Advanced Analysis of Dynamic Linker Hijacking: The Exploitation of LD_PRELOAD in Linux Environments

I. Introduction to Dynamic Linker Hijacking (MITRE T1574.006)

The exploitation of dynamic linking mechanisms represents a persistent and powerful attack vector in Linux environments. This technique, formally tracked by MITRE ATT&CK under T1574.006 (Dynamic Linker Hijacking), targets the core preparation phase of program execution.[1] Central to this vector is the environment variable LD_PRELOAD.

1.1. Overview of the Linux Dynamic Linking Process and ELF Binaries

Linux applications that are dynamically linked rely on the system's dynamic linker/loader, typically executed as ld.so or ld-linux.so, to resolve external dependencies during runtime.[3] When an application launches, the dynamic linker is responsible for locating necessary shared libraries (e.g., libc, libssl) and mapping their functions into the process's virtual memory space.[5] This process converts unresolved function symbols defined in the executable's ELF structure into concrete addresses, allowing the program to execute. This architectural design promotes code reuse and simplified system maintenance.

1.2. The Intentional Design and Unintended Abuse of LD_PRELOAD

The LD_PRELOAD environment variable was intentionally designed as a utility for debugging, testing, and performance optimization.[4] It specifies one or more user-defined shared objects (.so files) that the dynamic loader must load *before* all other shared libraries, including fundamental ones like the C standard library (libc).[3]

The primary attack primitive derived from this feature is **symbol interposition** or **function hooking**.[6] By defining a function with the same name as a function in a standard library (e.g., printf or system), an attacker's malicious shared object intercepts calls to that function. When the dynamic linker resolves the symbol at startup, it uses the definition provided by the preloaded library because it is first in the search order. This allows the attacker to inject arbitrary code into the execution flow of the target application.[4]

1.3. Comparison of Dynamic Linking Environment Variables

Understanding the hierarchy of dynamic linking controls is crucial for analysis. While LD_PRELOAD and LD_LIBRARY_PATH both influence the linking process, they serve fundamentally different purposes and have varying security implications.

The efficacy of LD_PRELOAD for targeted function hijacking surpasses other methods because it guarantees symbol precedence over all standard libraries. Conversely, LD_LIBRARY_PATH merely alters the directories searched for entire libraries, which is less precise.[1] If an attacker uses LD_LIBRARY_PATH, they may need to supply an entire, complex, possibly fragile library replacement. By contrast, using LD_PRELOAD requires only a small .so file containing the specific malicious function, resulting in a stealthier, simpler payload less likely to cause application instability.

Comparison of Dynamic Linking Environment Variables and Mechanisms

Variable/File	**Mechanism**	**Scope**	**Symbol Precedence**	**Secure Mode Behavior**	**Source(s)**
LD_PRELOAD (Environment Variable)	Loads specified shared objects first.	Per-process or per-user session.	Highest (overrides standard library functions).	Ignored if path contains /, or if not found in trusted path and setuid/setgid.[6]	[1]
/etc/ld.so.preload (System File)	System-wide list of preloaded shared objects.	Global (all dynamically linked programs).	High (loaded after LD_PRELOAD but before other libs).[6]	Loaded only if files are system libraries or have setuid bit enabled.[3]	[2]
LD_LIBRARY_PATH (Environme	Alters library search path	Per-process or per-user session.	Low (affects search order, not	Ignored completely.[6]	[1]

nt Variable)	(directories) .		symbol precedence) .		

II. Technical Foundations of LD_PRELOAD Abuse: Symbol Interposition

Effective exploitation requires a deep understanding of the dynamic linker's internal operation and specific ELF binary features.

2.1. Dynamic Loader Operations and Precedence

When a program is launched, the dynamic linker, ld.so, executes its search and resolution process. The linker loads libraries based on a specified sequence, which generally includes RPATH/RUNPATH definitions, paths specified by LD_LIBRARY_PATH, system configuration files (/etc/ld.so.conf), and default system directories (/lib, /usr/lib).[8]

By setting the LD_PRELOAD environment variable, an attacker forces the injection of their malicious library at the absolute beginning of this sequence.[3] When the linker begins to resolve symbols needed by the target application, it resolves to the first definition it finds in the constructed link map. Since the preloaded library is the first element, any functions it defines override those in subsequent standard libraries.[4] This mechanism ensures that the attacker-supplied code controls the execution flow for the hooked functions.

