TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study

Jinpeng Wei and Calton Pu
Georgia Institute of Technology
{weijp,calton}@cc.gatech.edu

ABSTRACT

Due to their non-deterministic nature, Time of Check To Time of Use (TOCTTOU) vulnerabilities in Unix-style file systems (e.g., Linux) are difficult to find and prevent. We describe a comprehensive model of TOCTTOU vulnerabilities, enumerating 224 file system call pairs that may lead to successful TOCTTOU attacks. Based on this model, we built kernel monitoring tools that confirmed known vulnerabilities and discovered new ones (in often-used system utilities such as *rpm*, *vi*, and *emacs*). We evaluated the probability of successfully exploiting these newly discovered vulnerabilities and analyzed in detail the system events during such attacks. Our performance evaluation shows that the dynamic monitoring of system calls introduces non-negligible overhead in microbenchmark of those file system calls, but their impact on application benchmarks such as Andrew and PostMark is only a few percent.

Categories and Subject Descriptors

D.4.3: File Systems Management – *Access methods*; D.4.5: Reliability –*verification*; D.4.6: Security and Protection – *Access controls*.

General Terms

Reliability, Experimentation, Security.

Keywords

Race detection

1 Introduction

TOCTTOU (Time Of Check To Time Of Use) is a well known security problem [1] in file systems with weak synchronization semantics (e.g., Unix file system). A TOCTTOU vulnerability requires two steps [2]. First, a vulnerable program checks for a file status. Second, the program operates on the file assuming the original file status remained invariant during execution. For example, *sendmail* may check for a specific attribute of a mailbox (e.g., it is not a symbolic link) in step one and then append new messages (as root) in step two. Because the two steps are not executed atomically, a local attacker (mailbox owner) can exploit the window of vulnerability between the two steps by deleting his/her mailbox and replacing it with a symbolic link to /etc/passwd. If the replacement is completed within the window and the new messages happen to be syntactically correct /etc/passwd entries with root access, then *sendmail* may unintentionally give unauthorized root access to a normal user (the attacker).

TOCTTOU vulnerabilities are a very significant problem. For example, between 2000 and 2004, we found 20 CERT [14] advisories on TOCTTOU vulnerabilities. They cover a wide range of applications from system management tools (e.g., /bin/sh, *shar*, *tripwire*) to user level applications (e.g., *gpm*, Netscape browser). A similar list compiled from BUGTRAQ [16] mailing list is shown in Table 1. The CERT advisories affected many operating systems, including: Caldera, Conectiva, Debian, FreeBSD, HP-UX, Immunix, MandrakeSoft, RedHat, Sun Solaris, and SuSE. In 11 of the CERT advisories, the attacker was able to gain unauthorized root access. TOCTTOU vulnerabilities are widespread and cause serious consequences.

Table 1: Reported TOCTTOU Vulnerabilities

Domain	Application Name
Enterprise applications	Apache, bzip2, gzip, getmail, Imp-webmail, procmail, openldap, openSSL, Kerberos, OpenOffice, StarOffice, CUPS, SAP, samba
Administrative tools	at, diskcheck, GNU fileutils, log-watch, patchadd
Device managers	Esound, glint, pppd, Xinetd
Development tools	make, perl, Rational ClearCase, KDE, BitKeeper, Cscope

At the same time, TOCTTOU vulnerabilities are also a very challenging research problem due to their non-deterministic nature. They are very hard to detect because the occurrence of a TOCTTOU vulnerability requires a pair of certain system calls along the execution path of an application combined with appropriate environmental conditions. So they are more elusive than say, a buffer overflow bug which is only a single point of failure. TOCTTOU vulnerabilities are also hard to exploit, because they are essentially race condition errors so whether an attack can succeed relies on whether the attacking code is executed within the usually narrow window of vulnerability (on the order of

milliseconds as shown in section 4.2). Furthermore, normal static program analysis tools for detecting race conditions cannot be applied directly, since the attack programs are usually unavailable until the vulnerabilities are discovered.

The first contribution of this paper is a model-based approach to detecting TOCTTOU attacks in Unix-style operating systems. During the 10 years since the first systematic study of TOCTTOU problem by Bishop [2][3], only partial solutions have been proposed for some instances of the problem [5][6][13]. In this paper, we develop a model and list a comprehensive enumeration of TOCTTOU vulnerabilities for the Linux virtual file system. To the best of our knowledge, this is the most complete study of TOCTTOU problem so far.

The second contribution of the paper is a systematic search for potential TOCTTOU vulnerabilities in Linux system utility programs. We implemented model-based software tools that are able to detect previously reported TOCTTOU vulnerabilities as well as finding some unknown ones (e.g., in the *rpm* software distribution program, the *vi/vim* and *emacs* editors). We conducted a detailed experimental study of successfully exploiting these vulnerabilities and analyze the significant events during a TOCTTOU attack against the native binaries of *rpm* and *vi*. By repeating the experiments, we also evaluated the probability of these events happening, as well as the success rate of these non-deterministic TOCTTOU attacks. These analyses provide a quantitatively better understanding of TOCTTOU attacks.

The rest of the paper is organized as follows. Section 2 summarizes the CUU model of TOCTTOU vulnerabilities. Section 3 describes a framework that detects TOCTTOU vulnerabilities through monitoring of TOCTTOU pairs. Section 4 presents a detailed analysis of events during the attacks on *rpm* and *vi*, including a study of attack success probability. Section 5 discusses the accuracy of the detection software tools and shows the measured overhead incurred by the tools. Section 6 summarizes related work and Section 7 concludes the paper.

2 The CUU Model of TOCTTOU

2.1 Broad Definition of TOCTTOU

A necessary condition for a TOCTTOU vulnerability to happen is a pair of system calls (referred to as "TOCTTOU pair" in this paper) operating on the same disk object using a file pathname. The first system call (referred to as "CU-call") establishes some preconditions about the file (e.g., the file exists, the current user has write privilege to the file, etc). The second system call (referred to as "Use-call") operates on the file, based on those preconditions. In our model, the preconditions about the file can be established either explicitly (e.g., **access** or **stat**) or implicitly (e.g., **open** or **creat**). Therefore, the TOCTTOU name is more restrictive than our model. Our model includes the original check-use system call pairs [2][3], plus use-use pairs. For example, a program may attempt to delete a file (instead of checking whether a file exists) before creating it. Consequently, the pair <**delete**, **create**> is also considered a (broadly defined) TOCTTOU pair.

2.2 An Enumeration of TOCTTOU pairs in Linux

We apply this model (called CUU) to the concrete situation of analyzing TOCTTOU problems in Linux. To get a complete list of TOCTTOU pairs, we first find the complete CUSet (the set of CU-calls) and UseSet (the set of Use-Calls). We select these two sets of kernel calls from the functional specification of Linux file system. We started from file system calls that require a pathname as input, and then filtered out those that are unlikely to be leveraged in a TOCTTOU attack. For example, **swapon** does not follow symbolic links so it is not included in the UseSet (Here we assume that all TOCTTOU attacks based on **swapon** are symbolic link kind attack). Finally we got the following CUSet and UseSet:

- CUSet = { **access, stat, open, creat, mknod, link, symlink, mkdir, unlink, rmdir, rename, execve, chmod, chown, truncate, utime, chdir, chroot, pivot_root, mount** }
- UseSet = { **creat, mknod, mkdir, rename, link, symlink, open, execve, chdir, chroot, pivot_root, mount, chmod, chown, truncate, utime** }

Although some system calls may appear unlikely candidates, they have been included after careful analysis. For example, **mknod** is in UseSet because it is able to create a new regular file, a function that is rarely known.

