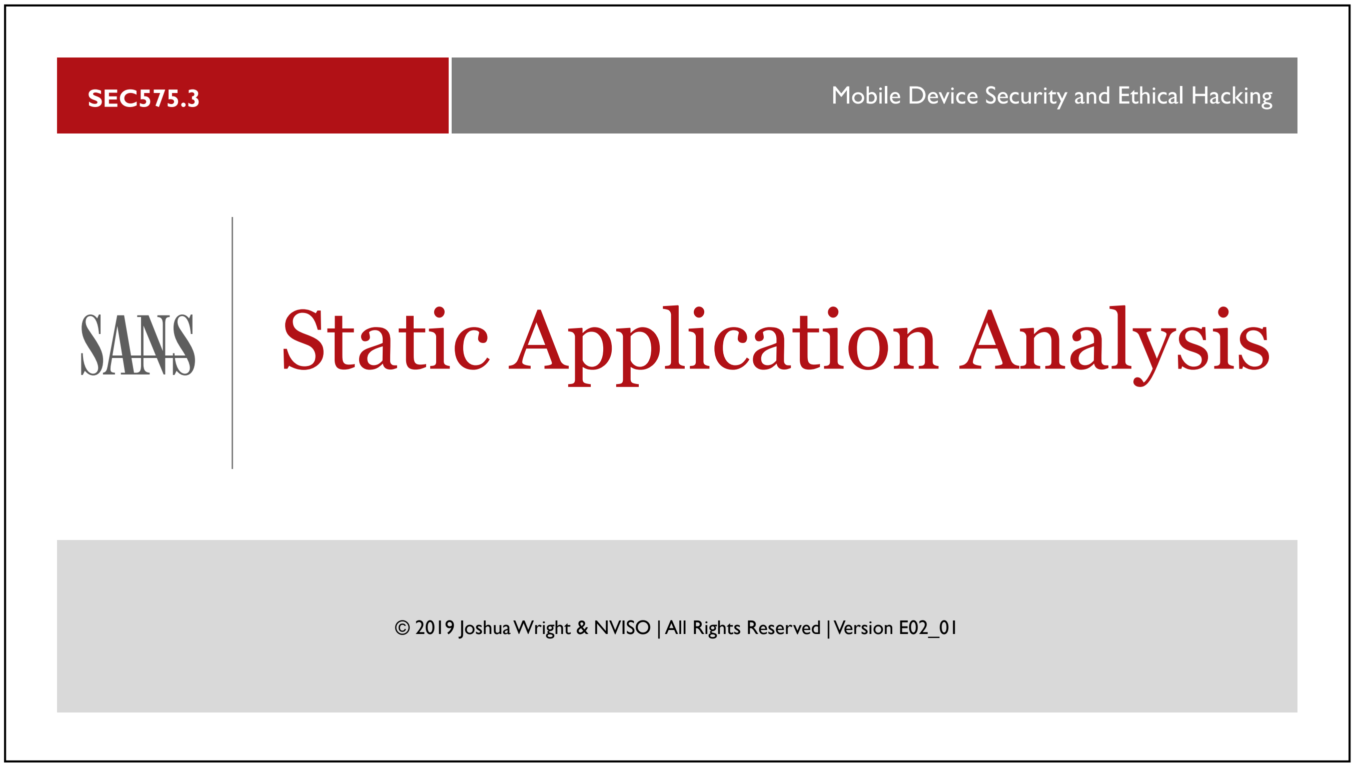
SEC575 | MOBILE DEVICE SECURITY AND ETHICAL HACKING

575.3
Static Application Analysis

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org



SEC575.3

Mobile Device Security and Ethical Hacking

SANS

Static Application Analysis

Welcome to Day 3 of Mobile Device Security and Ethical Hacking! Today's material focuses on static application analysis.

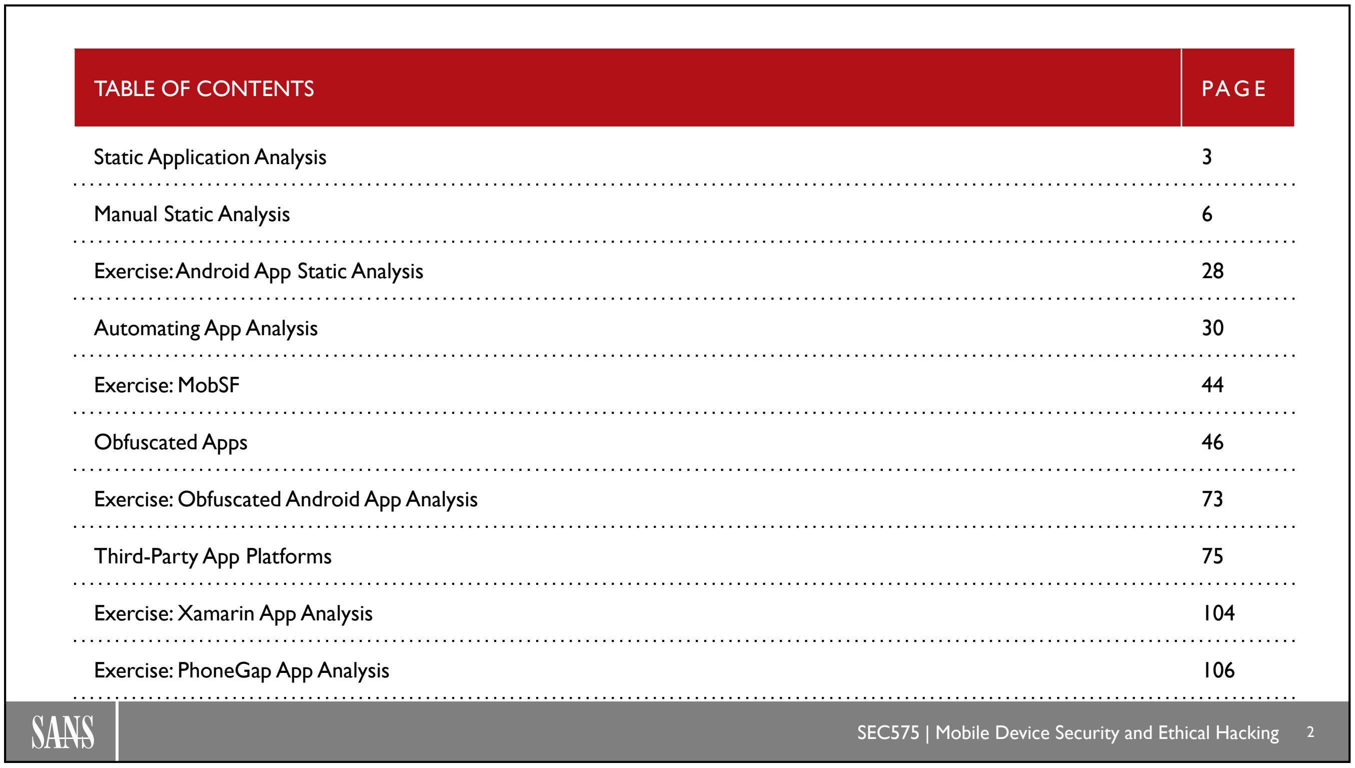
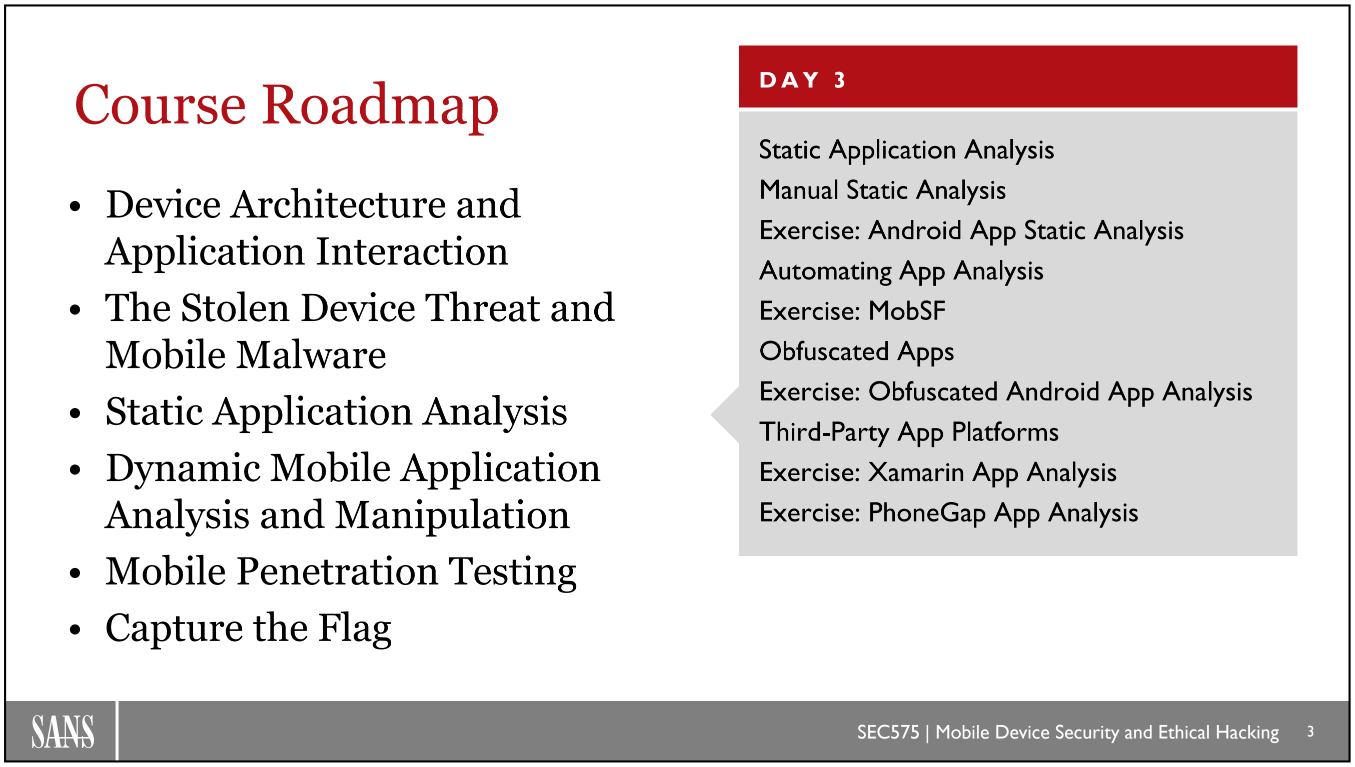


TABLE OF CONTENTS	PAGE
Static Application Analysis	3
Manual Static Analysis	6
Exercise: Android App Static Analysis	28
Automating App Analysis	30
Exercise: MobSF	44
Obfuscated Apps	46
Exercise: Obfuscated Android App Analysis	73
Third-Party App Platforms	75
Exercise: Xamarin App Analysis	104
Exercise: PhoneGap App Analysis	106

SEC575 | Mobile Device Security and Ethical Hacking 2

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

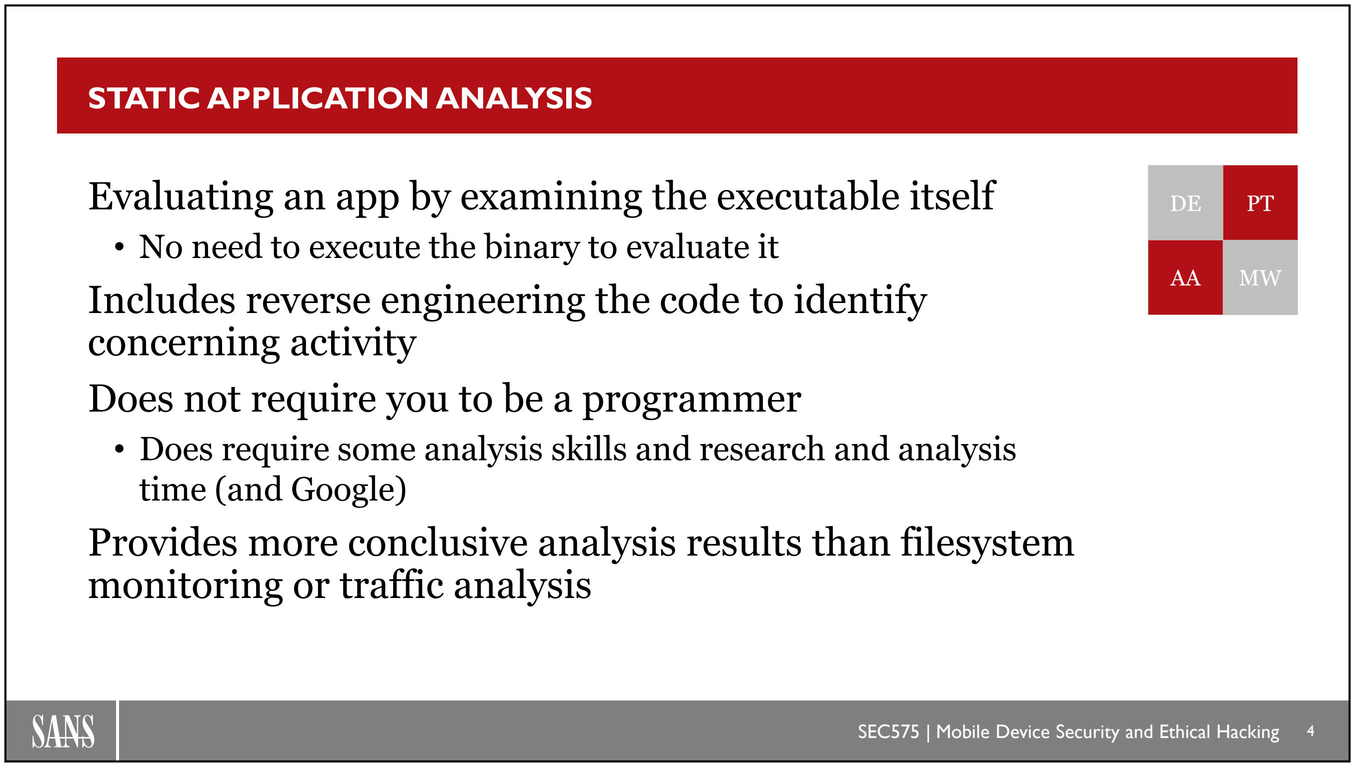
DAY 3

Static Application Analysis
Manual Static Analysis
Exercise: Android App Static Analysis
Automating App Analysis
Exercise: MobSF
Obfuscated Apps
Exercise: Obfuscated Android App Analysis
Third-Party App Platforms
Exercise: Xamarin App Analysis
Exercise: PhoneGap App Analysis

SANS

SEC575 | Mobile Device Security and Ethical Hacking 3

This page intentionally left blank.



STATIC APPLICATION ANALYSIS

Evaluating an app by examining the executable itself

- No need to execute the binary to evaluate it

Includes reverse engineering the code to identify concerning activity

Does not require you to be a programmer

- Does require some analysis skills and research and analysis time (and Google)

Provides more conclusive analysis results than filesystem monitoring or traffic analysis

DE	PT
AA	MW

SANS

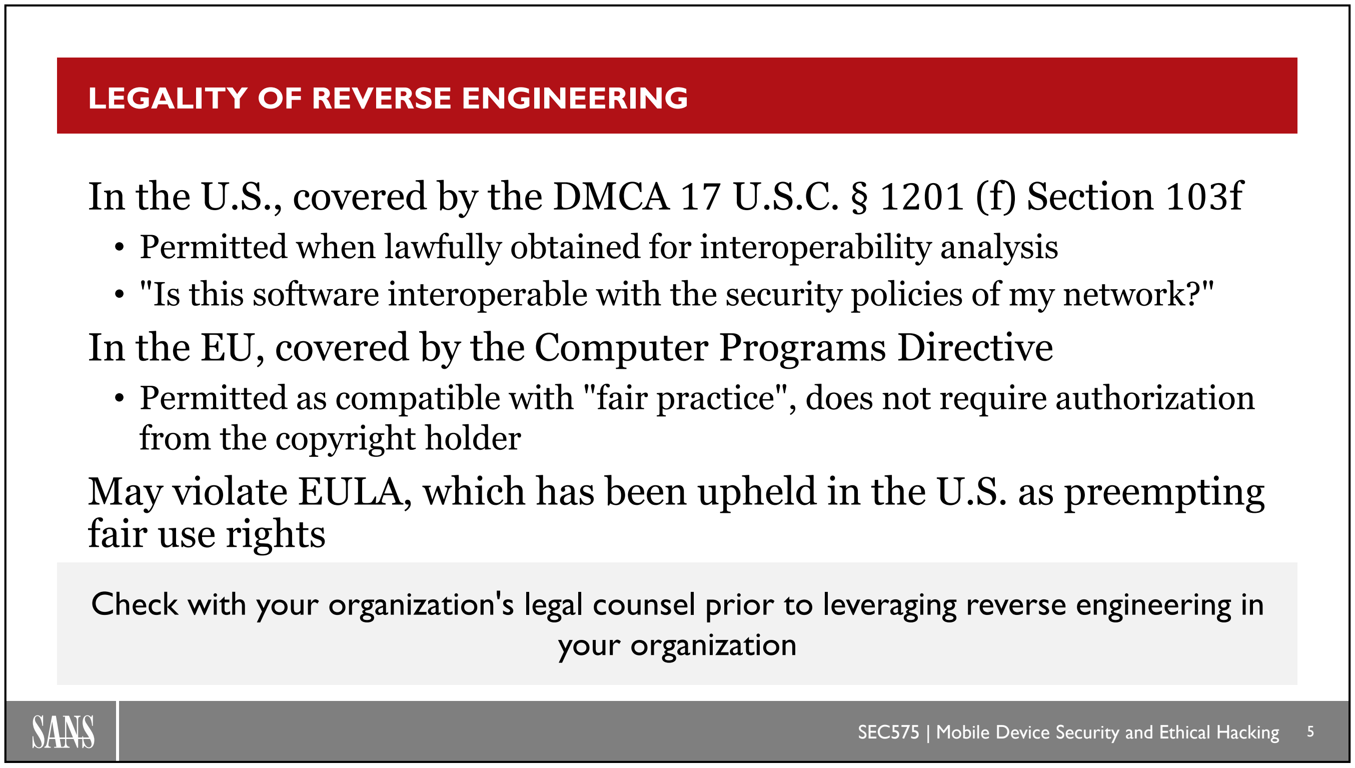
SEC575 | Mobile Device Security and Ethical Hacking 4

Static Application Analysis

Although filesystem application modeling and network activity analysis can tell us a lot about an application, there are cases where we need to further evaluate the functionality of an application in order to validate behavior or identify undesirable activity. Static analysis focuses on the evaluation of the executable itself without executing it, typically through reverse engineering the executable's code.

Reverse engineering does not require you to be a programmer, but it does require some analysis skills, research and analysis time, and a reference source to understand the APIs and systems used by an executable. We can use public resources, such as the SDK documentation, for the various application platforms and Google or other search engines to assist us in evaluating the functions and libraries used by the target executable.

Static analysis provides more conclusive analysis results than filesystem monitoring or traffic analysis. For example, if an application has code logic to deliver only a network payload at a certain time, then we might miss the behavior through filesystem modeling and network analysis. With static analysis, however, we can identify the malicious or undesirable behavior without having to meet the required payload delivery condition.



LEGALITY OF REVERSE ENGINEERING

In the U.S., covered by the DMCA 17 U.S.C. § 1201 (f) Section 103f

- Permitted when lawfully obtained for interoperability analysis
- "Is this software interoperable with the security policies of my network?"

In the EU, covered by the Computer Programs Directive

- Permitted as compatible with "fair practice", does not require authorization from the copyright holder

May violate EULA, which has been upheld in the U.S. as preempting fair use rights

Check with your organization's legal counsel prior to leveraging reverse engineering in your organization

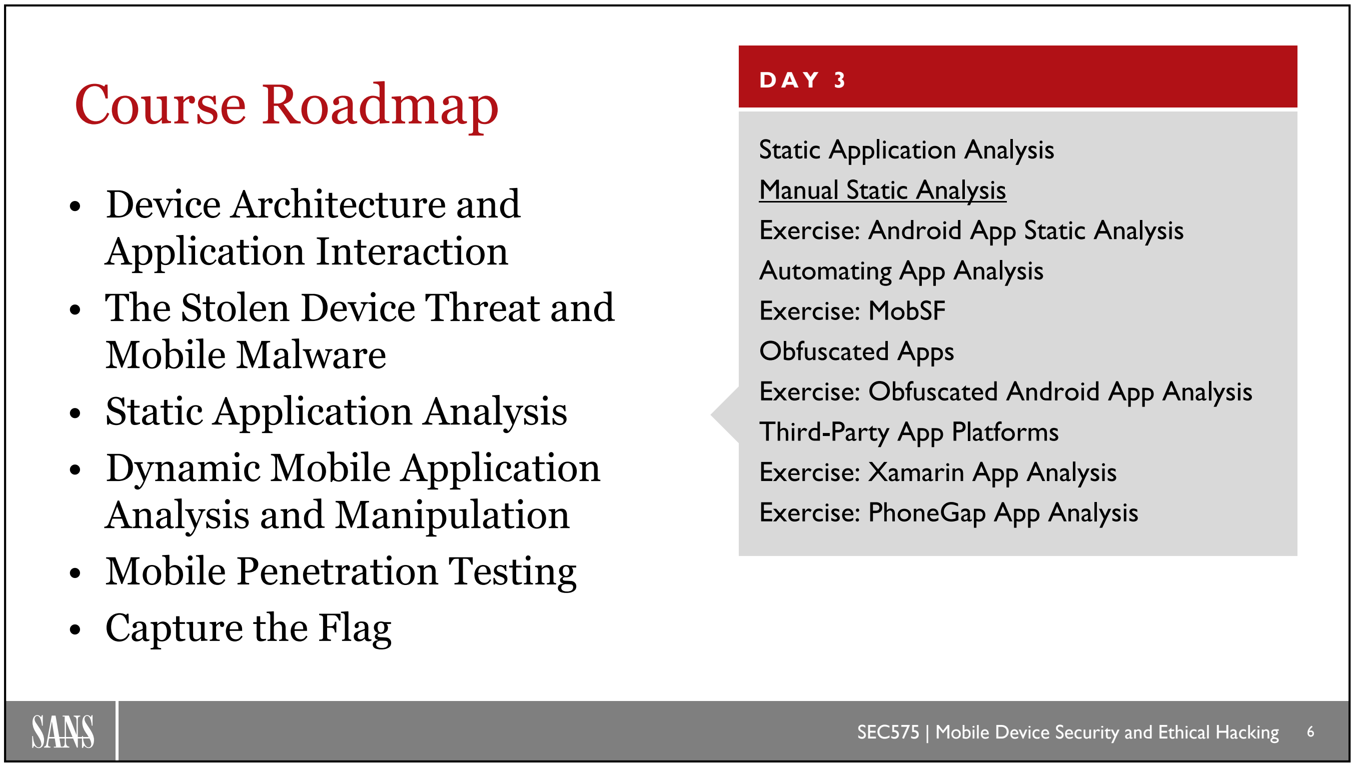
SANS SEC575 | Mobile Device Security and Ethical Hacking 5

Legality of Reverse Engineering

In the United States, the legality of reverse engineering closed source software is typically covered by the Digital Millennium Copyright Act, where it is permitted when the software is lawfully obtained for "interoperability analysis". From a security perspective, reverse engineering commercial software could be covered as trying to determine whether the behavior and activity of the software are in keeping with the security policies of the network.

In the EU, reverse engineering is typically covered by the Computer Programs Directive, where it is permitted as "fair practice” and does not require authorization from the copyright holder.

However, software vendors can create challenges when they state that reverse engineering is not permitted in their end user license agreement, which has been upheld in the United States as preempting fair use rights in *Bowers v. Baystate Technologies* (http://www.infoworld.com/d/developer-world/contract-case-could-hurt-reverse-engineering-337). Always check with your organization's legal counsel prior to leveraging reverse engineering in your organization to ensure that it is acceptable per organizational standards.



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 3

Static Application Analysis
Manual Static Analysis
Exercise: Android App Static Analysis
Automating App Analysis
Exercise: MobSF
Obfuscated Apps
Exercise: Obfuscated Android App Analysis
Third-Party App Platforms
Exercise: Xamarin App Analysis
Exercise: PhoneGap App Analysis

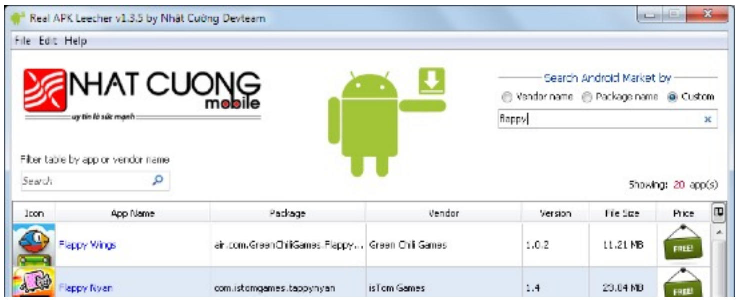
SANS SEC575 | Mobile Device Security and Ethical Hacking 6

This page intentionally left blank.

RETRIEVING ANDROID APPS

Option 1: Install app on rooted Android device, retrieve from filesystem
Option 2: Download app to host with Real APK Leecher
- Configure your device ID and credentials in app

Option 3: APK Downloader extension
Option 4: www.apkmonk.com


Real APK Leecher v1.3.5 by Nhật Cường Devteam

File Edit Help

Search Android Market by: ○ Vendor name ○ Package name [● Custom]

flappy

Filter table by app or vendor name

Search

Showing: 20 app(s)

Icon	App Name	Package	Vendor	Version	File Size	Price
	Flappy Wings	air.com.GreenChiliGames.Flappy...	Green Chili Games	1.0.2	11.21 MB	FREE!
	Flappy Nyan	com.istomgames.tappynyan	isTom Games	1.4	23.04 MB	FREE!

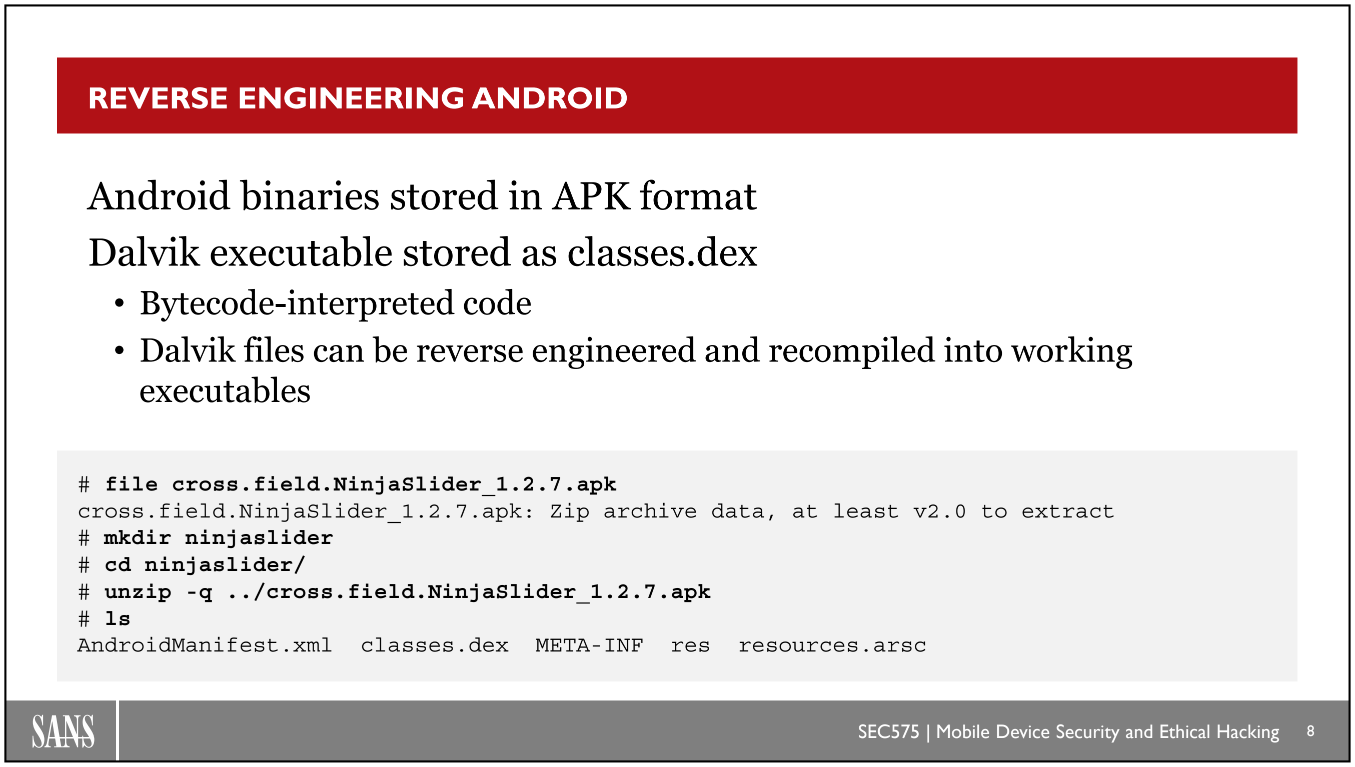
Retrieving Android Apps

Prior to starting the analysis of the Android application, we should retrieve the Android Package file (APK) for the app. There are four primary options for retrieving APK files. Option 1 is to install the app on your rooted Android device and retrieve the APK file from the filesystem in /data/app, /data/app-private, /mnt/asec, or /mnt/secure/asec. Download the file with "adb pull" or retrieve it from an SD card.

The second option is to download the file to your Windows host system using Real APK Leecher by Nhat Cuong. Available at http://forum.xda-developers.com/showthread.php?t=1563894, Real APK Leecher uses your Google Play login credentials and the device ID of an authorized Android device to search and retrieve APK files at an arbitrary directory location you specify in the application preferences. Paid apps can also be retrieved with Real APK Leecher, provided they are purchased in advance through your configured Google Play account.

A third option is to use the Chrome or Firefox browser with the APK Downloader extension available at http://codekiem.com/apk-downloader. When this extension is loaded, browsing to the Google Play store and viewing an app will display a new button "Download APK" next to the default "Install" button. Like the Real APK Leecher, the Chrome APK Downloader extension requires that a device ID and Google Play credentials be submitted for use with the extension.

Finally, the apkmonk.com site is an easy source to download free APK files. You may not get the most current version of an APK, though, and there is little vetting of the APKs distributed from the site, so use this resource with caution.



REVERSE ENGINEERING ANDROID

Android binaries stored in APK format

Dalvik executable stored as classes.dex

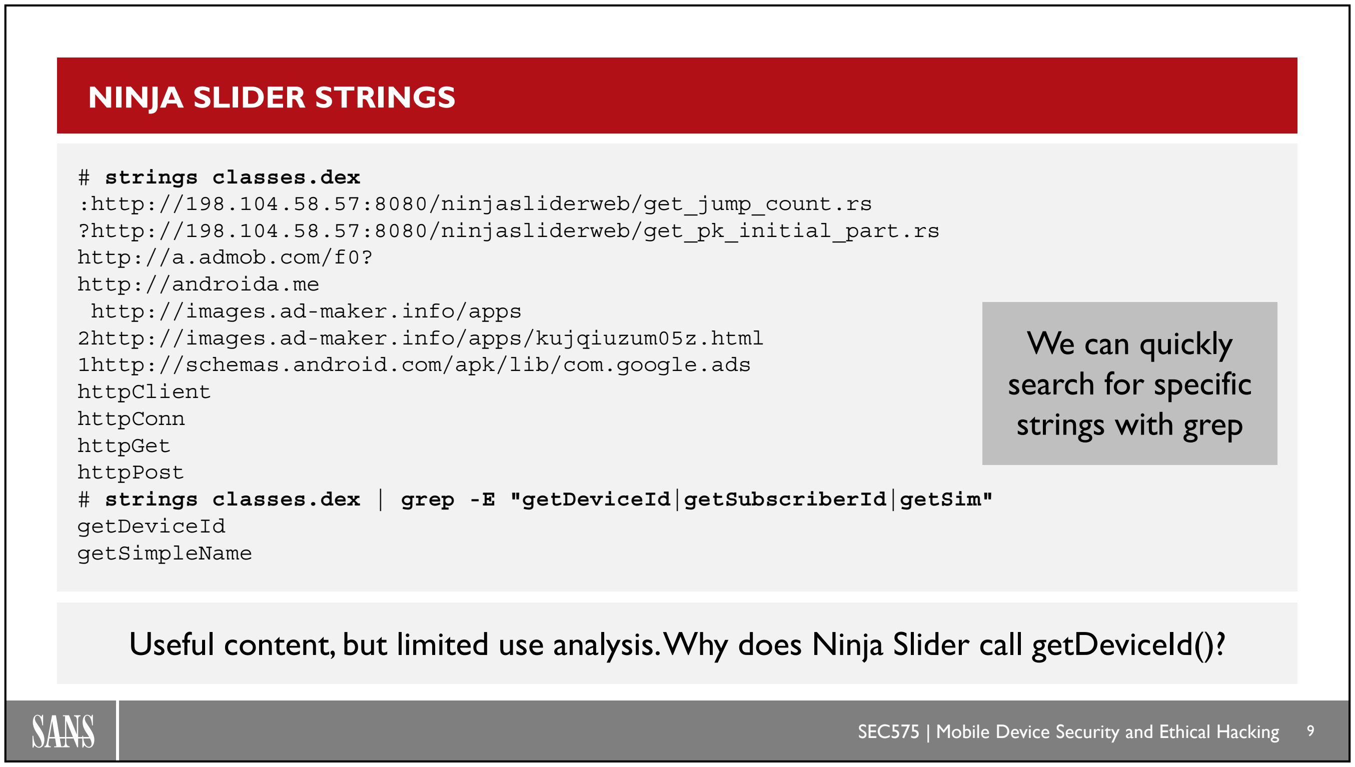
- Bytecode-interpreted code
- Dalvik files can be reverse engineered and recompiled into working executables

```
# file cross.field.NinjaSlider_1.2.7.apk
cross.field.NinjaSlider_1.2.7.apk: Zip archive data, at least v2.0 to extract
# mkdir ninjaslider
# cd ninjaslider/
# unzip -q ../cross.field.NinjaSlider_1.2.7.apk
# ls
AndroidManifest.xml  classes.dex  META-INF  res  resources.arsc
```

Reverse Engineering Android

The Dalvik executable in an APK file is always stored as "classes.dex". This file will be our target for static analysis. For our examples, we'll reverse engineer the Ninja Slider game that we introduced briefly earlier today.

In the example on this slide, we see that the file utility reports that the APK file is a zip archive. To extract the classes.dex file, create a temporary directory and unzip the APK file there. Unzipping the Ninja Slider app produces three files consisting of AndroidManifest.xml, classes.dex, and resources.arsc with two directories: META-INF and res.



NINJA SLIDER STRINGS

```
# strings classes.dex
:http://198.104.58.57:8080/ninjasliderweb/get_jump_count.rs
?http://198.104.58.57:8080/ninjasliderweb/get_pk_initial_part.rs
http://a.admob.com/f0?
http://androida.me
 http://images.ad-maker.info/apps
2http://images.ad-maker.info/apps/kujqiuzum05z.html
1http://schemas.android.com/apk/lib/com.google.ads
httpClient
httpConn
httpGet
httpPost
# strings classes.dex | grep -E "getDeviceId|getSubscriberId|getSim"
getDeviceId
getSimpleName
```

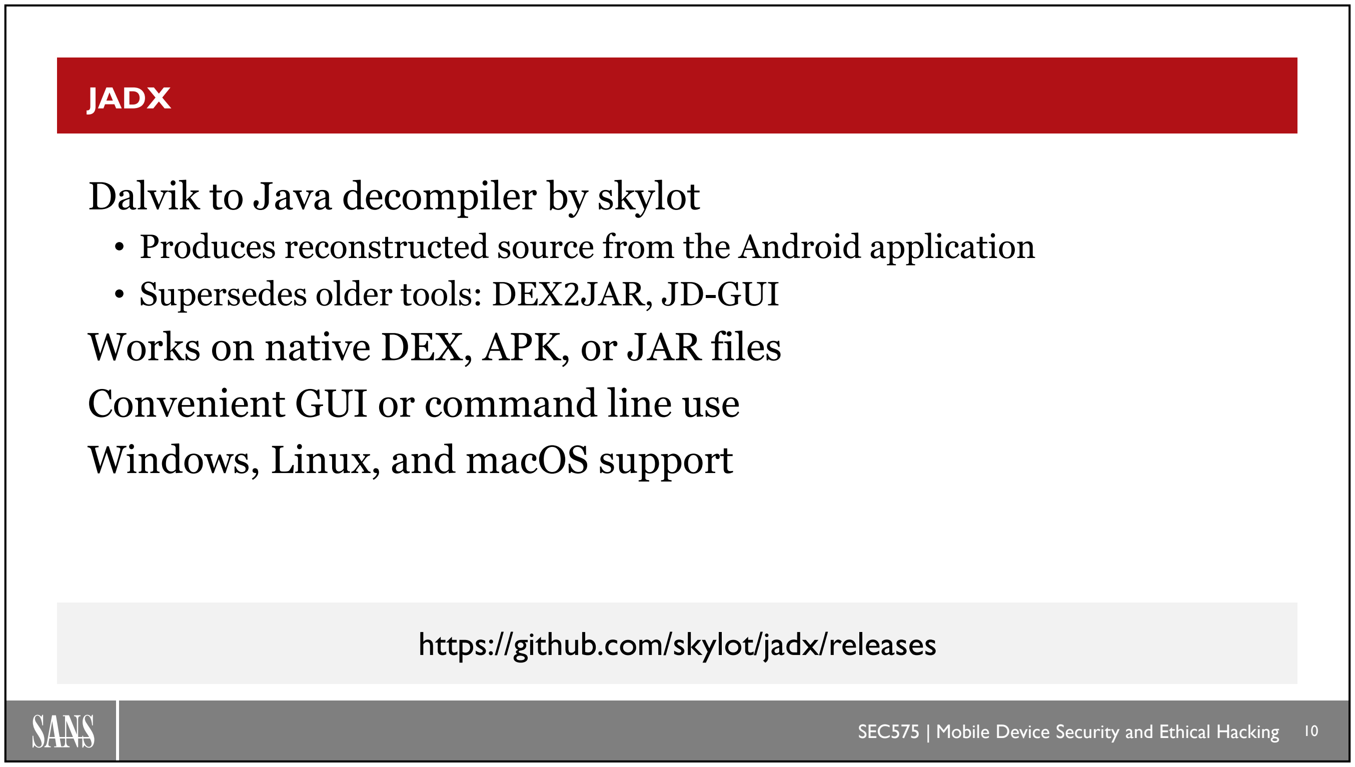
We can quickly search for specific strings with grep

Useful content, but limited use analysis. Why does Ninja Slider call getDeviceId()?

Ninja Slider Strings

Android executables are stored in an unencrypted format, allowing us to easily extract string information from the binary, as shown. As a large number of strings are returned, we can search through the results, looking for matching strings representing sensitive functions, including `getDeviceId()`, `getSubscriberId()`, and `getSim()`. In this example, the `getDeviceId()` and `getSimpleName()` functions are returned (the latter partially matching the `getSim()` function).

Knowing that Ninja Slider calls the `getDeviceId()` function is useful, but we don't know how it is used. We can use reverse engineering to determine why Ninja Slider calls this function and what it does with the resulting data.



JADX

Dalvik to Java decompiler by skylot

- Produces reconstructed source from the Android application
- Supersedes older tools: DEX2JAR, JD-GUI

Works on native DEX, APK, or JAR files

Convenient GUI or command line use

Windows, Linux, and macOS support

https://github.com/skylot/jadx/releases

SANS SEC575 | Mobile Device Security and Ethical Hacking 10

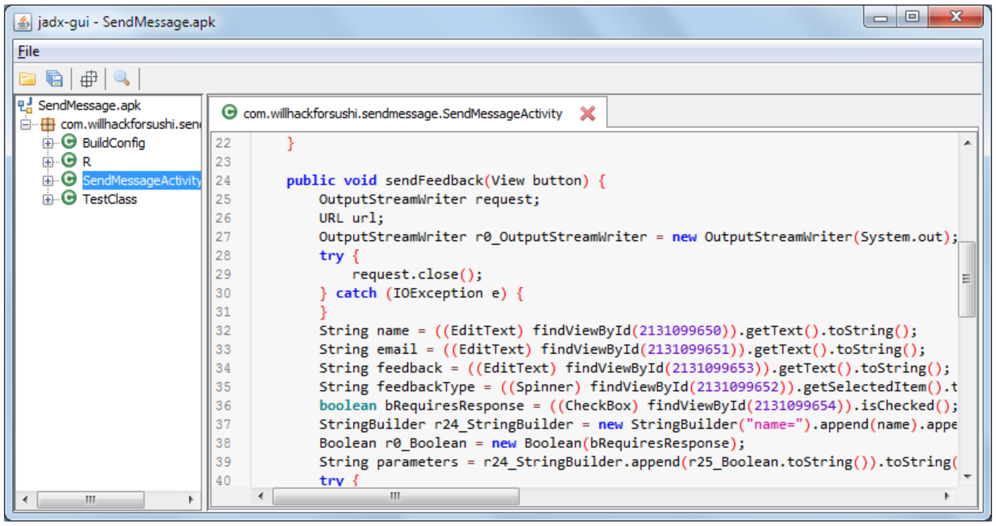
Jadx

Jadx is a Dalvik to Java decompiler written by skylot. Jadx reads from an APK file (or a DEX or Java JAR file) and decompiles the intermediate format bytecode instructions, reconstructing Java source code. The decompiled Java source is not an exact match for the original Java source, but it provides a good representation of what the original source looks like. This is a tremendously useful resource for mobile device security analysts who need to evaluate the functionality of Android applications.

As an Android decompiler, Jadx is superior to legacy tools, such as DEX2JAR and JD-GUI, that can also be used to decompile Android apps. Jadx provides a similar GUI interface to JD-GUI's decompiler but excels at decompiling troublesome code, including developer-selected anti-reverse engineering mechanisms.

Jadx is available as a convenient GUI and as a command line tool. Since Jadx is written in Java, it runs on Windows, Linux, and macOS. Jadx is available in source form on GitHub and as official releases at the URL shown on this page.

```
$ mkdir decompiled
$ jadx -d decompiled/ SendMessage.apk
INFO  - processing ...
INFO  - done
```



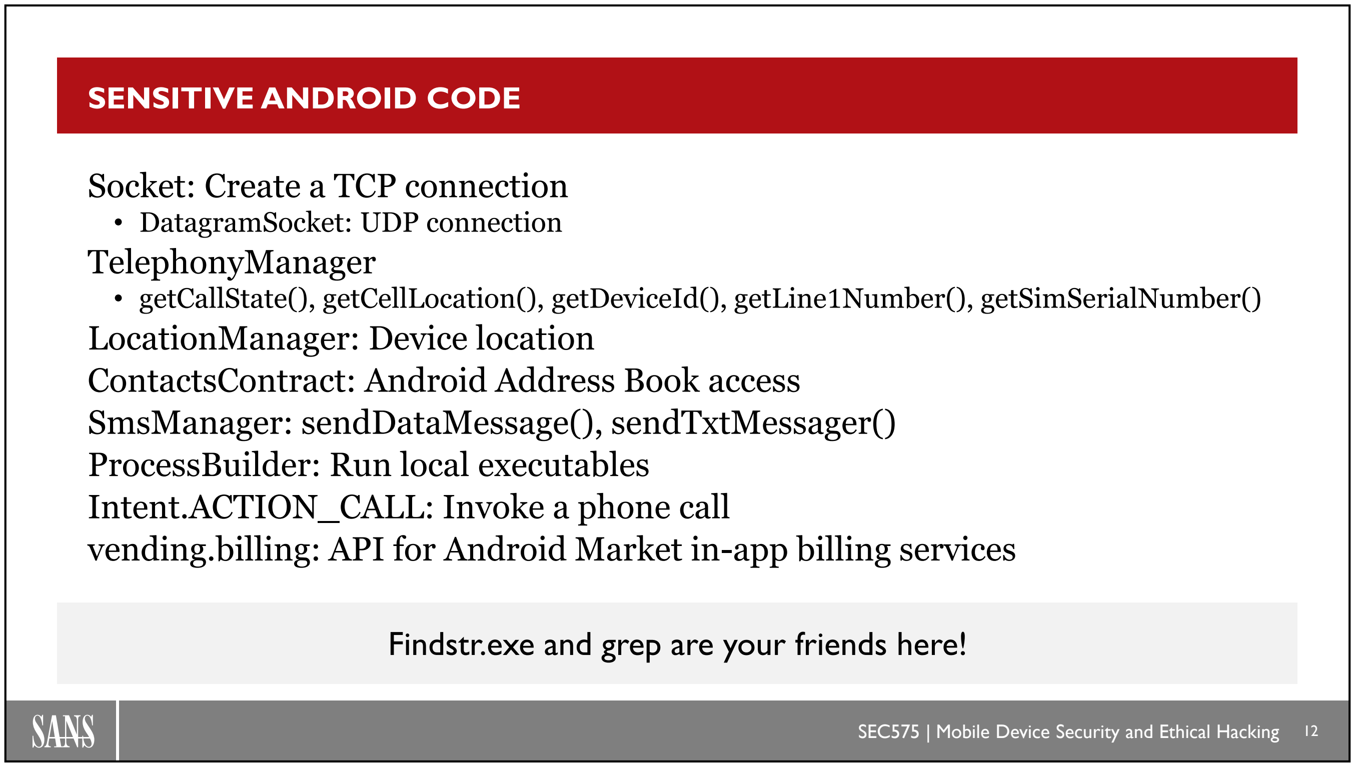
```
    }

    public void sendFeedback(View button) {
        OutputStreamWriter request;
        URL url;
        OutputStreamWriter r0_OutputStreamWriter = new OutputStreamWriter(System.out);
        try {
            request.close();
        } catch (IOException e) {
        }
        String name = ((EditText) findViewById(2131099650)).getText().toString();
        String email = ((EditText) findViewById(2131099651)).getText().toString();
        String feedback = ((EditText) findViewById(2131099653)).getText().toString();
        String feedbackType = ((Spinner) findViewById(2131099652)).getSelectedItem().t
        boolean bRequiresResponse = ((CheckBox) findViewById(2131099654)).isChecked();
        StringBuilder r24_StringBuilder = new StringBuilder("name=").append(name).appe
        Boolean r0_Boolean = new Boolean(bRequiresResponse);
        String parameters = r24_StringBuilder.append(r25_Boolean.toString()).toString(
        try {
```

Jadx Example

In the example on this page, we demonstrate the two uses of Jadx. First, from the command line, Jadx will recreate the source tree of the Android application from a starting directory point. Here we create an output directory to store the decompiled files and then use Jadx in the command line mode to reproduce the Android application source tree. The output directory will have several Java source code files (*.java) that can be viewed with any text editor or searched using `grep -R` *`searchterm`*.

In the second example, we launch the Jadx GUI and then browse to and select the APK file, allowing us to view the application package, class structure, and source code. The search functionality in the Jadx GUI is limited to searching the currently selected class file. If you want to search across the entire project with Jadx, decompile the APK file using the command line syntax and use the `grep` command to search.



SENSITIVE ANDROID CODE

Socket: Create a TCP connection
- DatagramSocket: UDP connection

TelephonyManager
- getCallState(), getCellLocation(), getDeviceId(), getLine1Number(), getSimSerialNumber()

LocationManager: Device location
ContactsContract: Android Address Book access
SmsManager: sendDataMessage(), sendTxtMessager()
ProcessBuilder: Run local executables
Intent.ACTION_CALL: Invoke a phone call
vending.billing: API for Android Market in-app billing services

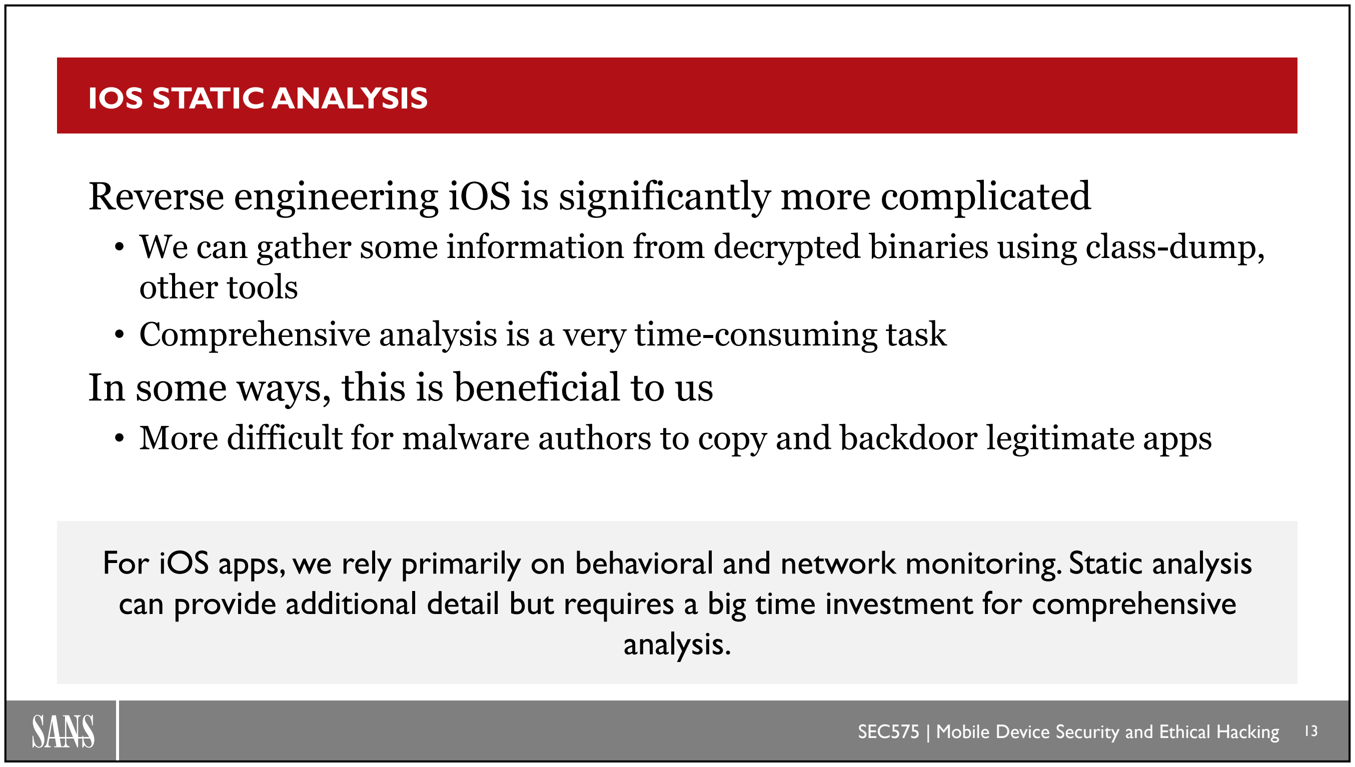
Findstr.exe and grep are your friends here!

Sensitive Android Code

After decompiling an Android app and viewing the sources with Jadx, there are several sensitive functions you can search for as a starting point for your analysis:

- **Socket/DatagramSocket:** Create a TCP or a UDP connection to an identified target by hostname or IP address.
- **TelephonyManager:** API for several sensitive functions.
  - **getCallState:** Determine whether the device is currently in a call.
  - **getCellLocation:** Get the location of the mobile device.
  - **getDeviceId:** Get the IMEI of the mobile device.
  - **getLine1Number:** Obtain the phone number of the mobile device.
  - **getSimSerialNumber:** Get the serial number of the SIM card.
- **LocationManager:** Provides additional access to location analysis services, including GPS latitude and longitude coordinates.
- **ContactsContract:** Interact with and manipulate Android contact book information.
- **SmsManager:** Send and retrieve SMS and MMS messages.
- **ProcessBuilder:** Run local executables on the platform.
- **Intent.ACTION_CALL:** A Java Intent for invoking a phone call on the mobile device.
- **vending.billing:** API for Android Market in-app billing services.

We can use findstr.exe, grep, or the Jadx search functionality to identify the presence of these functions and then evaluate the calling functions to determine the intent of using the sensitive data.



IOS STATIC ANALYSIS

Reverse engineering iOS is significantly more complicated

- We can gather some information from decrypted binaries using class-dump, other tools
- Comprehensive analysis is a very time-consuming task

In some ways, this is beneficial to us

- More difficult for malware authors to copy and backdoor legitimate apps

For iOS apps, we rely primarily on behavioral and network monitoring. Static analysis can provide additional detail but requires a big time investment for comprehensive analysis.

