Pig in a Poke: Automatically Detecting and Exploiting Link Following Vulnerabilities in Windows File Operations

Bocheng Xiang, Yuan Zhang, Fengyu Liu, Hao Huang, Zihan Lin, Min Yang
Fudan University, China

Abstract

Symbolic links are widely utilized in file operations on the Windows system to facilitate seamless interaction and enhance the overall user experience. However, developers' failure to properly validate symbolic links during the process of file operations has led to the Link Following Vulnerabilities (LFVulns), enabling attackers to manipulate system files arbitrarily.

In this paper, we conduct a comprehensive analysis of existing LFVulns and reproduce 42 of them for in-depth empirical research. Our findings uncover the root causes of LFVulns and identify key factors hindering their detection and exploitation. To bridge this gap, we developed `LinkZard`, a prototype for the automated detection and exploitation of LFVulns targeting Windows systems. `LinkZard` consists of two main phases. The exploration phase employs efficient file state fuzzing to better uncover potential vulnerabilities, while the exploitation phase locates sinks and utilizes code wrapping strategies to achieve automatic exploitation. We applied `LinkZard` to 120 commercial programs from vendors such as Microsoft, Apple, and Intel, successfully detecting and exploiting 55 zero-day vulnerabilities. We responsibly reported all identified vulnerabilities to the affected vendors. Up to now, 49 of them have been confirmed and patched, resulting in 15 CVE assignments and bounty rewards.

1 Introduction

Symbolic links [1] are widely used in file operations on the Windows system, significantly enhancing the user experience by providing features such as desktop shortcuts [2] and directory junctions [3]. While offering conveniences, the misuse of symbolic links also introduces significant security risks, referred to as Link Following Vulnerabilities (LFVulns), enabling arbitrary file manipulation and leading to critical consequences such as local privilege escalation (LPE) [4] and denial of service (DoS) [5].

The root cause of LFVulns lies in the lack of proper validation for symbolic links during the process of file operations. When files are controlled by a low-privileged attacker (e.g., a regular user), they can be crafted into symbolic links pointing to sensitive files. If a privileged program fails to validate the files and follows the links, it may result in arbitrary manipulation of sensitive files, thus resulting in LFVulns.

To the best of our knowledge, no prior work has focused on the detection and exploitation of LFVulns. The most relevant work, Jerry [6], identifies file-hijacking vulnerabilities caused by weak file permission controls. While the root causes of file hijacking vulnerabilities and LFVulns differ, their detection similarities enable Jerry to detect a limited set of LFVulns. However, it fails to thoroughly explore file operations and relies only on a dangerous single-file operation, resulting in a significant number of false negatives and false positives. Additionally, it lacks the capability for automated LFVulns exploitation.

Therefore, in this work, we are highly motivated to design an automatic approach for detecting and exploiting LFVulns in the Windows system. To gain a deeper understanding of LFVulns, we conducted an empirical study to investigate the underlying causes and exploits of existing LFVulns. Our findings reveal that such automation is a non-trivial task, and the following two challenges must be properly addressed.

- **Challenge-1: How to solve file state constraints for effective detection of LFVulns?** File state constraints are specific file conditions that must be satisfied before deeper file operations proceed. For example, in a log backup routine, a backup may only be triggered when the file size exceeds a threshold, typically expressed as a condition like `if(file.size > BACKUP_SIZE)`. Such constraints are common and serve as necessary preconditions for triggering LFVulns. However, due to the diversity of file states and the black-box nature of program functionalities, we lack effective methods to accurately solve and infer whether these constraints have been addressed.
- **Challenge-2: How can we automate the exploitation of LFVulns?** Automated exploitation requires locating sinks within complex file operations and applying suitable exploitation strategies. Here, we define a sink as a manually

defined sequence of high-risk file API operations that operate on the same file and collectively indicate an exploitable condition. Even after the sink is successfully located, exploitation strategies for pre-sink (i.e., constraints before the sink) and on-sink (i.e., constraints within the sink) constraints differ significantly.

In this paper, we propose a novel security analysis approach for the automated detection and exploitation of LFVulns in the Windows system, called `LinkZard`. Specifically, our approach is inspired by several key insights. First, file operations are often accompanied by file state queries, and these states are highly concentrated, which allows us to solve potential state constraints. Second, the sinks of LFVulns are composed of an invocation sequence of file operation APIs, which forms a method call graph. Besides, the entire program's set of file operations also constitutes a large graph. Therefore, locating the sink within a complex program can be formulated as a subgraph isomorphism problem. Based on these insights, `LinkZard` implements automated detection and exploitation of LFVulns through two key phases.

In the exploration phase, we implement a feedback-driven file state fuzzing strategy to dynamically solve file state constraints. To infer whether the constraints have been solved, we use a two-dimensional (i.e., operation count and operation types) analysis of file operations. Specifically, we obtain specific file state (e.g., file name, size) query information to guide the fuzzing process. Additionally, we have developed three efficient mutation operators that target these file states to effectively address these constraints. By comparing the type and quantity of file operations before and after fuzzing, we can infer whether the constraints have been solved, which ensures thorough exploration of privileged programs and efficient detection of LFVulns.

The exploitation phase consists of three processes. First, we formalize the file operations from the exploration phase into a File Operation Primitive Graph (FOPG), which comprehensively represents the invocation sequence between file operations. Based on our second insight, we leverage a subgraph matching [7] algorithm to locate the sink, and then categorize constraints as pre-sink or on-sink based on their position relative to the sink. Finally, we apply two distinct code-wrapping strategies to handle pre-sink and on-sink constraints. This approach's effectiveness lies in its reliance on constraint types rather than specific operations, making it applicable to LFVulns across various scenarios.

To evaluate the effectiveness of `LinkZard` in detecting and exploiting vulnerabilities, we constructed a benchmark comprising 42 known vulnerabilities. Our evaluation shows that `LinkZard` successfully detected 38 known vulnerabilities, outperforms the current state-of-the-art tool (i.e., Jerry [6]) with improvements of 29.41% in precision rate and 33.34% in recall rate. For the automatic exploitation, `LinkZard` successfully exploited 33 of them, achieving a success rate of 86.84% (33/38). Furthermore, we applied `LinkZard` to 120 programs from well-known vendors, including Microsoft, Apple, Intel, HP, and Tencent. We detected and successfully exploited 55 zero-day vulnerabilities in 49 of these programs. Considering the widespread use of these programs and the significant security threats the vulnerabilities pose to their user base, we responsibly reported all vulnerabilities to the vendors. To date, these vulnerabilities have been assigned 15 CVE identifiers. These evaluations confirm that `LinkZard` is highly effective in automating the detection and exploitation of LFVulns in real-world scenarios.

The contributions of this paper are summarized as follows:

- We introduce a threat model for LFVulns and, based on this model, present the first systematic empirical study of LFVulns, which reveals their root cause and key characteristics. Additionally, we provide several novel insights into the automated detection and exploitation of LFVulns.
- Building on these insights, we propose `LinkZard`, the first prototype for the automated detection and exploitation of LFVulns. `LinkZard` effectively solves file state constraints without human intervention and wraps exploitation code to achieve the successful exploitation of LFVulns.
- Our evaluation of 120 real-world popular programs shows that `LinkZard` can automatically detect and successfully exploit 55 zero-day vulnerabilities. All vulnerabilities were reported to the vendors, with 49 confirmed or patched. To date, 15 CVE IDs have been assigned.

2 Background & Problem Statement

2.1 Link Following Vulnerability

2.1.1 LFVuln Overview

Symbolic link [1] is a widely used mechanism in the Windows system. Developers who work with symbolic link files can follow the link and directly interact with the target file. This flexibility and transparency feature significantly enhances the functionality of the file system. However, due to insufficient checks on symbolic links, low-privileged attackers can exploit carefully crafted malicious symbolic link files to trick high-privileged programs into accessing and manipulating sensitive files, leading to Link Following Vulnerabilities (LFVulns). While the immediate consequences of LFVulns resemble those of path traversal vulnerabilities [8], such as arbitrary file moves or deletions, their root causes differ fundamentally. The root cause of LFVulns is presented in §3.2.

Figure 1 illustrates the attack workflow of LFVulns. The origin design intention by the developer was that the privileged program could directly delete the target file through a symbolic link file created by an administrator (green line). However, an attacker may exploit this functionality to delete arbitrary files without authentication (red line). Specifically, to exploit this vulnerability, ① a low-privileged attacker creates a symbolic link file that points to a protected file, which the

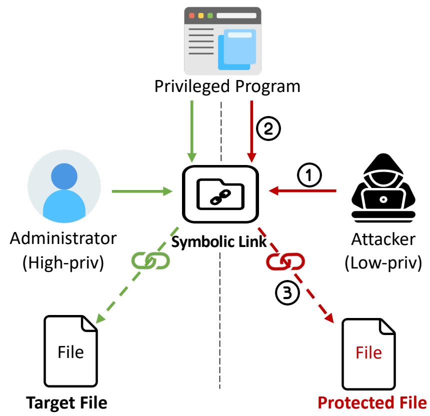


Figure 1: An Overview of Normal Symbolic Link and Link Following Vulnerabilities in File Deletion

attacker does not have permission to access. ② The privileged program fails to perform proper validation on the file and follows the symbolic link to operate on the target protected file with elevated privileges. ③ As a result, the protected file is unauthorizedly deleted, thereby compromising the confidentiality, integrity, and availability of the system.

2.1.2 Threat Model

In our threat model, we assume that the attackers gain access to the target system through means such as weak password attacks or malware infections, obtaining a standard user account (low-privileged). With this level of access, the attackers can interact with privileged applications through inter-process communication (IPC) [9] to trigger file operations, and for certain files, configure symbolic links to exploit vulnerabilities. We classify the security risks caused by the LFVulns into the following two categories.

- *Denial of Service (DoS):* Attackers can exploit LFVulns to create files essential for system initialization and kernel module loading, leading to Denial of Service. For example, creating a file at `C:\Windows\System32\cng.sys` can result in boot failures, ultimately rendering the system unusable.
- *Local Privilege Escalation (LPE):* Attackers can exploit LFVulns to delete sensitive files and even entire folders or move arbitrary files (e.g., a malicious DLL), leading to Local Privilege Escalation. For instance, an attacker can copy a malicious DLL to `C:\Windows\System32` directory to achieve privilege escalation through DLL side-loading [10] by exploiting LFVulns.

