Linux File Overwrite Privilege Escalation Techniques

Executive Summary

Linux privilege escalation fundamentally involves an unauthorized user or process gaining elevated permissions beyond their legitimate entitlements. [1, 2] Within this domain, file overwrite stands out as a particularly potent mechanism. It enables attackers to directly modify critical system files, configuration files, or even executable binaries, thereby gaining higher access levels, most commonly root privileges. [1, 2, 3] This manipulation can lead to full system compromise, data breaches, and persistent unauthorized access.

The methods for achieving file overwrite privilege escalation are diverse, ranging from exploiting deep-seated kernel vulnerabilities like Dirty COW (CVE-2016-5195) and Dirty Pipe (CVE-2022-0847) [1, 4, 5, 6] to leveraging common misconfigurations. These misconfigurations include improperly set permissions on sensitive files (e.g., `/etc/shadow`, `/etc/passwd`) [7], vulnerable SUID/SGID binaries [2, 8], and insecurely configured writable directories like `/tmp`. [9, 10] Furthermore, timing-dependent race conditions (Time of Check to Time of Use - TOCTOU) [11, 12], symbolic link (symlink) and hard link attacks [9, 13], abuse of misconfigured services or scheduled cron jobs [2, 3, 14], and flaws in package managers [15, 16, 17] all present avenues for file overwrite. The overarching impact is severe, encompassing system integrity compromise, data exfiltration, and the establishment of persistent backdoors. [2, 3, 13]

Technique Name	Brief Mechanism	Common Target Files/Components	Example CVE (if applicable)
Kernel Exploits	Exploiting kernel bugs to gain write access to read-only memory/files.	`/etc/passwd`, `/etc/shadow`, SUID binaries, kernel memory	CVE-2016-5195 (Dirty COW), CVE-2022-0847 (Dirty Pipe)
Writable Sensitive Files	Direct modification of critical configuration files due to lax permissions.	`/etc/shadow`, `/etc/passwd`, service configuration files	CVE-2025-4598, CVE-2025-25428

Technique Name	Brief Mechanism	Common Target Files/Components	Example CVE (if applicable)
SUID/SGID Binary Exploitation	Abusing privileged executables with flaws or misconfigurations to run arbitrary code.	Any file writable by root, especially `/etc/shadow`, `/etc/passwd`	CVE-2021-3156 (Baron Samedit), CVE-2024-29119
Race Conditions (TOCTOU)	Manipulating file paths between a security check and its subsequent use.	Temporary files, sensitive system files (e.g., `/etc/shadow`)	CVE-1999-1187, CVE-1999-1091
Symbolic Link (Symlink) Attacks	Redirecting privileged file operations to unintended targets via symlinks.	Temporary files, sensitive system files (e.g., `/etc/shadow`)	CVE-2020-8019, CVE-2021-25321
Misconfigured Services	Exploiting services running as root that allow file modification or command injection.	Service configuration files, any file writable by service user	General misconfigurations
Writable Cron Jobs	Modifying scripts executed by cron jobs or the crontab file itself.	Cron scripts, `/etc/crontab`	General misconfigurations
Systemd Service File Overwrite	Injecting malicious commands into systemd unit files.	Systemd unit files (e.g., `/etc/systemd/system/*.service`)	General misconfigurations
Package Manager Vulnerabilities	Exploiting flaws in package managers to write files outside intended paths.	Arbitrary filesystem locations, system binaries, configuration files	CVE-2014-7206 (APT), CVE-2019-16776 (NPM)
Insecure Temporary Files	Predicting or manipulating insecurely	Any file used by the vulnerable application, sensitive system data	CWE-377

Technique Name	Brief Mechanism	Common Target Files/Components	Example CVE (if applicable)
	created temporary files.		
Log Poisoning	Injecting malicious data into log files to achieve code execution when logs are processed.	Log files, scripts processing logs	CRLF Injection related vulnerabilities

Introduction to Linux Privilege Escalation

Privilege escalation is defined as the act of exploiting a bug, design flaw, or configuration oversight in an operating system or software application to gain elevated access to resources that an application or user is normally restricted from. [1, 2, 3, 18] In multi-user Linux environments, privileges dictate what a user is permitted to do, such as viewing, editing, or modifying system files. [1, 2] When an unprivileged user gains entitlements they are not authorized for, it constitutes privilege escalation. [1] This is a critical concern for information security, as it enables malicious activities like deleting files, accessing private information, or installing unwanted programs. [1, 2, 3] For attackers, privilege escalation is a means to an end, allowing them to deepen their access, persist within an environment, and execute more severe malicious actions. [2, 18]

The ability to read and write any sensitive file is a primary objective for attackers performing privilege escalation, as it facilitates persistence between reboots and the insertion of permanent backdoors. [1] File overwrite is a direct and highly impactful vector for achieving root access. By modifying critical system files that govern user authentication (e.g., password hashes), system configurations, or executable binaries, an attacker can directly alter the system's security posture to their advantage. [1, 6, 7] The emphasis on "file overwrite" as a specific type of privilege escalation highlights a fundamental attack surface. While many privilege escalation techniques exist, those leading to file overwrite are particularly potent because they directly impact the integrity and configuration of the operating system itself, offering persistent and often undetectable control. For instance, vulnerabilities like Dirty COW explicitly state that the attack does not leave traces in system logs [4], making detection significantly harder. This combination of persistence and stealth elevates file overwrite as a high-priority threat, underscoring the importance of file integrity monitoring and immutable infrastructure principles as crucial defensive measures for Linux systems.