SANS

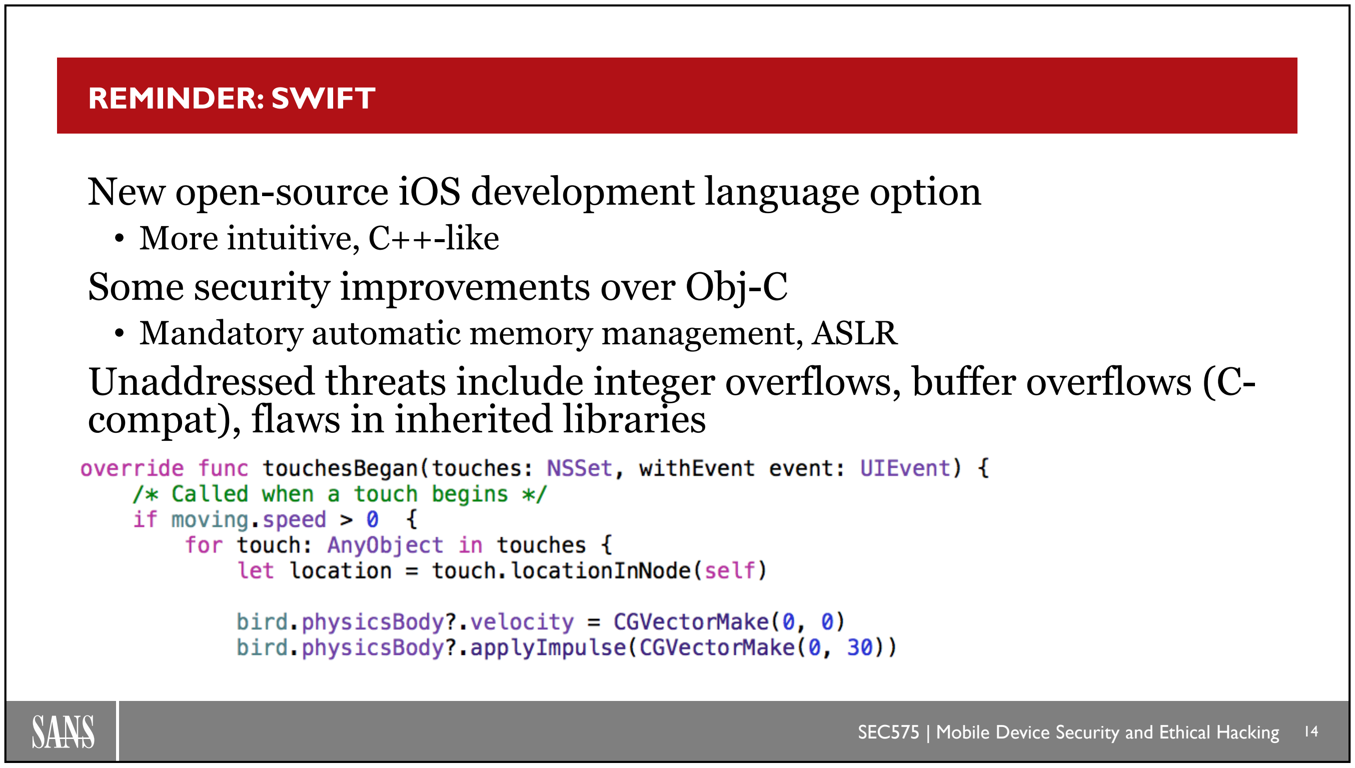
SEC575 | Mobile Device Security and Ethical Hacking 13

iOS Static Analysis

Reverse engineering in iOS is significantly more complex than reverse engineering Android apps. Some tools make this process easier, such as utilizing the data we can gather from strings, otool, and class-dump, but the analysis process is still very time-consuming and complex.

In some ways, the complexity of reverse engineering iOS binaries benefits the security of iOS systems because it is more difficult to take legitimate apps and add malicious code to them, and there is less theft of intellectual property from an application's source code. These factors serve as motivation for vendors to support iOS as their primary mobile device platform as opposed to Android.

For iOS apps, our primary analysis method is based on filesystem and network monitoring. Static analysis can be beneficial and might be the only option to evaluate binaries. Where the behavior to be evaluated isn't easily observed through network or filesystem monitoring, significant time and resource investment is required to return valuable results.



REMINDER: SWIFT

New open-source iOS development language option
- More intuitive, C++-like

Some security improvements over Obj-C
- Mandatory automatic memory management, ASLR

Unaddressed threats include integer overflows, buffer overflows (C-compat), flaws in inherited libraries

```
override func touchesBegan(touches: NSSet, withEvent event: UIEvent) {
    /* Called when a touch begins */
    if moving.speed > 0  {
        for touch: AnyObject in touches {
            let location = touch.locationInNode(self)

            bird.physicsBody?.velocity = CGVectorMake(0, 0)
            bird.physicsBody?.applyImpulse(CGVectorMake(0, 30))
```

SANS SEC575 | Mobile Device Security and Ethical Hacking 14

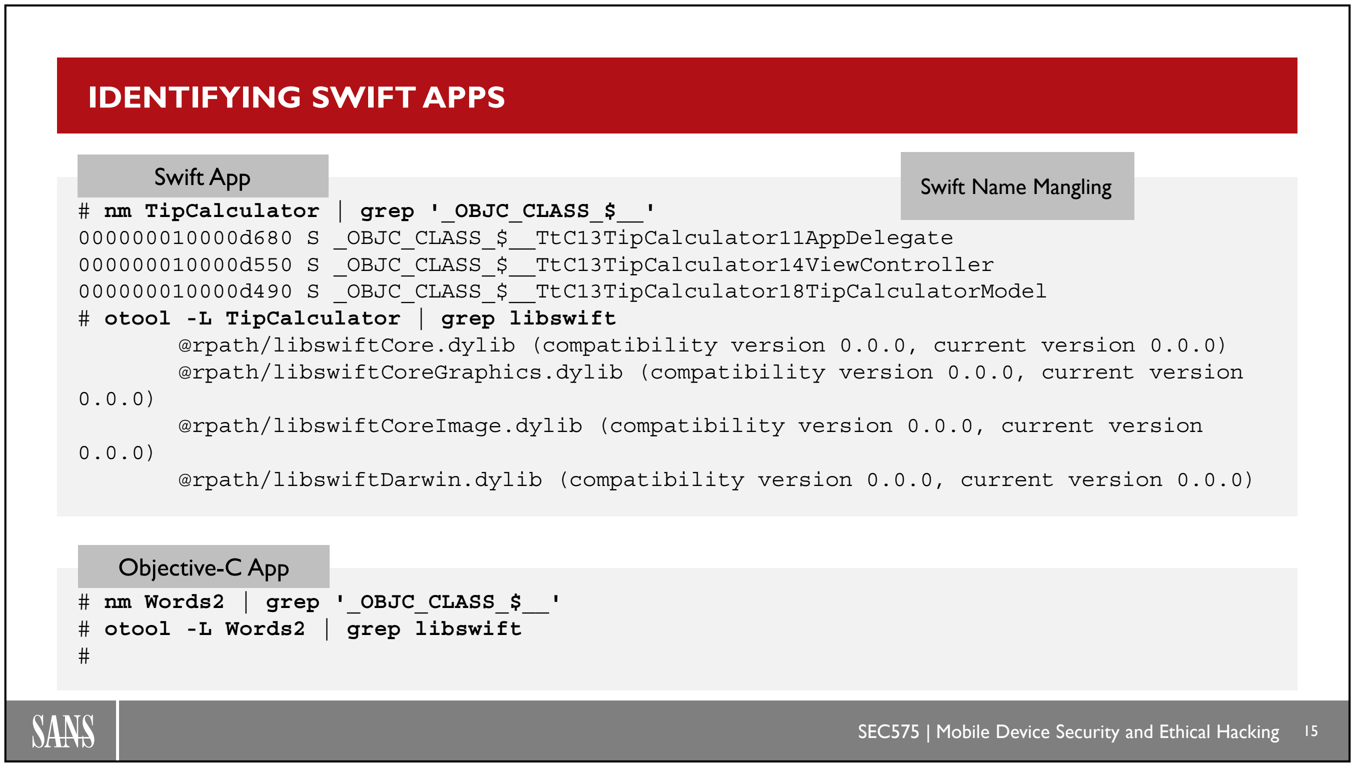
Reminder: Swift

A recent change in iOS development is the introduction of a second programming language for iOS and macOS applications: Swift. This new language option can be used in conjunction with Objective-C, depending on the developer's preference. At a high-level, Swift is more intuitive and C++-like, which will be attractive to new developers with C, C++, or Java experience who want to develop for iOS or OS X.

Swift also introduces some security enhancements over Objective-C. With Objective-C, memory management can be handled automatically or manually by the developer; manual memory management is more likely to be inconsistent, which can lead to exploitable vulnerabilities. Further, although Objective-C uses Address Space Layout Randomization for binaries, the use of ASLR is optional and can be disabled by a developer.

Swift improves memory management and ASLR support over Objective-C by mandating automatic memory management and ASLR support. However, Swift does not attempt to address other common threats, including integer overflows and buffer overflows in C-compatibility library functions. Code written native in Swift would not be exposed to buffer overflow threats, but a Swift application interacting with a legacy library does not offer protection against these threats.

The example code shown on this page is an excerpt from a Swift clone of Flappy Bird written by Nate Murray (https://github.com/fullstackio/FlappySwift/tree/master/FlappyBird; the original Flappy Bird app was written by Dong Nguyen). Swift code uses a function calling method that is C++-like (as opposed to message passing used in Objective-C) with the common "." notation for object access. Some of the Swift functionality is also influenced by more modern languages such as Python ("for touch: AnyObject in touches").



IDENTIFYING SWIFT APPS

Swift App

Swift Name Mangling

```
# nm TipCalculator | grep '_OBJC_CLASS_$__'
000000010000d680 S _OBJC_CLASS_$__TtC13TipCalculator11AppDelegate
000000010000d550 S _OBJC_CLASS_$__TtC13TipCalculator14ViewController
000000010000d490 S _OBJC_CLASS_$__TtC13TipCalculator18TipCalculatorModel
# otool -L TipCalculator | grep libswift
        @rpath/libswiftCore.dylib (compatibility version 0.0.0, current version 0.0.0)
        @rpath/libswiftCoreGraphics.dylib (compatibility version 0.0.0, current version
0.0.0)
        @rpath/libswiftCoreImage.dylib (compatibility version 0.0.0, current version
0.0.0)
        @rpath/libswiftDarwin.dylib (compatibility version 0.0.0, current version 0.0.0)
```

Objective-C App

```
# nm Words2 | grep '_OBJC_CLASS_$__'
# otool -L Words2 | grep libswift
#
```

SANS

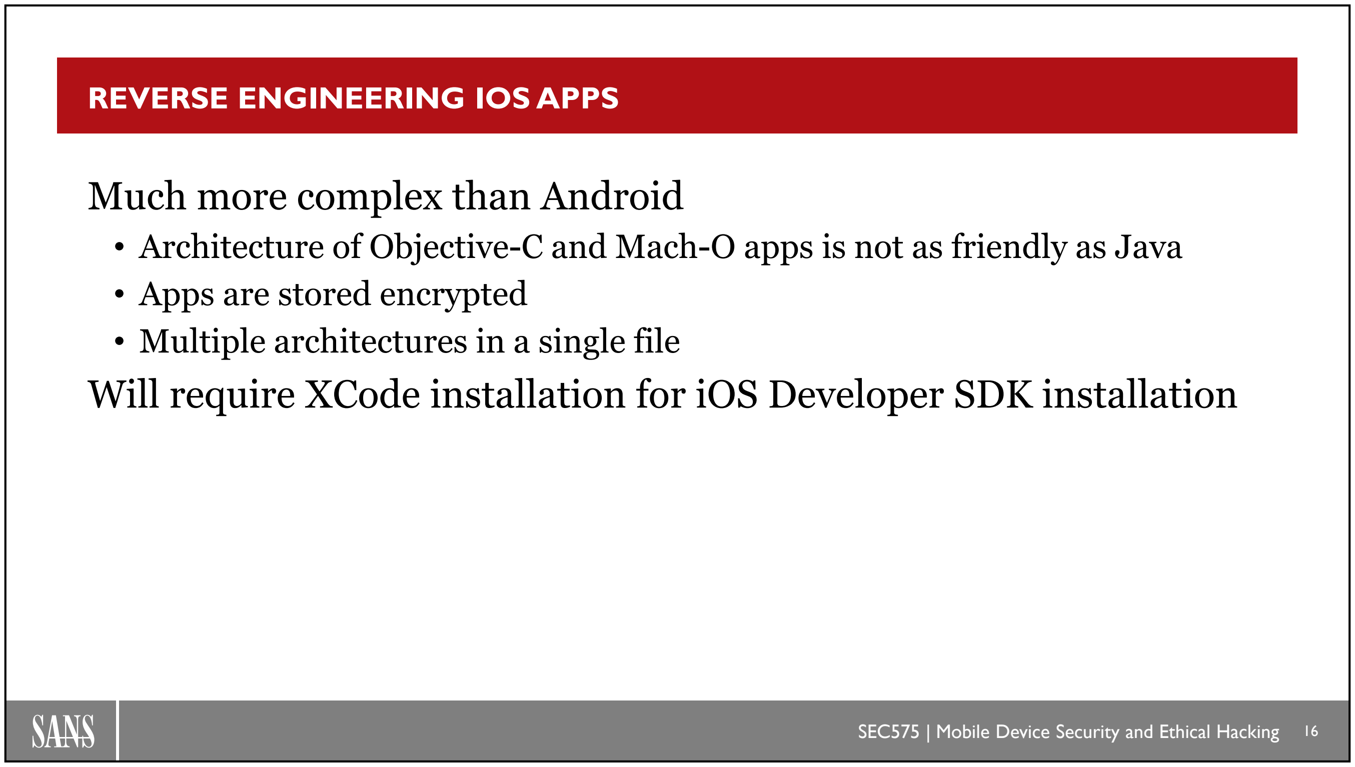
SEC575 | Mobile Device Security and Ethical Hacking 15

Identifying Swift Apps

On a jailbroken iOS device, we can evaluate an application binary to determine whether it was written in Swift or Objective-C using the "nm" and "otool" tools. In the first example on this page, the Swift compiled binary "TipCalculator" is evaluated with the nm tool, searching for the string "_OBJC_CLASS_$__". Despite the "OBJC" notation, this is a Swift binary, which we can see from the name mangling example "`__TtC13TipCalculator11AppDelegate`":

_	Mangled names start with an initial underscore.
`_T`	This is the marker for a Swift global symbol.
`tC`	This indicates that it is a *type class* object.
`13`	13 bytes follow that describe the class name "TipCalculator".
11	11 bytes follow that describe the method name "AppDelegate".
12	The otool utility cal also lists the linked libraries used by an executable with the "`-L`" argument, as shown. Searching for the string "libswift" indicates several linked libraries, further indicating that this is indeed a Swift executable.

The bottom example on this page shows the same technique to examine an Objective-C application; both "`grep`" commands return no matches for Objective-C programs.



REVERSE ENGINEERING IOS APPS

Much more complex than Android

- Architecture of Objective-C and Mach-O apps is not as friendly as Java
- Apps are stored encrypted
- Multiple architectures in a single file

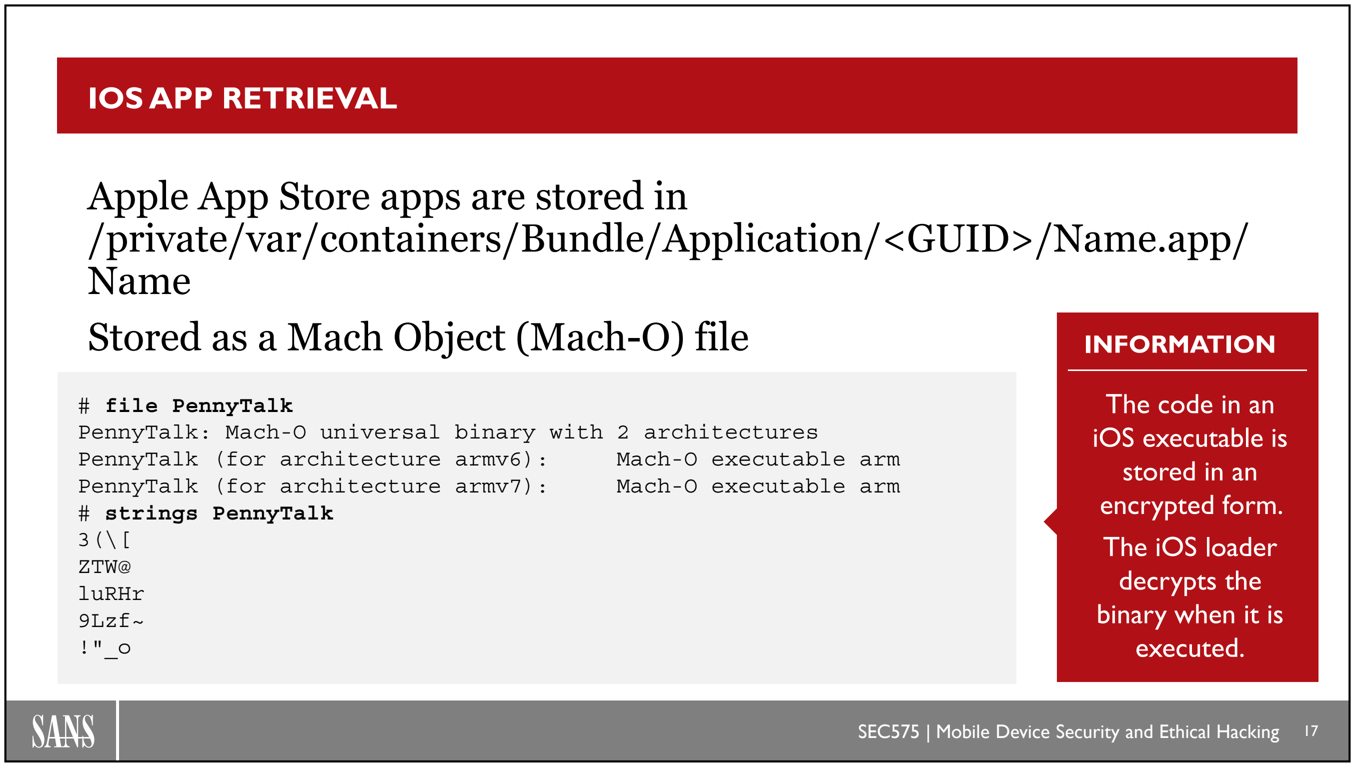
Will require XCode installation for iOS Developer SDK installation

SANS SEC575 | Mobile Device Security and Ethical Hacking 16

Reverse Engineering iOS Apps

Reverse engineering iOS apps is a much more complex process compared to the ease of obtaining Java sources for Android apps. This is primarily due to the architectural differences between the Objective-C language and the Mach-O executable architecture compared to Java and Dalvik files. In addition, iOS apps downloaded from the Apple App Store are stored in an encrypted format on the filesystem, combining binaries for multiple architectures into a single file.

We'll examine the tools and techniques available to us for reverse engineering of iOS apps next. Note that many of the techniques we'll examine will require a Mac and an installation of XCode to access the iOS Developer Software Development Kit (SDK).



IOS APP RETRIEVAL

Apple App Store apps are stored in /private/var/containers/Bundle/Application/<GUID>/Name.app/Name

Stored as a Mach Object (Mach-O) file

```
# file PennyTalk
PennyTalk: Mach-O universal binary with 2 architectures
PennyTalk (for architecture armv6):     Mach-O executable arm
PennyTalk (for architecture armv7):     Mach-O executable arm
# strings PennyTalk
3(\[
ZTW@
luRHr
9Lzf~
!"_o
```

INFORMATION

The code in an iOS executable is stored in an encrypted form. The iOS loader decrypts the binary when it is executed.

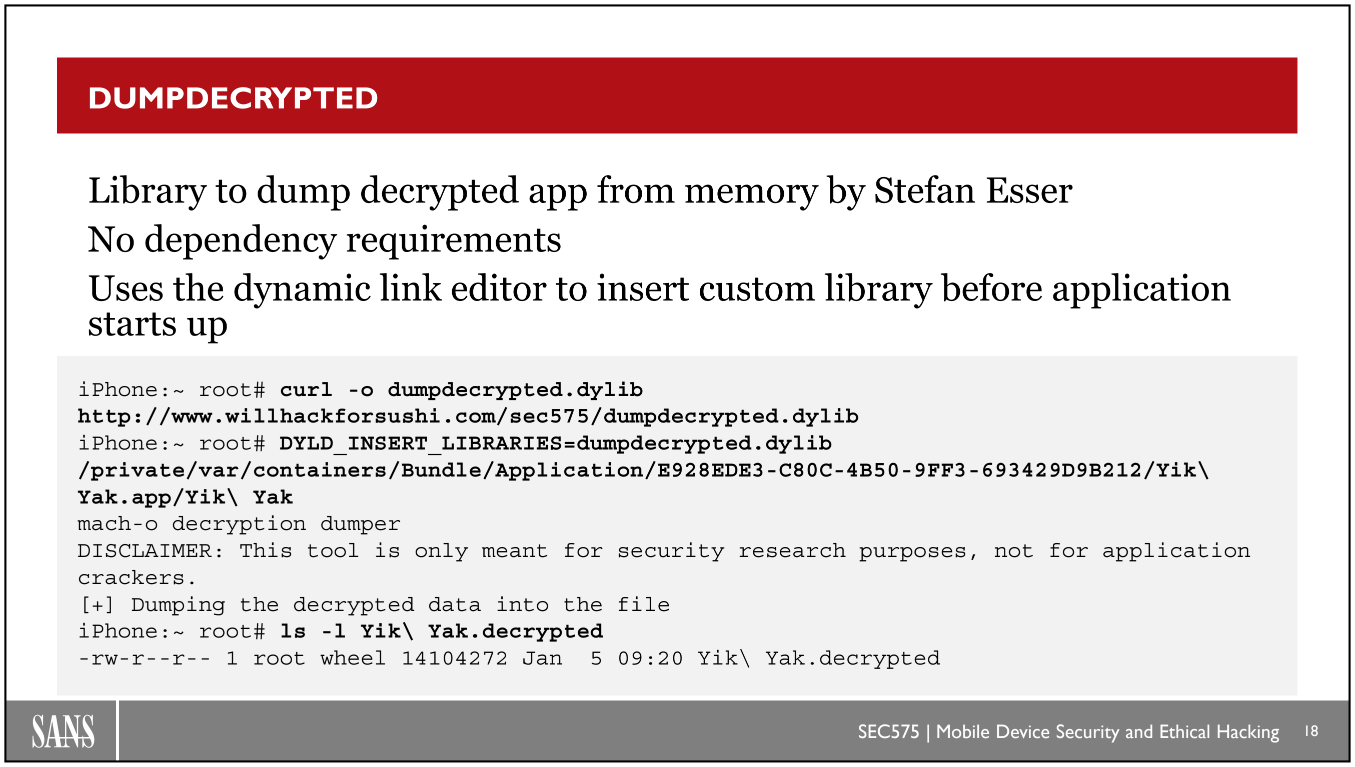
SEC575 | Mobile Device Security and Ethical Hacking 17

iOS App Retrieval

Apple App Store apps are stored on the iOS filesystem in `/private/var/containers/Bundle/Application/<GUID>`. Inside the GUID directory, the application is stored in a directory called "Name.app" where *Name* is the name of the application. Inside the Name.app directory is a file matching the directory name without the .app extension, representing the executable binary.

iOS binaries are stored as a Mach Object (Mach-O) file, which is a file format for executables accommodating multiple binaries in the same file designed for alternate platforms. Using the file utility, we can identify that the file uses the Mach-O format, and the executable binaries are stored in the file.

The executable is stored in an encrypted form in iOS, where it is decrypted when it is loaded by the operating system. In the stored format, however, it cannot be read and will not return useful data when we evaluate it with strings, as shown on this page.



DUMPDECRYPTED

Library to dump decrypted app from memory by Stefan Esser

No dependency requirements

Uses the dynamic link editor to insert custom library before application starts up

```
iPhone:~ root# curl -o dumpdecrypted.dylib
http://www.willhackforsushi.com/sec575/dumpdecrypted.dylib
iPhone:~ root# DYLD_INSERT_LIBRARIES=dumpdecrypted.dylib
/private/var/containers/Bundle/Application/E928EDE3-C80C-4B50-9FF3-693429D9B212/Yik\
Yak.app/Yik\ Yak
mach-o decryption dumper
DISCLAIMER: This tool is only meant for security research purposes, not for application
crackers.
[+] Dumping the decrypted data into the file
iPhone:~ root# ls -l Yik\ Yak.decrypted
-rw-r--r-- 1 root wheel 14104272 Jan  5 09:20 Yik\ Yak.decrypted
```

SANS SEC575 | Mobile Device Security and Ethical Hacking 18

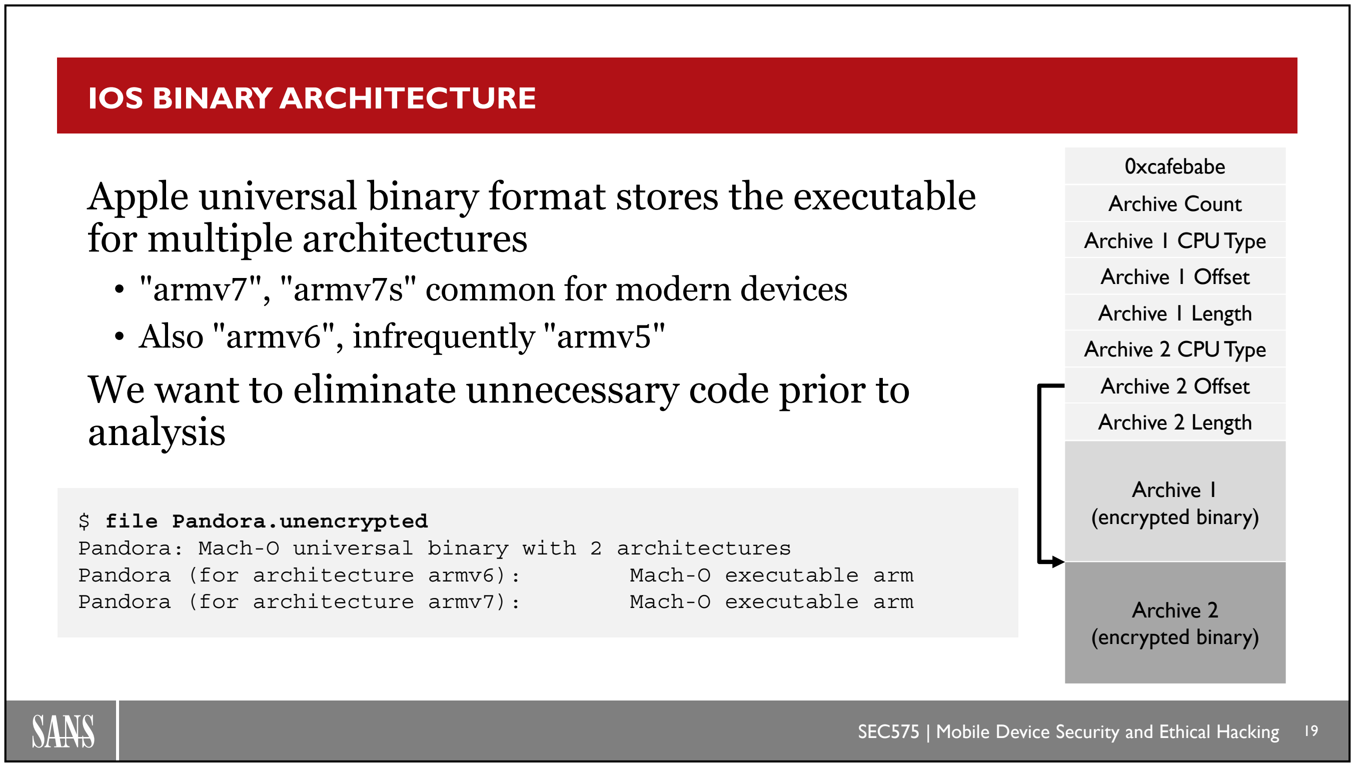
Dumpdecrypted

Dumpdecrypted is a library for iOS written by Stefan Esser, available at https://github.com/stefanesser/dumpdecrypted. Dumpdecrypted can be used to remove the encryption from an iOS executable without any extra dependencies or the manual steps associated with gdb.

Dumpdecrypted uses the iOS dynamic link editor to insert a custom library into the execution of a target application before the application starts up. By specifying a dynamic link library ("dylib" file) in the environment variable DYLD_INSERT_LIBRARIES, we can change the execution path of a target binary. Dumpdecrypted uses this feature on the command line, specifying the dumpdecrypted.dylib file following the DYLD_INSERT_LIBRARIES variable and then the name of an iOS executable target.

In the example on this page, the Yik Yak application is decrypted using Dumpdecrypted, specifying the path of the executable in the iOS containers/bundle directory as shown.

Dumpdecrypted is distributed as source code on the GitHub site. A compiled version of the library for iOS 9 and later targets is available on the author's website, as shown on this page.



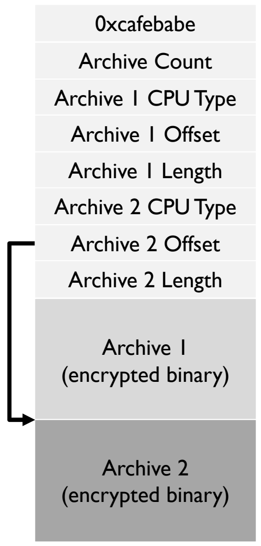
IOS BINARY ARCHITECTURE

Apple universal binary format stores the executable for multiple architectures

- "armv7", "armv7s" common for modern devices
- Also "armv6", infrequently "armv5"

We want to eliminate unnecessary code prior to analysis

```
$ file Pandora.unencrypted
Pandora: Mach-O universal binary with 2 architectures
Pandora (for architecture armv6):        Mach-O executable arm
Pandora (for architecture armv7):        Mach-O executable arm
```

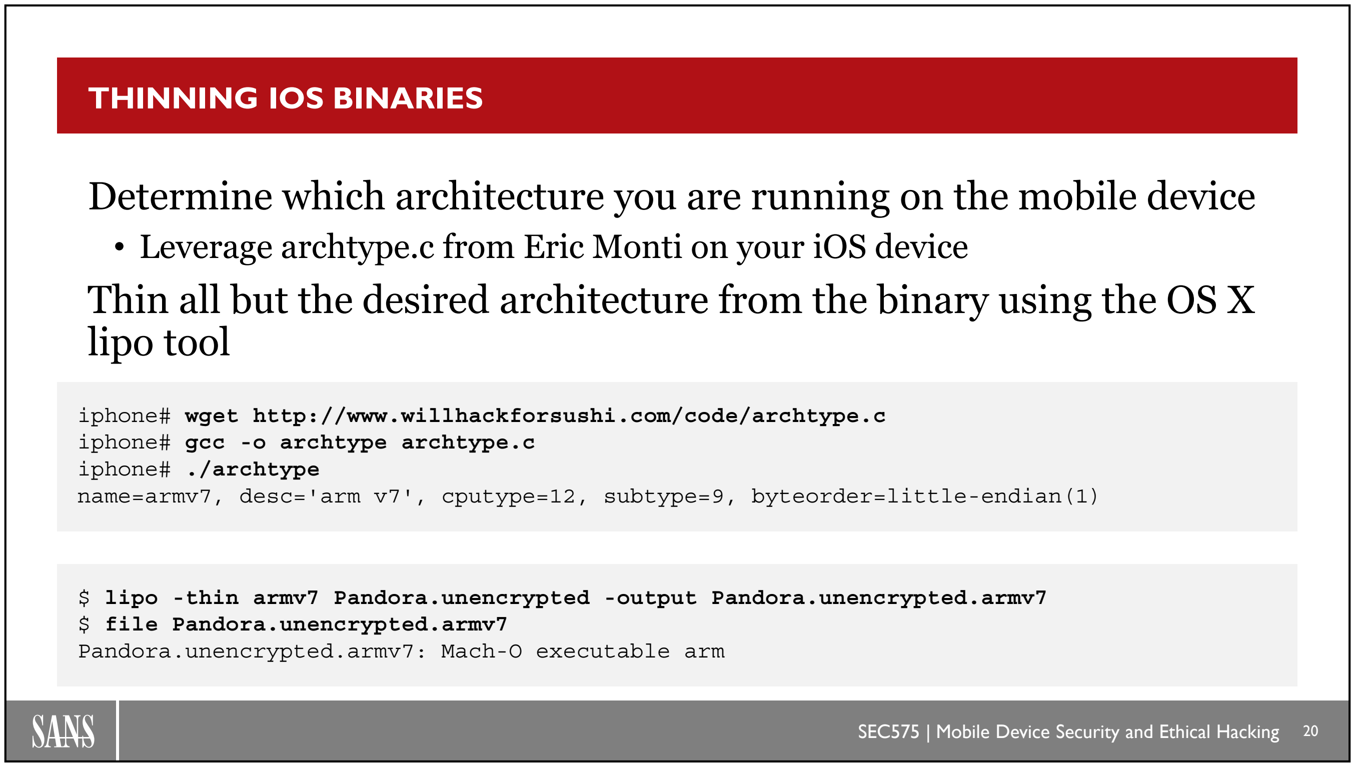


iOS Binary Architecture

As we've seen, Apple iOS stores executables in Mach-O format, embedding multiple executables in a single file for cross-platform compatibility. For modern apps, this typically represents a binary for the ARMv7 architecture and a second app for the more recent ARMv7s architecture (64-bit ARM), though it is possible to also see ARMv6 and even ARMv5 binaries in some apps.

The Mach-O file structure starts with a magic header of 0xcafebabe, followed by a numeric field indicating the number of binaries present in the file. For each archive, the CPU type and a numeric offset to the beginning of the executable binary for the specified CPU type is listed, as well as the length of the binary. This allows iOS (and other platforms using the Mach-O format such as OS X) to embed multiple binaries in the same file.

From a reverse engineering perspective, we want to eliminate as much complexity from the file as possible. If we are running the binary using an ARMv7s processor, then the ARMv6 binary has little meaning to us and should be removed to eliminate unrelated content from the target.



THINNING IOS BINARIES

Determine which architecture you are running on the mobile device

- Leverage archtype.c from Eric Monti on your iOS device

Thin all but the desired architecture from the binary using the OS X lipo tool

```
iphone# wget http://www.willhackforsushi.com/code/archtype.c
iphone# gcc -o archtype archtype.c
iphone# ./archtype
name=armv7, desc='arm v7', cputype=12, subtype=9, byteorder=little-endian(1)
```

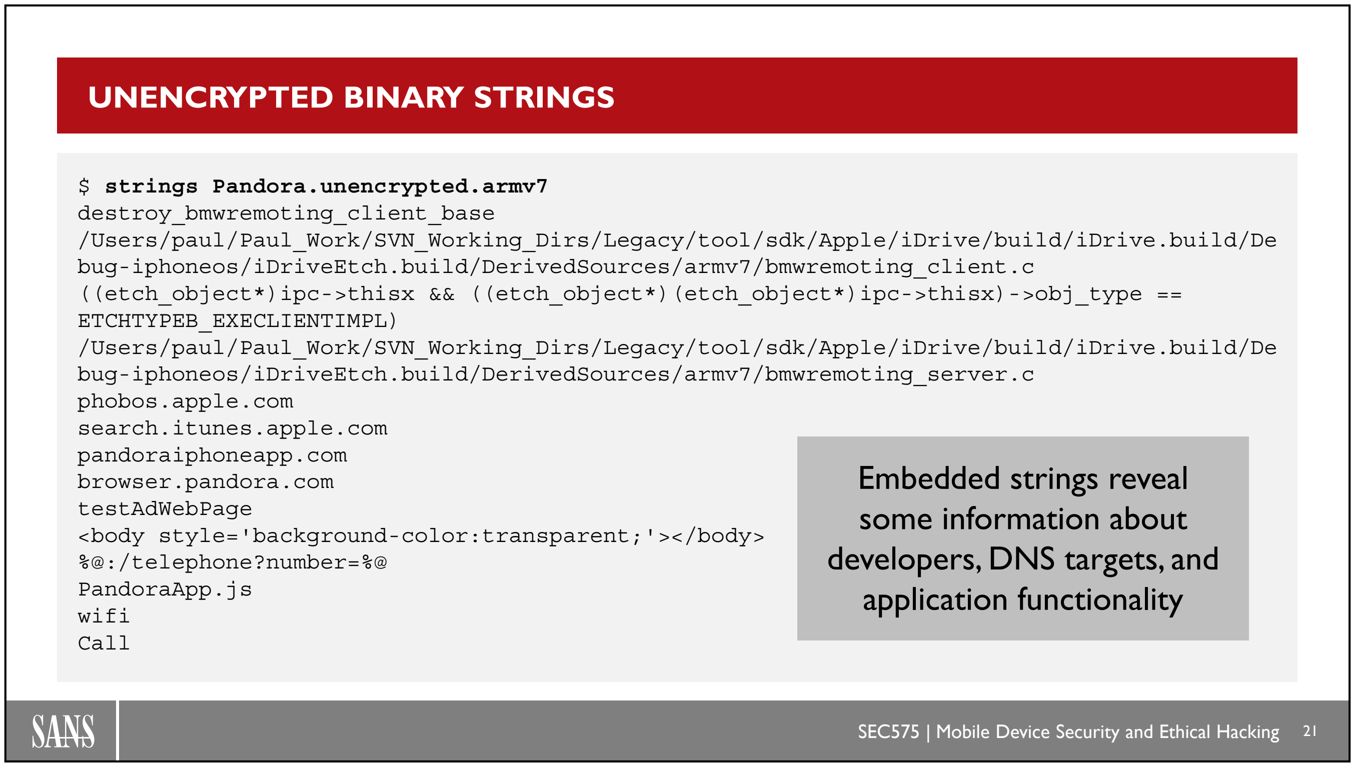
```
$ lipo -thin armv7 Pandora.unencrypted -output Pandora.unencrypted.armv7
$ file Pandora.unencrypted.armv7
Pandora.unencrypted.armv7: Mach-O executable arm
```

SEC575 | Mobile Device Security and Ethical Hacking 20

Thinning iOS Binaries

We can remove unnecessary binaries from Mach-O files using the lipo tool included on OS X systems with the XCode package installed. First, determine which CPU platform is in use on your iOS device by retrieving and compiling the simple archtype.c tool from Eric Monti. In the example on this page, running archtype on a first-generation iPad reveals that the processor type is ARMv7.

Referencing the unencrypted application on OS X, we can remove (or "thin out") all binary architectures except the target platform using the lipo tool, as shown. The resulting binary is unencrypted and includes only the appropriate target binary for our platform, which we'll use for our analysis.



UNENCRYPTED BINARY STRINGS

```
$ strings Pandora.unencrypted.armv7
destroy_bmwremoting_client_base
/Users/paul/Paul_Work/SVN_Working_Dirs/Legacy/tool/sdk/Apple/iDrive/build/iDrive.build/De
bug-iphoneos/iDriveEtch.build/DerivedSources/armv7/bmwremoting_client.c
((etch_object*)ipc->thisx && ((etch_object*)(etch_object*)ipc->thisx)->obj_type ==
ETCHTYPEB_EXECLIENTIMPL)
/Users/paul/Paul_Work/SVN_Working_Dirs/Legacy/tool/sdk/Apple/iDrive/build/iDrive.build/De
bug-iphoneos/iDriveEtch.build/DerivedSources/armv7/bmwremoting_server.c
phobos.apple.com
search.itunes.apple.com
pandoraiphoneapp.com
browser.pandora.com
testAdWebPage
<body style='background-color:transparent;'></body>
%@:/telephone?number=%@
PandoraApp.js
wifi
Call
```

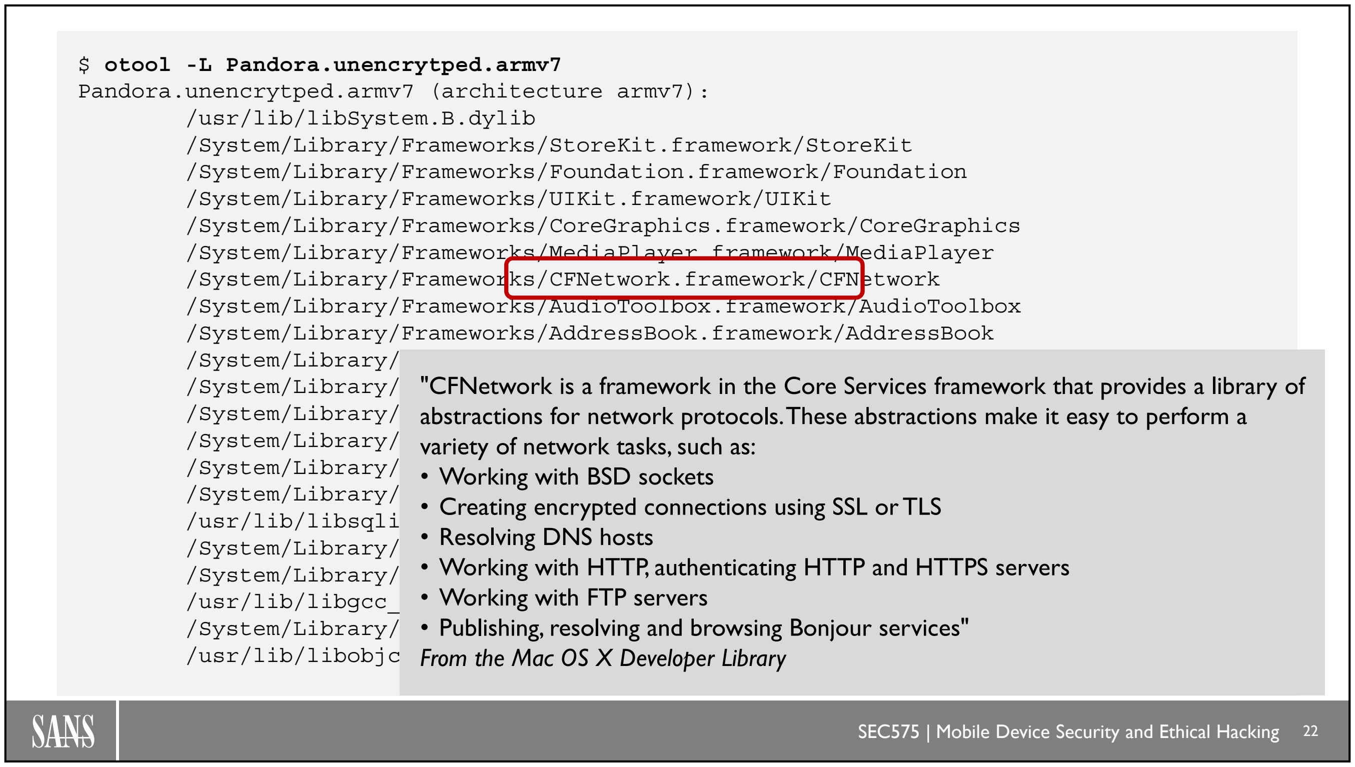
Embedded strings reveal some information about developers, DNS targets, and application functionality

SANS

SEC575 | Mobile Device Security and Ethical Hacking 21

Unencrypted Binary Strings

As noted with Dalvik files and other data, we can use the strings command to extract binary string content from the application. In the output on this slide, while evaluating the unencrypted Pandora application thinned for the ARMv7 architecture, we see several strings referencing the developer's directory structure that were embedded in the binary, as well as HTML code and several hosts referenced by the app.



```
$ otool -L Pandora.unencrytped.armv7
Pandora.unencrytped.armv7 (architecture armv7):
        /usr/lib/libSystem.B.dylib
        /System/Library/Frameworks/StoreKit.framework/StoreKit
        /System/Library/Frameworks/Foundation.framework/Foundation
        /System/Library/Frameworks/UIKit.framework/UIKit
        /System/Library/Frameworks/CoreGraphics.framework/CoreGraphics
        /System/Library/Frameworks/MediaPlayer.framework/MediaPlayer
        /System/Library/Frameworks/CFNetwork.framework/CFNetwork
        /System/Library/Frameworks/AudioToolbox.framework/AudioToolbox
        /System/Library/Frameworks/AddressBook.framework/AddressBook
        /System/Library/
        /System/Library/
        /System/Library/
        /System/Library/
        /System/Library/
        /System/Library/
        /usr/lib/libsqli
        /System/Library/
        /System/Library/
        /usr/lib/libgcc_
        /System/Library/
        /usr/lib/libobjc
```

"CFNetwork is a framework in the Core Services framework that provides a library of abstractions for network protocols. These abstractions make it easy to perform a variety of network tasks, such as:
- Working with BSD sockets
- Creating encrypted connections using SSL or TLS
- Resolving DNS hosts
- Working with HTTP, authenticating HTTP and HTTPS servers
- Working with FTP servers
- Publishing, resolving and browsing Bonjour services"

From the Mac OS X Developer Library

SANS SEC575 | Mobile Device Security and Ethical Hacking 22

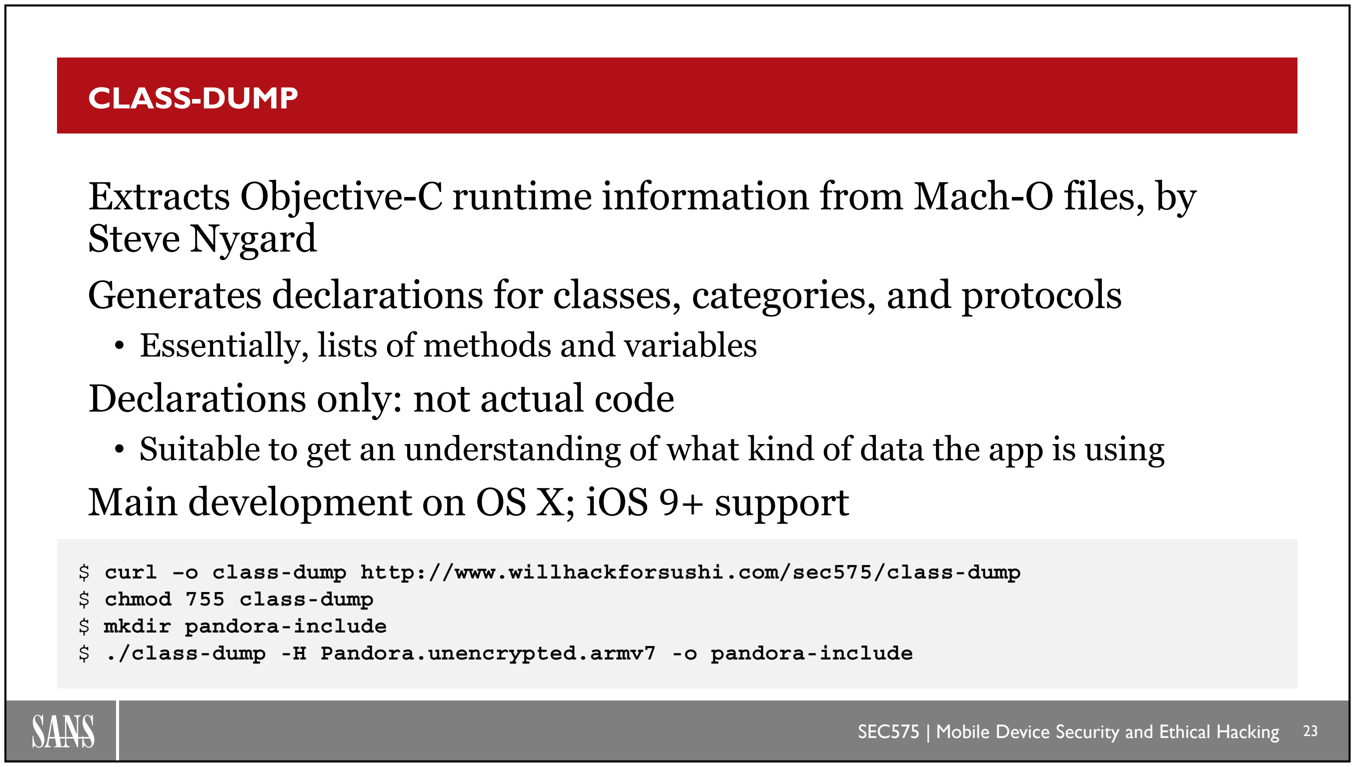
otool: Library Disclosure

We can return to the otool tool to identify the linked libraries associated with the unencrypted binary. These libraries represent functionality that is included in the binary, implemented via external binary files. We cannot determine exactly what functions are used in these libraries, but the opportunity exists for the developer to use them based on the library's functionality.

For example, the Pandora app reports that it uses the CFNetwork.framework/CFNetwork library. The Mac OS X Developer Library reports that this library is used for common network tasks, including socket connections and upper-layer protocols. Other interesting libraries for Pandora include access to the AddressBook.framework/AddressBook library, which is used to retrieve, modify, and create Address Book application entries.

These two libraries might provide legitimate functionality for the Pandora app, as it requires network access and the ability to share stations with other contacts in your Address Book. That said, this same functionality might be disconcerting in other applications, such as games or personal productivity tools.

A reference for iOS frameworks that can be evaluated to identify the functionality of a framework identified with otool is available at https://developer.apple.com/documentation/.



CLASS-DUMP

Extracts Objective-C runtime information from Mach-O files, by Steve Nygard

Generates declarations for classes, categories, and protocols

- Essentially, lists of methods and variables

Declarations only: not actual code

- Suitable to get an understanding of what kind of data the app is using

Main development on OS X; iOS 9+ support

```
$ curl -o class-dump http://www.willhackforsushi.com/sec575/class-dump
$ chmod 755 class-dump
$ mkdir pandora-include
$ ./class-dump -H Pandora.unencrypted.armv7 -o pandora-include
```

SANS SEC575 | Mobile Device Security and Ethical Hacking 23

class-dump

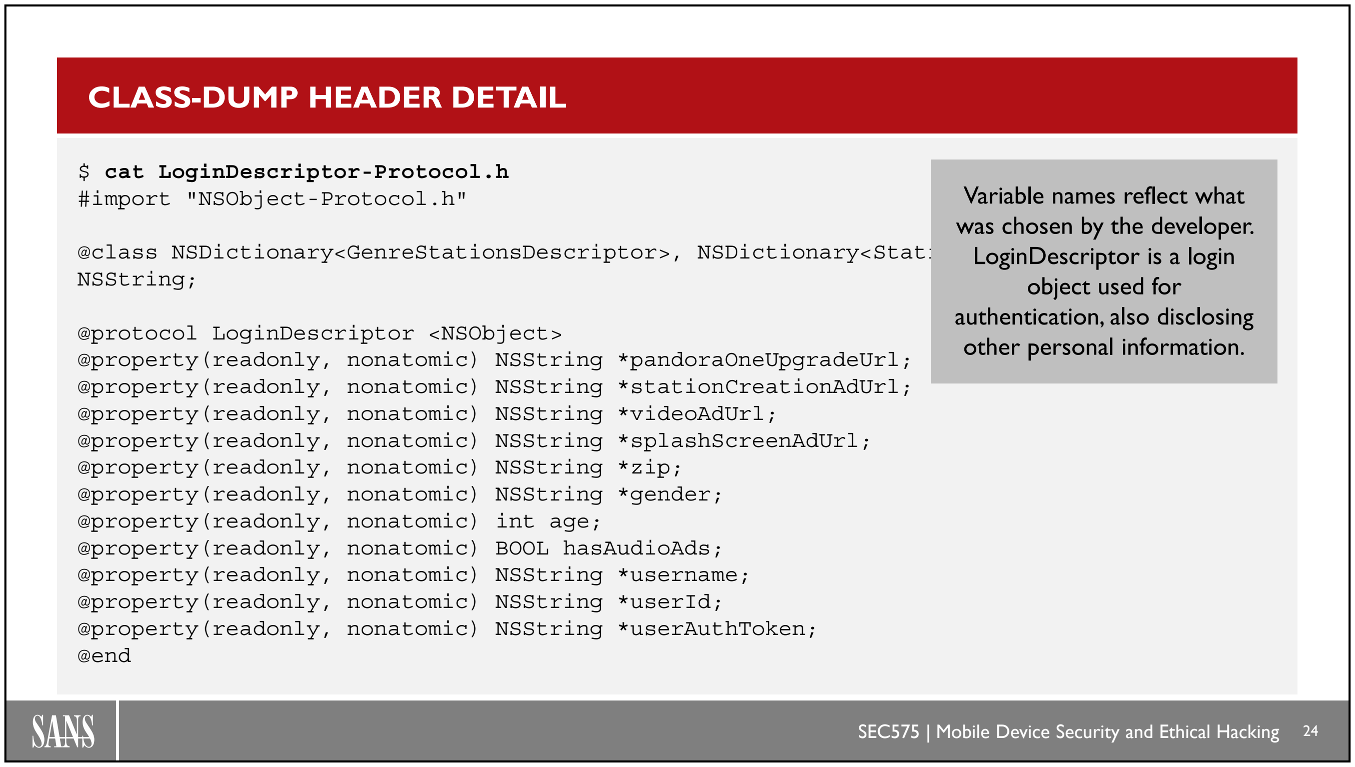
The class-dump tool is an open-source project by Steve Nygard to extract Objective-C runtime information from Mach-O files, available at http://stevenygard.com/projects/class-dump/. Using class-dump, we can evaluate an unencrypted iOS binary and generate Objective-C-compatible header files describing declarations for classes, categories, and protocols (essentially, various Objective-C functions that might call methods in other languages).

Note that due to the nature of the compiled binaries, class-dump is not able to produce source code in the same manner as Java decompilers and instead is limited to extracting only the function and header declarations. The data that class-dump is able to obtain is limited, but it gives us an idea as to the kind of data the app is collecting and using.

The class-dump tool is developed and primarily supported for OS X systems. Although some derivative versions of class-dump exist for Windows and Linux systems, these versions generally do not work well or do not support the latest versions of the iOS SDK.

Using class-dump is straightforward: create a directory where you want the class-dump data to be stored in, then run class-dump, specifying the Mach-O Objective-C binary with the "-H" parameter and the output directory with the "-o" parameter, as shown.

You can download the source code to class-dump from the GitHub page at https://github.com/nygard/class-dump. At the time of this writing, the latest release of class-dump (version 3.5) does not support the latest iOS release. A modified version of class-dump built to accommodate iOS 9 and later applications is available on the author's website at http://www.willhackforsushi.com/sec575/class-dump. This binary file is built for OS X 64-bit systems.