2.2. Function Hooking: The Code Injection Primitive

Malicious payloads are typically compiled as shared object files (.so) using compiler flags such as -shared -fPIC (Position Independent Code).[4]

Crucially, shared libraries support constructor primitives. A common implementation uses the _init() function, which the dynamic linker is designed to execute immediately upon successfully loading the shared library.[11] This timing is essential, particularly for privilege escalation attempts, as it grants code execution before the target application's main() function is called or before privileges can be dropped.[11] For simple attacks, the payload involves defining the target function (e.g., a custom printf implementation) and optionally using the _init() function for initial setup or immediate shell spawning.[4]

2.3. Advanced Stealth: Calling the Original Function with dlsym(RTLD_NEXT)

In many exploitation scenarios, simply replacing a function entirely will lead to the crash or malfunction of the target application, immediately signaling intrusion.[12] To maintain stealth and stability, advanced malware, such as userland rootkits, must perform their malicious action and

then transparently invoke the original, legitimate system function.[13]

This capability is achieved through the use of the dlsym library call in conjunction with the special handle RTLD_NEXT.[14] The RTLD_NEXT handle instructs the dynamic linker to resolve the requested symbol not to the current hooked function, but to the *next* definition of that symbol found further down the link map (i.e., the original function in the standard library).[14]

The ability to reliably interpose a function, execute malicious logic, and then restore normal application behavior by calling the original function (RTLD_NEXT) is technically a necessary feature for debugging but also forms the foundation for high-fidelity userland rootkits. Without this ability to wrap and delegate control, the rootkit would introduce unacceptable system instability and be trivial to detect via system malfunction.[12]

III. Exploitation Scenarios: Defense Evasion and Data Exfiltration

The primary post-exploitation use cases for LD_PRELOAD involve establishing stealth (defense evasion) and data theft (espionage).

3.1. Defense Evasion: Implementing Userland Rootkits

Userland rootkits employ LD_PRELOAD to manipulate the data returned by standard system utilities, allowing malware to achieve invisibility without escalating to kernel-level privileges.[1]

A common goal is **process hiding**. This is achieved by intercepting functions that read the /proc filesystem, which is the mechanism used by utilities like ps, top, and lsof to list running processes. Specifically, the rootkit hooks the readdir function.[17] When the hooked readdir is invoked, it checks the enumerated process list for identifiers (PIDs or names) belonging to the malicious component. If a malicious entry is identified, the hooked function simply filters or skips that entry before returning the list to the user utility, making the process effectively invisible to system administrators using standard tools. Open-source examples, such as libprocesshider, implement this exact TTP.[17]

Malware also employs function hooking for **file and network hiding**. Rootkits like Azazel hook I/O functions such as fopen to conceal files or network activity associated with their presence. If the hooked function determines the requested file or resource is related to the rootkit, it simply fails the call or returns altered data, thus preventing the user from seeing the malicious artifacts.[13] Notable malware families, including Symbiote, HiddenWasp, Azazel, and Medusa, have all been observed leveraging dynamic linker hijacking for stealth and persistence.[1]

3.2. Data Theft and Espionage

The technique of function hooking allows attackers to intercept data streams used by legitimate

applications, enabling targeted data theft.

A critical example is **credential harvesting**. The Symbiote malware is known for hooking libc's read function. By intercepting read calls, Symbiote analyzes the calling process (checking if it is ssh or scp, for instance) and extracts inputs, such as user credentials, before transparently passing the data to the legitimate application. This facilitates lateral movement and data exfiltration.[17]

Another technique is **sub-process injection** used by malware like Ebury. Ebury uses LD_PRELOAD to inject its malicious module into programs launched via SSH sessions. To ensure persistence across process boundaries, Ebury hooks key execution functions like system, popen, execve, and execvp within the parent process. Any subsequent calls to these functions trigger the injection of the malicious library into the newly spawned child process.[2]

The choice of functions targeted by dynamic linker hijacking directly correlates with the attacker's objective. If the goal is evasion, I/O functions like readdir or fopen are targeted. If persistence and lateral movement are primary goals, execution functions like execve are hooked. If espionage is the objective, data-handling functions like read are intercepted.[2] This correlation provides an immediate diagnostic indicator for security analysts assessing the purpose of a detected malicious library.

