Preventing Race Condition Attacks on File-Systems

Prem Uppuluri
Dept. of Computer Science
University of Missouri -
Kansas City, MO, 64110
uppulurip@umkc.edu

Uday Joshi
Dept. of Computer Science
University of Missouri -
Kansas City, MO, 64110
umjzn9@umkc.edu

Arnab Ray
Dept. of Computer Science
State University of New York at
Stony Brook, NY, 11790
arnabray@cs.sunysb.edu

ABSTRACT

Race condition attacks occur when a process performs a sequence of operations on a file, under the assumption that the operations are being executed "atomically". This can be exploited by a malicious process which changes the characteristics of that file between two successive operations on it by a victim process, thus, inducing the victim process to operate on a modified or different file. In this paper we present a practical approach to detect and prevent such race condition attacks. We monitor file operations and enforce policies which prevent the exploitation of the temporal window between any consecutive file operations by a process. Our approach does not rely on knowledge of previously known attacks. In addition, our experiments on Linux demonstrated that attacks can be detected with false alarms of less than 3% with performance overheads less than 8% of the processes execution time.

Keywords

Security, Race conditions, system calls

1. INTRODUCTION

Race condition attacks on file systems exploit the following flaw in process behavior: when a process performs a sequence of operations on a file, it assumes that the file does not change between any two successive operations. An attack can thus manifest itself by changing the file during the temporal interval between two successive operations on it by a victim process. We call the temporal window as a *race window*. Potential damage can be caused through the following two scenarios: (a) the victim process operates on the changed file, unwittingly, causing damage, or, (b) information is leaked/written from/into the file illegally if the victim's operations allow the attacker to get permissions on the file during the race window. A number of such attacks in the former category have been reported [6, 16]. While to our knowledge based on the CERT advisories, no attack in the later category has been reported, it has the potential to be exploited.

A classic example of a race condition attack is the attack on the program `Binmail` [14, 12] in UNIX. `Binmail` is a *setuid-to-root* program – a program which can be invoked by an ordinary user to perform some operations with super-user privileges. The `Binmail` program when delivering a mail to a user, checks for the existence of a temporary file using a `stat` system call. If the file exists, it deletes it. Next, it creates the file and writes the mail into it, using the `open` system call with one of `O_CREAT` (create the file if it does not exist) or `O_TRUNC` (truncate the file if it exists and is not empty) or `O_WRONLY` (open the file for writing if it exists and is empty) modes. An attacker can exploit the time interval between the `stat` call and the `open` call as shown in Figure 1. This attack induces `Binmail` to truncate the password file, `/etc/passwd`.

Previous research has usually focused on detecting only known race condition attacks [7, 10, 14] and/or do not detect attacks in scenario (b) [13, 21]. In this paper, we present a practical approach to detect and prevent both known and unknown race condition attacks. The core of our approach is in identifying, *non-commutable* file operations. Specifically, when a process P is performing two consecutive file operations O_1 and O_2 on a file, then any operation O' on that file by some other process, between O_1 and O_2, is said to be non-commutable with O_1 if O' *changes the filename-space* that results in damage either due to scenarios (a) or (b). The salient contributions of our paper are:

- Characterizing commutability of file operations in terms of independent sets [2] calculated on the Labeled Transition System (LTS) defined by the system-call behavior of the processes. This is achieved by constructing the system-call graph of a process, and then by providing a characterization of "safety" in the framework created. We use this characterization to develop policies. (Section 4 and Section 5).
- Addressing the practical issues in attack detection. In particular, we discuss the need to consider the associations between filename, inode number, file attributes and content when considering race conditions. In addition we discuss the issues related to false alarms and efficiency of detection. (Section 2 and Section 6).
- Demonstrate the practical feasibility of our approach.

