575.4

Dynamic Mobile Application Analysis and Manipulation

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

With the CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by SANS Institute to User for use in the SANS class associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCEPTING THIS COURSEWARE, YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, YOU AGREE THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND) SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If you do not agree, you may return the Courseware to SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

SANS acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There’s an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP and PMBOK are registered marks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.

SANS

Dynamic Mobile Application Analysis and Manipulation

Welcome to Day 4 of Mobile Device Security and Ethical Hacking! Today we will be performing dynamic mobile application analysis and manipulation.

TABLE OF CONTENTS

This page intentionally left blank.

Course Roadmap

- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 4

Dynamic Mobile Application Analysis and Manipulation
Android Dynamic Analysis with Drozer
Exercise: Manipulating Android Intents
iOS Dynamic Analysis with Needle
Modifying Mobile Applications
Exercise: Modifying Android Applications
Mobile Application Runtime Manipulation
Automated Runtime Manipulation with Objection
Exercise: Frida and Objection
Application Security Verification

This page intentionally left blank.

DYNAMIC MOBILE APPLICATION ANALYSIS

DE PT
AA MW

Static evaluation is valuable, but can become very complex

- Hard to follow code paths
- Application may load dynamic code
- Application may require server interaction

Dynamic analysis can solve these issues

- Monitor application behavior by executing it
- Inspect local storage and platform interaction
- Modify the behavior of the application at runtime

Dynamic Mobile Application Analysis

Static analysis is a very important step in analyzing a mobile application, but static analysis alone can be really slow and complex. On Android, it can be very difficult to follow the flow of the code throughout the app, keep track of the values of variables, or figure out what exactly is being stored in the application container. On iOS, static analysis is even more difficult, as decompilation will only generate low-level code that is very difficult to follow.

Dynamic analysis is the practice of running the application on a device or emulator and monitoring its behavior. Many different things can be monitored while the application is running. It is possible to monitor if specific methods are called, if platform features such as the keychain or biometrics are used, which data is stored or shared with other applications, etc. With the right tools, it is also possible to modify the behavior of the application at runtime.

Course Roadmap

- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 4

Dynamic Mobile Application Analysis and Manipulation
Android Dynamic Analysis with Drozer
Exercise: Manipulating Android Intents
iOS Dynamic Analysis with Needle
Modifying Mobile Applications
Exercise: Modifying Android Applications
Mobile Application Runtime Manipulation
Automated Runtime Manipulation with Objection
Exercise: Frida and Objection
Application Security Verification

This page intentionally left blank.

DROZER FRAMEWORK

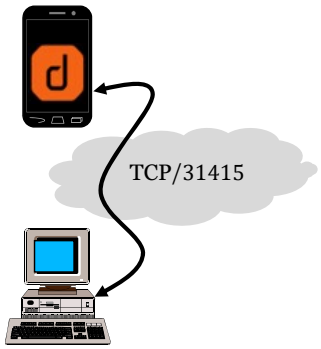
Client/server framework for evaluating and exploiting Android applications

- Formerly, Mercury Framework
- Runs in emulator or physical device

Framework like Metasploit includes analysis tasks and exploit modules

Opportunity to explore Android IPC mechanisms

- Identifying and crafting Intents



Drozer Framework

Drozer is a client/server framework for evaluating Android applications consisting of an Android application agent and a Windows or Linux console component. Drozer is extensible and includes both assessment and exploitation modules, making it similar to the Metasploit Framework project for Android devices.

Using Drozer, we can explore Android IPC mechanisms, enumerate application components, and craft custom Intent messages to targeted services, without having to write Java code. In this way, Drozer is a proof-of-concept framework (unlike Metasploit) because it does not resemble the malware that would normally exploit weaknesses in applications. For use in application assessment, however, Drozer is a valuable tool that allows us to quickly and easily experiment with Android applications.

Drozer is maintained by a team of developers at MWR InfoSecurity (@mwrlabs) and a community of contributors. Drozer is available at https://labs.mwrinfosecurity.com/tools/drozer/.

ANDROID COMPONENT EXPOSURE

Other applications can invoke and manipulate components

Developers need to protect access to sensitive components

- Not all components are sensitive

We can explore and demonstrate vulnerabilities with Drozer

- Practical exploitation would come from malware on the Android device

Android Component Exposure

Using Drozer, we can identify the exposed Android components, invoking them from the Drozer console to identify the threat of exposed components to other applications on the same platform. Note that not all Android components are sensitive; many Android applications include activities, receivers, and services for normal application functionality. Using Drozer, we can enumerate and test these components to identify exposed sensitive application functions.

Although we can explore and demonstrate Android vulnerabilities with Drozer, it is not designed to be an exploitation tool. Drozer requires the Drozer Agent to interact with the device, allowing us to explore applications without writing custom code. In this fashion, Drozer is more closely a proof-of-concept vulnerability assessment tool, demonstrating the threats that would otherwise be exposed to malicious Android applications running on the same device.

DROZER AGENT

Install, then start the agent. Click the OFF button to start the agent listener on TCP/31415.

Drozer Agent

Included with the Drozer software, the Drozer Agent ("agent.apk") first needs to be installed in the physical or emulated Android device. After installing the Drozer Agent, start the application and click the button marked OFF to start the agent listener on TCP/31415.

DROZER CONSOLE: STARTUP AND ENUMERATION

```
C:\>adb forward tcp:31415 tcp:31415
C:\>cd \drozer
C:\drozer>drozer.bat console connect
...
Selecting defd70b2666755d8 (unknown sdk 4.0.4)

drozer Console (v2.3.1)
dz> run app.package.list
...
com.mwr.dz (drozer Agent)
org.owasp.goatdroid.fourgoats (FourGoats)
dz> run app.package.attacksurface org.owasp.goatdroid.fourgoats
Attack Surface:
  4 activities exported
  1 broadcast receivers exported
  0 content providers exported
  1 services exported
    is debuggablez
```

Identify installed applications.

Attacksurface discloses activities, receivers, and services that are accessible for a specified application. We can evaluate each to discover the names of the exported components.

Drozer Console: Startup and Enumeration

Next, we step through the analysis of a target Android application, FourGoats. Written as an OWASP project for understanding and exploiting Android vulnerabilities by Jack Mannino, FourGoats is a social networking application included with the GoatDroid project. GoatDroid and FourGoats are available at https://github.com/jackMannino/OWASP-GoatDroid-Project.

After the Drozer Agent is running on an Android virtual device (AVD), we need to add a port forward from the local host to the Android device. This can be done by using the "adb forward" command, as shown on this page.

Next, invoke the "drozer.bat" script from a command prompt in the C:\drozer directory, adding the parameters "console connect", as shown. The Drozer Console connects to the local TCP/31415 port, connecting to the AVD.

From the "dz>" console prompt, we can interact with the Drozer Agent to enumerate and exploit the Android device and installed applications. Issuing the command "run app.package.list" identifies all the installed applications, as shown on this slide.

Another Drozer function is the app.package.attacksurface module. Using this module, we can quickly identify the exposed activities, receivers, content providers, and services on the specified application, as shown.

DROZER CONSOLE: ANALYSIS

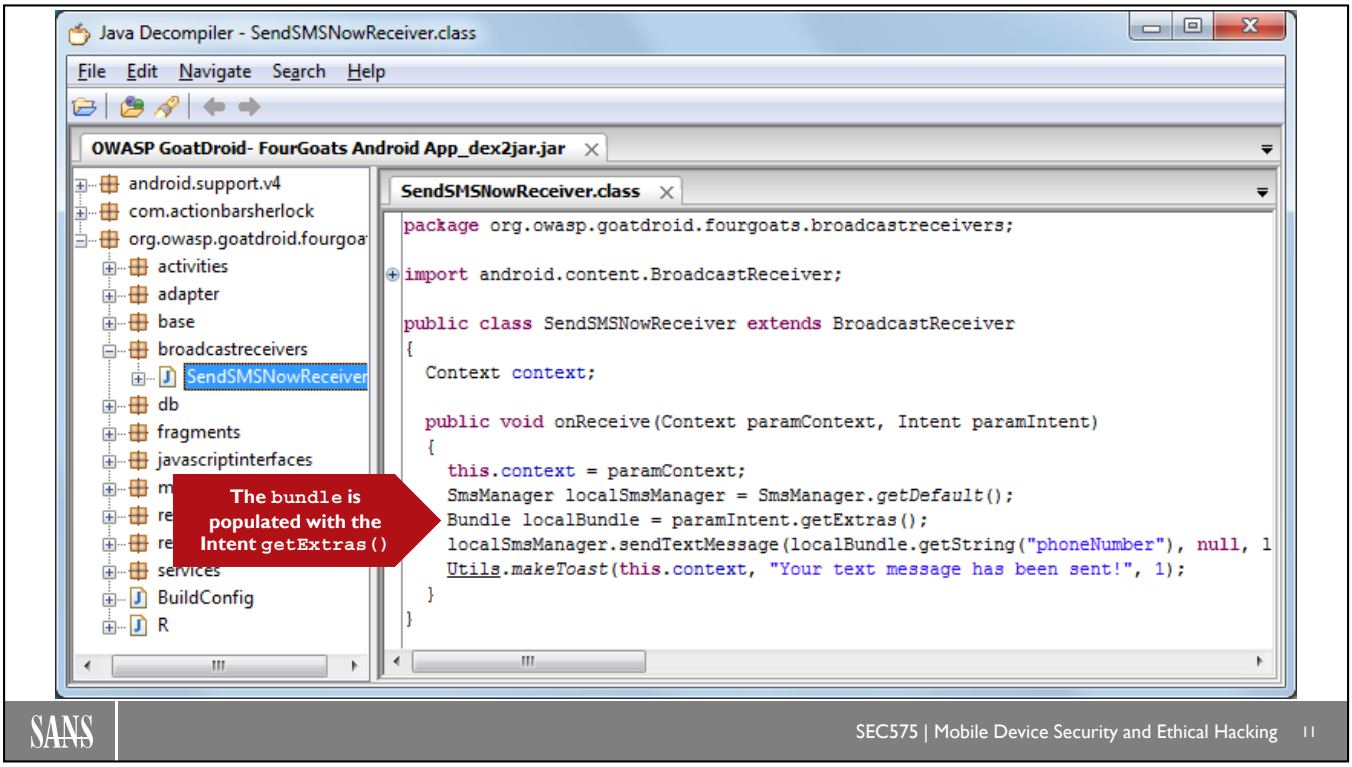
```
dz> run app.activity.info -a org.owasp.goatdroid.fourgoats
Package: org.owasp.goatdroid.fourgoats
  org.owasp.goatdroid.fourgoats.activities.Main
  org.owasp.goatdroid.fourgoats.activities.ViewCheckin
  org.owasp.goatdroid.fourgoats.activities.ViewProfile
  org.owasp.goatdroid.fourgoats.activities.SocialAPIAuthentication

dz> run app.service.info -a org.owasp.goatdroid.fourgoats
Package: org.owasp.goatdroid.fourgoats
  org.owasp.goatdroid.fourgoats.services.LocationService
    Permission: null

dz> run app.broadcast.info -a org.owasp.goatdroid.fourgoats
Package: org.owasp.goatdroid.fourgoats
  Receiver: org.owasp.goatdroid.fourgoats.broadcastreceivers.SendSMSNowReceiver
```

After identifying the exposed components, we can send intents to each to explore the application behavior. First, let's examine the associated sources to identify extra data sent with each component.

Drozer Console: Analysis

Targeting the org.owasp.goatdroid.fourgoats application, we can enumerate the names of the exposed activities, services, and receivers identified by the AttackSurface module.

Drozer includes three modules (app.activity.info, app.service.info, and app.broadcast.info) to explore Android components to enumerate activities, services, and receivers, respectively. In the results on this page, we see four activities, one service, and one receiver. Let's investigate the SendSMSNowReceiver function.



Component Extras

Although Drozer can identify the components associated with an Android application, it cannot enumerate any extra parameters that are processed with Android Intent messages. To identify the extra parameters for Intents, we need to disassemble the application and inspect the Java source, searching for the string "getExtras". For the org.owasp.goatdroid.fourgoats.SendSMSNowReceiver, we can navigate the application tree and identify the onReceive method, as shown on this slide.

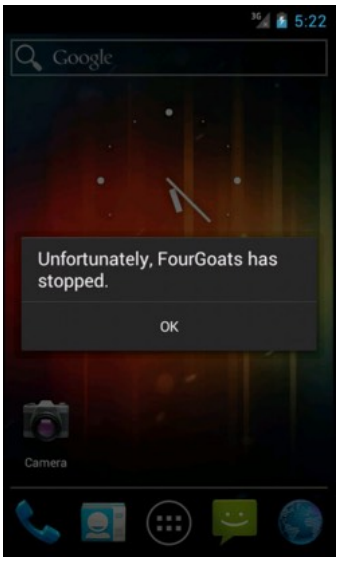
In the onReceive method, we see a Java Bundle declared as the return from paramIntent.getExtras(). The Java Bundle is a data type that stores associated data (similar to a Perl Hash or a Python Dictionary) and is typically used to retrieve extra parameters from an Intent.

Immediately following the getExtras() call, we see that an SMS message is sent using sendTextMessage, using the Java Bundle localBundle, and retrieving a string object called "phoneNumber". From this output, we see that an extra is associated with the SendSMSNowReceiver receiver, a string object known as "phoneNumber", referring to the phone number as the recipient of an SMS message.

DROZER CONSOLE: START EMPTY INTENT

```
dz> run app.broadcast.info -a
org.owasp.goatdroid.fourgoats
Package: org.owasp.goatdroid.fourgoats
  Receiver:
org.owasp.goatdroid.fourgoats.broadcastreceivers.Sen
dSMSNowReceiver
dz> run app.broadcast.send --component
org.owasp.goatdroid.fourgoats
org.owasp.goatdroid.fourgoats.broadcastreceivers.Sen
dSMSNowReceiver
```

Sending an empty Intent to SendSMSNowReceiver causes the app to crash. Valuable DoS finding, but not terribly interesting.



Drozer Console: Start Empty Intent

At this point, we know that the FourGoats application has a receiver component that we can invoke from other applications. Returning to the Drozer console, we can use the app.broadcast.send method to invoke the org.owasp.goatdroid.fourgoats application (using "--component"), invoking the SendSMSNowReceiver receiver. Upon sending this message to FourGoats, the application crashes. This is a potentially valuable DoS finding, but it is not terribly interesting. We can continue to explore this receiver component to identify additional vulnerabilities associated with the component.

DROZER CONSOLE: POPULATED INTENT

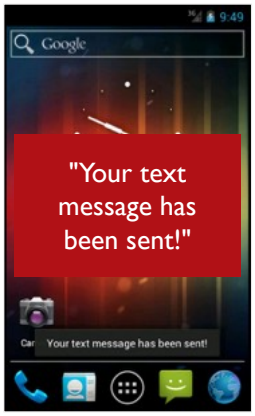
Reverse engineered source code excerpt

```
Bundle localBundle = paramIntent.getExtras();

localSmsManager.sendTextMessage(localBundle.getString("phoneNumber")
, null, localBundle.getString("message"), null, null);
```

```
--extra arguments: type parameter_name parameter_value
```

```
dz> run app.broadcast.send --component org.owasp.goatdroid.fourgoats
org.owasp.goatdroid.fourgoats.broadcastreceivers.SendSMSNowReceiver
--extra string phoneNumber 4015242911 --extra string message "Please
call me!"
```

```
C:\>adb logcat -t 2
D/SmsStorageMonitor(  180): SMS send size=0 time=1360680329412
D/gralloc_goldfish(  694): Emulator without GPU emulation detected.
```



Drozer Console: Populated Intent

Recall that when we reverse engineered the Android application, we saw that the SendSMSNowReceiver Receiver populated a Java Bundle localBundle with the return from the getExtras() function. Next, localBundle was referenced to retrieve a string object ("getString") with the object name "phoneNumber", as well as a second string with the object name "message", shown on the top of this page.

These Extras can be sent with an Android Intent using Drozer. The Drozer modules app.broadcast.send, app.service.send, and app.activity.send all accept an argument "--extra" that requires three parameters:

- Parameter type
- Parameter name
- Parameter value

Running the app.broadcast.send module again, we can add the "--extra string phoneNumber 4015242911" and "--extra string message 'Please call me!'" arguments, sending the custom Intent to the SendSMSNowReceiver Receiver component. This time, instead of crashing, FourGoats displays a message on the screen, "Your text message has been sent!" We can further confirm that the Intent triggered the delivery of an SMS message by monitoring the Android device logs with the "adb logcat" command, as shown.

Course Roadmap

- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 4

Dynamic Mobile Application Analysis and Manipulation
Android Dynamic Analysis with Drozer
Exercise: Manipulating Android Intents
iOS Dynamic Analysis with Needle
Modifying Mobile Applications
Exercise: Modifying Android Applications
Mobile Application Runtime Manipulation
Automated Runtime Manipulation with Objection
Exercise: Frida and Objection
Application Security Verification

This page intentionally left blank.

EXERCISE: MANIPULATING ANDROID INTENTS

Please log in to the SANS lab system for the exercise
This exercise takes approximately 40 minutes

Exercise: Manipulating Android Intents

Please log in to the SANS lab system for the Manipulating Android Intents exercise. This exercise takes approximately 40 minutes to complete.

Course Roadmap

- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 4

Dynamic Mobile Application Analysis and Manipulation
Android Dynamic Analysis with Drozer
Exercise: Manipulating Android Intents
iOS Dynamic Analysis with Needle
Modifying Mobile Applications
Exercise: Modifying Android Applications
Mobile Application Runtime Manipulation
Automated Runtime Manipulation with Objection
Exercise: Frida and Objection
Application Security Verification

This page intentionally left blank.

NEEDLE

Security testing framework for iOS by Marco Lancini/MWR
Requires jailbroken device
Runs on Linux/macOS and connects over SSH to target device
Metasploit-like console with modules

- Binary, Comms, Dynamic, Hooking, Static, Storage

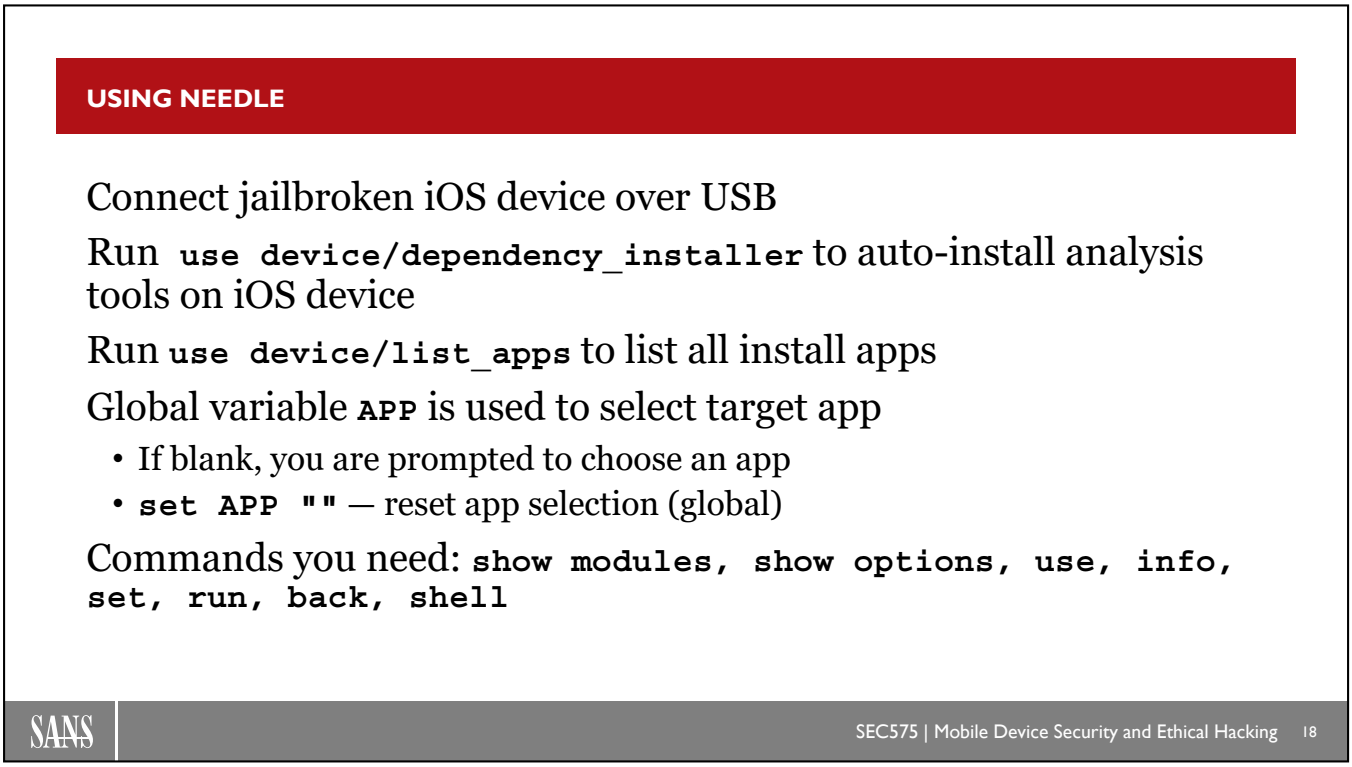
```
$ python needle.py
                __   _ _______ _______ ______      _______
               | \  | |______ |______ |     \ |      |______
               |  \_| |______ |______ |_____/ |_____ |______

  [MWR InfoSecurity (@MWRLabs) - Marco Lancini (@LanciniMarco)]

[needle] >
```


Needle

Needle is a modular iOS app assessment framework by Marco Lancini, distributed by MWR Labs. It runs on a macOS or Linux system, interacting with a jailbroken device over SSH. Using a Metasploit-like console and module concept, Needle allows analysts to quickly gather information about one or more target iOS apps.

The fundamental functionality of Needle isn't tremendously valuable, but the features accommodated with its modules are very useful for iOS app assessment. Needle modules are classified as binary (static executable flags), comms (working with network activity and related tasks such as certificate management), dynamic (working with apps while they're running), hooking (manipulating apps), static (static analysis), storage (data on the filesystem), and various (miscellaneous) functionality.

The setup instructions for Needle are well documented on the Needle wiki at https://github.com/mwrlabs/needle/wiki.



Using Needle

After completing the installation requirements, connect a jailbroken iOS device to your macOS or Linux system over USB. Needle will connect to a jailbroken device over SSH using the USB interface as the preferred configuration option, though it can also connect to an iOS device specified with a target IP address.

When working with a new jailbroken device and Needle for the first time, execute the `"device/dependency_installer" module`. This tells Needle to install the required functionality on the iOS device as needed. After using Needle for a while, you won't have to set this option, but it makes it easier to get started with Needle in the beginning.

Needle uses the global configuration variable `APP` to identify which app it is evaluating. By default, this value is blank, which will cause Needle to prompt you to choose an application when you run a module. Needle remembers the target app for all subsequent use until you exit and restart Needle or until you set the `APP` variable to an empty string (e.g., `""`), at which point Needle will prompt you to choose a new app target. Make sure the Needle iOS app is in the foreground to enumerate the list of available apps on the system.

Needle has several commands available, but the core commands you should understand are defined here:

- `show modules`: List the modules available to Needle for app assessment use
- `show options`: List the options available in the currently selected (*used*) module
- `use`: Load a specified module by name; press Tab twice to autocomplete from the list of available modules matching a partial module name
- `info`: Gather help and usage information for a module
- `set`: Control global or module-specific settings
- `run`: Run the loaded module
- `back`: Exit the module, returning to global configuration mode
- `shell`: Open an SSH shell on the iOS device (useful when you need to identify a path or other parameter on the iOS device without opening a new SSH session)

Next, we'll look at some of the more useful module options available in Needle and some examples of how they can be used for app analysis.

NEEDLE BINARY MODULES

Retrieves static information from installed iOS apps

- Needle often runs a local iOS binary and returns output to console

`binary/class_dump`	Decrypts binary, returns header information
`binary/compilation_checks`	Reports on developer options to secure binary
`binary/install`	Installs an IPA file
`binary/metadata`	Obtains application metadata, including URI handlers
`binary/pull_ipa`	Downloads an IPA file
`binary/shared_libraries`	Identifies the shared libraries for an app
`binary/strings`	Runs strings on a binary
`binary/universal_links`	Identifies all links associated with an app

Needle Binary Modules

Needle's binary module classification represents functionality to gather information about applications and the app executables, as well as basic analysis and system configuration tasks. Of the modules shown on this page, the `class_dump`, `compilation_checks`, and `metadata` modules are the most useful, since they represent functionality that cannot be easily captured with a single tool on the iOS device.

NEEDLE METADATA MODULE

```
[needle] > set APP com.facebook.messenger
[needle] > use binary/metadata
[needle][metadata] > run
[+] Target app: com.facebook.Messenger
[*] Retrieving app's metadata...
[+] Name                : Messenger.app
[+] App Version         : 48148560 (102.0)
[+] Data Directory      : /private/var/mobile/Containers/Data/Application/6B2B56AD-...-B29C-
499DA46AF0DF
[+] Architectures       : armv7, arm64
[+] Platform Version    : 10.2
[+] SDK Version         : iphoneos10.2
[+] Minimum OS          : 8.0
[+] URL Handlers
[+]               fb-messenger-api20131028
[+]               fb-messenger-public
[+]               fb-messenger-neue
[+]               fb-events-share
[+] Apple Transport Security Settings
[+]               NSAllowsArbitraryLoads                : 1
[*] No Application Extensions found
```

Tells us that Facebook Messenger allows for unencrypted HTTP traffic

Needle `metadata` Module

In the example on this page, I used the Needle `metadata` module to evaluate the Facebook Messenger application (note that output from this command has been modified for space). Here we see that the application version, the data directory, executable architectures, and URL handlers are all revealed. Most interesting is that the Facebook Messenger application has `NSAllowsArbitraryLoads` set to true (1), which indicates that the app accommodates the use of unencrypted HTTP traffic (this is not a default setting for iOS apps).

NEEDLE DYNAMIC MODULES

Retrieves dynamic information from running iOS apps
Slow to execute; on-console alternatives are often faster

`dynamic/detection/jailbreak_detection`	Reports if the app does jailbreak detection
`dynamic/ipc/open_uri`	Open a specified URI handler (just use Safari)
`dynamic/memory/heap_dump`	Searches memory for a specified string
`dynamic/monitor/files`	Monitor file open/read/write activity
`dynamic/monitor/pasteboard`	Monitor activity to the system pasteboard (clipboard)
`dynamic/monitor/syslog`	Monitor syslog activity (recording to a file)
`dynamic/watch/syslog`	Monitor syslog activity (in real time)

Needle Dynamic Modules

Where Needle's static modules collected information from a target app without running the app, the dynamic modules all rely on starting the application before collecting information. The need to run the app under special conditions for analysis makes these modules slower than static analysis, and often slower than the analysis applications that Needle runs on the iOS device itself.

Some of the dynamic modules are not very useful, such as `open_uri`, which opens a URL (you could just type the URL into Safari), or are more cumbersome than the companion command line alternatives (monitoring the Apple System Log is easier done with *ondeviceconsole* than Needle). However, the `heap_dump` module is particularly useful for searching for and identifying matching strings from the target application heap (allocated memory).

The ondeviceconsole command is available in the Cydia app store.

NEEDLE HEAP_DUMP MODULE

```
[needle] > use dynamic/memory/heap_dump
[needle][heap_dump] > set FILTER key
[needle][heap_dump] > run
[+] Target app: com.facebook.Messenger
[V] PID found: 33647
[*] Enumerating mach regions...
[*] Dumping memory (it might take a while)...
[V] Checking if we have dumps...
[*] Extracting strings...
       SELECT key, size, modification_time, access_time, tag, extra FROM
DiskCacheManifest;
       UPDATE DiskCacheManifest SET access_time = ?2 WHERE key = ?1;
       SELECT key, size, modification_time, access_time, tag, extra FROM
DiskCacheManifest WHERE key = ?1;
       PLOrderKeyObject
       TIKeyboardCandidateCoding
       I4TIKeyboardInputManager
       <key>back_camera_s</key>
       @@"TIKeyboardCandidate"
```

Specify any case-insensitive string as the FILTER parameter.

Needle `heap_dump` Module

In the example on this slide, I used the `heap_dump` module to search Facebook Messenger's allocated memory for the string *key*. The module uses the `gdb` and `strings` commands to retrieve the memory and search it for the target string but does so in such a way that would be cumbersome to do manually. Using Needle here is a big time-saver.

NEEDLE STORAGE MODULES

Automates data collection using local iOS tools

Convenient but not superior to using the individual tools themselves

`storage/caching/keyboard_autocomplete`	Dumps the contents of the **`dynamic-text.dat`** file
`storage/caching/screenshot`	Prompts user to background app, retrieves screenshot (useful for validating anti-snapshot code)
`storage/data/files_binarycookies`	Dumps the contents of WebKit cookie storage files
`storage/data/files_cached`	Dumps the contents of WebKit history files
`storage/data/files_plist`	Dumps the contents of plist files
`storage/data/files_sql`	Dumps the contents of SQLite database files
`storage/data/keychain_dump`	Dumps the contents of the keychain

Needle Storage Modules

Needle's storage modules automate the process of collecting data from an iOS device. The big advantage of using Needle here is that you can use a single interface to access data from several different file types that would otherwise require several different analysis tools. Instead of using Plist Editor for Windows, SQLiteSpy, Keychain_dumper, strings, and BinaryCookieReader, Needle uses the correct tool for the corresponding file format.

NEEDLE FILES_BINARYCOOKIES MODULE

```
[needle] > use storage/data/files_binarycookies
[needle][files_binarycookies] > run
[*] Looking for Binary Cookies files...
[*] The following Binary Cookies files have been found:
       0   - [NSFileProtectionCompleteUntilFirstUserAuthentication]
/private/var/mobile/Containers/Data/Application/6B2B56AD-B8DC-4F2A-B29C-
499DA46AF0DF/Library/Cookies/Cookies.binarycookies
[>][QUESTION] Please select a number: 0
[*] Pulling: /private/var/mobile/Containers/Data/Application/6B2B56AD-B8DC-4F2A-B29C-
499DA46AF0DF/Library/Cookies/Cookies.binarycookies ->
/Users/jwright/.needle/output/BinaryCookies_datadir_Library_Cookies_Cookies.binarycookies
Cookie : c_user=100003349494426; domain=.facebook.com; path=/; expires=Fri, 16 Feb 2018
Cookie : csm=2; domain=.facebook.com; path=/; expires=Fri, 16 Feb 2018;
Cookie : datr=rSlW6dCshgW2-FB0aY; domain=.facebook.com; path=/; expires=Sat, 16 Feb 2019;
Cookie : fr=0CvfalQDPAviLylY1.AWWkUG3QcGT2rCs.BYpbSe.aJ.AAA.0.0.BYpbSe.AWVUeYay;
domain=.facebook.com; path=/; expires=Fri, 16 Feb 2018;
Cookie : xs=202:EDjpP0LMZbg:2:1487254686:15392; domain=.facebook.com; path=/;
expires=Fri, 16 Feb 2018; Secure
```


Needle `files_binarycookies` Module

In the example on this page, I use the `files_binarycookies` module to read from the Facebook Messenger `Cookies.binarycookies` file. This file is used by any application using the WebKit framework to record cookie content, revealing authenticated cookie information that could be used in a sidejacking attack.