IV. Privilege Escalation Vectors and Security Bypasses

LD_PRELOAD is commonly implicated in Local Privilege Escalation (LPE) attempts, which exploit deviations from the standard security model.

4.1. The Secure Execution Mode Barrier

The Linux dynamic linker implements a crucial security measure known as **Secure Execution Mode**. This mode is designed to prevent low-privileged users from hijacking high-privilege binaries, such as those with the Set-User-ID (SUID) or Set-Group-ID (SGID) bits set.[9]

Secure mode is automatically triggered when a process's real and effective user IDs differ (e.g., running a SUID binary), or if the binary possesses special file capabilities.[20] When operating in secure mode, the dynamic linker enforces severe restrictions on environment variables, often ignoring them completely.[9] For LD_PRELOAD, specifically, pathnames containing slashes (/), which typically point to user-controlled directories, are ignored. Libraries are only preloaded if they are system libraries found in standard search directories or if they are themselves owned by root and have the SUID bit set, which is a rare and complex configuration.[6]

This secure mode is a necessary architectural defense, ensuring that a binary running with elevated privileges (where the effective user ID differs from the real user ID) cannot have its trusted execution flow altered by environment variables arbitrarily set by the lower-privileged user. This prevents environmental trust issues, which are often more insidious than traditional

memory corruption attacks.[2]

4.2. Local Privilege Escalation (LPE) via Misconfiguration

Despite robust security mechanisms, LPE is often possible due to misconfiguration, specifically within the sudo utility.

The primary LPE vector involves exploiting the sudo env_keep option. If a system administrator explicitly configures sudoers to retain the LD_PRELOAD environment variable (env_keep LD_PRELOAD), the security barrier enforced by the dynamic linker is nullified because sudo explicitly allows the low-privileged environment setting to persist into the execution of the target SUID binary.[11]

The LPE execution flow proceeds as follows:

1. The attacker confirms that the sudo policy allows LD_PRELOAD retention.
2. The attacker compiles a payload (shell.so) containing the required privilege escalation sequence—setuid(0), setgid(0), and system("/bin/bash")—within the special shared library constructor function _init().[11] The payload also typically calls unsetenv("LD_PRELOAD") to prevent the variable from leaking to subsequent processes.[11]
3. The attacker executes a program they are authorized to run via sudo, explicitly passing the variable: sudo LD_PRELOAD=/path/to/shell.so <target_program>.[11]
4. Because sudo failed to sanitize the environment, the dynamic linker loads shell.so.
5. The _init() function executes immediately, gaining control while the process still holds the elevated privileges conferred by sudo, successfully spawning a root shell.[11]

LPE can also occur in contexts involving Linux File Capabilities. Although capabilities trigger secure mode, preloading is still theoretically possible if the preloaded library meets strict ownership requirements (root-owned, set-uid) or if a complex root wrapper is used to manage capability retention using kernel primitives like prctl(PR_SET_KEEPCAPS, 1).[21]

4.3. Historical and Modern Glitches/Vulnerabilities

The history of Linux security contains numerous vulnerabilities stemming from the dynamic linker's environment variable processing. Historical CVEs highlight flaws where privileged applications failed to adequately cleanse the environment before execution. Examples include CVE-2001-0872 (OpenSSH failure to cleanse LD_PRELOAD with UseLogin enabled) and CVE-2001-0169 (glibc failure to verify SUID/SGID status of preloaded libraries from cache).[24]

More recently, issues such as the "Looney Tunables" vulnerability (CVE-2023-4911), which exploited buffer overflow vulnerabilities in glibc's handling of the GLIBC_TUNABLES environment variable, demonstrate that complex parsing logic within ld.so remains an attack surface. Such parsing bugs can potentially lead to LPE by bypassing environment sanitation

checks and enabling arbitrary code execution.[26]

V. Adversarial Persistence and Real-World TTPs

Beyond immediate LPE, LD_PRELOAD is a primary mechanism for achieving persistent control and maintaining a hidden presence on compromised systems.

