International Symposium on Secure Software Engineering (ISSSE'06), Arlington VA, March 13-15, 2006.

A Methodical Defense against TOCTTOU Attacks: The EDGI Approach

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

Abstract

TOCTTOU is a challenging and significant problem, involving two-step (check and use) file object access by a victim process and simultaneously an attacker access to the same file object in-between the two steps. We describe a model-based, event-driven defense mechanism (called EDGI), which prevents such attacks by stopping the second process in-between the two steps. Our main contribution is the systematic design and implementation of EDGI defense and its evaluation. EDGI has no false negatives and very few false positives. It works without changing application code or API. A Linux kernel implementation shows the practicality of the EDGI defense, and an experimental evaluation shows low additional overhead on representative workloads.

1. Introduction

TOCTTOU (Time Of Check To Time Of Use) is a well known security problem [1]. A classic example is *sendmail*, which used to check for a specific attribute of a mailbox (e.g., it is not a symbolic link) before appending new messages. Unfortunately, the mailbox owner (attacker) can exploit the window of vulnerability between the checking and appending by deleting his mailbox and replacing it with a symbolic link to /etc/passwd. If an attack message consists of a syntactically correct /etc/passwd entry with root access, then the attacker may gain root access.

The *sendmail* example also illustrates the complexity of a TOCTTOU attack [23], which requires (unintended) shared access to a file by the attacker and the victim (the *sendmail*), plus the two distinct steps (check and use) in the victim. This structural complexity makes the detection of TOCTTOU vulnerabilities difficult. For example, unlike buffer overflow problems, TOCTTOU victim programs are tricked into performing a relatively small action for the attacker and proceeding, without significant deviation from normal behavior. Furthermore, successful techniques for typical race condition detection such as static analysis are not directly applicable, since the attacker program is not available beforehand. Finally, TOCTTOU attacks are inherently non-deterministic and not easily reproducible, making the detection of such vulnerability even more difficult. These difficulties are illustrated by the TOCTTOU vulnerability recently found in *vi/vim* and *emacs* [23], which appears to be in place since the time those venerable programs were created.

The difficulty of detection contrasts with the simplicity of some of the proposed patches suggested in advisories and reports on TOCTTOU exploits from US-CERT [15], CIAC [16] and BUGTRAQ [17], including setting proper file/directory permissions and checking the return code of function calls. However, some other suggested programming fixes are varied and non-trivial: using random numbers to obfuscate file names, replacing mktemp() with mkstemp(), and using a strict umask to protect temporary directories. More significantly, none of these fixes can be considered a comprehensive solution for TOCTTOU vulnerabilities.

The main contribution of this paper is a model-based, event-driven defense mechanism (called EDGI) for preventing exploitation of TOCTTOU vulnerabilities. Although TOCTTOU vulnerabilities need not always involve file access [25], in this paper we focus on such vulnerabilities in Unix-style file systems. The EDGI defense has several advantages over previously proposed solutions. First, based on the CUU model [23][24], EDGI is a systematically developed defense mechanism with careful design (using ECA rules) and implementation. Assuming the completeness of the CUU model, EDGI can stop all TOCTTOU attacks. Second, with careful handling of issues such as inference of invariant scopes and time-outs, EDGI allows very few false positives. Third, it does not require changes to applications or file system API. Fourth, our implementation on Linux kernel and its experimental evaluation show that EDGI carries little additional overhead.

The relevance of our work is underscored by the severity and frequency of TOCTTOU vulnerabilities reported. 20 CERT [15] advisories on TOCTTOU vulnerabilities were reported between 2000 and 2004. In 11 of these advisories, the attacker was able to gain unauthorized root

access. A list compiled from BUGTRAQ [17] mailing list is shown in Table 1 ([23]).

The rest of the paper is organized as follows. Section 2 summarizes previous relevant research on TOCTTOU vulnerabilities and defenses. Section 3 outlines the CUU model and Section 4 describes the design of EDGI defense. Section 5 outlines a prototype implementation, and Section 6 describes the experimental evaluation of the implementation. Section 7 concludes the paper.

