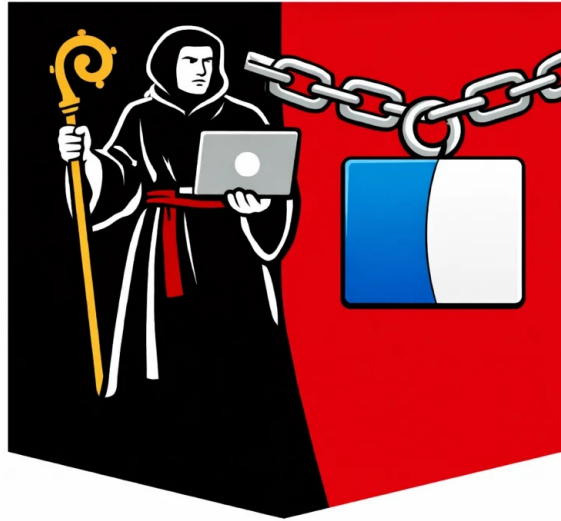# The Glarus Exploit:

The State of Link Exploitation on macOS 26

Golden Helm Securities — December 2025

December 22, 2025

Abstract

This research examines the current state of symbolic and hard link exploitation on macOS 26 (Tahoe), focusing on privilege escalation from standard user to root. I present Glarus, a vulnerability in the dirhelper system daemon that combines a string truncation bug with a time-of-check time-of-use (TOCTOU) race condition to achieve arbitrary file ownership changes.

The investigation reveals that Apple has implemented targeted hardening via TCC "Administer Computer" permissions specifically protecting authentication files from many link attacks that would provide straightforward access to root code execution. I explore other escalation paths, do initial documentation of Apple's defense-in-depth architecture, and discuss the implications for macOS symbolic and hard link security research. The aim of this research is to serve as both a definitive starting point and comprehensive summary of finding and exploiting symbolic and hard link vulnerabilities on macOS 26.

Note on Patch Status: The Glarus vulnerability was patched in **macOS 26.1 Beta 3**, released on October 13, 2025, two days prior to my disclosure to Apple. Despite being patched, Glarus remains a valuable case study demonstrating the power of chaining path manipulation bugs with symbolic and hard link primitives.

Table of Contents

1. Symbolic and Hard Link Overview on macOS

Most Mac users think of the filesystem simply as directories (folders) and files. However, macOS also supports two special file types—symbolic links (symlinks) and hard links—which have unique properties exploitable by attackers.

Symbolic Links

Instead of containing actual file data, a symlink is a special file that points to another location on the filesystem. Think of it as a redirect or shortcut. The following example demonstrates creating a symbolic link and examining its properties:

```
user@mac Desktop % ln -s /etc/openldap/ldap.conf sample_link
# Creates a symbolic link named "sample_link" pointing to ldap.conf

user@mac Desktop % ls -l sample_link
lrwxr-xr-x 1 user staff 23 Oct 17 15:54 sample_link -> /etc/openldap/ldap.conf
# The 'l' at the start indicates this is a link
# The arrow (->) shows what the symlink points to
```

Key properties of symbolic links:

• The target doesn't need to exist. A symlink is simply a string representing a path

• Symlinks can point to both directories and files

• When a program accesses the symlink, the system automatically redirects to the target

• The `lstat()` system call examines the link itself, while `stat()` follows the link to its target

• The `lchown()` function operates on a symlink itself rather than following it. The "l" prefix indicates "don't follow links"

Hard Links

Hard links are more restricted but more powerful. A hard link creates an additional directory entry pointing to the same inode (the underlying file data on disk). Both the original filename and the hard link reference identical data. They are effectively the same file with two names:

```
user@mac Desktop % echo "Hello, world" > original.txt
# Create a regular file with some content

user@mac Desktop % ls -li original.txt
2856323 -rw-r--r-- 1 user staff 13 Oct 17 10:14 original.txt
# The number 2856323 is the inode - the file's unique identifier on disk
# The '1' after the permissions is the link count

user@mac Desktop % ln original.txt hardlink.txt
# Create a hard link - another name for the same file data

user@mac Desktop % ls -li original.txt hardlink.txt
2856323 -rw-r--r-- 2 user staff 13 Oct 17 10:14 hardlink.txt
2856323 -rw-r--r-- 2 user staff 13 Oct 17 10:14 original.txt
# Both files share the same inode (2856323)
# Link count is now 2 - two directory entries point to this data
# Modifying either file changes both; deleting one leaves the other intact
```