5.1. Establishing Persistence via Global Preloading (/etc/ld.so.preload)

While LD_PRELOAD set as an environment variable is often session-specific, global persistence can be achieved by writing the malicious shared object path into the configuration file /etc/ld.so.preload.[6]

This file contains a list of shared objects loaded before execution for *all* dynamically linked programs on the system.[6] Modification of this file requires root access, but once achieved, it ensures the malicious code is injected into nearly every running process, granting system-wide persistence and code injection capability.[1] This is a hallmark TTP of sophisticated userland rootkits.

Malware utilizing this system-wide mechanism includes Hildegard and Rocke, which modified /etc/ld.so.preload to hook libc functions. Rocke specifically used this to conceal its dropper and cryptomining payload from system monitoring tools, effectively hiding processes across the entire machine.[2] The OrBit rootkit also employs this method to guarantee the early loading of its malicious shared object.[17]

5.2. Malware Utilization Case Studies

- **Ebury (S0377):** This sophisticated Linux malware targets SSH servers for backdoor access and credential harvesting. Ebury maintains persistence and achieves lateral movement by using LD_PRELOAD to inject its module into programs launched by SSH, specifically by hooking execution functions (execve, system).[2]
- **Symbiote (S1216):** This userland rootkit focuses on espionage and stealth. It hooks multiple libraries, including libc and libpcap, to conceal its activities. Symbiote also leverages dynamic linker hijacking to bypass authentication mechanisms by hooking Linux Pluggable Authentication Module (PAM) functions.[17]
- **Winnti for Linux:** This threat group demonstrated the reusability of these techniques by incorporating the open-source **Azazel rootkit**, which relies on LD_PRELOAD, to conceal the main backdoor component and its malicious operations.[17]

Malware leveraging dynamic linker hijacking can be categorized by the severity of their persistence mechanism. The lowest tier relies on user-set environment variables (easily cleared). The highest tier involves modification of /etc/ld.so.preload. Achieving this high-tier persistence requires initial root access but, once installed, is extremely resilient, surviving

system reboots and affecting all future processes.[2]

VI. Detection, Defense, and Mitigation Strategies

Defending against dynamic linker hijacking requires a layered approach focusing on execution context monitoring and system hardening.

6.1. Attacker Evasion Techniques

Sophisticated rootkits incorporate mechanisms to evade detection, primarily by cleaning up their execution environment. For instance, after executing a privilege escalation payload, the malicious library will often call unsetenv("LD_PRELOAD") within its _init() function. This prevents the variable from being passed to subsequent child processes and removes the environmental artifact that forensic tools might use to trace the initial injection, complicating analysis.[11]

6.2. Defender Perspective: Analytic Focus and Monitoring

Effective detection must target both the staging of the malicious payload and the execution event itself, focusing on the system call level.

Monitoring Process Execution Context: High-fidelity detection relies on auditing or tracing mechanisms (such as eBPF or the Linux Audit System) to monitor execve events (sched_process_exec).[2] Security analysts should flag any process execution where the LD_PRELOAD environment variable is explicitly defined, particularly if the process is a high-value binary or if the library path points to an unusual location (like /tmp or a user's home directory).[2] Furthermore, detecting the creation of new shared object (.so) files in non-standard user directories can indicate the staging phase of an attack.[2]

File Integrity Monitoring (FIM): Given the critical nature of system-wide persistence, continuous File Integrity Monitoring of /etc/ld.so.preload is mandatory. Defenders should monitor for write access (security_file_open with write permissions) or suspicious renaming operations on this file, as these signals indicate attempts to install a system-wide rootkit (e.g., Hildegard, Rocke).[2]

Advanced Behavioral Detection: Since userland rootkits must inject and execute additional code, they often introduce measurable delays into critical system calls. Novel detection frameworks leverage runtime tracing to measure the time stamps of functions within targeted system calls (like readdir or read). Statistical analysis of these function execution times can detect an increase in the runtime distribution—the unavoidable latency introduced by the malicious interception layer—which serves as a key anomalous behavioral signal, even if the rootkit successfully conceals its files and processes from standard tools.[28]