Table 1: Reported TOCTTOU Vulnerabilities [23]

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,

2. Related Work

The TOCTTOU problem was first identified by the Protection Analysis project [2] and the Research into Secure Operating Systems project [1]. Matt Bishop [3][4] was the first to describe a prototype analysis tool that looks for TOCTTOU pairs in applications and also discussed the non-deterministic nature of the problem. Since then, several attempts have been made to defend against subsets of TOCTTOU vulnerabilities. We discuss first the approaches using dynamic monitoring (as we do).

RaceGuard [6] uses kernel monitoring to protect against a specific pair of file system calls: <**stat, open**>, where **open** is used to create a temporary file in a publicly shared directory such as /tmp. RaceGuard caches the file name tested by the **stat** call (to be non-existent) and aborts the **open** call if it finds the file created after **stat**. While RaceGuard is effective at detecting and stopping <**stat, open**> attack, it is unclear how to define, detect or prevent attacks based on other such pairs called TOCTTOU pairs (see a more precise definition in Section 3).

Another example of defense for a specific TOCTTOU pair (<**access, open**>) is a probabilistic approach [7]. Their solution adds multiple <**access, open**> pairs (called strengthening rounds) following the original <**access, open**> pair. The strengthening rounds check the invariance of path-to-inode mapping following the first pair. For an attack to succeed, all the rounds must return the same mapping, which grows exponentially harder with the number of rounds. Although this approach may be applied to other TOCTTOU pairs as they are discovered, it requires a change of application source code.

A more generic defense is pseudo-transactions [14]. Their idea is to encapsulate TOCTTOU pairs in pseudo-transactions that preserve the file name mapping starting from the CU-call through to the Use-call. By disallowing some sequences of file system calls (the TOCTTOU pairs identified by them) and allowing others, they are able to prevent known TOCTTOU attacks. Pseudo-transactions support a flexible specification of policies for allowed and disallowed pairs. They derived heuristically a set of policies, which was empirically refined but unverified. In comparison, the EDGI approach has some similarities in implementation (Section 5), but our model-based design (Section 4) answers the questions on the validity of file system call pair allow/disallow policies.

As mentioned earlier, the complexity of TOCTTOU attacks is due to the synchronized access by two independent programs (attacker and victim) to a shared file, precisely between the two steps formed by a TOCTTOU pair. This complexity, plus a number of dynamic states (e.g., file names, ownership, and access rights), limits the usefulness of static analysis techniques that require the availability of all source code involved. Representative examples include: Bishop and Dilger's pattern matching tool [3][4], Meta-compilation [8] with compiler extensions, RacerX [9] with inter-procedural analysis, and MOPS [5] using model checking. Extensions to dynamic monitoring and analysis include dynamic online analysis tools such as Eraser [13] for finding race conditions in a multithreaded program and post mortem analysis tools [11] that can detect the result of exploiting a TOCTTOU vulnerability but not locate the error.

3. The CUU Model of TOCTTOU

A previous study of TOCTTOU vulnerabilities and their detection [23] introduced the CUU model of TOCTTOU vulnerabilities. In contrast to detection, the focus and main contribution of this paper are the design and implementation of the EDGI defense (Sections 4 through 6) to prevent TOCTTOU attacks. Since EDGI is also based on the CUU model, we summarize the model here to make this paper self-contained. A more theoretical question of the completeness of the CUU model is addressed in another paper [24].

3.1. Broad Definition of TOCTTOU

A necessary condition for a TOCTTOU vulnerability is a pair of system calls (referred to as "TOCTTOU pair") operating on the same file object using the file name. The first of the pair (called "CU-call") establishes some preconditions about the file name (e.g., the file's existence and the user's access privileges, etc). Based on those preconditions, the second of the pair (called "Use-call") operates on the file.

In our model (called the *CUU Model*), the preconditions (called *invariants* since they must remain true through the Use-call) about the file can be established either explicitly (e.g., **access** or **stat**[1]) or implicitly (e.g., **open** or **creat**). The CUU model includes both the original check-use system call pairs [3][4], and 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.