Hard link restrictions:

1. They can only point to files (not directories)

2. The target file must already exist

3. The link and target must be on the same filesystem/volume

4. On macOS, certain files are protected from hardlinking by SIP, TCC, and MACF policies

5. The `st_nlink` value returned by `stat()` indicates the number of hard links to a specific inode

2. macOS Filesystem Security Boundaries

The combination of hard and symbolic links presents attackers with numerous opportunities for privilege escalation. These links can be abused by someone with local code execution to redirect privileged operations to a different file. The privileges being targeted typically fall within four categories:

1. User to Root

Escalating from user to root offers clear advantages: read and write access to the entire filesystem, plus access to additional root-only kernel attack surface for further kernel escalation. There are many other user accounts on macOS that may be useful to escalate to (`_locationd`, `_appleinstalld`, `_trustd`, etc.), so this can be abstracted as a more general userA-to-userB security boundary. However, this research focuses on the user-to-root boundary because root is generally the highest privileged user account and therefore the best to target.

There are several other interesting security boundaries that can suffer from hard link and symbolic link vulnerabilities. They will not be the focus of this work but are listed here for a brief look at the bigger picture:

2. Sandbox Escape

Many processes are launched with a sandbox profile that creates a custom restricted environment in terms of filesystem access, system calls, and IPC capabilities. Links may provide escape routes from these sandboxed environments by redirecting privileged operations to restricted locations.

3. TCC (Transparency, Consent, and Control) Bypass

TCC protects access to sensitive user data including photos, messages, contacts, and location. Link vulnerabilities may bypass these protections by tricking privileged processes into accessing protected data on behalf of an unprivileged attacker.

4. SIP (System Integrity Protection) Bypass

SIP protects the operating system itself. A bypass would allow modification of protected system files in `/System`, `/usr`, `/bin`, and `/sbin`.

As I will demonstrate, these boundaries interact in complex ways. Apple has layered multiple protection mechanisms that must all be considered when evaluating attack vectors.

3. Historical User-to-Root Link Vulnerabilities

Symbolic and hard link exploits have a long history on Apple platforms. One notable example was the 2013 evasi0n iPhone jailbreak, which exploited `lockdownd`, a root process that would `chmod()` the file at `/var/db/timezone` on startup. By replacing that path with a symlink (via MobileBackup), the evasi0n team gained the ability to make any root-owned file world-writable.

This pattern, a root process performing file operations on an attacker-controllable path, remains the foundation of link-based attacks today.

4. Vulnerability Discovery Process

I monitored root processes calling `chown()`, `chmod()`, or `lchown()` using DTrace. Here is a small example:

```
#!/usr/sbin/dtrace -s

syscall::lchown:entry
/uid == 0/
{
    self->path = copyinstr(arg0);
    self->target_uid = arg1;
}

syscall::lchown:return
/self->path != NULL && self->target_uid != 0/
{
    printf("%s[%d] lchown(%s, uid=%d) = %d\n",
           execname, pid, self->path, self->target_uid, arg1);
    self->path = NULL;
}
```

This identified `dirhelper` (`/usr/libexec/dirhelper`), a root daemon providing directory management services to sandboxed applications. Because dirhelper is specifically designed to perform filesystem operations on behalf of sandboxed processes, it is an ideal escalation target. Any vulnerability in its path handling could allow a sandboxed attacker to manipulate files they shouldn't have access to.

5. Vulnerability Case Study: Glarus

The Glarus exploit yields an arbitrary `lchown()` primitive, the ability to change ownership of any file on the filesystem to the current user. It achieves this by combining two vulnerabilities.