Steps	Subject	Operation	Comment
1.	`binmail`	`tmp_file = mktemp()`	Generate unique temp filename
2.	`binmail`	`stat(tmp_file)`	Check for the existence of the file
3.	`binmail`	if file_exists(`tmp_file`) `unlink(tmp_file)`	if the file exists delete it
4.	`attacker`	`link -s /etc/passwd tmp_file`	attacker creates a symbolic link from `tmp_file` to a privileged file
5.	`binmail`	`open(tmp_file, O_CREAT|O_TRUNC|O_WRONLY)`	open resolves the symbolic link `tmp_file` to `/etc/passwd` and truncates the file

Figure 1: Attack on `binmail` program

Specifically we show through experimentation the effectiveness of our approach in attack detection as well as its efficiency and ease of use (Section 7)

In the next section we discuss the practical issues in detecting attacks.

2. PRACTICAL ISSUES

We motivate this section with the following examples:

- *Simple* `access-open` *attack.* This attack exploits the race window between the following two successive operations of a `setuid-to-root` process P: checking user's permissions to create/write a file Y using the `access` system call, and then opening the file Y using an `open` call for writing. Setuid-to-root programs perform these two operations to ensure that when a user wants to write into a file, the user has the permissions to do so. Without this check, the setuid-to-root program can be used to open any file writing, since it runs with administrator (root) privileges. An attacker can exploit this race window by running the setuid-to-root program. After the process executes the access call, to check if the attacker has permissions to write Y, the attacker deletes Y. She then creates a new symbolic link Y to some privileged file X. When the setuid-to-root program opens file Y as part of its second operation, it resolves the symbolic link Y and ends up opening X, causing the program to write into X. The attacker has exploited the race window with the observation that any user can create a symbolic link to a privileged file and the `open` system call does not check for the permissions of the user running the program – rather it uses the effective user id of the program, which is, root.

- *Directory redirection attack* (Figure 2): In this attack the attacker replaces the directory in which a file was being accessed by a setuid-to-root program. When the program opens the file, it is actually referring to a different file with the same filename.

- *filelogger attack* (Figure 3): `filelogger` is a privileged program which appends messages to a logfile. If the logfile reaches the maximum size (i.e., it fills the filesystem), the process moves it to another filesystem and creates a new logfile. The process checks the file size (using `stat`) before opening it (`open`) for writing. An attacker can exploit the time interval between `stat` and `open` by appending a large message to the file, forcing it to run out of space. The `open` system call then fails, possible causing loss of key log-messages.

- *Readfile attack*: (Figure 4) Consider a privileged program which opens a file (using `open`) and then changes the files permissions (using `chmod`), to ensure that it is not read by any other program. An attacker can read the information between the `open` and `chmod`, thus causing privileged *information leak*.

The above examples illustrate some key lessons:

- *Changes to filesystem name-space can manifest in different ways*: Files in UNIX are characterized by a unique id (*inode number*), one or more filenames, attributes such as permissions and the data blocks which store the file content. Hence, a *change to the filesystem* can be defined as a change in the association between either (a) inode number and file name, or (b) the inode number and file data or attributes.

- *An operation's malicious intent is application specific*: Defining if a particular operation during the race window constitutes an attack is dependent on application semantics. For instance, if two editors are simultaneously editing a file, it may be an error but not necessarily an attack, unlike in the `filelogger` attack. Similarly, two or more applications reading a file concurrently do not constitute an attack, unlike in the case of the readfile attack. Therefore, any approach to detect race condition attacks must consider application specific semantics. This is a key issue not only to detect stealthy attacks but also to reduce false alarms.

In addition to the above requirements, to detect attacks pro-actively, i.e, before they cause damage, the approach has to be efficient. This is a challenge because a race can be between any two file operations between any two processes – making the number of operations/files to be considered very high.

3. OVERVIEW OF OUR APPROACH

In our approach we define the notion of commutability. We then develop security policies which capture this notion. Policies are expressed as patterns over sequences of file related system calls and their arguments. The choice of using system calls is based on the observation that, *all* file operations manifest themselves as system calls. This observation is valid in most cases, since, the only way to modify files is (a) using systemcalls on UNIX or (b) by covert channels. Files can modified by covert channels, e.g., by mapping the file into memory using the `mmap` system call and then modifying the memory by directly modifying the `/dev/mem` file – an image of the main memory. We do not address covert channels in this paper.