This classification of CUSet and UseSet is not structured enough for a complete analysis because some CU-calls and Use-calls are semantically unrelated. For example, <**creat, chdir**> is not a meaningful pair because **creat** creates a regular file while **chdir** expects a directory as argument. So we need to subdivide CUSet and UseSet so that a TOCTTOU pair at least applies to the same kind of storage objects (e.g. regular file, directory, or link). Thus we define the following sets.

Definition 1: CreationSet contains system calls that create new objects in the file system. It can be further divided into three subsets depending on the kind of objects that the system call creates:

CreationSet = FileCreationSet ∪ LinkCreationSet ∪ DirCreationSet, where

FileCreationSet = {**creat, open, mknod, rename**}
LinkCreationSet = {**link, symlink, rename**}
DirCreationSet = {**mkdir, rename**}

Definition 2: RemoveSet contains system calls that remove objects from the file system. It can be further divided into three corresponding subsets:

RemoveSet = FileRemoveSet ∪ LinkRemoveSet ∪ DirRemoveSet, where
FileRemoveSet = {**unlink, rename**}
LinkRemoveSet = {**unlink, rename**}
DirRemoveSet = {**rmdir, rename**}

Definition 3: NormalUseSet contains system calls which work on existing storage objects and do not remove them. We subdivide them into two sets:

NormalUseSet = FileNormalUseSet ∪ DirNormalUseSet, where
FileNormalUseSet = {**chmod, chown, truncate, utime, open, execve**}
DirNormalUseSet = {**chmod, chown, utime, mount, chdir, chroot, pivot_root**}

Definition 4: CheckSet contains the system calls that establish preconditions about a file pathname explicitly.

CheckSet = {**stat, access**}

Using the above definitions, we divide the CUSet and UseSet into subsets:

CUSet = CheckSet ∪ CreationSet ∪ RemoveSet ∪ NormalUseSet
UseSet = CreationSet ∪ NormalUseSet

Based on the precondition established by the CU-call, we can divide the TOCTTOU pairs into two groups: Group 1 creates a new object and Group 2 operates on an existing object. We say that TOCTTOU vulnerabilities are *not* due to bad programming practices, since in Group 1 the CU-call establishes the precondition that the file pathname does not exist and in Group 2 the CU-call establishes the precondition that the file pathname exists.

Group 1 preconditions can be established either explicitly by CU-calls in the CheckSet, or implicitly by CU-calls in the RemoveSet. These are followed by Use-calls in a CreationSet of the corresponding type, e.g., the creation of a directory is only paired with a system call on a directory.

Group 1 = (CheckSet × CreationSet) ∪(FileRemoveSet × FileCreationSet) ∪ (LinkRemoveSet × LinkCreationSet) ∪ (DirRemoveSet × DirCreationSet).

Group 2 preconditions can be established by CU-calls in the CheckSet, or by CU-calls in the CreationSet (a file/directory/link exists after it is created), or by CU-calls in the NormalUseSet. These are followed by corresponding Use-calls. The link-related calls are paired with both FileNormalUseSet and DirNormalUseSet because a link can point to either a regular file or a directory.

Group 2 = (CheckSet × NormalUseSet) ∪ (FileCreationSet × FileNormalUseSet) ∪ (DirCreationSet × DirNormalUseSet) ∪ (LinkCreationSet × FileNormalUseSet) ∪ (LinkCreationSet × DirNormalUseSet) ∪ (FileNormalUseSet × FileNormalUseSet) ∪ (DirNormalUseSet × DirNormalUseSet).

Intuitively, Group 1 ∪ Group 2 completes the set of TOCTTOU pairs. A formal proof of the completeness of CUU is out of the scope of this paper and is addressed in another paper [20].

In summary, Table 2 shows these TOCTTOU pairs along two dimensions: the use of a storage object and whether the check was an explicit check or an implicit check. A total of 224 pairs have been identified using this table.

Table 2: Classification of TOCTTOU Pairs

Use	Explicit check	Implicit check
Create a regular file	CheckSet × FileCreationSet	FileRemoveSet × FileCreationSet
Create a directory	CheckSet × DirCreationSet	DirRemoveSet × DirCreationSet
Create a link	CheckSet × LinkCreationSet	LinkRemoveSet × LinkCreationSet
Read/Write/ Execute or Change the attribute of a regular file	CheckSet × FileNormalUseSet	(FileCreationSet × FileNormalUseSet)∪ (LinkCreationSet × FileNormalUseSet)∪ (FileNormalUseSet × FileNormalUseSet)
Access or change the attribute of a directory	CheckSet × DirNormalUseSet	(DirCreationSet × DirNormalUseSet)∪ (LinkCreationSet × DirNormalUseSet)∪ (DirNormalUseSet × DirNormalUseSet)

2.3 Known TOCTTOU Examples

We applied our model to known TOCTTOU vulnerabilities and show the results in Table 3.

Table 3: Real world applications known to have TOCTTOU vulnerability

Applications	TOCTTOU pair	Classification
BitKeeper, Cscope 15.5, CUPS, getmail 4.2.0, glint, Kerberos 4, openldap, OpenOffice 1.0.1, patchadd, procmail, samba, Xinetd	<**stat, open**>	CheckSet × FileCreationSet
Rational ClearCase, pppd	<**stat, chmod**>	CheckSet × FileNormalUseSet
logwatch 2.1.1	<**stat, mkdir**>	CheckSet × DirCreationSet
bzip2-1.0.1, gzip, SAP	<**open, chmod**>	FileCreationSet × FileNormalUseSet
Mac OS X 10.4 – launchd	<**open, chown**>	
Apache 1.3.26, make	<**open, open**>	
StarOffice 5.2	<**mkdir, chmod**>	DirCreationSet × DirNormalUseSet

3 Model-Based TOCTTOU Detection

3.1 Components of Practical Attacks

An actual TOCTTOU vulnerability consists of a victim program containing a TOCTTOU pair (described in Section 2) and an attacker program trying to take advantage of the potential race condition introduced by the TOCTTOU pair. The attacker program attempts to access or modify the file being manipulated by the victim through shared access during the vulnerability window between the CU-call and Use-call. For example, by adding a line to an unintentionally shared script file in the *rpm* attack (Section 4.2), the attacker can trick the victim into executing unintended code at a higher privilege level (root). In general, we say that a TOCTTOU attack is profitable if the victim is running at a higher level of privilege. In Unix-style OSs, this means the victim running as root and the attacker as normal user.

