Where Do You Want to Go Today?
Escalating Privileges by Pathname Manipulation*

Suresh Chari Shai Halevi Wietse Venema
IBM T.J. Watson Research Center,
Hawthorne, New York, USA

Abstract

We analyze filename-based privilege escalation attacks, where an attacker creates filesystem links, thereby "tricking" a victim program into opening unintended files. We develop primitives for a POSIX environment, providing assurance that files in "safe directories" (such as `/etc/passwd`*) cannot be opened by looking up a file by an "unsafe pathname" (such as a pathname that resolves through a symbolic link in a world-writable directory). In today's UNIX systems, solutions to this problem are typically built into (some) applications and use application-specific knowledge about (un)safety of certain directories. In contrast, we seek solutions that can be implemented in the filesystem itself (or a library on top of it), thus providing protection to all applications.*

Our solution is built around the concept of pathname manipulators*, which are roughly the users that can influence the result of a file lookup operation. For each user, we distinguish* unsafe pathnames *from* safe pathnames *according to whether or not the pathname has any manipulators other than that user or* `root`*. We propose a* `safe-open` *procedure that keeps track of the safety of the current pathname as it resolves it, and that takes extra precautions while opening files with unsafe pathnames. We prove that our solution can prevent a common class of filename-based privilege escalation attacks, and describe our implementation of the* `safe-open` *procedure as a library function over the POSIX filesystem interface. We tested our implementation on several UNIX variants to evaluate its implications for systems and applications. Our experiments suggest that this solution can be deployed in a portable way without breaking existing systems, and that it is effective against this class of pathname resolution attacks.*

*This work was supported in part by the Department of Homeland Security under grant FA8750-08-2-0091.

1. Introduction

In this work we take another look at the problem of privilege escalation via manipulation of filesystem names. Historically, attention has focused on attacks against privileged processes that open files in directories that are writable by an attacker. One classical example is email delivery in the UNIX environment (e.g., [9]). Here, the mail-delivery directory (e.g., `/var/mail`) is often group or world writable. An adversarial user may use its write permission to create a hard link or symlink at `/var/mail/root` that resolves to `/etc/passwd`. A simple-minded mail-delivery program that appends mail to the file `/var/mail/root` can have disastrous implications for system security. Other historical examples involve privileged programs that manipulate files under the world-writable `/tmp` directory [11], or even in a directory of the attacker's choice [10].

Over time, privileged programs have implemented safety mechanisms to prevent pathname resolution attacks. These mechanisms, however, are tailored specifically to the program's purpose, are typically implemented in the program itself, and rely on application-specific knowledge about the directories where files reside. We believe, however, that the application is fundamentally the wrong place to implement these safety mechanisms.

Recent vulnerability statistics support our position. The US National Vulnerability Database [16] lists at least 177 entries, since the start of 2008, for symlink-related vulnerabilities that allow an attacker to either create or delete files, or to modify the content or permissions of files. No doubt, the vast majority of these entries are due to application writers who simply were not aware of the problem. However, there are even vulnerabilities in system programs, which are typically better scrutinized. For example, an unsafe file `open` vulnerability was reported in the `inetd` daemon in Solaris 10 [12] when debug logging is enabled. This daemon runs with `root` privileges and logs debug messages to the file `/var/tmp/inetd.log` if that file exists. The file is opened using `fopen(DEBUG_LOG_FILE,`

"r+"). Since /var/tmp is a world writable directory a local unprivileged user can create a link to any file on the system, and overwrite that file as root with inetd debug messages. A similar example, related to unsafe unlink operation, is a reported vulnerability in the Linux rc.sysinit script [13] in the initscripts package before version 8.76.3-1. That vulnerability could be used by unprivileged users to delete arbitrary files by creating symbolic links from specific user-writable directories.

In addition to these examples, experiments that we run in the course of this work uncovered a number of (latent) privilege escalation vulnerabilities, where system processes write or create files as root in directories that are writable by unprivileged system process. In these cases, a compromise of the unprivileged system process could result in further privilege escalation. These vulnerabilities are described in Section 5.3.

These examples demonstrate that it is unrealistic to expect every application (or even every "important application") to implement defenses against these attacks. We contend that a system-level safety net would be more effective at stopping these problems than trying to fix every affected application, or trying to educate current and future generations of application writers. In a world where applications (and their fragments) are used in environments that are vastly different from what the application designers had in mind, it is unreasonable to expect that the applications themselves will distinguish between files that are safe to open and ones that are not.

In this work we seek a general-purpose mechanism that can be implemented in the file system or in a system library, that allows programs to open files that exist in an "unsafe" environment, knowing that they will not be "tricked" into opening files that exist in a "safe" environment. Specifically, we show how such a mechanism can be implemented over POSIX filesystems.

In a nutshell, our solution can be viewed as identifying "unsafe subtrees" of the filesystem directory tree, and taking extra precautions whenever we visit any of them during the resolution of a pathname. Roughly, a directory is unsafe for a certain user if anyone other than that user (or root) can write in it. Our basic solution consists of resolving a pathname component by component, enforcing the conditions that once we visit an unsafe node, in the remainder of the path we will no longer follow symbolic links or allow pathname elements of '..', nor will we open a file that has multiple hardlinks. Thus, once we resolve through an unsafe node, we will not visit nodes that exist outside the subtree rooted at that node.[1]

In contrast with many prior works on filename-based attacks, our work is *not primarily focused on race conditions* (such as access/open races [20, 4]). Rather, we directly addresses the privilege-escalation threat, which is the main motivation for many of these attacks. Here we focus on the pathname resolution mechanism, identify a simple security property that can be met *even in the presence of race conditions*, and show that this property can be used to prevent privilege-escalation attacks.

[1] We describe in Section 6.1 a more permissive variant that still provides the same protection against privilege-escalation attacks.

1.1. Our contribution

We focus on tightening the connection between files and their names. In most filesystems, programs access files by providing names (the pathnames), and rely on the filesystem to resolve these names into pointers to the actual files (the file handles). Unfortunately, the relation between files and their names in POSIX filesystems is murky: Files can have more than one name (e.g., due to hard or symbolic links), these names can be changed dynamically (e.g., by renaming a directory), filename resolution may depend on the current context (e.g., the current working directory), etc. This murky relation obscures the semantics of the name-to-file translation, and provides system administrators and applications writers with ample opportunities to introduce security vulnerabilities. Our solution builds on the following concepts:

- Ignoring the partition to directories and subdirectories, we view the entire path as just one name and examine its properties. We introduce the concept of *the manipulators* of a name, which roughly captures "anyone who can change the outcome of resolving that name." In POSIX filesystems, the manipulators of a path are roughly the users and groups that have write permission in any directory along this path. More precisely, U belongs to the manipulators of a name if the resolution of that name visits any directory that is either owned by U or that U has write permissions for.

- Using the concept of manipulators, we distinguish between *safe names* and *unsafe names*. Roughly, a name is safe for some user if only that user can manipulate it. Specializing this concept to UNIX systems, we call a name "system safe" if its only manipulator is root, and call it "safe for U" if the only manipulators of it are root and U. For example, typically the name /etc/passwd is "system safe", the name /home/joe/mbox is safe for user joe, and the name /var/mail/jane is not safe for anyone (as /var/mail is group or world writable).

- Once we have safe and unsafe pathnames, we can state our main security guarantee. We provide a procedure safe-open that ensures the following property:

If a file has safe names for user U, then safe-open *will not open it for U using an unsafe name.*

As we show in the paper, this property can be used to ensure that no privilege escalation via filesystem links occurs. For example, if /etc/passwd is system-safe, then no process running as root will safe-open this file due to a hard link or symbolic link that could have been created by a non-root process. In particular, a "simple minded" mail delivery program that uses our safe-open will be protected against the attack in the example from above. Also, we verified that this guarantee is sufficient to protect against the documented vulnerabilities in CVE.