5.1 Bug #1: String Truncation via Unchecked `strlcat()`

The dirhelper daemon exposes its functionality through a MIG (Mach Interface Generator) IPC interface. When reverse engineering MIG-based services, functions beginning with `__x` typically represent the entry points for each exposed routine. Through disassembly, I identified that `dirhelper_internal_server` has a unique code path when the calling process is inside a container:

```
// Pseudocode from reverse engineering
void handle_container_request(char *app_id, uint32_t buffer_size) {
    char path[buffer_size];  // Attacker-controlled size!

    // Get container path
    // e.g., "/Users/victim/Library/Containers/com.app/Data"
    sandbox_container_path_for_audit_token(audit_token, path, buffer_size);
```

```
    // Check if path ends with '/'
    size_t len = strlen(path);
    if (path[len-1] != '/') {
        strlcat(path, "/tmp/", buffer_size);  // ← VULNERABLE!
    } else {
        strlcat(path, "tmp/", buffer_size);
    }

    // Create directory and set ownership
    _makeDirectoryWithUIDAndGID(path, uid, gid, 0700);
}
```

The dirhelper daemon uses `strlcat()` to append "/tmp/" to container paths but does not check the return value. The `buffer_size` parameter is controlled by the calling process. Note, `strlcat()`'s size parameter includes the null terminator, so when `buffer_size` is set to `strlen(container_path) + 5`, only 4 characters can be appended before the null byte. Therefore, with a buffer of this size, the trailing "/" is truncated, resulting in a path ending in "/tmp" instead of "/tmp/".

Why does this matter? A path with a trailing slash must refer to a directory. Without the trailing slash, `/Users/.../Data/tmp` can refer to a directory, file, symbolic link, or hard link. This ambiguity enables the second vulnerability.

5.2 Bug #2: TOCTOU Race Condition

The truncated path is passed to `_makeDirectoryWithUIDAndGID`:

```
int _makeDirectoryWithUIDAndGID(const char *path, uid_t uid,
                                gid_t gid, mode_t mode) {
    int result;

    // Step 1: Create directory
    result = mkdir(path, mode);
    if (result != 0 && errno != EEXIST) {
        return -1;
    }

    // ← RACE WINDOW: Attacker can replace directory here!

    // Step 2: Change ownership
    result = lchown(path, uid, gid);  // ← Operates on whatever is at 'path'
    if (result != 0) {
        return -1;
    }

    return 0;
}
```

The daemon calls `mkdir()` then `lchown()`. While `lchown()` doesn't follow symlinks, the truncated path creates a race window: if an attacker replaces the directory with a hard link to a privileged file between these operations, they will gain ownership of whatever that hard link is.

The Problem: The race requires replacing a root-owned directory, but as a regular user, we cannot delete root-owned files.

The Solution: We don't need to delete it. We can move it. Since the directory exists inside a user-owned container folder (`Data`), we can rename the parent directory to effectively "remove" the root-owned child from that path.

5.3 The Swap Technique

The attack uses a directory swap via **rename operations**:

- **Before:** `Data/tmp` is a directory created by `mkdir()`

- **Swap Step 1:** `rename(Data, Data_backup)` — Move the real Data directory out of the way

- **Swap Step 2:** `rename(Fake, Data)` — Move our prepared Fake directory into position as Data

- **After:** `Data/tmp` now resolves to what was `Fake/tmp`

- **Result:** `lchown("Data/tmp")` changes ownership of `Fake/tmp`

If `Fake/tmp` is a hardlink to a root-owned file, the attacker now owns that file.