An important observation is that even though the victim is running at a higher level of privilege, the attacker must have sufficient privileges to operate on the shared file attributes, e.g., creation or deletion. This observation narrows the scope of potential TOCTTOU vulnerabilities. Table 4 shows a list of directories owned by root in Linux. Since normal users cannot change the attributes or content of files in these directories, these files are safe.

Table 4: Directories Immune to TOCTTOU

/bin	/root	/usr/dict	/var/db
/boot	/proc	/usr/kerberos	/var/empty
/dev	/sbin	/usr/libexec	/var/ftp
/etc	/usr/bin	/usr/sbin	/var/lock
/lib	/usr/etc	/usr/src	/var/log
/misc	/usr/include	/usr/X11R6	/var/lib
/mnt	/usr/lib	/var/cache	/var/run
/opt			

3.2 CUU Model-Based Detection Tools

Based on the CUU model, we designed a software framework and implemented software tools to detect actual TOCTTOU vulnerabilities in Linux. Figure 1 shows the four components of our detection framework, based on dynamic monitoring of system calls made by sensitive applications (e.g., those that execute with root privileges). The first component of our framework is a set of plug-in Sensor code in the kernel, placed in system calls listed in the CUSet and UseSet (Section 2.2). These Sensors record the system call name and its arguments, particularly file name (full path for unique identification purposes). For some system calls, other related arguments are also recorded to assist in later analysis, e.g., the *mode* value of **chmod**(*path, mode*). Some environmental variables are also recorded, including process id, name of the application, user id, group id, effective user id, and effective group id. This information will be used in the analysis to determine if a TOCTTOU pair can be exploited. We do not use standard Linux trace facilities such as *strace* for two reasons: First, *strace* does not output full pathname for files referred to using relative pathnames; Second, *strace* does not give enough environmental information such as effective user id.

The Sensors component also carries out a preliminary filtering of their log. Specifically, they identify the system calls on files under the system directories listed in Table 4 and filter them out, since those files are immune to TOCTTOU attacks. After this filter, remaining potentially vulnerable system calls are recorded in a circular FIFO ring buffer by **printk**.

The second component of our framework is the Collector, which periodically empties the ring buffer (before it fills up). The current implementation of the Collector is a Linux daemon that transforms the log records into an XML format and writes the output to a log file for both online and offline analysis.

The third component of our framework is the Analyzer, which looks for TOCTTOU pairs (listed in Table 2) that refer to the same file pathname. For offline analysis, this correlation is currently done using XSLT (eXtensible Stylesheet Language Transformations)

templates. This analysis proceeds in several rounds as follows.

Round 1: First, the Analyzer sorts the log records by file name, grouping its operation records such as the names and locations (sequence numbers) of system calls.

Round 2: Second, system calls on each file are paired to facilitate the matching of TOCTTOU pairs.

Round 3: Third, system call pairs are compared to the list in Table 2. When a TOCTTOU pair is found, an XSLT template is generated to extract the corresponding log records from the original log file.

Round 4: Fourth, the log records related to TOCTTOU pairs found are extracted into a new file for further inspection.

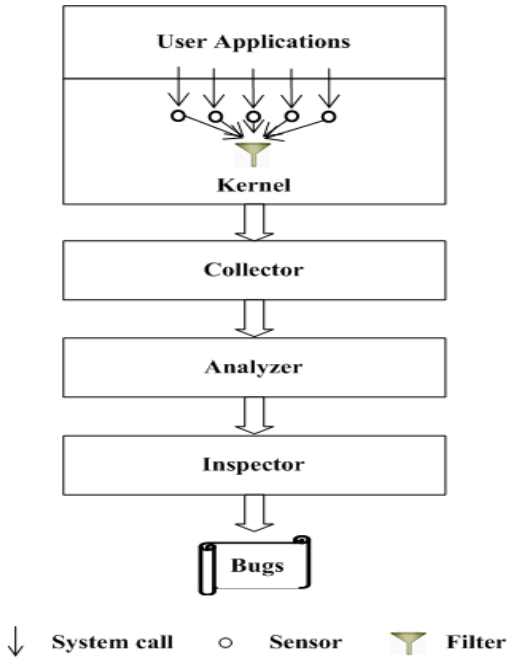

Figure 1: Framework for TOCTTOU Detection

The fourth component of our framework is the Inspector, which identifies the actual TOCTTOU vulnerability in the program being monitored. The Inspector links the TOCTTOU pair with associated environmental information, including file pathname, related arguments, process id, program name, user id, group id, effective user id, and effective group id. The Inspector decides whether an actual exploitation can occur.

For each TOCTTOU pair, the Inspector does the following steps:

- Check the arguments of the calls to see if these calls can be profitable to an attacker. For example, if the Use-Call is **chmod**, then a value of 0666 for the *mode* argument falls into this category because this **chmod** can be used to make /etc/passwd world-writable. On the other hand, a *mode* value of 0600 is not profitable because it will not give the attacker any permission on a file that he/she does not own. In this case the TOCTTOU pair in question is not a TOCTTOU vulnerability.
- Check the file pathname. For the **chmod** example, if the file is stored under a directory writable by an ordinary user, like his/her home directory, then continue to the next step; otherwise the TOCTTOU pair is not a TOCTTOU vulnerability.
- Check the effective user id. Continuing with the **chmod** example, if the effective user id is 0 (root), then report this TOCTTOU pair as a vulnerability; otherwise, the TOCTTOU pair is not a vulnerability.

It should be noted that the steps described above give only an outline of the Inspection process based on one attack scenario for one particular TOCTTOU pair. For different TOCTTOU pair and different attack scenario, the details of these checks can be different. For example, the same TOCTTOU pair as the above with a *mode* value of 0644 and the same other conditions is also considered a vulnerability because it can be exploited to make /etc/shadow readable by an attacker. Thus the Inspector requires a template (or signature) for each kind of attack scenario. Table 5 shows the set of templates used by the current implementation of the Inspector. For brevity, this table does not show the file pathname and effective user id which are checked in every template. This set may be expanded as new attack scenarios are found.

Table 5: Templates used in the Inspector

Use-Call	Arguments to check	Sample attack scenarios
chmod	*mode*	Gain unauthorized access rights to /etc/passwd
chown	*owner, group*	Change the ownership of /etc/passwd
chroot		Access information under a restricted directory
execve		Run arbitrary code
open	*mode, flag*	Mislead privileged programs to do things for the attacker, or steal sensitive information
truncate	*length*	Erase the content of /etc/passwd

4 Analysis of Real TOCTTOU Attacks

4.1 Experimental Setup

We applied our detection framework and tools to find previously unreported TOCTTOU vulnerabilities in Linux. Although the CUU model describes all the TOCTTOU pairs in Linux file systems, it is impractical to test all the execution paths of all the system software (or even a single program of any complexity). Our intent is to learn as much as possible about real TOCTTOU vulnerabilities through a detailed analysis. The experiments show that significant weaknesses can be found relatively easily using our framework and tools.

From the discussion in Section 3.1, we focus our attention on system software programs that use file system (outside the directories listed in Table 4) as a root. Each program chosen is downloaded, installed, configured, and deployed. Furthermore, we also build a testing environment which includes the design and generation of a representative workload for each application, plus the analysis of TOCTTOU pairs observed. Although this is a laborious process that requires high expertise, one could imagine incorporating such testing environments into the software release of system programs, facilitating future evaluations and experiments.

