Unsafe at Any Copy: Name Collisions from Mixing Case Sensitivities

Aditya Basu and John Sampson, *The Pennsylvania State University;* Zhiyun Qian, *University of California, Riverside;* Trent Jaeger, *The Pennsylvania State University*

https://www.usenix.org/conference/fast23/presentation/basu

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Unsafe at Any Copy: Name Collisions from Mixing Case Sensitivities

Aditya Basu[*] aditya.basu@psu.edu
John Sampson[*] jms1257@psu.edu
Zhiyun Qian[†] zhiyunq@cs.ucr.edu
Trent Jaeger[*] trj1@psu.edu

[*]*The Pennsylvania State University* [†]*University of California, Riverside*

Abstract

File name confusion attacks, such as malicious symlinks and file squatting, have long been studied as sources of security vulnerabilities. However, a recently emerged type, i.e., ***case-sensitivity-induced name collisions***, has not been scrutinized. These collisions are introduced by differences in name resolution under case-sensitive and case-insensitive file systems or directories. A prominent example is the recent Git vulnerability (CVE-2021-21300) which can lead to code execution on a victim client when it clones a maliciously crafted repository onto a case-insensitive file system. With trends including `ext4` adding support for per-directory case-insensitivity and the broad deployment of the Windows Subsystem for Linux, the prerequisites for such vulnerabilities are increasingly likely to exist even in a single system.

In this paper, we make a first effort to investigate how and where the lack of any uniform approach to handling name collisions leads to a diffusion of responsibility and resultant vulnerabilities. Interestingly, we demonstrate the existence of a range of novel security challenges arising from name collisions and their inconsistent handling by low-level utilities and applications. Specifically, our experiments show that utilities handle many name collision scenarios unsafely, leaving the responsibility to applications whose developers are unfortunately not yet aware of the threats. We examine three case studies as a first step towards systematically understanding the emerging type of name collision vulnerability.

1 Introduction

A fundamental file system design choice is whether it will allow file names to be case sensitive or not, and modern file systems are diverse in their selection. A *case-sensitive file system* is one that allows the definition of multiple files whose names differ only in their case, such as `Foo.c` and `foo.c`. In a *case-insensitive file system*, only one file can be defined whose names differ only in their case. Historically, UNIX file systems are case sensitive, whereas Windows file systems are case insensitive. Further, case-insensitive file systems may be either case preserving (e.g. Apple File System (APFS), NTFS, etc.) or not (FAT), where a *case-preserving file system* preserves the case chosen (i.e., either `Foo.c` or `foo.c`), rather than converting all names to one case choice (e.g., all lowercase). Importantly, while choices in case sensitivity for a single file system may appear to be arbitrary or aesthetically driven, the precise semantics of interactions between two file systems with different case sensitivities can range from subtle to ill-defined, with associated consequences.

Practitioners have long had concerns about the implications of leaving case sensitivity as an open design choice [31] Historically, these concerns were not considered as pressing when file systems were associated with their respective operating systems and associated singular assumptions about case. However, *individual systems now frequently support a mixture of case-sensitive and case-insensitive file systems*, creating opportunities for files to be moved between file systems with different cases and file identifier encodings. More troublingly, *several file systems now support allowing the choice of case for individual directories* [12], complicating file operations by having multiple case and encoding semantics within the same file system.

Security risks related to this design choice therefore appear to be increasing. First, the Windows Subsystem for Linux [58] (WSL) integrates Linux and Windows platforms leading to expectations that files may be routinely copied from Linux (i.e., case-sensitive) to Windows (i.e., case-insensitive) file systems. Second, Linux `ext4` now supports case-sensitive and case-insensitive naming in the same partition, configurable per directory [12, 34]. Linus Torvalds expressed concerns about adding such support to `ext4` [31], stating that such features often cause "actual and very subtle security issues".

Indeed, security issues caused by moving files from case-sensitive to case-insensitive file systems are starting to appear. For example, the `git` distributed version control system has suffered from multiple vulnerabilities (e.g., CVE-2014-9390, CVE-2021-21300), caused by how `git` clones repositories from case-sensitive file systems to case-insensitive file systems.

To exploit this, an adversary creates a repository in a case-sensitive file system with a directory whose name will *collide* (i.e., only differs in case) with a symbolic link (to another directory) added by `git` when the repository is cloned to a case-insensitive file system. The *name collision* between the directory and the symbolic link enables adversaries to overwrite the scripts that `git` executes. Such attacks can alter both the target resource's content and/or its metadata, including its permission assignments.

Researchers have long been aware of hazards that may occur during file system name resolution [3, 4], particularly that programmers must validate safe use of symbolic links and check for "squatted" files when creating a new files. Many defenses have been proposed [7–9, 30, 40–42, 50–52, 55]. However, to the best of our knowledge, ours is the first work studying how case interplays cause name collisions that lead to incorrect, and in some cases, vulnerable behaviors. We show that utilities and applications currently do not recognize unsafe use of case-insensitive file systems, leading to these problems. This paper demonstrates the potential implications of the name collision problem, focusing on Linux and its supported file systems, thereby motivating both more and broader (e.g., other OS-FS combination) investigations. We identify potential gaps in the existing contract between the applications and the underlying file system that results in unsafe behaviors (see §8). We make the following contributions:

- We examine the security and correctness implications of *name collisions*, when two distinct file system resources with two distinct names map to to a single name, due to file system case sensitivity and/or encoding mismatches.
- We show that improper handling of case-[in]sensitivity and encoding can result in silent data loss and corruption, symbolic link traversal, unexpected hardlink creation, insecure merging of directory contents, and data disclosure due to incorrect overwriting of file system resources.
- We developed an automated method to test common Linux utilities for unsafe reactions to name collisions, finding a wide variety of responses, many of which are unsafe and possibly exploitable.
- We demonstrate novel exploits on three programs `dpkg`, `rsync`, and Apache `httpd`, showing how they operate incorrectly in the face of name collisions and how they would be exploited when deployed on case-insensitive directories.

2 Background: From Cases to Collisions

Beyond traditional, i.e. operating-system-entailed, decisions made with respect to case sensitivity, even Linux files systems now represent a surprising diversity of case sensitivity decisions. In particular, the desire to support some non-native applications, such as WINE and Samba from Windows systems, has motivated Linux file systems to support the case-insensitive file naming used in these non-native file systems.

The ability to create case-insensitive file systems has long been possible in some Linux file systems, such as ZFS, JFS, and ciopfs. However, these options are applied to the entire filesystem, rather than just the relevant directories for individual applications. In 2019, Linux kernel version 5.2 added support for per-directory case-insensitivity to ext4 [12, 34]. Later in 2019, similar support was added to the Flash-Friendly File System (F2FS) in Linux kernel version 5.4 [13, 14]. For case-insensitive directories, these file systems are case-preserving in nature.

2.1 Motivations for Increasing Case Diversity

Samba Samba [45] implements the Common Internet File System (CIFS) protocol which allows for sharing file systems over a network. Its primary use is sharing files with Windows clients that expect a case-insensitive file system. Hence, Samba implements user-space case-insensitive lookups even if the underlying file system is case-sensitive. Furthermore, it allows turning on/off case-sensitivity and case-preservation on a per-mount basis [46]. Note that this feature only works for non-Windows clients, which means that the actual file system can contain files differing only in case. This can lead to unexpected behaviors where Samba will choose to show only a subset of files. Deleting files which have collisions will now show the alternate versions, thereby giving rise to inconsistent behavior from the end user's perspective.

