

Reference: [Computer Security by Stallings and Brown, Chapter 24](#)

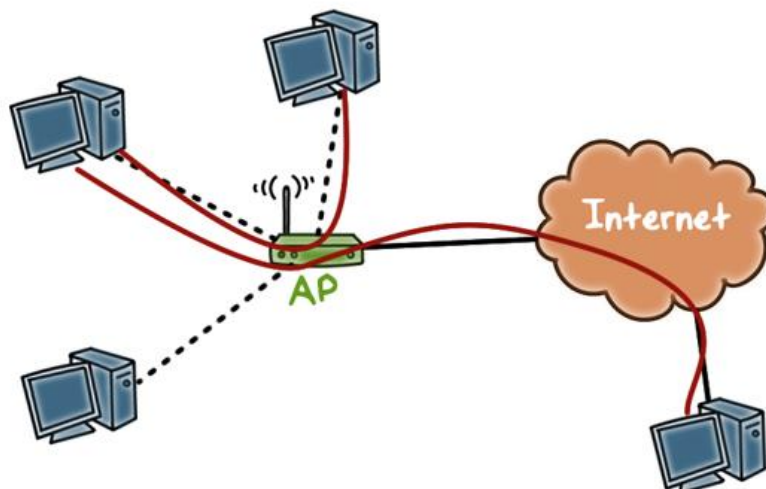
Wireless and Mobile Security

Lesson Introduction

- WiFi security
- iOS security
- Android security

Most users now use WiFi enabled devices, such as laptops, to connect to the network. In this lesson, we will first briefly discuss the WiFi security standards. A more recent trend is that more users now using smart phones for important tasks. Therefore, we will also cover iOS security and Android security.

Introduction to WiFi



Let's first briefly review the WiFi technology. A typical use of WiFi is to allow personal computers or devices enabled with WiFi in a locale such as a home to access the Internet through the access point, or AP. The AP connects to these devices wirelessly. And connects to the Internet through physical wiring. Such as to a router provided by the Internet service provider. Devices in a locale can also connect to each other wirelessly through the AP.

Instructor Notes: Windows devices account for 80% of malware infections transmitted via mobile networks

Introduction to WiFi



- No inherent **physical protection**
- **Broadcast** communications

In wireless networking, data is not transmitted via physical wiring. Instead data is transmitted in air which is an open medium. In other words, there is no inherent physical protection of communications. Also since there's no hard wiring connecting two devices for direct communications, in wireless networking, communication between two devices is

achieved by one device broadcasting and the other device listening to the broadcast.



WiFi Quiz

Select all that apply.

Which of the following are security threats to WiFi:

- ☐ Eavesdropping
- ☐ Injecting bogus messages
- ☐ Replaying previously recorded messages
- ☐ Illegitimate access to the network & its services
- ☐ Denial-of-service
- ☐ All the above

Given the characteristics of WiFi and wireless networking. Now let's do a quiz. Which of the following are security threats to WiFi? Select all that apply.

First, eavesdropping. This means attacker listening to communications. Second, injecting bogus messages. Third, replaying previously recorded messages. Fourth, illegitimate access to the network & its services. Fifth, denial of service. Sixth, all the above.

The answer is all the above. In fact, all of these threats are also threats to wired networking. And one is networking because there's no inherent physical protection and the nature of broadcast communication. These threats are even more serious.

- ☐ Eavesdropping
- ☐ Injecting bogus messages
- ☐ Replaying previously recorded messages
- ☐ Illegitimate access to the network & its services
- ☐ Denial-of-service
- ☒ All the above

Overview of WiFi Security



- Early solution was **based on WEP**
 - **seriously flawed**
 - not recommended to use
- **New security standard for WiFi is 802.11i**, implemented as WiFi Protected Access II (WPA2)

Now, let's discuss WiFi security. The earlier WiFi security standard is called Wired Equivalent Privacy, or WEP. WEP has been shown to be easily pickable even when it is correctly configured. Therefore, you should no longer use WEP. The new standard is 802.11i, and it is implemented as WPA2. WPA stands for WiFi Protected Access II.

Overview of 802.11i

Main advantages over WEP

- **access control model** is based on 802.1X
- **flexible authentication framework** (based on EAP
 - Extensible Authentication Protocol)
 - Carrier protocol designed to transport the messages of real authentication protocols (e.g., **TLS – Transport Layer Security**)

TLS. In other words, you can implement a host of different authentication methods on top of EAP. In other words, 802.1x can accommodate a host of different authentication methods.



Now let's take a look at 802.11i. 802.11i enforces access control and the access control protocol is based on another standard called 802.1x.