CLASS-DUMP HEADER DETAIL

```
$ cat LoginDescriptor-Protocol.h
#import "NSObject-Protocol.h"

@class NSDictionary<GenreStationsDescriptor>, NSDictionary<Stat
NSString;

@protocol LoginDescriptor <NSObject>
@property(readonly, nonatomic) NSString *pandoraOneUpgradeUrl;
@property(readonly, nonatomic) NSString *stationCreationAdUrl;
@property(readonly, nonatomic) NSString *videoAdUrl;
@property(readonly, nonatomic) NSString *splashScreenAdUrl;
@property(readonly, nonatomic) NSString *zip;
@property(readonly, nonatomic) NSString *gender;
@property(readonly, nonatomic) int age;
@property(readonly, nonatomic) BOOL hasAudioAds;
@property(readonly, nonatomic) NSString *username;
@property(readonly, nonatomic) NSString *userId;
@property(readonly, nonatomic) NSString *userAuthToken;
@end
```

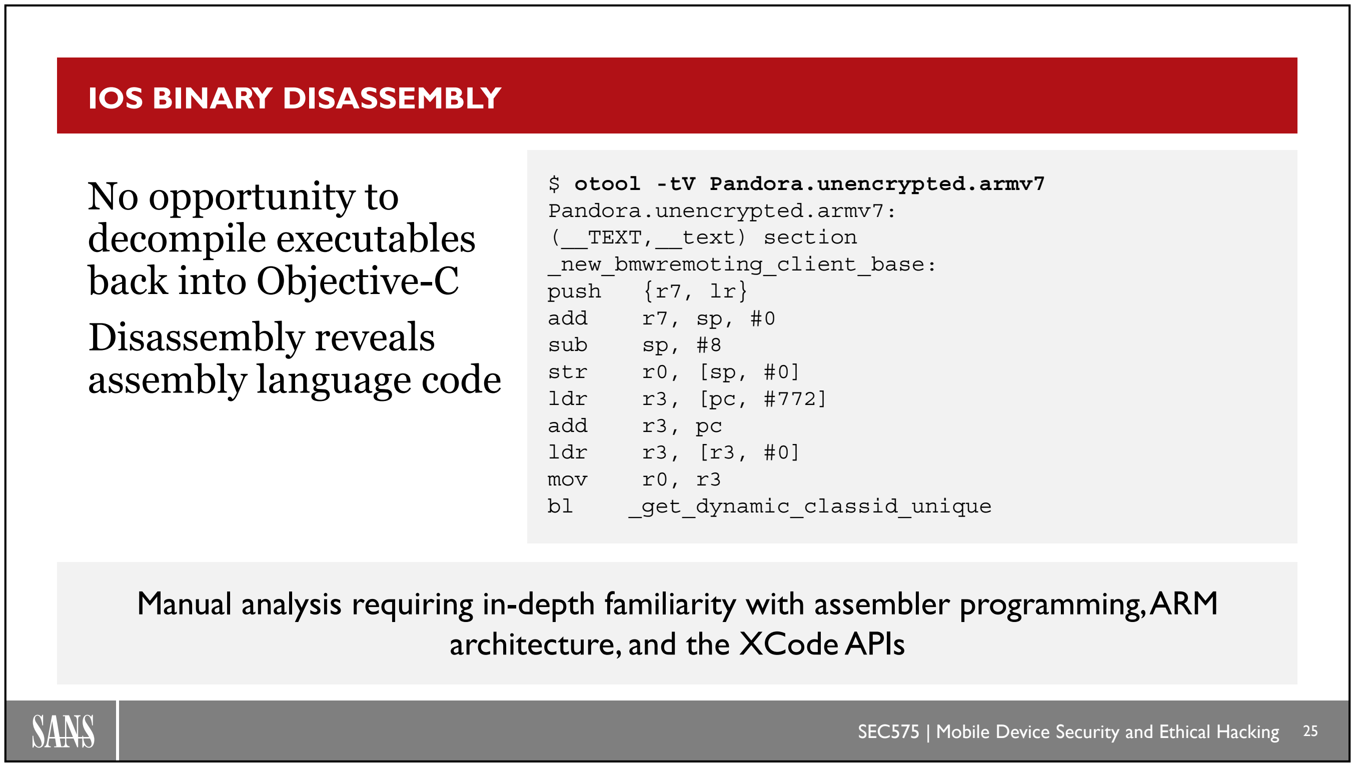
Variable names reflect what was chosen by the developer. LoginDescriptor is a login object used for authentication, also disclosing other personal information.

SANS

SEC575 | Mobile Device Security and Ethical Hacking 24

class-dump Header Detail

This page shows the output from class-dump analysis of the Pandora binary. class-dump generated several header files named for the import header used by Objective-C. In the class-dump output, the header detail matches what the original developer wrote with relative accuracy, describing the classes, protocols, and properties. In the output on this page, we see that the Pandora class for "GenreStationDescriptor" includes several variables, including zip, gender, age, and username information. This might indicate some sort of data collection or transmission in the app that includes the above information when it is accessible to the app.



IOS BINARY DISASSEMBLY

No opportunity to decompile executables back into Objective-C

Disassembly reveals assembly language code

```
$ otool -tV Pandora.unencrypted.armv7
Pandora.unencrypted.armv7:
(__TEXT,__text) section
_new_bmwremoting_client_base:
push    {r7, lr}
add     r7, sp, #0
sub     sp, #8
str     r0, [sp, #0]
ldr     r3, [pc, #772]
add     r3, pc
ldr     r3, [r3, #0]
mov     r0, r3
bl      _get_dynamic_classid_unique
```

Manual analysis requiring in-depth familiarity with assembler programming, ARM architecture, and the XCode APIs

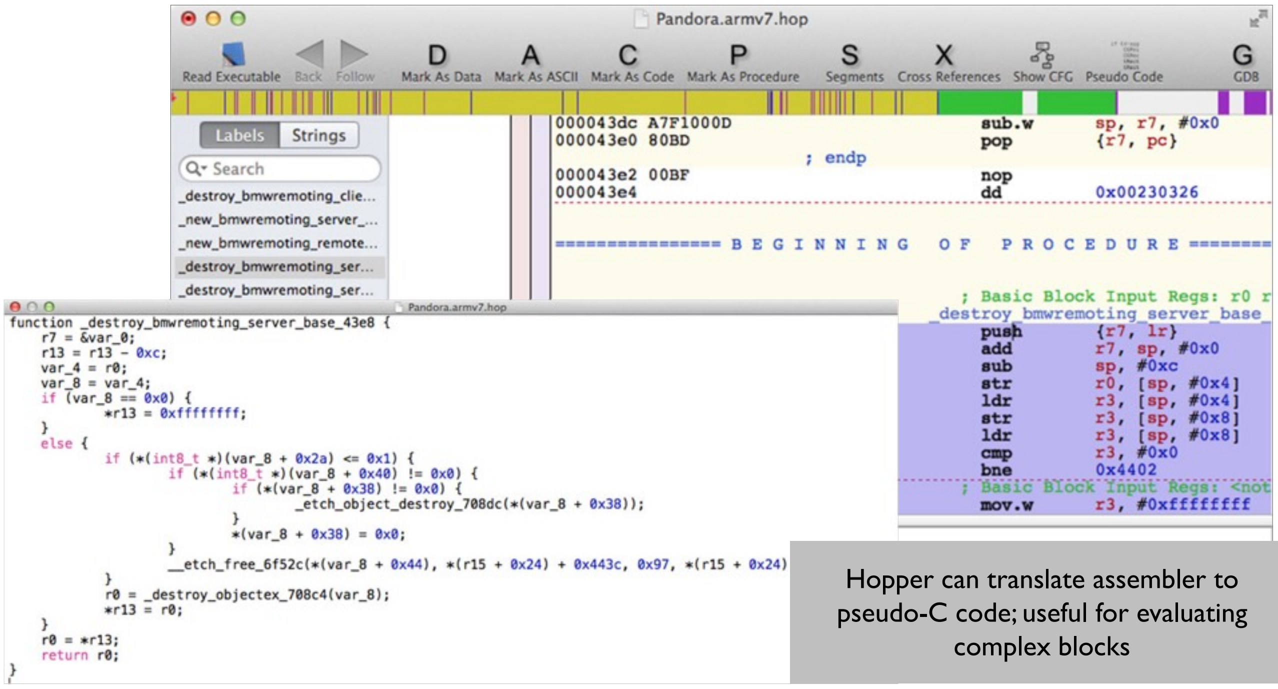
SANS

SEC575 | Mobile Device Security and Ethical Hacking 25

iOS Binary Disassembly

Unlike Dalvik files, there is no opportunity to take an iOS Mach-O binary and extract Objective-C source. Our only option for reverse engineering the actual code of an iOS app is to disassemble the code into low-level assembly language code. We can extract the assembly code using otool with the "-tV" argument, as shown on this page. The otool output will attempt to include the appropriate labels and variable names, wherever accessible.

Evaluating the functionality of a binary by reviewing the assembly language instructions is a very time-consuming task, requiring an in-depth familiarity with assembly language programming, ARM processor architecture, and the XCode APIs. This level of analysis might be outside of the scope of your analysis, though it might be the only alternative to obtain a comprehensive analysis of the functionality of an application.



```
000043dc A7F1000D                       sub.w      sp, r7, #0x0
000043e0 80BD                           pop        {r7, pc}
                           ; endp
000043e2 00BF                           nop
000043e4                                dd         0x00230326

================ B E G I N N I N G   O F   P R O C E D U R E ========

                                  ; Basic Block Input Regs: r0 r
                              _destroy_bmwremoting_server_base_
                                        push       {r7, lr}
                                        add        r7, sp, #0x0
                                        sub        sp, #0xc
                                        str        r0, [sp, #0x4]
                                        ldr        r3, [sp, #0x4]
                                        str        r3, [sp, #0x8]
                                        ldr        r3, [sp, #0x8]
                                        cmp        r3, #0x0
                                        bne        0x4402
                                  ; Basic Block Input Regs: <not
                                        mov.w      r3, #0xffffffff
```

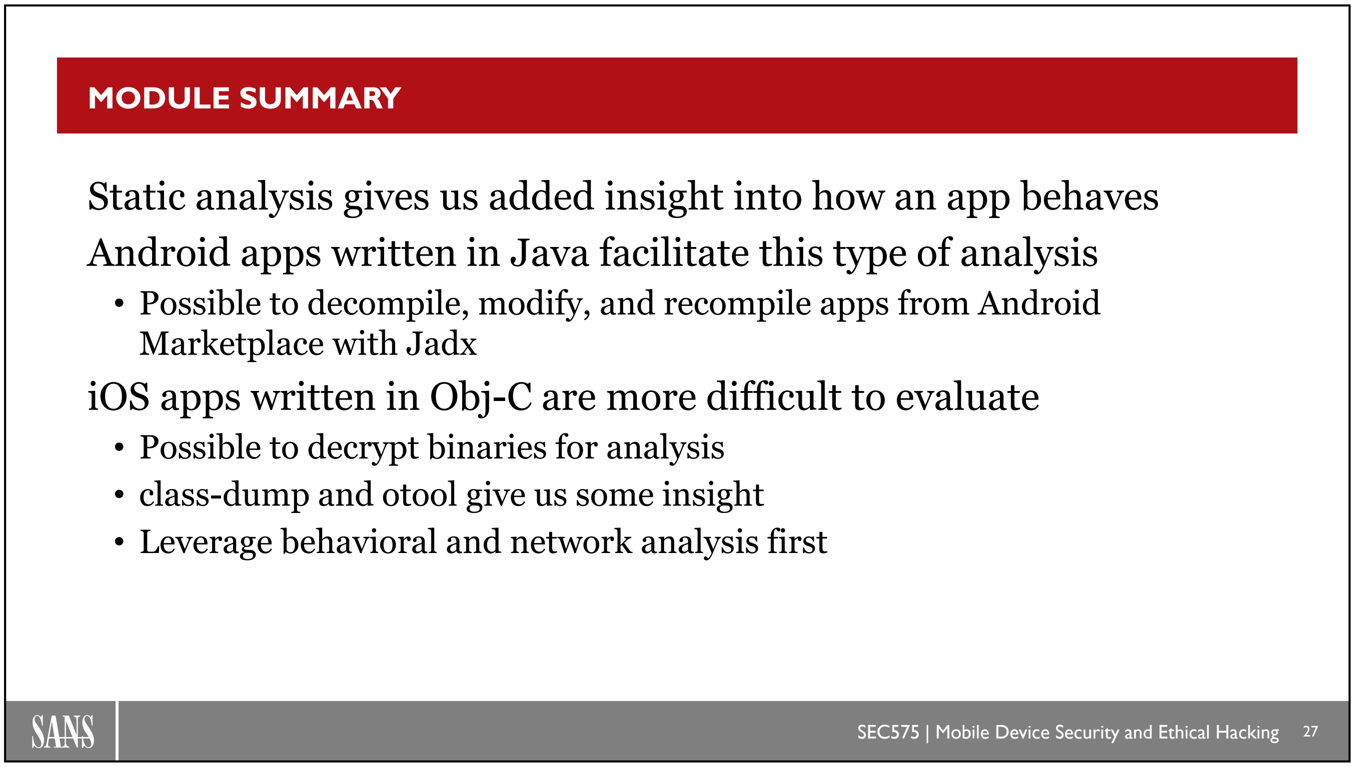
```
function _destroy_bmwremoting_server_base_43e8 {
    r7 = &var_0;
    r13 = r13 - 0xc;
    var_4 = r0;
    var_8 = var_4;
    if (var_8 == 0x0) {
            *r13 = 0xffffffff;
    }
    else {
            if (*(int8_t *)(var_8 + 0x2a) <= 0x1) {
                    if (*(int8_t *)(var_8 + 0x40) != 0x0) {
                            if (*(var_8 + 0x38) != 0x0) {
                                    _etch_object_destroy_708dc(*(var_8 + 0x38));
                            }
                            *(var_8 + 0x38) = 0x0;
                    }
                    __etch_free_6f52c(*(var_8 + 0x44), *(r15 + 0x24) + 0x443c, 0x97, *(r15 + 0x24)
            }
            r0 = _destroy_objectex_708c4(var_8);
            *r13 = r0;
    }
    r0 = *r13;
    return r0;
}
```

Hopper can translate assembler to pseudo-C code; useful for evaluating complex blocks

Hopper

Hopper is a commercial disassembler available for OS X systems for $89 (www.hopperapp.com). A feature-rich disassembler, Hopper implements a useful interface to disassemble Intel and ARM code, allowing you to quickly navigate through the blocks. With integrated commenting and documentation tools, Hopper parallels many of the features of the more costly Ida Pro tool for a tenth of the price.

Hopper has an interesting feature where any block of instructions can be translated to a pseudo-C block of code, as shown on this page. This is a useful feature when evaluating complex blocks with lots of branching, reducing the time required for analysis.



MODULE SUMMARY

Static analysis gives us added insight into how an app behaves

Android apps written in Java facilitate this type of analysis

- Possible to decompile, modify, and recompile apps from Android Marketplace with Jadx

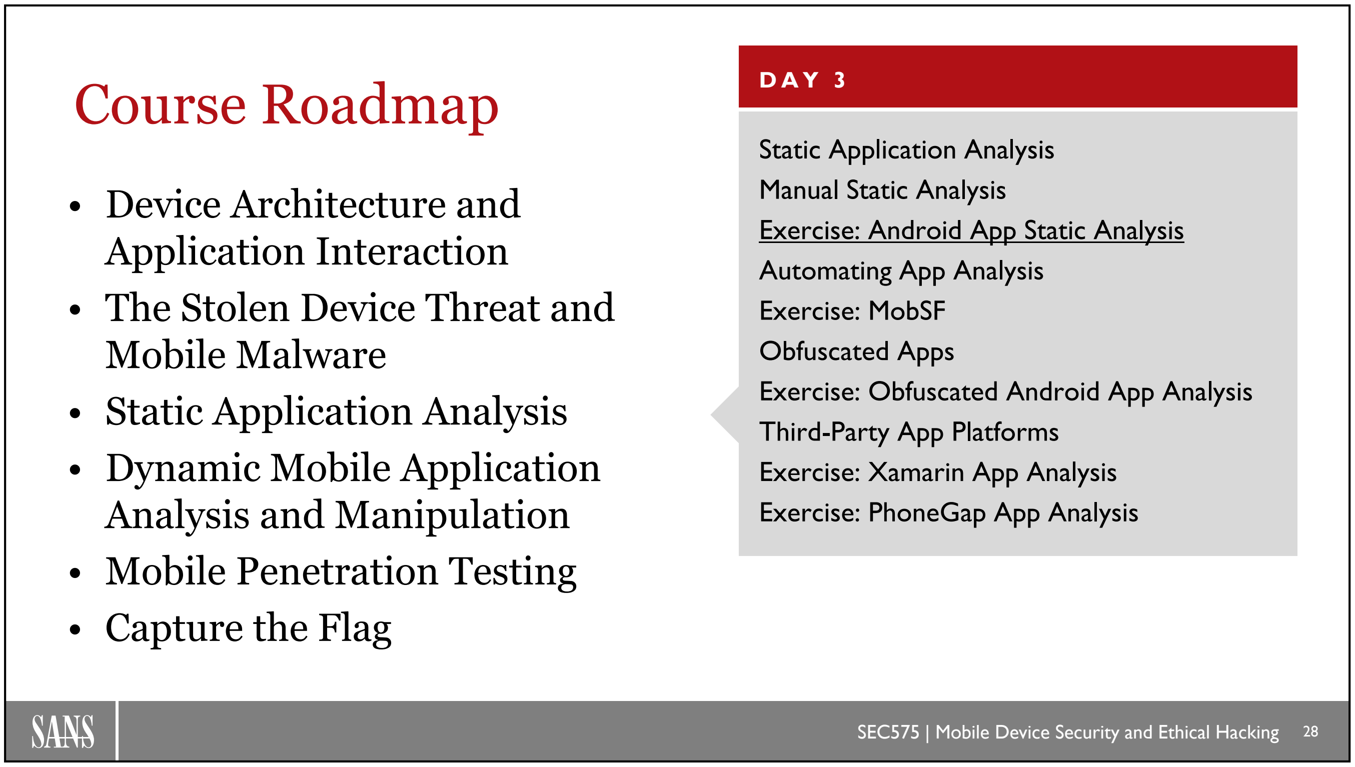
iOS apps written in Obj-C are more difficult to evaluate

- Possible to decrypt binaries for analysis
- class-dump and otool give us some insight
- Leverage behavioral and network analysis first

Module Summary

In this module, we reviewed techniques to perform static analysis on mobile device binaries. For Android applications written in Java, Jadx allows us to easily decompile Dalvik files for analysis. Accordingly, attackers can also decompile Dalvik files, creating modified versions that include malicious code.

On Apple iOS platforms, Objective-C programs stored in Mach-O files are more difficult to evaluate. Apple instituted some controls such as executable encryption that can be bypassed, but the compiled nature of applications proves to be of significant benefit in thwarting application static analysis.



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

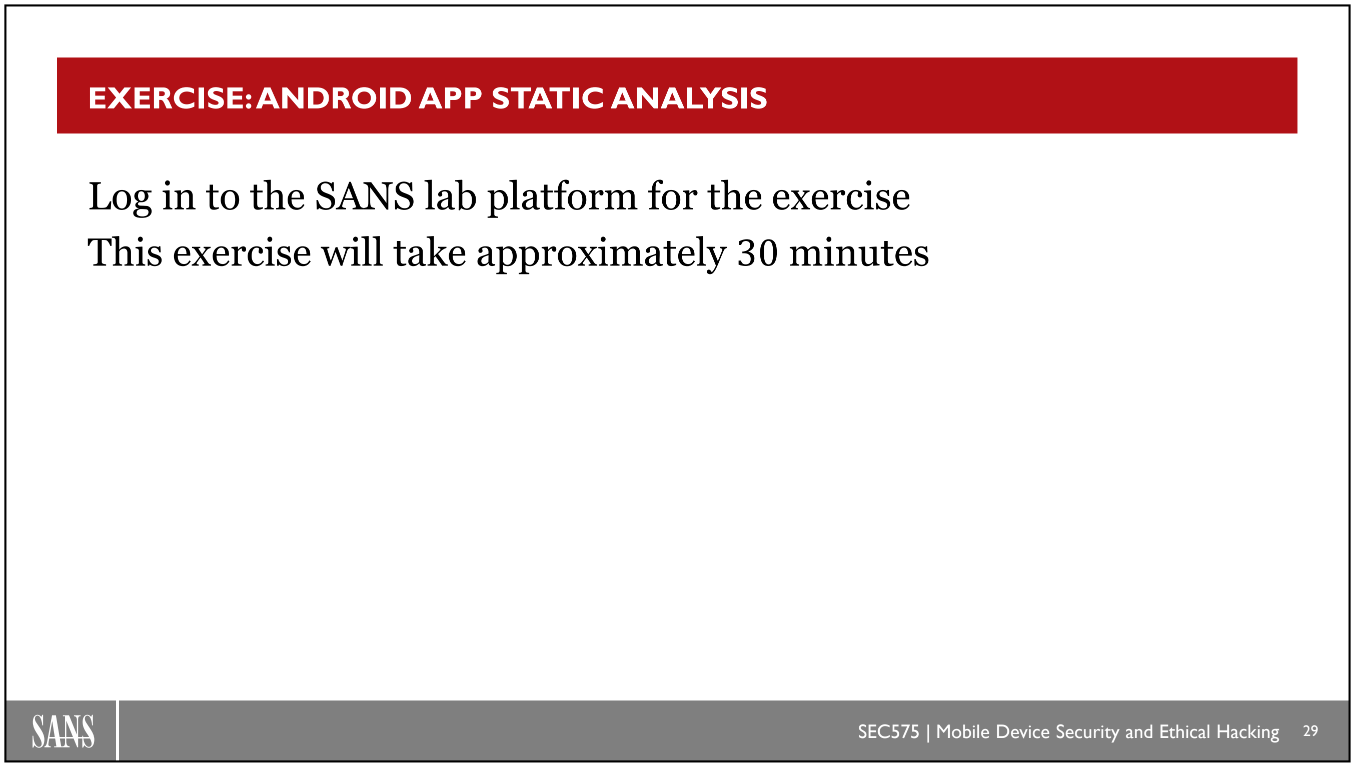
DAY 3

Static Application Analysis
Manual Static Analysis
Exercise: Android App Static Analysis
Automating App Analysis
Exercise: MobSF
Obfuscated Apps
Exercise: Obfuscated Android App Analysis
Third-Party App Platforms
Exercise: Xamarin App Analysis
Exercise: PhoneGap App Analysis

SANS

SEC575 | Mobile Device Security and Ethical Hacking 28

This page intentionally left blank.



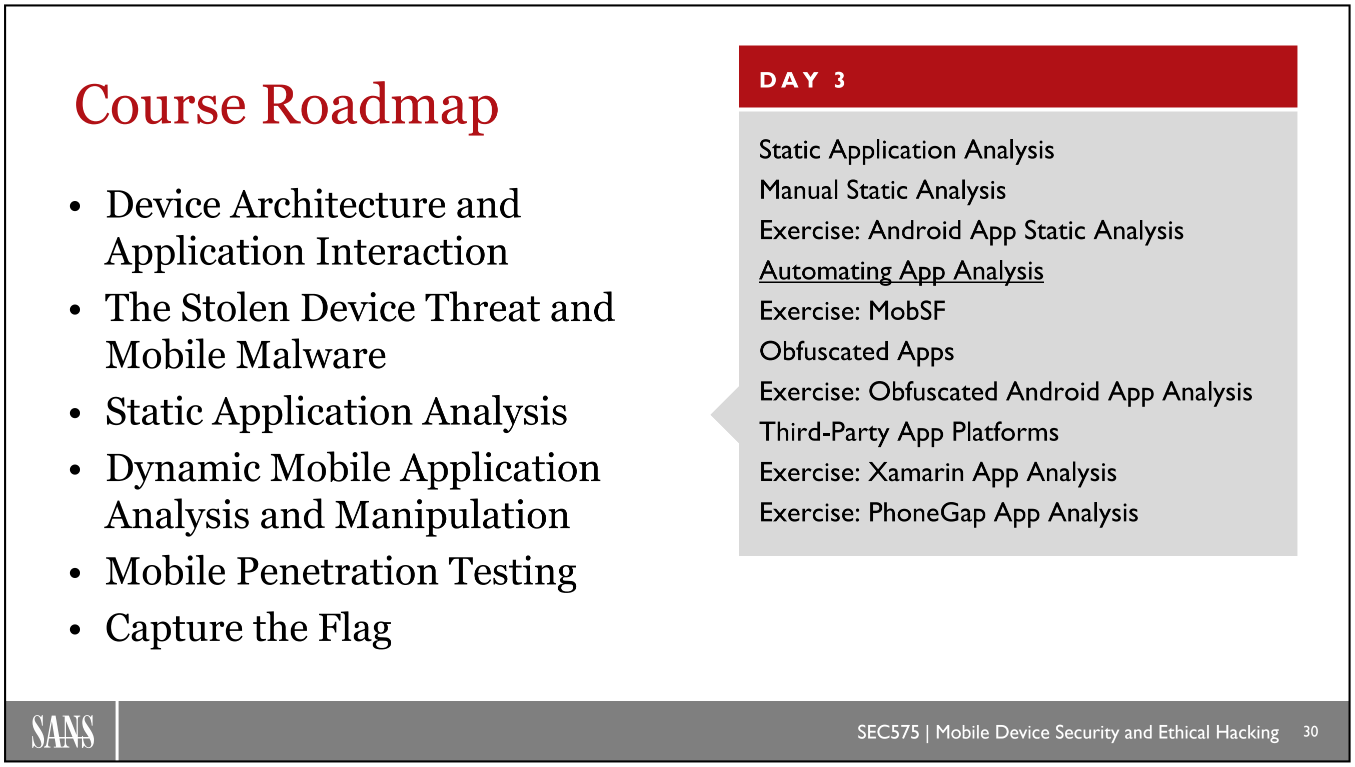
EXERCISE: ANDROID APP STATIC ANALYSIS

Log in to the SANS lab platform for the exercise

This exercise will take approximately 30 minutes

Exercise: Android App Static Analysis

Please log in to the SANS lab platform for the Android App Static Analysis exercise. This exercise will take approximately 30 minutes to complete.

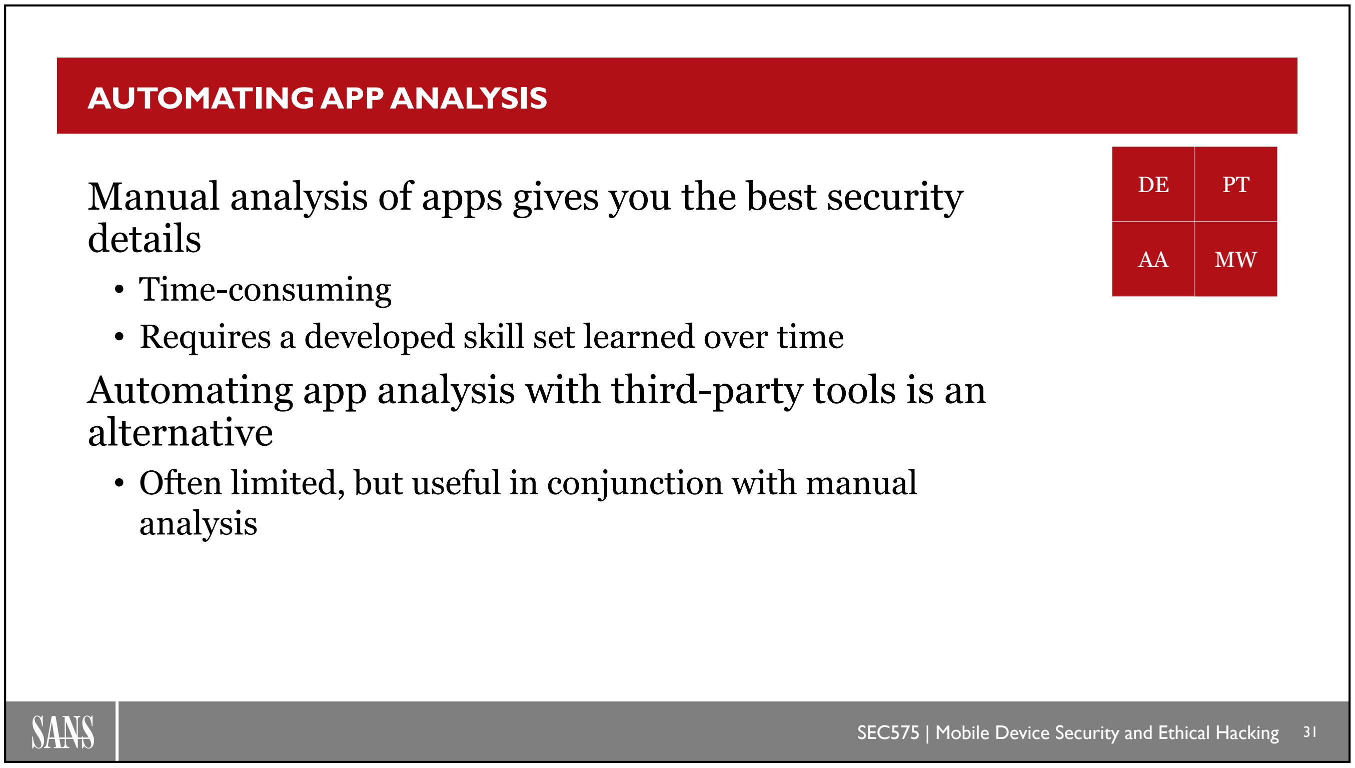


Course Roadmap

- 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 3

This page intentionally left blank.



AUTOMATING APP ANALYSIS

DE	PT
AA	MW

Manual analysis of apps gives you the best security details

- Time-consuming
- Requires a developed skill set learned over time

Automating app analysis with third-party tools is an alternative

- Often limited, but useful in conjunction with manual analysis

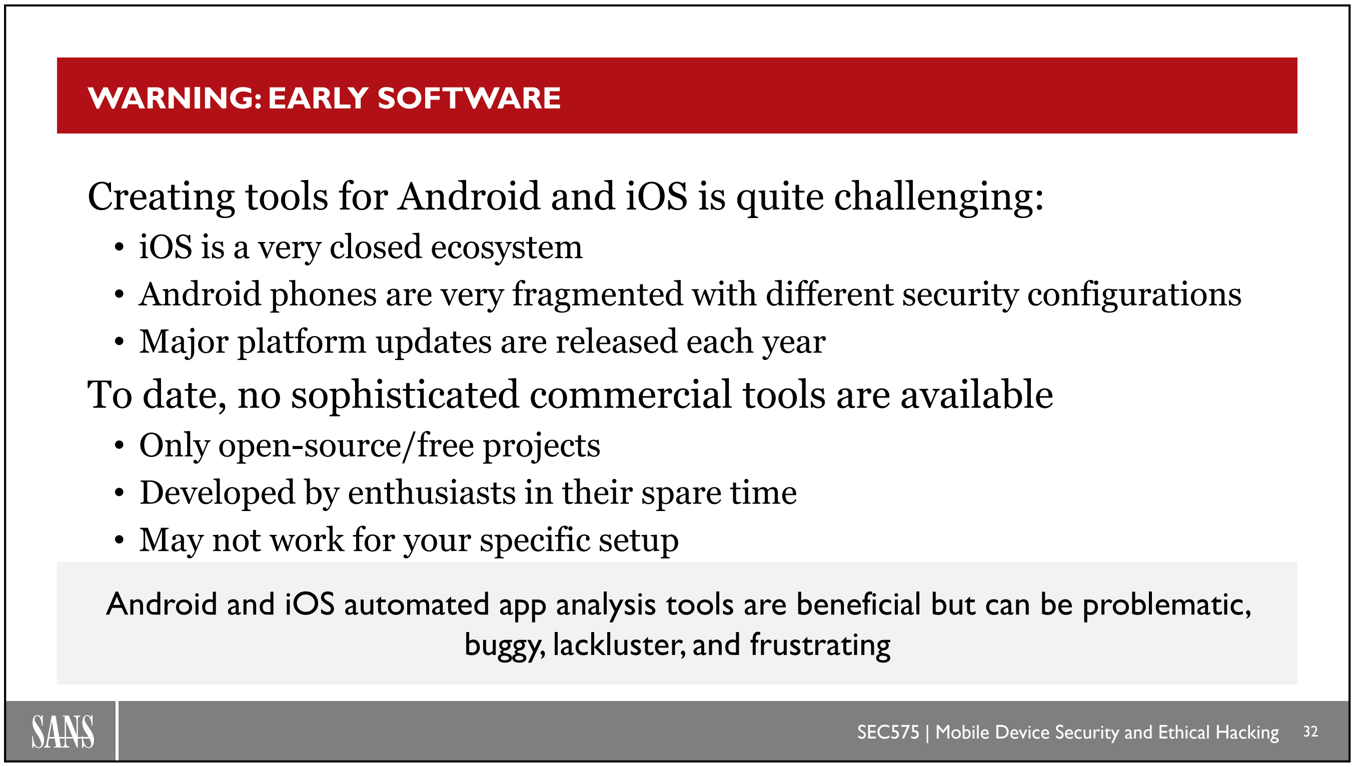
SEC575 | Mobile Device Security and Ethical Hacking 31

Automating App Analysis

So far, we investigated several manual analysis techniques to identify the behavior of a target application. From this analysis, you can gather useful information about the target application, including whether the app is reasonably secure for use in your organization.

Unfortunately, manual analysis of applications is a time-consuming task. Few analysts have the skills to perform app analysis on a regular basis while balancing other job responsibilities.

Fortunately, automated mobile device application analysis tools have recently become available. Using these tools, we can gather many of the app analysis details that are pertinent to understanding the security of the application. Automated app analysis tools may not provide all the answers needed about the security of an application, but they give us a quick way to assess an app that can then be further leveraged with manual analysis if required.



WARNING: EARLY SOFTWARE

Creating tools for Android and iOS is quite challenging:

- iOS is a very closed ecosystem
- Android phones are very fragmented with different security configurations
- Major platform updates are released each year

To date, no sophisticated commercial tools are available

- Only open-source/free projects
- Developed by enthusiasts in their spare time
- May not work for your specific setup

Android and iOS automated app analysis tools are beneficial but can be problematic, buggy, lackluster, and frustrating

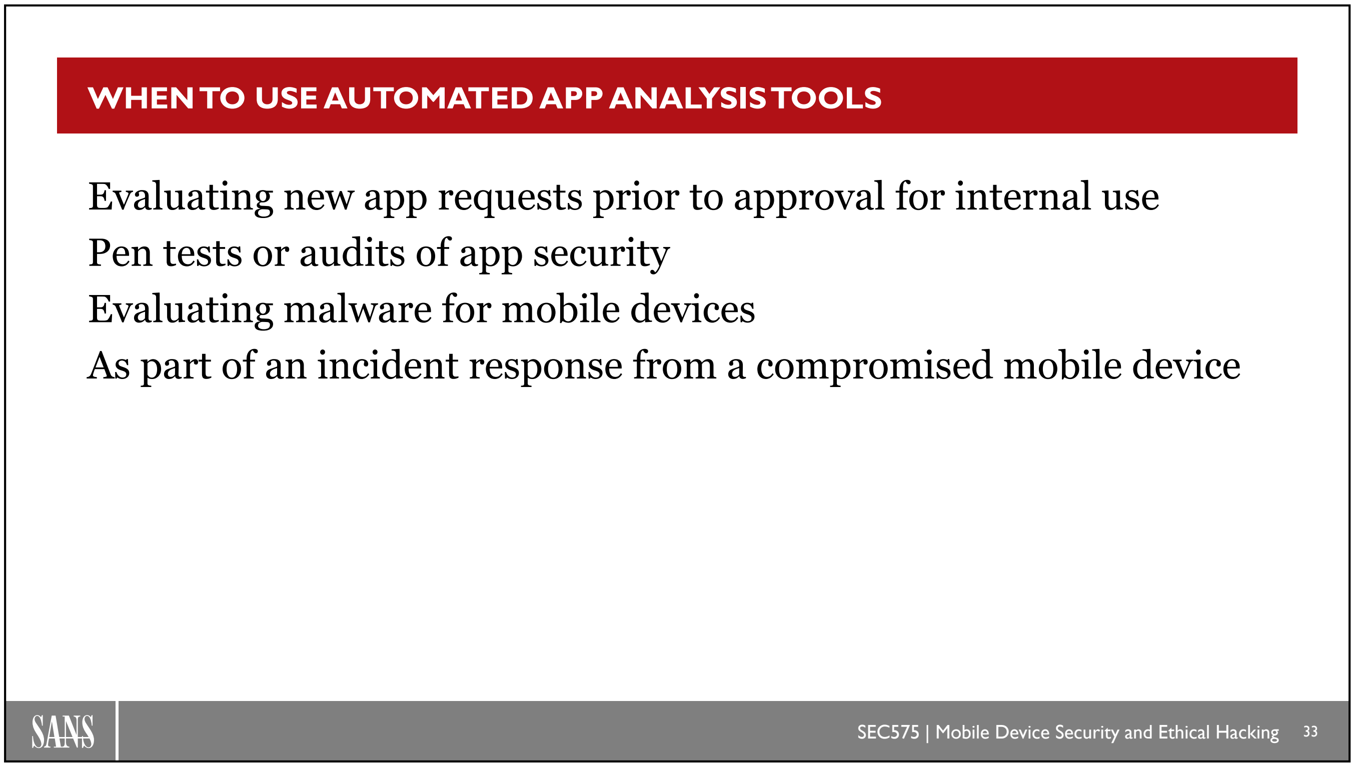
SANS SEC575 | Mobile Device Security and Ethical Hacking 32

Warning: Early Software

We examine mobile device application analysis tools in this module that are made freely available by their authors. Due to the closed nature of iOS and the high diversity of Android smartphones, it is important to understand that tools may not always work for your specific setup, and that some debugging may be required. Additionally, as mobile platforms usually release a new major version each year, tools often struggle to keep up with changes, which means it's difficult to test on the latest OS versions.

To date, no sophisticated commercial analysis tools have been made available, leaving only the open-source tools. As such, we need to set our expectations accordingly and expect command line or awkward text-based user interfaces, frequent crashes, and little to no project documentation.

We recognize that the open-source tools that are available are the gracious products of hard work by developers who often have many other responsibilities to keep them busy. These tools are a welcome addition to the open-source community, and we thank these developers for their contributions.



WHEN TO USE AUTOMATED APP ANALYSIS TOOLS

Evaluating new app requests prior to approval for internal use

Pen tests or audits of app security

Evaluating malware for mobile devices

As part of an incident response from a compromised mobile device

SANS SEC575 | Mobile Device Security and Ethical Hacking 33

When to Use Automated App Analysis Tools

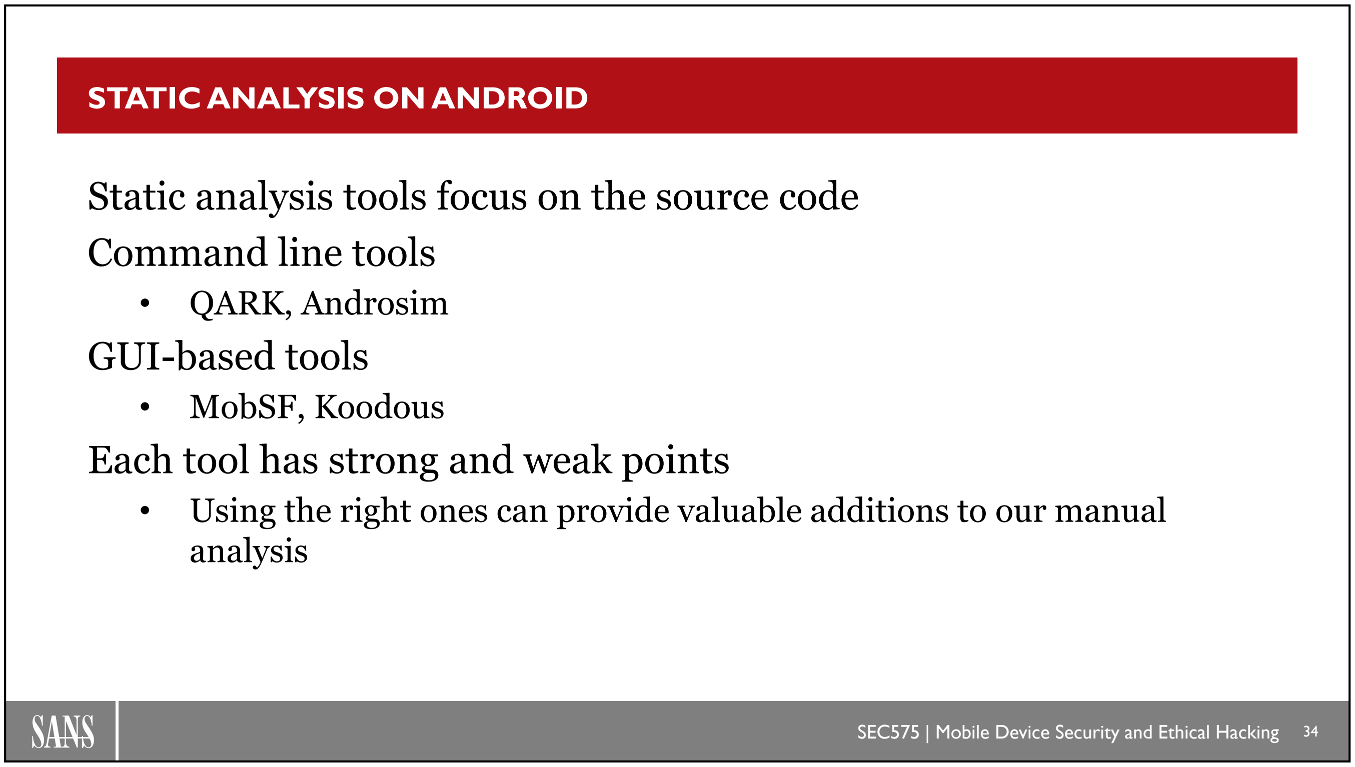
Before we jump into the tools themselves, let's look at when it's a good idea to use these tools.

If your organization is evaluating new application requests prior to inclusion in an enterprise app store, or otherwise approves them for use on personally owned or enterprise-owned mobile devices, automated analysis tools help simplify the approval process. By identifying security threats or exposure areas within an application, we can make a better-informed decision as to whether an app meets the requirements for approved applications.

As a penetration tester, we may identify the use of a mobile app in an organization and evaluate the app on a testing device. This analysis data may give the penetration tester ideas for exploitation opportunities due to the identification of flaws in the software.

If you are evaluating mobile device malware, either as a malware researcher or responding to a compromised device, automated app analysis tools can reveal the target hosts the malware communicates with and the data that is transmitted to the remote hosts. With this information, we can evaluate the impact and scope of the malware and possibly identify removal opportunities.

Finally, incident response analysis can also leverage automated app analysis tools following the identification of a compromised mobile device. By evaluating multiple applications installed on the compromised device, the incident response analyst may identify potential attack vectors used by an adversary to exploit the device, giving the analyst a lead to begin their analysis.



STATIC ANALYSIS ON ANDROID

Static analysis tools focus on the source code

Command line tools

- QARK, Androsim

GUI-based tools

- MobSF, Koodous

Each tool has strong and weak points

- Using the right ones can provide valuable additions to our manual analysis

SANS SEC575 | Mobile Device Security and Ethical Hacking 34

Static analysis tools work by analyzing the Java source code and other important Android app resources, such as the Android Manifest file, in order to detect potential security issues and vulnerabilities. Next, we will look at some useful automatic static analysis tools for Android applications. Some of them are command line based:

- **QARK:** detects potential vulnerabilities in APK files and source code
- **Androsim:** compares apps for changes to detect malicious clones

Others provide a full GUI:

- **MobSF:** provides a comprehensive overview of apps and highlights potential issues
- **Koodous:** online platform enabling collaborative app analysis

We'll see that each of them has its unique advantages and weaknesses —if we understand them and combine them properly, they can provide valuable input for our manual analysis.

MOBSF

Mobile Security Framework (MobSF) aims to be an all-in-one solution for static and dynamic analysis

Gives a good overview of the application

- (Exported) Activities, Services, Receivers, and Providers

Highlights potential issues

- Debug/backup enabled, unprotected receivers, sensitive information in logs...

Dynamic analysis is less stable

https://github.com/MobSF/Mobile-Security-Framework-MobSF

SANS | SEC575 | Mobile Device Security and Ethical Hacking 35

MobSF

The Mobile Security Framework (MobSF) tries to be an all-in-one solution for static and dynamic analysis for Android, iOS, and Windows apps. It provides you with much information on the application and can highlight many different potential security problems.

MobSF supports both dynamic and static analysis, but the dynamic analysis can be tricky to get up and running. Once dynamic analysis is working, MobSF will track various Android APIs to monitor platform interaction, such as cryptographic calculations, file operations, and network calls.

MOBSF INSTALLATION

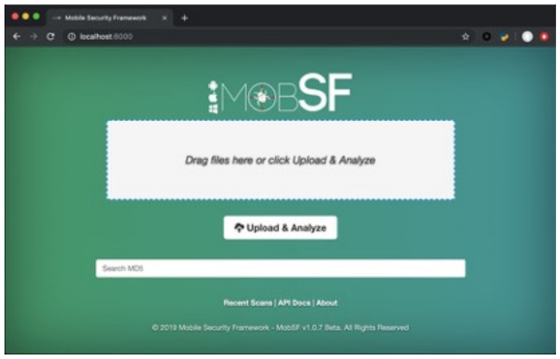
Fully dockerized:

- `docker pull opensecurity/mobile-security-framework-mobsf`
- `docker run -it -p 8000:8000 opensecurity/mobile-security-framework-mobsf:latest`

Access through http://localhost:8000

Upload IPA or APK

Supports REST API for integration into automated pipelines


Mobile Security Framework
localhost:8000
MOBSF
Drag files here or click Upload & Analyze
Upload & Analyze
Search MD5
Recent Scans | API Docs | About
© 2019 Mobile Security Framework - MobSF v1.0.7 Beta. All Rights Reserved

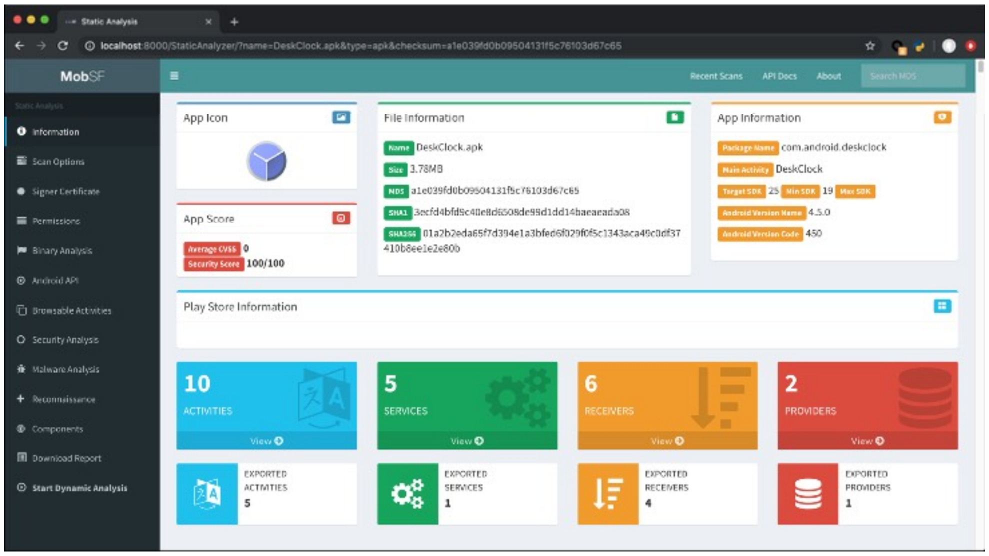
MobSF Installation

MobSF is most easily installed through docker. This has the advantage that your system will not be impacted by either MobSF or any of the applications that are analyzed.

After pulling the image from docker hub and launching an instance, the application can be accessed through port 8000 on the host machine. On the first screen, you can choose an IPA or APK to upload, after which the different built-in static analyzers start executing.

MobSF also provides a REST API that can be used to programmatically upload files, request scans, and export the results. This can be used in an automated build process where each new build of an application is checked for vulnerabilities.

MOBSF REPORTS



localhost:8000/StaticAnalyzer/?name=DeskClock.apk&type=apk&checksum=a1e039fd0b09504131f5c76103d67c65

File Information

Name	DeskClock.apk
Size	3.78MB
MD5	a1e039fd0b09504131f5c76103d67c65
SHA1	3ecfd4bfd9c40e8d6508de99d1dd14baeaeada08
SHA256	01a2b2eda65f7d394e1a3bfed6f029f0f5c1343aca49c0df37410b8ee1e2e80b

App Score

Average CVSS	0
Security Score	100/100

App Information

Package Name	com.android.deskclock
Main Activity	DeskClock
Target SDK	25
Min SDK	19
Max SDK	
Android Version Name	4.5.0
Android Version Code	450

10 ACTIVITIES · 5 SERVICES · 6 RECEIVERS · 2 PROVIDERS

EXPORTED ACTIVITIES 5 · EXPORTED SERVICES 1 · EXPORTED RECEIVERS 4 · EXPORTED PROVIDERS 1

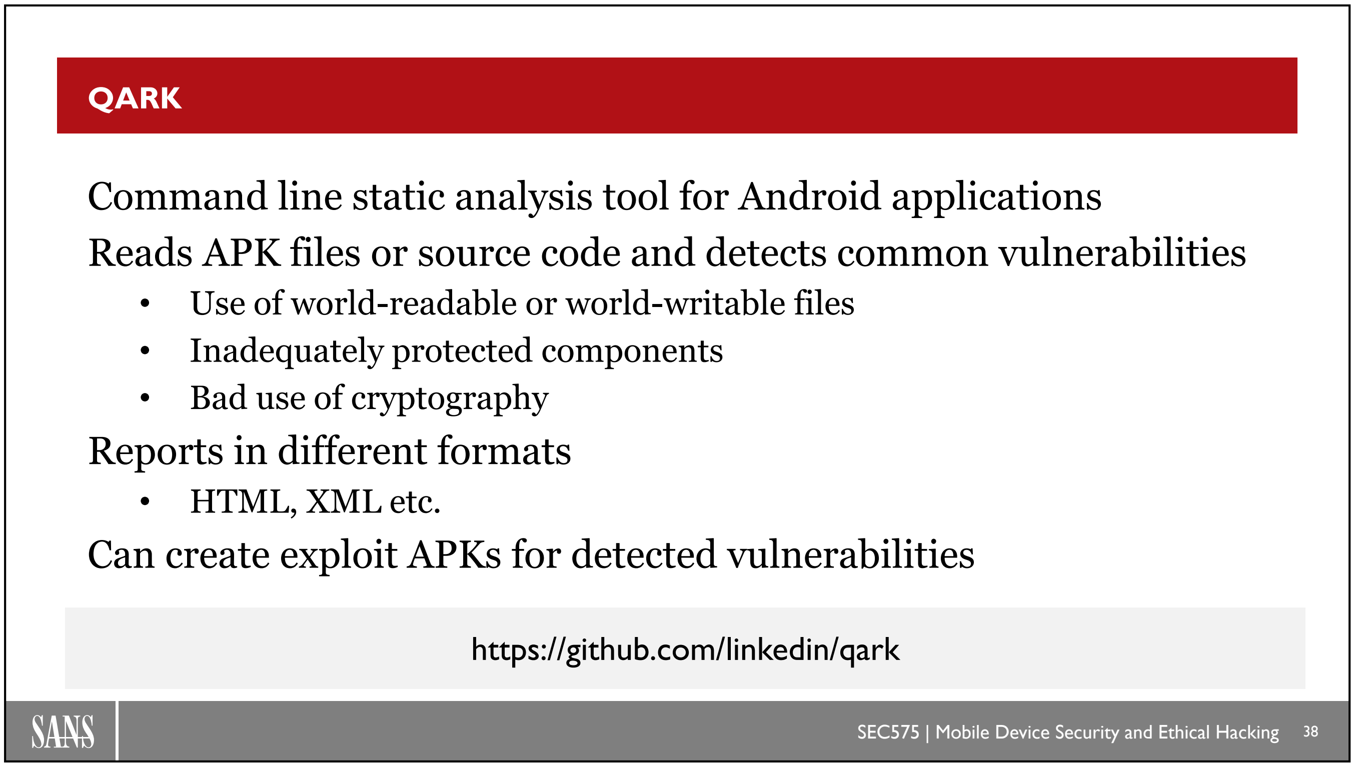
Typically many false positives, but still a great way to start an application analysis

MobSF Reports

The end result of a static analysis is an HTML report that gives an extensive overview of the application with the ability to start a dynamic analysis session. Among other things, MobSF will detect the following potential vulnerabilities:

- Weak signing certificate
- Dangerous permissions
- Unprotected services or providers
- Missing application hardening (backup, debug…)

Static analysis is typically much less powerful than dynamic analysis, and applications will trigger many different false positives. It is, however, still an excellent place to start application analysis, and a lot of the low-hanging fruit can easily be identified.



QARK

Command line static analysis tool for Android applications

Reads APK files or source code and detects common vulnerabilities

- Use of world-readable or world-writable files
- Inadequately protected components
- Bad use of cryptography

Reports in different formats

- HTML, XML etc.

Can create exploit APKs for detected vulnerabilities

https://github.com/linkedin/qark

SANS SEC575 | Mobile Device Security and Ethical Hacking 38

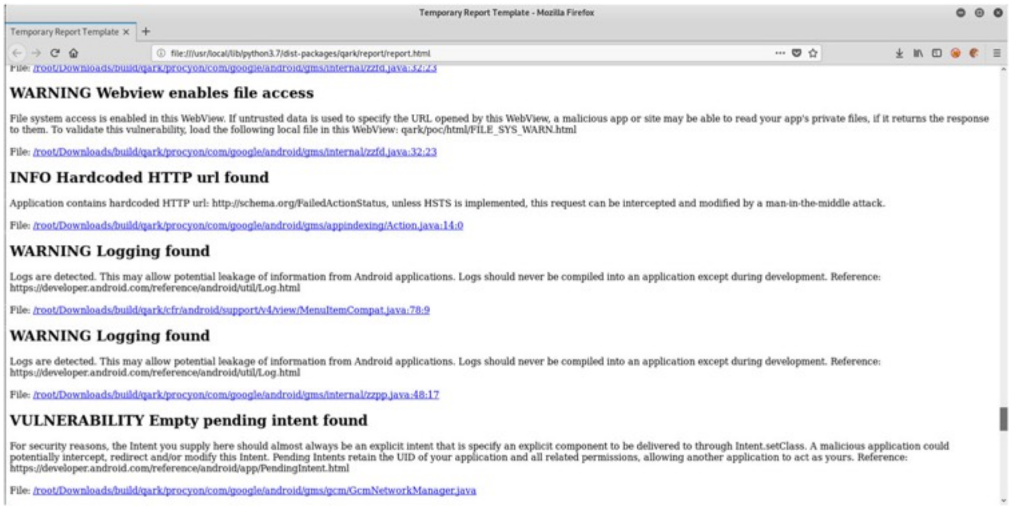
QARK

QARK is another command line static analysis tool for Android applications, written in Python. It accepts both APK files and Java source code. When given an APK file, QARK automatically decompiles it and runs the analysis on the decompiled source code. Some examples of vulnerabilities that can be detected by QARK include:

- **Applications using world-readable or world-writable files:** data in these files could be accessed or modified by other applications
- **Accidentally exported or inadequately protected components:** for example, services exported without restrictions
- **Weak or improper use of cryptography**: for example, cryptographic keys embedded in the app source code

QARK can generate reports in multiple formats, including HTML, XML, and JSON. An interesting additional functionality is that it can create exploits, meaning APKs that can exploit the vulnerabilities found in the target application. If you want to use this functionality, some additional setup is required, since building the exploit APKs depends on the Android SKD and build tools. Detailed instructions can be found here: https://github.com/linkedin/qark.

