

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Wormholes in the File System: Understanding the Misunderstanding of Symlinks

Yongheng Liu, Lei Zhang, Yuhang Zhao, and Yuzhou He, *Fudan University*

https://www.usenix.org/conference/usenixsecurity26/presentation/liu-yongheng

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Wormholes in the File System: Understanding the Misunderstanding of Symlinks

Yongheng Liu
Fudan University

Lei Zhang *
Fudan University

Yuhang Zhao
Fudan University

Yuzhou He
Fudan University

Abstract

Symlinks, a file system feature supported at the operating system level for more than four decades, are deeply integrated across the software stack, making them an indispensable component of modern computing environments. The symlink mechanism is highly flexible, allowing references to nearly any location in the file system and enabling complex resolution behaviors. However, this flexibility also introduces significant misunderstandings for developers, which has led to widespread misuse and become a primary vector for critical security vulnerabilities in file-handling software, e.g., arbitrary file write and code execution.

In this paper, we present the first systematic study of how symlinks are incorrectly handled and the substantial security risks. Specifically, we study how programming language standard libraries improperly utilize symlink-related system calls, and how such misuse and misunderstanding propagates to downstream applications which results in various inadequate path validations. To further understand whether these inadequate path validations could be exploited in real world, we design and implement five path validation bypass schemes and three arbitrary write methods. Our evaluation across 85 projects uncovered 16 arbitrary file write vulnerabilities, two of which have been assigned high-severity CVEs. We also demonstrate their severe impacts in real-world scenarios and propose multiple mitigation strategies. We have responsibly disclosed these issues to the affected vendors.

1 Introduction

Symlink, also known as symbolic link or soft link, constitute one of the most enduring and elegant abstractions in file system design. Since their introduction to BSD Unix in 1982 and subsequent standardization in POSIX, symlinks have served as a fundamental building block for system organization for over four decades [35]. Unlike a hard link, which is directly entry that refers to the same file's underlying data structure (inode), a symlink's file content consists of a pathname that references another file or directory. This mechanism provides significant flexibility. By simply updating a link's target, users can seamlessly switch between different software versions or toggle execution contexts, facilitating an advanced approach to system organization. Thus, symlinks are widely used in software installation, deployment, and environment configuration.

Building on this foundational design, symlinks are deeply integrated throughout the modern software stack. Operating systems provide native system calls for the creation and resolution of symlinks, and this functionality is supported and utilized by a wide range of downstream software. Core system utilities such as `pwd`, `ls` and `realpath`; standard libraries in programming languages, including `pathlib` in Python and `java.nio.file` in Java; development tools and package managers like git, npm and pip; and containerization platforms such as Docker all gracefully handle symlinks, relying on the operating system for transparent resolution. Consequently, symlinks have become a foundational mechanism of modern software ecosystems. Most applications benefit from the simplified and flexible path handling enabled by symlinks.

However, the flexibility of symlinks also introduces pervasive misunderstandings and substantial security risks. [9] The diversity and parametric complexity of file-related system calls often cause developers to overlook the specific requirement for symlink processing. When designing path-processing interfaces, standard library developers in different programming languages adopt inconsistent strategies for file type identification and path normalization. This inconsistency can even result in the same pathname resolving to divergent physical locations [22]. Downstream application developers, typically unaware of these internal implementation details, may misuse standard library interfaces. This misunderstanding leads to insufficient path sanitization and flawed path validation when handling path with symlinks.

Such symlink-induced inconsistencies in path resolution and inadequacies in path validation can lead to severe security vulnerabilities, including path traversal and arbitrary file write,

*Corresponding author

which can further escalate into Remote Code Execution or Denial of Service attacks. Therefore, identifying the misuse of symlinks, especially for the security flaw in path validation mechanisms, is a critical problem but not well-studied.

In this paper, we present the first systematic study of security issues arising from the misunderstandings of symlink. We investigate the irregular design of path resolution in programming language standard libraries as well as neglect of symlink's flexible features in path validation and sanitization. Starting from system calls, we investigate how these issues propagate to downstream applications and analyze their risks in real-world cases. However, for symlinks, existing work only focused on race conditions [7, 12] and link-following [36, 38] vulnerabilities. Such attacks typically exploit time-of-check-to-time-of-use (TOCTOU) gaps between symlink verification and usage by replacing the symlink target. In contrast, our work investigates security flaws rooted in the defective resolution of path containing symlinks. These vulnerabilities emerge from the combination of inconsistent symlink-handling implementations in programming language standard libraries and inadequate path validation within downstream applications. Moreover, the attacks studied in this paper do not require an attacker to first obtain a low-privilege account to manipulate symlinks; instead, they directly leverage the victim's high privileges to achieve severe security consequences.

Specifically, we first conduct a systematic investigation of the inconsistencies in path resolution interfaces across different programming languages arising from symlink-induced misunderstandings. We further study how these inconsistencies lead to confusion and misuse. Programming language standard libraries implement high-level abstraction of path handling by wrapping multiple system calls, thus introducing discrepancies in design logic and functionality details. By designing a variety of path test cases combining path patterns with symlink targets, our study finds four aspects of confusion in file type identification interfaces and four different strategies in path normalization interfaces. We also analyze the risks caused by such implementation inconsistencies.

Second, we study the weakness of existing path validation and sanitization mechanisms when handling path containing symlinks, and how these flaws can be leveraged to achieve arbitrary file writes. Considering the diversity of strategies in path traversal defense and misuse patterns in symlink resolution, such vulnerabilities cannot be effectively identified using a single unified approach,e.g., static analysis or symbolic execution. Instead, we leverage specific inconsistent behaviors during path resolution and design five bypass techniques as well as three write methods to exploit path traversal vulnerability. We focus on the extraction scenario because it involves intensive file writes and frequent symlink validation, and has implementations across diverse programming languages. By examining 85 projects across ten programming languages, we find 16 arbitrary file write vulnerabilities, two of which have been confirmed high-severity CVEs. The methodology is also transferable to other domains requiring path validation.

Finally, we summarize the real world impact of vulnerabilities arising from inadequate path validation induced by misunderstandings of symlinks. Using four different application scenarios, we analyze how deficient wrappers and the misuse of symlink-related interfaces propagate implicitly through the software stack, ultimately resulting in high-impact path traversal vulnerabilities. Furthermore, adhering to ethical research standards, we responsibly disclosed all identified vulnerabilities to the affected vendors and project maintainers. We also propose a set of tailored mitigation strategies for different developers to reduce the associated security risks.

The contributions of this paper are summarized as follows:

- We present the first systematic analysis of misunderstandings regarding symlinks across layers of the software stack. We categorize four distinct aspects of confusion in file type identification and four different design strategies employed in path normalization, analyzing the security risks stemming from these inconsistencies.
- We summarize five classes of path traversal defense mechanisms. By exploiting the improper handling of symlinks and the misuse of associated interfaces, we design five symlink-based validation bypass schemes and three arbitrary-write methods. Through evaluation of 85 open-source projects, we uncover 16 arbitrary file write vulnerabilities.
- We demonstrate the severe consequences of symlink-induced validation flaws through four real-world application scenarios. We further provide remediation guidance tailored to different developers and discuss the trade-offs in robust path validation.

2 Background

2.1 Symlinks in Software Stack

A symlink (a.k.a. symbolic link or soft link) is a special file type that references another file or directory. It differs from a hard link, whereas a symlink's file content consists of a string encoding the pathname, a hard link is directly entry that refers to the same file object. Consequently, a hard link necessarily refers to an existing target, whereas a symlink may be a path to a non-existent location. Therefore, if the target is moved, renamed, or deleted, the symlink is not automatically updated or removed; instead, it becomes a dangling link whose stored pathname no longer resolves to a valid target. As a fundamental system-level feature, symlinks are typically utilized in conjunction with broader file system operations. Application developers often interact with symlinks in an indirect manner. Through our analysis of symlink related call sequences in real-world applications, we characterize how symlink effects are transmitted layer by layer within the software stack: operating system layer, programming language layer and downstream

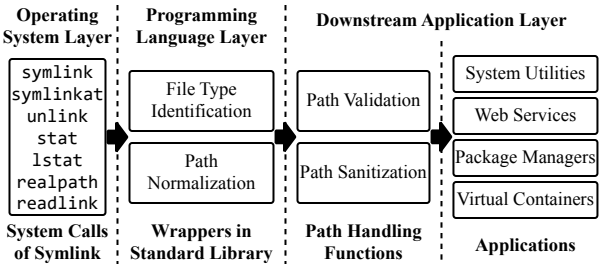


Figure 1: Symlink Resolving across Layers of Software Stack.

applications layer. This hierarchy is illustrated in Figure 1 and detailed in the following.