3.2. Enumeration of TOCTTOU pairs

Using the CUU model, we are able to enumerate systematically the set of TOCTTOU pairs in a file system. The question of completeness (the model's ability to enumerate all TOCTTOU pairs) is addressed in another paper [24]. Here, we present an intuitive argument to motivate the EDGI system.

The CUU model classifies the file system calls into four groups: creation, removal, normal use, and status check. These groups are defined abstractly, based on the system call functionality. The status check operations read file object state without changing it, while the other three groups are "use" operations that cause side effects. The four-group division is further divided into subsets due to the existence of several kinds of Unix-style file objects: regular files, directories, and symbolic links.

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.

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

RemovalSet = FileRemovalSet ∪ LinkRemovalSet ∪ DirRemovalSet.

Definition 3: NormalUseSet contains system calls which work on existing file objects and do not remove them. They are subdivided into two sets:

NormalUseSet = FileNormalUseSet ∪ DirNormalUseSet

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

This classification is generic for any Unix-style file system. To get the actual TOCTTOU pairs for a particular file system, one only needs to compute the corresponding sets in Definition 1 to 4, and then apply the formulas in Table 2. For example, by studying the functional specification of Linux file system, we get the following sets:

FileCreationSet = {**creat, open, mknod, rename**}

LinkCreationSet = {**link, symlink, rename**}

DirCreationSet = {**mkdir, rename**}

FileRemovalSet = {**unlink, rename**}

LinkRemovalSet = {**unlink, rename**}

DirRemovalSet = {**rmdir, rename**}

FileNormalUseSet = {**chmod, chown, truncate, utime, open, execve**}

DirNormalUseSet = {**chmod, chown, utime, mount, chdir, chroot, pivot_root**}

CheckSet = {**stat, access**}

As result, a total of 224 pairs have been identified for the Linux file system using these sets and their combinations shown in Table 2 (first introduced in [23]).

Table 2: Classification of TOCTTOU Pairs [23]

Use	Explicit check	Implicit check
Create a regular file	CheckSet × FileCreationSet	FileRemovalSet × FileCreationSet
Create a directory	CheckSet × DirCreationSet	DirRemovalSet × DirCreationSet
Create a link	CheckSet × LinkCreationSet	LinkRemovalSet × 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)

4. The EDGI Defense against TOCTTOU

4.1. Overview

We propose an event driven approach, called EDGI (**E**vent **D**riven **G**uarding of **I**nvariants), to defend applications against TOCTTOU attacks. The design requirements of EDGI are:

1. It should solve the problem within the file system, and does not change the API, so that existing or future applications need not be modified.

[1] There are several variants of **stat**, such as **lstat** and **stat64**. For simplicity of presentation we use **stat** as the general term to denote all its variants in this paper.

2. It should solve the problem completely, i.e., no false negatives.
3. It should not add undue burden on the system, i.e., very low rate of false positives.
4. It should incur very low overhead on the system.

EDGI consists of three design steps (described in the rest of this section), a concrete implementation (Section 5), and an experimental evaluation (Section 6). The first design step is to map the CUU model (summarized in Section 3) into invariants in a concrete file system (Linux in our case) and the kernel calls that preserve the invariants. The second design step uses ECA (event-condition-action) rules [20][21] to model the concrete invariant preservation methods, so we can have reasonable assurance the invariants are indeed preserved. The third design step completes the design by addressing the remaining issues such as the automated inference of invariant scope and inheritance of invariants by children processes.

In the EDGI approach, each CU-call creates an invariant that should be preserved through to the corresponding Use-Call. Specifically, a file certified to be non-existent by a CU-call should remain non-existent until the Use-call creates it. Similarly, a file certified to exist by a CU-call should remain the same file until the Use-call (by the same user) accesses it. Identifying and preserving these two invariants (non-existence of a file and the mapping from a pathname to a file object) are the main goals of EDGI approach.

The EDGI design treats an invariant as a sophisticated *lock*. The user invoking a CU-call becomes the owner of the lock, and the lock is usually held by the same user through the Use-call. Due to the complications of Unix file system, the invariant handling is more complicated than a normal lock compatibility table. Therefore, we represent the invariant handling using ECA rules, as explained in the following section. We note that we only use ECA rules as a model, since our implementation does not support general-purpose rule processing.

