Check for updates

PDF Download
3719027.3765217.pdf
16 December 2025
Total Citations: 0
Total Downloads: 676

Latest updates: https://dl.acm.org/doi/10.1145/3719027.3765217

RESEARCH-ARTICLE

Windows plays Jenga: Uncovering Design Weaknesses in Windows File System Security

DONG-UK KIM, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

JUN-YOUNG PARK, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

SANGHAK OH, Sungkyunkwan University, Seoul, South Korea

HYOUNGSCHICK KIM, Sungkyunkwan University, Seoul, South Korea

INSU YUN, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

Open Access Support provided by:

Sungkyunkwan University

Korea Advanced Institute of Science and Technology

Published: 19 November 2025

Citation in BibTeX format

CCS '25: ACM SIGSAC Conference on Computer and Communications Security
October 13 - 17, 2025
Taipei, Taiwan

Conference Sponsors:
SIGSAC

CCS '25: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (November 2025)
https://doi.org/10.1145/3719027.3765217
ISBN: 9798400715259

Windows plays Jenga: Uncovering Design Weaknesses in Windows File System Security

Dong-uk Kim*
justlikebono@kaist.ac.kr
KAIST
Daejeon, Republic of Korea

Junyoung Park*
parkjuny@kaist.ac.kr
KAIST
Daejeon, Republic of Korea

Sanghak Oh
sanghak@skku.edu
Sungkyunkwan University
Seoul, Republic of Korea

Hyoungshick Kim†
hyoung@skku.edu
Sungkyunkwan University
Seoul, Republic of Korea

Insu Yun†
insuyun@kaist.ac.kr
KAIST
Daejeon, Republic of Korea

Abstract

File systems are essential components of modern operating systems, with Windows being one of the most dominant platforms. Recently, a series of attacks have exploited the Windows file system to trigger serious security threats such as privilege escalation. Over the past several years, dozens of such attacks have been reported and even exploited in the wild. However, Microsoft has consistently addressed these issues with targeted patches rather than fundamental redesigns — resembling a precarious game of Jenga where security measures are stacked upon an unstable foundation.

In this paper, we present a five-step comprehensive analysis of the Windows file system's design weaknesses. First, we analyze how Windows differs from another operating system, Linux. Second, we investigated how these discrepancies lead to security vulnerabilities in real-world applications and identified 13 high-impact vulnerabilities, including 11 previously unknown ones. Third, we show that current compatibility layers in modern programming languages fail to handle these discrepancies properly. Specifically, we examined compatibility layers in six programming languages and found 27 non-compliant and 9 inconsistencies, rendering these layers unreliable. Fourth, through a user study involving 21 experienced developers, we found that most were unfamiliar with OS-level file system discrepancies and rarely implemented appropriate mitigations. Finally, we analyze existing countermeasures and discuss their limitations. Our findings reveal critical yet largely obscured security risks resulting from design flaws in the Windows file system. Furthermore, we suggest that Microsoft rethink its strategy and address these fundamental weaknesses.

CCS Concepts

• **Security and privacy → Operating systems security**.

*Equal Contribution
†Corresponding Authors

This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '25, Taipei
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765217

Keywords

Windows, Operating system, File system, Design weakness

ACM Reference Format:
Dong-uk Kim, Junyoung Park, Sanghak Oh, Hyoungshick Kim, and Insu Yun. 2025. Windows plays Jenga: Uncovering Design Weaknesses in Windows File System Security. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765217

1 Introduction

File systems are essential components of modern operating systems, providing a storage interface to users and applications. They not only manage the organization, but also operations, access control, and concurrency control of files and directories. While file systems across different operating systems share common functionalities, their implementations can vary significantly, leading to subtle yet critical discrepancies in their behaviors. These discrepancies become particularly concerning when they impact security-relevant behaviors, which can potentially lead to security vulnerabilities that can be exploited by attackers.

Windows, as one of the most widely deployed operating systems globally today, has been the target of numerous attacks exploiting its file system designs. Over the past several years, dozens of these attacks have been reported and even exploited in the wild, resulting in serious security threats including privilege escalation [65]. Despite the severity and frequency of these issues, Microsoft has consistently addressed them with targeted patches rather than addressing the fundamental design weaknesses [11, 35, 72]. A notable case is CVE-2020-16902, where Microsoft issued four patches for the same vulnerability due to initial fixes being incomplete, allowing attackers to bypass them [10]. Their strategy resembles a game of Jenga, where security measures are continuously stacked upon an unstable foundation. While this strategy may temporarily suppress the issues, it is not a sustainable solution for recurring security threats. Moreover, previous studies such as Jerry [82] focused on finding security vulnerabilities themselves, but not on the fundamental causes of these vulnerabilities. These studies provide limited insight into why such vulnerabilities frequently occur and how to prevent them from recurring.

In this paper, we present a comprehensive five-step analysis of the Windows file system, uncovering its design weaknesses that can lead to security vulnerabilities. First, we analyze how the Windows file system fundamentally differs from Linux, another major operating system that shares similar file system functionalities (§3). Through the comparison, we identify four critical discrepancies. Among these, we uncover two discrepancies, file locks and permission inversion, that have not been previously known to cause security vulnerabilities. All four discrepancies represent fundamental design differences that contribute to the recurring security vulnerabilities in Windows.

Second, we investigate how these discrepancies lead to security vulnerabilities in real-world Windows applications (§4). We identify 13 high-impact vulnerabilities that are caused by the discrepancies, including 11 previously unknown ones. Notably, these vulnerabilities are found in major vendors, such as Google, Mozilla, NVIDIA, Samsung, and Oracle. These vulnerabilities can be exploited to achieve privilege escalation, block antivirus scanning, or even escape from sandboxed environments.

Third, we show that current compatibility layers in modern programming languages are insufficient to handle these discrepancies correctly, although they are widely used for writing OS-agnostic code (§5). In particular, we identify 27 non-compliant and 9 inconsistent behaviors in handling file links across six programming languages, making these layers unreliable. Moreover, we find that no compatibility layers are available for file locks and file permissions, leaving developers without cross-platform abstractions for these critical security-relevant features.

Fourth, through a user study, we found that developers are largely unaware of the discrepancies between Windows and Linux file systems and rarely implement mitigations against file system security vulnerabilities (§6). Our study of 21 professional software developers revealed a universal lack of knowledge about these OS-level discrepancies. Even participants with security backgrounds failed to apply essential countermeasures such as checking link existence, proper share mode locking, or explicit permission settings. This knowledge gap is particularly concerning as developers must create their own protections when compatibility layers fail.

Finally, we analyze existing countermeasures and discuss their limitations (§7). These countermeasures, which are rare and not widely adopted, are not effective in preventing these security vulnerabilities. Moreover, we find that even experienced developers at major software vendors struggle to implement effective solutions.

Our findings uncover fundamental design weaknesses in the Windows file system that can impact security. Microsoft has consistently addressed these weaknesses with targeted patches, prioritizing backward compatibility and convenience over security principles. Unfortunately, these weaknesses are not merely implementation bugs that can be patched away but rather systemic weaknesses that require comprehensive redesign to be fully addressed. Without such fundamental changes, applications will continue to be vulnerable to file-related attacks, regardless of how many individual patches developers apply. Therefore, we suggest that Microsoft rethink its strategy and address these fundamental weaknesses.

Our contributions are as follows:

- We examine the discrepancies between Windows and Linux file systems that contribute to recent security vulnerabilities. We also identify two new discrepancies that have not been previously reported (i.e., file locks and permission inversion).
- We identify 13 high-impact vulnerabilities that arise from these discrepancies.
- We demonstrate that either existing compatibility layers or the common practices of developers fail to address the discrepancies.
- We analyze existing countermeasures and demonstrate their challenges in mitigating these vulnerabilities.

Due to the space limit, we release our Appendix online [76].

2 Overview

In this section, we introduce our threat model and research questions. We also explain the reasons for comparing the Windows file system to Linux, and clarify the scope of our contribution.

2.1 Threat Model

We consider a privilege escalation threat model [9, 54, 63], where the adversary initially holds limited privileges (e.g., an unprivileged user account or sandboxed process) and aims to escalate to higher-privilege contexts (e.g., Administrator, SYSTEM, or kernel mode) or interfere with privileged processes (e.g., disrupting system updates or antivirus operations) to enable further attacks. To reflect scenario-specific differences, we explicitly provide the context and impact of each attack at the start of its discussion (see §4).

2.2 Research Questions

In this paper, we study the design weaknesses of the Windows file system. To demonstrate them, we reveal the security-relevant discrepancies between Windows and Linux file systems, and how these discrepancies can lead to security issues. To achieve these goals, we answer the following research questions:

RQ1. How are Windows and Linux file systems designed differently? In §3, we examine the file system designs of Windows and Linux and identify four key discrepancies. Among them, two discrepancies involving file locks and permission inversion are newly identified and have not been previously reported as causes of security vulnerabilities.

RQ2. How serious are these discrepancies? In §4, we examine the real-world vulnerabilities that are caused by these discrepancies. In summary, we discovered 13 high-impact vulnerabilities affecting 11 major vendors, of which 11 can be exploited to escalate privileges, one to block antivirus scanning, and one to escape sandboxes.

RQ3. How do the existing compatibility layers handle these discrepancies? In §5, we examine the compatibility layers in 6 programming languages such as C and C++. Unfortunately, we found that the compatibility layers are not working well, as we found 27 non-compliant behaviors and 9 inconsistent behaviors.

RQ4. How aware are developers of these discrepancies, and how do they respond? In §6, we conduct a user study to assess developer awareness of these discrepancies and examine the mitigation strategies they adopt in practice. Our results show that developers are largely unaware of the discrepancies, and they rarely take effective countermeasures.

RQ5 What would be the countermeasures to mitigate these vulnerabilities? In §7, we examine the existing countermeasures to mitigate these discrepancies. As a result, we find four countermeasures, each of which unfortunately has limitations. Furthermore, we find that even experienced developers at major software vendors struggle to implement effective countermeasures, highlighting the need for Microsoft to address these fundamental design weaknesses in the Windows file system.

2.3 Purpose of the Linux Comparison

In this study, we compare the Windows and Linux file systems for two main purposes.

For intuitive explanation. We use Linux as a reference point to help readers better understand the design of the Windows file system. Since Linux is more familiar to many developers [70] — owing to its open-source nature, wide academic adoption, and extensive documentation — this comparison provides an intuitive framework for explaining Windows-specific behaviors. Furthermore, understanding Linux behavior is essential for our later analysis of compatibility layers (§5).

