Developing Secure iOS Applications

YOW! Connected 2014

Michael Gianarakis

About me

@mgianarakis

Practice Manager at Securus Global

Application security for most of the last decade

Good guy that thinks like a bad guy

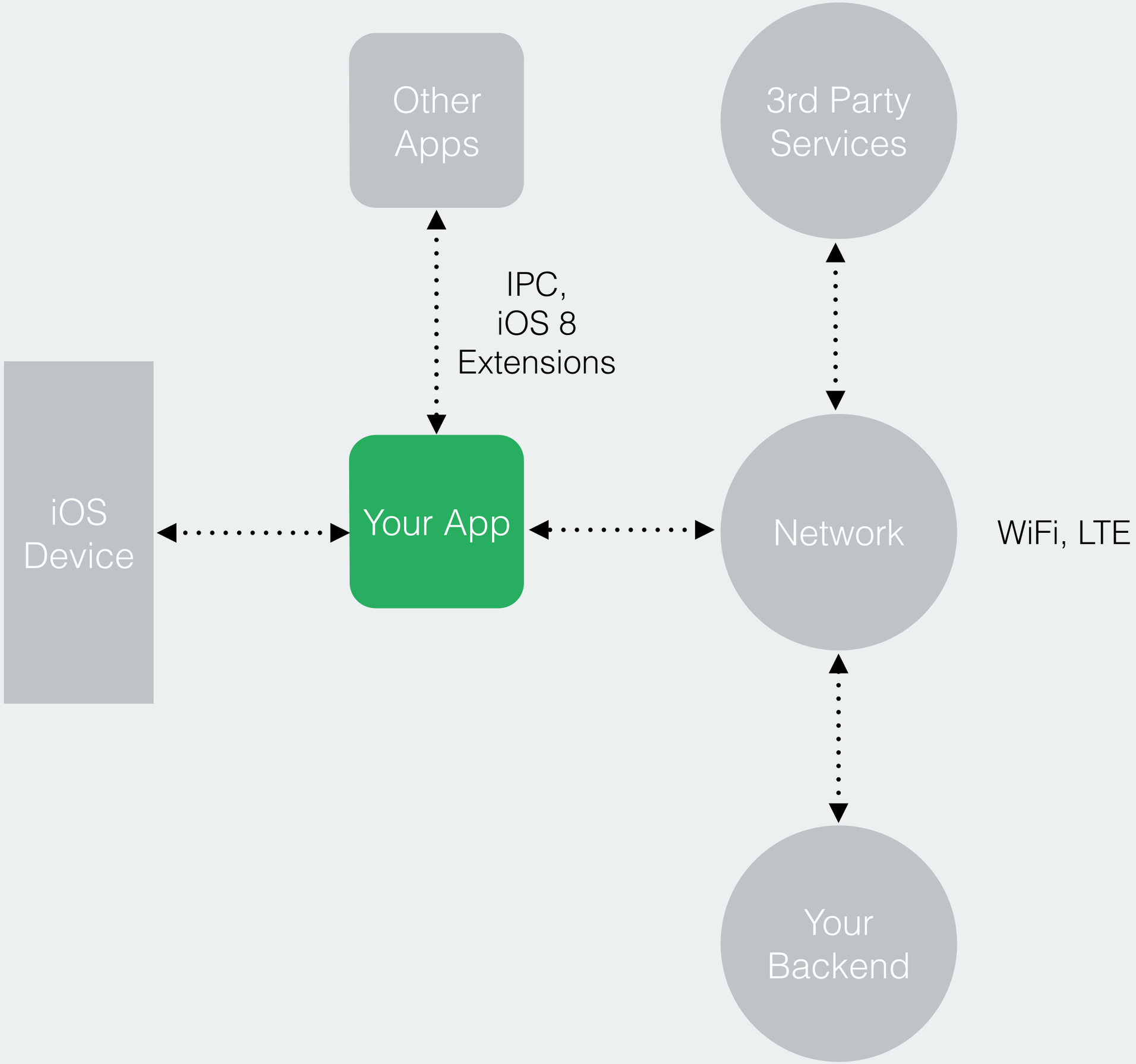
This presentation

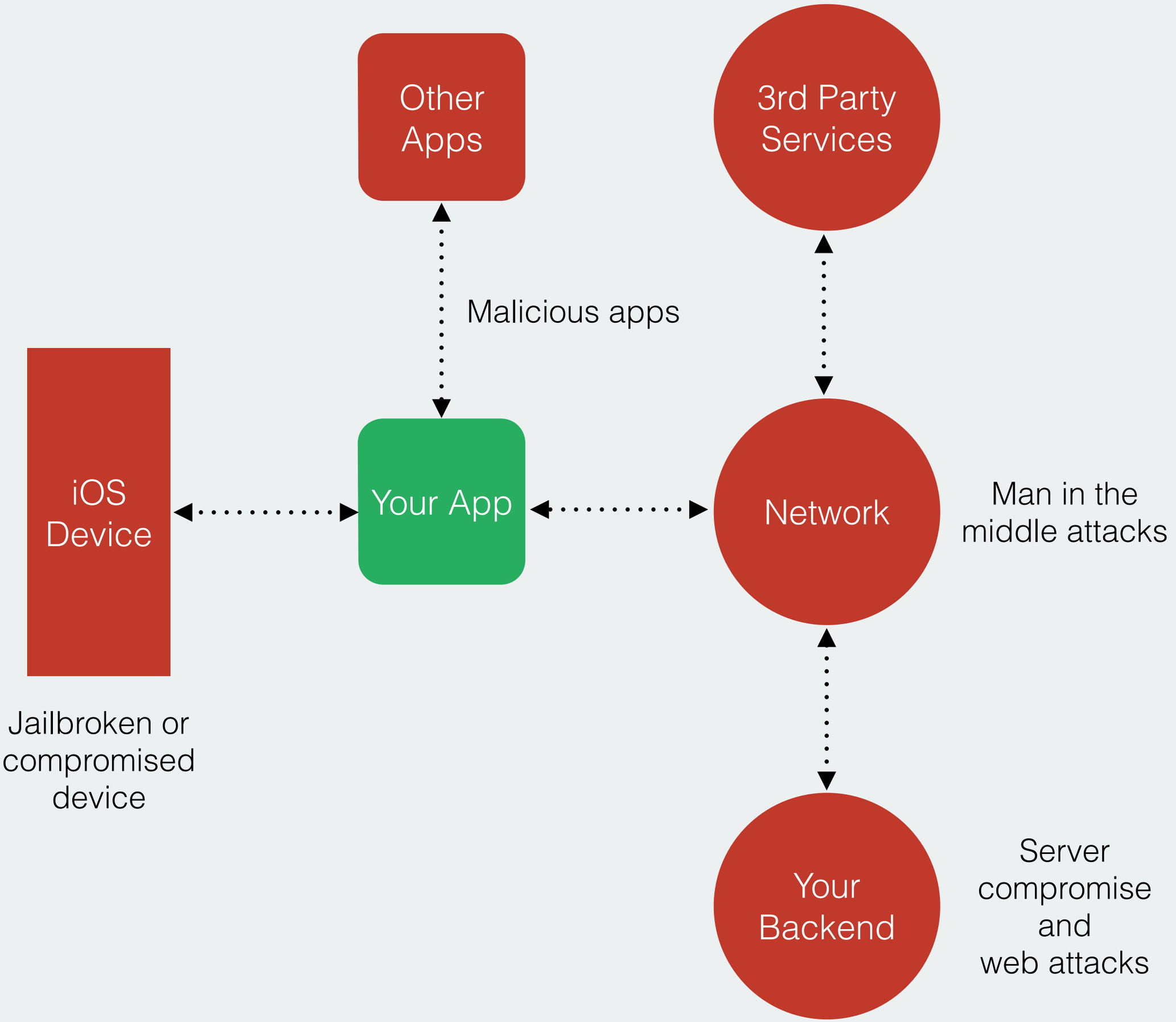
- Aims to give a good overview of how to design secure iOS applications
- Focuses on the key security issues commonly found in iOS applications
- Not a guide to hacking iOS applications

Topics

- Overview of the iOS attack surface
- Common iOS security issues
- Secure iOS application design principles
- Secure development approaches for key security iOS issues

iOS Application Attack Surface


Other Apps
3rd Party Services
IPC, iOS 8 Extensions
iOS Device
Your App
Network
WiFi, LTE
Your Backend


Other Apps
3rd Party Services
Malicious apps
iOS Device
Your App
Network
Man in the middle attacks
Jailbroken or compromised device
Your Backend
Server compromise and web attacks

Understanding the risk profile of your app

- Important first step that is often overlooked.
- Drives the security design of your application
- Key considerations:
  - What information is in the app and how sensitive/valuable is it?
  - Who are the likely threat actors? How capable are they?
  - What are the likely attack scenarios for each threat actor?
  - What is your tolerance for risk?
  - Regulatory requirements?

Common iOS Application Security Issues

Common Issues

- Based on working with a range of companies and a variety of applications including:
    - Mobile banking and trading apps
    - Board paper apps
    - Internal business apps
    - Retail apps
    - Social networking apps
    - Games

Common Issues

- Binary and Runtime Security Issues
- Transport Layer Security Issues
- Data Security Issues

Get these right and you are well and truly ahead of the pack

Secure iOS Design Principles

Don't trust the client or the runtime environment

Users will connect to untrusted networks

Don’t store anything sensitive on the device

Binary and Runtime Security

Binary and Runtime Security

- As a reflective language, Objective-C can observe and modify it's own behaviour at runtime.
- Can be great from a development perspective
- Not so much from a security perspective

Binary and Runtime Security

- The ability to create and call ad hoc classes and methods on the fly allows attackers to manipulate and abuse the runtime of the application:
    - Bypass security locks
    - Break logic checks
    - Escalate privilege
    - Steal information from memory.
- Implementing runtime security checks significantly increases the security of your application
- You should always assume that your application will run on a compromised device

Aside - Compromised Devices

Myths surrounding jailbreaking

- Generally speaking, iOS is one of the most secure operating systems available

- One of the defining security features of iOS is the chain of trust. Every bit of code is signed and approved by Apple.

- Break that trust and all bets are off

Myths surrounding jailbreaking

- Some of the security issues discussed in this presentation will require a compromised device

- This is usually the cause of a lot of misunderstanding when it comes to assessing the impact of these issue

Myth 1 - Public jailbreaks are the only jailbreaks

- I often hear "but there is no jailbreak for version x.x.x so we are safe"

- **FACT:** Public jailbreaks for every version of iOS released so far

- **FACT:** Not all users update to the latest firmware

- **FACT:** iOS vulnerabilities are extremely valuable, there are many more exploits than just those that people give out for free on the internet

Myth 2 - Jailbreaks require physical access to the device

- I often hear "well someone would need to steal the device to compromise it"

- **FACT:** Not all exploits require physical access.

    - jailbreakme.com TIF image processing vulnerability and PDF processing vulnerability required only a user visit a web page in Safari

    - Remote code execution via SMS found by Charlie Miller (https://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf)

    - Nitro JIT vulnerability to download and execute unsigned code via POC app that was approved also found by Charlie Miller (http://reverse.put.as/wp-content/uploads/2011/06/syscan11_breaking_ios_code_signing.pdf)

Myth 3 - Jailbreaks are always obvious to the user

- It is a commonly held misconception that if Cydia is not on the device then it is not jailbroken.

- Cydia is an app that is commonly installed with public jailbreaks but is certainly not a prerequisite or in any way required for a jailbreak.

- In fact, any kind of targeted exploit is unlikely to install Cydia in the first place.

Myth 4 - Jailbreaking is Apple's problem

- **FACT:** The security of your app, your data and your users is your responsibility.