Notably, to better illustrate how arbitrary file deletion can result in privilege escalation, we present a commonly exploited technique below. Windows systems include an automatic rollback mechanism [11] in their installer framework to restore the system to its original state upon installation failure. During installation, a privileged installer service creates a protected directory, i.e., `C:\Config.msi`, which stores a rollback script (`.rbs`) and a corresponding rollback file (`.rbf`). One exploitation technique abuses LFVulns to cause a privileged program to delete this directory. The attacker can then recreate it and inject malicious `.rbs` and `.rbf` files. When the rollback mechanism is triggered, the malicious scripts execute with elevated privileges, resulting in privilege escalation.

2.2 LFVulns Exploitation on Windows

Attackers typically rely on two key mechanisms to exploit the LFVulns in Windows file operations: *Pseudo-Symbolic Links* and *Opportunistic Locks*.

Pseudo-Symbolic Links. Actually, creating a traditional symbolic link in Windows requires administrator privileges [12]. Interestingly, attackers can leverage two mechanisms in Windows to achieve the functionality of symbolic links without needing administrator privileges. (1) Directory Junctions [3] (aka. Mount Points) enable the linking of one directory to another target directory and do not require administrator privileges. This makes them accessible to attackers, who can create directory junctions using the command `mklink /j <source> <target>`, setting the `source` directory as a mount point linked to the `target` directory. (2) Object Manager Symbolic Links (ObjSymlinks) exist within the Object Manager's [13] namespaces and reference various system objects, such as devices, files, or directories. These namespaces, functioning as special directories, can also be mounted using directory junctions. For instance, the command `mklink /j <source> \RPC Control` mounts `source` directory to the `RPC Control` namespace. Notably, some namespaces (e.g., `\RPC Control\`) are writable by low-privileged users, allowing attackers to create ObjSymlinks within these namespaces that point to arbitrary files.

In general, to construct a pseudo-symbolic link, attackers first leverage directory junctions to mount the directory path of a vulnerable file into writable namespaces in the object manager. They then create a symbolic link within these namespaces that points to the target file.

Opportunistic Lock. An Opportunistic Lock (OpLock) [14] temporarily blocks access to a file, granting exclusive control during specific operations. File operations in programs often occur sequentially, and when certain conditions are required to trigger LFVulns, the lack of a mechanism to pause a privileged program's file operations might cause the pseudo-symbolic link to be created after the vulnerable file operation has been completed, leading to exploitation failure. In such cases, the attacker must first meet the required condition, then use Oplock to pause file operations, creating a stable time window to configure the pseudo-symbolic link before the vulnerable operation occurs. Once the Oplock is released, the exploitation is completed.

In summary, attackers can leverage two key Windows-specific mechanisms to reliably construct pseudo-symbolic links within the exclusive time window provided by Oplock, enabling the exploitation of elevated program privileges to access or manipulate sensitive files.

2.3 Real-World Example

Privileged Program Vulnerable Code

```
1  void IPC_CleanTempFile() {
     // Operates on an attacker-controlled directory
2    wchar_t * dirPath = GetTempFilePath();
3    sprintf(searchPath, L"%s\\*.tmp", dirPath);
     // Locate .tmp files in the specified directory
4    hFind = FindFirstFile(searchPath, &fdata);
5    do {
6      wchar_t filePath[MAX_PATH];
7      PathCombine(filePath, dirPath, fdata.cFileName);
       /** Remove located temporary files */
8      DeleteFile(filePath);
9    } while (FindNextFile(hFind, &fdata));
10 }
```

Attacker Exploit Code

```
1  void Exploit() {
2    HANDLE hFile = CreateFile(L"attack.tmp");
3    Oplock oplock = new Oplock(hFile);
     // Use Oplock to create a time window
4    oplock->SetOplock(Trigger_Callback, hFile);
5    oplock->Release();
6  }
7  void Trigger_Callback(HANDLE hFile){
8    wchar_t* dirPath = GetFileDir(hFile);
9    wchar_t* symPath = L"\\RPC Control\\attack.tmp"
10   CreateJunction(dirPath, L"\\RPC Control");
11   CreateObjSymlink(symPath, victim_file);
12 }
```

Figure 2: CVE-2024-491**: Windows W** Service LFVuln vulnerability and its exploitation code fragment

We use a real-world LFVuln we detected in a Windows system service (i.e., W** service, anonymized for ethical reasons) to better demonstrate the LFVuln. In Figure 2, we present the simplified code (decompiled using IDA-Pro [15]) of the vulnerable privileged program alongside the exploit code snippet. This vulnerability arises from the improper deletion of files without strict access control by a privileged program, allowing an attacker to exploit this flaw for arbitrary file deletion and subsequent privilege escalation.

The red dashed arrow in Figure 2 represents the normal file operation path. The `IPC_CleanTempFile` function clears temporary files. Lines 2-4 locate the first `.tmp` file in the temporary directory using `GetTempFilePath` and `FindFirstFile`. The directory obtained through `GetTempFilePath` (Line 2) is not protected by any access control policy and thus can be controlled by attackers. Lines 5-9 iterate through all matching `.tmp` files using `FindNextFile` and delete them sequentially via `DeleteFile` (Line 8). Due to the privileged program failing to validate symbolic links for the files being deleted, this ultimately leads to an LFVuln.

The red solid arrows in Figure 2 represent the file operation path during the exploitation phase. The exploitation code follows a multi-step process. First, in the `exploit` function, Lines 2-4 create a `.tmp` file and set an Oplock (Line 3) with a callback function `TriggerCallback` (Line 7). This ensures that the privileged program cannot delete the file after identifying it until the Oplock is released, creating a stable time window for further operations. Second, within `TriggerCallback` (Line 7), the path information is constructed (Lines 8-9), followed by configuring a directory junction (Line 10) and an ObjSymlink (Line 11) to set up the pseudo-symbolic link, linking the `.tmp` file path to `victimFile`. Finally, the Oplock is released (Line 5), causing the privileged program to follow the pseudo-symbolic link and erroneously delete `victimFile`.

Such vulnerabilities are prevalent in privileged programs, as file operations are fundamental to almost all programs. This example demonstrates that detecting and exploiting these vulnerabilities can be relatively complex and challenging.

2.4 Existing Work and Limitations

To the best of our knowledge, Jerry [6] represents the most closely related state-of-the-art work in this area. Specifically, Jerry interacts with programs using randomized GUI actions and command-line options, detecting file-hijacking vulnerabilities by identifying weakly permissioned files in program operations. It flags potential vulnerabilities when actions such as creation, deletion, or even reading are performed on attacker-controlled files lacking strict access control policies.

However, due to the differing root causes of the two vulnerabilities, this approach exhibits limitations in the context of LFVulns. First, Jerry does not comprehensively explore file operations, resulting in a low recall rate (only 57.14% in our dataset). This limitation arises from its exclusive focus on the existence of files, thereby overlooking numerous other potential file states (e.g., file name) that can impact the program's file operation process. Second, Jerry's detection strategy is overly coarse-grained, reporting vulnerabilities based on one single file operation (e.g., reporting a vulnerability whenever the program reads an attacker-controlled file). This detection strategy leads to numerous false positives and heavily relies on expert knowledge for analysis, thereby requiring substantial human effort. Finally, Jerry lacks support for exploiting LFVulns. The complexity of exploitation steps means that even after detecting vulnerabilities, significant expert effort and time are required to achieve successful exploitation.

3 Empirical Study

In this section, we present the methodology and findings of our empirical study on LFVulns. We begin by detailing the process of LFVuln collection (in §3.1), followed by a comprehensive analysis of 145 LFVulns across multiple dimensions. In summary, we analyzed and manually reproduced 42 vulnerabilities as part of our empirical study. This study aims to address three critical questions, i.e., ❶ What is the root cause of LFVulns (in §3.2), ❷ What are the types of dangerous file operations (sinks) that lead to LFVulns? (in §3.3), and ❸ What factors significantly hinder the detection and exploitation of LFVulns? (in §3.4)

3.1 Vulnerability Collection and Analysis

To enable a comprehensive study of LFVulns, we collected all recorded cases from the past five years through a cross-referenced analysis of keyword-based identification and CWE [16] mapping. Specifically, we performed systematic keyword searches in the CVE database [17] using terms such as *Link Following*, *Privilege Escalation*, and *Symbolic*. Concurrently, we queried the NVD (National Vulnerability Database) [18] for CWEs related to symbolic links, specifically *CWE-{59-64}* [19–24], in order to collect CVE-identified vulnerabilities associated with these weaknesses. The intersection of CVEs identified through both methods served as the confirmed dataset of LFVulns. For the remaining results, we manually examined vulnerability descriptions and associated CWEs to ensure their relevance. This methodology yielded a total of 408 LFVulns recorded.

Subsequently, focusing on LFVulns in Windows applications, we excluded vulnerabilities specific to Linux, Mac, and Android, with 79, 46, and 2 vulnerabilities removed from each platform, respectively. This filtering resulted in 281 vulnerabilities specific to the Windows platform.

Finally, we excluded vulnerabilities from the dataset for which detailed exploitation information was unavailable. Specifically, we first removed cases with no disclosed details or insufficient information to enable meaningful analysis. Subsequently, we collected publicly available exploit code from platforms such as GitHub [25], HackerOne [26], and ExploitDB [27]. For vulnerabilities with sufficient details but lacking published exploit code, manual reproduction was conducted by three authors through a cross-validation process to ensure accuracy and consistency. Ultimately, we curated a dataset of 145 vulnerabilities for empirical study, successfully reproducing 42 of them as ground truth (detailed in Table 5 in Appendix B) for further evaluation §5.2. The entire process of dataset collection, analysis, and reproduction required approximately two months to complete.

3.2 Root Causes

Finding 1: The root cause of LFVulns originates from the inadequate validation of symbolic links during the process of file operations.

In Windows file operations, symbolic links are widely utilized to enable seamless interaction, enhancing the overall user experience. However, as a feature of file operations in Windows, following symbolic links is enabled by default—a behavior that developers often overlook. Consequently, programs lack adequate validation of symbolic links during the process of file operations. When files are under attacker control (e.g., files in `C:\Windows\temp`), this oversight allows attackers to craft symbolic links and arbitrarily manipulate system files. Furthermore, the diversity of file operations in the Windows system complicates developers' ability to ensure adequate validation of symbolic links across all operations. This results in a "weakest link" [28] effect in security, where the absence of proper validation in any single functionality involving file operations can lead to the emergence of LFVulns.

3.3 Sink Types

Finding 2: There are four distinct types of dangerous file operations (sinks) leading to LFVulns, each of which can be represented by specific sequences of file operation APIs.