Samba's requirement of case-insensitive matching, which is done in user-space, incurs a huge performance overhead [37] thereby motivating the support for case-insensitivity in the ext4 file system [34–36]. Other programs/systems such as Wine [57], Network File System (NFS), SteamOS [48, 49] and Android [32, 59] would also benefit from in-kernel case-insensitivity support.

ext4 For ext4, the idea is that the filesystem at large can be configured to be "casefolding," which permits the mixing of case-sensitive and case-insensitive directories in the same file system. When creating an ext4 file system, the *casefold* option is applied, e.g., `mkfs -t ext4 -O casefold /dev/sda`. Setting the `+F` inode attribute on an empty directory makes it case-insensitive, e.g., `mkdir foo; chattr +F foo`. Note that case-insensitive directories can contain case-sensitive directories. This means that for a given path, `/foo/bar/bin/baz`, any of `foo`, `bar` and `bin` can either be case-sensitive or case-insensitive.

tmpfs `tmpfs` recently added case-insensitivity support [33]. The use cases are similar to that of ext4 with the addition of supporting sandboxing and container tools such as Flatpak.

2.2 Name Collisions

A *name collision* occurs when a file system maps two distinct names of two distinct resources to the same name. Name collisions can cause problems to occur if the names of distinct resources *collide* when those resources are replicated to a target directory that does not provide a 1:1 mapping for all replicated objects. Suppose one directory has two files with distinct names in that file system. Should those files be copied to a second directory in which the two file names collide (i.e., are mapped to the same name), then only one file will be created, which may be either of the original files or an unpredictable combination of the two files' content and metadata. Variation in case sensitivity between two file systems is a common origin of collisions, but diversity in other encoding properties, such as character choice (e.g., FAT does not support “, :, ∗, etc. [1]) and canonicalization processes, can lead to the same effect. For example, NTFS uses UTF-16 while APFS (macOS) and ext4 (Linux) use UTF-8 and older file systems can use other encoding schemes, such as iso8859-1.

Modern encoding schemes such as Unicode (e.g., UTF-8, UTF-16, etc.) have support for non-English characters that requires *case folding* [6] to perform case-insensitive matching. Unlike traditional techniques, case folding uses lookup tables to transform each character of the filename to a predetermined case. Furthermore, individual characters in Unicode can have multiple binary representations. Hence, a normalization scheme also needs to be applied to the case folded filename to ensure that the same characters are encoded using identical binary sequences. Consider the filenames `floß`, `FLOSS` and `floss`. All can coexist on a case-sensitive file system supporting reasonable character encodings, but since case-folding for both `floß` and `FLOSS` is `floss`, attempting to move these files to a case-insensitive system may only preserve one of the original triple.

In addition, *case folding rules and normalization techniques can differ across file systems*. The *locale* (or language) also influences the case folding rules. Due to such differences, ‘temp_200K’ (where K = Kelvin Sign or Unicode code point U+212A) and ‘temp_200k’ are considered identical on NTFS and APFS, but on ZFS[2] these filenames are considered different when using case-insensitive lookups. As a result, when two files of these names are copied from a ZFS file system to an NTFS file system, they will collide and only one filename and only one file will be created. For clarity and conciseness, we will use examples of ASCII-based, case-insensitive matching throughout the rest of the paper.

We propose a taxonomy for *name confusions*, shown in Figure 1, that captures the types of incorrect program behaviors that may stem from the ambiguous uses of names for file system resources. *Name collisions* are a subset of this broader class. Name confusions may be caused by three reasons: (1) because multiple names may refer to the same resource (i.e., *aliasing*); (2) because an adversary may create a resource of that name before the victim (i.e., *squat*); and (3) because the multiple resources may be associated with the same name (i.e., *collisions*). Of these, however, name collisions are the least explored for their correctness and security implications. As Linux is adding more support for case-insensitivity, it is crucial to understand the pitfalls and problems such functionality may incur. This work aims to study these issues.

3 From Collisions to Calamities

Name collisions can impair system functionality by modifying the content and/or metadata of files and directories in unexpected ways. Some name collisions have already led to security vulnerabilities [24]. In this section, we define the conditions in which a name collision occurs, the conditions under which such a collision may be exploitable by an adversary, and describe a known vulnerability that is caused by a name collision.

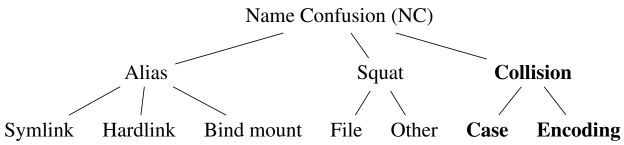


Figure 1: Taxonomy of *name confusion vulnerabilities* divided into *alias* (i.e., multiple names for a resource), *collision* (i.e., multiple resources for a name), and squat (temporal ambiguities in names vs. resources) classes

3.1 Causes of Name Collisions

A process may cause a name collision under the following conditions.

- There exists a *source resource* (e.g., file or directory) in a case-sensitive file system, whose name is *source name*.
- The process uses a relocation operation to place the source resource in a *target directory*, where the target directory is a case-insensitive or case-preserving directory. Examples of relocation operations include copy (e.g., `cp`, `rsync`, or an archive operation, such as `tar` or `unzip`) or move (e.g., `mv`).
- The relocation operation produces a *destination name* from the source name for the name of the source resource when placed in the target directory.
- There is a *target resource* with a *target name* whose name differs from the source name, but maps to the same name as the destination name does in the target directory (e.g., due to differences in case-folding rules between the source and target directory).
- If the process is authorized to modify the target resource, the process’s relocation operation results in a name collision between the target and source resources.
- If the relocation operation proceeds despite the name collision, then the target resource’s content and/or its metadata may be modified using the source resource content and/or metadata.

When these conditions are met, a name collision occurs such that the target resource in the target directory will be modified using the source resource. In most cases, modifying a target resource using a source resource of a different name is an unexpected result. We test how common Linux utilities react to name collisions and examine case studies where name collisions cause incorrect operation.

Given the above conditions, there are several clear scenarios where the movement of files involving the following types of file systems (following the categorization in §2.2) could result in name collisions:

[1] `http://elm-chan.org/fsw/ff/doc/filename.html`

[2] By default, the ZFS file system does not perform normalization. We use this default behavior for the given example.

- Case-sensitive and case-insensitive file systems.
- Two distinct case-insensitive file systems with different case folding rules, e.g. ZFS to NTFS, etc.
- Two file system whose locales are different but they still use the same file system format (such as ext4).
- A single file system that supports per-directory case-insensitivity, e.g. ext4.

Clearly, name collisions may impact system functionality by causing collateral damage to resources supposedly unrelated to the operation, even removing the target resource entirely. In addition, name collisions may be used to exploit the process performing the relocation operation in a version of a *confused deputy attack* [25]. An adversary only requires write access to the source directory to produce source names that may lead to name collisions to perform an attack. We note that adversaries require fewer permissions to perform attacks using name collisions than other name confusion classes, which require write access to a directory used in name resolution of the target resource [54]. Thus, remote attacks using file system archives, such as tarballs and zip files, as well as file repositories, such as GitHub, can be the sources of attacks.

In practice, to perform a successful attack using a name collision, the victim process has to help the adversary in two ways. First, the victim process has to use the source resource in a relocation operation planted by an adversary as described above. In addition to archives, other activities, such as backups, may provide opportunities for exploitation of name collisions. In addition, ad hoc user actions copying files, e.g., from Linux to Windows in the Windows Subsystem for Linux, may result in unexpected and exploitable collisions. Second, the target directory of the relocation operation has to be predictable by the adversary to enable them to produce a source name that leads to a colliding destination name. Archives make this task much easier because the archive itself may be crafted to provide the target resource that is exploited by creating a collision with another archive file. A recent vulnerability in the `git` distributed revision control system demonstrates exactly this, as described below.