802.1x is flexible, because it is based on the Extensible Authentication Protocol or EAP. EAP is designed as a carrier protocol and its purpose is to transport the messages of the real authentication protocols such as

Overview of 802.11i

Main advantages over WEP

- authentication process results in a shared session key (which **prevents session hijacking**)
- different functions (encryption, integrity) use **different keys derived from the session key** using a one-way function
- **integrity protection** is improved
- **encryption function** is improved

In addition, 802.11i supports good or strong security practices. For example, it uses different keys for encryption versus integrity protection. And it also uses stronger encryption methods. In particular, it uses AES.



The more and advanced EAP methods such as TLS provide mutual authentication. This means that it will limit man in the middle attacks. Because it authenticates both the server and the client to each other. Furthermore, this EAP method results in key material, which can be used to generate dynamic encryption keys. That means the encryption keys will change dynamically over time. In



WiFi Security Standards Quiz

Choose the best answer:

Now let's do a quiz on WiFi security standards. Which security standard should be used for WiFi? Choose the best answer. Is it WEP? Or WPA2?

Which security standard should be used for WiFi?

☐

WEP

☐

WPA2

The answer is WPA2, because without having shown to be easily breakable, even when it is correctly configured, and the new security standard is WPA2. Therefore, you should use WPA2 as the security standard for WiFi.

☐

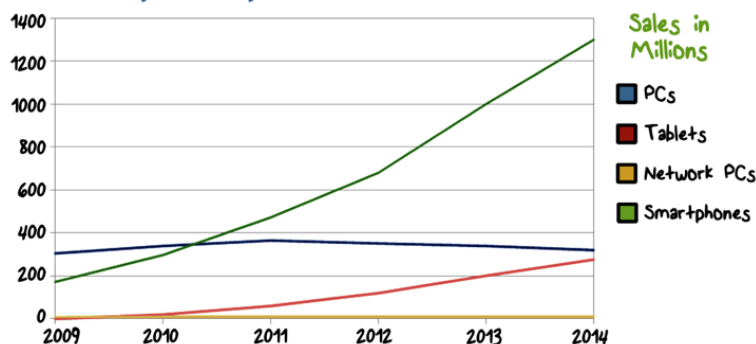
WEP

☒

WPA2

Overview of Smartphone Security

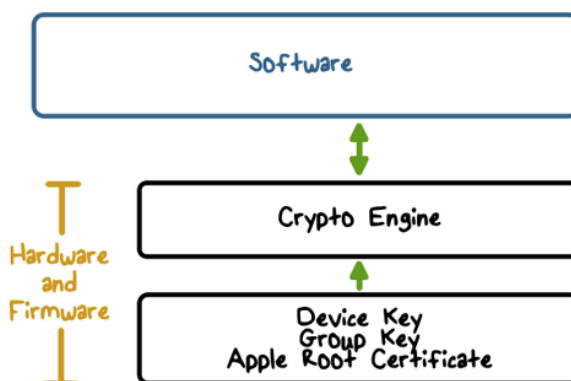
PC, Tablet, Network PC and Smartphone



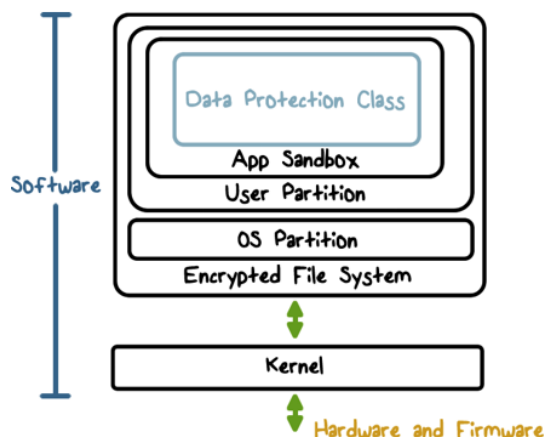
Now let's discuss smartphone security. Here we plot out the sales figures of different devices over the years. It is clear that more people now are using smartphones and they're using them for more and more important tasks. Therefore, it is important for us to understand the security of smartphones.

First let's take a look at the security of iOS. The iOS security architecture combines both software and hardware features to provide security to an iOS device such as an iPhone or iPad. It has built in crypto capabilities to support data protection, such as confidentiality and integrity. For example, notice that the crypto engine and the crypto keys are embedded in the hardware.

Overview of iOS Security



Overview of iOS Security



And of course, beta protection is applied to not just the system files but also application data. The security architecture also provides various strong isolation mechanisms, for example, it uses App Sandbox to protect app security. This enables the apps to run securely. For example, apps cannot interfere with each other and also this ensure the integrity of the system. That is, even if an app is compromised, it's damage to the system is very limited.