Steps	Subject	Operation	Comment
1.	setuid program	access(file, W_OK)	determine if it is ok for the real user to write into file. Assume file is in /tmp/a/b/c/file directory
2.	attacker	replace(/tmp/, /etc)	replace the /tmp directory with some other directory
3.	setuid program	open(file, O_RDWR)	opens the file, but this refers to a different file since the directory has been changed

Figure 2: Abstract attack based on directory relocation

Steps	Subject	Operation	Comment
1.	`filelogger`	`lf=stat(log_file)`	check the size of log_file
2.	`filelogger`	`if (size(lf)<MAX_ALLOWED+bytes)`	file size should be within some system defined limit
3.	attacker	`append(log_file, HUGE_DATA)`	append data to make log_file go over the limit
4.	`filelogger`	s = write(log_file, buf, bytes)	attempt to write the buffer, but s = no file space !

Figure 3: Attack on `filelogger`

System call policies are specified using a high-level policy language called, behavior modeling specification language (BMSL) [20, 23]. BMSL extends the familiar pattern-matching constructs of regular expressions (regexs) to the domain of events (in this paper system calls) with arguments.

Two types of policy specifications are developed in our approach: generic and application specific. Generic policies capture commutability relation between file operations that are applicable to most of the processes. On the other hand, application specific policies are tailored to the semantics of specific applications. Specifications in BMSL are compiled into efficient enforcement mechanisms called detection engines (DEs). The DEs match the system calls with the policies in the specification using an efficient pattern matching algorithm [22] and trigger a reaction when an attack is detected.

The runtime system is shown in Figure 5. It includes the following: a `configuration file` which allows a user to associate specific applications with their specific DEs (if applicable), a kernel runtime system (`KRT`), which intercepts all the system calls both at the time of entry into the kernel and at the time of their return, a base kernel detection engine (`KDE`) whose functionality is to keep track of files being operated on by the processes, and, DEs which enforce the security policies. In this figure, each process $P_1, \cdots P_n$ is monitored by a DE DE_n – the enforcement mechanism for the generic specification. Certain processes such as $P1$ are also being monitored by one or more application specific policies (DE_1).

4. DEFINING COMMUTABILITY

A *process graph* is an abstract representation of a process's externally observable behavior. In general, in a multithreaded operating system each process executes in its own space and makes a series of system calls. [For the purpose of our discussion all system calls are file operation related]. A system call graph is a process graph or a Labeled Transition System (LTS) that encodes sequences of system calls made by the process in terms of states and transitions – with each state of the system call graph representing a "snapshot" [valuation to variables] of the process before (after) executing a system call transition while the labels are simply the system call names.

When a system consists of multiple process graphs operating in parallel, then the set of traces of the entire system may be looked upon as the interleaving of the individual traces of the process graphs. This brings us to the issue of characterizing file commutativity in the LTS framework for representing system call behavior of processes. In order to do this we adapt the concept of *independent* actions.

Let α be a high privileged file operation [i.e., it is performed by a privileged process] and β be a low privileged file operation [i.e., it is performed by a normal process]. The two actions α and β are independent if for all states s: (1) if α is enabled at a state s then after the execution of α it should be possible for β to be enabled; (2) if β is enabled at a state s then after the execution of β it should be possible for α to be enabled and (3) if both α and β are enabled at the same state s then the result of executing α first and β second would be *equivalent* to the result of executing β first and α next. In other words, α and β cannot enable or disable each other and their order is irrelevant with respect to a particular kind of *equivalence*. For our purpose, this equivalence is defined with respect to identical file-system name spaces. For reasons of space we do not provide the formal definition of the equivalence relation but merely provide the intuition that two states are deemed to be equivalent if they have the same "file-system view" that is they cannot be distinguished by the file system. [Note two equivalent states may provide very different valuations to non-file system parameters]

The automata theoretic characterization of file commutability is significant because it lifts the purely mathematical concept of commutability to a trace-based finite state domain. In other words we transform the denotational characterization of commutability to an equivalent operational characterization which in turn enables us to formulate implementation polices in a more natural fashion. We use this commutability definition to develop security policies using system calls.