3.2 An Example Collision Vulnerability

Security vulnerabilities due to filename collisions across different file systems have been demonstrated in the wild. Consider a recent vulnerability in the `git` distributed version control system (CVE-2021-21300). This vulnerability results in remote code execution after cloning a maliciously crafted repository created on a case-sensitive file system to a case-insensitive file system.

Figure 2 depicts the maliciously crafted repository structure. Note that this directory structure works correctly on a case-sensitive file system. However, on case-insensitive file systems, the presence of the 'a' (small) and 'A' (capital) directories creates a collision that exposes a vulnerability. This collision results in a vulnerability when using `git`'s out-of-order checkout machinery. Git Large File Storage (LFS) uses out-of-order checkouts for downloading binaries in the background. Say the repository creator (adversary) marks 'A/post-checkout' for an out-of-order checkout. When a user clones this repository to a case-insensitive file system (e.g., NTFS), `git` performs a sequence of operations that: (1) replaces 'A' with the symbolic link 'a' and (2) writes the script file 'A/post-checkout' to '.git/hooks/post-checkout' due to the symbolic link 'a'. After the files are downloaded, `git` runs the script '.git/hooks/post-checkout' that the adversary provided, which is obviously undesirable.

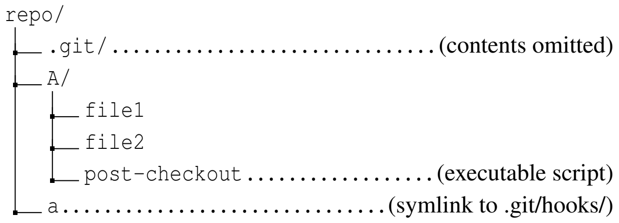


Figure 2: Example for Git CVE-2021-21300

In this case, a maliciously crafted `git` repository can be designed to provide a target resource of the symbolic link 'a', which when collided by 'A' in resolving the source resource 'A/post-checkout' redirects the operation to a directory chosen by the adversary using the symbolic link.

3.3 The State of Defenses for Name Confusions

Currently, operating systems provide no innate defenses to prevent name collisions, leaving the challenge to programmers. However, researchers have studied problems due to other types of name confusions extensively, proposing a variety of defenses [7–9, 30, 40–42, 44, 50–52]. However, researchers have shown that comprehensive program defenses are expensive [55] and that system-only defenses will always be prone to some false positives [5]. Leveraging limited program information [28, 53] still results in some false positives.

As a result, library commands for opening files have been extended in a variety of ways to prevent name confusions from occurring. The `open` command has been extended with flags to detect file squats (i.e., `O_EXCL|O_CREAT` to detect the presence of an existing file during file creation) and prevent unexpected use of aliases (i.e., `O_NOFOLLOW` to prevent following symbolic links). However, the use of squats and aliases is desirable in some applications, despite their risks. Further complicating the matter is that adversaries may exploit the gap between when a program validates a file system resource and opens that resource to create name confusions, known as time-of-check-to-time-of-use (TOCTTOU) attacks [4, 39]. The `openat` command has been added to enable programmers to avoid TOCTTOU attacks, by opening a file from a validated directory (i.e., file descriptor to the directory of the desired file). However, the successful use of `openat` requires the programmer to check for unwanted squats or aliases themselves. An alternative is proposed by the `openat2` command instead controls *how* files may be opened, such as requiring all file components accessed to be descendants of the directory from which the operation originates. However, `openat2` cannot prevent name confusions for some cases (e.g., using links across file systems). `openat` and `openat2` reduce the attack surface of squat and alias attacks, but do not eliminate them entirely, and depend on the programmer's additional actions to configure these commands and to check for TOCTTOU attacks.

At present, the above commands make no effort to help programmers address name collisions. As a result, utilities to perform copy and move operations and applications that may utilize file systems with multiple or mixed (e.g., ext4 and F2FS) case sensitivities or encodings may not detect and resolve name collisions correctly. We will examine the possible defenses for name collisions in §8.

4 Overview

In this paper, we aim to explore the impact that name collisions may have on file system security. To do this, we propose to examine three research questions.

RQ1: *How do applications invoke utilities that may allow unsafe name collisions?* In §6, we examine Linux packages to determine the most common options that applications employ for the utilities used to perform copies. We examine how frequently application packages use utilities in copy operations by scanning their scripts for such operations, as shown in Table 1.

RQ2: *When do the utilities for performing copy operations allow unsafe name collisions?* Recall that §3.1 defines the conditions under which an unsafe name collision may occur. This research question asks whether the utilities that applications may use to perform copy operations (e.g., `cp` and `tar`) prevent unsafe effects when a name collisions occur. For these utilities and the common options found in RQ1, we examine a variety of name collision scenarios to determine whether the utilities allow name collisions and their unsafe effects to occur as shown in Table 2a.

RQ3: *What correctness and security problems are caused by name collisions?* In §7, we examine three case studies where we show how name collisions cause programs to behave incorrectly. In particular, we show concretely how applications can be vulnerable to name collisions when target resources are deployed on case-insensitive or case-preserving file systems.

Impacts: A preview of our result is that: (1) many applications rely on these utilities to copy file system resources and repositories/archives; (2) the utilities used to copy file system resources and repositories/archives often allow unsafe name collisions, although the specific responses vary in ad hoc ways; and (3) applications currently lack defenses against name collisions, which can lead to incorrect operation and exploitable vulnerabilities.

5 Testing for Name Collisions

This section details an automated tool for testing the responses of common Linux utilities used for relocation operations to name collisions. As described in §3.1, a name collision is caused by creating a source name that will be converted to a destination name by the relocation operation that is equal to a target name in the target directory of the operation. Thus, our aim is develop a method to automate the generation of source resources with names that will lead to name collisions when relocated to case-insensitive targets and identify when operations allow the name collision to occur, detecting the effects of those operations.

5.1 Test Case Generation

The individual test cases are generated to test file system resources of various types, including regular files, directories, symbolic links (to files and directories), hard links, pipes, and devices. In addition, we have found that creating collisions in non-trivial directory structures may also lead to incorrect behaviors. Figure 3 shows an example test case where the directories as well as their contents result in a collision when transferred to a case-insensitive file system. As a result, we aim to generate test cases that result in name collisions at different depths of the directory being copied, as evidenced by the collision between directory names at depth 2 (i.e., "dir" and "DIR") and the impact on colliding resources of different types (i.e., a regular file "foo" and a pipe "foo").

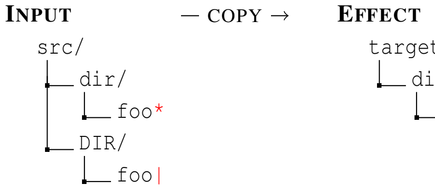


Figure 3: An example of squashing case-sensitive directory names and file names of two different types. Here, '*' means a regular file and '|' means file type is a named pipe.

Since we are testing the behavior of various utilities that perform relocation operations, we can control the source and target names in creating test cases. As a result, the choice of names is trivial. We create source directories that contain both the target resource (i.e., a resource copied first from the source to the target) and the source resource (i.e., a resource copied later by the utility to collide with the target resource (i.e., now in the target directory). This is similar to the way name collisions would occur when copying an archive or repository that causes a collision, as the `git` vulnerability. Since different utilities may process resources in different orders, we generate test cases with both orderings of resources that may cause collisions.

The only decisions then are what are the resource types of the source and target resources and where to place them in the source directory hierarchy to cause the desired collision to be created. Symbolic links, pipes, and devices only create interesting behaviors when used as target resources. For symbolic links, the unsafe effect is to follow the link to another file system resources, which only happens with the symbolic link as the target resource. For pipes and devices, the unsafe effect is to send the source resource's content to the pipe or device, which also is only possible if these are target resources.