QARK ANALYSIS RESULTS



```
Temporary Report Template - Mozilla Firefox
file:///usr/local/lib/python3.7/dist-packages/qark/report/report.html

File: /root/Downloads/build/qark/procyon/com/google/android/gms/internal/zzfd.java:32:23

WARNING Webview enables file access

File system access is enabled in this WebView. If untrusted data is used to specify the URL opened by this WebView, a malicious app or site may be able to read your app's private files, if it returns the response
to them. To validate this vulnerability, load the following local file in this WebView: qark/poc/html/FILE_SYS_WARN.html

File: /root/Downloads/build/qark/procyon/com/google/android/gms/internal/zzfd.java:32:23

INFO Hardcoded HTTP url found

Application contains hardcoded HTTP url: http://schema.org/FailedActionStatus, unless HSTS is implemented, this request can be intercepted and modified by a man-in-the-middle attack.

File: /root/Downloads/build/qark/procyon/com/google/android/gms/appindexing/Action.java:14:0

WARNING Logging found

Logs are detected. This may allow potential leakage of information from Android applications. Logs should never be compiled into an application except during development. Reference:
https://developer.android.com/reference/android/util/Log.html

File: /root/Downloads/build/qark/cfr/android/support/v4/view/MenuItemCompat.java:78:9

WARNING Logging found

Logs are detected. This may allow potential leakage of information from Android applications. Logs should never be compiled into an application except during development. Reference:
https://developer.android.com/reference/android/util/Log.html

File: /root/Downloads/build/qark/procyon/com/google/android/gms/internal/zzpp.java:48:17

VULNERABILITY Empty pending intent found

For security reasons, the Intent you supply here should almost always be an explicit intent that is specify an explicit component to be delivered to through Intent.setClass. A malicious application could
potentially intercept, redirect and/or modify this Intent. Pending Intents retain the UID of your application and all related permissions, allowing another application to act as yours. Reference:
https://developer.android.com/reference/android/app/PendingIntent.html

File: /root/Downloads/build/qark/procyon/com/google/android/gms/gcm/GcmNetworkManager.java
```

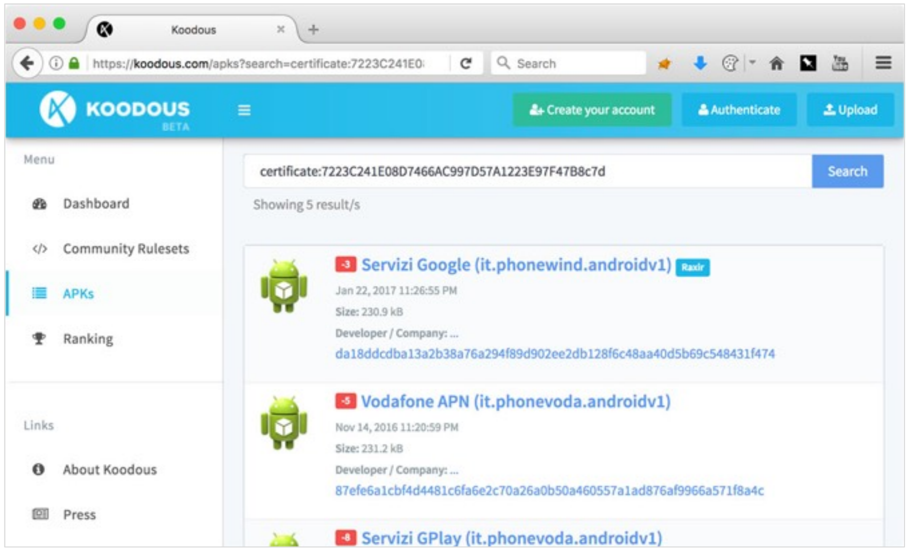
Finding explanations and links to decompiled source code

QARK Analysis Results

Reports generated by QARK are somewhat crudely formatted but contain all the essential information. For each potential finding, QARK provides a short description, as well as a link to the associated source code.

In the example provided, it detects, among others, some potentially insecure settings in a WebView used by the app, which we need to investigate further to determine if they translate to an actual vulnerability. QARK cannot replace manual analysis, but it can be a helpful starting point for discovering app vulnerabilities.

KOODOUS



certificate:7223C241E08D7466AC997D57A1223E97F47B8c7d

Showing 5 result/s

-3 Servizi Google (it.phonewind.androidv1) Raxir
Jan 22, 2017 11:26:55 PM
Size: 230.9 kB
Developer / Company: ...
da18ddcdba13a2b38a76a294f89d902ee2db128f6c48aa40d5b69c548431f474

-5 Vodafone APN (it.phonevoda.androidv1)
Nov 14, 2016 11:20:59 PM
Size: 231.2 kB
Developer / Company: ...
87efe6a1cbf4d4481c6fa6e2c70a26a0b50a460557a1ad876af9966a571f8a4c

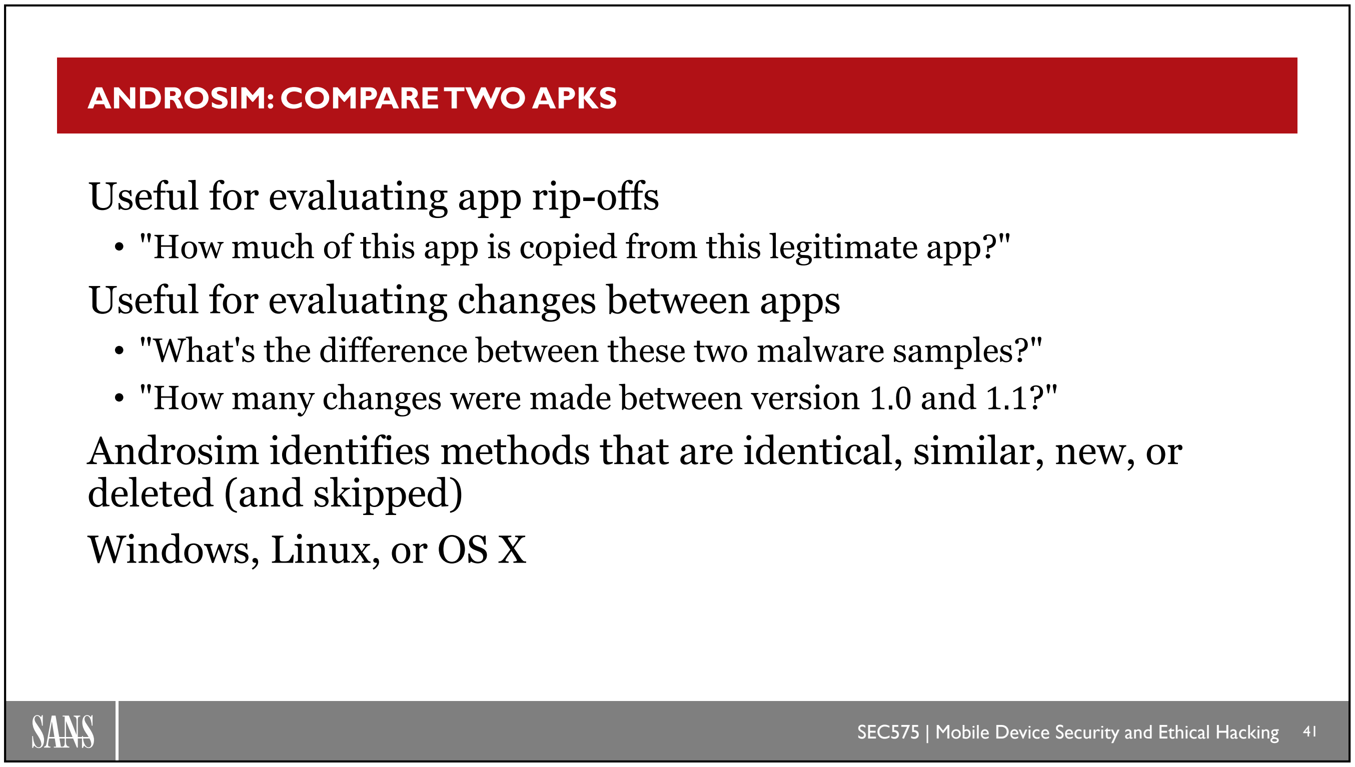
-8 Servizi GPlay (it.phonevoda.androidv1)

Online collaborative research site for Android applications. Search by SHA1 hash of signing certificate to identify all known apps published with the same signing key.

Koodous

The Koodous service at koodous.com is designed as a "collaborative platform that combines the power of online analysis tools with social interactions between the analysts over a vast APKs repository." Even without creating an account, users can search for details surrounding submitted APKs, including the SHA1 hash of a certificate. Using the SHA1 hash of an RSA file from an application, you can use Koodous to identify any other APKs known to the service that were signed with the same certificate.

In the example on this page, we submit a certificate hash from a malware sample reportedly produced by Hacking Team, an Italian malware company. This signature was found in five other applications as well. Koodous provides some information about the other applications, including permission declarations and certificate details, but does not provide direct links to the APK files.



ANDROSIM: COMPARE TWO APKS

Useful for evaluating app rip-offs

- "How much of this app is copied from this legitimate app?"

Useful for evaluating changes between apps

- "What's the difference between these two malware samples?"
- "How many changes were made between version 1.0 and 1.1?"

Androsim identifies methods that are identical, similar, new, or deleted (and skipped)

Windows, Linux, or OS X

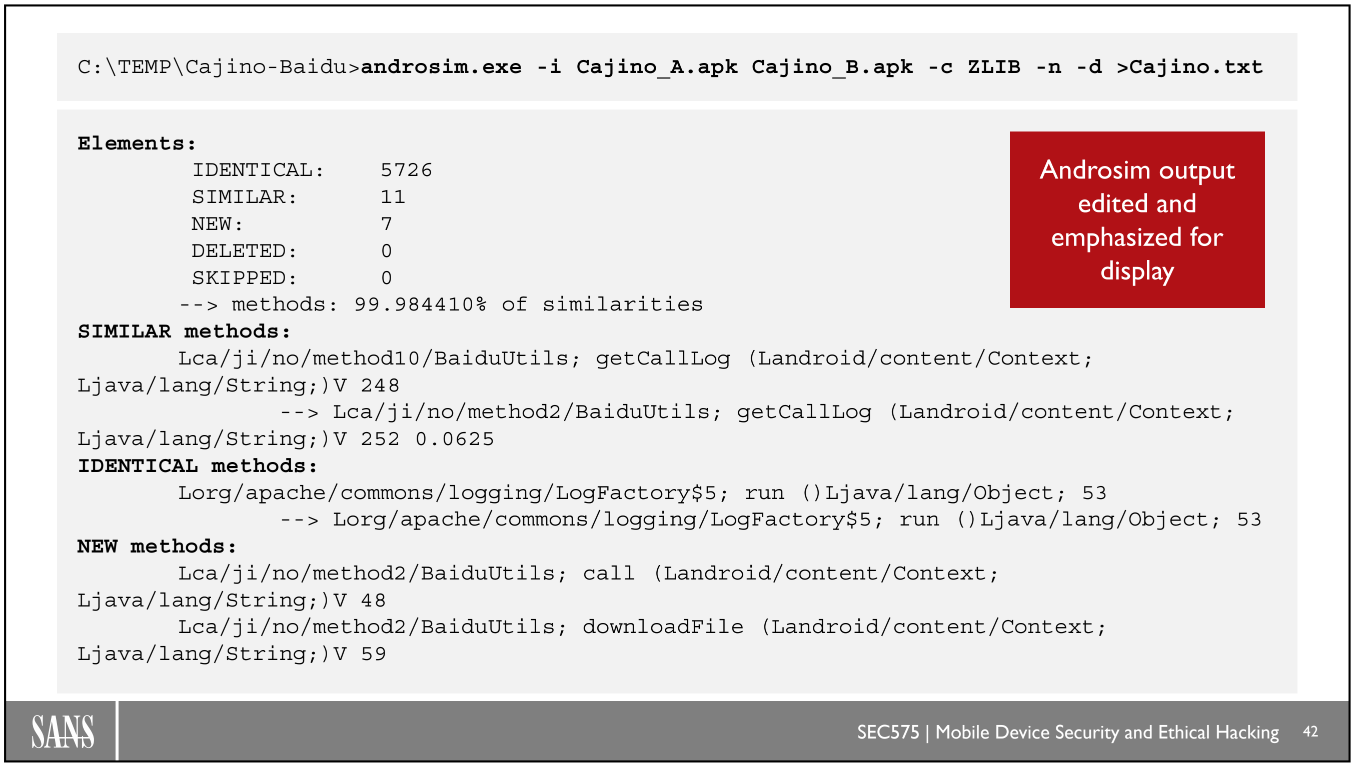
SEC575 | Mobile Device Security and Ethical Hacking 41

Androsim: Compare Two APKs

The Androsim utility allows us to compare two Android packages to identify the number of identical, similar, new, or deleted (and skipped) methods. This tool offers several practical analysis opportunities:

- To identify app rip-offs, where a different author posts a similar app to the Google Play store that we believe to be derived from a legitimate app
- To identify changes between two apps, such as comparing two malware samples from the same malware family or changes between two versions of an application

Androsim is a command line tool for Windows, Linux, or OS X systems, bundled with the Androguard tools. The Androguard (and Androsim) source code is available at https://github.com/androguard/androguard/ with official releases available at https://github.com/androguard/androguard/releases. Installation instructions are available at https://code.google.com/p/androguard/wiki/Installation.



```
C:\TEMP\Cajino-Baidu>androsim.exe -i Cajino_A.apk Cajino_B.apk -c ZLIB -n -d >Cajino.txt
```

```
Elements:
         IDENTICAL:    5726
         SIMILAR:      11
         NEW:          7
         DELETED:      0
         SKIPPED:      0
        --> methods: 99.984410% of similarities
SIMILAR methods:
        Lca/ji/no/method10/BaiduUtils; getCallLog (Landroid/content/Context;
Ljava/lang/String;)V 248
                --> Lca/ji/no/method2/BaiduUtils; getCallLog (Landroid/content/Context;
Ljava/lang/String;)V 252 0.0625
IDENTICAL methods:
        Lorg/apache/commons/logging/LogFactory$5; run ()Ljava/lang/Object; 53
                --> Lorg/apache/commons/logging/LogFactory$5; run ()Ljava/lang/Object; 53
NEW methods:
        Lca/ji/no/method2/BaiduUtils; call (Landroid/content/Context;
Ljava/lang/String;)V 48
        Lca/ji/no/method2/BaiduUtils; downloadFile (Landroid/content/Context;
Ljava/lang/String;)V 59
```

Androsim Example

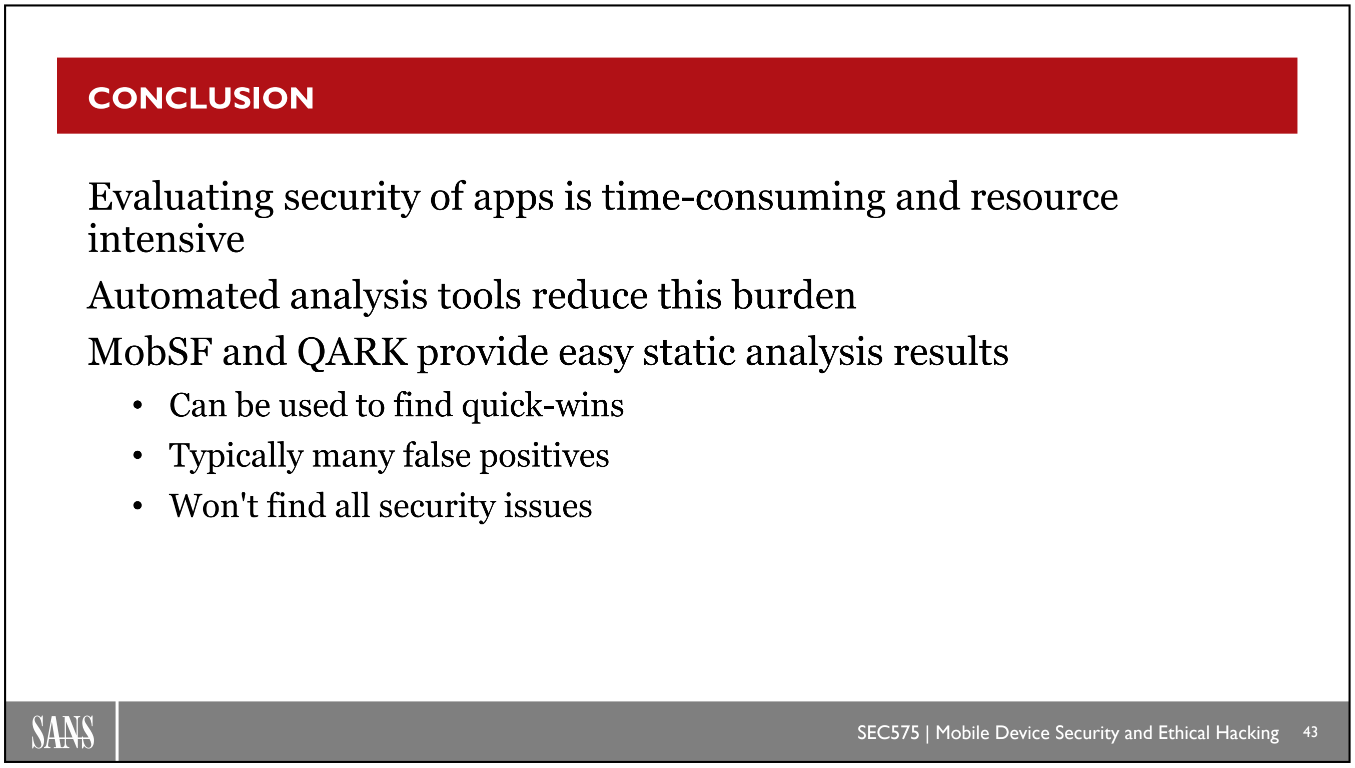
In the example on this slide, the author compared two samples of the Cajino malware for Android. Cajino is a Remote Access Trojan (RAT) capable of recording audio from the device microphone, sending SMS, making phone calls, deleting and downloading files, retrieving location information, and collecting personal information (including SMS, contacts, and camera pictures). Cajino uses the Baidu Cloud Storage (BCS) APIs to harvest the data and controls the infected Android devices using the Baidu Cloud Push notification service (similar to Google Cloud Messaging). (Additional analysis on the Cajino malware is available at http://b0n1.blogspot.com/2015/03/remote-administration-trojan-using.html.)

Using the Androsim utility for Windows, the two Cajino Android packages are compared, decompressed using the standard ZLIB decompression routing, and compared (using only the first APK as the basis for establishing the similarities metric with "-n"). Detailed information about the changed methods is displayed with the "-d" argument.

The Cajino output is long and should be redirected to a file for subsequent inspection. Here Androsim indicates that there are 5,726 identical methods between the two samples, which includes both the Cajino malware code and the packaged libraries. Only 11 methods are different (similar) between the two samples, and 7 samples are new in Cajino_B.apk.

The bold output on this page was added by the author for emphasis. We've removed most of the similar method output for space; here the ca.ji.no.method10.BaiduUtils.getCallLog() method is shown to be similar to the sample B ca.ji.no.method2.BaiduUtils.getCallLog() method. In sample A, the getCallLog() method is 248 lines of code, compared to the sample B method, which is 252 lines. The similarity ratio between the two is .0625, or 6.25%.

From this output, we can focus our analysis on the getCallLog() method as being slightly different than the previous version, and we can also investigate the new functions call() and downloadFile() shown in the NEW methods section. For malware analysis, the IDENTICAL methods listing isn't particularly useful, but in a scenario where you are trying to identify a copycat application, the metrics shown for identical methods would be useful to identify stolen or pirated code.



CONCLUSION

Evaluating security of apps is time-consuming and resource intensive

Automated analysis tools reduce this burden

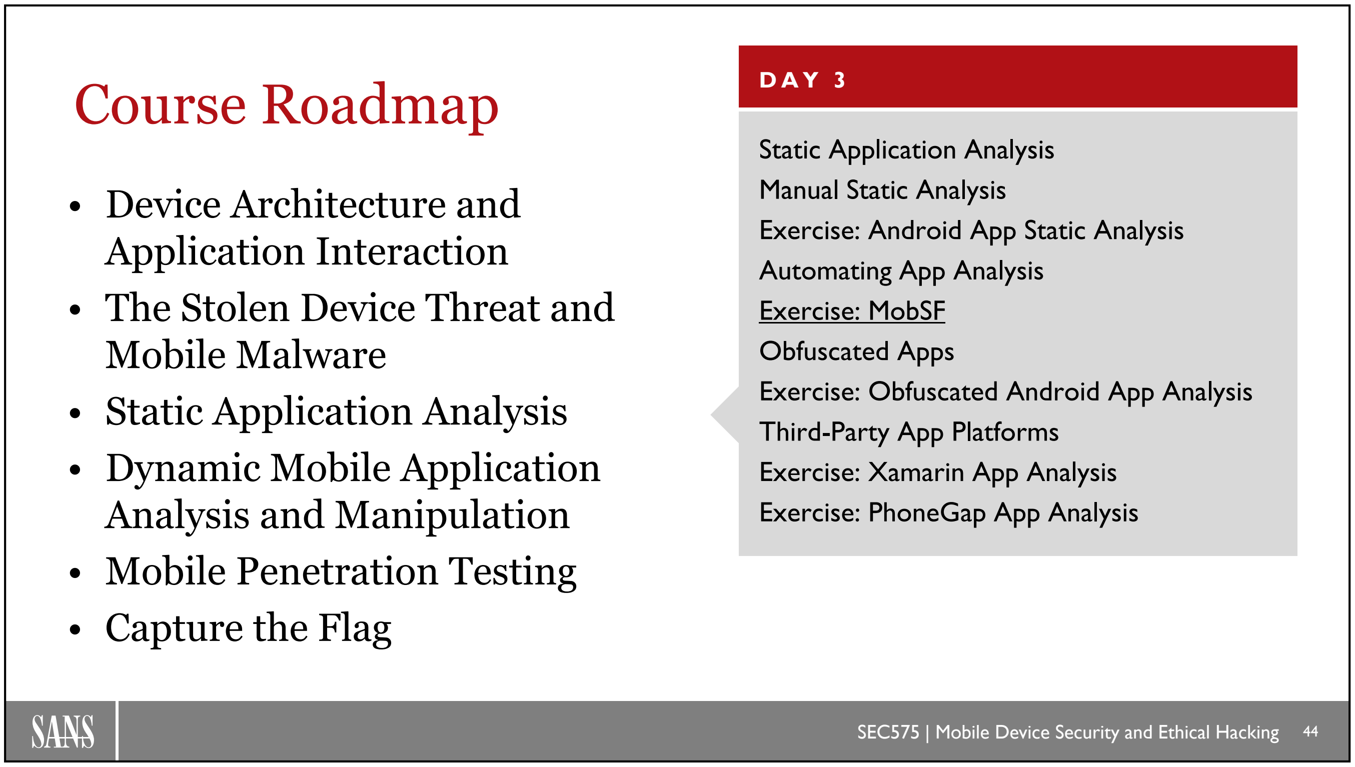
MobSF and QARK provide easy static analysis results

- Can be used to find quick-wins
- Typically many false positives
- Won't find all security issues

SANS | SEC575 | Mobile Device Security and Ethical Hacking 43

Conclusion

Evaluating mobile device applications is an important component for a strong mobile device security strategy, requiring significant skill and expertise for effective analysis. Automated analysis tools reduce the burden associated with the analysis. Even if the results from automated analysis tools are not comprehensive, their data can be useful as part of an initial analysis of an application. At a minimum, automated analysis tools reduce the total amount of time investment necessary for application analysis.



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 3

Static Application Analysis
Manual Static Analysis
Exercise: Android App Static Analysis
Automating App Analysis
<u>Exercise: MobSF</u>
Obfuscated Apps
Exercise: Obfuscated Android App Analysis
Third-Party App Platforms
Exercise: Xamarin App Analysis
Exercise: PhoneGap App Analysis

SANS | SEC575 | Mobile Device Security and Ethical Hacking 44

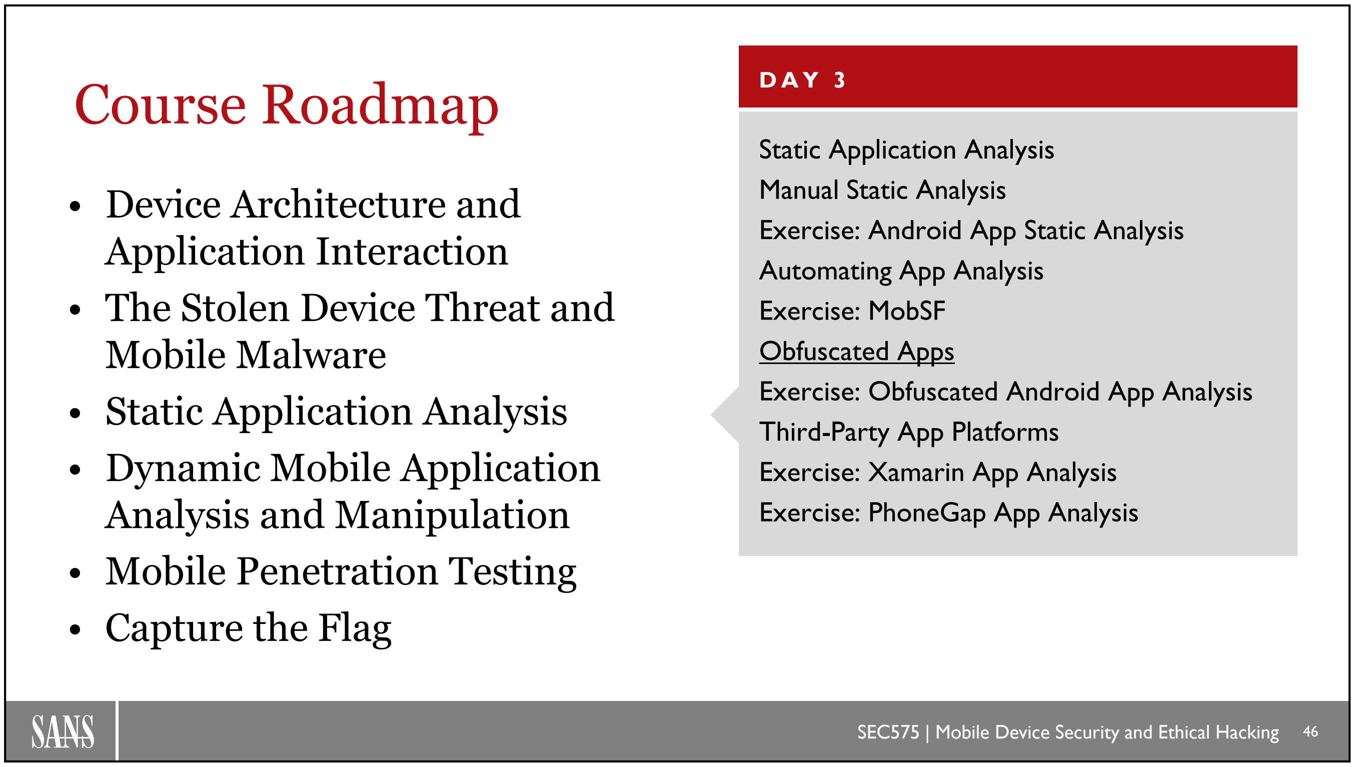
This page intentionally left blank.

EXERCISE: MOBSF

Log in to the SANS lab platform for the exercise
This exercise will take approximately 30 minutes

Exercise: MobSF

Please log in to the SANS lab platform for the MobSF exercise. This exercise will take approximately 30 minutes to complete.



Course Roadmap

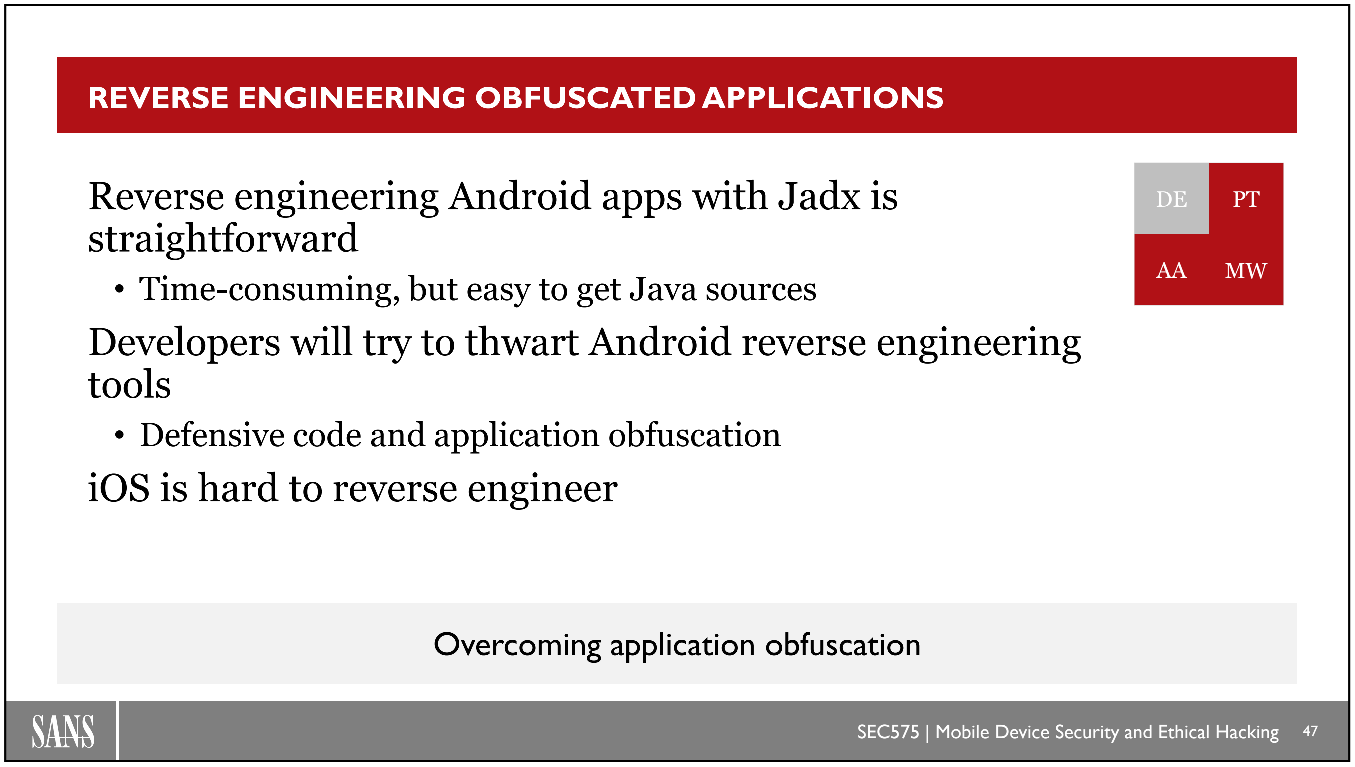
- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 3

Static Application Analysis
Manual Static Analysis
Exercise: Android App Static Analysis
Automating App Analysis
Exercise: MobSF
Obfuscated Apps
Exercise: Obfuscated Android App Analysis
Third-Party App Platforms
Exercise: Xamarin App Analysis
Exercise: PhoneGap App Analysis

SANS SEC575 | Mobile Device Security and Ethical Hacking 46

This page intentionally left blank.



REVERSE ENGINEERING OBFUSCATED APPLICATIONS

DE	PT
AA	MW

Reverse engineering Android apps with Jadx is straightforward

- Time-consuming, but easy to get Java sources

Developers will try to thwart Android reverse engineering tools

- Defensive code and application obfuscation

iOS is hard to reverse engineer

Overcoming application obfuscation

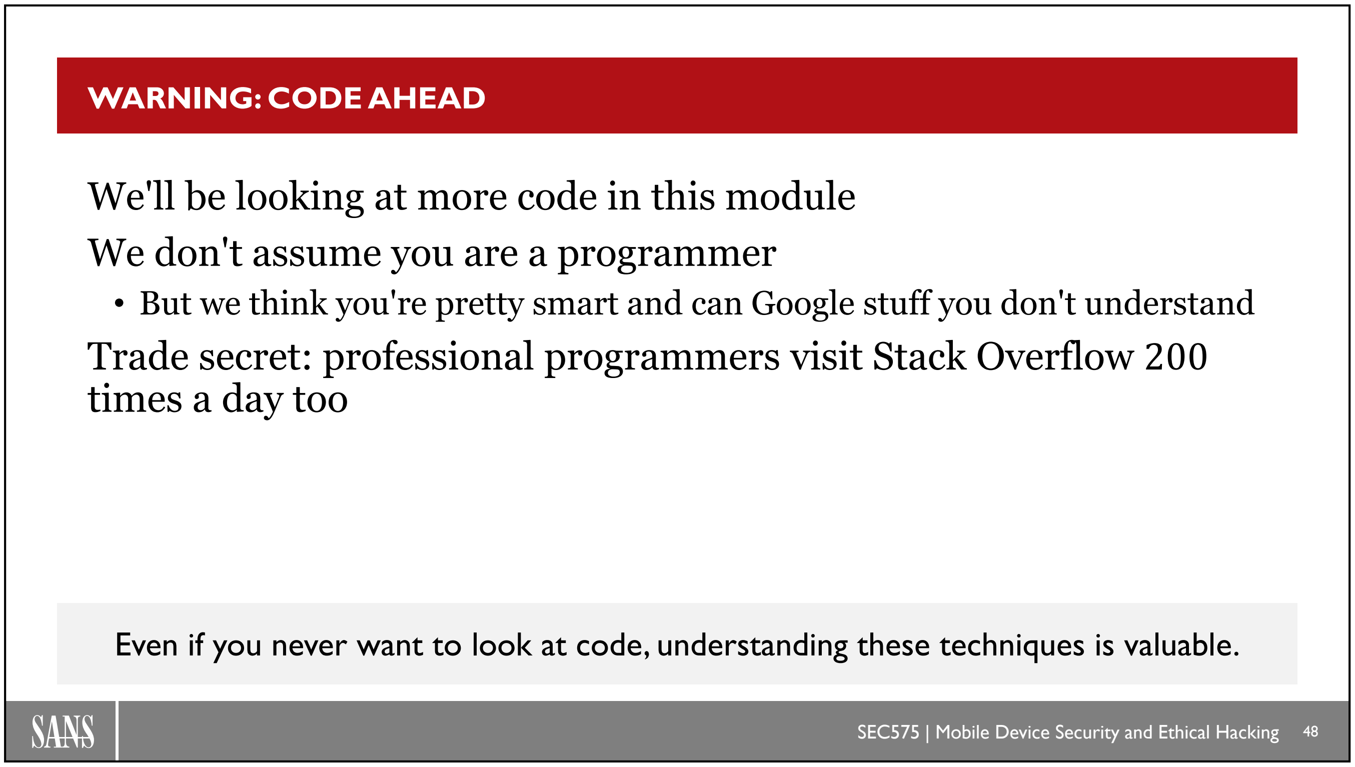
SANS SEC575 | Mobile Device Security and Ethical Hacking 47

Reverse Engineering Obfuscated Applications

In the last module, we looked at techniques for reverse engineering Android and iOS applications. While iOS apps are compiled to machine language instructions, most Android apps include the Dalvik bytecode instructions in an intermediate language format, which makes it straightforward to use Jadx and other decompilers to reconstruct the Java source. For mobile application reverse engineers, Android is a straightforward platform to attack: simply open the APK file in Jadx and start analyzing the reconstructed source. This can take hours or days to produce meaningful results, but it is a big advantage for an analyst and a disadvantage for the app author.

To overcome this problem, many application developers will utilize application obfuscation, a mechanism that allows the developer to retain the original source code in their development environment, but ship the functional application with obfuscated class, method, and variable names. Alternatively, developers might turn to third-party platforms, such as PhoneGap, Xamarin, and Unity, to develop their applications, packaging the application into an Android or an iOS application (or both).

As security analysts, we need to have the capability to handle these corner cases, where understanding the application source might not be as simple as just reading the reconstructed Java with Jadx. In this module, we'll look at overcoming Android application obfuscation first, then we'll look at third-party platforms.



WARNING: CODE AHEAD

We'll be looking at more code in this module

We don't assume you are a programmer

- But we think you're pretty smart and can Google stuff you don't understand

Trade secret: professional programmers visit Stack Overflow 200 times a day too

Even if you never want to look at code, understanding these techniques is valuable.

SANS SEC575 | Mobile Device Security and Ethical Hacking 48

Warning: Code Ahead

Before we jump into this module, I want to take a minute to give you fair warning: there will be code ahead. More than a little code. Probably code in programming languages that you haven't seen before.

Many people in the IT field who are not programmers have hesitance or discomfort when dealing with code. I find that many people have had a bad experience with coding in college (they never "got it") or disliked coding when they tried so much that they eschew any mention of programming at all.

Here's the deal, though: if you are doing mobile device security analysis (and, in particular, mobile application security analysis) you're going to have to look at code from time to time. This does NOT mean that you have to be a programmer capable of sitting down and blasting out 1,000+ lines of code in one sitting! It does mean that you're going to have to look at stuff with an open mind, searching stuff online you aren't sure about, and take good notes.

For those of you who are still doubtful, I'd like to let you in on a trade secret. Professional programmers, the people who code every day for work, even they visit StackOverflow.com about 200 times a day to look stuff up. Programming concepts become ingrained after a while, and style and skill develop over time, but with all the rapid changes in languages, and the numerous libraries and idiosyncrasies in languages themselves, professional programmers have to look stuff up too.

If I haven't convinced you yet, let me try one more tactic. Even if you never want to look at code, understanding the techniques that we're covering is still valuable. Even if you aren't going to reverse engineer applications, understanding the practice, the techniques, and the opportunity for an adversary to do so makes you a better analyst and allows you to make informed decisions about risk.

PROGUARD

Included with Android Studio (free), turned off by default

Turned on, ProGuard substitutes source when the APK is built

- Renaming classes, methods, and variables to obfuscated names.
- `com.google.util.concurrent (pageResult)` becomes `com.a.a.a(a)`

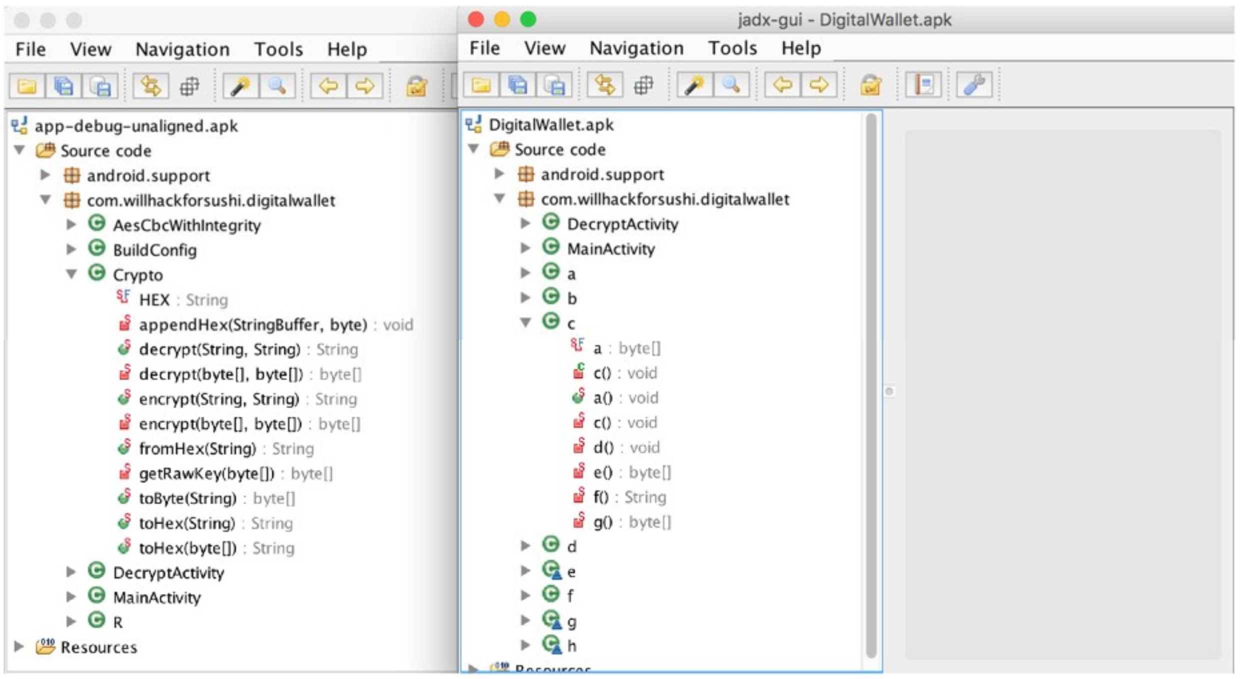
For reverse engineering, ProGuard makes our life hard(er)

SEC575 | Mobile Device Security and Ethical Hacking 49

ProGuard

The most common Android obfuscation tool is ProGuard. Included with Android Studio for free, ProGuard allows a developer to turn on application obfuscation when the app is built. The developer still maintains the code that he or she has written, but the output generated by the build process is converted to an obfuscated format before being distributed as an application.

ProGuard is turned off by default in Android Studio, but when a developer turns it on and requests a new build, the library will rename classes (collections of functionality and data in the form of variables), methods (the functions themselves), and variables to obfuscated names. For example, the class, method, and variable `com.google.util.concurrent(pageResult)` is converted into `com.a.a.a(a)`. This makes our lives harder as analysts because while `com.google.util` is a class we can look up, `concurrent` is a method we can read about, and pageResult is a variable we can relate to, `com.a.a.a(a)` has no intuitive meaning.


File View Navigation Tools Help
jadx-gui - DigitalWallet.apk
File View Navigation Tools Help

```
app-debug-unaligned.apk
▼ Source code
  ▶ android.support
  ▼ com.willhackforsushi.digitalwallet
    ▶ AesCbcWithIntegrity
    ▶ BuildConfig
    ▼ Crypto
        HEX : String
        appendHex(StringBuffer, byte) : void
        decrypt(String, String) : String
        decrypt(byte[], byte[]) : byte[]
        encrypt(String, String) : String
        encrypt(byte[], byte[]) : byte[]
        fromHex(String) : String
        getRawKey(byte[]) : byte[]
        toByte(String) : byte[]
        toHex(String) : String
        toHex(byte[]) : String
    ▶ DecryptActivity
    ▶ MainActivity
    ▶ R
▶ Resources
```

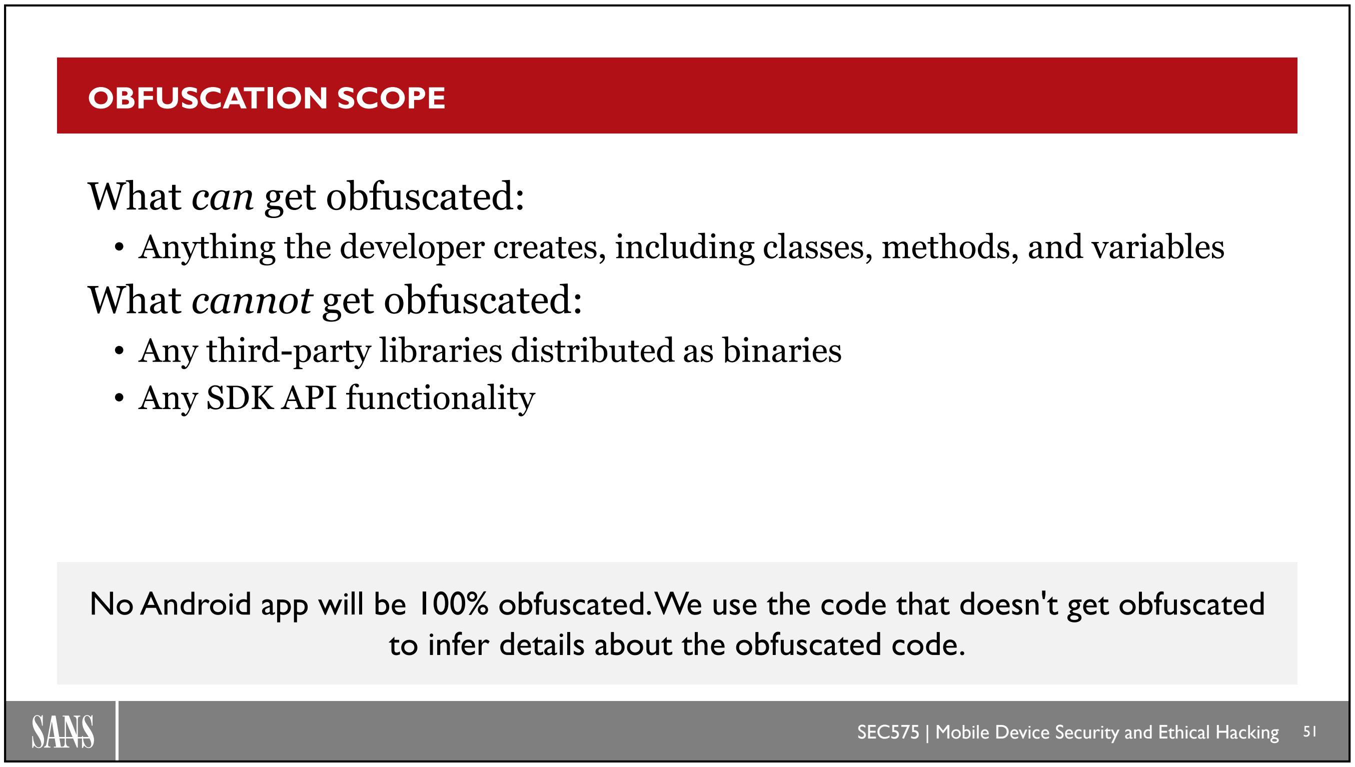
```
DigitalWallet.apk
▼ Source code
  ▶ android.support
  ▼ com.willhackforsushi.digitalwallet
    ▶ DecryptActivity
    ▶ MainActivity
    ▶ a
    ▶ b
    ▼ c
        a : byte[]
        c() : void
        a() : void
        c() : void
        d() : void
        e() : byte[]
        f() : String
        g() : byte[]
    ▶ d
    ▶ e
    ▶ f
    ▶ g
    ▶ h
▶ Resources
```

SANS SEC575 | Mobile Device Security and Ethical Hacking 50

Obfuscated Android Methods

The example on this page shows the output from Jadx decompiling an application written by this author. Both examples are from the sample application.

The example on the left shows the application without ProGuard obfuscation, disclosing class names (`com.willhackforsushi.digitalwallet.Crypto`), method names (`decrypt`, `encrypt`, `getRawKey`), and more. The example on the right is the same application, compiled with ProGuard turned on. The developer still has the original sources with descriptive names to work with, but the reverse engineer does not get the original content.



OBFUSCATION SCOPE

What *can* get obfuscated:

- Anything the developer creates, including classes, methods, and variables

What *cannot* get obfuscated:

- Any third-party libraries distributed as binaries
- Any SDK API functionality

No Android app will be 100% obfuscated. We use the code that doesn't get obfuscated to infer details about the obfuscated code.

SANS SEC575 | Mobile Device Security and Ethical Hacking 51

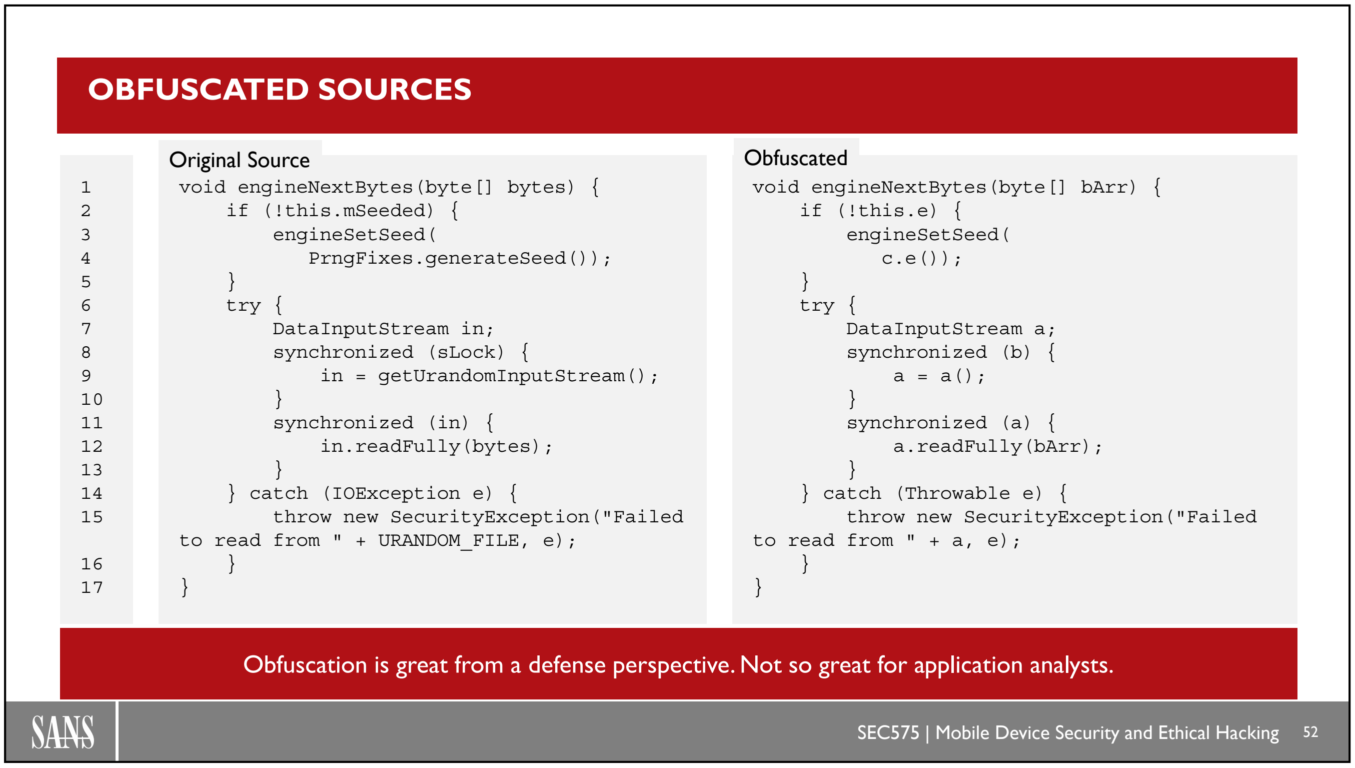
Obfuscation Scope

When a developer uses an obfuscation tool, some components of the source will become obfuscated, while others remain consistent with the original source.

An obfuscator can rename any of the code written by the developer, including classes, methods, and variables. ProGuard takes advantage of this to make the code hard to read, changing `getPassword()` to `c()`.

However, ProGuard cannot rename all of the code used within an application. Obfuscators cannot rename any of the classes, methods, or variables referenced in third-party libraries (when distributed as a binary). Most importantly, the obfuscator cannot rename any parts of the standard Android SDK API.

An Android app will never be 100% obfuscated. Knowing this, we can use the code that doesn't get obfuscated to infer details about the obfuscated code. Let's take a look at an example.



OBFUSCATED SOURCES

Original Source

```
1   void engineNextBytes(byte[] bytes) {
2       if (!this.mSeeded) {
3           engineSetSeed(
4              PrngFixes.generateSeed());
5       }
6       try {
7           DataInputStream in;
8           synchronized (sLock) {
9               in = getUrandomInputStream();
10          }
11          synchronized (in) {
12              in.readFully(bytes);
13          }
14      } catch (IOException e) {
15          throw new SecurityException("Failed
    to read from " + URANDOM_FILE, e);
16      }
17  }
```

Obfuscated

```
1   void engineNextBytes(byte[] bArr) {
2       if (!this.e) {
3           engineSetSeed(
4              c.e());
5       }
6       try {
7           DataInputStream a;
8           synchronized (b) {
9               a = a();
10          }
11          synchronized (a) {
12              a.readFully(bArr);
13          }
14      } catch (Throwable e) {
15          throw new SecurityException("Failed
    to read from " + a, e);
16      }
17  }
```

Obfuscation is great from a defense perspective. Not so great for application analysts.

SANS

SEC575 | Mobile Device Security and Ethical Hacking 52

Obfuscated Sources

The two examples on this page show what an obfuscated method looks like compared to the original source. Using this comparison, we can determine what is changed in the source of an Android application when the developer turns on ProGuard.

On line 1 in the method declaration, we see the method name `engineNextBytes` is not changed. This is because the name is referenced elsewhere outside of the class, and ProGuard opts to retain the original name to avoid breaking application functionality. If the method is only used within a class, ProGuard would rename it to `a`, `b`, `c`, etc. What is changed on line 1 is the parameter variable name. Originally called `bytes`, ProGuard obfuscates the name, leaving Jadx to use the generic name `bArr` for *byte array*.