The sinks are manually defined sequences of high-risk file API operations targeting the same file, distilled from our study to capture representative vulnerable file operations. These APIs, part of the fundamental Windows APIs [29] provided by Microsoft, encapsulate all file operations. Table 4 in Appendix A presents the API sequences for each sink. We detail the four sink types below and use a real-world example to illustrate an API sequence for a sink in our study.

❶ Unsafe Creation, Write, and Overwriting (49, 33.8%). This sink arises from the frequent use of file creation and write operations in programs, leading to a higher prevalence of LFVulns under this type. These vulnerabilities are often exploited to achieve arbitrary file creation and writing.

❷ Unsafe Copying and Moving (14, 9.7%). When both the source and destination files of copy or move operations are attacker-controlled, LFVulns in this category can be leveraged to achieve arbitrary file movements, including file deletion and creation.

❸ Unsafe Access Control Configuration (14, 9.7%). This sink arises when privileged programs assign permissive ACL [30] to files without verifying whether the file is a symbolic link. Such unsafe access control configurations allow attackers to redirect permissions to arbitrary files.

❹ Unsafe Deletion (68, 46.8%). This sink is the most prevalent in our dataset, primarily due to two factors: *first*, programs frequently create and delete temporary files as part of normal operations; *second*, the inherent design of deletion processes in Windows often involves following directory junctions, significantly increasing the likelihood of LFVulns.

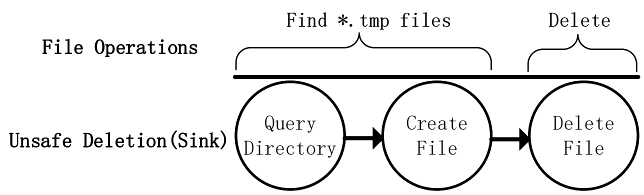


Figure 3: The Sink of Unsafe Deletion in the Real-world LFVuln

Figure 3 depicts a sink in the real-world LFVuln(§2.3). The file operations performed by the privileged program are implemented through a sequence of API calls (QueryDirectory → CreateFile → DeleteFile). This sequence represents the privileged program first performing a specific file state query targeting the file directory. After completing the query, it proceeds to delete these files. When the files are under the control of an attacker, this results in an LFVuln, allowing the attacker to achieve arbitrary file deletion.

3.4 File State Constraints

Finding 3: File state constraints are the most critical factor hindering both the automated detection and exploitation of LFVulns. These constraints are widely distributed across 66 vulnerabilities, accounting for 46% (66/145) of the total dataset analyzed.

File states represent the unique characteristics of a file, encompassing attributes such as file name, size, content, and time-related properties (e.g., creation and modification timestamps). It is worth noting that we adopt the term *state* rather than *metadata*, following its usage in recent research [31] [32], [33], as it more accurately reflects the dynamic conditions of files observed during runtime. These attributes collectively define the state of a file, influencing its behavior and interactions during file operations.

Programs often require a set of specific file states to be satisfied before performing deeper or more critical file operations. We define these prerequisite states, which may block further file operations if unmet, as *file state constraints*. File state constraints refer to conditions where the absence or mismatch of specific file states causes the privileged program to terminate or skip subsequent operations on the file. Our study found that, among the 66 vulnerabilities, approximately 41% are related to file name constraints, where the file name must adhere to specific formats (e.g., UUID, timestamps) or extensions (e.g., .log, .tmp). Around 21% of the vulnerabilities are linked to file content constraints (e.g., Magic Number [34]). Interestingly, we observed that certain antivirus software is affected by these vulnerabilities. Such software typically runs with elevated privileges and incorrectly deletes malicious files during virus scanning, leading to LFVulns.

File state constraints can be classified into two categories based on their presence at different positions relative to the sink within the file operation sequence:

Pre-sink constraints (12/66, 18.2%). Pre-sink constraints refer to all file state constraints that exist before a privileged program's file operation triggers the sink. These constraints present a significant barrier to the detection of LFVulns. When pre-sink constraints remain unsolved, further exploration of privileged program file operations becomes infeasible. As a result, potential sinks in deeper file operations are not triggered, leading to missed vulnerabilities. Moreover, for pre-sink constraints, once the attacker solves the constraint, they can create a pseudo-symbolic link at any point before the sink is triggered, enabling successful exploitation.

On-sink constraints (54/66, 81.8%). Unlike pre-sink constraints, on-sink constraints refer to file state constraints present within the dangerous API sequences (sinks) of a privileged program's file operation. This type of constraint significantly interferes with the automated exploitation of LFVulns. In contrast to pre-sink constraints, as the constraints reside within the dangerous API sequences (sinks), their solution prompts the privileged program to proceed with the subsequent API calls. As a result, the attacker must race against the program, competing for the time window between constraint resolution and the program's execution of the subsequent operation to create the pseudo-symbolic link. However, in all the ground truth vulnerabilities, we found that for LFVulns with on-sink constraints, the Oplock is consistently used as a stable exploitation method. However, automating Oplock-based exploitation remains a significant challenge.

4 The Methodology of `LinkZard`

In this section, we first summarize the challenges we encountered and our key insights (in §4.1.1) and present our proposed solutions (in §4.1.2). We then thoroughly elaborate on the two main phases in `LinkZard`: the exploration phase (in §4.2) and the exploitation phase (in §4.3).

4.1 Overview

4.1.1 Challenges and Insights

Given the severe security threats posed by LFVulns, our objective is to develop an effective approach for detecting and exploiting potential LFVulns. To achieve this goal, we face two straightforward key challenges:

- **Challenge 1: How to solve file state constraints for effective detection of LFVulns?** As indicated in §3.4, 46% of the vulnerabilities exhibit file state constraints, highlighting that efficiently detecting LFVulns necessitates addressing these constraints inherent in program file operations. To the best of our knowledge, no existing work effectively addresses file state constraints in LFVulns. Although dynamic

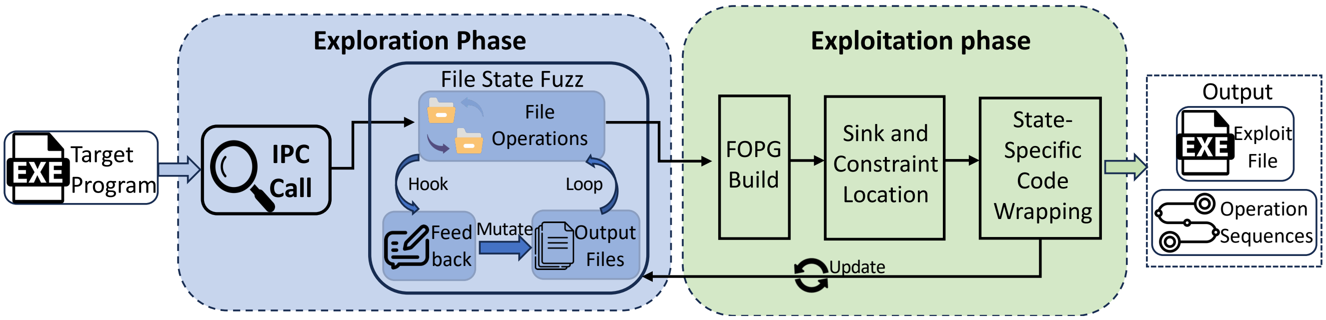


Figure 4: `LinkZard` Architecture: Workflow Overview for Link Following Vulnerabilities

and static constraint-solving techniques offer valuable insights, significant challenges remain. On one hand, the complexity of application functionalities poses substantial difficulties for static approaches such as static analysis and symbolic execution, which are prone to well-documented issues like path explosion [35] and code obfuscation [36]. On the other hand, advanced dynamic fuzzing tools for file inputs lack feedback mechanisms and mutation strategies tailored to file states. For instance, tools like AFLSmart [37] and Peach Fuzz [38] primarily focus on byte-level or structural mutations, making them ineffective for resolving the file state constraints inherent in LFVulns. Furthermore, our research indicates that current solutions for addressing file state constraints heavily rely on manual implementation.

- **Challenge 2: How can we automate the exploitation of LFVulns?** The complexity of application functionality leads to long sequences of file operations. The challenge of automating exploitation lies in two main aspects: first, we are unable to pinpoint the exact position within the file operation sequence where the attacker's series of actions should be performed. Traversing all positions and file operations results in an exponential increase in the complexity of the exploitation process. Second, even if the correct position is identified, for pre-sink constraints, we can create a pseudo-symbolic link and solve the constraint to complete the exploitation. However, for on-sink constraints, there is a lack of prior knowledge on how to solve the constraints while simultaneously using an Oplock to stabilize the exploitation. As shown in §2.3, this LFVuln exhibits on-sink constraints. Successful exploitation requires resolving the state constraints before the *FindFirstFile*, while setting an Oplock and performing various exploitation-related file operations, such as creating a pseudo-symbolic link.

Insights. For the two challenges outlined above, our insights are distinct yet effective in addressing each one.

First, as described in *Challenge 1*, accurate feedback and efficient mutation operators are two crucial components in fuzzing file states. Our key observation is that file operations are typically accompanied by specific state queries, which often correspond to file states imposing constraints. These state constraints are primarily concentrated on attributes such as file name format, content, size, and time. This concentration allows us to focus on a subset of critical file states rather than the entire state space. By applying targeted mutation operators to these states, we can leverage the information obtained from state queries as feedback to guide and prioritize the mutation process. This feedback-driven approach not only improves the efficiency of addressing file state constraints but also ensures that the fuzzing process remains scalable for complex programs with diverse file operations.

Second, as outlined in *Challenge 2*, the sink represents a shorter invocation sequence of file operation API calls, capturing the essential steps of a vulnerable operation. This sequence can be formed into a method call graph, while the entire set of file operations constitutes a large graph. This size disparity suggests that the task of locating the sink within the file operations can be transformed into a subgraph isomorphism problem. The key insight here is that by recognizing this structural difference, we can apply graph-matching techniques to locate the sink. Building on this insight, we leverage the principles used to solve the subgraph isomorphism problem [7] to address this challenge, enabling efficient and accurate sink location.

4.1.2 Solutions

Following the insights, we propose `LinkZard`, the first prototype for the automated detection and exploitation of LFVulns. As shown in Figure 4, it consists of two main phases: exploration and exploitation phase.

