File Hijacking Vulnerability: The Elephant in the Room

Chendong Yu [*], Yang Xiao [*] [§], Jie Lu[†], Yuekang Li[‡], Yeting Li[*], Lian Li[†],
Yifan Dong[*], Jian Wang[*], Jingyi Shi[*], Defang Bo[*] and Wei Huo[*]
[*] Institute of Information Engineering, CAS, China
[†] SKLP, Institute of Computing Technology, CAS, China
[‡] University of New South Wales
{yuchendong, xiaoyang}@iie.ac.cn, {lujie}@ict.ac.cn, {yuekang.li}@unsw.edu.au, {liyeting}@iie.ac.cn,
{lianli}@ict.ac.cn, {yifan.dong}@foxmail.com, {wangjian, shijingyi, bodefang, huowei}@iie.ac.cn

***Abstract*—Files are a significant attack vector for security boundary violation, yet a systematic understanding of the vulnerabilities underlying these attacks is lacking. To bridge this gap, we present a comprehensive analysis of File Hijacking Vulnerabilities (FHVulns), a type of vulnerability that enables attackers to breach security boundaries through the manipulation of file content or file paths. We provide an in-depth empirical study on 268 well-documented FHVuln CVE records from January 2020 to October 2022. Our study reveals the origins and triggering mechanisms of FHVulns and highlights that existing detection techniques have overlooked the majority of FHVulns. As a result, we anticipate a significant prevalence of zero-day FHVulns in software.**

We developed a dynamic analysis tool, JERRY, which effectively detects FHVulns at runtime by simulating hijacking actions during program execution. We applied JERRY to 438 popular software programs from vendors including Microsoft, Google, Adobe, and Intel, and found 339 zero-day FHVulns. We reported all vulnerabilities identified by JERRY to the corresponding vendors, and as of now, 84 of them have been confirmed or fixed, with 51 CVE IDs granted and $83,400 bug bounties earned.

I. INTRODUCTION

My understanding is that sort of thing is outside of our security model.... Together, I fear we have a vulnerability on our hands that we need to fix, a bigger vulnerability than I originally anticipated. [1]

— A Git maintainer commented on CVE-2022-24765

Modern operating systems, such as Windows and MacOS, implement security boundaries to separate code and data of different trust levels. Those boundaries serve as a fundamental protection mechanism to isolate sensitive data and high-privileged execution environments [23]. However, these security boundaries can be easily breached by File Hijacking Vulnerabilities (FHVulns). FHVulns represents a type of security flaw where an attacker can breach the security boundaries by manipulating files, including file paths and contents, and they can result in severe security issues such as arbitrary code execution [37], privilege escalation [38], and data loss [39].

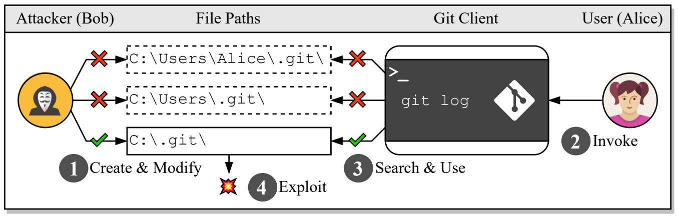


Fig. 1: A file hijacking vulnerability identified by JERRY (CVE-2022-24765).

Fig. 1 depicts an example of an FHVuln (CVE-2022-24765) in the official Git client, caused by improper handling of file paths, leading to arbitrary command execution. This vulnerability has existed for 18 years since the inception of the Git project and is exploitable in all mainstream operating systems, including Windows, Linux and MacOS. To illustrate the exploitation of this vulnerability, let us assume Bob is the attacker and Alice is the victim. ❶ Bob *creates* the C:\.git\ directory where he can modify any contents within the directory. Note that Bob does not have the right to modify the following directories: C:\Users\Alice\ and C:\Users\. However, anyone in the *Authenticated Users* group can create directories under the path C:\. ❷ Alice *invokes* the Git command, such as git log in her home directory (C:\Users\Alice\). ❸ If no .git\ directory exists in the invoking directory C:\Users\Alice\, the Git client will recursively *search* for .git\ in its parent directories. In this case, the Git client will locate the C:\.git\ directory created by Bob and consider it as the Git home directory. ❹ Bob can put githooks [42] in the directory C:\.git\, which will be executed by the Git client. As a result, Bob can trick Alice into executing arbitrary commands.

There have been various studies on different aspects of FHVulns. For example, research [59], [70], [76], [77] focused on detecting files with weak permissions, which can result in security boundary violations. More recently, a number of works aimed at detecting dangerous file operations that could potentially breach security boundaries, such as process creation [45], [49], dynamic library loading [44], file cre-

[1]All the excerpts in this paper are from discussions among developers while fixing the vulnerabilities reported by us.
[*] Also with Key Laboratory of Network Assessment Technology, CAS.
[*] Also with Beijing Key Laboratory of Network Security and Protection Technology.
[*] Also with School of Cyber Security, UCAS, Beijing, China
[§] Yang Xiao is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23038
www.ndss-symposium.org

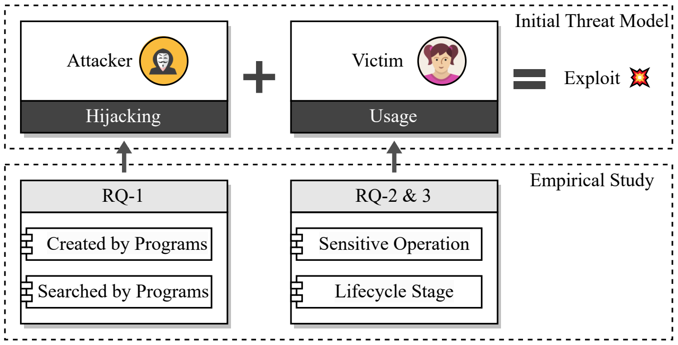


Fig. 2: The threat model of file hijacking vulnerabilities and overview of our empirical study.

ation/deletion [6], [9], and temporary directory creation [72]. Despite considerable efforts in this direction, eradicating FHVulns remains challenging for researchers and practitioners. For instance, existing static permission-checking techniques are unable to detect the vulnerability shown in Fig. 1 due to the lack of awareness of program behaviors. Through our analysis of existing studies, we have found that there is still a lack of systematic study and thorough understanding of FHVulns.

A. Empirical Study

Prior to conducting the empirical study, we present a summary of the threat model for FHVulns. As illustrated in Fig. 2, there are two necessary conditions that must be met in order to exploit an FHVuln: ❶ the attacker must be able to hijack certain files or directories (***Requirement 1***); ❷ the vulnerable software must interact with the hijacked files or directories through sensitive operations (***Requirement 2***). Based on the threat model, we performed the first empirical study on FHVulns. The objective of this study was to gain insights into the origins and triggering mechanisms of FHVulns by addressing the following research questions:

- **RQ1** *What are the origins of the hijacked files?*
- **RQ2** *What types of operations are dangerous vulnerability-triggering operations?*
- **RQ3** *When in the software lifecycle (installation, uninstallation, ...) are file hijacking vulnerabilities triggered?*

We collected 268 CVE records [35] documenting FHVulns from January 2020 to October 2022 to conduct the empirical study. During the study, we made three observations, corresponding to the research questions. ❶ (***Observation*** 1) Hijacked files can originate from two sources: files *created* by the vulnerable program with weak permissions, allowing attackers to manipulate them, or files created by the attacker and encountered by the vulnerable program through specific *search* strategies. The first origin accounts for 10.1% (27) of the total vulnerabilities, while the majority of vulnerabilities (89.9%, 241) come from the second origin. Existing techniques primarily focus on detecting files with weak permissions, i.e., vulnerabilities with the first origin, while missing out on the majority of FHVulns. ❷ (***Observation*** 2) Not all accesses to hijacked files can result in vulnerabilities. For example, operations such as querying file sizes are generally safe. Through our study, we identified six types of operations that can lead to vulnerabilities. None of the existing techniques address all six types of operations. ❸ (***Observation*** 3) FHVulns can be triggered at any stage of the software lifecycle, with the majority of vulnerabilities being triggered during the stage of software program startup. No existing technique addresses every stage in the software lifecycle to detect FHVulns.

B. Detection Tool

Based on our observations from the study, we have identified a significant prevalence of zero-day FHVulns in software, as existing detection techniques have overlooked the majority of these vulnerabilities. To address this gap, we have developed a new FHVuln detection tool called JERRY [2]. JERRY monitors the target software throughout its entire software lifecycle (Observation 3), and records executed file operation traces at each stage. Next, JERRY analyzes recorded execution traces to discover file paths created or searched by the target software, which may be generated or manipulated by attackers. Thus, JERRY is able to simulate the hijacking of files from both origins (Observation 1). Finally, JERRY reports a vulnerability if the target software performs dangerous file operations on the hijacked files (Observation 2).

To assess the effectiveness of JERRY, we constructed a benchmark comprising 51 known FHVulns. Our evaluation revealed that JERRY successfully detected 50 out of the 51 vulnerabilities with no false positives, surpassing all the baselines by detecting at least 13 more vulnerabilities. Furthermore, we applied JERRY to 438 popular programs from notable vendors such as Microsoft, Google, Adobe, Intel, Dell, etc. We identified 339 previously unknown FHVulns in 176 programs and earned $83,400 through the bug bounty programs. Notably, some of these vulnerabilities had persisted for a long time, as evidenced by the 18-year-old vulnerability depicted in Fig. 1, which was present from the outset of the Git project. These findings underscore the prevalence of FHVulns and the need to address them, as they have been overlooked in the past.

We conducted an in-depth analysis of the newly detected vulnerabilities and yielded two findings for RQ1 and RQ2: ❶ (***Finding*** 1) The software-tailored search strategy has not received extensive attention. ❷ (***Finding*** 2) Reading operations result in a higher number of FHVulns, and pose a higher risk than we thought. These findings are derived from new types of FHVulns and thus they are not observable from existing FHVulns. Furthermore, we conducted a detailed discussion on FHVulns in both Windows and Unix-like systems, identified the reasons behind their widespread occurrence, and discussed the responsibilities for mitigating them. Our findings provide valuable insights for future research in improving detection tools and developing effective defenses against FHVulns.

This paper makes the following contributions.

- We, for the first time, provided a clear definition of FHVuln's threat model. Using this threat model, we conducted the first empirical study on FHVulns, revealing the origins and triggering mechanisms of FHVulns.
- We developed a dynamic analysis tool, JERRY, to detect FHVulns and applied it to 438 popular programs and uncovered 339 zero-day FHVulns. All vulnerabilities identified by JERRY were reported to the vendors, resulting in

[2]JERRY is a mouse featured in the cartoon *Tom & Jerry*. According to certain myths, elephants are afraid of mice.

84 of them being confirmed or fixed, with 51 CVE IDs granted and $83,400 in bug bounties earned.
- We conducted an in-depth analysis of the newly discovered FHVulns and made new findings that were not observable from existing FHVulns. Additionally, we provided insights on multiple topics which can enable future research.

This paper is coupled with a companion website: https://sites.google.com/view/iamjerry. We will release JERRY together with our study results through this website.

II. Empirical Study

This does happen, and is something we historically have tried to prevent in the installer.
— A Google developer commented on CVE-2023-2939

In this section, we first present the data collection and analysis process for our empirical study, then conduct a comprehensive investigation on 268 FHVulns to answer the research questions (RQ1-RQ3) introduced in Section I. Finally, we provide more details about the threat model and impacts of FHVulns, derived from the study.

