Intro to iOS Application Security Testing

A Codemash Pre-Compiler - January 7, 2020

Presented By: Hans Weisheimer
Email: hans@ath0.io
Twitter: @hweisheimer
GitHub: https://github.com/hweisheimer/ios-testing

Prologue

We're cheating today. This course is intended to teach techniques that work on stock ("jailed") devices, and we will stick as closely as possible to that. However, in the interest of giving Windows users are a more meaningful experience, and to work around some bugs that I've recently run into, we will be using Jailbroken virtual devices.

This is being done to avoid a single problematic requirement for jailed devices: that the application be launched with an attached debugger in order to disable signature checks on injected code. This can't be done from a Windows host (afaik), and I've run into a lot of pairing issues lately with libimobiledevice on macOS. I've included notes in the "Run the Application" section that outline the normal launch process.

Chapter One – Progress Bars and Busywork

Obviously, everyone went through the pre-requisites ahead of time and is definitely not trying to install several GB worth of stuff right now. Good job. Appendix A won't be of any interest to you.

Device Setup

Open Safari (Mac) or Chrome (Windows). On Mac, the virtual device graphics perform poorly in Chrome and Firefox.

Navigate to https://hans.corellium.com/

Sign in to Corellium using the credentials provided (see instructor if you didn't receive a slip of paper). You should see a project with one device, an iPhone 7. Click on the device name, then click to view the screen. Tap the home button twice to unlock the device. If your taps do not register, ensure that you are physically pressing down on your touchpad.

Download the OPVN file and import it into your OpenVPN client. Connect to the VPN.

Physical Devices: You will access these over USB, and omit the `--network` and `--host` options later on.

Proxy Setup

Start **Burp Suite**, proceed as a temporary project, and open to the **Proxy** tab. Click the **Interception** button to turn off auto-intercept. Under the **Options** tab, add a listener for port 8080 on the 10.11.x.x interface. This is the Corellium VPN interface. Make note of your PC's IP address for the next step.

On the virtual device, open **Settings** -> **WiFi** -> **Corellium**. Tap **Configure Proxy**, select **Manual**, enter the address and port number of your PC's proxy interface, and tap **Save**.

Open **Safari** on the device, navigate to http://httpforever.com/ and confirm that the site loads.

Return to Burp Suite's Proxy tab, view HTTP history, and confirm that httpforever.com was accessed through the proxy.

We will deal with the CA Certificate later – for now, we want it to be untrusted.

Install the Application

In the interest of time (and supporting Windows users), the application has already been patched to include the Frida Gadget and re-signed. Information on this process is in the appendix.

Download the IPA file from https://hans-ios-testing.s3.us-east-2.amazonaws.com/DVIA-v2-frida-codesigned.ipa

From the Corellium device page, navigate to the **App Browser** tab and click **Install IPA**. Select the downloaded IPA file.

Because these devices are enrolled in the Apple Developer Program, you will not need to explicitly trust the developer account. If you sign apps with a free account, you would need to trust the developer certificate under **Settings** -> **General** -> **Device Management**.

Run the Application

Tap the DVIA-v2 icon on the home screen, and proceed to the next chapter. The information below is for future reference.

In a more typical development setup (Mac + stock physical device), you would need to launch the application with an attached debugger. This bypasses some code signing checks, allowing Frida to inject new code at runtime. To do so, you would:

1. Install ios-deploy
2. Unzip the IPA – you will see a `Payload` directory containing `DVIA-v2.app`
3. Run `ios-deploy -b Payload/DVIA-v2.app -L` (uploads the app, launches it with lldb attached, then exits lldb)

Alternatively, Frida's interception API can be disabled in `FridaGadget.config` by setting `code_signing` to `required`. Most of today's activities will work in this mode, but the more interesting Frida/Objection features will not. Examples of things that will not work in this mode are: Certificate pinning bypass, Biometric auth bypass, Jailbreak detection/simulation, and pretty much all of the "hooking" commands.

If you'd like to experiment with this alternative build, it can be downloaded here: https://hans-ios-testing.s3.us-east-2.amazonaws.com/DVIA-v2-frida-codesigned-no-intercept.ipa

Chapter Two – Hax0ring

Exercise 1: Snooping on Traffic

Open the **DVIA-v2** application and select **Network Layer Security** from the menu. Type some things into the alleged credit card form.

Vulnerability: Tap **Send Over HTTP** and observe the data being transmitted over (unencrypted) HTTP in Burp Suite. HTTPS is ubiquitous at this point, and should be used virtually everywhere. Whenever sensitive data is involved (like credit card data!), it's mandatory.