- **FACT:** The security risk to your organisation is your responsibility. You will be the one that suffers the impact if your app is compromised and you should be actively managing that risk.

- **FACT:** Apple won't secure your app for you.

Myth 5 - You can completely protect against jailbreaking

- **FACT:** Unfortunately you will never be able to completely secure an application.

- The idea is to raise the bar for the security of your app so that it becomes too difficult or costly to attack.

</aside>

Debug Check

- A simple and effective method of securing the runtime is to implement anti-debugging control.
- There may be reasons to allow an application to run on a jailbroken device but there are no legitimate reasons for debugging a production application

Debug Check

- Debugging protection can be implemented using the following techniques:
    - Periodically monitoring the process state flag to determine if application process is being debugged and then terminating the process.
    - Using the ptrace request `PT_DENY_ATTACH` two prevent future tracing of the process. This can be done with the following function call from within the application:
        - `ptrace(31,0,0,0)`
- These techniques should be used in conjunction with each other to provide a more robust anti-debugging control.

Jailbreak Detection

- There are multiple ways to implement jailbreak detection but the two most common methods are:
  - File system check
  - Sandbox integrity check

Jailbreak Detection

- File system checks test for the presence of files that are typically found on jailbroken devices.

- Some examples of files to look for:
  - `/Library/MobileSubstrate/MobileSubstrate.dylib`
  - `/var/cache/apt`
  - `/var/lib/apt`
  - `/bin/bash and /bin/sh`
  - `/usr/sbin/sshd and /etc/ssh/sshd_config`

Jailbreak Detection

- When implementing a file system check you should ensure you check for multiple files using different methods such as `fileExistsAtPath` and `isReadableFileAtPath`

Jailbreak Detection

- A sandbox integrity check tests that the integrity of Apple's sandbox is intact on the device, and that the application is running inside it.

- A sandbox integrity check works by attempting to perform an operation that would not typically succeed on a device that has not been jailbroken.

- A common test is to attempt to use the fork function to spawn a new child process which will succeed if the sandbox has been compromised or if the application is running outside of the sandbox

Jailbreak Detection

- Check out the IMAS project's Security Check module https://github.com/project-imas/security-check

Inline Functions

- Any sensitive functions such as the security checks should be compiled as inline functions

- Inlining functions complicates attacks that involve editing memory and patching the application by forcing an attacker to find and patch all occurrences of the code.

- Specify a function as inline using the `__attribute__((always_inline))` compiler directive.

Inline Functions

- This directive may not always be honoured however. To minimise the chance that this will happen configure the following:
    - Set the `-finline-limit` compiler flag high enough to allow for inlining of your code (e.g. 5000)
    - Set the optimisation level at least `-O1`
    - Don't use `-unit-at-a-time` (note this is automatically set at `-O3` and above
    - Don't use `-keep-inline-functions` if your functions are static

Address Space Validation

- Any time malicious code is injected into your application it must be loaded into an address space
- Validating the address space adds complexity to attacks as it forces an attacker to inject the malicious code into the existing address space with the valid code or attacking dynamic linker

Address Space Validation

- The `dladdr` function in the dynamic linker library returns information about the address space a particular function belongs to.

- Passing it the function pointer of a class's method implementation can determine whether it came from your app, Apple's frameworks, or an unknown/malicious source

Avoid Simple Logic

- Simple logic checks for security functions and business logic are susceptible to to manipulation.
- Examples include:
    - `- (BOOL) isLoggedIn`
    - `BOOL isJailbroken = YES`
    - `- (void) setDebugFlag`

Avoid Simple Logic

```
cy# [[redacted]JailbreakDetector.messages['hasExistenceOfJailbreakFiles'] = function() {return false;}
function () {return false;}
cy# [[redacted]JailbreakDetector.messages['hasEvidenceOfSymbolicLinking'] = function() {return false;}
function () {return false;}
cy# [[redacted]JailbreakDetector.messages['hasSandboxIntegrityBeenCompromised'] = function() {return false;}
function () {return false;}
cy# [jb hasExistenceOfJailbreakFiles]
0
cy# [jb hasEvidenceOfSymbolicLinking]
0
cy# [jb hasSandboxIntegrityBeenCompromised]
0
```

```
cy# [[redacted]JailbreakDetector sharedInstance].jailbreakSymbolicLinkPaths = @[]
@[]
cy# [[redacted]JailbreakDetector sharedInstance].jailbreakFilePaths =@[]
@[]
```

Securing Memory

- Instance variables stored within Objective-C objects can be easily mapped inside the Objective-C runtime.

- Never store anything in memory until the user has authenticated and data has been decrypted.

- Manually allocate memory for sensitive data rather than storing the data (or pointers to this data) inside Objective-C instance variables.

- Wipe sensitive data from memory when it's not needed.

Securing Memory

```
iPhone:~ root# ps aux | grep [redacted]
mobile    2722   2.2 29.6   555048 153188   ??  Ss    3:05PM   0:17.06 /var/mobile/Applications/188E2BF8-81B5-4781-98B9-AAA92932B0
6E/[redacted]
root      2732   0.0  0.0   273928      0 s003  R+    3:11PM   0:00.00 grep [redacted]
iPhone:~ root# cycript -p 2722
cy# UIApp.delegate.logonPIN
@"2580"
cy# UIApp.delegate.deviceToken
@"jdTbSh6l8Y"
cy# UIApp.delegate.AESKey
@"<58b3cffb 73e24bcf ae9310d8 8dc239a6 c11f460a ecae45ce b540a3cc 3e647b5f>"
cy# UIApp.delegate.mobileNumber
@"0432384204"
cy# UIApp.delegate.customerName
@"[redacted]"
cy# [AccountCache sharedInstance]
@"<AccountCache: 0x1e94cd20>"
cy# accountTest = new Instance(0x1e94cd20)
@"<AccountCache: 0x1e94cd20>"
cy# accountTest.accountList
cy# accountTest.primaryAccount
null
```

```
cy# *account_test
{isa:@"[redacted]ccount",alerts:null,usage:@"1111110",permissionBitMask:327743,accountName:@"[redacted]",displayAccountName:@"[redacted]
[redacted]t",accountNumber:@"[redacted]-451732569",displayAccountNumber:@"[redacted]451732569",accountType:@"[redacted]",defaultBalance:@"<Money: 0x1d972d30>",otherBa
lance:@"<Money: 0x1d972f50>",everydayAccount:0,creditCardAccount:0,vali[redacted]nt:0,accountUserProperty:@"<AccountUserProperty: 0x1d972d00>",subA
ccountType:@"I3",originalDisplayAccountName:null,preferredAccount:1}
```

Transport Layer Security

Transport Layer Security

- Users will connect their devices to untrusted networks
- If your app handles sensitive data you should be encrypting all traffic between the app and the back end.