NEEDLE POTENTIAL

Needle is mostly a wrapper around command line tools on iOS

- Modules abstract you from remembering the command line details
- Also, abstracts you from getting detail

Needle saves you time

- Optionally create a resource file to run all your favorite modules for each app you want to evaluate
- You can always use the command line tools if you want

Needle is extensible and straightforward

- If the community backs Needle, like the Metasploit Framework, lots of future potential

Needle Potential

Needle is a useful tool, leveraging the functionality of several different iOS command line tools in one consistent interface. Needle helps you collect information about a target application without working with dozens of different command line tools. Unfortunately, this also abstracts you from getting some of the detail individual tools return.

Using Needle, you can accelerate your analysis of iOS apps. If needed, you can refer to individual tools for additional information, but Needle is a good place to start your analysis. Needle also includes the ability to specify a *resource* file, representing a list of commands to run when you start Needle, making it possible to script up your standard analysis tasks.

The most exciting part about Needle is the extensible and straightforward framework it represents. If a large community of users and developers uses Needle and enhances its functionality (similar to the support seen by the Metasploit Framework), then Needle has a lot of future potential.

Course Roadmap

- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 4

Dynamic Mobile Application Analysis and Manipulation
Android Dynamic Analysis with Drozer
Exercise: Manipulating Android Intents
iOS Dynamic Analysis with Needle
Modifying Mobile Applications
Exercise: Modifying Android Applications
Mobile Application Runtime Manipulation
Automated Runtime Manipulation with Objection
Exercise: Frida and Objection
Application Security Verification

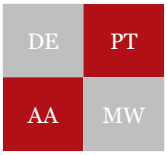
This page intentionally left blank.

MODIFYING MOBILE APPLICATIONS

Next to monitoring an app, we can manipulate its behavior

App manipulation can help to analyze an app

- Force an app to use HTTP, not HTTPS
- Disable emulator detection
- Run on jailbroken device



Can evade in-app purchase systems or change app functionality

- Illegitimately obtain resources in games without paying or cheating validation systems

Use your skills for good, not evil.

Modifying Mobile Applications

When we evaluate mobile device applications, it is sometimes necessary to modify the behavior of the application. This can be extremely useful as part of our analysis process and simplify the task at hand. For example, through app modification, we could force applications to use HTTP instead of HTTPS for all network traffic, or disable Android emulator detection, or get an iOS app to run on a jailbroken device. Of course, these same techniques could be used to evade in-app purchase systems or to gain access to pay for application functionality illegitimately.

In this module, we look at the techniques behind modifying Android applications as part of the application analysis process. Remember to use your skills for good, not evil.

IOS APP MANIPULATION

iOS apps are compiled in ARM

Modifying iOS applications requires patching the executable with new assembly instructions

- Requires advanced reverse engineering skills
- Error-prone
- Very time consuming

Application needs to be resigned correctly to run on non-jailbroken devices

iOS App Manipulation

iOS applications are compiled to ARM and run directly on the device. There is no intermediate language, as there is on Android. This makes it very difficult to reverse engineer the functionality of an iOS application. Without ARM knowledge and a good reverse engineering mindset, finding the correct location for a piece of functionality, such as jailbreak detection, will be very difficult, if not impossible. Once you have identified the functionality that you want to modify, you will also need to patch the binary using ARM code. Finally, the application needs to be correctly resigned to be run on non-jailbroken devices, which is also a tricky process.

Patching an iOS binary is time-consuming and error-prone and is therefore rarely used to modify an application. Luckily, other techniques exist to modify the behavior of iOS applications at runtime, which are discussed in the next section.

ANDROID APP MANIPULATION

We can decompile Android apps to Java but cannot recompile from Java

- Decompilation does not attempt to preserve app functionality

Possible to manipulate app using runtime debugger

- Complex because we have to attach to the Dalvik VM

Best option is to decompile app to an intermediate format that preserves functionality (if not readability)

Android App Manipulation

As we saw earlier, it is possible to take an Android APK file and convert it into a Java JAR file, which can then be evaluated with reverse engineering tools to get back to Java source code. Unfortunately, this decompilation process does not preserve the functionality of the Android application, instead favoring the ability to read and understand the Java code instead of being able to recompile.

It is possible to evaluate Android applications using a standard machine code runtime debugger process (such as the GNU debugger, or "gdb"), but this is a complex process because we have to attach to the Dalvik virtual machine where other instructions are executing, not just our target application.

To manipulate Android applications, our best option is to decompile an application into an intermediate format that preserves the application functionality. The intermediate format can be read (with patience), and it can be manipulated and recompiled into a valid Dalvik executable.

DECOMPILE TO SMALI

Smali is an intermediate disassembler/assembler designed for Dalvik executables

- Icelandic for "assembler"

Disassembles to Dalvik bytecode format

- Not a friendly or readable format

Maintains application integrity upon disassembly

Decompile to Smali

Smali is an intermediate format disassembler designed to work with Dalvik executables (through the APK file). Smali files represent the classes and methods of a .dex file by turning it into low-level Dalvik bytecode text. This format is not friendly to read or interpret, but it has a significant advantage over the JD-GUI or Procyon disassemblers in that the smali files maintain application integrity upon disassembly. This allows you to decompile and then recompile the application from the smali sources.

APKTOOL

Automates decompilation of an APK file into Dalvik bytecode

Creates a directory structure matching that of an Android source project

```
C:\TEMP>apktool.bat d cross.field.NinjaSlider_1.2.7.apk
I: Baksmaling...
testI: Loading resource table...
I: Loaded.
I: Loading resource table from file:
C:\Users\jwright\apktool\framework\1.apk
I: Loaded.
I: Decoding file-resources...
I: Decoding values*/* XMLs...
I: Done.
I: Copying assets and libs...
```

Apktool Decompiled Structure

Apktool

Apktool automates the decompilation of an APK file into the smali format, creating a directory structure that matches that of an Android source project. In the example on this slide, the Apktool "apktool.bat" script decompiles (specified with the "d" argument) the Ninja Slider app into the current directory, creating the associated resource and configuration files needed to recompile the application.

DALVIK BYTECODE

```
invoke-static {v7}, Ljava/lang/String;->valueOf(I)Ljava/lang/String;

    move-result-object v4
    .line 177
    :cond_1
    new-instance v5, Lcom/crossfd/android/NinjaSlider/utility/RestWebServiceClient;
    const-string v7, "http://198.104.58.57:8080/ninjasliderweb/submit_point.rs"
    invoke-direct {v5, v7},
Lcom/crossfd/android/NinjaSlider/utility/RestWebServiceClient;-
><init>(Ljava/lang/String;)V
    .line 179
    .local v5, restClient:Lcom/crossfd/android/NinjaSlider/utility/RestWebServiceClient;
    new-instance v2, Ljava/util/ArrayList;
```

For application analysis, smali code isn't as useful as Java decompilation. However, we can also rebuild the app after making changes to manipulate the Android application.

Dalvik Bytecode

After decompiling the application, the decompiled Dalvik bytecode source is written to the "smali" directory, an example of which is shown on this slide.

The smali code is awkward to read because it is intended to be represented in a low-level format that is turned into machine-level code with the Dalvik interpreter native to the Android platform. From an application analysis perspective, the smali disassembly isn't useful to us. However, the big advantage with smali disassembly is that the code can then be recompiled into a working executable, even after applying modifications to the sources.

DALVIK BYTECODE INTRODUCTION

Uses declared registers as placeholders for objects, variables

- vo: local register, po: parameter register

Syntax is always destination, then source

Object types are specified with a capital letter identifying the type

```
.locals 3
const-string v2, "I like Java more than I thought I did"
invoke-super {p0, p1}, Landroid/app/Activity;->onCreate()V
```


Dalvik Bytecode Introduction

The Dalvik bytecode represented in smali files declares the functionality of the application classes and methods. Each method uses declared registers as placeholders for objects and variables using an incremental numbering system, where v0 is the first local register (used within the method), and p0 is the first parameter register (used to share data between methods). In the example on this slide, the ".locals" argument indicates that three registers are required for this method.

The Dalvik bytecode syntax is designed such that the operand is on the left, followed by the destination, then the source. The const-string opcode places the string "I like Java …" (the source) into the register v2 (the destination).

Object types in Dalvik bytecode are identified with a capital letter to indicate the type. In this example, the invoke-super opcode takes two arguments (p0 and p1), which are passed to the onCreate method of the android/app/Activity class ("android/app/Activity" is a class, denoted with a leading "L"). The onCreate method returns a void type (indicated with a capital "V" at the end).

DALVIK BYTECODE TYPES

Types in Dalvik bytecode use an uppercase letter to denote the type

Influences the instructions that follow

- Many instructions work only with specific types

```
invoke-virtual {v1}, Landroid/Toast;->show()V
invoke-direct {}, Lcom/sushi/app->DoIt()J
invoke-interface {v3}, Ljava/util/Iterator;->hasNext()Z
move-result v0
if-eqz v0, :cond_1
```

Marker	Type
V	void
Z	Boolean
B	byte
S	short
C	char
I	int
J	long (64 bits)
F	float
D	double (64 bits)
L	object

Dalvik Bytecode Types

Every programming language has the concept of variable types, and Dalvik is no exception. Types in the Dalvik bytecode format use an uppercase letter to denote the type, a list of which are shown in the table on this page. In the examples on this page, the first line has an object type ("L") for android/Toast, where the show() method returns a void type ("V"). In the second line, the DoIt() method returns a long value, while hasNext() returns a Boolean (true or false) value.

The type used for variables often denotes the instructions that follow because some instructions can only perform operations on specific variable types (or they truncate the data in some cases). In the example on this slide, after calling hasNext(), which returns a Boolean value, the move-result opcode takes the Boolean return value and places it in v0, which is then acted upon with the if-eqz opcode.

Three examples of method invocation are displayed on this page. The type of method invocation is selected based on the object type being called:

- **invoke-virtual**: Invokes a normal method, one that is not marked as private (an internal method that is not intended for calling by other code outside of the object), static (a method that is accessible from an object type that has not yet been instantiated), or final (a method that cannot be overridden with another method declaration) by the developer
- **invoke-direct**: Invokes a direct method (typically one that is marked private)
- **invoke-interface**: Invokes a method in an object whose type is not known or is not validated prior to invocation

DALVIK COMMON INSTRUCTIONS

Opcode	Explanation
const vx, literal32	Put the 32-bit integer constant into vx for later comparisons
invoke-super {params}, method	Invokes the method of the immediate parent with the specified params, comma-separated
invoke-direct {p}, method	Invokes the method with the specified params
invoke-virtual {p}, method	Invokes the method with the specified params
move vx, vy	Moves the content of vy into vx
move-result vx	Moves the return value of the previous method into vx
if-eqz vx, target_loc	Skips to target_loc if vx is equal to zero
if-nez vx, target_loc	Skips to target_loc if vx is not equal to zero
check-cast vx, type	Checks if the references in vx matches the specified type

http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html

Dalvik Common Instructions

In total, the Dalvik bytecode has approximately 250 opcodes available, which is a lot by comparison to machine code languages, such as Intel assembly. Common opcodes are shown on this slide so that you have a quick reference.

On his website at the URL shown on this page, Gabor Paller has documented a list of Dalvik bytecode opcodes, with example usage. This is an excellent reference and should be kept handy for anyone reverse engineering Dalvik bytecode.

SAMPLE APPLICATION: HELLOWORLD

Examine simple Android app: HelloWorld.apk

Compare Java source to Dalvik bytecode

```
public class MainActivity extends Activity {

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    TextView t =
       (TextView)findViewById(R.id.textView1);
       t.setText("Hello, World!");
    }
}
```

Sample Application: HelloWorld

The best way to gain a better understanding of the Dalvik bytecode language is to look at a sample application written in Java and then examine the Dalvik bytecode interpretation of the Java source. The source for a basic Android application, "HelloWorldApp", is shown on this page. The app displays only the string "Hello, World!" on the activity.

In this example, the method onCreate is invoked with the parameter savedInstanceState with a type of Bundle. (A Java Bundle is a collection of string objects that can be represented in various ways.)

In the onCreate() method, the method invokes the parent onCreate method (super.onCreate), then indicates which screen should be shown for this activity (setContentView). Next, the method declares a variable "t" of type TextView, which is set to the TextView object on the activity called textView1. Finally, the label "Hello, World!" is placed in the TextView object.

From a Java perspective, this is a simple program. Now we examine this same functionality in the Dalvik bytecode format.

DECOMPILED FILE STRUCTURE

```
C:\>apktool d helloworld.apk
I: Baksmaling...
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Copying assets and libs...
C:\>cd helloworld\smali\com\example\helloworld
C:\helloworld\smali\com\example\helloworld>dir/w
[.]                 [..]                BuildConfig.smali
MainActivity.smali  R$attr.smali        R$drawable.smali
R$id.smali          R$layout.smali      R$string.smali
R$style.smali       R.smali
```

Output directory structure matches package name(s).

Files with $ are inner-class files that are automatically generated. R.smali is a mapping to resources in the UI. BuildConfig.smali is generated by Eclipse. MainActivity.smali is the code for the application.

Decompiled File Structure

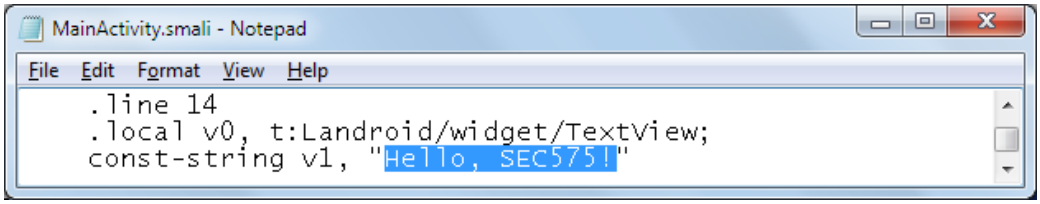
When we decompile the helloworld.apk application (you can follow along by downloading the file from E:\lab-files\helloworld.apk) with apktool, it creates a directory structure that closely resembles the original Java source directory structure. From the current directory, apktool creates a directory of the base package name ("helloworld") and within that directory, it creates a directory "smali" where all the Dalvik bytecode files are written. The directory structure within the "smali" directory matches that of the packages and classes used within the application (e.g., helloworld\smali\com\example\helloworld).

Of the ".smali" files that are created, the files with a dollar sign in the filename are inner-class files, automatically generated by the compiler. The R.smali file is a mapping of the UI resources to handles that are referenced within the application. The BuildConfig.smali file is generated by the Eclipse IDE. The file MainActivity.smali represents the functionality of the MainActivity class that is the most interesting content for us to examine. Let's look at this file next, section by section.

The output from `apktool` shown on this page has been modified for space.

EDITING DALVIK BYTECODE: SMALL CHANGES

Making small changes to bytecode is straightforward




```
C:\>apktool b helloworld
I: Checking whether sources has changed...
I: Smaling...
I: Checking whether resources has changed...
I: Building resources...
I: Building apk file...
```


Editing Dalvik Bytecode: Small Changes

Let's make a small change to the HelloWorldApp code, changing the text label to "Hello, SEC575!". Small changes such as this one are straightforward, where we can simply change the string on line 30 to display the content of our choosing as shown on this page. We can rebuild the APK file with apktool, specifying the "b" argument to build the APK file.

Apktool generates an APK file in the "dist" directory. This file cannot be used as is, however—it must first be signed.

GENERATE SIGNING KEYS

```
C:\>mkdir keys
C:\>"c:\Program Files\Java\jdk1.8.0_20\bin\keytool.exe" -genkey -v -keystore
keys/helloworld.keystore -alias HelloWorld -keyalg RSA -validity 10000
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Joshua Wright
omitted for space
Is CN=Joshua Wright, OU=Unknown, O=Will Hack for Sushi, L=Providence, ST=Rhode Island,
C=US correct?
  [no]:  yes

Generating 1,024 bit RSA key pair and self-signed certificate (SHA1withRSA) with a
validity of 10,000 days
        for: CN=Joshua Wright, OU=Unknown, O=Will Hack for Sushi, L=Providence,ST=Rhode
Island, C=US
Enter key password for <HelloWorld>
        (RETURN if same as keystore password):
[Storing keys/helloworld.keystore]
```

Your JDK path may be different

Generate Signing Keys

To sign the APK file, we need to generate a keystore that has the private signing keys and individual keys for each application we sign. For this job, we use the Java Development Kit (JDK) keytool.exe executable.

First, create a directory where the keystore files will be stored. Next, run the keystore.exe executable. The JDK installer places files in "C:\Program Files" (regardless of whether it is a 32-bit or a 64-bit system), but the path changes with each incremental revision of the JDK. Enter the majority of the path as shown in the beginning of the example shown on this slide, and then press Tab to auto-expand the remainder of the directory path.

In this example, we generate new keys with keytool.exe ("-genkey"), storing the keystore file in keys/helloworld.keystore. We give the application signature an alias of "HelloWorld", using the RSA signing algorithm with a lifetime of 10,000 days. Keytool prompts you to enter certificate information, which can be accurate or inaccurate, prior to generating the keystore.

SIGNING THE MODIFIED APK

```
C:\>"c:\Program Files\Java\jdk1.8.0_20\bin\jarsigner.exe" -verbose -keystore
keys\helloworld.keystore helloworld\dist\helloworld.apk HelloWorld
Enter Passphrase for keystore:password
   adding: META-INF/MANIFEST.MF
   adding: META-INF/HELLOWOR.SF
   adding: META-INF/HELLOWOR.RSA
  signing: res/drawable-hdpi/ic_launcher.png
  signing: res/drawable-mdpi/ic_launcher.png
  signing: res/drawable-xhdpi/ic_launcher.png
  signing: res/drawable-xxhdpi/ic_launcher.png
  signing: res/layout/activity_main.xml
  signing: AndroidManifest.xml
  signing: classes.dex
  signing: resources.arsc
jar signed.

Warning:
```

Ignore this warning

Signing the Modified APK

After the keystore is created, we can use the jarsigner utility (also in the JDK bin directory) to sign the APK file. Specify the file location to the keystore file with "-keystore", the filename of the APK file, and the alias used when you generated the keystore ("HelloWorld" in this example).

Jarsigner signs the APK file and issues a warning about timestamp validity, which can be safely ignored.

ZIPALIGNING THE MODIFIED APK

```
C:\>zipalign -fv 4 helloworld.apk helloworld_aligned.apk
Verifying alignment of helloworld_aligned.apk (4)...
      50 META-INF/MANIFEST.MF (OK - compressed)
    9938 META-INF/HELLOWOR.SF (OK - compressed)
   20144 META-INF/HELLOWOR.RSA (OK - compressed)
   21264 resources.arsc (OK)
  128746 res/drawable-hdpi/ic_launcher.png (OK - compressed)
  129050 res/drawable-mdpi/ic_launcher.png (OK - compressed)
  129349 res/drawable-xhdpi/ic_launcher.png(OK - compressed)
  129645 res/drawable-xxhdpi/ic_launcher.png (OK - compressed)
  129952 res/layout/activity_main.xml (OK - compressed)
  ...
  383662 AndroidManifest.xml (OK - compressed)
  384344 classes.dex (OK - compressed)
Verification succesful
```


Zipaligning the modified APK

After the APK has been signed, it needs to be zipaligned. This is an optimization step that aligns all uncompressed data within the APK to 4-byte boundaries. This way, Android can directly access different portions of the APK through mmap(), which reduces the amount of required RAM to run the application.

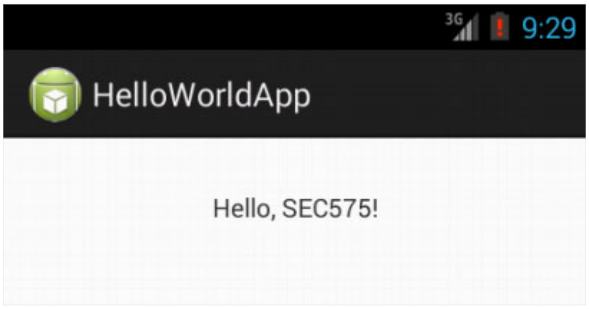
Zipalign works by modifying the extra field of the Local File Headers of the zip file. Since jarsigner signs the content of the zip file (and not the entire zip), the alignment needs to be performed after the zip has been signed.

Zipalign is included in the Android SDK Build tools.

TEST THE MODIFIED APPLICATION

```
C:\>adb uninstall com.example.helloworld
Success

C:\>adb install helloworld\dist\helloworld_aligned.apk
1817 KB/s (299717 bytes in 0.161s)
        pkg: /data/local/tmp/helloworld.apk
Success
```

Modifying existing content is straightforward; adding new code is slightly more complex.



Test the Modified Application

With the modified HelloWorld.apk signed with your new key, we can uninstall the previous version and install the modified version, as shown on this page. The HelloWorldApp displays the modified TextView label, as shown.

This example is straightforward because our change to the application is minor. We can even add new code to the smali files to achieve extra functionality at the cost of additional complexity.

ANDROID APP MANIPULATION

Manipulating Android apps with Apktool/Dalvik bytecode can be simple or elaborate

- Complexity of the target Android application
- Complexity of desired changes
- Familiarity with Dalvik bytecode

Requires practice and reference sources to be effective

Android Security Cookbook, Makan, Alexander-Brown

Other references included in notes

Android App Manipulation

In this module, we looked at the process and techniques for manipulating Android applications. Using Apktool, we saw how it is possible to take an APK file and reverse it to the Dalvik bytecode in smali files that can be examined and manipulated. The process of manipulating Android applications can vary in complexity from simple to elaborate, depending on the complexity of the target Android application, the complexity of the desired changes, and your familiarity with Dalvik bytecode syntax.

To be efficient in your ability to modify Android applications, it is necessary to practice on target apps and to have Dalvik bytecode reference sources available. Some useful reference sources are:

- http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html
- http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html
- http://source.android.com/devices/tech/dalvik/instruction-formats.html

Furthermore, guides that walk you through sample target applications can also be useful. *Android Security Cookbook* by Makan and Alexander-Bown (Packt Publishing) is a great book with a small but useful section dedicated to Android application reverse engineering through the reading and manipulation of Dalvik bytecode. Other online resources are:

- http://lazyswamp.blogspot.com/2013/08/this-is-post-to-help-to-understand-how.html
- http://androidcracking.blogspot.com/2010/09/way-of-android-cracker-0.html
- https://apkudo.com/?p=775

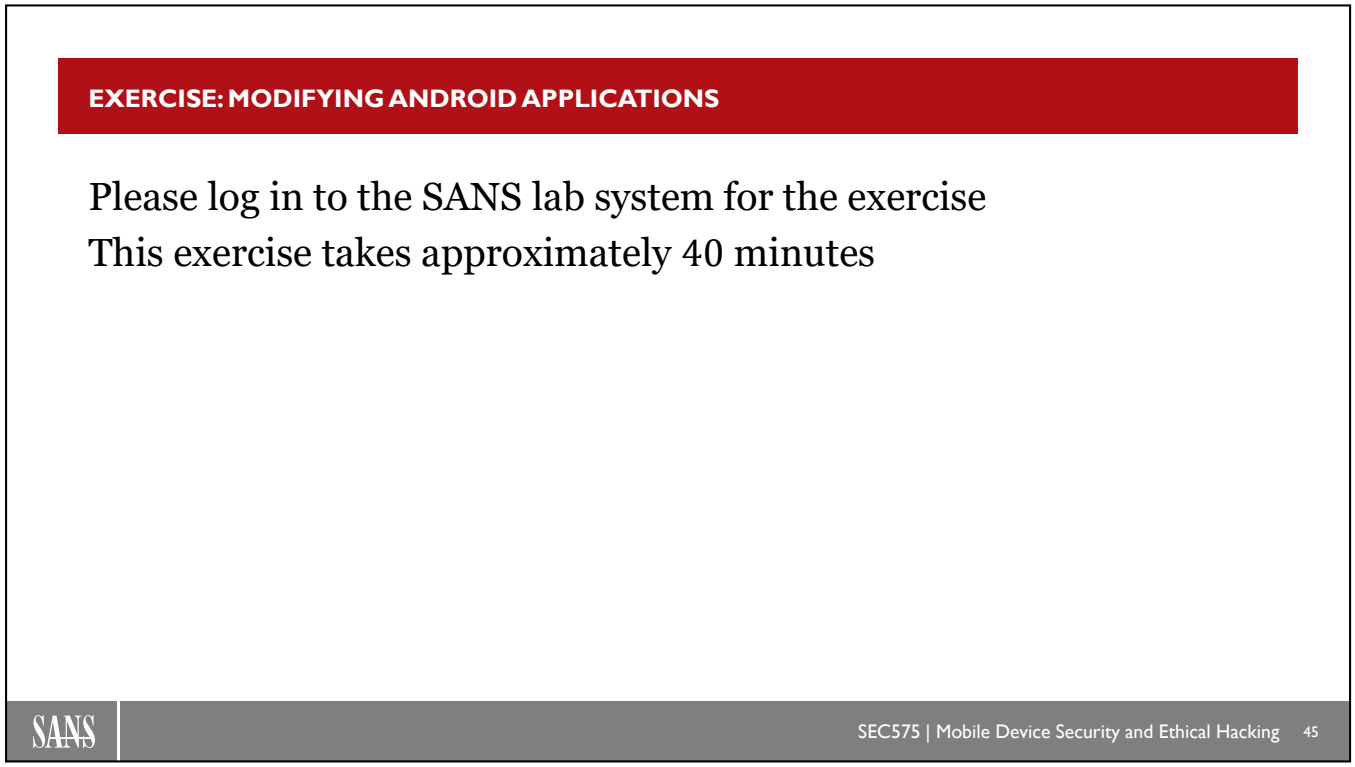
Course Roadmap

- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 4

Dynamic Mobile Application Analysis and Manipulation
Android Dynamic Analysis with Drozer
Exercise: Manipulating Android Intents
iOS Dynamic Analysis with Needle
Modifying Mobile Applications
Exercise: Modifying Android Applications
Mobile Application Runtime Manipulation
Automated Runtime Manipulation with Objection
Exercise: Frida and Objection
Application Security Verification

This page intentionally left blank.



Exercise: Modifying Android Applications

Please log in to the SANS lab system for the Modifying Android Applications exercise. This exercise takes approximately 40 minutes to complete.

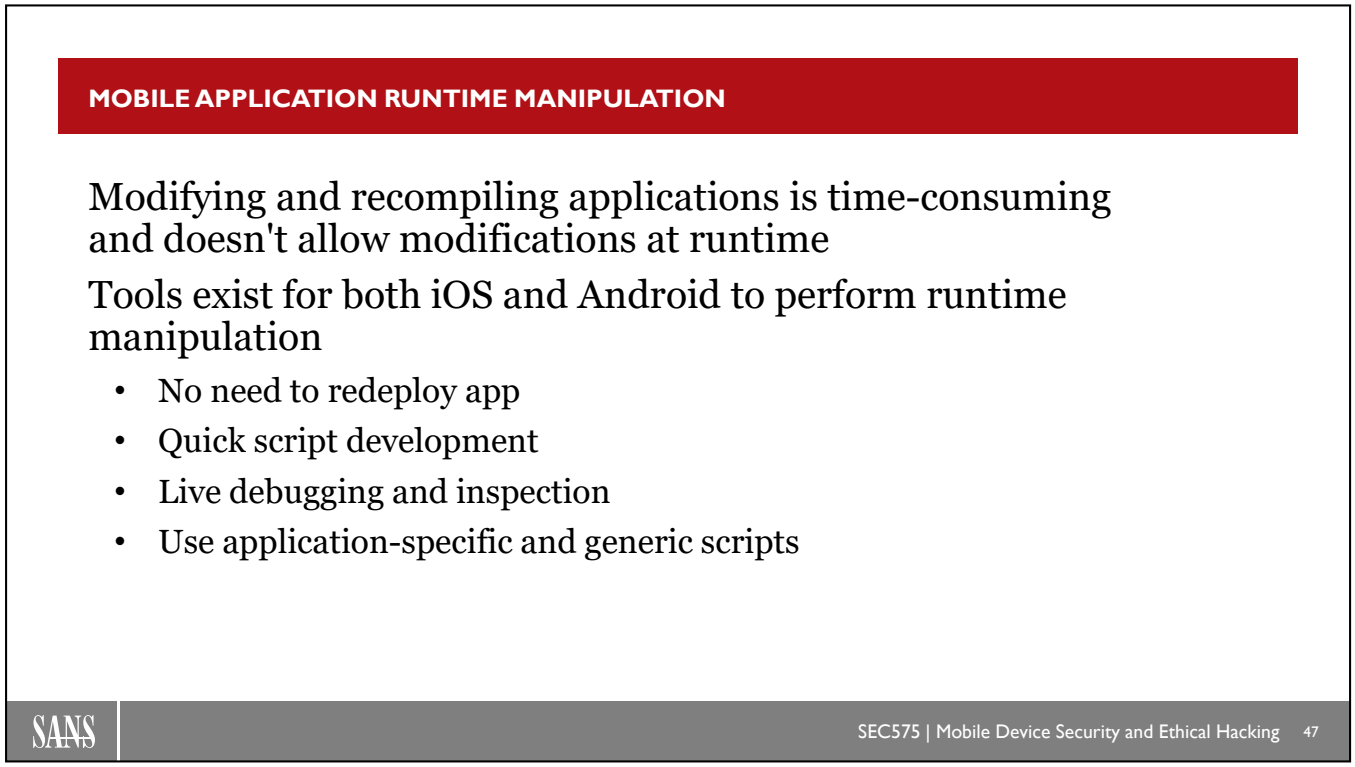
Course Roadmap

- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 4

Dynamic Mobile Application Analysis and Manipulation
Android Dynamic Analysis with Drozer
Exercise: Manipulating Android Intents
iOS Dynamic Analysis with Needle
Modifying Mobile Applications
Exercise: Modifying Android Applications
Mobile Application Runtime Manipulation
Automated Runtime Manipulation with Objection
Exercise: Frida and Objection
Application Security Verification

This page intentionally left blank.



Mobile Application Runtime Manipulation

Decompiling, modifying, and recompiling an application can be useful, but it's very time-consuming and not always easy. On iOS, it's even more difficult, as the application binary needs to be patched and the application will only be able to run on jailbroken devices. Luckily, both Android and iOS applications can be modified at runtime, which can be extremely useful as part of our analysis process and greatly simplifies the task at hand.

Using dynamic instrumentation brings many advantages over static code manipulation. There is no need to go through the lengthy procedure of recreating the application, which allows you to quickly develop scripts and test them on the original application. You also have the possibility to inspect variables and method calls at runtime and make decisions based on that information. Finally, it is possible to use both generic scripts that target the operating system or popular libraries, or application-specific scripts that are written for a specific application version.

IOS APP MANIPULATION

Decompilation in high-level Obj-C is not an option

Possible to manipulate apps at runtime through machine language code

- Using gdb, but time-consuming analysis

Can manipulate the app by accessing reflective runtime properties of Obj-C

- Access variables and objects
- Replace functions and methods with new code
- Dynamically change application behavior

iOS App Manipulation

In iOS, applications are compiled to execute natively on the iOS device ARM processor. This makes it impossible to reverse an iOS application to its Objective-C (Obj-C) source. It would be possible to reverse the iOS app to machine-level assembly code with gdb, but evaluating or manipulating that code would be a cumbersome process for all but the simplest tasks.