4.2. Invariant Maintenance

The EDGI approach adopts a modular design and implementation strategy by separating the EDGI invariant processing from the existing kernel. The invariant-related information is maintained as extra state information for each file object. When an invariant-related event is triggered, the corresponding set of conditions is evaluated and if necessary, appropriate actions are taken to maintain the invariant.

The invariant-related information for each file object includes its state (free or actively used), a tainted flag, invariant holder user id and a process list. In detail:

- refcnt – the number of active processes using the file object. When refcnt = 0, the file object is free.
- tainted – when refcnt > 0, this flag means whether the name to disk object binding can be trusted.
- fsuid – the user id of the processes that are actively using the file object.
- gh_list – a doubly-linked list, in which each node contains a process id and the timestamp of the last system call made by the process on the file object.

Two kinds of events trigger condition evaluation:

- File system calls such as **access, open, mkdir**, etc.
- Process operations: **fork, execve, exit**.

The conditions evaluated by each event and their associated actions are summarized in Table 3 (f denotes the file object). The conditions refer to the file object status (whether the invariant is the non-existence of the file or the file object mapping), and actions include the creation, removal and potentially more complex invariant maintenance actions.

4.3. Inferring Invariant Scope

An astute reader may have noticed that the invariant maintenance rules in Table 3 are not restricted to a TOCTTOU pair, but extend to a sequence of file system calls. Our discussion has been restricted to TOCTTOU pairs so far. In some programs, multiple accesses to the same file are made through additional Use-calls. Since this is a legitimate use of files, the EDGI system must maintain the same invariant through all the Use-calls of such a sequence by the same user. During the time such a sequence of accesses exists, the file object is said to be *actively used*. Otherwise the file object is said to be *free*.

The interval during which the file object is actively used forms the scope of its invariant. The scope varies in length, depending on the number of consecutive Use-calls made by the application. Consequently, a significant technical challenge is to correctly identify this scope - the boundaries of the TOCTTOU vulnerability window of the application. Since current Unix-style file systems are oblivious to such application-level semantics, we need to *infer* the scope, so no changes are imposed on the applications or the file system interfaces.

The inference of invariant scope is guided by the CUU model, which specifies the initial TOCTTOU pair explicitly. The Use-call of the initial pair becomes the CU-call of the next pair, completed by the following Use-call. Let us assume that the CUU model correctly captures the TOCTTOU problem. Intuitively, the initial pair can be considered the base of an induction proof, guaranteeing the maintenance of the invariant from the CU-call through the Use-call. The additional Use-calls become the steps of the induction. (Details of the proof can be found elsewhere [24].) The sequence continues until the program ends, a time-out or preemption occurs (see Section 4.4).

Table 3: Invariant Maintenance Rules in EDGI

Name	Event	Condition	Action
Incarnation rule	Any system call on f	refcnt == 0	Set f's state as actively used (refcnt++); set its tainted flag as false, fsuid as current user id, record current pid and current system time in the gh_list.
Reinforcement rule	Any system call on f	refcnt > 0 and fsuid == current user id and tainted == false	Record current pid and current system time in the gh_list.
Abort rule	Any system call on f	refcnt > 0 and fsuid == current user id and tainted == true	Record current pid and current system time in the gh_list. Return an error.
Root preemption rule	Any system call on f	refcnt > 0 and fsuid != current user id and current user id == root	Remove all invariant holders information from the gh_list; set f's fsuid as current user id, set refcnt as 1, tainted as false, record current pid and current system time in the gh_list.
Owner preemption rule	Any system call on f	refcnt > 0 and fsuid != current user id and current user id != root and fsuid != root and current user is the owner of f	Remove all invariant holders information from the gh_list; set f's fsuid as current user id, set refcnt as 1, tainted as false, record current pid and current system time in the gh_list.
Invariant maintenance rule 1	Any system call in the RemovalSet (3.2) on f	refcnt > 0 and fsuid != current user id	Traverse the gh_list to get the latest timestamp t, compute the interval between t and current time, if it is less than threshold MAX_AGE, deny the current request, otherwise grant the current request and set tainted as true.
Invariant maintenance rule 2	Any system call in the CreationSet (3.2) on f	refcnt > 0 and fsuid != current user id	Traverse the gh_list to get the latest timestamp t, compute the interval between t and current time, if it is less than threshold MAX_AGE, deny the current request, otherwise grant the current request and set tainted as true.
Clone rule	fork (parent, child)	True	For each file object that has parent in its gh_list, record child and current system time, and increment the refcnt.
Termination rule	Exit	True	Remove current pid from the gh_list of each file object that has it on its gh_list, and decrement the corresponding refcnt.
Distract rule	Execve	True	Remove current pid from the gh_list of each file object that has it on its gh_list, and decrement the corresponding refcnt.