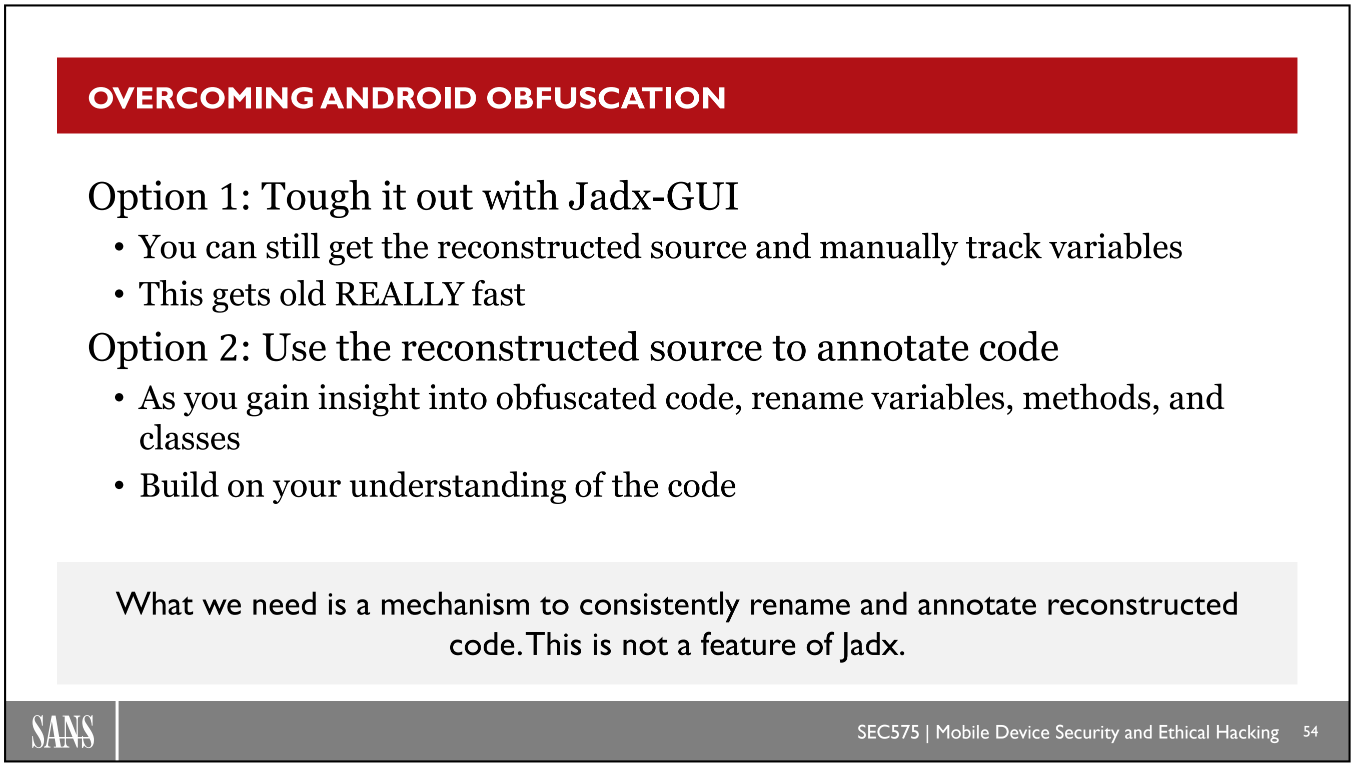
On line 2 of the original source, the code references a *class* variable `this.mSeeded`. Instead of retaining this variable name chosen by the developer, ProGuard leaves us with `this.e`, a generic name that offers no insight into what it could mean.

ProGuard's renaming of code continues on line 4. As the first parameter of the `engineSetSeed` call (on line 3), the developer calls `PrngFixes.generateSeed()`, which we infer as a mechanism to establish the seed value necessary for a pseudorandom number generator (PRNG) to return random values. In the obfuscated version, however, we see a function call to `c.e()`. Here ProGuard has renamed the `PrngFixes` class to `c`, and the `generateSeed` method to `e`, leaving the reverse engineering analyst with little insight into what is actually happening in this code. Furthermore, the name `e` is used in multiple places, many of which are not related variables or methods, just name collisions.

Notice how, on line 7, the developer declares the variable `in` that is of the type `DataInputStream`. In the obfuscated code, this variable is renamed to `a`. This is similar to what we've seen before, but look at lines 16–17 as well: the original source used the constant `URANDOM_FILE` as the variable, and the obfuscated source uses `a` again. The initial declaration of `a` on line 7 is inside the `try {}` block and is accessible only within that block. The reference to `a` on line 17 is outside the try block, so they are treated as two different variables (despite having the same name).

It's important to note that ProGuard *cannot* change all the code during the obfuscation process. For example, the line 3 `engineSetSeed()` method is invoking functionality from a third-party library that is not obfuscated—ProGuard cannot change the name `engineSetSeed()` without breaking the application functionality. Similarly, the object type name `DataInputStream` is a standard type that ProGuard cannot obfuscate. `SecurityException()` is likewise an Android built-in method that escapes obfuscation as well. ProGuard can obfuscate a lot, but not everything, leaving some artifacts for us to use to interpret the code.

Confusing? Yes. Obfuscation makes it harder to reverse engineer code. That's a security improvement for the developer, but it's not insurmountable for the analyst to overcome.



OVERCOMING ANDROID OBFUSCATION

Option 1: Tough it out with Jadx-GUI

- You can still get the reconstructed source and manually track variables
- This gets old REALLY fast

Option 2: Use the reconstructed source to annotate code

- As you gain insight into obfuscated code, rename variables, methods, and classes
- Build on your understanding of the code

What we need is a mechanism to consistently rename and annotate reconstructed code. This is not a feature of Jadx.

SANS | SEC575 | Mobile Device Security and Ethical Hacking 54

Overcoming Android Obfuscation

As mobile application analysts, we need to be able to overcome the hardship imposed by application obfuscation when trying to understand what is really happening in an application. We have two primary options.

Option 1 is to tough it out with Jadx-GUI. Even though the application is obfuscated, we can open the APK in Jadx and get the reconstructed source code. The classes, methods, and variables don't make any sense, and there will be a lot of repeated variable names, which makes analysis more difficult, but you could conceivably spend time reading and understanding the code. Or you might get really lucky and find just the right block of code that is the interesting bit you're looking for. Maybe. It's not likely.

Option 2 is to use Jadx to build the reconstructed code and then annotate the code as you review it. At first, the obfuscated code in reconstructed form won't make much sense, but after a little while, you will figure out small pieces of functionality, correctly identify what a variable or method actually does, and build on your understanding of the code. By gaining insight into the obfuscated application a little at a time, you can annotate the reconstructed sources and turn them into something that is more easily understood.

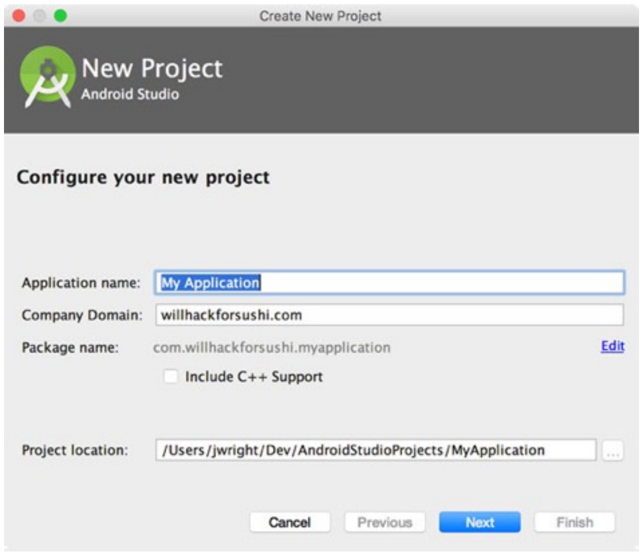
The ability to annotate reconstructed source is not a feature of Jadx. Jadx will build the reconstructed source, but the GUI presents it as is, without the ability to add comments or rename classes, methods, and variables. To annotate the source, we need to utilize a second tool.

ANDROID STUDIO

Free IDE for Android development from Google

Can import source from the filesystem into a new project

Allows the user to refactor and comment on reconstructed source



Configure your new project

Application name: My Application
Company Domain: willhackforsushi.com
Package name: com.willhackforsushi.myapplication Edit
Include C++ Support
Project location: /Users/jwright/Dev/AndroidStudioProjects/MyApplication

Android Studio

Android Studio is a free Integrated Development Environment (IDE) for Android development from Google. Fortunately, Android Studio will happily import the reconstructed source code generated by the `jadx` utility as a new project. Once imported into Android Studio, we can work with the reconstructed code like the original developer would, gaining the ability to annotate the code with comments as we start to understand the app functionality, and to *refactor* methods, classes, and variable names.

JADX TO ANDROID STUDIO

Using `jadx` at the command line, create reconstructed source from APK

Start Android Studio, click Import project

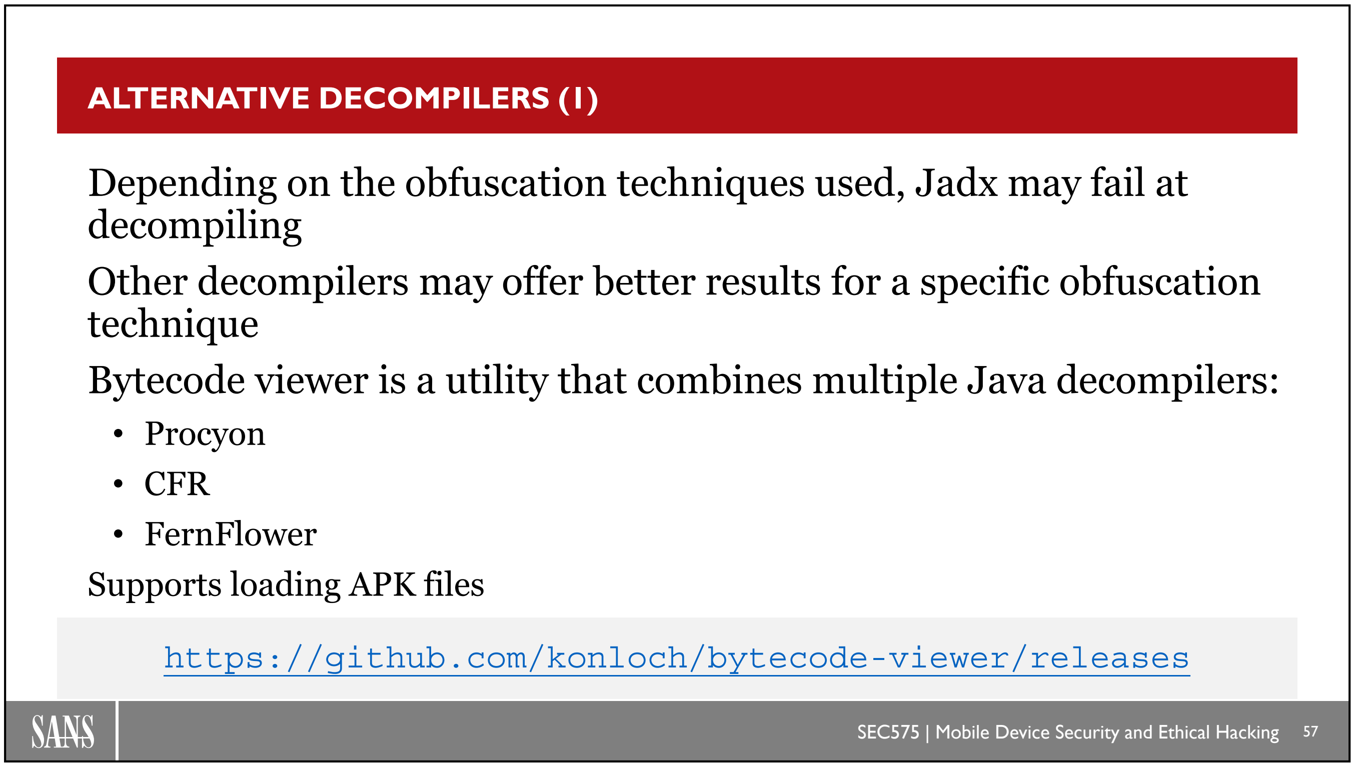
- Choose the Jadx output directory, then click Next | Next | Next | Next | Next | Next | Finish

```
$ mkdir digitalwallet
$ jadx -d digitalwallet/ DigitalWallet.apk
...
$ ls digitalwallet/
AndroidManifest.xml  com/  android/  res/
```

Jadx to Android Studio

The process to bring an APK file into Android Studio is fairly straightforward. First, create a directory where you will store the reconstructed source. Next, use the jadx utility as shown on this page to extract the APK file bytecode and build the reconstructed Java source code.

After you extract the reconstructed source code, open Android Studio and click *Import project*. When prompted, choose the Jadx directory, then finish the wizard, accepting all the defaults. Easy!



ALTERNATIVE DECOMPILERS (1)

Depending on the obfuscation techniques used, Jadx may fail at decompiling

Other decompilers may offer better results for a specific obfuscation technique

Bytecode viewer is a utility that combines multiple Java decompilers:

- Procyon
- CFR
- FernFlower

Supports loading APK files

https://github.com/konloch/bytecode-viewer/releases

SANS SEC575 | Mobile Device Security and Ethical Hacking 57

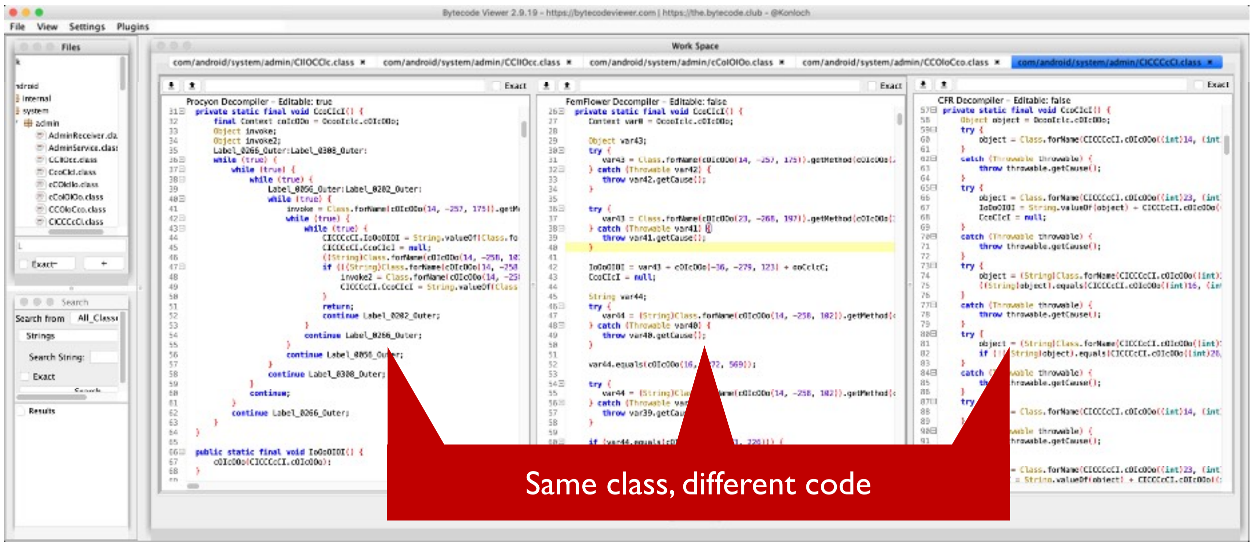
Alternative Decompilers (1)

Decompiling code from compiled code to readable code is always a difficult problem. Jadx usually does a good job, but depending on the technique that was used, Jadx may produce code that is very hard to read. Even worse, some obfuscators specifically create constructs that target popular decompilers and try to make the decompilers crash to slow down the reverse engineering process.

Fortunately, Jadx isn't the only available Java decompiler. Other popular decompilers are Procyon, CFR, FernFlower, and JD. Each of these decompilers may work better or worse on specific applications and it is therefore a good idea to try multiple decompilers on the same application. Where one decompiler might crash, another may succeed.

The application bytecode viewer is a simple utility that combines these decompilers in one easy-to-use application. Although it was originally created as a Java decompilation tool, it now also supports APK files.

ALTERNATIVE DECOMPILERS (2)


Bytecode Viewer 2.9.19 - https://bytecodeviewer.com | https://the.bytecode.club - @Konloch
File View Settings Plugins
Files
Work Space
com/android/system/admin/CllOCClc.class
com/android/system/admin/CCllOcc.class
com/android/system/admin/cColOlOo.class
com/android/system/admin/CCOloCco.class
com/android/system/admin/CICCCcCl.class
Exact
Search
Search from All_Classes
Strings
Search String:
Results
Same class, different code

Procyon Decompiler - Editable: true

```
private static final void CcoClcI() {
    final Context coIcODo = OcooIclc.cOIcODo;
    Object invoke;
    Object invoke2;
    Label_0266_Outer:Label_0308_Outer:
    while (true) {
        while (true) {
            while (true) {
                Label_0056_Outer:Label_0202_Outer:
                while (true) {
                    invoke = Class.forName(cOIcODo(14, -257, 175)).getM
                    while (true) {
                        while (true) {
                            CICOCcCI.IoOoOIOI = String.valueOf(Class.fo
                            CICOCcCI.CcoClcI = null;
                            ((String)Class.forName(cOIcODo(14, -258, 10
                            if (!((String)Class.forName(cOIcODo(14, -258
                                invoke2 = Class.forName(cOIcODo(14, -25
                                CICOCcCI.CcoClcI = String.valueOf(Class
                            }
                            return;
                            continue Label_0202_Outer;
                        }
                        continue Label_0266_Outer;
                    }
                    continue Label_0056_Outer;
                }
                continue Label_0308_Outer;
            }
            continue;
        }
        continue Label_0266_Outer;
    }
}

public static final void IoOoOIOI() {
    cOIcODo(CICOCcCI.cOIcODo);
}
```

FernFlower Decompiler - Editable: false

```
private static final void CcoClcI() {
   Context var0 = OcooIclc.cOIcODo;

   Object var43;
   try {
      var43 = Class.forName(cOIcODo(14, -257, 175)).getMethod(cOIcODo(
   } catch (Throwable var42) {
      throw var42.getCause();
   }

   try {
      var43 = Class.forName(cOIcODo(23, -268, 197)).getMethod(cOIcODo(
   } catch (Throwable var41) {
      throw var41.getCause();
   }

   IoOoOIOI = var43 + cOIcODo(-36, -279, 123) + ooCclcC;
   CcoClcI = null;

   String var44;
   try {
      var44 = (String)Class.forName(cOIcODo(14, -258, 182)).getMethod(c
   } catch (Throwable var40) {
      throw var40.getCause();
   }

   var44.equals(cOIcODo(16,    72, 569));

   try {
      var44 = (String)Cla      Name(cOIcODo(14, -258, 182)).getMethod(c
   } catch (Throwable var
      throw var39.getCau
   }

   if (var44.equals(cO                 , 220)) {
```

CFR Decompiler - Editable: false

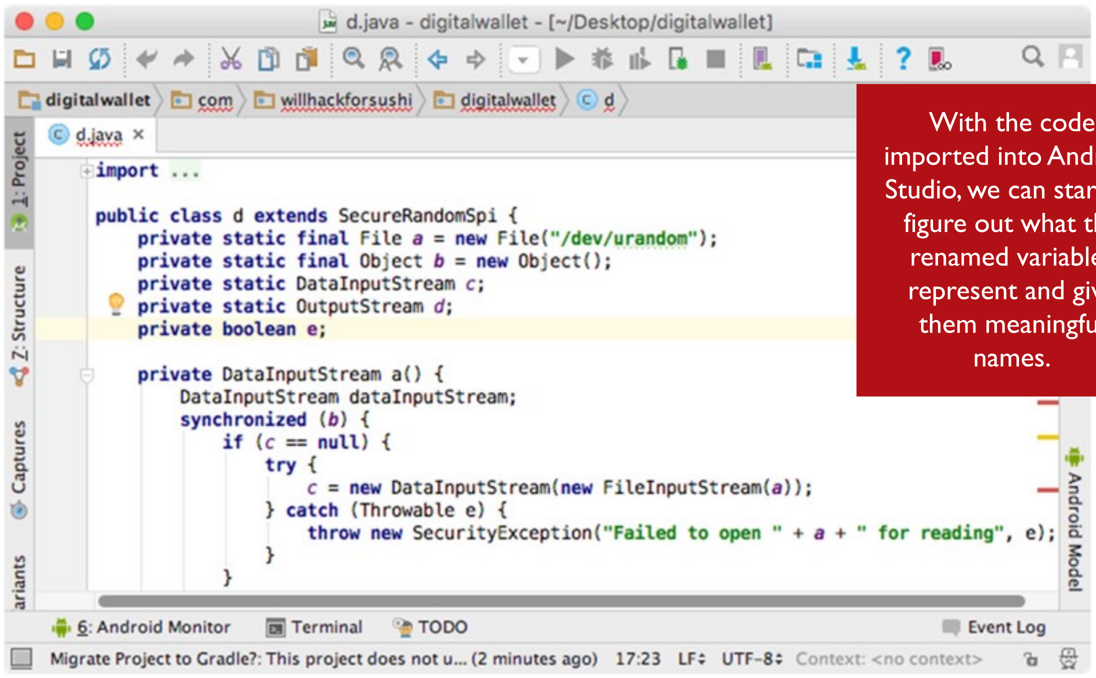
```
private static final void CcoClcI() {
    Object object = OcooIclc.cOIcODo;
    try {
        object = Class.forName(CICCCcCI.cOIcODo((int)14, (int
    }
    catch (Throwable throwable) {
        throw throwable.getCause();
    }
    try {
        object = Class.forName(CICCCcCI.cOIcODo((int)23, (int
        IoOoOIOI = String.valueOf(object) + CICCCcCI.cOIcODo(
        CcoClcI = null;
    }
    catch (Throwable throwable) {
        throw throwable.getCause();
    }
    try {
        object = (String)Class.forName(CICCCcCI.cOIcODo((int)
        ((String)object).equals(CICCCcCI.cOIcODo((int)16, (in
    }
    catch (Throwable throwable) {
        throw throwable.getCause();
    }
    try {
        object = (String)Class.forName(CICCCcCI.cOIcODo((int)
        if (!((String)object).equals(CICCCcCI.cOIcODo((int)20,
    }
    catch (T    wable throwable) {
        th    hrowable.getCause();
    }
    try
           = Class.forName(CICCCcCI.cOIcODo((int)14, (int
    }
           wable throwable) {
           hrowable.getCause();

           = Class.forName(CICCCcCI.cOIcODo((int)23, (int
           = String.valueOf(object) + CICCCcCI.cOIcOOo(
```

Alternative Decompilers (2)

In this example, the same class is loaded in the Procyon (left), FernFlower (middle), and CFR (right) decompilers. While the functionality of the three generated blocks of code is the same, Procyon has created a structure that is very hard to follow with six nested while loops, labels, and strange continue statements. Both FernFlower and CFR have found an alternative Java structure that fits the given Java bytecode, and they have created a much more readable code fragment.

Because decompilers will normally not rename variables or method names, it is possible to mix and match between different decompilers, taking the best decompiled code wherever possible.



d.java - digitalwallet - [~/Desktop/digitalwallet]

digitalwallet > com > willhackforsushi > digitalwallet > d

```
import ...

public class d extends SecureRandomSpi {
    private static final File a = new File("/dev/urandom");
    private static final Object b = new Object();
    private static DataInputStream c;
    private static OutputStream d;
    private boolean e;

    private DataInputStream a() {
        DataInputStream dataInputStream;
        synchronized (b) {
            if (c == null) {
                try {
                    c = new DataInputStream(new FileInputStream(a));
                } catch (Throwable e) {
                    throw new SecurityException("Failed to open " + a + " for reading", e);
                }
            }
```

Migrate Project to Gradle?: This project does not u... (2 minutes ago) 17:23 LF: UTF-8: Context: <no context>

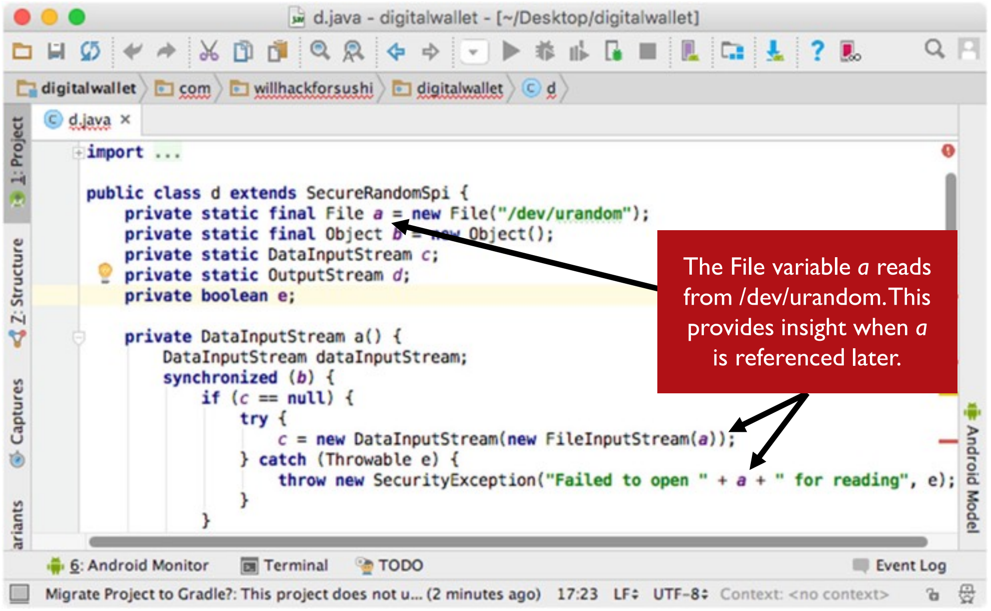
With the code imported into Android Studio, we can start to figure out what the renamed variables represent and give them meaningful names.

Imported Code

After Android Studio finishes importing the reconstructed code, you will have content that looks similar to the example shown on this page. So far, this isn't any better than what we got with Jadx-GUI's output. However, within Android Studio, we can start to figure out small pieces of functionality in the reconstructed code and rename the functional components so they are more intuitive and meaningful to us.

To demonstrate this, we'll look through this small piece of code (and another sample to demonstrate another point not present in this block of code) in the form of several *tips* that you can use in your own assessments.

TIP 1: USE STRINGS


d.java - digitalwallet - [~/Desktop/digitalwallet]
digitalwallet
com
willhackforsushi
digitalwallet
d
d.java
1: Project
Z: Structure
Captures
ariants
Android Model
6: Android Monitor
Terminal
TODO
Event Log
Migrate Project to Gradle?: This project does not u... (2 minutes ago) 17:23 LF: UTF-8: Context: <no context>

```
import ...

public class d extends SecureRandomSpi {
    private static final File a = new File("/dev/urandom");
    private static final Object b = new Object();
    private static DataInputStream c;
    private static OutputStream d;
    private boolean e;

    private DataInputStream a() {
        DataInputStream dataInputStream;
        synchronized (b) {
            if (c == null) {
                try {
                    c = new DataInputStream(new FileInputStream(a));
                } catch (Throwable e) {
                    throw new SecurityException("Failed to open " + a + " for reading", e);
                }
            }
```

The File variable *a* reads from /dev/urandom. This provides insight when *a* is referenced later.

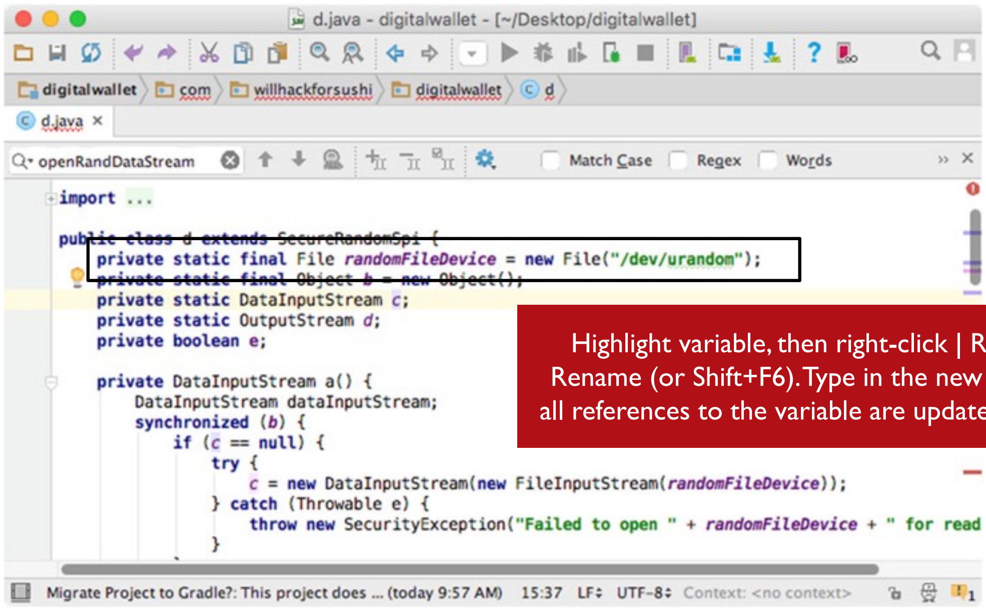
60

Tip 1: Use Strings

One of the quickest ways to gain insight into code is to read the strings embedded in methods. In the example on this slide, we see the string `"/dev/urandom"` near the top of the code, which is the parameter passed to the `File()` method. The return of this method is saved in the `a` variable.

Later in the code, the variable `a` is referred to again, as the parameter to the `FileInputStream()` method and as a string in the `SecurityException()` method. By looking at how this variable is used and the string used to create the variable in `File()`, we see that the variable `a` is really better described as `randomFileDevice`.

REFACTOR AND RENAME



```
import ...

public class d extends SecureRandomSpi {
    private static final File randomFileDevice = new File("/dev/urandom");
    private static final Object b = new Object();
    private static DataInputStream c;
    private static OutputStream d;
    private boolean e;

    private DataInputStream a() {
        DataInputStream dataInputStream;
        synchronized (b) {
            if (c == null) {
                try {
                    c = new DataInputStream(new FileInputStream(randomFileDevice));
                } catch (Throwable e) {
                    throw new SecurityException("Failed to open " + randomFileDevice + " for read
                }
```

```
Migrate Project to Gradle?: This project does ... (today 9:57 AM)   15:37   LF÷   UTF-8÷   Context: <no context>
```

Highlight variable, then right-click | Refactor | Rename (or Shift+F6). Type in the new name, and all references to the variable are updated together.

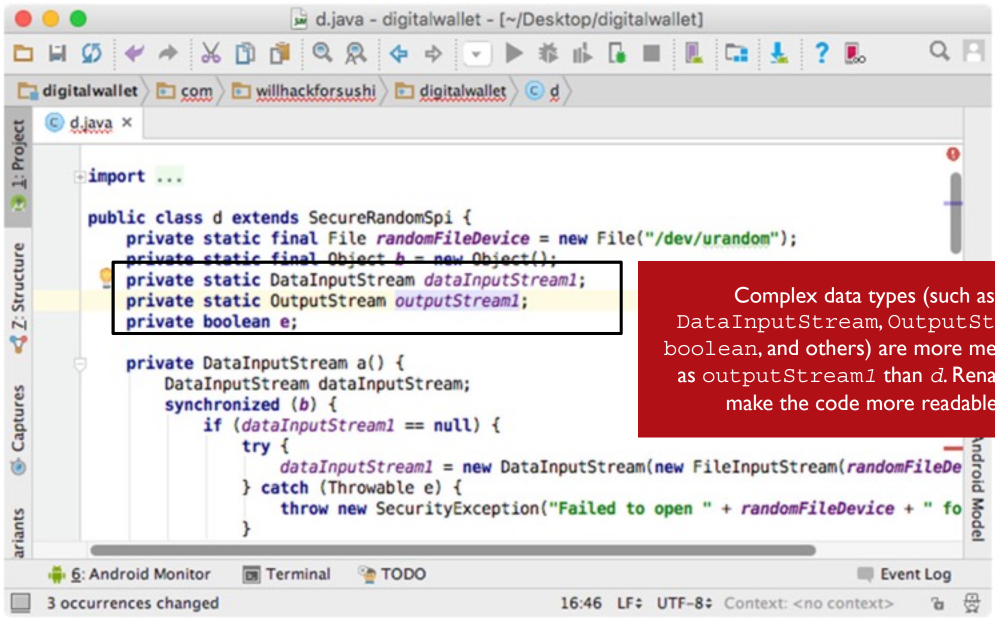
61

Refactor and Rename

In order to make the reconstructed code more legible, we can rename the variable `a` to `randomFileDevice`. You could simply edit all the references to `a` that you see on the page, but it is easy to get tripped up and make a mistake (for example, renaming a variable "`a`" that is in a different scope and represents a different variable altogether, or missing a reference). Instead, we can highlight the variable instance we want to change and let Android Studio handle the renaming process for all the other variables. After highlighting "`a`", right-click on the text and select Refactor | Rename (or press Shift+F6), then type in the new name. All the references to the variable are updated to the new name automatically!

This seems like a little change, but it's a huge help. Android Studio will refactor and rename all the variables to the intuitive names you choose, allowing you to continue your analysis of the obfuscated code.

TIP 2: USE COMPLEX TYPES



d.java - digitalwallet - [~/Desktop/digitalwallet]

digitalwallet 〉 com 〉 willhackforsushi 〉 digitalwallet 〉 d

```
import ...

public class d extends SecureRandomSpi {
    private static final File randomFileDevice = new File("/dev/urandom");
    private static final Object b = new Object();
    private static DataInputStream dataInputStream1;
    private static OutputStream outputStream1;
    private boolean e;

    private DataInputStream a() {
        DataInputStream dataInputStream;
        synchronized (b) {
            if (dataInputStream1 == null) {
                try {
                    dataInputStream1 = new DataInputStream(new FileInputStream(randomFileDe
                } catch (Throwable e) {
                    throw new SecurityException("Failed to open " + randomFileDevice + " fo
                }
```

3 occurrences changed   16:46 LF: UTF-8: Context: <no context>

Complex data types (such as `DataInputStream`, `OutputStream`, `boolean`, and others) are more meaningful as `outputStream1` than *d*. Rename to make the code more readable.

Tip 2: Use Complex Types

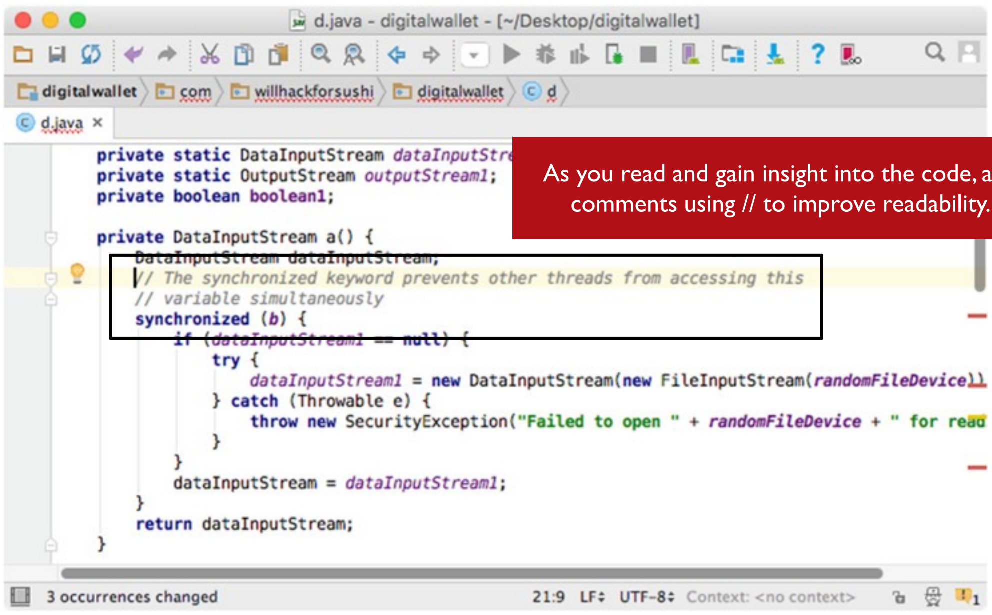
Many of the Android variables will be complex data types. That is, instead of a simple data type such as a `String` or an `Integer`, you will see object types such as `DataInputStream`, `OutputStream`, `boolean`, and more.

Then ProGuard obfuscates the code; it will change the intuitive names chosen by the developer for complex data types to the familiar `a`, `b`, `c`, etc. Even if you don't know exactly what the variable is used for, you can rename it based on the complex object type to make it more intuitive.

In the example on this page, the `DataInputStream` object was formerly `c`, and `OutputStream` was formerly `d`. Using the refactor and rename option, I've renamed the variables to `dataInputStream1` and `outputStream1`. Subsequent references to these variables later in the code are also updated, making it clearer that the variable is being used as the complex data type (instead of `c`).

As you continue your analysis of the code, you can rename these variables again later. `dataInputStream1` may become `randomDataInputStream` after more analysis, making the variable name even more intuitive.

TIP 3: ADD YOUR OWN COMMENTS



```
private static DataInputStream dataInputStre
private static OutputStream outputStream1;
private boolean boolean1;

private DataInputStream a() {
    ~~DataInputStream dataInputStream;~~
    // The synchronized keyword prevents other threads from accessing this
    // variable simultaneously
    synchronized (b) {
        ~~if (dataInputStream1 == null) {~~
            try {
                dataInputStream1 = new DataInputStream(new FileInputStream(randomFileDevice))
            } catch (Throwable e) {
                throw new SecurityException("Failed to open " + randomFileDevice + " for read
            }
        }
        dataInputStream = dataInputStream1;
    }
    return dataInputStream;
}
```

d.java - digitalwallet - [~/Desktop/digitalwallet]
digitalwallet
com
willhackforsushi
digitalwallet
d
d.java
3 occurrences changed
21:9 LF÷ UTF-8÷ Context: <no context>

As you read and gain insight into the code, add comments using // to improve readability.

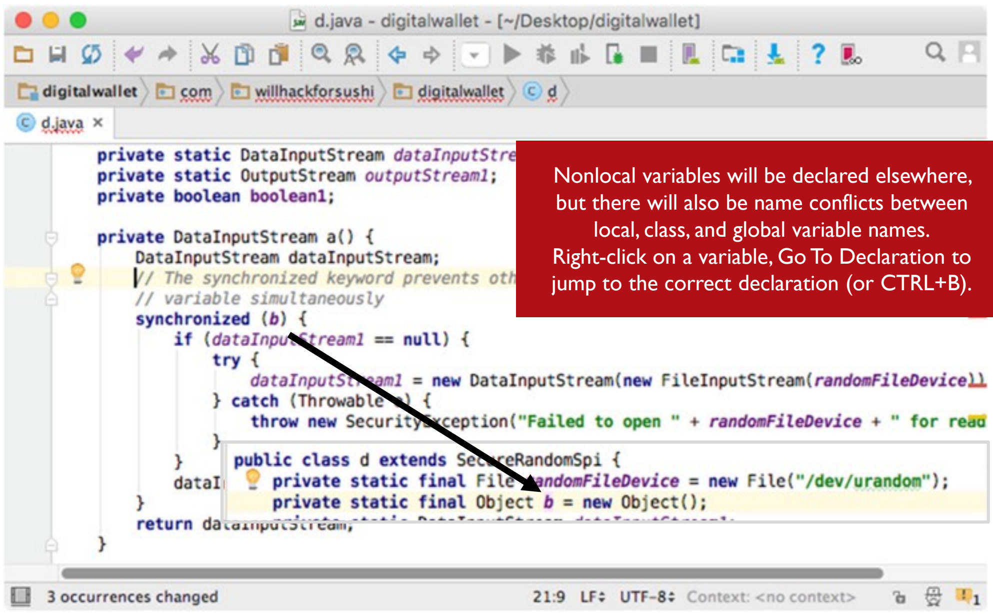
63

Tip 3: Add Your Own Comments

I'm not embarrassed to admit that when writing this slide, I needed to look up what the `synchronized()` method was for (it's used for threading so multiple threads don't try to control the same variable at the same time). After reading a few StackOverflow articles, I returned to this function and added a comment, starting with two slashes, as shown on this page.

Add comments to the code you've reviewed. Even if it's just to make sure that you've *taken a look*, that's OK. It's OK to ask questions in your comments, such as "Is this where the random data is retrieved?" Adding comments will improve readability and help you quickly remember what a block of code or a variable or a whole class is for. This becomes particularly useful hours, days, or months later when you look back on code but have since forgotten what you once knew about it.

TIP 4: USE GO TO DECLARATION


d.java - digitalwallet - [~/Desktop/digitalwallet]
digitalwallet
com
willhackforsushi
digitalwallet
d
d.java
3 occurrences changed
21:9 LF UTF-8 Context: <no context>

```
private static DataInputStream dataInputStre
private static OutputStream outputStream1;
private boolean boolean1;

private DataInputStream a() {
    DataInputStream dataInputStream;
    // The synchronized keyword prevents oth
    // variable simultaneously
    synchronized (b) {
        if (dataInputStream1 == null) {
            try {
                dataInputStream1 = new DataInputStream(new FileInputStream(randomFileDevice))
            } catch (Throwable e) {
                throw new SecurityException("Failed to open " + randomFileDevice + " for read
            }
        }
        dataI
    }
    return dataInputStream;
}
```

```
public class d extends SecureRandomSpi {
    private static final File randomFileDevice = new File("/dev/urandom");
    private static final Object b = new Object();
```

Nonlocal variables will be declared elsewhere, but there will also be name conflicts between local, class, and global variable names. Right-click on a variable, Go To Declaration to jump to the correct declaration (or CTRL+B).

64

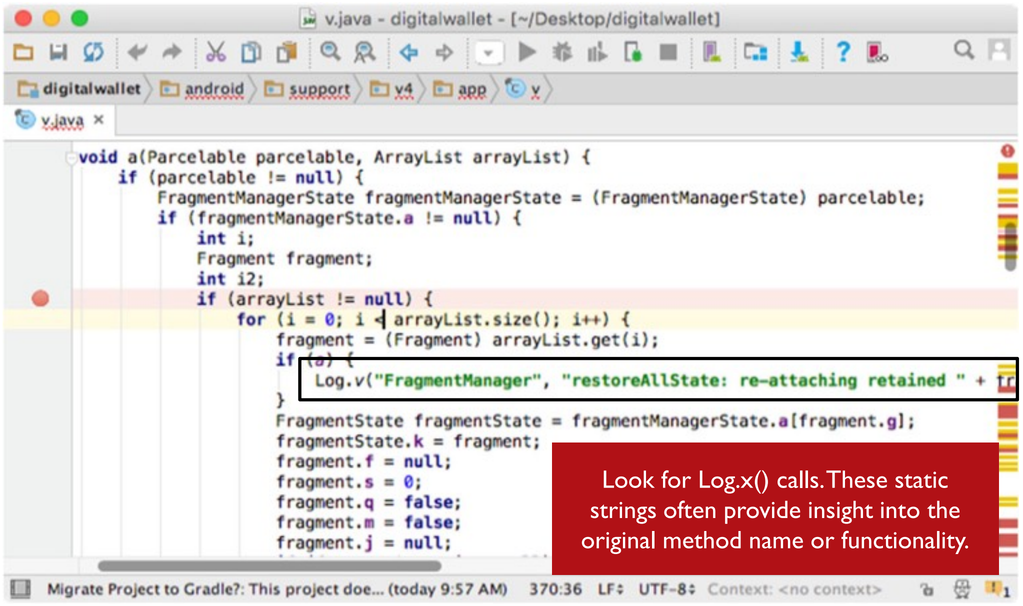
Tip 4: Use Go To Declaration

For this next tip, we need to talk a little about Java variables and *scope*. In Java (and other languages), a variable can be declared in a whole bunch of different places, and how and where the programmer can access the variable will change. For example, if the developer declares `Boolean isJobFinished` at the top of the class before the methods are declared, then `isJobFinished` is a class variable and is accessible anywhere in the class (these are called member variables). By contrast, a developer can declare a variable at the top of a method and have it accessible anywhere in the method (but not outside the method—these are called local variables). Variables can also be declared inside of curly brackets { } and are only accessible in that local block of code (these are called block scope or loop variables).

Now that you've grasped that concept: Why is this important to us for analyzing obfuscated Android apps? Often you will look at a block of code and see a variable only to wonder, "What the HECK is that?" It's not always easy to identify where the variable was declared since it could be a member, local, or block scope variable. This brings up to tip #4: use *Go To Declaration*.

If you're looking at a variable and wondering what the variable type is, or where it was declared, right-click on the variable and choose Go To Declaration (or press CTRL+B). Android Studio will move your cursor to the location where the variable is declared, at the appropriate scope. Once you get the declaration of the variable (and the type), you can start looking for other references to figure out what it does (then refactor and rename it appropriately).

TIP 5: USE LOGGING



```
void a(Parcelable parcelable, ArrayList arrayList) {
    if (parcelable != null) {
        FragmentManagerState fragmentManagerState = (FragmentManagerState) parcelable;
        if (fragmentManagerState.a != null) {
            int i;
            Fragment fragment;
            int i2;
            if (arrayList != null) {
                for (i = 0; i < arrayList.size(); i++) {
                    fragment = (Fragment) arrayList.get(i);
                    if (a) {
                        Log.v("FragmentManager", "restoreAllState: re-attaching retained " + fr
                    }
                    FragmentState fragmentState = fragmentManagerState.a[fragment.g];
                    fragmentState.k = fragment;
                    fragment.f = null;
                    fragment.s = 0;
                    fragment.q = false;
                    fragment.m = false;
                    fragment.j = null;
```

```
Migrate Project to Gradle?: This project doe... (today 9:57 AM)   370:36   LF‡   UTF-8‡   Context: <no context>
```

Look for Log.x() calls. These static strings often provide insight into the original method name or functionality.

Tip 5: Use Logging

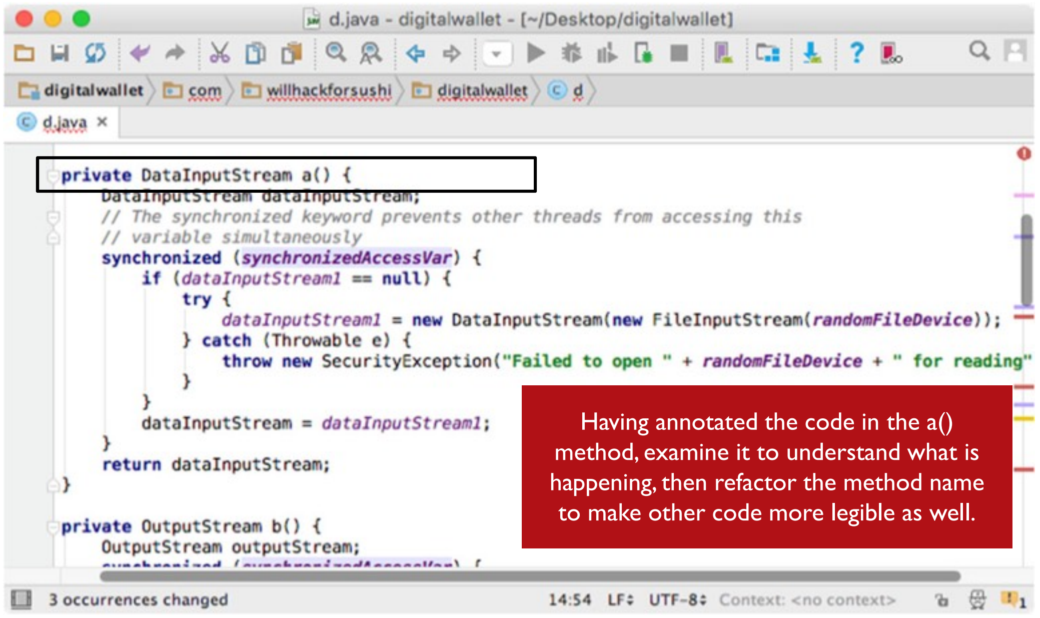
This tip is similar to Tip 1 (Use Strings), but it's really valuable, so I'm emphasizing it here. Developers will often include logging messages (with `Log.v` for verbose logging, `Log.i` for informational logging, `Log.e` for error logging, etc.). The logging message is usually a combination of a string and variables. If you're lucky, the developer will use a consistent format for *what* is included in the logging string.

In the example on this page, the `Log.v` method takes two parameters: a tag (which is usually the class name), then the message itself. These strings are not adjusted by ProGuard, even though the declaration of the class name and the method name *are* obfuscated.

From this logging message, we can surmise that the class is called `FragmentManager`, and the method is called `restoreAllState`. Next up: refactor and rename the class and the method, making it easier for you to read other code where this class and method are used.

Not every developer will use consistent class tags or include the method name in the logging string, but it's fairly common and a big win for the application analyst.

TIP 6: UNDERSTAND, THEN RENAME METHODS



```
private DataInputStream a() {
    DataInputStream dataInputStream;
    // The synchronized keyword prevents other threads from accessing this
    // variable simultaneously
    synchronized (synchronizedAccessVar) {
        if (dataInputStream1 == null) {
            try {
                dataInputStream1 = new DataInputStream(new FileInputStream(randomFileDevice));
            } catch (Throwable e) {
                throw new SecurityException("Failed to open " + randomFileDevice + " for reading"
            }
        }
        dataInputStream = dataInputStream1;
    }
    return dataInputStream;
}

private OutputStream b() {
    OutputStream outputStream;
```

3 occurrences changed 14:54 LF: UTF-8: Context: <no context>

Having annotated the code in the a() method, examine it to understand what is happening, then refactor the method name to make other code more legible as well.

66

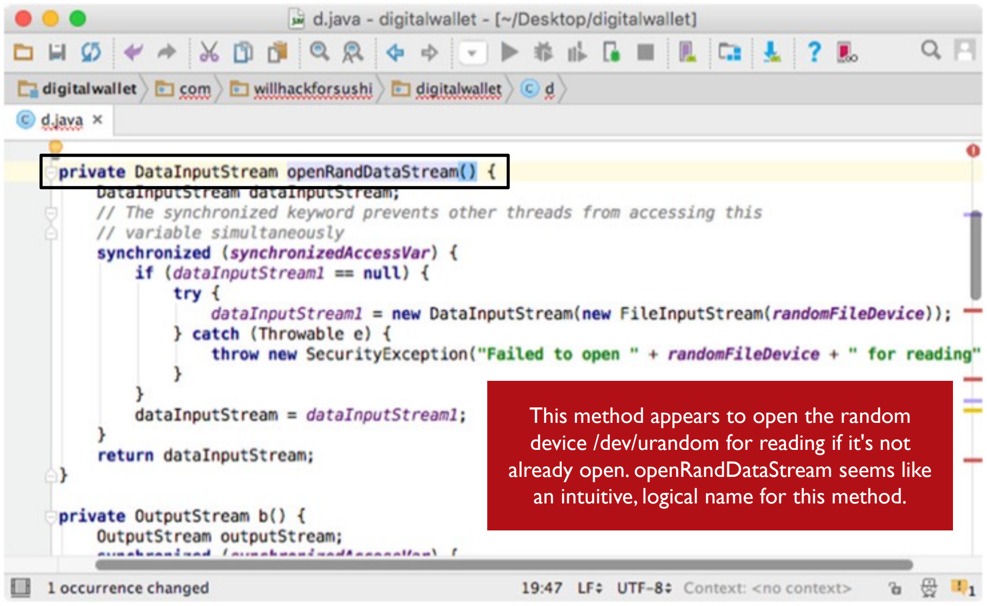
Tip 6: Understand, then Rename Methods

So far, we've looked at multiple tips to take obfuscated code and turn it into something reasonable. In the example on this page, the method `a()` returns the type `DataInputStream`. But what does the method actually do?

From our comment, we recall that the `synchronized` method is making sure other threads aren't interfering with the data. Next, the complex variable `dataInputStream1` is checked to see if it's `null`, and if it is, the method opens the `randomFileDevice` (`/dev/urandom`), returning a file handle populated in `dataInputStream1`. If this fails, the code raises a security exception. Finally, the `dataInputStream` variable is set to the `dataInputStream1` value and returned at the end of the method.

In the beginning, this code looked more complex, with the `a`, `b`, `c`, `d` variables and calls to methods that we weren't familiar with. After a little analysis, though, we can refactor and rename the method `a()` to something more intuitive.

RENAMED METHOD: OPENRANDDATASTREAM



```
private DataInputStream openRandDataStream() {
    DataInputStream dataInputStream;
    // The synchronized keyword prevents other threads from accessing this
    // variable simultaneously
    synchronized (synchronizedAccessVar) {
        if (dataInputStream1 == null) {
            try {
                dataInputStream1 = new DataInputStream(new FileInputStream(randomFileDevice));
            } catch (Throwable e) {
                throw new SecurityException("Failed to open " + randomFileDevice + " for reading"
            }
        }
        dataInputStream = dataInputStream1;
    }
    return dataInputStream;
}

private OutputStream b() {
    OutputStream outputStream;
    synchronized (synchronizedAccessVar) {
```

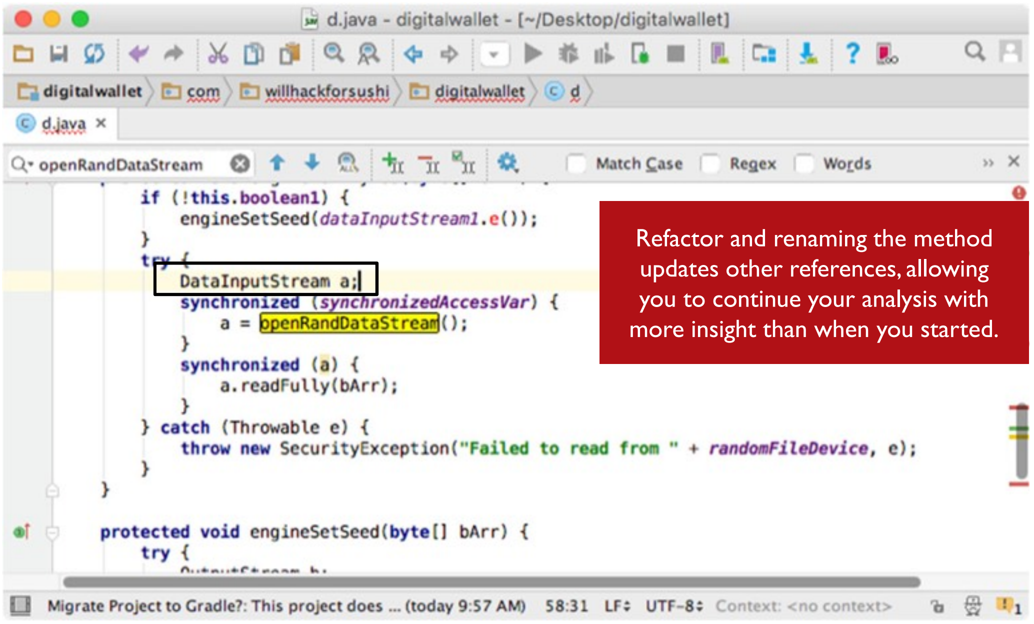
```
1 occurrence changed                19:47  LF÷  UTF-8÷  Context: <no context>
```

This method appears to open the random device /dev/urandom for reading if it's not already open. openRandDataStream seems like an intuitive, logical name for this method.