Fortunately, another option is available. It is possible to manipulate the application by accessing the reflective runtime properties of Objective-C. Through this mechanism, we can access internal variables and objects in the code at runtime and dynamically change the application behavior. Aiding us in this task is the library extension for jailbroken devices, Cydia Substrate.

CYDIA SUBSTRATE

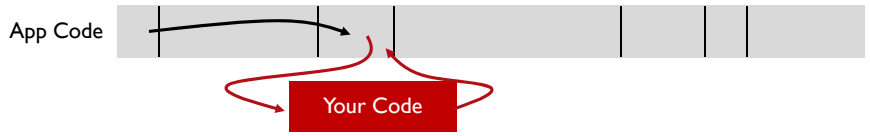
Library developed by Saurik

Redirect functions and manipulate variables in a running app

- Programmatically or in a short script

Used by many Cydia tweaks to bypass restrictions within applications

Utilizes Obj-C reflective properties



Cydia Substrate

Cydia Substrate is a development library developed by Saurik, the same developer who wrote the Cydia installer and is active in the iOS jailbreak community. Using this library, developers can write their own code to take advantage of a common characteristic of all iOS apps to manipulate the runtime behavior of a target app.

Objective-C is a reflective, object-oriented programming language. A reflective programming language allows a developer, at runtime, to examine and modify the behavior of a program. Using Objective-C's reflectivism, a developer can change the type and values, properties, and functions of a program as needed without recompiling the code. To meet this behavior, all Objective-C programs include the names of all variables and functions used in the code, as we saw in using the class-dump tool.

Cydia Substrate allows a developer to write a program that attaches to a running program and manipulates the behavior of the app using Objective-C reflectivism in any way desired. For example, an attacker may write a tool using Cydia Substrate that attaches to the Apple iOS SpringBoard app, manipulating the device unlock function to bypass the need for a passcode by skipping that code block. We examine a method for doing this attack shortly.

Cydia Substrate is a popular library for jailbroken iOS users because many of the Cydia tweak packages modify the common behavior of iOS apps to enhance the platform (such as changing font use in the iOS SpringBoard or squeezing five icons in the dock of an iPhone). One popular way to leverage Cydia Substrate without writing Objective-C code is to use the Cycript tool.

CYCRIPT

Scripting language in JavaScript syntax, accessing Obj-C objects

- Install using Cydia after installing Cydia Substrate

Can attach to any running iOS process, leveraging Cydia Substrate function redirection capabilities

Manipulates program in memory

- Nonpersistent modifications

Primarily a developer tool, but fun

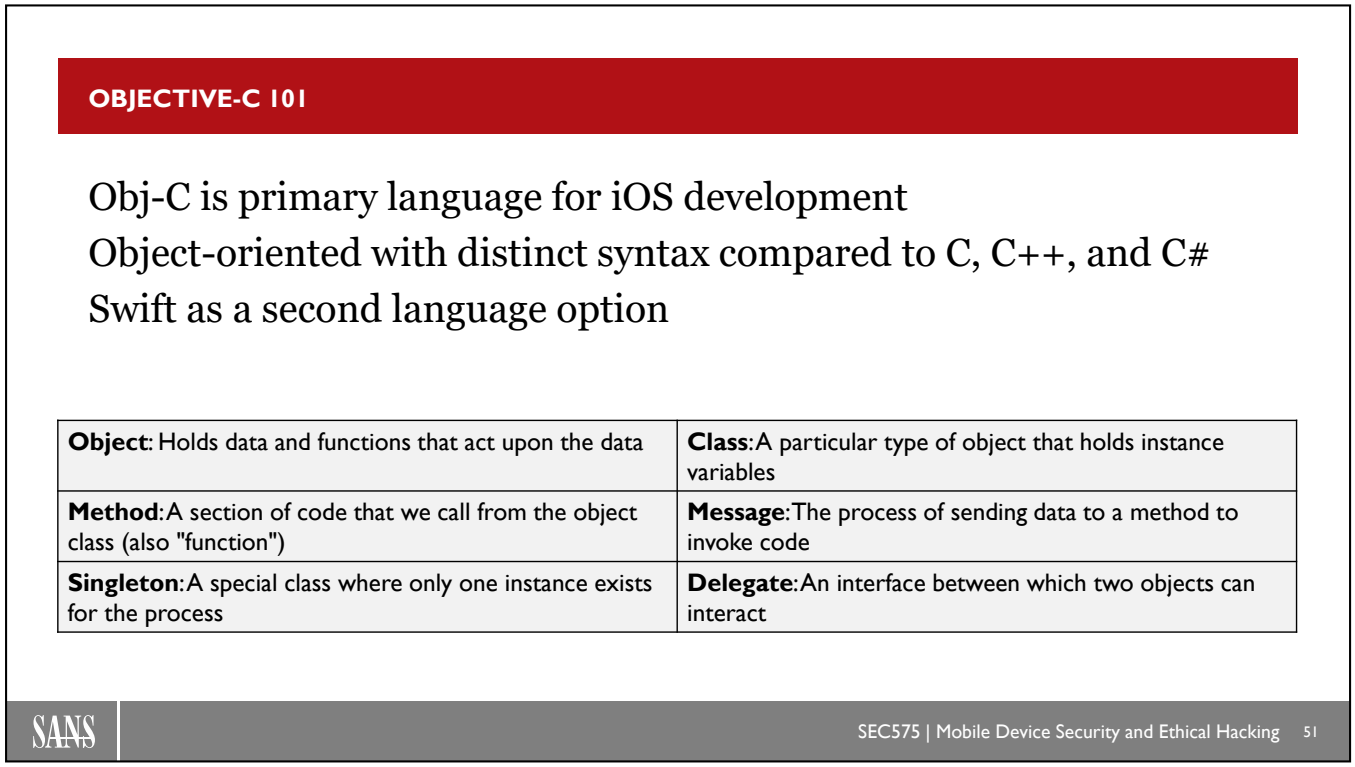
Cycript

Cycript is a scripting language that uses JavaScript syntax to access Objective-C object data. Cycript uses Cydia Substrate to attach to a running application and manipulate the program's functionality. Using the Cycript command line tool, you can interactively write short code blocks to manipulate an application or write complex scripts to be run from the command line.

When we manipulate applications with Cycript, we attach to the running process and manipulate the system in a non-persistent way: Changes made with Cycript are made in memory only and are lost when the application terminates and restarts. Several solutions to automate the application of Cycript changes are available, including the capability to invoke a Cycript script each time the application launches.

To develop with Cycript, you must have some familiarity with Objective-C message passing, the Apple iOS APIs, and the JavaScript programming language. This may be a significant barrier to entry for most users, but with a little knowledge about how Cycript works, we can use prepared script to achieve useful functionality on iOS.

Cycript is typically installed with the Cydia package manager on iOS, but it can be downloaded from https://cydia.saurik.com/api/latest/3. Cycript documentation is available at http://www.cycript.org.



Objective-C 101

To understand the power and functionality of Cycript, it is necessary to first gain a basic understanding of the Objective-C programming language. In iOS, developers use Objective-C (or Swift, starting in iOS 8) to write applications, which is a distinct departure from the syntax methods used in C, C++, and C#. We look at some basic Objective-C syntax in this module, but first, let's get a basic understanding of concepts in Objective-C:

- **Object**: In Objective-C, programmers use objects in their code, which can combine groups of data (variables) and functionality in the form of methods that take action on that data.
- **Class**: An Objective-C class is an instantiated object that holds instance variables and methods; a class is a specific type of an object that has been allocated by the programmer for use.
- **Method**: A method is a section of code that is called from the object class (sometimes referred to as a function).
- **Message**: In Objective-C, methods are not called directly; instead, they are sent messages that instruct the method to take an action.
- **Singleton**: A singleton class is a special class where only one instance of the class exists for the process; iOS uses singleton classes to share data between multiple classes in the running process.
- **Delegate**: A delegate is an object interface used by two independent objects for communication and interaction.

Now that we have some basic vocabulary, let's look at some syntax of Objective-C compared to the (hopefully more familiar) syntax of C++.

LAUNCHING CYCRIPT

Identify the process by Mach-O executable name
Attach with Cycript by PID or process name

```
# ps -ef | grep Navigator
  501  8889     1   0   0:00.00 ??         0:00.49 /var/mobile/Applications/E9E54F01-
0E82-4C4E-B3FE-00EFF1DA5063/Navigator.app/Navigator
    0  8891  8865   0   0:00.00 ttys000    0:00.01 grep Navigator
# cycript -p Navigator
cy#
```


Launching Cycript

To launch Cycript, identify the name of the process to which you want to attach. The process name is usually a variation of the application name, although not always. In the example on this slide, we search the process list for the string Navigator, which is the process name for the DirecTV iPad app.

Cycript can attach to a process by the process ID or the process name with the "-p" parameter, as shown.

EXPLORE APP CLASSES

Cycript has a built-in function "ObjectiveC.classes" that enumerates all app classes
- This is a lot of output

Write to a file for easier parsing
Alternative: class-dump

```
cy# ObjectiveC.classes
{PFUbiquitySaveSnapshot:#"PFUbiquitySaveSnapshot",TAGPValueAndStatic:#"TAGPValueAndStatic",UIMotionEve
nt:#"UIMotionEvent", ... //NSDictionary
cy# [ObjectiveC.classes allKeys]
@["PFUbiquitySaveSnapshot","TAGPValueAndStatic", ...    //NSArray
cy# var classes = [[ObjectiveC.classes allKeys] componentsJoinedByString:@"\n"];
//NSString
cy# [classes writeToFile:"/var/mobile/cycriptoutput.txt" atomically:NO encoding:4 error:NULL]
# ^D
# sort /var/mobile/cycriptoutput.txt | head
AAAccount
AAAccountManager
AAAccountMigrator
```


Explore App Classes

The recommended technique for identifying the names of available classes in an application instance is to retrieve the header data using class-dump after decrypting an application. As an alternative, Cycript enables us to explore the classes available using the built-in ObjectiveC.classes object. Entering this value by itself prints a lot of output, including each class name and address, and likely exceeds the buffer size of your terminal window. To limit the output to just the names of the classes themselves, send ObjectiveC.classes the allKeys message as shown, which returns an NSArray object.

NSArray objects can be converted to string objects by passing the componenentsJoinedByString message with an optional argument identifying a delimiter that is inserted between each string. In the example on this slide, the "classes" variable is declared and holds the string object returned by joining the NSArray string objects into a single string, delimited by a newline character ("\n"). Passing the writeToFile message to the string object allows us to write the contents of the classes list to a file that we can then sort and examine as desired.

EXPLORE THE CURRENT WINDOW

UIApp.keyWindow is the current open window interface

UIApp.keyWindow.rootViewController provides access to the current window

Use printMethods to explore object name


```
cy# UIApp.keyWindow
#"<UIWindow: 0x146787e0; frame = (0 0; 320 480); autoresize = W+H; gestureRecognizers =
<NSArray: 0x14590c10>; layer = <UIWindowLayer: 0x1466ca50>>"
cy# UIApp.keyWindow.rootViewController
#"<WHFSViewController: 0x14665fc0>"
cy# printMethods(WHFSViewController)
[{selector:@selector(isJailbroken),implementation:0x22f21},{selector:@selector(labelOutpu
t),implementation:0x23035},{selector:@selector(setLabelOutput:),implementation:0x23055},{
selector:@selector(buttonClicked:),implementation:0x22fcd}]
```


Explore the Current Window

The UIApp member variable keyWindow is used to refer to the current open window interface, which can consist of multiple "screens" of content that overlay each other, depending on the use of the application. The keyWindow.rootViewController is the object that provides access to the current window.

We can examine the methods of the current open window to get a perspective on what is accessible at the current moment based on the application behavior. You can use the command "UIApp.keyWindow.rootViewController" to identify the name of the class that is the current rootViewController.

In the example on this slide, a simple app that tests for jailbroken devices indicates that the device is jailbroken with the message displayed on the screen. Looking at the UIapp.keyWindow.rootViewController output, the class WHFSViewController is the current rootViewController. Using the printMethods() function, we can explore the available methods in the WHFSViewController object, the first of which is called "isJailbroken".

EXAMINE METHODS

iPhoneDevWiki has sample functions for exploring applications

- printMethods(className);
- methodsMatching(className, regex);

Paste this code into your Cycript session and explore the selected Class

```
cy# printMethods(WHFSViewController)
[{selector:@selector(isJailbroken),implementation:0x420e1},{selector:@selector(labelOutpu
t),implementation:0x421f1},{selector:@selector(setLabelOutput:),implementation:0x42219},{
selector:@selector(buttonClicked:),implementation:0x42119},{selector:@selector(didReceive
MemoryWarning),implementation:0x42019},{selector:@selector(viewDidLoad),implementation:0x
41fd5},{selector:@selector(.cxx_destruct),implementation:0x42249},{selector:@selector(fil
eExistsAtPath:),implementation:0x4205d}]
cy# methodsMatching(WHFSViewController,/file/)
[["c12@0:4@8","fileExistsAtPath:"]]
```


Examine Methods

The website iPhoneDevWiki has a page titled Cycript Tricks (http://iphonedevwiki.net/index.php/Cycript_Tricks) that includes several Cycript code snippets that are useful for evaluating applications. Functions including printMethods and methodsMatching allow us to search a specific object to identify the accessible methods in the object. The printMethods function displays the accessible methods in the identified object, while methodsMatching displays the methods that match a regular expression string match, as shown on this slide.

To use these functions, keep the Cycript Tricks page open in your browser (or a convenient document) and paste them into your Cycript session as needed. (The methods themselves are included below for reference.)

In the example on this slide, we look at the output for the object WHFSViewController, a simple app that tests if a device is jailbroken. In this output, we see several methods, including isJailbroken, that look interesting. Using methodsMatching, we can search for methods that contain a specific string, such as "file", as shown.

CYCRIPT COMPLETION

Cycript helps you navigate objects with tab completion

Enter the beginning of the object, method, or iVar name

- Press Tab twice

Getting into the habit of using tab completion reduces your typing and helps eliminate typos

```
cy# skipLi[Tab][Tab]
cy# skipLimitState->[Tab][Tab]
dailySkipCounter    stationSkipCounter
isa                 version
cy# skipLimitState->sta[Tab][Tab]
cy# skipLimitState->stationSkipCounter
6
```


Cycript Completion

One of the powerful features of Cycript is the ability to use tab completion to navigate and explore object methods and iVars. Enter the beginning of the object name and press Tab twice ("[Tab][Tab]" is used in the examples). When Cycript can uniquely identify the object you have entered, it completes the remainder of the object name. If there are multiple matches, Cycript lists them and returns you back to the partially completed object name so you can choose how to complete it. If there are no matches, Cycript shows no output, and you are left with the partially entered object name.

Get into the habit of using tab completion when using Cycript. Even if you know the name of the object you are accessing, you will be faster with tab completion and less likely to make a typo when entering the object name.

Now let's look at an example of using tab completion to evaluate the Pandora application, manipulating instance variables.

SETTING INSTANCE VARIABLES

Cycript can manipulate instance variables

- Use this to change how applications behave

Example: Pandora Online Radio and skip limiting

"Free Pandora accounts permit 6 skips per hour per station, for up to 24 total skips per day across all stations. The daily skip limit helps us prevent having to pay royalties on songs that are not being heard." help.pandora.com, Skip Limit

Setting Instance Variables

Cycript can display methods and instance variables using the tryPrintIvars() function or with tab completion, but it can also manipulate these variables. We can use this when testing applications to see how changes we introduce change the functionality of the application. As an example, let's look at the Pandora application for iOS. Pandora limits users to the number of songs they can skip per hour and per day as part of an effort to keep the costs of the service down. We can examine this functionality in Cycript and manipulate it by changing the value of instance variables.

UNLIMITED SKIPS

```
cy# var skipLimit = [ SkipLimitState sharedSkipLimits ]
#"<SkipLimitState: 0x15516cd0>"
cy# skipLimit->[Tab][Tab]
_dailySkipCounter    _stationSkipCounter  _version            isa
cy# skipLimit->_stationSkipCounter->[Tab][Tab]
_counts                  _maximumCountPerDuration
_duration                isa
cy# skipLimit->_stationSkipCounter->_maximumCountPerDuration
6
cy# skipLimit->_stationSkipCounter->_duration
3600
cy# skipLimit->_stationSkipCounter->_duration = 0
0
```

3600 is the number of seconds in a day. Changing this to 0 causes the stationSkipCounter to reset much more frequently.

Pandora watches server logs to track use; don't do this

Unlimited Skips

In this example, we examine the behavior of the Pandora app on iOS, version 5.6.3. By analyzing the Pandora classes with class-dump, we see the class SkipLimitState that has a singleton called sharedSkipLimits. (Remember that a singleton is an Obj-C object that can exist only once in a running program.) In the beginning of this example, we declare a variable skipLimit that points to the sharedSkipLimits singleton.

To examine the methods and instance variables in the skipLimit variable, enter the beginning of the variable name followed by "->" and press Tab twice. This displays four objects. If we investigate the _stationSkipCounter object, we see an additional four objects, including _maximumCountPerDuration and duration.

It's likely that either of these instance variables could be manipulated to change the behavior of the Pandora application: changing _maximumCountPerDuration from 6 to a large number prevents Pandora from alerting the user following six skips within the duration period. Alternatively, we can change the _duration value from 3600 (the number of seconds in an hour) to 0, which causes the duration for tracking the number of skips to a value where the app never exceeds the number of skips for the duration.

Note that when you skip tracks with Pandora, the app talks to the Pandora servers to retrieve the new audio information. It's likely that Pandora watches the server logs to track user behavior where the number of skips for a given period exceeds what they anticipate. Although it's useful to use this example with Cycript for demonstration, we don't recommend you use this technique to subvert Pandora for regular use.

Now that we've examined some techniques to explore applications, we have enough information to look at manipulating applications through a process called *method swizzling*.

METHOD SWIZZLING

Identify the class and the method you want to manipulate

- class-dump or with Cycript

Create the new function you want to apply in JavaScript

```
ControllerName.messages['methodName'] = function() {
    // Your function code, for example:
    return false;
}
```


Method Swizzling

Method swizzling is the process where we identify a method in an object of the target application and redirect it to a custom JavaScript function that we create. First, we identify the name of the object and the method that we want to manipulate, then we create a short JavaScript function that is executed in its place.

The example on this slide illustrates the basic outline of method swizzling. For the controller object, we invoke the messages method, identifying the method we want to manipulate inside the square brackets. We set that value equal to the new JavaScript function that we create.

INVALID WORDS

```
# cycript -p WordsWithFriendsFreeiPad
cy# [ObjectiveC.classes allKeys]
@["UINib","BurstlyCurrencyUpdateInfo","GEOTileLoader","NSSortedArray","AARegisterRequest"
,"PLLocationController",...
cy# ObjectiveC.classes["WordGameDictionary"]
#"WordGameDictionary"
// Paste printMethods()
cy# printMethods(WordGameDictionary)
[{selector:@selector(isValidWord:),implementation:0x241f71},{selector:@selector(getDawgAr
ray),implementation:0x241e41},...]
cy# WordGameDictionary.messages['isValidWord:'] = function() {return true;}
function () {return true;}
```

Invalid Words

The popular game Words with Friends for iPad can be manipulated to disable the client-side logic that validates the word content played in the game (at the time of this writing). Attaching Cycript to the WordsWithFriendsFreeiPad process, we can list all the classes using the `[ObjectiveC.classes allKeys]` method. This generates a lot of output, but one class that looks interesting is WordGameDictionary.

Using the printMethods function from the iPhone Development Wiki, we can examine the accessible methods in the WordGameDictionary object. One interesting method is "isValidWord". Redirecting this method to a JavaScript function that always returns true modifies the behavior of the game so that we can play any combination of letters as a valid word, as shown on this slide.

JAILBREAK DETECTION EVASION

Jailbreak detection methods vary across apps
Redirect simple "isJailbroken" method to return "false"

```
cy# UIApp.keyWindow.rootViewController
#"<WHFSViewController: 0x14665fc0>"
cy# printMethods(WHFSViewController)
[{selector:@selector(isJailbroken),implementation:0x22f21},{selector:@selector(labelOutpu
t),implementation:0x23035},{selector:@selector(setLabelOutput:),implementation:0x23055},{
selector:@selector(buttonClicked:),implementation:0x22fcd},{selector:@selector(didReceive
MemoryWarning),implementation:0x22ea1},{selector:@selector(fileExistsAtPath:),implementat
ion:0x22ecd},{selector:@selector(viewDidLoad),implementation:0x22e75},{selector:@selector
(.cxx_destruct),implementation:0x23069}]
cy# WHFSViewController.prototype.isJailbroken = function () { return false; }
```


Jailbreak Detection Evasion

Returning to our TestForJailbreak application, we can also use method swizzling to change an app's behavior to evade jailbreak detection. In the printMethods() output for WHFSViewController, we see the method "isJailBroken". We can redirect that function to one of our own choosing that always returns false to avoid jailbreak detection for this app.

More complex applications may instrument other techniques for jailbreak detection, but an attacker with time, patience, and creativity can always find a method where that level of testing can be subverted.

SWIFT AND CYDIA SUBSTRATE

No support from Cydia Substrate for native Swift

- Can manipulate Swift-linked Obj-C libraries

Swift methods marked with the @objc header can be manipulated with Cycript

- Exposed when developers want compatibility between Obj-C and Swift

Future changes to Cydia Substrate could include Swift swizzling by patching virtual function lookup table in memory

Swift and Cydia Substrate

Unlike Objective-C binaries, native Swift binaries cannot be manipulated using Cydia Substrate and Cycript. Swift binaries that call Objective-C libraries can be manipulated by redirecting the Objective-C code, but this technique cannot be applied to manipulate native Swift code.

In some cases, developers may write libraries in Swift with the intention of including backward compatibility with Objective-C programs. This is done by marking Swift headers with the "@objc" directive. In limited cases, Cydia Substrate can be used to manipulate these Swift applications.

To date, there has been little public discussion or research on techniques for manipulating Swift applications using Cydia Substrate or other techniques. It is possible that future releases of Cydia Substrate could patch running Swift applications by manipulating virtual function lookup tables in memory, a technique that can similarly be applied to C++ applications.

IOS APP MANIPULATION

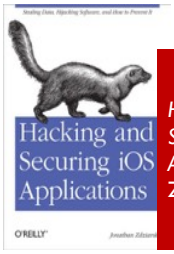
Manipulating iOS apps with Cycript is an advanced analysis technique

- Requires knowledge about iOS SDK development with Obj-C
- Requires time to explore and experiment with target applications

Powerful tool for iOS app testing

Objective-C Programming:The Big Nerd Ranch Guide, 2nd Edition, Hillegass and Ward

iOS Programming:The Big Nerd Ranch Guide, Keur, Hillegass, and Conway



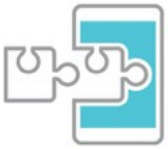
Hacking and Securing iOS Applications, Zdziarski

iOS App Manipulation

Manipulating iOS apps with Cycript is an advanced analysis technique, requiring a lot of background knowledge in Objective-C, JavaScript, the iOS SDK and iOS APIs (Cocoa), and iOS reverse engineering tools. Some useful books for learning these topics include *Objective-C Programming: The Big Nerd Ranch Guide*, *2nd Edition,* by Hillegass and Ward (Big Nerd Ranch Guides) for Objective-C, and *iOS Programming: The Big Nerd Ranch Guide,* by Keur, Hillegass, and Conway (Big Nerd Ranch Guides) for iOS API and SDK information. Although a little older, *Hacking and Securing iOS Applications* by Jonathan Zdziarski (O'Reilly Press) remains a great introduction to using Cycript for application manipulation.

XPOSED

- Android framework to install extra "modules"
  - Improved battery management
  - Disable auto-sleep for specific apps
  - UI modifications
- Requires rooted device
- Allows hooking into applications
- Modules are written in Java and need to be compiled to APKs
  - Slow development, but very accessible
- Still popular for end user plugins, less for security audits



Xposed

Xposed is a framework that allows customizing Android devices by installing extra modules. These modules can add several features to the Android OS, such as:

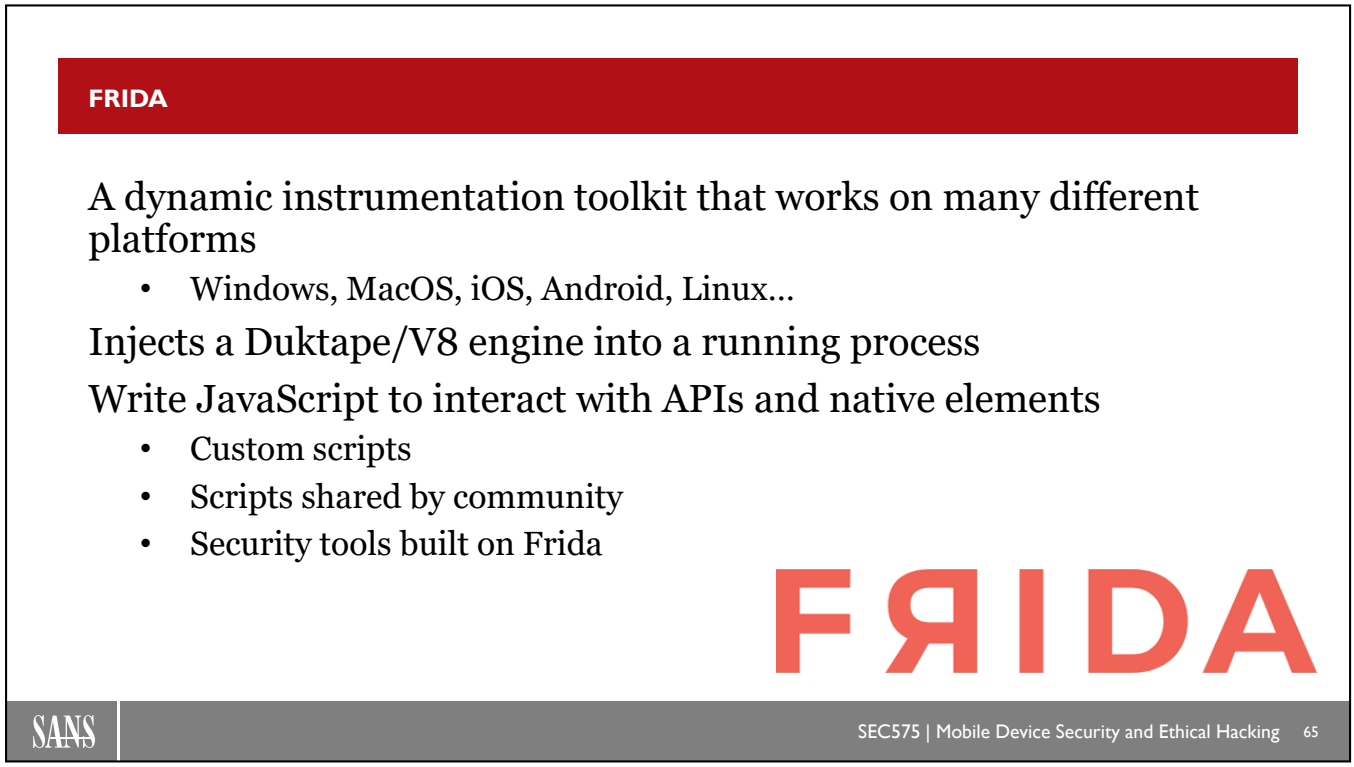
- Better battery management
- Preventing specific apps from auto-sleeping
- Modifying the UI (for example, changing the appearance of the clock)

To accomplish the above, Xposed modules hook into applications in order to modify their behavior as desired.

Xposed is distributed with an app called Xposed Installer, which allows users to search for and install modules in a fashion similar to how the app store works. The Xposed Installer app can only be installed on rooted devices.

Modules are developed in Java and compiled into APK files that then have to be installed on the device. In comparison to the tools that we will see next, this is a very time-consuming process. The upside is that the module itself is programmed in Java, which makes it very easy to write new code to interact with the targeted application.

There is still an active Xposed community of users that creates modules to modify parts of the operating system or applications. With the arrival of Magisk, the popularity is declining as Magisk Modules offer many of the features that Xposed modules provide, while being much less intrusive to the device.



Frida

Frida is a reverse engineering toolkit developed by Ole André V. Ravnås, a Norwegian security researcher working at NowSecure. More specifically, Frida is a dynamic code instrumentation toolkit that allows you to inject snippets of code or even full libraries into native apps on Windows, macOS, GNU/Linux, iOS, Android, and QNX. Frida is a relatively new tool but has been rapidly gaining in popularity due to its powerful nature and the wide support for various platforms. Various APIs exist for different platforms that allow you to interact with the application, hook functions, replace implementations, or interact with the filesystem through the application. This could be useful in various scenarios:

- A new, fancy iOS application is released and you wish to interop with it and build an Android version. However, the application is encrypting all the traffic, so it's impossible to trace the API by using a proxy tool. Frida can be used to hook in the application and log all the requests in plaintext before they're sent to the server.
- You wish to extend an existing Windows application deployed at the customer to provide additional logging and diagnostic techniques without rebuilding and redeploying the full application. Using Frida, it's possible to add the required logging mechanisms and just send the Frida script instead of a new version of the application.
- Perform a thorough black-box test of an application of which you do not have the build resources or the source code. Frida allows you to bypass SSL pinning and jailbreak detection, which could make penetration testing quite difficult.

In order to run your injected code snippets, Frida injects Google's V8 or Duktape JavaScript engine into the target process of the native application. This JavaScript engine executes the injected JavaScript code with full access to low level memory, system level API's and even allows for calling native functions inside the binary. To facilitate this, a Frida client on your device is communicating with a Frida server running inside the injected binary.

Since any script can be injected inside native apps, it allows for an endless amount of instrumentation techniques. Therefore, Frida heavily relies on custom scripts that all have their own purpose. Scripts for recurring processes like bypassing jailbreak/root detection or SSL pinning are often shared by the community.

Although Frida has a steep learning curve, many security researchers are finding their way to Frida and building tools on top of it in order to facilitate these recurring processes. A notable example is the Objection framework, which provides an intuitive GUI for interacting with the application and performing typical tasks such as jailbreak detection evasion and the circumvention of SSL pinning. Other tools building on top of Frida are Needle and Passionfruit.

In the following slides, we merely focus on using Frida for instrumenting native mobile applications (iOS and Android).

FRIDA FOR ROOTED DEVICES: FRIDA-SERVER

frida-server runs with full permissions on the target device and has full access to applications and filesystem

(Almost) all applications can be injected

Requires least amount of setup

Install frida-server binary on device

- adb
- Cydia
- SSH

Frida for Rooted Devices: frida-server

Frida can be run both on jailbroken/rooted devices and non-jailbroken/rooted devices. Since the setup on jailbroken/rooted devices is a bit more straightforward and requires the least amount of setup, we will cover this one first.

In general, frida can be injected into any application, although it is possible for applications to actively try to prevent this, just like applications can try to prevent a debugger from being attached.

As mentioned earlier, Frida sets up a communication channel between a client component running on the device of the tester and a server component (frida-server) running on the target device.

In order to install the frida-server on your jailbroken/rooted device, one of the following methods could be used:

- On iOS, the package manager Cydia could be used to install and start frida-server directly. Add “build.frida.re” as package source to find the correct package.
- On Android, our own frida-server Magisk Module can be used to install frida-server and automatically starts the server after a device boot (https://github.com/TheDauntless/Magisk-Frida-Server).
- Download one of the pre-built frida-server binaries for iOS or Android from Frida's releases page on GitHub (https://github.com/frida/frida/releases) and install them using adb (Android) or ssh (iOS).
  - **Android**:
    1. adb push frida-server /data/local/tmp/
    2. adb shell "chmod 755 /data/local/tmp/frida-server“
    3. adb shell "/data/local/tmp/frida-server &“

.