4.4. Remaining Issues

There are some additional issues that need to be resolved for an actual implementation. First, if we consider the invariants as similar to locks, then the question of dead-lock and live-lock arises. For example, it is possible that an invariant holder is a long-running process which only touches a file object at the very beginning and then never uses it again. Consequently, a legitimate user may be prevented from creating/deleting the file object for a long time, resulting in denial of service. This problem can be addressed by a time out mechanism. If an invariant holder process does not access a file object for an exceedingly long time, the invariant will be temporarily disabled to allow other legitimate users to proceed. (Timeout is discussed in Section 6.2.)

If the time-out results in simple preemption (i.e., breaking the lock), then this method may be used to attack very long application runs. To prevent the preemption-related attack, we use a *tainted* bit to mark the preemption. After a preemption-related file creation or deletion, the invariant no longer holds. EDGI marks the file object as *tainted*, so the next access request from the original invariant holder will be aborted.

The second and related problem is the relationship between the current invariant holder and the next process attempting to access the file object. Up to now, we have assumed a symmetric relationship, without distinguishing

legitimate users from attackers. In reality, we know some processes are more trustworthy than others. Specifically, in Unix environments we trust the file object owner and root processes completely. Consequently, we allow these processes to "break the lock" by preempting other invariant holders. Concretely, when the file object owner or root process attempt to access a file object, they immediately become the invariant holder, and the invariant for the former holder is removed.

The third issue is the inheritance of invariants by children processes. For example, after a user process checks on a file object and becomes an invariant holder, it spawns a child process, and terminates. In the mean time, the child process continues, and uses the file object. In the simple solution, the invariant is removed when the owner (parent) process terminates. In this case an attacker can achieve a TOCTTOU attack before the child process uses the file. Thus we must extend the scope of invariants to the child process at every process creation. This invariant inheritance extension is analogous to the invariant scope extension discussed in Section 4.3.

A final question is whether the EDGI approach is a complete solution, capable of stopping all TOCTTOU attacks. In this paper, we describe a systematic design (Section 4) and implementation (Section 5), based on the CUU model. We outline here an informal argument for the correctness of the design, i.e., EDGI protects all the TOCTTOU pairs identified by the CUU model. For every file system call, the rules summarized in Table 3 are checked and followed. The first time a CU-call is invoked on a file object, that user becomes the file object's invariant holder. At any given time there is at most one invariant holder for each file object. Users other than the invariant holder are not allowed to create or remove the file object (including changes to mapping between the name and disk objects). The EDGI defense is designed to stop all TOCTTOU pairs identified by the CUU model. The proof of CUU model completeness [24] is beyond the scope of this paper.

5. Linux Implementation of EDGI

We have implemented the design described in the previous section in the Linux file system. The implementation consists of modular kernel modifications to maintain the invariants for every file object and its user/owner. We outline the process that remembers the invariant holder of each file object (Section 5.1) and then the maintenance of the invariants (Section 5.2).