A. Data Collection and Analysis

In order to conduct a comprehensive study of FHVulns, it was essential to gather a substantial dataset of such vulnerabilities. To achieve this, we compiled a dataset by collecting all documented FHVulns from the CVE (Common Vulnerabilities and Exposures) database [35] for the period of January 2020 to October 2022. During this period, a total of 52,719 CVEs were recorded. To focus specifically on FHVulns in programs running on Windows, Linux, and Mac operating systems [3], we excluded vulnerabilities in Web and IoT devices, as well as memory corruption vulnerabilities like buffer overflows and double frees. This filtering process resulted in a dataset of 23,032 vulnerabilities.

Next, we utilized an iterative process to systematically identify FHVulns using keyword-based searches. Initially, we queried the CVE database with the following keywords: *access control*, *hijack*, and *permission*. Then, we manually examined each resulting vulnerability to filter out unrelated vulnerabilities that did not fit our threat model (Fig. 2), and extracted additional keywords. This two-step process was repeated until no new keywords were extracted. In the end, the following additional keywords: *uncontrolled*, *symbol*, *search*, and *install*, were extracted and used in our query.

Finally, we excluded vulnerabilities that lacked sufficient details. Specifically, we only considered vulnerabilities that had information describing the hijacked files, the triggering stage, and the triggering file operations. For example, we filtered out CVE-2021-0057 [20], while retaining CVE-2020-36167 [17]. In total, we collected 268 CVEs with sufficient details.

After the data collection phase, our study followed the same empirical research approach in [80] to investigate the 268 FHVulns. The 268 FHVulns were evenly distributed among three authors of this paper and each author examined his assigned FHVulns to label their corresponding hijacked file origins (RQ1), sensitive operations (RQ2), and triggering lifecycles (RQ3). Subsequently, the authors performed a peer-review process where each labeled vulnerability was cross reviewed by the other two authors. in which each author reviewed the labeling of the vulnerabilities assigned to the other two authors. Whenever discrepancies arose in the labeling results, the three authors held a collective discussion to reconcile the differing viewpoints. In cases where a consensus could not be reached, an additional author was recruited to join the discussion until a resolution was achieved. The labeling and analysis of the 268 FHVulns took approximately three months.

[3]We focused on the mainstream operating systems because they provide full-fledged security boundaries and the relevant vulnerabilities are more likely to be reproducible.

B. RQ1: Origins of Hijacked Files

***Observation 1*: Most (89.9%) hijacked files are due to the five search strategies employed by the programs and the underlying operating systems, while the rest come from files created by programs with weak permissions.**

One of the necessary triggering conditions (***Requirement 1***) requires the existence of hijacked files. Therefore, the key to understanding FHVulns lies in understanding the origins of hijacked files, specifically where these files come from. While it has been previously studied that files created by programs with weak permissions can be hijacked, this origin only accounts for a small portion of the studied vulnerabilities, specifically 10.1% (27) of the total.

The majority of hijacked files (241, 89.9%) are a result of different search strategies employed by the studied programs and their underlying operating systems. When the absolute file path is not provided, operating systems or programs typically use different file search strategies to locate the target file. Frequently, those search strategies could potentially return directories with weak permissions, which allows attackers to plant malicious files. In our investigation, we have identified 5 search strategies that could potentially return directories with weak permissions.

❶ Path Search Order (9, 3.4%). Nine vulnerabilities were found to be due to path search order in the underlying system. In the Windows system, when a command is executed, the system first attempts to locate the invoked executable file in the current working directory (CWD) before searching paths defined in the *PATH* environment variable. This strategy poses a security risk when the CWD has weak permissions. Take a vulnerability in Github Cli [19] for example. When executing the command gh, the CLI client will try to locate the executable file *git.exe* in CWD. Once CWD (e.g., C:\ProgramData) has weak permissions, attackers can place a malicious executable file *git.exe* in the directory.

❷ Linux Paths on Windows (12, 4.5%). Cross-system migration is a common practice in software development. In our study, we identified 12 vulnerabilities resulting from improper path adaptation during migration from Linux to Windows. Specifically, when Linux paths do not exist on the Windows system, the system searches from the C:\ directory. As C:\ is writable by default, this allows attackers to easily plant malicious files. Take CVE-2019-5443 [11] for example. The program *curl* reads the default configuration file /usr/local /ssl/openssl.cnf. On windows, this file does not exist, and the system searches for the configuration file C:\usr\local\ssl\ openssl.cnf instead. Since the path C:\ has weak permissions, an attacker can easily hijack the file and control the engine configuration, leading to arbitrary code execution.

❸ **Unquoted Paths (46, 17.1%).** A total of 46 vulnerabilities have been identified to be caused by unquoted paths. An unquoted path refers to a file path that is not enclosed in quotation marks. When an unquoted path contains spaces or special characters, the Windows system truncates the path and searches for the file using the truncated path. For instance, in CVE-2020-13884 [14], the *Citrix* Workspace program invokes the *CreateProcess* API to execute the file *TrolleyExpress.exe* with an unquoted path C:\ProgramData\ Citrix\Citrix Workspace 1911\TrolleyExpress.exe. As a result, Windows system truncates the path with spaces and searches for the file using the path C:\ProgramData\Citrix\Citrix.exe , effectively loading the file *Citrix.exe* instead of *Trolley-Express.exe*. Since the path C:\ProgramData\Citrix has weak permissions, attackers could hijack *Citrix.exe*, resulting in privilege escalation.

❹ **Symbolic Links (52, 19.4%).** Improper handling of symbolic links has been found to result in 51 vulnerabilities. A symbolic link (also known as a soft link or symlink) is a type of file in a computer's file system that serves as a reference or pointer to another file or directory [54]. When accessing a symbolic link, the operating system follows the link to locate the target file or directory. CVE-2022-39845 [39] is such an example which leads to arbitrary directory deletion. Specifically, the uninstaller of Samsung Kies [28] attempts to delete a non-existent directory C:\ProgramData\Samsung \DeviceProfile\Cache. Since attackers can have write permission to C:\ProgramData\Samsung, an attacker can create the symbolic link C:\ProgramData\Samsung\DeviceProfile\ Cache targeting an arbitrary directory, e.g., C:\. Consequently, the uninstaller will delete all files in C:\.

❺ **Dynamically Loaded Libraries (122, 45.5%).** There are 122 vulnerabilities that can be triggered by loading dynamic libraries, such as dynamic link library (DLL) files on Windows and shared object (SO) files on Linux. When a DLL is loaded in Windows, the system searches for the target DLL in the CWD, system directories, windows directories, and directories declared in the *PATH* environment variable, in that order. On the other hand, when an executable file in Linux loads SO files, the system first searches for loaded files under the directories declared in the *DT_RPATH* and *DT_RUNPATH* sections of the executable file. If the search fails, the system then searches directories declared in the environment variable *LD_LIBRARY_PATH* and in the configuration file /etc/ld.so. conf, as well as in default directories /lib (/lib64) and /usr/lib (/usr/lib64). During the search process for dynamic libraries, the operating system looks for these libraries in searched directories. However, if a directory with weak permissions is encountered during this search, it could potentially allow an attacker to hijack the dynamic libraries that are to be loaded. DLL hijacking [21], [41] has recently gained significant attention, and most of the existing vulnerabilities studied in our research are caused by DLL hijacking.

C. RQ2: Sensitive operations

One of the necessary triggering conditions (***Requirement 2***) requires the vulnerable program to operate on hijacked files. Nevertheless, not all operations are susceptible to such attacks. Operations such as querying information (e.g., file size), closing files, or enumerating files in a directory are generally safe. In our study, we identified and categorized six types of dangerous operations that can potentially lead to exploitation: *moving*, *creating*, *deleting*, *reading*, *process creation*, and *image loading*.

***Observation 2*:** There are six types of dangerous operations on hijacked files subject to file hijacking attacks. Among the six types of operations, *process creation* and *image loading* are most frequently exploited (accounting for 28.4% and 45.1% of total vulnerabilities, respectively). The other four types of dangerous operations are *moving* (1.1%), *reading* (7.1%), *creating* (8.2%), and *deleting* (10.1%).

❶ **Moving (3, 1.1%)**, ❷ **Creating (22, 8.2%)**, and ❸ **Deleting (27, 10.1%).** These three types of operations are dangerous when accessing symbolic links, as discussed in CVE-2022-39845. Such operations on hijacked symbolic links can lead to data loss and destruction of file integrity.

❹ **Reading (19, 7.1%).** Programs commonly read different settings from configuration files, which are subject to file hijacking attacks. For instance, attackers may tamper with a database connection string in configuration files which can redirect subsequent queries to a malicious database. Alternatively, attackers can insert harmful file paths in the configuration file, tricking the program to execute malicious code.

❺ **Process Creation (76, 28.4%).** Process creation involves creating a new process to execute a file, which can lead to arbitrary code execution if the file can be hijacked. For instance, during installation, programs may download installation files from internet first then launch the downloaded files to initiate the installation process. Hijacking the downloaded files can trick the programs to execute arbitrary commands.

❻ **Image Loading (121, 45.1%).** Image loading refers to loading dynamically loaded libraries, i.e., DLL and SO files on Windows and Linux, respectively. Hijacked libraries can lead to arbitrary code execution.

D. RQ3: Software lifecycle

We divide the entire software lifecycle into six distinct stages: installation, uninstallation, updating, repairing, starting up, and usage. Each stage performs different types of actions.

***Observation 3*:** While the majority (62.3%) of FHVulns are exploited during the *Starting up* stage, FHVulns can be triggered at any stage during the software lifecycle, i.e., *Installation* (17.2%), *Uninstallation* (4.5%), *Updating* (1.9%), *Repairing* (3.7%) and *Usage* (10.4%).

❶ **Installation (46, 17.2%).** 46 vulnerabilities are triggered during software installation. The installation stage typically involves actions such as checking system requirements, copying files, setting up configurations, etc. For instance, the installation process often copies files from the installation media (e.g., a downloaded package) to specified locations on the hard drive. It is worth noting that the installation process often requires high privileges, and file hijacking vulnerabilities in this stage are particularly dangerous: attackers can hijack files in the installation process to gain high privileges.

❷ **Uninstallation (12, 4.5%).** 12 vulnerabilities occur at the uninstallation stage, all involving symbolic links. When uninstalling a software, the uninstallation process removes all files (e.g., executable files and configuration files) associated with the software from the system. As discussed in CVE-2022-39845 [39], the uninstallation process may follow a hijacked symbolic link to delete arbitrary directories, resulting in data loss and system damage.

❸ **Updating (5, 1.9%)** and ❹ **Repairing (10, 3.7%)**. The two stages perform similar actions, such as replacing an existing version of software with a newer or correct version. Five and ten vulnerabilities are triggered in the two stages, respectively.

❺ **Starting Up (167, 62.3%).** The majority of vulnerabilities (167) happen during the starting up stage. When starting up a program, the program usual conducts a series of tasks to initialize execution environments, including locating executable files, loading dynamic libraries, reading configuration settings, etc. All above tasks are subject to file hijacking attacks.

❻ **Usage (28, 10.4%).** Compared to other stages, actions in the usage stage are program specific and they can change significantly for different software programs, depending on the functionality provided by the program. For instance, a text editor may create, modify, and delete files as the user edits and saves documents. Similarly, an email client may create and delete files as it downloads and stores emails.

Although the number of file operations during the usage stage is much larger than in the other stages, FHVulns are more frequently observed in the installation and starting up stages instead.

E. Threat Model and Impact

Most FHVulns can only be exploited locally, but there are a few FHVulns that can also be exploited remotely [16]. In our threat model, we assume that the attacker has authenticated into the operating system with normal user privileges. As a result, the attacker has permissions to create files or file paths in some directories (e.g. C:\ProgramData in Windows and /tmp in Linux). As shown in Fig. 2, the attacker is able to control (RQ1) certain files or file paths. Then, the victim process (program) will interact (RQ2) with the hijacked files or file paths in software lifecycle (RQ3). By exploiting FHVuln, the attacker can attain the privileges of another user (horizontal privilege escalation) [18] or even escalate privileges to become a root/administrator (vertical privilege escalation) [13].