We implemented our safe-open procedure as a library function over POSIX file systems, and also generalized it to other POSIX interfaces that resolve pathnames such as safe-unlink, safe-chmod, etc. (cf. Section 4). We performed whole-system measurements with several UNIX flavors, and find that system-wide safe pathname resolution can be used without "breaking" real software. During these measurements we also uncovered a number of new (latent) vulnerabilities (cf. Section 5.3), that would be fixed using our safe-open.

We mention that our work on safe pathname resolution was done in the context of a more general framework. In a companion paper [6] we describe an abstract filesystem interface where file operations are permitted only on the names with which the file was created. We then describe an implementation that uses the safe resolution procedure described in this paper, and formally prove that it realizes the abstract filesystem interface. (That formal proof is carried out in the framework of "universal composability" [5], which is used in cryptography to prove that a system realizes its specifications in all adversarial settings.)

1.2. Related Work

Much of the prior work on pathname safety has focused on time-of-check/time-of-use race vulnerabilities (TOCTTOU) in privileged programs [1, 2, 8, 3, 20, 4]. Our work is not focused on this problem, instead it directly addresses the privilege-escalation issue that underlies many of these race-condition vulnerabilities: Rather than trying to prevent race conditions, we modify the name-resolution procedure to ensure that privilege-escalation cannot happen even if an attacker is able to induce race conditions.

In early analysis of filesystem race vulnerabilities in privileged programs, Bishop discusses safe and unsafe pathnames, and introduces a can-trust library function that determines whether an untrusted user could change the name-to-object binding for a given pathname [1]. Later, a more formal analysis with experimental validation was done by Bishop and Dilger [2].

Our safe-open function implements a user-level pathname resolver that examines pathname elements one by one; its structure is therefore similar to that of the access-open function by Tsafrir *et al.* [20, 21]. While their user-level name resolver applies access checks to each path element in a manner that defeats race attacks, our safe-open function is not primarily concerned with access checks. Instead, we apply a "path safety" check to each path element.[2]

In the context of system call introspection monitors for TOCTTOU vulnerabilities, Garfinkel [14] considered remedies which could also potentially apply to the problem of unsafe pathname resolution. These remedies include disallowing the creation of symlinks to files which the calling process does not have write permissions to, as well as denying access to files through symlinks. As noted in his paper, these solutions can mitigate the problem but they do not solve it. For instance, they do not address pre-existing symlinks, and fail in the face of symlinks in intermediate components of the pathnames. In contrast, our solution directly addresses the underlying problem of unsafe pathname resolution.

Another approach to system call introspection is implemented in the Plash sandboxing system [18]. Here, a replacement C library delegates file-system operations to a fixed-privilege, user-level, process that opens files on behalf of monitored applications and that enforces a confinement policy. While this approach provides great expressiveness, it would not be suitable for system-wide deployment as envisaged with our safe-open function. (For example it is not clear how to address privilege changes by the calling process, or how this solution scales with the number of processes.)

Addressing filename manipulations is in some ways complementary to dealing with the "confused deputy" problem: Both problems are used as a vehicle for privilege escalation, and some aspects of the solution are common, but the problems themselves appear to be different: For example, the "simple minded" mail-delivery program from above knows that it uses its root privileges for writing /var/mail/root, so in this sense it is not a confused deputy (since it is not being tricked into using some extra privilege that it happens to hold). The problems with UNIX privilege-managing functions were systematically analyzed by Chen, Wagner and Dean; these authors also provide a more rational API for privilege management [7]. Their approach was later extended by Tsafrir, Da Silva and Wagner

[2]Our solution could have been implemented using a variant of the general framework from [21, Sec. 7], but that variant would have to be considerably more complex to deal with issues such as change of privileges or permissions, thread safety, etc.

to include also group privileges [19].

Mazieres and Kaashoek advocate a better system call API that among others allows processes to specify the credentials with each system call [15]. Our `safe-open` function could benefit from such features (especially when opening files on behalf of `setgid` programs, cf. Section 6.3).

2. Names, Manipulators, and Safe-Open

For presentation simplicity, we initially consider only a simplified setting where (a) all filenames are absolute paths, (b) every filesystem is mounted only once in the global name tree, and (c) no concurrency issues are present. (The last item means that we simply assume that no permission changes occur concurrently with our name resolution procedure.) We discuss relative pathnames at the end of this section, multiple mount points and dynamic permissions are discussed in Section 3.

2.1. Names and Their Manipulators

Roughly speaking, a *manipulator* of a name is any entity that has filesystem permissions that can be used to influence the resolution of that name. A manipulator can create a name (i.e., cause the filesystem to resolve that name to some file), delete it (causing name resolution to fail) or modify it (causing the name to be resolved to a different file). In the context of POSIX systems, a *manipulator* of a path in a POSIX filesystem is any `uid` that has write permission in — or ownership of — any directory that is visited during resolution of that path.[3]

For example, consider the files `/etc/passwd`, `/home/joe/mbox`, and `/tmp/amanda/foo` from a common UNIX system. The permissions of the relevant directories are:

```
drwxr-xr-x root root /
drwxr-xr-x root root /etc
drwxr-xr-x root root /home
drwx------ joe  joe  /home/joe
drwxrwxrwt root root /tmp
drwxr-xr-x root root /tmp/amanda
```

Then the only manipulator of the name `/etc/passwd` is `root` (since only `root` can write in either `/` or `/etc/`), and the manipulators of the name `/home/joe/mbox` are `root` and `joe`. On the other hand, all the users on that machine are manipulators of `/tmp/amanda/foo`, since everyone can write in `/tmp`.[4] Moreover, if we had the symbolic links:

```
/home/joe/link1 -> /etc/passwd
/home/joe/link2 -> /tmp/amanda
```

then the manipulators of the name `/home/joe/link1` are `root` and `joe`, and the manipulators of the name `/home/joe/link2/foo` include all the users on that machine (since resolution of this last name goes through the world-writable `/tmp`).

We note that this description is "static", in that it refers to the permission structure as it exists at a given point in time. Nonetheless, in Section 3.2 we show that our solution (which is based on this "static" notion) prevents privilege escalation via pathname manipulations even in settings where the filesystem (and its permissions) can change in a dynamic fashion. Roughly speaking, this is because in POSIX systems only manipulators of a path can add new manipulators to it, and no manipulator can remove itself from the set of manipulators of a path.[5]

Safe and unsafe names. For POSIX systems, we say that a name is *system-safe* (or safe for `root`) if `root` is the only manipulator of that name. A name is safe for some other `uid` if its only manipulators are `root` and `uid`. Otherwise the name is *unsafe*.

2.2. The `Safe-Open` Procedure

Our `safe-open` procedure is a refinement of the safety mechanisms used by the Postfix mail system [22] to open files under the world-writable directory `/var/mail`. The basic approach taken by Postfix is to verify that the opened file is not a symbolic link and does not have multiple hard links. This approach works for the special case of `/var/mail`, but it is not quite applicable as a general-purpose policy, for two reasons:

It is too strict. There are cases where applications have a legitimate need to open a file with multiple hard links or a symbolic link.[6] Moreover, blanket refusal to open files with multiple hard links would enable an easy denial-of-service attack: simply create a hard link to a file, and no one will be able to open it.

It is not strict enough. Refusing to open links does not provide protection against manipulation of higher-up directories. For example, consider a program that tries to open the file `/tmp/amanda/foo`. Even if this file

[3] See Section 6.3 for a discussion about `gids`.

[4] The directory `/tmp` typically has the sticky bit set, which prevents non-`root` users from removing other user's files from `/tmp`. But it does not prevent users from moving other user's files *into* `/tmp`. For this reason, everyone must be considered a manipulator of the directory `/tmp/amanda`, even though this directory can be removed only by `root`.

[5] The last statement depends on the fact that only `root` can use the `chown` system call.

[6] For example, old implementations of Usenet news kept a different directory for every newsgroup and a different file for every article, and when an article was sent to more than one group, then it will be stored with multiple hard links, one from each group where this article appears.

does not have multiple links, it may still not be safe to open it: For example, the attacker could have created /tmp/amanda/ as a symbolic link to /etc, and the program opening /tmp/amanda/foo will be opening /etc/foo instead.