5.1. Invariant Holder Tracking

Invariant holder tracking is accomplished by maintaining a hash table of pathnames that keeps track of the processes that are actively using each file object. The index to this hash table is the file pathname, and for each entry, a list of process ids is maintained. Our modular implementation

```
    Input: dentry d
    Output: 0 – succeed, -1 – the binding of d is tainted.
 1  if d.refcnt = 0
 2  then d.fsuid ← current user id, record current pid and
    current time in d.gh_list, d.refcnt++, d.tainted ← false,
    return 0.
    else
 3      if d.fsuid = current user id
 4      then record current pid and current time in
        d.gh_list, if d.tainted = false
 5                  then return 0
 6                  else  return -1.
        else
 7        if current user id = root
 8        then remove all invariants on d.gh_list,  d.fsuid ←
          root, record current pid and current time in
          d.gh_list, d.refcnt ←1, d.tainted ← false, return 0.
          else
 9          if d.fsuid = root
10          then return 0.
            else
11              if current user id is the owner of d
12              then remove all invariants on d.gh_list,
                d.fsuid ← current user id, record current pid
                and current time in d.gh_list, d.refcnt←1,
                d.tainted ← false, return 0.
13              else return 0.
```

Figure 1: Invariant Holder Tracking Algorithm

augments the existing directory entry (dentry) cache code and extends its data structures with the fields introduced in Section 4.2: fsuid, refcnt, tainted, gh_list.

Before a system call uses a file object by name, it first needs to resolve the pathname to a dentry. Our implementation instruments the Linux kernel to call the invariant holder tracking algorithm after each such pathname resolution. There are two possible approaches to implementing this algorithm. The first is to instrument the body of every system call (e.g., **sys_open**) that uses a file pathname as argument. The second is to instrument the pathname resolution functions themselves (in the Linux case, **link_path_walk** and **lookup_hash**).

The first approach has the disadvantage that instrumented code has to spread over many places, making testing and maintenance difficult. Although techniques such as Aspect Oriented Programming (AOP) [22] could help, we were unable to find a sufficiently robust C language aspect weaver tool that can work on Linux kernel. The second approach has the advantage that only a few (in the Linux case, exactly two) places need to be instrumented, making the testing and maintenance relatively easy. We chose the second approach for our implementation.

The invariant holder tracking algorithm GH is shown in Figure 1. This algorithm effectively implements the

rules summarized in Table 3, and it is called right before **link_path_walk** and **lookup_hash** successfully returns.

Line 1-2 of the invariant holder tracking algorithm addresses the situation where a new invariant holder is identified: invariant related data structure is initialized, including the invariant holder user id (fsuid), the invariant holder process id, the tainted flag, and a timestamp. After these steps, the invariant maintenance part (Section 5.2) will start applying this invariant. We can see that the same sequence also occurs in Line 8 and 12, where a new invariant holder is decided due to preemption.

Line 3-6 address the situation where an existing invariant holder accesses the file object again. Notice that the tainted flag is checked to abort the invariant holder process if the name to disk binding of the file object has been changed by another user's process (See 5.2).

Line 8 corresponds to the preemption of invariant from a normal user to the root discussed in Section 4.4. Similarly, line 12 handles the preemption by file object owner.

The invariant holder tracking algorithm needs the current process id and current user id runtime information, which are obtained from the *current* global data structure maintained by the Linux kernel.

5.2. Invariant Maintenance

The second part of implementation is invariant maintenance by thwarting the attacker's attempt to change the name to disk binding of a file object, which in turn is achieved by deleting or creating a file object. We instrumented two kernel functions to perform invariant checks:

- **may_delete**(d): this function is called to do permission check before deleting a file object d. We add invariant checking after all the existing checks have been passed: If d.refcnt > 0 and the current user id is not the same as d.fsuid, traverse d.gh_list to get the last access timestamp; if it is younger than MAX_AGE, return –EBUSY (file object in use and cannot be deleted). Otherwise set d.tainted as true and return 0.
- **may_create**(d): this function is called to do permission check before creating a file object, similar invariant checking is added after all the existing checks have been passed.

The **may_create** kernel function is called by all the system calls in the CreationSet (Section 3.2) and the **may_delete** function is called by all the system calls in the corresponding RemovalSet. These invariant checks implement the Invariant Maintenance Rules 1 and 2 in Table 3.

5.3. Engineering of EDGI Software

Table 4 shows the size of EDGI implementation in Linux kernel 2.4.28. The changes were concentrated in one file (dcache.c), which was changed by about 55% (LOC means lines of code). The other changes were small, with less than 5% change in one other file (namei.c), plus single-line changes in three other files. This implementation of less than 1000 LOC was achieved after careful control and data flow analysis of the kernel, plus some tracing. We consider this implementation to be highly modular and relatively easily portable to other Linux releases.