Out of the collected FHVulns, 86.6% were rated as critical or high based on their Common Vulnerability Scoring System (CVSS) score [40], which indicates the severity of the vulnerability. In comparison, out of all 52,719 vulnerabilities, only 56.8% were rated as critical or high, highlighting the greater impact of FHVulns on security. Based on the sensitive operations (Section II-C) of the victim program, the impact of FHVulns can be mainly categorized into two types: code execution (e.g process creation) and data corruption (e.g deleting). Furthermore, it was observed that the number of FHVulns was much higher in Windows compared to Unix-like systems. The reasons for this discrepancy will be explored in detail in Section VII-A.

III. The Methodology of Jerry

A. Overview

Fig. 3 overviews our detection tool Jerry at a high level. The tool processes the input target program together with a corresponding configuration file, which specifies how to interact with the target program at different stages in its lifecycle. Details of configuration files are described in the website of the tool [55]. The tool monitors execution of the target system at each lifecycle stage and reports all discovered vulnerabilities from the target program, together with key events triggering the vulnerabilities.

As shown in Fig. 3, Jerry employs a four-step iterative process to detect FHVuln. The intuition behind is to trigger as many program behaviors (execution traces) as possible. ❶ The *Event Trace Generator* executes the target program at each stage and records executed file operation traces. ❷ The *FHVuln Detector* examines each execution traces and a FHVuln will be reported if the trace performs dangerous operations on hijacked files. ❸ The *Path Pool Maintainer* collects files encountered in the event trace and puts them into the path pool. In this step, Jerry also checks if the file refers to a normal file or a directory. ❹ The *Path Hijacker* tries to hijack and create each file in the path pool. In this regard, the path hijacker acts as an attacker with no administrator/root privilege to mimic the hijacking actions. Since new files were created by the *Path Hijacker*, the *Event Trace Generator* is triggered again to discover more execution traces. This iterative process repeats until a fixed point where no new file is encountered.

Algorithm 1 describes the entire workflow of Jerry. In the *main* function (lines 27 – 42), Jerry first splits the *usage* stage into several sub-stages consisting of UI interaction sequences(lines 28 –31). For each stage, the loop (lines 32 – 41) performs the iterative process until the path pool does not change, i.e., no new paths is found.

B. Event Trace Generator

The event trace generator in Jerry (lines 1 – 6 in Algorithm 1) interacts with the target program to trigger as many different execution traces as possible. Observation 3 indicates that FHVulns can be triggered at any stage of the software lifecycle. At distinct stages, the target programs are interacted differently. Specifically, Jerry leverages the package manager to interact with the target program during the installation, uninstallation, updating, and repairing stages. For the starting up and usage stages, Jerry employs a customized user interface (UI) explorer to interact with the target program.

All stages, except for the usage stage, do not require complex interactions with the target program. For instance, for the installation stage, the package manager can automatically

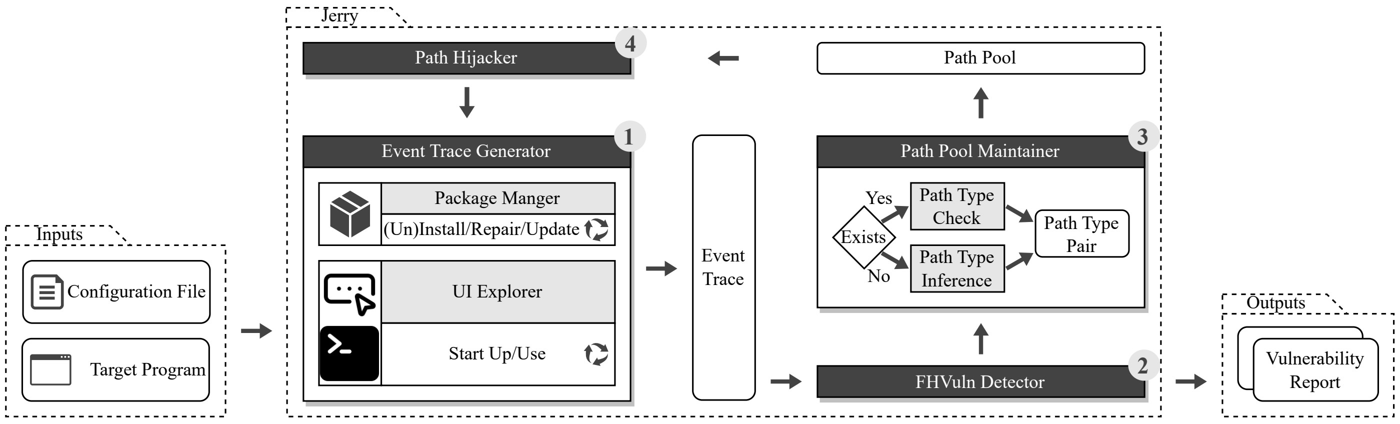
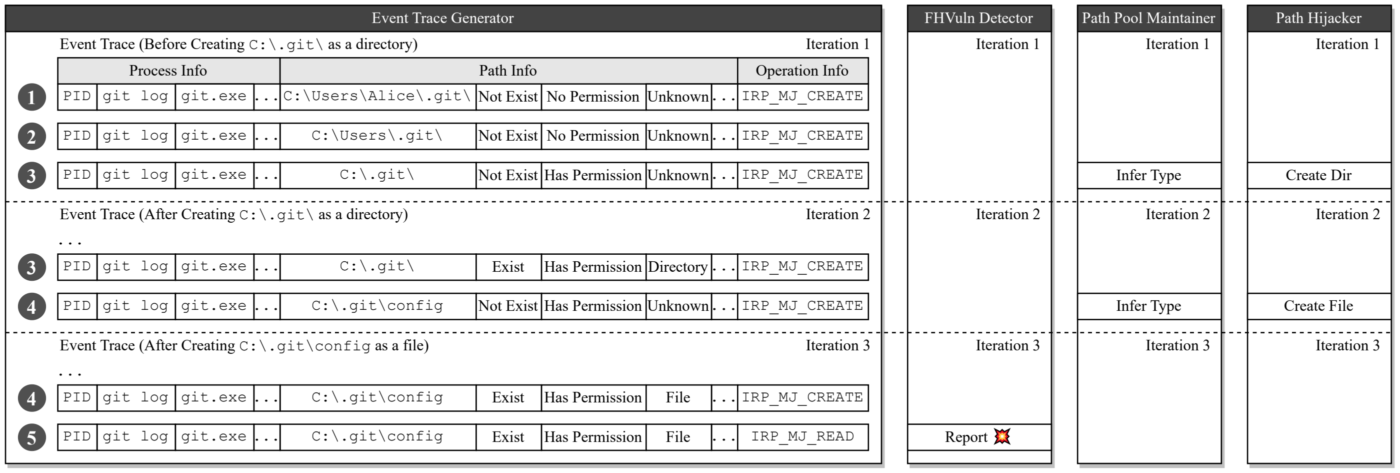


Fig. 3: Overview of JERRY.



	Process Info				Path Info					Operation Info
1	PID	git log	git.exe	...	C:\Users\Alice\.git\	Not Exist	No Permission	Unknown	...	IRP_MJ_CREATE
2	PID	git log	git.exe	...	C:\Users\.git\	Not Exist	No Permission	Unknown	...	IRP_MJ_CREATE
3	PID	git log	git.exe	...	C:\.git\	Not Exist	Has Permission	Unknown	...	IRP_MJ_CREATE
3	PID	git log	git.exe	...	C:\.git\	Exist	Has Permission	Directory	...	IRP_MJ_CREATE
4	PID	git log	git.exe	...	C:\.git\config	Not Exist	Has Permission	Unknown	...	IRP_MJ_CREATE
4	PID	git log	git.exe	...	C:\.git\config	Exist	Has Permission	File	...	IRP_MJ_CREATE
5	PID	git log	git.exe	...	C:\.git\config	Exist	Has Permission	File	...	IRP_MJ_READ

Fig. 4: A demonstration of how JERRY detects the vulnerability in Fig. 1.

invoke the installation program without extra user interactions. However, the usage stage may involve complex command line options or graphical user interface (GUI) interactions. For target programs with command line interfaces, JERRY acquires the command line options from the configuration file and generates different option combinations to use the program. Take the Git client as an example, JERRY can generate commands like git status or git log according to the configuration file. For target programs with graphical user interfaces (GUI), JERRY uses an automated GUI testing tool [89] to interact with the program and performs simple actions such as button clicking. As such, JERRY may miss program behaviors requiring complicated user interactions, suggesting false negatives.

JERRY monitors target program execution and records file access events. Fig. 4 shows an example of the event traces for the vulnerability in Fig. 1. Each event provides detailed information describing the accessing process, the accessed files, and the performed operation. For the accessing process, we record its process ID, its command line option (or GUI events), and the executed program. For the accessed files, we record file names, their existence and permissions (whether can be manipulated by the path hijacker or not), and their types (directory or file). JERRY evaluates whether files or file paths are susceptible to hijack by examining the access control list (ACL) [31]. It verifies whether attackers possess the necessary permissions to modify the content of an existing file. In the scenario of a non-existent file path, JERRY relies on the ACL of its existing parent directory to determine the feasibility of creating files or file paths within it. The performed operation is the corresponding windows driver API to access the file. All above information are included in the vulnerability report to help reproduce the bug.

C. FHVuln Detector

The FHVuln detector (lines 7 – 12 in Algorithm 1) analyzes event traces to identify FHVulns. If an event performs dangerous operations over a hijacked file, it is considered as vulnerability and JERRY will generate a vulnerability report recording the event trace with the triggering event highlighted.

In Fig. 4, event ❺ in iteration 3 performs the IRP_MJ_READ [26] operation on C:\.git\config, which is a hijacked file. Hence, JERRY reports it as a vulnerability. Note that event ❹ in iteration 3 is not a bug-triggering event since the operation IRP_MJ_CREATE [25] only opens a stub to the object pointed by the path.

The impact of FHVulns is closely associated with sensitive operations. Of the six categories of sensitive operations, the impact of five can be directly determined by the operation itself. Process creation and image loading enable arbitrary

Algorithm 1: The workflow of JERRY

```
Input: $p$: the target program under test
Input: $c$: the corresponding configuration file
Output: $\mathcal{R}$: the set of vulnerability reports
1  def get_event_trace(p,c,stage):
2      if $stage \in \{INSTALL, UNINSTALL, REPAIR, UPDATE\}$ then
3          $trace \leftarrow run\_package\_manager(p, c, stage)$;
4      else
5          $trace \leftarrow interact\_with\_ui(p, c)$;
6      return $trace$;
7  def detect_fhvuln(trace):
8      for $event \in trace$ do
9          if $is\_vulnerable(event)$ then
10             $r \leftarrow generate\_report(trace)$;
11             $\mathcal{R} \leftarrow \mathcal{R} \cup \{r\}$;
12             return $r$;
13 def update_path_pool($\mathcal{P}$, trace):
14     $\mathcal{P}' \leftarrow collect\_paths(trace)$;
15     for $path \in \mathcal{P}'$ do
16         if $path \notin \mathcal{P}$ then
17             if $exist(path)$ then
18                 $type \leftarrow check\_type(path)$;
19             else
20                 $type \leftarrow infer\_type(path)$;
21             $path.type \leftarrow type$;
22             $\mathcal{P} \leftarrow \mathcal{P} \cup \{path\}$;
23 def manipulate_path($\mathcal{P}$):
24     for $path \in \mathcal{P}$ do
25         if $\neg exist(path)$ then
26             $create\_file\_or\_dir(path)$;
27 def main(p , c ):
28     $\mathcal{R} \leftarrow \emptyset$ ;
29     $c.stages \leftarrow \{INSTALL, UNINSTALL,$
30     $REPAIR, UPDATE, START\_UP\}$;
31     $c.stages \leftarrow c.stages \cup explore\_ui\_usage()$ ;
32     for $stage \in c.stages$ do
33         $trace \leftarrow [\ ]$;
34         $\mathcal{P} \leftarrow \emptyset$;
35         do
36             $\mathbb{P} \leftarrow \mathcal{P}$;
37             $trace \leftarrow get\_event\_trace(p, c, stage)$;
38             $\mathcal{R} \leftarrow \mathcal{R} \cup \{detect\_fhvuln(\mathcal{R}, trace)\}$;
39             $update\_path\_pool(\mathcal{P}, trace)$;
40             $manipulate\_path(\mathcal{P})$;
41         while $\mathbb{P} \neq \mathcal{P}$ ;
42     return $\mathcal{R}$ ;
```

code execution, while creating, moving, deleting operations result in data corruption. Only FHVulns related to reading operation require manual analysis, as reading data alone may not necessarily pose an actual security impact.