Key Signals for Detecting Dynamic Linker Hijacking

Detection Signal (Event)	Source Location	TTP Indication	Analytic Action	Source(s)
sched_process_exec with LD_PRELOAD set	Environment variables list in process metadata.	Initial execution, LPE attempt, or targeted injection (Ebury).	Flag executions of privileged or unexpected binaries with this variable defined.	[2]
Write access to /etc/ld.so.preload	File Integrity Monitoring (FIM).	System-wide persistence attempt (Hildegard, Rocke).	Alert immediately; only system configuration tools should modify this file.	[2]
Creation of *.so files in user directories	File creation events (/tmp, user home).	Staging of malicious payload for subsequent execution.	Correlate file creation with later execve events involving LD_PRELOAD.	[2]

6.3. System Hardening and Configuration Mitigation

Effective mitigation requires addressing the underlying configuration risks:

1. **Sudo Hardening:** The most critical defense against LPE is ensuring that the sudo configuration strictly adheres to security best practices by sanitizing the execution environment. Administrators **must not** configure env_keep LD_PRELOAD or similar directives, as this directly bypasses the dynamic linker's secure mode.[2]
2. **Execution Prevention (M1038):** Application control solutions can be configured to restrict which shared libraries can be loaded by legitimate programs, preventing the injection of

unknown or malicious .so files.[2]

3. **Code-Level Defenses:** For developers building highly sensitive applications, defensive programming techniques can be employed. This includes implementing checks in the application's initialization routine to explicitly check for and clear dangerous environment variables like LD_PRELOAD. If the environment variable is present in a privileged context, the application should use _exit() to abort immediately, preventing the malicious library from executing its constructor.[21]

VII. Conclusion

The LD_PRELOAD mechanism is a potent vector for Linux exploitation, uniquely situated at the intersection of developer convenience and system architecture. Its exploitation enables a wide range of adversarial activities, from high-privilege shell spawning via configuration flaws to the establishment of undetectable userland rootkits for espionage and long-term persistence.

The analysis demonstrates that the efficacy of this technique relies on its superior symbol resolution priority, which allows for function interposition to occur before any other library is loaded. The core threat model has evolved: modern attackers are less reliant on simple LPE (due to strong enforcement of secure execution mode) and instead focus on exploiting subtle configuration weaknesses (sudo env_keep) or achieving permanent, system-wide compromise via modification of /etc/ld.so.preload. Furthermore, attackers have refined their payloads using dlsym(RTLD_NEXT) to ensure stealth and application stability, dramatically increasing the complexity required for detection.

To effectively counter these threats, security posture must extend beyond signature-based detection. The recommendations emphasize architectural hardening, particularly the rigorous sanitization of environments for privileged operations (e.g., sudo), coupled with continuous, high-fidelity runtime monitoring and integrity checks on critical system configuration files and process execution context. The dynamic linker remains a critical and frequently targeted component of the Linux security architecture.

Works cited

1. Linux Detection Engineering - A Continuation on Persistence Mechanisms — Elastic Security Labs, accessed October 29, 2025, https://www.elastic.co/security-labs/continuation-on-persistence-mechanisms
2. Hijack Execution Flow: Dynamic Linker Hijacking, Sub-technique T1574.006 - Enterprise, accessed October 29, 2025, https://attack.mitre.org/techniques/T1574/006/
3. What Is the LD_PRELOAD Trick? | Baeldung on Linux, accessed October 29,