Our tools were implemented on Red Hat 9 Linux (kernel 2.4.20) to find TOCTTOU vulnerabilities in about 130 commonly used utility programs. The script-based experiments consist of about 400 lines of shell script for 70 programs in /bin and /sbin. This script takes about 270 seconds to gather approximately 310K bytes of system call and event information. The other 60 programs were run manually using an interactive interface. From this sample of Linux system utilities, we found 5 potential TOCTTOU vulnerabilities (see Table 6).

The experiments were run on an Intel P4 (2.26GHz) laptop with 256M memory. The Collector produces an event log at the rate of 650 bytes/sec when the system is idle (only background tasks such as daemons are running), 11KB/sec during the peak time a large application such as OpenOffice is started, and 2KB/sec on average. The Analyzer processes the log at the speed of 4KB/sec.

From the list in Table 6, we wrote simple attack programs that confirmed the TOCTTOU vulnerabilities in *rpm, emacs* and *vi*. We discuss the attack on *rpm* and *vi* in detail (Sections 4.2 and 4.3, respectively), and outline the others in Section 4.4.

4.2 rpm 4.2 Temp File Vulnerability

rpm is a popular software management tool for installing, uninstalling, verifying, querying, and updating software packages in Linux. When *rpm* installs or removes a software package, it creates a temporary script file in directories such as /var/tmp or /var/local/tmp. This shell script is used to install or remove help documentation of the software package. Since the access mode of this file is set to 666 (world-writable), an attacker can insert arbitrary commands into this script. Given the privileges required for installing software (usually root), this is a significant vulnerability. The TOCTTOU pair involved is <**open**, **open**>: the first **open** creates the script file for writing the script; and the second **open** is called in a child process to read and execute the script.

Table 6: Potential TOCTTOU Vulnerabilities

Application	TOCTTOU errors	Possible exploit
vi	**<open, chown>**	Changing the owner of /etc/passwd to an ordinary user
rpm	**<open, open>**	Running arbitrary command
emacs	**<open,chmod>**	Making /etc/shadow readable by an ordinary user
gedit	**<rename, chown>**	Changing the owner of /etc/passwd to an ordinary user
esd (Enlightened Sound Daemon)	**<mkdir, chmod>**	Gaining full access to another user's home directory

Table 7: Baseline vulnerability of rpm

Package Operation	Install (rpm –i)		Uninstall (rpm –e)	
	Average	Stdev	Average	Stdev
t (μsec)	125,188	9,930	110,571	10,961
v (μsec)	5,053	20	4,218	102
v/t	4.1%	---	3.8%	---

4.2.1 Baseline Analysis of *rpm*

In our evaluation of the TOCTTOU vulnerability in *rpm*, we start by measuring the total running time of *rpm* (denoted by t) and the window of vulnerability (the time interval between the two **open**s, denoted by v). We ran *rpm* (as root) 100 times, alternatively installing and uninstalling a package named *sharutils-4.2.1-14.i386.rpm*, and measured t and v for each invocation. From Table 7 we can see that the window of vulnerability is relatively narrow (less than 5%), since the two **open**s are separated only by a few milliseconds.

4.2.2 An Experiment to Exploit *rpm*

The second part of our evaluation is to measure the effectiveness of an attack trying to exploit this apparently small window of vulnerability. This experiment runs a user-level attack process in a loop. It constantly checks for the existence of a file name with the prefix "/var/tmp/rpm-tmp". A victim process (*rpm* run by root) installs a software package and creates a script file of that name. Note that *rpm* inserts a random suffix as protection against direct guessing, but a directory listing command bypasses the need to guess the full pathname. If a file name of the expected prefix appears, the attacker appends the command "*chown* attacker:attacker /etc/passwd" to it. If the append happens during the window of vulnerability, then the child process of *rpm* will execute the script and the inserted command line, making the attacker the owner of /etc/passwd. When *rpm* finishes, the test program checks whether the attacker has become the owner of /etc/passwd.

Due to the non-deterministic nature of these experiments, we ran the experiment 100 times in a batch. After running several batches, we found a surprisingly high average number of 85 successful attacks per batch, considering the apparently narrow window of vulnerability shown in Table 7.

4.2.3 Event Analysis of *rpm* Exploit

To fully understand what happened during the TOCTTOU attack, we analyze the important system events during the experiment. Figure 2 shows the events in a successful exploit of *rpm*. In Figure 2, the dark (upper) line shows the events of the *rpm* process, and the lower line shows the events of the attacker process. The attacker process stays in a loop looking for file names of interest. When the *rpm* process creates the file (just before the 200 msec clock tick), the attacker detects it and appends the *chown* line to the temporary script and goes back to the loop.

The two timelines show that even though the CPU consumption during the window of vulnerability is relatively small, the *rpm* process causes interrupts that lengthen the window, represented by dotted upper line. Specifically, there are at least two scheduling actions within the *rpm* vulnerability window: *rpm* creates a new process to execute *bash*, which creates another new process to execute an external executable file (/sbin/install-info). Each process creation causes rpm to yield CPU to the scheduler. Figure 2 shows that the attacker process is scheduled as a result and the attack succeeds. Consequently, the two scheduling actions created by *rpm* make the attack more likely to succeed because *rpm* yields the CPU in the window of vulnerability.