- **iOS**:
    - scp frida-server root@xx.xx.xx.xx:/tmp/frida-server
    - chmod 755 /tmp/frida-server
    - /tmp/frida-server -D

Once frida-server is installed on the jailbroken/rooted device, it should be run with root privileges. On both iOS and Android, this can easily be done on a rooted/jailbroken device by running the frida-server binary as the root user.

Afterwards it runs with full permissions on the target device and has access to all applications and the whole filesystem. It is capable of spawning an existing program, attaching to a running program, or hijacking one as it's being spawned, and then run your injected JavaScript code inside of it.

FRIDA-SERVER

frida-server runs on target machine and injects code

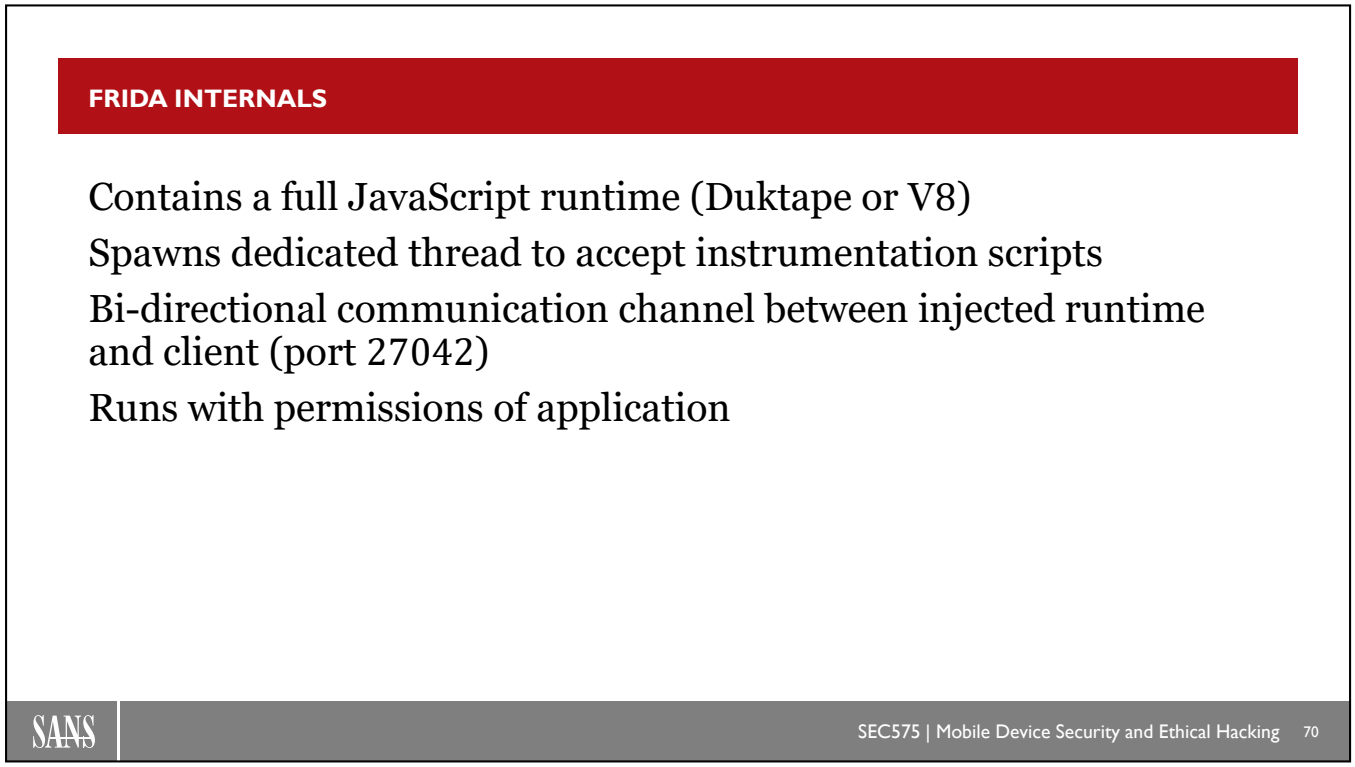
Multi-step process:

- Bootstrapper code is written into target memory
- Thread is hijacked to execute bootstrapper code
- Bootstrapper code loads frida-agent.so
- Similar behavior to malware

frida-server

After frida-server is installed, it is capable of injecting code inside the process of running applications. In order to achieve this, the frida process writes bootstrapper code into the memory of the target process of the application. Next, frida will hijack an existing thread in the target process and let it execute the bootstrapper code. The bootstrapper code loads frida-agent.so into the memory space of the target. The frida-agent.so is the core of the frida framework and allows users to run their injected code and interact with native functions inside the target process.

This bootstrapping process is quite similar to how some malware infects applications.



Frida Internals

The Frida agent contains a full JavaScript runtime. By default, the Duktape JS runtime is chosen over Google's V8 JS runtime since Duktape is more memory efficient and garbage collection is more optimal. If you will heavily inject JavaScript code or require ES6 support, you might be better off to switch to the V8 engine.

Once the Frida agent is running, it will set up a bi-directional channel between the Frida client running on the host machine and itself in the target process runtime by listening on port 27042. The agent spawns a dedicated thread to listen for incoming connections and accepts instrumentation scripts originating from the Frida client, which will then be executed by the injected JS engine.

Since a hijacked thread of the targeted app spawned the Frida agent, it runs with the same permissions as the app. Unless an app is explicitly launched with root privileges, apps still run with limited privileges in a sandboxed environment, even on jailbroken or rooted devices.

FRIDA CLI (1)

Collection of official support tools

```
> pip install frida-tools
```

frida –U org.telegram.messenger	Interactive CLI interacting with the JavaScript runtime
frida-ps –Ua	Listing injectable processes
frida-trace -m "-[InitialViewController *]" -U -f com.apple.safari	Trace function calls through pattern matching
frida-discover –U org.telegram.messenger	Discover internal functions for tracing
frida-ls-devices	List available frida-server connections
frida-kill -U 12345	Kill a process

Frida CLI (1)

Frida comes with a bunch of official support tools that aid the instrumentation process. These are available as a Python pip package and can easily be installed on the host machine.

> pip install frida-tools

Take note that the "frida-tools" contain the client part of Frida, while the pip module "Frida" only contains the server part running on the target device (frida-server).

The toolkit includes the following six tools:

- **frida:** This is the Frida CLI. The Frida CLI is the main way to interact with the injected Frida runtime on the target device. By adding the -U parameter, it's possible to connect to a frida-server running on a connected device.
- **frida-ps:** This tool gives an overview of the available processes that can be instrumented. By adding the -a parameter, only processes of applications are listed. This makes it easy to find the correct package name of an application running on a connected device.
- **frida-trace**: This tool allows you to search for invocations of methods based on a pattern. In the example, all methods of the InitialViewController are monitored by specifying the method using the –m flag. Frida-trace will print when one of the methods is invoked, and it does so by creating JavaScript files containing hooking code for the different methods. These hooks can be copied, customized, and used in the Frida CLI through the –l flag.
- **frida-discover:** This tool allows you to discover internal functions in a program, which can then be traced by using frida-trace.
- **frida-ls-devices:** Shows all of the available devices to target with Frida. It will typically have at least two targets: a direct communication to the local host and a TCP connection to the local host.
- **frida-kill**: Allows you to kill a process in case it is unresponsive.

One of them is the Frida CLI which interacts with the JavaScript runtime running on the Frida agent inside the injected app. Additionally, it exposes platform bindings of iOS and Android

FRIDA CLI (2)

Interactive command line interface with JavaScript runtime

Exposes platform bindings

Connect to local or remote frida-server

Can launch processes or attach to running processes

- Interesting for early instrumentation

Can load scripts and listen for changes

Frida CLI (2)

The Frida CLI tool inside the Frida toolkit (the "*frida*" binary) interacts with the JavaScript runtime running on the Frida agent inside the injected app. This allows us to hook into different classes and methods to change their behavior at runtime and interact with the platform bindings of iOS and Android to launch activities, read files, and much more.

This interactive command line interface tool emulates some features of IPython, allowing developers and testers to easily prototype and debug applications.

The Frida CLI can either be run on the host machine and connect to the remote frida-server running on the targeted mobile device or it can be run from the mobile device and connect to the local frida-server. The Frida CLI allows users to launch new processes or attach to an existing process and interact with platform-specific bindings. This could particularly be helpful for early instrumentation techniques, for example, listing local variables, viewing exposed interfaces, or dumping classes.

Since everything is directly interpreted by the V8 or Duktape runtime in the target process, the internals are exposed as JavaScript objects. In order to list or interact with these objects, one could just read out the declared objects or perform function calls.

Alternatively, scripts containing multiple commands or function calls can be loaded from the filesystem, which are automatically reloaded once modifications are made. This allows you to write scripts from an IDE (Integrated Development Environment) and test them without having to reload the application.

FRIDA CLI: STARTUP OPTIONS

```
> frida -h
Usage: frida [options] target

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -D ID, --device=ID    connect to device with the given ID
  -U, --usb             connect to USB device
  -R, --remote          connect to remote frida-server
  -H HOST, --host=HOST  connect to remote frida-server on HOST
  -f FILE, --file=FILE  spawn FILE
  -n NAME, --attach-name=NAME
                        attach to NAME
  -p PID, --attach-pid=PID
                        attach to PID
  --debug               enable the Node.js compatible script debugger
  --enable-jit          enable JIT
  -l SCRIPT, --load=SCRIPT
                        load SCRIPT
  -c CODESHARE_URI, --codeshare=CODESHARE_URI
                        load CODESHARE_URI
  -e CODE, --eval=CODE  evaluate CODE
  -q                    quiet mode (no prompt) and quit after -l and -e
  --no-pause            automatically start main thread after startup
  -o LOGFILE, --output=LOGFILE
                        output to log file
```

Many options are supported. Most useful arguments:

-f ➔ (re)starts the application
-p ➔ attaches to running application
-l ➔ Loads script
-U ➔ Connect to USB device
--no-pause ➔Resume app after launch

73

Frida CLI: Startup Options

There are many different ways to use Frida. For Android and iOS instrumentation, the following options are of interest:

- **--usb (-u):** This will always be necessary if your target is on a connected USB device. Without this flag, Frida will target applications running on the host system.
- **--file (-f):** The package name of the application to be instrumented. When using this option, the application will be (re)launched automatically.
- **--no-pause:** By default, Frida will pause the execution of the application until the %resume command is entered. This allows you to enter your scripts on the CLI before any code is executed. This flag will automatically continue code execution without pausing.
- **--load (-l):** The path of the script that will be loaded before the application execution is resumed.
- **--attach-pid (-p):** Attach to a running process. This is practical if you want to attach to a running process that shouldn't be restarted, such as a system process.

FRIDA CLI: INJECTING INTO TELEGRAM

```
> frida -Uf org.telegram.messenger
...
Spawned `org.telegram.messenger`. Use %resume to let the main thread start executing!
[LGE Nexus 5::org.telegram.messenger]-> %resume
```

Resume the startup process

```
[LGE Nexus 5::org.telegram.messenger]-> Process.arch
"arm"
[LGE Nexus 5::org.telegram.messenger]-> Process.enumerateModulesSync()[6]
{
    "base": "0xb5a09000",
    "name": "libandroid_runtime.so",
    "path": "/system/lib/libandroid_runtime.so",
    "size": 1019904
}
[LGE Nexus 5::org.telegram.messenger]->
```

Interact with the Process interface to print the architecture and loaded modules

Frida CLI: Injecting into Telegram

In this example, Frida is injected into the Telegram app on a Nexus 5 device.

First, the application is resumed so that it finishes its startup process. If this is not done within a set number of seconds, Android will terminate the application because its startup time was too long.

The code shows two commands that are executed, both using the Process interface. *Process.arch* shows the current architecture of the device, while *Process.enumerateModulesSync()* returns an array of all the native modules that are currently loaded into the process. As an example, the sixth loaded module is printed, which is the *libandroid_runtime.so*. For each module, the base address, the name, the path, and the size are given.

FRIDA:WHAT IT CAN DO

Frida allows you to hook functions to:

- Inspect arguments and return values
- Change implementations

But you can do anything the application can do:

- Interact with other applications
- Create instances and call methods
- Read keychains and local storage

Still limited by permissions requested by app

Frida: What It Can Do

Frida allows injecting and executing of your own scripts in the target application. These scripts can either hook into existing classes and methods and alter the implementation at runtime or perform any other action that the app is capable of. Frida is often used to inspect and log arguments and return values to get an insight into the purpose and internal workings of specific functions without analyzing their full source code. Next to adding logging capabilities, the arguments and return values could be dynamically overwritten to trigger or disable certain flows in the application. For example, a method that verifies the certificate of the remote server could be altered to always return "True" to allow penetration testers to perform a man-in-the-middle attack.

However, Frida is not limited to hooking into existing functions. In fact, it's possible to do anything the hooked application can do. We could inject a script that interacts with other apps by creating new intents and activities. We are able to create new instances and call their methods of existing classes in the target application. It is even possible to read keychain entries related to the target application and consult its data stored in the local storage. In short, the possibilities of Frida are endless.

Do note the custom code is still injected in the target process of the application; therefore, we remain restricted to the permissions requested by the target app. For example, it's not possible to inject a script that allows us to track the device's location if the target app has no permission to perform location tracking.

Frida CLI: Loading Custom Scripts

Since all scripts injected by Frida are executed by the V8 or Duktape engine injected in the target process, interacting with the internals is done via JavaScript. All the JavaScript code that interacts with the Java runtime should be wrapped inside a Java.perform call. The Java.perform guarantees that the current thread is attached to the Java VM and executes the function specified as argument. If the current thread is not linked to the Java VM or the Java VM in unavailable, Frida will fail to execute the injected script.

Injected scripts are automatically reloaded once modifications are made. This makes it straightforward to write scripts in your preferred IDE and test them without reloading the application on changes in the scripts.

The code snippet interacts with the Java and Process interface in order to list the current Android version and the amount of loaded modules.

The following command injects hook.js inside the Telegram app:

```
$ frida -U -l hook.js -n org.telegram.messenger --no-pause
```

In this case, Frida is launched with the "--no-pause" flag. This will force the application to continue code execution without manually entering the %resume command after the script is loaded in the target process.

FRIDA: INSPECTING METHOD ARGUMENTS

```
function myFunction()
{
    var cryptoClass = Java.use("com.vault.util.Crypto");

    cryptoClass.setKey.implementation = function(key)
    {
        console.log("Key: " + key);
        this.setKey(key);

    }
}
Java.perform(myFunction);
```

Don't forget to call the original function

```
package com.vault.util;

public class Crypto
{
    public String secretKey;
    public void setKey(String key)
    {
        this.secretKey = key;
    }
}
```


Frida: Inspecting Method Arguments

Frida is capable of interacting with classes and methods loaded inside the target process. In some cases it could be particularly useful to override existing methods to add additional logging capabilities or change the behavior of the application by altering the passed arguments or the return value.

In the following example, we will add logging capabilities to a crypto class in order to reveal a key used for encryption/decryption in the application. The code on the right shows the target method that we wish to override: "setKey".

"setKey" is a method of the "Crypto" class inside the "com.vault.util" package. Note that this code could be obtained by decompiling the Android application using apktool and dex2jar in case you are targeting the app from a black-box perspective.

The injected script will first create a wrapper for the "Crypto" class inside the "com.vaul.util" package using "`Java.use`". In order to replace a method implementation of this class, we can select the relevant method ("`setKey`" in this case) and override the implementation method with our own code. Here we "extend" the method with a line to log the key passed as argument. Afterwards, the original method is called to not perturb the application and proceed with the intended behavior.

FRIDA: INSPECTING RETURN VALUES

```
function myFunction()
{
    var cryptoClass = Java.use("com.vault.util.Crypto");

    cryptoClass.decrypt.implementation = function(data)
    {
        console.log("Decrypting: " + data);
        var returnValue = this.decrypt(data);
        console.log("Decrypted: " + returnValue);
        return returnValue;


    }
}
Java.perform(myFunction);
```

Don't forget to return the actual result

```
package com.vault.util;

public class Crypto
{
    private String decryptedString;
    public String decrypt(String data)
    {
        // Long and complex decryption algorithm
        return decryptedString;
    }
}
```


Frida: Inspecting Return Values

While the previous example logged the argument(s) passed to a specific method, it is also possible to log the return values of a method.

The setup of the injected script is exactly the same: We write a function that creates a wrapper for the “Crypto” class inside the “com.vaul.util” package and override the implementation of the “setKey” method. Afterwards, the function is passed to Java.perform to execute it in the target process.

In this case, we first call the original method and save the return value in the “returnValue” parameter. This allows us to log the return value and return it afterwards to not break the intended flow of the application.

Logging the return value could be useful to quickly reveal the outcome of a complex function or algorithm without statically analyzing the code.

FRIDA: MODIFYING ARGUMENTS (1)

```
function myFunction()
{
    var cryptoClass = Java.use("com.vault.util.Crypto");

    cryptoClass.enableSecurity.implementation = function(data)
    {


        data = false;
        this.enableSecurity(data);
        console.log("Security disabled");
    }
}
Java.perform(myFunction);
```

Simply give other arguments to the original function

```
public class Crypto
{
    public boolean securityEnabled;
    public void enableSecurity(boolean enabled)
    {
        this.securityEnabled = enabled;
    }
}
```


Frida: Modifying Arguments (1)

While the previous scripts did not alter the behavior of the application but just added logging capabilities, it is also possible to alter the arguments passed to a specific method or change its return value. This could be used to bypass security features like SSL pinning or root/jailbreak detection if they are poorly implemented.

Again, the setup is exactly the same compared to the two previous examples. From the original source code, we can deduct that the “enableSecurity” is a setter method expecting a Boolean value. In order to always set “securityEnabled” to false (and possibly disable additional security features), we can override the method by changing the Boolean argument and calling the original method afterwards. This guarantees the app will continue its normal behavior other than that security features are disabled that are dependent on the “securityEnabled” variable.

FRIDA: MODIFYING ARGUMENTS (2)

```
// Java.perform and cryptoClass declaration removed for brevity
cryptoClass.getSecurity.implementation = function()
{
    var sec = this.getSecurity();
    console.log("Security modified from " + sec + " to " + !sec);
    sec = !sec;
    return sec
}
```

Toggle the return value

```
public class Crypto
{
    public boolean securityEnabled;
    public boolean getSecurity()
    {
        return this.securityEnabled;
    }
}
```

```
cryptoClass.getSecurity.implementation = function()
{
    return false;


}
```

Or just always return false, but beware of side effects

Frida: Modifying Arguments (2)

Instead of changing the setter to always return "false" whenever the app calls the setter method, we could alter the getter instead. Imagine the "securityEnabled" variable is set to true by default and could only be switched by an authorized user. In this case, it's not possible for a regular user to toggle security on and off and trigger the app to call the "enableSecurity" setter method.

A possible solution would be to modify the "`enableSecurity`" method as in the previous example and manually call the method afterwards. While this may work for a simple setter method like "enableSecurity", some more complex methods may be dependent on additional logic that has not been set up when calling the method manually.

Alternatively, we could alter the "`getSecurity`" method and always return the opposite of what's stored in the "`securityEnabled`" variable, as shown in this example. For the purpose of clarity, an additional line of logging is inserted.

In the second code snippet, the "`getSecurity`" method is forced to always return false. Note the original method is not called in this case. However, it's advised to always call the original function and just alter the arguments and/or return value in order to not break the application.
While in this case the "`getSecurity`" getter is just a simple method returning a Boolean variable, the impact should be limited. However, in some cases, the original method introduces some side effects that could be forgotten or left out when the original method is not invoked by the overridden method. Depending on the type of side effects, this could alter the intended behavior or even break the application.

FRIDA: MODIFYING IMPLEMENTATION

```
// Java.perform and cryptoClass declaration removed for brevity
cryptoClass.decrypt.implementation = function(data)
{
    var encStr = ["TmV2ZXI=", "R29ubmE=", "R2l2ZSB5b3U=", "VXA="]
    for(var i = 0; i< encStr.length; i++)
    {
        console.log("Decrypted: " + this.decrypt(encStr[i]));
    }
    return this.decrypt(data);
}
```

Override function to decrypt all our encrypted strings

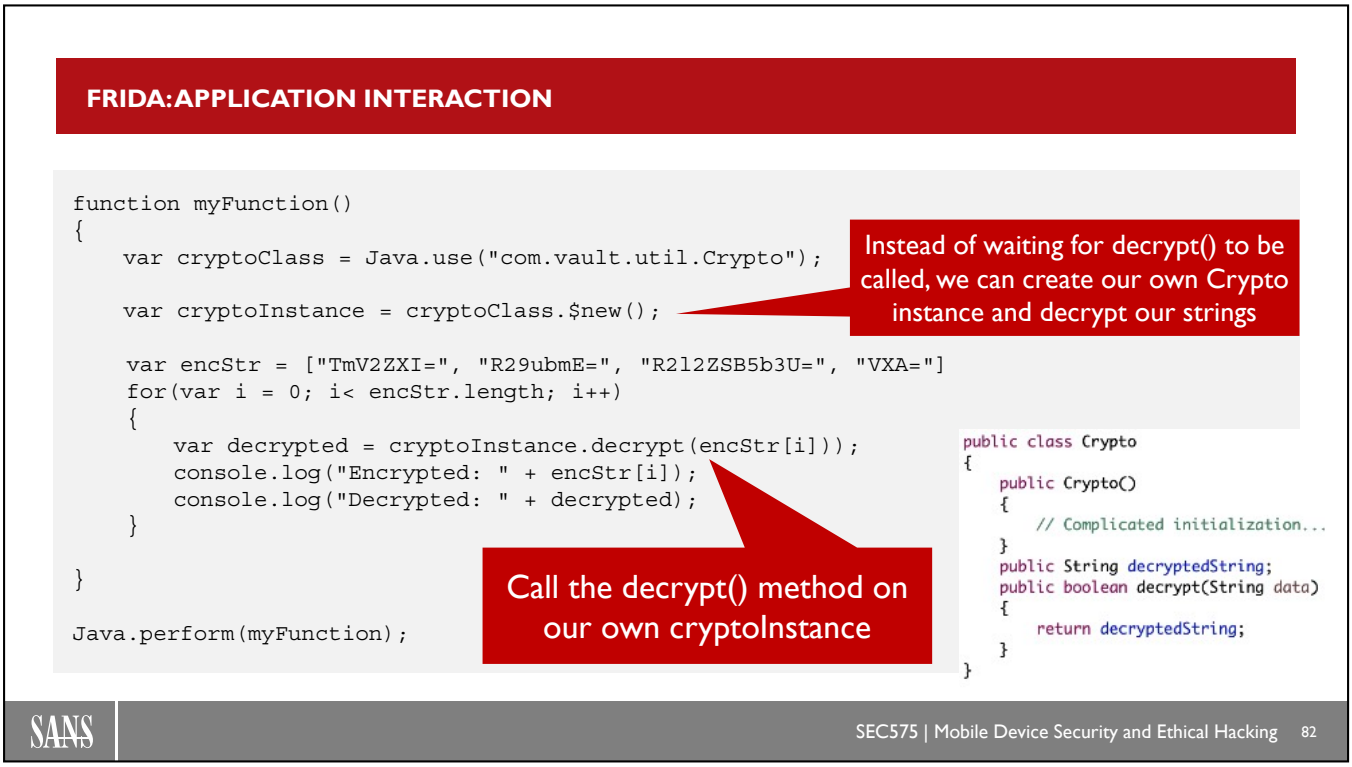
Keep the application running by performing the original call

```
package com.vault.util;

public class Crypto
{
    private String decryptedString;
    public String decrypt(String data)
    {
        // Long and complex decryption algorithm
        return decryptedString;
    }
}
```


Frida: Modifying Implementation

Imagine we discovered encrypted strings in the local storage, but we are unable to decrypt them by statically looking at the code. We could wait for the application to decrypt all the encrypted strings, but that might take a very long time, depending on the logic of the application. Instead, we could alter the implementation of the decrypt method in the "Crypto" class to pass our discovered strings. We override the method by first adding some additional logic to decrypt our strings and call the original method afterwards. Because the app will only call the decrypt function when it's ready to do so (i.e., other possible variables might be initialized for the decryption process), our discovered strings will be decrypted.

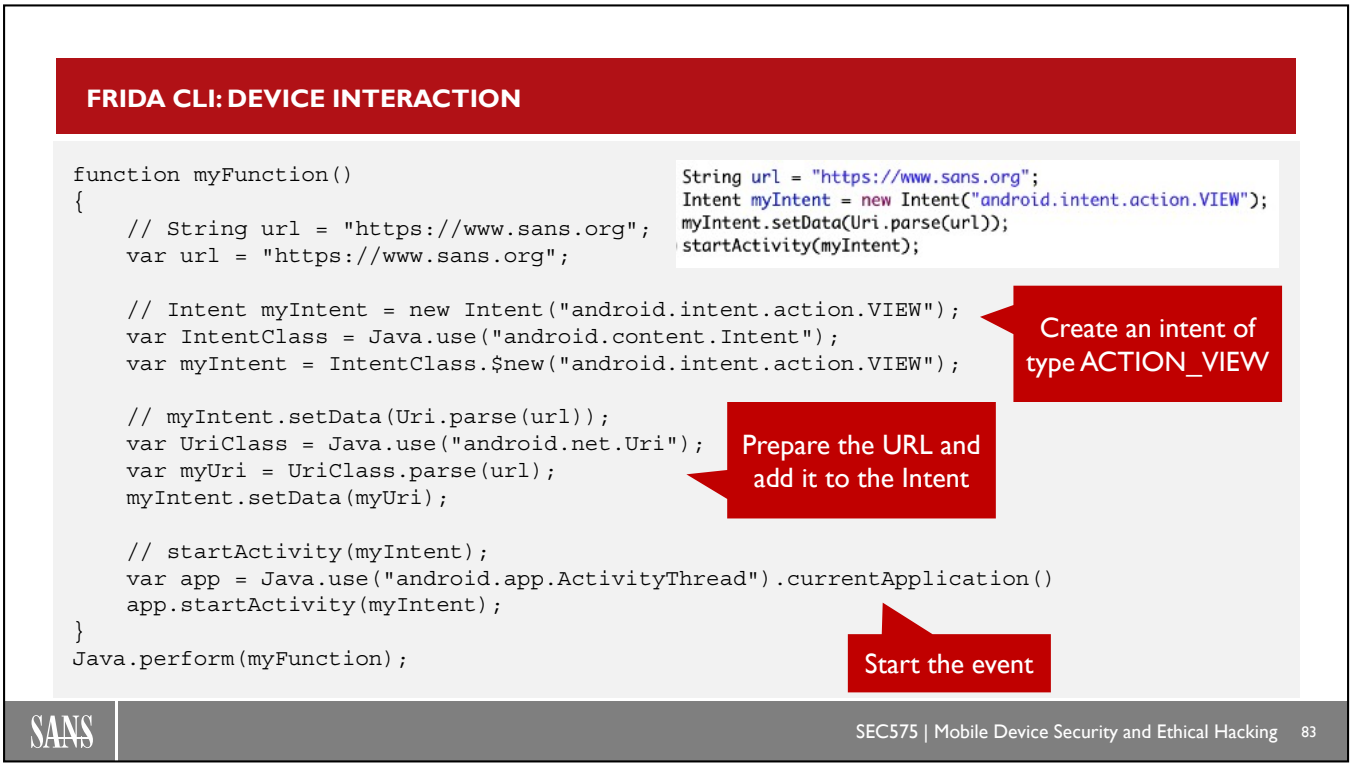
Again, do not forget to call the original method to proceed the expected behavior of the application. In this case, the original data argument passed to the method will not be decrypted and returned when the original method is not called. This will most likely break the application.



Frida: Application Interaction

In the previous example, we have to wait until the application calls the decrypt function just once in order to decrypt all of our string we discovered in the local storage. However, this might still take some time, depending on the logic of the application. Instead of waiting for the application to call the overridden decrypt function, we could create a new "Crypto" instance by calling the "$new" method on the JavaScript wrapper object. "$new" will invoke the original constructor of the class and initialize all the variables required for the decryption process. Afterwards, we add the same logic to decrypt all the discovered strings but call the decrypt method of our own crypto instance.

In this case, we do not have to invoke the original method since this code is executed once the script is injected in the target process and we're not in the normal flow of the application yet.



Frida CLI: Device Interaction

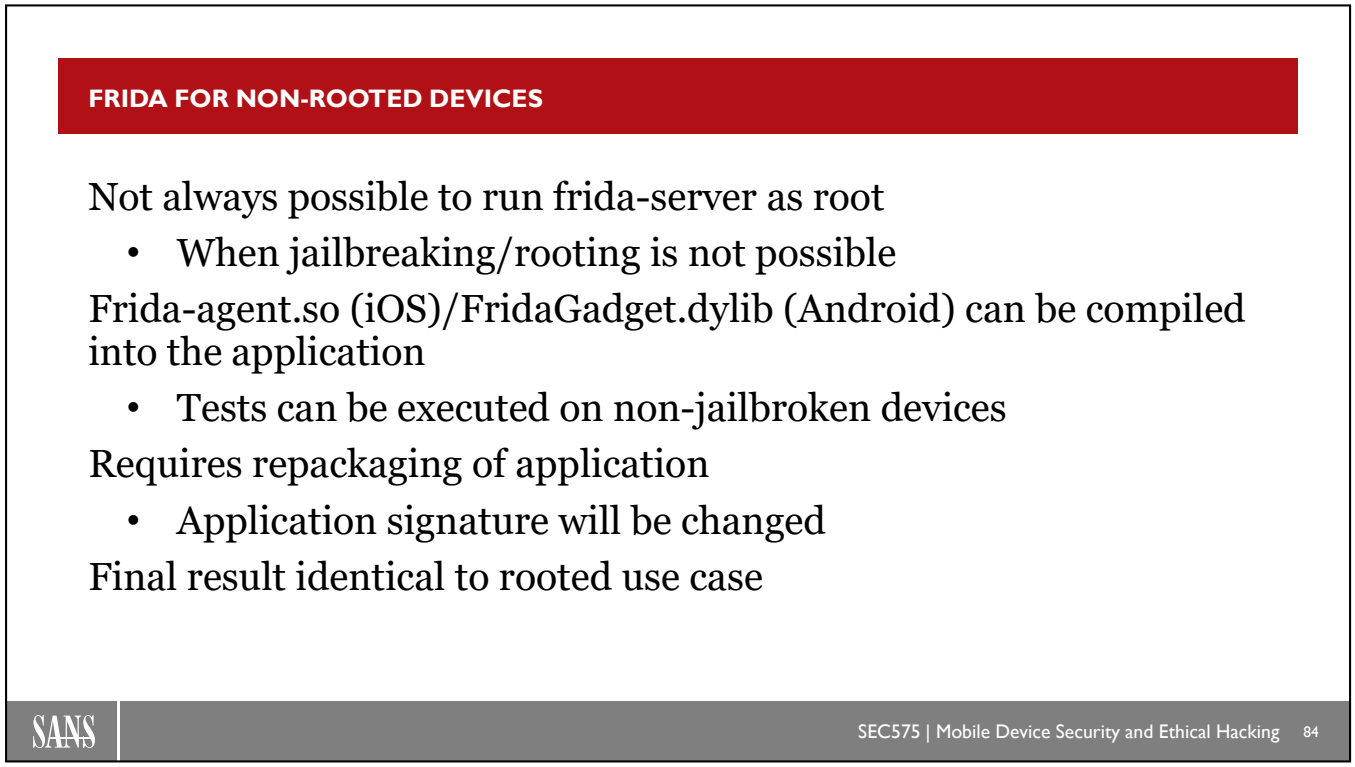
Based on the previous examples, it's clear the possibilities of Frida are endless. In fact, you can do anything the hooked application can (and is allowed to) do.