Operating System Layer. The operating system layer provides the fundamental system calls required for symlink management. These include operations for link creation (e.g., `symlink`, `symlinkat`), file type identification (e.g., `stat`, `lstat`), and path resolution (e.g., `realpath`, `readlink`). All symlink related behaviors in higher-level software stacks are implemented via these underlying system calls.

Programming Language Layer. Building upon system calls, the programming language layer provides higher-level abstractions within standard libraries to facilitate the handling of paths containing symlinks. These libraries primarily offer deeply encapsulated functionality for file type identification and path normalization. Specifically, file type identification interfaces are typically implemented by returning specialized file type objects or providing predicates to determine whether a path refers to a certain type. Path normalization interfaces focuses on path transformation, including the conversion between absolute and relative paths, the elimination of dotdot ("..") elements, and the resolution of symlinks. Notably, the specific implementation logic for these functionalities varies considerably across standard libraries.

Downstream Application Layer. This layer encompasses a wide array of downstream applications. To ensure the security of operations in file system, these applications typically implement custom path-checking and filtering mechanisms. Downstream applications then leverage either proprietary logic or third-party wrappers to facilitate a broad spectrum of functional scenarios.

Path Handling Functions. This refers to the essential validation and sanitization procedures performed during file operations, ensuring the safety of subsequent actions such as file writes and symlink creation. Path traversal defenses are typically implemented at the application level. Applications determine whether a candidate path remains within the intended restricted directory by invoking standard library functions or employing manual string processing techniques. Path handling is a prerequisite for applications across many scenarios.

Applications. The use of symlinks in downstream applications is often implicit, and most users are largely unaware of symlink manipulation. For instance, symlinks are fundamental to system configuration utilities and software distribution; web servers use them to enable or disable sites; package managers rely on them for version switching. More examples of the extensive application of symlinks are detailed in Table 1.

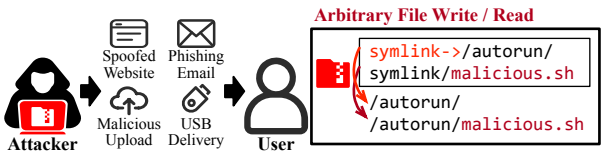


Figure 2: Threat Model of Improper Symlink Resolution.

Table 1: Symlink Usage in Multiple Scenarios.

Application Scenario	Examples
System Utilities	systemd, tar
Web Service Management	Nginx, Apache HTTPD
Package Management	Homebrew, apt
Virtual Container Management	Docker
Version Control	sdkman, nvm
Large-scale Data Support	DVC, ClearML

2.2 Threat Model

In the threat model considered in this paper, the adversary conducts the attack by constructing a chain of symlinks. In Unix-like systems, creating symlinks does not require root privileges, and symlinks are widely used by ordinary applications in everyday functionality. Consequently, an attacker can induce a victim to unknowingly create a chain of symlinks whose targets are controlled by the attacker. These link targets typically contain carefully crafted path patterns. Owing to the victim's misunderstanding of symlink resolution mechanisms and the incorrect use of path-handling interfaces, path validation is often inadequate. During subsequent file operations, such as file writes, these inadequately validated paths can finally point outside the intended directory, ultimately resulting in unauthorized writes to sensitive locations.

This scenario is frequently encountered yet often overlooked in the context of archive extraction. Since archive formats were originally designed to maintain directory structure consistency for software distribution, many extraction libraries provide native support for restoring symlinks. This is the primary scenario where symlinks are "passively" created from untrusted sources. Moreover, extraction workflows inherently involve extensive file-write operations, which can also be used by attacker directly. Attackers can employ social engineering, such as phishing emails or malicious websites, to deceive end users into downloading and extracting a crafted archive. For remote targets, the attacker typically

```c
static void
make_parent(char *path){
  sep = strrchr(path, '/');
  if(sep==NULL||sep==path)
    return;
  *sep = '\0';
  if(lstat(path, &sb)==0){
    if(S_ISDIR(sb.st_mode)){
      *sep = '/';
      return;  }
    unlink(path);  }
  make_parent(path);
  mkdir(path, 0755);
  *sep = '/';
}
```

```c
static void
extract(archive *a,
        archive_entry *e){
 /* ensure that parent
  * directory exists */
 make_parent(realpathname);
 if(S_ISDIR(filetype))
   extract_dir(a,e,
           realpathname);
 else extract_file(a,e,
           &realpathname);
}
```

(a) Code Snippets of Motivating Example

```
PoC-Archive
  ├─[symlink] link -> /
  └─[regular] link/tmp/poc.txt
```

(b) Proof-of-Concept Demonstration

Figure 3: Motivating Example in lib***ive.

focuses on web servers hosting applications with automated decompression features, for example, cloud storage platforms, community forums, and collaborative document editors. In this scenario, an attacker directly uploads the crafted archive to trigger the server's automatic extraction logic.

The security impact of such an attack depends on the destination of the unauthorized file write. By redirecting writes to sensitive directories such as `/etc/cron.d/`, where scripts are automatically executed by the system, the attacker can achieve Remote Code Execution (RCE) with the victim's privileges. Alternatively, by overwriting critical configuration files, the attacker can induce a Denial of Service (DoS) conditions. Below we use a real world example to illustrate the insecure usage of symlinks and the attack consequences.

2.3 Motivating Example

Figure 3 presents a real-world zero-day vulnerability to illustrate the security implications of improper symlink resolution and the misunderstanding of system interfaces. The vulnerability is found in lib***ive, a foundational library frequently utilized as the default extraction tool in many operating systems, and encapsulated as standard extraction library in some languages, which has received over 3.4k stars on GitHub. To address the symlink requirements in scenarios such as software distribution, the library allows the creation of symlinks pointing to arbitrary locations. And to mitigate the risk of arbitrary write vulnerabilities, the library incorporates checks during file writing.

The library utilizes the `make_parent()` function to perform sanity checks, ensuring that all directories in the final file path exist and preventing arbitrary writing via symlinks. This function analyzes the path components from the end to the beginning, using `lstat()` to determine each file type. If a symlink with the same name as a directory component is encountered, the function deletes the link and creates a directory instead, thus avoiding writing through symlinks.

However, this defensive mechanism is fundamentally flawed because `lstat()` exclusively evaluates the terminal component of the provided path string, remaining indifferent to the symlink status of intermediate components. Consequently, an attacker can construct a proof-of-concept (PoC), as shown in the Figure 3 (b).

The attacker can first direct the symlink to an arbitrary location (in this case, the root directory). Then, the attacker writes files into a subdirectory of this path (for example, writing to `/tmp`). Since the `make_parent()` function checks the path components from the end towards the beginning, the first component to be examined is the `link/tmp` path. Although `link` is a symlink, the `lstat()`function only assesses the type of the `tmp` directory component. As a result, the sanitizer function incorrectly identifies the entry as a valid directory, allowing the attacker to conduct an arbitrary write attack.

3 Problem Statement

To elucidate the misunderstandings of symlink across different software stack layers and the resulting security implications, this paper addresses the following research questions:

RQ1: What are the symlink-induced inconsistencies in path resolution interfaces across different programming languages?

Standard libraries in different programming languages encapsulate low-level system calls to provide high-level abstractions for path resolution. However, the implementation logic and functional semantics of these interfaces often diverge, which leads to inconsistencies in path handling and often confuses developers. To investigate this issue, we focus on two categories of functionality that critically depend on correct symlink handling: file type identification interfaces and path normalization interfaces. The implementation of these two functionalities is directly influenced by language designers' misunderstandings in symlink resolution.

Specially, first, we examine whether file type Identification interfaces accurately determine file types when processing paths containing symlinks, and we identify four types of confusion that users may encounter when using these interfaces. Second, we evaluate the correctness and robustness of path normalization interfaces when resolving paths that involve symlinks. We summarize four different strategies in path handling. In particular, we focus on the inconsistencies in path transformation arising from the diverse resolution strategies adopted by different standard libraries. The detailed discussion of this part is presented in section 4.

RQ2: How can the existing path traversal defense mechanisms bypassed through symlink manipulation?

Path traversal defenses are predominantly implemented at the application level during file-processing operations. Their

designs exhibit substantial variability, but showing weakness when handling symlinks. This inadequacy of path validation arises from the combined effect of developers' misunderstandings of symlink and the improper wrappers provided by programming languages.

In this phase of our research, we first provide a systematical taxonomy of current path protection strategies. We then design a set of validation bypass methods that leverage symlink resolution flexibilities. To evaluate the effectiveness of our proposed bypasses, we conduct large-scale experiments in archive extraction scenario. This scenario necessitates rigorous path filtering while naturally supporting diverse file operations, such as file writing and the creation of various file types including symlinks. A more detailed description of this part is provided in section 5.