Protocol Version

- If using `NSURLSession` you can specify the minimum supported TLS protocol version by setting the `TLSMinimumSupportedProtocol` property in the `NSURLSessionConfiguration`

Cipher Suites

- You should disable support for low and medium strength cipher suites on the server

- Medium:

```
SSLv2
DES-CBC-MD5 Kx=RSA Au=RSA Enc=DES(56) Mac=MD5
SSLv3
DES-CBC-SHA Kx=RSA Au=RSA Enc=DES(56) Mac=SHA1
TLSv1
EXP1024-DES-CBC-SHA Kx=RSA(1024) Au=RSA Enc=DES(56) Mac=SHA1 export
EXP1024-RC4-SHA Kx=RSA(1024) Au=RSA Enc=RC4(56) Mac=SHA1 export
DES-CBC-SHA Kx=RSA Au=RSA Enc=DES(56) Mac=SHA1
```

- Low:

```
SSLv2
EXP-RC2-CBC-MD5 Kx=RSA(512) Au=RSA Enc=RC2(40) Mac=MD5 export
EXP-RC4-MD5 Kx=RSA(512) Au=RSA Enc=RC4(40) Mac=MD5 export
SSLv3
EXP-RC2-CBC-MD5 Kx=RSA(512) Au=RSA Enc=RC2(40) Mac=MD5 export
EXP-RC4-MD5 Kx=RSA(512) Au=RSA Enc=RC4(40) Mac=MD5 export
TLSv1
EXP-RC2-CBC-MD5 Kx=RSA(512) Au=RSA Enc=RC2(40) Mac=MD5 export
EXP-RC4-MD5 Kx=RSA(512) Au=RSA Enc=RC4(40) Mac=MD5 export
```

Certificate Validation

- Always perform certificate validation checks
- When using Apple frameworks iOS will accept a certificate if it's signed by a CA in the trust store
- Certificate validation checks will often be overridden in development to avoid issues with self-signed certs.
- Typically this is encapsulated in a variable somewhere or in some other simple logic that can be subverted.

Certificate Validation

```
8798  @interface [redacted]AppConfiguration : XXUnknownSuperclass <[redacted]INCMSHTTPClientConfiguration> {
8799  @private
8800    BOOL _allowInsecureCookieTransport;
8801    NSURL* _companyLogoEndpointURL;
8802    NSURL* _[redacted]UserServicesEndpointURL;
8803    NSURL* _contentServerEndpointURL;
8804    NSURL* _refreshScopeEndpointURL;
8805    NSString* _secureContentBasePath;
8806    NSURL* _loginEndpointURL;
8807    NSURL* _getFileEndpointURL;
8808    NSURL* _[redacted]EndpointURL;
8809    NSURL* _[redacted]EndpointURL;
8810    BOOL _allowUntrustedSSLCertificates;
8811  }
8812  @property(retain, nonatomic) NSString* secureContentBasePath; // G=0x1f753d; S=0x1f7571;
8813  @property(retain, nonatomic) NSURL* contentServerEndpointURL; // G=0x1f7455; S=0x1f7489;
8814  @property(retain, nonatomic) NSURL* [redacted]erServicesEndpointURL;  // G=0x1f73e1; S=0x1f7415;
8815  @property(retain, nonatomic) NSURL* companyLogoEndpointURL; // G=0x1f736d; S=0x1f73a1;
8816  @property(assign, nonatomic) BOOL allowInsecureCookieTransport; // G=0x1f7301; S=0x1f7335;
8817  @property(assign, nonatomic) BOOL allowUntrustedSSLCertificates;  // G=0x1f7781; S=0x1f77b5;
8818  @property(retain, nonatomic) NSURL* getFileEndpointURL; // G=0x1f7625; S=0x1f7659;
8819  @property(retain, nonatomic) NSURL* refreshScopeEndpointURL;  // G=0x1f74c9; S=0x1f74fd;
8820  @property(retain, nonatomic) NSURL* loginEndpointURL; // G=0x1f75b1; S=0x1f75e5;
8821  @property(retain, nonatomic) NSURL* [redacted]EndpointURL; // G=0x1f7699; S=0x1f76cd;
8822  @property(retain, nonatomic) NSURL* [redacted]EndpointURL; // G=0x1f770d; S=0x1f7741;
8823  +(id)defaultConfiguration;  // 0x1f6949
8824  -(void).cxx_destruct; // 0x1f77ed
8825  -(id)debugDescription;  // 0x1f6d41
8826  -(BOOL)allowsInsecureTransport; // 0x2da09
8827  -(BOOL)allowsUntrustedCertificates; // 0x2d9cd
8828  -(id)secureCookiePath;  // 0x2d995
8829  -(id)baseURL; // 0x2d95d
8830  @end
```

Certificate Validation

- To avoid having to do this, use valid certs in development (e.g. free certs)
- If you do need to disable validation, ensure that the production builds essentially “hard code” validation.
- Consider certificate pinning.

Certificate Pinning

- Mobile apps are great candidates for certificate pinning as they only need to communicate with a small, clearly defined set of servers

- Certificate pinning improves security by removing the need to trust the certificate authorities.