Renamed Method: `openRandDataStream`

To me, this code looks like a helper function that opens the `/dev/urandom` device for random data, returning a file handle that another process can read from. Instead, calling the method `a()`, I've renamed it to `openRandDataStream()` (again, using the refactor and rename option in Android Studio).

REPEAT THIS PROCESS



d.java - digitalwallet - [~/Desktop/digitalwallet]
digitalwallet
com
willhackforsushi
digitalwallet
d
d.java
openRandDataStream
Match Case
Regex
Words

```
        if (!this.boolean1) {
            engineSetSeed(dataInputStream1.e());
        }
        try {
            DataInputStream a;
            synchronized (synchronizedAccessVar) {
                a = openRandDataStream();
            }
            synchronized (a) {
                a.readFully(bArr);
            }
        } catch (Throwable e) {
            throw new SecurityException("Failed to read from " + randomFileDevice, e);
        }
    }

    protected void engineSetSeed(byte[] bArr) {
        try {
```

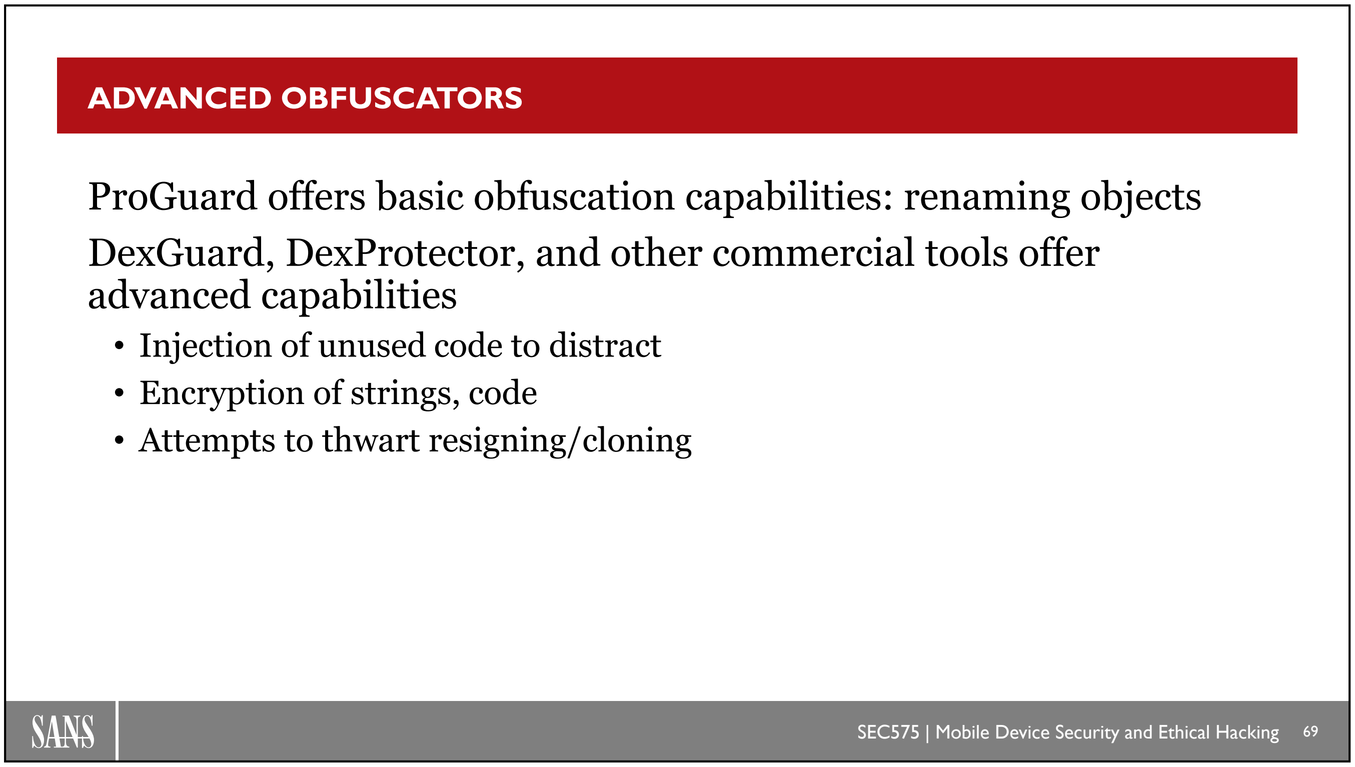
Migrate Project to Gradle?: This project does ... (today 9:57 AM) 58:31 LF: UTF-8: Context: <no context>

Refactor and renaming the method updates other references, allowing you to continue your analysis with more insight than when you started.

Repeat This Process

In practice, a short method like the one we evaluated will take 1–2 minutes to figure out and rename. It seems like a small victory when you consider that there are likely hundreds or thousands of methods and classes waiting to be analyzed as well. However, because we use the refactor and rename functionality, all the references to the former `a()` method are now called `openRandDataStream()`, providing you with a strong point of reference when analyzing subsequent code (as shown on this page).

My recommendation when applying these tips is to start small. Focus on the details first, refactoring and renaming, adding comments as you continue your analysis. After enough of the small stuff, the larger picture becomes more and more clear.



ADVANCED OBFUSCATORS

ProGuard offers basic obfuscation capabilities: renaming objects

DexGuard, DexProtector, and other commercial tools offer advanced capabilities

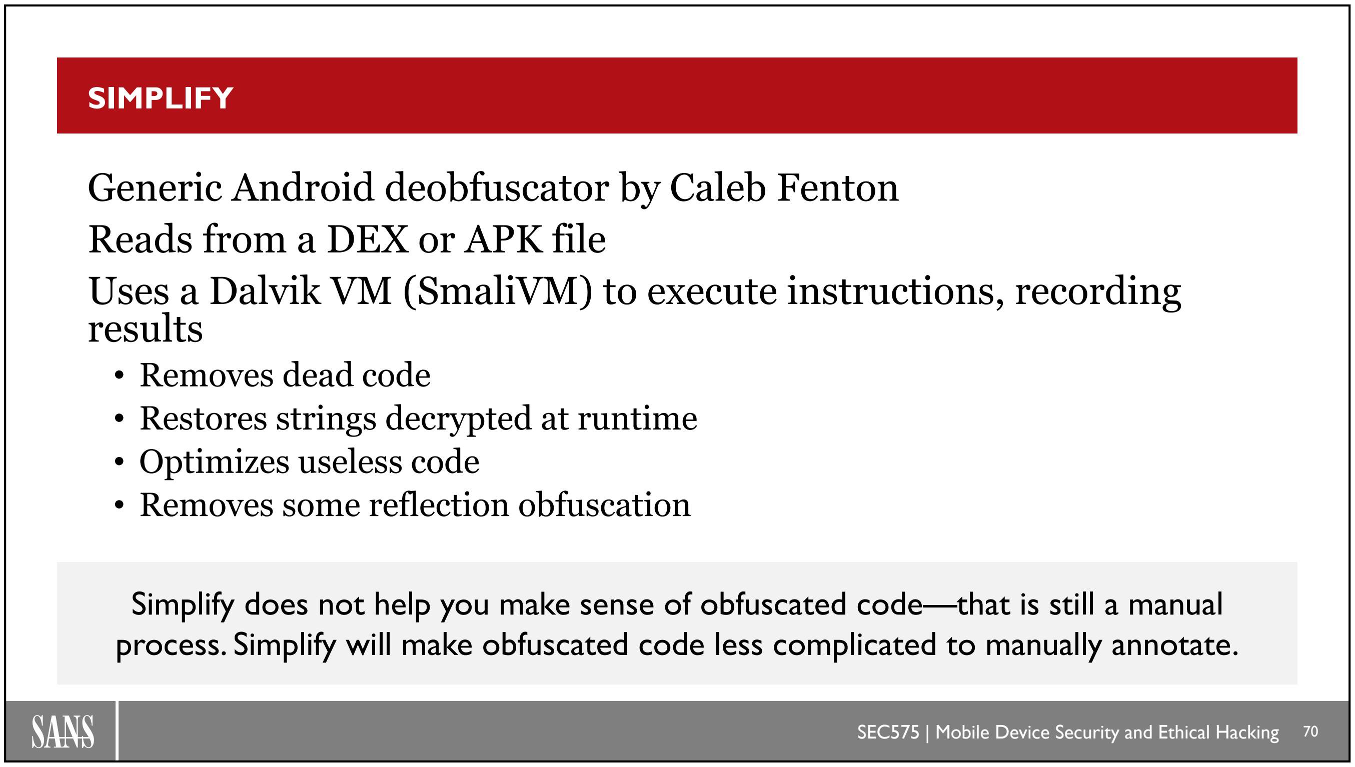
- Injection of unused code to distract
- Encryption of strings, code
- Attempts to thwart resigning/cloning

SEC575 | Mobile Device Security and Ethical Hacking 69

Advanced Obfuscators

Since it is included with Android Studio for free, ProGuard is the most popular obfuscator for Android applications. However, ProGuard offers only basic obfuscation capabilities in the form of renaming objects. Commercial obfuscators, including DexGuard and DexProtector, offer more advanced obfuscation capabilities, including the injection of unused code to distract analysis, encryption of strings and code, and attempts to thwart resigning and application cloning.

ProGuard is straightforward in how it obfuscates an application. Third-party obfuscators require more complex analysis techniques.



SIMPLIFY

Generic Android deobfuscator by Caleb Fenton

Reads from a DEX or APK file

Uses a Dalvik VM (SmaliVM) to execute instructions, recording results

- Removes dead code
- Restores strings decrypted at runtime
- Optimizes useless code
- Removes some reflection obfuscation

Simplify does not help you make sense of obfuscated code—that is still a manual process. Simplify will make obfuscated code less complicated to manually annotate.

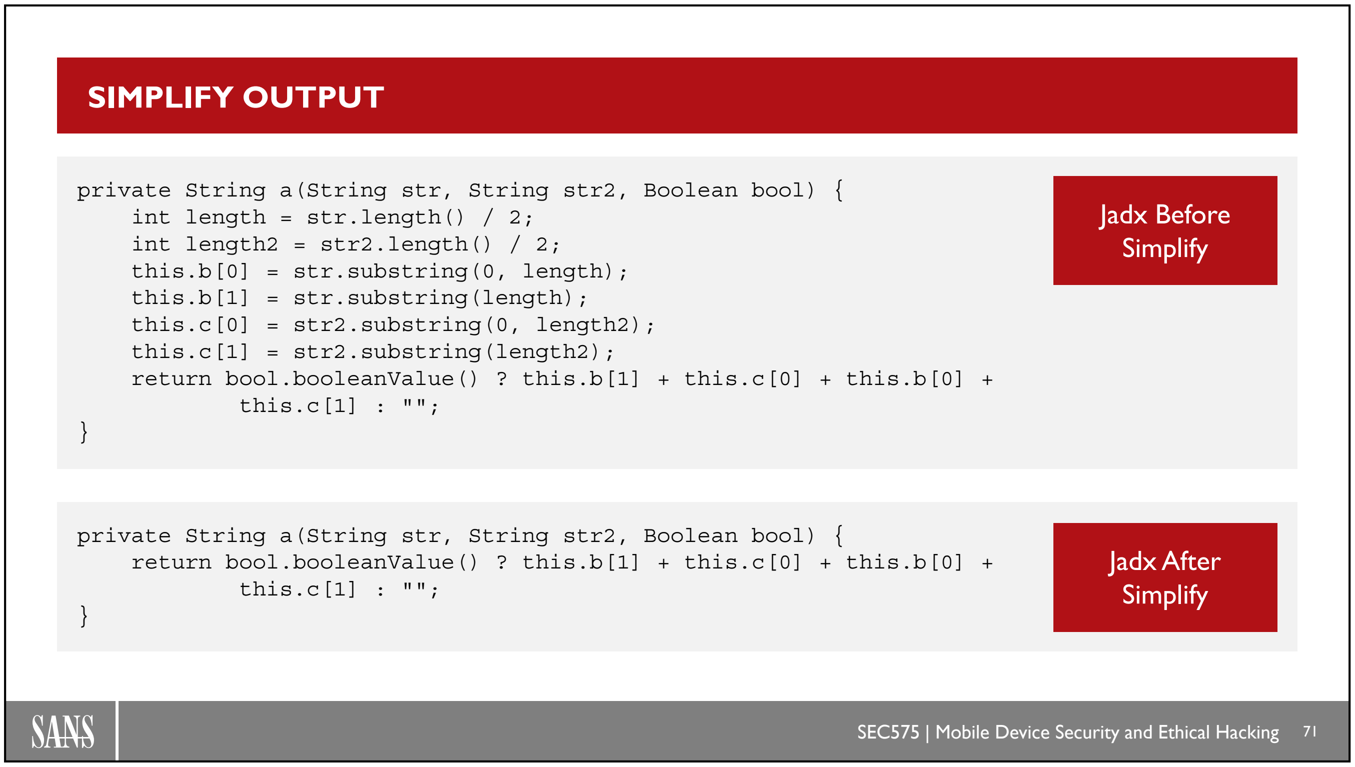
SANS SEC575 | Mobile Device Security and Ethical Hacking 70

Simplify

Simplify is a generic Android deobfuscator by Caleb Fenton (https://github.com/CalebFenton/simplify). Simplify reads from a DEX or APK file and attempts to deobfuscate the application through several techniques, including the use of SmaliVM (also by Fenton), a Dalvik VM that can execute instructions and record the results.

Through SmaliVM, Simplify can detect and remove dead code (code that doesn't do anything or code that isn't reached), it can observe the decryption of strings and restore the original declarations, optimize useless code, and remove some forms of reflection obfuscation (where the application manipulates code references at runtime). These are tremendous features that make the analysis of commercially obfuscated applications much simpler.

However, Simplify does not help you make sense of the obfuscated code. The task of annotating reconstructed code in Android Studio does not go away with Simplify. Rather, Simplify just makes the obfuscated code less complicated to manually annotate.



SIMPLIFY OUTPUT

```
private String a(String str, String str2, Boolean bool) {
    int length = str.length() / 2;
    int length2 = str2.length() / 2;
    this.b[0] = str.substring(0, length);
    this.b[1] = str.substring(length);
    this.c[0] = str2.substring(0, length2);
    this.c[1] = str2.substring(length2);
    return bool.booleanValue() ? this.b[1] + this.c[0] + this.b[0] +
            this.c[1] : "";
}
```

Jadx Before Simplify

```
private String a(String str, String str2, Boolean bool) {
    return bool.booleanValue() ? this.b[1] + this.c[0] + this.b[0] +
            this.c[1] : "";
}
```

Jadx After Simplify

SANS SEC575 | Mobile Device Security and Ethical Hacking 71

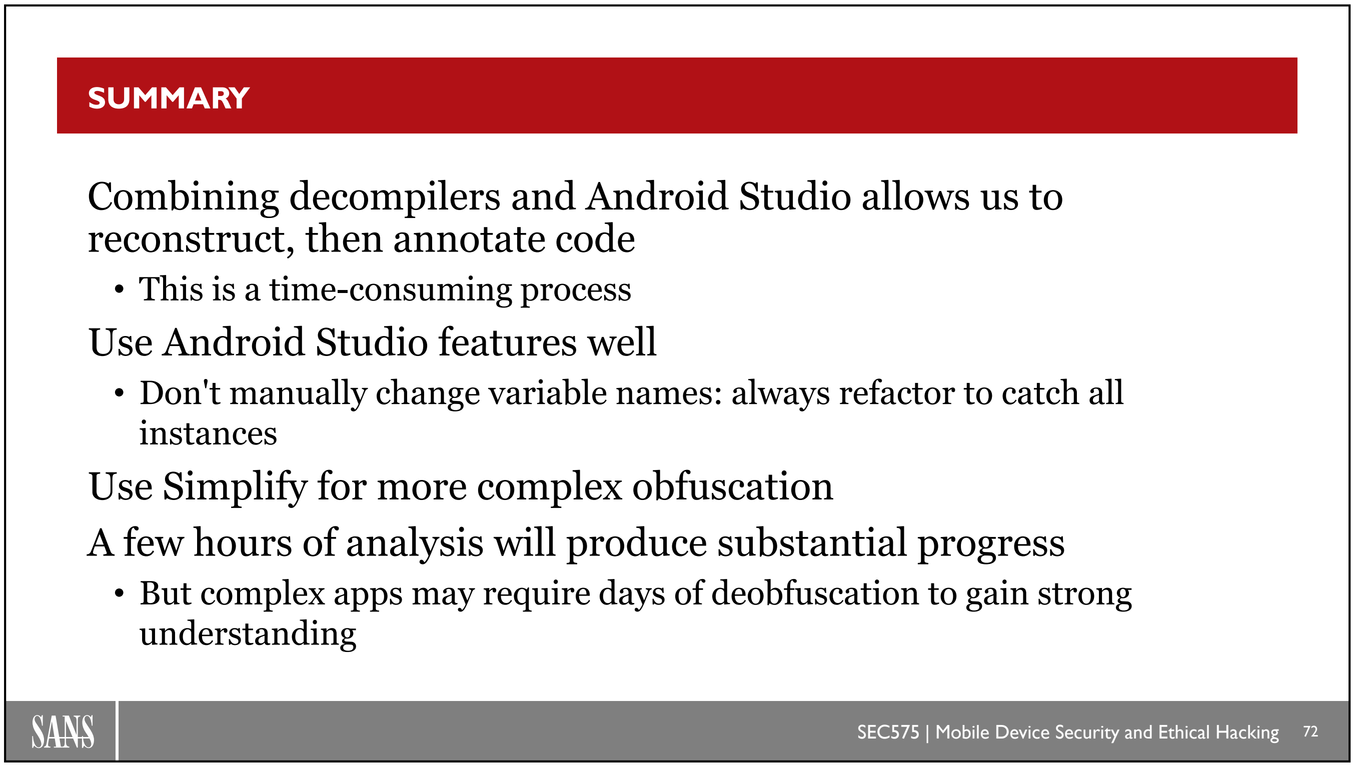
Simplify Output

Simplify reads from an APK or DEX file and will attempt to deobfuscate the entire application by default. This can be problematic in some cases since it will take a long time to deobfuscate large applications, or problematic code can cause Simplify to crash. Instead, you can limit Simplify's processing and deobfuscation to specific objects by specifying the `-it` argument, followed by the class name and `;->`. This will limit Simplify to only the methods in that class name, as shown in the example below (output modified for space):

```
$ java -jar simplify.jar -it 'com/willhackforsushi/securenotepad/d;->'
SecureNotePad.apk
Simplified 6 methods from 1 classes in 3163 ms.
Total optimizations:
             constantized ifs = 0
             constantized ops = 8
             dead assignments removed = 36
             dead ops removed = 0
             dead results removed = 17
Writing output to SecureNotePad_simple.apk
```

Simplify will produce a new output APK file that can be evaluated using the tools we've discussed so far. The examples on this page show the reconstructed code from Jadx before and after deobfuscation with Simplify. In the first example, the code calculates the length of a pair of strings, then uses `substring` to manipulate and shuffle the strings. The app developer added this code to make it more difficult for the analyst to reverse engineer the reconstructed code when trying to figure out what the method does. Simplify's output eliminates all of this nonsense, skipping right to the return value, having determined that the prior code was *dead* or useless, so it eliminated it.

Simplify does not help to annotate the code, but it can be useful to simplify the reconstructed output, making it easier to analyze the source. This is tremendously valuable to our analysis, and a wonderful tool to keep around when you want to perform static analysis on obfuscated Android applications.



SUMMARY

Combining decompilers and Android Studio allows us to reconstruct, then annotate code

- This is a time-consuming process

Use Android Studio features well

- Don't manually change variable names: always refactor to catch all instances

Use Simplify for more complex obfuscation

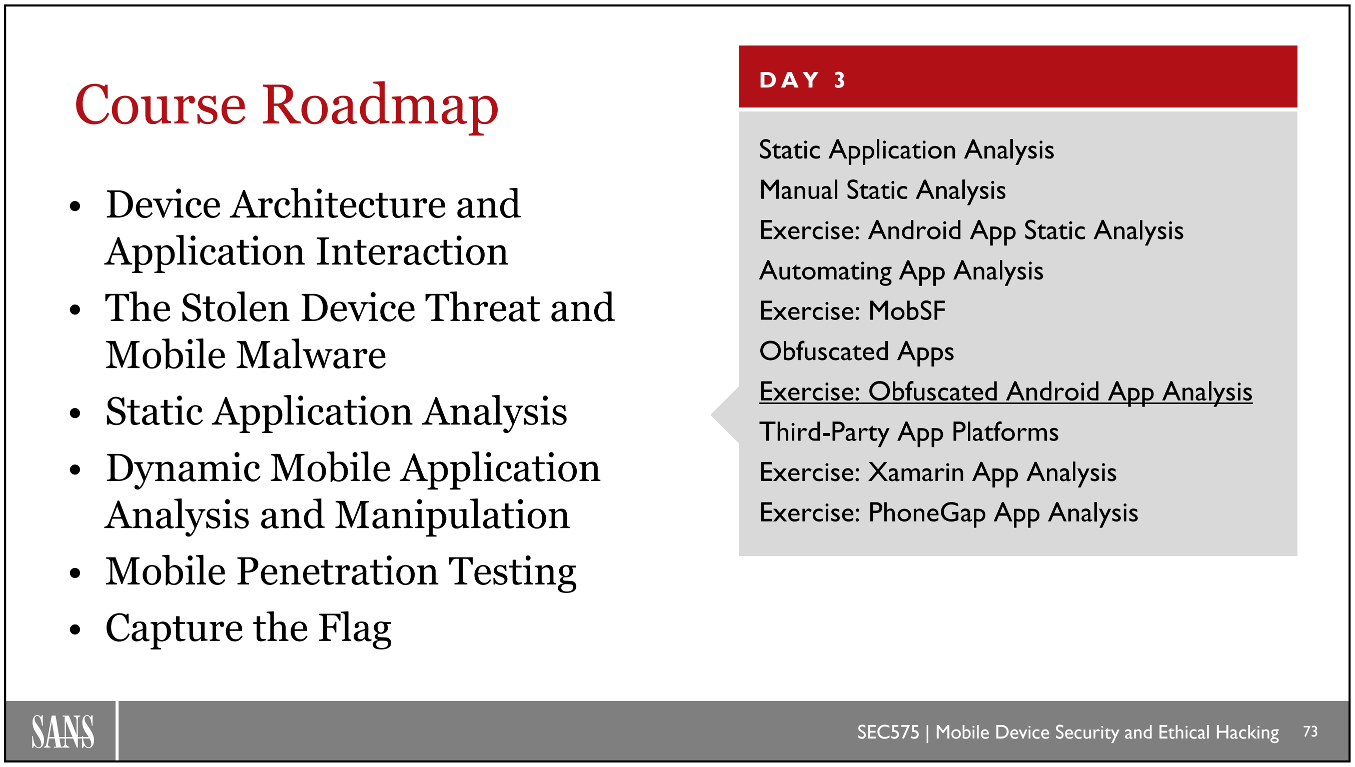
A few hours of analysis will produce substantial progress

- But complex apps may require days of deobfuscation to gain strong understanding

Summary

In this module, we look at the techniques to apply to deobfuscate an Android application. Using Jadx or one of the other decompilers to produce reconstructed Java, then using Android Studio to refactor, rename, and annotate the code, you can produce a meaningful body of work that helps you understand what is happening within the target application. This is a time-consuming process, which requires patience. After a slow start, however, you will find that substantial progress can be made within a few hours, making it possible to gain new insight and a strong understanding of the target application.

Obfuscation beyond renaming of objects requires additional analysis. Fenton's Simplify tool helps with this process, using a Dalvik virtual machine to execute methods, analyze the product, and create a simplified version of the target application. Simplify will help to remove dead code, will decrypt strings, and perform other deobfuscation tasks, but it does not help with application annotation. Simplify is a valuable tool but does not eliminate the need to annotate an application and carefully evaluate the functionality to really understand what an application does.



Course Roadmap

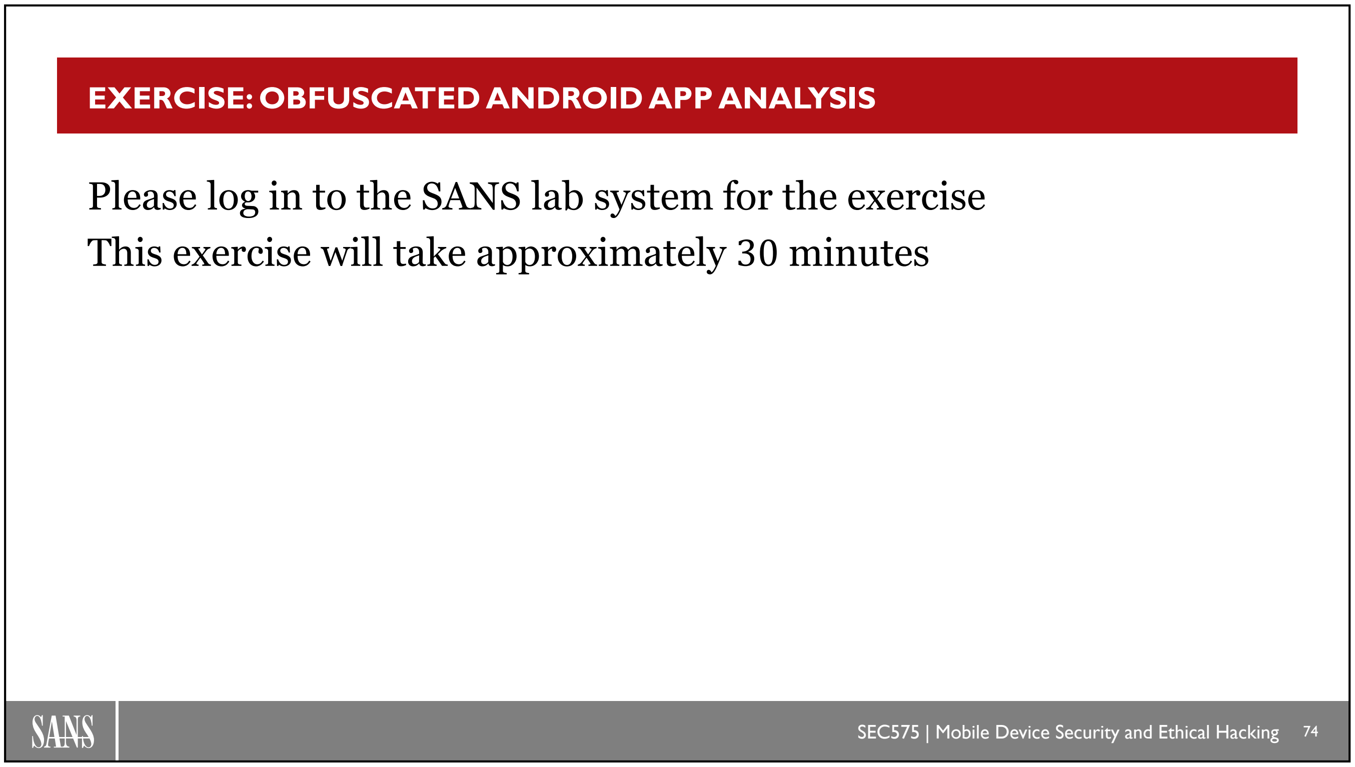
- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 3

Static Application Analysis
Manual Static Analysis
Exercise: Android App Static Analysis
Automating App Analysis
Exercise: MobSF
Obfuscated Apps
<u>Exercise: Obfuscated Android App Analysis</u>
Third-Party App Platforms
Exercise: Xamarin App Analysis
Exercise: PhoneGap App Analysis

SEC575 | Mobile Device Security and Ethical Hacking 73

This page intentionally left blank.



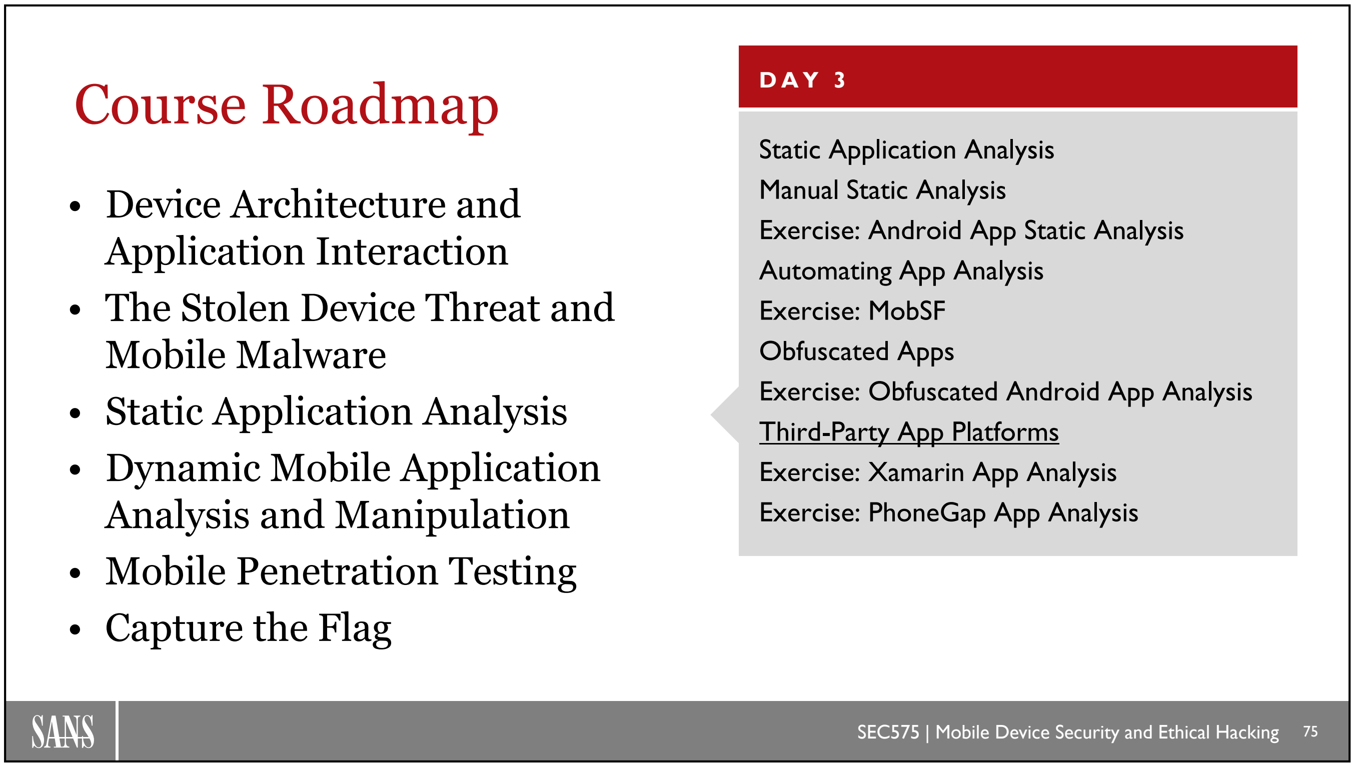
EXERCISE: OBFUSCATED ANDROID APP ANALYSIS

Please log in to the SANS lab system for the exercise
This exercise will take approximately 30 minutes

SANS | SEC575 | Mobile Device Security and Ethical Hacking 74

Exercise: Obfuscated Android App Analysis

Please log in to the SANS lab system for the Obfuscated Android App Analysis exercise. This exercise will take 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

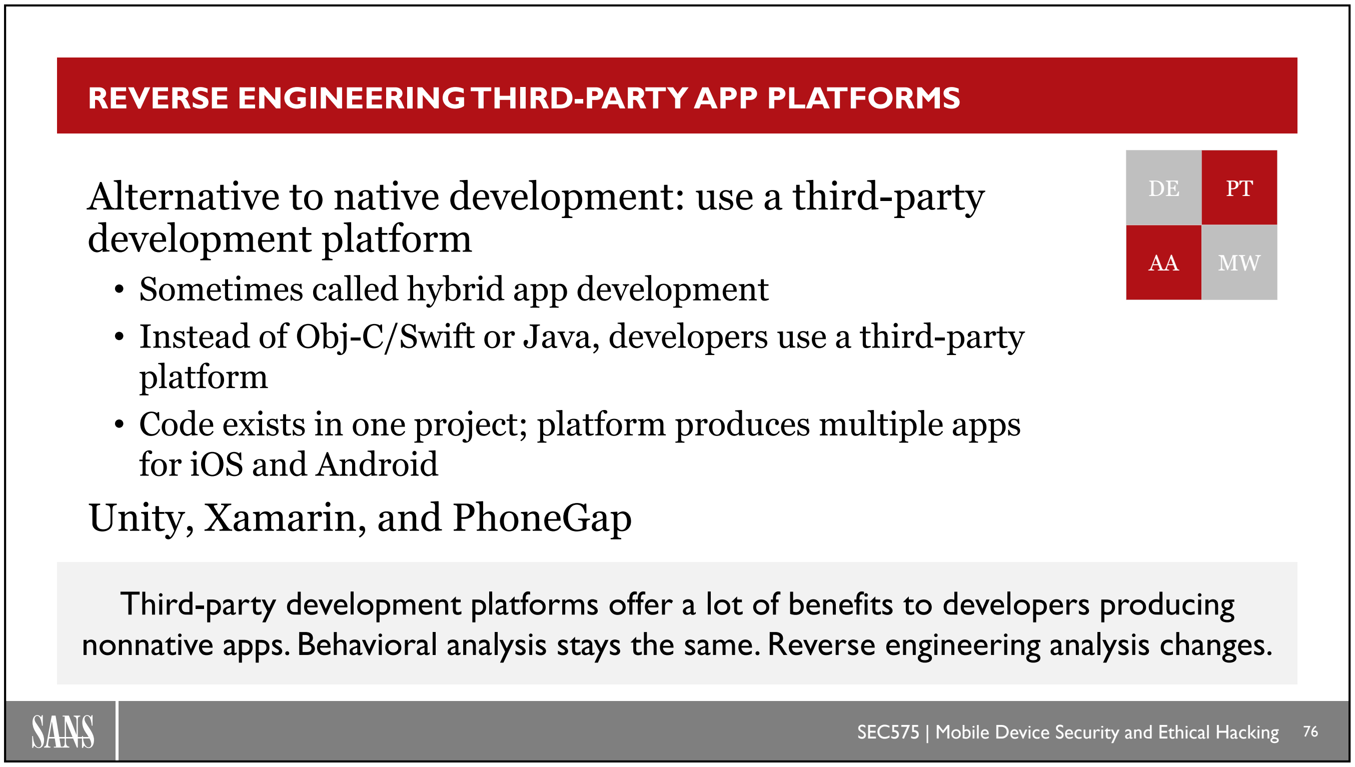
DAY 3

Static Application Analysis
Manual Static Analysis
Exercise: Android App Static Analysis
Automating App Analysis
Exercise: MobSF
Obfuscated Apps
Exercise: Obfuscated Android App Analysis
<u>Third-Party App Platforms</u>
Exercise: Xamarin App Analysis
Exercise: PhoneGap App Analysis

SANS

SEC575 | Mobile Device Security and Ethical Hacking 75

This page intentionally left blank.



REVERSE ENGINEERING THIRD-PARTY APP PLATFORMS

DE	PT
AA	MW

Alternative to native development: use a third-party development platform

- Sometimes called hybrid app development
- Instead of Obj-C/Swift or Java, developers use a third-party platform
- Code exists in one project; platform produces multiple apps for iOS and Android

Unity, Xamarin, and PhoneGap

Third-party development platforms offer a lot of benefits to developers producing nonnative apps. Behavioral analysis stays the same. Reverse engineering analysis changes.

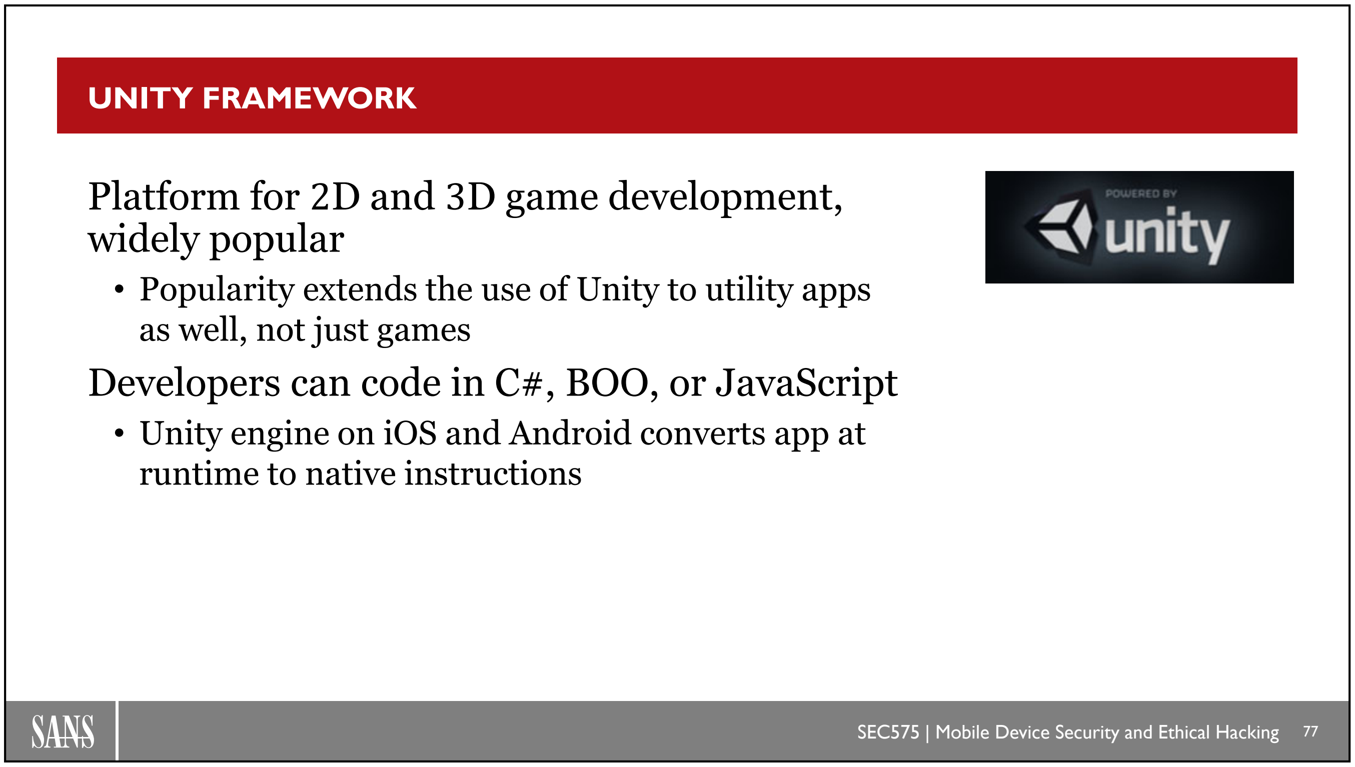
SANS

SEC575 | Mobile Device Security and Ethical Hacking 76

Reverse Engineering Third-Party Development Platforms

In this module, we'll continue our look at reverse engineering mobile applications. Instead of looking at natively built applications, however, we'll focus on third-party development platforms. Sometimes called hybrid app development, these applications aren't developed in Objective-C or Swift for iOS or Java for Android. Instead, they are written using a third-party platform and language, allowing the developer to write the app once and then *build* it for iOS, Android, and other platforms.

Third-party development platforms offer developers a lot of advantages, including the ability to focus their efforts on a single product, instead of writing one app for iOS and another for Android and inevitably having feature disparity between the two products. Several companies offer these platforms for developers, with the three most popular from Unity, Xamarin, and PhoneGap. As a mobile security analyst, the tasks for behavioral analysis of these platforms (network packet captures, HTTP proxies, filesystem data analysis, etc.) don't change. However, we need to use different techniques to reverse engineer these third-party platforms. We'll examine those techniques with practical examples in this module.



UNITY FRAMEWORK

Platform for 2D and 3D game development, widely popular

- Popularity extends the use of Unity to utility apps as well, not just games

Developers can code in C#, BOO, or JavaScript

- Unity engine on iOS and Android converts app at runtime to native instructions

SANS SEC575 | Mobile Device Security and Ethical Hacking 77

Unity Framework

The Unity Framework is a widely popular platform for 2D and 3D game development. More than just a platform for mobile gaming, Unity is widely used for commercial gaming products for Windows, Mac, and even console gaming platforms, including the Microsoft Xbox, Sony PlayStation, and more.

The popularity of the Unity platform has created a large number of talented Unity developers as well, leading to the use of Unity for mobile applications beyond just gaming applications. Developers can write code for the Unity framework in C#, BOO, or JavaScript, allowing developers already familiar with these languages to quickly pick up Unity as a platform as well.

An application written in Unity is quickly ported to Android and iOS, where the developer chooses the desired platform as a build option. The Unity engine produces an APK or IPA file for the target mobile platform, which can then be submitted for inclusion in the Google Play or Apple App Stores.

UNPACKING UNITY

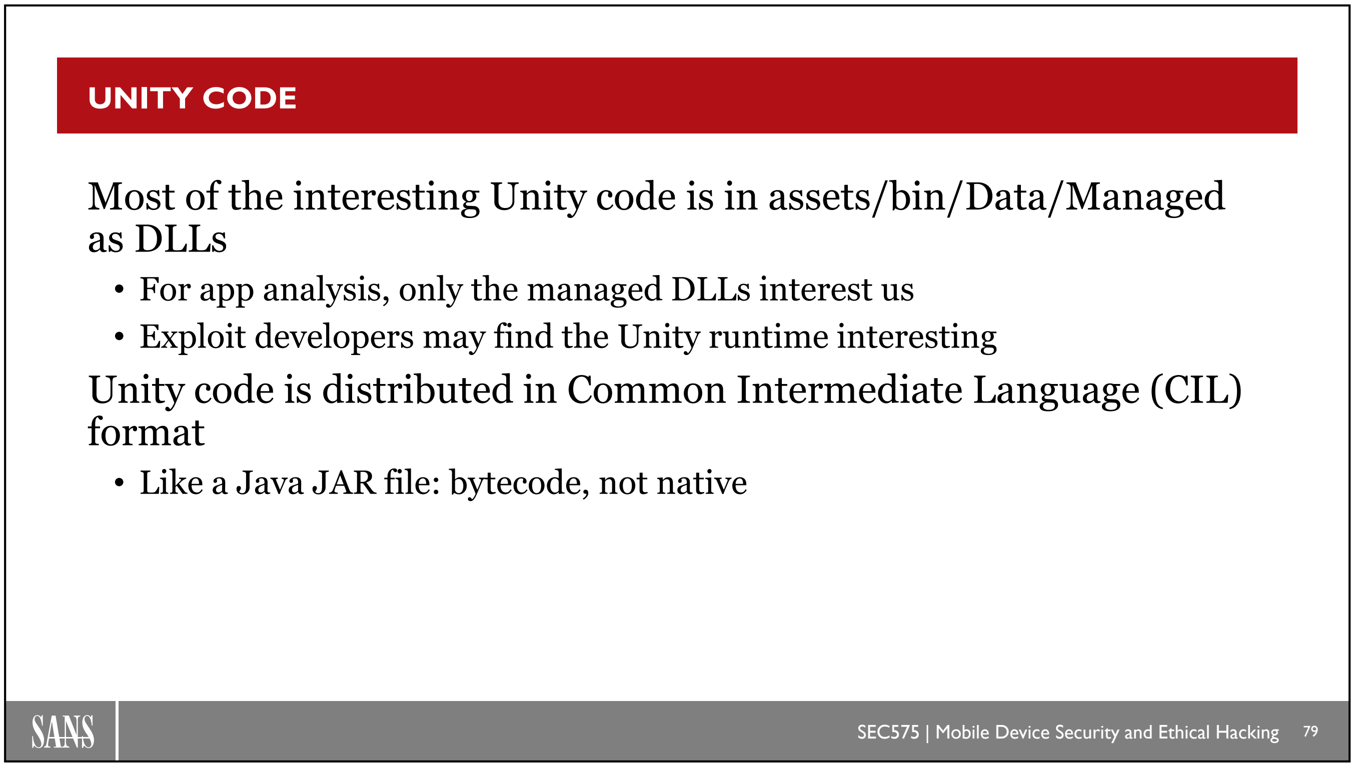
```
$ mkdir pokemongo
$ unzip -d pokemongo/ com.nianticlabs.pokemongo_0.51.0.apk
  inflating: pokemongo/assets/bin/Data/0000000000000000f000000000000000
  inflating: pokemongo/assets/bin/Data/00174a48ac5822646a93874b20a9d676
```

File(s)	Description
AndroidManifest.xml	Minimal Android app declarations
classes.dex	Minimal Android app to launch the Unity Engine (for ARM, or x86)
lib/*	Natively compiled libraries for the Unity Engine
assets/bin/Data/*	Shaders, textures, and other Unity objects
assets/bin/Data/Managed/Assembly*.dll	Unity app binaries

Unpacking Unity

An Android application using the Unity framework is still distributed as an APK file and can be retrieved and extracted in the same way as any other APK file as well. In the example on the top of this page, I've retrieved the Pokémon Go application from apkmonk.com. Unzipping the APK file reveals all the Android app components, including several that we are already familiar with, including the `AndroidManifest.xml` and `classes.dex` files.

Unlike a native app, however, the `classes.dex` file does not include the code that makes up the Pokémon Go app. Instead, `classes.dex` is a small stub of code that launches the Unity engine for the appropriate Android processor architecture. The Unity engine then uses the Unity-compiled binaries stored in the `assets/bin/Data/Managed` directory (typically, `Assembly*.dll` files) to launch the application, using the files in `assets/bin/Data/*` for graphics (shaders, textures) and other Unity application assets.



UNITY CODE

Most of the interesting Unity code is in assets/bin/Data/Managed as DLLs

- For app analysis, only the managed DLLs interest us
- Exploit developers may find the Unity runtime interesting

Unity code is distributed in Common Intermediate Language (CIL) format

- Like a Java JAR file: bytecode, not native

SANS SEC575 | Mobile Device Security and Ethical Hacking 79

Unity Code

By unzipping the Unity-created APK file, we see the managed code files distributed as DLLs in `assets/bin/data/Managed`. The Unity runtime itself is distributed with each Unity binary in the `lib` directory, but that is of little interest if our focus is on the app itself (exploit developers may find the Unity runtime interesting for the purpose of bug discovery).

The Unity DLLs distributed in the APK file are not natively compiled. Instead, the DLLs are distributed in the Common Intermediate Language (CIL) format. CIL is a standard mechanism to distribute CPU-agnostic code, very similar to a Java JAR file. Like a Java JAR file, we can use reverse engineering tools to reconstruct the original source code used to build the CIL files.

ANDROID OUIJA APP

Virtual Ouija board published by Redwerk Board

User asks questions, Ouija board "communicates with spirits"

Developed in Unity

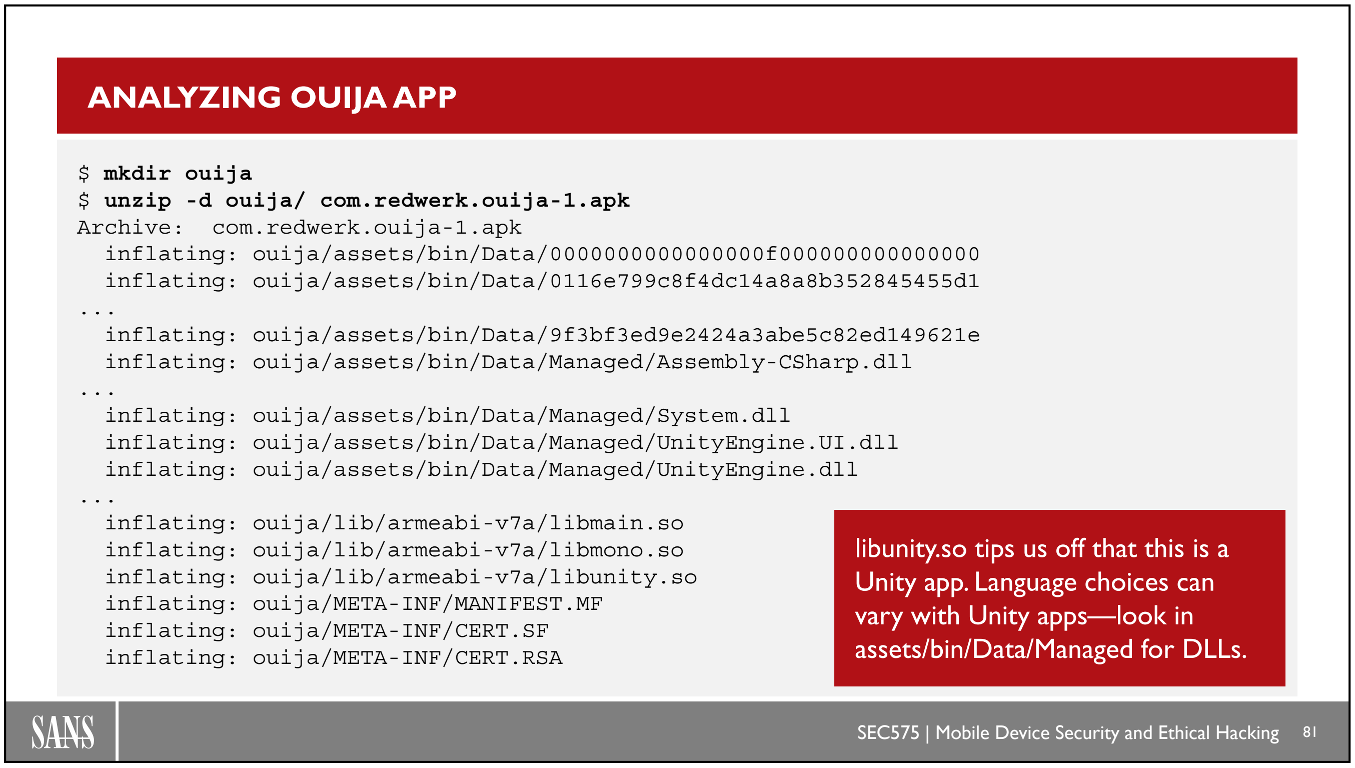
What does it really do?

"Converse with ghosts and spirits right from your phone"

Android Ouija App

As we look at the tools and techniques for Unity application reverse engineering, we'll use the Virtual Ouija board as our target.

When you launch the Ouija app, you can ask the *ghost* questions. The ghost responds by moving the planchette across the board, spelling out the ghost responses. As part of a larger study of Third-Party mobile application frameworks, I wanted to find out what the app really did.



ANALYZING OUIJA APP

```
$ mkdir ouija
$ unzip -d ouija/ com.redwerk.ouija-1.apk
Archive:  com.redwerk.ouija-1.apk
  inflating: ouija/assets/bin/Data/0000000000000000f000000000000000
  inflating: ouija/assets/bin/Data/0116e799c8f4dc14a8a8b352845455d1
...
  inflating: ouija/assets/bin/Data/9f3bf3ed9e2424a3abe5c82ed149621e
  inflating: ouija/assets/bin/Data/Managed/Assembly-CSharp.dll
...
  inflating: ouija/assets/bin/Data/Managed/System.dll
  inflating: ouija/assets/bin/Data/Managed/UnityEngine.UI.dll
  inflating: ouija/assets/bin/Data/Managed/UnityEngine.dll
...
  inflating: ouija/lib/armeabi-v7a/libmain.so
  inflating: ouija/lib/armeabi-v7a/libmono.so
  inflating: ouija/lib/armeabi-v7a/libunity.so
  inflating: ouija/META-INF/MANIFEST.MF
  inflating: ouija/META-INF/CERT.SF
  inflating: ouija/META-INF/CERT.RSA
```

libunity.so tips us off that this is a Unity app. Language choices can vary with Unity apps—look in assets/bin/Data/Managed for DLLs.

Analyzing Ouija App

I purchased the Ouija app from the Google Play store for $4. After installing it on a rooted Android Nexus 7, I downloaded the APK file using `adb pull` to my system for analysis. The first step was to unzip the APK, the partial output of which is shown on this page.

From this output, we see familiar Android APK files, but we also see several files in the `assets` and `lib` directories. The `lib` directory includes the Unity runtime executables for different Android architectures (ARMv7 is shown on this page, `armeabi-v7a`). The runtime executables are going to be the same for all Unity builds (for a specific version of the Unity framework). We are more interested in the files in the `assets/bin/Data/Managed` directory.