Kernel-Level Vulnerabilities Leading to File Overwrite

Kernel exploits are specialized programs that leverage vulnerabilities within the Linux kernel to execute arbitrary code with elevated permissions, typically resulting in superuser (root) access. [1, 2] The generic workflow involves tricking the kernel into running an attacker's payload in kernel mode, manipulating critical kernel data (such as process privileges), and subsequently launching a shell with the newly acquired root privileges. [1] For a kernel exploit to be successful, four conditions must generally be met: the presence of a vulnerable kernel, a matching exploit, the ability to transfer the exploit onto the target system, and the capability to execute it. [1] Systems running older kernel versions are particularly susceptible due to the prevalence of publicly known vulnerabilities. [2]

Dirty COW (CVE-2016-5195): Exploiting Copy-on-Write (COW) Race Conditions

Dirty COW (CVE-2016-5195) is a notorious local privilege escalation bug that exploits a race condition in the Linux kernel's memory-management subsystem. [1, 4, 19] Specifically, it targets how the kernel handles the copy-on-write (COW) breakage of private read-only memory mappings. [1, 4, 19] This race condition, when exploited with precise timing, allows an unprivileged local user to gain write access to memory mappings that are intended to be read-only. [1, 4, 19] Copy-on-write is a technique designed for efficient duplication of resources that might be modified; if a resource is copied but not changed, the system avoids creating a new resource, allowing the copy and the original to share the same resource. A new resource is only created if a modification occurs. The Dirty COW vulnerability manipulates this mechanism to achieve unauthorized write access. [19]

The core impact of Dirty COW is the ability to modify binaries and files that are typically read-only and only modifiable by privileged users. [4] For instance, an attacker could alter a critical system binary like `/bin/bash` to include malicious code. When an unsuspecting user then executes this infected program, the malicious code runs with the same elevated privileges as the original program. [4] A significant and concerning characteristic of Dirty COW is that the exploitation process itself does not leave any traces in the system logs [4], making detection extremely challenging. The primary and most effective countermeasure is to apply patches or upgrade to a newer, non-vulnerable kernel version. [4, 19] This vulnerability was present in the Linux kernel since version 2.6.22 (September 2007) and was patched in versions 4.8.3, 4.7.9, 4.4.26, and newer. [4]

Dirty Pipe (CVE-2022-0847): Arbitrary File Write through Pipe Buffer Manipulation

Dirty Pipe (CVE-2022-0847) is a local privilege escalation vulnerability affecting Linux kernels from version 5.8 onwards. [5, 6, 20] It enables a local attacker to bypass file permissions and write arbitrary data to any file, provided certain conditions are met. [6] The vulnerability was discovered in the context of the `splice()` system call, which is designed to efficiently move data between a file descriptor and a pipe without data crossing the usermode/kernelmode address space boundary. [6] Normally, when a file is read, its memory pages are copied into a memory-managed space called the page cache, reducing hard disk I/O. The bug manifests when a file is being read into a pipe via `splice()` and, concurrently, arbitrary data is written into the pipe. This erroneous state causes the newly written data to end up in the same page cache used by the file, effectively writing to the file even if it was originally opened in read-only mode (`O_RDONLY`). [6]

Successful exploitation requires the following conditions: the target file must be readable by the attacker; the overwritten offset within the file must not fall on a page boundary (typically 4096 bytes); the write operation cannot cross a page boundary; the file cannot be resized during the exploit; and the file must be backed by the page cache (i.e., a regular file). [6] Dirty Pipe allows local attackers to easily gain root privileges by rewriting sensitive system files, such as `/etc/passwd` (to change root's password hash), or by hijacking `setuid-root` binaries by overwriting their ELF (Executable and Linkable Format) with malicious code. [6, 20] It is conceptually similar to Dirty COW in its ability to write to read-only resources, but Dirty Pipe specifically targets read-only *files*, whereas Dirty COW targeted read-only *memory maps*. [6] This vulnerability impacts Linux kernels from 5.8 until versions before 5.16.11, 5.15.25, and 5.10.102. [5, 6] The recommended remediation is to upgrade the Linux kernel to a patched version. [6]

The existence of both Dirty COW and Dirty Pipe, discovered years apart, points to a recurring pattern of fundamental kernel vulnerabilities related to memory and file I/O handling that enable arbitrary writes to read-only files. This suggests a deep-seated and systemic challenge in kernel development and auditing, particularly concerning the complexities of race conditions in highly optimized subsystems like memory management and inter-process communication. The striking similarity in their impact (arbitrary file write) and root cause (race conditions in core kernel mechanisms like copy-on-write and pipe handling) indicates that these are not isolated, one-off bugs. Instead, they represent manifestations of a deeper, inherent complexity in kernel design and optimization, where subtle timing interactions can introduce severe security flaws. This recurring pattern underscores that even with extensive code review and testing, identifying and mitigating such race conditions in highly concurrent kernel environments remains exceptionally difficult. It highlights the critical importance of continuous kernel security research, rapid patch deployment, and potentially, a re-evaluation of certain kernel design patterns to minimize these classes of vulnerabilities.

Other Notable Kernel Exploits Enabling File Overwrites

The "Full Nelson Exploit" is another example of a kernel vulnerability that leverages a race condition, specifically in the way Linux kernels handle `pipe_buf` structures. This exploitation allows an attacker to corrupt kernel memory, overwrite critical data structures like credentials, and ultimately escalate their privileges to root. [2]

Vulnerability Name	CVE ID	Affected Kernel Versions	Mechanism Summary	Specific File Overwrite Capability	Detection/Traceability
Dirty COW	CVE-2016-5195	2.6.22 - 4.8.3 (patched)	Race condition in Copy-on-Write (COW) memory handling.	Turn read-only memory mappings into writable, modifying binaries/files.	Does not leave traces in system logs.
Dirty Pipe	CVE-2022-0847	5.8 - 5.16.11 (patched)	Race condition in `splice()`	Write arbitrary data to read-only files backed by page cache.	No specific mention of logging, but arbitrary file writes are detectable.