Vulnerability: Tap **Send Over HTTPS**. Do you see the traffic in Burp Suite again? That's weird, because we never trusted Burp's CA certificate. The application is not performing proper certificate validation, but where is the offending code/config?

> **Hint**: Tap the **App Transport Security** at the bottom of the screen.
>
> To view the application's `Info.plist` file, execute the following on your laptop:
>
> ```
> objection --network --host 10.11.0.1 run ios plist cat Info.plist
> ```
>
> Look for the `NSAppTransportSecurity` key – do you see anything suspicious?

Once you've determined that an application uses HTTPS in a sane way, you will need to trust the proxy's CA certificate in order to observe/intercept the traffic. Let's do that now:

> Open **Safari** on your device and navigate to http://burp. Tap the **CA Certificate** link, and follow the prompts to install the Configuration Profile.
>
> Open **Settings**, tap **Profile Downloaded**, then tap **Install** (it will ask a few times).
>
> Go to **Settings** -> **General** -> **About** -> **Certificate Trust Settings**, and enable full trust for **PortSwigger CA**.
>
> Return to Safari, navigate to https://codemash.org, and confirm that the site loads.
>
> Return to Burp Suite's **Proxy** tab, view HTTP history, and confirm that codemash.org was accessed through the proxy.

Activity: Bypass Certificate Pinning

Certificate Pinning adds extra validation on top of the TLS defaults. If your threat model demands it, it's a way to protect against Mallory in the Middle attacks where an attacker has convinced a user to install a CA certificate, or convinced a trusted CA to issue them a certificate for the target domain.

Developers can "pin" to a specific public key or certificate that they own and use, or to a CA intermediate certificate. This introduces some amount of fragility, as a number of routine changes could potentially break your application.

Regardless of one's stance on certificate pinning, testers should be prepared to work around it when present.

Back in the **DVIA-v2** app's **Network Layer Security** screen, Tap **Send Using Certificate Pinning** – oh no, the request is blocked!

Open the Objection REPL on your laptop:

```
objection --network --host 10.11.0.1 explore
```

Now, we'll start a "job" (background task) to fake out some common pinning checks. At the REPL prompt, type:

```
ios sslpinning disable
```

Tap the button again, and look at the traffic in Burp - success!

To stop a running job in Objection, use the `jobs list` and `jobs kill [id]` commands (`jobs kill` didn't work quite right the last time I tried).

Bonus: On a Mac, you can monitor an iOS device's raw network traffic using **Remote Virtual Interfaces**. It's like having a mirror port for your phone! These interfaces are managed using the `rvictl` command. After setting up an interface, you can use sniffing tools such as Wireshark to observe *any* type of traffic (whereas Burp Suite and its ilk are limited to HTTP). If you suspect that an application uses non-HTTP protocols, this is useful.

Exercise 2: Perusing the Filesystem

Take some time to browse the application's filesystem. Keep in mind that it will be a little sparse until we use more of the features.

When you first enter the Objection REPL, you will be in the Bundle path. This contains the application binary, Info.plist (the app's manifest file), and all of the app's various resources. There are three other locations where the application may store data at runtime: Library, Caches, and Documents.

To find these paths, type **`env`** at the prompt. Copy one of the paths and type **`cd [path]`**. You can use **`ls`** to show the current directory's contents (sorry, it doesn't take arguments).

To print the contents of a file, use **`file cat [name]`**. To download files, use **`file download [name]`**. Helper commands are available for some common formats like plists and SQLite DBs (we'll get to these later). You can also execute commands on your laptop with **`!command`** – this is handy if you want to download a file and then run some local tools on it, without leaving the REPL.

Exercise 3: SQ~~Loot~~Lite

SQLite databases are found in great abundance in mobile applications. If you've followed a "build your first mobile app" tutorial, you've undoubtedly created one yourself. Common examples include:

- WebView caches, including cookies
- Cached web service responses (some request frameworks offer to cache these for you)
- Transient domain-specific data for the application, such as a user profile or a list of transactions
- Data managed by the Core Data framework (Apple's version of an ORM)
- Sketchy credential stores

SQLite databases exist as a trio of files - the base file, plus [name]-wal and [name]-shm. You may come across other variations on this pattern, depending on the DB's operating mode.

These databases can be found all over the app sandbox. Directly managed DBs will tend to be in `Documents`, while framework-managed DBs are often in the `Library` path. Cached data tends to be located in… `Caches`.

As with the last exercise, use Objection's `env` command to get the absolute path for each, then `cd` to that path and rummage around.