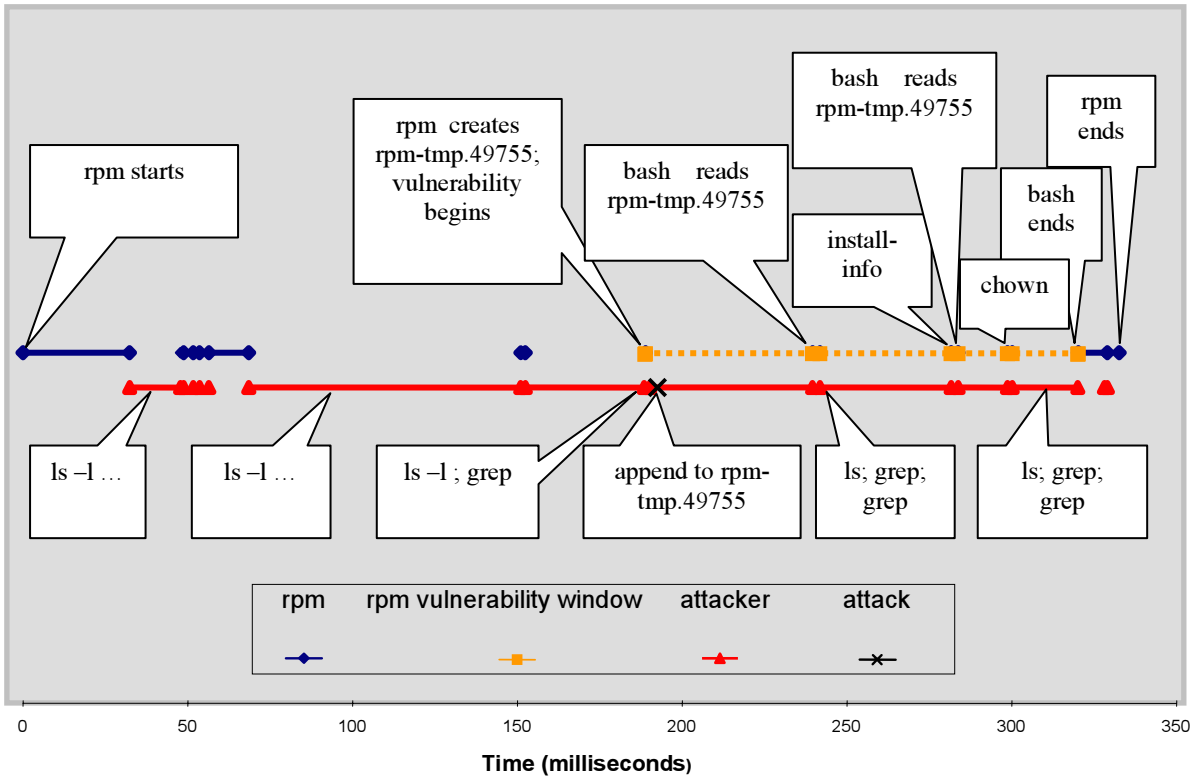

Figure 2: Event Analysis of *rpm* Exploit

In our experiments, we also found another reason more attacks succeed than indicated by the short window of vulnerability. Specifically, we observed that in some cases the appending to the script file by the attacker happened after the second **open** of *rpm* (outside the window), but the attack still succeeds. In these cases, we believe that append started after *bash* opened the script file (the second **open** of *rpm*), but it finished before *bash* reached the end of the script. Since *bash* interprets the script line by line, there is a good chance of executing the newly appended line. These two explanations (CPU yielding and slow interpretation of the script) help explain the lengthening of the window of vulnerability and the high attack success rate of 85%.

4.3 vi 6.1 Vulnerability

The Unix "visual editor" *vi* is a widely used text editor in many UNIX-style environments. For example, Red Hat Linux distribution includes *vi* 6.1. Using our tools, we found potential TOCTTOU vulnerabilities in *vi* 6.1. Specifically, if *vi* is run by root to edit a file owned by a normal user, then the normal user may become the owner of sensitive files such as /etc/passwd.

The problem can be summarized as follows. When *vi* saves the file being edited, it first renames the original file as a backup, then creates a new file with the original name. The new file is closed after all the content in the edit buffer is written. If *vi* is running as root, the initial owner and group of this new file is root, so *vi* needs to change the owner and group of the new file to its original owner and group. This forms an <**open, chown**> window of vulnerability every time *vi* saves the file. During this window, if the file name can be changed to a link to /etc/passwd, then *vi* can be tricked into changing the ownership of /etc/passwd to the normal user.

4.3.1 Baseline Analysis of *vi*

Using the same method of the *rpm* study, we measured the percentage of time when *vi* is running in its vulnerability window as it saves the file being edited. In *vi*, this depends on the edited file size. In our experiments, we bypass the user typing time to avoid the variations caused by human participation.

We define the save window *t* as the time *vi* spends in processing one "save" command, and the vulnerability window *v* during which TOCTTOU attack may happen. We measured 60 consecutive "saves" of the file for *t*, and timestamp the **open** and **chown** system calls for *v*. Since the "save" time of a file depends on the file size, we did a set of experiments on different file sizes. Figure 4 shows the time required for a "save" command for files of sizes from 100KB to 10MB. We found a per file fixed cost that takes about 14msec for the small (100KB) file and an incremental cost of 9msec/MB (for files of size up to 10MB).

Since **chown** happens after the file is completed, the window of vulnerability *v* follows approximately the same incremental growth of 9msec/MB (see Figure 4). Figure 3 shows the window of vulnerability to be relatively long compared to the total "save" time. It gradually grows to about 80% of the "save" total elapsed time for 10MB files. This experiment tells us that *vi* is more vulnerable when the file being edited is larger. For a small file (100KB size) the window of vulnerability is still about 5% of the "save" time.

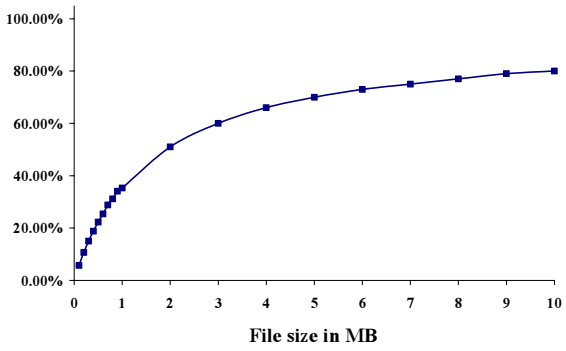

Figure 3: Window of Vulnerability Divided by Total Save Time, as a Function of File Size

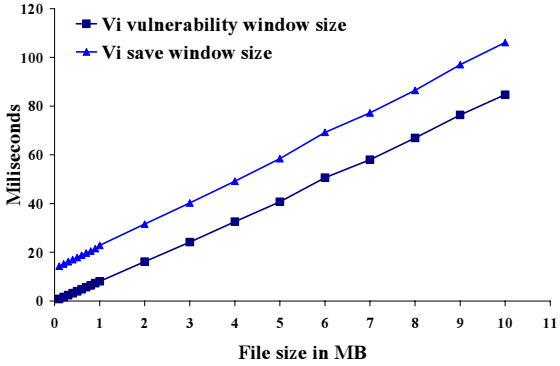

Figure 4: Vulnerability and Save Window Sizes of *vi*

4.3.2 An Experiment to Exploit *vi*

Unlike a batch program such as *rpm*, which is easily run from a script, *vi* is designed for interactive use by humans. To eliminate the influence of human "think time" in the experiments, we wrote another program to interact with *vi* by sending it commands that simulate human typing. This reduces the run-time and the window of vulnerability to minimum. The experiment runs a *vi* (as root) editing a file owned by the attacker in the attacker's home directory. The editing consists of either appending or deleting a line from the file and the experiment ends with *vi* exiting.

The attack consists of a tight loop constantly checking (by **stat**-ing) whether the owner of the file has become root, which signifies the start of the window of vulnerability. Once this happens, the attacker replaces the file with a symbolic link to /etc/passwd (as shown in Figure 7). When *vi* exits, it should change the ownership of /etc/passwd to the attacker. The attacker program checks for this ownership change. If *vi* finishes and /etc/passwd is still owned by root, the attack fails.

Contrary to the surprisingly high probability of success in the *rpm* case, we found a relatively low probability of success in the *vi* case (see Figure 5 and Figure 6), despite a relatively wide window of vulnerability. This leads to a more careful analysis of the system events during the attack.

4.3.3 Event Analysis of *vi* Exploit

Although the window of vulnerability may be wide, an attack will succeed only when:

1. *vi* has called **open** to create the new file,
2. *vi* has not called **chown,**
3. *vi* relinquishes CPU, voluntarily or involuntarily, and the attacker is scheduled to run, and
4. the attacker process finishes the file redirection during this run.