As a result, the automated test generation produces test cases consisting of source and target resources of all combinations of potentially unsafe resource types and places these test cases at depth one and/or two of the file system hierarchy. For `rsync`, we specifically found an issue caused by a collision at depth two, but not at depth one (see §7.2).

5.2 Detecting Collision Effects

The key idea is to record the file system operations sufficiently to detect that an unsafe name collision has occurred. Since we design the test cases to create a name collision on a relocation operation, we want to detect when such an operation is a successful collision. Then, we need to determine the impact of the operation to classify the effect according to one of the ten effect options defined in §6.1.

We monitor file system operations using `auditd` to detect successful collisions. An example of a log indicating a collision is shown in Figure 4. In this example, a *create* operation creates a target resource named "root" using `openat` command, but a later *use* operation to the same resource (i.e., same device-inode pair, see below) is associated with a name "ROOT", which differs from the name used when the resource was created. Note that although the target resource was created on a case-insensitive file system, multiple names may be used that are resolved to the same name.

We say that a collision is successful when we detect a *use* of a target resource with a different name than that used to create the target resource. To detect such collisions, we first identify the file system operations that *create* a target resource, recording its combination of device[3] and inode identifiers, which form a unique resource identifier and its pathname. In Figure 4, the name component "root" will be important to detecting the collision. We then capture all the file system operations that *use* the target resource. In Figure 4, the pathname of the *use* operation differs between "root" and "ROOT", indicating a name collision.

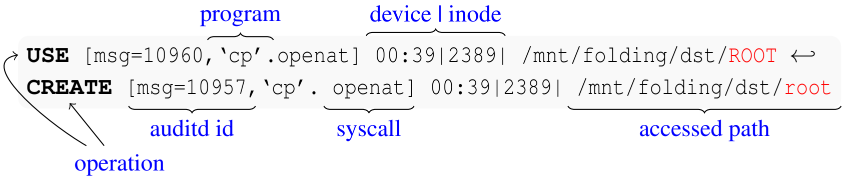
We also record a positive when a *use* operation deletes and replaces a resource from a prior *create* operation, as some collisions may cause the target resource to be deleted and the source resource to replace it. We validate that there is a *create* operation for the colliding destination name to verify the cause of the deletion is a collision.

To detect the effect of a name collision, we examine the resulting resource that now maps to the target name. We compare the source resource and target resource content and metadata to the resultant resource to determine whose content and/or metadata (i.e., source, target, or neither) the resource has. For tests on directories and hardlinks, we examine the directories and the resultant directory entries.

6 Name Collisions on Linux Copy Utilities

In this section, we examine how common Linux utilities that applications use to copy files from one part of the file system to another react when the copy operation causes a name collision in a case-insensitive directory. We note that the impact on move operations is similar because in most cases it simply performs a copy first and then deletes the source. However, when both the source and target are on the same file system, the underlying file system may directly relocate the contents of the source. This can result in unusual consequences on file systems that support per-directory case-[in]sensitivity. E.g., on ext4, moving a case-sensitive directory into a case-insensitive directory will preserve case-sensitive characteristics of the moved (or source) directory. However, when copying, the directories are newly created and these directories inherit the case-[in]sensitive characteristic from the parent directory. If the copy does not preserve attributes on directories, then all new directories will be case-insensitive under this scenario. Even though move works differently in certain cases, the collisions that may result from move have the same effect as that of copy. Hence, we only assess Linux utilities that perform copy operations below.

To quantify the ubiquity of these utilities, we survey their use by packages on Debian 11.2.0. We retrieve all packages from the Debian installation DVD and count the number of times the copy utilities are used inside the packages' scripts. The results are summarized in Table 1. Note that the listed uses of these utilities are lower bounds because we do not parse executable binaries. Hence, we miss uses where the utilities are invoked via system calls such as `system(...)`, `execve(...)`, etc.



```
USE [msg=10960,'cp'.openat] 00:39|2389| /mnt/folding/dst/ROOT ↩
CREATE [msg=10957,'cp'. openat] 00:39|2389| /mnt/folding/dst/root
```

Figure 4: Example violation reported by name collision testing.

[3] On Unix-like systems, each device is assigned a major and minor number. `auditd` reports these numbers (in hexadecimal) as XX:YY, where XX is the minor number and YY is the major number. Each file system mount point can be uniquely identified using these numbers.

Table 1: Prevalence of copy utilities

tar		zip		cp		cp*		rsync	
10	mc	21	texlive-plain-generic	78	hplip-data	12	dkms	28	mariadb-server
8	perl-modules	15	aspell	32	dkms	2	udev	5	duplicity
7	libkf5libkleo-data	11	libarchive-zip-perl	22	libltdl-dev	2	debian-reference-it	4	texlive-pictures
6	pluma	7	texlive-latex-recommended	20	autoconf	2	debian-reference-es	2	vim-runtime
6	mc-data	5	texlive-pictures	18	ucf	1	zsh-common	1	rsync
	...		...		...		...		...
107	TOTAL	69	TOTAL	538	TOTAL	25	TOTAL	42	TOTAL

We calculate the number of times that each command (tar, zip, etc.) is used inside scripts from various packages. We investigate 4752 .deb packages from the installation disk (DVD #1) of *Debian 11.2.0*. Only the top-five packages are shown (entries are sorted in descending order for each command).

6.1 Collecting Responses to Name Collisions

The name collision test cases and the responses of copy utilities are shown in Table 2a. The 'Target Type' column represents the resource type of the target resource that may be overwritten. The 'Source Type' represents the resource type of the source that collides with the target. The rest of the columns represent individual utilities and their responses to name collisions between a source resource of the source type and a target resource of the target type.

Below is a comprehensive list of the types of responses observed. Only "Deny" and "Rename" prevent name collisions from causing unsafe and possibly exploitable behaviors, although both may block legitimate functionality in some cases. "Ask the User" may result in an unsafe response if the user allows the target resource to be overwritten. Note that more than one response is possible for each test case.

Delete & Recreate ($\times$) *Delete* the target resource and *create* a new resource based on the source resource. The new resource's type, as well as its data and metadata, is determined by the source resource. The target resource is lost without any notification.

Overwrite ($+$) *Overwrite* the data and metadata of the target resource using the source resource. Unlike *Delete & Recreate*, the name of the target resource is preserved. If file `foo` is being overwritten with file `FOO`, then the final file will be named `foo` but will have the contents and metadata of file `FOO`.

Corrupt (C) Contents of a resource that is not involved in name collision (i.e., not the target resource) is modified. For a more in-depth discussion, refer to §6.2.5.

Metadata Mismatch ($\neq$) After a successful copy of a given source resource, some metadata, such as its name, UNIX permissions, user or group ID, extended attributes, or timestamp, remain from the target resource, creating a resource with a *mismatch* between the data (from the source) and the metadata (from the target).

Follow Symlink (T) Follow (or traverse) *symbolic links*, *even when explicitly directed not to do so*.

Rename (R) The source name is *renamed* automatically to avoid creating a name collision, such as by appending a counter, resulting in a copy of the source resource in the target resource's directory with a non-colliding name.

Ask the User (A) To resolve a collision, *ask the user* to choose from a list of actions, such as to overwrite the target resource, skip copying the source resource, rename the target resource, abort, etc.
Note that the user can still choose a response that results in adverse consequences. For instance, if the user chooses to overwrite the target, the target's data and metadata are modified using the source.

Deny (E) *Deny* the copy associated with a collision and report an error.

Crashes (∞) Collisions can result in the program hanging (e.g., going into an infinite loop) or *crashing*.

