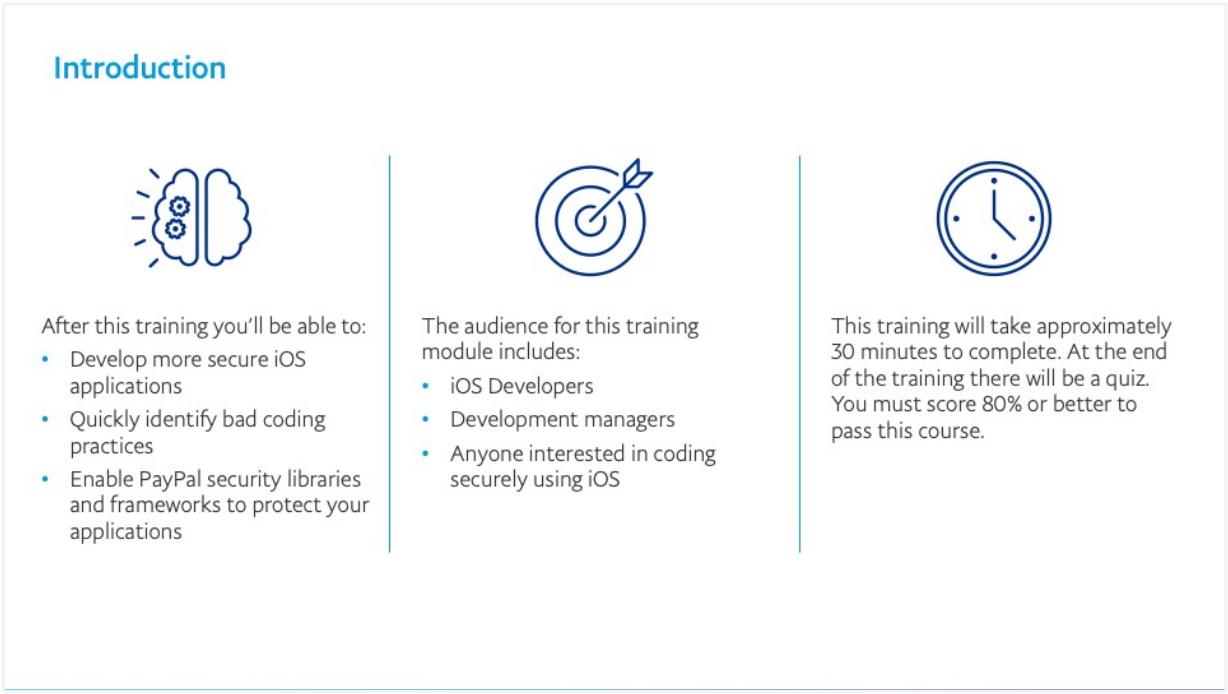
Welcome to iOS Security Training.



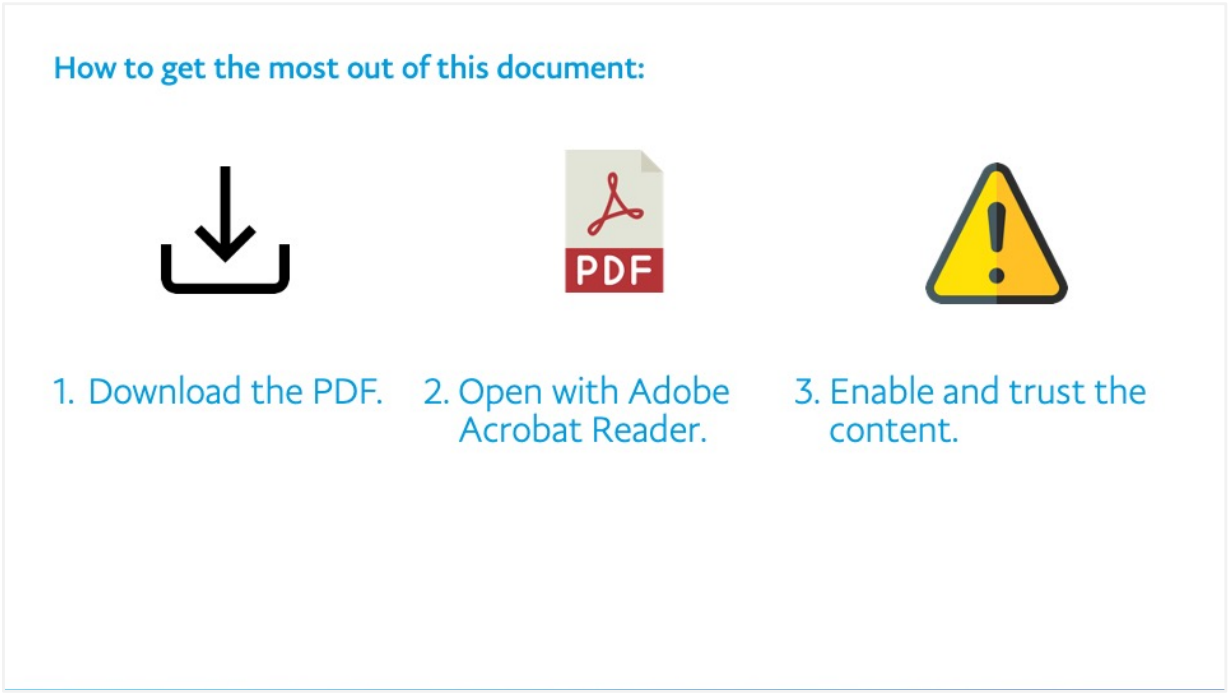
PayPal faces many challenges associated with information-related risks. Hackers, fraud, and denial-of-service attacks are all threats that must be dealt with daily. A key part of providing our customers with the best service possible is robust and reliable applications. Tools alone cannot create secure code, however. PayPal is committed to providing our developers with the best training available and has worked with our partner to create a comprehensive secure software development training series.