To implement a general-purpose safe-open, we therefore refine these rules. Our basic procedure is as follows: While resolving the name, we keep track of whether the path so far is *safe* or *unsafe* for the effective uid of the calling process. When visiting a directory during name resolution, we call it unsafe if it is group- or world-writable, or if its owner is someone other than root or the current effective uid of the calling process (and otherwise we call it safe). When resolving an absolute path, we start at the root in safe mode (if the root directory is safe). As long as the resolver only visits safe directories, we are in a safe mode, can follow symbolic links or '..', and can open files with multiple hard links. However, once the resolver visits an unsafe directory, we switch to unsafe mode, and in the remainder of the path, disallow symbolic links or '..', and refuse to open a file with multiple hard links.[7] We note the following about this solution:

- A safe name that can be opened by POSIX open will also be opened by safe-open: If a name is safe then the safe-open procedure will visit only safe directories, and therefore will not abort due to symlinks or multiple hardlinks. Any directory that is visited during name resolution in open will also be visited by safe-open, and the file will eventually be opened.
- A file with only one name (which can be opened by POSIX open) will be opened by safe-open: This is similar to the previous argument, if the file has just one name then this name cannot include symbolic links and the file cannot have multiple hard links. Hence safe-open will succeed in opening it if POSIX open does.
- For a file with multiple unsafe names, each of these names may or may not be opened by safe-open. Note that if many names point to the same file, then there must be "merge points" where either we have a symbolic link pointing to a directory (or to the file) or multiple hard links pointing to this file. When safe-open resolves these names, it agrees to follow these "merge points" if it visited only safe directories before they occur, and refuses to follow them if it visited an unsafe directory.

  For one example, safe-open will agree to open the unsafe name /home/joe/link2/foo from Section 2.1 when running with effective uid of joe, since the "merge point" occurred while visiting the directory /home/joe/, still in a safe mode. On the other hand, safe-open will refuse to open this name when running with effective uid of root, since the directory /home/joe/ is not safe for root.

[7] See Section 6.1 for more permissive variants of this procedure.

Implementing this safe-open procedure in the filesystem itself (i.e., in the kernel) should be straightforward: All we need is to add a check for permissions and ownership on every directory, updating the safety flag accordingly. Arguably, this is the preferred mode of implementation, but it requires changes to existing filesystems. Alternatively, we describe an implementation of safe-open as a library function in user space. This implementation roughly follows the procedure of Tsafrir et al. [20, 21] for user-level name resolution, but adds to it the safe-mode vs. unsafe-mode behavior as described above. We discuss this implementation in Section 4.

Relative paths and openat. The procedure for resolving relative paths (or for implementing openat) is essentially the same as the one for absolute paths, except that we need to know if the starting point (e.g., the current working directory) is safe or not. In a kernel implementation, it is straightforward to keep track of this information by adding flags to the handle structure. Some care must be taken in situations where the directory permissions change (e.g., via chmod or chown) or when the privileges of the current process change, but no major problems arise there. Keeping track of this information in a library implementation is harder, but even there it is usually possible to get this information, and reasonable defaults can be used when the information is unavailable (e.g., after an exec call). We refer to Appendix A for more details about relative paths and openat.

3. Our Security Guarantee

Recall the security guarantee that we set out to achieve:

If a file has names safe for user U, then safe-open *will not open it for U using an unsafe name.*

In other words, if a file has both safe and unsafe names, then safe-open should fail on all the unsafe names. (At the same time it succeeds on all the safe names, as noted above.) We note again that as stated, this guarantee applies only to a static-permission model, where permissions and ownership of directories do not change during the name resolution. However, as we discuss at the end of this section, protection against privilege escalation attack is ensured *even when the attacker makes arbitrary permission changes for directories that it owns.* The only thing that we must assume

is that non-adversarial entities do not induce a permission-change race against our name resolution.[8] Our analysis below also assumes that each directory tree appears only once in the file system tree (i.e. no loop-back mounts, etc.), and that each directory has at most one parent (i.e., one hard link with a name other than '`.`' or '`..`').[9] A short discussion of mount points can be found later in this section.

We now turn to proving this security guarantee. Consider a file that has both safe and unsafe names (for a specific `uid`), fix one specific unsafe name, and we show that `safe-open` must fail when it tries to open that name (on behalf of a process with this effective `uid`). We distinguish two cases: either the file has just one hard link, or it has more than one.

- *Case 1: more than one hard link.* Note that when `safe-open` is called with the unsafe name, it will apply name resolution while checking the safety of the name as it resolves it. As the resolution of this name goes through a directory which is unsafe for this `uid`, then `safe-open` will arrive at the last directory in this name resolution in unsafe mode (assuming that it arrives there at all). Since the file has more than one hard link, `safe-open` will then refuse to open it.

- *Case 2: exactly one hard link.* In this case, there is a single path from the root to this file in the directory tree (i.e. we exclude names that contain symbolic links). Below we call this the "canonical path" for this file and denote it by `/dir1/dir2/.../dirn/foo`.

  Clearly, every pathname that resolves to this file must visit all the directories on the canonical path. (Moreover, the last directory visited in every name resolution must be `dirn`, since it holds the only hard link to `foo`.) Since we assume that the file has safe names for `uid`, it follows that all the directories in this canonical path must be safe for `uid`.

  Consider now the directories visited while resolving the unsafe name. Being unsafe, we know that the resolution of this name must visit some unsafe directory, and that unsafe directory cannot be on the canonical path. Therefore, during the resolution of an unsafe name, `safe-open` must visit some unsafe directory (and therefore switch to unsafe mode) before arriving at the final directory `dirn`.

  Consider the last directory *not on the canonical path* that was visited while resolving this unsafe name. We call this directory `dir0`. Then it must be the case that `safe-open` switched to unsafe mode when visiting `dir0` or earlier (because after `dir0` it only visited safe directories). Now, since the canonical path begins with the root '`/`', then `safe-open` could not descend into the canonical path from above. Hence moving from `dir0` to the next directory was done either by following a symbolic link or by following '`..`', but this is impossible since `safe-open` does not follow symbolic links or '`..`' when in unsafe mode.

This completes the proof of our security guarantee.

Multiple mount points. We note that all the arguments from above continue to hold even when a filesystem is mounted at multiple points in the global name space, as long as all the mount points are system-safe. However, our security guarantee breaks if we have the same filesystem mounted in several directories, some safe and others not. In this case, going down a "canonical" unsafe name for a file, we have no way of knowing that the same file also have a safe name (via a different mount point). The same problem arises when parts of the filesystem are exposed to the outside world, e.g., via NFS. In this case, what may appear as a safe directory to a remote user may be unsafe locally (or the other way around).

3.1. Using the Security Guarantee to Thwart Privilege Escalation

The security guarantee that we proved above provides one with an easy way of creating files that applications cannot be "tricked" into opening using adversarial links: Namely, create the file with a safe name. For example, if the name `/path-to/foo` is system safe, then no process running as `root` can use `safe-open` to open the same file with a name that includes a link that was created (or renamed, or moved to its current location) by a non-`root` user. This is because such a link would have to be created in (or moved to) an unsafe directory, making the name unsafe and causing `safe-open` (running as `root`) to fail on it.

This observation can be used to defeat privilege escalation attacks. Consider a file that needs to be protected against unauthorized access (where access can be read, write, or both). Hence the file is created with restricted access permissions. To ensure that this protection cannot be overcome by the attacker creating adversarial links, we create this file with a name that is safe for all the `uid`s that have access permission for it. (That is, if only one `uid` has access permission to the file then the name should be safe for that `uid`, and otherwise the name should be system-safe.)

We now claim that an attacker that cannot access the file, also cannot create a link that would be followed with

[8] The distinction between adversarial and non-adversarial entities is inherent in privilege-escalation attacks, since one must distinguish between privileges held by the attacker and those held by the victim(s).

[9] Nearly all contemporary POSIX implementations either do not allow processes to create additional hard links to directories (e.g., FreeBSD, Linux) or restrict this operation to the super-user (e.g., Solaris, HP-UX). A notable exception is MacOS.