Vulnerability Name	CVE ID	Affected Kernel Versions	Mechanism Summary	Specific File Overwrite Capability	Detection/Traceability
			system call and pipe buffer handling.		
Full Nelson	Not specified	Not specified	Race condition in `pipe_buf` structures.	Corrupt kernel memory, overwrite credentials/data.	Not specified.

Exploiting File and Directory Permission Misconfigurations

Writable Sensitive Files

The `/etc/shadow` and `/etc/passwd` files are foundational to user authentication and access control on Linux systems. [7] Their compromise is a direct path to privilege escalation. If `/etc/shadow` is misconfigured to be writable by an unprivileged user (e.g., `user001` having write access due to misconfiguration [7]), an attacker can directly replace the root user's password hash with a new, known hash. [7] After modifying and saving the file, the attacker can then use the `su root` command with the new password, instantly gaining root access. [7] This represents a direct and highly effective privilege escalation method.

Even if only read access to `/etc/shadow` is available, an attacker can extract the hashed root password. [3, 7] These hashes can then be cracked offline using specialized tools like John the Ripper. [3, 7] The `unshadow` command can be used to combine `/etc/passwd` and `/etc/shadow` contents into a format suitable for cracking. [7] Once the root password is successfully cracked, the attacker can use `su root` to gain full root privileges. [3, 7] CVE-2025-4598 describes a systemd-coredump flaw that could allow access to `/etc/shadow` content via a SUID process coredump, enabling sensitive data exposure. [21, 22] CVE-2025-25428 highlights a hardcoded password vulnerability in `/etc/shadow` for specific devices, allowing root login. [21]

Beyond user authentication files, misconfigured permissions on other system or service configuration files can be exploited. If a service configuration file, particularly one for a service running with root privileges, is writable by a low-privileged user, an attacker can modify it to inject arbitrary code or alter service behavior for privilege escalation. [2, 23] For example, a web server running as root with a misconfigured PHP script could be exploited to execute malicious scripts. [2]

The recurrent appearance of `/etc/shadow` and `/etc/passwd` as targets, whether through direct write permissions, read-and-crack, or indirect kernel/SUID exploits, underscores their status as the "crown jewels" for Linux privilege escalation. Any attack path, no matter how convoluted, that eventually leads to modifying or accessing these files is a high-priority threat.

Gaining control over them effectively grants full administrative control over user accounts and, by extension, the entire system. This reinforces the absolute necessity for extremely stringent access controls, robust file integrity monitoring, and immediate patching for *any* vulnerability that could potentially allow modification or unauthorized access to these files. They are the ultimate prize for a privilege escalation attacker.

SUID/SGID Binary Exploitation

SUID (Set User ID) and SGID (Set Group ID) bits are special permissions in Unix-like systems that allow an executable file to run with the privileges of the file's owner or group, respectively, rather than the privileges of the user who executes the file. [2, 8, 24] This mechanism is essential for legitimate programs that require elevated permissions to perform specific tasks, such as the `passwd` utility which needs root privileges to modify `/etc/shadow`. [23] Attackers actively seek out SUID/SGID binaries that are either misconfigured or contain exploitable vulnerabilities. [2, 8] If a SUID binary has a flaw (e.g., a buffer overflow, a command injection vulnerability, or insecure handling of file operations), an attacker can exploit it to execute arbitrary code with the elevated privileges of the file owner (often root). [2, 8, 23, 24] This can directly lead to gaining a root shell or overwriting privileged system files. [2]

A common enumeration technique for attackers is to use the `find` command to locate SUID binaries across the filesystem. For example, `find / -perm -4000 -type f -ls 2>/dev/null` searches for files with the SUID bit set. [1, 2]

Common vulnerable SUID binaries include:

- **`find`:** If the `find` binary has the SUID bit set, an attacker can leverage it to spawn a root shell, for instance, by executing `sudo find. -exec /bin/bash \;` . [2, 3]
- **Editors/Interpreters:** Programs like `awk`, `more`, `less`, `vi`, `nmap`, `perl`, `ruby`, `python`, and `gdb` should *never* have the SUID bit set. [1, 2] If they do, an attacker can easily read or overwrite any files on the system by using their shell escape features or arbitrary code execution capabilities. [1]
- **`sudoedit` (CVE-2021-3156 - Baron Samedit):** A critical heap overflow vulnerability in the `sudo` program (specifically `sudoedit`) allowed unprivileged users to gain root privileges without authentication, including the ability to be listed in the `sudoers` file. [5, 25] This vulnerability affected `sudo` versions from 1.8.2 to 1.8.31p2 and 1.9.0 to 1.9.5p1. [25]
- **`systemd-coredump` (CVE-2025-4598):** A flaw allowing an attacker to force a SUID process to crash and replace it with a non-SUID binary to access the original's privileged coredump, potentially reading sensitive data like `/etc/shadow`. [21, 22] This is a race condition scenario.
- **`Netdata`'s `ndsudo` (CVE-2024-29119):** This SUID `root`-owned executable, shipped with Netdata, runs a restricted set of external commands. However, its search paths are supplied by the `PATH` environment variable, allowing an attacker to control where `ndsudo` looks for these commands. If the attacker has write access to a directory in the `PATH`, they can place a malicious executable there, leading to local privilege escalation. [22]

The persistent vulnerability of SUID/SGID binaries, even common and critical utilities like `find` or `sudo`, reveals a fundamental tension between system functionality and security. The necessity for certain programs to run with elevated privileges creates an inherent and often exploited attack surface. This indicates that secure development practices for privileged binaries, including meticulous input validation and avoidance of dangerous functions (e.g., `system()`), are paramount, and that simply relying on default configurations is insufficient. The continuous discovery of critical vulnerabilities in widely used SUID programs suggests that even mature,

well-audited system components are not immune to flaws. This indicates a need for not only strict file permission management and regular SUID/SGID audits but also a deeper emphasis on "secure by design" principles in the development of *any* privileged binary. This includes rigorous threat modeling, static/dynamic code analysis, and a commitment to the principle of least privilege in the functionality of the binary itself, not just its execution context.