After this training you'll be able to:

- Enable security best practices to protect your application
- Identify and mitigate bad coding practices
- Understand common mobile application vulnerabilities
- Develop more secure Android applications

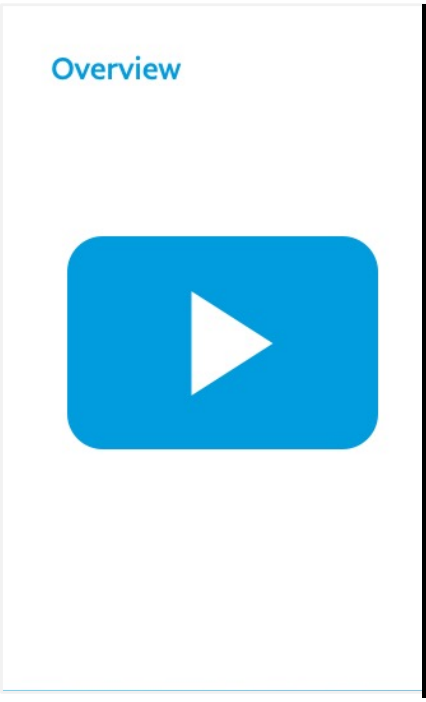
The audience for this training module includes:

- iOS developers
- Development managers
- Anyone interested in coding securely using iOS.



How to get the most out of this document:

1. Download the PDF. Viewing in browser limits the functionality.
2. Open the PDF with Adobe Acrobat Reader or similar program.
3. Enable and trust the content. To be able to play the video content in the PDF, you must enable the content. Choose “Trust this document always.”

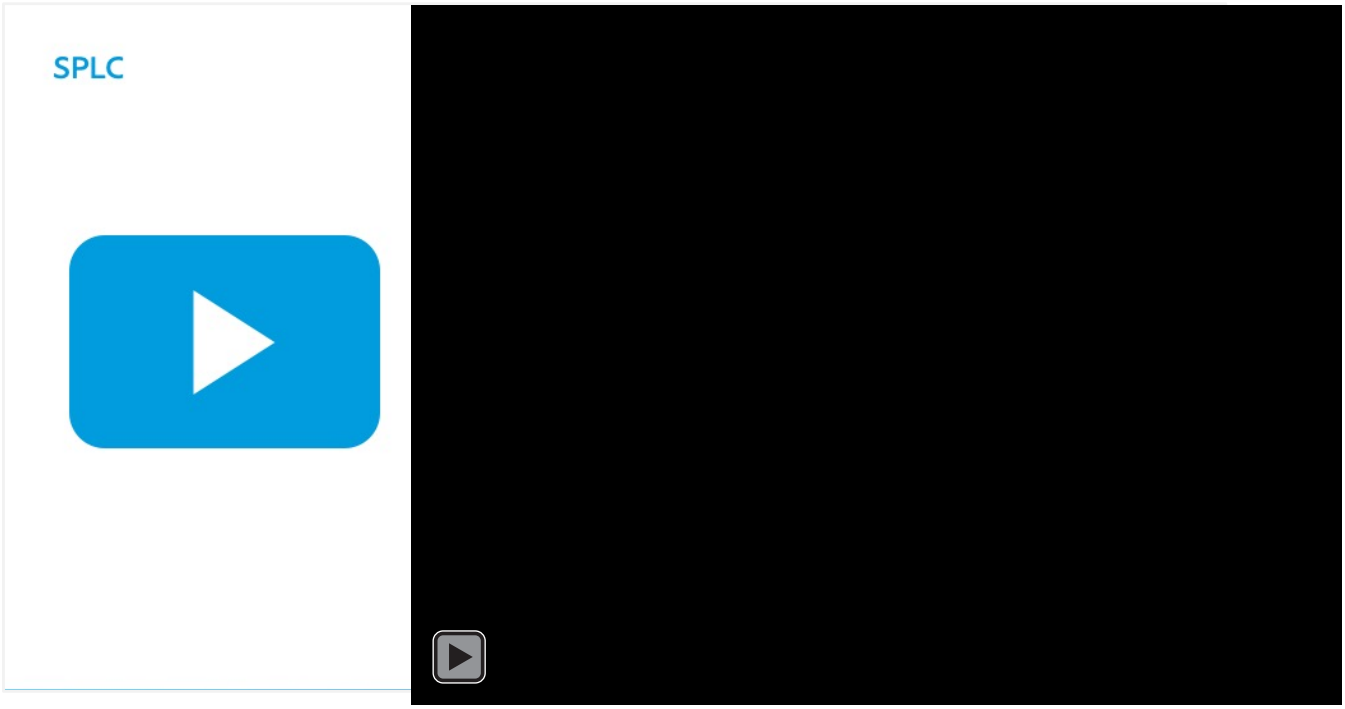


Everyday, PayPal enables hundreds of millions of customers to send, receive, and spend their money. Customers value the convenience and security we provide through our many products and platforms. It is critical, as PayPal employees, that we support these brand promises to our customers.

Understanding common security vulnerabilities and how to mitigate them is one step in protecting our customers. As product developers, we must be customer champions – following secure coding practices to develop and deploy secure applications.

Knowledge obtained from this course will help guide you in making secure coding choices. Additionally, we will share some useful resources.