`safe-open` by anyone with access permission for this file. Note that the attacker must have a different `uid` than anyone who can access the file.[10] Hence a directory where the attacker can create a link must be unsafe for anyone who can access the file, and therefore `safe-open` will not follow links off that directory.

3.2. Dynamic Permissions

The argument above covers the static-permission case, where permissions for directories do not change during the execution of `safe-open`. We now explain how it can be extended to the more realistic dynamic-permission model.

Consider a potential privilege-escalation attack, where an attacker that cannot access a certain file tries to cause a victim program to access that file on its behalf. Notice that in this scenario it must be the case that the attacker does not have `root` privileges, and also has a different effective-`uid` than the victim. (Otherwise no privilege escalation is needed — the attacker could access the file by itself.[10])

Consider now a file F that can be accessed by the effective-`uid` of the victim (denoted by U) but not by the effective-`uid` of the attacker (denoted by U'), consider a particular execution of `safe-open` by the victim, and assume that:

(a) at the time that the procedure is invoked, the file F has some name that is safe for U, and that name remains unchanged throughout the execution, and

(b) the pathname argument to `safe-open` is not a U-safe name for the file F when the procedure is invoked.

Under these conditions, we show that this `safe-open` procedure will not open the file F, *barring a concurrent filesystem operation by* `root` *or* U *on pathname elements that* `safe-open` *examines.* Put in other words, the attacker can only violate our security guarantee if it can induce a race condition *between two non-adversarial processes* (i.e., the `safe-open` procedure and another process with `uid` of either the victim or `root`). Assume therefore that these two conditions hold, and in addition

(c) neither `root` nor U did any concurrent filesystem operation on any pathname element examined by this `safe-open`.

We observe that any pathname element that `safe-open` examines and that resides in a U-safe directory at the time where the procedure was invoked, must remain in the same state throughout this `safe-open` execution. The reason is that being U-safe, only U and `root` have permissions to change anything in the directory, and by our assumption (c) neither of them made any changes to that pathname element.

Imagine now that the state of the filesystem is frozen at the time when the `safe-open` procedure is invoked, and consider the way the pathname argument to `safe-open` would be resolved. We have two cases: either all the directories visited by this hypothetical name resolution are U-safe, or some of them are not. The easy case is when all of them are U-safe: then it must be the case that the hypothetical name resolution does not resolve to the file F (or else it would be a U-safe name for F, contradicting our assumption (b)). But it is easy to show (by induction) that the same directories will be visited also in the actual name resolution, all of them would be in exactly the same state, and therefore also the actual name resolution as done by `safe-open` would not be resolved to F.

Assume, then, that the hypothetical name resolution would visit some unsafe directories, and let `dir0` be the first U-unsafe directory to be visited. The same easy inductive argument as above shows that all the directories upto (and including) `dir0` are also visited by the actual name resolution. We now know that the owner of `dir0` remains the same throughout the execution of `safe-open` (since by assumption (c) `root` did not make any changes in directories that were examined by `safe-open`). If the owner is different than U and `root`, then `safe-open` will switch to unsafe mode when it gets to `dir0`. If the owner is U or `root` then it must be the case that the directory was group- or world-writable when `safe-open` was invoked (since it was unsafe in the hypothetical resolution), and thus it must still be group- or world-writable when `safe-open` examines it (since by our assumption (c) U and `root` did not change that directory). We therefore conclude that the hypothetical and actual name resolutions proceeded identically upto (and including) `dir0`, and they both switched to unsafe mode upon visiting `dir0`.

In particular it implies that `safe-open` arrived at the final directory in unsafe mode, so it would only open F if F had a single hard link at the time that the procedure returned. Recall now that by our assumptions (a), this single hard link must be at the end of a U-safe pathname. But we know that `safe-open` visited at least one unsafe directory, so its traversal must have merged back into the safe pathname at some point after visiting `dir0`. As in the static case, this must have happened by following a symbolic link or '`..`', which is a contradiction.

Preventing privilege-escalation in the dynamic setting. Once we established the security guarantee in the dynamic setting, we can show how to use it to prevent privilege escalation even in a filesystem where permissions can change. In addition to creating the protected files with safe names,

[10] See Section 6.3 for a short discussion of `setgid` programs.

we also need to ensure that (a) we never reduce the write permissions of a non-empty directory that was group- or world-writable or chown a non-empty user directory back to root; and (b) we do not change permissions or ownership in the safe name and do not delete it while there are still programs that have the file open.

It is not hard to see that as long as (a) and (b) do not happen, then the conditions that we set in our dynamic-system proof hold, and hence no privilege-escalation can result from adversarial filesystem actions. Seeing that condition (a) is really needed is also easy: indeed if the attacker creates an adversarial link in a world-writable directory and then the victim chmods the directory and removes the world-writable permission, then safe-open will happily follow the adversarial link. Demonstrating that (b) is needed is a bit more tricky: Consider for example the file /etc/passwd, which is only writable by root, and consider the following sequence of operations:

1. Some user program P opens /etc/passwd for read and keeps the handle,
2. The attacker creates another hard link /var/mail/root to he same file,
3. A confused administrator deletes /etc/passwd, and
4. The mail-delivery program uses safe-open to open /var/mail/root, and then writes into it.

Note that safe-open will succeed under these conditions, since now /var/mail/root is the only name for this file (and in particular the file has only one hard link). But when the program P goes to read from its file descriptor, it will see the data that the mail-delivery program wrote there.

4. Implementing safe-open for POSIX Filesystems

We implemented safe-open as a library routine over the POSIX filesystem interface. The routine performs user-level name resolution, similar to the routines of Tsafrir *et. al* [20, 21], while adding the pathname safety check in every directory. That is, the routine goes through each component of the path to be opened, checks for the manipulators of each directory, and marks a directory unsafe if it has manipulators other than root and the current process' effective uid. Once it encounters an unsafe directory, in the remainder of the path, it does not follow symlinks or '..', and does not open a file with multiple hardlinks. A pseudocode description of our implementation is found in Figures 1 and 2 in Appendix B.

4.1. Race conditions

Our name-resolution procedure is not particularly vulnerable to filesystem-based adversarial race conditions, in that it would correctly label safe/unsafe directories regardless of concurrent actions of any attacker (as long as the euid of the attacker is neither root nor the victim's euid). There are only two points in our code where we need to guard against check/use conditions:

(A) We must never open a symbolic link. If the O_NOFOLLOW flag is available then we can use it for that purpose, but to get get the same effect in a truly portable code we implement the lstat-open-fstat-lstat pattern.

(B) The other check/use window in our code is between the time that we check permissions and conclude that we are in a safe directory and the time that we read a symbolic link or open a file or directory. As we explained in Section 3.2, this check/use window is only open to races against processes with the same effective uid as the process calling safe-open (or root), not to races against an adversarial process trying to escalate privileges. As permission-changing actions by benign processes are quite rare, we believe that this window does not pose a major threat. We can even check the directory permissions both before and after reading a symlink (or opening a file or directory) to further narrow this window (and then this race cannot happen as long as non-adversarial processes do not revoke write permissions on non-empty directories).

4.2. Thread safety

Implementing user-level name resolution requires that we work with handles to directories, using either the current working directory (which may not be thread safe) or the openat, readlinkat and fstatat interfaces, which are part of a recent POSIX standard [17]. These interfaces duplicate existing pathname-based interfaces but add another parameter, a file descriptor for a directory. When used with a relative name, these calls now work relative to the specified directory instead of the current working directory.

The new interfaces are implemented in current Solaris and Linux versions. On systems without support for the openat family of function calls, we emulate their functionality inside a synchronized block: Maintaining a handle to the directory currently visited, we store the current working directory, change directory with fchdir to the visited directory, explore the next path element (for example, with open or lstat), then restore the original current working directory. To make the emulation signal-safe we also need to suspend signal delivery while in the protected block.

4.3. Read permissions on directories

Our user-level `safe-open` implementation relies on the ability to open all the intermediate directories (e.g., to `fstat` them or to use them with `openat`). Each path component, except the final one, is opened in a `O_RDONLY` mode. For this implementation to work, the process must have read permission on each non-final component in the path (in addition to the search permission that is required to look up the next pathname component in that directory). This is different from the regular POSIX `open` that only requires search permission on each directory component.