2025, https://www.baeldung.com/linux/ld_preload-trick-what-is
4. LD_PRELOAD and Dynamic Library Hijacking in Linux | by Hem Parekh | Medium, accessed October 29, 2025, https://medium.com/@hemparekh1596/ld-preload-and-dynamic-library-hijacking-in-linux-237943abb8e0
5. LD_PRELOAD a Powerful Tool!!! - Medium, accessed October 29, 2025, https://medium.com/@akankshakoul/ld-preload-a-powerful-tool-c5170f90d9d0
6. ld.so(8) - Linux manual page - man7.org, accessed October 29, 2025, https://man7.org/linux/man-pages/man8/ld.so.8.html
7. User-space library rootkits revisited: Are user-space detection mechanisms futile? - arXiv, accessed October 29, 2025, https://arxiv.org/html/2506.07827v1
8. What is the order that Linux's dynamic linker searches paths in?, accessed October 29, 2025, https://unix.stackexchange.com/questions/367600/what-is-the-order-that-linuxs-dynamic-linker-searches-paths-in
9. The Linux Security Journey — Secure Execution Mode | by Shlomi Boutnaru, Ph.D., accessed October 29, 2025, https://medium.com/@boutnaru/the-linux-security-journey-secure-execution-mode-325137c3c76a
10. Guide: Using dynamic linking/loading to interpose on malloc | Nicolas van Kempen, accessed October 29, 2025, https://nvankempen.com/2025/06/04/guide-using-dynamic-loader-interpose-malloc.html
11. Linux Privilege Escalation - TryHackMe, accessed October 29, 2025, https://tryhackme.com/room/linprivesc
12. Function Hooking Using LD_PRELOAD | by David de Villiers - InfoSec Write-ups, accessed October 29, 2025, https://infosecwriteups.com/a-gentle-introduction-to-function-hooking-using-ld-preload-1714124a6eb9
13. Hooking libc using Go shared libraries | Gopher Academy Blog, accessed October 29, 2025, https://blog.gopheracademy.com/advent-2015/libc-hooking-go-shared-libraries/
14. dlsym(3) - Linux manual page - man7.org, accessed October 29, 2025, https://man7.org/linux/man-pages/man3/dlsym.3.html
15. C/C++ Function Hooking with LD_PRELOAD - Modest Destiny, accessed October 29, 2025, https://blog.modest-destiny.com/posts/c-cpp-function-hooking-with-ld-preload/
16. Correct usage of `LD_PRELOAD` for hooking `libc` functions | Tudor Brindus, accessed October 29, 2025, https://tbrindus.ca/correct-ld-preload-hooking-libc/
17. Linux rootkits explained – Part 1: Dynamic linker hijacking | Wiz Blog, accessed October 29, 2025, https://www.wiz.io/blog/linux-rootkits-explained-part-1-dynamic-linker-hijacking

18. gianlucaborello/libprocesshider: Hide a process under Linux using the ld preloader (https://sysdig.com/blog/hiding-linux-processes-for-fun-and-profit/) - GitHub, accessed October 29, 2025, https://github.com/gianlucaborello/libprocesshider
19. Emulating the Persistent and Stealthy Ebury Linux Malware - AttackIQ, accessed October 29, 2025, https://www.attackiq.com/2024/09/12/emulating-ebury-malware/
20. How to globally enable ld.so secure-execution mode for all applications?, accessed October 29, 2025, https://security.stackexchange.com/questions/241063/how-to-globally-enable-ld-so-secure-execution-mode-for-all-applications
21. Does using linux capabilities disable LD_PRELOAD - Stack Overflow, accessed October 29, 2025, https://stackoverflow.com/questions/18058426/does-using-linux-capabilities-disable-ld-preload
22. Heck, glibc by default still allows LD_PRELOAD pretty much everywhere, and mos... | Hacker News, accessed October 29, 2025, https://news.ycombinator.com/item?id=36269642
23. Unusual LD_PRELOAD/LD_LIBRARY_PATH Command Line Arguments | Elastic Security [8.19], accessed October 29, 2025, https://www.elastic.co/guide/en/security/8.19/unusual-ld-preload-ld-library-path-command-line-arguments.html
24. LD_PRELOAD - CVE: Common Vulnerabilities and Exposures, accessed October 29, 2025, https://www.cve.org/CVERecord/SearchResults?query=LD_PRELOAD
25. Remote LD_PRELOAD Exploitation - elttam, accessed October 29, 2025, https://www.elttam.com/blog/goahead/
26. Looney Tunables: A Deep Dive into the glibc's ld.so Vulnerability (CVE-2023-4911), accessed October 29, 2025, https://www.relianoid.com/blog/looney-tunables-a-deep-dive-into-the-glibcs-ld-so-vulnerability-cve-2023-4911/
27. LD_PRELOAD code injection detected - Aqua Security, accessed October 29, 2025, https://aquasecurity.github.io/tracee/latest/docs/events/builtin/signatures/ld_preload/
28. Trace of the Times: Rootkit Detection through Temporal Anomalies in Kernel Activity - arXiv, accessed October 29, 2025, https://arxiv.org/html/2503.02402v1
29. Eliminate LD_PRELOAD and other Dangerous Environment Variables - Whonix Forum, accessed October 29, 2025, https://forums.whonix.org/t/eliminate-ld-preload-and-other-dangerous-environment-variables/10594