- iSEC partners have a great library for implementing certificate pinning https://github.com/iSECPartners/ssl-conservatory

Data Security

Two types of data security issues....

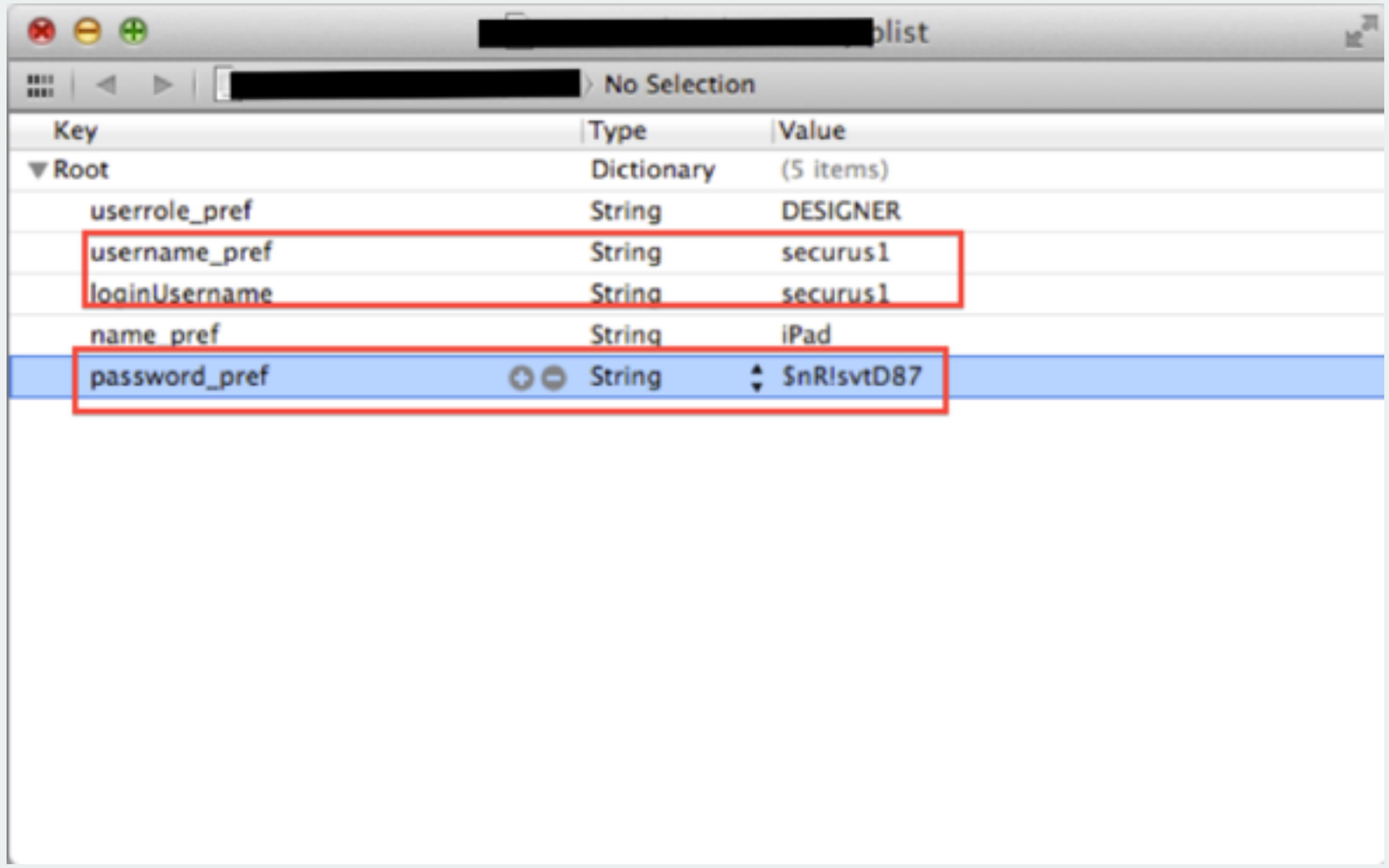
- Sensitive data stored on the device by the application that was not secured appropriately by the developer
- Unintended data leakage via system functionality

Insecure Data Storage

- It's common to find sensitive data stored insecurely on the device.
- As a general rule sensitive information should not be stored on the device at all.
- Really consider the need to store sensitive information on the device. Generally it is functionally and technically possible to not store the data at all.

Property List Files

- Don't store sensitive information using **`NSUserDefaults`**. Plist files are unencrypted and can easily be retrieved from the device and backups



SQLite Databases

- Avoid storing sensitive information in SQLite databases if at all possible.

- If you need to store sensitive information encrypt the database using SQLCipher https://github.com/sqlcipher/sqlcipher

- If you are using Core Data the Encrypted Core Data module of the IMAS project leverages SQLCipher to encrypt all data that is persisted https://github.com/project-imas/encrypted-core-data/

Data Protection APIs

- If you need to write files with sensitive information to the device (not credentials) at a minimum write files with an appropriate data protection attribute

- If you don't need to access the file in the background use `NSFileProtectionComplete` (for `NSFileManager`) or `NSDataWritingFileComplete` (for `NSData`).

- With this attribute set the file is encrypted on the file system and inaccessible when the device is locked.

Data Protection APIs

- If you need to access the file in the background use `NSFileProtectionCompleteUnlessOpen` for (`NSFileManager`) or `NSDataWritingFileProtectionCompleteUnlessOpen` (for `NSData`)

- With this attribute set the file is encrypted on the file system and inaccessible while closed.

- When a device is unlocked an app can maintain an open handle to the file even after it is subsequently locked, however during this time the file will not be encrypted.

Unintended Data Leakage

- Sensitive data captured and stored automatically by the operating system.
- Oftentimes developers are unaware that the OS is storing this information.