This restriction is of only temporary nature: a recent POSIX standard [17] introduces the `O_SEARCH` flag to open a directory for search operations only, and a future `safe-open` implementation can migrate to this.

4.4. Opening files without side effects

Upon arriving at the last path element (i.e., the file to be opened), our `safe-open` implementation may still need to verify that it is not a symbolic link. We again use the `lstat`-`open`-`fstat`-`lstat` pattern, but we must guard against potential side-effects of opening the file. For instance, opening the file with the flag `O_TRUNC` in combination with either `O_WRONLY` or `O_RDWR` will truncate the file before the `safe-open` procedure can determine that it opened an unexpected file. To fix this problem, we must first remove the `O_TRUNC` flag when opening the file, and if no error occurs then call `ftruncate` on the newly opened handle before returning it.

Somewhat similarly, if `safe-open` unexpectedly opens a target which is not a regular file (such as a `FIFO` or a `tty` port), then the open call could block indefinitely. This can be addressed only with cooperation by the application: when an application never intends to open a blocking target then it could specify the flag `O_NONBLOCK`.

4.5. Implementing `safe-create`, `safe-unlink`, and other primitives

Building on the same ideas, we can implement safe versions of other POSIX interfaces, such as `safe-create` for creation of new files, `safe-unlink` for removing them, etc. For many of these primitives, the implementation can be almost trivial: follow the same steps as with `safe-open` to reach the final directory[11]; in the final step, `safe-create` creates the file (with flags `O_CREAT` and `O_EXCL`), and `safe-unlink` removes the target which may be a symlink or a file with multiple hardlinks.

[11] Some primitives (such as `unlink` and `mkdir`) do not follow a symlink that appears as the final pathname component; the `safe-unlink` and `safe-mkdir` functions must of course behave accordingly.

Our generalized pathname safety policy is easy enough to express: "*when resolving a pathname through an unsafe directory, in the remainder of the path don't follow '`..`' or symbolic links, and don't open or change attributes of files with multiple hardlinks.*" Articulating the exact security properties that you get may take some care. For example, the security property that you get from `safe-create` is this: "When called with an unsafe name, `safe-create` will fail to create the file if the resulting file could also have a safe name."

Implementing safe versions of POSIX interfaces with more than one pathname (i.e., `safe-rename` and `safe-link`) can be problematic on systems that don't support `renameat` and `linkat`. The emulation of these functions is complicated by the fact that a process can have only one current working directory at a time; as a workaround one could perhaps utilize temporary directories with random names as intermediaries.

Current POSIX standards still lack some primitives that operate on existing files by file handle instead of file name, but this may change as standards evolve. For example, the recently-standardized `O_EXEC` (open file for execute) flag [17] enables the implementation of a family of `fexec` primitives that execute the file specified by a file handle.[12] Based on these primitives one could implement `safe-exec` versions that can recover from accessing an unexpected file, similar in the way that `safe-open` recovers before performing an irreversible operation. We note that executing files in unsafe directories is a minefield, and leave the development of a suitable safety policy as future work.

5. Experimental validation

We conducted extensive experiments to validate our approach for safe pathname resolution. Our goals in these experiments were (a) to check whether existing applications would continue to work when they run over a POSIX interface that implements safe pathname resolution; and (b) to see if we can identify yet-undiscovered vulnerabilities related to applications that follow unsafe links.

5.1. Testing apparatus

We implemented our safe name resolution and tested several "live" systems, to see what applications actually use unsafe links, and for what purpose. To cover a wide range of operating systems and production environments, we opted for implementing our procedure in a "shim" layer between the applications and `libc`. That is, we built a li-

[12] Support for these is already implemented in some Linux and BSD versions.

brary that intercepts filesystem calls, and instructed the run-time linker to load it before the regular `libc`. We used this to instrument dynamically-linked programs including setuid and setgid programs.[13] This approach makes it easier to test existing systems, but it may not be able to interpose on calls between functions within the same library. In addition it is necessary to intercept some library calls not related to file access, to prevent the accidental destruction of environment variables or file handles that our "shim" layer depends on.

In the interposition library, we implemented the safe pathname resolution and used it in the filesystem calls `open`, `fopen`, `creat`, `unlink`, `remove`, `mkdir`, `rmdir`, `link`, `rename`, `chmod`, `chown`, and the `exec` family. With `openat` and related functions, we did not implement yet safe pathname resolution with respect to arbitrary directory handles; in our measurements, such calls were a tiny minority. So far we only instrumented calls that involve absolute pathnames, or pathname lookups relative to the current directory.

We also kept some state related to the current working directory in our library, in order to implement safe name resolution for relative pathnames. (The same approach can be used for the directory handles used by `openat` and related functions, but we did not implement this yet.) A more detailed description of the implementation and its intricacies is provided in Appendix C.

5.2. Measurements of UNIX systems

We ran our pathname safety measurements on several out-of-the-box UNIX systems, specifically Fedora Core 11, Ubuntu 9.04, and FreeBSD 7.2 for i386 (both server and desktop versions). These systems were run on VMware workstation 5 for Linux and Windows hosts, and on real hardware. We instrumented the top-level system start-up and shutdown scripts, typically `/etc/rc.d/rc` or `/etc/init.d/rc`, and were able to monitor system and network daemon processes as well as desktop processes.[14] In all of these experiments, we configured our library to run in a report-only mode, where policy violations are logged but the intended operation is not aborted. (In fact, following the complete pathname resolution, our library will simply make the underlying system call on the original arguments and return the result.)

We ran these systems in their out-of-the-box configurations, and also tested some applications including the Gnome desktop, browsing with several Firefox versions (including plugins for popular multi-media formats), office document browsing, printing with Adobe Acroread, software compilation with gcc, and software package installation. The vast majority of these tests passed without a hitch. Most systems and applications never attempted an operation that would violate our safety policy, and thus they would have worked just as well had we configured our safe name resolution in enforcing mode. One notable exception is the web-server application, discussed in Section 5.5.

5.3. Latent vulnerabilities

In the course of our experiments we uncovered a number of latent privilege escalation vulnerabilities. The latent vulnerabilities occur where privileged system processes write or create files as `root` in directories that are writable by an unprivileged process. In these cases, a compromise of an unprivileged process could result in further privilege escalation:

- The Common UNIX Printing System (CUPS) saves state in files `job.cache` and `remote.cache`. These files are opened with `root` privileges and with flags `O_WRONLY|O_CREAT|O_TRUNC`, in directory `/var/cache/cups` which is writable by group `lp` (on some systems group `cups`). The CUPS software uses this group when running unprivileged helper processes for printing, notification, and more. If an unprivileged process is corrupted, an attacker could replace the state files by hard or symbolic links and destroy or corrupt a sensitive file.
- On Fedora Core 11, a similar latent problem exists with files under directory `/var/log/cups`.
- During MySQL startup, the `mysqld` daemon opens a file *hostname*`.lower-test` with flags `O_RDWR|O_CREAT` as `root`, under directory `/var/lib/mysql` which is owned by the `mysql` user. If the `mysqld` daemon is corrupted later when it runs with user `mysql` privileges, an attacker could replace this file by a hard or symbolic link and corrupt a sensitive file when MySQL is restarted.
- The Hardware Abstraction Layer daemon subsystem opens a file with flags `O_RDWR|O_CREAT` as `root`, in directory `/var/run/hald`. This directory is owned by user `haldaemon`, who also owns several daemon processes. Some of these processes listen on a socket that is accessible to local users.

[13] While the LD_PRELOAD environment variable was sufficient to instrument most programs, instrumenting setuid and setgid programs required additional steps. We stored run-time linker instructions in `/etc/ld.so.preload` on Linux, and in `/var/ld/ld.config` on Solaris; we modified the run-time linker `/libexec/ld-elf.so.1` on FreeBSD.

[14] For this instrumentation, we disabled security software such as AppArmor and SELinux to avoid interference between our instrumentation and their enhanced security policies.