Unsupported file type ($-$) Does not support copying a resource if the source resource is of this file type. Note that if hardlinks are not recognized by a utility, then it simply creates a fresh copy of the underlying file.

The exact command-line flags used used to generate Table 2a are listed in Table 2b. To identify these flags, we analyzed 4,752 `.deb` packages on Debian 11.2.0's installation DVD. We found that the most commonly used flags enabled the following functionality.

- Support recursively copying all directories.
- Support copying symbolic-links and hard-links as-is but *do not follow* them.
- Preserve metadata such as UNIX permissions, extended attributes (xattr), timestamps, and owner/group IDs (uid/gid).

Before examining the responses in Table 2a, we briefly note some additional context for two of the columns.

cp vs. cp* Both of these represent the same executable binary. The difference is in the way the command-line arguments are passed to the binary. Specifically, the format of specifying the source directories is different.

Consider that the source directory (to be copied) is `foo`. For `cp`, we will pass it as `foo/` while for `cp*` we will use `foo`. Note the trailing / is missing in the latter case. Just this difference significantly changes the behavior of `cp` as noted in Table 2a.

We use the `cp*` method of invocation coupled with shell completion, e.g., '`cp src/* /target`' where the shell re-

Table 2: Name Collision Responses for Popular Linux Utilities

Name Collision between Target Type	Source Type	`tar`	`zip`	`cp`	`cp*`	`rsync`	Dropbox
file	file	×	*A*	*E*	+≠	+≠	*R*
symlink (to file)	file	×	*A*	*E*	+*T*	+≠	*R*
pipe/device	file	×	–	*E*	+	+	–
hardlink	file	×	–	*E*	+≠	+≠	–
hardlink	hardlink	*C*×	–	*E*	*C*×	*C*+≠	–
directory	directory	+≠	+≠	*E*	+≠	+≠	*R*
symlink (to directory)	directory	+	∞	*E*	*E*	+*T*	*R*

(a) This table shows results of copying files/directories from a case-sensitive to a case-insensitive file system. `cp*` refers to `cp` being used with shell completion. For e.g., '`cp * /target`' which copies all items from the current directory to `/target` directory.

Utility	Version	Flags
`tar`	1.30	`-cf/-x`
`zip`	3.0	`-r -symlinks`
`cp`	8.30	`-a`
`rsync`	3.1.3	`-aH`

(b) This table lists the version of utilities and command-line flags used for the experiments. For `tar`, `-cf` was used to create the archive and `-x` to expand the archive.

× Delete existing file and create new file
+ Overwrite existing file. For directories, merge their contents.
≠ Mismatch between content and metadata

A Ask user to resolve the collision
T Follow (or traverse) symlink
C Corrupts non-colliding files
E Deny operation and report error

∞ Program crashes, or hangs
– Ignore unsupported file type (for hardlinks create regular file instead)
R Rename colliding file/directory

places `src/*` with each individual entry present inside `src` sans the trailing `/`. When testing the `cp` method, we change the command to '`cp src/ /target`'.

Dropbox Strictly speaking, *Dropbox* [11] is not a copy utility but a popular file synchronization utility. It is intended to replicate entire directories across multiple machines and file systems.

We mention Dropbox to highlight its distinct response to handling *potential* name collisions. Even when the underlying file system is case-sensitive, Dropbox treats it as *case-insensitive*. It proactively renames the files and directories to avoid name collisions that could occur if they were transferred to a case-insensitive file system. Note, however, that its renaming strategy is not even uniform across platforms: For example, the Dropbox application appends "(Case Conflicts)", "(Case Conflicts 1)", etc. to the file/directory names in case of a potential collision, whereas, when using their web-based interface, they append "(1)", "(2)", etc. instead.

6.2 Unsafe Responses to Name Collisions

Several responses shown in Table 2a demonstrate that utilities often allow unsafe responses to name collisions. In this section, we examine some of the more concerning responses to show how utilities delegate responsibility for security against name collisions to the applications that invoke them. For the examples in upcoming sections, `src/` and `target/` are on case-sensitive and case-insensitive file systems respectively.

6.2.1 Silent data loss with *tar, cp* & rsync*

Name collisions involving files generally result in silent data loss. From Table 2a, we can see that `tar` deletes and recreates (×) files when collisions occur. Hence, when there is a name collision between `foo` and `FOO`, only one of these files will remain in the target directory. The other file is permanently lost without any notification.

Similar to `tar`, `cp*` and `rsync` also lose files silently. However, their behavior of overwriting (+) files results in other problems that are discussed later in this section.

Unlike `tar`, `zip` and `cp` will ask a user for next steps (*A*) or report an error (*E*) respectively. Hence, they are not prone to silently losing files.

6.2.2 Merge directories with *tar, zip, rsync & cp**

Name collisions involving two directories results in their contents (files, directories, etc. inside the directory) being merged. All of `tar, zip, rsync,` and `cp*` will silently merge directory contents without notifying the user. Figure 5 highlights this issue using a directory listing.

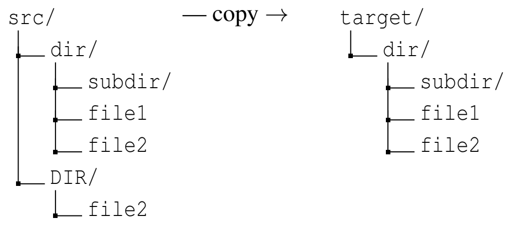


Figure 5: Impact of merging directories

In this example in Figure 5, the data of file `file2` is overwritten by the content written last in the copy operation. For example, if `src/DIR`'s contents are written last, then its content for `file2` is preserved and `src/dir`'s is lost.

Furthermore, when the colliding directories have different UNIX permissions, a collision results in metadata mismatch (≠). With respect to Table 2a, the UNIX permissions of the target resource are overwritten with permissions of the source resource.

In Figure 5, consider `src/dir/` with perms=700 and an adversary who creates `src/DIR/` with perms=777. After a copy (using any of the above utilities), `target/dir/` will have perms=777 effectively giving the adversary permission to the contents of the original `src/dir/`.

6.2.3 Stale names

Whenever utilities resort to overwriting (+), we end up with stale file/directory names. For example, consider a name collision between a target resource `foo` (file content: 'bar') and a source resource `FOO` (file content: 'BAR'). After copying with `rsync` or `cp*`, we will end up with file `foo` whose contents are 'BAR'.

The problem with such name collisions is that to the end user (or other programs), it will appear that `foo` was successfully copied while in reality `FOO` was copied. Just using the filename is not enough to discern which files were successfully copied. This is especially true for case-preserving file systems where the user has the expectation of the filenames being preserved. Hence, it is not unreasonable for the user to expect `foo` should contain `bar`.

6.2.4 Symbolic link traversal at target

Name collisions between *symlink (to file) and a regular file* results in `cp*` following the symlink (T) and overwriting (+) its target's contents with that of the regular source file. With regards to Table 2a, if the target resource is a symbolic link and the source resource is a file, then `cp*` ends up following the symlink and writing data to the resource referenced by the symlink.

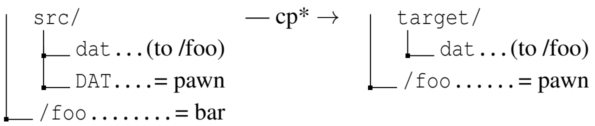


Figure 6: Following symlink

Figure 6 illustrates this case with an example. `src/dat` is a symbolic link to `/foo` and `/foo` contains 'bar'. Mallory (our adversary) does not have write access to `/foo` but does have access to `src/`. She creates `src/DAT` which contains 'pawn'.