For discovering Windows-specific behaviors. Comparing Windows with Linux also helped us identify Windows-specific design aspects in a principled manner. Given the substantial body of research on Linux file systems, this comparative approach allowed us to filter out well-understood common features (e.g., [50, 59]) and focus on distinctive Windows behaviors, some of which led to the discovery of design weaknesses in the Windows file system.

Clarification on the scope of the comparison. We emphasize that this comparison does not imply Linux is flawless, nor that all differences between Linux and Windows result in security vulnerabilities. As we show in later sections, Linux also exhibits limitations — for example, advisory file locks (e.g., flock) have restrictions with non-voluntary processes (§3.2), and its C++ standard library implementation has specification violations (§5). Moreover, we identified several discrepancies between the two systems that are unrelated to security. Due to space constraints, we focus in this paper only on those discrepancies with security implications.

2.4 Our Contribution

Although some of the discrepancies we discuss in this paper — link traversal [4, 7, 11, 12, 16–18, 31, 35, 72, 75, 81, 83] (§3.1) and permission inheritance [82] (§3.3.1) overlap with previous works, our study makes two key novel contributions.

First, two out of four identified discrepancies (i.e., file locks and permission inversion) are new and have not been reported to cause security vulnerabilities. Regarding file locks, while we acknowledge that using oplocks in exploitation is popularized by James Forshaw [4, 7, 12, 31, 35, 81, 83], file locks have not been identified as direct causes of vulnerabilities (e.g., DoS). Also, to the best of our knowledge, there is no public material that correlates security vulnerabilities to file lock design choices (i.e., advisory vs. mandatory locks), share mode locks, and permission inversion.

Second, our study provides a deeper analysis of previously known issues, including link traversal and permission inheritance. In particular, our study presents real-world case studies that illustrate how these known problems can compound with other discrepancies to create new security risks (§4). We further examine whether existing compatibility layers effectively mitigate such issues (§5), study how well developers recognize these discrepancies (§6), and analyze the real-world deployment of mitigation mechanisms (§7). We believe that our study can offer insights into the design weaknesses of the Windows file system, making it a valuable resource for both researchers and practitioners.

Table 1: Types of links in Windows. Each column describes the restrictions of the links. JDSymlink is the most preferred since it has no specific restrictions.

Link type	Privilege	References	Link location
NTFS symbolic link	Administrator	**None**	**None**
NTFS junction	**None**	Directory	**None**
DosDevice	**None**	**None**	NT namespace
JDSymlink	**None**	**None**	**None**

3 Discrepancies between Windows and Linux File Systems

In this section, we answer **RQ1** by investigating the discrepancies between Windows and Linux file systems. We identify four key discrepancies in three aspects: one in file link traversal (§3.1), one in file locks (§3.2), and two in file permissions (§3.3).

3.1 Discrepancies in File Link Traversal

One major discrepancy between Windows and Linux file systems is how they handle links. This is because Windows supports unique types of links (e.g., DosDevice) that are not present in Linux. In the following, we first introduce the different types of links in Windows (§3.1.1), and then discuss the differences in how Windows and Linux handle links in file operations (§3.1.2).

3.1.1 Background: Links in Windows. Windows provides several types of links, which are not present in Linux. Among them, we provide a brief introduction to four frequently used types as shown in Table 1: 1) NTFS symbolic links, 2) NTFS junctions, 3) DosDevices, and 4) JDSymlinks.

NTFS symbolic link. NTFS symbolic links [50] are similar to Linux symbolic links, allowing references to files or directories in different locations. A key difference is that Windows requires Administrator privileges to create symbolic links. This is a security measure implemented by Microsoft; this prevents attackers from exploiting them in redirecting file operations to sensitive files [43]. However, this security measure limits the functionality of NTFS symbolic links, as applications should maintain copies of files in multiple locations instead of simply using symbolic links. Detailed discussion is provided in [76].

NTFS junction. NTFS junctions [59] are links that work similarly to NTFS symbolic links. However, they can only reference directories, not files. On the other hand, unlike symbolic links, they are less restrictive as they can reference directories across different volumes and do not require Administrator privileges to create.

DosDevice. DosDevice [13, 46] (often referred to as *object manager symbolic link*) is a special type of symbolic link that maps a simpler name to a device or file path. For example, the name `C:` is assigned to a device `\Device\HarddiskVolumeX` as a DosDevice. DosDevice can be created by a regular user without Administrator privileges.

Table 2: Link traversal behaviors of native file operations in Windows and Linux. We compare Linux system calls and Windows APIs for three types of links: Linux symbolic links, Windows NTFS symbolic links, and Windows JDSymlinks. Note that POSIX functions (compatibility layers) will be discussed later in §5.

Operation	Linux		Windows		
	System calls/flags	Linux symbolic link	Windows APIs/flags	NTFS symbolic link	JDSymlink
open	`open` [39]	✓	`CreateFile` [55]	✓	✓
open (w/ a flag)	`O_NOFOLLOW` [39]	✗	`FILE_FLAG_OPEN_REPARSE_POINT` [55]	✗	✓
remove	`remove` [40]	✗	`DeleteFile` [58]	✗	✓
rename	`rename` [41]	✗	`MoveFile` [56]	✗	✓

✓/✗: follows/does not follow links.

However, the link can only be placed in *NT namespace* [60], which is a namespace for kernel objects in Windows. As this namespace is separated from the file system, DosDevice cannot be placed as a file in the file system, restricting its usage.

JDSymlink. DosDevice is frequently combined with a junction to create a complete and unrestricted symbolic link [7, 31, 35, 75, 83]. While neither a junction nor a DosDevice can independently act as a complete and unrestricted symbolic link, they can do so when combined. This combination, which will be referred to as *JDSymlink* (i.e., Junction + DosDevice Symbolic link), follows a two-step process: 1) it navigates the junction to the directory `\GLOBALROOT\RPC Control` of NT namespace, and 2) it follows the DosDevice to the destination file. For example, consider a JDSymlink located at `C:\temp\link` that points to `C:\target\file.txt`. A file operation on the link will first traverse the junction `C:\temp` to `\GLOBALROOT\RPC Control`. Then, it will follow the DosDevice created at `\GLOBALROOT\RPC Control\link` to the target file `C:\target\file.txt`. As a result, the operation effectively follows the JDSymlink to the target file.

JDSymlink is preferred over the NTFS symbolic link for exploitation purposes, as it can be created by a regular user without elevated privileges. Both junctions and DosDevices can be created by a regular user. Additionally, the location `\GLOBALROOT\RPC Control` is accessible by all users as it serves as an endpoint for RPC calls, which all processes need to access. These factors make JDSymlink accessible to all users, and thus a popular choice for attackers to redirect file operations to sensitive files [16, 31, 83].

3.1.2 Link traversal in Windows and Linux. We compared the link traversal behaviors of native file operations on Windows and Linux. Specifically, we compared the behaviors of `open`, `remove`, and `rename` on Linux system calls and Windows APIs. Also, we additionally compared the behaviors of these file operations on JDSymlinks, as it is frequently used in exploiting real-world security vulnerabilities [16, 31, 83]. Note that these native file operations are different from the compatibility layers provided by programming languages, which will be discussed in §5.

Table 2 shows the results of the comparison. While Windows and Linux handle links equivalently for Windows and Linux NTFS symbolic links, Windows handles JDSymlinks differently. In total, we found three cases: `open` with a flag, `remove`, and `rename`. First, Windows follows links for `open` even if a flag for link following prevention (i.e., `O_NOFOLLOW` in Linux and `FILE_FLAG_OPEN_REPARSE_POINT` in Windows) is provided. This is unexpected because the flag is intended to prevent link traversal. In particular, we found that this flag works well for Linux symbolic links and NTFS symbolic links, but not for JDSymlinks. Second, Windows follows links for `remove`, deleting the target file instead of the link itself. The expected behavior is to delete the link itself, as shown in Windows and Linux NTFS symbolic links. Third, Windows follows links for `rename`, moving the target file instead of the link itself. The expected behavior is to move the link itself, as shown in Windows and Linux NTFS symbolic links.

Security implications. These unexpected behaviors can lead to security vulnerabilities. For example, even if a developer uses a flag to prevent link traversal when opening files, it is simply ignored for JDSymlinks. Consequently, attackers can exploit this to redirect file operations to sensitive files. What is worse is that the flag name `FILE_FLAG_OPEN_REPARSE_POINT` implies it should work with *reparse points* [61], which JDSymlinks are composed of. This can confuse developers when diagnosing security issues. Finally, the behaviors of remove and rename can be exploited by attackers to redirect file operations to arbitrary files, as will be discussed in §4.

3.2 Discrepancies in File Locks

Advisory vs. mandatory locks. Operating systems typically adopt two types of file locks: *advisory locks* and *mandatory locks* [8]. Advisory locks require processes to voluntarily use the locking mechanism, meaning they must explicitly call functions to acquire and release the locks. In contrast, mandatory locks enforce the locking mechanism at the kernel level, denying access to locked files even for processes that do not explicitly use the locking mechanism.

Both locks have their own pros and cons in terms of compatibility and availability. Advisory locks are less compatible with existing applications than mandatory locks. Mandatory locks can cover non-voluntary applications due to their enforcement of the locking mechanism. However, regarding availability, this can cause more severe issues, as mandatory locks make locked files completely unavailable. For example, unprivileged processes can lock files that privileged processes need, disrupting their operations. On the other hand, advisory locks still allow non-voluntary applications to access locked files.

Locks in Windows and Linux. While Linux uses advisory locks, Windows uses mandatory locks. Specifically, Linux implements advisory locks through the `flock` [38] system call, allowing processes to invoke it to acquire and release locks. In contrast, Windows implements various types of mandatory locks, including *share mode locking* [52, 55] and *opportunistic locking (oplock)* [64]. Share mode locking is configured when opening a file (i.e., `CreateFile` [55]). When no specific sharing mode (e.g., read or write) is specified, access by other file operations is denied, leading to errors during execution. Opportunistic locking (oplock) locks files similarly to

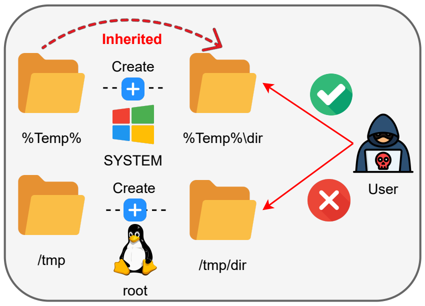

Figure 1: Comparison of permission inheritance between Windows and Linux. Windows files/directories inherit permissions from their parent directories, while Linux's do not.