Solution for Challenge 1: Exploration Phase. In this phase, we first interact with privileged programs through `IPC Call`. Guided by *Finding 2* and our insights, to detect LFVulns efficiently, we designed a feedback-driven file state fuzz, algorithm 1 outlines the overall process of file state fuzz. The fuzzing leverages API hooking for state queries to obtain specific types of state query information, including all file states mentioned in §3.4. This feedback effectively guides the selection of file states for mutation. We employ three distinct and efficient mutation strategies to target concentrated file state constraints. Following this, we perform a two-dimensional analysis of file operations to infer whether the state constraints have been resolved. `LinkZard` transitions from exploration to exploitation upon inferring that the file state constraints have been solved during fuzzing. It is important to note that during the file state fuzzing phase, our focus is not on distinguishing between pre-sink or on-sink constraints, but rather on determining whether the constraints have been successfully solved. Ultimately, we get a complete set of file operations performed by the privileged program.

Solution for Challenge 2: Exploitation Phase. In light of our insights, the exploitation phase consists of three key processes. In the first process, all file operations of privileged programs are structured into a *File Operation Primitive Graph (FOPG)*, a directed graph that captures the file operations and their sequential relationships. Next, in the second process, we perform graph matching to identify the portion of the FOPG that matches the API sequence of the sink. Upon locating the sink, this indicates the detection of an LFVuln. Subsequently, we traverse the graph to determine whether the constraint is pre-sink or on-sink, which is crucial for exploitation. In the third process, we perform state-specific code wrapping. The wrapping process involves the following steps: (1) Extract the exploitation fragment's state-specific dependency information from the sink and constraint files, including file state (e.g., file name and path) and sink type (e.g., Unsafe Deletion). (2) For both types of constraints, we apply distinct wrapping strategies to assemble the exploitation code, which is then compiled and executed to achieve the exploitation of LFVulns. Notably, this approach depends on the constraint type rather than the file operation itself, making it applicable to LFVulns across various operational scenarios. After each exploitation, if successful, it results in output; otherwise, the privileged program's file operations are updated, and the process returns to the exploration phase.

4.2 Exploration Phase

4.2.1 IPC Call

To achieve direct and efficient interaction with privileged programs, we leverage IPC (Inter-Process Communication) [9] to establish direct communication. Specifically, we utilize two parallel channels by analyzing typical communication mechanisms, i.e., Service Management [39] and RPC (Remote Procedure Call) [40], to explore interaction methods with privileged programs. When the input executable file is registered as a service program, we establish an effective communication channel with the privileged program by issuing control commands (e.g., start, stop, restart) to the target service via the Service Manager, as these commands often trigger extensive file operations within the privileged program. In parallel, we systematically analyze the RPC interfaces exposed by the input program, synthesize RPC stubs by inferring parameter types from IDL metadata [41], and adapt the majority of parameter types, with random parameter values generated during each interface invocation. We adopt a bottom-up and intuitive approach for handling complex nested interface parameters (e.g., nested structures). Specifically, we begin by recursively identifying and constructing sub-structures whose members are primitive types rather than other structures. These sub-structures are then iteratively integrated into their parent structures, continuing this process until the entire nested structure of the interface parameter is fully constructed. For unknown parameter types, we use `NULL` as a placeholder to ensure successful invocation. By combining these two approaches, we achieve efficient and direct interaction with privileged programs, thereby allowing the triggering of file operations by the privileged programs.

4.2.2 File State Fuzz

We designed comprehensive feedback and mutation strategies to effectively fuzz file states, and then leveraged the two-dimensional analysis of file operations to infer and solve file state constraints, as illustrated in algorithm 1.

Algorithm 1: File State Fuzz Workflow

Input: Initial set of privileged file operations O_i
Output: Expanded set of file operations O_o

Initialization: $\mathcal{F} \leftarrow$ ExtractFiles(O_i);
while *True* **do**
  $O_o \leftarrow \emptyset$;
  foreach $f \in \mathcal{F}$ **do**
    $\mathcal{S}_f \leftarrow$ Feedback(O_i, f);
    $\mathcal{F}' \leftarrow$ Mutate$(f, \mathcal{S}_f)$;
    $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}'$;
  foreach $f' \in \mathcal{F}$ **do**
    $O' \leftarrow$ GetOperations(f');
    if *InferResolvedConstraints*(O_i, O') **then**
      Mark(f');
      $O_i \leftarrow O_i \cup O'$;
      break;
$O_o \leftarrow O_i$;
return O_o

For feedback, the large number of user-level APIs related to file state queries and their layered encapsulation would require extensive modeling. To avoid such large-scale modeling, we adopt a more robust approach by leveraging kernel-level hook techniques to monitor two primary state queries: *File State Queries* [42] and *Directory State Queries* [43], which together constitute the complete set of file state queries. The former encompasses nearly all file state queries, including file names, sizes, time attributes, and permissions. The latter represents the entire set of directory state queries, such as enumerating specific files within a directory. Specifically, we hook into a total of 12 state queries, distributed across the two primary types described above, which together cover 28 distinct file states, with each query mapping to one or more of these file states. The details are outlined in Table 6 in Appendix C. Upon encountering a file state constraint, the program issues a corresponding file state query, which our instrumentation intercepts and observes. This enables us to leverage the observed file state queries as feedback to drive targeted mutations of the mapped file states, thereby significantly reducing the mutation state space and effectively solving the file state constraints. The design eliminates the need for elaborate modeling and optimizes feedback utilization to effectively guide the mutation process.

For mutation, we implemented three distinct mutation operators to efficiently solve file state constraints:

(1) Z3-based Constant Value Injection: For state information involving pattern matching or fuzzy queries, we use Z3 [44] to solve for constant values and inject them into the corresponding states. For instance, when wildcard file names are present, Z3 solves the wildcard to generate specific constants, which are then used for mutations.

(2) Similarity-based Mutation: When state information exhibits similarities, we calculate similarity using edit distance [45] and iteratively alter different components to generate new states. For example, if file name queries include `.log.1` and `.log.2`, we calculate their edit distance and mutate to create similar file names, such as `.log.3` and `.log.0`.

(3) Flip-based Mutation: Inspired by traditional fuzzing techniques, we implement random flips on file states. This operator randomly flips attributes to generate new file states. For instance, a file can be flipped to a directory, or a non-compressed file can be flipped to a compressed one.

For inferring whether constraints have been solved, considering the black-box nature of programs, which prevents directly determining if constraints are resolved, we infer whether file state constraints have been solved after mutation by analyzing two dimensions of file operations: *(1) Operation count*. Checking if the number of operations on the mutated file increases. *(2) Operation types*. Verifying if the types of operations increase. However, relying on a single dimension, like operation count, is insufficient, as an increase (e.g., additional read operations due to a larger file size) may not indicate constraints. By contrast, when both dimensions exhibit increases, it can be inferred that state constraints have been successfully resolved. Subsequently, files that successfully resolve state constraints are marked, and the process is terminated to transition into the exploitation phase.

4.3 Exploitation Phase

4.3.1 File Operation Primitive Graph Build

In this process, `LinkZard` constructs the File Operation Primitive Graph (FOPG) to systematically represent the relationships between all file operations performed by the privileged program. We first introduce the FOPG along with formal definitions of its nodes and edges, followed by a detailed description of the FOPG construction process.

Definition 1 (FOPG). The File Operation Primitive Graph (FOPG) is a directed graph that uses specialized nodes and edges to represent the file operations of privileged programs and their sequences. Each path obtained through the traversal of the FOPG reflects a complete and sequential set of operations performed by the program on a target file and its dependent files (e.g., parent directories).

Definition 2 (FOPG Node). In the FOPG, each node represents a distinct file operation performed by the privileged program. A node is formalized as a tuple:

$$N = \langle O_s, O_d, O_p, R \rangle, \qquad (1)$$

where O_s denotes the operation subject; notably, if the program executes the file operation under an impersonated user context [46], the operation subject is defined as the impersonated user. O_d represents the target file of the operation, O_p specifies the type of operation performed (e.g., write, rename), and R indicates the operation's return value, reflecting its outcome (e.g., success or a system-defined error code [47]).

Definition 3 (FOPG Edge). An FOPG edge $e \in E$ represents a directed connection between two nodes, capturing the sequential relationship between file operations. This sequentiality inherently determines the directionality of edges, which plays a critical role in sink localization and subsequent code wrapping during the exploitation phase.

We construct the FOPG based on the comprehensive sequences of file operations performed by privileged processes during the exploration phase. In this graph, each file operation is formalized as a node, and directed edges are added to reflect the sequential execution of operations. In particular, operations targeting the parent and sibling directories of a node's file are incorporated as its predecessors, as inter-directory dependencies are technically essential for the creation of pseudo-symbolic links during exploitation. This consideration is explicitly integrated into the construction of the FOPG. Meanwhile, pruning strategies are employed to enhance efficiency and focus on essential paths. Specifically, operations targeting strictly protected paths are excluded, and repeated operations on the same file are removed to eliminate redundancy, retaining only the critical operations.

4.3.2 Sink and Constraint Location

In this process, we focus on locating the sinks and identifying the associated constraints within the FOPG. By drawing on the principles used to solve the subgraph isomorphism problem [7], we implement a graph matching approach to identify a corresponding subgraph within the FOPG that matches the sink, which also means `LinkZard` detects an LFVulns vulnerability. Subsequently, we perform graph traversal to determine whether the file state constraints are pre-sink or on-sink.

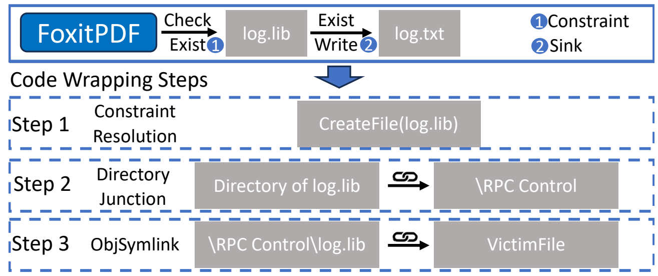
Sink Location. Given the FOPG $G_{\text{op}} = (N_{\text{op}}, E_{\text{op}})$, where N_{op} represents the nodes of the FOPG and E_{op} represents the edges between these nodes, we aim to locate a subgraph $G_{\text{sink}} = (N_{\text{sink}}, E_{\text{sink}})$ that corresponds to the sink. Thus, we attempt to map each node in the sink's subgraph to a node in the FOPG.

$$f : N_{\text{sink}} \to N_{\text{op}} \quad (2)$$
$$\forall (v_i, v_j) \in E_{\text{sink}}, \quad (f(v_i), f(v_j)) \in E_{\text{op}}. \quad (3)$$

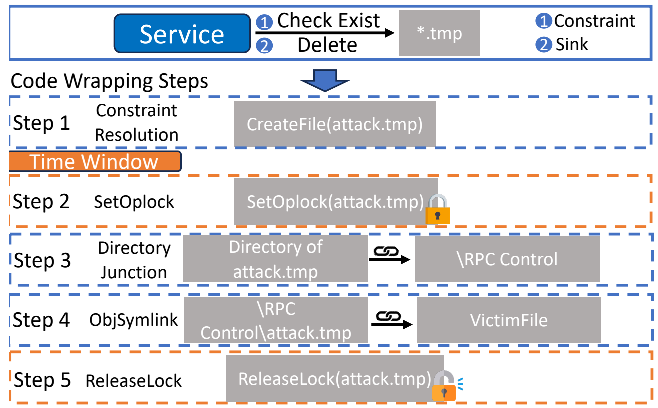
Equation 3 illustrates the core process. Specifically, the node mapping ensures that two nodes correspond to the same API operation, while the edge mapping guarantees that the sequence of operations is consistent.