D. Path Pool Maintainer

The path pool maintainer (lines 13 – 22 in Algorithm 1) infers the types (directory or file) of newly encountered files and adds them to the path pool. It is trivial to acquire the type of an already existing file by simply querying the file. However, it is tricky to analyze whether a non-existent file is a directory or a normal file. Hence, JERRY applies a simple heuristic to infer file types according to their usages.

The heuristic is based on the observation that when accessing a normal file, programs commonly check the existence of its parent directory while such a check is unnecessary when accessing a directory. For instance, in Fig. 4, event ❸ in iteration 1 accesses the C:\.git\ path without checking the parent's existence. Hence, the path C:\.git\ is regarded as a directory. In contrast, event ❹ in iteration accesses the path C:\.git\config after opening a stub for C:\.git\. Thus, it is inferred as a file. In addition to the above heuristic, JERRY also applies other heuristics such as using common file extensions (e.g., exe and dll) to recognize typical files. If a path is involved in directory-specific operations (such as children traversal), then it is identified as a directory.

As a fallback, JERRY also employs a trial-and-error mechanism [4] to handle paths whose types cannot be inferred. An encountered path with unknown type is by default considered as a file. If the file is used by file-specific operations later on, then the guess is correct. Otherwise, if the path is accessed by directory-specific operations, the guess is wrong and the path hijacker will create a directory instead.

E. Path Hijacker

The path hijacker (lines 23 – 26) mimics the behavior of an attacker by creating and hijacking non-existent files for the target program. For exe and dll files, JERRY replaces them with manually crafted files. For other file types, it utilizes a specially created blank file consisting of newline and space characters for hijacking. This is because JERRY aims to ensure that the program runs as possible even after the file has been hijacked. If the file is accessed by creating, moving or deleting operations, the path hijacker will create a symbolic link pointing to a special location (e.g. C:\symbolic\) for monitoring.

To act like an attacker, the path hijacker is a standalone process in JERRY, which executes with the permissions of a normal user without root or administration rights.

IV. IMPLEMENTATION

We implemented JERRY on Windows. The implementation consists of 4.5k lines of C code based on the Microsoft Windows Driver Kit, together with 2.5k lines of Python code. Instead of elaborating every step of JERRY in detail, we focus on the implementation details of the event trace generation step since this step involves many external tools to interact with the target program and monitor its execution.

Event Trace Generator. The primary objective of this module is to automate the installation, uninstallation, updating, repairing, starting up and usage of targeted programs. We utilized Chocolatey [65], a command line manager to install, uninstall and update software. For repairing, we monitor the behavior of installation and implement it for software packaged by Windows Installer [57] We start up the target software by directly invoking the program in the command line.

To automate the use of targeted programs, JERRY supports command-line interface (CLI) and graphical user interface (GUI) in a different manner. For software with command-line interface, JERRY automates its usages by randomly combing

[4]For the clarity of presentation, this trial-and-error mechanism is not reflected in Fig. 3 and Algorithm 1.

distinct command-line options in the corresponding configuration file. For software with GUI interface, we utilize the UIAutomation tool [89] to automate button click events.

Event Monitor Module. We have developed a monitoring tool at the kernel driver level using the Minifilter framework [73] to track the events of files. JERRY uses interfaces [24] provided by Minifilter to perform hooking before each file operation function. If the target file exists, JERRY records its permissions and path. Otherwise, when the parent directory of the file has weak permissions, JERRY records the permission and path of the parent directory.

V. EVALUATION

I would not be at all surprised if many holes remain.
— A Git maintainer commented on CVE-2022-41953

A. Evaluation Setup

We evaluated the effectiveness and efficiency of JERRY by comparing it against three baseline tools in detecting known vulnerabilities, and subsequently applying the tool to detect new vulnerabilities in widely used real-world projects.

Benchmarks. We evaluated JERRY with two distinct benchmark suites: the "*known vulnerability suite*" and the "*unknown vulnerability suite*". ❶ *The known benchmark suite* is extracted from the 268 CVEs studied in Section II, where **51** reproducible CVEs are included. Among the rest of the 217 unreproducible CVEs, 147 lack sufficient reproducing instructions, 52 only exist in unavailable vulnerable versions, and we fail to install the target programs for the remaining 18 CVEs. The details of each CVE in the known benchmark suite, including its CVE ID, affected programs, triggering stages, and triggering operations, are listed in TABLE A1 in the appendix. ❷ *The unknown benchmark suite* consists of **438** popular software programs. We collected a total of 663 software packages, including all software with more than 100,000 downloads on Chocolatey [65], and all pre-installed Windows software from top PC vendors including DELL, Lenovo, and HP. Among the 663 software packages, 225 packages were excluded because they either had not been updated after 2020, or could not be successfully installed. Finally, we obtained a total of **438** software programs. All programs in this suite are widely-used applications and vulnerabilities reported in the suite will have significant security impacts. Details of the *unknown vulnerability suite* will be revealed on the website [55].

Baselines. We compared JERRY against three base line tools: PrivescCheck, JERRY-Crassus and LPET. PrivescCheck [49] is a static tool which detects executable files with weak permission by scanning the access control metadata provided by Windows. Crassus [36] detects FHVulns by analyzing event traces monitored by ProcMon [50], identifying existing or non-existent exe files, dll files and openssl.cnf which can be hijacked. However, Crassus does not have a built-in module to automatically trigger the events. To enable a meaningful comparison, we extended Crassus by incorporating the event trace generator module and replaced our monitor with ProcMon, named it JERRY-Crassus. LPET [72] is a dynamic tool designed to discover FHVulns when using software. We cannot directly compare JERRY with LPET due to the unavailability of the LPET tool and the limited information on its hijacking verification mechanism, making it challenging to replicate. However, LPET shares a similarity strategy with Crassus, and we can estimate the effectiveness of LPET for qualitative analysis based on the strategic differences between these two tools.

TABLE I: **The Overall Evaluation Results on Known Vulnerabilities.** The column # reported means the number of results reported by corresponding tool.

Tool	# reported	TP	FP	FN	Precision	Recall
PrivescCheck	34	20	14	31	58.8%	39.2%
JERRY-Crassus	44	37	7	14	84.1%	72.5%
JERRY	**50**	**50**	**0**	**1**	**100.0%**	**98.0%**

Metrics. In this evaluation, we adopt two commonly used metrics, namely positive predictive value (a.k.a precision) and true positive rate (a.k.a recall), to compare the effectiveness of JERRY against the two baseline tools. Eq. (1) and (2) show the equations to compute precision and recall, where TP, FP and FN represent true positives, false positives and false negatives, respectively.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

Evaluation Configuration. Installation and updates of a software will change the system state, which may not be undone. To avoid such impacts, we conduct our experiments on a virtual machine hosted on a Lenovo laptop with 14 Intel Core i9 processors, 32 GB of memory and a 512 SSD drive, running on Windows 11. The virtual machine is a VMWare image configured with 4 CPU cores, 16GB of memory, and a 200 GB hard drive, also running Windows 11.

B. Effectiveness on Known Vulnerabilities

We evaluated the effectiveness of JERRY on the 51 known vulnerabilities by comparing it with two existing tools, PrivescCheck and JERRY-Crassus. The overall evaluation results are shown in TABLE I. We also give the detailed results for each CVE in the appenix (TABLE A1).

Overall Results. JERRY significantly outperformed both baseline tools with respect to precision and recall. It successfully identified 50 out of 51 vulnerabilities without any false positives. In other words, the precision and recall of JERRY are 100.0% and 98.04% (50/51), respectively. In comparison, PrivescCheck and JERRY-Crassus were less effective. Specifically, PrivescCheck (resp. JERRY-Crassus) detected 20 (resp. 37) out of 51 vulnerabilities with a precision of 58.8% (resp. 84.1%) and a recall of 39.2% (resp. 72.5%).

In addition, we analyzed the vulnerabilities commonly detected by each tool to further evaluate the effectiveness of JERRY. As shown in TABLE A1, the vulnerabilities detected by PrivescCheck and JERRY-Crassus can all be identified by JERRY. Besides, there are 13 vulnerabilities uniquely detected by JERRY, whereas no vulnerabilities can be uniquely detected by any baselines. The result echoes the high effectiveness of

JERRY in terms of the number of identified known vulnerabilities.

FN Analysis for JERRY. JERRY only missed one vulnerability, which all other baseline tools also missed. After investigation, we found that the missed vulnerability was CVE-2020-26284 [15], which existed in the file parsing module of the Go language standard library. The vulnerability required parsing specific content and invoking the relevant module to trigger it.

FP/FN Analysis for PrivescCheck and JERRY-Crassus. We also analyzed the false positives/false negatives in PrivescCheck and JERRY-Crassus. For false positives, PrivescCheck reported 14 false positives, which resulted from its method of scanning the parent directory permissions when encountering an executable. In particular, if the parent directory has weak permissions, it will report a potential FHVuln as a result of DLL hijacking. However, it is possible that the executable is not loading any missing DLLs from that directory, which results in false positives. In addition, for non-existent files, JERRY-Crassus reported all the recorded dll files and openssl.cnf where the parent directories have weak permissions as FHVulns. However, it did not verify successful hijacking operations, leading to seven false positives. Expanding JERRY-Crassus to include other file types may increase true positives, but also results in much more false positives without post-hijack operation verification.

For false negatives, excluding the vulnerability (i.e., CVE-2020-26284) mentioned above that none of the three tools could detect, PrivescCheck failed to detect 30 vulnerabilities in total. 20 of these vulnerabilities resulted from its neglect of certain operations that could cause vulnerabilities, such as reading (10), creating (5), deleting (4), and moving (1). Moreover, PrivescCheck missed one vulnerability (i.e., CVE-2020-15145) because it only scanned the permissions in the installation directories listed in the registry. The remaining nine vulnerabilities were overlooked because it only considered the vulnerabilities during the installation phase. Similarly, JERRY-Crassus missed 13 vulnerabilities, with CVE-2020-26284 excluded. Among them, 10 false positives were reported because the tool did not take into account the impact of creating (5), deleting (4), and moving (1) operations. In addition, it only scanned a limited range of openssl.cnf, resulting in missing 3 out of 10 vulnerabilities related to read operations.

Comparison with LPET. LPET shares similar detection rules with JERRY-Crassus but extends the rule set to include bat files while excluding the openssl.cnf file. As LPET contains a verification module, it can detect 33 FHVulns without any false positives.