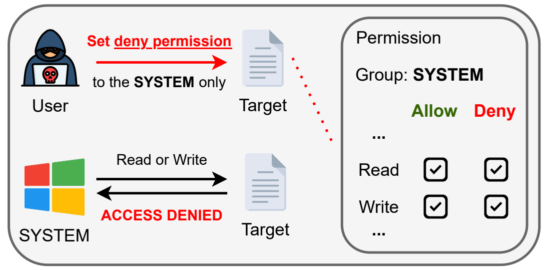

Figure 2: Example of permission inversion in Windows. Files created by regular users can deny access to `SYSTEM`.

share mode locking, but it delays other file operations rather than causing them to fail. This is useful when a process wants to avoid causing errors for other processes.

Security implications. Two mandatory locks in Windows, share mode locks and opportunistic locks, can cause security issues. Share mode locks can be exploited to inject faults into file operations, disrupting the expected file access behaviors of other applications. For example, when an unprivileged process locks a file that a privileged process needs, causing the privileged process to fail in accessing the file. This allows attackers to exploit this issue to trigger the fallback mechanisms of the privileged processes, potentially leading to privilege escalation. We will discuss this in more detail in §4.4. Opportunistic locks can be exploited to delay the execution of privileged processes, potentially causing a denial of service. This issue is sometimes used to win the file race condition in exploiting vulnerabilities, as we will discuss in §4.2.

One interesting thing to note is that Linux initially implemented a mandatory lock in `v2.6.12-rc2` [79], but later removed it in `v5.15` [32]. This lock has been known to cause file race conditions, as noted in existing documentation [37, 79]. For example, if a process acquires a lock immediately after another process has started writing to a file, the lock can be unreliable. Since file operations check for a lock only at the beginning of the operation, a lock can be obtained in the middle of an ongoing file operation. Therefore, the process may experience data changes even though the lock is acquired. This and many similar issues eventually led to the decision to remove the mandatory lock in Linux.

3.3 Discrepancies in File Permissions

Windows and Linux also have two discrepancies in file permissions: *permission inheritance* and *permission inversion.*

3.3.1 Permission Inheritance. Windows and Linux are fundamentally different in managing file permissions. In Linux, file permissions are primarily bound to file ownership. When creating a file, the system explicitly sets permissions based on the user who creates the file. The file owner has fine-grained control over access rights; they can configure read, write, and execute permissions separately for the owner, groups, and others. These permission settings are bound to the file itself and persist even when the file is moved to different locations in the file system.

Unlike in Linux, Windows primarily binds file permissions to the parent directory through inheritance. When creating a file, it automatically inherits the permissions from its parent directory by default [47, 62]. While Windows provides mechanisms to disable permission inheritance, this requires explicit configuration that developers mostly overlook (i.e., `SetNamedSecurityInfo` [57] with `PROTECTED_DACL_SECURITY_INFORMATION` [49]). Therefore, files possess permissions of their parent directories most of the time, which may be more permissive than desired.

Figure 1 shows an example of file permissions in Windows and Linux. Typically, Linux uses `/tmp` as a temporary directory, and Windows uses the environment variable `%Temp%`. In Linux, if a privileged user creates a directory under `/tmp`, permissions are explicitly set based on the owner, preventing access by regular users. However, in Windows, a directory under `%Temp%` (e.g., `C:\Windows\Temp`) inherits permissions from `%Temp%`, which is permissive enough for regular users to create new files or directories within it.

Security implications. Due to this inheritance mechanism, files in Windows may unintentionally allow access to regular users based on where they are located. This contrasts with Linux, where files with restrictive permissions remain inaccessible to unprivileged processes regardless of their location. In Windows, if a file is placed in a user-writable directory, it inherits those permissive permissions from the parent directory. This potentially allows regular users to access it. As we will discuss in §4, this inheritance behavior has led to real-world security vulnerabilities.

3.3.2 Permission Inversion. In Windows, as shown in Figure 2, regular users can create files with inverted permissions that block access for higher privileged processes while allowing access to regular users. This is due to the absence of a strict file permission hierarchy between users and groups. Unlike Linux, where the `root` user can access any file regardless of permissions, Windows does not differentiate between high-privileged and regular users in this manner. In other words, while high-privileged users generally have more permissive access, they cannot access every file while ignoring the permissions of the file itself. This is because Windows uses a list of Access Control Entries (ACEs) [45] to manage file permissions, which specifies access allowance and denial for each user and group. Due to no differentiation, regular users can add an ACE that denies access to higher privileged users, blocking their access. As a result, the file will be inaccessible to higher privileged users, while allowing access from regular users.

Security implications. Regular users creating such files can potentially cause security issues. For example, an attacker can deploy a malicious file with an ACE denying access from antivirus software, to block the software from scanning the file. Moreover, this can prevent the file from being deleted even when it is somehow recognized as a malicious file. Additionally, a malicious user can create such a file to interfere with the file operations of privileged processes by replacing the target files with malicious ones, potentially causing a denial of service. We will discuss these issues in more detail in §4.5.

Answer to RQ1: Windows and Linux have four key discrepancies in file systems: 1) link traversal with JDSymlink, 2) advisory and mandatory locks, 3) permission inheritance, and 4) permission inversion.

4 Real-World Cases

In this section, we answer **RQ2** by presenting real-world cases that are related to the issues in §3. We analyzed 25 real-world software (detail in [76]) and identified related vulnerabilities to verify whether our findings are also applicable to experienced developers. As a result shown in Table 3, we identified and reported 13 vulnerabilities across software from 11 different major vendors, including Google, Mozilla, NVIDIA, Samsung, and Oracle. In the following, we present interesting cases from these findings and categorize their root causes.

4.1 Analysis Methodology

To demonstrate the impact of the discrepancies discussed in §3, we attempted to systematically select candidate programs for in-depth manual analysis. Given that most real-world Windows software is closed-source and typically large, systematic selection is crucial for efficient analysis. To achieve this, we first filtered programs that request UAC elevation by parsing their manifest files, ensuring that the programs are executed with elevated privileges. Next, we analyzed the Import Address Table (IAT) of each binary to identify the usages of relevant file operations. For statically linked binaries that do not have an IAT, we used tools such as IDAPython [22], Lumina [23], and FLIRT [21] to trace the file operations. After identifying candidate programs through these processes, we manually analyzed them to discover vulnerabilities related to the discrepancies. We explain the details in [76].

4.2 Case 1: Firefox Updater EoP to SYSTEM

This vulnerability case demonstrates how challenging it is to mitigate the issues we presented. Firefox has an automatic updater that runs in the background to check and install updates. This updater operates with elevated privileges through a Windows service, allowing it to apply updates without user interaction, such as UAC prompts. The updater applies the mechanism of privilege separation to mitigate potential security vulnerabilities. In other words, the updater is executed in two distinct processes: one with the regular user privilege and the other with the SYSTEM privilege. Additionally,

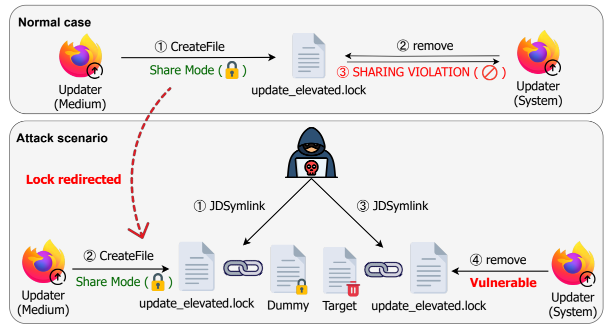

Figure 3: Firefox updater EoP vulnerability overview. Note that the lock file `update_elevated.lock` is located under `C:\ProgramData`, which is permissive to regular users.

although the updater attempted to ensure file integrity by using a restricted share mode lock during the update process, it failed to achieve this goal. Figure 3 illustrates how an attacker bypasses these mechanisms to trigger vulnerable file operations.

Permission inheritance. Due to the inheritance of permissions in the `C:\ProgramData` directory, an attacker with a user privilege can create a JDSymlink on a file used by a privileged updater (① in the attack scenario). During the update, Firefox updater creates a file to `update_elevated.lock`, which indicates the execution context of the update process (②). If `update_elevated.lock` exists, this means that the updater process is executed via the Windows service. Otherwise, it means that the updater process is executed by a privileged user. The update process keeps a handle to the file open using `CreateFile` with a `dwShareMode` of `NULL` (i.e., restricted share mode lock), to prevent external deletion or modification of the `update_elevated.lock` file. Later, during the completion of the update process, the `update_elevated.lock` file is deleted, and this file deletion is performed with `SYSTEM` privileges (④). Unfortunately, the permission of a directory that `update_elevated.lock` is located was inherited from the `C:\ProgramData`, so an attacker with user privileges can create a JDSymlink on `update_elevated.lock` (③).

Link traversal. An attacker can use a JDSymlink to bypass the Firefox updater's file-locking mechanism, redirecting the vulnerable file deletion (③). In general, since the updater already holds a handle to the `update_elevated.lock` file with a restricted `dwShareMode`, an attacker cannot create a JDSymlink to this file. However, an attacker can still exploit this file operation because even `CreateFile` also can be redirected through a JDSymlink. That is, the attacker can redirect the `CreateFile` call with a restricted `dwShareMode` to another file using a JDSymlink, and later replace it with another JDSymlink for exploitation before the file deletion operation occurs. In conclusion, although Firefox updater implemented measures to prevent attackers from modifying the `update_elevated.lock` file, we were able to bypass these measures using a combination of JDSymlink and opportunistic locks (see §3.2).

Opportunistic locks. Although the series of processes described above may seem unlikely to occur reliably, an attacker can exploit opportunistic locks to precisely and reliably control the file operation sequence to win the race condition [4, 13, 15]. In this case, an

Table 3: Discovered vulnerabilities in real-world software.