REVERSE ENGINEERING OUIJA

Ouija has Assembly-CSharp.dll

We can use a C# decompiler to analyze Ouija's code

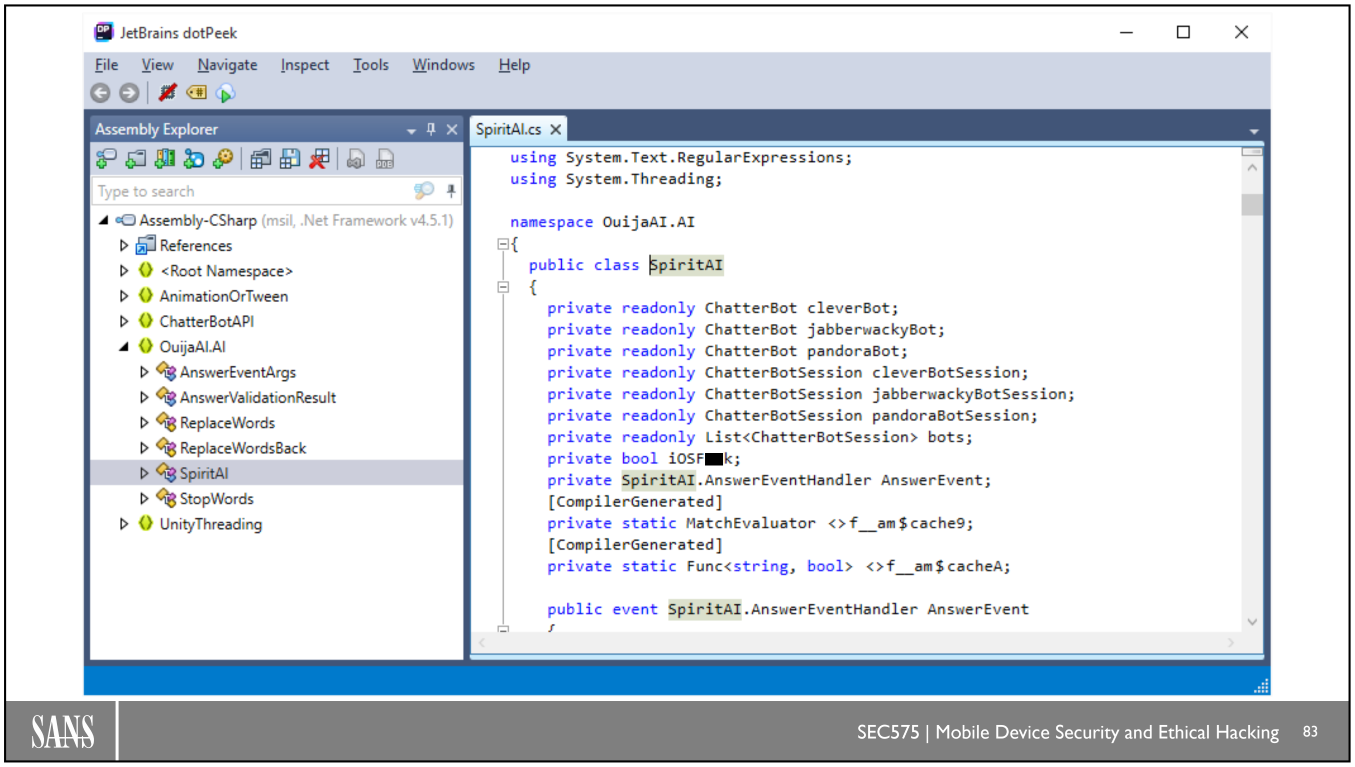
JetBrains dotPeek decompiles CIL files, revealing reconstructed C# source

- Free, closed source tool
- "Jadx for C#"

Reverse Engineering Ouija

In the Managed directory for the Ouija app, we see the file Assembly-CSharp.dll. The filename reveals that Redwerk wrote the Ouija app in C#. Since the C# DLL is distributed in CIL format, we can use a .NET decompiler tool such as the JetBrains dotPeek tool (https://www.jetbrains.com/decompiler/).

The dotPeek decompiler is analogous to what Jadx does for native Android applications, except instead of reconstructing Dalvik bytecode into Java source, it produces reconstructed C# from CSharp CIL files.



```
using System.Text.RegularExpressions;
using System.Threading;

namespace OuijaAI.AI
{
  public class SpiritAI
  {
    private readonly ChatterBot cleverBot;
    private readonly ChatterBot jabberwackyBot;
    private readonly ChatterBot pandoraBot;
    private readonly ChatterBotSession cleverBotSession;
    private readonly ChatterBotSession jabberwackyBotSession;
    private readonly ChatterBotSession pandoraBotSession;
    private readonly List<ChatterBotSession> bots;
    private bool iOSF[redacted]k;
    private SpiritAI.AnswerEventHandler AnswerEvent;
    [CompilerGenerated]
    private static MatchEvaluator <>f__am$cache9;
    [CompilerGenerated]
    private static Func<string, bool> <>f__am$cacheA;

    public event SpiritAI.AnswerEventHandler AnswerEvent
```

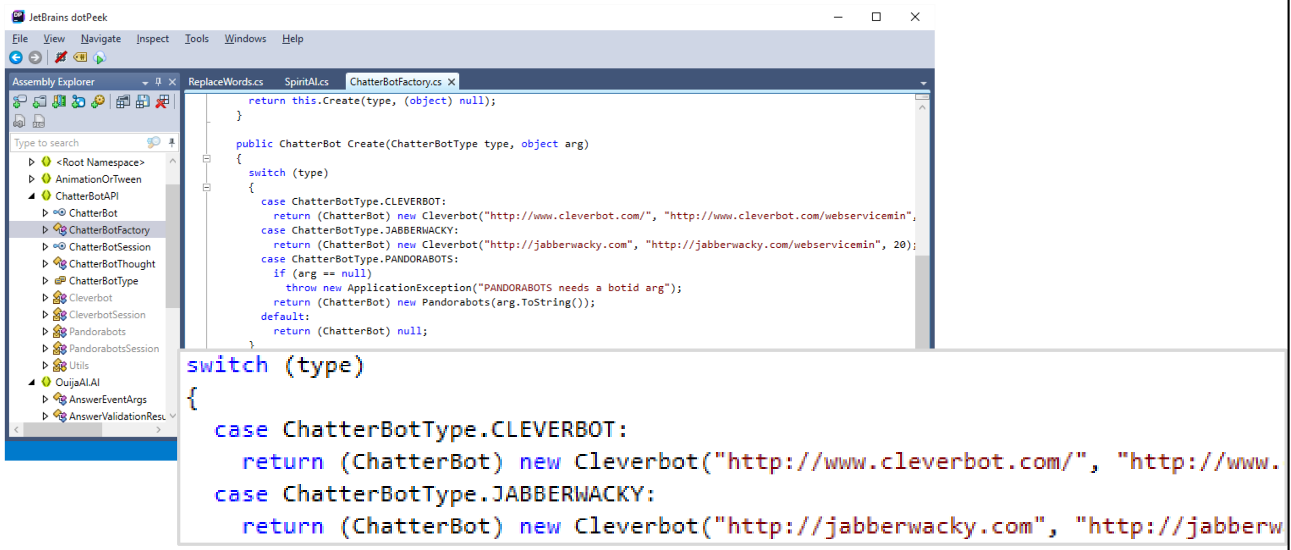
JetBrains dotPeek

The dotPeek tool only runs on Windows. Opening the `Assembly-CSharp.dll` file reveals the class names, methods, and variables, as shown on this page. Like our experience with Jadx, after opening the file we can start to evaluate the reconstructed code to gain a better understanding of the application.

One major difference with this technique compared to reverse engineering native Android applications is that dotPeek returns reconstructed C# code, not Java code. This creates an additional burden on the analyst, who may not be familiar with the C# language. Fortunately, several of the fundamental elements of C# are not terribly dissimilar to Java, and it is a fairly straightforward language to understand quickly.

The `SpiritAI` class shown on this page reveals that several `ChatterBot` variables are declared: `cleverBot`, `jabberwackyBot`, and `pandoraBot`, with companion `ChatterBotSession` variables. I wasn't sure what the `ChatterBot` variable was, so I started to back up a few steps and investigate the `ChatterBotAPI` class where that type is defined.

OUIJA: CHATTERBOTFACTORY



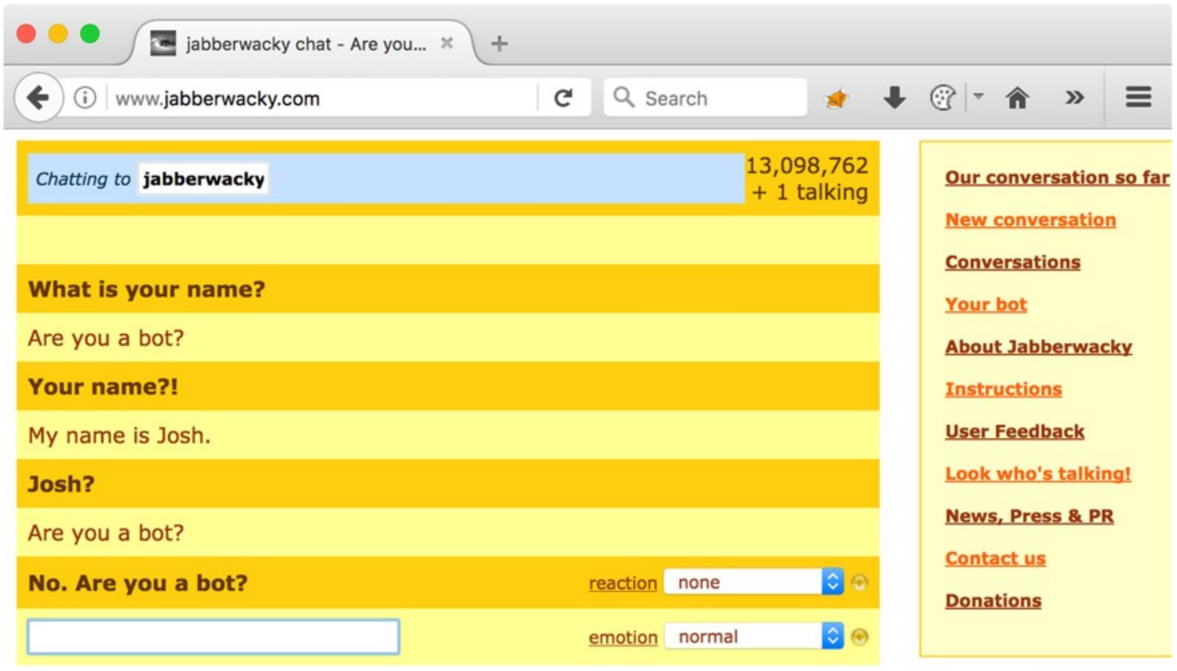
```
      return this.Create(type, (object) null);
    }

    public ChatterBot Create(ChatterBotType type, object arg)
    {
      switch (type)
      {
        case ChatterBotType.CLEVERBOT:
          return (ChatterBot) new Cleverbot("http://www.cleverbot.com/", "http://www.cleverbot.com/webservicemin",
        case ChatterBotType.JABBERWACKY:
          return (ChatterBot) new Cleverbot("http://jabberwacky.com", "http://jabberwacky.com/webservicemin", 20);
        case ChatterBotType.PANDORABOTS:
          if (arg == null)
            throw new ApplicationException("PANDORABOTS needs a botid arg");
          return (ChatterBot) new Pandorabots(arg.ToString());
        default:
          return (ChatterBot) null;
      }
```

```
switch (type)
{
  case ChatterBotType.CLEVERBOT:
    return (ChatterBot) new Cleverbot("http://www.cleverbot.com/", "http://www.
  case ChatterBotType.JABBERWACKY:
    return (ChatterBot) new Cleverbot("http://jabberwacky.com", "http://jabberw
```

Ouija: ChatterBotFactory

Navigating to the `ChatterBotAPI` class, I started to investigate the `ChatterBotFactory` object. In C#, the `Create()` method is called automatically when an object is instantiated (declared for the first time). Looking at the C# code in the `Create()` method reveals several URLs pointing to sites similarly named to the class object variables, including www.cleverbot.com and jabberwacky.com.


jabberwacky chat - Are you...

www.jabberwacky.com

Chatting to jabberwacky 13,098,762 + 1 talking

What is your name?

Are you a bot?

Your name?!

My name is Josh.

Josh?

Are you a bot?

No. Are you a bot?

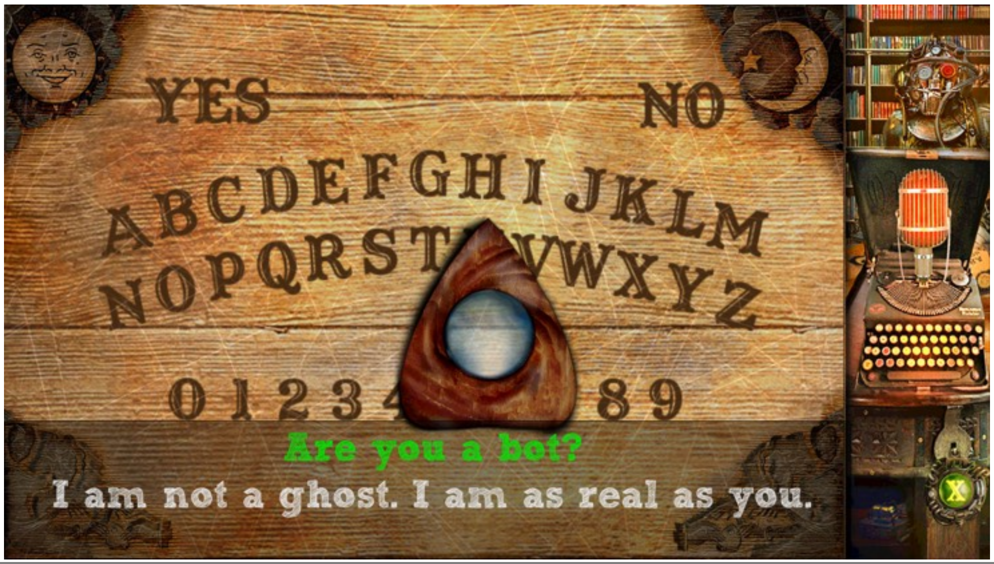
reaction: none

emotion: normal

Jabberwacky

Navigating to these websites reveals that they are free chat bot services. Note in the output shown here how I asked the Jabberwacky service if it was a bot, to which it answered "No".

ARE YOU A BOT?



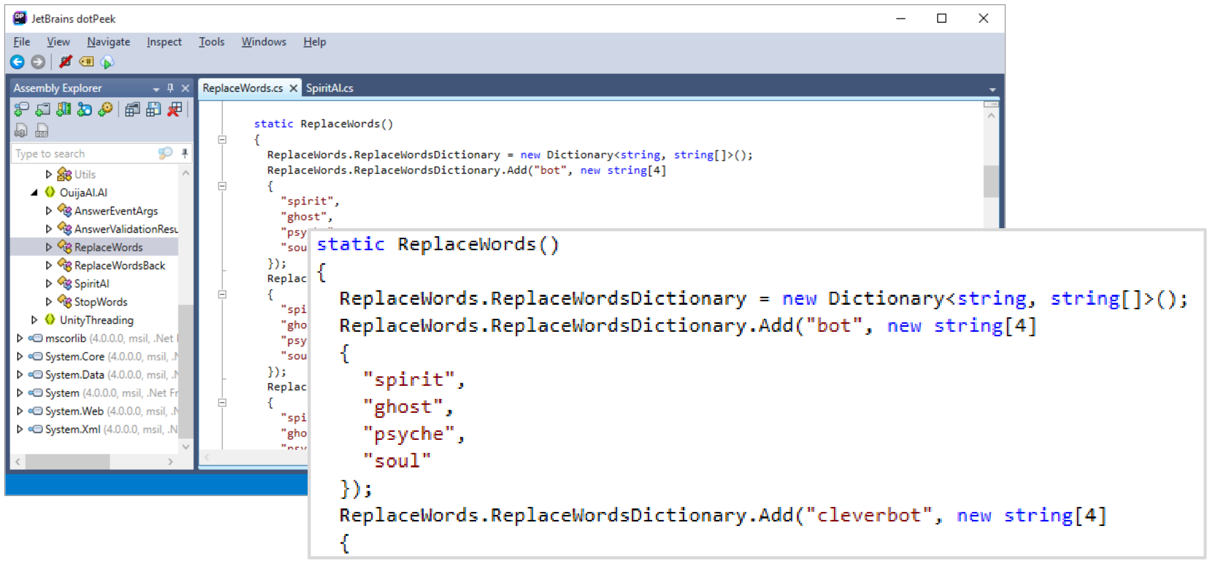
Are you a bot?

I am not a ghost. I am as real as you.

Are You a Bot

If you ask the same question to the Ouija app, however, you get a modified response. My question used the word "bot“, but the response used the word "ghost".

OUIJA: REPLACEWORDS()



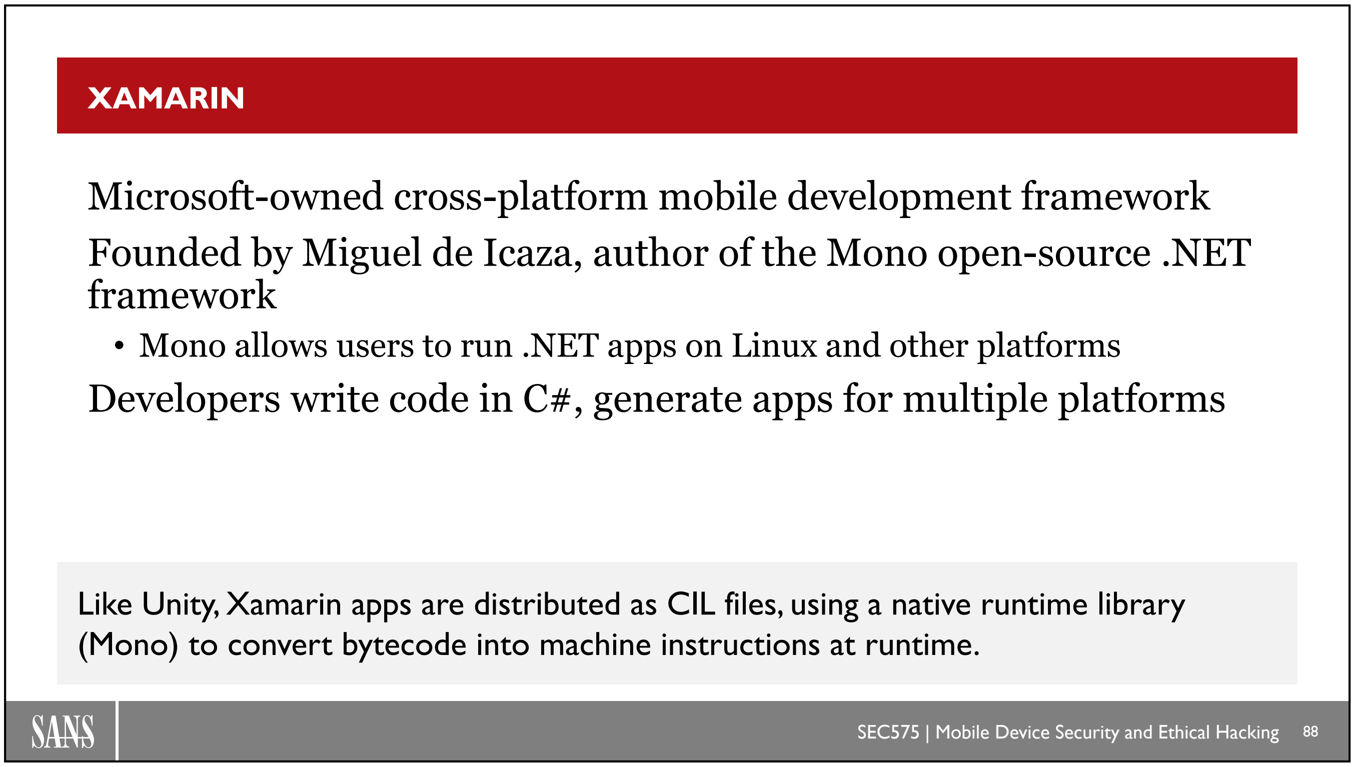
```
static ReplaceWords()
{
  ReplaceWords.ReplaceWordsDictionary = new Dictionary<string, string[]>();
  ReplaceWords.ReplaceWordsDictionary.Add("bot", new string[4]
  {
    "spirit",
    "ghost",
    "psyche",
    "soul"
  });
  ReplaceWords.ReplaceWordsDictionary.Add("cleverbot", new string[4]
  {
```

SANS SEC575 | Mobile Device Security and Ethical Hacking 87

Ouija: ReplaceWords()

Returning to dotPeek, more analysis of the OuijaAI class reveals code that automatically replaces instances of the word "bot" with a random word selected from "spirit", "ghost", "psyche", or "soul". The developers appear to have taken some effort to obscure the use of the chatbot services in their app, though perhaps a lackluster attempt.

For most Unity applications, the reverse engineering process proceeds similarly to the Ouija app. For more complex applications, there will be more code to review, but the analysis steps remain the same.



XAMARIN

Microsoft-owned cross-platform mobile development framework

Founded by Miguel de Icaza, author of the Mono open-source .NET framework

- Mono allows users to run .NET apps on Linux and other platforms

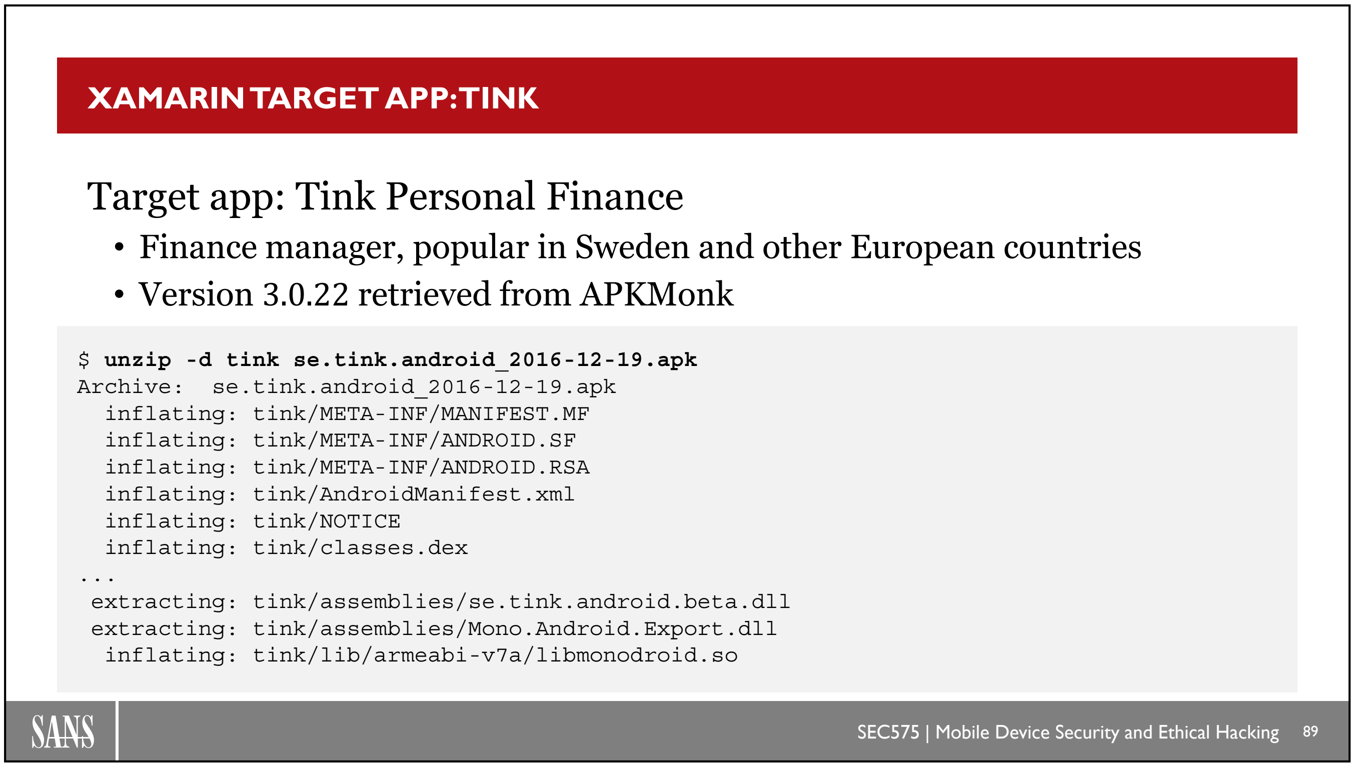
Developers write code in C#, generate apps for multiple platforms

Like Unity, Xamarin apps are distributed as CIL files, using a native runtime library (Mono) to convert bytecode into machine instructions at runtime.

Xamarin

Xamarin is a Microsoft-owned cross-platform mobile development framework founded by Miguel de Icaza. De Icaza is the author of the Mono open-source framework that allows applications to run .NET apps on Linux and other platforms. Using Xamarin, developers create their apps using C# and then generate runtime versions of their apps for multiple platforms.

Like Unity, the Xamarin apps are distributed as CIL files with a DLL extension. The CIL files are interpreted at runtime and converted to native machine instructions using the Mono framework.



XAMARIN TARGET APP: TINK

Target app: Tink Personal Finance

- Finance manager, popular in Sweden and other European countries
- Version 3.0.22 retrieved from APKMonk

```
$ unzip -d tink se.tink.android_2016-12-19.apk
Archive:  se.tink.android_2016-12-19.apk
  inflating: tink/META-INF/MANIFEST.MF
  inflating: tink/META-INF/ANDROID.SF
  inflating: tink/META-INF/ANDROID.RSA
  inflating: tink/AndroidManifest.xml
  inflating: tink/NOTICE
  inflating: tink/classes.dex
...
 extracting: tink/assemblies/se.tink.android.beta.dll
 extracting: tink/assemblies/Mono.Android.Export.dll
  inflating: tink/lib/armeabi-v7a/libmonodroid.so
```

SANS

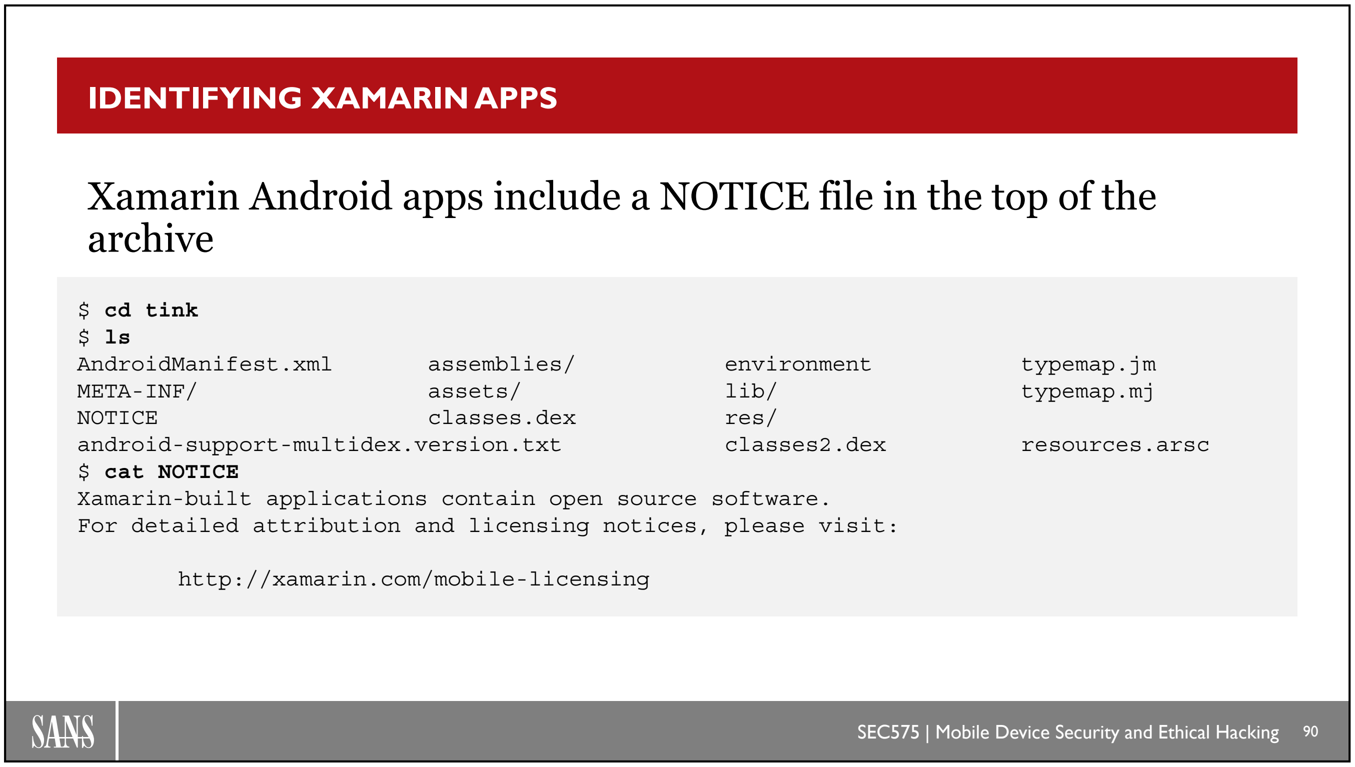
SEC575 | Mobile Device Security and Ethical Hacking 89

Xamarin Target App: Tink

For our investigation into reverse engineering Xamarin apps, we'll target the Tink Personal Finance app. Tink is a finance management helper application developed in Sweden and popular in other European countries as well. For the purposes of this analysis, I retrieved the Tink app from APKMonk.com, version 3.0.22.

Unzipping the Tink APK file shows several common Android files, similar to what we saw with Unity applications. Instead of distributing DLL files in the `assets` directory, Xamarin apps distribute the DLL files in the `assemblies` directory, with the Mono runtime libraries distributed in the `lib` directory.

Note that some of the output of the unzip command shown on this page has been trimmed for space.



IDENTIFYING XAMARIN APPS

Xamarin Android apps include a NOTICE file in the top of the archive

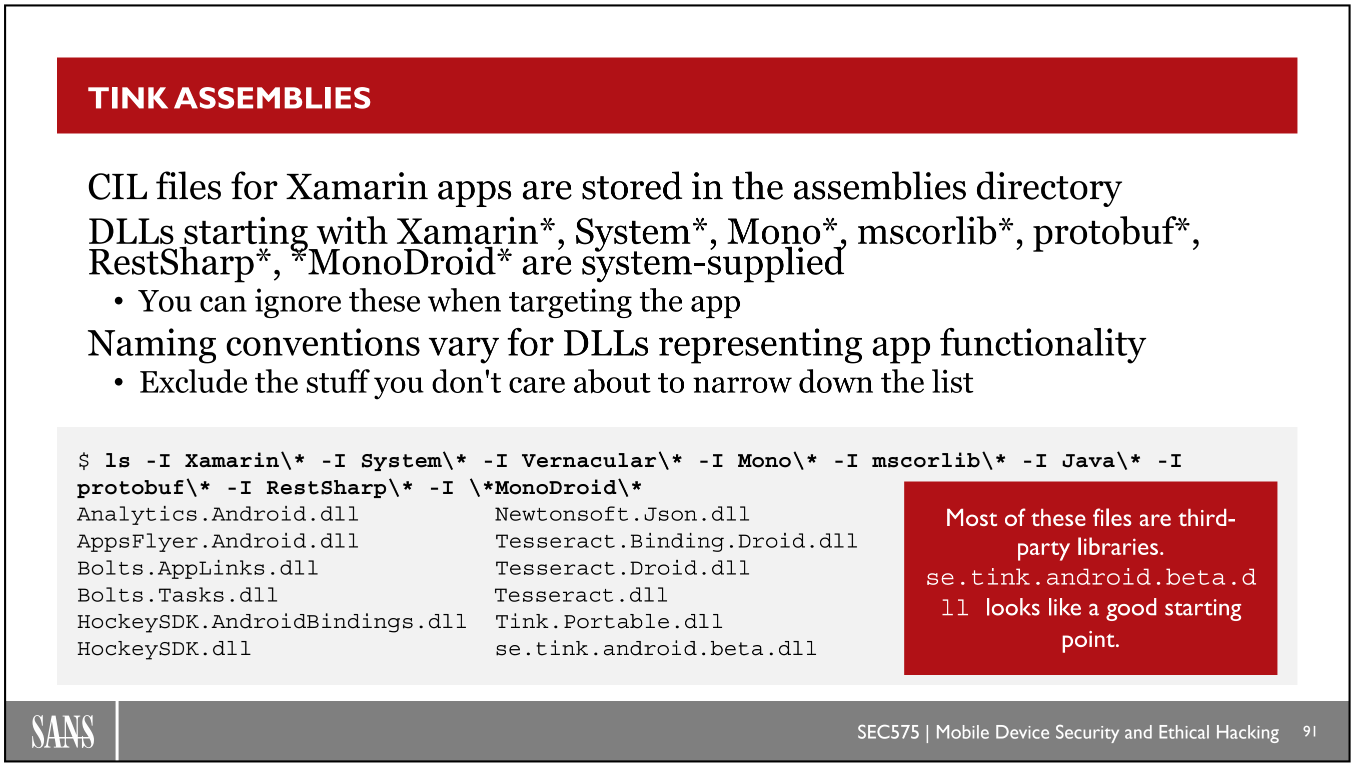
```
$ cd tink
$ ls
AndroidManifest.xml       assemblies/          environment          typemap.jm
META-INF/                 assets/              lib/                 typemap.mj
NOTICE                    classes.dex          res/
android-support-multidex.version.txt           classes2.dex         resources.arsc
$ cat NOTICE
Xamarin-built applications contain open source software.
For detailed attribution and licensing notices, please visit:

       http://xamarin.com/mobile-licensing
```

SEC575 | Mobile Device Security and Ethical Hacking 90

Identifying Xamarin Apps

As an open-source framework, Mono uses other open-source components. As a result, Xamarin needs to declare that it uses open-source software with a link for attribution of copyright owners. This makes it easy for us to quickly identify apps built with Xamarin by examining the `NOTICE` file in the top-level directory of the APK data, as shown on this page.



TINK ASSEMBLIES

CIL files for Xamarin apps are stored in the assemblies directory

DLLs starting with Xamarin*, System*, Mono*, mscorlib*, protobuf*, RestSharp*, *MonoDroid* are system-supplied

- You can ignore these when targeting the app

Naming conventions vary for DLLs representing app functionality

- Exclude the stuff you don't care about to narrow down the list

```
$ ls -I Xamarin\* -I System\* -I Vernacular\* -I Mono\* -I mscorlib\* -I Java\* -I
protobuf\* -I RestSharp\* -I \*MonoDroid\*
Analytics.Android.dll          Newtonsoft.Json.dll
AppsFlyer.Android.dll          Tesseract.Binding.Droid.dll
Bolts.AppLinks.dll             Tesseract.Droid.dll
Bolts.Tasks.dll                Tesseract.dll
HockeySDK.AndroidBindings.dll  Tink.Portable.dll
HockeySDK.dll                  se.tink.android.beta.dll
```

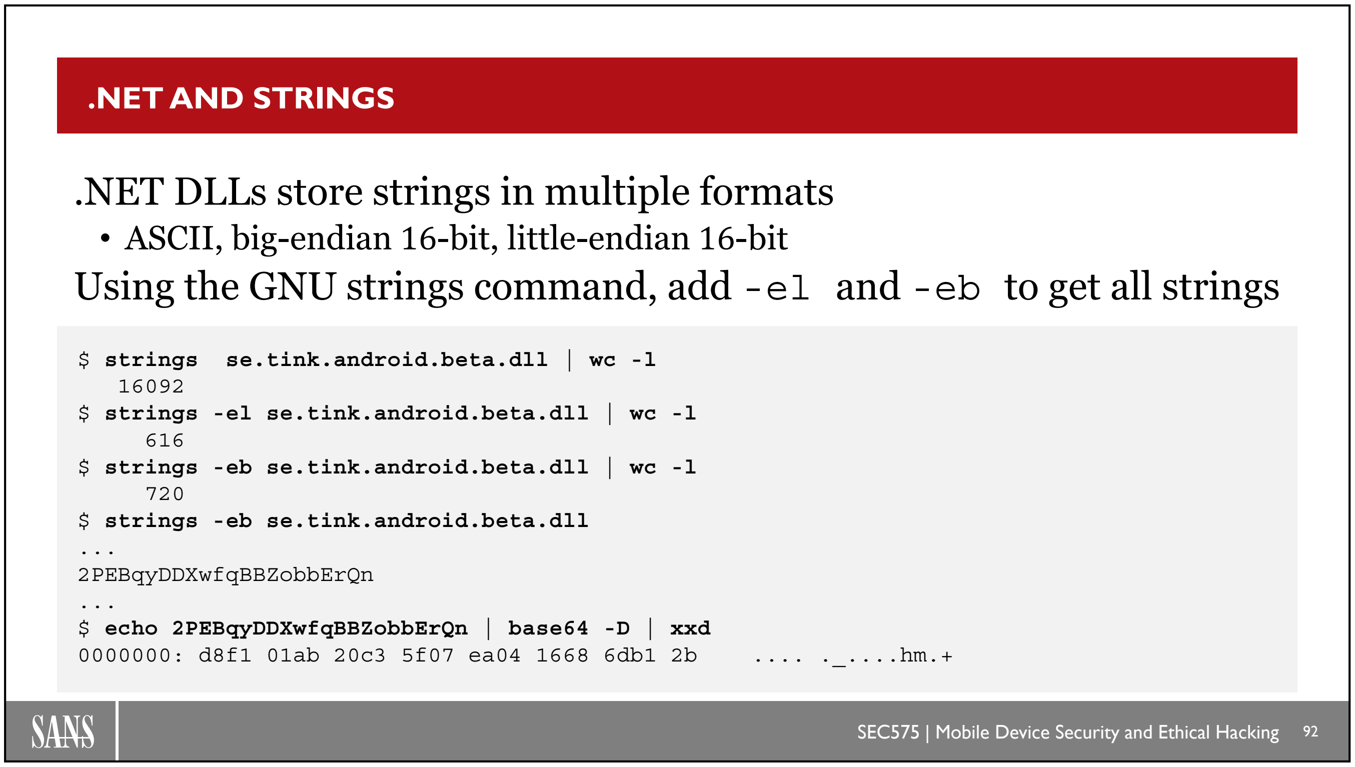
Most of these files are third-party libraries. `se.tink.android.beta.dll` looks like a good starting point.

Tink Assemblies

After unzipping the APK file and confirming that the developers built Tink with the Xamarin framework, I started to investigate the DLL files in the assemblies directory. In this directory, there were 40 different DLL files—most of which can be ignored, since they are not specific to the Tink application.

For Xamarin apps, DLL files starting with `Xamarin*`, `System*`, `Vernacular*`, `Mono*`, `mscorlib*`, `Java*`, `protobuf*`, `RestSharp*`, and `*MonoDroid*` can be ignored since they are part of the Xamarin framework and not the Tink app itself. This left 12 remaining DLL files, most of which appear to be third-party libraries, including the Bolts Framework developed by Facebook (https://github.com/BoltsFramework/Bolts-Android) and the Tesseract Framework for optical character recognition (OCR, https://github.com/rmtheis/tess-two). While it's valuable to recognize the libraries that make up an app's functionality, our primary analysis will be on the Tink application itself.

Using the short list of DLLs, the `se.tink.android.beta.dll` applications looks like a good starting point for application analysis. In practice, there is a little guessing involved in choosing the right entry point for the Xamarin application, but using these techniques of excluding system DLLs and identifying DLLs that represent third-party libraries helps to reduce the list of applications to choose from.



.NET AND STRINGS

.NET DLLs store strings in multiple formats

- ASCII, big-endian 16-bit, little-endian 16-bit

Using the GNU strings command, add `-el` and `-eb` to get all strings

```
$ strings  se.tink.android.beta.dll | wc -l
   16092
$ strings -el se.tink.android.beta.dll | wc -l
     616
$ strings -eb se.tink.android.beta.dll | wc -l
     720
$ strings -eb se.tink.android.beta.dll
...
2PEBqyDDXwfqBBZobbErQn
...
$ echo 2PEBqyDDXwfqBBZobbErQn | base64 -D | xxd
0000000: d8f1 01ab 20c3 5f07 ea04 1668 6db1 2b    .... ._....hm.+
```

SANS SEC575 | Mobile Device Security and Ethical Hacking 92

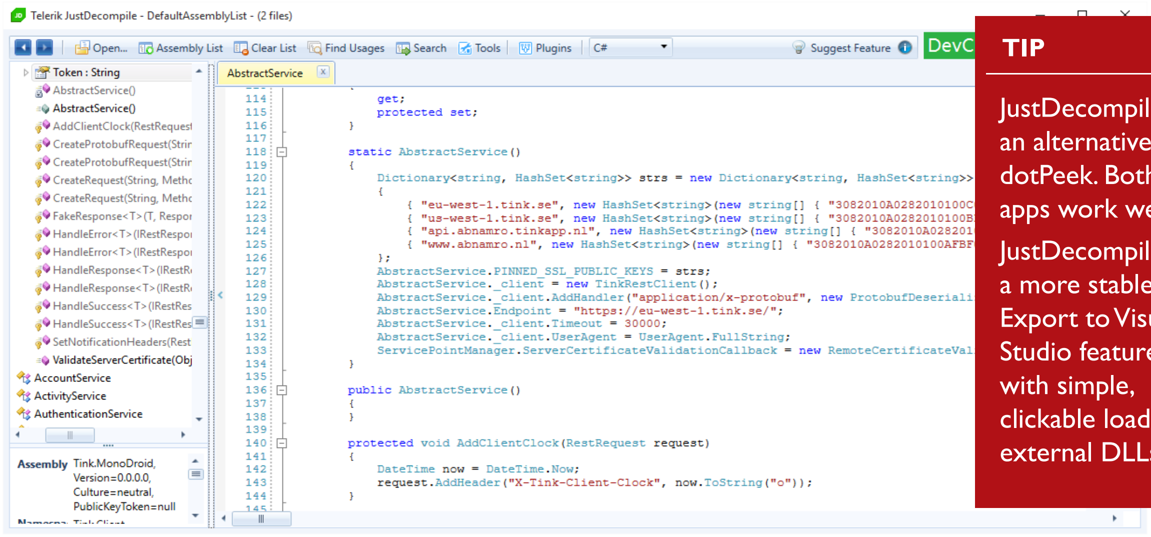
.NET and Strings

Don't forget about the useful strings command when reverse engineering applications. For Xamarin (and Unity) applications, we need to remember that the `strings` command only identifies ASCII strings by default. Since .NET uses both big-endian and little-endian strings as well, we need to run the `strings` command several times, specifying the different encoding options we're interested in seeing the results of.

In the example on this page, the first `strings` command reveals the greatest number of strings, but adding `-el` to identify little-endian Unicode strings and `-eb` to identify big-endian Unicode strings also produces a lot of output. For Tink, the big-endian Unicode encoding option revealed a Base64 encoded string that was not present in the default `strings` ASCII output.

Decoding the Base64 value reveals what could have been a randomly selected value. Next, we want to figure out where in the applications this string is used to identify if the value is sensitive or will otherwise expose the application in an interesting way.

TELERIK JUSTDECOMPILE


Telerik JustDecompile - DefaultAssemblyList - (2 files)
Open...
Assembly List
Clear List
Find Usages
Search
Tools
Plugins
C#
Suggest Feature
DevC
Token : String
AbstractService()
AbstractService()
AddClientClock(RestRequest
CreateProtobufRequest(Strir
CreateProtobufRequest(Strir
CreateRequest(String, Methc
CreateRequest(String, Methc
FakeResponse<T>(T, Respor
HandleError<T>(IRestRespo
HandleError<T>(IRestRespo
HandleResponse<T>(IRestR
HandleResponse<T>(IRestR
HandleSuccess<T>(IRestRes
HandleSuccess<T>(IRestRes
SetNotificationHeaders(Rest
ValidateServerCertificate(Obj
AccountService
ActivityService
AuthenticationService
AbstractService

Assembly Tink.MonoDroid, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

```
            get;
            protected set;
        }

        static AbstractService()
        {
            Dictionary<string, HashSet<string>> strs = new Dictionary<string, HashSet<string>>
            {
                { "eu-west-1.tink.se", new HashSet<string>(new string[] { "3082010A0282010100C
                { "us-west-1.tink.se", new HashSet<string>(new string[] { "3082010A0282010100B
                { "api.abnamro.tinkapp.nl", new HashSet<string>(new string[] { "3082010A028201
                { "www.abnamro.nl", new HashSet<string>(new string[] { "3082010A0282010100AFBF
            };
            AbstractService.PINNED_SSL_PUBLIC_KEYS = strs;
            AbstractService._client = new TinkRestClient();
            AbstractService._client.AddHandler("application/x-protobuf", new ProtobufDeseriali
            AbstractService.Endpoint = "https://eu-west-1.tink.se/";
            AbstractService._client.Timeout = 30000;
            AbstractService._client.UserAgent = UserAgent.FullString;
            ServicePointManager.ServerCertificateValidationCallback = new RemoteCertificateVal
        }

        public AbstractService()
        {
        }

        protected void AddClientClock(RestRequest request)
        {
            DateTime now = DateTime.Now;
            request.AddHeader("X-Tink-Client-Clock", now.ToString("o"));
        }
```

TIP

JustDecompile is an alternative to dotPeek. Both apps work well.

JustDecompile has a more stable Export to Visual Studio feature, with simple, clickable loading of external DLLs.

Telerik JustDecompile

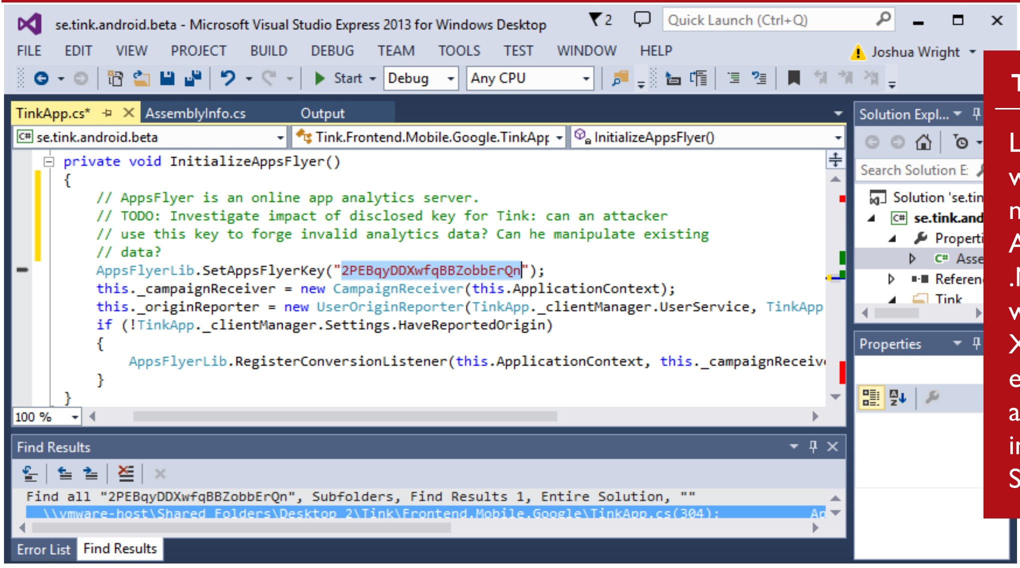
In the material on reverse engineering Unity applications, we examined the dotPeek application, which reconstructs C# code from .NET CIL (DLL files). dotPeek would also work for reverse engineering Xamarin applications, but I wanted to demonstrate another .NET decompiler option, namely the Telerik JustDecompile tool.

JustDecompile is a free tool and part of a larger collection of .NET analysis tools called DevCraft (which is a commercial tool, and while valuable for .NET developers, probably not something you need just for Xamarin or Unity reverse engineering). JustDecompile works similarly to dotPeek where you open a DLL and are presented with reconstructed C# code, as shown on this page. However, JustDecompile also includes dynamic loading of external libraries (click on a method in the reconstructed code; if it is not already part of the reconstructed code in the chosen library, JustDecompile will ask you if it can open the DLL and reconstruct the library code as well) and a more stable *Export to Visual Studio* feature.

From the open DLL, click the *Tools* button in the toolbar and choose *Export to Visual Studio*. JustDecompile will create a new Visual Studio product populated with the reconstructed C# code. Like we saw earlier with the Jadx code and Android Studio, exporting the DLL file (from Xamarin or Unity apps) and opening in Visual Studio allows us to annotate the code, making it easier for us to read and understand the code.

JustDecompile is available for free at http://www.telerik.com/products/decompiler.aspx.

IMPORT INTO VISUAL STUDIO, ANNOTATE



```
private void InitializeAppsFlyer()
{
    // AppsFlyer is an online app analytics server.
    // TODO: Investigate impact of disclosed key for Tink: can an attacker
    // use this key to forge invalid analytics data? Can he manipulate existing
    // data?
    AppsFlyerLib.SetAppsFlyerKey("2PEBqyDDXwfqBBZobbErQn");
    this._campaignReceiver = new CampaignReceiver(this.ApplicationContext);
    this._originReporter = new UserOriginReporter(TinkApp._clientManager.UserService, TinkApp
    if (!TinkApp._clientManager.Settings.HaveReportedOrigin)
    {
        AppsFlyerLib.RegisterConversionListener(this.ApplicationContext, this._campaignReceiv
    }
}
```

```
Find all "2PEBqyDDXwfqBBZobbErQn", Subfolders, Find Results 1, Entire Solution, ""
  \\vmware-host\Shared Folders\Desktop 2\Tink\Frontend.Mobile.Google\TinkApp.cs(304):
```

TIP

Like we've seen with Android native apps and Android Studio, .NET apps written with Unity or Xamarin are also easily annotated and deobfuscated in Microsoft Visual Studio.

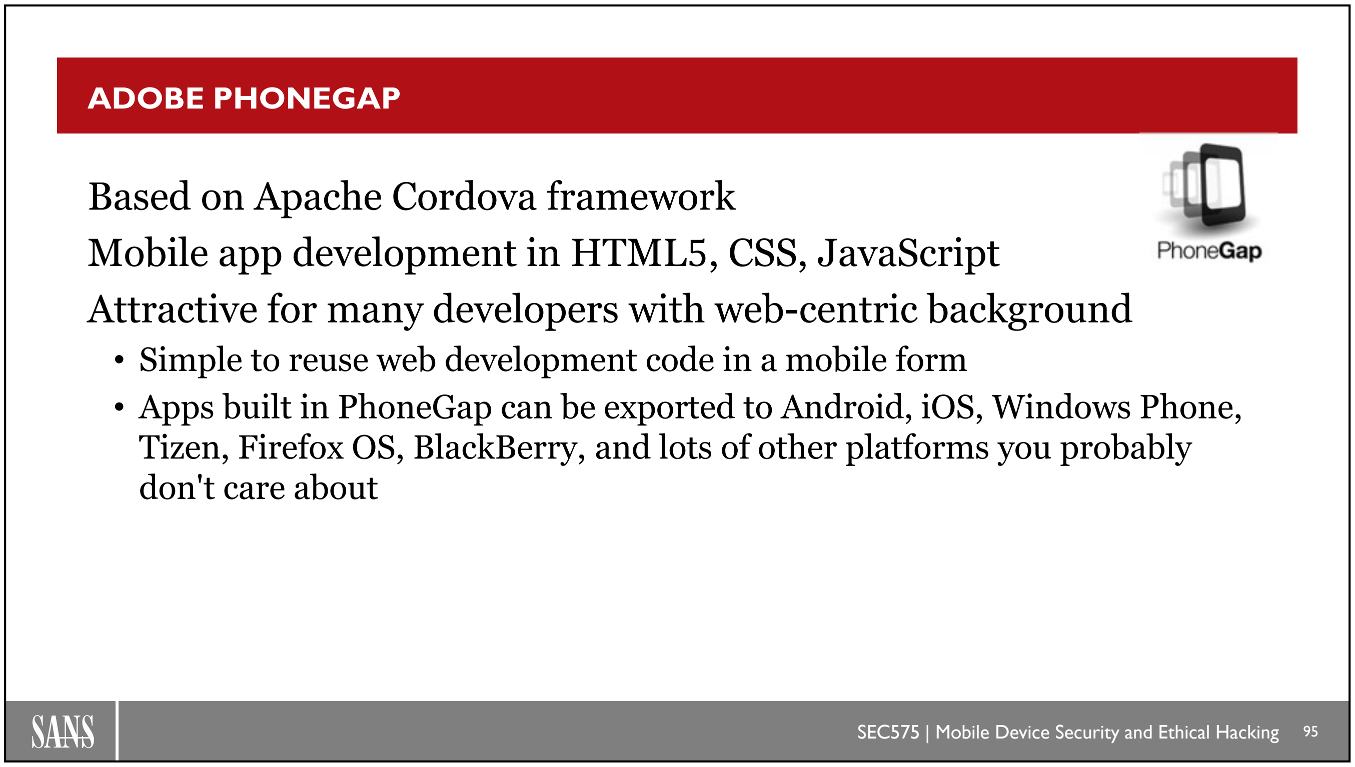
Import Into Visual Studio, Annotate

Opening the JustDecompile-generated Visual Studio project using Visual Studio Express (available for free at https://www.visualstudio.com/vs/visual-studio-express/) and searching for the Base64 encoded string reveals that it is present in the InitializeAppsFlyer() method, as shown on this page.

AppsFlyer is a mobile device analytics platform used for tracking application use information (where app users are geographically, how frequently they use the application, etc.) The key disclosed in the reconstructed code is specific to the Tink application and uniquely identifies all of the traffic to the AppsFlyer application as originating from the Tink app.

From an exposure perspective, the disclosure of this API key is likely a low- to medium-risk threat. An attacker could forge falsified data to poison the analytics database results using this key. Further investigation is required to determine if the key is also used for a secondary access mechanism (such as logging into the AppsFlyer administrative management page) that would represent a higher-risk threat.

For Unity and Xamarin applications, both distribute .NET CIL files, making reverse engineering straightforward with dotPeek or JustDecompile. Next, we'll look at an entirely different application architecture with the PhoneGap framework.



ADOBE PHONEGAP

Based on Apache Cordova framework

Mobile app development in HTML5, CSS, JavaScript

Attractive for many developers with web-centric background

- Simple to reuse web development code in a mobile form
- Apps built in PhoneGap can be exported to Android, iOS, Windows Phone, Tizen, Firefox OS, BlackBerry, and lots of other platforms you probably don't care about

SANS SEC575 | Mobile Device Security and Ethical Hacking 95

Adobe PhoneGap

Adobe PhoneGap is a framework for platform-independent mobile app development built on the open-source Apache Cordova framework. Using PhoneGap, developers build their applications using conventional HTML5, CSS, and JavaScript.

PhoneGap is an attractive platform for mobile app development because it utilizes the core development skills that are already used for website creation. This allows developers with a web development background to quickly build mobile applications, but it also represents an opportunity to reuse code built for websites in a mobile application.

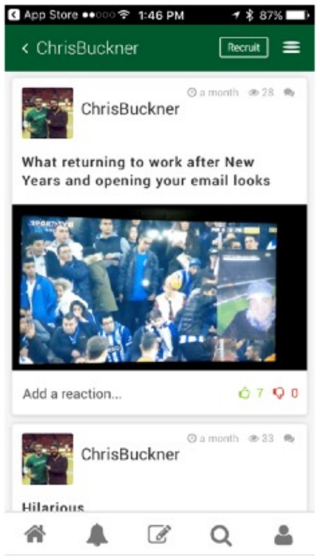
PhoneGap apps can be exported into Android and iOS applications but also lots of other platforms that you probably don't care about. This is the power of the framework: existing web development skills, web application code reuse, and access to multiple platforms from one interface.