RQ3: How do misunderstandings of symlinks and inadequate path validation lead to real-world impacts?

This research question examines the practical hazards arising from the improper handling of symlinks in file processing. We analyze how deficient wrappers and the misuse of symlink-related interfaces propagate implicitly through the software stack, ultimately resulting in critical security vulnerabilities.

By providing concrete case studies, we illustrate the inherent stealthiness and severity of these flaws. We responsibly disclose the identified vulnerabilities and actively engage with developers and communities. Finally, we propose a set of diverse mitigation strategies to reduce the associated security risks. Detailed analysis and discussion are provided in section 6 and section 7.

4 Path Resolution Inconsistency

In this section, we conduct a systematic behavior classification and comprehensive evaluation of two fundamental categories of path resolution interfaces: file type identification and path normalization. We collect these two categories of interfaces across 10 programming languages (C, C++, Java, Python, Go, PHP, JavaScript, Rust, C#, Perl). By analyzing and examining the interface behavior with diverse test cases, we reveal four aspects of file type identification confusion and four inconsistent strategies of path normalization .

4.1 File Type Identification Confusion

File type identification interfaces are primarily responsible for determining the type of file system object from an input path string, e.g. regular file, directory, and symlink. Because these primitives govern how paths are interpreted, their behavior directly affects symlink handling. To evaluate these functions, we collect and analyze 35 interfaces across 10 programming languages, as summarized in Table 2. These interfaces employ diverse methodologies for determining file types. Based on their functionality, we classify them into two categories.

Table 2: Path Identification Interfaces. # Follow Symlink indicates the number that resolve symlinks. # Has Follow Option represents the number that provide a configurable option for disabling symlink resolution.

Functionality	#	# Follow Symlink	# Has Follow Option
Return File Type	18	7	2
Is Directory	7	6	1
Is Regular	7	6	1
Is Symlink	3	0	0
Total	35	19	4

The first category consists of interfaces that directly return the file type, which are typically implemented as wrappers around system calls including `stat` or `lstat`. The second category includes interfaces that evaluate whether a given path corresponds to a specific file type, which is considered as a directory, a regular file, or a symlink.

We differentiate these interfaces based on their behavior when the terminal component of a path is a symlink, specifically, whether the interface identifies the symlink itself or the object it references (i.e., "link-following" and "non-following" behavior). In addition, we characterize whether an interface provides an explicit `NO_FOLLOWLINK` option to facilitate a more comprehensive analysis of consistency across different configurations.

File Type Identification Confusion refers to inconsistent resolution behaviors that arise when symlinks with different target types, or when symlinks situate at different positions within a path. We construct test cases by combining symlinks that point to different target files types with diverse path patterns. From this combinatorial space, we selected ten representative cases as test inputs to evaluate inconsistencies in file type identification behavior.

We categorize file type identification confusion into four distinct types: non-existing link confusion, intermediate link confusion, dot directory confusion, trailing slash confusion. Table 3 presents a high-level summary of these four confusion types across the 35 file type identification interfaces, including the number of affected interfaces for each confusion category, as well as representative examples from different programming languages. We now describe the details.

Aspect 1: Intermediate Link Confusion. This confusion arises when a symlink appears at an intermediate position in a

Table 3: Path Identification Confusions and Examples.

Identification Confusion	#	Examples
Aspect1: Intermediate Link Confusion	15	lstat[C]
Aspect2: Dot Directory Confusion	13	Files.isSymbolicLink[Java]
Aspect3: Trailing Slash Confusion	14	
Slash After Symlink[–>Dir]	12	fs::symlink_metadata[Rust]
Slash After Symlink[–>Regular]	2	Files.isRegularFile[Java]
Aspect4: Dangling Link Confusion	11	filesystem::status[C++]

path rather than as the final component. Among all interfaces capable of identifying symlinks, none continue to classify such path as a symlink, even when a dedicated symlink identification function is used. Instead, they return file type of the concrete object ultimately reached through the symlink. If a developer is unaware of these semantics, they may implement insufficient path filtering mechanisms. Indeed, the inadequate protection discussed in our motivating example is a consequence of this specific confusion.

Aspect 2: Dot Directory Confusion. This category examines whether appending a current directory pointer ("`/.`") to a symlink that targets a directory induces inconsistent behavior. The observed behavior is closely consistent with Aspect 1. Notably, however, in C# the corresponding interface can still identify the path as a symlink. A plausible explanation is that the implementation normalizes the path by removing the trailing "`/.`" segment prior to checking file type.

Aspect 3: Trailing Slash Confusion. This confusion refers to inconsistent behaviors observed when a trailing slash ("`/`") is appended to a symlink. We evaluated two scenarios: links targeting directories and links targeting regular files. When the symlink points to a directory, appending a trailing slash yields behavior that is largely consistent with Aspect 2. Except for a small number of interfaces that can still identify the symlink, most interfaces instead report the type of the resolved directory. When the symlink points to a regular file, the expected behavior is either to identify the link itself or to return an error. However, notably, two Java interfaces continue to classify the path as a regular file.

Aspect 4: Dangling Link Confusion. This confusion arises when resolving a symlink whose target does not exist (i.e., a dangling symlink). Our results show divergence in how interfaces interpret such links. Most interfaces attempt to resolve the target and immediately raise an error indicating that the file or directory does not exist, whereas a subset of interfaces correctly classify the path as a symlink. Such errors can mislead developers into conflating the existence of the symlink with the existence of its target.

4.2 Inconsistency in Path Normalization

Path normalization refers to the family of functions provided by standard libraries to transform resolve divergent logical paths into a standardized format. Such transformation is a necessary prerequisite for reliable path validation as file systems permit a single physical location to be represented by multiple logical paths through the use of special patterns and symlinks. However, normalization strategies are not implemented consistently across interfaces. We consider three categories of components that may arise during path resolution: basic elements, file types, and errors. Based on the official Linux documentation [1, 3] on path resolution [2, 14], we select a set of representative patterns for each category. Detailed descriptions of these patterns are provided in Table 4. We examined 34 path transformation interfaces from 10 programming languages with the combination of these path patterns.

4.2.1 Path Normalization Strategies

We conducted an analysis using a combination of manual inspection and test cases by composing fundamental path elements in 34 interfaces, notably, eight of which perform only basic conversions between relative and absolute paths. We identified several distinct path normalization strategies and summarized in Table 5. Details are as follows:

Strategy 1: Direct Invocation to OS Interface. APIs in this category normalize paths by directly invoking the operating system's `realpath` syscall. This approach follows a left-to-right sequential resolution. It fails immediately when encountering exceptional conditions such as a nonexistent intermediate component. Examples include Java's `Path.toRealPath` and Rust's `fs::canonicalize`.

Strategy 2: Lexical Normalization First, then Syscall for the Remainder. Instead of following the OS's left-to-right resolution order, these implementations first resolves lexical patterns (e.g., "dotdot" components) internally and then invoke `realpath` to resolve the remaining path that may contain symlinks. This deviation in resolution order can change both behavior and results. Specifically, if a symlink is immediately followed by a "`..`" component, the link will be lexically purged without ever being resolved. JavaScript `fs.realpathSync` follows this strategy.

Strategy 3: Lexical Normalization plus Manually Symlink Resolution. Interfaces in this category manually handle both path patterns and symlinks in user space, typically by iteratively invoking `readlink` syscall. A salient difference from operating system `realpath` is that they may successfully resolve nested symlinks deeper than the system limit (commonly 40) whereas a standard syscall would return a "Too many levels of symlinks" error. Python's `os.path.realpath` exemplifies this manual strategy.

Strategy 4: Lexical-only Processing without Symlink Resolution. Interfaces in this group operate purely on the path string, simplifying syntactic patterns while ignoring symlinks entirely. While efficient, these interfaces are unsuitable for path validation where symlinks may point outsides. However, inconsistent naming conventions across languages confuse the user and may lead to misuse. For example, Go's `path.Clean` and Java's `Path.normalize` just handle the special path patterns without considering symlinks.

In addition to the divergency mentioned above, the implementations exhibit inconsistencies in their handling of exceptions. For example, upon encountering resolution errors, some APIs propagate failures by throwing exceptions, whereas others catch such errors for robustness and return a partially resolved path. These divergent error-handling policies may lead to great different resolution results.

Table 4: Minimal Path Component Elements in Path Resolution. We generate diverse path test cases by combining path patterns across different categories and by repeatedly reusing the same pattern.