#	Software	Status	CVE ID	Impact	Description
1	Google Chrome	Fixed	CVE-2024-1694	EoP	Redirection of file move operation on a code red file
2	Google Chrome	Fixed	CVE-2024-7977	EoP	Improper file deletion on `chrome_installer.log` file during the update
3	Mozilla Firefox	Fixed	CVE-2025-2817	EoP	Redirection of file deletion operation on `update_elevated.log` file
4	Mozilla Firefox	Confirmed	-	Sandbox Escape	Blocking file operations of Firefox updater using share mode locks
5	Oracle Java	Fixed	CVE-2025-50063	EoP	Redirection of file deletion operation on `baseimagefam8` file
6	NVIDIA App	Confirmed	-	EoP	Redirection of file deletion operation on `mouseLut.json` file
7	Samsung Magician	Fixed	CVE-2025-32098	EoP	Redirection of file deletion operation on `Optimization` directory
8	Docker Desktop	Fixed	CVE-2025-3224	EoP	Redirection of file deletion operation on `config` directory
9	360 Total Security Antivirus	Fixed	-	DoS	Blocking file access of the antivirus core process using deny permissions
10	Gen Digital CCleaner	Fixed	CVE-2025-3025	EoP	Redirection of file deletion operation during the cleaning feature
11	libcurl	Fixed	-	EoP	Redirection of file deletion when using logging options
12	MalwareBytes Antivirus	Duplicate	CVE-2024-6260	EoP	Redirection of file deletion on `updatrpkg` directory
13	Foxit Reader	Fixed	CVE-2024-38393	EoP	Improper write access permissions granted to user for `Addon` directory

attacker can delay a file deletion operation with an opportunistic lock targeting the `update_elevated.lock` file. During this delay, the attacker can reliably create a JDSymlink pointing to the target file, before the deletion operation occurs. Afterward, the attacker can release the lock to proceed with the deletion operation.

4.3 Case 2: Firefox Updater DoS in Sandboxed Process

This case demonstrates that these issues can be exploited even within processes that have strictly limited privileges, such as a browser sandbox. Firefox updater attempts many file operations on the `Mozilla Firefox` directory in the `C:\ProgramData`. Therefore, we investigated whether the permissions for such files or directories are strictly restricted for sandboxed processes and found that they are not. Additionally, we examined how these issues can lead to critical security vulnerabilities. Figure 4 illustrates the overall flow of this vulnerability. As a result, this issue can be exploited to cause a continuous Denial of Service (DoS) to the Firefox browser. This forces the user to remain on a lower version of Firefox.

Permission inheritance. Due to permission inheritance on the `C:\ProgramData` directory, even the browser sandbox process could access update-related files. Firefox employs a complex sandbox architecture, details of which can be found in Mozilla's official documentation [66]. While the update is in progress, the updater process logs the status of the update to `update.status` file. As in the Firefox updater case, the path inherits permissions from `C:\ProgramData`. Unfortunately, even the network process running with Untrusted integrity and the data decoder and GPU processes running with Low integrity were able to perform read operations on the `update.status`.

Abusing the share mode locking. If an attacker has gained code execution in a sandboxed process, they can exploit the share mode locking mechanism to cause a continuous DoS of Firefox updater. The attacker can preemptively open a handle to the `update.status` file with restrictive `dwShareMode` flags, as the sandboxed process has read permissions for the `update.status` file. This would result in the update process being unable to perform file operations on `update.status`, due to restricted `dwShareMode`, resulting in an `ERROR_SHARING_VIOLATION` error code. In addition, the `update.status` file has a critical role in sharing update progress between separate updater processes. If Firefox updater fails to obtain a write handle to `update.status`, it treats this as a critical internal error and

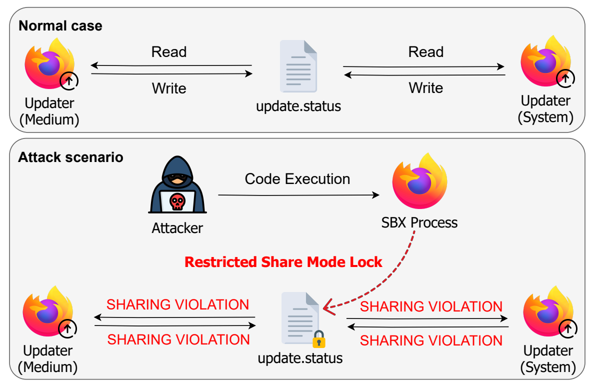

Figure 4: Firefox updater DoS vulnerability overview.

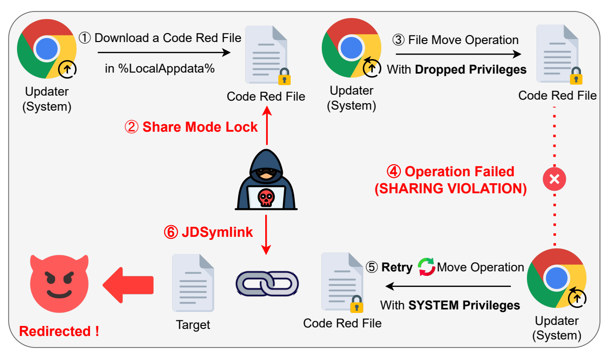

Figure 5: Chrome updater EoP vulnerability overview.

terminates the update process. Afterward, the update will not be performed until the updater attempts the new update process several times. Furthermore, when using another browser-based application with a permission similar to the Firefox sandboxed process, it can continuously interrupt the update process even if Firefox is closed.

4.4 Case 3: Chrome Updater EoP to SYSTEM

Chrome also has a similar issue with Firefox. Notably, the Chrome updater is aware of the potential security risks and attempts to

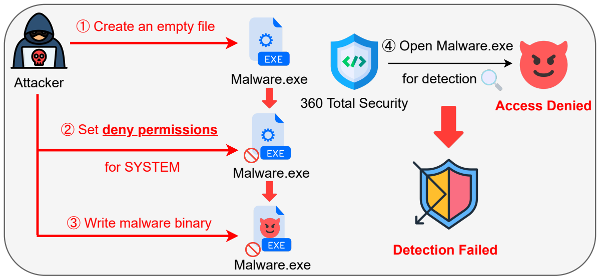

Figure 6: 360 Total Security antivirus DoS overview.

mitigate them by performing file operations with low privileges using impersonation that is described in §7.2.

However, by abusing the share mode locking, an attacker can force the Chrome updater to perform vulnerable file operations with SYSTEM privileges. Figure 5 (and code in [76]) shows the process of how an attacker abuses these mechanisms and redirects a file operation executed with elevated privileges.

Permission inheritance. When urgent updates are required, the Chrome updater downloads a file known as Code Red (①). The file is downloaded to the path `%LocalAppdata%{GUID}`, which inherits permissions from `%USERPROFILE%\AppData`. Since this path is user-writable, an attacker with user privileges can modify this directory or sub-files.

Abusing the share mode locking. As seen in the figure, the updater first performs the file move operation with the logged-on user's privileges (③). However, if the operation fails for any reason (④), it retries the operation with elevated privileges (⑤). Therefore, if an attacker who has obtained the user privileges opens the file with `dwShareMode` set to `NULL` or `FILE_SHARE_READ` (②), the initial attempt will fail, and the file move operation will be executed with SYSTEM privileges.

Link traversal. Similarly, by making a JDSymlink on the path, the attacker can redirect the privileged file move operation (⑥). From the perspective of applying the exploit primitive [83], `MoveFileExW` has the same effect as file deletion on the source path. As a result, an attacker with user privileges can redirect the file move operation during the Chrome update process, making it into an arbitrary file deletion, and achieves SYSTEM privileges.

4.5 Case 4: 360 Total Security Antivirus DoS

We found that even antivirus software developers fail to properly consider the Windows file permission structure. In general, antivirus software must be granted with ultimate permissions to effectively detect or remove malware within the system. However, we demonstrate that abusing the permission structure can block antivirus software from detecting and removing malware. Figure 6 shows how an attacker can interrupt the operation of the antivirus.

Permission inversion. We discovered that by using deny permission, it is possible to create malware that can never be detected by 360 Total Security antivirus. As previously mentioned, a user with the Write DAC permission on a file can also set "deny" permissions for users in other groups (②). In other words, an attacker can configure permissions on a file such that they have access while the SYSTEM group is denied access. For example, if a malicious dropper configures a file with such deny permissions and subsequently writes malicious content to it, any attempt by the antivirus to read a file for the detection will fail, returning an `ERROR_ACCESS_DENIED` error code. As a result, the attacker can create a file that is completely impervious to antivirus detection.

> **Answer to RQ2:** We demonstrated that these discrepancies can lead to critical security vulnerabilities from 11 different vendors. Additionally, we showed that such vulnerabilities could arise in various components such as browser sandboxes, antivirus software, and software updater.

5 Compatibility Layers between Windows and Linux

In this section, we answer **RQ3** by examining the compatibility layers between Windows and Linux.

Target. We investigate the implementation of file operations in six programming languages: C, C++, Python, Java, Go, and Rust, focusing on five common operations: `open`, `create`, `remove`, `copy`, and `rename`. For each operation, we selected the most representative function that provides fine-grained control options (e.g., `O_NOFOLLOW`) for handling links. We also investigated the functions' behaviors with three types of link-related flags: 1) link traversal prevention, 2) link ignoring, and 3) exclusive creation.

Results. Because some behaviors are not clearly defined in the specification, we categorized our results into three groups: *conformant*, *non-compliant*, and *inconsistent*. Conformant behaviors are those that either align with the specification or behave the same on Windows and Linux. Non-compliant behaviors are those that do not follow the specification, while inconsistent behaviors are those that are not clearly defined in the specification and behave differently on Windows and Linux.

Our results demonstrate that the compatibility layers between Windows and Linux do not properly handle these discrepancies. In particular, we found 27 non-compliant behaviors and 9 inconsistent behaviors as summarized in Table 4 (detail in [76]). The table is organized into four parts: 1) specification, 2) Linux implementation, 3) Windows with NTFS symbolic links, and 4) Windows with JDSymlinks. For each function and flag combination, we first examined the specification to understand the expected link traversal behaviors. We then investigated the actual behaviors on Linux and Windows with NTFS symbolic links to assess how well they follow the specifications. Finally, we analyzed Windows with JDSymlinks to understand how this other link type affects the behavior of these functions. Detailed results are provided in [76].

Non-compliant and inconsistent behaviors. Notably, compatibility layers fail to properly handle the link traversal behaviors of JDSymlinks. Since these layers are implemented on top of the native Linux system calls and Windows APIs, they inherit the problematic behaviors we identified earlier in §3.1.2. As demonstrated before, Windows APIs incorrectly follow JDSymlinks in all operations including `open`, `remove`, and `rename` — even when a link traversal prevention flag is given. Compatibility layers should have provided mechanisms to prevent or mitigate these problematic behaviors, but they failed to do so.

Table 4: Behaviors of file-related functions on Windows and Linux.