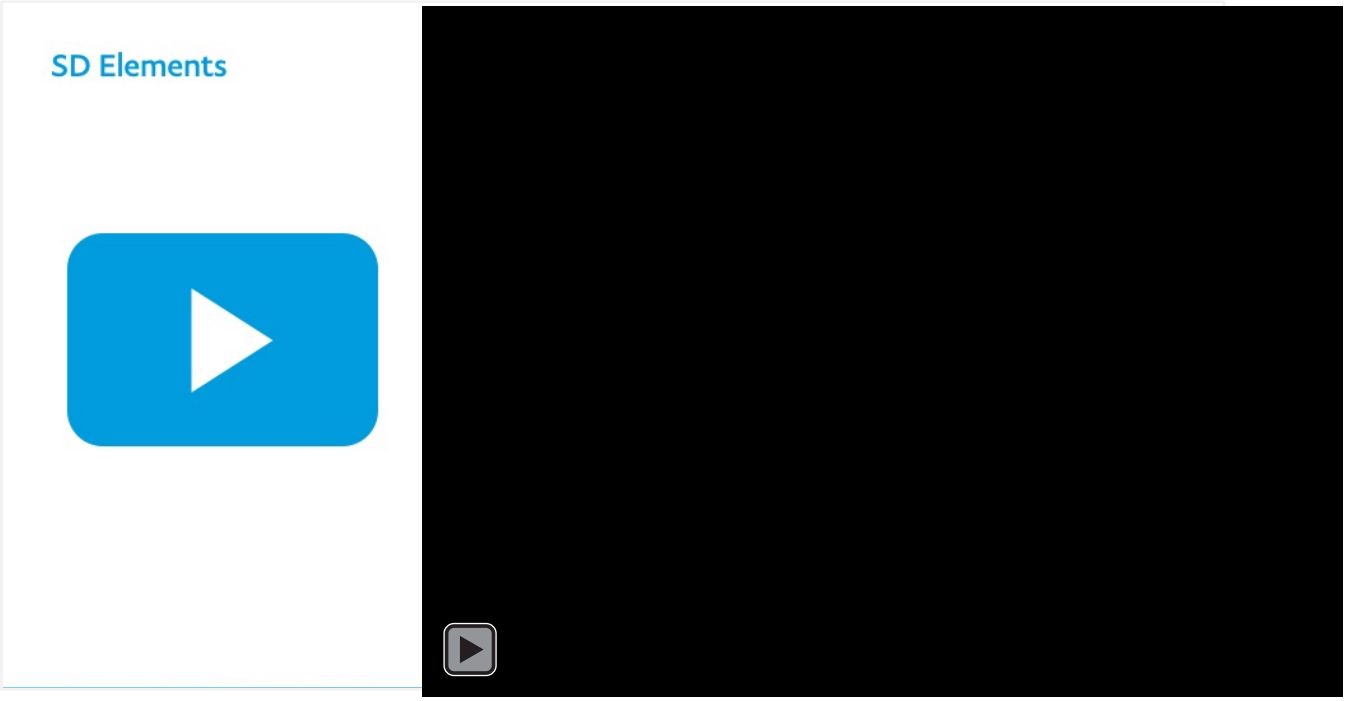
Let's begin!



Here at PayPal, we provide a standard approach and process for delivering secure PayPal products called the Secure Product Lifecycle (SPLC).

The SPLC follows the product development lifecycle, and provides products, frameworks, and processes for building secure products. With this, we can address application vulnerabilities faster and more cost-effectively, thereby ensuring customer trust in the PayPal brand.

Learn more about PayPal's SPLC and application security products by visiting go/splc.

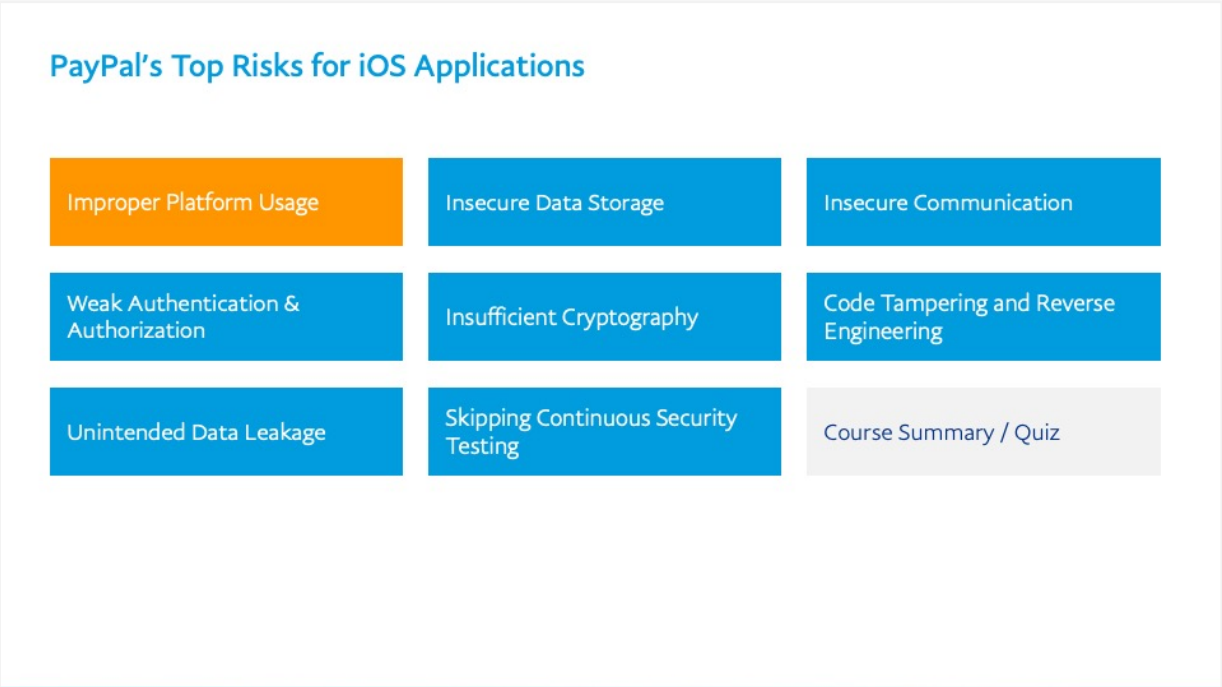


Evaluating the security concerns of an application during the design phase of the SPLC is the easiest way to deploy a secure application to our customers. As part of PayPal's SPLC, development teams have access to SD Elements, which provides security requirements or user stories after completing a simple survey.