The sinks introduced in §3.3 are formalized into sink graphs following the same procedure described in §4.3.1, and are subsequently treated as subgraphs for matching within the FOPG. We first begin by performing a depth-first traversal of the FOPG to explore all possible file operation sequences. Subsequently, for each visited node, we attempt to map the first node of the sink to it in the FOPG. If the mapping is successful, we then verify whether the successor nodes and their corresponding edges in the sink can be mapped to their counterparts in the FOPG. If any mismatch occurs, we backtrack to the previous node and attempt a different mapping. This process continues until a valid mapping for the entire sink is found, thereby accurately locating the sink in the FOPG. Notably, we observe that enforcing strict subgraph matching can lead to elevated false negatives in vulnerability detection. This is primarily due to the presence of extraneous file operations interleaved within the execution of the sink operation in privileged programs. To address this, we refine our mapping strategy: once a sink node is successfully matched to a node in the FOPG, its successor nodes are flexibly matched against multiple successor nodes of the corresponding FOPG node. This relaxation mitigates the risk of false negatives caused by intervening irrelevant operations and improves the robustness of sink localization.

Constraint Location. In this process, we perform graph traversal within the FOPG to identify the constraints associated with the sink. Specifically, we recursively explore all the incoming edges to the sink, identifying the set of all predecessor nodes N and their incoming edges. The goal is to determine whether any of these predecessor nodes N has an operation target O_d that corresponds to a file with an existing state constraint, i.e., a file that has been previously marked as f' in *Exploration Phase*. If such a node is found, it indicates the presence of a *pre-sink* constraint. For *on-sink* constraints, we check each node N within the sink. If the operation target O_d of any node corresponds to a file with an existing state constraint, it indicates the presence of an *on-sink* constraint.



(a) Pre-sink Code Wrapping in FoxitService (CVE-2024-38***)



(b) On-sink Code Wrapping in W** Service (CVE-2024-22***)

Figure 5: Illustration of Pre-sink and On-sink Code Wrapping in LFVulns Exploitation by `LinkZard`

4.3.3 State-Specific Code Wrapping

This process is responsible for wrapping state-specific code to achieve the final exploitation of LFVulns. As mentioned in §4.1.2, the effectiveness of this assembly method lies in its focus on constraint types rather than specific file operations, making it applicable to LFVulns across different scenarios. Broadly, this is a two-step process: (1) extracting file state information from the located sink and associated constraints, and (2) applying distinct wrapping strategies for pre-sink and on-sink constraints to fully assemble the exploitation code.

Specifically, for the first step, since the exploitation code for pseudo-symbolic links and opportunistic locks is highly dependent on specific states (e.g., file name, path), we extract the state information from the sink and the files with existing constraints. This includes file names and file paths. The extracted state information for the former primarily serves to wrap the Oplock exploitation code, while the state information for the latter is used for wrapping the pseudo-symbolic link exploitation code. In the second step, we elaborate on the wrapping strategies for pre-sink and on-sink constraints and how the extracted state information is mapped to the exploitation code, ultimately assembling the complete exploitation code for compilation.

❶ For pre-sink constraints, our wrapping strategy involves first assembling the code to generate a file that satisfies the state constraints, primarily by replicating the same file marked during the *exploration phase*. Next, we use the target file's name and path from the sink operation to assemble the pseudo-symbolic link exploitation code. 1) The code to create a directory junction is assembled by using the extracted path to create the junction, pointing to a writable namespace (e.g., `\RPC Control`). 2) The code to create an ObjSymlink is assembled by using the extracted file name to create the symbolic link, pointing to the victim file. Figure 5a illustrates the code wrapping steps performed by `LinkZard` for the exploitation with pre-sink constraints in FoxitPDF.

❷ For on-sink constraints, our wrapping strategy first involves assembling the code to generate a file that satisfies the state constraints. Next, we combine this extracted file name and path to set the Oplock and use the state of the sink operation's target file to create a pseudo-symbolic link. This includes: 1) assembling the code snippet to implement the Oplock for the constrained file, ensuring that the privileged program temporarily halts and resumes once the Oplock is released to complete the subsequent file operations in the sink; and 2) assembling the pseudo-symbolic link exploitation code, similar to the pre-sink strategy, using the state of the sink operation's target file. Figure 5a demonstrates the detailed code wrapping steps performed by `LinkZard` for the exploitation with on-sink constraints in the W** service.

5 Evaluation

5.1 Experimental Setup

Experiments. Our evaluation aims to answer the following research questions:

- RQ1: How does `LinkZard` compare to state-of-the-art tools, i.e., Jerry [6], in terms of its vulnerability detection capabilities for known LFVulns? (in §5.2)
- RQ2: How effective is `LinkZard` in detecting and exploiting unknown LFVulns in real-world programs? (in §5.3)
- RQ3: How do the key processes of `LinkZard` contribute to its performance in detecting and exploiting Link Following vulnerabilities? (in §5.4)

Implementation. We implemented `LinkZard` on Windows, consisting of approximately 5.6k lines of C code and 2.8k lines of Python code. The C code, developed using the Microsoft Windows Driver Kit (WDK) [48], captures privileged file operations and provides detailed feedback on file state information. The Python code utilizes this information to carry out the exploration and exploitation phases. All experiments were conducted on a VMware virtual machine hosted on an HP laptop running Windows 11. The host system is equipped with a 13th Gen Intel Core i7-13700 processor, 32 GB of RAM, and a 1 TB SSD. The virtual machine is configured with 4 CPU cores, 16 GB of memory, and a 100 GB virtual hard drive, also running Windows 11.

Table 1: Comparison between `LinkZard` and `Jerry-Ext` in Known Vulnerability Dataset (RQ1)

Baselines	Detection					Exploitation
	TP	FP	FN	Prec (%)	Recall (%)	Success Rate (%)
`Jerry-Ext`	24	10	18	70.59%	57.14%	/
`LinkZard`	38	0	4	100.00%	90.48%	86.84% (33/38)

Dataset. We evaluate `LinkZard` using two datasets: ① *Known Vulnerability Dataset:* This dataset consists of the 42 successfully reproduced LFVulns described in our empirical study. Table 5 shows the details of the dataset. ② *Unknown Vulnerability Dataset:* This dataset includes 120 highly popular programs and system-critical foundational service programs, with all popular programs having over 50K downloads from Chocolatey [49]. Notably, all selected programs run with elevated privileges, either directly as system services or by relying on privileged components for critical functionality. These programs span a broad spectrum, encompassing widely-used user applications to low-level drivers and hardware-based service programs, highlighting the dataset's comprehensiveness in ensuring representation across all layers of the software stack.

Time Setup. We assign a 20-minute time budget for testing each program, during which `LinkZard` automatically alternates between two phases. Specifically, it transitions from the exploration phase to the exploitation phase upon solving file state constraints. If the exploitation attempt fails, `LinkZard` updates file states and resumes exploration.

5.2 Comparison on Known Vulnerabilities

In this section, we evaluate the effectiveness of `LinkZard` by conducting a comparative analysis against state-of-the-art techniques (i.e., Jerry [6]) using the Known Vulnerability Dataset.

Baseline Setup: Jerry-Ext. We adapted and enhanced Jerry, as it was not originally designed for privileged programs and thus cannot effectively interact with them. To ensure a fair comparison, we extend Jerry by integrating the *IPC Call* component, resulting in *Jerry-Ext*. This extension ensures compatibility with all programs within the Known Vulnerability Dataset, providing comprehensive coverage and improved applicability.

Result Overview. In terms of detection, `LinkZard` significantly outperforms `Jerry-Ext`, successfully detecting 38 vulnerabilities out of 42, identifying 14 more vulnerabilities than `Jerry-Ext`. Additionally, `LinkZard` achieves precision and recall rates of 100.00% and 90.48%, respectively. It is worth noting that the high precision of `LinkZard` stems from our use of sinks, i.e., API file operation sequences, rather than individual file operations, to detect LFVulns. In contrast, the precision and recall rates of `Jerry-Ext` are notably lower, with values of 70.59% and 57.14%. Furthermore, `LinkZard` maintains a high exploitation success rate of 86.84%, whereas `Jerry-Ext`, due to its inability to perform exploitation, has no data available in this regard.

As shown in Table 5 in Appendix B, we provide a detailed breakdown of the performance of `LinkZard` and `Jerry-Ext` on the known vulnerability dataset. While all vulnerabilities detected by `Jerry-Ext` are also identified by `LinkZard`, the latter additionally identifies 14 more vulnerabilities, highlighting its superior detection capability. Furthermore, `LinkZard` successfully exploits 33 of these vulnerabilities, demonstrating its effectiveness not only in detection but also in exploitation. These results underscore `LinkZard`'s high efficiency and robustness in addressing known vulnerabilities, making it valuable for both detection and exploitation tasks.

False Negative Analysis. We conducted a comprehensive analysis of all false negatives. For the 4 false negatives by `LinkZard`, the underlying cause lies in the need to satisfy non-file-state external constraints to trigger the vulnerabilities. For instance, CVE-2020-0668 [50], a local privilege escalation vulnerability in *Windows Service Tracing*, necessitates prior modification of registry values to exploit the issue. Due to the high variability of such external constraints, it becomes impractical for `LinkZard` to accommodate them. In addition to these 4 cases, `Jerry-Ext` exhibited 14 additional false negatives. This limitation stems from its inability to address file-state constraints. An illustrative example is CVE-2023-45253 [51], where the file size must exceed the *maxSizeRollBackups* threshold to trigger an unsafe file move, a constraint that `Jerry-Ext` cannot address.