Countermeasures include never setting the SUID bit on programs that allow shell escapes, file editors, compilers, or interpreters. [1] Furthermore, directories such as `/etc/sudoers.d` should maintain strict permissions, being writable only by root. [1]

SUID/SGID Binary	Common Vulnerability Type	Exploitation Example	Impact on File Overwrite	Mitigation Note
`find`	Misconfiguration (SUID bit set)	`sudo find. -exec /bin/bash \;` to get root shell.	Allows arbitrary command execution as root, enabling file overwrite.	Do not set SUID bit on `find`.
`vi`, `more`, `less`, `awk`, `perl`, `python`, `ruby`, `nmap`, `gdb`	Misconfiguration (SUID bit set)	Using shell escape features or arbitrary code execution within the program.	Allows reading/overwriting any file on the system.	Never set SUID bit on editors/interpreters.
`sudoedit`	Heap overflow (CVE-2021-3156)	Crafted command line input to trigger overflow and gain root.	Allows gaining root privileges, potentially modifying `sudoers` file.	Patch `sudo` to fixed versions (e.g., 1.9.5p2+).
`systemd-coredump`	Race condition (CVE-2025-4598)	Force SUID process crash, replace with non-SUID binary to access coredump.	Read sensitive data like `/etc/shadow` from coredump.	Patch `systemd-coredump` and address race conditions.
`ndsudo` (Netdata)	`PATH` manipulation (CVE-2024-29119)	Place malicious executable in attacker-writable	Execute arbitrary programs with root permissions.	Restrict `PATH` for SUID binaries; ensure `ndsudo` is patched.

SUID/SGID Binary	Common Vulnerability Type	Exploitation Example	Impact on File Overwrite	Mitigation Note
		directory in `PATH`.		

Insecurely Configured Writable Directories (e.g., `/tmp`)

World-writable directories, notably `/tmp` and `/var/tmp`, are frequently targeted in privilege escalation attacks. [9, 10, 26] Their universal write permissions make them ideal locations for attackers to create files. If privileged programs create temporary files with predictable names in such directories without adequately checking for their prior existence, an attacker can exploit this. By pre-creating a file or a symbolic link with the expected name, the attacker can redirect the privileged program's operations to an arbitrary target file, leading to unauthorized overwrites. [9, 27]

Race Conditions (Time of Check to Time of Use - TOCTOU)

A Time of Check to Time of Use (TOCTOU) attack is a cybersecurity vulnerability that arises when a system checks the state of a resource or condition at one point in time, but then uses or relies on that resource or condition at a later time. [2, 10, 11, 12, 26, 27] The critical vulnerability lies in the brief time gap between the "check" and the "use." During this interval, an attacker can manipulate the resource or condition, leading to unauthorized access, data corruption, or other security breaches. [2, 11] Race conditions, which are a broader category of concurrency issues, are a common form of TOCTOU vulnerabilities in UNIX-like systems, particularly in shared directories such as `/tmp` and `/var/tmp`. [10] These can occur due to interference from untrusted processes or even unintended interactions between trusted processes sharing resources. [10, 27]

Exploiting Temporary File Creation and Symbolic Links in Race Conditions

A prevalent TOCTOU exploitation vector involves privileged programs that create temporary files. These programs often create files in world-writable directories (like `/tmp`) and may use predictable naming conventions. [9, 26, 27] If such a program first checks for the file's existence or permissions (e.g., using `access()`) and then proceeds to open the file (e.g., using `fopen()`) in a separate, non-atomic step, a TOCTOU window is created. [2, 11, 12, 27] An attacker can exploit this by rapidly creating a symbolic link (symlink) with the predictable temporary file name, pointing it to a sensitive target file (e.g., `/etc/shadow`). [2, 9, 11, 12, 27] During the brief interval between the privileged program's check and its subsequent file operation, the attacker's symlink can redirect the program's write operation to the sensitive target file. [2, 9, 11, 12, 27] This effectively allows the attacker to overwrite files they would normally not have permission to modify. [12, 27]

For example, a vulnerable program might check if `/tmp/XYZ` is writable, then proceed to open it. An attacker, in a rapid, concurrent operation, can replace `/tmp/XYZ` with a symlink to `/etc/shadow` in the brief time window, causing the privileged program to inadvertently write its data to `/etc/shadow`. [11, 12, 27]

The long history and high volume of symlink-related CVEs (over 1590 entries from 1999 to 2024 [9]) underscore that this is not a new or niche vulnerability, but a persistent and fundamental challenge in secure programming on Unix-like systems. This indicates a systemic issue where developers consistently fail to account for the indirect nature of file paths when handling privileged operations. The sheer volume and longevity of symlink vulnerabilities suggest that the problem extends beyond individual coding errors. It points to a deeper, systemic oversight or misunderstanding in how file system interactions are designed and secured within applications. Simply advising developers to "check for symlinks" is clearly insufficient, as the TOCTOU window often bypasses such checks. This implies a need for fundamental shifts towards "secure by default" programming paradigms, where atomic file operations (`O_EXCL`, `mkstemp()`, `O_TMPFILE`) are the standard, and predictable temporary file names are actively discouraged. This highlights the importance of robust API design and kernel features that inherently mitigate these risks, rather than relying solely on developer vigilance.