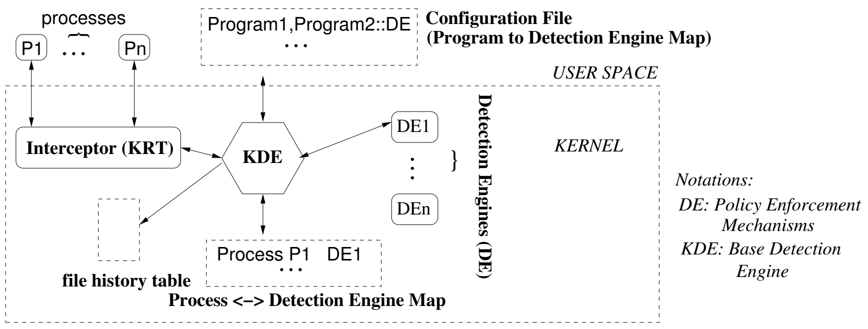
5. SECURITY POLICIES

BMSL policies is used to specify the commutability relationshiops. Associated with them are reactions which are triggered when actual file related system calls match the policies.

As a first step to policy development (to make it easier to

Steps	Subject	Operation	Comment
1.	`readfile`	`lf=chmod(privX,777)`	change permissions to world read/write `privX`
2.	`attacker`	`open and read(privX)`	open and read the filefile
3.	`readfile`	do something and `chmod(privX, 700)`	remove permissions

Figure 4: Attack on `readfile`



processes
P1
Pn
Program1,Program2::DE
Configuration File (Program to Detection Engine Map)
USER SPACE
Interceptor (KRT)
KDE
DE1
DEn
Detection Engines (DE)
KERNEL
Process P1 DE1
file history table
Process <-> Detection Engine Map
Notations:
DE: Policy Enforcement Mechanisms
KDE: Base Detection Engine

Figure 5: Runtime System Architecture

handle the large number of system calls) we group system calls and associate abstract events with each group. Policies are written using the abstract events. Use of abstract events rather than low-level system calls in policies allow them to be portable. This is because the system call API among UNIX variants mostly differ only in the type and number of parameters in each system call and not in their names. Examples of such groups are shown in Figure 6. Each group has a set of system calls, separated by disjunction (`||`) and associated with an abstract event. For instance, the $fileWriteOps(pathname)$ event, maps to a disjunction of all the system calls which perform a write operation on a pathname such as,
`open(pathname, O_RDWR)` and `creat`. The `pid` argument tracks the process making the call. Semantically, when a process executes a system call, the call matches the abstract event, if and only if, it matches one of the system calls represented by the event. A more extensive discussion of grouping was presented in [22].

As a next step, we identify pairs of abstract events which are not commutable for most of the programs. Specifically, when any of following operations occur within a race window, the events causing these operations are deemed non-commutable:

a. changing the filename-inode association.

b. replacing the file contents when the victim process is performing write operations on the file.

In the first case, the victim process starts operating on a completely different file. This causes the intended behavior of the victim process to change – and hence, can be considered as an attack. In such a case, the security policies terminate the process executing the malicious operation.

The second case is however not clear-cut. Such cases can also be part of intended behavior, e.g., multiple processes writing into a logfile. To prevent false alarms, in such cases, the policies trigger a reaction which *delay*'s the process causing the potentially malicious system call by putting the process in a wait queue and moving it to the `runqueue` queue only when the victim process closes the file or exits. This type of `delay` reaction was, to our knowledge, first proposed by [21]. Note that, the `delay` reaction has the simple effect of serializing concurrent file transactions and hence can be used for all file-related race conditions. We however, use it conservatively, since we believe that interactive processes can get unresponsive if they are delayed. Figure 7 shows the security policy that specify these checks in BMSL. In the policies "`|`" symbol associates a condition/assignment with a system call, the "·" stands for a sequence operation and the "*" is the standard Kleene-closure symbol.