The survey gathers information about your application including the platforms, languages, and frameworks used, and the types of data processed. SD Elements will then generate a series of security stories outlining steps required to improve the overall security of your application.

We will now show you how to add a project to SD Elements:

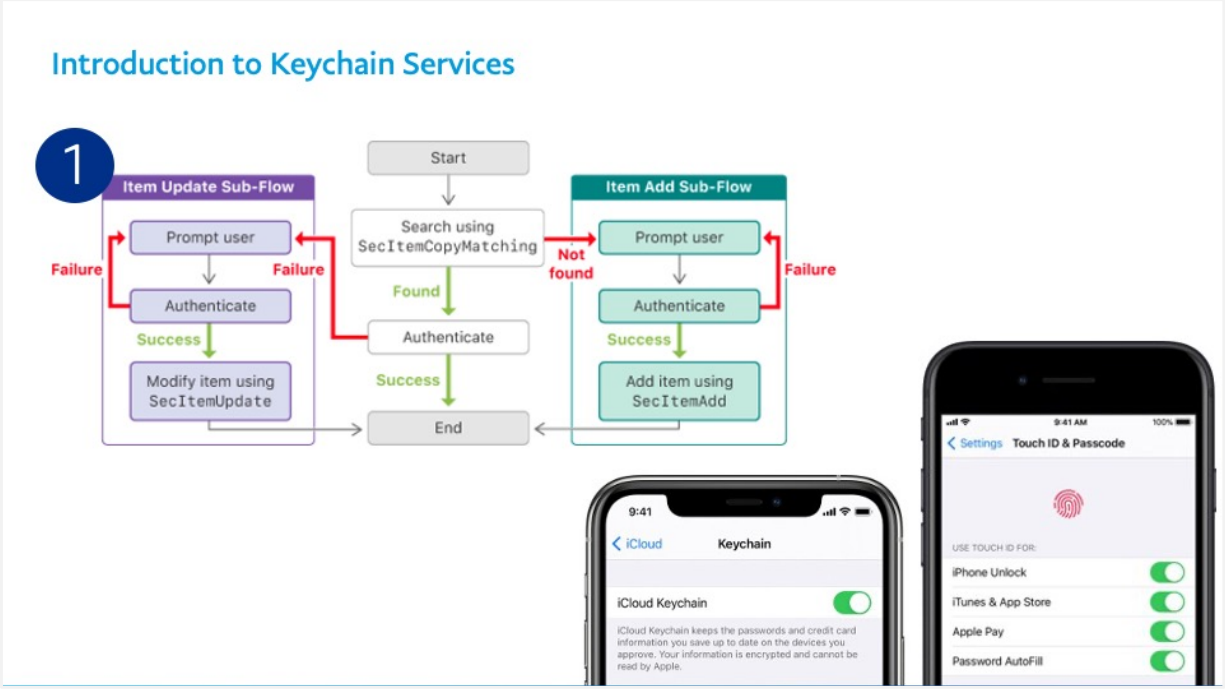
- Click on the "Components" drop-down menu.
- Click the plus icon on the top right corner.
- Input the component name and click **Create**.
- Click the plus icon on the top right corner to create a new release.
- Give a name for the new release and provide a Jira epic ID.
- Click **Create**.
- Back to the top right corner, click the plus icon.
- Select the correct framework from the survey profile and click the **Continue to survey** button.
- Answer all the required questions to complete the survey.
- Once the survey is completed, SD Elements will list out all security user story tasks.
- Details on how to complete a task are provided under each task item.
- Learn more about how to use SD Elements by visiting go/sde-jira.



This course is divided into 9 sections based on PayPal's Top Security Risks for iOS Applications including:

- Improper Platform Usage
- Insecure Data Storage
- Insecure Communication
- Weak Authentication and Authorization
- Insufficient Cryptography
- Code Tampering and Reverse Engineering
- Unintended Data Leakage
- Skipping Continuous Security Testing

Let's start by reviewing Improper Platform Usage.



Apps often need access to sensitive user data such as passwords, but keeping the data secure can come at a cost. If data is stored unencrypted, it creates a security risk. If an app repeatedly prompts the user for their password, it can cause a bad user experience, requiring the user to use simple passwords or write them down.

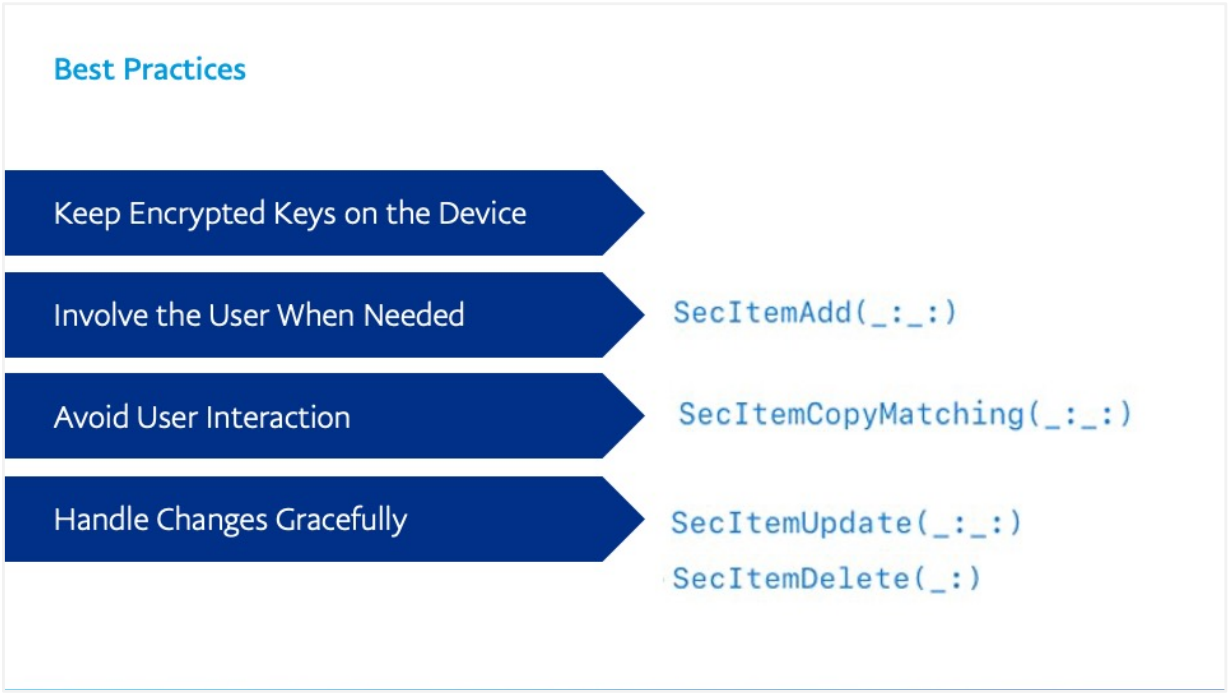
Keychain Services helps solve this problem by providing easy access to encrypted storage. Your app uses the Keychain, along with minimal user interaction, to provide a good user experience. For example, Figure 1 shows the process of storing an Internet password.