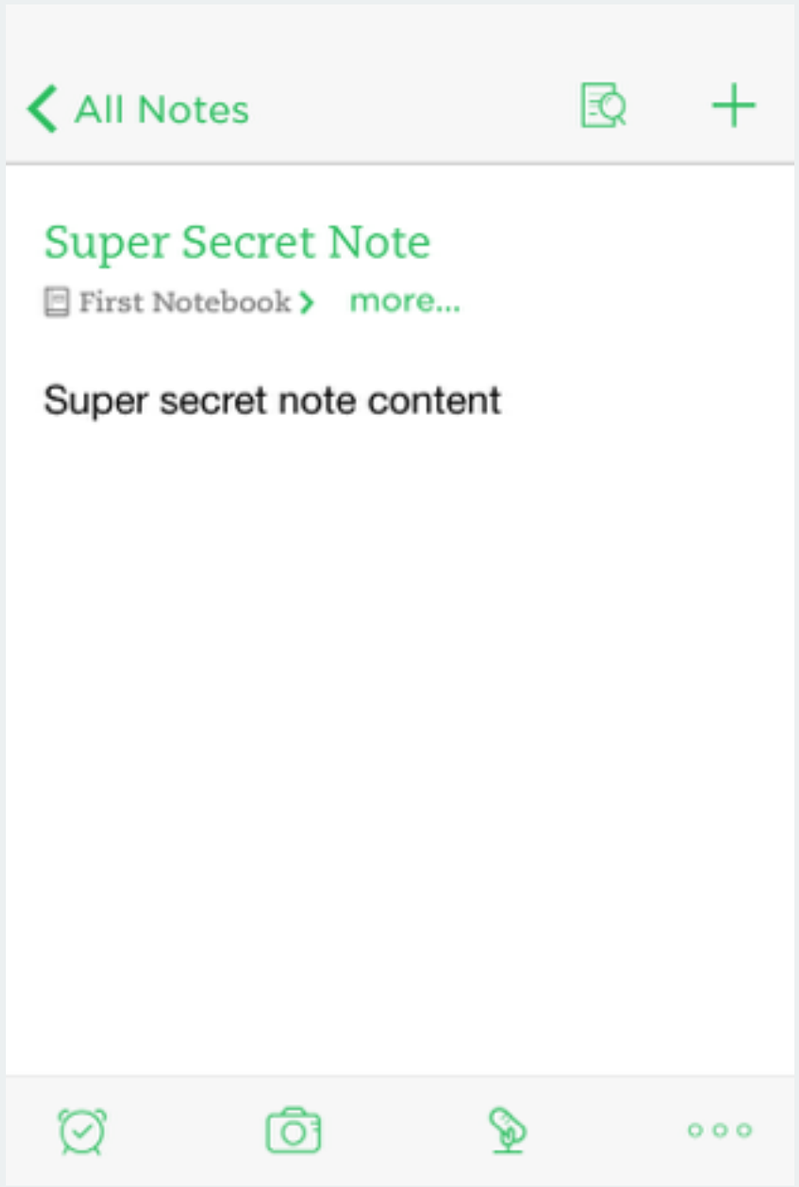
Background Screenshot Cache

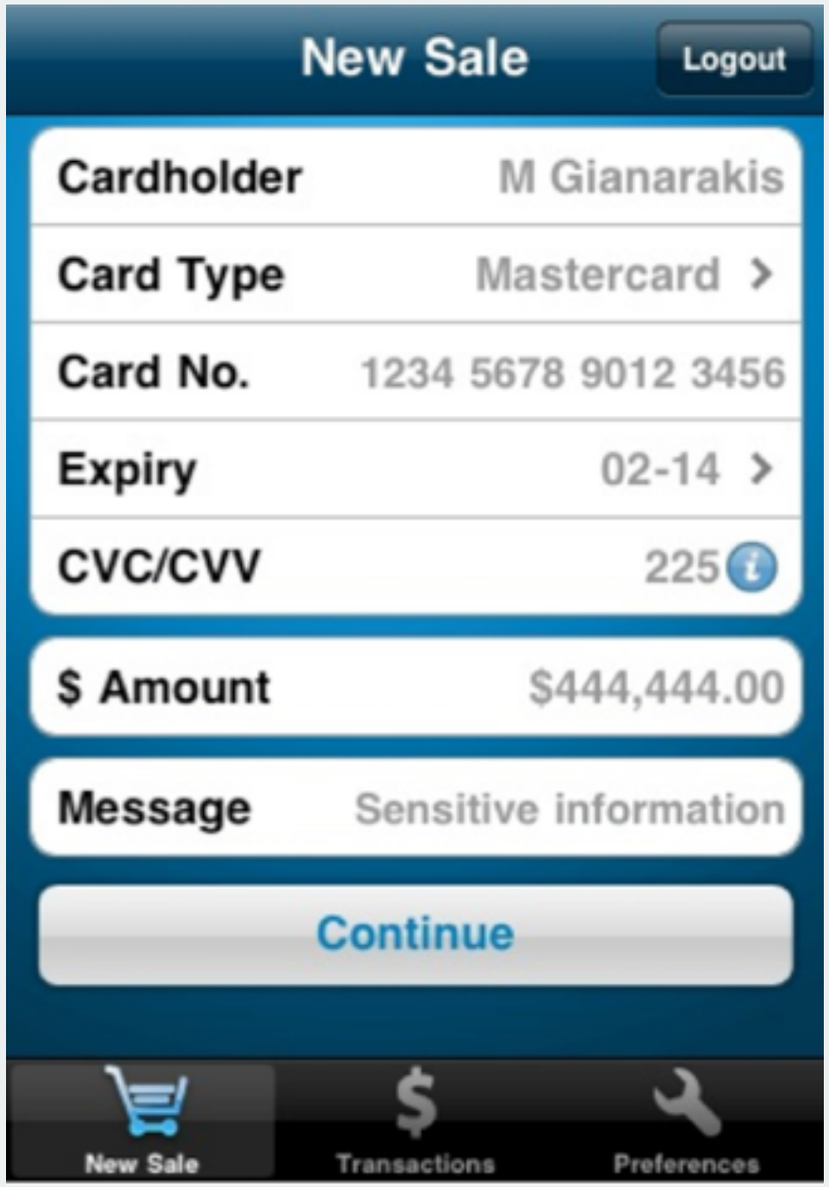
- Every time an application suspends into the background, a snapshot is taken to facilitate the launch animation.