In addition, we developed application-specific policies for the `filelogger` and `readfile` programs, to demonstrate the need for a high level language such as BMSL. Specifically, the following policies were developed:

- appending to a file, when `filelogger` process is accessing it is disallowed.
- reading a file, when `readfile` process is accessing it is disallowed.

Figure 8 shows the BMSL policies corresponding to the above operations. Note that, for the `filelogger` program, the response $move\_file$ is launched - it moves the log file into backup and creates a new file with that pathname and then allows the append operation to proceed. This prevents any false alarms when, for instance, multiple file logger programs are accessing the same file.

6. IMPLEMENTATION

Our code was implemented as a loadable kernel module on Linux. The module code was kept simple, running to about 1000 lines of code. In addition to the runtime infrastructure discussed in Section 3, some of the key implementation details are:

- *Runtime structures and functions to keep the low-level details of virtual file systems (VFS) hidden from policy*

```
/* write system calls on a pathname */
    $writeOps(path, pid) := \{open(path, O\_RDWR|O\_CREAT|O\_APPEND|O\_WRONLY|O\_TRUNC, pid)$
        $||truncate(path, length, pid)||creat(path, mode, pid)||mkdir(path, mode, pid)\}$
/* replace operations on a pathname */
    $fileReplaceOps(path, pid) := \{unlink(path, pid)||link(path, pid)||symlink(path, pid)$
        $||rename(path, newpath, pid)||rmdir(path, pid)\}$
/* privileged operations on a file - can be performed only by a root */
    $privilegedFileOps(path, pid) := \{chown(path, owner, group, pid)||chroot(path, pid)\}$
/* file attribute change system calls */
    $fileAttribChange(path, pid) := \{chmod(path, mode, pid)||utimes(path, buf, pid)\}$
/* file attribute check system calls */
    $fileAttribCheck(path, pid) := \{stat(path, buf, pid)||lstat(path, buf, pid)||access(path, pid)\}$
/* write or privileged system calls */
    $writeOrPrivilegedOps(path, pid) := \{writeOps(path, pid)||privilegedFileOps(path, pid)||fileAttribCheck(path, pid)\}$
/* append operations */
    $fileAppendOps(path, pid) := \{open(path, O\_APPEND, pid)\}$
/* all file operations: disjunction of all the above events */
    $allFileOps(path, pid) := writeOps(path, pid)|| \cdots ||fileAppendOps(path, pid)$
```

Figure 6: Classification of file related calls. Disjunction between the calls is represented using `||` symbol

```
/* A file cannot be replaced after a stat or access call */
1.  $(fileAttribCheck(path, pid))$
        $\cdot(!fileOps(path, pid))* \cdot (fileReplaceOps(path, pid2)|pid \neq pid2) \rightarrow \{term(pid2)\}$
/* No process can must change the inode number of the file when it is being accessed by another process */
2.  $((allFileOps(path, pid) \cdot (!allFileOps(path, pid))*$
        $\cdot(fileReplaceOps(path, pid2)|pid \neq pid2) \rightarrow term(pid2)$
/* Checks for the change in the contents of an inode across two successive operations by the same process */
3.  $((writeOps||PrivilegedFileOps(path, pid) \cdot (!closeFileOps(path, pid))*$
        $\cdot(fileContentReplaceOps(path, pid2)|pid \neq pid2) \rightarrow delay(pid2)$
```

Figure 7: Generic policies. `&&` : conjunction, $\cdot$: sequence, `|`: such that, `*`: kleene-closure

writers. In addition, these structures were designed to prevent conflicts and redundant information from being maintained when multiple processes access the same file. The salient implementations were:

– *Methods to access VFS file details:* They probe the `struct nameidata` of VFS, for inode number, file size and mode.

– *Structure to keep track of files being accessed by processes:* We maintain, globally, two instances of a hashtable, called `file_history`. Each instance is indexed by either the filename or inode number and points to the structure:

```
struct fileinfo {
  int pid[256]; // pid's accessing the file
  char* pathname; //canonicalized pathname
  int de_ino; //  file's inode number
  int refcount; //#processes accessing file
}
```

An entry in the hash-table is deleted when `refcount=0`.