Category	Pattern	Tag	Description

Basic Element	dot
	dotdot
	slash
		`.`
		`..`
		`/`
			Three fundamental path elements are examined, including the current directory indicator (dot), the parent directory indicator (dotdot), and the path component separator (slash). Notably, a single slash denotes the root directory.

File Type	regular file
	directory
	symlink
		`[regular]`
		`[dir]`
		`[symlink]`
			Three file types are considered: regular files, directories, and symlinks. Hard links are not included, as they exhibit behavior nearly identical to that of regular files. These file types may appear as intermediate or terminal components in a path, or as the target of a symlink.

Error	non-existent location
	max nest depth
	loop symlink
	long filename
	long pathname
		`ENOENT`
		`SYMLOOP_MAX`
		`ELOOP`
		`NAME_MAX`
		`PATH_MAX`
			Five error conditions are tested. Non-existent locations are common to be the symlink target, given the dangling links. Max nest depth refers to the system's limit on the number of symlinks that can chain to one another. If a symlink resolves into a cycle, this issue will also arise. Long filename refers to a file name that exceeds the system's length limit. Long pathname indicates that the entire path being resolved exceeds the system's limits.

Table 5: Path Normalization Interfaces.

Strategy	#	Examples
Strategy 1	11	Path.toRealPath[Java] Path::canonicalize[Rust]
Strategy 2	2	fs.realpathSync[JavaScript] fs::weakly_canonical[C++]
Strategy 3	3	os.path.realpath[Python] filepath.EvalSymlinks[Go]
Strategy 4	10	Path.normalize[Java] os.path.normpath[Python]
Absolute–Relative Path Transformation	8	fs::absolute[C++] File::Spec::canonpath[Perl]
Total	34	

4.2.2 Problems Arising from Inconsistent Normalization

These divergent implementation strategies can cause both the path resolution and validation diverging from the file system's results. We illustrate inconsistencies with two cases. Table 6 shows the resolution results with different strategies.

Case 1: Resolution-order Mismatch Invalidates Path Checks. Consider a symlink `symlink` that points to the current directory (`"."`). An application attempts to access `symlink/..` and then write a file under that path. Under Strategy 2 that simplifies `".."` lexically before resolving symlinks, the `".."` component eliminates the preceding `symlink` component, and the path is interpreted as the current directory `"."`. However, at the system level, the kernel resolves path components sequentially from left to right. It first follows `symlink` to obtain `"./"`, then applies `".."` to reach the parent directory `".."`. Consequently, this discrepancy allows the file operation to escape the intended directory, bypassing the application's path-validation logic.

Table 6: Normalization Results with Different Strategies.

Case	Tag	Normalization Strategy			
		S1	S2	S3	S4
Case 1	`symlink[->.]/..`	`..`	`.`	`.`	`.`
Case 2	`symlinkN[ ->symlinkN-1[->..... ->dir]]`	Err	Err	`dir`	`symlinkN`

Case 2: Manual Symlink Resolution Induces Unexpected Exceptions. Consider a deeply nested symlink chain in which `symlinkN` points to `symlinkN-1`, `symlinkN-1` points to `symlinkN-2`, and so on, for N>40 (the maximum number of nested symlinks since Linux 4.2 [2]). Because each symlink has a well-defined link target, for strategies that implement manual resolution via `readlink`, the application may successfully traverse an arbitrary number of links and treat this path as resolvable. Nevertheless, the operating system typically enforces a maximum symlink traversal depth and will reject such paths with an error (Too many levels of symlinks) at the time of file operation. This mismatch results in unexpected runtime exceptions.

5 Path Validation Inadequacy

In this section, we investigate the prevalence of vulnerabilities arising from developer misunderstanding and the subsequent misuse of symlink interfaces. We focus our analysis on archive extraction, a quintessential scenario where symlink manipulation can be triggered by untrusted external inputs. Specifically, we analyze the inadequacy in existing checks and validations of paths containing symlinks, and we show how incomplete filtering can be exploited to achieve arbitrary file write attack.

5.1 Methodology

We provide a high-level overview of our systematic methodology for analyzing and detecting path traversal vulnerabilities arising from insufficient path validation. Our approach is structured into the following three steps:

- **Step 1: Summarizing Path Traversal Defense Mech-**

anisms. Through manual analysis of archive extraction tools and libraries implemented in different programming languages, we summarize the five main categories of path traversal defense mechanisms.

- **Step 2: Constructing Bypass Strategies for Arbitrary Symlink Targets**. Leveraging the identified design inconsistencies in symlink resolution and the defense mechanism flaws, we develop four targeted bypass strategies. These strategies enable an extracted archive to create symlinks that ultimately resolve to attacker-chosen, arbitrary file system locations.
- **Step 3: Exploiting Arbitrary File Writes**. Recognizing that many defensive measures are primarily enforced during the file-writing stage, we develop specialized exploitation mechanisms for write operations. By coupling these exploitation methods with the arbitrary symlink targets achieved in Step 2, we finally exploit arbitrary file write vulnerabilities.

5.2 Understanding Path Traversal Defense

We analyze 10 projects, covering low-level system libraries, language standard libraries, and third-party libraries. We summarize the path sanitization schemes, showing in Table 7.

Table 7: Path Traversal Defense in Different Projects.

ID	Language	Project	Defense Mechanism				
			D1	D2	D3	D4	D5
1	C	BusyBox Tar	✔			✔	
2	C	GNU Tar	✔		✔	✔	
3	C	Info-ZIP UnZip	✔			✔	
4	C/C++	7-Zip	✔				✔
5	C/C++	libarchive	✔		✔		
6	Rust	tar-rs	✔	✔	✔		
7	Rust	zip2	✔	✔	✔		
8	Python	zipfile	✔				✔
9	Python	tarfile	✔	✔			
10	Python	py7zr	✔	✔			

Defense 1: String-based Path Processing. This approach partitions path strings and emulates the interpretation of special characters within the path. Typically, these methods strip leading slashes and employ a path stack to resolve and eliminate "`..`" sequences, effectively mimicking operating system behavior. If the processed path does not contain the target directory as a prefix, the system either triggers an error or forcibly appends the string to the base path. A primary advantage of string-based processing is that it avoids file system system calls, resulting in lower overhead and higher performance.

Defense 2: Resolving Path with System Calls. This defense mechanism relies on operating system calls to normalize file paths thereby eliminating "`..`" components and resolving symlinks. Implementation details vary across programming languages and libraries: some rely directly on native system calls for full resolution, whereas others employ a hybrid approach, invoking the system interface specifically for symlink resolution while manually handling "`/`", "`.`" and "`..`". After normalization, the resolved path is validated to ensure that it remains confined to the permitted directory subtree. Some language runtimes employ caching mechanisms to reuse prior resolution results and reduce performance costs.

Defense 3: Restricting File Creation through Symlinks. In some extraction utilities that permit symlinks to be created, a common mitigation is to prohibit writing file through existing symlinks [18, 28]. The extractor enforces that the intended write destination is a directory. In practice, if the target is identified as a symlink, the application typically deletes the existing entry and initializes a new directory in its place. Notably, however, the criterion used to determine whether an entry represents a directory differs across implementations. Some applications rely on string-based rules, such as detecting a trailing slash in the path, while others utilize system calls to query the actual file mode.

Defense 4: Enforcing the Creation Order for Different Entry Types. During archive extraction in some basic libraries and tools, in order to support symlinks pointing to any needed location, a defense mechanism is to impose a strict creation order over entry types. All symlink entries are first collected into a list. After completing the extraction of all regular files, the symlinks are then created. This approach provides a temporal safeguard to prevent path traversal via symlink.

Defense 5: Abandoning Symlink Creation. Given the complexity and security risks associated with symlinks, some extraction libraries decline to process symlink. Instead, they either emit a regular text file that records the target path or simply ignore symlink entries. Furthermore, some specialized libraries do not provide high-level, automated extraction interfaces. These libraries shift the responsibility for path validation and security enforcement to downstream users.

It is important to note that the above mitigation strategies are frequently integrated to enhance robustness. Although these defenses make substantial efforts to mitigate path traversal while preserving the functional utility of symlinks, they still cannot fully eliminate the risk of arbitrary writes.

5.3 Symlink-Based Path Validation Bypass

We design five distinct bypass strategies that exploit the inherent flexibility of symlinks and the discrepancies identified earlier between language-level path normalization interfaces and OS-level system-call resolution. Building on existing defense strategies, we design following techniques for bypassing path checking mechanisms and Figure 4 shows the payload in archive of each bypass strategy.