Then the administrator starts the copy using: `cp -a src/* target/`. At this point, `cp` first creates the symlink `target/dat`. Then it overwrites (+) this symlink with the contents of `src/DAT`, effectively updating the file `/foo`. After the copy has completed, `/foo` contains 'pawn'.

*cp** has no command-line options to prevent traversal of symbolic links at the target. Only link traversal at the source can be turned off via command-line flags.

6.2.5 The case of *hardlink – hardlink* name collisions

During a copy when hardlinks (whose targets are different) collide, it can corrupt (*C*) other non-colliding files and create spurious hardlinks. Table 2a shows that this behavior is exhibited by `tar, cp*`, and `rsync`. An interesting observation is that, regardless of whether the utility's behavior is *Delete & Recreate* (×) or *Overwrite* (+), this problem affects both.

To understand this scenario, consider Figure 7 that uses `rsync` to perform the copy. The same color coding represents files that are hard-linked to each other. So `src/hfoo` and `src/zzz` are hard-linked, representing the same file. These files contain 'foo'. Similarly, `src/hbar` and `src/ZZZ` are hard-linked and they contain 'bar'.

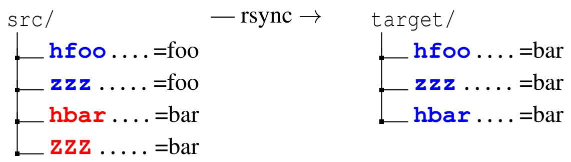


Figure 7: *hardlink – hardlink* name collision

After copying using `rsync`, `target/` contains three files that are all hard-linked to each other. Unlike the `src/` directory, `target/hfoo`, `target/hbar`, `target/zzz` are all hardlinks of each other and they contain 'bar'.

Additionally, note that although the name collision happened between `zzz` and `ZZZ`, the contents of `hfoo` were replaced. Even `tar`, which deletes the old file and recreates it, exhibits this behavior.

The following order of operations undertaken by `rsync` result in this behavior.

1. Copy `src/hbar` to `target/hbar`. Now `target/hbar` contains 'bar'.
2. Copy `src/zzz` to `target/zzz`. Now `target/zzz` contains 'foo'.
3. In `target/`, hardlink `ZZZ` to `hbar`. Due to name collision, this effectively changes `zzz` to be hard-linked to `hbar`. Now `target/zzz` contains 'bar'.
4. In `target/`, hardlink `hfoo` to `zzz`. Now `target/foo` contains 'bar'. Additionally, all three files inside `target/` are hard-linked to each other.

The above copy is semantically different from the `src/`. Specifically, name collision results in *distinct* sets of files getting hard-linked with each other at the `target/`.

7 Case Studies

In this section, we examine case studies where name collisions cause unsafe behaviors, some of which are exploitable.

7.1 `dpkg` Package Manager

`dpkg` is the package manager on Debian OS and its derivatives such as Ubuntu. `dpkg` packages are compressed tarballs with extension `.deb`. When `dpkg` processes a package, it tracks all files it creates during package installations in a database. Before installing a new package, `dpkg` leverages this database to ensure that any files of previously installed packages will not by overwritten by this new package thereby preventing potentially malformed packages from corrupting the system.

On the other hand, we have observed that `dpkg` will allow a package installation to replace any file whose name is not in its database, even privileged user files. Thus, as long as a file in a package has a filename that does not match the filename of another package's file, `dpkg` will install the file, silently replacing any existing file.

However, regardless of the underlying file system, the above database is matched in a *case-sensitive* manner. This allows new packages to *replace files* of previously installed packages via name collisions effectively circumventing the safeguards in dpkg.

In addition, and perhaps even more seriously, dpkg may allow an adversary to replace a package's customized config file with the default, reverting important changes. deb packages can mark certain files as configuration (or config) files. During package upgrades, if dpkg spots modifications to these config files then it prompts the user to review the changes.

However, the config files are also matched in a case-sensitive manner. Under name collisions, dpkg will just replace the original package's config file with the config file of the new package. For services, such as sshd, httpd, etc., config files are critical to their security, so such overwrites can potentially make the system vulnerable..

Reporting We have reported these issues to the maintainers of dpkg. The maintainers of dpkg have since updated their package documentation [10] to warn end user communities not to use dpkg where targets may be case-insensitive (i.e. specific directories, or entire file systems).

During our discussions, we analyzed 74,688 packages and found 12,237 filenames from those packages would collide if a case-insensitive file system were used, breaking multiple packages that contain these files. The name collision problem is fundamentally entrenched into the way dpkg is implemented because it reasons about *names* without involving the underlying file system(s).

7.2 Rsync

rsync demonstrates vulnerable behavior when processing name collisions involving *directories*. During copy, the default behavior of rsync is to simply recreate the symbolic links present at source. However, when colliding directories contain sub-directories and symbolic links with the same name, the collision causes rsync to suffer from link traversal[4].

Consider the source directory listed in Figure 8. Here, the directories topdir/ and TOPDIR/ only differ in case. So when copying to a case-insensitive file system, rsync will encounter a name collision.

```
src/
├── topdir/
│   └── secret/ .......................... symlink to /tmp
└── TOPDIR/
    └── secret/
        └── confidential ..................... regular file
```

Figure 8: Case-sensitive source that rsync is copying

We use the following command to perform the copy:

```
rsync -a src/ dst/
```

where,

-a recursively copy directories, preserve symlinks, timestamps, and discretionary access control permissions

src/ is case-sensitive

dst/ is case-insensitive

After the copy is completed, the newly created files are shown in Figure 9. Note that the file named confidential ends up in /tmp.

```
dst/
│ └── TOPDIR/
│     └── secret/ .......................... symlink to /tmp
└── /tmp/confidential ........................ link traversal
```

Figure 9: After copying to case-insensitive destination

rsync has created the /tmp/confidential file by following the symbolic link dst/TOPDIR/secret.

Below, we describe how this situation can be exploited. Consider an adversary who wants to access a confidential file in TOPDIR/ to which she lacks any access. However, she knows that TOPDIR/ is processed by a backup operation using rsync. If she can create a sibling directory topdir/, to which she will have read-write access, she can direct rsync to write the confidential file (inside TOPDIR/) to any directory of her choosing by creating a symbolic link inside topdir/ to that directory.

Reporting We reported this issue to the rsync maintainers, and they told us that user's should not use rsync with non-case honoring file systems. However, we have concerns about the user community following such a recommendation in this case, since rsync is often used by individuals.

In the course of these discussions, we learned the cause of the incorrect behavior. rsync assumes a one-to-one mapping of directories between source and target file systems. When a name collision results in two source directories being mapped to a single directory in the target, rsync can be tricked into incorrectly predicting the target file type. In the presented scenario, a symbolic link src/topdir/secret (to a directory) is incorrectly inferred to be a regular directory src/TOPDIR/secret.

rsync uses the O_NOFOLLOW flag with open() to prevent link traversal and uses openat()/openat2() to contain link traversals within a directory hierarchy, but this strategy fails when the symbolic link is treated as a directory.

7.3 Apache httpd

Security of certain applications relies on the security parameters of the underlying file system. One such application is Apache's httpd. It allows access to the underlying file system via the HTTP protocol, relying on the UNIX Discretionary

[4] In this case, the name collision makes the alias exploitable, again combining name confusions.

Access Control (DAC) permissions[5] to mediate the access. For example, files are accessible over HTTP only if: (i) its UNIX group is www-data and has read permission for the group, or (ii) has world-readable UNIX permissions.

Using utilities for copying directories between systems can silently alter these DAC permissions in unintended ways, leading to serious security lapses. We illustrate this scenario using Apache httpd and migration of its data using tar. To study the impact of name collisions on the security parameters, we assume that the migration happens from a case-sensitive to a case-insensitive file system. The behavior of tar discussed below draws from the discussion of Table 2a.