7. EXPERIMENTATION

We conducted experiments to test: the effectiveness of our prototype in detecting known/unknown attacks at runtime, and its performance. All the experiments were conducted on a Pentium 4 machine with 256 Mb RAM running Red Hat 7.3. In the experiments, the prototype was used to monitor all programs on the system. In addition, we specifically ran, I/O intensive applications (to measure performance), setuid to root programs and commonly used daemons providing remote services. The specific programs were: `gcc`, `mount`, `tar`, `gzip`, `filelogger`, `make`, `in.ftpd`, `in.telnetd` and `sshd`.

7.1 Attack Datasets

To create an attack dataset, we searched through CERT advisories [6] and the GIAC practical repository on race condition attacks [16]. The latter categorizes race condition attacks into: file redirection, setuid scripts and relocated subdirectory attacks. In the CERT advisories we found attacks in the file redirection and setuid scripts categories. We selected the following three attacks from these two categories: `periodic` cron process temporary file attack, `mkdir-chown` file redirection attacks, and setuid to root attack: the `access-open` attack. For the relocated subdirectory attack category, we developed an attack called `mount` random directory attack – a concrete version of the relocated-subdirectory attack discussed earlier in Figure 2. In addition, we developed an attack, which has not yet been exploited called the `filelogger` attack discussed earlier in Figure 3 and `readfile` of Figure 4. Note that since, the experiments were on RedHat 7.2, in which all flaws which these attacks exploit were fixed. Hence, to run the attacks the source code of the programs was downloaded and the flaws were reinserted.

```
/* After fileLogger checks a file's size, and before it opens it for writing, no unprivileged process can append the file */
1.  $fileAppendOps(path, pid)$ | $path \in LogFileList$ && $pid \in PrivilegedProcessList$
        $\cdot(!fileOps(path, pid))$*$\cdot$ $fileAppendOps(path, pid2) \rightarrow \{move\_file(path)\}$
/* No reading of the file in race window for the readfile program */
2.  $(fileAttribChange(path, mode, pid)$ | $pid = readfile\_pid$ && $mode = 777)$
        $\cdot(!fileAttribChange(path, mode, pid)$ | $mode = 700)$*
        $\cdot(fileReadOps(path, pid2)$ || $fileWriteOps(path, pid2))$
            $\rightarrow \{term(pid2)\}$
```

Figure 8: Application specific policies

7.2 Attack Detection

We executed the attacks randomly. When only the generic policies were used, we could detect all the attacks except: `filelogger` and `readfile`, which were detected by the application specific policies. Table 9 summarizes these results.

7.3 Performance Measurement

Performance is affected by overheads due to: (a) system call interception, (b) pattern matching of system calls with the DEs and (c) storage and lookup of file information. Out of these parameters, we were only concerned with (b) and (c). This is because, system call interception introduces constant overheads. Specifically, I/O bound processes (such as tar and gzip) typically have interception overheads of about 20%, while the overhead due to CPU bound processes (such as `in.ftpd` – ftp server) is 5% [11]. This overhead is directly proportional to the number of system calls being executed and is independent of the nature of DEs – the contribution of this paper.

To measure overheads due to (b) and (c), we developed the following test suite:

- `gcc` on a large file (~5200 source lines), which measures the overhead due to (b) and (c) without taking into consideration the overheads caused by possible interferences by other `gcc` processes also accessing the same `file_history` tables.
- multiple gcc's, by compiling our prototype (using a make file). Specifically, this compiles 28 C++ source files, over 8 subdirectories. It resulted in 28 total gcc invocations, of which we noticed around 22 parallel invocations. The overhead here include overheads due to performing multiple inode lookups as well as takes into account interferences by processes in accessing same `file_history` tables.
- two I/O intensive programs *gzip* and *tar*, which make numerous file operations. Both these programs were run on our prototype directory.