The most robust mitigation against TOCTOU vulnerabilities in file operations is the use of atomic operations. This typically involves using the `O_EXCL` flag in conjunction with `O_CREAT` when calling `open()`. [9, 11] This flag atomically checks if the file exists before creating it and returns an error if it does, or if the path is a symlink (even if the target doesn't exist). [9] For secure temporary file creation, the `mkstemp()` library function is highly recommended as it incorporates this atomic behavior and generates random, hard-to-guess names. [9] Linux also provides specific kernel-level countermeasures like `protected_symlinks` (which restricts symlink following in sticky-bit world-writable directories) and the `O_TMPFILE` flag for `open()` (which creates unnamed, unlinked temporary files, making symlink attacks impossible). [9]

Symbolic Link (Symlink) and Hard Link Attacks

Mechanisms of Symlink and Hard Link Manipulation for File Overwrite

Symbolic links, or symlinks, are special files that act as pointers to other files or directories within the filesystem. [9, 13] In a symlink attack, an adversary manipulates these links to trick an application or user into accessing or modifying unintended files. [13] The attacker creates a symlink that points to a sensitive or restricted file. When a vulnerable application, often running with elevated privileges, subsequently follows this symlink, it inadvertently performs its intended operation (e.g., writing data) on the target file instead of the intended one, but with its own (higher) permissions. [13] This allows attackers to bypass access controls and gain unauthorized access to sensitive data, leading to data breaches, system integrity compromise, and privilege escalation. [13]

Hard links are distinct from symlinks in that they are simply additional directory entries that point directly to an existing file's inode. [9] Unlike symlinks, they are not special files themselves and share the exact same data and metadata (including permissions) as their original target file. [9] While hard links cannot directly change the permissions or ownership of a file (as permissions

are tied to the inode, not the link itself), they can be used in conjunction with other vulnerabilities. For example, if a privileged process performs operations like `chown()` or `chmod()` on a hard-linked file, it will affect the original file's inode. The key principle enabling both symlink and hard link attacks is that a user only needs write permissions in the directory where they create the link, not special permissions on the target file itself. [9]

Exploiting `open()` with `O_CREAT` and Predictable File Names

A common vulnerability pattern arises when privileged programs attempt to create files using the `O_CREAT` flag with the `open()` system call. [9] If these programs create temporary files with predictable names in world-writable directories (like `/tmp`) without first verifying that a file with that name doesn't already exist, they become susceptible to symlink attacks. [9] The `open()` system call, when encountering a symlink with `O_CREAT`, will follow the symlink *before* checking for file existence and creating the file. This means it will attempt to create the missing *target* of the symlink. [9]

This behavior allows an attacker to pre-create a symlink pointing to an arbitrary, sensitive file (e.g., `/etc/shadow`). When the privileged program then attempts to create its temporary file, it will instead operate on the attacker's chosen target, potentially overwriting or modifying its contents with the privileged program's permissions. [9] This can lead to denial of service by corrupting important system files or, more critically, privilege escalation by gaining write access to sensitive files. [9]

The document highlights several historical and recent CVEs demonstrating this vulnerability:

- **CVE-1999-1187 (PINE mail reader):** Exploited by creating a symlink to a non-existent `.rhosts` file. When PINE opened, it created a world-writable `.rhosts` file, granting the attacker access. [9]
- **CVE-1999-1091 (tin newsreader):** Created a world-writable log file with a fixed name (`/tmp/.tin_log`) without existence checks, allowing similar exploitation. [9]
- **CVE-2020-8019 (syslog-ng package):** An install script vulnerability allowed symlinking to `/etc/shadow` and changing its ownership to an unprivileged user, enabling password modification. [9]
- **CVE-2021-25321 (arpwatch daemon):** Involved a `chown()` operation on a symlink, changing `/etc/shadow` ownership to an unprivileged `runtime` user. [9]
- **CVE-2022-35631 (Velociraptor):** A forensics tool created a temporary file based on a configuration name, trusting the monitored system too much. [9]
- **CVE-2025-25428:** A generic reference to symlink attacks manipulating files like `/etc/shadow` for privilege escalation. [21]

To effectively counter these attacks, programs should *always* use atomic file creation methods. The `O_EXCL` flag, used in conjunction with `O_CREAT` in `open()`, is crucial as it atomically checks if the file exists before creating it and returns an error if it does, or if the path is a symlink (even if the target doesn't exist). [9] For temporary files, the `mkstemp()` library function is highly recommended, as it employs this secure technique and generates random, hard-to-guess names. [9] Linux also implements specific kernel-level countermeasures: `protected_symlinks` (which, when enabled, restricts following symlinks in sticky-bit world-writable directories unless the symlink is owned by the process or has the same owner as the directory) and the `O_TMPFILE` flag for `open()` (which creates a file with no links in the filesystem, guaranteeing a new inode and making symlink attacks impossible). [9]

Abusing Misconfigured Services and Scheduled Tasks

Exploiting Services Running with Root Privileges that Allow File Modification

A significant privilege escalation vector arises from Linux services that are misconfigured to run with `root` privileges while simultaneously allowing arbitrary code execution or file modification through their interfaces. [1, 2, 23] For example, if a web server is configured to run as `root` and has a command injection vulnerability in a PHP script, an attacker can leverage this flaw to execute malicious commands directly as `root`. [1, 2] Similarly, certain database services, if running with excessive privileges, can be exploited. For instance, MySQL User Defined Function (UDF) exploits can allow arbitrary commands to be executed from the MySQL shell, which will run as `root` if the MySQL service itself is running with root privileges. [1] This violates the principle of least privilege, creating a high-impact target for attackers.

The danger of services running as `root` is a foundational principle of least privilege. Its repeated appearance as an attack vector suggests that despite this being a well-known security tenet, it remains a common misconfiguration in practice. This indicates a gap between security best practices and their implementation, likely due to convenience or legacy system requirements. This highlights a critical gap between theoretical security knowledge and practical implementation. It suggests a strong need for automated configuration auditing tools, stricter default service configurations, and a cultural shift within organizations to deeply understand and mitigate the security implications of running any service with more privileges than absolutely necessary.