False Positive Analysis. `Jerry-Ext` has 10 false positives, primarily due to its reliance on single-file operations to determine the presence of vulnerabilities. Notably, all of these false positives occurred when privileged programs performed secure file read operations, which `Jerry-Ext` mistakenly identified as vulnerabilities. In contrast, `LinkZard` leverages sink for vulnerability detection as previously mentioned in §4.3.2, a method that significantly improves precision, as evidenced by its markedly higher accuracy in the experimental results.

Exploitation Result Analysis. For exploitation, `LinkZard` achieves a success rate of 86.84% (33/38) for known vulnerabilities. Considering the complexity of privileged program file operations and the challenges of exploiting LFVulns, we believe this is a significant result. This is particularly impressive given that our exploitation approach focuses on constraint-based exploitation code wrapping rather than individual file operations, making it applicable to LFVulns across various file operation scenarios. Additionally, we analyzed the five failures in exploitation within the detected LFVulns and found that successful exploitation required meeting additional preconditions. For example, CVE-2024-21111 [52] requires a tricky technique to first clear the folder, which `LinkZard` is unable to achieve.

5.3 RQ2: Identifying Unknown Vulnerabilities

To evaluate `LinkZard`'s capability in identifying unknown vulnerabilities, we applied `LinkZard` to the *Unknown Vulnerability Dataset*, which consists of 120 untested programs.

Result Overview. Table 2 presents the real-world vulnerabilities detected and exploited by `LinkZard`. Overall, `LinkZard` successfully identified and exploited 55 zero-day vulnerabilities across 49 programs, with only 5 false positives.

False Positive Analysis. Our analysis of the 5 false positives revealed that 3 of them were caused by the implementation of process-level mitigations in the application, specifically through the use of *Redirection Guard* [53], which protects against LFVulns by preventing unauthorized redirections during file operations. These mitigations allowed `LinkZard` to detect the vulnerabilities as potential threats, but the exploitation attempts failed due to the protective mechanisms in place. The remaining 2 false positives stemmed from custom defenses added to file operations by the application, which specifically check whether a file is in the form of a symbolic link, thereby partially mitigating the vulnerability.

Vulnerability Disclosure. Among the 49 confirmed and fixed vulnerabilities, 15 have been assigned CVE identifiers, while 25 were acknowledged with bug bounty rewards. Notably, a portion of these zero-day vulnerabilities was discovered in high-impact programs with over 5 million downloads, including Tencent Meeting, Enterprise WeChat, Foxit PDF, and Adobe C***. Additionally, some zero-day vulnerabilities were identified in critical infrastructure and foundational services with widespread impact, such as default Windows system services (e.g., Windows Image Acquisition) and Intel's driver-level services (e.g., Intel(R) *** Service). Furthermore, we observed that software developed by prominent vendors, including Microsoft, Apple, Intel, JetBrains, VMware, and Tencent, also exhibited a significant number of LFVulns. We responsibly reported all vulnerabilities to the vendors and maintained continuous communication to ensure that all vulnerabilities with detailed information received the vendors' approval for disclosure.

5.4 RQ3: Ablation Studies

In our ablation study, we systematically evaluated the impact of incrementally removing key processes of `LinkZard` on false positives and false negatives. Specifically, we created two variants by isolating each process from `LinkZard`, and the details of these variants are outlined as follows:

- $\texttt{LinkZard}_{\texttt{NF}}$ (*No Fuzz*): In this variant, we disable only the *File State Fuzz* process while keeping the other processes intact. The aim is to assess the impact of this process on the vulnerability detection capabilities of `LinkZard`, with a particular focus on its influence on the false negative rate.
- $\texttt{LinkZard}_{\texttt{NS}}$ (*No Sink*): In this variant, we isolate sinks and to ensure the usability of `LinkZard`, we adopt Jerry's

Table 2: Part of Real-world LFVulns Detected and Exploited by `LinkZard` in the Unknown Vulnerability Dataset. (RQ2) The abbreviations Dos and LPE in the *Sec. Risk* column represents Denial of Service and Local Privilege Escalation, respectively. The symbol "★" indicates Critical Infrastructure and Foundational Services. Note that these entries do not include specific download metrics.

#	Software Name	Vendor	Download	Sec. Risk	Status
1	WeCom	Tencent	250M	Dos	Fixed
2	Tencent Meeting	Tencent	200M	Dos	Fixed
3	WeChat Input	Tencent	20M	Dos	Fixed
4	Foxit PDF	Foxit Software	9.8M	LPE	CVE-2024-38***
5	Foxit PDF	Foxit Software	9.8M	Dos	Fixed
6	Adobe *** Software	Adobe	7.6M	Dos	Confirmed
7	W*** Software	Elastic	3.2M	LPE	Confirmed
8	Microsoft PC Manager	Microsoft	3.1M	LPE	CVE-2024-49***
9	Microsoft ***	Microsoft	3.1M	Dos	Confirmed
10	iTunes for Windows	Apple	2.7M	LPE	CVE-2024-44***
11	H*** Software	Elastic	2.5M	LPE	Confirmed
12	Microsoft OfficePlus	Microsoft	2.4M	LPE	CVE-2024-38***
13	H*** Software	Elastic	2.4M	LPE	Confirmed
14	M*** Software	Elastic	1.7M	LPE	Confirmed
15	Azure Software 1	Microsoft	1.7M	LPE	CVE-2024-38***
16	Azure Software 2	Microsoft	1.7M	LPE	Fixed
17	F*** Software	Elastic	1.5M	LPE	Confirmed
18	V*** Software	Veeam	1.3M	LPE	Confirmed
19	Sonos Controller S2	Sonos	1.3M	Dos	Confirmed
20	O*** Software	ManageEngine	1.0M	LPE	CVE-2024-98**
21	E*** Software 1	ManageEngine	1.0M	LPE	Fixed
22	E*** Software 2	ManageEngine	1.0M	LPE	Fixed
23	A*** Software	ManageEngine	1.0M	LPE	Fixed
24	P*** Software	Trend Micro	635K	LPE	Reported
25	T*** Software	Trend Micro	594K	Dos	Reported
26	MalwareBytes Adwcleaner	MalwareBytes	521K	LPE	Confirmed
27	TeamCity	JetBrains	483K	LPE	CVE-2024-43***
28	Cato SDP Client	Cato Networks	425K	LPE	Reported
29	Warp VPN	Cloudflare	366K	LPE	Fixed
30	A*** Software	Avast	325K	LPE	CVE-2024-72**
31	P*** Software	PaperCut	324K	Dos	Confirmed
32	Synology BeeDriver	Synology	209K	Dos	CVE-2024-11***
33	Adobe ***	Adobe	203K	LPE	Confirmed
34	Amazon Kinesis Agent	Amazon	201K	LPE	Fixed
35	HP *** Hub	HP	198K	Dos	Confirmed
36	Comodo ***	Comodo	165K	Dos	Reported
37	S*** Software	SonicWALL	157K	Dos	Reported
38	TOTAL SECURITY	G DATA	150K	Dos	Fixed
39	T***	G DATA	150K	LPE	Reported
40	A*** Software	Elastic	148K	LPE	Confirmed
41	W*** Software	Wacom	123K	Dos	Confirmed
42	Splashtop Business Access	Splashtop	108K	Dos	Fixed
43	R*** Software	Rockwell	100K	LPE	Confirmed
44	Azure Monitor Agent	Microsoft	67K	Dos	CVE-2024-38***
45	WatchGuard EPDR	WatchGuard	50K	Dos	CVE-2025-01**
46	Windows WiaRPC Service	Microsoft	★	LPE	CVE-2024-38***
47	Windows WmsRepair Service	Microsoft	★	LPE	CVE-2024-49***
48	*** Device	Realtek	★	Dos	CVE-2024-11***
49	Intel *** Center	Intel	★	Dos	Confirmed
50	*** Device	Realtek	★	LPE	Fixed
51	Windows FSRM	Microsoft	★	Dos	Fixed
52	Windows Backup Service	Microsoft	★	Dos	Fixed
53	Windows *** Manager	Microsoft	★	Dos	Confirmed
54	Windows *** Manager	Microsoft	★	LPE	Confirmed
55	VMware *** Client	VMware	★	LPE	Confirmed

detection strategy by using single-file operations as the sink while keeping the others unchanged. The purpose of this modification is to evaluate the impact of the sink on the accuracy of vulnerability detection.

Table 3: Ablation Study (RQ3) on Two Variants of `LinkZard` in the Known Vulnerability Dataset

Baselines	TP	FP	FN	Prec (%)	Recall (%)
$\texttt{LinkZard}_{\texttt{NF}}$	19	0	23	100.00	45.24
$\texttt{LinkZard}_{\texttt{NS}}$	38	10	4	79.17	90.48
`LinkZard`	38	0	4	100.00	90.48

We conducted experiments on *the Known Vulnerabilities Dataset* using two variants. Table 3 provides detailed results on the vulnerability detection capabilities of the two `LinkZard` variants. $\texttt{LinkZard}_{\texttt{NF}}$ exhibits 23 false negatives, resulting in a recall rate of only 45.25%, highlighting the critical role of the *File State Fuzz* process in vulnerability discovery. Simultaneously, $\texttt{LinkZard}_{\texttt{NS}}$ experiences a substantial increase in false positives, with precision dropping by approximately 79.17%, highlighting the importance of the sinks in ensuring detection accuracy. Through this ablation study, we validate that the different processes of `LinkZard` play a crucial role in maintaining the effectiveness of vulnerability detection.

6 Case Studies

In this section, we showcase two LFVulns to demonstrate `LinkZard`'s effectiveness: one identified in a highly popular application and another uncovered within the foundational infrastructure of prominent commercial software.

Case A: Microsoft OfficePlus (Downloaded by Over 2.4M Users). Microsoft OfficePlus [54] is a commercial subscription application introduced within the Office. The application registers a privileged service named `OfficePlusService`. When interacted via IPC, the service queries the size of the `MSOfficePLUSService.log` file located in the program directory (`C:\ProgramData\Microsoft OfficePLUS`). If the file size exceeds a specified threshold, the service creates a new file, `MSOfficePLUSService1.log`, and appends subsequent content to it. This process continues until `MSOfficePLUSService10.log` is created, after which it deletes the original `MSOfficePLUSService.log`. The deletion operation contains a sink, which is detected by `LinkZard` as an LFVuln. Using `LinkZard`'s file state mutators, specifically targeting file size and name similarity, the file operations were successfully explored. Through code wrapping, `LinkZard` successfully exploits this LFVuln to achieve arbitrary file deletion, ultimately enabling local privilege escalation. Given the significant potential impact of this vulnerability, we promptly reported it to MSRC [55]. As a result, the issue was assigned a CVE (CVE-2024-38***) and was rewarded with a vulnerability bounty.