From top-down point of view, the methodical design and implementation process benefited from the CUU model as a starting point. Then, the ECA rules facilitated the reasoning of invariant maintenance. The rules were translated into the Invariant Holder Tracking algorithm. These steps give us the confidence that the invariants are maintained by EDGI software.

Conversely, from a bottom-up point of view, the Linux kernel was organized in a methodical way. For example, it has exactly two functions (**may_delete** and **may_create**) controlling all file object status changes. By guarding these two functions, we were able to guard all 224 TOCTTOU pairs identified by the CUU model. This kind of function factoring in the Linux kernel contributed to the modular implementation of EDGI.

Table 4: Linux Implementation of EDGI

Source File	Modified Places	Original LOC	Added LOC
fs/dcache.c	4	1307	749
fs/namei.c	5	2047	84
fs/exec.c	1	1157	1
kernel/exit.c	1	602	1
kernel/fork.c	1	896	1

6. Experimental Evaluation of EDGI

6.1. Discussion of False Negatives

The EDGI system design follows the CUU model. In Section 4.4 we included an informal argument for the completeness of the CUU model, details of which can be found in [24]. If the ECA rules summarized in Table 3 captures all the TOCTTOU pairs identified by the CUU model, and the invariant holder tracking algorithm in Figure 1 implements all the rules in Table 3, and our Linux kernel implementation (Section 5) is correct, then our implementation should have zero false negatives.

We have run all the attack experiments we could find, including known TOCTTOU vulnerabilities such as *logwatch* 2.1.1 [19] and new vulnerabilities recently detected, including *rpm*, *vi/vim*, and *emacs*. In all the experiments the EDGI system is able to stop the attacker program.

One exception to the invariant maintenance rules is the preemption by programs running as root, which are allowed to gain the invariant and change file object status at

will. We consider this exception to be safe, since if an attacker has already obtained root privileges, there is no further gain for using TOCTTOU attacks.

6.2. Discussion of False Positives

As mentioned in Section 4.4, our conservation maintenance of invariants may introduce long delays, if an invariant holder runs for a long time. These long delays can be considered a kind of false positives, since they may or may not be necessary. Our implementation introduces a time-out mechanism to mitigate this problem. If another user's process wants to create/delete the file object and encounters the last access time by the invariant holder to be older than the time-out period, the new process is allowed to preempt the invariant and the file object is marked as tainted. If the original invariant holder attempts to use the file object again, then we have found a real conflict. The current implementation aborts the original invariant holder, although other design choices are possible.

The determination of a suitable time-out period, called MAX_AGE in Table 3, is probably dependent on each specific workload and a research question. If it is too short, an attacker may use it to abort a long running legitimate process by attempting to write to a shared file. If it is too long, another legitimate process may be delayed for a long time. We have experimentally chosen a MAX_AGE of 60 seconds.

6.3. Overhead Measurements

We use a variant of the Andrew benchmark [10] to evaluate the overhead introduced by EDGI defense mechanism. The benchmark consists of five stages:

1. Recursively create 110 directories with **mkdir**.
2. **copy** 744 files (total 12MB).
3. **stat** 1715 files and directories.
4. **grep** these files and directories (total 26MB).
5. Compile 150 source files.

The experiments were run on a Pentium III 800MHz laptop with 640MB memory, running Red Hat Linux in single user mode. We report the average and standard deviation of 20 runs for each experiment in Table 5, which compares the measurements on the original Linux kernel and on the EDGI-augmented Linux kernel. The same data is shown as bar chart in Figure 2.

The Andrew benchmark results show that EDGI generally has a moderate overhead. The only exception is **stat**, which has 47% overhead. The explanation is that **stat** takes less time than other calls (such as **mkdir**), but the extra processing due to invariant holder tracking (now part of pathname resolution) has a constant factor across different calls. This constant overhead weighs more in short system calls such as **stat**. Fortunately, **stat** is used relatively rarely, thus the overall impact remains small.