C. Effectiveness on Unknown Vulnerabilities

To verify if JERRY is able to identify unknown vulnerabilities, we further applied JERRY to 438 real-world projects. It found 339 zero-day vulnerabilities in 176 out of 438 software programs with 21 false positives. Among the 339 identified vulnerabilities, 126 were attributed to new paths discovered during the testing process. Some of these zero-day vulnerabilities were found in high impact software programs with over 500 million downloads, such as Adobe Reader DC, Chrome, Visual Studio, Git for Windows, VMware and so on. So far, a total of 84 of the identified vulnerabilities have been confirmed (and fixed). Among the 84 confirmed vulnerabilities, 51 were assigned CVE IDs, and 21 were rewarded with a total of $83,400 bug bounties. TABLE II shows part of the real-world vulnerabilities that were reported by JERRY and confirmed by developers. Meanwhile, we also applied the two other baseline tools to explore these projects, and the results were not satisfactory. Specifically, PrivescCheck only found 39 vulnerabilities (11.5% of JERRY), while JERRY-Crassus detected 143 vulnerabilities (42.2% of JERRY). In all 143 vulnerabilities detected by JERRY-Crassus, 6 FHVulns result from issues with openssl.cnf, 124 FHVulns are due to overly-permissive dll files, and 13 FHVulns are related to exe files. Since there are no FHVulns related to bat files in our evaluation, LPET will report 137 FHVulns, excluding the 6 related to openssl.cnf. This demonstrates the significant advantage of JERRY over PrivescCheck, JERRY-Crassus and LPET in discovering zero-day FHVulns.

FP Analysis for JERRY. JERRY has reported 21 false positive issues, all of which are related to read operation. For example, we discovered instances where a program reads file content without utilizing it or where the data is rendered useless. For instance, the Azure CLI leverages the "knack" package to load configuration settings, utilizing a bottom-up search policy to locate the ".azure/config" file. This process involves reading the contents of all identified files during initialization. However, the configuration settings used can be controlled by a parameter, making the content of the configuration file useless. It is worth noting that some configuration files may not be suitable for Windows systems due to the use of Linux file paths, which may lead to hijacking. Additionally, it is important to mention that certain configurations can only be utilized in Linux systems (e.g., C:\etc\gcrypt\fips_enabled).

Case Studies. We present two examples to demonstrate how the user interface exploration and path inference features of JERRY facilitate the detection of critical FHVulns.

Case-A. While testing Visual Studio, the UI Explorer of JERRY initially clicks the button labeled as "Git" and opens a menu list. Subsequently, from the menu list, the UI Explorer selects the item labeled as "Open git in cmd". Visual Studio then searches for git.exe in CWD, which has weak permissions. JERRY detects this file operation and places a malicious git.exe in CWD. Moreover, it confirms that the malicious git.exe is successfully loaded into Visual Studio. JERRY automatically documents the detailed steps of the entire exploit process, as well as the potential security impacts (i.e., arbitrary code execution) for this vulnerability. We reported this FHVuln along with the generated steps to MSRC and received confirmed feedback quickly. Due to its severity, MSRC rewarded us with **$30,000**.

Case-B. Gem [52] is a package manager of Ruby programming language. While starting up, Gem attempts to access a file or directory with the path C:\ProgramData\gemrc. As the path does not exist, Gem stops to access the file or directory and continues executing the rest of the code. During testing, JERRY detected this behavior and attempted to create a malicious file with the path C:\ProgramData\gemrc for Gem. Gem was able to access the file and continued executing the rest of the code. As the rest of the code includes a reading operation, the malicious file was successfully loaded

TABLE II: **Part of Real-world FHVulns Detected by Jerry and Confirmed by Developers.** The abbreviations Ins, Uni, Up, Rep, SU and Us represent Installation, Uninstallation, Updating, Repairing, Starting Up and Usage, respectively. The abbreviations PC, IL, RD, CT, MV and DT represent Process Creation, Image Loading, Reading, Creating, Moving and Deleting, respectively. The Symbol "★" indicates that the corresponding software is pre-installed.

No.	Software Name	# Download	Stage	Operation	Status
1	Adobe Reader DC	465,124,436	Ins	CT	Confirmed
2	Adobe Reader DC	465,124,436	Uni	DT	Confirmed
3	Chrome	97,544,900	Ins	CT	CVE-2023-2939
4	Chrome	97,544,900	Ins	RD	Fixed
5	Firefox	40,111,618	Uni	DT	CVE-2023-4052
6	JRE8	24,394,580	Ins	CT	Fixed
7	Visual Studio	10,670,579	Ins	CT	CVE-2023-21567
8	Visual Studio	10,670,579	Us	PC	Confirmed
9	Git for Windows	10,256,420	Ins	PC	CVE-2022-31012
10	Git for Windows	10,256,420	SU	RD	CVE-2022-24765
11	Git for Windows	10,256,420	Us	PC	CVE-2022-41953
12	Git for Windows	10,256,420	Us	PC	CVE-2023-23618
13	Git for Windows	10,256,420	SU	PC	CVE-2023-29012
14	Git for Windows	10,256,420	SU	RD	CVE-2023-29011
15	Openssh for Windows	5,884,392	SU	RD	CVE-2022-26558
16	Sysinternals	5,859,086	SU	IL	Confirmed
17	Nodejs	5,353,689	SU	RD	Confirmed
18	DellCommandUpdate	4,210,082	Ins	DT	CVE-2023-23698
19	DellCommandUpdate	4,210,082	Ins	CT	CVE-2023-28071
20	Visual Studio Code	4,172,599	Us	PC	CVE-2022-38020
21	Dotnet SDK	3,016,753	SU	IL	CVE-2023-28260
22	Dotnet SDK	3,016,753	Us	IL	CVE-2023-33126
23	Dotnet SDK	3,016,753	Us	RD	CVE-2023-33135
24	iTunes for Windows	2,382,592	SU	IL	CVE-2023-32351
25	Dropbox	2,290,276	Uni	DT	Confirmed
26	Azure Cli	1,197,993	SU	IL	Fixed
27	Gvim	1,897,408	Ins	PC	CVE-2022-37172
28	Php	1,665,675	Ins	PC	CVE-2022-45307
29	Azure pipeline agent	1,376,209	Ins	PC	CVE-2022-45306
30	Ruby	1,369,541	Ins	PC	CVE-2022-45301
31	Ruby	1,369,541	SU	RD	Fixed
32	StrawberryPerl	1,187,107	Ins	PC	CVE-2022-36564
33	Intel Software 1	945,347	Ins	CT	Fixed
34	Intel Software 1	945,347	Ins	DT	Fixed
35	VMWare Tools	819,878	SU	RD	CVE-2022-22977
36	VMWare Tools	819,878	SU	DT	Fixed
37	Msys2	683,078	Ins	PC	CVE-2022-37172
38	Bazel	314,066	SU	RD	Confirmed
39	MySQL	278,425	SU	PC	CVE-2022-39403
40	MySQL	278,425	SU	RD	CVE-2022-39402
41	MySQL	278,425	SU	RD	CVE-2022-39404
42	Github Cli	226,930	SU	PC	Fixed
43	ZeroTierOne	177,047	SU	IL	CVE-2022-1316
44	WPS Office	122,094	Ins	IL	Fixed
45	WPS Office	122,094	Ins	IL	Fixed
46	WPS Office	122,094	SU	IL	Fixed
47	Intel Software 2	★	Ins	CT	Fixed
48	Intel Software 3	★	Ins	PC	Fixed
49	Intel Software 4	★	Ins	PC	Fixed
50	Intel Software 5	★	SU	IL	Fixed
51	Dell Command Intel vPro	★	Uni	DT	CVE-2023-23697
52	Dell Command Integration Suite	★	Uni	DT	CVE-2023-24572
53	Dell Command Monitor	★	Uni	DT	CVE-2023-24573
54	Dell Command Monitor	★	Uni	DT	CVE-2023-28049

TABLE III: **Average Running Time (s) of Jerry-NoInfer and Jerry.** The columns Ins, Uni, Upd, Rep, and SU mean Installation, Uninstallation, Updating, Repairing, and Starting Up, respectively.

Tool	Ins	Uni	Upd	Rep	SU	Usage
Jerry-NoInfer	8039.4	1417.6	3871.9	1206.9	556.17	33.8
Jerry	1128.1	414.2	893.5	254.7	115.5	15.8

by Gem. Since C:\ProgramData\gemrc contains a URL to the central repositories, this FHVuln affects thousands of software packages built with Ruby. In other words, attackers can hijack links that lead to software package downloads. By redirecting these links to repositories under their control, attackers can plant malicious code in the software within the repositories. This nefarious activity has the potential to cause significant damage to the software supply chain.

However, all existing methods are unable to detect it as they can only be performed on existing files.

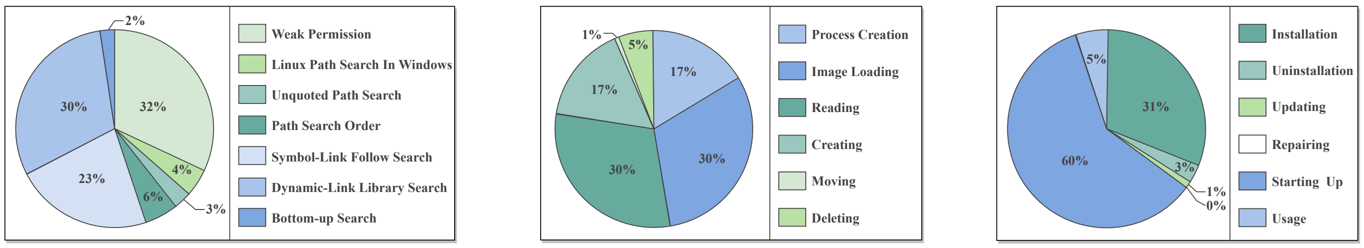
D. *Efficiency*

In addition to the effectiveness of Jerry, another important evaluation metric is the time it takes to detect these vulnerabilities. In this section, we evaluated the efficiency of Jerry by performing an ablation study on all 489 software programs from the benchmark and real world. In particular, we considered Jerry-NoInfer, which does not use our proposed path type inference and tested these paths which cannot decide whether file or directory by our heuristics directly one by one. Our experience shows that the strategies we used can increase speed without reducing false positives.

The results of this evaluation are shown in TABLE III. As we can see here, in terms of average running time, Jerry achieved at least 2.14 faster in the usage stage and 7.13 faster in the installation stage because there are only a few paths that can be hijacked in the usage stage, but in the installation stage, there are much more paths that can be hijacked than other stages. Therefore, overall, Jerry significantly reduces its running time while ensuring the same effect.

E. *Responsible Disclosure*

As demonstrated in Section II, FHVulns can lead to severe consequences, such as arbitrary code execution, privilege escalation, data loss, and open redirect attacks. In light of these risks, we took the responsibility of disclosing all the 339 vulnerabilities we identified in 176 software programs with a detailed report and Proof of Concept (PoC) to vendors of the affected software. We reported 48 of these vulnerabilities via third-party vulnerability coordination platforms, including HackerOne [43], Bugcrowd [34], and Intigriti [46]. Furthermore, we reached out to 107 companies or organizations to report 291 vulnerabilities through their dedicated email addresses and forms for reporting security vulnerabilities. As per responsible disclosure practices, we will not publicly release any unfixed vulnerabilities until the developers address them. It is worth noting that all vulnerabilities with detailed information in our paper have been fixed by developers. At present, 84 of the identified vulnerabilities have been either confirmed or fixed, and 51 CVE identifiers have been assigned to these issues.



(a) Origins (b) File Operations (c) Stages

Fig. 5: Distribution of new FHVulns on different origins, file operations, and stages.

VI. Analysis of New FHVulns

To provide guidance for future studies on FHVulns, we conducted an empirical analysis of the 339 newly detected vulnerabilities to re-address RQ1 and RQ2.