In this example, we show how to use Frida to browse to a specific URL. The corresponding Java code can be found in the upper right corner. The Java line in question, which is 'translated' to JavaScript, can be found in comments.

First, we declare and set the URL variable to the URL we wish to access. Afterwards, we create a new wrapper around the Intent class and create a new instance of the "VIEW" intent using the "$new" method.

The URL should first be wrapped inside an URI object since the "setData" method of the intent instance expects a URI object. The URI object can be created by constructing an "`android.net.Uri`" class and calling the static "`parse`" method.

To launch an activity with the freshly created intent instance, we create a wrapper around the main ActivityThread class and ask for the "`currentApplication`" object, which encapsulates the whole application. Afterward, the "`startActivity`" method is called to actually open the URL.



Frida for Non-Rooted Devices

It may not always be possible to run frida-server as the root user. An example of this is when an iOS application needs to be tested on the latest iOS version when jailbreaking the device is not possible yet. Still, it's possible to perform dynamic instrumenting using Frida on a non-jailbroken/rooted device. In this case, Frida-agent.so or FridaGadget.dylib, a dynamic library, should be included in the Frida runtime of an iOS or Android application, respectively.

However, the installation process slightly differs from jailbroken/rooted mobile devices. It requires either the help from a developer who can voluntarily include the external library, or by decompiling the application, adding the library and compiling it again.

In order to install and run the Frida gadget on a non-jailbroken/rooted device, the frida-agent.so gadget should be compiled into the targeted application. Therefore, the frida-server is tied to a specific app and it's only possible to inject JavaScript code in this patched app. Note that this is different from a jailbroken/device where the frida-server is injected into memory and thus is able to target any installed application.

To compile the Frida gadget inside the target application, the application should be repackaged and code signed to load the gadget when the app starts. For Android, objection has automated the process as far as possible and the following guide can be used:

- https://github.com/sensepost/objection/wiki/Patching-Android-Applications

For iOS, the process is a bit more complex since the patched app needs a trusted mobile provisioning file containing certificate information and the entitlements groups. The process requires a (free) Apple Developer account and should be performed on the macOS operating system. A clear guide on how to patch an iOS application to inject the Frida gadget can be found here:

- https://github.com/sensepost/objection/wiki/Patching-iOS-Applications

Note this process will change the signature of the application, so additional work may be needed in case the application contains signature verification controls.

Once you are able to install and launch the patched app running the Frida gadget, the full Frida instrumentation toolkit can be used as if it's running on a jailbroken/rooted device.

Course Roadmap

- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 4

Dynamic Mobile Application Analysis and Manipulation
Android Dynamic Analysis with Drozer
Exercise: Manipulating Android Intents
iOS Dynamic Analysis with Needle
Modifying Mobile Applications
Exercise: Modifying Android Applications
Mobile Application Runtime Manipulation
Automated Runtime Manipulation with Objection
Exercise: Frida and Objection
Application Security Verification

This page intentionally left blank.

OBJECTION

Runtime mobile exploration toolkit built on top of Frida
Supports both iOS and Android
Interactive console to interact with filesystem
Implements bypasses for common mobile hardening techniques

- Bypass SSL pinning
- Bypass and simulate rooted/jailbroken environments
- Bypass Touch ID

View and dump platform-specific data structures

- iOS Keychain
- SQLite databases
- .plist files
- Local storage

Objection

As Frida has a steep learning curve, researchers have been creating tools on top of Frida to facilitate some recurring processes when exploring mobile applications. One of the most popular frameworks built on top of Frida for instrumenting mobile applications at runtime is Objection. Objection, coming from the concatenation of "object" and "injection", provides an intuitive GUI for interacting with iOS and Android applications and the filesystem. The framework is heavily dependent on Frida; therefore, all features offered by Objection could be performed by Frida, as well by injecting your own scripts. However, Objection bundles scripts for the most common processes when performing security research or penetration testing of mobile applications and makes is very straightforward to bypass some prevalent mobile hardening techniques.

Objection has built-in scripts for tackling SSL pinning, circumventing rooted/jailbroken device detection and bypassing Touch ID on iOS, and much more. Additionally it provides controls to view and dump some platform-specific data structures for storing data related to the application like the iOS Keychain, SQLite databases, and .plist files with configuration details.

OBJECTION: CONNECTING

Install through pip3

```
> pip3 install objection
```

Run frida-server on target device

Make sure device is connected through USB

Call with package name as gadget

```
$ objection --gadget org.telegram.messenger explore
Using USB device `LGE Nexus 5`
Agent injected and responds ok!

     _   _         _   _
 ___|_|_|_|___ ___| |_|_|___ ___
| . | . | |  _|  _|  _| | . |   |
|___|___| |___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.6.6

[tab] for command suggestions
org.telegram.messenger on (google: 9) [usb] #
```

Start the Objection REPL within the target application

Objection: Connecting

Objection is available as a Python3 package and can be installed using pip3.

> pip3 install objection

Note that it is recommended to install Objection in a virtual environment such that Objection will be installed in an isolated Python virtual environment. This way Objection dependencies will have no effect on any locally installed dependencies within the operating systems module path.

As Objection is using Frida under the hood to inject custom scripts executed by the Duktape/V8 JavaScript engine, it requires a frida-server running on the target mobile device. A bi-directional channel between Objection running on the host machine and the frida-server on the target device will be set up. The frida-server spawns a dedicated thread to listen for incoming connections (by default on port 27042) and accepts instrumentation scripts originating from Objection, which will then be executed by the injected JavaScript engine.

By default, Objection connects to remote devices through USB. However, it is possible to connect over a network's connections by specifying the correct startup options.

The Objection REPL can be started using the "explore" command. The command requires a gadget parameter, which in most cases equals the package name of the target application. The package name of all applications running on the target device connected over USB can be retrieved using frida-ps:

> frida-ps –U

OBJECTION: STARTUP OPTIONS

```
$ objection --help
Usage: objection [OPTIONS] COMMAND [ARGS]...

Options:
  -N, --network            Connect using a network connection instead of USB.
                           [default: False]
  -h, --host TEXT          [default: 127.0.0.1]
  -p, --port INTEGER       [default: 27042]
  -ah, --api-host TEXT     [default: 127.0.0.1]
  -ap, --api-port INTEGER  [default: 8888]
  -g, --gadget TEXT        Name of the Frida Gadget/Process to connect to. [default: Gadget]
  -S, --serial TEXT        A device serial to connect to.
  -d, --debug              Enable debug mode with verbose output.
  --help                   Show this message and exit.

Commands:
  api          Start the objection API server in headless mode.
  device-type  Get information about an attached device.
  explore      Start the objection exploration REPL.
  patchapk     Patch an APK with the frida-gadget.so.
  patchipa     Patch an IPA with the FridaGadget.dylib.
  run          Run a single objection command.
  version      Prints the current version and exists.
```


Objection: Startup Options

Some interesting startup options are the following:

- **--network (-N):** It is possible to connect to the target device over a network. In this case, this flag should be specified alongside the correct host and port flag. By default, Objection connects to remote devices over USB.
- **--gadget (-g):** The package name of the application to be instrumented. This can be found using the frida-ps command.
- **--serial (-S):** If multiple devices are connected, the proper device can be selected using this flag.

As of version 1.5.0 launched in February 2019, Objection offers an API service where interaction with the target app is possible by sending HTTP requests containing JavaScript code. The main functionality of Objection is encapsulated in the explore command. Explore starts a Read-Eval-Print-Loop (REPL), accepting and executing objection-specific commands.

OBJECTION: RECONNAISSANCE

```
$ objection -g org.telegram.messenger explore
Using USB device `LGE Nexus 5`
Agent injected and responds ok!
     _   _         _   _
 ___| |_|_|___ ___| |_|_|___ ___
| . | . | | -_|  _|  _| | . |   |
|___|___| |___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.6.6

[tab] for command suggestions
org.telegram.messenger on (google: 9) [usb] # env

Name                    Path
----------------------  ------------------------------------------------------------
cacheDirectory          /data/user/0/org.telegram.messenger/cache
codeCacheDirectory      /data/user/0/org.telegram.messenger/code_cache
externalCacheDirectory  /storage/emulated/0/Android/data/org.telegram.messenger/cache
filesDirectory          /data/user/0/org.telegram.messenger/files
obbDir                  /storage/emulated/0/Android/obb/org.telegram.messenger
packageCodePath         /data/app/org.telegram.messenger-1/base.apk
```

Start the Objection REPL within the target application

List interesting directories related to the application

89

Objection: Reconnaissance

After successful connection with the Frida agent running in the target process using the explore command, Objection displays the target device and application and starts the REPL. In the example shown, we are connecting to a Telegram app running on an LGE Nexus 5 device.

At this stage, commands can be entered into the REPL as needed. The Objection REPL offers a whole list of custom commands, which facilitates instrumentation of mobile applications at runtime. To make things easier, the REPL offers auto-completion and entering a `[tab]` will show a list of command suggestions.

For example, the “env” command can be used to enumerate interesting directories that relate to the application in question. It will print out the locations of the applications Library, Caches, and Documents directories. From here on, the security researcher can search for interesting data related to the application.

OBJECTION: RECONNAISSANCE

```
org.telegram.messenger on (google: 9) [usb] # pwd print
Current directory: /data/user/0/org.telegram.messenger/files
org.telegram.messenger on (google: 9) [usb] # ls ..
Type       Last Modified            Read    Write    Hidden    Size
---------  -----------------------  ------  -------  --------  -------
Directory  2019-06-21 09:25:17 GMT  True    True     False     4.0 KiB  cache
Directory  2019-06-21 09:25:17 GMT  True    True     False     4.0 KiB  code_cache
Directory  2019-06-21 09:25:17 GMT  True    False    False     4.0 KiB  lib
Directory  2019-06-21 10:39:57 GMT  True    True     False     4.0 KiB  shared_prefs
Directory  2019-06-21 09:35:36 GMT  True    True     False     4.0 KiB  no_backup
Directory  2019-06-21 11:58:39 GMT  True    True     False     4.0 KiB  files

Readable: True  Writable: True

org.telegram.messenger on (google: 9) [usb] # !cat /etc/passwd
Running OS command: cat /etc/passwd

root:*:0:0:System Administrator:/var/root:/bin/sh
...
```

Unix-based interaction with the filesystem

Execute commands on host OS

90

Objection: Reconnaissance

The Objection REPL implements some well-known Unix commands like “pwd”, “cd”, “ls”, and “rm” to interact with the mobile system. By default, the objection REPL will start in your applications main bundle path. In the example, “pwd” and “ls” are used to print the path of the current working directory and list the files. It’s also possible to quickly run commands on the host OS itself by prefixing the command with a “!”.

OBJECTION: INTERACTING WITH TELEGRAM

```
org.telegram.messenger on (google: 9) [usb] # android hooking list activities
org.telegram.messenger.GoogleVoiceClientActivity
org.telegram.messenger.OpenChatReceiver
org.telegram.ui.ExternalActionActivity
org.telegram.ui.IntroActivity
org.telegram.ui.LaunchActivity
org.telegram.ui.PopupNotificationActivity...
...
Found 18 classes

org.telegram.messenger on (google: 9) [usb] # android intent launch_activity
org.telegram.ui.IntroActivity
(agent) Starting activity org.telegram.ui.IntroActivity...
(agent) Activity successfully asked to start.
```

Search activities registered in the app

Invoke activities

91

Objection: Interacting with Telegram

While the previous slide showed some OS independent commands to interact with the remote filesystem, commands with more specific purposes are prefixed with either “android” or “ios”.

In the first example, we enumerate all the activities registered in Telegram application on Android. Any of these activities can be launched by supplying it as an argument to the “android intent launch_activity” command. In the following example, we are manually launching the Telegram IntroActivity, which displays the introduction screen on the target device.

OBJECTION: EASY HOOKING (1)

```
org.telegram.messenger on (google: 9) [usb] # android hooking search classes login
org.telegram.ui.LoginActivity
org.telegram.ui.LoginActivity$1
org.telegram.ui.LoginActivity$2
org.telegram.ui.LoginActivity$3
...
Found 37 classes
```

Search classes related to login

```
org.telegram.messenger on (google: 9) [usb] # android hooking list class_methods
org.telegram.ui.LoginActivity
private void org.telegram.ui.LoginActivity.clearCurrentState()
private void org.telegram.ui.LoginActivity.needFinishActivity()
private void
org.telegram.ui.LoginActivity.needShowAlert(java.lang.String,java.lang.String)
private void org.telegram.ui.LoginActivity.needShowProgress(int)
...
Found 65 method(s)
```

List class methods

92

Objection: Easy Hooking (1)

We discussed hooking into specific classes and methods to inspect and change arguments and/or return values using Frida. In order to achieve this, we had to write our own JavaScript code snippets that selected the correct class and override the implementation of the class method in question.

The same result can be achieved using Objection. Objection encapsulates this functionality in the "ios hooking" and "android hooking" commands for iOS and Android, respectively. In the example, we search for a class related to login by leveraging the search functionality of the hooking library. It's possible to either search in the list of registered classes or else directly in all the methods of each class registered in the application.

Once we find the proper class, we can list all the methods declared in the class by using the "list class_methods" functionality. Here we can see the "`org.telegram.ui.LoginActivity`" class implements 65 different methods. For brevity reasons, only two are displayed.

OBJECTION: EASY HOOKING (2)

```
org.telegram.messenger on (google: 9) [usb] # android hooking watch class_method
org.telegram.ui.LoginActivity.needFinishActivity

(agent) Attempting to watch class org.telegram.ui.LoginActivity and method
needFinishActivity.
(agent) Hooking org.telegram.ui.LoginActivity.needFinishActivity()
(agent) Registering job dwyv48fajb9. Type: watch-method for:
org.telegram.ui.LoginActivity.needFinishActivity
```

Watch for specific methods to be triggered by the application

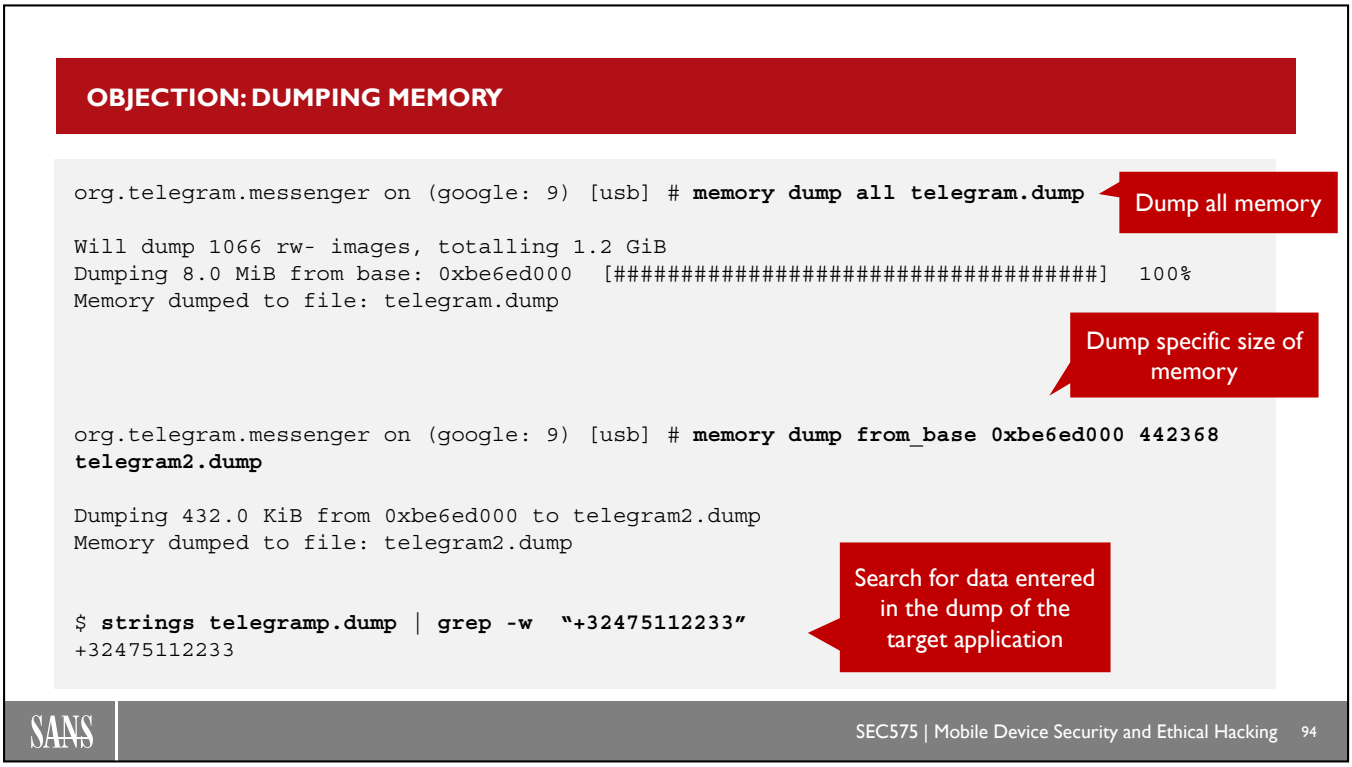
```
org.telegram.messenger on (google: 9) [usb] # android hooking set return_value
org.telegram.ui.LoginActivity.needFinishActivity true

(agent) Attempting to modify return value for class org.telegram.ui.LoginActivity
        and method needFinishActivity.
(agent) Hooking org.telegram.ui.LoginActivity.needFinishActivity()
(agent) Registering job pgp8u6ge5bs. Type: set-return for:
        org.telegram.ui.LoginActivity.needFinishActivity
```

Change the return value of a specific method to a Boolean value

93

Objection: Easy Hooking (2)

Now we are able to set the return value of any of the discovered methods at runtime using the "`android hooking set return_value`" command and specifying the method and return value as arguments. Currently, Objection only supports setting return values to a Boolean value. However, if more advanced hooking setups are required, Frida is the preferred solution



Objection: Dumping Memory

Objection makes it straightforward to dump the current processes' memory using the "memory dump" command. Either all memory can be dumped that is marked as readable and writable (rw-) to a file specified by local destination or else memory can be dumped starting from a base address. To dump from a specific base address, a number of bytes should be specified. For example, for addresses and sizes, the "memory list modules" command may be used. Do note specifying addresses or sizes that are outside of the current processes' sandbox have a high chance of crashing the application since it usually does not have permission to read outside of the sandbox.

In the first example, all memory of the Telegram process is dumped, while the second example dumps the memory starting from address 0xbe6ed000 for a size of 442368 bytes.

OBJECTION: BYPASSING SECURITY CONTROLS

```
org.telegram.messenger on (google: 9) [usb] # android sslpinning disable
(agent) Custom TrustManager ready, overriding SSLContext.init()
(agent) Found com.android.org.conscrypt.TrustManagerImpl, overriding
TrustManagerImpl.verifyChain()
(agent) Found com.android.org.conscrypt.TrustManagerImpl, overriding
TrustManagerImpl.checkTrustedRecursive()
(agent) Registering job oumj12pdfw. Type: android-sslpinning-disable
```

Bypass SSL pinning

```
org.telegram.messenger on (google: 9) [usb] # android root disable
(agent) Registering job m4jpk7zntri. Type: root-detection-disable

org.telegram.messenger on (google: 9) [usb] # android root simulate
(agent) Registering job gxjf29fhjy. Type: root-detection-enable
```

Bypass root detection or simulate a rooted environment

Objection: Bypassing Security Controls

Objection has built-in scripts to bypass some prevalent security controls, such as SSL pinning, root detection, and Touch ID. Just like all the other functionalities in Objection, these are based on custom hooks injected by Frida. While these scripts are designed to work in many different environments and implementations, it might happen that the hooks fail to succeed due to specific implementations of these security controls.

In the first example, SSL pinning is disabled by overriding the application-specific TrustManager, which is able to only accept connections from trusted servers, to an empty TrustManager

The second example attempts to disable root detection or simulate a rooted environment. Under the hood, this is achieved using the following techniques:

- Test whether the build tags contain “test-keys”. If this is the case, the build was signed with a custom key generated by a third-party developer instead of the official key from an official developer. This may indicate the device is rooted.
- Verify whether it is possible to run the “su” command. Running the “su” command without a username will automatically gain you root privileges if the device is rooted. On non-rooted devices, the “su” binary is not available.
- Search the filesystem for specific files and binaries that are linked to rooted devices. This includes, for example, the “su”, “busybox”, and “daemonsu” binaries.

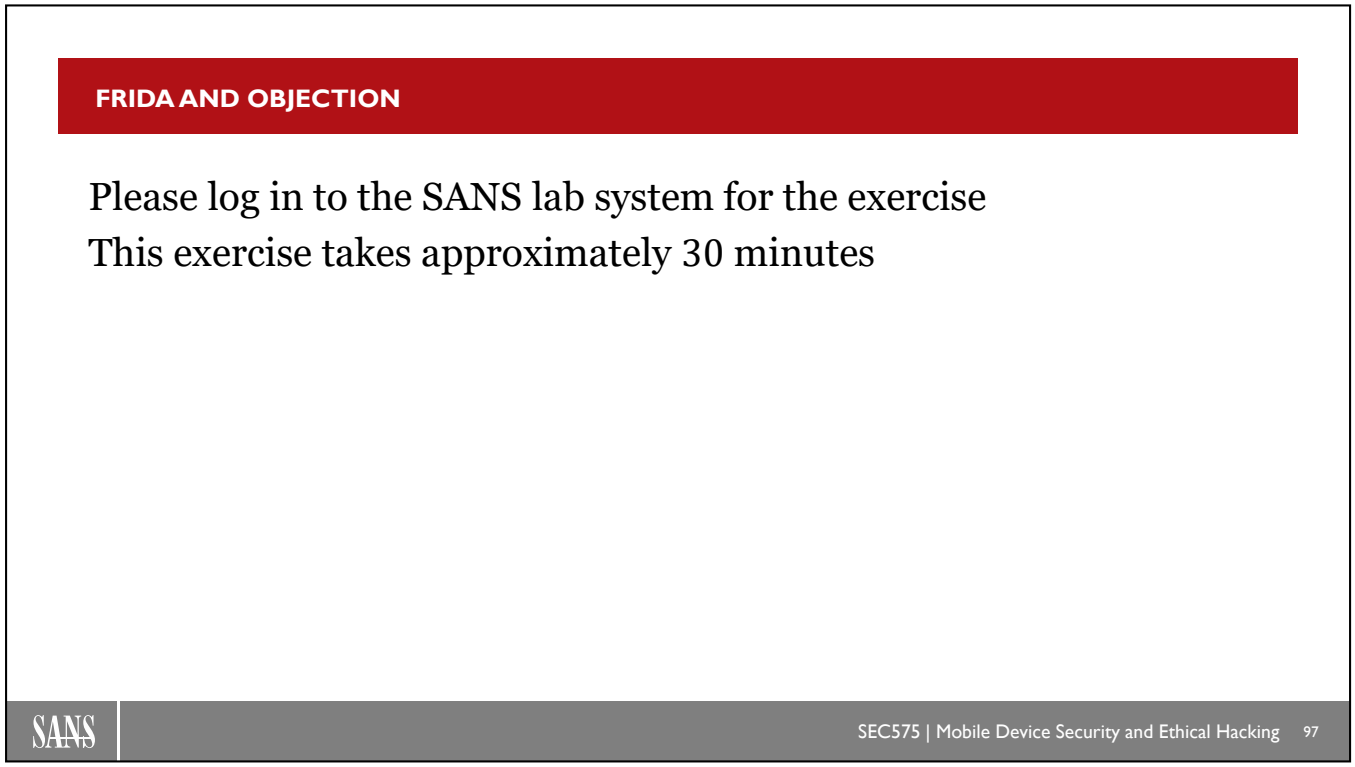
Course Roadmap

- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 4

Dynamic Mobile Application Analysis and Manipulation
Android Dynamic Analysis with Drozer
Exercise: Manipulating Android Intents
iOS Dynamic Analysis with Needle
Modifying Mobile Applications
Exercise: Modifying Android Applications
Mobile Application Runtime Manipulation
Automated Runtime Manipulation with Objection
Exercise: Frida and Objection
Application Security Verification

This page intentionally left blank.



Frida and Objection

Please log in to the SANS lab system for the Frida and Objection exercise. This exercise takes approximately 30 minutes to complete.

Course Roadmap

- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 4

Dynamic Mobile Application Analysis and Manipulation
Android Dynamic Analysis with Drozer
Exercise: Manipulating Android Intents
iOS Dynamic Analysis with Needle
Modifying Mobile Applications
Exercise: Modifying Android Applications
Mobile Application Runtime Manipulation
Automated Runtime Manipulation with Objection
Exercise: Frida and Objection
Application Security Verification

This page intentionally left blank.

APPLICATION SECURITY VERIFICATION

For onboarding applications, a quick test may be enough

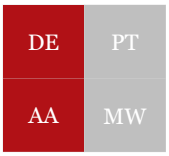
If you are responsible for ensuring your customer creates a secure application, a full audit is required

Audits are a mix of dynamic and static analysis

- Sometimes dynamic is easier, sometimes static

Audits are time intensive

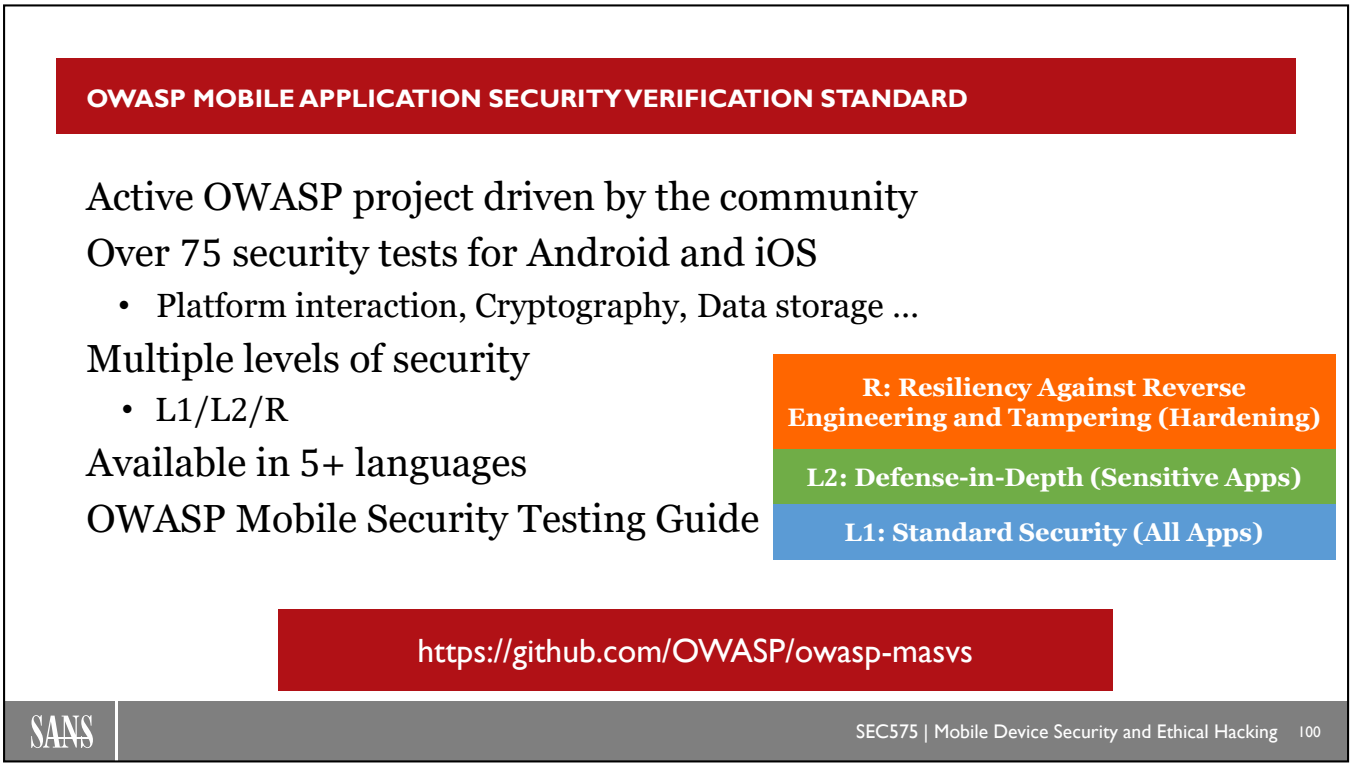
- Use tools to speed up, but know what the tools are doing



Application Security Verification

If you need to evaluate the security of an application that is going to be introduced in your company or organization, a quick test might be sufficient. However, if you are responsible for verifying that an application created by your customer is secure, you must perform a full security audit.

Security audits usually consist of a mix between dynamic and static analysis. Depending on the application tested and the environment, dynamic analysis might be easier to perform than static analysis, or vice versa. In any case, full audits are time-intensive procedures and in order to perform them in a reasonable amount of time, you can use tools to speed up your work—provided that you know how to use them and, more importantly, how to interpret their results.



OWASP Mobile Application Security Verification Standard

The Mobile Application Security Verification Standard (MASVS) is an OWASP project, actively developed and maintained by members of the security community (this course's author is a co-author of the project). It aims to be a comprehensive list of security controls that must be implemented by applications in order to achieve an acceptable level of security.

The MASVS is constantly being updated and its current version contains over 75 security tests for both the Android and iOS platforms. The tests are organized in eight categories, including for example:

- **Platform interaction:** concerns the secure use of platform APIs and components
- **Cryptography:** ensures the application follows the best cryptographic practices
- **Data storage:** verifies that sensitive data is adequately protected

The tests are also associated with three security levels:

- **Level 1:** tests mandatory for every application
- **Level 2:** tests mandatory for applications that require a high level of security, for example, mobile banking applications
- **Level R:** tests related to application hardening and anti-reverse engineering, such as root detection

The MASVS has been translated into more than five languages, including, for example, Japanese, French, and German. It is also supported by OWASP's Mobile Security Testing Guide (MSTG), available here (https://github.com/OWASP/owasp-mstg), which contains detailed technical explanations and instructions for performing the MASVS checks.

APP REPORT CARDS

A full security assessment takes multiple days to complete, depending on

- Size of the application
- Security verification level
- Used technology and framework

App Report Cards were developed before community standards were available and contain a basic security assessment

We will evaluate two apps based on the App Report Cards

App report cards help to make analysis and reporting consistent

App Report Cards

A full security assessment can take quite some days to complete. The MASVS is very elaborate and can take a lot of time to properly complete. The time it takes to perform a full analysis depends on multiple aspects:

- The size of the application: A small coupon app will have a lot less functionality than a banking app
- Security Verification Level: A coupon app will require a Level 1 assessment, a banking app will require a Level 2 verification, and a mobile game may require extensive application hardening and reverse engineering protection
- Used technology and framework: Native applications are typically the easiest and most supported by different tools. However, third-party frameworks exist, such as Xamarin, PhoneGap, React Native, Flutter … These can make tests go very slowly, since they all require a custom workflow

One of the authors of this course has developed App Report Cards, even before the MASVS existed. The controls in the App Report Cards are a set of basic requirements that an application should adhere to. Analyzing an application based on these cards can give a very good first impression on the security posture of the applications.