FANREACT

Social media platform for fans to discuss sports and share reactions

- Text, pictures, and video uploads

Showcased at PhoneGap.com as an amazing app

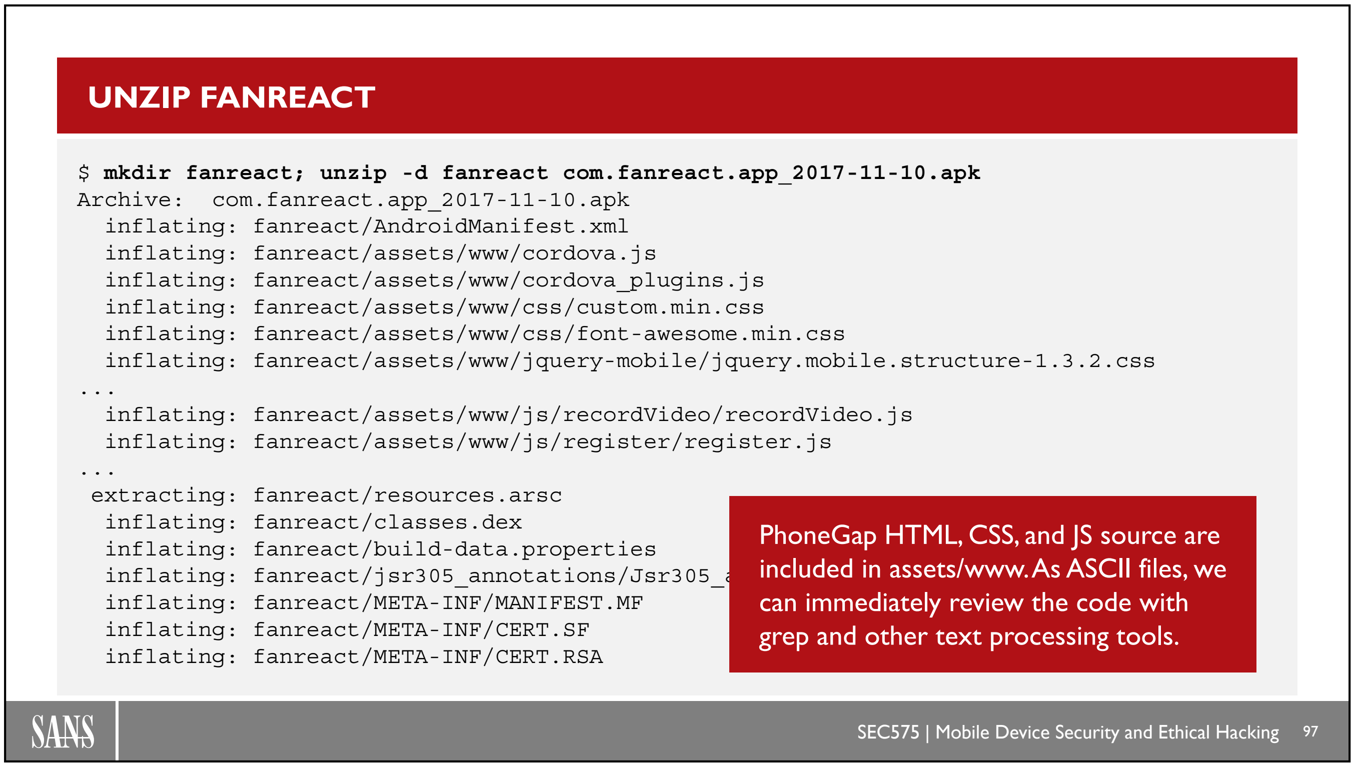


FanReact™

FanReact

For our analysis of PhoneGap applications, I chose to investigate the FanReact application. FanReact is a social media platform for sports fans to discuss sports and share reactions to sporting events in the form of text, pictures, and video uploads. FanReact is showcased at the Adobe PhoneGap website (http://phonegap.com/app/fanreact/) as an "amazing app".

This analysis is based on the FanReact version 1.5.0 application, available at the APKMonk website (http://www.apkmonk.com/app/com.fanreact.app/). I identified multiple vulnerabilities in the FanReact application during this analysis. I made several attempts to report the vulnerabilities to the FanReact organization through their support team but received no responses.



UNZIP FANREACT

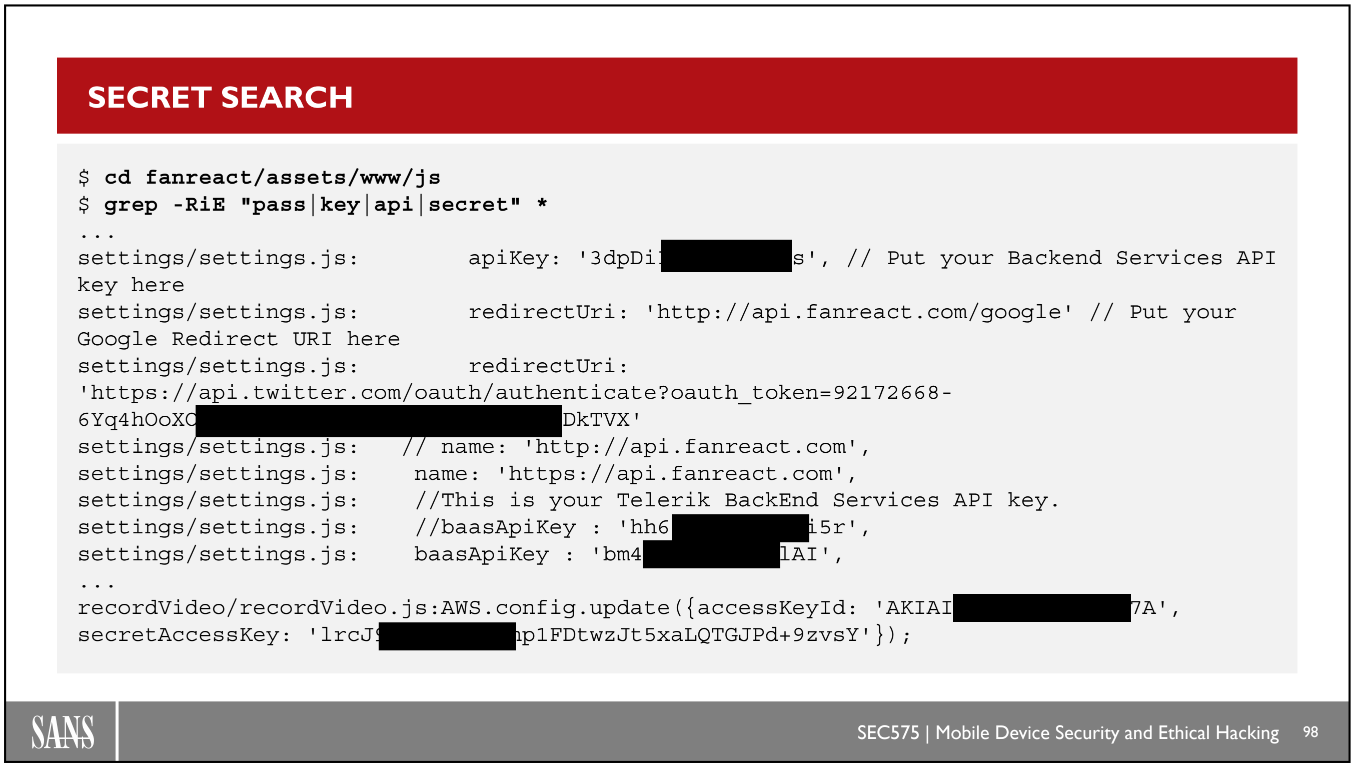
```
$ mkdir fanreact; unzip -d fanreact com.fanreact.app_2017-11-10.apk
Archive:  com.fanreact.app_2017-11-10.apk
  inflating: fanreact/AndroidManifest.xml
  inflating: fanreact/assets/www/cordova.js
  inflating: fanreact/assets/www/cordova_plugins.js
  inflating: fanreact/assets/www/css/custom.min.css
  inflating: fanreact/assets/www/css/font-awesome.min.css
  inflating: fanreact/assets/www/jquery-mobile/jquery.mobile.structure-1.3.2.css
...
  inflating: fanreact/assets/www/js/recordVideo/recordVideo.js
  inflating: fanreact/assets/www/js/register/register.js
...
 extracting: fanreact/resources.arsc
  inflating: fanreact/classes.dex
  inflating: fanreact/build-data.properties
  inflating: fanreact/jsr305_annotations/Jsr305_a
  inflating: fanreact/META-INF/MANIFEST.MF
  inflating: fanreact/META-INF/CERT.SF
  inflating: fanreact/META-INF/CERT.RSA
```

PhoneGap HTML, CSS, and JS source are included in assets/www. As ASCII files, we can immediately review the code with grep and other text processing tools.

SANS SEC575 | Mobile Device Security and Ethical Hacking 97

Unzip FanReact

Like our other analysis targets, we start with unzipping the APK file. PhoneGap HTML, CSS, and JS source are included in the `assets/`www directory. As ASCII files, we can immediately review code with `grep` or other text processing tools.



SECRET SEARCH

```
$ cd fanreact/assets/www/js
$ grep -RiE "pass|key|api|secret" *
...
settings/settings.js:        apiKey: '3dpDi[redacted]s', // Put your Backend Services API
key here
settings/settings.js:        redirectUri: 'http://api.fanreact.com/google' // Put your
Google Redirect URI here
settings/settings.js:        redirectUri:
'https://api.twitter.com/oauth/authenticate?oauth_token=92172668-
6Yq4hOoXC[redacted]DkTVX'
settings/settings.js:    // name: 'http://api.fanreact.com',
settings/settings.js:    name: 'https://api.fanreact.com',
settings/settings.js:    //This is your Telerik BackEnd Services API key.
settings/settings.js:    //baasApiKey : 'hh6[redacted]i5r',
settings/settings.js:    baasApiKey : 'bm4[redacted]lAI',
...
recordVideo/recordVideo.js:AWS.config.update({accessKeyId: 'AKIAI[redacted]7A',
secretAccessKey: 'lrcJ[redacted]p1FDtwzJt5xaLQTGJPd+9zvsY'});
```

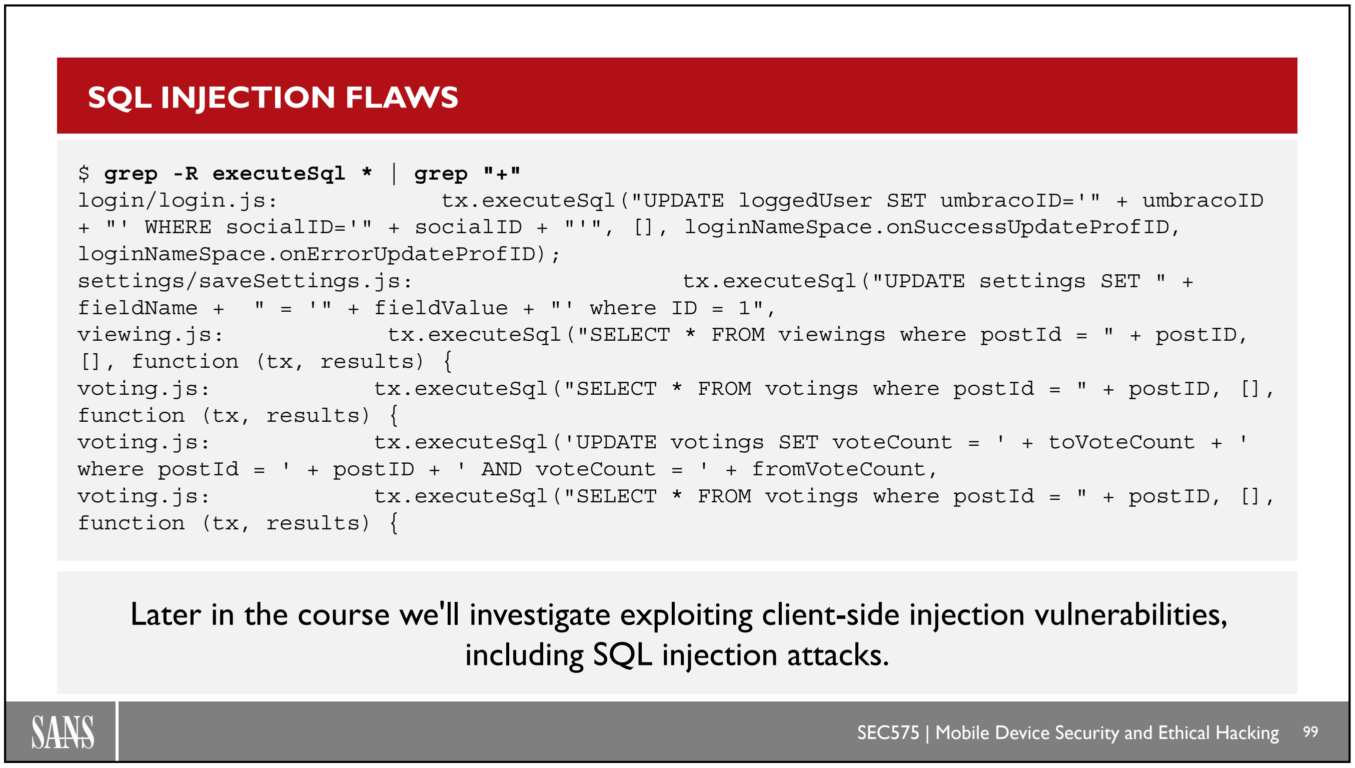
SANS SEC575 | Mobile Device Security and Ethical Hacking 98

Secret Search

JavaScript code is the most interesting component to us when analyzing PhoneGap applications. Located in the `assets/www/js` directory, we can use a recursive (`-R`), case-insensitive (`-i`) `grep` to search the JavaScript files using a basic regular expression (`-E`). Here we specify several strings we are interested in seeing, including `pass`, `key`, `api`, and `secret`, delimited with the vertical bar (`|`).

This command returned over 2,000 results. Filtering the output with grep `-v` to eliminate third-party libraries and other code that wasn't interesting (`grep -RiE "pass|key|api|secret" * | grep -Ev "/kendo|/underscore|/jquery|^import|^AmazonAWS"`) reduced the output to a more manageable 187 results, including the output shown on this page (actual keys here are partially obscured).

The most concerning entry here is the last, disclosing the Amazon AWS key ID and secret access key. With this information, an attacker could take over the AWS infrastructure for a target infrastructure unless the key itself is constrained for limited functionality. Ouch!



SQL INJECTION FLAWS

```
$ grep -R executeSql * | grep "+"
login/login.js:           tx.executeSql("UPDATE loggedUser SET umbracoID='" + umbracoID
+ "' WHERE socialID='" + socialID + "'", [], loginNameSpace.onSuccessUpdateProfID,
loginNameSpace.onErrorUpdateProfID);
settings/saveSettings.js:                   tx.executeSql("UPDATE settings SET " +
fieldName +  " = '" + fieldValue + "' where ID = 1",
viewing.js:            tx.executeSql("SELECT * FROM viewings where postId = " + postID,
[], function (tx, results) {
voting.js:            tx.executeSql("SELECT * FROM votings where postId = " + postID, [],
function (tx, results) {
voting.js:            tx.executeSql('UPDATE votings SET voteCount = ' + toVoteCount + '
where postId = ' + postID + ' AND voteCount = ' + fromVoteCount,
voting.js:            tx.executeSql("SELECT * FROM votings where postId = " + postID, [],
function (tx, results) {
```

Later in the course we'll investigate exploiting client-side injection vulnerabilities, including SQL injection attacks.

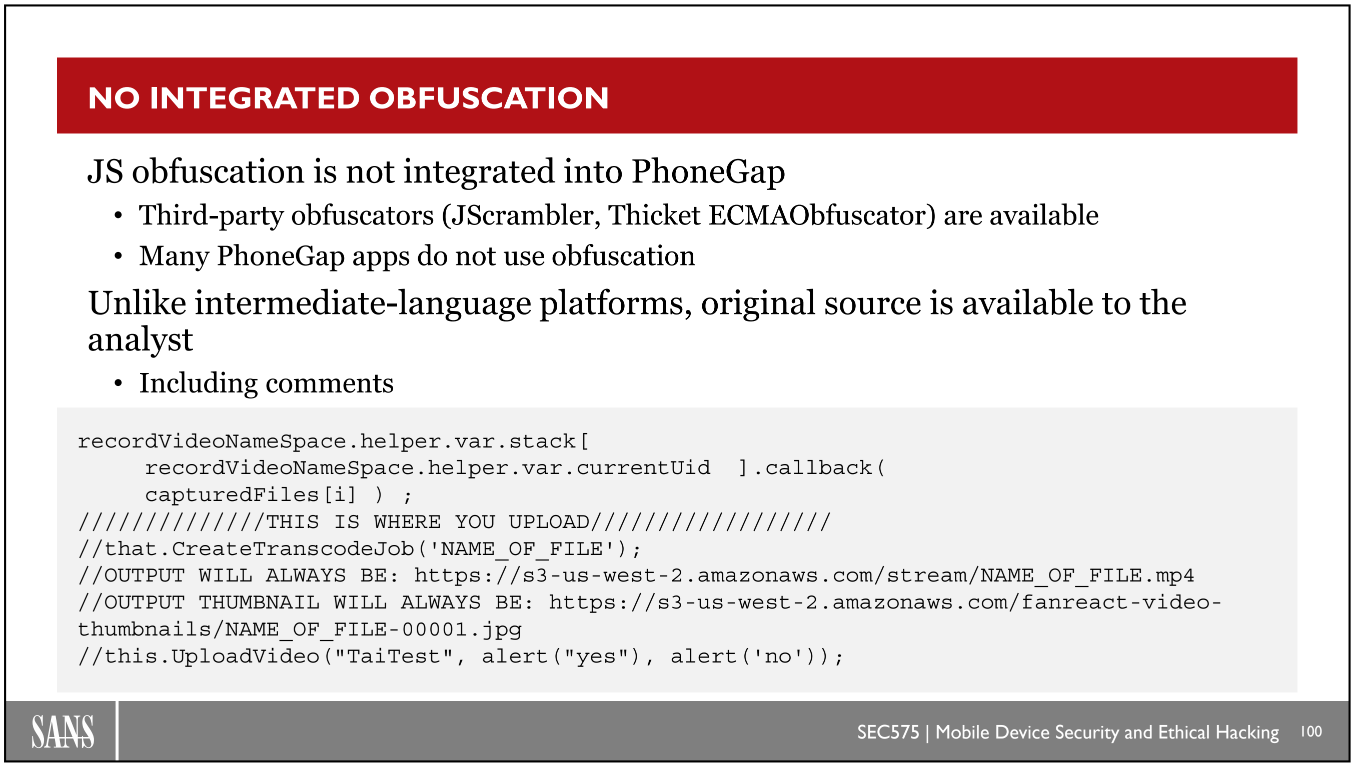
SANS

SEC575 | Mobile Device Security and Ethical Hacking 99

SQL Injection Flaws

Later in the course, we'll examine SQL injection flaws, but while I was evaluating the FanReact sources, I also did a quick check for SQL injection. In the output on this page, we see several examples where SQL statements are constructed using string concatenation. Examining these parameters in a Burp Suite proxy session, it appears that an attacker with privileged network access (MITM) could manipulate these parameters as well, making it possible to conduct client-side SQL injection attacks against FanReact users.

We'll investigate SQL injection attacks and client-side injection attacks in more depth later in the course. Here I wanted to demonstrate the ease with which we can identify vulnerabilities in this particular PhoneGap application, leveraging the available JavaScript source and simple command line searches.



NO INTEGRATED OBFUSCATION

JS obfuscation is not integrated into PhoneGap

- Third-party obfuscators (JScrambler, Thicket ECMAObfuscator) are available
- Many PhoneGap apps do not use obfuscation

Unlike intermediate-language platforms, original source is available to the analyst

- Including comments

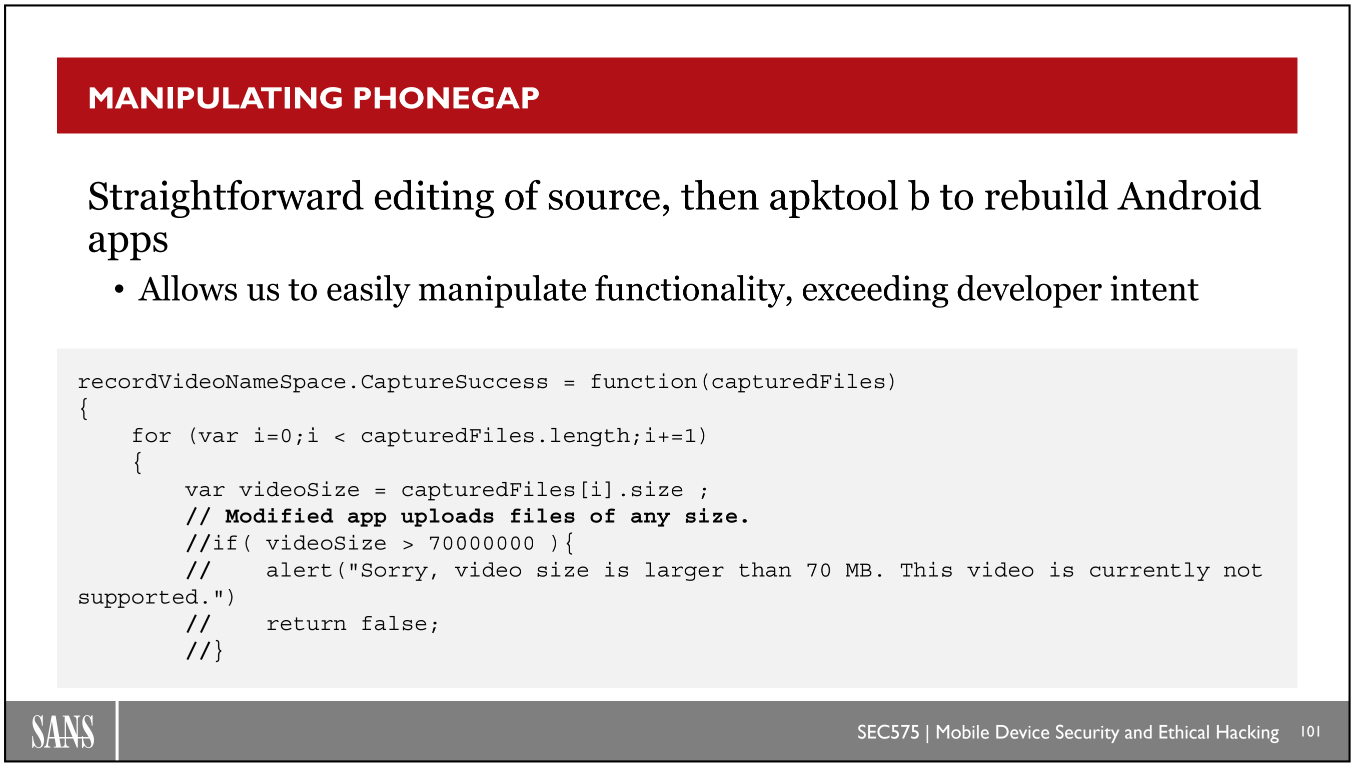
```
recordVideoNameSpace.helper.var.stack[
     recordVideoNameSpace.helper.var.currentUid  ].callback(
     capturedFiles[i] ) ;
//////////////THIS IS WHERE YOU UPLOAD/////////////////
//that.CreateTranscodeJob('NAME_OF_FILE');
//OUTPUT WILL ALWAYS BE: https://s3-us-west-2.amazonaws.com/stream/NAME_OF_FILE.mp4
//OUTPUT THUMBNAIL WILL ALWAYS BE: https://s3-us-west-2.amazonaws.com/fanreact-video-
thumbnails/NAME_OF_FILE-00001.jpg
//this.UploadVideo("TaiTest", alert("yes"), alert('no'));
```

SANS SEC575 | Mobile Device Security and Ethical Hacking 100

No Integrated Obfuscation

PhoneGap does not include JavaScript obfuscation capabilities, so most PhoneGap applications distribute the original JavaScript files written by the developers. It is possible to integrate third-party JavaScript obfuscators in the PhoneGap build process, such as JScrambler (https://jscrambler.com/en/) and Thicket's ECMAObfuscator (https://www.semanticdesigns.com/Products/Obfuscators/ECMAScriptObfuscator.html).

From a reverse engineering perspective, PhoneGap applications not only disclose the original source code, class, method, and variable names, but also the comments in the code. In the example on this page, we see the comments left by a developer that describe the functionality for uploading MP4 files for transcoding and the exact paths where the files will be available. Combined with the previously disclosed AWS key, an attacker could use this functionality to turn the FanReact AWS service into a hosting site for MP4 files outside of the FanReact application functionality.



MANIPULATING PHONEGAP

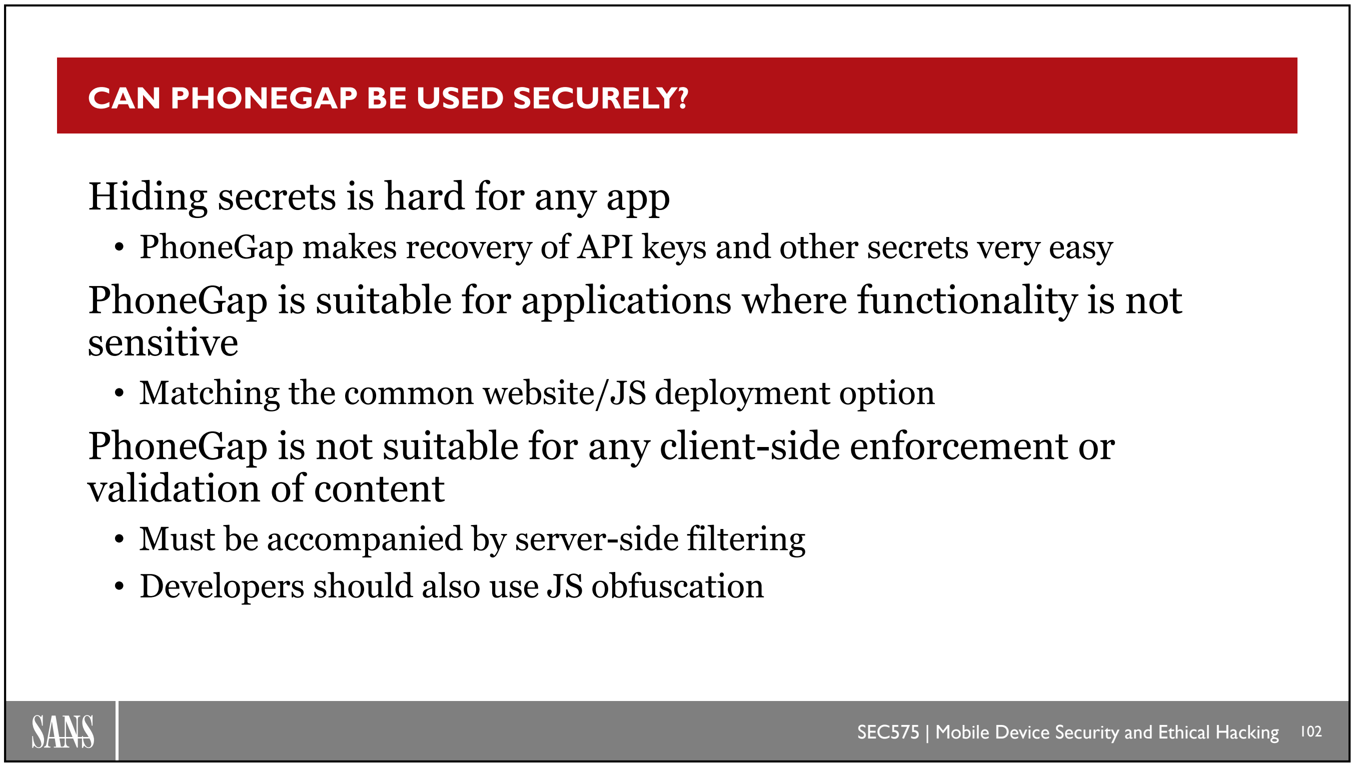
Straightforward editing of source, then apktool b to rebuild Android apps

- Allows us to easily manipulate functionality, exceeding developer intent

```
recordVideoNameSpace.CaptureSuccess = function(capturedFiles)
{
    for (var i=0;i < capturedFiles.length;i+=1)
    {
        var videoSize = capturedFiles[i].size ;
        // Modified app uploads files of any size.
        //if( videoSize > 70000000 ){
        //    alert("Sorry, video size is larger than 70 MB. This video is currently not
supported.")
        //    return false;
        //}
```

Manipulating PhoneGap

It's easy to change the functionality for PhoneGap applications. Simply decompile the application using `apktool d`, then edit the JavaScript source and recompile using `apktool b`. For example, FanReact does not allow users to upload videos larger than 70 MB, raising an alert when the user tries to do so. To build a version of the application that does not have this limitation, we can simply comment out the video size check (as shown on this page) and rebuild the application.



CAN PHONEGAP BE USED SECURELY?

Hiding secrets is hard for any app

- PhoneGap makes recovery of API keys and other secrets very easy

PhoneGap is suitable for applications where functionality is not sensitive

- Matching the common website/JS deployment option

PhoneGap is not suitable for any client-side enforcement or validation of content

- Must be accompanied by server-side filtering
- Developers should also use JS obfuscation

SEC575 | Mobile Device Security and Ethical Hacking 102

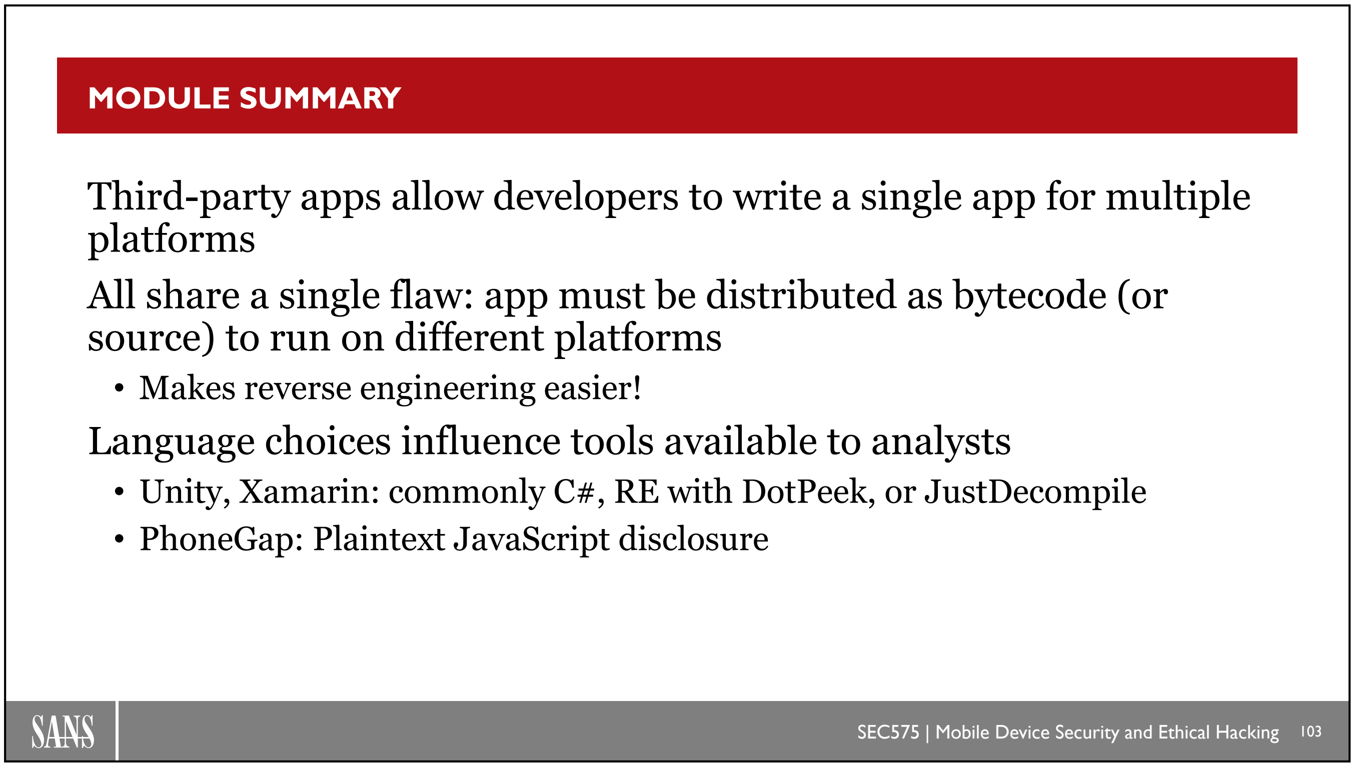
Can PhoneGap Be Used Securely?

My quick analysis of the FanReact app written with PhoneGap reveals several vulnerabilities. It's reasonable for a developer to ask if those vulnerabilities are inherent to the PhoneGap architecture or unique to the FanReact app itself.

The disclosure of API keys and secrets in the FanReact app is a common issue for many mobile applications. In general, it's hard to hide secrets for any app, yet many frameworks, third-party libraries, and external services expect developers to embed secrets anyway. Unlike other platforms, PhoneGap-developed apps make it very easy to search for and identify these secrets, but any iOS or Android app would similarly be vulnerable to this kind of secret disclosure. Secure data storage in the iOS Keychain or the Android Keystore would make disclosure of these keys harder, but ultimately, they will always be accessible to a privileged attacker on the device (e.g., an attacker with a jailbroken or rooted mobile device).

PhoneGap is suitable for applications where the functionality of the application is not particularly sensitive or secretive. This closely matches the web application model where JavaScript is easily accessible but not relied upon for all system functionality (e.g., server-side processing always validates input from the client).

PhoneGap is not suitable for an application where it must rely on client-side enforcement for content validation and should always be deployed with server-side filtering. Developers should also consider using JavaScript obfuscation in their PhoneGap apps to make analysis more difficult.



MODULE SUMMARY

Third-party apps allow developers to write a single app for multiple platforms

All share a single flaw: app must be distributed as bytecode (or source) to run on different platforms

- Makes reverse engineering easier!

Language choices influence tools available to analysts

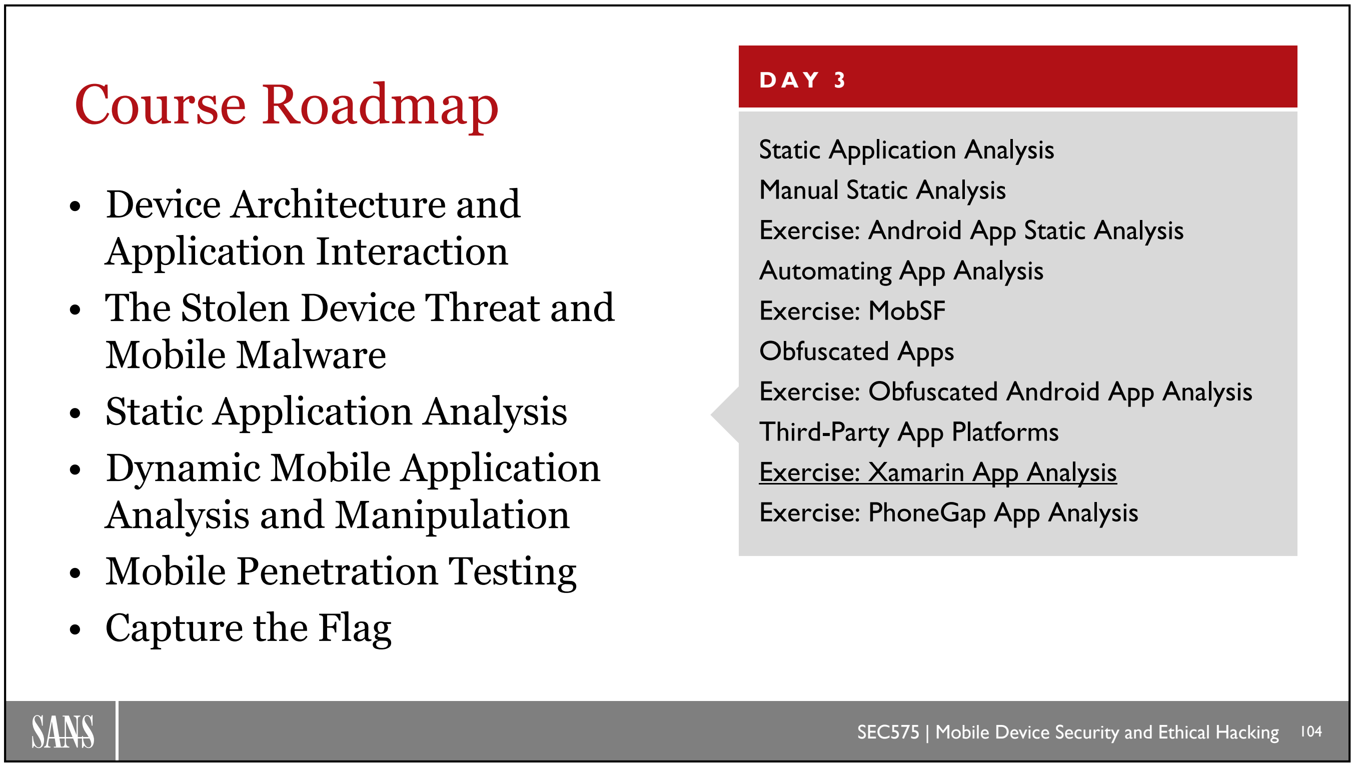
- Unity, Xamarin: commonly C#, RE with DotPeek, or JustDecompile
- PhoneGap: Plaintext JavaScript disclosure

Module Summary

In this module, we looked at the techniques for reverse engineering third-party development platforms from Unity, Xamarin, and PhoneGap. While behavioral analysis techniques for these applications remain the same, we need to employ different tools and techniques to perform static analysis.

From a developer perspective, third-party development platforms offer a tremendous advantage: write your code in one language and generate applications for all the platforms you want to support. From a business perspective, cost is reduced, support is simplified, and the time to market is reduced. These are significant motivators for organizations to want to adopt third-party development platforms.

Unfortunately, security suffers when third-party development platforms are used. All three of the platforms we looked at share a common flaw: the app must be distributed as bytecode CIL (or source) to run on different platforms with different processor architectures. This makes our job as reverse engineering analysts easier, allowing us to use .NET decompilation tools for Unity and Xamarin applications, or simply text tools for PhoneGap applications.



Course Roadmap

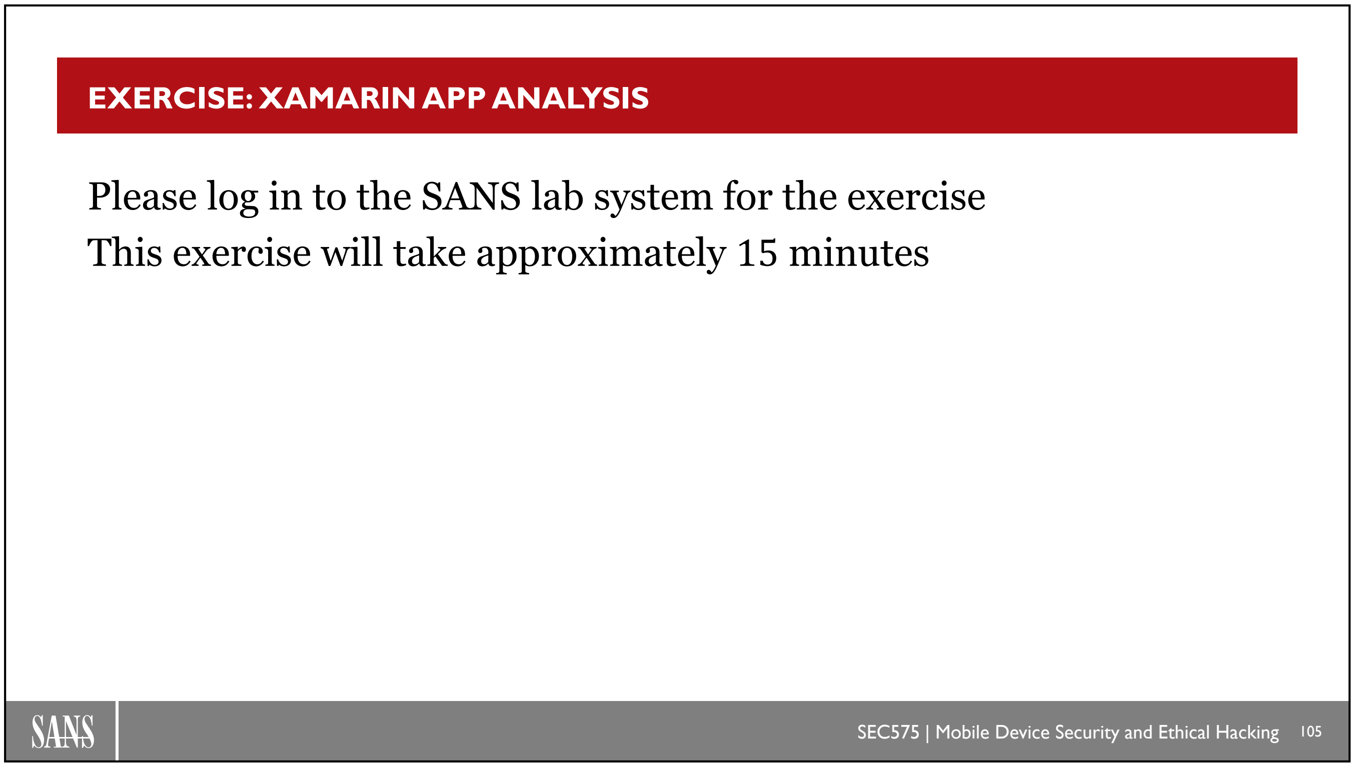
- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 3

Static Application Analysis
Manual Static Analysis
Exercise: Android App Static Analysis
Automating App Analysis
Exercise: MobSF
Obfuscated Apps
Exercise: Obfuscated Android App Analysis
Third-Party App Platforms
Exercise: Xamarin App Analysis
Exercise: PhoneGap App Analysis

SANS | SEC575 | Mobile Device Security and Ethical Hacking 104

This page intentionally left blank.



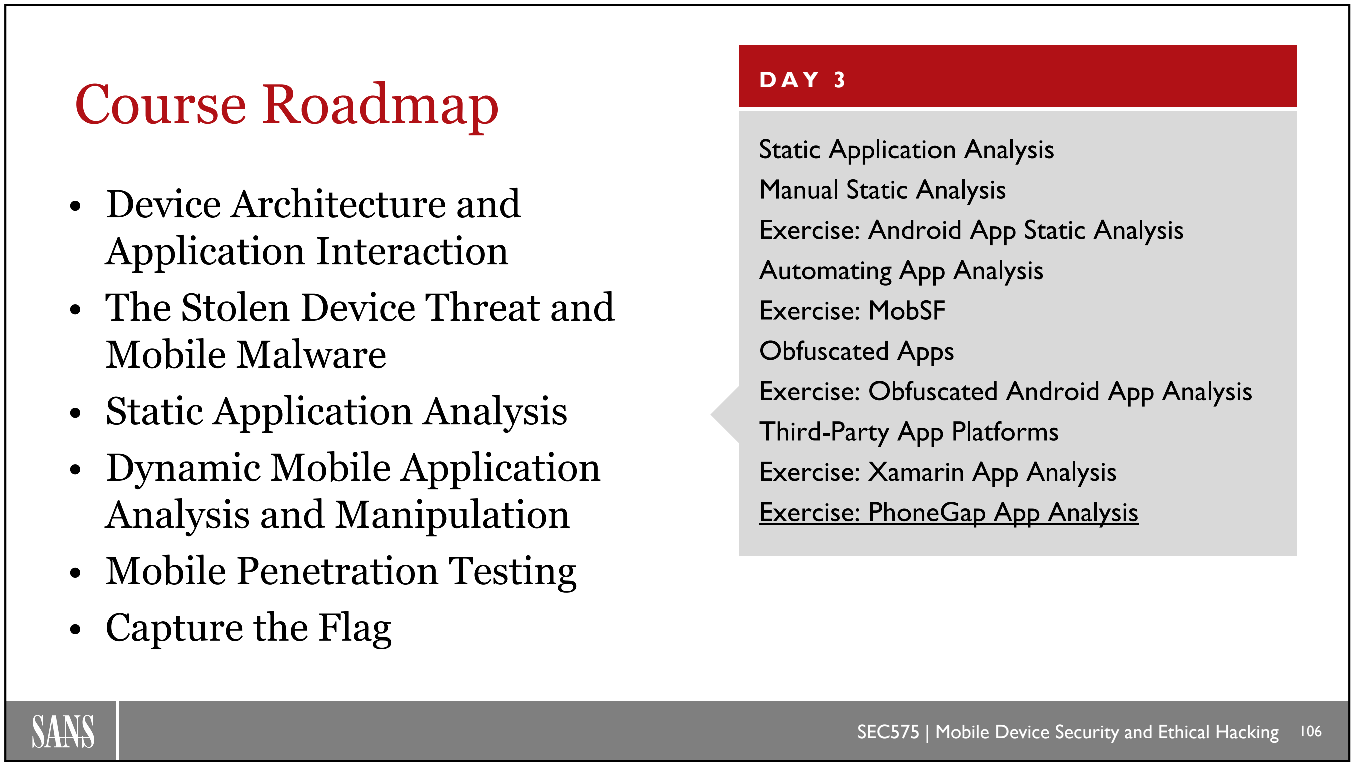
EXERCISE: XAMARIN APP ANALYSIS

Please log in to the SANS lab system for the exercise
This exercise will take approximately 15 minutes

SANS SEC575 | Mobile Device Security and Ethical Hacking 105

Exercise: Xamarin App Analysis

Please log in to the SANS lab system for the Xamarin App Analysis exercise. This exercise will take approximately 15 minutes to complete.



Course Roadmap

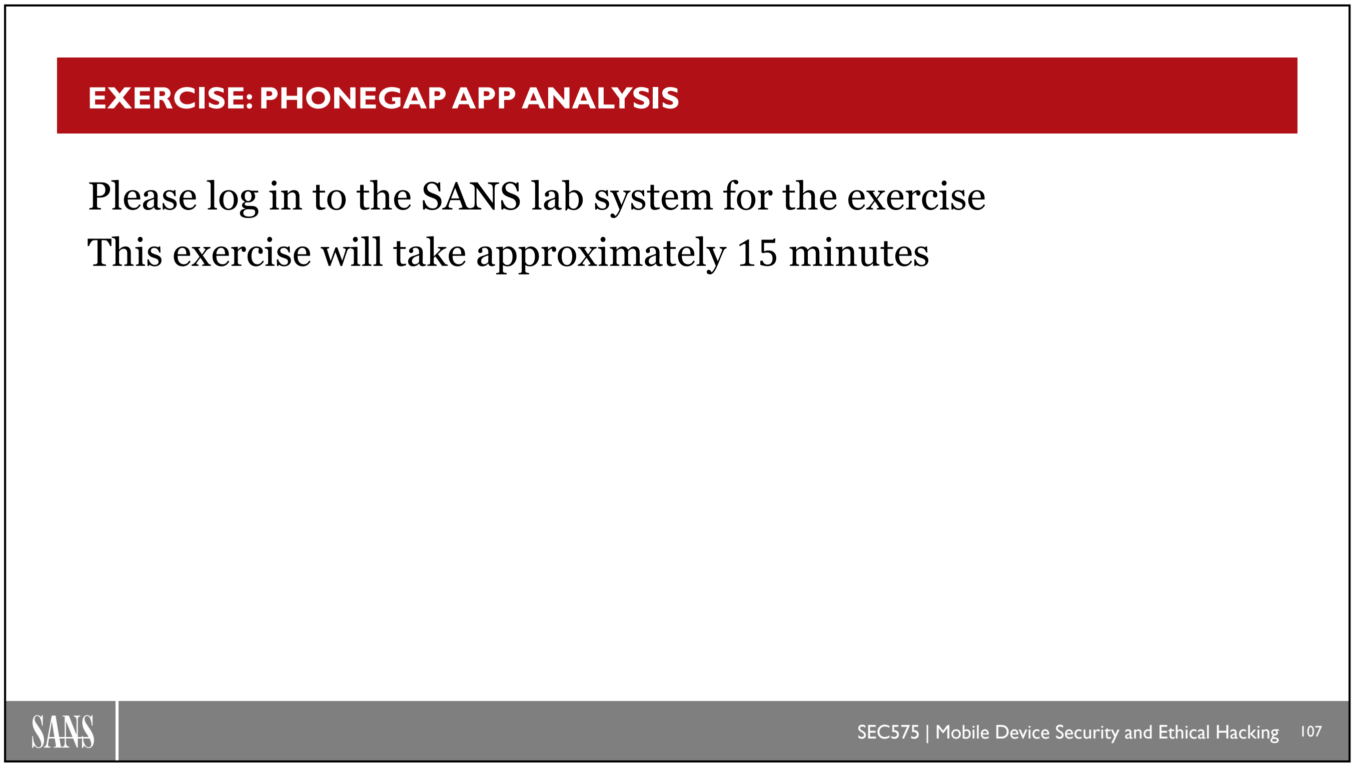
- Device Architecture and Application Interaction
- The Stolen Device Threat and Mobile Malware
- Static Application Analysis
- Dynamic Mobile Application Analysis and Manipulation
- Mobile Penetration Testing
- Capture the Flag

DAY 3

Static Application Analysis
Manual Static Analysis
Exercise: Android App Static Analysis
Automating App Analysis
Exercise: MobSF
Obfuscated Apps
Exercise: Obfuscated Android App Analysis
Third-Party App Platforms
Exercise: Xamarin App Analysis
Exercise: PhoneGap App Analysis

SANS SEC575 | Mobile Device Security and Ethical Hacking 106

This page intentionally left blank.



EXERCISE: PHONEGAP APP ANALYSIS

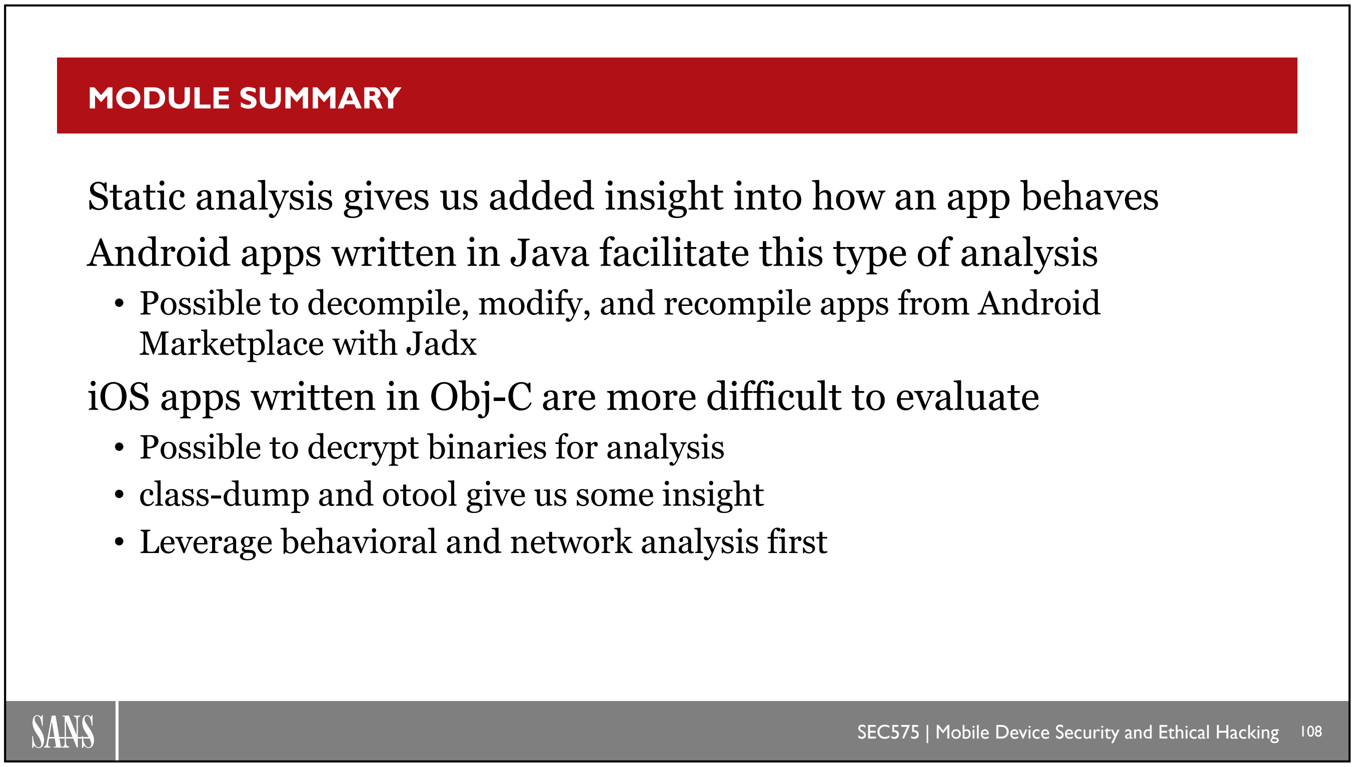
Please log in to the SANS lab system for the exercise
This exercise will take approximately 15 minutes

SANS

SEC575 | Mobile Device Security and Ethical Hacking 107

Exercise: PhoneGap App Analysis

Please log in to the SANS lab system for the PhoneGap App Analysis exercise. This exercise will take approximately 15 minutes to complete.



MODULE SUMMARY

Static analysis gives us added insight into how an app behaves

Android apps written in Java facilitate this type of analysis

- Possible to decompile, modify, and recompile apps from Android Marketplace with Jadx

iOS apps written in Obj-C are more difficult to evaluate

- Possible to decrypt binaries for analysis
- class-dump and otool give us some insight
- Leverage behavioral and network analysis first

SANS

SEC575 | Mobile Device Security and Ethical Hacking 108

Module Summary

In this module, we reviewed techniques to perform static analysis on mobile device binaries. For Android applications written in Java, different decompilers allow us to easily decompile Dalvik files for analysis to both SMALI and Java. Accordingly, attackers can also decompile Dalvik files, creating modified versions that include malicious code.

On Apple iOS platforms, Objective-C programs stored in Mach-O files are more difficult to evaluate. Apple instituted some controls such as executable encryption that can be bypassed, but the compiled nature of applications proves to be of significant benefit in thwarting application static analysis.