Operating System Vulnerabilities Quiz

Select three operating systems with **the most vulnerabilities in 2014**:

- | | |
|--|--|
| <input type="checkbox"/> Apple Mac OS X | <input type="checkbox"/> Apple iOS |
| <input type="checkbox"/> Linux Kernel | <input type="checkbox"/> Microsoft Windows Server 2012 |
| <input type="checkbox"/> Microsoft Windows Vista | <input type="checkbox"/> Microsoft Windows 7 |
| <input type="checkbox"/> Microsoft Windows 8 | |

Now let's do a quiz on the vulnerabilities of operating systems. Select three operating systems with the most vulnerabilities in 2014. Is it Mac OS X, iOS, Linux, Microsoft Windows Server, Microsoft Windows Vista, Microsoft Windows 7, or Microsoft Windows 8?

Instructor Notes: Most Vulnerable Operating Systems and Applications in 2014

So the top three operating systems with the most vulnerabilities in 2014 are Mac OS X, iOS, and Linux Kernel. These answers may be surprising because you might have thought that Apple platforms are

inherently more secure. However, as Apple platforms, in particular iOS devices, gain market shares, they become a high target for the attackers. Therefore mobile abilities are being discovered and exploited.

- | | |
|--|--|
| <input checked="" type="checkbox"/> Apple Mac OS X | <input checked="" type="checkbox"/> Apple iOS |
| <input checked="" type="checkbox"/> Linux Kernel | <input type="checkbox"/> Microsoft Windows Server 2012 |
| <input type="checkbox"/> Microsoft Windows Vista | <input type="checkbox"/> Microsoft Windows 7 |
| <input type="checkbox"/> Microsoft Windows 8 | |

Hardware Security Feature

- Each iOS device has a **dedicated AES-256 crypto engine**
- Manufacture Keys**
 - Apple provides the Device ID (UID) and the device group ID (GID) as AES 256 Bit keys
 - While the UID is unique to each device, the GID represents a processor class (e.g., Apple A5 processor)
 - The UID and GID keys are directly burned into the silicon** and can only be accessed by the Crypto Engine

Now let's take a look at the hardware security support in iOS devices. Each iOS device has a dedicated AES-256 crypto engine built into the direct memory access path between the flash storage and the main system memory. This makes file encryption highly efficient.

Recall that AES-256 means that the key length is 256-bit. The device's unique ID or UID, and the device group ID or GID, are AES-256 bit keys fused or compelled into the application processor, a secure enclave during manufacturing. This means that no software or firmware can read them directly. They can see only the results of encryption or decryption operations performed by the dedicated AES engines implemented in silicon using the UID or GID as a key. This is an important feature because the keys are stored securely in the hardware. The UIDs are unique to each device and are not recorded by Apple or its suppliers.

The GIDs are common to all processors in a class of devices. For example, all devices using the Apple A8 processor. The GIDs are used for tasks such as delivering system installation and updates. Again, a unique feature of iOS devices is that the UID and GID keys are directly burned in the silicon and they can be only accessed by the Crypto Engine. And the Crypto Engine itself is part of the hardware.

iOS Trusted Bootchain



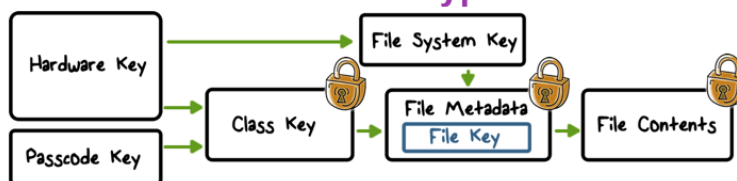
The security of a device should be established when the device is turned on initially. iOS uses trusted bootchain to achieve this goal.

When an iOS is turned on, each application processor immediately executes code from boot only

memory known as the bootrom. This immutable code known as the hardware root of trust, is laid down during chip fabrication and is implicitly trusted. In other words, this code is burnt into the hardware.

The bootrom code contains the Apple root CA public key, which is used to verify that the low level loader, LLB is signed by Apple properly before allowing it to load. This is the first step in the chain of trust where each step ensures that the next is signed by Apple properly. When the LLB finishes its tasks, it verifies and runs the next stage below the iBoot. Which in turn verifies and runs the iOS kernel. This secure bootchain helps ensure that the lowest levels of software are not tampered with. And allows iOS to run only on validated Apple devices.

File Data Encryption



- Every file is encrypted with a unique **File Key**, that is generated when the file is created
- The file key is wrapped with a **Class Key** and stored in the file's metadata
- The metadata is encrypted with the **File System Key**
- The **Class key** is protected by the **Device UID** and (if configured for some files) the User Passcode

In addition to the hardware crypto capabilities built into each iOS device, Apple uses a technology called data protection to further protect data stored in flash memory on the device.