The first two conditions have been studied in the baseline experiment. The fourth condition depends on the implementation of the attacker program. For example, if the attacker program is written in C instead of shell script, it will be less likely to be interrupted.

The third condition is the least predictable. In our experiments, we have found several reasons for *vi* to relinquish CPU. First, *vi* may suspend itself to wait for I/O. This is likely since the window of vulnerability includes the writing of the content of the file, which may result in disk operations. Second, *vi* may use up its CPU slice. Third, *vi* may be preempted by higher priority processes such as *ntpd*, *kswapd*, and *bdflush* kernel threads. Even after *vi* relinquishes CPU, the second part of the condition (that the attacker process is scheduled to run) still depends on other processes not being ready to run.

This analysis illustrates the highly nondeterministic nature of a TOCTTOU attack. To achieve a statistically meaningful evaluation, we repeat the experiments and compute the probability of attack success. To make the experimental results reproducible, we eliminated all the confounding factors that we have identified. For example, in each round of experiments, we ran *vi* at least 50 times, each time on a different file, to minimize file caching effects. We also observed memory allocation problems after large files have been used. To relieve memory pressure, we added a 2-second delay between successive *vi* invocations.

Figure 5 shows the success rate for file sizes ranging from 100KB to 1MB averaged over 500 rounds. We see that for small files, there is a rough correlation between the size of window of vulnerability and success rate. Although not strictly linear, the larger the file being edited, the larger is the probability of successfully attacking *vi*.

Figure 6 shows the results for file sizes ranging from 2MB to 4MB, with a stepping size of 20KB, averaged over 100 rounds. Unlike the dominantly increasing success rate for small file sizes, we found apparently random fluctuations on success rates between file sizes of 2MB and 3MB, probably due to race conditions. For example, files of size 2MB have success rate of 4%, which is lower than the 8% success rate of file size 500KB in Figure 5. The growing success trend resumes after files become larger than 3MB.

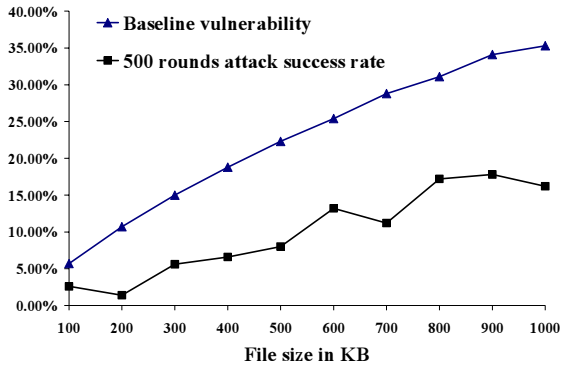

Figure 5: Success Rate of Attacking *vi* (small files)

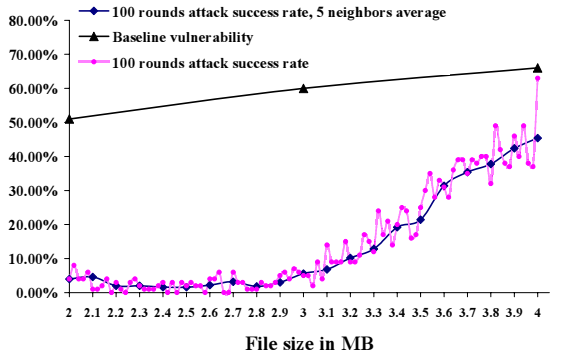

Figure 6: Success Rates of Attacking *vi* (large files)

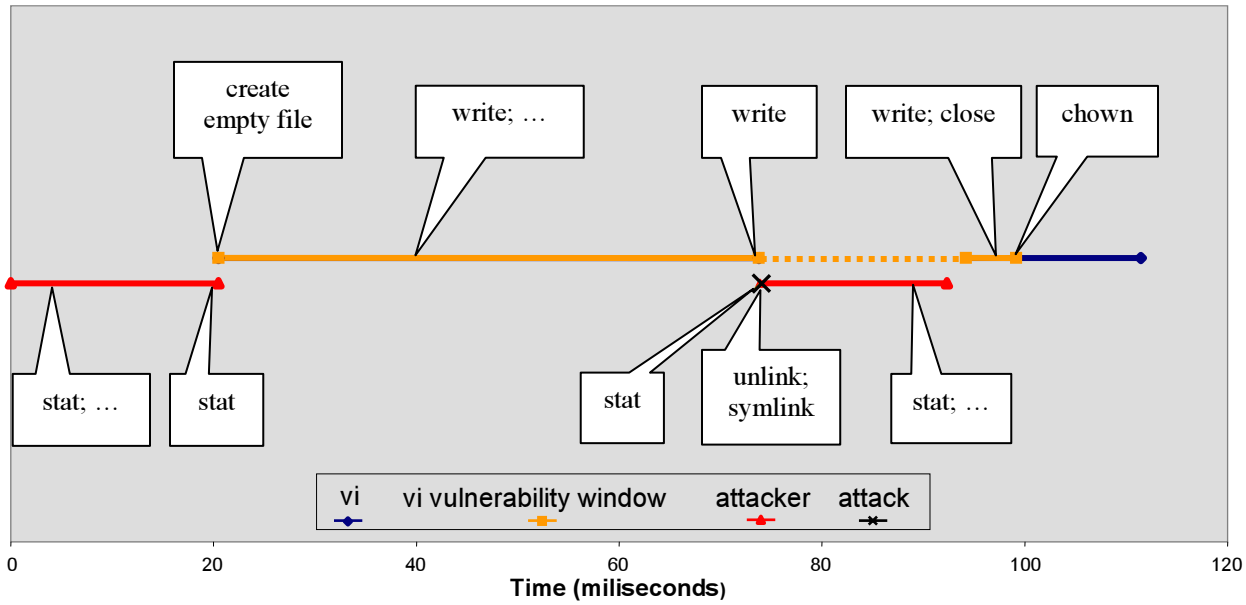

Figure 7: Event Analysis of the *vi* Exploit

4.4 Other Vulnerabilities

In our experiments, we identified 5 TOCTTOU pairs (see Table 6) and confirmed 3 of them through direct attacks (*rpm*, *vi*, and *emacs*). Due to its similarity to the *vi* experiments (Section 4.3), the analysis of the attack of *emacs* is omitted here.

We also tried to attack *gedit*, the fourth vulnerability discovered, but we found a very low probability of successful attack. Like *vi*, *gedit* becomes vulnerable when it saves the file being edited. Unlike *vi*, *gedit* writes to a temporary scratch file, then renames the scratch file to the original file name, and calls **chown**. Thus the window of vulnerability is between the **rename** and the directly following **chown**, a very short time that reduces the probability of successful attack. A full analysis of *gedit* experiments is beyond the scope of this paper.

