTERM SYSTEM UTILITY	MALICIOUS PROGRAM
1. access(a/logfile)	
	2. rm(a/logfile) 3. symlink(c/target, a/logfile)
4. if yes, $fd_{a/logfile}$ = open(a/logfile, O_APPEND);	

Table 10: xterm attack description

XTERM SYSTEM UTILITY	MALICIOUS PROGRAM
1. {r(a), **r(a/logfile)**}	
	2. {r(a), r(a/logfile), w(a), unmap(a/logfile)} 3. {r(a), w(a), w(a/logfile), **map(a/logfile → c/target)**}
4. {r(a), **r(a/logfile → c/target)**, w(a/logfile → c/target), map($fd_{a/logfile}$ → c/target)} followed by some w($fd_{a/logfile}$ → c/target)	

Table 11: Sequence of actions for xterm attack

Let *logfile* be in the directory *a*. Since *a/logfile* is a file, r(a/logfile) in step 1 as part of the *access(2)* call corresponds to checking for write permissions of the file with the real userid. The race condition exists between steps 1 and 4 in Table 10. The attacker can create a symbolic link by the same name *a/logfile* to a protected object *c/target*. The subsequent *open(2)* appends the data to the protected file. RCAD Algorithm flags the attack using the actions marked in bold in Table 11 at lines 1, 3, and 4.
Other known binding based race condition attacks include *passwd(1)*, *sendmail(1)* and *rm(1)*. Our algorithm has been used to flag these and the results appear in [9].

6. Conclusion

6.1 Summary of our Contributions

In this paper, we have presented a unified approach for detecting binding based race condition attacks. Although rare and difficult to execute, these attacks are extremely critical and have been a major work of analysis in recent times. Our work informs the programmer or system administrator if the system utility has any period of vulnerability in an interleaved execution trace, which could be exploited by a given malicious program resulting in unintended behavior or output. It also helps the system administrator to flag the attack from the logs maintained on a system. With the available trace of system calls representing all events taking place during execution, binding based race condition attacks can be detected without flagging false positives. We have primarily considered the Unix file system. Our approach can be extended to other operating systems/file systems.

Note that our approach is not complete. Determining whether an arbitrary program has a generic vulnerability appears to be an extremely hard problem. Hence, we need to analyze a restricted set of problems. Most often, this restriction is specified by the desired security properties of the system utility. We have proposed in this paper an algorithm to determine if a program satisfies some of these properties.

6.2 Comparison with Related Work

Our work was inspired by the work of Bishop and Dilger [4], Cowan, et. al. [6], and Ko and Redmond [11]. We have used the known binding based race condition attack descriptions available in [1], [2], [12], [15].

Bishop and Dilger's seminal paper [4] formally defines the notion of a TOCTTOU flaw and presents a methodology for detecting race condition attacks using static analyzers. The static analysis tool requires source code and a human analyst to analyze the output of the tool. Bishop and Dilger also present theorems showing that detecting TOCTTOU flaws statically is undecidable and discuss the possibility of a dynamic detector.

Raceguard [6] is a kernel enhancement that attempts to stop attacks with low performance costs. Raceguard does this with sufficient speed and precision so that the attack can be halted before it takes effect. Raceguard has been implemented, tested, and its performance has been measured. However, it does not defend against all types of race condition attacks.

Solar Designer [18] is a solution to race condition attacks which proposes that *setuid* programs should not follow symbolic links in directories with the sticky bit set unless the root owns them. Rather than attacking the "race" aspect, the Openwall enhancement to the Linux kernel attacks the inclination of privileged programs to follow symbolic links. Under Openwall, programs that are *setuid* root will not follow symbolic links to a file in a directory in which the sticky bit is set (e.g. /tmp) because most race condition attacks involve temporary file race vulnerabilities and a *setuid* program. The use of "sticky bit" minimizes the compatibility problems imposed by this approach since symbolic links are useful, and temporary files are largely created in the /tmp file system. While this approach is effective in many cases, it obtains mixed results as some programs (wrongly) can create temporary files in other file systems, e.g. the current working directory.

In comparison, we examine the execution trace of the system utility and the malicious program at the system call level to detect binding based race condition attacks. Our work does not necessitate the presence of the source code of the system utility or the malicious program. RCAD algorithm requires the interleaved execution trace of the system utility and the malicious program, which can be obtained from log files. No human intervention is required for analyzing the results and the algorithm does not detect any false positives.

6.3 Directions for Future Work

We are currently developing a proof of concept prototype. This prototype will enable us to provide a tool for dynamic detection of binding based race condition attacks. We are also working on extending our approach to other attacks. The results of our work, when complete, will appear elsewhere.

7. References

[1] 8LGM, "[8lgm]-Advisory-7.UNIX.passwd.11-May-1994," available from *fileserv@bagpuss.demon.co.uk* (May 1994).

[2] 8LGM, "[8lgm]-Advisory-5.UNIX.mail.24-Jan-1992, " available from *fileserv@bagpuss.demon.co.uk* (Jan 1992).

[3] Bach, M. J., "The Design of the Unix Operating System," Prentice-Hall, Englewood Cliffs, NJ, 1987.

[4] Bishop, M., and Dilger, M., "Checking for Race Conditions in File Accesses," *In The USENIX Association, Computing Systems*, pages 131--152, Spring 1996.

[5] Carl, Landwher, et al., "A taxonomy of computer program security flaws," Technical report, Naval Research Laboratory, November 1993.

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

[7] Denning, D., "An Intrusion Detection Model," *IEEE Transactions on Software Engineering* **SE-13** (2) pp. 222–232 (Feb. 1987).
[8] Garvey, T. D., and Lunt, T. F., "Model-Based Intrusion Detection," *Proceedings of the Fourteenth National Computer Security Conference*, pp. 372–385 (Oct. 1991).
[9] Goyal, B., Sitaraman, S., and Venkatesan, S., Technical Report UTDCS-02-03, Feb. 2003, *Department of Computer Science*, University of Texas at Dallas, Richardson, TX, USA, available at http://www.utdallas.edu/~venky/publications/race.pdf
[10] Ko, C., and Cheuk, W., "Execution Monitoring Of Security-Critical Programs In A Distributed System: A Specification-Based Approach," Ph.D. Dissertation, University Of California, Davis 1996.
[11] Ko, C., and Redmond, T., "Noninterference and Intrusion Detection," *2002 IEEE Symposium on Security and Privacy*, May 12 - 15, Berkeley, California.
[12] Recursive directory removal race condition, available at *http://mail.gnu.org/pipermail/bug-fileutils/2002-March/002433.html*
[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] *Sendmail Vulnerability*, CERT Advisory CA-90:01 (Jan. 1990), available from *cert.org* via anonymous *ftp.*
[16] Sherif, J. S., and Dearmond, T. G., "Intrusion Detection: Systems and Models," *Eleventh IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'02)*, June 10 - 12, 2002 Pittsburgh, Pennsylvania, USA.
[17] Singhal M., and Shivratri N.G., "Advanced Concepts in Operating Systems," McGraw-Hill, ISBN: 007057572X, 1994.
[18] "Solar Designer," Root Programs and Links, available at *http://www.openwall.com/linux/*
[19] Sun Microsystems, Man pages: *Rdist – Remote File Distribution Program*.