A benefit for mobile users utilizing the Keychain services is that passwords that are more complex and hard-to-remember can be stored easily. iOS provides Keychain encryption out of the box so that the developer does not introduce their own encryption methods.

By using access control lists and Keychain access groups, a developer can decide which apps and data require encryption and which do not. Biometric authentication including TouchID can be used for authenticating mobile apps and accessing stored Keychain items.

Misusing an operating system feature or failing to use platform security controls - including the Keychain, Touch ID, and other security controls - properly can create security risks. Additionally, users may opt of using it or in some cases, find ways to bypass the Touch ID option, making a user's mobile vulnerable to potential hacking.

Improper platform usage is a common risk with average detectability and can cause severe impact on affected apps. To ensure Keychain is used correctly, follow the following best practices.



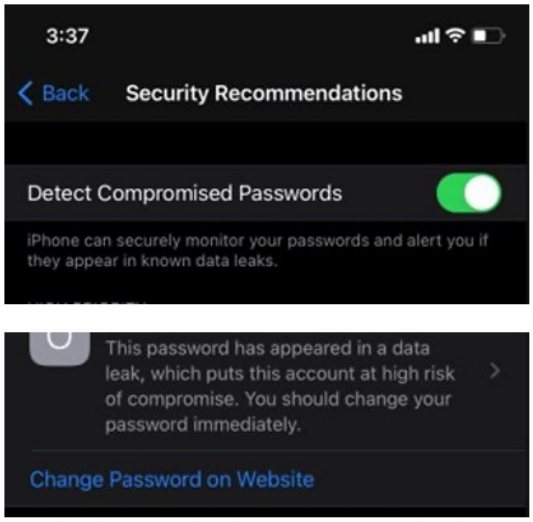
Keep Encrypted Keys on the Device: Do not allow Keychain encryptions through the server route. Instead keep the encrypted keys in one device only, so that it cannot be exploited in other devices or via the server. Secure the app by using the Keychain to store the app's secret, which should have a dedicated access control list. The user authentication policy of the access control list can be enforced by the OS.

Involve the User When Needed: The first time an app needs credentials, no password is stored in the Keychain. After the user has provided credentials that successfully authenticate, the app stores them using a call to the SecItemAdd(_:_:) function. The app now continues with its regular network access. Later, when the server requires reauthentication, the app can retrieve the credentials from the Keychain instead of prompting the user again.

Avoid User Interaction: A secure network resource requires periodic reauthentication. For example, if the user relaunches an app after having been away for a while, reauthentication will be required. The app searches the keychain for the user's password using the SecItemCopyMatching(_:_:) function. If the password is found and the app successfully uses it to authenticate, no user interaction is needed.

Handle Changes Gracefully: Occasionally, the user changes credentials outside the scope of the app. For example, you might offer a web interface to the same service where the user can change or reset their password. In this situation, a subsequent keychain item search in the app produces an out-of-date password that fails to authenticate. In this case, the app prompts the user for new credentials by calling the SecItemUpdate(_:_:) function to modify the existing stored value. The user might also decide to disconnect entirely from the network service. In response, your app should "forget" the corresponding credentials, along with performing any other actions required to log out. Use the SecItemDelete(_:) function to remove the password from the keychain entirely.