Case B: Commercial Software infrastructure Infrastructure M*** (anonymized for ethical reasons), a leading IT operations management software provider, is reported to support over 60% of Fortune 500 companies in managing IT infrastructure, data centers, business systems, and security. During our analysis, we identified a vulnerability in a widely used WAF-related infrastructure component integrated across multiple M*** commercial applications. Specifically, the vulnerability resides in the directory `C:\Windows\Temp\waf_fileupload`, where the infrastructure periodically checks for JSP files containing malicious content and performs insecure deletion if such files are detected. Using `LinkZard`, we leveraged directory query feedback to perform file state fuzzing, successfully triggering the vulnerability and enabling local privilege escalation. Upon reporting the issue to the vendor, we learned that over 10 internal commercial applications reused this vulnerable infrastructure, amplifying the severity of the threat. In response, the vendor promptly patched the vulnerability. This vulnerability was assigned CVE-2024-9***, and we received an official acknowledgment from the vendor.

7 Discussion and Limitations

Adaptability. Our prototype for detecting and exploiting LFVulns was specifically designed for privileged programs within the Windows system. The primary motivation for this choice lies in the fact that Windows possesses the largest user base among desktop operating systems [56] and exhibits the most complex mechanisms for file operations and symbolic links. Since LFVulns arise from the lack of proper validation for symbolic links by developers, such vulnerabilities are prevalent across all system platforms that support symbolic links, including Unix-like [57] systems (Linux and Mac). The core ideas of `LinkZard` can be extended to other operating systems with a simpler implementation than on Windows.

Mitigation. Currently, there are two main mitigation approaches for LFVulns: (1) Redirection Guard [53], introduced by Microsoft in Windows, mitigates LFVulns at the process level by preventing privileged programs from following insecure symbolic links. However, this measure is only applicable to Windows 11 22H2 and later, making it incompatible with applications that prioritize version compatibility. (2) Strict Access Control [58], the most commonly used mitigation approach due to its simplicity. Developers should enforce strict access control policies on directories and files involved in LFVulns to prevent arbitrary file manipulation by attackers.

Limitations. Our implementation of feedback-driven fuzzing focuses exclusively on file states, as we observed that file state constraints are the primary obstacle to exploring file operations. However, an analysis of FN (in §5.2) revealed that environment variables and registry values can also impact exploration. Despite this, we did not include fuzzing for these factors due to the disproportionate cost-to-benefit ratio, making their exclusion an acceptable trade-off for `LinkZard`.

8 Related Work

File-related Vulnerability Detection. A significant body of research has focused on detecting vulnerabilities related to file access control [6, 59–65], broadly categorized into static and dynamic approaches. In static detection techniques, [59] [60] leverage static program analysis and access control policy analysis to identify file system vulnerabilities. Shaikh et al. [63] proposed a decision tree-based anomaly detection algorithm to identify inconsistencies in access control policies. Dynamic detection methods [6] [65], employ monitors to track file system events during a software's lifecycle, identifying file operations targeting weakly permissioned files. However, in the context of LFVulns, these approaches do not address the challenges posed by file state constraints in vulnerability detection and exploitation, limiting the applicability of both static and dynamic methods for the effective detection and exploitation of LFVulns.

Windows-related Vulnerability. In recent years, extensive research has focused on the detection and exploitation of vulnerabilities in Windows systems [66–72]. Choi et al. [69, 70] combined empirical studies of Windows API fuzzing with automated analysis of function dependencies to implement automated API fuzzing techniques. [71] [67] conducted fuzzing on Windows applications to detect vulnerabilities, including buffer overflows and denial-of-service attacks. For exploitation, Jung et al. [71] utilized CPU-level operating system instrumentation to exploit race condition vulnerabilities in the Windows kernel, while Gu et al. [72] analyzed thread-unsafe interfaces in COM objects to exploit data race vulnerabilities. Despite these advancements, limited attention has been given to the security challenges arising from dynamic file operations. This gap strongly motivates our work on revealing, detecting, and exploiting LFVulns in the Windows system.

9 Conclusion

This paper presents the first systematic study of LFVulns. Inspired by findings from empirical research, we developed `LinkZard`, the first framework for automatically detecting and exploiting Link Following vulnerabilities in Windows file operations. To date, `LinkZard` has identified and successfully exploited 55 zero-day vulnerabilities across 120 real-world applications. We responsibly disclosed all vulnerabilities to the respective vendors, resulting in 49 confirmed and fixed cases, with 15 CVE identifiers assigned and bug bounties rewarded. We believe `LinkZard` could provide actionable guidance for the prevention and remediation of LFVulns.

Acknowledgement

We thank the anonymous reviewers for the helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (62172105, U2436207). Yuan Zhang and Min Yang are the corresponding authors. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - Fudan University 21TQ1400100 (21TQ012). Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

10 Ethics Considerations

This work poses no ethical concerns. All testing activities were conducted within our locally set-up offline environment, ensuring that there was no interaction with or impact on any real-world systems or user data. We have proactively reported all vulnerabilities we discovered and assisted developers in fixing these vulnerabilities. As a result, 15 CVE identifiers have been assigned as a confirmation for our efforts.

11 Open Science

In alignment with the open science policy, we are committed to fully following the conference's artifact evaluation guidelines. Upon the acceptance of our paper, we will publicly release the source code of LinkZard[1], along with the datasets and baselines used for evaluation in our research. This initiative aims to enhance the reproducibility and replicability of scientific findings, ensuring that our work can be verified and built upon by other researchers in the field.

[1] https://doi.org/10.5281/zenodo.15617437

References

[1] Microsoft. Symbolic links. https://learn.microsoft.com/en-us/windows/win32/fileio/symbolic-links, 2024.

[2] Wikipidia. Shortcut. https://en.wikipedia.org/wiki/Shortcut_(computing)#Microsoft_Windows.

[3] Microsoft. Hard links and junctions. https://learn.microsoft.com/en-us/windows/win32/fileio/hard-links-and-junctions.

[4] Zero Day Initiative. Breaking barriers and assumptions: Techniques for privilege escalation on windows (part 1). https://www.zerodayinitiative.com/blog/2024/7/29/breaking-barriers-and-assumptions-techniques-for-privilege-escalation-on-windows-part-1, 2024. Accessed: 2024-07-29.

[5] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice*. Pearson, 4th edition edition, 2017. Discusses Denial-of-Service attacks and their implications.

[6] 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 of the Network and Distributed System Security Symposium*, 2024.

[7] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[8] OWASP Foundation. Path traversal. https://owasp.org/www-community/attacks/Path_Traversal, 2024.

[9] Inter-process communication. https://learn.microsoft.com/zh-cn/windows/win32/ipc/interprocess-communications.

[10] CrowdStrike. Dll side-loading: How to combat threat actor evasion techniques. https://www.crowdstrike.com/en-us/blog/dll-side-loading-how-to-combat-threat-actor-evasion-techniques/, 2024.

[11] Microsoft. Rollback installation (windows installer). https://learn.microsoft.com/zh-cn/windows/win32/msi/rollback-installation, 2024.

[12] Create symbolic links (windows vista). https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-vista/cc766301(v=ws.10)?redirectedfrom=MSDN#create-symbolic-links.

[13] Windows kernel mode object manager. https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-object-manager.

[14] Opportunistic locks. https://learn.microsoft.com/en-us/windows/win32/fileio/opportunistic-locks.

[15] Hex-Rays. Hex-Rays IDA Pro. https://www.hex-rays.com/, 2020.

[16] MITRE Corporation. Common weakness enumeration (cwe). https://cwe.mitre.org/, 2024.

[17] MITRE Corporation. Common vulnerabilities and exposures (cve). https://cve.mitre.org/, 2024.

[18] National Institute of Standards and Technology (NIST). National Vulnerability Database. https://nvd.nist.gov/, 2024.

[19] Cwe-59: Link following error. https://cwe.mitre.org/data/definitions/59.html.

[20] Cwe-60: Unix path link problems. https://cwe.mitre.org/data/definitions/60.html.

[21] Cwe-61: Unix symbolic link (symlink) following. https://cwe.mitre.org/data/definitions/61.html.

[22] Cwe-62: Unix hard link. https://cwe.mitre.org/data/definitions/62.html.

[23] Cwe-63: Windows path link problems. https://cwe.mitre.org/data/definitions/63.html.

[24] Cwe-64: Improper handling of symbolic links in windows. https://cwe.mitre.org/data/definitions/64.html.

[25] Github. https://github.com, 2024.

[26] Hackerone. https://hackerone.com, 2024.

[27] Exploit database: Exploits for penetration testers and vulnerability researchers. https://www.exploit-db.com/exploits/example, 2024.

[28] Iván Arce. The weakest link revisited [information security]. *IEEE Security & Privacy*, 1(2):72–76, 2003.

[29] Microsoft. Windows api. https://learn.microsoft.com/zh-cn/windows/win32/apiindex/windows-api-list, 2024.

[30] Access control lists (acls). https://learn.microsoft.com/en-us/windows/win32/secauthz/access-control-lists.

[31] Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. Ferry:{State-Aware} symbolic execution for exploring {State-Dependent} program paths. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4365–4382, 2022.

[32] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. {StateFuzz}: System {Call-Based}{State-Aware} linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3273–3289, 2022.

[33] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, 2022.

[34] Wikipedia. symbolic link. https://en.wikipedia.org/wiki/Magic_number_(programming)#In_files.

[35] Cristian Cadar and Patrice Godefroid. Symbolic execution for software testing: Three decades later. In *Communications of the ACM*, pages 82–90, 2013.

[36] Babak Yadegari, Brian Johannesmeyer, Jason Whitaker, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 674–691, 2015.

[37] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[38] Peach Fuzzer. Peach fuzzing platform. https://peachtech.gitlab.io/peach-fuzzer-community/.

[39] Microsoft. Services - windows server. https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/sc-config.

[40] Microsoft. Remote procedure call (rpc) - windows server. https://learn.microsoft.com/en-us/windows/win32/rpc/remote-procedure-calls-overview.

[41] Microsoft. Microsoft interface definition language (midl). https://learn.microsoft.com/en-us/windows/win32/midl/midl-start-page, 2025.

[42] Microsoft. Irp_mj_query_information. https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-query-information, 2024.

[43] Microsoft. Irp_mj_directory_control. https://learn.microsoft.com/en-us/previous-versions/windows/drivers/ifs/irp-mj-directory-control, 2024.