A. Distribution of New FHVulns

Origin Distributions. Fig. 5(a) shows the distribution of new FHVulns based on their origins. The chart clearly indicates that weak permissions of existing files was the most common origin of FHVulns, accounting for more than one-third (32%) of all occurrences. Dynamically loaded libraries was the second most frequent origin, accounting for 30% of the FHVulns. The remaining 38% was distributed among several other categories, including Linux paths on Windows (4%), unquoted paths (3%), path search order (6%), and bottom-up search (2%).

Operation Distributions. Fig. 5(b) illustrates the distribution of various file operations. The chart shows the most prevalent types of file operations are reading (30%) and image loading (30%), followed by process creation (17%) and creating (17%). On the other hand, deleting (5%) and moving (1%) are the least frequent operations.

Stage Distributions. Fig. 5(c) presents the distribution of various stages. The starting up stage accounts for the majority of FHVulns, representing 60% of the total occurrences. The installation stage follows with 31%, while the remaining stages, namely usage, uninstallation, and updating, account for only a minor fraction of 9%.

B. New Findings

Our investigation has yielded two noteworthy findings that differ from the existing observations and can provide guidance for future studies.

> ***Finding 1*****:** The bottom-up search strategy, a software-tailored search strategy that led to eight new FHVulns in fundamental software like Git and Dotnet SDK that had gone unnoticed for 18 years, has not received extensive research attention.

Observation 1 reveals that five search strategies account for the majority (89.9%) of hijacked files, which also applies to newly detected 223 FHVulns (65.8%). However, all five strategies listed in the observation are built-in search strategies of the operating systems. After investigating the newly detected FHVulns, we found a new type of search strategy, the bottom-up search strategy, which is software-tailored. The bottom-up search strategy employs an iterative approach, starting the search in the current working directory and progressively navigating upwards towards parent directories. Unlike built-in search strategies, this new type of strategy is OS-neutral and also results in eight FHVulns in our real-world experiment results. For instance, the vulnerability illustrated in Fig. 1 remained concealed in Git for 18 years, from the initial release of Git. Another similar FHVuln, CVE-2023-33135 [75], was found in Dotnet SDK [48], which is the core functionality required to create Dotnet projects. Dotnet uses its own search strategies to locate the *.config* directory from the CWD to the root directory. When Dotnet is executed in a directory without a Dotnet project, it will eventually locate C:\.config , which can be hijacked by the attackers as C:\ is world-writable. By controlling the configuration in C:\.config\dotnet –tools.json, attackers can execute arbitrary code by making Dotnet download malicious tools from websites.

> ***Finding 2*****:** Reading operations result in more FHVulns than we studied (**30.4%** vs **7.1%**), and they are more dangerous than we think.

Observation 2 reveals that there are six types of sensitive operations that can be exploited, which aligns with our findings on new vulnerabilities. However, a notable difference exists between existing and new FHVulns. Specifically, among existing FHVulns, only 7.1% of them involved reading operation, with 89.5% (17/19) of these cases attributed to reading the hijacked openssl.cnf file in the OpenSSL library. In contrast, for new FHVulns, the percentage of reading operation is as high as 30.4% (103/339), representing a significant increase. Furthermore, the types of FHVulns involving reading operation in new FHVulns are more diverse than those in existing ones. Hijacking files with reading operation can lead to severe consequences, including arbitrary code execution (as illustrated in Fig. 1), URL redirection, and sensitive information disclosure. For example, CVE-2023-2939 identified by Jerry enables attackers to hijack the *settings.dat* file, which contains a URL that Chrome uses to transmit diagnostic information to its servers. Attackers can replace the URL with their own URL to obtain the diagnostic information. Thus, reading operation poses greater risks than previously considered and requires more attention in the future, as will be discussed in Section VIII.

VII. Discussions

It is actually incomplete on Windows: a malicious user can create C:\.git !

— A Git developer commented on CVE-2022-24765

A. What is the reason for the higher number of FHVulns in Windows compared to Unix-like systems?

We conducted a thorough analysis of FHVulns in Windows and Unix-like systems (Linux and Mac) [56] and identified two root causes based on the design principles that govern these operating systems.

More Existing Directories with Weak Permissions. In Unix-like system, only the /tmp directory is world-writable. Its purpose is to accommodate data that is only required for a brief period, such as temporary files generated by programs running on the system. In Windows system, besides the directory C:\Windows\Temp which is like /tmp, there are another two directories with weak permissions that lead to FHVulns, i.e., C:\ProgramData and C:\ root directories. While C:\ProgramData is designed to be used for storing shared program data and configuration files, some software programs may mistakenly store sensitive configuration files and executable files in this directory. This can increase the risk of FHVulns occurring. Moreover, the C:\ root directory has weak permissions due to historical reasons [4], [82]. This has resulted in numerous hijacking issues. When new folders are created in C:\, they inherit the permissions of C:\ and become world-writable, which further exacerbates security risks.

More Searched Directories with Weak Permissions. Many search strategies, like bottom-up search strategy, path search order, symbolic links and dynamically loaded libraries, are applicable to both Unix-like systems and Windows systems. However, these strategies are associated with relatively fewer FHVulns in Unix-like systems compared to Windows. One reason for this is that in Windows, when searching for executable files, the operating system looks in CWD as well as in directories specified in the PATH environment variable. In contrast, Unix-like systems limit their search to directories listed in the PATH variable only [3]. In terms of symbolic links, Windows has special directory links known as junction points [27], which do not undergo strict permission checks during creation. This feature is not present in Unix-like systems. Additionally, Windows searches for dynamically loaded libraries (DLLs) within the software program directory and the CWD, whereas Unix-like systems do not [21]. These differences mean that search-related vulnerabilities have a larger attack surface in Windows compared to Unix-like systems.

B. Why do FHVulns still widely exist?

While there have been efforts to identify FHVulns, they continue to persist in real-world software. In fact, our research identified 339 zero-day FHVulns in 438 popular software programs, including Git, Adobe, and Chrome. After conducting a detailed analysis, we concluded that there are two main categories of reasons contributing to the widespread existence of these vulnerabilities.

Lack of Awareness. During the process of reporting and fixing these vulnerabilities, we had in-depth discussions with the developers and identified three main reasons why they introduced FHVulns. **Firstly**, they were not aware of the existence of FHVulns. Many of them only learned about FHVulns for the first time through our reports. Interestingly, some developers initially disagreed with the threat model we proposed in Fig. 2, thinking that their software was only for single-user, until they had deeper discussions with other developers and realized that their software was for multi-user. **Secondly**, developers lacked secure usage of APIs. The operating system provides many secure API options for developers to use. For example, in the API LoadLibraryEx [47], the parameter *LOAD_LIBRARY_SEARCH* can specify the search priority to avoid malicious DLLs being loaded before the target DLLs. However, in actual development, developers may not deliberately use these parameters. **Thirdly**, developers lacked security responsibility. During the discussion, we found that some developers believed that files with weak permissions were introduced due to users' incorrect usage, and the software itself did not need to strengthen its defenses.

Lack of Effective Tool. Due to the lack of systematic research on FHVulns, existing tools have difficulty detecting FHVulns comprehensively. Based on ***Observation 1***, it can be concluded that the majority of FHVulns are caused by five search strategies, and the paths of the hijacked files are often nonexistent. As a result, existing approaches are ineffective in identifying these vulnerabilities, since they do not take into account the search strategies and generate limited event traces (Section III-B). Furthermore, these approaches have limited scope, as they only focus on a subset of file operations, leading to numerous FHVulns going undetected.

C. Who is responsible for the FHVulns?

The responsibility for the security of a file or directory with weak permissions lies with the software developer in cases where the software creates the file or directory. When such a file or directory is created by the administrator (e.g., global configure file for all users), the system administrator assumes responsibility for its security. However, if a user provides a potentially hijacked file or directory that is subsequently used by the software, the party responsible for its security is unclear. Through deep discussions with developers, most of them tend to rely on providing warnings to users when there is a risk with the current file or directory. However, some developers believe that merely warning users without taking measures to prevent or limit potential risks may be viewed as irresponsible. Developers should, therefore, consider implementing protective measures in the software. For example, to address CVE-2022-24765, Git now requires users to whitelist trusted files and directories with weak permissions. However, such defenses may be inconvenient for some users, limiting their ability to use the software. After fixing CVE-2022-24765, many users complained on social media platforms about Git's whitelist mechanism. Therefore, developers must strike a balance between protecting user safety and experience to provide secure and accessible software products. Moreover, it is recommended that developers of the Windows operating system revisit the system design to address the issue of FHVulns at the root level. However, it may be not realistic since the new design of the operating system has introduced challenges for backward compatibility.

VIII. Limitation and Future Work

It's probably much more effective to make certain git config variables require that the user that runs the command also owns the file.

— Linus Torvalds commented on CVE-2022-24765

In this section, we discuss limitation, possible improvements for JERRY and suggestions for defending against FHVulns based on our observations (Section II), new findings (Section VI-B) and the discussions (Section VII).

A. What are the limitations of JERRY?

As mentioned in Section III-C, JERRY detects FHVuln when both hijacking and sensitive operations are satisfied. However, FHVulns related to read operation can lead to false positives, necessitating supplementary manual analysis to accurately evaluate the impact. This represents a major limitation of the JERRY framework. In addition, JERRY utilizes chocolatey to automate the installation, uninstallation, and updating stage of software. When testing software not hosted on the Chocolatey website, it is necessary to implement a corresponding local script. In the usage stage, JERRY only supports simple user interactions, which may lead to false negatives.

B. Where can we make improvements to JERRY?

Mitigating False Positives and False Negatives. To improve the effectiveness of JERRY, we can employ several strategies to reduce both false positives and false negatives. First, for FHVulns that involve reading operations, JERRY does not consider the specific way in which the hijacked file content is used by the software, leading to 21 false positives. For example, JERRY reported a false positive for the libgcrypt library [69], which attempts to read the configuration file at C:\etc\gcrypt\fips_enabled on Windows. However, the configuration in this file is only used in libgcrypt for Linux [68]. In other words, even if the content of C:\etc\gcrypt\fips_enabled is read, it is not used by libgcrypt for Windows and therefore does not pose any security impacts. Therefore, we may combine taint analysis [63], [90] and program understanding [78], [87] techniques to figure out where the file content propagates.

In addition, we can employ static analysis techniques [64], [91] to detect more FHVulns by identifying potential vulnerabilities. It is challenging to trigger certain components of software that contain FHVulns through dynamic testing. Hence, static analysis can be utilized to identify suspected FHVulns along with the trigger conditions, which can guide JERRY to trigger these vulnerabilities. Furthermore, we found that some FHVulns (e.g., CVE-2020-26284) can only be triggered via specific user-provided contents. To address this issue, mutation-based techniques [61], [79] can be employed to generate diverse contents.

More Attack Vectors. In addition to files, other resources such as registries and named pipes can also breach security boundaries. For example, a study found that two service registry keys with weak permissions were present on Windows 7, and can enable attackers to obtain *SYSTEM* privilege by modifying their value [32]. Additionally, hijacked named pipes can also be used for privilege escalation [33]. Therefore, further exploration of resource objects for hijacking is worth considering. Moreover, some FHVulns can exist in multiple operating systems simultaneously. For example, software can be designed with system-independent search strategies (e.g., bottom-up search strategy in Finding 1), which can be the origins of FHVulns.

C. How to defend against and mitigate FHVulns?

After reporting the new FHVulns to developers, we engaged in many discussions with them about how to fix and mitigate FHVulns. We conducted an extensive analysis and summary of these discussions to provide software developers, system administrators, and users with effective recommendations for defending against FHVulns.