Platforms (Link Type)	Languages	open			create		remove	copy						rename	
		N	P	XC	N	XC	N	N(S)	N(D)	P(S)	P(D)	I(S)	I(D)	N(S)	N(D)
Specification	C	✓	✗	✗	✓	-	✗	-	-	-	-	-	-	✗	✗
	C++	✓	-	-	-	-	✗	✓	✓	✗	✓	✗	✗	✗	✗
	Python	✓	✗	✗	-	-	?	✓	?	✗	?	-	-	✗	✗
	Java	?	-	-	?	-	?	✓	✗	✗	?	-	-	?	?
	Go	?	-	?	?	-	?	-	-	-	-	-	-	?	?
	Rust	?	-	✗	?	✗	✗	✓	✓	-	-	-	-	✗	✗
Linux	C	✓	✗	✗	✓	-	✗	-	-	-	-	-	-	✗	✗
	C++	✓	-	-	-	-	✗	✓	✗	✗	✗	✗	✗	✗	✗
	Python	✓	✗	✗	-	-	✗	✓	✓	✗	✓	-	-	✗	✗
	Java	✓	-	-	✓	-	✗	✓	✗	✗	✗	-	-	✗	✗
	Go	✓	-	✗	✓	-	✗	-	-	-	-	-	-	✗	✗
	Rust	✓	-	✗	✓	✗	✗	✓	✓	-	-	-	-	✗	✗
Windows (NTFS Symlink)	C	✓	-	✓	✓	-	✗	-	-	-	-	-	-	✗	✗
	C++	✓	-	-	-	-	✗	✓	✓	✗	✢	✗	★	✗	✗
	Python	✓	-	✓	-	-	✗	✓	✓	✗	✓	-	-	✗	✗
	Java	✓	-	-	✓	-	✗	✓	✗	✗	✗	-	-	✗	✗
	Go	✓	-	✓	✓	-	✗	-	-	-	-	-	-	✗	✗
	Rust	✓	-	✗	✓	✗	✗	✓	✓	-	-	-	-	✗	✗
Windows (JDSymlink)	C	✓	-	✓	✓	-	✓	-	-	-	-	-	-	✓	✻
	C++	✓	-	-	-	-	✓	✓	✓	✓	✓	✓	✓	✓	✻
	Python	✓	-	✓	-	-	✓	✓	✓	✓	✓	-	-	✓	✻
	Java	✓	-	-	✓	-	✓	✓	✓	✓	✓	-	-	✓	✻
	Go	✓	-	✓	✓	-	✓	-	-	-	-	-	-	✓	✻
	Rust	✓	-	✓	✓	✓	✓	✓	✓	-	-	-	-	✓	✻

✓/✗: follows/does not follow the link, ?: not specified, -: not available.
■: conformant (i.e., compliant or consistent), ■: non-compliant, ■: inconsistent among implementations.
N: no flags, P: link traversal prevention, I: ignore link (skip), XC: exclusive creation (O_EXCL | O_CREAT). S/D: source/destination.
✢: follows link when file → link, but not when link → link.
★: does not follow link when file → link, but errors in link → link.
✻: partially follows JDSymlinks — follows junctions of JDSymlinks, but not DosDevices.

Moreover, compatibility layers had more non-compliant behaviors than the native implementations. For example, open in C and Python followed Windows NTFS symbolic links even when the exclusive creation flags were given. According to the specification, open with the exclusive creation flag should fail if it is given a link. Additionally, we identified a notable non-compliant behavior in the Linux copy implementation, where it fails to follow destination symbolic links despite being required to do so by the specification. This behavior was unexpected as Linux implementations generally follow specifications strictly. We reported this issue to the gcc developers, and they acknowledged it as a defect.

Linux copy implementation used open with O_CREAT | O_EXCL flags on the destination to avoid overwriting existing files by default. However, open with these flags also fails when the destination is a symbolic link, even if the target location is empty. While this behavior of open is intentional to prevent privileged applications from creating files in unintended locations [27], copy does not intend this behavior. As a result, incorrect usage of open with exclusive creation flags results in a non-compliant behavior of copy, which specifies that it should follow symbolic links on the destination.

Missing compatibility layers. We attempted to investigate the implementation of file locks (§3.2) and file permissions (§3.3), but *could not find suitable compatible implementations across our target languages.* This is likely due to the significant differences in their designs between the two operating systems. In particular, for file locks, Windows uses mandatory file locking, whereas Linux uses advisory file locking. These two locks are fundamentally different in their blocking behaviors for non-voluntary applications, which is discussed in §3.2. Therefore, finding a unified implementation for file locks is challenging.

Similarly, creating a unified implementation for file permissions is difficult due to the distinct designs of the two operating systems. Windows uses an Access Control Entry (ACE) list to manage file permissions [45], whereas Linux uses a mode tied to the file. In Windows, permission checking involves traversing the ACE list, while in Linux, it only requires checking the access mode (i.e., 0o777). Consequently, setting a file to allow all users to read is not straightforward in Windows, as a single deny ACE can override an allow read ACE. Removing the problematic deny ACEs is an option, but it may not be desirable for developers who need more restrictive permissions for specific users or groups.

Nevertheless, chmod [26, 36, 51] could be considered for a unified implementation for file permissions. However, it is limited to configuring only read and write permissions on Windows. The Windows implementation of chmod does not actually modify the ACE list but instead sets or clears the read-only attribute flag on the file when write permission is toggled. This means that chmod with read permission does not guarantee the ability to read a file if there are explicit deny ACEs in the permission list. Furthermore, chmod cannot handle permission inheritance, which is a crucial aspect of file permissions discussed in §3.3.

Answer to RQ3: Compatibility layers between Windows and Linux are insufficient to handle the discrepancies between the two operating systems, having many non-compliant and inconsistent behaviors in handling file links. Moreover, no compatibility layers are available for file locks and file permissions.

6 User Study

To answer **RQ4**, we conducted a user study to assess how aware developers are of the discrepancies discussed in §3.

6.1 Study Protocol

At the beginning of the study, participants were provided with a consent form that outlined the scope of data collection, including the recording of screen activity and voices. They were informed about the overall study procedures and explicitly reminded of their right to withdraw at any time without penalty. To maintain ecological validity and observe authentic coding behaviors, we intentionally withheld the study's core research objectives, minimizing the risk of bias associated with increased security awareness [1, 3, 68]. After completing the programming tasks, we conducted an exit interview to explore participants' awareness of file system-related discrepancies across operating systems. Each participant received a $20 compensation, an amount guided by prior research [28, 73]. In addition, participants were informed that this was a research study and were motivated by interest in security, rather than financial incentives. On average, participants spent 88.4 minutes completing the study (σ = 45.2). Prior to the main study, we conducted a pilot study with two participants, using their feedback to refine the study structure, interview questions, and procedural guidelines.

6.2 User Study Design

Our user study consisted of two parts: a programming task and an interview.

6.2.1 Programming Task Design. We asked participants to implement a cross-platform logging system that runs with elevated privileges on both Windows and Linux. We provided virtual machines running both Windows and Linux for 24 hours to each participant, allowing them to use enough time to complete the task. Also, we allowed our participants to use any resources they wanted, including installing their own IDEs, searching the Internet, and using AI tools. This is to ensure that our participants can complete the task as they would usually do in a real-world software development scenario. We provided skeleton code and asked participants to implement two key components: *logging directory* setup and *file operations*.

Logging directory. We asked our participants to first choose a proper temporary path on Windows and Linux. After the path is chosen, the logging system should create a directory named with the current date within the chosen path. After creating the logging directory, the system will create and manage both the log file and backup file within this directory. Participants were informed that the logging system would run with elevated privileges as part of a system service. Our goal was to observe how developers would naturally implement such a component under realistic conditions, without specific guidance on security risks or defenses.

We note that, in practice, improper access control on privileged logging directories can expose systems to local attacks. On Windows, secure configurations typically require setting appropriate Access Control Lists (ACLs) [45] and adjusting inheritance rules [62]. On Linux, this involves setting restrictive file mode bits and ownership. These platform-specific steps are crucial for preventing unauthorized access to sensitive log files.

File operations. We asked our participants to implement three file operations: `open`, `remove`, and `rename`. Specifically, `open` writes log messages to the log file; `remove` deletes the backup file once a backup is completed; and `rename` renames the log file to a backup file when its size exceeds a predefined threshold.

The task design intentionally reflected real-world development scenarios, where file operations performed under elevated privileges can carry implicit security risks such as unintended symbolic link traversal. We sought to observe whether participants would recognize and address these risks in their implementation.

6.2.2 Interview Design. After completing the programming task, we conducted interviews to explore participants' awareness and typical practices related to OS-level file system differences. We asked them to reflect on the behavioral differences between Windows and Linux file systems and how they would address such issues in real-world development. Participants also discussed their prior knowledge and experience with these platform-specific characteristics, as well as possible mitigation strategies they might consider in their own code. Following the interviews, they completed a post-task survey that included demographic questions and a security quiz designed to assess their background in secure software development. Details of the interview and survey instruments are provided in the Supplementary Materials.

6.3 User Study Results

Demographics. We recruited 21 professional software developers through prior industry collaborations and personal contacts. The developers were employed at a diverse range of software companies, including a global IT corporation with over 30,000 employees and a smaller company with a team of approximately 20 employees. Of the 21 participants, 16 (76.2%) were aged between 26 and 35, while 3 (14.3%) fell within the 18-25 age group, and 2 (9.5%) were in the 36-45 age range. In terms of gender, 18 (85.7%) participants identified as male.

Participants in our study had an average of 10.7 years of programming experience (σ = 5.7). Python was the most frequently used programming language, reported by 17 participants (81.0%), followed by C/C++ with 7 participants (33.3%). Visual Studio Code (VSCode) [44] was the most commonly used integrated development environment (IDE), with 12 participants (57.1%) indicating it as their primary development tool. Linux was the most frequently used operating system, reported by 18 participants (85.7%). For programming-related assistance, ChatGPT [69] was the most widely used resource, with 17 participants (81.0%) regularly relying on it.

We evaluated participants' security knowledge through a quiz [67], classifying those who answered all questions correctly (6 out of 21) as having a security background, and the rest (15 participants) as having limited security knowledge. Additionally, when asked about prior experience with security, all but one participant self-reported exposure to security topics, such as completing an undergraduate course in computer security. Detailed demographics are described in [76].

Functionality and security of developers' code. We analyzed participants' submitted implementations for both functional correctness and security. Nineteen out of twenty-one participants submitted code that compiled and satisfied the task requirements—specifically, correctly writing to the log file and performing the backup operation at least twice—indicating that most were able to meet the intended specification. According to the post-task interviews, all participants reported using AI-powered coding assistants (e.g., ChatGPT [69], Cursor [2], GitHub Copilot [19]), and twelve additionally referred to online resources such as Stack Overflow and official API documentation. However, despite this widespread use of automated and community-based support, our security analysis (detailed in [76]) revealed that all participants introduced vulnerabilities involving unchecked symbolic links, insecure use of privileged temporary directories, and unsafe file system API usage.