The fifth vulnerability is the Enlightened Sound Daemon (*esd*), which creates a directory /tmp/.esd then changes the access mode of this directory to 777, giving full permissions (read/write/execute) to all users. Besides, this directory is under /tmp, a place where any user can create files or directories. So a possible attack is to create a symbolic link /tmp/.esd before the **mkdir** call of *esd* and let the link point to some directories owned by the running user (such as his/her home directory). If *esd* does not check whether its **mkdir** call succeeds, then it will change the access mode of the running user's home directory to 777. Then an attacker has full access to the running user's home directory. We postponed our experiments on *esd* since this TOCTTOU vulnerability has been reported in BUGTRAQ [17].

Overall, we consider the CUU model-based detection framework to be a success. With a modest number of experiments, we confirmed known TOCTTOU vulnerabilities and found several previously unreported ones. However, this offline analysis only covers the execution paths exercised by the workloads, so it cannot guarantee the absence of TOCTTOU vulnerabilities when none is reported.

In this paper, our research focuses on the scheduling aspects of TOCTTOU attacks in uniprocessor environments. Multiprocessors, hyper-threaded uniprocessors, or multi-core processors are beyond the scope of this paper and subject of ongoing research.

5 Evaluation of Detection Method

5.1 Discussion of False Negatives

As mentioned in Section 4.1, our tools are not designed for exhaustive testing. While we attempted to generate representative workloads for the 130 programs tested, we cannot guarantee coverage of all execution paths. The coverage problem may be alleviated by improvements in the testing technology and documentation.

More fundamentally, the CUU-Model covers pairs of file system calls, assuming that a precondition is established by the CU-call before the Use-call relies on it. In programs where preconditions are not explicitly established (a bad programming practice), e.g., a program creates a temporary file under a known name without first **stat**-ing the existence of the file, exploits may

happen outside the CUU model. The problem of complex interactions among more than a pair of system calls is an open research question. (Currently, there are no known examples of such complex vulnerabilities.)

5.2 Discussion of False Positives

Tool-based detection of vulnerabilities typically does not achieve 100% precision. The framework described in Section 3 is no exception. There are some technical sources of false positives:

1. Incomplete knowledge of search space: The list of immune directories (Table 4) is not complete because of the dynamic changes to system state (e.g. newly created root-owned directories under /usr/local), which leads to false positives.
2. Artifacts of test environment: If the test cases themselves uses /tmp or the home directory of an ordinary user, our tools have to report related TOCTTOU pairs, which are false positives. For example, the initial test case for *cpio* uses a temporary directory /tmp/cpio, so the tools reported a <**stat, chdir**> on this directory.
3. Coincidental events: Because our tools do system-wide monitoring, they capture file system calls made by every process. Sometimes two unrelated processes happen to make system calls on the same file that appear to be a TOCTTOU pair.
4. Incomplete knowledge of application domain: Not every TOCTTOU pair is profitably exploitable. For example, the application *rpm* invoked by "--addsign" option contains a <**stat, open**> pair, which can open any file in the system for reading, such as /etc/shadow. However, *rpm* can not process /etc/shadow because it is not in the format recognizable by *rpm*. So it is unlikely that this pair can be exploited to undermine a system.

By improving the kernel filter (source 1), re-designing test cases (source 2), and reducing concurrent activities (source 3), we reduced the false positive of our tools; for example, in one experiment testing 33 Linux programs under /bin, the false positive rate fell from 75% to 27%. However, source 4 is hard to remove due to the differences among application domains.

5.3 Overhead Measurements

To evaluate the overhead of our detection framework, we ran a variant of the Andrew benchmark [9]. The benchmark consists of five stages. First, it uses **mkdir** to recursively create 110 directories. Second it copies 744 files with a total size of 12MB. Third, it **stat**s 1715 files and directories. Fourth, it *grep*s (scan through) these files and directories, reading a total amount of 26M bytes. Fifth, it does a compilation of around 150 source files. For every stage, the total running time is calculated and recorded. We run this benchmark for 20 rounds and get the average. To mitigate the interference from other processes during the run, we start Red Hat in single-user mode (without X window system and daemon processes such as *apmd, crond, cardmgr, syslogd, gpm, cups* and *sendmail*). To get an estimation of the overhead of our system, we run this experiment on a Linux box without modifications to get the baseline results, and then a Linux box with our monitoring tools (without the Analyzer and the Inspector which are used offline). For the latter case, we ran the experiment under two different directories to see the influence of file pathname to the overhead. The total running time of these five stages for the experiments is shown in Figure 8 and Table 8.

The results show a relatively higher overhead for **mkdir, copy** and **stat** when the benchmark is run under an ordinary user's home directory (denoted Vulnerable Dir in Figure 8 and Table 8). But when the benchmark is run under /root (denoted Immune Dir in Figure 8 and Table 8), the overhead becomes much lower (dropping from 144% to 14% for **stat**). This difference shows that **printk**s in the kernel and the Collector daemon process contribute significantly to the overhead, because the filter in kernel suppresses most log messages caused by the benchmark when it runs in a directory immune to TOCTTOU (Table 4), therefore the **printk**s and Collector have much less work to do. The other source of overhead comes from the Sensor (including the filter and a query of the internal /proc file system data structure to map a process id to the complete command line to assist the Inspector). However, the overhead of our detection tools is amortized by application workload, as shown for compilation.

PostMark benchmark [11] is designed to create a large pool of continually changing files and to measure the transaction rates for a workload approximating a large Internet electronic mail server. Since mail server software such as *sendmail* had well known TOCTTOU problems, PostMark seems to be another representative workload to evaluate the performance overhead of our software tools.

When PostMark benchmark is running, it first tests the speed of creating new files, and the files have variable lengths that are configurable. Then it tests the speed of transactions. Each transaction has a pair of smaller transactions, which are either read/append or create/delete.

On the original Linux kernel the running time of this benchmark is 30 seconds. On our modified kernel, with all the same parameter settings, the running time is 30.35 seconds when the experiment is run under /root (an immune directory), and 35 seconds when the experiment is run under a vulnerable directory. So the overhead is 1.17% and 16.7% for these two cases, respectively. This result also shows that the **printk**s and the Collector contribute significantly to the overhead.

Table 8: Andrew Benchmark Results (msec)

Functions	Original Linux	Modified Linux Immune Dir		Modified Linux Vulnerable Dir	
		Time	Overhead	Time	Overhead
mkdir	2.8 ±0.06	3.0 ±0.10	7.1%	4.1 ±0.05	46%
copy	59.2 ±0.49	64.8 ±2.2	9.5%	80.8 ±0.46	36%
stat	61.1 ±0.55	69.4 ±0.41	14%	149.3 ±3.5	144%
grep	543.1 ±2.4	576.2 ±5.9	6.1%	645.3 ±3.7	19%
compile	20,668 ±66	20,959 ±90	1.4%	21,311 ±195	3.1%