- The Tomcat subsystem opens a file with flags O_WRONLY|O_APPEND|O_CREAT as root in directory /var/cache/tomcat6. This directory is owned by user tomcat6, who also owns a process that provides service to remote network clients.
- On Fedora Core 11, directory /var/lock is writable by group lock, which is also the group of a setgid program /usr/sbin/lockdev. System startup scripts create "lock" files as root with flags O_WRONLY|O_NONBLOCK|O_CREAT|O_NOCTTY. If the lockdev program has a vulnerability, an attacker could replace a lock file by a hard or symbolic link and corrupt a sensitive file.
- XAMPP [24] (an integrated package of Apache, MySQL, PHP and other components) on Linux opens files, for error logging, as root in the directory /opt/lampp/var/mysql which is owned by the uid nobody. A corrupted process running as nobody can replace this with a link to any file on the system which would then be overwritten. We note that XAMPP runs a number of daemons providing network services as the nobody user, including httpd.

In all these cases, our safe name resolution would protect the system from privilege escalation if the unprivileged processes are corrupted.

5.4. Policy violations

During our "whole system" tests we ran into a surprisingly small number of actual safety policy violations. These turned out to be specific to particular platforms, and were caused by quirks in the way that directory ownership and permissions were set up:

- On FreeBSD 7.2, the man command could trigger policy violations when a user requested a manual page. FreeBSD stores pre-formatted manual pages under directories owned by user man (instead of root as with many other UNIX systems). According to our policy, these directories are unsafe for users other than man. This resulted in policy violations with pre-formatted manual page files that had multiple hard links.

  FreeBSD adopted this approach so that pre-formatted manual pages can be maintained by a non-root process. This limits the impact of vulnerabilities in document-formatting software. However, we find the benefits of this approach dubious: document-formatting software still runs with root privileges when the super-user requests a manual page for software that is not part of the base system. By default, no pre-formatted manual pages exist for this software category, and this is where the biggest risk would be.
- The FreeBSD package manager triggered warnings about following '..' when removing a temporary directory tree under /var/tmp; these could be addressed by a more permissive policy (cf. Section 6.1).
- On Fedora Core 11, the Gnome desktop software triggered policy violations that we did not experience with other systems. The violations happened when a process with gdm user and group privileges attempted to follow symbolic links under directory /var/lib/gdm. This directory is writable by both owner gdm and group gdm.

  These policy violations can be avoided with a more sane configuration that uses owner gdm write permission only. Our "live" measurements show that group gdm is used only by processes that run as user gdm. With a single-member group like gdm, owner gdm permission is sufficient, and group gdm write permission is unnecessary. (We found similar issues with XAMPP for Linux, which installs with directories that have owner nobody and group root with group write permission.)

5.5. A web-server application

Most of our measurements were done on bare-bones systems that we instantiated specifically for the purpose of running the experiments. The only production system that we had access to was a Debian 5.0 system running an Apache web server and some other services. On that system we did not attempt a whole-system measurement, but instead only run specific services under our measurement apparatus. Also on that system most services did not report any policy violations, with the notable exception of the web server.

The web site on that system is managed cooperatively by several users, where different users are responsible for different parts of the site, and with no attempt for any protection between these users. As a result, the web-tree is a mesh of directories with different owners, many of them writable by the web-administrator group (whose members include all these different users). Roughly speaking, the entire web-tree on that system is an UNsafe subtree. Moreover, some dynamic-content parts of the web site make heavy use of symbolic links, e.g., for using the same script in different contexts.

It is clear that our safe-open procedure will break this web site, but this is more an artifact of our particular choice of implementation than of the security guarantee that we set out to ensure. Indeed, in Section 6.1 we describe a more permissive implementation of safe-open that still ensures the same security guarantee, but would not break this web site. (The idea is that we can follow symbolic links off un-

safe directories, as long as we ensure that the file that we get to at the end does not have any safe names.)

5.6. Conclusions

Our experiments seem to indicate that our approach to safe name resolution is both effective and realistic. On one hand, it fixes *all 177 symlink-related vulnerabilities reported in CVE since January 2008*, and also provides protection against the (latent) vulnerabilities that we identified in our experiments. On the other hand, most systems will continue working without a problem even if this safety measure was implemented. The few that break can be "fixed" either by implementing a more standard permission structure for the relevant directories or by implementing the more permissive variant of `safe-open` from Section 6.1.

We stress that in our experiments, we did not identify even a single example where there is a legitimate need to open files that would be inherently disallowed by our approach to safe name resolution.

6. Variations and Extensions

6.1. A more permissive `safe-open`

Our `safe-open` procedure does not follow symbolic links off an unsafe directory, but is not hard to see that this policy is more restrictive than what we really need for our security guarantee. Indeed, we only need to ensure that `safe-open` fails on an unsafe name *if the file to be opened has any other name that is safe*. It turns out that a small modification of `safe-open` can ensure the same security guarantee while allowing more names to be opened.

The idea is to keep two safe/unsafe flags rather than one. Both flags begin in a *safe* state and switch to *unsafe* state when visiting an unsafe directory, but one flag is "sticky", in that once in *unsafe* state it stays in this state until the end of the name resolution, while the other is reset to the `safe` state whenever we are about to follow a symbolic link with an absolute path. That is, the second flag is reset to `safe` state whenever we are about to return to the root directory.

With these two flags, we can follow arbitrary symbolic links, and can also follow '`..`' as long as the second flag is in *safe* mode. When we finally reach the file to be opened, we abort the procedure only if (a) the "sticky" flag is in *unsafe* mode and the file has more than one hard link, or (b) the two flags have different values. (In the second case, the "sticky" flag indicates that the given pathname was unsafe, while the resettable flag indicates that as part of the name resolution we followed some safe name to arrive at the file.) The reason that this more permissive procedure works, is that if a file with only one hard link has any safe names, then its "canonical" name (i.e., the one with no symbolic links) must be safe. Moreover, this name must be the one followed by the time that the name-resolution arrives at the file itself.

As we described it, this more permissive version still refuses to follow '`..`' when the second flag is in *unsafe* mode. This can be easily remedied, however: we simply drop the restriction on following '`..`', and instead just reset the second flag to *safe* mode after every '`..`'.

6.2. An alternative `safe-open` using extended attributes

On some systems, a much more direct approach is also possible. Recall that the problem that we try to address is that an adversary without permissions to a file is able to add names to the filesystem that resolve to that file. If the filesystem supports extended attributes, then we can avoid this problem simply by including with the file an attribute that lists all the permissible names for that file. The `open` procedure, after opening the file, will look for this extended attribute, and if found it will compare its pathname argument against the list of permissible names, and will abort if there is a mismatch. For example, the file `sudo` in `/etc/init.d/` will have a permitted-names attribute listing the names `/etc/init.d/sudo` and `/etc/rcS.d/S75sudo`, and no program will ever be able to open it using any other name.

This simple solution looks quite attractive, but it necessitates proper management of the additional attribute. In particular, we must decide who may set this attribute (and under what conditions). For example, when we add to our filesystem a symbolic link:

```
/var/spool/mail -> /var/mail
```

do we need to modify the permitted-name attribute in all the files under `/var/mail/`? We leave all these questions to future work.

6.3. Group permissions

Recall that our `safe-open` procedure only uses `uids` to determine safety of directories, which means in particular that we treat two processes with the same `uid` as equal and do not try to protect one from the other. This leaves open the possibility of privilege escalation by acquiring group privileges: namely, an adversarial process may try to trick another process with the same effective `uid` but more group privileges into opening a file that the adversarial process itself cannot open.

In the work we do not try to protect against such attacks, indeed protection between different processes with the same effective `uid` is virtually impossible in most POSIX systems. We mention that it is not hard to change the

safe-open procedure itself so that it considers the gid rather than the uid for the purpose of determining directory safety, but this would require a change in the interface, since the calling application would need to somehow indicate that it wants to use this gid-based safety check instead of the default uid-based check.