This finding highlights that even experienced developers often lack the knowledge or resources needed to properly mitigate OS-level file system vulnerabilities. Notably, when these issues were discussed during post-task interviews, several participants attempted to justify their insecure implementations. One participant (P13) expressed this view, stating: *"Network-layer defense is what really matters. If an attacker already has local access, the system is probably exposed to other attacks anyway, so I don't really bother defending against that when developing software."* However, this perception is also problematic. In practice, local, unprivileged users can significantly contribute to security vulnerabilities. Without clear guidelines or secure default settings, developers may overlook file system security, even in privileged components.

Developers' awareness of security discrepancies. To understand developers' awareness and practices regarding OS-level file system discrepancies, we conducted semi-structured interviews following the programming task.

In the first part of the interview, we asked participants whether they were aware of discrepancies in file system behaviors between operating systems, particularly between Windows and Linux. Notably, none of the 21 participants demonstrated substantial knowledge of these discrepancies. While two acknowledged superficial distinctions—such as variations in path separators or reliance on conditional compilation directives (e.g., `#ifdef`)—their understanding remained limited. When prompted about common cross-platform development practices, only two participants mentioned relevant solutions—one referenced abstracting file operations through wrapper functions, while another cited using portable libraries such as Standard Template Library (STL) and Boost.

In the second part of the interview, we examined participants' awareness of and mitigation strategies for three key file system security issues: (1) symbolic links, (2) file locking (including `CreateFile`'s *share mode locking* and *oplock*), and (3) the use of privileged temporary directories. Ten participants recognized that symbolic links could be exploited by attackers to redirect access to unauthorized files, but only one participant (P19) demonstrated knowledge of effective mitigation strategies, specifically proposing to verify the inode number after opening a file. In contrast, none of the participants showed an understanding of the semantics or risks associated with file locking, including share modes or oplocks. Meanwhile, nine participants correctly identified the risks related to privileged temporary directories and described appropriate mitigations, such as controlling access permissions. Notably, only five participants appeared in both the symbolic link and temporary directory groups, suggesting that awareness and mitigation capabilities were fragmented rather than consistently distributed across participants.

Limitations. We identify three key limitations that should be considered when interpreting our findings. First, our task design necessarily abstracted away many contextual factors present in real-world software development—such as team collaboration, production deadlines, and organizational security policies—which may influence how developers prioritize or balance security in practice. Second, although we avoided providing explicit security instructions to prevent priming effects, the task structure may have implicitly guided participants toward vulnerable patterns. This limitation is partially mitigated by the fact that security requirements are often explicitly stated in real product development settings. More importantly, despite the potential influence of task framing, none of the participants produced secure implementations, suggesting that the observed vulnerabilities reflect broader challenges in secure software development. Lastly, our findings may have limited generalizability due to the composition of our participant pool. While the tasks required C/C++ programming, most participants reported Python as their primary language. Nevertheless, C/C++ was the second most frequently used language, and all participants demonstrated sufficient proficiency to complete the tasks.

Answer to RQ4: Our user study reveals that developers have limited awareness of OS-level file system discrepancies between Windows and Linux and rarely implement effective mitigation strategies. While some participants recognized risks such as symbolic link attacks or insecure temporary directories, their understanding was generally insufficient to inform secure coding decisions. Notably, none were aware of Windows-specific file locking mechanisms.

7 Countermeasures

In this section, we answer **RQ5** by investigating our target applications (list in [76]) for the existing countermeasures to mitigate security vulnerabilities that are caused by the discrepancies. Also, we discuss the challenges of mitigating these vulnerabilities given the weak design of Windows file system.

7.1 Kernel-Level Countermeasures

Windows redirection trust mitigation. To prevent link traversal attacks, Windows provides a mitigation called *Redirection Trust Mitigation* [53]. When enabled, this mitigation prevents processes from following junctions. This can be effective against JDSymlinks as they rely on junctions, blocking their traversal. The mitigation can be enabled per process through the Windows API or group policy settings.

However, this mitigation is coarse-grained, meaning that a process with this mitigation enabled cannot follow any junctions that are created by a non-privileged actor [34]. This can be problematic as legitimate junctions may be needed for applications to function properly. For example, in the Microsoft's winget-pkgs, the mitigation prevented a user from updating or uninstalling a package because the winget process was blocked from accessing its own legitimate junction [42]. When this mitigation is enabled, applications may struggle to function properly as they cannot traverse these predefined junctions. Moreover, this mitigation is poorly documented, thereby developers are hardly aware of this mitigation. We could not find any applications that use this mitigation other than a few Microsoft's applications, such as Microsoft Office 365, Windows installer and spooler service.

Custom kernel drivers. To solve the problems of file locking and permission checking, some vendors implement custom Windows kernel drivers. For example, Avast antivirus software uses a kernel driver to perform file operations without interference from attackers. This driver replaces standard file operation handlers with its own implementation that bypasses normal permission checking and locking mechanisms. This capability is crucial for antivirus functionality — when scanning or quarantining malicious files, the antivirus must be able to access files regardless of any locks or permission restrictions that malware may try to impose. As we saw in §4.5, without this kernel-level access, attackers can prevent the antivirus from quarantining detected malware by manipulating file permissions and locks.

However, implementing custom kernel drivers requires significant expertise and resources. It demands careful engineering and a deep understanding of the Windows kernel and file system internals, since driver bugs can cause system-wide crashes. Additionally, deploying drivers on Windows requires submitting them to Microsoft for validation and signing [25] — a process that is both costly and time-consuming. Due to these high barriers to entry, we found that only a small number of applications implement this mitigation, especially antivirus software.

7.2 Application-Level Countermeasures

We found some applications implement their countermeasures at the application level. In total, we found four countermeasures used in these applications : 1) checking link existence, 2) share mode locking, 3) explicit permissions, and 4) impersonation.

Checking link existence. First, applications can prevent unintended link traversals by checking for links before file operations. They can check either *final path* or for *reparse point*. Chrome checks the final path by comparing the resolved path with the original path [20]. Alternatively, the NVIDIA App checks for reparse points by trying to remove the link itself if the target is a reparse point [61], which is true for all types of links. After detecting a link, applications can reject the operation or verify the link target.

Share mode locking. Second, as shown in §4.2, Firefox updater uses a share mode locking to prevent external modification of the `update_elevated.lock` file. As demonstrated in §3.2, on Windows, if there is already an open handle to a specific file, any access modes not specified in the `dwShareMode` of that handle are denied, even to processes with higher privileges. Using this mechanism, programs can prevent an attacker from modifying the file while it is in use. However, we found that this could be bypassed by redirecting the file operation that applies the share mode locking.

Explicit permissions. Third, Microsoft Office 365 explicitly configures restrictive permissions on its data directory (i.e., `C:\ProgramData\Microsoft`) to prevent unauthorized access. The data directory does not inherit permissions from its parent directory (i.e., `C:\ProgramData`), which has more permissive settings. This ensures its critical application data remains protected regardless of the parent directory's settings.

Impersonation. Fourth, Chrome updater drops its privileges on file operations by *impersonating* the user before performing the operations. It masquerades as the user and performs the file operations with reduced privilege. If they are unexpectedly redirected to privileged files, the operation will fail due to the lack of privilege. This way, attackers cannot tamper with the files they cannot access.

Limitations. While these countermeasures can be effective when properly implemented, they have limitations. They require thorough testing to ensure comprehensive coverage, as missing even one check can create a security vulnerability. Moreover, checking link traversals can be bypassed using time-of-check-time-of-use (TOCTOU) attacks, where an attacker passes the checks with regular files and then replaces them with links afterward.

Furthermore, some countermeasures, such as impersonation, may break the functionality of applications. Specifically, it is hard to determine whether the file operation can successfully run with the reduced privileges. Due to this, Chrome updater retries the operation with the original privileges if it fails the operation with the reduced privileges. However, as we saw in §4.4, this retry mechanism can be exploited by attackers to bypass the mitigation.

7.3 Challenges of Mitigating the Vulnerabilities

Responses from developers of major software vendors. Even experienced developers at major software vendors struggle to implement effective countermeasures. This was evident in the responses we received to our vulnerability reports. For example, when we suggested impersonation as a fix for a vulnerability in Firefox updater (§4.2), the bug triage owner remarked that impersonation is "tricky" to implement correctly. Similarly, Docker's security team responded to our report with an argument that the underlying issue is a Windows bug, rather than a flaw in Docker. Finally, the maintainer of curl acknowledged the difficulty of addressing the reported vulnerability, noting that he was uncertain of a better solution than they currently do. He also mentioned that although he has tried multiple approaches previously, these approaches still introduced other significant challenges and risks.

Our recommendation. We strongly believe that the root cause of these vulnerabilities lies in the fundamental design weaknesses of the Windows file system. Without addressing these core design issues, similar security problems will continue to emerge across the ecosystem. These vulnerabilities affect all Windows applications, not just Microsoft's own products for which they have created targeted patches. Based on the responses from developers we contacted, it is clear they are not at fault for these vulnerabilities but rather victims of the file system's design flaws. We argue that Microsoft should fulfill its responsibility to provide a secure and reliable base for all applications to run on, rather than pushing the burden onto application developers to mitigate the vulnerabilities.

Responses from Microsoft. To deliver our concerns to Microsoft, we reported the fundamental issues with their real-world case examples to their response center. We also recommended that they should at least update the descriptions of file-operation-related APIs on MSDN to address the issue clearly, in case resolving the root causes takes significant time. However, their response was disappointing. While they acknowledged the issue, they categorized its severity as *low* and indicated that there were no immediate plans for remediation. In other words, these issues will likely remain unaddressed, leaving the whole ecosystem of Windows applications affected by these vulnerabilities.

Answer to RQ5: We found that existing countermeasures unfortunately have limitations. Moreover, we found that even experienced developers struggle to implement effective countermeasures. We believe the root cause lies in fundamental Windows file system design weaknesses. Therefore, Microsoft should address this directly rather than through targeted patches.