- Stored unencrypted in the app container.

- This allows attackers to view the last thing a user was looking at when the application was backgrounded.

Background Screenshot Cache





Background Screenshot Cache

- Place content clearing code in the state transition methods `applicationDidEnterBackground` and `applicationWillResignActive`
  - The simplest way to clear the screen contents is to set the key window's hidden property to YES.

```
- (void)applicationDidEnterBackground:(UIApplication *)application
 {
     [ UIApplication sharedApplication ].keyWindow.hidden = YES;
 }
```

Background Screenshot Cache

- Be careful, if there are any other views behind the current view, these may become visible when the key window is hidden.

- Other methods:
    - Manually clearing sensitive info from the current view
    - Placing the launch image in the foreground

- Note: `ingoreSnapshotOnNextApplicationLaunch` will not prevent the screenshot from being taken

URL Caching

- iOS automatically caches requests and responses for protocols that support caching (e.g. HTTP/HTTPS) and stores them unencrypted in a SQLite database in the app container.

- Make sure to configure the server to send the appropriate cache control headers:

```
Cache-Control: no-cache, no-store
Expires: 0
Pragma: no-cache
```

URL Caching

- Can also control caching in the client.
- For NSURLConnection implement the connection:willCacheResponse: delegate method

```
- (NSCachedURLResponse *)connection:(NSURLConnection *)connection
                  willCacheResponse:(NSCachedURLResponse *)cachedResponse
{
    return nil;
}
```

- For `NSURLSession` set the `URLCache` property to `NULL`

Logging

- Less of an issue since iOS 7
- Pre-iOS 7 any app was able to view the device console and thus any sensitive data that was logged.
- Still a good idea to make sure nothing sensitive is logged.

Logging

- If you must log sensitive information in development for debugging purposes special care should be taken to ensure that this code is not pushed to production.

- One way to do this is to redefine `NSLog` with a pre-processor macro such as `#define NSLog(…)`

Pasteboard

- When the Cut or Copy buttons are tapped, a cached copy of the data stored on the device's clipboard - `/private/var/mobile/Library/Caches/com.apple.UIKit.pboard/pasteboard`.

- If the application handles sensitive data that may be copied to the pasteboard consider disabling Copy/Paste functionality for sensitive text fields (note this is done by default for password fields)

Pasteboard

- To disable pasteboard operations for sensitive fields subclass `UITextView` and override the `canPerformAction:withSender:` method to return `NO` for actions that you don't want to allow or just disable the menu completely

```
- (BOOL)canPerformAction:(SEL)action withSender:(ID)sender {
     UIMenuController *menuController = [UIMenuController sharedMenuController];
     if (menuController) {
          [UIMenuController sharedMenuController].menuVisible = NO;
     }
 return NO;
 }
```