We note that our approach for safe name resolution is quite coarse with respect to group permissions, in that group write permissions always make a directory unsafe for everyone. This is justified when the directory gid is the primary or secondary gid of multiple UNIX accounts, since multiple accounts are manipulators. However, contemporary UNIX-es have many gids that are associated with only one uid (or maybe none at all, e.g. when the gid is only used by the execution of a setgid program). In general we cannot anticipate all possible ways that a gid may be activated, and hence we consider the directory unsafe in all these cases. This may trigger spurious policy violations in some configurations, but in our experiments we did not find configurations where such policy violations cannot be resolved.

We also note that in conjunction with the more permissive variant from above, this behavior lets administrators bypass much of our safety mechanisms: To forgo most of our safety protections for some subtree (without otherwise changing any permissions), it is sufficient to make the root of that subtree writable, e.g., by the root group. Assuming that only root is a member of this group, this will not change any real permissions in the system, but will make that entire subtree unsafe, and therefore permit opening of the files in it also using other unsafe names, even ones with symbolic links. (This trick does not help if there are multiple hardlinks, however.)

7. Conclusion

In this paper we considered the problem of privilege escalation via manipulation of filesystem pathnames, which effect name resolution in system calls such as open, unlink, etc. While many privileged programs take measures to protect against such attacks, these measures are always very application specific. We propose a more general approach of having safe pathname resolution as part of the filesystem itself or a system library, thereby protecting all applications by default.

We introduced the concept of the *manipulators* of a pathname, that include anyone who can influence the outcome of the pathname resolution. In POSIX these are the users who either own or can write in any directory visited during the pathname resolution. Using this concept, we call a pathname *safe* for U if the only manipulators of the pathname are root and U. We described a general routine safe-open, ensuring that if a file has safe names then safe-open will not open that file with an *unsafe* name, and demonstrated that this guarantee can be used to thwart filename-based privilege escalation attacks. This is useful not only for privileged programs that run in known-to-be hostile environments, but also for programs written by naive developers, and programs that are being deployed in unforeseen environments with unexpected file permission semantics.

We implemented our safe name resolution routine in a library, using portable code over the POSIX interface, and performed extensive experiments to validate the applicability of our solution to current operating systems and applications. We verified that this solution uniformly protects system against the documented cases of applications and daemons vulnerable to pathname manipulation attacks, as well as against some new (latent) vulnerabilities that we uncovered. We also instrumented current versions of Ubuntu 9.04, Fedora Core 11 and FreeBSD 7.2 to run every process through a program which interposes calls to file manipulation and related calls and checks if the corresponding operation manipulates a safe pathname. These experiments confirmed that very few existing systems break when used over our safe name resolution, and the handful of cases where our solution produces false positives can be handled either by implementing a more standard permission structure for the relevant directories or by using a more permissive variant of our solution.

References

[1] M. Bishop. Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux. Technical Report CSE-95-8, University of California at Davis, Sep 1995.

[2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. Computing Systems 9(2), pp. 131–152, Spring 1996.

[3] N. Borisov, R. Johnson, N. Sastry, and D. Wagner. Fixing races for fun and profit: how to abuse atime. In 14th USENIX Security Symposium, pp. 303–314, Jul 2005.

[4] X. Cai, Y. Gui, R. Johnson. Exploiting Unix File-System Races via Algorithmic Complexity Attacks In *IEEE Symposium on Security and Privacy*, Oakland, California May 2009.

[5] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS*, pages 136–145, 2001.

[6] R. Canetti, S. Chari, S. Halevi, B. Pfitzmann, A. Roy, M. Steiner, W. Venema. Composable Security Analysis of Operating System Services. Technical Report RC24900, IBM T. J. Watson Research Center, 2009.

[7] H. Chen, D. Wagner, and D. Dean. Setuid Demystified. In *USENIX Security Symposium*, pages 171–190, 2002.

[8] D. Dean and A. J. Hu. Fixing races for fun and profit: how to use access(2). In 13th USENIX Security Symposium, pp. 195–206, Aug 2004.

[9] CERT Coordination Center. CERT Advisory CA-1995-02. http://www.cert.org/advisories/CA-1995-02.html, January 26, 1995. (Accessed December 2009).

[10] CERT Coordination Center. CERT Advisory CA-1993-17. http://www.cert.org/advisories/CA-1993-17.html, November 11, 1993. (Accessed December 2009).

[11] Common Vulnerabilities and Esposures. CVE-2001-0529. http://cve.mitre.org/, March 9, 2002. (Accessed December 2009).

[12] Security Vulnerability in inetd(1M) Daemon When Debug Logging is Enabled CVE-2008-1684. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1684, September 2008. (Accessed December 2009).

[13] initscripts Arbitrary File Deletion Vulnerability. CVE-2008-3524. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-3524, September 2008. (Accessed December 2009).

[14] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS 2003, San Diego, California, USA.

[15] D. Mazieres and F. Kaashoek. Secure applications need flexible operating systems. In IEEE Workshop on Hot Topics in Operating Syst. (HOTOS), p. 56, 1997.

[16] National Vulnerability Database. http://nvd.nist.gov/. (Accessed December 2009).

[17] The Open Group Base Specifications Issue 7; IEEE Std 1003.1-2008. http://www.opengroup.org/. (Accessed December 2009).

[18] M. Seaborn. Plash: tools for practical least privilege. http://plash.beasts.org/. (Accessed December 2009).

[19] D. Tsafrir, D. Da Silva, and D. Wagner. The murky issue of changing process identity: revising "setuid demystified", USENIX ;login, 33(3), pages 55–66. 2008.

[20] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably Solving File TOCTOU Races with Hardness Amplification. In *Proceedings of* USENIX Conference on File and Storage Technologies (FAST), pages 189–206, 2008.

[21] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably preventing file race attacks with user-mode path resolution. Technical Report RC24572, IBM T. J. Watson Research Center, June 2008.

[22] W. Venema. The Postfix mail transfer agent. http://www.postfix.org/. (Accessed December 2009).

[23] R. N. M. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *Proceedings of the 1st USENIX Workshop on Offensive Technologies*, Boston, August 2007.

[24] K. Seidler. The XAMPP software. http://www.apachefriends.org/. (Accessed December 2009).

A Relative pathnames

When resolving a pathname relative to an initial directory (i.e. the current directory or a directory handle with functions such as `openat`), the resolver needs to determine if the initial directory is safe, before following the same steps as with absolute pathnames (Section 2.2). For this, the implementation needs to maintain safety information about directory handles, including the implicit directory handles for the current and root directories of all processes.

The per-handle safety information needs to be initialized when a directory handle is instantiated with functions such as `open`, `chdir` or `chroot`, and the safety information needs to be propagated when a directory handle is copied with functions such as `dup`, `fcntl`, `fork`, or with functions that transmit a file handle over an inter-process communication channel. Maintaining this information is straightforward in the file system itself (i.e. in the operating system kernel). We discuss our user-level approach in Appendix C.

In a simplistic implementation, each directory handle has a static flag that indicates if the directory is safe. However, additional care is needed with processes that change their effective `uid` (for example, a process that invokes the `seteuid` function, or a process that executes a file with the setuid bit turned on). As the result of an effective `uid` change, a directory that was safe may become unsafe or vice versa. As a further complication, the safety of a directory depends on the program execution history. For example, a handle for directory `/etc` is normally safe for everyone, but that same directory handle would be safe only for `joe` if a pathname resolved through a symbolic link under `/home/joe`.