To protect sensitive directories, httpd can be configured to only allow authenticated users to access specific directories. A commonly used approach is to configure authentication via the .htaccess file [1] which lists the valid users/groups allowed to access a specific directory over HTTP. All subdirectories inside the sensitive directory are also protected. We show that the use of additional security-oriented files can be exploited under the presence of name collisions.

Scenario httpd serves the contents of www/ (of Figure 10) over HTTP. Initially, www/ is stored on a case-senstive file system. The directory hidden/ is inaccessible over HTTP since the *others* permissions are cleared. Next, protected/ is configured to be accessible only to specific users using the .htaccess file.

```
www/
├─ hidden/ ............................. perm=700
│  └─ secret.txt
├─ protected/..........group=www-data, perm=750
│  ├─ .htaccess ............(only allow valid users)
│  └─ user-file1.txt
└─ index.html
```

Figure 10: www/ on case-sensitive file system

Adversary A UNIX user called Mallory has read-write access to www/ directory. However, DAC permissions prevent her from accessing hidden/ directory because its owner is another user. Additionally, protected is inaccessible since Mallory does not belong to the group www-data.

She modifies www/ as shown in Figure 11 and adds the HIDDEN/ and PROTECTED/ directories with the intent of gaining access to hidden/ and protected/ via a name collision.

Vulnerability tar is used to migrate the adversary-modified www/ directory to another system that uses a *case-insensitive* file system. Figure 12 shows the state of the file system once the tarball (archive format of tar) is extracted.

Now, the previously inaccessible hidden/ directory is now accessible over HTTP. Additionally, since the .htaccess file is cleared, unauthenticated users will be allowed to view protected/ over HTTP.

```
www/
├─ hidden/ ..................................perm=700
│  └─ secret.txt
├─ HIDDEN/ ..................................perm=755
├─ protected/ ..............group=www-data, perm=750
│  ├─ .htaccess .................(only allow valid users)
│  └─ user-file1.txt
├─ PROTECTED/ ............................... perm=755
│  └─ .htaccess ............................(empty file)
└─ index.html
```

Figure 11: Adversary modified www/ on the case-sensitive file system

```
www/
├─ hidden/ ..................................perm=755
│  └─ secret.txt
├─ protected/ ............................... perm=755
│  ├─ .htaccess ............................(empty file)
│  └─ user-file1.txt
└─ index.html
```

Figure 12: www/ after migrating to case-insensitive file system

Reporting We have reported this scenario to the Apache maintainers, but have not yet reached a resolution. Using Table 2a, we can reason about the above problems. Under a *directory* – *directory* collision, tar incorrectly modifies metadata. This happens for the hidden/ – HIDDEN/ collision. Here, DAC permissions of the latter are applied to the former resulting in the leakage of secret files.

For *directory* – *directory* collisions, tar will also merge contents of both directories. For protected/ – PROTECTED/ collision, this merger results in the empty .htaccess file overwriting the original one that restricts access to authorized users. The end result is that all users are now allowed access to the new protected/ directory.

8 Potential Defenses

As discussed in the context of name confusion attacks in general in §3.3, it can be difficult to produce defenses to prevent name collisions as well. In this section, we discuss some options and their limitations.

Name collisions are due to differences in case folding rules among file systems, e.g., case sensitivity and encodings, so it is difficult to ensure that name collisions cannot happen. Suppose a system has only one file system. Even then, an archive constructed on another file system using conflicting case folding rules may cause name collisions to occur when expanding the archive. Since user-space programs cannot determine the case-folding rules that may be applied to a file, user-space solutions alone will be unreliable. In addition, they may be prone to TOCTTOU attacks [3,4]. Thus, extending library calls like realpath to detect name collisions will not sufficiently solve the problem. In addition, system solutions

[5] If the system supports Mandatory Access Control (MAC), then DAC is used in conjunction with MAC.

lack knowledge of the programmer intent that caused the collision and hence, a systems-only defense for name confusion will suffer from false positives [5].

For example, one idea may be to write a wrapper to vet archives prior to expansion operations (e.g., tar and zip) to validate that each file in the archive will result in a distinct file after expansion. One way to do this is to check for name collisions among all the files in the archive. Although the notion that no two files in an archive should collide seems intuitively reasonable, there are critical drawbacks to this defense. First, the target directory may already have files that may result in collisions, limiting its utility. Second, targets that support per-directory case-sensitivity can switch between case-sensitive and case-insensitive lookups when resolving a filepath, leading to incorrect assumptions about case-sensitivity and being prone to race conditions. Finally, the case folding rules applied by such a wrapper are not guaranteed to be the same as those of the target directory.

As a result, we envision that defenses for name collisions will evolve in a manner similar to defenses for name confusions that utilize the open commands (i.e., openat and openat2). Consider that these commands have flags to check whether a file of a corresponding name exists at creation time, only opening that file when created anew (i.e., O_CREAT|O_EXCL). This call prevents a name collision from overwriting an existing file, but it may be too strong a defense. Suppose one really wants to overwrite files of the same name, but prevent name collisions from modifying files that actually have differing names (i.e., that only match due to case folding). In this case, a new flag is necessary, such as O_EXCL_NAME, which prevents opening a file when the names differ, but not when such names match. Using this flag would enable the virtual file system to compare names in a case-insensitive manner (i.e., based on the case folding and normalization for target directory) to detect collisions and compare names in a case-sensitive manner to determine matches. However, at present, the virtual file system cannot choose the type of matching (case-sensitive or case-insensitive), nor can it identify the type of matching done by the underlying physical file system.

Unfortunately, even with variants of the open command and other defenses, such as FileProvider classes in Android, programmers continue to make mistakes that lead to errors and vulnerabilities. The challenge is for programmers to determine the intent of their operation, understand the threats faced in such an operation, and configure these complex, low-level commands in such a way that they block the threats while satisfying the intent. Until file system APIs enable this combination of requirements, errors will remain common.

9 Related Work

Researchers have proposed defenses to thwart name confusion attacks for alias and squat cases. To the best of our knowledge, no defenses for name collisions have been proposed.

System Defenses Researchers have long known about name confusion attacks [3,4] and have proposed a variety of system defenses [7–9, 30, 40–42, 44, 50–52]. In a system defense, the operating system aims to enforce an invariant that prevents name confusion attacks from succeeding. However, as discussed in §8, without programmer intent such defenses will suffer from false positives [5]. Hybrid defenses have also been proposed [53,55] where the operating system introspects into the process to leverage program state along with file system state in enforcement. Even though false positives are reduced, these techniques lack explicit programmer intent to fully eliminate all false positives.

Program Defenses As a result, systems provide APIs for programmers to decide how to handle name confusion attacks. Several file system APIs include flags to avoid using symbolic links entirely (e.g., O_NOFOLLOW flag for the open system call), but in many cases programmers want to be able to use symbolic links. Researchers have proposed program-specific defenses to configure APIs or program frameworks for preventing name confusion attacks [27,43,47,56]. More advanced commands for file allow programmers to manage *how* files are open, including the impact of symbolic links. For example, the openat system call enables the user to open a directory first to validate its legitimacy before opening the remaining path. openat2 explicitly constrains how name resolution is performed to reduce the potential for attacks.