Suggestions for Software Developers. To address FHVulns, software developers should avoid using paths with weak permissions (***Requirement 1***) and verify the status of files before using them (***Requirement 2***). To block ***Requirement 1***, ❶ *Avoiding using Built-in directories with weak permissions.* Software developers should either place sensitive files to paths with strict permissions (i.e., C:\Program Files) or public paths (i.e., C:\ProgramData) while actively setting right permissions for sensitive files. For example, most public configuration files, such as system config of rubygem, are placed in public paths such as C:\ProgramData, to ensure that all users can assess and share them. Therefore, during the installation, software should create a new configuration file and assign high-level permissions to it. ❷ *Correcting API Usages.* It is necessary to read the API document carefully to ensure that they correctly use the API and its parameters. Our study revealed that the majority of FHVulns arising from the dynamic loaded libraries search strategy can be attributed to the CWD. Consequently, it is recommended that developers employ *LoadLibraryEx* instead of *LoadLibraryA* and manage the search strategy by configuring the *LOAD_LIBRARY_SEARCH* flag to prevent including CWD in the search paths. [41] Moreover, when performing sensitive actions like creating or deleting files, it is crucial to proactively set relevant parameters to prevent symbolic link resolution issue [58]. Specifically, the function *CreateFileA* requires the *FILE_FLAG_OPEN_REPARSE_POINT* parameter, as stated in the document [74]. ❸ *Design software-tailored search strategies cautiously.* Based on Finding 1, it is evident that software-tailored search strategies may lead to various FHVulns. Thus, developers need to design search strategies with caution, taking into account multiple user scenarios. To block ***Requirement 2***, ❹ *Checking file status before use.* Developers ought to verify whether sensitive files have been hijacked by checking their ownership. If the file's owner differs from that of the user, the software should either terminate its operation or prompt users to confirm whether they wish to proceed with potentially "hijacked" sensitive files.

Suggestions for System Administrators. It was observed that approximately 23 software programs in our real-world experiments were installed in directories with weak permission, and only four of them set an appropriate permissions to files. ❶ As a result, software developers should consider changing their software's default installation directory (the first suggestion for software developers), and system administrators can ensure double security by employing the custom installation mode and installing the software in a directory with high permissions.

Additionally, we found that various direct subdirectories of the root directory C:\, which were not initially present in the system, were vulnerable to hijacking due to the excessively permissive permission settings of the Windows OS during initialization. ❷ Therefore, it is advisable for system administrators to remove the write permission of *Authenticated Users* [29] group from the C:\ directory. This action can significantly decrease the likelihood of path hijacking, while having minimal side effects. (***Requirement 1***)

Suggestions for Software Users. Besides, it is crucial for software users to be aware of security risks and follow safe practices when using software. Particularly, users should avoid launching software program in paths with weak permissions, such as C:\ProgramData\. (***Requirement 1***) Additionally, before using software in publicly writable paths, users need to check for any suspicious files. (***Requirement 2***) For instance, when using software downloaded from the internet in publicly writable paths, users ought to verify its integrity by checking the checksum.

IX. Related Work

A. File-related Vulnerability Detection

Static Analysis. To detect file-related vulnerabilities, various static analysis techniques have been proposed. Some techniques [62], [70], [76] focused on identifying weak permissions in the Windows system by scanning permission metadata. PrivescCheck [49] statically scanned the permission metadata in Windows and report potential file-related vulnerabilities. Bauer et al. [59] employed association rule mining techniques to detect access control misconfiguration, while Parkinson et al. [77] utilized rule mining methods to identify incorrect permission configurations. Shaikh et al. [81] proposed a decision tree and data classification-based approach for detecting incomplete and inconsistent (allow or deny) policies in access control datasets. These methods analyzed the permissions of existing files statically to identify file-related vulnerabilities. In the context of FHVulns, these techniques can only detect the FHVulns with the first origin, neglecting the majority of the FHVulns (***Observation*** 1).

Dynamic Analysis. Many approaches employed dynamic techniques to detect file-related vulnerabilities. Spartacus [53] analyzed the traces recorded by ProcMon [50] to detect DLL hijacking or binary hijacking. Crassus [36] considered the permissions of the path when detecting file-related vulnerabilities. Researchers also focused on finding file-related vulnerabilities caused by symbolic link automatically by triggering APIs provided by Windows service [10]. Vetle [86] utilized ProcMon to monitor file system events with "NOT FOUND" result, and verified file hijacking vulnerabilities through manual testing. Similarly, LPET [72] designed a dynamic file system event monitor to detect temporary directories with weak permissions during the software installation, updating, and uninstallation stage. In the context of FHVulns, these techniques are like the *FHVuln detector* module in Jerry. They may miss out on some sensitive operations (***Observation*** 2). Moreover, most of them lack the ability to generate the event traces during different software lifecycle stages (***Observation*** 3).

B. File-related Vulnerability Exploitation

Several researchers have focused on exploiting DLL hijacking vulnerabilities while minimizing the impact on the normal functioning of software programs. As a promising approach, they have proposed DLL proxying attacks [12], [22]. Numerous studies have been conducted on security issues caused by symbolic links [1], [5]–[8], [58], [92]. Some of the researchers [9] used symbolic link testing tools [2] to dig into these vulnerabilities and investigated how symbolic links can be used to execute arbitrary code [92]. Basu et al. [58] discovered that on case-insensitive systems, symbolic links can lead to problems such as content overwriting and permission modification. These approaches can be combined with Jerry to amplify the security impacts of detected FHVulns.

C. General Hijacking Studies

Researchers have explored various types of hijacking beyond file hijacking. For instance, hijacking named pipes on Windows can result in privilege escalation, and a tool called PipeViewer [30] has been developed to identify such vulnerabilities. Account hijacking has also been studied comprehensively by researchers like Avinash Sudhodanan et al. [85]. In web security, session hijacking [60], [66] and cookie hijacking [67], [83], [84] are common types of hijacking incidents. Additionally, techniques like control flow hijacking [51], [88] and data flow hijacking [71] are often used to exploit memory corruption vulnerabilities. However, our research specifically focuses on file hijacking, which can lead to a breach of security boundaries in operating systems. This type of attack can compromise software security and allow attackers to gain elevated privileges, execute arbitrary code, and so on, making it critical to understand file hijacking.

X. Conclusion

This paper presents the first empirical investigation of FHVulns. The study observations inspired us to create a dynamic analysis tool, known as Jerry, for detecting FHVulns. To date, Jerry has identified 339 zero-day FHVulns in 438 real-world software programs. We have taken full responsibility for disclosing all of the zero-day FHVulns, of which 84 have been either confirmed or resolved, with 51 being assigned CVE identifiers. Moreover, our findings on the existing and newly detected vulnerabilities can further help to guide future work on detecting and fixing FHVulns.

Acknowledgment

The authors would like to express their gratitude to the anonymous reviewers for their helpful feedback on an earlier version of this paper. This work is partly supported by National Key R&D Program of China under Grant #2022YFB3103900, Strategic Priority Research Program of the CAS under Grant #XDCO2030200 and Chinese National Natural Science Foundation (Grants #62032010, #62202462).

References

[1] "A link to the past - abusing symbolic links on windows," https://infocon.org/cons/SyScan/SyScan%202015%20Singapore/SyScan%202015%20Singapore%20presentations/SyScan15%20James%20Forshaw%20-%20A%20Link%20to%20the%20Past.pdf, 15.

[2] "Symoliclink testing tools," https://github.com/googleprojectzero/symboliclink-testing-tools, 15.

[3] "How does unix search for executable files," https://superuser.com/questions/238987/how-does-unix-search-for-executable-files, 2010.

[4] "Windows 7 Special Directories (Folders)," https://wiki.carleton.edu/pages/viewpage.action?pageId=9961710, 2011.

[5] "Between a Rock and a Hard Link," https://googleprojectzero.blogspot.com/2015/12/between-rock-and-hard-link.html, 2015.

[6] "Windows 10 Symbolic Link Mitigations," https://googleprojectzero.blogspot.com/2015/08/windows-10hh-symbolic-link-mitigations.html, 2015.

[7] "Windows Exploitation Tricks: Arbitrary Directory Creation to Arbitrary File Read," https://googleprojectzero.blogspot.com/2017/08/windows-exploitation-tricks-arbitrary.html, 2017.

[8] "Windows exploitation tricks: Exploiting arbitrary file writes for local elevation of privilege," https://googleprojectzero.blogspot.com/2018/04/windows-exploitation-tricks-exploiting.html, 2018.

[9] "Abusing privileged file operations," https://troopers.de/downloads/troopers19/TROOPERS19_AD_Abusing_privileged_file_operations.pdf, 2019.

[10] "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, 2019.

[11] "CVE-2019-5443," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5443/, 2019.

[12] "Windows Privilege Escalation - DLL Proxying," https://itm4n.github.io/dll-proxying/, 2019.

[13] "CVE-2020-13542," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-13542/, 2020.

[14] "CVE-2020-13884," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-13884/, 2020.

[15] "CVE-2020-26284," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-26284, 2020.

[16] "CVE-2020-27955," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-27955/, 2020.

[17] "CVE-2020-36167," https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-36167, 2020.

[18] "CVE-2020-8224," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8224/, 2020.

[19] "GitHub CLI can execute a git binary from the current directory," https://github.com/cli/cli/security/advisories/GHSA-fqfh-778m-2v32/, 2020.

[20] "CVE-2021-0057," https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-0057, 2021.

[21] "ld.so(8) — Linux manual page," https://man7.org/linux/man-pages/man8/ld.so.8.html/, 2021.

[22] "SuperDllHijack," https://github.com/anhkgg/SuperDllHijack/, 2021.

[23] "Windows security servicing criteria," https://www.microsoft.com/en-us/msrc/windows-security-servicing-criteria, 2021.

[24] "Writing Preoperation and Postoperation Callback Routines," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/writing-preoperation-and-postoperation-callback-routines/, 2021.

[25] "IRP_MJ_CREATE," https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-create, 2022.

[26] "IRP_MJ_READ," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/irp-mj-read, 2022.

[27] "Junction Point," https://learn.microsoft.com/en-us/sysinternals/downloads/junction, 2022.

[28] "Samsung Kies," https://www.samsung.com/cn/support/kies/, 2022.

[29] "Understand special identities groups," https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/manage/understand-special-identities-groups/, 2022.

[30] "A tool that shows detailed information about named pipes in Windows," https://github.com/cyberark/PipeViewer/, 2023.

[31] "Access Control List," https://learn.microsoft.com/en-us/windows/win32/secauthz/access-control-lists, 2023.

[32] "An Unconventional Exploit for the RpcEptMapper Registry Key Vulnerability," https://itm4n.github.io/windows-registry-rpceptmapper-exploit/, 2023.

[33] "Breaking Docker Named Pipes SYSTEMatically: Docker Desktop Privilege Escalation," https://www.cyberark.com/resources/threat-research-blog/breaking-docker-named-pipes-systematically-docker-desktop-privilege-escalation-part-1, 2023.

[34] "Bugcrowd," https://bugcrowd.com/, 2023.

[35] "Common Vulnerabilities and Exposures," https://cve.mitre.org/, 2023.

[36] "Crassus Windows privilege escalation discovery tool," https://github.com/vu-ls/Crassus/, 2023.

[37] "CVE-2022-25969," https://cve.mitre.org/cgi-bin/cvename.cgi?name=2022-25969, 2023.

[38] "CVE-2022-26319," https://cve.mitre.org/cgi-bin/cvename.cgi?name=2022-26319, 2023.

[39] "CVE-2022-39845," https://cve.mitre.org/cgi-bin/cvename.cgi?name=2022-39845, 2023.

[40] "CVSS," https://nvd.nist.gov/vuln-metrics/cvss, 2023.