To account for processes that change execution privilege, we propose that each directory handle would include a field specifying the `uid` that the directory's pathname prefix was "safe for" when the pathname was resolved. Namely, this field will indicate `root` if the directory was reached via a system-safe pathname, it will indicate a single non-root `uid` if it was reached via a pathname that has only `uid`

```
/* Resolve a pathname and open the target file */

safe_open(path, open_flags, is_safe_wd)
{
    if (path is absolute) {
        is_safe_wd = 1; dirhandle = null;
    } else {
        dirhandle = open(".", O_RDONLY) or return error;
    }
    return safe_lookup(dirhandle, path, is_safe_wd,
                       lookup_flags_for_open,
                       open_action_func, open_flags);
}

/* Call-back to open the final pathname component */

open_action_func(dirhandle, name, is_safe_wd, open_flags)
{
    truncate = (open_flags & O_TRUNC);
    flags = (open_flags & ~O_TRUNC);

    filehandle = openat(dirhandle, name, flags)
                 or return error;
    fst = fstat(filehandle) or return error;
    /* lstatat(args) is local alias for
       fstatat(args, AT_SYMLNK_NOFOLLOW) */
    lst = lstatat(dirhandle, name) or return error;

    if (fst and lst don't match) return EACCESS;
    check dirhandle permissions again,
          and update is_safe_wd if unsafe;
    if (!is_safe_wd && name is "..") return EACCES;
    if (!is_safe_wd && fst is not a directory
        && fst has multiple hard links)
        return EACCES;
    if (truncate) ftruncate(filehandle,0)
                  or return error;
    return filehandle;
}
```

Figure 1. The top-level `safe_open` and a call-back function `open_action_func`.

and `root` as manipulators, and it will indicate *no-one* if the pathname had more than one non-root manipulator. When resolving a pathname relative to an initial directory, one determines the safety of the initial directory by combining the "safe for" `uid` from the handle with fresh information about the owner and writers for the initial directory itself.

B Pseudo-Code Implementation

The `safe-lookup` procedure described in Figure 2 implements our safe pathname resolution principle and is the common routine used to implement `safe-open`, `safe-create` and other safe versions of the POSIX interface. The specifics of each individual function are reflected in the arguments `lookup_flags`, `action_func` and `action_args`. The top-level function `safe_open` and the corresponding parameters to `safe_lookup` are described in Figure 1.

```
/* Resolve pathname, invoke action on final component */

safe_lookup(dirhandle, path, is_safe_wd, lookup_flags,
            action_func, action_args)
{
    if (path is empty) return ENOENT;
    if (path is absolute) {
        dirhandle = open("/", O_RDONLY) or return error;
        fst = result of lstat("/") or return error;
        skip leading "/" in path, and replace
             path by "." if the result is empty;
    } else
        fst =result of fstat(dirhandle) or return error;
    while (true) {
        /* check dirhandle permissions */
        if (fst.owner not in [root, euid]
            || anyone not in [root, euid] can write)
            is_safe_wd = false;

        split path into first and suffix,
              and replace all-slashes suffix by ".";
        lst = result of lstatat(dirhandle, first)
              or return error;

        /* the meaning of "final pathname component"  *
         * depends on lookup_flags, it has different  *
         * meaning for open, unlink, etc.             */
        if (first component is final pathname component)
            return action_func(dirhandle, first,
                               is_safe_wd, action_args);

        if (first component is a symlink) {
            newpath = readlinkat(dirhandle, first)
                      or return error;
            check dirhandle permissions again,
                  and return EACCES if unsafe;

            /* symlink at end of pathname */
            if (suffix == null)
               return safe_lookup(dirhandle, newpath,
                               is_safe_wd, lookup_flags,
                               action_func, action_args);

            /* other symlink */
            [newhandle, fst] =
             safe_lookup(dirhandle, newpath, is_safe_wd,
                         lookup_flags, null, null)
              or return error;

        } else {
            /* first component is not a symlink */
            newhandle = openat(dirhandle,first,O_RDONLY)
                        or return error;
            check dirhandle permissions again,
                  and update is_safe_wd if unsafe;

            if (!is_safe_wd && name is "..")
               return EACCES;
            fst = result of fstat(newhandle)
                  or return error;

            if (first component is not a directory)
               return ENOTDIR;

            lst = result of lstatat(dirhandle, first)
                  or return error;
            if (lst does not match fst) return EACCES;

            /* reached the end of readlinkat result */
            if (suffix == null) return [newhandle, fst];
        }
        path = suffix;
        dirhandle = newhandle;
    }
}
```

Figure 2. The `safe_lookup` recursive call.

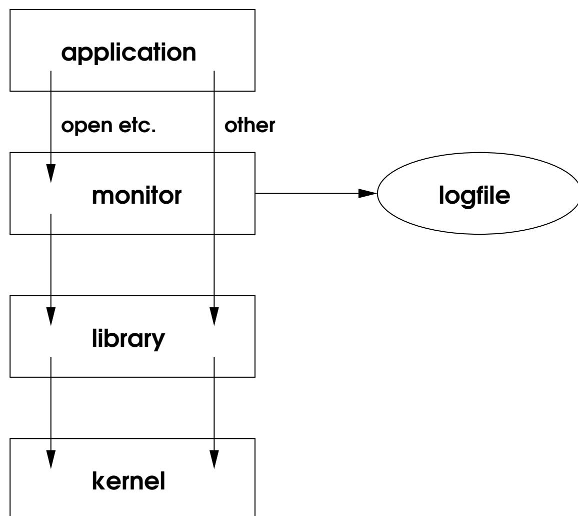

Figure 3. In-process monitor architecture.

C User-level implementation

As mentioned earlier in this paper, a kernel-based implementation of safe pathname resolution is straightforward: while visiting each pathname element one at a time, maintain a safety flag and apply the safety policy for following symbolic links, "`..`", and for files with multiple hard links as appropriate. With a kernel-based implementation, maintaining per-handle directory safety information is also straightforward. This approach is preferable, but only after it has been demonstrated that safe pathname resolution does not break well-behaved programs.

To demonstrate the feasibility of our pathname safety policy, we chose an approach that is based on library-call interposition with an in-process monitor. This approach works with dynamically-linked programs, including programs that are setuid or setgid, and it provides acceptable performance on Linux, FreeBSD and Solaris systems. We opted against external-process monitors such as `strace` or `truss`: they suffer from TOCTOU problems, they cause considerable run-time overhead, and they don't have direct access to the monitored process's effective `uid` which is needed for pathname safety decisions.

As illustrated in figure 3, the monitor is implemented as a library module that is loaded into the process address space between the application and the libraries that are dynamically linked into the application. Depending on configuration, the monitor can log function calls such as `open` with the effective `uid`, and can log whether or not a call violates our pathname safety policy. For the purpose of the feasibility test the monitor does not enforce policy, but instead passes control to the real `open` etc. function. The in-process monitor for Linux, FreeBSD and Solaris is implemented in about 2000 lines of K&R-formatted C code, comments not included, plus a small shell script that implements the command-line interface.

Besides interposing on functions such as `open` that require pathname resolution, our in-process monitor interposes on additional functions to ensure proper operation of the monitor itself. For example, the monitor intercepts function calls such as `close` and `closefrom`, to prevent the logging file handle from being closed by accident. The monitor intercepts function calls such as `execve` to ensure consistent process monitoring when a new program is executed. Upon `execve` entry, the monitor exports environment variables to control run-time linker behavior and to propagate monitor state, and it resets the close-on-exec flag on the logging file handle. When the `execve` call returns in the newly-loaded program, the monitor restores private state from environment variables before the application's code starts execution.

To track per-handle directory safety state, an in-process monitor would need to interpose on functions that copy file handles such as `dup` or `fcntl`. Interposition is not necessary with process-creating primitives such as `fork` or `vfork`, since these are not designed to share the in-kernel file descriptor table or process memory between parent and child processes. On the other hand, the Linux `clone` and BSD `rfork` process-creating primitives are designed so that they can share the file descriptor table or process memory, meaning that changes made by one process will affect the other process. This behavior complicates a user-level monitor implementation, and is not yet supported by our monitor.

Our preliminary in-process monitor maintains a safe/unsafe flag for directory handles created with `open` so that it can check programs that use the `open-fchdir` idiom. The monitor does not yet check lookups relative to a directory handle. In our measurements, we found that the `openat` etc. functions are used by only few programs, and that those functions are called almost exclusively with absolute pathnames or with pathnames relative to the current directory. The monitor currently does not propagate the per-process current directory and root directory safety state across function calls such as `execve`. Instead, it initializes their safety state on the fly at program start-up time. Without modification to monitored applications, it is not practical for a user-level monitor to track safety flags for directory handles that are sent over an inter-process communication channel. Fortunately, such usage is rare.