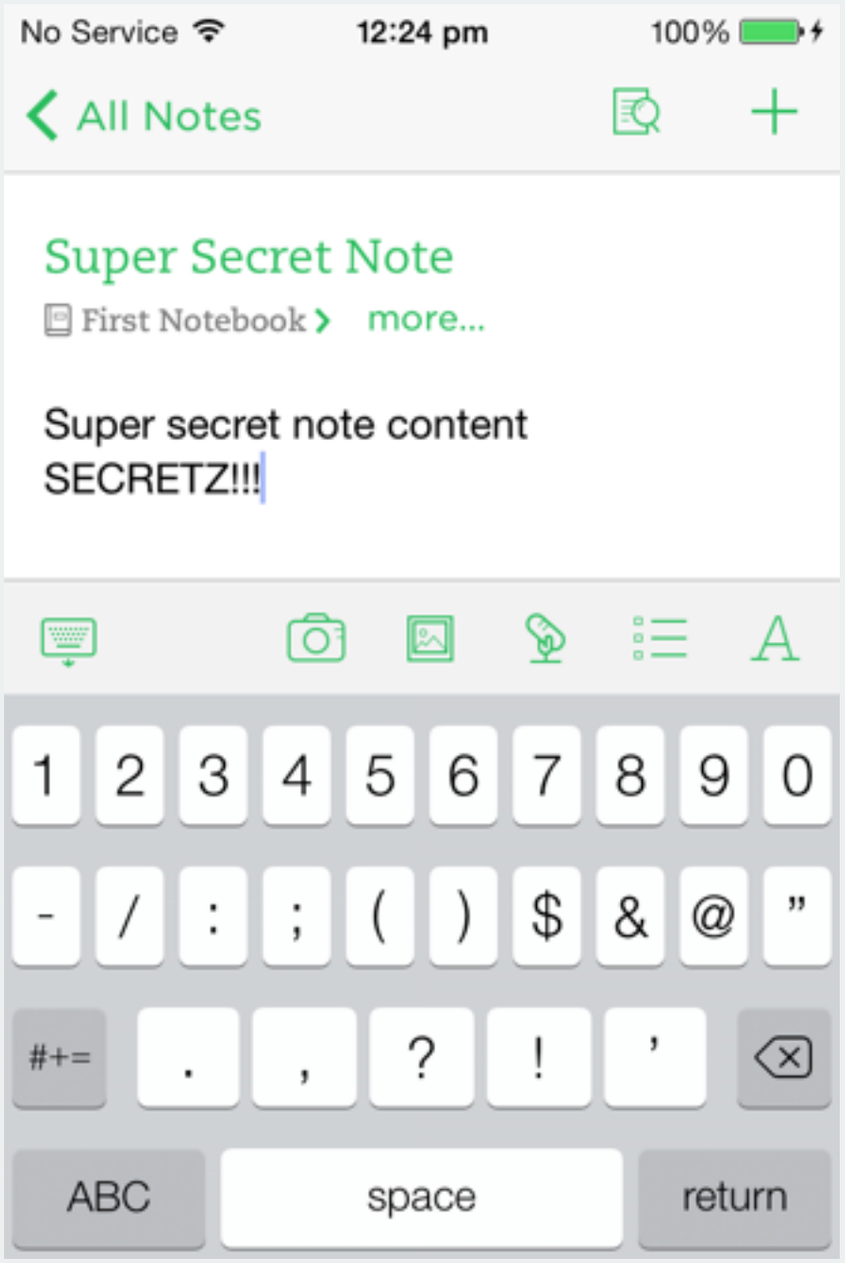
Autocorrect Cache

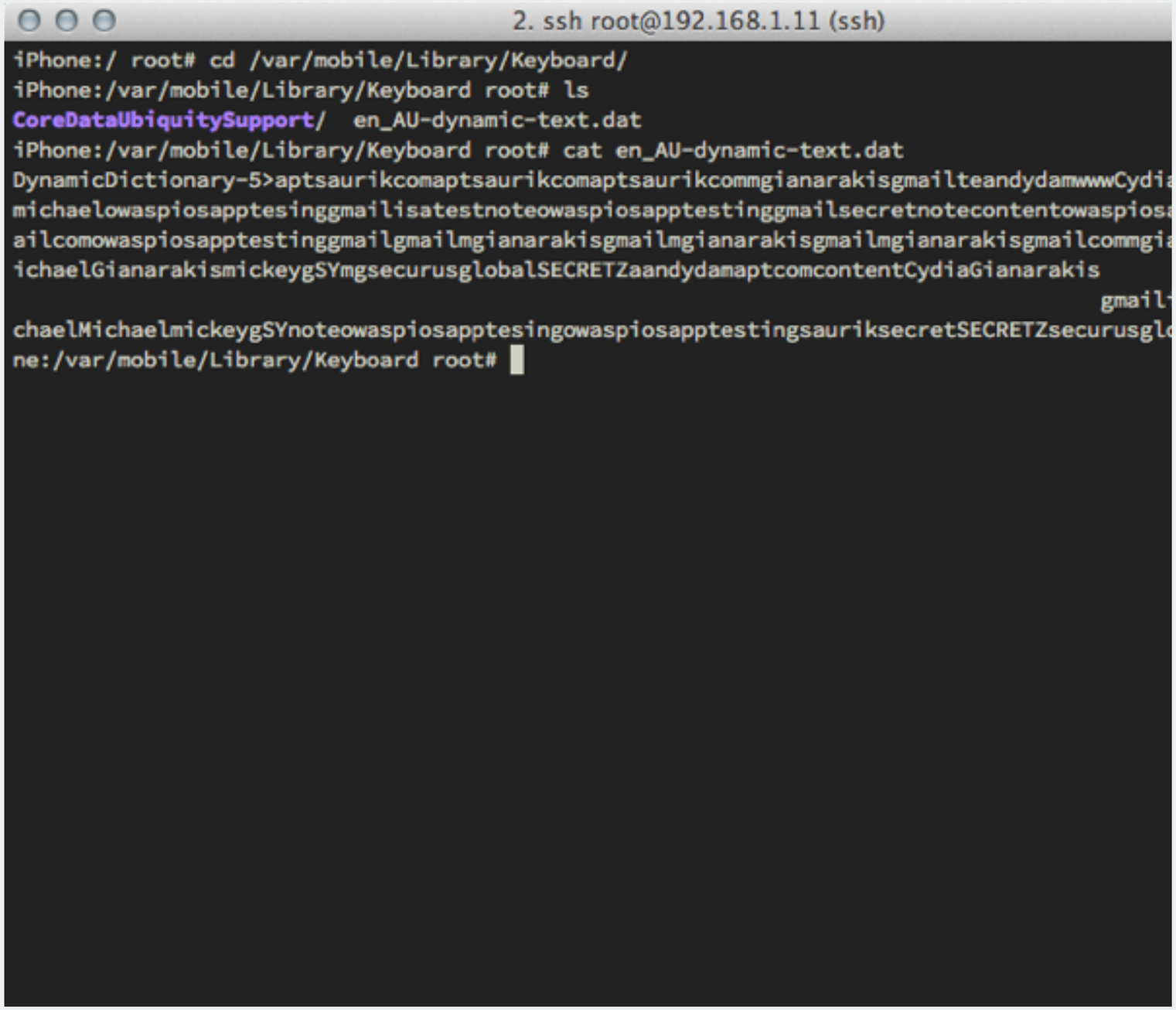
- iOS keeps a binary keyboard cache containing ordered phrases of text entered by the user – `/private/var/mobile/Library/Keyboard/dynamic-text.dat`

- To avoid writing data to this cache you should turn autocorrect off in text fields whose input should remain private.

- To turn autocorrect off, the developer should set `textField.autocorrectionType` to `UITextAutocorrectionTypeNo` for all sensitive text fields

Autocorrect Cache





Cryptography

- Getting cryptography right in mobile apps is difficult.
- The primary challenge is key management.
- Most apps fall prey to storing the key with the lock.
- Need to use a key derivation function.

Key Management

- Don't hard code an encryption key in the binary as it can easily be retrieved running strings on the binary.

- Storing an encryption key in the Keychain is not secure on compromised devices.

- On a compromised device encryption keys can also be retrieved from memory. You should manually allocate memory for encryption keys rather than storing them in Objective-C instance variables.

- Wipe keys from memory when not needed.

Key Management

- Use a key derivation function to derive a key to use to encrypt the master key.
- Should be based on a secret input such as the user's password.
- Don't just use a cryptographic hash of the key.
- Use a sensible salt such as the `identifierForVendor`

Insecure Algorithms

- MD5, SHA1, RC4, MD4 algorithms are weak and should not be used.

Custom Encryption Algorithms

- Don't do it.

www.securusglobal.com

@mgianarakis

eightbit.io