Figure 10 presents the results. I/O intensive programs show greater overheads than CPU bound since they make a larger percentage of file related calls than the CPU intensive programs.

Program	Percent of file-related calls	Overhead percent (s)
`gcc`	10%	1.33%
`make`	30%	3.171%
`tar`	73%	7.34%
`gzip`	49%	3.69%

Figure 10: Total overhead due to enforcement of policies.

7.4 Discussion of Effectiveness Results

7.4.1 False Alarm Rate

The prototype detected all the attacks with few false alarms. This is mainly due to the `delay` reaction. Another reason for few false alarms is the use of application-specific policies. We must note that, if the attack data set were larger, we might have experienced more false alarms. Our results, however, indicate the high signal-to-noise ratio of our approach.

7.4.1.1 Novel attacks.

The specifications depend on the knowledge of program behavior and not on the actual attack signatures. For instance, the specifications for the `filelogger` program considered the behavior of `filelogger`. The signature of the attack, which involved, a malicious append operation was not considered. Hence, our approach differs from traditional misuse based approaches. A standardized intrusion detection evaluations on a similar scale as that of DARPA Lincoln Labs evaluations of 1998,1999,2000 [15] needs to be however performed to better justify the ability of our approach to detect known/unknown attacks.

8. RELATED WORK

Preventing race conditions is a well researched area in concurrent and multi-threaded systems [3, 18]. More recent works have looked at races in multi-threaded programs. They include approaches based on static analysis [5, 19], runtime detection of races [8, 17] and a combination of both [1]. However, these approaches cannot be directly applied to detecting race condition attacks, because of the following reasons:

- The underlying assumption in these approaches are that they have the source code available for all the processes involved in a race. However, this is not always the case. Moreover, in race condition attacks, the information about all the programs involved in the attack may not available. This is because any malicious process may cause the attack by interfering with the execution of a privileged process.
- Not all races conditions can be characterized as at-

Attack Name	Description	Percent instances detected	
		(gc)	(ps/pg)
`periodic`	attacker guesses name of temporary file used by privileged process and replaces it	100	100
mkdir-chown	Exploits time interval between a process creating a directory and changing its ownership	100	100
Simple access-open	See Section 2	0	100
mount attack	See Figure 2	100	100
`filelogger`	See Figure 3	0	100
`readfile`	See Figure 4	0	100

Figure 9: Attack Information. *gc*: **only generic specification,** *ps/pg*: **program specific/program group specifications**

tacks. For instance a race condition in which a user opens the same file using two editors is not an attack – just an error.

In the area of computer security, there have been approaches which are aimed at preventing race condition attacks. These include the static analysis approach of [4], where programs are statically analyzed to detect operations that have been exploited previously. However, it suffers from the same problem as other static analysis approaches – explosion in state space. In addition, this approach can only prevent known attacks. Raceguard [7] is a practical mechanism to prevent race conditions in UNIX. However, it only prevents race conditions on temporary files.

The non-interference technique [13], provides a theoretical formulation of what constitutes a race condition attack. In doing so they draw from the previous works on concurrent systems [3, 18]. However, their approach is abstract. Though, they use a subset of their formalisms to create a prototype to detect attacks, they neither discuss any of the practical challenges nor have they presented any results. Moreover, they do not address the issue of attacks that cause information leak.

[21] describe a practical approach to dynamically detect and prevent attacks. Their approach is based on monitoring file operations using what they call as *deny* and *allow* specifications. A deny specification allows all commutations of file operations except those which are explicitly denied. This is similar to a misuse-based approach as they deny only the attacks which have been exploited. The allow specification is used to detect unknown attacks. This denies all commutations of file operations except those which are explicitly specified to be allowed. A key contribution of this work is the `delay` reaction we discussed earlier. While this approach is practical, it differs from our approach in the following ways:

- They only consider filename-space associations. For instance, their approach cannot detect attacks which only change the contents of a file not its characteristics.
- They use only inode numbers of files to determine if file operations commute. Hence, their approach cannot detect the directory relocation attack of Figure 2, in which inode numbers of each and every subdirectory in the pathname of the file have to be considered for successful detection.
- Allow/deny policies are universally enforced on all the processes. This leads to one of the following problems: either their allow policies are very strict, causing lots of false alarms, or are very flexible thus being unable to detect unknown attacks. They do not discuss the false alarms when using the allow policy. We should note though that in their approach the default action when they detect an attack is to delay the other process. Hence, false alarms simply slow down the processes but do not effect their working.