In the **DVIA-v2** app, select **Local Data Storage**, then **YapDatabase**. Enter some bogus credentials and tap **Save in YapDatabase**.

Open the Objection REPL and navigate to the Library path. You will see an `Application Support` directory, and within that a `YapDatabase.sqlite` file. Open it with the command: `sqlite connect YapDatabase.sqlite`

At the SQLite prompt, you can dump the schema with .schema, list the tables with .tables, or execute normal SQL statements. Let's look at the `database2` table:

```
select * from database2
```

We see rows labeled `username` and `password`, but they hold `BLOB` data! Copy the username `BLOB` hex data (everything after `0x`), then open **Burp Suite**. Go to the **Decoder** tab and paste the data into the first box. Select the **Decode as…** dropdown and choose **ASCII Hex**. There are a few strings we can pick out right away, including the value of the username. Repeat with the password `BLOB`.

To close the file and return to the Objection REPL, type `exit`

Bonus: Any idea what this binary format is? `bplist` is the big clue. See if you can build something to decode and pretty-print this data. You'll find this type of data in many iOS databases, so it will come in handy.

Exercise 4: Dumping the KeyChain

The KeyChain sometimes gets a little *more* credit than it deserves. To be fair, of all the options available, it is the most appropriate place to store sensitive information like credentials. This data is not particularly difficult to retrieve, though. Think about how to reduce the risks associated with storing sensitive data. For example:

- Store properly-scoped access tokens rather than account passwords
- If users must log in when they open the app, consider using a key derivation technique (like PBKDF2) and only storing sensitive data in encrypted form

Caveat to the above rant: Entries which require TouchID/FaceID are relatively magical and I will throw no further shade upon them. If you attempt to dump one of these entries, you'll be greeted with a biometric challenge.

Activity: Steal some KeyChain secrets

> Open the DVIA-v2 app, select **Local Data Storage**, then select **KeyChain**. Enter a super secret string, and tap **Save in KeyChain**.
>
> Now open the Objection REPL and enter the command: `ios keychain dump`
>
> You will see some rather benign entries (such as analytics tokens), along with a new entry containing your super secret string.

Android Sidenote: Many Android applications treat SharedPrefs like a KeyChain equivalent because the basic key/value pattern is similar. But SharedPrefs are stored in plain old XML files. A closer analog would store encrypted data in SharedPrefs, and an encryption key in the Android KeyStore. It's a bit of complexity that is often skipped or overlooked. In short, you'll find good loot in SharedPrefs.

Exercise 5: Misc Data Stores

Activity: Find sensitive data in a plist file

We looked at the application bundle's `Info.plist` file earlier, but these are used for more than just app manifests.

In the **DVIA-v2** app, enter some bogus credentials in the **Local Data Storage** -> **Plist** screen. Can you find it on the filesystem?

Remember that Objection has a built-in plist pretty-printer: `ios plist cat [file]`

Hint: Most intentionally-created files live under the Documents path.

Activity: Examine UserDefaults

UserDefaults is a preference storage system built into iOS. We'd expect to find a lot of boring app settings there, but occasionally it's misused to store sensitive information.

This time, go to the **Local Data Storage** -> **UserDefaults** screen, and enter something memorable.

This is an easy one, because you don't even need to search for a file – use the Objection command:

```
ios nsuserdefaults get
```

Exercise 6: Biometric Authentication

Corellium devices do not support TouchID or FaceID. Time permitting, we will see a demo.

On a real device, the more naïve callback-based implementation can be bypassed using the objection command: `ios ui biometrics_bypass`

A more secure implementation involves flagging a KeyChain entry as requiring biometric authentication. It may be locked to the current finger/face set (as of the entry's creation), or to any matching finger/face (including those which are added later). When that entry is accessed, a biometric prompt is triggered. Hand-wavy crypto-math and hardware enclaves are involved, so hooking method calls won't do you much good.

Android supports a similar pair of implementations, but it was added more recently and is more difficult to use securely. Hence, it is common to find cross-platform modules that implement biometric authentication more securely on iOS than on Android (assuming they get it right at all).

Honorable Mention: Logging

Unfortunately, the device logging data leakage exercise in DVIA-v2 crashes! Be aware that you're likely to find interesting information in the console logs, though. You can access these from **XCode** under **Window** -> **Devices and Simulators** -> [Device] -> **Open Console**.

You can find all sorts of things in leftover debug logging: full request/response data, credentials, personal data, encryption keys, access tokens, … you get the idea. Keep an eye out for sketchy logging statements during code review!