Bypass 1: Basic Symlink Exploitation. This approach

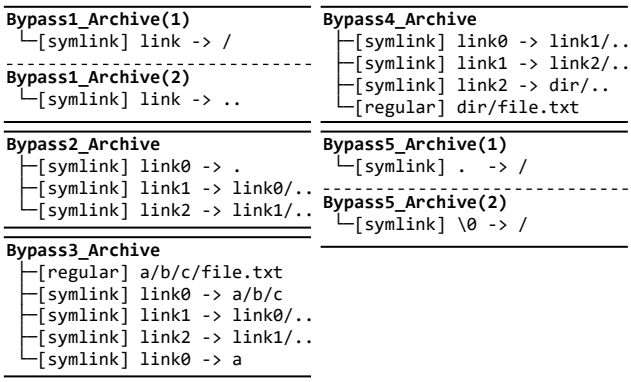


Figure 4: Bypass Payload in Archives.

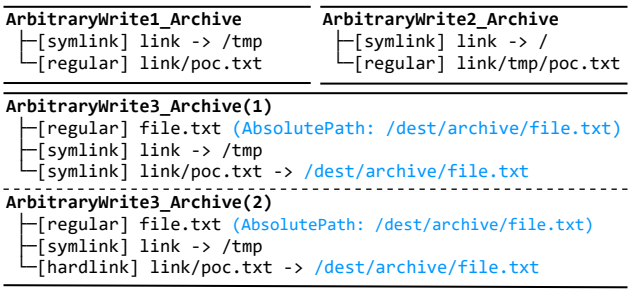


Figure 5: Arbitrary Write Payload in Archives.

represents the most direct exploitation vector, yet it remains useful in some libraries that fail to account for symlink. An attacker can directly set the symlink target to the root directory ("/") or use repeated ".." components to backtrack to the root directory, thereby escaping the intended extraction scope.

Bypass 2: Exploiting Inconsistent Path Resolution. This strategy builds upon the observations detailed in section 4, Case 1. As illustrated in Figure 4, we construct a chain of symlinks where each link points to the parent directory of its predecessor link. Because the path normalization functions performs lexical processing first, the ".." component eliminates the preceding symlink, which erroneously suggests the path remains within the current working directory. However, during the actual file-writing phase, the operating system's resolution mechanism follows the symlinks sequentially. The final resolved path is therefore equivalent to a traversal sequence composed of many ".." segments, enabling the attacker to bypass the path check.

Bypass 3: Cascading Redirection via Symlink Updates. This strategy leverages nested symlink dereferencing and the cascading effects induced by modifying a base link. As illustrated Figure 4, the attacker first creates a base link, `link0`, pointing to a deep subdirectory. We then iteratively create additional symlinks, each referencing the parent directory of the previous symlink. Unlike Bypass 2, even when the extractor relies on OS-level resolution, each individual symlink appears legitimate and thus passes validation. Finally, we recreate `link0` to point to the current directory. While link0 remains nominally "safe", its mutation causes the entire dependent chain to resolve outside. Notably, this strategy is not a universal substitute for Bypass 1, as some extraction libraries do not overwrite existing entries with the same name.

Bypass 4: Validation Evasion via Dangling Symlinks. This strategy exploits the fact that symlinks may be dangling, creating a temporal gap between path validation and the restoration of a link's validity. The attacker first creates `link0` pointing to a non-existent path `link1/..` . During path validation, resolution aborts because link1 does not exist, and the implementation may therefore classify the link as benign. Subsequently, the attacker creates link1 pointing to `link2/..`, and so on, iteratively building the resolution chain. Finally, by pointing `linkN` to a existent location, the attacker reconstructs a traversal chain similar to Bypass 2. This approach, however, does not replace Bypass 2 or Bypass 3 in general, as its success depends on whether extraction tools restore dangling symlinks.

Bypass 5: Manipulating the Extraction Destination Directory. This strategy is designed to circumvent security checks enforced at the time of file writing. While many extraction utilities permit the creation of symlinks, they often strictly validate the final destination of write operations by normalizing both the target path and the designated extraction destination. However, a vulnerability exists when the utility fails to protect the extraction root itself. By assigning a symlink a name as "." (dot) or even an empty string "\0", the attacker can use the path concatenation logic to set the symlink name as the extraction destination directory. When this directory is successfully replaced by a symlink pointing to the system root, all subsequent extraction operations are therefore deemed "legal". This bypass applies only in settings where the extractor does not create the destination directory in advance.

Through these strategies, the primary objective is to establish a symlink pointing to the root directory, enabling arbitrary reads. However, such a link alone is insufficient for exploitation, as many libraries may permit arbitrary redirection while still restricting subsequent write operations. The next step, therefore, is to leverage these symlinks to trigger arbitrary file writes.

5.4 Arbitrary Write via Symlink

We exploit API misuse to trigger arbitrary write in this subsection. Building upon the arbitrary redirection achieved in the previous phase, these strategies achieve arbitrary file write capability. Figure 5 shows the payload in archive of each arbitrary write method.

Write Method 1: Direct File Writing. The simplest approach is to write a file through a symlink that points to the

target location. If the extractor performs no validation of write destination, this immediately yields an arbitrary file write.

Write Method 2: Sub-directory Writing via Symlink. This strategy leverages the misuse of symlink identification interfaces, which is also the technique used in our Motivating Example. Instead of writing directly to the symlink, the attacker targets a file within a sub-directory under the link. Because the validation logic often only inspects the ultimate file object and not its parent components, the interface erroneously identifies the operation as "safe," thereby bypassing the security filter. In practice, this method can effectively subsume Method 1.

Write Method 3: Creating Links via Symlink. This approach is designed to bypass constraints hat enforce a strict extraction order, such as extracting regular files before symlinks. While such utilities may allow arbitrary symlink creation after files are extracted, they often overlook the risk of creating secondary symlinks or hardlinks (hardlink creation is supported by formats such as TAR) through an existing symlink. If the extraction path is known or predictable, creating arbitrary links that reference the extracted files is functionally equivalent to writing files to arbitrary locations.

Table 8 illustrates the relationship between the file type identification confusion and path normalization inconsistency discussed in section 4, and the bypass strategies presented in section 5. A bypass strategy may exploit one or more inconsistencies in interface design, while simultaneously leveraging developer misuse and misunderstanding of these symlink-related interfaces.

Table 8: Relationship between path resolution inconsistencies and bypass strategies. ✔ indicates that the corresponding bypass strategy leverages developers misuse and misunderstanding of the associated inconsistency.

		Identification Confusion				**Path Normalization**			
		A1	A2	A3	A4	S1	S2	S3	S4
Bypass Method	B1								✔
	B2					✔	✔	✔	
	B3						✔	✔	
	B4				✔				
	B5		✔						
Write Method	W1								✔
	W2	✔		✔					
	W3								✔

5.5 Detection Results

We detail our experimental dataset, the implementation of our testing framework, and the results of our vulnerability discovery process below.

Dataset. We constructed a dataset of archive extraction projects from ten programming languages. The dataset includes both language standard libraries and third-party open-source extraction libraries collected from GitHub and major package ecosystems (e.g., Maven and npm). During collection, we excluded libraries that provide only compression/decompression algorithms without performing file system extraction. Among the 85 collected projects, we further removed eight libraries that do not expose a unified extraction interface and instead require users to explicitly define the handling of each entry type. The final filtered dataset is summarized in Table 9.

Following the officially recommended usage documented by each project, we either compiled the library directly or implemented lightweight wrappers to normalize the invocation interface, ultimately enabling command-line execution of each library.

Table 9: Dataset to Detecting Path Traversal.

Language	**Compiler/ Interpreter**	**Collection Source**	**#**	**# Has Wrapper**
C	GCC 13.3.0	GitHub	16	14
C++	G++ 13.3.0			
Java	OpenJDK 25.0.1	Maven	7	6
Python	Python 3.14.2	PyPI	5	5
Go	Go 1.22.2	Go Modules	11	9
JavaScript	Node.js 25.3.0	npm	27	25
PHP	PHP 8.3.6	GitHub	6	6
Rust	cargo 1.92.0	GitHub	3	3
C#	.NET 10.0.100	GitHub	3	2
Perl	Perl 5.38.2	CPAN	7	7
Total			85	77

Implementation. We extensively customized Python-based tooling for ZIP, TAR, and 7z archives to support adding archive entries of arbitrary types and contents. We then integrated our proposed bypass and writing strategies into an automated exploit workflow. All experiments were conducted within a Docker container running Ubuntu 24.04.

The workflow consists of three stages: First, an initial set of test “seed” archives is generated to evaluate how extraction tools restore symlinks. These archives contain symlinks targeting various locations, including relative paths, absolute paths, and paths containing parent-directory references (i.e., "..."). By observing whether a tool restores symlinks and how it processes their target paths, we can identify extraction tools that may exhibit potentially vulnerable behavior.