Check Password Compromise

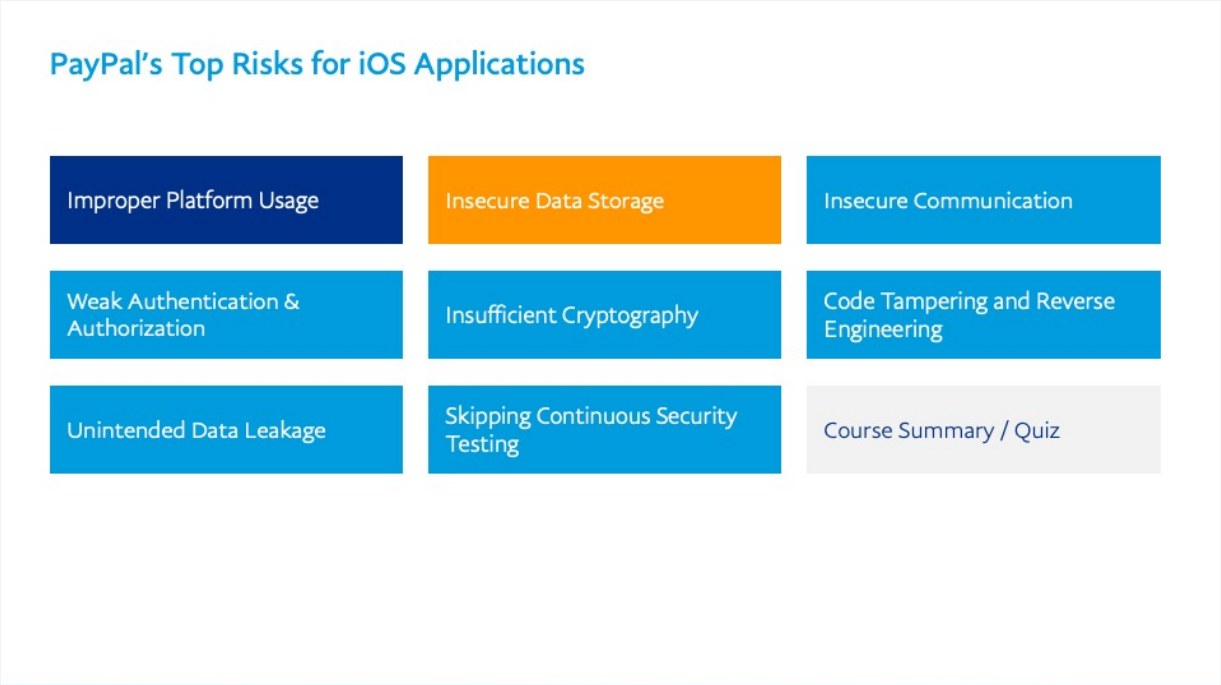


One of the most useful features in the newest iOS update will let you know if your password is compromised in a data breach.

Here's how you can check:

- Open the **Settings** app on your iPhone.
- Scroll down and tap on the **Passwords** section.
- Tap on the **Security Recommendations** and toggle the **Detect Compromised Passwords** setting.

If your password has been compromised, the system will automatically suggest you change it.



Next, let's look at Insecure Data Storage.

Mobile phones are personal devices. Users store their contacts, photos, videos, emails, messages, and other private information on their phones. There are major consequences if these devices are lost and fall into the hands of an attacker. If the user has locked the device with a passcode or biometric security, the information stored within the device can be considered safe, but what if it is not locked?

For some users, turning off security features can mean improving desired functionality and ease of use.

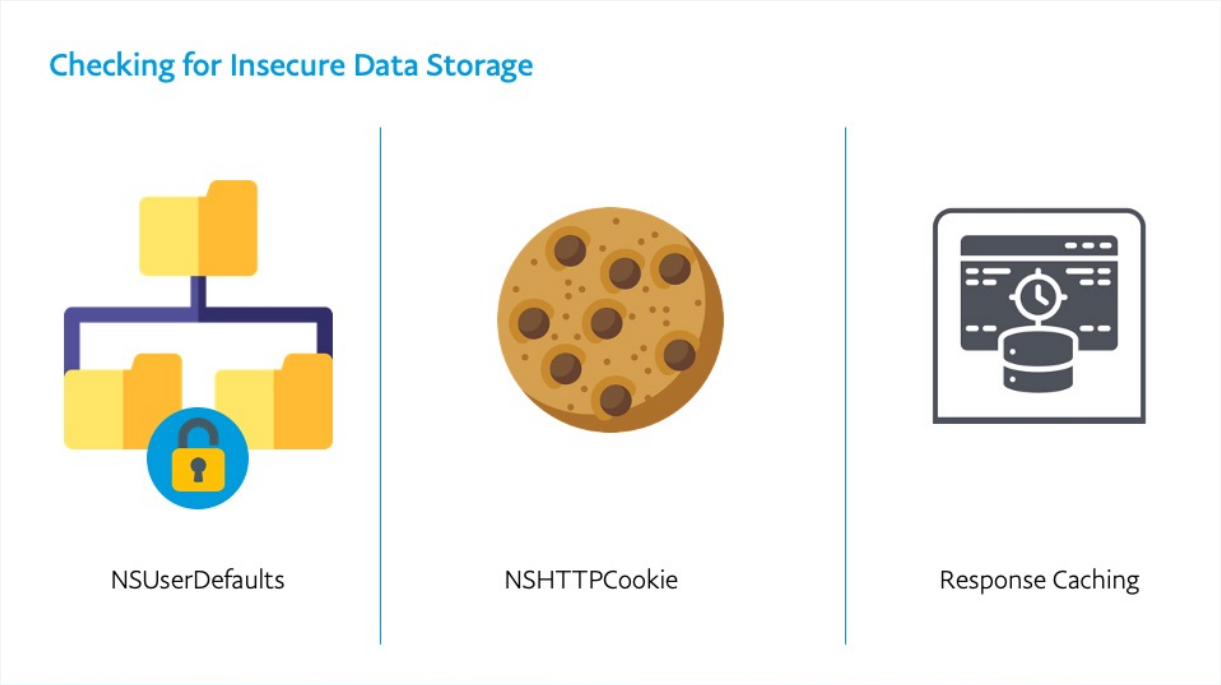
Some general patterns are:

- Users not using screen lock passcodes, since they can affect response time for various actions on their device
- Users jailbreaking their device to get more features, but not being aware of the compromise to security
- Users not being careful about what apps they install and potentially installing malware on their device

App developers should conclude that users will have the lowest level of security on their devices and should not rely on user behavior as a reliable layer of security.

As developers, we cannot stop a device from being jailbroken and falling into the hands of attackers. We must do our best to keep an app's data safe and secure. According to a recent report, insecure data storage is the most common issue in mobile applications. 76% of mobile applications had insecure data storage, putting passwords, financial information, personal data, and correspondence at risk. In addition, high-risk vulnerabilities were found in 38% of iOS mobile applications.

We will next review some good security practices for storing app data in iOS.



Insecure Data Storage in NSUserDefaults
The **NSUserDefaults** class provides a programmatic interface for interacting with the default system. The default system allows an app to customize its behavior according to user preferences. Data saved by NSUserDefaults can be viewed in the application bundle. This class stores data in a property list or plist file, but it's meant to be used with small amounts of data. NSUserDefault files should not be used for storing sensitive data.