Manipulating Writable Cron Jobs or Scripts Executed by Cron

Cron jobs are automated tasks scheduled to run at specific intervals on a Linux system, often with elevated privileges. [2, 23] A significant vulnerability arises if a low-privileged user has write permissions to a script executed by a cron job, or to the `crontab` configuration file itself. By modifying a writable cron script or by directly editing the `crontab` file (e.g., using `crontab -e` if permissions allow), an attacker can inject malicious commands. [2, 3, 23] These commands will then be executed by the user context of the cron job, which is frequently `root`. For instance, an attacker could insert a command to copy `/bin/bash` to `/tmp/bash` and set the SUID bit on it (`chmod +s /tmp/bash`), effectively creating a root shell. [3] Once the malicious command is injected, the attacker simply waits for the scheduled cron job to execute it, thereby escalating privileges. [2, 3, 23]

Overwriting Systemd Service Configuration Files

`systemd` is the widely used init system in modern Linux distributions, managing system services and their execution parameters. [14, 28] If a low-privileged user gains the ability to modify a `systemd` service configuration file (e.g., `/etc/systemd/system/example.service`) that is configured to run with `root` privileges, they can inject arbitrary commands into the service's execution directives. An attacker can insert a malicious payload, such as a reverse shell command, into the `ExecStart` directive within the section of a writable `systemd` unit file. [14]

After modifying the file, the attacker would then need to trigger a daemon reload (`systemctl daemon-reload`) and restart the affected service (`systemctl restart example.service`). Once restarted, the service will execute the injected malicious command with its configured privileges, typically `root`, granting the attacker a root shell. [14] The `sudo systemctl` command itself can be vulnerable to privilege escalation through the modification of service configuration files, as noted in general advisories. [14]

The exploitation of cron jobs and systemd service files for privilege escalation highlights that the *automation* and *configuration management* layers of Linux systems are significant attack surfaces. Attackers target the mechanisms designed for system maintenance and reliability, turning them into tools for persistence and privilege escalation. This reveals that the very tools and processes designed to automate and manage system operations (cron, systemd) become critical attack vectors when their underlying configuration is not adequately secured. The implicit trust placed in these automated tasks, especially those configured to run as root, makes them high-value targets for attackers seeking persistent and privileged access. This implies that security audits must extend beyond executable binaries to include a thorough review of configuration files for all automated services and tasks, ensuring strict file permissions, integrity checks, and adherence to the principle of least privilege in their operational context.

Vulnerabilities in Package Managers and Update Mechanisms

Exploiting Flaws in `APT`, `DNF`/`YUM`, `NPM` for Arbitrary File Writes during Package Operations

Package managers like `APT` (Debian/Ubuntu), `DNF` / `YUM` (Red Hat/CentOS/Fedora), and `NPM` (Node.js) are fundamental components for software installation, updates, and dependency management on Linux systems. They typically operate with elevated privileges to perform their functions. [5, 15, 16, 17, 29, 30, 31] Vulnerabilities within these managers can be severely exploited to achieve arbitrary file writes or other forms of privilege escalation during package operations. [15, 16, 17, 29, 32, 33]

The Advanced Package Tool (`APT`) has had notable vulnerabilities. For instance, CVE-2014-7206 involved `APT` incorrectly creating a temporary file when handling the changelog command. A local attacker could exploit this flaw to overwrite arbitrary files on the system. [17] While `DNF` is designed as a robust package manager [30, 31], general vulnerabilities in package managers can arise from how they process and extract package contents, particularly during installation or upgrade routines. Such flaws could allow malicious packages to write files to unintended locations.

Major JavaScript package managers, including `npm`, `yarn`, and `pnpm`, have been affected by "filesystem takeover vulnerabilities". [16] These vulnerabilities allow malicious actors to perform arbitrary file overwrites, even when packages are installed with flags intended to prevent script execution (e.g., `--ignore-scripts`). [16] This exploitation often occurs because `npm` manages executables by maintaining symlinks between globally installed packages and a specific directory. A malicious package can be crafted to overwrite these symlinks or inject malicious files into critical system paths. [16] Arbitrary file overwrite during package installation can lead to

severe consequences, including injecting malware, altering lockfiles to compromise future installations, or generally "poisoning" the filesystem with attacker-controlled content. [16] Regular updates and timely patching of package managers and the underlying operating system are crucial to mitigate these risks. [5, 18, 34]

Path Traversal Vulnerabilities in Update Processes

Path traversal, also known as directory traversal, is a vulnerability where user-controlled input is used to construct file paths without proper sanitization. [32] This allows an attacker to inject sequences like `../` to "traverse" directories and escape the intended base path. [32] In update mechanisms or file synchronization tools (like `rsync`), if path traversal vulnerabilities exist, a malicious server or client could manipulate file paths to write files outside of the intended destination directory. [33] This can be particularly dangerous when options like `--inc-recursive` or `--safe-links` are used without sufficient symlink verification, allowing an attacker to place arbitrary malicious files in critical system locations. [33] Arbitrary file write outside the designated directory can lead to placing malicious files in arbitrary system paths, potentially named after legitimate directories, which can then be leveraged for privilege escalation. [33] Recent `rsync` vulnerabilities (CVE-2024-12087, CVE-2024-12088) illustrate this. These flaws involved path traversal due to improper symlink verification and race conditions during symlink handling, ultimately allowing arbitrary file writes to unintended locations. [33]

The vulnerabilities in package managers and update mechanisms are particularly insidious because they exploit the very processes designed to *secure* and *maintain* the system. This indicates that the supply chain of software, from package creation to distribution and installation, is a critical and often overlooked attack surface for privilege escalation. Attackers are turning the system's own maintenance and security update mechanisms against it. The fact that flaws in these critical components can lead to arbitrary file writes during seemingly benign operations (like `npm install` or `apt update`) highlights a significant weakness in the software supply chain. This implies a pressing need for rigorous security audits of package manager codebases, secure development practices for package creators (e.g., preventing malicious `bin` key manipulation in `package.json`), and robust input validation within all update utilities. It also suggests that even "official" software updates could become vectors for compromise if the package manager itself is flawed or compromised, underscoring the need for layered trust and verification.