Data protection allows the device to respond to common events. Such as incoming phone calls but also enables a high level of encryption for user data. Critical system apps such as

messages, mail, calendar, contacts, photos and health data values use data protection by default. And third party apps install on iOS 7 or later receive this protection automatically.

Data protection is implemented by constructing and managing a hierarchy of keys. And views on the hardware encryption technologies going to each iOS device. Data protection is controlled on a profile basis by signing each file a class. And access to the file is determined by whether the class keys have been unlocked. Each time a file is created, data protection creates a new 256 bit profile key and gives it to the hardware AS engine which uses the key to encrypt a file as it is written to flash memory and the encryption is sent through ASCBC mode. The profile key is wrapped or encrypted with one or several class keys. And the wrapped per file key is stored in the files metadata. That is, the file key is used to encrypt the file contents and the key itself is encrypted using the class key and the encrypted or wrapped key is stored in the files metadata.

The file metadata itself is encrypted using a file system key. Then when a file is opened or accessed this metadata is decrypted using the file system key. This will reveal the route to profile key then the profile key is unwrapped with the class key. The profile key then can be used to decrypt the file as it is read from the flash memory.

The metadata of all files in the file system is encrypted using the same random key. We call it the file system key. This key is created when iOS is first installed, or when the device is wiped by the user. The class key is protected with the hardware UID, and for some classes, with the user's passcode.



Security Quiz

Mark all the answers that are true.

- ☐ All cryptographic keys are stored in flash memory
- ☐ Trusted boot can verify the kernel before it is run
- ☐ All files of an app are encrypted using the same key

Mark all the answers that are true.

First, all cryptographic keys are stored in flash memory. Second, trusted boot can verify the kernel before it is run. Third, all files of an app are encrypted using the same key.

First all cryptographic keys are stored in flash memory. This is false because the device's UID and GID are fused into the processors, therefore they're not stored in the flash memory. Second, trusted boot can verify the kernel before it is run.

This is true because that's purpose of

trusted boot at each stage can verify the next is signed properly by Apple. Therefore, the kernel can be verified before it is run. Third, all files of an app are encrypted using the same key. This is false because as we have discussed, each file has a profile key to encrypt its contents.



All cryptographic keys are stored in flash memory

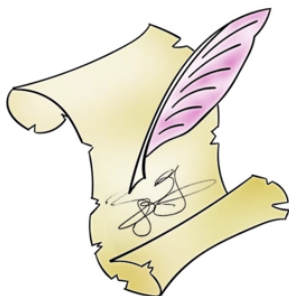


Trusted boot can verify the kernel before it is run



All files of an app are encrypted using the same key

Mandatory Code Signing



•All executable code has to be signed by a trusted party

•Apps from App Store are signed by Apple

•No dynamic code generation or self-modifying

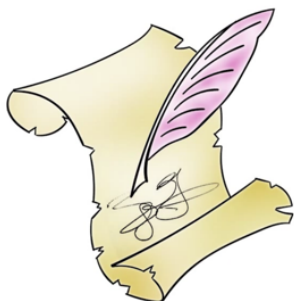
Once the iOS kernel has started, it controls which user processes and apps can be run to ensure that all apps come from a known or approved source and have not been tampered with.

iOS requires that all executable code be signed using an Apple issued certificate. More specifically, apps provided with the device, such as

Mail or Safari, are already signed by Apple. Third party apps must also be validated and signed using an Apple-issued certificate. This mandatory co-signing extends the concept of chain of trust from the iOS to the apps, and prevents third party apps from loading unsigned code, or using self modifying code.

At run time co-signature checks of all executable memory pages are being performed as the pages are loaded to ensure that our apps has not been modified since it was installed or last updated. This is done through a user-space daemon and is enforced by the kernel. In summary, iOS enforces mandatory code signing and identification to extend the chain of trust from iOS to the apps.

Mandatory Code Signing



•Code signing check

- Enforced by kernel, handled by a user-space daemon

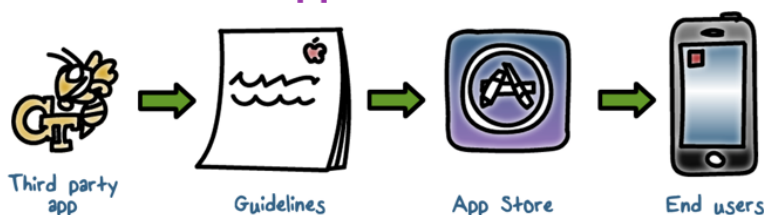
•Mandatory code signing

As we have discussed, iOS requires military code signing. This means that, apps that come with the device, such as MAIL or Safari, are signed by Apple. And the third party apps must be verified and signed, using an Apple-issued certificate. Now let's take a look at the process by which a third party app developer can obtain the Apple issued certificate.