Data is also stored in cleartext and can be easily discovered by an attacker if they get access to a device. The data stored with these methods can also be exposed through iTunes backups. You can check to see if NSUserDefaults objects are stored in Preferences by opening the <Bundle_Identifier>.plist file in the Library/Preferences directory.

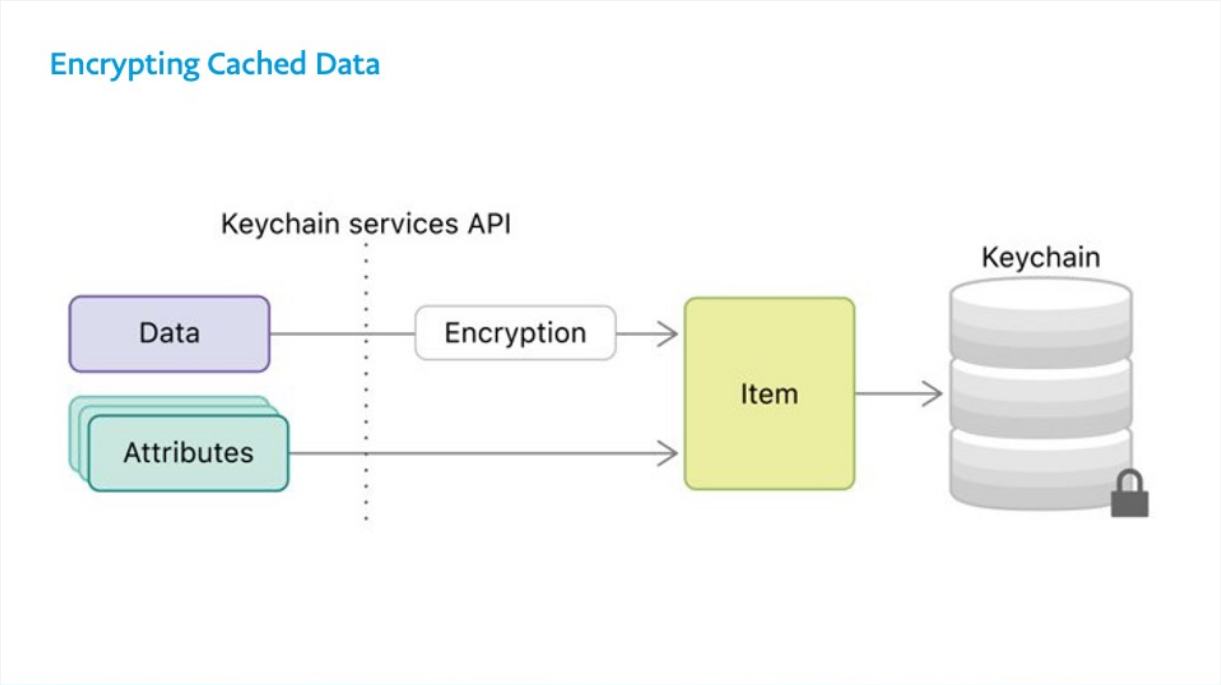
Insecure Data Storage in NSHTTPCookie
It is dangerous for an application to store sensitive data in NSHTTPCookie objects. This is because objects of this class are immutable which means that even if they are deleted or overwritten, they will continue to persist in memory. They are stored in the Local Data Storage in binary form. They can be dumped using Objective C or Swift functions. Cookies are sent in the request headers in an application's traffic.

Response Caching
By default, many iOS application frameworks enable response caching. This can lead to sensitive information disclosure as the server's responses might contain victim's sensitive information in clear text. Responses may be found cached in a database file or in a binary file in the Library/Caches directory. We can read database files using any DB browser.

iOS applications can use SQLite databases to store data required by the application. However, by default, data stored on SQLite databases is not encrypted.

SQLCipher for Xcode is an open-source encryption tool for SQLite databases in iOS apps. Be sure not to hard-code passwords to encrypted databases in places such as source code, shared preferences, or elsewhere in your app. Doing so can provide attackers an opening to accessing sensitive information.

Resources:
https://www.zetetic.net/sqlcipher/ios-tutorial/
https://github.com/sqlcipher/sqlcipher



If apps need to store sensitive data including usernames, passwords, API tokens, AUTH tokens, or other sessions' related information, developers should use the iOS Keychain. Keychain items can only be shared between apps from the same developer.

In order to ensure Keychain items are protected, especially when the device is locked, developers should not use the **kSecAttrAccessibleAlways** protection class. Instead, use the strongest protection class available for each Keychain item:

kSecAttrAccessibleWhenUnlockedThisDeviceOnly: Data in this Keychain is accessible to the app only while the device is unlocked.

kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly: Data in this Keychain items is not accessible until the device is unlocked by the user, even after a restart. This is recommended for items that need to be accessed by background applications.

Both these classes ensure that items with this attribute do not migrate to a new device. This means that even after restoring a device from a backup of a different device, these items will not be present.

Data Protection Classes

Class	Protection type
Class A: Complete Protection	(NSFileProtectionComplete)
Class B: Protected Unless Open	(NSFileProtectionCompleteUnlessOpen)
Class C: Protected Until First User Authentication	(NSFileProtectionCompleteUntilFirstUserAuthentication)
Class D: No Protection	(NSFileProtectionNone)

New files created on an iOS device can use one of the four following data classes.