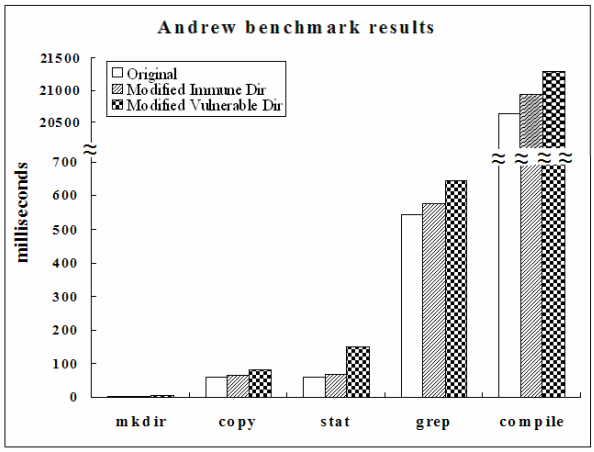

Figure 8: Andrew Benchmark Results

6 Related Work

The impression that TOCTTOU vulnerabilities are due to bad programming practices is probably created by the patches and solutions suggested in advisories and reports on TOCTTOU exploits from US-CERT [14], CIAC [15] and BUGTRAQ [16]. For example, many of the reported problems link temporary files to another file to be manipulated. Examples of TOCTTOU pairs are <**stat, mkdir**> and <**stat, open**>. Typical solutions suggested for patching these problems include:

- Using random number to obfuscate file names.
- Replacing mktemp() with mkstemp().
- Using a strict umask to protect temporary directories.
- Dropping privileges to those of an explicitly configured user.
- Setting proper file/directory permissions.
- Checking the return code of function calls.

Although these suggestions are useful, they cannot detect nor prevent these exploits. CUU model provides a systematic approach to detection and prevention (outside the scope of this paper).

In recent years, static analysis of source code has been used to find bugs in systems software. Significant examples include: Bishop and Dilger's prototype analysis tool that used pattern matching to look for TOCTTOU pairs [2][3]; Meta-compilation [7] using compiler-extensions to check conformance to system specific rules; RacerX [8] decoupling compilation from rule-checking plus inter-procedural analysis; MOPS [4] using model checking to verify program security properties. These static analysis tools are limited in the detection of real TOCTTOU problems due to difficulties with dynamic states (e.g., file names, ownership, and access rights) and unavailability of attacker programs for race condition checking.

Dynamic monitoring and analysis have been used to gain insights into a system's behavior in many different settings, such as file access prediction in mobile computing [18][19]. In the particular area of software security, dynamic monitors observe application execution to find software bugs. These tools can be further classified into dynamic online analysis tools and post mortem analysis tools. Eraser [12] is an online analysis tool that uses lockset analysis to find race conditions (unsynchronized access to shared variables) in a multithreaded program. Calvin et al [10] proposed a post mortem analysis tool for security vulnerabilities (including TOCTTOU) related to privileged programs (setuid programs). However, this tool can only detect the result of exploiting a TOCTTOU vulnerability but cannot locate the error.

In the area of general mechanisms to defend against TOCTTOU attacks, solutions have been proposed for specific TOCTTOU pairs. For example, Dean and Hu [6] add multiple <**access, open**> pairs (called strengthening rounds) to reduce the probability of successful attack against the TOCTTOU pair <**access, open**>. RaceGuard [5] prevents the temporary file creation race condition in UNIX systems, specifically, the <**stat, open**> TOCTTOU pair. Tsyrklevich and Yee [13] described a protection mechanism (called pseudo-transaction) to prevent race conditions between specified system call pairs. Although the pseudo-transaction mechanism is sufficiently general, their specification of TOCTTOU pairs was based on heuristics. The CUU model is a generalization of previous work watching for specific TOCTTOU pairs. Our work also complements mechanisms such as pseudo-transactions by providing a complete model (with 224 identified TOCTTOU pairs) to monitor all potentially dangerous interactions.

7 Conclusion

According to CERT [14] advisories and BUGTRAQ [16] reports, TOCTTOU problems are both numerous

and serious. We describe the CUU model and framework to detect TOCTTOU vulnerabilities. The model consists of 224 pairs of dangerous file system calls (the TOCTTOU pairs) and we implemented the detection framework for offline analysis of TOCTTOU vulnerabilities. The CUU model is programming language-independent. The software tools work without changes or access to application source code.

Using offline analysis, we confirmed known TOCTTOU attacks such as *esd* [17]. Running a relatively modest set of experiments (about 130 system utility programs), we also found and confirmed previously unreported TOCTTOU vulnerabilities in (the unmodified, original version of) *rpm*, *emacs* and *vi*.

To understand better TOCTTOU vulnerabilities, we recorded and analyzed in detail the main events in the attack scenarios. These analyses support a quantitative evaluation of the likelihood of success for each attack (ranging from very unlikely in the *gedit* case to highly likely in the *rpm* case at 85%). This evaluation is a non-trivial task for non-deterministic concurrent programs. We also measured and found modest performance overhead of our tools by running the Andrew and PostMark benchmarks (a few percent additional overhead for application level benchmarks).

The CUU model-based analysis of TOCTTOU vulnerabilities also suggests online defense mechanisms similar to pseudo-transactions [13]. This is a topic of active research and beyond the scope of this paper.

8 Acknowledgement

This work was partially supported by NSF/CISE IIS and CNS divisions through grants CCR-0121643, IDM-0242397 and ITR-0219902. We also thank the anonymous FAST reviewers for their insightful comments.

9 References

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

[2] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. Computing Systems, 9(2):131–152, Spring 1996.

[3] Matt Bishop. Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux. Technical Report 95-8, Department of Computer Science, University of California at Davis, September 1995.

[4] Hao Chen, David Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS), pages 235--244, Washington, DC, November 2002.

[5] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities. In Proceedings of the 10th USENIX Security Symposium, Washington DC, August 2001.

[6] Drew Dean and Alan J. Hu. Fixing Races for Fun and Profit: How to use access(2). In Proceedings of the 13th USENIX Security Symposium, San Diego, CA, August 2004.

[7] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In Proceedings of Operating Systems Design and Implementation (OSDI), September 2000.

[8] Dawson Engler, Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP'2003).

[9] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system, Transactions on Computer Systems, vol. 6, pp. 51-81, February 1988.

[10] Calvin Ko, George Fink, Karl Levitt. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. Proceedings of the 10th Annual Computer Security Applications Conference, page 134-144.

[11] PostMark benchmark. http://www.netapp.com/tech_library/3022.html

[12] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.

[13] Eugene Tsyrklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In Proceedings of the 12th USENIX Security Symposium, pages 243–256, Washington, DC, August 2003.

[14] United States Computer Emergency Readiness Team, http://www.kb.cert.org/vuls/

[15] U.S. Department of Energy Computer Incident Advisory Capability. http://www.ciac.org/ciac/

[16] BUGTRAQ Archive http://msgs.securepoint.com/bugtraq/

[17] BUGTRAQ report RHSA-2000:077-03: esound contains a race condition. http://msgs.securepoint.com/bugtraq/

[18] Ahmed Amer and Darrell D. E. Long. Noah: Low-cost file access prediction through pairs. In Proceedings of the 20th IEEE International Performance, Computing and Communications Conference (IPCCC '01), April 2001.

[19] Tsozen Yeh, Darrell D. E. Long, and Scott Brandt. Performing file prediction with a program-based successor model. In Proceedings of the 9th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '01), pages 193~202, Cincinnati, OH, August 2001.

[20] Calton Pu, Jinpeng Wei. A theoretical study of TOCTTOU problem modeling. In preparation.