In order to develop apps on iOS devices, developers must register with Apple and join the iOS Developer Program. The real world identity of each developer, whether an individual or a business, is verified by Apple before their certificate is issued. This certificate enables developers to sign apps and submit them to the app store for distribution. As a result, all apps in the app store have been

submitted by an identifiable person or organization, serving as a deterrent to the creation of malicious apps. Furthermore, all apps in the app store have been reviewed by Apple, to ensure that they operate as described, and don't obtain obvious bugs or other programs. iOS devices are only allowed to download apps from the official Apple app store. And because of this restricted app distribution model, plus co-signing, it is very difficult to upload malware to the app store and have devices download such malware.

Restricted App Distribution Model



- Third-party apps have to be **reviewed by Apple**. The apps that passed the review are signed by Apple
- iOS devices are only allowed to download apps through the App Store**



App Store Security Quiz

Choose the best answer




In 2013 researchers were able to bypass Apple's App store security. **What method did they use?**

- ☐ Uploaded malware disguised as an app without authorization, bypassing the review and check process.
- ☐ Uploaded an app that after it passed the review process morphed into malware.
- ☐ Uploaded an app that led users to a site that contained malware.

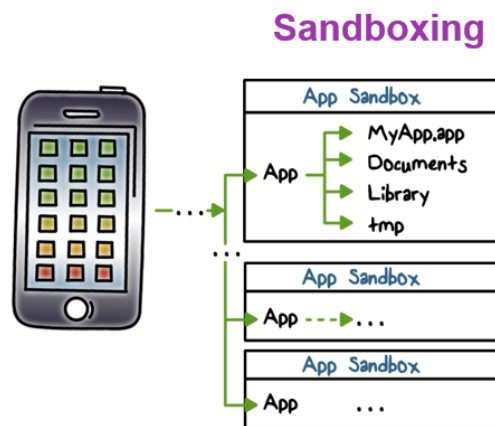
uploaded an app that led users to a site that contained malware.

However, despite these measures there are still security holes. Let's do a quiz. Choose the best answer. In 2013, researchers were able to bypass Apple's app store security. What method did they use? First, uploaded malware disguised as an app without authorization, bypassing the review and check process. Second, uploaded an app that after it passed the review process morphed into malware. Third,

First, upload the malware disguised as an app without authorization bypassing the review and check process. We know that's not possible given the restricted app distribution model by iOS. Second, uploading an app that after it passed the review process morphed into malware. This is possible and, indeed, this is the method that the researcher had used. In the spirit of full disclosure, I was one of the researchers. A link to an article about the research can be found in the instructor's notes. Third, uploading an app that led users to a site that contained malware. This is not the method that the researchers used. In effect, iOS would prevent malware from being downloaded from a site and run on the device.

-  Uploaded malware disguised as an app without authorization, bypassing the review and check process.
-  Uploaded an app that after it passed the review process morphed into malware.
-  Uploaded an app that led users to a site that contained malware.

Instructor Notes: Researchers Outwit Apple



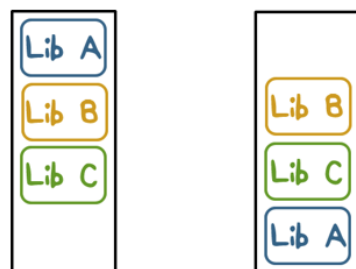
- **Each app has a unique home directory** for its files
- **Apps are restricted** from accessing files stored by other apps or from making changes to the device

Once an app is verified to be from an approved source, iOS enforces security measures, designed to prevent it from compromising other apps, or the rest of the system. All third party apps are sandboxed, so that they are restricted from accessing files stored by other apps, or from making changes to the device. This prevents apps from gathering or modifying information stored by other apps. Each app has a

unique home directory for its files, which is randomly assigned when the app is installed. If a third-party app, needs to access information other than its own, it does so only by using services explicitly provided by our iOS. System files and resources are also excluded from user's apps. The majority of iOS, as well as the third party apps, run as a non-privileged user mobile. The entire iOS petition is mounted as read only. Unnecessary tools, such as remote locking services, are not included in the system software. The iOS APIs, do not allow apps to escalate their own privileges to modify the apps or iOS itself. To summarize, by sandboxing a third party app, each app has its unique directory, and can access its own files, any access to other files will be restricted.

Address Space Layout Randomization

- Stack, heap, main executable, and dynamic libraries.