9. LIMITATIONS

Our approach does not detect attacks which use covert channels, e.g., modifying the `/dev/mem` file. This file is an image of the main memory of the system. Hence, if a file is already loaded into the main memory, a user can change its attributes or content by editing this file. However, not only does this require a user to have super-user privileges, it is also a difficult task that can easily crash the system if the edits are not accurate.

Another example of such a covert channel is: files which are already mapped into the main memory can be modified using system calls which directly effect memory, such as `brk`. Filenames do not appear as arguments for such calls. Hence, our approach cannot detect such attacks. However, modifying the memory map of a file by a process, other than the one opening the file, is disallowed in Linux unless that process is a lightweight process created by the main process using `clone` system call. Though there can be other such covert channels, we did not find any such attacks (or the above mentioned limitations) reported either in CERT advisories [6] or in [16].

A more significant limitation is that we did not address race conditions that might occur within the execution of a system call itself [24, 9]. A possible solution would be to use lower-level events such as Linux Security Module (LSM) hooks [24].

10. REFERENCES

[1] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free java. In *5th ICVMC*, volume 2937 of *LNCS*, 2004.

[2] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state-space

exploration. *Formal Methods in System Design*, 18:97–116, 2001.

[3] Arthur Bernstein. Analysis of programs for parallel processing. In *IEEE Trans. Electronic Comput. EC-15 5, 757-763*, 1966.

[4] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[5] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *16th OOPSLA*, October 2001.

[6] CERT. http://www.cert.org/advisories/index.html.

[7] Crispin Cowan, Steve Beattie, Chris Wrigh, and Greg Kroah-Hartman. Raceguard. In *10th USENIX Security Symposium*, 2001.

[8] Stefan Savage et al. Eraser: A dynamic data race detector for multithreaded programs. *ACM TCS*, 15(4):391–411, 1997.

[9] Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *NDSS*, 2003.

[10] K. Ilgun. A real-time intrusion detection system for unix. In *IEEE S&P*, 1993.

[11] K. Jain and R Sekar. User-level infrastructure for system call int erposition: A platform for intrusion detection and confinement. In *ISOC NDSS*, 2000.

[12] C. Ko. *Execution Monitoring of Security-Critical Programs in a Distributed System*. PhD thesis, UCDavis, Dec 1996.

[13] Calvin Ko and Timothy Redmond. Noninterference and intrusion detection. In *IEEE S&P*, 2003.

[14] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, CERIAS lab, Purdue University, December 1995.

[15] R. Lippmann, J.W. Haines, D. Fried, J. Korba, and K. Das. The 1999 darpa off-line intrusion detection evaluation. In *Computer Networks*, 2000.

[16] J. Craig Lowery. A tour of tocttou. In *SANS GSEC practical v1.4b*, 2002.

[17] José F. Martínez and Josep Torrellas. Applying thread-level speculation to explicitly parallel applications. In *ASPLOS*, San Jose, CA, 2002.

[18] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[19] Bruno Blanchet Patrick. A static analyzer for large safety-critical software. In *citeseer.nj.nec.com/581205.html*.

[20] R. Sekar and Prem Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *USENIX Security Symposium*, 1999.

[21] Eugene Tsyrklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *USENIX Security Symposium*, 2003.

[22] P. Uppuluri and R. Sekar. Experiences with specification-based intrusion detection. In *RAID, LNCS*, 2001.

[23] Prem Uppuluri. *Intrusion Detection/Prevention Using Behavior Specfications*. PhD thesis, SUNY Stony Brook, August 2003.

[24] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah. Linux security modules. In *citeseer.nj.nec.com/wright02linux.html*, 2002.