The app report cards are available at https://github.com/joswr1ght/MobileAppReportCard, including completed examples for the applications we evaluate in this module.

In addition to evaluating mobile application security using app report cards, we also examine the techniques that developers can apply to resolve identified flaws. Whether in an application security analyst capacity or as a penetration tester, it is important to provide recommendations to customers on how to address identified threats. For mobile app security, this is often in the form of code examples that can be shared with developers.

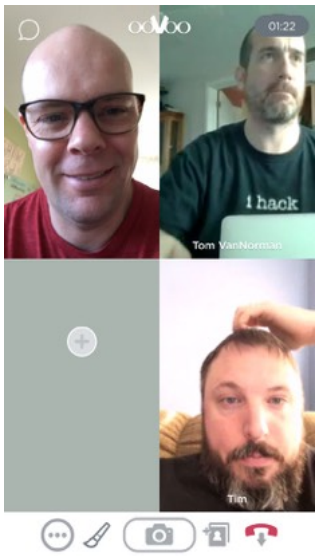
ooVoo

Multiview video, audio, and text chat tool

- Competes with Skype
- Sells international dialing plans

Focuses on high-quality video chat (up to 12 parties)

100 million users



ooVoo

First, we use the app report card to evaluate the security of iOS apps. Our evaluation target is ooVoo for iOS, the multiview video, audio, and text chat tool. ooVoo competes with Skype and, like Skype, offers international dialing plans through its application while generating additional revenue through advertising.

ooVoo focuses on providing high-quality video chat services, supporting up to 12 concurrent video feeds for a single chat. ooVoo claims to support more than 100 million users, with an audience of 65% under 25 (http://www.oovoo.com/fact-sheet/ooVoo_FactSheet_0807V5.pdf).

APP BINARY TESTS

Ensure app is compiled to use PIE
Ensure stack canary protection is used
Ensure app uses ARC memory management

```
# otool -hv ooVoo | grep PIE
   MH_MAGIC     ARM         V7  0x00      EXECUTE    60       6072   NOUNDEFS DYLDLINK
TWOLEVEL WEAK_DEFINES BINDS_TO_WEAK PIE
# otool -I -v ooVoo | grep stack_chk
0x00744ba0  1171 ___stack_chk_fail
0x008f4020  1172 ___stack_chk_guard
0x008f5d20  1171 ___stack_chk_fail
# otool -I -v ooVoo | grep _objc_release
0x007444c0  1605 _objc_release
0x008f5b68  1605 _objc_release
```

App should include PIE flag, stack_chk_guard/stack_chk_fail, and _objc_release symbols

App Binary Tests

The iOS Integrated Development Environment (IDE) Xcode offers several options for developers that can improve the security of their application. Notably, developers should use the following features when building iOS apps:

- **Position Independent Executable**: When enabled, the application loads at a randomly selected memory address when it launches, reducing the likeliness of an attacker predicting memory locations as part of a memory corruption exploit. PIE support is on by default in modern versions of Xcode.
- **Stack Canary**: To validate the integrity of the stack, a "canary" value is placed on the stack before calling a function and is validated again when the function ends. If the developer is still preserving backward compatibility with old versions of iOS (before iOS 4.3), this feature is not included in the binary.
- **Automatic Reference Counting**: Xcode uses ARC by default to let the platform handle memory management and garbage collection, instead of requiring the developer to free memory after use. The use of ARC prevents common memory corruption flaws, such as double-free threats. Note that linking ARC support with Xcode does not necessarily mean that all code uses ARC for memory management; source code auditing is required to further evaluate how a developer is managing memory on the system.

We can manually confirm that apps are built with PIE, stack canary, and ARC support using the otool command, as shown on this slide. Ensure that the app indicates the PIE flag, includes the __stack_chk_guard and __stack_chk_fail, and includes the _objc_release symbols to ensure that PIE, stack protection, and ARC support are linked with the application, respectively.

In the case of ooVoo, shown on this slide, the application is compiled to use PIE and is linked to use stack canary protection and ARC memory management.

IDENTIFY URL HANDLERS

Examine Info.plist file, CFBundleUrlSchemes array

- Needle displays these parameters in the binary/metadata module

Invoking any handler launches app, trigger functionality

```
CFBundleURLTypes = Array {
       Dict {
           CFBundleURLName = com.oovoo.iphone.free
           CFBundleURLSchemes = Array {
               oovoo.special.scheme
               oovoo
```

Evaluating what the app does with these schemes and what URL parameters it uses requires app reverse engineering, a time-consuming task. Identify any behavior you can from using the app or through published SDKs.

Identify URL Handlers

Identify all the custom URL handlers used in the application. Prior to iOS 8, apps could only communicate with each other through custom URL parameters, so many apps retain that functionality. If an app opens the custom URL (or if we, as testers, enter the custom URL in Mobile Safari), the app that registered the handler launches automatically. This is unlike modern iOS extensions (such as the Share extension or the Action extension) that can only be invoked by the user through screen taps.

Applications that register their own custom URL handlers parse the URL path and parameters as needed to take action. For example, the Twitter URL handler "twitter://" expects a URL in the following form to invoke a populated "submit new post" View Controller:

twitter://post?message=hello%20world&in_reply_to_status_id=12345

Some apps may not parse the content of the URL well, leading to possible memory or stack corruption flaws, or even simpler logic flaws in the handling application. Each application should be evaluated to identify all the registered URL handlers, and each handler should be tested to identify the mishandling of malformed URL requests.

We can identify the custom URL handlers in Needle or by inspecting the iOS app Info.plist file. After opening the Info.plist file, investigate the CFBundleUrlSchemes array of elements to identify the list of URL handlers, as shown on this slide.

Identifying the custom URL handlers used by an application is straightforward; however, evaluating the format of how the URL parameter is used is difficult. There is no simple mechanism to evaluate the parameters expected in the target app custom URL handler. It is possible to identify parameters through reverse engineering analysis of the applications (in some cases, using a simple "strings" command or through reverse engineering tools, such as Hopper or IDA Pro), or additional details may be published for some applications in Software Development Kit (SDK) APIs.

For ooVoo, two URL handlers are registered: "oovoo" and "oovoo.special.scheme". The ooVoo SDK does not document these URL handlers, although we can explore the Android version of the ooVoo app (for example, JadX reverse engineering) for additional information on what these URL handlers do, assuming they are similarly used on Android devices. Though not guaranteed to be compatible between iOS and Android versions of the application, the capability to easily retrieve Java source from the ooVoo for Android application makes it easy to quickly evaluate URL handler details, as shown here:

```
$ mkdir ooVoo-android

$ ./JadX/bin/jadx -d ooVoo-android/ com.oovoo-3062.apk

$ grep -R oovoo:// ooVoo-android/

ooVoo-android//com/oovoo/specialcache/ae.java:    private static final String
CONTACT_BASE_URI = "oovoo://localhost/contacts/";

ooVoo-android//com/oovoo/specialcache/ae.java:    private static final String
FACEBOOK_BASE_URI = "oovoo://localhost/facebook/";

$ grep -R CONTACT_BASE_URI ooVoo-android/

ooVoo-android//com/oovoo/specialcache/ae.java:    private static final String
CONTACT_BASE_URI = "oovoo://localhost/contacts/";

ooVoo-android//com/oovoo/specialcache/ae.java:        return CONTACT_BASE_URI
+ Long.toString(phoneUser.getRowContactID());
```

In this example, we create a temporary directory, "oovoo-android", then use the command line version of JadX to convert the ooVoo Android APK file to Java source. This command creates a lot of output that has been removed for space here.

Next, we search for the "oovoo://" URL handler prefix, which returns two string constants, "oovoo://localhost/facebook/" and "oovoo://localhost/contacts/". These string constants are presumably used to invoke ooVoo functionality but require additional parameters. Searching for the string constant name "`CONTACT_BASE_URI`" reveals the earlier reference and a new reference indicating that ooVoo concatenates the "oovoo://localhost/contacts/" URL with the string returned from the `phoneUser.getRowContactID()`. Additional research is needed to identify the value returned by this call and if it can be used on the iOS platform to manipulate the ooVoo application.

EXAMINE ASL MESSAGES

Developers use the Apple System Log to capture event information

Logging information should not contain sensitive information

- Any other iOS app gathers ASL logging messages

Monitor ASL using ondeviceconsole or Xcode

```
# apt-get install com.eswick.ondeviceconsole
# ondeviceconsole | grep -i oovoo
Dec 19 06:33:46 Joshua-Wrights-iPad ooVoo[2451] <Warning>: MOPUB: Banner view
(9f0c8420aef142b5ac5128a7d1a1d392) loading ad with MoPub server URL:
http://ads.mopub.com/m/ad?v=8&udid=ifa:XXXX&id=9f0c8420aef142b5ac5128a7d1a1d392&nv=3.2.0&
q=age_g:5,age:39,gen:m&o=p&sc=1.0&z=-0500&mr=1&dnt=1&ct=2&av=2.2.8&dn=iPad2%2C5
```


Examine ASL Messages

On iOS, any running application can retrieve the contents of the Apple System Log (ASL) without special permissions. A malicious app running on an iOS device could use this permission to retrieve logging information from other applications, potentially gathering sensitive information from the platform. It is not uncommon to see iOS applications disclosing credential information in the ASL, including password and PIN values.

When evaluating iOS apps, examine the contents of the ASL by connecting the iOS device to a Mac OS X system running Xcode. Use the Xcode Device Organizer function to examine the contents of logging entries to identify sensitive information that could be retrieved by other applications.

As an alternative to Xcode, you can install the command line tool ondeviceconsole on a jailbroken iOS device to retrieve logging information, as shown on this slide. In this example, the logging information is filtered with grep for the string "oovoo", revealing an ASL entry corresponding to the advertising network MoPub. The URL being retrieved from the ads.mopub.com server reveals sensitive parameters about this author, including my age and gender. Here, the logging entry reveals two disturbing characteristics: 1. ooVoo is logging sensitive information to the ASL that can be retrieved by other running iOS apps, and 2. ooVoo is sharing personal information gathered from Facebook integration with advertisers.

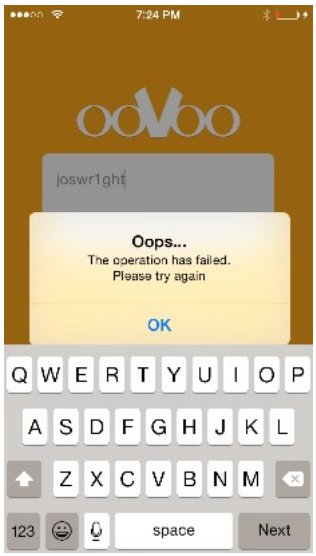
TLS CERTIFICATE HANDLING

How does app respond to invalid certificates?

- Are certificates validated?
- Is the user given the choice to continue despite a certificate error?

Configure network settings to use a proxy server for TLS

- Use app features, before and after authentication, to test certificate handling



TLS Certificate Handling

One of the most important steps in evaluating iOS or Android applications is to evaluate network activity, including how the app handles untrusted TLS certificates. In the best case, the iOS app discontinues network activity when an untrusted certificate in a TLS exchange is identified. In some cases, the app prompts the user to continue or reject the certificate, which is not ideal.

The test procedure is straightforward: Configure the iOS device to use an HTTP proxy server, such as Burp Suite (http://portswigger.net/burp/), and attempt to use the application for a network-sensitive function (such as logging in). The application should reject the certificate without the option to continue, despite the untrusted certificate. Repeat this process for other app functionality because some apps may behave differently when presented with a bad certificate for network authentication than other app activities.

In the example on this slide, ooVoo presents an "Oops..." error to the user when the bad certificate is identified. This is ideal and the default behavior for iOS TLS connections. If the TLS app continues despite the TLS certificate error, or if an option to continue despite the TLS certificate error is displayed, then the developer took specific action to bypass the default fail condition implemented by the TLS handler on iOS.

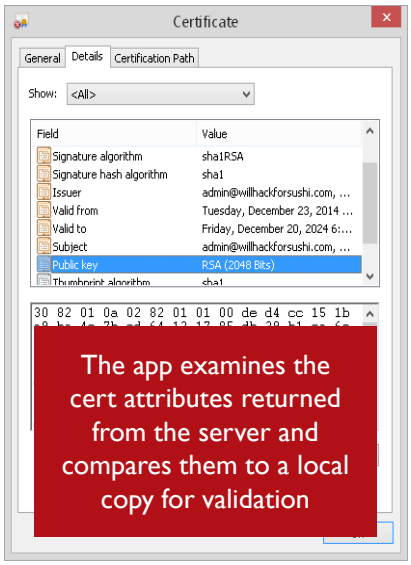
CERTIFICATE PINNING

Websites rely on CA trust for certificate validation

- Web browsers are not designed to support a specific website, use abstract trust

Mobile apps can validate a specific server certificate

- Not relying solely on CA trust (which can be modified locally)
- Instead, by matching a local copy of the server public key to the server copy

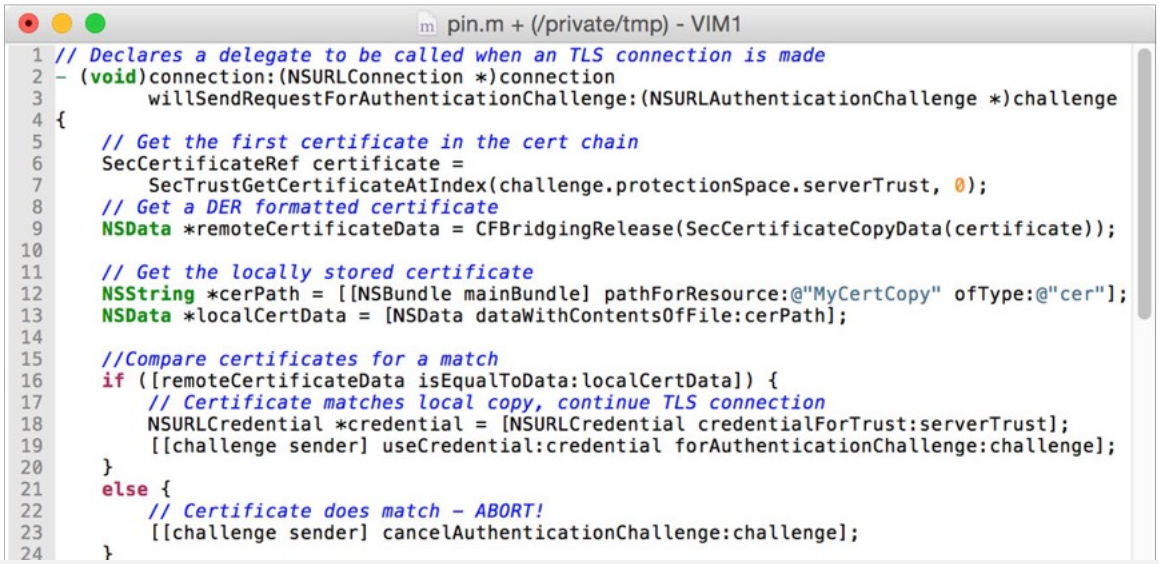


Certificate Pinning

An enhancement to strict TLS validation is the use of certificate pinning. Typically, web browsers visiting a TLS-protected webpage validate the certificate parameters from the TLS website and the certificate parameters of the issuing certificate authority (CA). Web browsers don't have the option to add other logic to certificate processing and validation (although some enhancements to the HTTP protocol are being evaluated to expand certificate validation options).

Mobile applications can apply additional confirmation techniques to validate the TLS certificate. Instead of relying primarily on the trust of the issuing CA, a mobile application can generate a self-signed or privately signed certificate and validate specific parameters in the certificate issued by the application server when the TLS connection is established. For example, the mobile application can maintain a local copy of the app server certificate public key and reject all TLS connections where the exact public key is not received from the server.

This validation process is known as certificate pinning, and it addresses several issues with TLS certificate validation. First, the use of certificate pinning removes the need to trust that any one of several CAs trusted by the platform is not compromised because even a compromised CA issuing certificates that purport to be "www.oovoo.com" do not have the same public key that is validated locally. Furthermore, the CA trust does not protect against a local adversary (or a third-party adversary) who adds a new CA to the trusted list of CA (such as when, as pen testers, we add the Burp Suite CA to the trust on iOS to inspect TLS traffic). Finally, this is a low-cost option as well because it is not necessary to purchase a certificate for use in application certificate pinning from a public CA.



iOS Obj-C code to compare the server-returned certificate to a locally stored resource copy, "MyCertCopy"

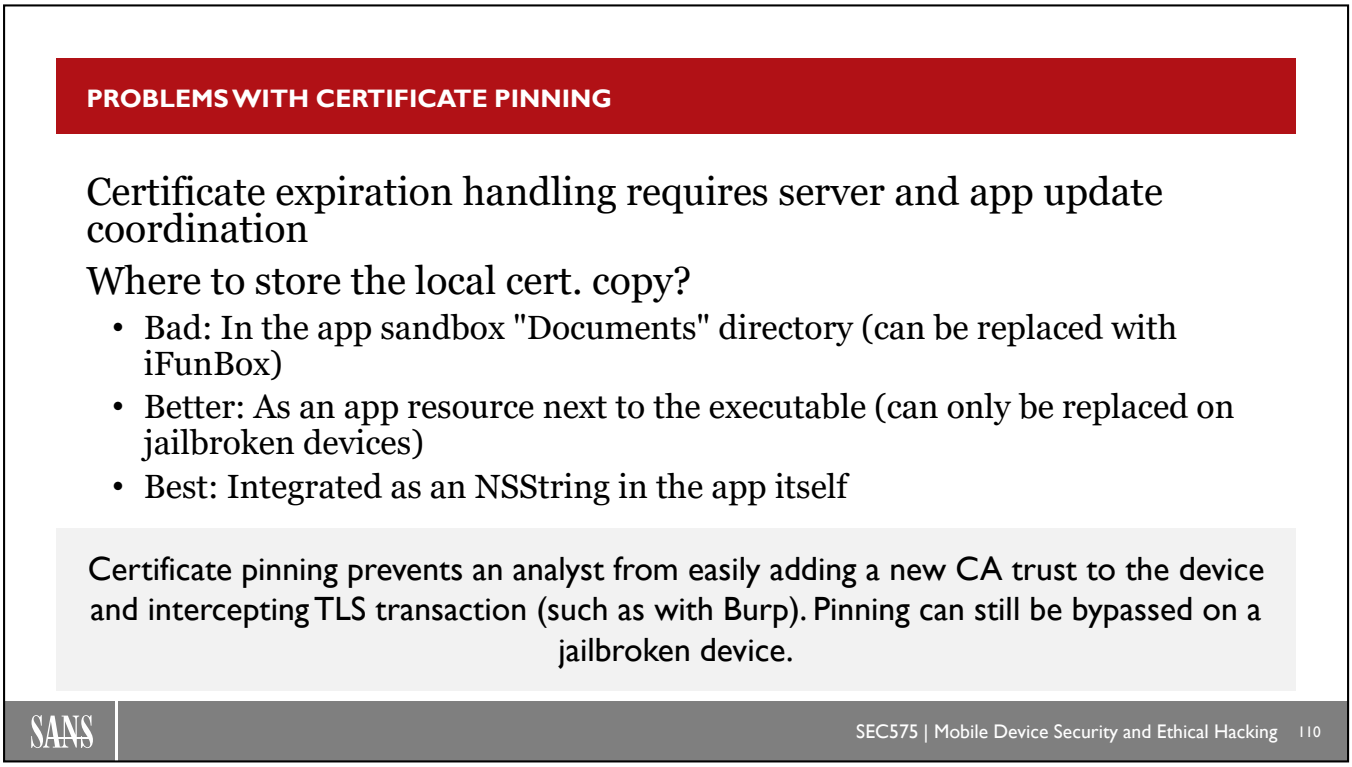
iOS Certificate Pinning

The Objective-C code on this page shows an example of certificate pinning for iOS. In this code, a delegate method is declared that is called each time the application creates a TLS connection using the NSURLConnection class (the default class for HTTP and HTTPS activity).

In lines 6 and 9, the code retrieves the certificate from the server and formats it for the Data Encoding Rules (DER) format. Next, in lines 12 and 13, it reads from a local application resource, "MyCertCopy", representing a local file in DER format (using the common ".cer" filename extension), possibly stored with the other application files in the sandbox Documents directory.

In line 16, the code compares the two certificates to ensure they match. If the certificate from the server matches the certificate stored locally, then the method delegate indicates that the certificate should be used. If the certificates do not match, then the method delegate indicates that the TLS connection should be canceled, allowing the calling method to raise an error or alert the user as desired.

Note that the certificate that is used for validation is stored on the filesystem in this example. This is not a requirement, as we see next, but it has to be stored somewhere on the iOS device for validation against the server certificate.



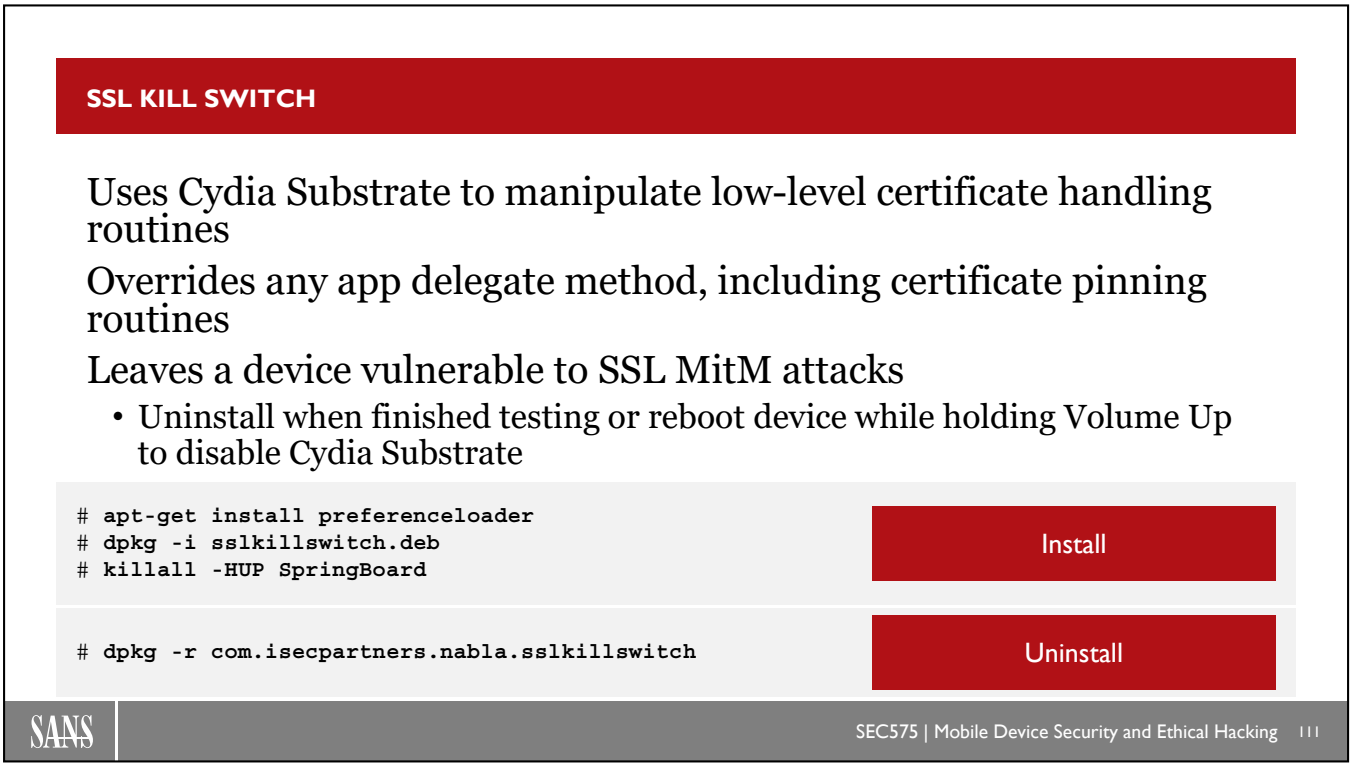
Problems with Certificate Pinning

Certificate pinning has some benefit for the security of an application, but it also introduces new challenges.

Without certificate pinning, the server administrator can change the certificate on the server as often as needed. As long as the certificate is issued by an authority that the iOS device trusts, the certificate can be revoked, reissued, and changed as necessary. For certificate pinning scenarios, changing the server certificate is a more complex process because it requires changes in the mobile application as well. For this reason, pinned certificates usually have a long period of validity before expiration. Changes to the server certificate must be coordinated with the client, usually with an overlapping period of time where one or more certificates are sufficient for the TLS connection.

A secondary issue is the challenge of storing the certificate on the iOS device itself. Three primary techniques are used for storing the certificate for validation:

- **Store the certificate in the app Sandbox "Documents" directory**: This is not a good idea because an attacker can replace any file in the sandbox Documents directory with a different certificate, allowing him to bypass the pinning method.
- **Store the certificate as an application resource**: Storing the certificate as an application resource (e.g., as one of the application files next to the main executable) is preferred because the certificate cannot be replaced through iFunBox on a jailed device. However, on a jailbroken device, any file can be replaced, allowing an attacker to replace the certificate with one of his own choosing.
- **Integrate the certificate into the iOS app code as an NSString object**: Instead of reading the certificate from the filesystem, the certificate content itself can be embedded as an NSString object in the iOS app. This makes it harder for an attacker to replace because it requires manipulation of the iOS app itself. An attacker can still manipulate the iOS app using Cydia Substrate or a binary hex editor to replace the certificate with an alternate copy, but it requires a jailbroken device and a more sophisticated attacker.



SSL Kill Switch

As penetration testers and mobile security analysts, we still need the ability to eavesdrop on TLS connections, even when certificate pinning is in use. Although it is often not possible to bypass certificate pinning on jailed iOS devices, we can install SSL Kill Switch on jailbroken devices to bypass all certificate validation.

SSL Kill Switch is written by Alban Diquet from iSec Partners and is available at https://github.com/iSECPartners/ios-ssl-kill-switch

It uses Cydia Substrate (formerly Mobile Substrate, by Saurik) to manipulate low-level TLS certificate validation routines, always returning successful validation of TLS certificates regardless if certificate pinning or more common certificate validation procedures are used.

SSL Kill Switch requires the preferenceloader package from Cydia and is then installed manually using the example shown on this page. (Alternatively, you can download preferenceloader directly from http://apt.thebigboss.org/repofiles/cydia/debs2.0/preferenceloader_2.2.3-3.deb and then install using dpkg.) After installation, restart SpringBoard.

When SSL Kill Switch is used, all TLS activity is susceptible to MitM attacks, making the iOS device where SSL Kill Switch is installed susceptible to attack. To prevent your test device from being exploited by an attacker, remove SSL Kill Switch when you are finished using it for testing (and then restart SpringBoard) or reboot the iOS device while holding the Volume Up button to disable Cydia Substrate altogether.

TESTING CERTIFICATE PINNING

Import the CA certificate from the proxy server on the iOS device to establish trust

- Does the TLS error persist? If so, pinning is used

Where is the local copy of the certificate used for comparison?

- Examine app sandbox directory for certificates using iFunBox; can it be replaced?

On a jailbroken device, install SSL Kill Switch to disable certificate validation, inspect traffic

ooVoo does not use certificate pinning, which makes it easy to inspect TLS tunnel traffic

Testing Certificate Pinning

Now that we've covered some of the use cases for certificate pinning, let's look at a recommended testing procedure for iOS apps.

After testing to make sure the app rejects invalid certificates and that it does not prompt the user to bypass invalid certificates (as we saw in the last analysis step), import the CA certificate from the proxy server on the iOS device (for Burp Suite, browse to http://burp/cert and tap on the certificate link to install). After the certificate is installed, kill and restart the target iOS application and attempt to use it to access TLS-protected application servers again. If the TLS error persists, then the application uses certificate pinning.

Next, try to identify the location of the local certificate used for pinning. Is it in the application sandbox directory? Can it be replaced through iFunBox on a jailed iOS device?

Finally, on a jailbroken device, use SSL Kill Switch to disable certificate validation altogether to gain access to the TLS traffic.

For ooVoo, no certificate pinning is used.

TRAFFIC ANALYSIS

Inspect network traffic to identify information disclosure threats
- Is all traffic TLS encrypted? For non-TLS traffic, is sensitive data disclosed?
- Who is generating the traffic (app or linked library)?

Capture traffic using rvictl and tcpdump
- Evaluate all protocols and hosts with Wireshark

```
$ rvictl -s 6e141e682db2e389439c1587a61efa9873b9f525
Starting device 6e141e682db2e389439c1587a61efa9873b9f525 [SUCCEEDED]
$ sudo tcpdump -ni rvi0 -s0 -w ios-traffic.pcap
```

Display filter: none

Protocol	% Packets	Packets	% Bytes	Bytes	Mbit/s	End Packets
Transmission Control Protocol	89.70 %	3090	98.24 %	1871074	0.064	2596
XMPP Protocol	1.25 %	43	0.66 %	12524	0.000	43
Secure Sockets Layer	10.33 %	356	7.85 %	149478	0.005	332
Secure Sockets Layer	0.70 %	24	1.19 %	22729	0.001	24
Hypertext Transfer Protocol	2.67 %	92	2.34 %	44506	0.002	44

Here, we learn that ooVoo uses XMPP, SSL/TLS, and HTTP. Inspect the payload content of unencrypted protocols for information disclosure threats.