Memory Layout

attempted Lib C attack attempts to trick a device into executing malicious code, by manipulating memory addresses of the stack and system libraries. By randomizing the addresses of these system libraries and stacks, it will be hard for such an attack to succeed, because the addresses of these system libraries and stacks are very hard to guess.



Apple Security Quiz

Choose the best answer

What weaknesses were exploited by researchers in the Apple apps security in 2015?

- ☐ The malware was uploadable to the Apple Apps store.
- ☐ The malware was able to bypass Sandbox security
- ☐ The malware was able to hijack browser extensions and collect passwords.
- ☐ All of the above.

The correct answer is all of the above. The link in the instructor's notes point to a paper on this topic. The researchers were able to upload malware to the Apple Apps store, because the Apple Apps store review process is not bulletproof, which means that a malware can be disguised and pass the review process. And the Sandbox implementation has security bugs that can be exploited by the malware to bypass the Sandbox security. Similarly, browser is a piece of very complex software and also has security bugs. So again, the malware was able to hijack the browser and collect users' passwords.

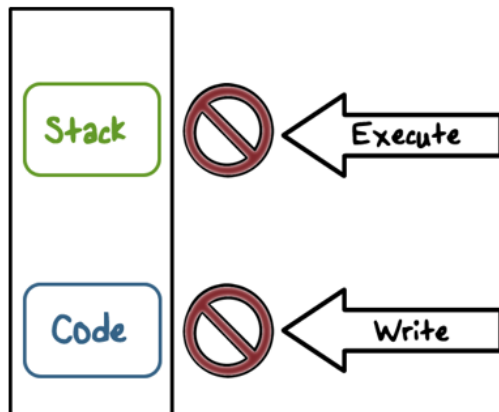
- ☐ The malware was uploadable to the Apple Apps store.
- ☐ The malware was able to bypass Sandbox security
- ☐ The malware was able to hijack browser extensions and collect passwords.
- ☒ All of the above.

iOS also has several other run time security measures. Address space layout randomization or ASLR protects against exploitation of memory corruption bugs. With ASLR, when an app is run, all memory regions are randomized. Randomly arranging the memory addresses of executable code system libraries, stacks, and heaps, and other related programming contracts, reduces the likelihood of many sophisticated exploits. For example, an

Again, any software can have security holes, therefore it is not surprising that despite all the efforts of security engineering, iOS still has security holes. What iOS security weaknesses were exploited by researches in the 2015? First, the malware was uploaded to the Apple App store. Second, the malware was able to bypass Sandbox security. Third, the malware was able to hijack browser extensions and collect passwords. Fourth, all the above.

Instructor Notes: Unauthorized Cross-App Resource Access on MAC OS X and iOS

Data Execution Prevention



Another runtime security measure that iOS uses is Data Execution Prevention. iOS uses the ARM processor's eXecute Never feature, which marks memory pages as nonexecutable. More specifically, iOS marks pages that are writable in runtime as nonexecutable. For example, memory pages that contain stack will not be executable. On the flip side, memory pages that are executable are marked as nonwritable. For example, the code pages can be executable, but they're not writable.

Data Execution Prevention is an implementation of the policy that makes writable and executable mutually exclusive. If a page is writable, meaning that at runtime, new data or code can be returned on that memory page, then this page is not executable. This prevents code-injection attacks because, to inject code, the attacker must write code into a memory page. But since we make writable and executable mutually exclusive, after the code can be returned in the memory page, that page can not be executable. Therefore, even if the attacker can write code into memory, he cannot force execution on that page. Therefore, he cannot execute his malicious code.

Data Execution Prevention



- **Stack and Heap** are not executable
- **W^X** policy enforced on code pages

Prevents code-injection attacks

Passcodes and Touch ID



- Touch ID provides **convenience**
- Passcode enables **data protection**
- **Maximum failed** attempts
- Progressive passcode **timeout**

To discourage brute force passcode attacks, the iOS interface enforces escalating time delays after the entry of an invalid passcode at the last screen. Users can choose to have the device automatically wiped if the passcode is entered incorrectly after ten consecutive attempts.