Second, archives are constructed using bypass methods with the goal of establishing symlink chains that ultimately reference locations outside the intended extraction directory. Based on the results of the first stage (e.g. support for parent-directory references or dangling symlinks), we select appropriate bypass methods. We then build archives containing a chain of symlinks to conduct further evaluation.

Finally, write methods are employed to evaluate arbitrary

Table 10: The vulnerabilities we find in the total 85 projects. The first 16 leads to arbitrary write, and the last two leads to arbitrary read. In fact, all arbitrary write inherently imply arbitrary read capability. ✔ means it is vulnerable to our bypass method.

ID	Language	Library or Software	Version	Vulnerable Format	Bypass Method					Write Method			Status
					B1	B2	B3	B4	B5	W1	W2	W3	
1	Python	py***zr	1.1.0	7z		✔		✔		✔	✔	✔	CVE-2026-***79
2	JavaScript	com***sing	2.0.0	tar	✔					✔	✔		CVE-2026-***84
3	JavaScript	tarb***ract	0.0.6	tar		✔	✔	✔		✔	✔	✔	Confirmed
4	JavaScript	zi***ib	1.1.2	zip					✔	✔			Assigned
5	C	bu***x/tar	1.37.0	tar	✔	✔	✔	✔				✔	Confirmed
6	C	k***/zip	0.3.5	zip	✔	✔		✔		✔	✔	✔	Assigned
7	C	sc***s/star	1.7.0	tar				✔		✔	✔		Confirmed
8	C/C++	lib***ive	3.9.0	zip	✔	✔	✔	✔			✔		Confirmed
9	C/C++	mi***ng	4.0.10	zip	✔	✔	✔	✔	✔	✔	✔	✔	Assigned
10	Go	it***vior	60...d75b	zip, tar	✔	✔	✔	✔	✔	✔	✔	✔	Confirmed
11	Go	m***iver	4.0.0	tar	✔	✔		✔	✔	✔	✔	✔	Confirmed
12	Go	cont***hive	1.59.1	tar	✔	✔	✔	✔	✔	✔	✔	✔	Confirmed
13	Go	m***hive	0.2.0	tar	✔	✔	✔	✔	✔	✔	✔	✔	Assigned
14	Go	go.p***age	1.61.0	tar	✔	✔	✔	✔	✔	✔	✔	✔	Confirmed
15	Perl	Arch***ract	0.88	zip	✔	✔		✔	✔	✔	✔	✔	Confirmed
16	PHP	p***_tar	1.6.0	tar		✔	✔	✔		✔	✔		Confirmed
17	JavaScript	t***s	3.1.1	tar		✔	✔	✔		—	—	—	Confirmed
18	JavaScript	t***z	1.0.1	tar		✔	✔	✔		—	—	—	Confirmed

file write capabilities. By combining different write techniques with various bypass strategies, we assess whether arbitrary file writes can be achieved.

Vulnerabilities. We identified arbitrary file Write vulnerabilities by verifying the presence of unauthorized files at target destinations. For arbitrary file read, we adopt a conservative criterion to account for the fact that some libraries do not consider symlinks that point to arbitrary locations inherently dangerous. So we only consider libraries that prevent links pointing to the root directory ("/") or deep directory traversal ("..." sequences) but fail to sanitize our bypass techniques. In total, we discovered 16 arbitrary write vulnerabilities and two additional arbitrary read vulnerabilities, as detailed in Table 10. For ethical and anonymous consideration, the names of the affected projects have been anonymized.

5.6 Comparison with Previous Work

To the best of our knowledge, the work most closely related to ours is ZIPDIFF [37]. ZIPDIFF analyzes inconsistencies in how different decompression parsers handle the same zip archive, focusing on discrepancies in file parsing and metadata handling. It employs randomized mutation of zip-file to check whether two parsers exhibit divergent behaviors.

We reproduced the open-source implementation of ZIPDIFF and evaluated it on 20 zip tools from our dataset, eight of which contain symlink vulnerabilities. ZIPDIFF fails to detect any of them.

Exploiting symlink-based path traversal requires carefully crafted symlink targets and creation order, with consideration of language-level symlink interfaces. ZIPDIFF employs simple path mutation strategies and cannot generate ZIP inputs with such complex symlink compositions.

Our work also differs from the standard Zip Slip vulnerability. Traditional Zip Slip attacks achieve path traversal by directly referencing the root directory or using parent-directory components (".."), thereby escaping the controlled directory. Our work focuses on path validation flaws caused by language-level interfaces. Different programming languages may resolve the same symlink-containing path to different physical locations. As a result, even identical validation logic can be secure in one language yet vulnerable in another.

6 Case Study

6.1 Operating System Utility

Vulnerabilities within lib***ive directly propagate to widely-used operating systems. F***SD, a widely adopted open-source Unix-like operating system, is extensively utilized in many infrastructures. As a core component of F***SD, lib***ive is integrated as the base system utility to power the default archive management.

In real-world scenarios, attackers can distribute malicious ZIP archives via phishing email attachments or spoofed websites. When a user extracts these archives containing malicious symlinks, the lib***ive resolves these links silently, enabling attackers to execute arbitrary file writes on the target

file system, finally causing remote code execution. This operation is superficially indistinguishable from the extraction of a benign archive, leaving the user entirely unaware of the malicious symlinks creation.

6.2 IoT Device Toolkit

Bu***ar is a compact and portable UNIX utility for extraction, serving as the default toolkit for resource-constrained environments such as IoT firmware, mobile device, and lightweight container images. This utility is frequently applied to data-processing workflows on IoT devices, involving processing externally supplied archive files as part of normal operation.

When processing archives containing crafted symlink layouts, Bu***ar may write files to unintended locations. This behavior can lead to remote code execution and denial of service. Given Bu***ar's prevalence across IoT and mobile environments, this vulnerability class presents a wide deployment surface.

6.3 Library in Other Applications

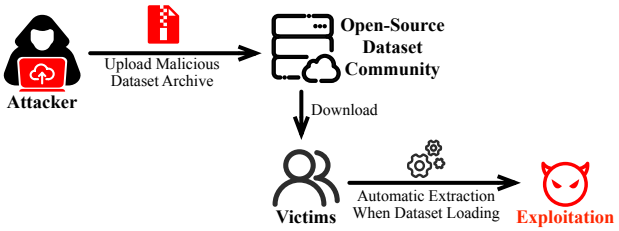


Figure 6: Real World Impact in Machine Learning Scenario.

Vulnerabilities in py***zr can affect across the software supply chain into diverse application scenarios. For instance, the Hug***sets library, a foundational component in the machine learning ecosystem with over 21.2k GitHub stars, facilitates the access and loading of open-source datasets. When a user invokes `load_dataset()`, the library automatically handles downloading, caching, and extraction in the background. The datasets library calls various extraction libraries to perform these operations, remaining entirely transparent to the user.

Figure 6 illustrates the attack procedure. Attackers can exploit this supply-chain vulnerability by constructing dataset archive containing malicious symlinks. By uploading such dataset to open-source dataset community and inducing victims to load them, the attacker triggers the automatic extraction process. This enables arbitrary file writes on the victim's system, subsequently allowing the attacker to place backdoors in victim's server.

6.4 Web Service

Wor***ess is a widely deployed web application platform that is commonly extended through plugins to provide custom network-facing services. In operational practice, many plugins support archive upload and server-side extraction service in workflows. Plugin developers often rely on packaged archive-handling libraries rather than implementing custom extraction logic.

In this setting, improper handling of symlinks during extraction can directly translate into a web attack surface. We identify a representative Wor***ess plugin Mu***ion (335 stars in GitHub) that integrates an archive library with unsafe symlink handling. By uploading a crafted archive, an attacker can induce the server to write files to arbitrary locations outside the limited directory, reaching full control of the deployment server. Critically, this risk is difficult for web application developers to recognize, since symlink resolution occurs implicitly inside the extraction library.

7 Lessons Learned

7.1 Mitigation and Suggestion

We propose several mitigation strategies informed by our analysis of robust library implementations and historical vulnerability patches. These recommendations aim to enhance existing defense strategies across multiple dimensions. Furthermore, we discuss developers' perspectives on the issues of symlinks as well as the associated design trade-offs.

For Flawed Path Validation. Effective path validation and filtering are critical for securing file system operations. Based on our evaluation of successful defensive patterns and the patching strategies adopted in historical vulnerability fixes, we summarize the following guidelines to strengthen current validation frameworks:

Always perform validation at write time. Given the inherent flexibility of symlinks and the potential for path ambiguity, path validation should be enforced whenever a write operation is performed, including regular file, a symlink, or another file type. In addition, developers should better use syscall-based path resolution for each write operation to ensure symlinks do not escape restricted directories. This keeps consistency between the resolved path and the actual write target, enabling robust verification.