8 Discussion

Exploit primitive. The exploit primitive [83] in Windows allows attackers to transform a file deletion or move vulnerability into privilege escalation. This primitive abuses the rollback mechanism of the Windows Installer when an MSI package installation fails. Attackers can leverage the vulnerability using a link (e.g., JDSymlink) to redirect a file operation to remove the `C:\Config.Msi` directory, which is protected from regular users. To achieve this, attackers can transform a file operation into a directory operation by using the `::$INDEX_ALLOCATION` stream [48]. For example, executing a native file deletion function `DeleteFileA("C:\Config.Msi::$INDEX_ALLOCATION")` will successfully delete the `C:\Config.Msi` directory. After removing the directory, attackers can recreate it with permissive access controls, in order to place malicious rollback script files within this directory. When an MSI installation fails, these script files will be executed with SYSTEM privileges, effectively achieving privilege escalation.

Microsoft has implemented a mitigation for this exploit primitive by preventing the native `DeleteFile` API from deleting directories through the `::$INDEX_ALLOCATION` stream. However, this mitigation is limited to Windows 11 24H2, leaving other supported versions such as Windows 10 22H2 and Windows 11 23H2 unprotected. Moreover, security researchers and attackers can continuously discover new exploit primitives by analyzing OS internals [29]. Therefore, we believe that any vulnerability related to file operations should be treated as a critical security issue.

In-the-wild exploitation. Vulnerable file operations in Windows are actively exploited in the wild. CVE-2025-21391 [65] serves as an example, demonstrating how attackers achieve privilege escalation to SYSTEM by exploiting a vulnerable file deletion issue within a Windows native component. Although an exploit for this vulnerability was detected by MSRC and subsequently patched in January 2025, the fundamental issues discussed in our research still remain.

9 Ethical Considerations

Responsible disclosure. As shown in Table 3, all vulnerabilities discovered during our research were reported to the respective vendors. All vulnerabilities were confirmed by the respective vendors except the one reported to 360 Total Security Antivirus. For the confirmed vulnerabilities, vendors have either already released patches or are actively working on fixes. In the case of 360 Total Security, the vendor raised an issue regarding reproducibility of the vulnerability following our report. In response, we quickly provided proof of the exploit and a detailed report. However, despite our effort on contacting them multiple times, we received no response thereafter. Subsequently, we found that this vulnerability has been patched silently in the newer version. Specifically, a kernel driver named `BAPIDRV64.sys` was used to address this vulnerability.

User study ethics. Our university's Institutional Review Board (IRB) reviewed and approved the study protocol. To follow ethical standards, we limited personal data collection to only those questions that are necessary for the study. We anonymized participants using unique identifiers and included a "prefer not to say" option in all demographic questions to respect their privacy and autonomy. To ensure ecological validity and mitigate potential harm from vulnerabilities, we provided participants with their own virtual machines and instructed them to configure the environments according to their individual needs.

10 Related Work

Researchers have proposed mechanisms to exploit, detect, and mitigate file-related vulnerabilities. However, no work has delved into the fundamental design weaknesses of the Windows file system, how these weaknesses can lead to security vulnerabilities, and why such vulnerabilities keep recurring.

Exploitation of file-related vulnerabilities. Researchers have extensively studied file-related vulnerabilities and their exploitation techniques. Several works have explored exploiting file operations in Windows privileged services [4, 7, 11, 12, 17, 18, 31, 35, 71, 72, 75, 81], demonstrating how attackers can abuse these operations to escalate privileges. Specifically, researchers have shown that exploit primitives, such as arbitrary file deletion [83] or creation [16], can arise from file-related vulnerabilities and be used to achieve privilege escalation. Additionally, researchers have developed tools like symbolic link tester [13] to support the development and testing of such exploits. Finally, Basu et al. [5] investigated name collisions between file systems with different case sensitivities, highlighting the scenarios where these vulnerabilities can be exploited. These works demonstrate that file-related vulnerabilities discussed in this paper can be exploited to achieve privilege escalation.

Detection of file-related vulnerabilities. Researchers have worked extensively on detecting file-related vulnerabilities. ProcMon [74] aids security analysts in detecting file-related vulnerabilities by monitoring file operations. It provides traces of file operations so that analysts can review them to identify potential vulnerabilities. LPET [24], Crassus [80], PrivescCheck [30], and Jerry [82] detect vulnerabilities by monitoring file operations and reporting potentially vulnerable operations that meet specific rules. Notably, Jerry [82] employs a user interface explorer to simulate and trigger various file operations within a program. Additionally, STING [78] and PathSentinel [33] use dynamic and static analysis on programs, respectively, to detect file-related vulnerabilities on Linux and Android systems.

Mitigation of file-related vulnerabilities. Researchers have proposed several mitigation techniques to prevent file-related vulnerabilities. Chari et al. [6] proposed safe procedures for pathname resolution in file operations of UNIX systems. JIGSAW [77] uses dynamic analysis to automatically derive programmer expectations and enforce them on deployment. More recently, Windows has implemented several mitigations [14, 17, 43, 53] aimed at preventing the exploitation of file-related vulnerabilities.

11 Conclusion

In this paper, we presented a comprehensive analysis of Windows file system design weaknesses that lead to recurring security vulnerabilities. We revealed four critical discrepancies between Windows and Linux file systems, contributing to high-impact vulnerabilities in real-world software. Also, we demonstrated that either compatibility layers or developers are unable to properly handle these discrepancies. Finally, we showed that existing countermeasures are limited in their effectiveness, and even experienced developers struggle to mitigate these vulnerabilities. Based on these findings, we believe the root cause of these vulnerabilities lies in fundamental design weaknesses in the Windows file system. We recommend that Microsoft should consider redesigning the file system to provide a secure foundation for the Windows ecosystem.

12 Acknowledgment

We thank the anonymous reviewers for providing valuable feedback. The authors from KAIST were supported by IITP (No. RS-2024-00331961), NRF (No. 2022K1A3A1A9109426711), and BK21 FOUR (No. 4120200113769). The authors from Sungkyunkwan University were supported by KISA (No. 2780000017) and IITP (No. RS-2022-II220688).

References

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2017. Comparing the Usability of Cryptographic APIs. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.

[2] Anysphere. 2023. Cursor. https://cursor.com. Accessed: 2025-09-09.

[3] Hala Assal and Sonia Chiasson. 2019. "Think Secure from the Beginning": A Survey with Software Developers. In *Proceedings of the 37th CHI Conference on Human Factors in Computing Systems (CHI)*. New York, NY.

[4] Luca Barile. 2021. Exploiting TOCTOU Vulnerability Using OpLock and Junctions. https://lucabarile.github.io/Blog/toctou/index.html. Accessed: 2025-09-09.

[5] Aditya Basu, John Sampson, Zhiyun Qian, and Trent Jaeger. 2023. Unsafe at Any Copy: Name Collisions from Mixing Case Sensitivities. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, CA.

[6] Suresh Chari, Shai Halevi, and Wietse Z Venema. 2010. Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[7] @clavoillotte. 2019. An Introduction to Privileged File Operation Abuse on Windows. https://offsec.almond.consulting/intro-to-file-operation-abuse-on-Windows.html. Accessed: 2025-09-09.

[8] Jonathan Corbet. 2015. Optional Mandatory Locking. https://lwn.net/Articles/667210. Accessed: 2025-09-09.

[9] The MITRE Corporation. 2018. CAPEC-233: Privilege Escalation. https://capec.mitre.org/data/definitions/233.html. Accessed: 2025-09-09.

[10] SSD Disclosure. 2020. CVE-2020-16902. https://ssd-disclosure.com/ssd-advisory-windows-installer-elevation-of-privileges-vulnerability/. Accessed: 2025-09-09.

[11] James Forshaw. 2015. Between a Rock and a Hard Link. https://googleprojectzero.blogspot.com/2015/12/between-rock-and-hard-link.html. Accessed: 2025-09-09.

[12] James Forshaw. 2015. A Link to the Past. https://infocon.org/cons/SyScan/SyScan%202015%20Singapore/SyScan%202015%20Singapore%20presentations/SyScan15%20James%20Forshaw%20-%20A%20Link%20to%20the%20Past.pdf. Accessed: 2025-09-09.

[13] James Forshaw. 2015. symboliclink-testing-tools. https://github.com/googleprojectzero/symboliclink-testing-tools. Accessed: 2025-09-09.

[14] James Forshaw. 2015. Windows 10^H^H Symbolic Link Mitigations. https://googleprojectzero.blogspot.com/2015/08/windows-10hh-symbolic-link-mitigations.html. Accessed: 2025-09-09.

[15] James Forshaw. 2015. Windows Drivers are True'ly Tricky. https://googleprojectzero.blogspot.com/2015/10/windows-drivers-are-truely-tricky.html. Accessed: 2025-09-09.

[16] James Forshaw. 2018. Windows Exploitation Tricks: Exploiting Arbitrary Object Directory Creation for Local Elevation of Privilege. https://googleprojectzero.blogspot.com/2018/08/windows-exploitation-tricks-exploiting.html. Accessed: 2025-09-09.

[17] James Forshaw. 2023. Windows: System Drive Replacement During Impersonation EoP. https://project-zero.issues.chromium.org/issues/42451595. Accessed: 2025-09-09.

[18] Søren Fritzbøger. 2020. CVE-2020-1088 — Yet Another Arbitrary Delete EoP. https://medium.com/csis-techblog/cve-2020-1088-yet-another-arbitrary-delete-eop-a00b97d8c3e2. Accessed: 2025-09-09.

[19] GitHub. 2021. GitHub Copilot. https://github.com/features/copilot. Accessed: 2025-09-09.

[20] Google. 2025. Chrome GPU Link Mitigation. https://chromium.googlesource.com/chromium/src/+/master/content/browser/gpu/gpu_data_manager_impl_private.cc#125. Accessed: 2025-09-09.

[21] Hex-Rays. 2025. FLIRT. https://docs.hex-rays.com/user-guide/signatures/flirt. Accessed: 2025-09-09.

[22] Hex-Rays. 2025. IDAPython. https://docs.hex-rays.com/developer-guide/idapython. Accessed: 2025-09-09.

[23] Hex-Rays. 2025. Lumina. https://hex-rays.com/lumina. Accessed: 2025-09-09.

[24] Can Huang, Xinhui Han, and Guorui Yu. 2020. LPET – Mining MS-Windows Software Privilege Escalation Vulnerabilities by Monitoring Interactive Behavior. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*. Virtual.

[25] HWCert-Migrated. 2019. Driver Signing Changes in Windows 10. https://techcommunity.microsoft.com/blog/windowshardwarecertification/driver-signing-changes-in-windows-10/364859. Accessed: 2025-09-09.

[26] The IEEE and The Open Group. 2024. chmod. https://pubs.opengroup.org/onlinepubs/9799919799/functions/chmod.html. Accessed: 2025-09-09.

[27] The IEEE and The Open Group. 2024. open. https://pubs.opengroup.org/onlinepubs/9799919799/functions/open.html. Accessed: 2025-09-09.