10 Conclusion

Interactions among file systems with differing encoding/case-sensitivity semantics can lead to name collisions when performing maliciously crafted, or even ostensibly benign, copy operations. We explored the impact that these name collisions can have on file system security. Current operating systems do not directly prevent name collision-based attacks, delegating that responsibility to the programmers. In investigating the utilities used to copy file system resources and repositories/archives, we demonstrate that they often allow unsafe name collisions and lack the sort of uniformity in name-collision handling against which safer use policies could be easily crafted. Further, we show that many applications rely on potentially unsafe use of these utilities, opening themselves up to exploitable vulnerabilities. We examine three case studies demonstrating concrete vulnerabilities to name collisions. Finally, we suggest directions for future research to systematically defend against name collision attacks.

Artifacts

The artifacts produced during the work can be found at https://github.com/mitthu/name-confusion. It contains scripts to generate the test cases and run commands required to create Table 2a. Furthermore, it contains the tool for analyzing auditd traces and extracting relevant create-use pairs (see §5.2). Finally, there are proof-of-concept scripts to reproduce the vulnerabilities in dpkg and rsync.

References

[1] Apache HTTP Server: Authentication and authorization. https://httpd.apache.org/docs/2.4/howto/auth.html#gettingitworking.

[2] Bazaar's handling of case insentitive file systems. http://doc.bazaar.canonical.com/bzr.1.12/developers/case-insensitive-file-systems.html.

[3] Richard Bisbey, Gerald Popek, Jim Carlstedt, et al. Protection errors in operating systems: Inconsistency of a single data value over time. Technical report, University Of Southern California Marina Del Rey Information Sciences, 1975.

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

[5] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting UNIX file-system races via algorithmic complexity attacks. In *2009 30th IEEE Symposium on Security and Privacy*, pages 27–41. IEEE, 2009.

[6] Case mapping vs. case folding. https://www.w3.org/TR/charmod-norm/#definitionCaseFolding.

[7] Suresh Chari, Shai Halevi, and Wietse Z. Venema. Where do you want to go today? Escalating privileges by pathname manipulation. In *NDSS*. Citeseer, 2010.

[8] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[9] Drew Dean and Alan J. Hu. Fixing races for fun and profit: How to use access (2). In *13th USENIX Security Symposium (USENIX Security 04)*, pages 195–206, 2004.

[10] dpkg FAQ: diff of updated documentation. https://wiki.debian.org/Teams/Dpkg/FAQ?action=diff&rev2=78&rev1=77.

[11] Dropbox. https://www.dropbox.com/.

[12] Linux kernel documentation (v5.2): ext4 support. https://www.kernel.org/doc/html/v5.2/admin-guide/ext4.html.

[13] Linux kernel documentation: Flash-friendly file system (F2FS). https://docs.kernel.org/filesystems/f2fs.html.

[14] F2FS: Support case-insensitive file name lookups (patch). https://patchwork.kernel.org/project/linux-fsdevel/patch/20190719000322.106163-3-drosen@google.com/.

[15] ciopfs: case insensitive on purpose filesystem. https://www.brain-dump.org/projects/ciopfs/.

[16] ext3ci – case insensitive ext3 filesystem for Linux 2.6.32. http://bill.herrin.us/freebies/.

[17] Linux kernel documentation: FUSE. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[18] IOMap. https://www.mono-project.com/docs/advanced/iomap/.

[19] JFS filesystem (man page). https://www.unix.com/man-page/redhat/8/mkfs.jfs/.

[20] Linux kernel documentation: NTFS3. https://docs.kernel.org/filesystems/ntfs3.html.

[21] NTFS-3G driver. https://github.com/tuxera/ntfs-3g.

[22] XFS filesystem (man page). https://manpages.ubuntu.com/manpages/trusty/man8/mkfs.xfs.8.html.

[23] ZFS on Linux project. https://zfsonlinux.org/.

[24] Git's patch for CVE-2021-21300. https://github.com/git/git/commit/684dd4c2b414bcf648505e74498a608f28de4592.

[25] Norman Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22:36–38, October 1988.

[26] Daniel Kachakil. Multiple vulnerabilities in Android's Download Provider (CVE-2018-9468, CVE-2018-9493, CVE-2018-9546). https://ioactive.com/multiple-vulnerabilities-in-androids-download-provider-cve-2018-9468-cve-2018-9493-cve-2018-9546/.

[27] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. *ACM SIGOPS Operating Systems Review*, 41(6):321–334, 2007.

[28] James A. Kupsch and Barton P. Miller. How to open a file and not get hacked. In *2008 Third International Conference on Availability, Reliability and Security*, pages 1196–1203. IEEE, 2008.

[29] Yu-Tsung Lee, William Enck, Haining Chen, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Daimeng Wang, Giuseppe Petracca, and Trent Jaeger. PolyScope: Multi-Policy access control analysis to compute authorized attack operations in android

systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2579–2596, 2021.

[30] Kyung-Suk Lhee and Steve J. Chapin. Detection of file-based race conditions. *International Journal of Information Security*, 4(1):105–119, 2005.

[31] Linus Torvalds's comments on case-insensitive file systems. https://patchwork.kernel.org/project/linux-fsdevel/cover/20181206230903.30011-1-krisman@collabora.com/#22369005.

[32] Eliminating Android wrapfs "hackery". https://lwn.net/Articles/718640/.

[33] mm: shmem: Add case-insensitive support for tmpfs. https://lwn.net/Articles/850214/.

[34] Case-insensitive ext4. https://lwn.net/Articles/784041/.

[35] Filesystems and case-insensitivity. https://lwn.net/Articles/772960/.

[36] Case-insensitive filesystem lookups. https://lwn.net/Articles/754508/.

[37] Network filesystem topics. https://lwn.net/Articles/685431/.

[38] Slava Makkaveev. Man-in-the-Disk: Android apps exposed via external storage. https://research.checkpoint.com/2018/androids-man-in-the-disk/.

[39] William S. McPhee. Operating system integrity in OS/VS2. *IBM Systems Journal*, 13(3):230–252, 1974.

[40] OpenWall Project - Information security software for open environments. http://www.openwall.com/.

[41] Jongwoon Park, Gunhee Lee, Sangha Lee, and Dong-Kyoo Kim. RPS: An extension of reference monitor to prevent race-attacks. In *Pacific-Rim Conference on Multimedia*, pages 556–563. Springer, 2004.

[42] Mathias Payer and Thomas R. Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 215–226, 2012.

[43] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 161–176, 2009.

[44] Calton Pu and Jinpeng Wei. A methodical defense against TOCTTOU attacks: The EDGI approach. In *Proceedings of the 2006 International Symposium on Secure Software Engineering*, 2006.

[45] Samba: Implementation of SMB/CIFS protocol. https://www.samba.org/.

[46] smb.conf.5 (man). https://www.samba.org/samba/docs/4.15/man-html/smb.conf.5.html.

[47] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, 1999.

[48] Case-sensitive filesystems not supported on Mac. https://help.steampowered.com/en/faqs/view/0395-A862-13F3-6E82.

[49] SteamOS. https://store.steampowered.com/steamos.

[50] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *FAST*, volume 8, pages 1–18, 2008.

[51] Eugene Tsyrklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[52] Prem Uppuluri, Uday Joshi, and Arnab Ray. Preventing race condition attacks on file-systems. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 346–353, 2005.

[53] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. JIGSAW: Protecting resource access by inferring programmer expectations. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 973–988, 2014.

[54] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. STING: Finding name resolution vulnerabilities in programs. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 585–599, 2012.

[55] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. Process firewalls: Protecting processes during resource access. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 57–70, 2013.

[56] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[57] Wine: Wine is not an emulator. https://www.winehq.org/.

[58] What is the Windows subsystem for Linux? https://docs.microsoft.com/en-us/windows/wsl/about.

[59] Diving into SDCardFS: How Google's FUSE replacement will reduce I/O overhead. https://www.xda-developers.com/diving-into-sdcardfs-how-googles-fuse-replacement-will-reduce-io-overhead/.