[44] Z3Prover. The z3 theorem prover. https://github.com/Z3Prover/z3, 2024.

[45] Wikipedia. Edit distance. https://en.wikipedia.org/wiki/Edit_distance, 2024.

[46] Impersonation - windows process security. https://learn.microsoft.com/en-us/windows/win32/com/impersonation, 2024.

[47] Microsoft. System error codes (windows). https://learn.microsoft.com/en-us/windows/win32/debug/system-error-codes, 2023.

[48] Microsoft. *Windows Driver Kit (WDK)*, 2023.

[49] Chocolatey Software, Inc. Chocolatey - the package manager for windows. https://chocolatey.org/.

[50] CVE-2020-0668. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0668.

[51] CVE-2023-45253. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-45253.

[52] Cve-2024-21111. https://www.cve.org/CVERecord?id=CVE-2024-21111.

[53] Policy csp - configureredirectionguardpolicy. https://learn.microsoft.com/en-us/windows/client-management/mdm/policy-csp-printers#configureredirectionguardpolicy.

[54] Microsoft. Microsoft officeplus. https://www.officeplus.cn/, 2024.

[55] Microsoft. Microsoft security response center (msrc). https://msrc.microsoft.com/, 2024.

[56] Wikipedia. Usage share of operating systems. https://en.wikipedia.org/wiki/Usage_share_of_operating_systems, 2024.

[57] Wikipedia. Unix like. https://en.wikipedia.org/wiki/Unix-like, 2024.

[58] Microsoft. What is access control. https://www.microsoft.com/en-us/security/business/security-101/what-is-access-control.

[59] Jiadong Lu, Fangming Gu, Yiqi Wang, Jiahui Chen, Zhiniang Peng, and Sheng Wen. Static detection of file access control vulnerabilities on windows system. *Concurrency and Computation: Practice and Experience*, 34(16):e6004, 2022.

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

[61] Miao Cai, Hao Huang, and Jian Huang. Understanding security vulnerabilities in file systems. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 8–15, 2019.

[62] Jinghan Sun, Shaobo Li, Jun Xu, and Jian Huang. The security war in file systems: An empirical study from a vulnerability-centric perspective. *ACM Transactions on Storage*, 19(4):1–26, 2023.

[63] Riaz Ahmed Shaikh, Kamel Adi, and Luigi Logrippo. A data classification method for inconsistency and incompleteness detection in access control policy sets. *International Journal of Information Security*, 16:91–113, 2017.

[64] Lujo Bauer, Scott Garriss, and Michael K Reiter. Detecting and resolving policy misconfigurations in access-control systems. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):1–28, 2011.

[65] Can Huang, Xinhui Han, and Guorui Yu. Lpet–mining ms-windows software privilege escalation vulnerabilities by monitoring interactive behavior. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2089–2091, 2020.

[66] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[67] Ivan Fratric. Winafl: A fork of afl for windows. https://github.com/googleprojectzero/winafl, 2016.

[68] DoHoon Lee, YoungHan Choi, and Jae-Cheol Ryou. Api fuzz testing for security of libraries in windows systems: From faults to vulnerabilites. In *2008 Third International Conference on Convergence and Hybrid Information Technology*, volume 2, pages 578–584. IEEE, 2008.

[69] YoungHan Choi, HyoungChun Kim, and DoHoon Lee. An empirical study for security of windows dll files using automated api fuzz testing. In *2008 10th International Conference on Advanced Communication Technology*, volume 2, pages 1473–1475. IEEE, 2008.

[70] YoungHan Choi, HyoungChun Kim, HyungGeun Oh, and Dohoon Lee. Call-flow aware api fuzz testing for security of windows systems. In *2008 International Conference on Computational Sciences and Its Applications*, pages 19–25. IEEE, 2008.

[71] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[72] Fangming Gu, Qingli Guo, Lian Li, Zhiniang Peng, Wei Lin, Xiaobo Yang, and Xiaorui Gong. {COMRace}: detecting data race vulnerabilities in {COM} objects. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3019–3036, 2022.

Table 4: File Operation API Sequences in Four Sink Types

Sink Types	API Sequences
Unsafe Creation, Write, and Overwriting	• **Sequence 1:** `CreateFile(OpenResult: Created)` • **Sequence 2:** `CreateFile(Disposition: Opened) → WriteFile` • **Sequence 3:** `CreateFile(Disposition: OverwriteIf) → WriteFile` • **Sequence 4:** `CreateFile(Disposition: Opened) → QueryBasicInformationFile → CreateFile(OpenResult: Created)`
Unsafe Coping and Moving	• **Sequence 1:** `CreateFile(Desired Access: DELETE) → QueryBasicInformationFile → SetRenameInformationFile` • **Sequence 2:** `CreateFile[source file] → QueryBasicInformationFile → CreateFile[target file] → SetBasicInformationFile`
Unsafe Access Control Configuration	• **Sequence 1:** `CreateFile(Desired Access: Write DAC) → QuerySecurityFile → SetSecurityFile` • **Sequence 2:** `CreateFile(Desired Access: Read Control) → QuerySecurityFile → CreateFile(Desired Access: Write DAC) → SetSecurityFile`
Unsafe Deletion	• **Sequence 1:** `CreateFile(Desired Access: DELETE) → DeleteFile` • **Sequence 2:** `QueryDirectory → CreateFile(Desired Access: DELETE) → DeleteFile`

A Sink Details

Table 4 presents the file operation API sequences corresponding to different types of sinks identified in the ground truth.

Table 5: The Detailed Evaluation Results on Known Vulnerabilities.

#	CVE ID	Affected Software	Sink Type	Detection		Exploitation (`LinkZard`)
				Jerry-Ext	`LinkZard`	
1	CVE-2024-8405	PaperCut NG/MF	FC	✓	✓	✓
2	CVE-2024-7235	AVG AntiVirus Free	FC	–	✓	✓
3	CVE-2024-7234	AVG AntiVirus Free	FD	–	✓	✓
4	CVE-2024-7232	Avast Free Antivirus	FD	–	✓	✓
5	CVE-2024-7231	Avast Cleanup Premium	FD	–	✓	✓
6	CVE-2024-7228	Avast Free Antivirus	FC	–	✓	✓
7	CVE-2024-45316	SonicWall Connect Tunnel	FD	✓	✓	✓
8	CVE-2024-45315	SonicWall Connect Tunnel	FC	✓	✓	✓
9	CVE-2024-35204	Veritas System Recovery	FC	✓	✓	✓
10	CVE-2024-3037	PaperCut NG/MF	FD	✓	✓	✓
11	CVE-2024-28916	Xbox Gaming Service	FM	✓	✓	–
12	CVE-2024-27460	Poly Plantronics Hub	FD	✓	✓	✓
13	CVE-2024-21111	VirtualBox	FD	–	✓	–
14	CVE-2024-20656	Visual Studio	ACC	–	✓	–
15	CVE-2023-50915	GOG Galaxy	FC	✓	✓	✓
16	CVE-2023-50917	Intel Driver & Support Assistant	FC	✓	✓	✓
17	CVE-2023-42099	Intel Driver & Support Assistant	FD	–	✓	✓
18	CVE-2023-36874	Windows Error Reporting Service	FD	–	✓	✓
19	CVE-2023-35342	Windows Image Acquisition Service	ACC	✓	✓	✓
20	CVE-2023-32470	Dell Digital Delivery	FC	✓	✓	✓
21	CVE-2023-29343	SysInternals Sysmon for Windows	FD	–	–	–
22	CVE-2023-28892	Malwarebytes	FD	✓	✓	✓
23	CVE-2023-28869	NCP Secure Enterprise Client	FC	✓	✓	✓
24	CVE-2023-28868	NCP Secure Enterprise Client	FD	✓	✓	✓
25	CVE-2023-21752	Windows Backup Service	FD	–	✓	✓
26	CVE-2023-20178	Cisco AnyConnect Secure Mobility Client	FD	–	–	–
27	CVE-2022-45697	Razer Central	FD	–	✓	✓
28	CVE-2022-43293	Wacom Driver	FC	✓	✓	✓
29	CVE-2022-38699	Armoury Crate	FC	✓	✓	✓
30	CVE-2022-38604	Wacom Driver	FD	✓	✓	✓
31	CVE-2022-32450	AnyDesk	FC	–	–	–
32	CVE-2022-28225	Yandex Browser	FM	✓	✓	✓
33	CVE-2022-22718	Windows Print Spooler Service	FC	✓	✓	✓
34	CVE-2022-22262	ROG Live Service	FD	✓	✓	✓
35	CVE-2022-21999	Windows Print Spooler Service	FC	✓	✓	✓
36	CVE-2021-25261	Yandex Browser	FM	✓	✓	✓
37	CVE-2020-9682	Adobe Creative Cloud	FC	✓	✓	✓
38	CVE-2020-15401	IOBit Malware Fighter Pro	FD	–	✓	✓
39	CVE-2020-14990	IOBit Advanced SystemCare	FD	–	✓	✓
40	CVE-2020-0668	Windows Service Tracing	FM	–	–	–
41	CVE-2019-19248	Electronic Arts Origin	ACC	–	✓	✓
42	CVE-2019-13382	SnagIT	FM	✓	✓	✓

B The Detailed Evaluation Results on Known Vulnerabilities (RQ2)

Table 5 break down the evaluation result of RQ2. The abbreviations **FC**, **FM**, **FD**, and **ACC** represent Unsafe Creation, Write, and Overwriting, Unsafe Moving and Copying, Unsafe Deletion, and Unsafe Access Control Configuration, respectively.

Table 6: Mapping of File State Queries to Corresponding File States

Class	**File States**	**Class**	**File States**
FileNameInformation	File Name	FileBasicInformation	Creation Time
	File Name Length		Modification Time
FileDirectoryInformation	Is File		Access Time
	Is Directory		Last Write Time
	The Directory Size		Is Hidden File
FileEndOfFileInformation	File Size		Is Archive File
FileReparsePointInformation	Is Symlink		Is Temporary File
	Is Mount Point		File Content
	Is Appexeclink		
FileHardLinkInformation	Hard Link File		
FileCompressionInformation	Compressed File Size	FileSecurityInformation	File DACL
	Compressed File Format		File Owner and Group
FileStreamInformation	Alternate Data Stream	FileStandardInformation	Number of Links
	Data Stream Name		Data Stream Attributes
FileAlternateNameInformation	Short File Name	FileAlignmentInformation	File Alignment

C Feedback Metrics

Table 6 provides a detailed mapping between file state queries used as feedback and their corresponding file states.