PostMark benchmark [12] 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. PostMark first tests the speed of creating new files, and 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 40.0 seconds. On EDGI-augmented kernel, with all the same parameter settings, the running time is 40.1 seconds (Again these results are averaged over 20 rounds). So the overhead is 0.25%. This result corroborates the moderate overhead of EDGI.

Table 5: Andrew Benchmark Results (in milliseconds)

Functions	Original Linux	Modified Linux	Overhead
mkdir	6.35 ±0.21	6.43 ±0.19	1.3%
copy	217.0 ±1.5	218.6 ±1.4	0.7%
stat	132.0 ±1.9	193.6 ±0.8	47%
grep	777.0 ±4.3	870.1 ±5.3	12%
compile	53,971 ±434	55,615 ±367	3.0%

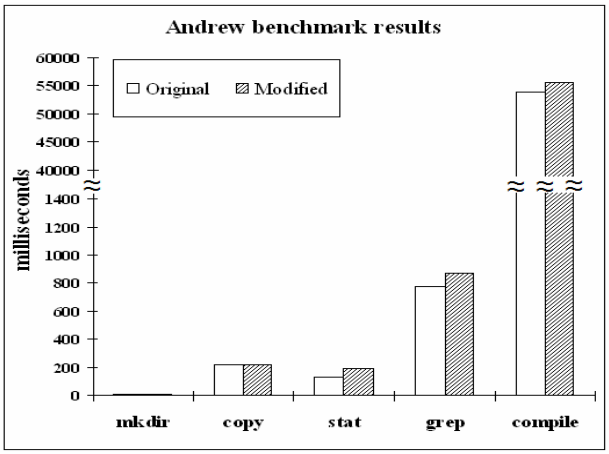

Figure 2: Andrew Benchmark Results

7. Conclusion

TOCTTOU (Time Of Check To Time Of Use) is a long-standing security problem that is both numerous and serious [15][17]. Compared to other vulnerabilities such as buffer overflow, TOCTTOU attacks are complex, requiring two-step accesses to a file by a victim process (checking and use of the file) combined with an attacker process changing the file object mapping to storage objects precisely in-between the two steps.

In this paper, we describe a model-based, event-driven approach (called EDGI) to prevent the exploitation of TOCTTOU vulnerabilities. The main idea of EDGI is to guard the file object invariant created by the checking step (e.g., the pathname mapping to disk objects) through the completion of the use step, thus protecting the potential victim from shared access by any attacker.

The EDGI defense has several advantages. First, EDGI is a systematically developed defense mechanism (based on the CUU model [23][24]) with careful design (using ECA rules) and implementation. Assuming the completeness of the CUU model, EDGI can stop all TOCTTOU attacks (no false negatives). Second, with careful handling of issues such as inference of invariant scopes and timeouts, EDGI allows very few false positives. Third, it does not require changes to applications or file system API by inferring automatically the scope of invariants to be protected. Fourth, our modular implementation on Linux kernel and its experimental evaluation show that EDGI carries little additional overhead.

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 ISSSE 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] R. Bisbey and D. Hollingsworth. Protection Analysis Project Final Report. ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute, May 1978.

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

[4] 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.

[5] 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), Washington, DC, November 2002.

[6] 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.

[7] 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.

[8] 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.

[9] 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).

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

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

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

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

[19] Security holes in logwatch. http://xforce.iss.net/xforce/xfdb/8652.

[20] Dennis R. McCarthy, Umeshwar Dayal. The Architecture Of An Active Data Base Management System. SIGMOD Conference 1989: 215-224

[21] David Harel. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3):231–274, June 1987.

[22] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. Aspect-Oriented Programming. Proceedings European Conference on Object-Oriented Programming, 1997.

[23] Jinpeng Wei, Calton Pu. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study. 4th USENIX Conference on File and Storage Technologies (FAST '05), San Francisco, CA, December 2005.

[24] Calton Pu, Jinpeng Wei. A theoretical study of TOCTTOU problem modeling. Submitted for publication.

[25] S. Chen, J. Xu, E. C. Sezer, P. Gauriar and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. USENIX Security Symposium, Baltimore, MD, August 2005.