To prevent unauthorized use of a device, a user can use passcode or touch ID. Touch ID is the fingerprint sensing system that makes secure access to the device faster and easier. That is, it provides a degree of convenience. By setting up a device passcode, the user automatically enables data protection. iOS supports four digit and arbitrary length alphanumeric passcodes. To further



iOS Quiz

Mark all the true answers

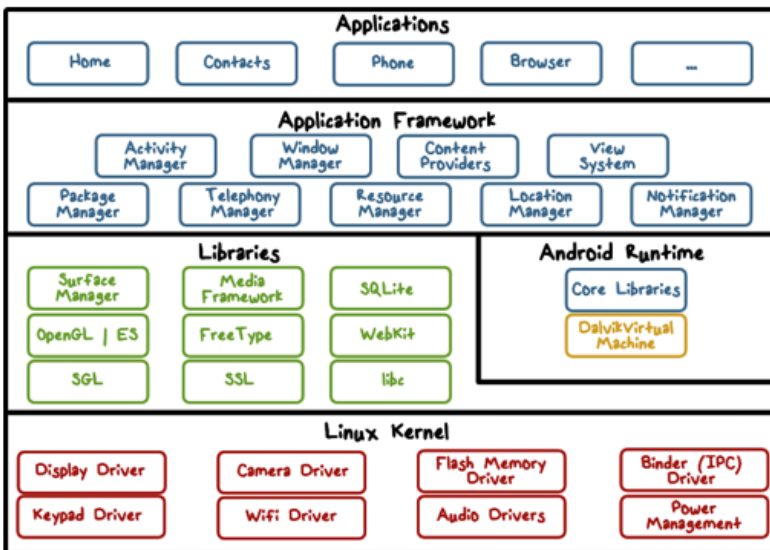
- ☐ Each app runs in a sandbox and has its own home directory for its files
- ☐ All iOS apps must be reviewed and approved by Apple
- ☐ iOS apps can be self-signed by app developers

Mark all answers there correct. First, each app runs in a sandbox and has its own home directory for its files. Second, all iOS apps must be reviewed and approved by Apple. Third, iOS apps can be self-signed by app developers.

First, each app runs in a sandbox and has its own home directory for its files. This is true. Second, all iOS apps must be reviewed and approved by Apple. This is true. Third, iOS apps can be self-signed by app developers. This is false. Self signing means that the app developers can issue its own certificate, and use the public key in a certificate to sign its own apps, and this is clearly false.

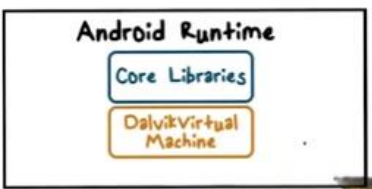
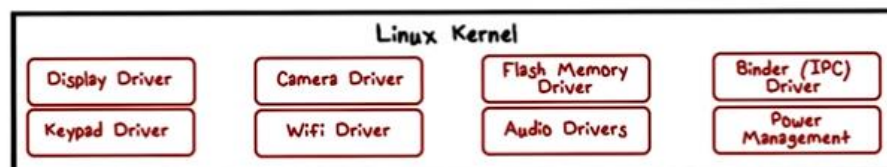
- ☒ Each app runs in a sandbox and has its own home directory for its files
- ☒ All iOS apps must be reviewed and approved by Apple
- ☐ iOS apps can be self-signed by app developers

Android Security Overview



Now let's take a look at Android security. Here's an overview of the architecture of Android which is based on Linux. Android is implemented in the form of a software stack architecture consisting of a Linux kernel, a runtime environment and corresponding libraries, an application framework and a set of applications.

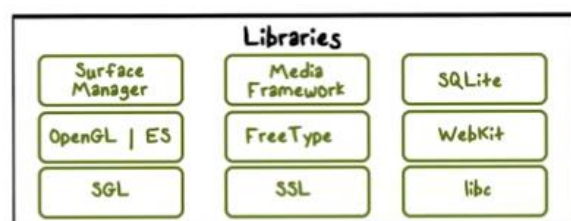
At the lowest level is the Linux Kernel. It provides a level of abstraction within device hardware and the upper layers of the Android software stack.

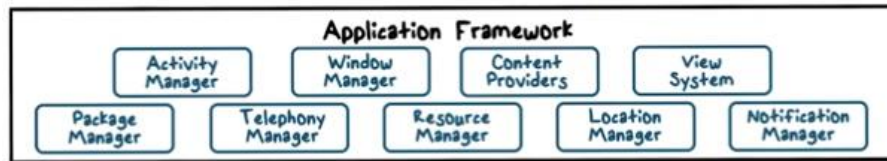


Each application running on the Android device runs its own instance of the Dalvik virtual machine. Apps are commonly written in Java, and compiled to bytecode of the Java virtual machine, which is then translated to Dalvik bytecode. The Dalvik executable format is very compact and it is designed for systems such as smart phones that are constrained in terms of memory and processor speed. By using

Dalvik, Android can achieve performance and efficiency. The Android core libraries are Java based libraries for application development. For example, for web browsing, data access and database queries, graphics rendering, and so on.

The Android call libraries do not actually perform much of the actual work. And are in fact, essentially Java wrappers around a set of C and C++ based libraries. These C and C++ libraries are included to fulfill a wide range of functions, including 2D or 3D graphics rendering, secure second layer, etc.