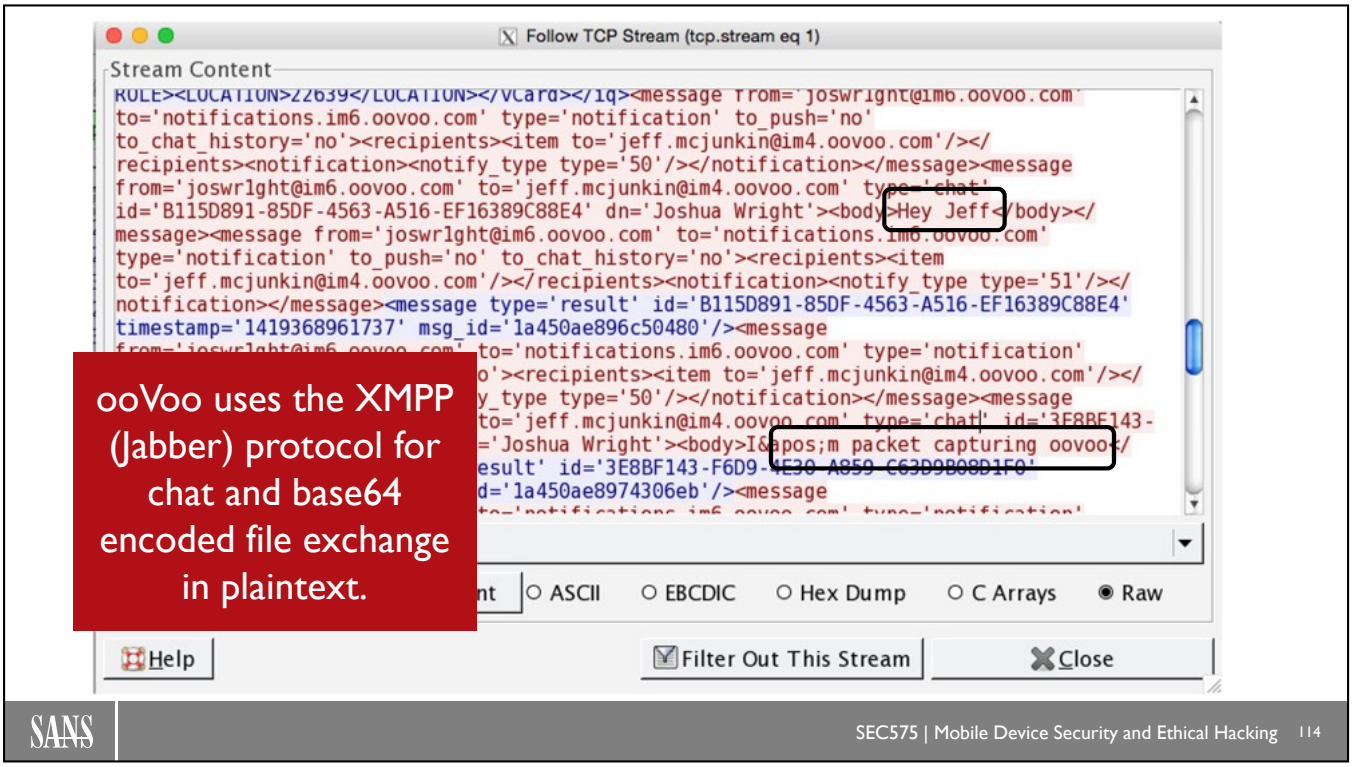
Traffic Analysis

Next, capture the network traffic from the target application and identify some characteristics of the network activity. Enumerate the list of servers, ports, and protocols used by the application. For each connection, identify if the traffic is native to the application or created by a third-party library, such as an ad network. Finally, identify which traffic is encrypted and which is unencrypted. For unencrypted traffic, examine the network activity to identify potentially sensitive information disclosure threats.

Although more difficult to evaluate, it is useful to understand which part of the application is generating the traffic. Identified traffic may look suspicious, but the attribution of that activity should be further evaluated: Is the suspicious traffic generated by the application itself or from any of the linked libraries used by the application? In the case of traffic generated by linked libraries, is the traffic destined to the application publisher or to a third party (such as an ad network provider)?

For iOS, remember that we can use the rvictl utility included with the Xcode command line tools (available through Xcode and the Mac App Store) to force an iOS device to communicate through a Mac for all network activity. We can capture this network traffic using the tcpdump utility, as shown on this slide.

In the case of ooVoo, Wireshark reveals that the protocol uses a combination of HTTP, SSL (HTTPS), and XMPP. XMPP is a messaging protocol popularized by the Jabber chat protocol. The Jabber protocol can use TLS for confidentiality and integrity protection, so it is important to further evaluate the network activity.



ooVoo XMPP Traffic

Evaluating the XMPP protocol activity in Wireshark for ooVoo further reveals that the application does not encrypt chat, file exchange, screen sharing, or video chat conversations.

EVALUATE FILE SYSTEM USE

Kill app, copy all files to "A" directory

Restart app, make a minimal change, kill app, copy all files to "B" directory

Compare two sets, evaluate changes

```
$ diff -rq A B
Files A/Library/Application Support/ooVoo/ooVoo2.sqlite and B/Library/Application
Support/ooVoo/ooVoo2.sqlite differ
Files A/Library/Application Support/ooVoo/ooVoo2.sqlite-shm and B/Library/Application
Support/ooVoo/ooVoo2.sqlite-shm differ
Files A/Library/Application Support/ooVoo/ooVoo2.sqlite-wal and B/Library/Application
Support/ooVoo/ooVoo2.sqlite-wal differ
$ sqlite3 "B/Library/Application Support/ooVoo/ooVoo2.sqlite"
sqlite> select * from ZOOVEVENT;
4|13|1|0|0|4|3||3||1|4|1|||||||||||440693322.055|19850a92ce0da028|Test 2|||Test 2||
13|13|4|0|0|4|4|4|||1|4|1|||||||||||441061761.737|1a450ae896c50480|Hey Jeff|||Hey Jeff||
14|13|4|0|0|4|4|4|||1|4|1|||||||||||441061769.996|1a450ae8974306eb|I'm packet capturing
oovoo|||I'm packet capturing oovoo||
```


Evaluate File System Use

Next, evaluate the application to identify how it is recording data to the filesystem. One option is to use the application normally (login, access resources, and more) and then manually evaluate all the files in the filesystem. This is a valuable analysis task, but it can also be time-consuming, depending on the number of resources that need to be evaluated.

A second option is to identify changes in the filesystem following the completion of specific tasks, such as logging in for the first time. After installing the application and running it for the first time (but without sharing sensitive information), kill the application. Use iFunBox or iExplorer to copy all the files in the app sandbox directory to the filesystem as the "base" data set. (In this example, the author used the directory "A" to store all the files.) Next, start the application again and log in but without excessive application use after logging in to minimize the number of files written to the filesystem that have to be evaluated. Kill the application again and copy the files to the "B" data set.

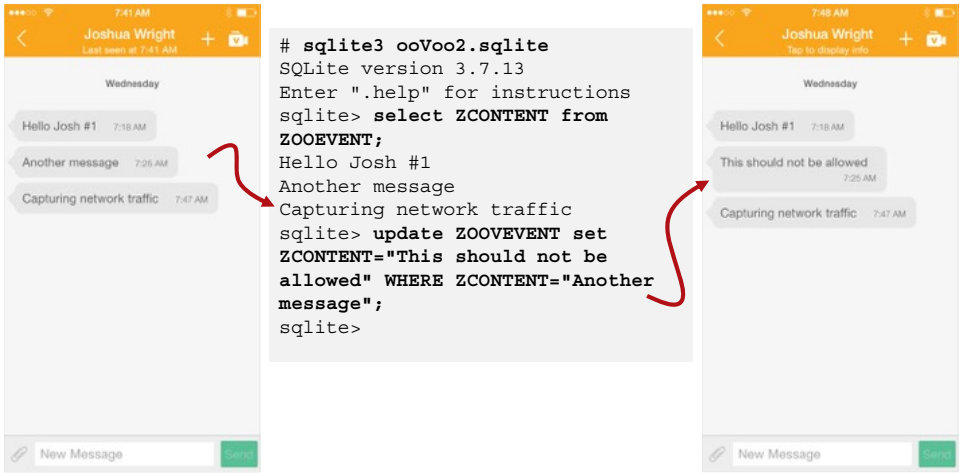
From Linux or OS X, we can use the diff utility with the arguments -r (recursively examine all files in all subdirectories) and -q (limit the diff output to identify new files or changed files, but suppress the differences identified). Specifying the base data set and the changed data set allows us to identify the files that changed between the two application invocations, allowing us to focus our analysis on a smaller subset of files.

In the case of ooVoo, this technique was used to identify filessystem changes following a chat conversation. Differences between the "A" and "B" copies of the application sandbox indicate several changes, including a txt file and a SQLite database file. In the case of the SQLite database file, note that files with "sqlite-shm" and "sqlite-wal" extensions are also changed; these are normal SQLite files for shared memory storage and write-ahead logging, respectively, and do not have to be evaluated independent of the main database file. Inspecting the main database file with the command line sqlite3 tool and retrieving the contents of the database table ZOOVEVENT indicates that the chat log history is stored in plaintext, along with other numeric and possible date stamp identifiers.

FILESYSTEM CHECKSUMS

Does the app use checksums to detect filesystem changes?

Less important in ooVoo, particularly important for games and other apps recording in-app purchases



116

Filesystem Checksums

The next step is to evaluate if the application takes steps to validate the contents of files on the filesystem. Developers should take steps to ensure the integrity of files that are used in the application have not been modified. This is somewhat less important in an application such as ooVoo where chat conversation history and cached files are stored, but particularly important for applications where in-app purchases are used as a revenue source.

For example, consider the game Plants vs. Zombies 2 by PopCap Games. At the time of this writing, PvZ2 is the #76 top free game for iOS, grossing over $20,000 daily in revenue (http://thinkgaming.com/app-sales-data/top-free-games/). As a free game, revenue for PvZ2 comes from in-app purchases, notably the use of gems to aid in game play, ranging from $5 for 50 gems to $20 for 250 gems. When a user makes an in-app purchase for gems, PvZ2 records the gem count in an app sandbox file named "pp.dat". If an attacker edits the pp.dat file with a hex editor, he can change the number of recorded gems at file offset 0x166 to an arbitrary number, bypassing the purchase incentive properties of the game. As a clear revenue loss flaw in PvZ2, it behooves developers to take steps to mitigate such changes, including the use of filesystem checksum validation.

For ooVoo, no integrity monitoring is used to validate the contents of logging files used to record chat conversations. This is likely only problematic in forensics cases where the contents of chat conversations cannot be relied upon when an adversary has exploited or otherwise has access to a victim's iOS device.

KEYCHAIN USE: KEYCHAIN_DUMPER

```
# wget --no-check-certificate https://github.com/ptoomey3/Keychain-
Dumper/raw/master/keychain_dumper
# chmod 755 keychain_dumper
# ./keychain_dumper | grep -A4 com\.oovoo
Service: com.oovoo.iphone.free
Account: mobileConnectDeviceUniqueIdentifier
Entitlement Group: QGW54X6PW3.com.oovoo.iphone.free
Label: (null)
Generic Field: mobileConnectDeviceUniqueIdentifier
Keychain Data: C25D263A-C371-43B5-9ABE-ED5B3BB41B5B

--
Service: com.oovoo
Account: joswr2ght
Entitlement Group: QGW54X6PW3.com.oovoo.iphone.free
Label: (null)
Generic Field: <7b226175 746f223a 66616c73 657d>
Keychain Data: Ijv2HLajdP
```

Run keychain_dumper with -a to dump all Keychain content

Keychain Use: keychain_dumper

One popular technique for evaluating the Keychain use for an application is to use the application normally and then dump the contents of the Keychain with the keychain_dumper tool by Patrick Toomey (https://github.com/ptoomey3/Keychain-Dumper). Download the utility using wget from a jailbroken iOS device and run as shown to extract the contents of the Keychain. By default, keychain_dumper only extracts Keychain Entitlements; run with the "-a" argument, keychain_dumper extracts all Keychain data, including generic passwords, internet passwords, identity information, certificates, and RSA keys.

IOS KEYCHAIN ACCESSIBILITY ATTRIBUTE

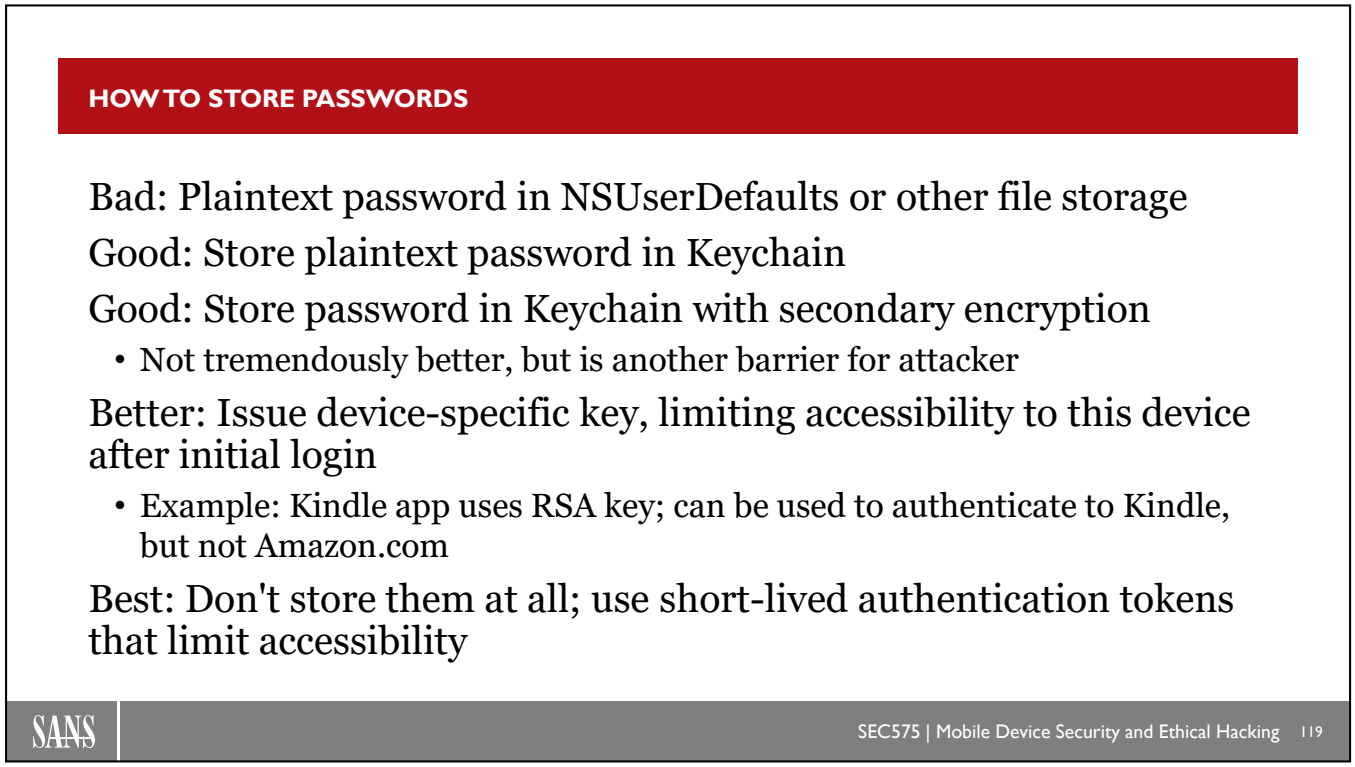
Attribute	Data Is ...	Use When
kSecAttrAccessibleWhenUnlocked	Only accessible when device is unlocked.	Authorized user, transfer across devices through backup + restore.
kSecAttrAccessibleAfterFirstUnlock	Accessible while locked. But if the device is restarted, it must first be unlocked for data to be accessible again.	Today extensions ("widgets") that need Keychain data and accessibility from a locked device or background running tasks; backed up.
kSecAttrAccessibleAlways	Always accessible.	Never use this.
kSecAttrAccessibleWhenUnlockedThisDeviceOnly	Only accessible when device is unlocked. Data is not migrated via backups.	Like kSecAttrAccessibleWhenUnlocked, but not transferred with backup.
kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly	Accessible while locked. But if the device is restarted, it must first be unlocked for data to be accessible again. Data is not migrated via backups.	Like kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly, but not transferred with backup.
kSecAttrAccessibleAlwaysThisDeviceOnly	Always accessible. Data is not migrated via backups.	Never use this.

iOS Keychain Accessibility Attribute

When passwords and other sensitive content are identified in the Keychain, evaluate the storage attribute associated with the Keychain entry. When developers use the Keychain, they indicate the level of accessibility for the Keychain item with one of the attributes shown on this slide.

In the case of ooVoo, Keychain data is stored with the `kSecAttrAcessibleWhenUnlocked` attribute, meaning that the Keychain-stored credential is available to the application when the iOS device is unlocked (regardless of the use of a PIN or password used to unlock the device). Secrets stored with this attribute are also retained in backups, which allows a user who purchases a new device to restore a previously backed-up iOS device to a new device and regain access to ooVoo without re-entering the password.

The use of the `kSecAttrAcessibleWhenUnlocked` attribute is probably sufficient for ooVoo, but threatens credential compromise when the Mac or Windows system is compromised where a backup is stored. An attacker who compromises a host system with an iTunes backup can use the Elcomsoft Phone Breaker (EPB) utility to dump the contents of the Keychain (the `kSecAttrAcessibleWhenUnlocked` attribute). If the developer wanted to avoid this risk, the Keychain attribute could be marked with the `kSecAttrAccessibleWhenUnlockedThisDeviceOnly` attribute and require the user to re-enter the password after restoring a backup to a new device.



How to Store Passwords

It is useful to help developers understand different techniques that can be used to store passwords and what the positive and negative implications are of each technique. In order of worst to best:

- **Plaintext passwords in NSUserDefaults or other files**: The NSUserDefaults object is common in iOS apps to record the application preferences specified by a user. Because these preferences are stored in the filesystem in plaintext (in /private/var/mobile/Containers/Data/Application/*GUID*/Library/Preferences/*appname*.plist), this is not a recommended practice. Similarly, storing passwords in plaintext in files that are accessible within the sandbox is not recommended.
- **Store plaintext password in Keychain**: This is a better practice than storing the password on the filesystem, but still not a positive security practice. An attacker who can jailbreak the device can easily extract the credential using keychain_dumper.
- **Store password in Keychain with secondary encryption**: Storing the password in the Keychain after encrypting it is better, but not tremendously so. The key used to encrypt the password has to be stored somewhere as well, which would also be accessible to the attacker.
- **Issue device-specific key**: Instead of storing the password at all, use the password for initial authentication and then dispose of it. The authentication server can issue a device-specific key granting subsequent authentication but limited to the specific device. This prevents an attacker who is able to dump the Keychain from retrieving unlimited access to the account, and also prevents him from using the plaintext password as a mechanism to log in to other, unrelated application or website resources. This practice is effectively used by the Amazon Kindle app on iOS, which issues an RSA key after initially authenticating the user, which is only useful for accessing Kindle resources and cannot be used to access Amazon.com.

The best practice is to not store the password at all on the iOS device. Instead, use short-lived authentication tokens that limit server resource accessibility. Although this requires the user to re-enter his credentials often, it effectively mitigates most of the threat associated with loss of compromised devices.

DEBUG DETECTION

TIP

Developers can detect an attached debugger and take action: stop execution, crash, report user details to a server, and so on.

Makes it difficult to perform dynamic reverse engineering.

```
#import <dlfcn.h>
#import <sys/types.h>
#import <stdio.h>
typedef int (*ptrace_ptr_t)(int _request, pid_t _pid, caddr_t _addr, int _data);
void disable_dbg() {
    ptrace_ptr_t ptrace_ptr = (ptrace_ptr_t)dlsym(RTLD_SELF, "ptrace");
    ptrace_ptr(31, 0, 0, 0); // PTRACE_DENY_ATTACH = 31
}
int main() {
#ifndef DEVEL
    disable_dbg();
#endif
    printf("Hello, World\n");
}
```

```
ipad # ./debugdetect
Hello, World
ipad # gdb -q debugdetect
(gdb) run
Starting program:
/private/var/root/debugdetect

Program exited with code 055.
```


Debug Detection

iOS apps should include code scattered throughout the application functionality to identify if the application has been invoked by, or has been attached with, a debugger. Tools such as the GNU Debugger (gdb) are commonly used to reverse engineer applications and are used with scripts to decrypt iOS applications. If the app detects a debugger attached, it can stop execution, subtly change application functionality, or report the details of the system (IP address, geolocation information, username entered into the application, and so on) to a remote server.

The code shown on this slide was developed to identify the presence of a debugger and halt execution when found. The `disable_dbg()` function invokes the `ptrace()` function using the `PTRACE_DENY_ATTACH` flag (`ptrace()` is included in the C library for iOS, but is not exposed in the API, so it is invoked with `dlsym()`). When `ptrace()` is invoked with the `PTRACE_DENY_ATTACH` request type, the application halts execution and exits immediately. If a debugger is not attached, the application continues to execute normally.

Because `disable_dbg()` also prevents developers from debugging the app, use conditional compilation to include this functionality in the application code. In the example on this slide, the `main()` function invokes `disable_dbg()` only when the `DEVEL` definition is not set, allowing developers to retain use of debugging in development, but prevent debug access in production.

This code, and instructions to building from an OS X system with Xcode command line tools, is available at https://gist.github.com/joswr1ght/fb8c9f4f3f9a2feebf7f (or visit https://gist.github.com/joswr1ght and scroll to "iosdebugdetect").

To test if the app does debug detection, invoke it with gdb, as shown from this slide (remember to invoke the actual Mach-O binary; for ooVoo, executable is stored in /private/var/mobile/Containers/Bundle/Application/*GUID*/ooVoo.app/ooVoo). Enter the "run" command to start the binary and watch for application termination, ASL entries, network traffic, or other subtle behavior that could indicate debug detection. ooVoo did not exhibit any behavior that would indicate debug detection.

Unfortunately, adding debug detection does not prevent an attacker from creating an unencrypted app bundle with DumpDecrypted. DumpDecrypted uses library preload functionality to inject its code prior to execution to create an unencrypted binary, preventing the developer from detecting the debugger.

```
1   @interface UrlConnection : NSObject
2   @property (strong) NSString *url;
3   - (void)connect;
4   @end
5
6   @implementation UrlConnection
7   - (void)connect {
8           // Connect to a server, or other behavior the
9           // attacker wants to manipulate
10  }
11  @end
```

Class Header File

```
12  Class ucclass = objc_getClass("UrlConnection");
13  SEL sel = sel_getUid("connect");
14  // Save the original address for the UrlConnection implementation
15  IMP ucconnectimp = class_getMethodImplementation(ucclass, sel);
16  IMP runtimeimp;
17  while(1) {
18      [NSThread sleepForTimeInterval:10.0f];
19      ucclass = objc_getClass("UrlConnection");
20      sel = sel_getUid("connect");
21      runtimeimp = class_getMethodImplementation(ucclass, sel);
22      if (runtimeimp != ucconnectimp) printf("Modification Detected");
23  }
```

Runtime Code

Detecting Runtime Modification

Although debug detection identifies a debugger, such as gdb attached to a binary, it does not detect runtime modification using Cydia Substrate or Cycript. However, a developer can add code to build a table of critical method addresses (in Objective-C, "Implementations") at startup and compare method addresses later to those recorded at startup to identify runtime modification.

In the code example on this slide, there are two files: first, a class header file that implements an object called `UrlConnection` with a method called `connect`. The code for the connect method is not important to us (and omitted here, for space), other than it could be something an attacker would want to manipulate.

The second file is the code of the application at startup, instantiating the `UrlConnection` class on line 12 as the `ucclass` variable. A Selector (`SEL`) is then retrieved for the connect method in the `UrlConnection` object. Next, an Implementation (`IMP`) variable `ucconnectimp` is declared, using `class_getMethodImplementation()` to record the address of the `UrlConnection` class and method.

With the `ucconnectimp` Implementation declared, the application runs normally. Periodically, the developer should examine the current address of the `ucconnect` object and compare it to the previously declared `ucconnectimp` Implementation. If the periodically checked Implementation (here, `runtimeimp`) does not match the earlier Implementation recording (`ucconnectimp`), it indicates that the application has been manipulated by an attacker, likely through Cydia Substrate.

A complete example of this code, with a target application, is posted at https://gist.github.com/joswr1ght/4480de9862284fc74f6e (or visit https://gist.github.com/joswr1ght and search for "catchredir"). When this app is compiled and run, it produces output displaying the address of the previously recorded and current `ucclass` variable every 10 seconds, as shown:

```
# ./catchredir
pointer 0x3bd85 and 0x3bd85
pointer 0x3bd85 and 0x3bd85
```

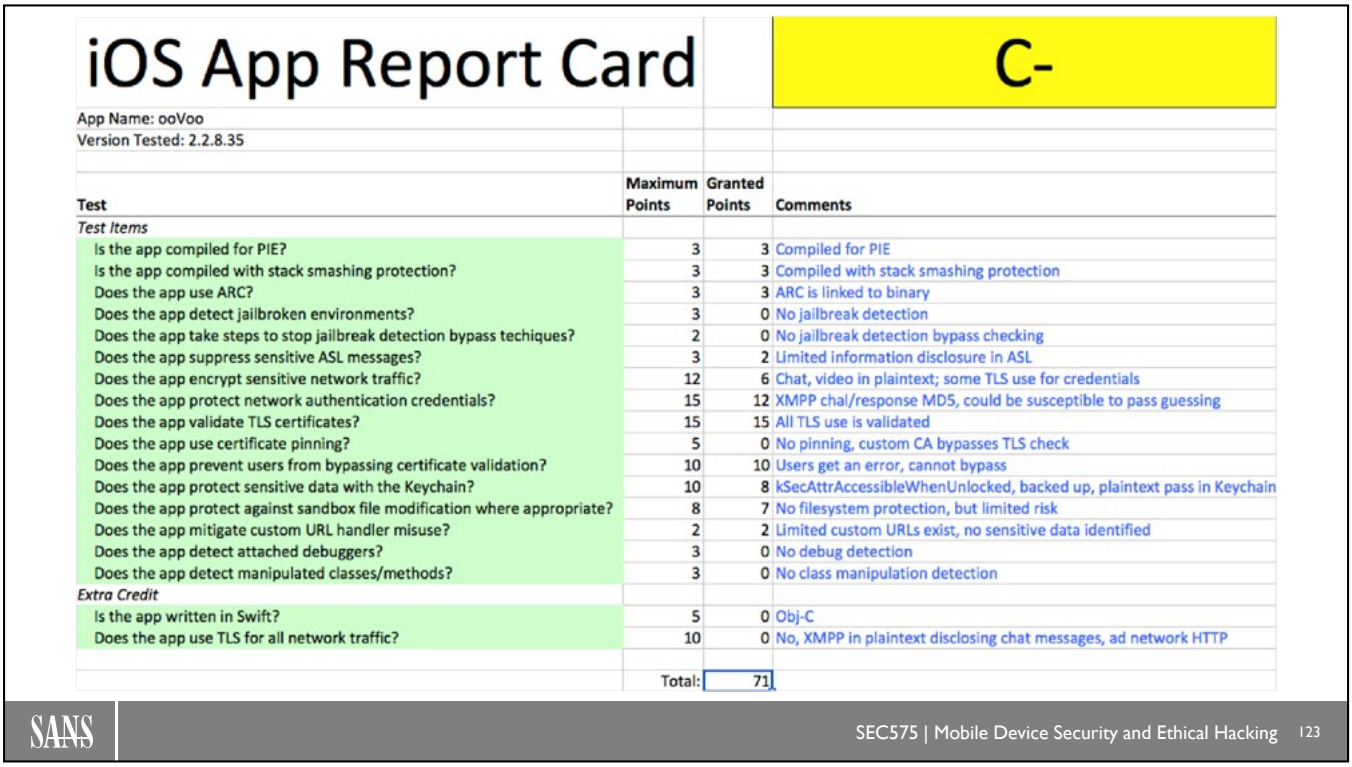
In a second window, attach to the running catchredir process using Cycript and redirect the connect method of the UrlConnection class to a local function, as shown:

```
# cycript -p catchredir
cy# UrlConnection = ObjectiveC.classes["UrlConnection"]
UrlConnection
cy# UrlConnection.messages['connect'] = function() { return; }
function () {return;}
```

With 10 seconds, the output from the catchredir process changes, indicating the redirected address of the attacker-specified function:

```
pointer 0x3bd85 and 0x3bd85
pointer 0x3bd85 and 0x324000
Modification Detected
```

For ooVoo, no runtime manipulation detection system could be identified.



iOS App Report Card

C-

App Name: ooVoo
Version Tested: 2.2.8.35

Test	Maximum Points	Granted Points	Comments
Test Items			
Is the app compiled for PIE?	3	3	Compiled for PIE
Is the app compiled with stack smashing protection?	3	3	Compiled with stack smashing protection
Does the app use ARC?	3	3	ARC is linked to binary
Does the app detect jailbroken environments?	3	0	No jailbreak detection
Does the app take steps to stop jailbreak detection bypass techiques?	2	0	No jailbreak detection bypass checking
Does the app suppress sensitive ASL messages?	3	2	Limited information disclosure in ASL
Does the app encrypt sensitive network traffic?	12	6	Chat, video in plaintext; some TLS use for credentials
Does the app protect network authentication credentials?	15	12	XMPP chal/response MD5, could be susceptible to pass guessing
Does the app validate TLS certificates?	15	15	All TLS use is validated
Does the app use certificate pinning?	5	0	No pinning, custom CA bypasses TLS check
Does the app prevent users from bypassing certificate validation?	10	10	Users get an error, cannot bypass
Does the app protect sensitive data with the Keychain?	10	8	kSecAttrAccessibleWhenUnlocked, backed up, plaintext pass in Keychain
Does the app protect against sandbox file modification where appropriate?	8	7	No filesystem protection, but limited risk
Does the app mitigate custom URL handler misuse?	2	2	Limited custom URLs exist, no sensitive data identified
Does the app detect attached debuggers?	3	0	No debug detection
Does the app detect manipulated classes/methods?	3	0	No class manipulation detection
Extra Credit			
Is the app written in Swift?	5	0	Obj-C
Does the app use TLS for all network traffic?	10	0	No, XMPP in plaintext disclosing chat messages, ad network HTTP
	Total:	71	

ooVoo Report Card

The ooVoo report card based on these analysis steps is presented here. There is certainly flexibility in the grading process for subjective analysis, although this author believes that a grade of C- was generous, considering the unencrypted disclosure of video and chat sessions.

ANDROID APP REPORT CARD

Shares some common analysis tasks as iOS devices

- TLS verification, user bypass of certificate failure, check rooted, and so on

Several analysis tasks can be automated with Drozer

Some analysis tasks require app disassembly with JadX

Android App Report Card

Next, we look at the steps to evaluate an Android application using a report card for Android apps. Some of the analysis steps for evaluating Android apps do not differ from the steps for analyzing iOS apps, notably the evaluation of network traffic, TLS validation, and certificate checks.

While Needle provides some insight into the iOS app analysis process, for Android, we rely on Drozer. Drozer provides several modules that are effective to evaluate Android apps that we investigate in this module.

Furthermore, we have much easier access to source code in Android apps than we have for iOS. We also make use of the Java disassembly through JadX.

ZILLOW FOR ANDROID

Mobile app supporting Zillow.com

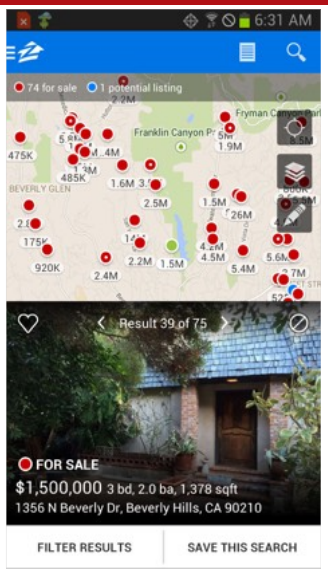
Real estate database, supported by advertising

Use GPS or ZIP Code information to browse homes for sale or rent

Save favorites, past searches by logging in

- Integrated Zillow authentication or through Google, Facebook

Get preapproval for mortgages through app



Zillow for Android

To examine the steps for evaluating Android applications, we complete a report card analysis for the Zillow for Android application. Zillow for Android is the mobile application supporting the Zillow.com website, providing access to real estate listings. Zillow relies on advertising revenue for the service.