Retain and use the sanitized path results. Our vulnerability analysis reveals that many flaws arise when a path is sanitized to be safe after normalization, but subsequent file operations continue to use the original, unsanitized path. Retaining and directly operating on the sanitized path can effectively prevent risks caused by later symlink target changes.

Carefully select the symlink resolution interfaces and options. Developers should carefully select programming language interfaces that correctly handle symlinks and explicitly

configure their behavior. When performing sensitive path manipulations or file operations, options such as `O_NOFOLLOW` should be enabled to prevent automatic symlink resolution.

Ensure the existence of the write path in advance. The write path should be verified to exist prior to file operations, with each path component validated individually. This strategy avoids unintended operations mediated through symlinks.

For Extraction Functionality. In addition to adopting the stricter path protection strategies discussed above, we recommend explicitly exposing the parameter of whether symlinks are restored during extraction. Such interface designs allows downstream users to selectively enable or disable symlink extraction based on their requirements, avoiding unnecessary restoration of symlinks. For terminal-based decompression utilities, we recommend implementing notifications when a symlink is being established. Such design enables users to be aware of filesystem changes, mitigating potential security implications caused by unintended symlinks.

For Programming Language. To minimize developer confusion and misuse, programming language developers had better adopt path resolution strategies that are consistent with those of the operating system. In addition, we suggest the documentation should more clearly specify how symlinks are handled and highlight the potential security risks. It is also a good choice to introduce specialized interfaces specifically designed for handling paths that involve symlinks.

7.2 Trade-off in Path Validation

Based on our discussions with developers and an analysis of responses in publicly disclosed historical vulnerabilities, we observe that enabling robust but efficient path validation remains a challenging problem. The necessity of invoking system calls for accurate symlink resolution introduces significant performance overhead, particularly in scenarios involving the extraction of numerous small files. Below, we summarize the trade-offs adopted by developers to balance security with performance efficiency.

Some library developers have chosen to remove unified archive extraction abstractions altogether, exposing only low-level file extraction interfaces. In this design, downstream application developers need to manually implement custom file-handling and path-filtering logic. As a matter of fact, this approach transfers security responsibility from the extraction library to the application layer.

Developers also frequently cite performance requirements as a primary constraint. Adequate symlink resolution relies on system calls, which is expensive. That is why many designs initially rely on lexical-level path manipulation rather than system-level checks. This latency is exacerbated in archives containing a high volume of small files or deep directory structures. Many developers mentioned alternative mechanisms are required to achieve effective but robust path protection.

8 Discussion

8.1 Other Symlink Usage Scenarios

Our study focuses on the restoration of symlinks within three archival formats (ZIP, TAR, and 7z). We find other archive formats also support the restoration of symlinks. For example, the RAR format also provides such mechanisms. However, due to its proprietary nature and licensing restrictions, we are unable to reverse-engineer the format or generate customized archives for systematic testing. Other formats, such as CPIO, are infrequently used and lack widely adopted supporting libraries.

We also manually examined other scenarios where symlinks are used. By analyzing the usage of symlink creation APIs across open-source projects in GitHub, we observe that most symlinks are governed by predefined logic with deterministic targets and unaffected by external input. Besides, the threats in those contexts are usually race conditions and link following vulnerabilities, which is less related to the threat model considered in this paper.

8.2 Other Operating Systems

Symlink mechanism also exists on Windows. Creating symlinks on Windows typically requires administrative privileges. Since most extraction tools run under standard user privileges, the risk of symlink-based path traversal is inherently mitigated for the majority of end-users. Nevertheless, in server-side environments, where administrative service accounts may use vulnerable libraries to extract archives, such attacks may still be possible. Given the differences between Windows and Unix-like permission and security models, a comprehensive analysis of symlink–related path vulnerabilities on Windows is left to future work.

9 Related Work

Symlink Related Vulnerabilities. Prior research has extensively investigated vulnerabilities arising from improper handling of symlinks, with a focus on the exploitation and detection of race conditions and link following attacks. In the area of race conditions, these studies typically exploit the TOCTOU window (time-of-check-to time-of-use) by rapidly replacing symlinks to bypass security checks, thereby achieving privilege escalation or file tampering [7, 12, 28]. Bishop et al. [6] first highlighted race condition issues in UNIX file operations. Subsequent efforts by Wei et al. [34] identified numerous potentially vulnerable TOCTOU system call pairs, while Tsafrir et al. [32] extended this analysis to real-world setuid programs containing symlink race vulnerabilities. Raducu et al. [27] provided a comprehensive survey of file-related TOCTOU attacks and defenses, covering variants involving symlinks, hard links, and other types.

Research on link following focused on the use of symlinks to redirect file operations, resulting in file hijacking or unintended access to sensitive files [38]. Xiang et al. [36] introduced automated methods to detect and exploit link follow vulnerability in Windows file operations.

Although these studies have effectively unveiled exploitation techniques and detection methods for symlinks and link following within specific operating system environments, they focus on runtime races or targeted exploitation, often overlooking flaws in the symlink resolution. Furthermore, rather than requiring pre-existing access as a low-privilege user to escalate permissions, our approach allows for the direct inheritance of the victim's elevated privileges. This makes the attack both more stealthy and more severe.

Path Traversal. Many studies has focused on the detection and mitigation of Path Traversal attacks [4, 20, 21, 39]. For example, Akhoundali et al. [4] extended the detection methodologies to support large-scale codebases, while Li et al. [21] implemented Path Traversal detection for Node.js using Object Dependency Graph (ODG) analysis. Recent studies also investigated the impact of path traversal in various scenarios, such as bypass authentication in URL handling [38]. In the domain of extraction, the path traversal vulnerability often identified as ZIP Slip [29]. In contrast to these works, our research focuses on path traversal attacks specifically induced by symlink resolution. This vulnerability stems from a misunderstandings of symlink resolution across the entire software stack, triggering stealthy and severe arbitrary write attacks.

Other Attacks via Archive. Prior research on archive formats explored parsing ambiguities, which have long been exploited for antivirus evasion and detection bypass. Early demonstrations used parser ambiguities in ZIP and related formats to create files interpreted differently by security tools and target applications [5, 15, 24, 33]. More recent work [16, 17] has examined polyglot archives for evasion purposes.

The most closely related work is ZIPDIFF [37], which offers the first systematic study of ZIP parsing ambiguities in various aspects. ZIPDIFF analyzes inconsistencies in how different decompression parsers handle the same zip archive, focusing on discrepancies in file parsing and metadata handling. It employs mutation of ZIP file to check whether two parsers exhibit divergent behaviors. Although ZIPDIFF considered the inconsistencies of path parsing, it lack a deeper analysis of creation order and sequence of symlink files.

Beyond parsing ambiguities, archive formats are vulnerable to resource exhaustion via zip bombs [8, 10, 25]. These attacks exploit high compression ratios, overlapping entries, or recursive structures to consume excessive CPU, memory, or disk resources upon extraction.

10 Conclusion

Symlinks, a file system feature supported at the operating system level, has become an indispensable component of modern computing environments. The symlink is allowed to reference to nearly any location in the file system and supports complex resolution behaviors. However, this flexibility also introduces significant misunderstandings for developers, which has led to widespread misuse and become a primary vector for critical security vulnerabilities.

In this paper, we present the systematic study of how symlinks are incorrectly handled and the substantial security risks. Using fine-grained test cases based on composing path elements, we identify four aspects of confusion in file type identification and categorized four inconsistent implementation strategies in path normalization. We recognize and summarize five existing path traversal defense mechanisms. We also implement five path validation bypass schemes and three arbitrary write methods. Our bypass and arbitrary write methods exploit 16 arbitrary file write vulnerabilities across 85 projects. We also demonstrate their severe impacts in real world scenarios and propose multiple mitigation strategies for different developers. We have responsibly disclosed these issues to the affected stakeholders.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U25A6023). We also acknowledge Alibaba Cloud Computing Co., Ltd. for its support of the research project associated with this work. Lei Zhang is the corresponding author.

Ethical Considerations

Vulnerability Discovery and Validation. All vulnerability analysis and validation procedures in this study were performed within strictly controlled, local environments. For vulnerabilities identified in open-source libraries, we verified arbitrary file write capabilities through local code audits and Proof-of-Concept (PoC) testing. Downstream impact assessments were executed in isolated settings, utilizing virtual machines and containerized sandboxes. Our experiments involved no interaction with real users, production systems, or public services, and no real-world attacks were conducted.