[28] Katharina Krombholz, Karoline Busse, Katharina Pfeffer, Matthew Smith, and Emanuel Von Zezschwitz. 2019. "If HTTPS Were Secure, I Wouldn't Need 2FA"-End User and Administrator Mental Models of HTTPS. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[29] Jonas L. 2020. From Directory Deletion to SYSTEM Shell. https://secret.club/2020/04/23/directory-deletion-shell.html. Accessed: 2025-09-09.

[30] Clément Labro. 2020. PrivescCheck. https://github.com/itm4n/PrivescCheck. Accessed: 2025-09-09.

[31] Clément Lavoillotte. 2019. Abusing Privileged File Operations. https://troopers.de/downloads/troopers19/TROOPERS19_AD_Abusing_privileged_file_operations.pdf. Accessed: 2025-09-09.

[32] Jeff Layton. 2021. fs: Remove Mandatory File Locking Support. https://web.git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=linux-5.15.y&id=f7e33bdbd6d1bdf9c3df8bba5abcf3399f957ac3. Accessed: 2025-09-09.

[33] Yu-Tsung Lee, Hayawardh Vijayakumar, Zhiyun Qian, and Trent Jaeger. 2024. Static Detection of Filesystem Vulnerabilities in Android Systems. *arXiv preprint arXiv:2407.11279* (2024).

[34] Gal De Leon. 2022. Exploring Windows Redirection Trust Mitigation. http://unit42.paloaltonetworks.com/junctions-windows-redirection-trust-mitigation/. Accessed: 2025-09-09.

[35] Xuefeng Li and Zhiniang Peng. 2022. Exploiting Errors in Windows Error Reporting in 2022. https://powerofcommunity.net/assets/v0/poc2022/XuefengLi.pdf. Accessed: 2025-09-09.

[36] Linux man-pages project. 2024. chmod(2) — Linux Manual Page. https://man7.org/linux/man-pages/man2/chmod.2.html. Accessed: 2025-09-09.

[37] Linux man-pages project. 2024. fcntl(2) — Linux Manual Page. https://man7.org/linux/man-pages/man2/fcntl.2.html. Accessed: 2025-09-09.

[38] Linux man-pages project. 2024. flock(2) — Linux Manual Page. https://man7.org/linux/man-pages/man2/flock.2.html. Accessed: 2025-09-09.

[39] Linux man-pages project. 2024. open(2) — Linux Manual Page. https://man7.org/linux/man-pages/man2/open.2.html. Accessed: 2025-09-09.

[40] Linux man-pages project. 2024. remove(3) — Linux Manual Page. https://man7.org/linux/man-pages/man3/remove.3.html. Accessed: 2025-09-09.

[41] Linux man-pages project. 2024. rename(1) — Linux Manual Page. https://man7.org/linux/man-pages/man1/rename.1.html. Accessed: 2025-09-09.

[42] lordgreg. 2024. winget-pkgs: Issue #188562. https://github.com/microsoft/winget-pkgs/issues/188562. Accessed: 2025-09-09.

[43] Microsoft. 2008. Security Policy Settings New for Windows Vista. https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-vista/cc766301(v=ws.10). Accessed: 2025-09-09.

[44] Microsoft. 2015. Visual Studio Code. https://code.visualstudio.com. Accessed: 2025-09-09.

[45] Microsoft. 2021. Access Control Entries. https://learn.microsoft.com/en-us/windows/win32/secauthz/access-control-entries. Accessed: 2025-09-09.

[46] Microsoft. 2021. Defining an MS-DOS Device Name. https://learn.microsoft.com/en-us/windows/win32/fileio/defining-an-ms-dos-device-name. Accessed: 2025-09-09.

[47] Microsoft. 2021. File Security and Access Rights. https://learn.microsoft.com/en-us/windows/win32/fileio/file-security-and-access-rights. Accessed: 2025-09-09.

[48] Microsoft. 2021. File Streams (Local File Systems). https://learn.microsoft.com/en-us/windows/win32/fileio/file-streams. Accessed: 2025-09-09.

[49] Microsoft. 2021. SECURITY_INFORMATION. https://learn.microsoft.com/en-us/windows/win32/secauthz/security-information. Accessed: 2025-09-09.

[50] Microsoft. 2021. Symbolic Links. https://learn.microsoft.com/en-us/windows/win32/fileio/symbolic-links. Accessed: 2025-09-09.

[51] Microsoft. 2022. _chmod, _wchmod. https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/chmod-wchmod?view=msvc-170. Accessed: 2025-09-09.

[52] Microsoft. 2022. Creating and Opening Files. https://learn.microsoft.com/en-us/windows/win32/fileio/creating-and-opening-files. Accessed: 2025-09-09.

[53] Microsoft. 2022. PROCESS_MITIGATION_REDIRECTION_TRUST_POLICY structure (winnt.h). https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process-mitigation-redirection-trust-policy. Accessed: 2025-09-09.

[54] Microsoft. 2022. STRIDE Model. https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats. Accessed: 2025-09-09.

[55] Microsoft. 2023. CreateFileA Function (fileapi.h). https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea. Accessed: 2025-09-09.

[56] Microsoft. 2023. MoveFile Function (winbase.h). https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-movefile. Accessed: 2025-09-09.

[57] Microsoft. 2023. SetNamedSecurityInfoA Function (aclapi.h). https://learn.microsoft.com/en-us/windows/win32/api/aclapi/nf-aclapi-setnamedsecurityinfoa. Accessed: 2025-09-09.

[58] Microsoft. 2024. DeleteFileA Function (fileapi.h). https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-deletefilea. Accessed: 2025-09-09.

[59] Microsoft. 2024. Hard Links and Junctions. https://learn.microsoft.com/en-us/windows/win32/fileio/hard-links-and-junctions. Accessed: 2025-09-09.

[60] Microsoft. 2024. Naming Files, Paths, and Namespaces. https://learn.microsoft.com/en-us/windows/win32/fileio/naming-a-file. Accessed: 2025-09-09.

[61] Microsoft. 2024. Reparse Points. https://learn.microsoft.com/en-us/windows/win32/fileio/reparse-points. Accessed: 2025-09-09.

[62] Microsoft. 2025. ACE Inheritance Rules. https://learn.microsoft.com/en-us/windows/win32/secauthz/ace-inheritance-rules. Accessed: 2025-09-09.

[63] Microsoft. 2025. Microsoft Vulnerability Severity Classification for Windows. https://www.microsoft.com/en-us/msrc/sdlbugbar. Accessed: 2025-09-09.

[64] Microsoft. 2025. Opportunistic Locks. https://learn.microsoft.com/en-us/windows/win32/fileio/opportunistic-locks. Accessed: 2025-09-09.

[65] Microsoft. 2025. Windows Storage Elevation of Privilege Vulnerability. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2025-21391. Accessed: 2025-09-09.

[66] Mozilla. 2025. Firefox Sandbox Architecture. https://wiki.mozilla.org/index.php?title=Security/Sandbox/Architecture. Accessed: 2025-09-09.

[67] Sanghak Oh, Kiho Lee, Seonhye Park, Doowon Kim, and Hyoungshick Kim. 2024. Poisoned ChatGPT Finds Work for Idle Hands: Exploring Developers' Coding Practices with Insecure Suggestions from Poisoned AI Models. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[68] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. 2014. It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*. New Orleans, LA.

[69] OpenAI. 2022. OpenAI ChatGPT. https://chatgpt.com. Accessed: 2025-09-09.

[70] Stack Overflow. 2024. Technology | 2024 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2024/technology#1-operating-system. Accessed: 2025-09-09.

[71] Tze Bin Bryan Pak. 2024. What's Under the Hood? Root Cause and Patch Analyses of Elevation of Privilege Vulnerabilities in the Windows Operating System. In *IRC Conference on Science, Engineering and Technology*. Springer, 168–180.

[72] Zhiniang Peng, Xuefeng Li, and Lewis Lee. 2021. Diving in to spooler: Discovering LPE and RCE Vulnerabilities in Windows Printer. https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Diving-Into-Spooler-Discovering-Lpe-And-Rce-Vulnerabilities-In-Windows-Printer.pdf. Accessed: 2025-09-09.

[73] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*. Copenhagen, Denmark.

[74] Mark Russinovich. 2024. Process Monitor v4.01. https://learn.microsoft.com/en-us/sysinternals/downloads/procmon. Accessed: 2025-09-09.

[75] Eran Shimony. 2019. Follow the Link: Exploiting Symbolic Links with Ease. https://www.cyberark.com/resources/threat-research-blog/follow-the-link-exploiting-symbolic-links-with-ease. Accessed: 2025-09-09.

[76] Dong uk Kim, Junyoung Park, Sanghak Oh, Hyoungshick Kim, and Insu Yun. 2025. Windows plays Jenga: Uncovering Design Weaknesses in Windows File System Security. https://github.com/kaist-hacking/Jenga.

[77] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. 2014. JIGSAW: Protecting Resource Access by Inferring Programmer Expectations. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA.

[78] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. 2012. STING: Finding Name Resolution Vulnerabilities in Programs. In *Proceedings of the 21st USENIX Security Symposium (Security)*. Bellevue, WA.

[79] Andy Walker. 2007. Mandatory File Locking For The Linux Operating System. https://www.kernel.org/doc/Documentation/filesystems/mandatory-locking.txt. Accessed: 2025-09-09.

[80] wdormann. 2022. Crassus Windows Privilege Escalation Eiscovery Tool. https://github.com/vu-ls/Crassus. Accessed: 2025-09-09.

[81] Wenxu Wu. 2019. Battle of Windows Service: A Silver Bullet to Discover File Privilege Escalation Bugs Automatically. https://i.blackhat.com/USA-19/Wednesday/us-19-Wu-Battle-Of-Windows-Service-A-Silver-Bullet-To-Discover-File-Privilege-Escalation-Bugs-Automatically.pdf. Accessed: 2025-09-09.

[82] Chendong Yu, Yang Xiao, Jie Lu, Yuekang Li, Yeting Li, Lian Li, Yifan Dong, Jian Wang, Jingyi Shi, Defang Bo, et al. 2024. File Hijacking Vulnerability: The Elephant in the Room. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[83] Simon Zuckerbraun. 2022. Abusing Arbitrary File Deletes to Escalate Privilege and Other Great Tricks. https://www.zerodayinitiative.com/blog/2022/3/16/abusing-arbitrary-file-deletes-to-escalate-privilege-and-other-great-tricks. Accessed: 2025-09-09.