Other File Overwrite Privilege Escalation Vectors

Insecure Temporary File Creation by Applications

Beyond the specific TOCTOU race conditions, applications frequently create temporary files for various operations. If these temporary files are created insecurely—for example, by using predictable names, assigning overly permissive file permissions, or storing them in vulnerable, world-writable locations without proper checks—they become susceptible to exploitation. [35, 36] An attacker can exploit insecure temporary file creation by predicting the file name and pre-creating a malicious file or a symbolic link in its place. When the privileged application attempts to create or write to its temporary file, it inadvertently operates on the attacker-controlled file or the target of the symlink. [27, 35] This can lead to a range of impacts, including data corruption, denial of service, or, critically, privilege escalation if the temporary file is later used to update

sensitive system data (e.g., a database table controlling administrative access). [27] To prevent these vulnerabilities, temporary files must not be predictable. [36] Developers should use secure functions that generate random, unique file names and handle file creation atomically, such as `Path.GetRandomFileName()` in C# or `mkstemp()` in C. [9, 36]

Log Poisoning and Arbitrary Log File Writes Leading to Code Execution

Log poisoning is an attack technique where an attacker injects malicious data into log files. This malicious data is crafted in such a way that when a privileged application later reads or processes the log, it interprets the injected data as executable code or commands, leading to code execution or privilege escalation. [37, 38, 39, 40] If an application logs user-supplied input without proper sanitization, an attacker can inject control characters (like Carriage Return and Line Feed - CRLF) or actual malicious code into the log entry. [37] For example, in web applications, CRLF injection can cause log analysis systems to misinterpret log entries. [37] If a privileged log analysis tool, a web server displaying logs, or a script that includes log content then processes this poisoned log, it might inadvertently execute the injected code. [32, 37] While often initiated in web contexts, successful remote code execution on the server can then be leveraged as a stepping stone for local privilege escalation. [32, 38] If the log file itself is overwritten with malicious content and subsequently executed by a privileged process (e.g., a cron job that runs a script including log data), it could directly lead to privilege escalation. [32] Log poisoning can lead to various attacks, including cross-site scripting (XSS), page hijacking, or, in the context of privilege escalation, remote code execution on the underlying server. [37, 38]

The common thread among these vectors, like insecure temporary files and log poisoning, is the failure of applications to properly validate and sanitize *all* forms of user input or dynamically generated content. This indicates that privilege escalation isn't solely a system-level issue but also a pervasive application-level security flaw, where seemingly innocuous file operations or logging functions can be weaponized. This category highlights that application-level vulnerabilities, even those not directly related to system-level privileges or kernel flaws, can serve as crucial stepping stones to file overwrite privilege escalation. The root cause is often a lack of "secure by default" programming practices, insufficient input validation, or the misuse of file handling APIs. This implies that a truly holistic security strategy must extend beyond traditional system hardening to include rigorous application security testing (e.g., Static Application Security Testing - SAST, Dynamic Application Security Testing - DAST, and penetration testing) to identify and remediate these "softer" targets that can eventually be weaponized for full system compromise.

Countermeasures include strict input validation and sanitization, which are essential to prevent malicious data from entering logs. This includes removing or encoding newline characters and other control characters before logging user-supplied content. [37] Furthermore, log analysis tools must be designed to parse log files securely, without executing or misinterpreting injected data. [37]

Mitigation and Prevention Strategies

Mitigating Linux file overwrite privilege escalation requires a multi-layered approach, addressing vulnerabilities at the kernel, system configuration, application, and process levels.

Implementing the Principle of Least Privilege

A cornerstone of robust security, this principle dictates that users, applications, and processes should be granted only the absolute minimum permissions necessary to perform their intended functions. [18, 23, 34, 40, 41] This significantly reduces the attack surface and limits the potential impact should an account or process be compromised. [18, 34] For instance, avoiding the use of the `ALL` keyword in `sudoers` configurations is a direct application of this principle. [42] This also extends to creating separate user accounts for specific services or applications, granting them only the necessary permissions, and removing all unused accounts or unnecessary existing privileges. [38] Access to directories and files should be restricted as much as possible, ideally leaving it only for the file/directory owner. [38]

Regular System Patching and Kernel Updates

Maintaining up-to-date software is paramount. This includes the Linux kernel, the operating system distribution, and all installed applications. [1, 4, 5, 6, 18, 19, 34, 41] A vast number of privilege escalation attacks exploit known vulnerabilities, making timely and consistent patching a critical defense mechanism. [18, 34] Modern package managers often provide automated update features that should be leveraged to ensure systems are protected against the latest threats. [5, 30] For critical kernel vulnerabilities like Dirty COW, live-patching solutions may be available to minimize downtime. [19]

Secure Coding Practices

Developers must adopt secure coding practices to prevent file overwrite vulnerabilities. This includes:

- **Atomic File Operations:** When creating or modifying files, especially temporary ones, developers should use atomic operations to prevent Time of Check to Time of Use (TOCTOU) race conditions. This means using flags like `O_EXCL` in conjunction with `O_CREAT` for `open()` calls, or utilizing functions like `mkstemp()` that inherently handle atomic and unpredictable temporary file creation. [9] The `O_TMPFILE` flag for `open()` can also create files with no links in the filesystem, making symlink attacks impossible. [9]
- **Input Validation and Sanitization:** All user-supplied input and dynamically generated content must be rigorously validated and sanitized to prevent injection attacks, including CRLF injection for log poisoning. [37] This prevents malicious data from being written to logs or other files that could later be interpreted as executable code.
- **Avoiding Dangerous Functions:** Developers should avoid using functions that are prone to command injection or other vulnerabilities when handling user input, especially in privileged contexts. [43]
- **Principle of Least Privilege in Application Design:** Applications should be designed to run with the minimum necessary privileges. If a component does not require root privileges, it should not be configured to run as such. [1, 23]