The application framework is a set of services that collectively, from the environment, for Android apps to run. The

application framework allows apps to be constructed using usable, interchangeable, and replaceable components. Furthermore, an app can publish its capabilities along with any corresponding data, so that they can be found and reused by other apps.

At the top of the Android software stack are the apps. These include apps that come



with the device such as home, contacts, and phone and browser. A lot of third party apps that the user has downloaded after he purchases the device.

Application Sandbox



- Each application runs with its UID in its own Dalvik virtual machine
 - Provides CPU protection, memory protection
- Applications announces permission requirement
 - Create a **whitelist model** – user grants access – Ask user at install time
 - Inter-component communication reference monitor checks permissions

Running apps in virtual machines means that they're essentially sandboxed in runtime. This means that the apps cannot directly interfere with the operating system and other apps. Nor can they directly access the device hardware. Each app is granted a set of permissions at install time, and cannot perform operations that require permissions it does not have. More specifically, the Android platform takes advantage of the Linux

user-based protection as a means of identifying and isolating application resources. The Android system assigns a unique user ID, or UID, to each Android app and runs it as that user in a separate process. The kernel enforces security between apps and the system at the process level through standard Linux facilities such as user and group IDs that are assigned to the apps. By default, apps cannot interact with each other and apps have limited access to the operating system. If app A tries to do something malicious like reading app B's data or downloading the form without permission, then the operating system protects against this, because app A does not have the appropriate user privileges. This sandbox model is simple, but on the other hand, it is based on decades of Unix style user separation of processes and file permissions. An app can announce the permissions that it needs, and user approval is required at install time. The permissions are typically, back up. The permissions are typically implemented by mapping them to Linux groups that have the necessary read-write access to relevant system resources, such as files and sockets. And therefore, they are ultimately enforced by the Linux kernel.



Android Sandbox vs iOS Sandbox

Android	iOS
App announces permission requirement	Apps have same permissions
Installation-time approval	First-usage time approval
App may have more powerful permissions	Limited permissions



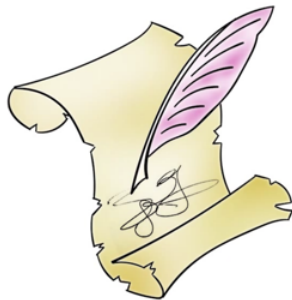
The Android Sandbox and the iOS Sandbox have different implementations. But from a security point of view, one of the main differences is how they handle permissions.

For Android, an app can announce the permissions they requires and the users can approve these permissions and install time. The

apps can ask for very powerful permissions.

For iOS, all the apps have the same set of basic permissions. If an app needs to access system resources or data, such as the users address book, user approval is required at the first time. In general, iOS apps have only limited permissions.

Code Signing



- All apps **self-signed by developers**
- **Code signing is used for**
 - Facilitating **application upgrades**
 - **Code/data sharing** between applications
 - Lets apps run in the same process

Another major difference between Android security and iOS security is code signing.

Android takes a very different approach from iOS. In particular, all apps are self-signed by developers. A developer can create his own power key, self sign it to create a certificate for its public key and then use that key to sign his apps. In other words, third party Android apps

are not signed by a central authority. There's no vetting process for third party app developers. This means that anybody can become an Android app developer, self-sign its apps and upload it to Android Play Store.

So, if code signing is not used to make sure that apps are from the vetted or verified developer, what is it used for? Code signing is used for making sure updates for an app are coming from the same developer. In addition, code signing is used to manage the trust relationship between apps, so that they can share code and data.



Android Apps Quiz

Mark all the true answers



- ☐ Android apps can be self-signed
- ☐ Android apps can have more powerful permissions than iOS apps

First, Android apps can be self-signed. This is correct.
Second, Android apps can have more powerful permissions than iOS apps. This is also true.

- ☒ Android apps can be self-signed
- ☒ Android apps can have more powerful permissions than iOS apps

Wireless and Mobile Security Lesson Summary

- Use WPA2 for WiFi security
- iOS has cryptographic keys and modules built into its device hardware, uses mandatory code signing and a very restricted app distribution model, and runs app in a sandbox with run-time protection such as ASLR and DEP
- Android is based on Linux and the sandbox model is based on Unix-style user separation, and its apps are self-signed

signed.

For Wi-Fi security, we should use the WPA2 standard. iOS has these cryptographic keys and modules built into its device hardware. It uses mandatory code signing and very restricted app distribution model. It runs an app in a sandbox with run-time protection such as ASLR and DEP. Android is based on Linux and sandbox is based on Unix style user separation. These apps are self