Disclosure and Remediation Collaboration. Upon confirming the identified flaws, we followed responsible disclosure practices, adopting the notification methodologies proposed by Stock et al. regarding large-scale vulnerability disclosure [30,31]. We reported all findings through private channels to the respective maintainers, specifically utilizing the GitHub Security Advisory mechanism [13] and direct developer email communication [31] to prevent premature exposure of technical details. We provided maintainers with comprehensive vulnerability reports, including full PoCs and reproduction steps. Throughout the process, we engaged in iterative communication with maintainers, contributing to remediation discussions, providing patch suggestions, and assisting with patch validation. At the time of writing, all disclosed vulnerabilities have been acknowledged by maintainers.

Supply Chain Impact and Ecosystem Coordination. Given the widespread adoption of the affected libraries, these vulnerabilities pose indirect risks to downstream projects via the software supply chain. Drawing on research by Zimmermann et al. [40] and Ladisa et al. [19] on software supply chain security, we assessed the propagation scope and potential impact of these flaws. In coordinating disclosure, we followed the multi-party vulnerability coordination guidelines established by FIRST [11], as well as the security response protocols of package managers such as PyPI [26] and npm [23]. Vulnerability details were reported to relevant communities and platforms through trusted third parties [31]. Furthermore, we proactively provided patch information to high-impact downstream projects, advising prompt updates to patched versions. To support ecosystem-wide defense while preventing misuse, this paper focuses on root causes and mitigation strategies, omitting technical specifics that could be directly weaponized.

Open Science

In accordance with the USENIX Security Open Science policy, we provide the following artifacts necessary to evaluate the contributions of this paper.

Datasets. We provide usage examples of 35 file type identification APIs and 34 path normalization APIs across 10 programming languages, as described in section 4. In addition, we include all decompression libraries and their wrapper implementations evaluated in section 5. These datasets enable reviewers to reproduce the measurements and analyses related to API behavior and inconsistencies across languages.

Test Cases. We provide the complete set of test cases constructed by combining base path patterns with symlink targets, as described in section 4. The test cases systematically cover representative symlink scenarios, including intermediate links, dot directory patterns, trailing slashes, and dangling symlinks. These cases are designed to evaluate behavioral inconsistencies among standard library APIs across different programming languages.

Source Code for Crafting Archives. We provide the source code for archive construction tooling supporting ZIP, TAR, and 7z formats. The repository also includes implementations of the bypass strategies, arbitrary file write methods described in section 5. These artifacts allow reviewers to reproduce the archive generation process and verify the reported behaviors.

Environment Configuration. We provide configurations and setup scripts that enable researchers to replicate our testing environment and reproduce the experimental results reported in this paper.

All artifacts are publicly available through the following Zenodo repository: https://doi.org/10.5281/zenodo.20325582. The repository has been accessible since submission time and will remain available throughout the review process. Upon acceptance, the anonymized repository will be updated to a permanent, non-anonymized public location.

References

[1] 4.1. Index Nodes — The Linux Kernel documentation, 2026. https://www.kernel.org/doc/html/latest/filesystems/ext4/inodes.html.

[2] path_resolution - linux manual page, 2026. https://man7.org/linux/man-pages/man7/path_resolution.7.html.

[3] symlink(7) - Linux manual page, 2026. https://man7.org/linux/man-pages/man7/symlink.7.html.

[4] Jafar Akhoundali, Hamidreza Hamidi, Kristian Rietveld, and Olga Gadyatskaya. Eradicating the unseen: Detecting, exploiting, and remediating a path traversal vulnerability across github. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, pages 542–558, 2025.

[5] S Alvarez and T Zoller. The death of av defense in depth?-revisiting anti-virus software. *CanSecWest, Vancouver, BC, Canada*, 2008.

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

[7] 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.

[8] Marie Cara. I came to drop bombs: Auditing the compression algorithm weapons cache. *Blackhat USA*, 2016.

[9] Ran Chen, Martin Clochard, and Claude Marché. A formally proved, complete algorithm for path resolution with symbolic links. *Journal of Formalized Reasoning*, 10(1):51–66, 2017.

[10] David Fifield. A better zip bomb. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[11] FIRST. Guidelines and practices for multi-party vulnerability coordination and disclosure, 2026. https://www.first.org/global/sigs/vulnerability-coordination/multiparty/guidelines-v1.1.

[12] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *In Proc. Network and Distributed Systems Security Symposium*, 2003.

[13] GitHub. About GitHub security advisories, 2026. https://docs.github.com/en/code-security/security-advisories.

[14] The IEEE and The Open Group. symlink, the open group base specifications issue 8, ieee std 1003.1-2024, 2026. https://pubs.opengroup.org/onlinepubs/9799919799/functions/symlink.html.

[15] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *2012 IEEE Symposium on Security and Privacy*, pages 80–94. IEEE, 2012.

[16] Luke Koch, Sean Oesch, Amul Chaulagain, Mary Adkisson, Samantha Erwin, and Brian Weber. Toward the detection of polyglot files. In *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, pages 120–128, 2022.

[17] Luke Koch, Sean Oesch, Amir Sadovnik, Brian Weber, Amul Chaulagain, Matthew Dixson, Jared Dixon, Mike Huettel, Cory Watson, Jacob Hartman, et al. On the abuse and detection of polyglot files. In *Proceedings of the ACM on Web Conference 2025*, pages 4810–4822, 2025.

[18] 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, March 2008.

[19] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *44th IEEE Symposium on Security and Privacy (S&P)*, pages 1509–1526, 2023.

[20] Yu-Tsung Lee, Hayawardh Vijayakumar, Zhiyun Qian, and Trent Jaeger. Static detection of filesystem vulnerabilities in android systems. *arXiv preprint arXiv:2407.11279*, 2024.

[21] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node. js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 143–160, 2022.

[22] Zhi Li, Weijie Liu, XiaoFeng Wang, Bin Yuan, Hongliang Tian, Hai Jin, and Shoumeng Yan. Lost along the way: Understanding and mitigating path-misresolution threats to container isolation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*, pages 3063–3077, 2023.

[23] npm, Inc. Reporting malware in an npm security policy, 2026. https://docs.npmjs.com/reporting-a-vulnerability-in-an-npm-package.

[24] Gregory R Panakkal. Leaving our zip undone: How to abuse zip to deliver malware apps. In *Virus Bulletin Conference*, 2014.

[25] Giancarlo Pellegrino, Davide Balzarotti, Stefan Winter, and Neeraj Suri. In the compression {Hornet's} nest: A security study of data compression in network services. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 801–816, 2015.

[26] Python Packaging Authority. PyPI security policy and vulnerability reporting, 2026. https://pypi.org/security/.

[27] Razvan Raducu, Ricardo J Rodríguez, and Pedro Álvarez. Defense and attack techniques against file-based toctou vulnerabilities: A systematic review. *IEEE Access*, 10:21742–21758, 2022.

[28] Chris Riddoch. The trouble with symbolic links. *LWN.net*, July 2022.

[29] Snyk. Zip slip vulnerability, 2018. https://security.snyk.io/research/zip-slip-vulnerability.

[30] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. Didn't you hear me? – towards more successful web vulnerability notifications. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[31] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1015–1032, 2016.

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

[33] Mario Vuksan, Tomislav Pericin, and Brian Karney. Hiding in the familiar: Steganography and vulnerabilities in popular archives formats. *Proceedings of Black Hat Europe*, 2010.

[34] Jinpeng Wei and Calton Pu. Tocttou vulnerabilities in unix-style file systems: An anatomical study. In *FAST*, volume 5, pages 12–12, 2005.

[35] Wikipedia. Symbolic link. https://en.wikipedia.org/wiki/Symbolic_link, 2026. Accessed: 2026-02-06.

[36] Bocheng Xiang, Yuan Zhang, Fengyu Liu, Hao Huang, and Zihan Lin. Pig in a poke: Automatically detecting and exploiting link following vulnerabilities in windows file operations. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 7019–7038, 2025.

[37] Yufan You, Jianjun Chen, Qi Wang, and Haixin Duan. My {ZIP} isn't your {ZIP}: Identifying and exploiting semantic gaps between {ZIP} parsers. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 431–450, 2025.

[38] Chendong Yu, Yang Xiao, Jie Lu, Yuekang Li, Yeting Li, Lian Li, Yifan Dong, Jian Wang, Jingyi Shi, Defang Bo, et al. File hijacking vulnerability: The elephant in the room. In *Proceedings 2024 Network and Distributed System Security Symposium (2024). https://api. semanticscholar. org/CorpusID*, volume 267621808, 2024.

[39] Qiyi Zhang, Fengyu Liu, Zihan Lin, and Yuan Zhang. Be aware of what you let pass: Demystifying url-based authentication bypass vulnerability in java web applications. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 1874–1888, 2025.

[40] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 995–1010, 2019.