Epilogue

I hope this gives everyone a good place to start, and I'll end with some brief thoughts on a couple of paths forward.

For development teams: Look into the extension and scripting capabilities of Objection and Frida, and think about how you can add some security tests to your current automated test suite.

For security folks: To dive deeper, you'll really want to use jailbroken devices. This has traditionally been painful, but the checkm8 bootrom vulnerability has changed things a bit. Also, get a Mac if you don't have one – decent used MacBook Pros are relatively affordable.

Appendix A – Pre-Requisites

Since everyone totally installed this stuff ahead of time, let's just say this is for reference. The USB sticks also contain some of this "reference material".

XCode: App Store, USB, or
https://download.developer.apple.com/Developer_Tools/Xcode_11.3/Xcode_11.3.xip

XCode Command Line Tools: App Store, USB, or
https://download.developer.apple.com/Developer_Tools/Command_Line_Tools_for_Xcode_11.3/Command_Line_Tools_for_Xcode_11.3.dmg

Install the following with Homebrew (https://brew.sh/) or another package manager:

- python (3.x)
- libusbmuxd
- libimobiledevice
- libideviceinstaller
- ios-deploy

Install some Python packages:
pip install frida frida-tools objection

If pip throws an SSL error on macOS, try running `/Applications/Python [version]/Install Certificates.command`
Reference: https://stackoverflow.com/a/53310545 (thanks, Scott!)

Burp Suite Community Edition: USB, or https://portswigger.net/burp/communitydownload

OpenVPN Connect Client: USB, or
https://openvpn.net/downloads/openvpn-connect-v2-windows.msi
https://openvpn.net/downloads/openvpn-connect-v2-macos.dmg

To use remote virtual interfaces to sniff network traffic on an iOS device, some extra steps are required starting with XCode 11: https://www.agnosticdev.com/content/how-install-rvictl-xcode-11 (or maybe not? I found `rvictl` in a location that simply wasn't on my $PATH - `/Library/Apple/usr/bin/rvictl`)

Appendix B – Patching Applications

In **XCode**, go to **Preferences** -> **Accounts**. Add you Apple ID and create an **Apple Development** certificate.

In order to generate a provisioning file for the code signing process, you will need to build and run an iOS project on the attached device. A blank application works fine for this.

If you need to use a custom Frida configuration, prepare a directory tree consisting of `Payload/AppName.app/Frameworks/`, and place a `FridaGadget.config` file within. Next, merge the config file into the IPA with:

```
zip -r [your-app].ipa Payload
```

Find your code signing signature by running:

```
security find-identity -p codesigning -v
```

Copy the hexadecimal signature for your Apple Development identity, and run:

```
objection patchipa -s [your-app].ipa -c hex-value
```

You should see a new file named `[your-app]-frida-codesigned.ipa`. Sideload the application using `ios-deploy`, Cydia Impactor, or whatever else fits into your workflow.

Appendix C - Resources and Further Reading

Frida Docs: https://frida.re/docs/home/
Frida Codeshare: https://codeshare.frida.re/
Objection Wiki (also has good Frida info): https://github.com/sensepost/objection/wiki

OWASP Mobile Security Testing Guide - Excellent resource for mobile appsec testers. You will see a lot of other tools used here, and the assumption is that you will be using a jailbroken device.
https://mobile-security.gitbook.io/mobile-security-testing-guide/

OWASP Mobile Application Security Verification Standard (MASVS) - Of more interest to those in traditional "delivery" roles: analysts, testers, and developers. Use this to guide your security requirements.
https://mobile-security.gitbook.io/masvs/

Other Tools:

> Passionfruit: Web UI that would support most of what we did today. YMMV in getting it to work.
> https://github.com/chaitin/passionfruit

Stuff that I made or borrowed for this pre-compiler:

> This document, odds & ends: https://github.com/hweisheimer/ios-testing
> Fork of Prateek Gianchandani's DVIA-v2 app: https://github.com/hweisheimer/DVIA-v2

Good talks on Frida and Objection:

> Leon Jacobs - Meticulously Modern Mobile Manipulations
> DEFCON 27
> https://www.youtube.com/watch?v=7LKXSYFrYAM
>
> Kev Cody - How to Frida Good
> SnowFROC 2019, mDevCamp 2019
> https://slideslive.com/38916544/how-to-frida-good?locale=en

Wondering what devices to buy for testing? Micro Center is selling new iPhone SEs for $129, and these will run iOS 13. An iPhone 7 can usually be had for $300 or less. Anything up to the 8 and (original) X can be jailbroken with checkm8/checkra1n, regardless of iOS version.