Filesystem State Diagram:

BEFORE (mkdir)	DURING (swap)	AFTER (lchown)
```Container/``` ```███ Data/ [user]``` ```█ ███ tmp/ [root] ←mkdir``` ```███ Fake/ [user]``` ``` ███ tmp [root]``` ``` █ hardlink to``` ``` /etc/target```	```Container/``` ```███ Data_backup/ [user]``` ```█ ███ tmp/ [root]``` ```███ Data/ [user] ←renamed from ███ Data/ [user]``` ```█ ███ tmp [root]``` ```█ █ hardlink``` ```███ (Fake moved to Data)```	```Container/``` ```███ Data_backup/ [user]``` ```█ ███ tmp/ [root]``` ```███ Data/ [user]``` ``` ███ tmp [USER] ←lchown!``` ``` █ hardlink``` ``` /etc/target [USER]```
dirhelper creates Data/tmp	Attacker renames: Data → Data_backup Fake → Data	lchown follows path to our hardlink!

> **Why rename() instead of symlink()?** Apple's sandbox infrastructure specifically blocks creating symlinks named "Data" in container directories. While you can delete or rename a directory named "Data", you cannot create a symlink with that name. The rename-based approach bypasses this restriction entirely by physically moving directories rather than creating symbolic links.

Directory Structure:

```
~/Library/Containers/com.example.dirhelper-client/
███ Data/                    ← Real container data directory
█    ███ (empty, or app data)
███ Data_backup/             ← Where Data moves during swap
█    ███ tmp/                ← Root-created directory ends up here
███ Fake/
     ███ tmp                 ← Hardlink to target file (same inode!)
```

The hardlink must be created from an unsandboxed process because the App Sandbox blocks hardlinks to system files:

```
// Inside sandbox: EPERM regardless of file permissions
link("/etc/pam.d/sudo", "./Fake/tmp");  // Returns -1, errno=EPERM
```

Two-Process Coordination: The exploit requires two processes working in concert:

1. Sandboxed Client Process: A containerized application that sends the crafted MIG request to dirhelper. This process triggers the vulnerable code path because it runs inside a container, causing dirhelper to use the container-specific logic.

2. Unsandboxed Race Process: A separate user-space process (not sandboxed) that creates the hardlink to the target file in `Fake/tmp`, monitors for the `mkdir()` via kqueue filesystem events, and

performs the rapid rename swap (`rename(Data, Data_backup)` followed by `rename(Fake, Data)`) to win the race.

These two processes run concurrently. The unsandboxed process sets up the directory structure and hardlink, then waits. When the sandboxed client sends the request, dirhelper calls `mkdir()`, which the race process detects. The race process then rapidly performs the two rename operations before dirhelper can call `lchown()`. The C-based implementation achieves microsecond precision using kqueue for filesystem monitoring. Now that we can point the lchown to an arbitrary file, through hard links, we need to decide what to target.

6. Apple's Defense-in-Depth Architecture

6.1 Why Target Authentication Files?

Files like `/etc/pam.d/sudo` and `/etc/sudoers` have been prime targets for privilege escalation attacks because modifying them provides immediate, reliable root access. The attack is straightforward: if you can change the ownership of these files to your user, you can then modify them to bypass root authentication entirely.

For `/etc/pam.d/sudo`, adding a single line like `auth sufficient pam_permit.so` at the top allows any user to run sudo without a password. For `/etc/sudoers`, adding `username ALL=(ALL) NOPASSWD: ALL` achieves the same result.

This attack pattern has a long history. At Pwn2Own 2016, lokihardt demonstrated a macOS privilege escalation by exploiting sudo's timestamp files in `/var/db/sudo/`. By manipulating file ownership and timestamps, an attacker could trick sudo into believing the user had recently authenticated, bypassing the password prompt entirely.

Given this history of file-based attacks on sudo and PAM, Apple has implemented specific protections for these high-value targets.
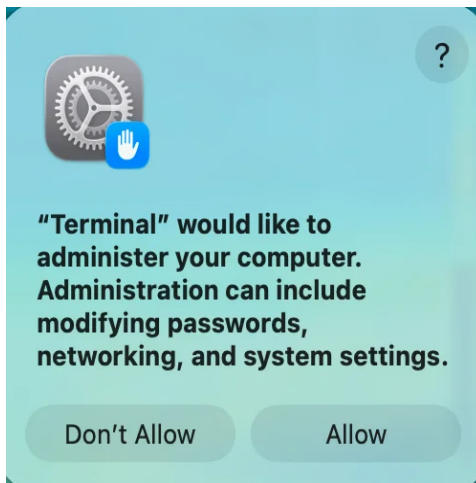
6.2 TCC "Administer Computer" Protection

When attempting to create a hardlink to `/etc/pam.d/su`, I observed:

```
$ ln /etc/pam.d/su test
# System dialog: "Terminal would like to administer your computer"
# [Don't Allow] [Allow]

# If "Don't Allow":
error: kernel System Policy: ln(22867) deny(1) file-link /private/etc/pam.d/su
```

Note: Clicking "Allow" will cause the exploit to proceed (no password required). This is a one-time prompt per application.

Crucially, these files do NOT have SIP flags:

```
$ ls -lO /etc/sudoers /etc/pam.d/sudo
-r--r--r-- 1 root wheel compressed 283 Sep 8 23:15 /etc/pam.d/sudo
-r--r----- 1 root wheel compressed 1709 Sep 8 23:15 /etc/sudoers
```

The `compressed` flag is APFS compression, not security-related. There's no `restricted` or `sunlnk` flag. This reveals a sparsely documented protection layer: while Apple's TCC protection of the `/etc/pam.d` directory was briefly noted when Monterey was released in October 2021 (see SentinelOne and JumpCloud), the specific behavior of blocking hardlink creation to authentication files, and the exact list of protected files, has not been publicly documented by Apple or security researchers to my knowledge.

Key Finding: Three-Layer Defense Architecture

Layer	Mechanism	Protection
Layer 1	SIP	Protects /System, /usr, /bin, /sbin via SF_RESTRICTED flag
Layer 2	TCC "Administer Computer"	Protects auth files via MACF policy (path-based)
Layer 3	Application-level checks	sudo verifies /etc/sudoers owned by root

6.3 Testing Results

I tested hardlink creation to various system files on a fresh macOS 26 VM:

File	Protection	Hardlink Result
`/etc/pam.d/sudo`	TCC Protected	Prompt required
`/etc/pam.d/su`	TCC Protected	Prompt required
`/etc/sudoers`	Unprotected	✓ Succeeds, but sudo rejects non-root ownership
`/etc/newsyslog.conf`	Unprotected	✓ Succeeds silently

`/etc/ssh/sshd_config`	Unprotected	✓ Succeeds silently
`/etc/hosts`	Unprotected	✓ Succeeds silently
`/etc/shells`	Unprotected	✓ Succeeds silently

7. Exploitation Techniques

Research Note: As discussed in §6.1, the most direct authentication file targets are blocked by TCC. The techniques below represent initial findings on alternative targets, not a comprehensive survey. Arbitrary chown/chmod primitives on macOS deserve significantly more research. There are likely many additional files and attack vectors. I encourage the security community to explore this space further.

7.1 Alternative Targets

With authentication files protected by TCC, exploitation requires targeting files outside this protection layer:

Target	Attack	Trigger	Impact
`/etc/newsyslog.conf`	Add rotation rule to copy payload to /etc/sudoers.d/	Daily job	Root (delayed)
`/etc/hosts`	Redirect domains to attacker IP	Immediate	Phishing/Supply chain
`/etc/ssh/sshd_config`	Enable root login, add authorized keys	SSH restart	Remote root (if SSH enabled)
`/etc/shells`	Add malicious shell path	User login	Persistence
`/Library/Launch Daemons/*.plist`	Modify third-party daemon config	Reboot	Root execution

The "Last Mile" Problem: Some targets like newsyslog.conf require a trigger event (daily job, reboot). Others like /etc/hosts take effect immediately.

7.2 Third-Party LaunchDaemons

While a fresh macOS installation contains no third-party LaunchDaemons, any installed third- party software that creates LaunchDaemons provides an easy escalation target. Common examples include:

• Virtualization software (VMware, Parallels, VirtualBox)

• Security tools (antivirus, endpoint protection)

• Cloud sync clients (Dropbox, Google Drive, OneDrive)

• Development tools (Docker, database servers)

Third-party LaunchDaemons in `/Library/LaunchDaemons/` are not protected by SIP or TCC.

7.3 Additional Considerations

The above table is not exhaustive. Any root-owned configuration file outside SIP/TCC protection that influences privileged execution is a potential target. Researchers should explore:

• Database configuration files

• Web server configurations (Apache, nginx)

• Cron-equivalent scheduled task configs

• Application-specific privilege escalation paths

8. Conclusions and Recommendations

Hard and symbolic links remain a viable attack vector on macOS Tahoe. Apple's three-layer defense (SIP, TCC "Administer Computer", application-level validation) blocks many of the best attacks on authentication files, but semi-arbitrary file ownership changes remain a powerful exploit primitive.

Key Recommendations

Given the current environment in macOS with regards to symbolic and hard links, here are some closing thoughts:

For Engineers and Developers:

• Be extra careful with string operation return values especially when it involves paths. Truncation, not just overflows, can result in vulnerabilities

• Avoid relying on `lchown` to protect you when the path involves components of a different user

• When possible always use the `AT_SYMLINK_NOFOLLOW` option when doing path operations

• Assume paths can completely change contents between system calls when in different user directories

• Use atomic operations and file descriptors whenever possible

For Security Researchers:

• A running dtrace log of chown/chmod can be enough to discover suspicious code

• When an arbitrary chown/chmod is achieved focus on files outside TCC Admin protections

• Chaining vulnerabilities may be necessary to exploit, don't give up easily

• Third-party software installed dramatically expands the link target attack surface

• Arbitrary chown/chmod primitives deserves more research

9. Reporting Timeline

Report ID	Date	Description
OE1103366985318	August 4, 2025	Initial dirhelper issue

OE11032565505713	August 14, 2025	Additional findings
—	October 13, 2025	macOS 26.1 Beta 3 released (Glarus TOCTOU patch)
OE11004064159426	October 16, 2025	Glarus vulnerability reported
—	December 12, 2025	macOS 26.2 released (fix for OE1103366985318)

Proof of Concept

Available at: [GitHub link upon publication]

Repository Structure:

```
glarus_poc_final/
███ README.md
███ QUICKSTART.md
███ src/
█    ███ dirhelper_client.c
█    ███ race_swap.c
█    ███ entitlements.plist
███ scripts/
█    ███ build.sh
█    ███ setup.sh
█    ███ exploit.sh
█    ███ cleanup.sh
███ docs/
     ███ TECHNICAL_ANALYSIS.md
```

Source Files:

• `dirhelper_client.c` — MIG client that triggers the string truncation vulnerability. Sends crafted buffer size to dirhelper causing path truncation from "/Data/tmp/" to "/Data/tmp". Includes macOS 26 compatibility via mig_get_reply_port().

• `race_swap.c` — TOCTOU race condition binary using kqueue for filesystem monitoring. Detects mkdir() and rapidly executes rename-based swap (Data→Data_backup, Fake→Data) before lchown() executes.

• `entitlements.plist` — Sandbox entitlements for the containerized client application.

Scripts:

• `build.sh` — Compiles dirhelper_client.c into a sandboxed app bundle and builds the race_swap binary.

• `setup.sh` — Creates the exploit directory structure with Data/ and Fake/ directories, and creates the hardlink from Fake/tmp to the target file.

• `exploit.sh` — Main orchestrator that coordinates the race binary and triggers dirhelper in a loop until the race is won.

• `cleanup.sh` — Removes exploit artifacts and restores the container to a clean state.

Usage:

```
$ cd glarus_poc_final
$ ./scripts/build.sh
$ ./scripts/exploit.sh /etc/hosts
```

On success, the target file's ownership will be changed to the current user. The exploit may require multiple attempts due to the race condition timing.

References

- evasi0n Jailbreak - Timezone Vulnerability (2013): theiphonewiki.com
- lokihardt, Pwn2Own 2016 - Safari + macOS Exploit: blog.hackerschool.de
- SentinelOne, "Apple's macOS Monterey | 6 Security Changes" (2021): sentinelone.com
- JumpCloud, "Granting Permissions for Monterey PAM" (2021): jumpcloud.com
- CWE-367 (TOCTOU Race Condition)
- strlcat(3) man page