Zillow for Android uses GPS or ZIP Code information with Google Maps integration to give the user an interface to browse homes for sale or rent. Users with Zillow.com accounts or integrated authentication through Google or Facebook can log in and save favorites and searches for later use. Authenticated users can also apply for mortgage preapproval through the Zillow for Android app.

```
dz> run app.package.attacksurface com.zillow.android.zillowmap
  5 activities exported
  5 broadcast receivers exported
  0 content providers exported
  2 services exported
dz> run app.activity.info -a com.zillow.android.zillowmap
Package: com.zillow.android.zillowmap
  com.zillow.android.re.ui.homesmapscreen.RealEstateMapActivity
  com.zillow.android.re.ui.homeslistscreen.HomesListActivity
  com.fsr.tracker.app.SurveyInviteActivity
  com.fsr.tracker.app.OnExitSurveyInviteActivity
  com.fsr.tracker.app.SurveyActivity

dz> run app.broadcast.info -a com.zillow.android.zillowmap
Package: com.zillow.android.zillowmap
  Receiver: com.zillow.android.re.ui.notification.GcmBroadcastReceiver
  Receiver: com.zillow.android.re.ui.notification.ZillowADMMessageHandler$Receiver
  Receiver: com.fiksu.asotracking.InstallTracking
  Receiver: com.zillow.android.location.ZillowLocationManagerGeofenceReceiver
  Receiver: com.zillow.android.re.ui.widget.AmazonWidgetBroadcastReceiver

dz> run app.service.info -a com.zillow.android.zillowmap
Package: com.zillow.android.zillowmap
  com.zillow.android.re.ui.notification.GeofenceNotificationMuteIntentService
    Permission: null
  com.zillow.android.re.ui.daydream.ZillowDaydream
    Permission: null
```

Use Drozer's attacksurface module to identify app IPC opportunities

App Intent Handling

First, we examine the Intent handling procedures in the application, using the Drozer app.package.attacksurface module. This module enumerates the activities, broadcast receivers, content providers, and services used by the application. For each of the four service types, we can further evaluate the services using the app.activity.info, app.broadcast.info, app.provider.info, and app.service.info modules, respectively.

For the Zillow application, the app.package.attacksurface module identifies five activities, five broadcast receivers, and two services. These components are further enumerated, as shown on this slide.

This output reveals several interesting characteristics about the Zillow for Android application. The app.activity.info module reveals that two of the modules are published by Zillow.com, but the other three are published by fsr.com, which is ForeSee, an Answers Company. (Notably, ForeSee does not own the fsr.com domain; ForeSee is at www.foresee.com.) These modules are likely part of the ForeSee cxMeasure product for mobile application data collection and analytics.

The Zillow for Android application also registers a broadcast receiver, com.fiksu.asotracking.InstallTracking. Fiksu is a marketing and advertising library to aggregate and optimize ad delivery through multiple ad network providers.

Although the names of the components revealed on this slide provide some insight into the Zillow application, they must be further evaluated using a combination of reverse engineered source code analysis and manual interaction. Next, we will run down some of the Zillow for Android activities and further evaluate threats associated with these components.

INVOKING ACTIVITIES

Apps can invoke other app Activities at any time

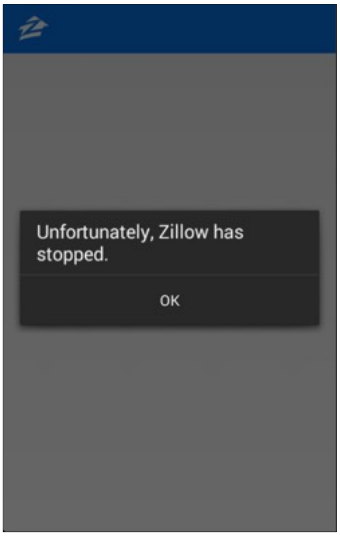
Track down Activities that interpret extra data from the Intent

- getExtra, getSerializableExtra

Time-consuming task

- Can reveal application vulnerabilities that can be exploited by other apps

```
dz> run app.activity.start --component
com.zillow.android.zillowmap
com.zillow.android.re.ui.homeslistscreen.HomesListActivity
dz> run app.activity.start --component
com.zillow.android.zillowmap
com.zillow.android.re.ui.homesmapscreen.RealEstateMapActivity
dz> run app.activity.start --component
com.zillow.android.zillowmap
com.fsr.tracker.app.SurveyInviteActivity
```



Invoking Activities

On the Android platform, any app can invoke any other app's Activities and potentially send data to the Activity for processing through the Intent Extras. If the target application does not validate the Intent Extra data properly, it can lead to privilege escalation or sensitive data disclosure vulnerabilities, allowing an attacker to perform actions with the privileges of the target application or read files from the target application data directory.

For our analysis, we can invoke the other application Activities using Drozer and the app.activity.start module, as shown here. The first two invoked Activities display normal screens for the Zillow application, but the third Activity causes the application to crash. Examining the output of "adb logcat", an error message is captured when the Activity was invoked, as shown:

```
E/AndroidRuntime( 2035): java.lang.RuntimeException: Unable to start activity
ComponentInfo{com.zillow.android.zillowmap/com.fsr.tracker.app.SurveyInviteAc
tivity}: java.lang.NullPointerException
```

The `java.lang.NullPointerException` is an indicator that the application anticipated data in the form of an Activity Extra when the Activity was invoked, but the data was not present in our invocation of the Activity. For each component enumerated by the app.package.attacksurface Drozer module, we should examine the associated Java code to enumerate the Activity Extra data passed to the component by searching for `getExtra()` or `getSerializableExtra()` functions. This can be a time-consuming task, but it is the only effective way to identify vulnerabilities in the handling of application components.

ZILLOW AND FORESEE ANALYTICS

Source for SurveyInviteActivity

```
// Method in com.fsr.tracker.app.SurveyInviteActivity package
public void onCreate(Bundle savedInstanceState) {
    // ... omitted for space
    // Extras is the package, CONTEXT_CONFIG is "CONTEXT_CONFIG", a string
    // getSerializableExtra gets the extra data from the Intent
    this.configuration = (Configuration)
            getIntent().getSerializableExtra(Extras.CONTEXT_CONFIG);
    TrackingContext.start(getApplication(), this.configuration);
    // ... omitted for space
}
```

This is the data supplied in the Intent.

com.fsr.tracker.domain.Configuration

```
public class Configuration implements Serializable {
    public static final int DEFAULT_EXIT_EXPIRY_DAYS = 7;
    public static final String DEFAULT_LOGGING_URL = "http://events.foreseeresults.com/rec/process?event=logit";
    public static final int DEFAULT_MAX_REPEAT_DAYS = 60;
    public static final String DEFAULT_SERVICE_URL = "http://survey.foreseeresults.com/survey/";
    public static final String DEFAULT_SURVEY_URL_BASE = "http://survey.foreseeresults.com/survey/display?";
    public static final String NOTIFICATION_ICON_NAME = "ic_notification";
    public static final String NOTIFICATION_LAYOUT_NAME = "notification";
    private static Logger logger;
    private String clientId;
    private Map<String, String> cpps;
    private String customLogoPath;
    private boolean debug;
    private int exitExpiryDays;
    private int exitExpiryMinutes;
    private LocalNotificationSurvey localNotificationSurvey;
    private String loggingUrl;
```

The attacker can control the Intent content, overwriting any of these default parameters, including the URL where analytic data is submitted.

128

Zillow and ForeSee Analytics

Using JadX to reverse engineer the Zillow APK, we can navigate to the Activity that triggered the crash condition. This Activity is not specific to Zillow but is from a linked library used for customer data analytics for ForeSee in `com.fsr.tracker.SurveyInviteActivity`.

The source code for this Activity includes an `onCreate()` method as the entry point to the Activity, which expects an Activity Extra called `Extras.CONTEXT_CONFIG` (here, the "Extras" part refers to the package name, which is included in `com.fsr.tracker.app.Extras`). Examining the Extras package reveals that `CONTEXT_CONFIG` is simply a string of "CONTEXT_CONFIG".

When we invoked the Activity earlier with Drozer, we did not specify an Activity Extra. This led to an empty value in the `this.configuration` variable in the example, which, when referenced later in the code, led to the NullPointerException.

The `getSerializableExtra()` function here indicates that, when invoked, the Activity expects to receive *serialized* data from the Activity Extra associated with the name `CONTEXT_CONFIG`. The concept of serialized data on Android is that the data being sent is formerly an application object, perhaps a class or other structured data, that is converted to an intermediate data form that can be unserialized and returned to the native data type by the receiving application.

Shortly after the data is received in the `onCreate()` method, the application invokes the `TrackingContext.start()` method, passing `this.configuration` (the serialized data received in the Activity Extra) as a parameter. Tracing the code further reveals that the ForeSee code uses this serialized data to specify several parameters that are used for application data tracking, populating the `Configuration` class. The `Configuration` class (shown on this slide) accepts several parameters, including `DEFAULT_LOGGING_URL`, `DEFAULT_SERVICE_URL`, and `DEFAULT_SURVEY_URL_BASE`, as shown.

This behavior represents a vulnerability in the ForeSee code that affects Zillow. A malicious app on the platform could invoke the ForeSee `com.fsr.tracker.app.SurveyInviteActivity` and supply a custom form of the serialized data to repopulate the Configuration object with different values. This would cause the customer analytic data to be sent to a different server, leading to an information disclosure threat on the platform.

This level of analysis is painfully time-consuming and required several hours of source code analysis on the part of this author to investigate this Activity. However, this level of analysis is required to perform application vulnerability analysis and enumerate the threat surface for a given target application.

APP LOGGING

Prior to Android 4.1, all logs were accessible to any app

- Sensitive content in logging is an information disclosure threat

Evaluate Android logging with "adb logcat"

- Filter using grep/egrep

Encourage developers to log only when BuildConfig.DEBUG is set

```
$ adb logcat | egrep -i "http|https|cookie|login|md5|sha1|auth|pass"
I/AuthZen ( 1535): Signing key fetched successfuly!
W/GmsApplication( 1535): Using Auth Proxy for data requests.
I/GoogleHttpClient( 1535): Falling back to old SSLCertificateSocketFactory
I/GLSUser ( 1535): [ ChannelManager ] Attempting to channel bind connection HttpClient.
```

```
if (BuildConfig.DEBUG) Log.i(TAG, "Message"); // Do This!
```


App Logging

Another analysis step for evaluating an Android application is to examine an application logging message. Prior to Android 4.1, all application logs were accessible to any other app on the system. Sensitive information disclosed in logging represents an information disclosure threat.

We can evaluate application logs by reading the output of "adb logcat". This command generates a large number of logs, so it is often useful to limit the output by searching for specific strings with grep or egrep. In the example on this slide, the "adb logcat" output is filtered using several strings, revealing mostly network activity logs generated by Zillow.

None of these logging entries represent significant information disclosure threats for users.

Encourage developers to perform conditional logging, writing logging messages only when the application is running in a debug configuration. A simple `if` statement is shown on this page, logging only when the BuildConfig.DEBUG flag is set. When the application is compiled for release, it does not generate the logging messages, eliminating any concern of information disclosure threats.

PERMISSION EVALUATION

```
dz> run app.package.info -a com.zillow.android.zillowmap
Package: com.zillow.android.zillowmap
  Application Label: Zillow
  Process Name: com.zillow.android.zillowmap
  Version: 6.2.3378
Uses Permissions:
  - android.permission.INTERNET
  - android.permission.ACCESS_WIFI_STATE
  - android.permission.ACCESS_FINE_LOCATION
  - android.permission.ACCESS_COARSE_LOCATION
  - android.permission.READ_PHONE_STATE
  - android.permission.GET_ACCOUNTS
  - android.permission.WRITE_EXTERNAL_STORAGE
  - com.google.android.providers.gsf.permission.READ_GSERVICES
  - com.zillow.android.zillowmap.permission.MAPS_RECEIVE
  - com.google.android.c2dm.permission.RECEIVE
  - android.permission.USE_CREDENTIALS
  - com.amazon.device.messaging.permission.RECEIVE
  - android.permission.READ_EXTERNAL_STORAGE
```

Identifying permissions that are not needed is difficult and requires familiarity with the app source. Investigate permissions that appear unnecessary.

Permission Evaluation

A security evaluation of an Android application would be remiss if it didn't review the permission declarations used by the application. Evaluating the implication of these permissions is difficult, however.

Developers should not declare a requirement for permissions that are not used. Even if the permission is not used within the app, if it is declared as required in the AndroidManifest.xml file, the end user has to approve that the app uses the declared permission when it is installed. Requiring users to approve permissions that are not necessary can lead to distrust within an application, with users becoming inured to the risks of malicious applications using similar permissions.

Although the Android SDK indicates the necessary permission declarations for specific classes and methods, there is no published reverse mapping that reveals the functions that are available when the permission is declared. Lacking such a taxonomy, there is no straightforward mechanism to identify unnecessary permission declarations. Analysts should use the declared permissions to perform a conjectured analysis of the application: if the application declares the need for RECEIVE_SMS, but there is no reasonable anticipated use of this permission, then the application should be further scrutinized at a source code level with JadX to identify any code that is associated with this permission.

In the example on this slide, we use the Drozer app.package.info module to evaluate the Zillow application. The output from this module has been trimmed to fit on the screen, but it enumerates application details, including the locally installed UID information, the application version information, and the declared permissions. The permissions declared by Zillow are all anticipated for use in an application that uses GPS, performs authentication against Google or Facebook services on the local device, and uses Google services (including Google Maps). Some of the permission declarations indicate that this application is also packaged for distribution in the Amazon App Store.

CERTIFICATE DETAILS

```
$ unzip -l com.zillow.android.zillowmap.apk | grep META-INF
   136637  12-18-14 08:33   META-INF/MANIFEST.MF
   136758  12-18-14 08:33   META-INF/ZILLOWAN.SF
      994  12-18-14 08:33   META-INF/ZILLOWAN.RSA
$ unzip -qq com.zillow.android.zillowmap.apk META-INF/ZILLOWAN.RSA
$ openssl pkcs7 -inform DER -in META-INF/ZILLOWAN.RSA -noout -print_certs
subject=/C=US/ST=Washington/L=Seattle/O=Zillow, Inc./OU=Engineering/CN=Zillow for Android
issuer=/C=US/ST=Washington/L=Seattle/O=Zillow, Inc./OU=Engineering/CN=Zillow for Android
```

```
$ unzip -l com.cheezyapp.apk META-INF/CERT.RSA
$ openssl pkcs7 -inform DER -in META-INF/CERT.RSA -noout -print_certs
subject=/O=Snowrabbit Team/OU=Snowrabbit/CN=Rabbit
issuer=/O=Snowrabbit Team/OU=Snowrabbit/CN=Rabbit
```

Certificate details should reflect the app publisher. Many apps do not include sufficient information to identify the publisher, which is a sign of poor development practices.

Certificate Details

An analyst should investigate the certificate details used to sign the Android application. The certificate details should sufficiently identify the application publisher. In many cases, the certificate details do not provide this information, providing only partial locale or developer aliases.

The Android application APK file includes a directory called META-INF, which includes certificate and signature information. The file ending with the "RSA" extension is the public key used to validate the application signature. Extract this file and inspect the contents from the command line using the openssl utility on Linux or OS X, as shown.

For Zillow, the certificate is sufficiently filled in, identifying the location and common name information for the publisher. In the second example provided for context, a "cheezy" application vendor publishes the app with the organizational name Snowrabbit.

The lack of certificate details does not necessarily introduce a vulnerability in the Android application, but it does indicate a poor development practice. For applications where no care is taken to enter sufficient certificate information, it is likely that other application vulnerabilities also exist.

Zillow uses a mix of HTTP and HTTPS activity

- HTTPS for authentication; HTTP for other activity, including disclosure of sensitive mortgage preapproval details
- Disclosure of contact details sent to agents

Network Analysis

Like iOS applications, Android applications should be evaluated to identify the network activity generated, enumerating the hosts, protocols, and ports in use. The use of encrypted or unencrypted activity should be noted, including the validation of TLS server certificates.

In the case of Zillow for Android, the application uses a mix of HTTP and HTTPS activity. HTTPS activity is used for authentication to the Zillow server and for the retrieval of stored information, such as property favorites and saved searches. The Zillow for Android application is not vulnerable to sidejacking attacks because activity that would disclose cookie information is limited to HTTPS.

However, other functionality in the Zillow application sends sensitive information entered by the user over an HTTP exchange. The Zillow for Android application includes form functionality to get preapproval for a mortgage or to schedule an appointment with a real estate agent to visit a property. This functionality requires that the user enters contact information, including name, phone number, and a free-form message field, and is sent over HTTP.

FILESYSTEM USE

Upload BusyBox binary on rooted device (or emulator)
Tar contents of app data directory
- Copy for offline analysis; use diff to identify filesystem changes

Does the app detect modifications to files?

```
$ pwd
/Users/jwright/Hack/android/B/com.zillow.android.zillowmap/files
$ file [zZ]*
ZMMSavedLoanRequests:          Java serialization data, version 5
ZillowFavoritePropertyType:    Java serialization data, version 5
ZillowRentalsFavoriteHomesNew: Java serialization data, version 5
zillowAdConfig:                Java serialization data, version 5
```

Zillow saves configuration preferences as Java serialized data objects, reading them back into memory as program variables, but does not check for malicious modifications

Filesystem Use

Like iOS application evaluation, Android applications should be evaluated to identify the use of filesystem resources. Analysis can be done manually by evaluating individual files or through differential analysis following a specific application action, such as logging in or accessing a sensitive resource. For sensitive application files, the application should detect modifications to files that have been changed on the filesystem.

Zillow for Android uses the filesystem to store configuration objects, as well as cached images and XML attributes for viewed properties. The configuration objects are interesting because Zillow stores programmatic objects as serialized data on the filesystem, as shown on this slide. Instead of storing user settings and preferences in XML or other ASCII-based configuration files, Zillow stores the internal memory structures to the filesystem and restores these files when the application is run without checking the integrity of the files. An adversary with access to the filesystem could modify these serialized data objects to maliciously manipulate the Zillow application functionality (such as changing the URLs used in the application or declaring and running arbitrary code).

Such an attack vector would be unlikely unless there is a secondary vulnerability in the platform that also grants an attacker access to the sandbox directory. Vulnerabilities such as the TowelRoot exploit affecting all Android devices prior to Android 4.4.4 could be used to obtain such access through a malicious application.

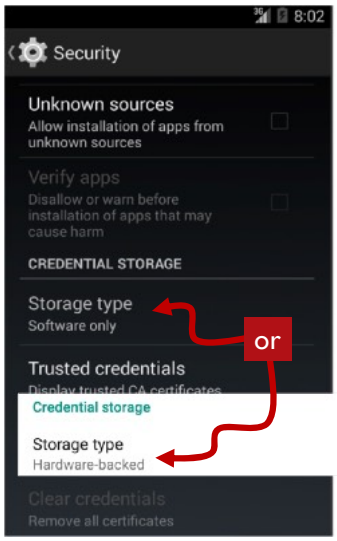
KEY STORAGE:ANDROID KEY STORE

Android 4.3 and later supports the Android Key Store

- Official feature for key storage
- Supports hardware-based storage (some hardware, developers can test for availability)

Apps should use this API for key storage, where needed

- Better not store passwords



Key Storage: Android Key Store

Android developers have multiple options for storing a user's password, though the preferred mechanism is to use the Android Key Store. Although this feature was available in earlier versions of the Android platform, it wasn't officially supported in the Android SDK until Android 4.3 (API 18).

The Android Key Store feature allows a developer to use an asymmetric encryption algorithm for protecting sensitive content. If the hardware itself supports secure key storage, then the asymmetric key is stored in the hardware module; developers can test to determine if the hardware device supports hardware-backed key storage or not (also displayed in the Settings app, as shown on this slide).

Developers should use the Android Key Store feature to protect sensitive content, including passwords, where needed. Of course, it is better not to store passwords at all on the platform if possible.

ANDROID KEY STORE USE

Initialize Key Store by generating RSA (4.3+) or ECC (4.4+) keys

- Asymmetric key stored in hardware, when available

Use to encrypt a symmetric key (or passphrase)

Store encrypted value in filesystem

- No support for storing symmetric keys

Sample implementation linked in notes

Android Key Store Use

The Android Key Store provides an interface to generate an asymmetric RSA key (Android 4.3 and later) or an Elliptic Curve Cryptography (ECC) key (Android 4.4 and later). The storage of this asymmetric key is abstracted to the developer; instead, he gets a named reference that the key-generating application can use to encrypt and decrypt content.

If a developer needs to store a symmetric key (such as an AES key) or a user's password, he can use the Key Store feature to generate an asymmetric key that is stored in hardware, using the API to encrypt the secret. The Key Store API does not provide a facility for storing encrypted secrets; developers can write to the filesystem to save the encrypted AES key or user password information.

The API is fairly straightforward for the Key Store feature; a sample implementation is available as part of the Android open source code at

https://android.googlesource.com/platform/development/+/master/samples/Vault/src/com/example/android/vault/SecretKeyWrapper.java.

To evaluate an Android application's use of key handling, retrieve the application source using JadX and search for the Key Store initialization object call `KeyStore.PrivateKeyEntry`. If this object reference is not present in the Android source, the developer may have chosen an alternate mechanism for key storage. Returning to the application source, evaluate the Activity where the user's password is entered and follow the source to identify any references to the password field and how it is stored on the system.

In the case of Zillow for Android, the application does not store the user's password. Instead, the password is used for authentication to the backend server, which returns a session cookie with a 20-year expiration period. This cookie is stored in plaintext in the webviewCookiesChromium.db file in /data/datacom.zillow.android.zillowmap/databases.

DEBUG DETECTION

Apps built for debugging allow a developer to trace and modify execution
- Some Android developers leave apps in debug mode unknowingly

Other running apps can attach to debug-enabled app and modify execution
- Gaining privileges of the target app

Test by inspecting AndroidManifest.xml or Drozer

```
$ unzip com.zillow.android.zillowmap.apk AndroidManifest.xml
Archive:  com.zillow.android.zillowmap.apk
  inflating: AndroidManifest.xml
$ java -jar AXMLPrinter2.jar AndroidManifest.xml | grep debuggable
              android:debuggable="false"
```

```
dz> run app.package.debuggable -f com.zillow.android.zillowmap
dz>
```


Debug Detection

Android apps should not be published in the Google Play store or other marketplaces when built for debugging. When an app runs on an Android device with debug support, the app creates a socket file named "@jdwp-control". Under normal development practices, the IDE (Eclipse or Android Studio) reads and writes from this socket file to control the app's execution. If an app is deployed when built for debugging, other malicious applications on the same platform can also read and write to this socket file, manipulating the execution of the vulnerable application to escalate the malicious app's privileges, including arbitrary reading and writing to the target app data directory.

In 2013, Tyrone Erasmus of MWR Labs (one of the authors of Drozer) presented his findings relating to the real-world deployment of apps with debug mode enabled. In a sample of 1,000 apps from the Google Play store, Erasmus noted that 5% of them were deployed with debug functionality (https://media.blackhat.com/bh-eu-12/Erasmus/bh-eu-12-Erasmus-Heavy-Metal_Poisoned_Droid-Slides.pdf).

We can test Android applications for the presence of debug functionality by examining the AndroidManifest.xml file, searching for the string "android:debuggable", as shown on this slide. Drozer also includes a module app.package.debuggable that indicates apps that include debug functionality. Run without the "-f" argument, app.package.debuggable examines all the installed apps on the device.

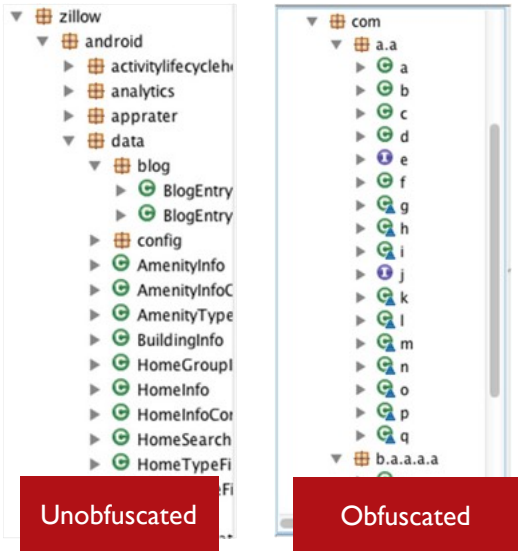
Zillow for Android is used in both examples on this slide and does not ship with debug functionality enabled.

OBJECT OBFUSCATION

Android IDE obfuscators make it harder to read decompiled code

ProGuard is free for Eclipse, Android Studio

DexGuard is a commercial alternative with more features



Object Obfuscation

In order to make it harder to evaluate Android application source code, developers should leverage source code obfuscators. Source code obfuscators remove all the logical package, class, method, and variable names selected by the developer to make the code readable with nonsense characters. Advanced source code obfuscators can even encrypt the names of objects and implement anti-reverse engineering tricks to make application disassembly harder.

The source code obfuscator does not require that the developer gives up a well-named environment for enhanced security. Instead, the obfuscator works when the application is compiled, renaming all the objects and generating a reference file that is used transparently for debugging reference. Without the reference file, however, an attacker would only have the nonsense names available for reference.

In the example on this slide, the Zillow for Android application was opened in the JadX GUI on the left. Zillow developers did not use obfuscation when the app was built, so the app reveals friendly package, class, and method names, as well as variables in those methods. This significantly aids the analyst and the attacker in analyzing the application for flaws.

The example on the right is taken from the Google Earth for Android application. The Google developers used an obfuscator, which converted their friendly package name to "a.a", with methods named "a", "b", "c", etc. This obfuscated naming makes it more complex to read and understand the source.

The Eclipse and Android Studio IDEs include support for ProGuard, a free Android obfuscator. Instructions for using ProGuard are available on the Android Developer site at http://developer.android.com/tools/help/proguard.html. A commercial obfuscator from the author of ProGuard is DexGuard (https://www.saikoa.com/dexguard), which offers advanced features, including encryption and anti-reverse engineering features.

APPLICATION SIGNATURE VERIFICATION

Attacker may resign and redistribute apps
Developers should check app signature validity

- Communicate unauthorized app versions or stop execution

```
$ mkdir keys
$ keytool -genkey -v -keystore keys/ZillowFalseKey.keystore -alias ZillowFalseKey -keyalg
RSA -validity 10000
$ jarsigner -keystore keys/ZillowFalseKey.keystore com.zillow.android.zillowmap.apk
ZillowFalseKey
Enter Password for keystore:
jar signed.
$ adb shell pm list packages | grep zillow
package:com.zillow.android.zillowmap
$ adb uninstall com.zillow.android.zillowmap
Success
$ adb install com.zillow.android.zillowmap.apk
```


Application Signature Verification

It is fairly common for never-do-well attackers to take a popular application from the Android store and repackage and redistribute it. In some cases, the application is modified to add malicious functionality, but in others, it is simply resigned under a different developer alias and reposted to Google Play or other app stores.

Developers can take action within the application source to thwart this type of repackaging attack. When an application package is distributed, the developer generates a keystore (not to be confused with the Android Key Store mechanism) with a public and a private signing key. The application is signed, and the public key is distributed with the application in the META-INF directory within the APK file.

When an attacker repackages an Android application, the application is re-signed using a different private key. The different private key has a correspondingly different public key, which is distributed with the modified APK file instead of the original developer public key. To thwart this kind of activity, developers can check the hash of the public key distributed with the application, comparing it to a hardcoded hash of the known-good public key. If the two hashes do not match, then the developer knows that the application has been repackaged and can stop operation or report the condition to a backend server.

Using the Android `PackageInfo` class, a developer can retrieve the signature value for the running application:

```
PackageInfo packageInfo =
context.getPackageManager().getPackageInfo(context.getPackageName(),
PackageManager.GET_SIGNATURES);

Signature[] appSignatures = packageInfo.signatures;
```

The array of signature information can then be checked against a statically stored byte array or string representing the known valid signature.

This technique was first demonstrated by Keith Makan and Scott Alexander-Bown in their book *Android Security Cookbook* (Packt Publishing). This technique has been further adapted in a complete example by the author (https://github.com/joswr1ght/ValidateSigningCertificate).

To test if the app performs signature validation, simply uninstall the original app, re-sign it using the keytool utility (as shown on this slide), and install it again. Investigate if the application exhibits any behavior that would indicate signature validity checking, signs of changes in application functionality, or new network activity.

Android App Report Card

D

App Name: Zillow (com.zillow.android.zillowmap)
Version Tested: 6.2.3378
Analyst: Joshua Wright (jwright@willhackforsushi.com)

Test	Maximum Points	Granted Points	Comments
Test Items			
Does the app mitigate custom Intent handling misuse?	6	3	Third-party intents can be manipulated for information disclosure
Does the app declare the minimum number of permissions necessary?	2	2	Several permissions declared, mostly needed for ad network use
Is the app signed with accurate and complete certificate details?	2	2	Certificate is complete
Does the app suppress sensitive system log messages (before Android 4.1)?	3	3	No sensitive logging entries were identified
Does the app encrypt sensitive network traffic?	12	8	Some use of TLS, but not for mortgage preapproval
Does the app protect network authentication credentials?	15	8	Password sent over TLS, app is susceptible to sidejacking
Does the app validate TLS certificates?	15	15	App rejects untrusted certificates
Does the app use certificate pinning?	5	0	No pinning use
Does the app prevent users from bypassing certificate validation?	10	10	App displays an error when an untrusted certificate is received
Does the app detect rooted environments?	2	0	App does not detect rooted/emulated environments
Does the app protect against data directory file modification where appropriate?	8	6	No, stores data as serialized Java objects, susceptible to manipulation, lir
Does the app protect sensitive data such as passwords?	10	6	App does not store user password, stores session token with 10 year exp
Is the app built to prevent debugger attachment?	6	6	App is built for release, not debug
Does the app validate the source of the package installer?	2	0	App does not check for package installer source
Does the app validate signature integrity?	5	0	App does not check for consistent signature
Does the app use class and method name obfuscation?	3	0	App does not use ProGuard or other class/method obfuscation
Extra Credit			
Does the app use TLS for all network traffic?	10	0	App uses HTTP for common requests, disclosing session cookie
	Total:	66	

Zillow Report Card

This slide shows the results of the Android App Report Card for the Zillow for Android application. Overall, the application received a lower grade than the ooVoo for iOS application did, although the applications appear to have an approximately even security posture.

The reason the Zillow for Android application received a lower grade is the requirement for more actions on the part of the developer for an Android application than what would be comparable for an iOS platform. The iOS platform offers more inherent security controls than the Android platform. For example, with iOS 8, we have app IPC capability, but it is only initiated by the user (except for URL handlers). In Android, there are broadcast receivers, activities, services, and content providers that all have to be independently evaluated for security flaws. In iOS, applications are natively compiled, making disassembly difficult, while Android developers must take several additional steps to prevent attackers from seeing the Java source of their application.

SUMMARY

The MASVS and app report cards were designed to achieve consistency in evaluation

We can apply the skills we've built to evaluate iOS, Android app security

- This time-consuming process varies in complexity between apps and platforms

For each evaluation task, we can offer developers advice on resolving security concerns

- Which makes us better penetration testers

Summary

In this module, we introduced the concept of using verification sheets such as the MASVS and app report cards for improved consistency in analyzing iOS and Android application security. These simple spreadsheets provide a mechanism to apply the skills we've built in this course to evaluate applications. The process of completing a comprehensive application security analysis is not simple and requires significant time investment on the part of the analyst, but we can avoid missing important analysis steps by tracking our results in the app report card format.

When performing a security analysis of a mobile application, it is important to not only point out the flaw in the application, but also provide a recommendation for developers to rectify the vulnerability. In this module, we looked at techniques to not only evaluate and identify flaws but to also make recommendations that can be applied to resolve the identified vulnerabilities for both iOS and Android developers. Being able to supply recommendations for identified flaws makes the analysis process more valuable for the customer, and it makes us better penetration testers.