Strict File and Directory Permissions

System administrators must enforce strict permissions on sensitive files and directories:

- `/etc/shadow` and `/etc/passwd`: These files should have highly restrictive permissions, typically readable only by root and writable only by root. [7] Any deviation is a critical misconfiguration. [7]
- **SUID/SGID Binaries:** The SUID bit should never be set on programs that allow shell escapes (e.g., `find`, `nmap`), file editors (`vi`, `more`, `less`), compilers, or interpreters (`perl`, `python`, `ruby`). [1] Regular audits should be conducted to identify and correct any improperly configured SUID/SGID binaries. [8, 41]
- **World-Writable Directories:** Directories like `/tmp` and `/var/tmp` should be managed carefully. While they are world-writable by design, kernel-level protections like `protected_symlinks` should be enabled to mitigate symlink attacks within them. [9]
- **Configuration Files:** Configuration files for services and scheduled tasks (e.g., `crontab`, `systemd` unit files) should have permissions that prevent modification by unprivileged users. [1] The `/etc/sudoers.d` directory, in particular, should only be writable by root. [1]

Continuous Monitoring and Auditing

Proactive security measures are essential for detecting and responding to privilege escalation attempts:

- **Security Audits:** Regularly audit file permissions, review system logs for suspicious activity, and perform penetration testing to identify vulnerabilities and potential paths for privilege escalation. [2]
- **File Integrity Monitoring (FIM):** Implement FIM solutions to detect unauthorized changes to critical system files, configuration files, and SUID/SGID binaries.
- **Log Monitoring:** Monitor authentication logs for unusual login patterns, failed login attempts, or logins from unfamiliar locations. [34] System logs record detailed information about user activities and network connections, making proper log management crucial for detecting abnormal activities. [23]
- **Endpoint Detection and Response (EDR):** Leverage EDR solutions to detect malicious activity, suspicious processes, and unexpected privilege changes. [18, 34]

System Hardening Techniques

Beyond specific vulnerability patching, general system hardening reduces the overall attack surface:

- **Strong Passwords and MFA:** Enforce the use of strong, unique passwords and multi-factor authentication (MFA) to prevent credential theft. [2, 18, 38]
- **Restrict Root Logins:** Disable direct root logins via SSH. [2]
- **Application Control:** Limit which directories or files can execute code, especially for critical processes or user directories, to prevent unauthorized scripts or binaries from running. [18]
- **Security-Enhanced Linux (SELinux) or AppArmor:** Deploy mandatory access control (MAC) frameworks like SELinux or AppArmor to enforce granular confinement policies and limit what processes can do, even if they gain elevated privileges. [41]

Conclusions

Linux file overwrite privilege escalation remains a critical and persistent threat, stemming from a complex interplay of kernel vulnerabilities, system misconfigurations, and insecure application development practices. The analysis reveals that attackers frequently target core system files like `/etc/shadow` and `/etc/passwd`, as their compromise offers direct and comprehensive control. The recurring nature of kernel bugs like Dirty COW and Dirty Pipe, which exploit fundamental memory and I/O handling, highlights the inherent difficulty in securing highly optimized kernel subsystems against subtle race conditions. Similarly, the long history of symlink and TOCTOU vulnerabilities underscores a systemic challenge in secure programming, where developers often overlook the non-atomic nature of file system operations.

Furthermore, the report identifies that seemingly benign components like package managers and automated services (cron, systemd) become potent attack vectors when misconfigured or flawed. This illustrates that the supply chain and the system's own maintenance mechanisms are significant, often overlooked, attack surfaces. Finally, application-level flaws, such as insecure temporary file creation and log poisoning, serve as crucial stepping stones, demonstrating that privilege escalation is not solely a system-level concern but also a pervasive application security issue.

The persistent nature of these vulnerabilities, despite decades of security research and best practices, points to a fundamental gap between theoretical knowledge and practical implementation. This gap often arises from prioritizing convenience, legacy compatibility, or a lack of deep understanding of complex system interactions.

Recommendations

To effectively counter Linux file overwrite privilege escalation, organizations should adopt a holistic and proactive security posture:

1. **Prioritize Patching and Updates:** Implement a rigorous patch management program for the Linux kernel, operating system, and all installed software. Leverage automated update mechanisms and apply patches promptly, especially for critical kernel vulnerabilities and widely used SUID binaries.
2. **Enforce Least Privilege:** Strictly adhere to the principle of least privilege across all users, services, and applications. Grant only the minimum necessary permissions for tasks, avoid running services as root unless absolutely essential, and meticulously configure `sudoers` files to limit command execution.
3. **Implement Secure Coding Practices:** Educate developers on secure file handling, emphasizing atomic operations (e.g., `O_EXCL`, `mkstemp()`, `O_TMPFILE`) and robust input validation/sanitization to prevent TOCTOU, symlink, insecure temporary file, and log poisoning vulnerabilities.
4. **Strengthen System Configurations:** Conduct regular security audits of file and directory permissions, paying particular attention to sensitive files (`/etc/shadow`, `/etc/passwd`), SUID/SGID binaries, world-writable directories (`/tmp`), cron jobs, and systemd service files. Implement mandatory access control frameworks like SELinux or AppArmor.
5. **Enhance Monitoring and Detection:** Deploy comprehensive file integrity monitoring, robust log analysis, and Endpoint Detection and Response (EDR) solutions. These tools are crucial for detecting unauthorized file modifications, suspicious process activity, and potential privilege escalation attempts that might otherwise go unnoticed.
6. **Secure the Software Supply Chain:** Recognize package managers and update mechanisms as critical attack surfaces. Implement measures to verify the integrity of packages, audit package

manager configurations, and ensure secure development practices for all software components, including third-party dependencies.