[41] "Dynamic link library search order," https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order/, 2023.

[42] "githooks - Hooks used by Git," https://git-scm.com/docs/githooks, 2023.

[43] "HackerOne," https://hackerone.com/, 2023.

[44] "Hijack Execution Flow: DLL Search Order Hijacking," https://attack.mitre.org/techniques/T1574/001/, 2023.

[45] "Hijack Execution Flow: Path Interception by Search Order Hijacking," https://attack.mitre.org/techniques/T1574/008/, 2023.

[46] "Intigriti," https://www.intigriti.com/, 2023.

[47] "LoadLibraryEx," https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryexa, 2023.

[48] ".NET is the free, open-source, cross-platform framework for building modern apps and powerful cloud services." https://dotnet.microsoft.com/en-us/, 2023.

[49] "PrivescCheck- Privilege Escalation Enumeration Script for Windows," https://github.com/itm4n/PrivescCheck, 2023.

[50] "Process Monitor," https://learn.microsoft.com/en-us/sysinternals/downloads/procmon, 2023.

[51] "Return Oriented Programming," https://en.wikipedia.org/wiki/Return-oriented_programming/, 2023.

[52] "RubyGems," https://rubygems.org/, 2023.

[53] "Spartacus," https://github.com/Accenture/Spartacus, 2023.

[54] "Symbolic link," https://en.wikipedia.org/wiki/Symbolic_link, 2023.

[55] "The website of Jerry," https://sites.google.com/view/iamjerry, 2023.

[56] "Unix like," https://en.wikipedia.org/wiki/Unix-like, 2023.

[57] "Windows Installer," https://learn.microsoft.com/en-us/windows/win32/msi/windows-installer-portal, 2023.

[58] A. Basu, J. Sampson, Z. Qian, and T. Jaeger, "Unsafe at any copy: Name collisions from mixing case sensitivities," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 183–198.

[59] L. Bauer, S. Garriss, and M. K. Reiter, "Detecting and resolving policy misconfigurations in access-control systems," *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 1, pp. 1–28, 2011.

[60] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan, "Cookiext: Patching the browser against session hijacking attacks," *Journal of Computer Security*, vol. 23, no. 4, pp. 509–537, 2015.

[61] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," *2015 IEEE Symposium on Security and Privacy*, pp. 725–741, 2015.

[62] H. Chen, N. Li, C. S. Gates, and Z. Mao, "Towards analyzing complex operating system access control configurations," in *Proceedings of the 15th ACM symposium on Access control models and technologies*, 2010, pp. 13–22.

[63] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, and Z. Xue, "Sharing more and checking less: Leveraging

common input keywords to detect bugs in embedded systems," in *USENIX Security Symposium*, 2021.

[64] Q. A. Chen, Z. Qian, Y. Jia, Y. Shao, and Z. M. Mao, "Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks," *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[65] Chocolatey Community, "Chocolatey Package," https://community.chocolatey.org/packages, 2023.

[66] K. Drakonakis, S. Ioannidis, and J. Polakis, "The cookie hunter: Automated black-box auditing for web authentication and authorization flaws," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1953–1970.

[67] M. Ghasemisharif, A. Ramesh, S. Checkoway, C. Kanich, and J. Polakis, "O single sign-off, where art thou? an empirical analysis of single sign-on account hijacking and session management on the web." in *USENIX Security Symposium*, 2018, pp. 1475–1492.

[68] Gnupg, "Gnupg," https://www.gnupg.org/documentation/manuals/gcrypt/Configuration.html, 2023.

[69] Google, "Crashpad," https://chromium.googlesource.com/crashpad/crashpad, 2023.

[70] S. Govindavajhala and A. W. Appel, "Windows access control demystified," *Princeton university*, 2006.

[71] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 177–192.

[72] C. Huang, X. Han, and G. Yu, "LPET–mining MS-windows software privilege escalation vulnerabilities by monitoring interactive behavior," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 2089–2091.

[73] MSDN, "Minifilter Framework," https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-concepts, 2021.

[74] MSDN, "CreateFileA," https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea, 2023.

[75] msrc, "CVE-2023-33135," https://msrc.microsoft.com/update-guide/vulnerability/CVE-2023-33135, 2023.

[76] P. Naldurg, S. Schwoon, S. Rajamani, and J. Lambert, "Netra: seeing through access control," in *Proceedings of the fourth ACM workshop on Formal methods in security*, 2006, pp. 55–66.

[77] S. Parkinson, V. Somaraki, and R. Ward, "Auditing file system permissions using association rule mining," *Expert Systems with Applications*, vol. 55, pp. 274–283, 2016.

[78] J. Patrick-Evans, M. Dannehl, and J. Kinder, "Xfl: Naming functions in binaries with extreme multi-label learning," 2021.

[79] H. Peng, Z. Yao, A. A. Sani, D. J. Tian, and M. Payer, "Gleefuzz: Fuzzing webgl through error message guided mutation," *USENIX Security'23*, 2023.

[80] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[81] R. A. Shaikh, K. Adi, and L. Logrippo, "A data classification method for inconsistency and incompleteness detection in access control policy sets," *International Journal of Information Security*, vol. 16, pp. 91–113, 2017.

[82] C. Siechert and E. Bott, *Microsoft Windows XP inside out*. Microsoft Press, 2002.

[83] S. Sivakorn, A. D. Keromytis, and J. Polakis, "That's the way the cookie crumbles: Evaluating https enforcing mechanisms," in *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, 2016, pp. 71–81.

[84] S. Sivakorn, I. Polakis, and A. D. Keromytis, "The cracked cookie jar: Http cookie hijacking and the exposure of private information," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 724–742.

[85] A. Sudhodanan and A. Paverd, "Pre-hijacked accounts: An empirical study of security failures in user account creation on the web," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1795–1812.

[86] Vetle, "Finding privesc with procmon," https://bordplate.no/presentations/finding_privesc_with_procmon.pdf, 2019.

[87] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *ArXiv*, vol. abs/2109.00859, 2021.

[88] W. Wu, Y. Chen, X. Xing, and W. Zou, "Kepler: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities." in *USENIX Security Symposium*, 2019, pp. 1187–1204.

[89] yinkaisheng, "UIAutomation," https://github.com/yinkaisheng/Python-UIAutomation-for-Windows, 2023.

[90] X. Zhang, X. Wang, R. Slavin, and J. Niu, "Condysta: Context-aware dynamic supplement to static taint analysis," *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 796–812, 2021.

[91] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," *2013 35th International Conference on Software Engineering (ICSE)*, pp. 652–661, 2013.

[92] S. Zuckerbraun, "Abusing arbitrary file deletes to escalate privilege and other great tricks," https://www.thezdi.com/blog/2022/3/16/abusing-arbitrary-file-deletes-to-escalate-privilege-and-other-great-tricks, 22.

APPENDIX

In this appendix, we list detailed evaluation results on known vulnerabilities in TABLE A1, including CVE-ID, affected software, stage, operation, and results of PrivescCheck, JERRY-Crassus, and our tool JERRY.

TABLE A1: **The Detailed Evaluation Results on Known Vulnerabilities.** The abbreviations Ins, Uni, Up, Rep, SU and Us represent Installation, Uninstallation, Updating, Repairing, Starting Up and Usage, respectively. The abbreviations PC, IL, RD, CT, MV and DT represent Process Creation, Image Loading, Reading, Creating, Moving and Deleting, respectively.

No.	CVE ID	Affected Software	Stage	Operation	PrivescCheck	Jerry-Crassus	Jerry
1	CVE-2022-41604	ZoneAlarm	Ins	MV			✓
2	CVE-2022-39845	Samsung Kies	Uni	DT			✓
3	CVE-2022-39286	Jupyter Core	Us	PC		✓	✓
4	CVE-2022-38611	Watchdog Anti-Virus	SU	IL	✓	✓	✓
5	CVE-2022-36415	Scooter Beyond Compare	Uni	IL		✓	✓
6	CVE-2022-35861	pyenv	Us	RD			✓
7	CVE-2022-29320	MiniTool Partition Wizard	SU	PC	✓	✓	✓
8	CVE-2022-28964	Avast Premium Security	Us	CT			✓
9	CVE-2022-27535	Kaspersky VPN	Us	DT			✓
10	CVE-2022-27094	Sony PlayMemories Home	SU	PC	✓	✓	✓
11	CVE-2022-24826	git-lfs	Us	PC		✓	✓
12	CVE-2022-24767	Git for Windows	Uni	IL		✓	✓
13	CVE-2021-46368	TRIGONE Remote System Monitor	SU	PC	✓	✓	✓
14	CVE-2021-45975	Acer Care Center	Up	IL	✓	✓	✓
15	CVE-2021-43463	Ext2Fsd	SU	PC	✓	✓	✓
16	CVE-2021-43460	System Explorer	SU	PC	✓	✓	✓
17	CVE-2021-43455	FreeLAN	SU	PC	✓	✓	✓
18	CVE-2021-43454	AnyTEXT Seacher	SU	PC	✓	✓	✓
19	CVE-2021-42923	ShowMyPC	SU	IL		✓	✓
20	CVE-2021-38570	Foxit Readinger	Uni	DT			✓
21	CVE-2021-37363	Gestionale Open	Ins	PC	✓	✓	✓
22	CVE-2021-36753	Sharkrdp bat	Us	PC	✓	✓	✓
23	CVE-2021-3606	OpenVPN	SU	RD	✓	✓	✓
24	CVE-2021-31847	McAfee Agent	Rep	IL		✓	✓
25	CVE-2021-30490	ViewPower	Ins	PC	✓	✓	✓
26	CVE-2021-28098	Forescout CounterACT	SU	CT			✓
27	CVE-2021-22000	VMware Thinapp	SU	IL		✓	✓
28	CVE-2020-9442	OpenVPN Connect client	SU	IL		✓	✓
29	CVE-2020-8224	Nextcloud Desktop Client	SU	RD		✓	✓
30	CVE-2020-6654	Eaton's 9000x Programming and Configuration Software	Ins	IL	✓	✓	✓
31	CVE-2020-5992	NVIDIA GeForce NOW	SU	RD		✓	✓
32	CVE-2020-5977	NVIDIA GeForce Experience	SU	RD			✓
33	CVE-2020-27643	1E Client	Us	CT			✓
34	CVE-2020-26941	NOD32 Antivirus ESET	Ins	CT			✓
35	CVE-2020-26538	Foxit Readinger	SU	PC		✓	✓
36	CVE-2020-26284	Hugo	Us	PC			
37	CVE-2020-26050	SaferVPN	SU	RD		✓	✓
38	CVE-2020-25744	SaferVPN	Us	CT			✓
39	CVE-2020-25043	Kaspersky Secure Connection	Ins	DT			✓
40	CVE-2020-22809	Windscribe	SU	PC	✓	✓	✓
41	CVE-2020-15843	ActFax	SU	IL	✓	✓	✓
42	CVE-2020-15264	Boxstarter	SU	IL	✓	✓	✓
43	CVE-2020-15261	Veyon	SU	PC	✓	✓	✓
44	CVE-2020-15145	Composer	Ins	PC		✓	✓
45	CVE-2020-13885	Citrix Workspace App	Uni	IL		✓	✓
46	CVE-2020-13884	Citrix Workspace App	Uni	PC		✓	✓
47	CVE-2020-13866	WinGate	Ins	PC	✓	✓	✓
48	CVE-2020-13542	LogicalDoc	Ins	PC	✓	✓	✓
49	CVE-2020-12431	Splashtop Software	Up	RD			✓
50	CVE-2020-10143	Macrium Reflect	SU	RD		✓	✓
51	CVE-2020-10140	Acronis True Image	SU	IL		✓	✓