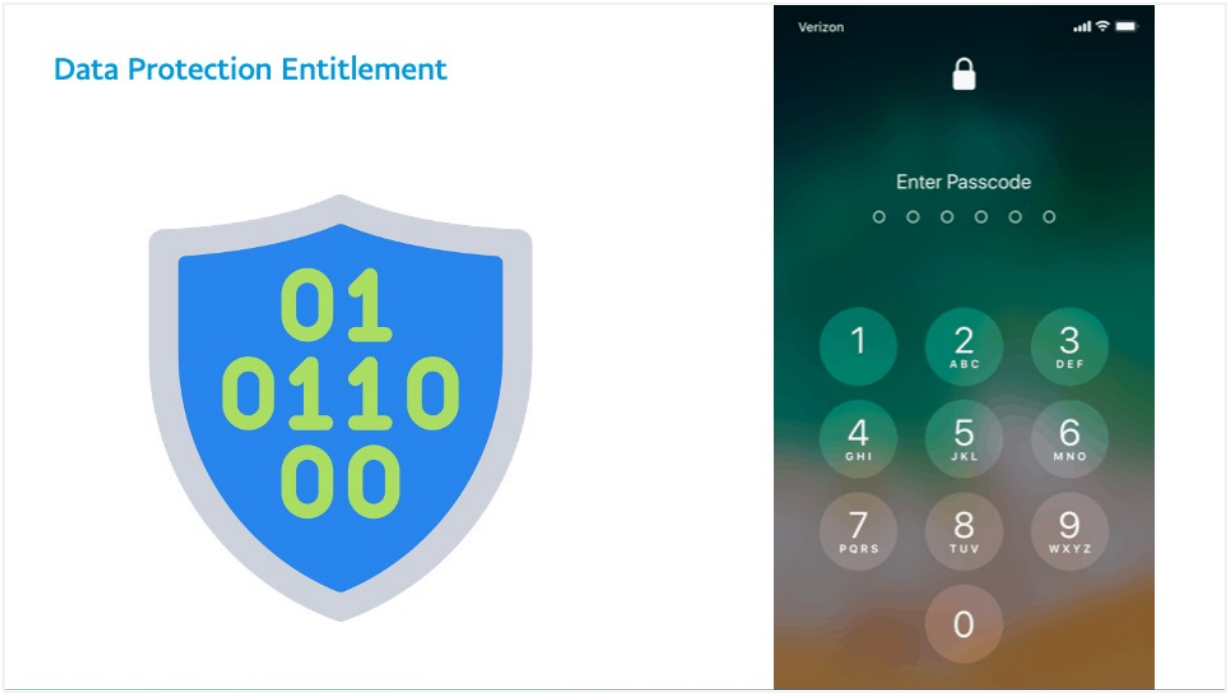
Complete Protection: The class key is protected with a key derived from the user passcode and device UID. The decrypted class key is discarded after the user locks the device and data is inaccessible until the user unlocks their device.

Protected Unless Open: This protection class is useful for tasks performing in the background when the device is locked.

Protected Until First User Authentication: This class only encrypts user's data until the very first-time users unlock their device. The class key is not removed from memory.

No Protection: The class key for this data protection class is protected only with the UID. This protection class exists for fast remote wiping where the immediate deletion of the class key makes the data inaccessible.

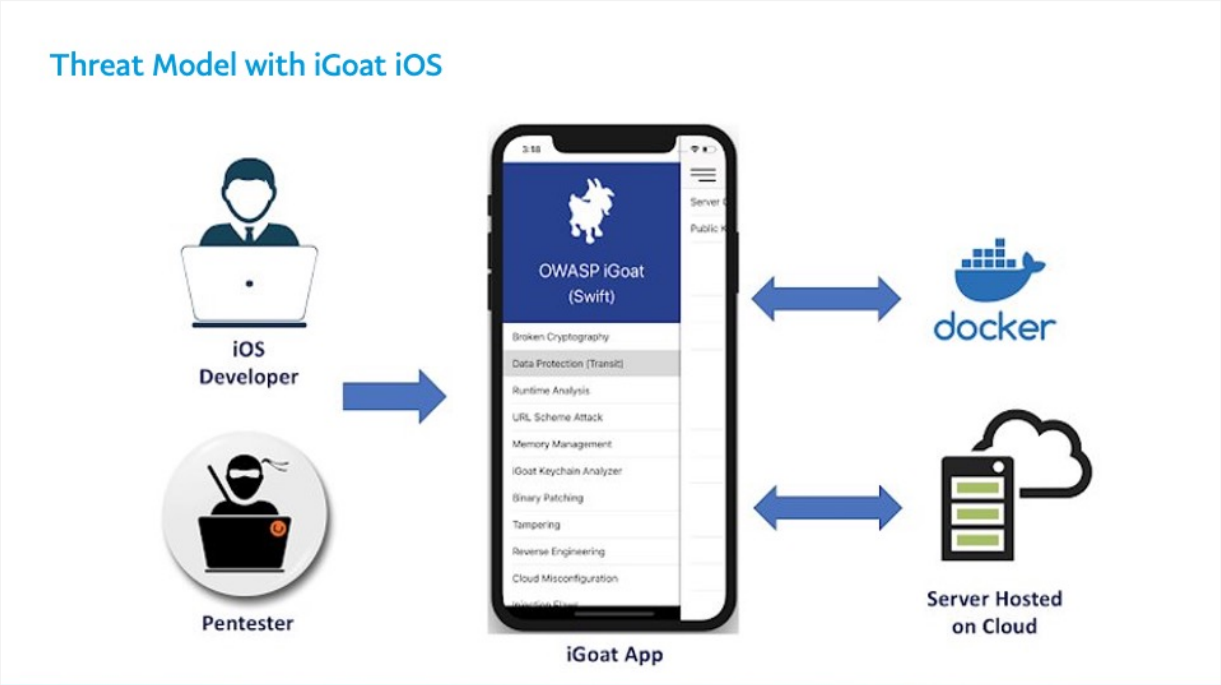
Understanding when to use each class is one step in protecting a device's data.



The default data protection class for app files since iOS 7 is **NSFileProtectionCompleteUntilFirstUserAuthentication**.

This class only encrypts user's data until users unlock their device for the first time. To improve the app data security, iOS provides the **NSFileProtectionComplete** class. This protection class provides the strongest data protection for the app since it encrypts the data whenever the device is locked. In the case a user's device is stolen, it would be very hard for an attacker to extract app user data.

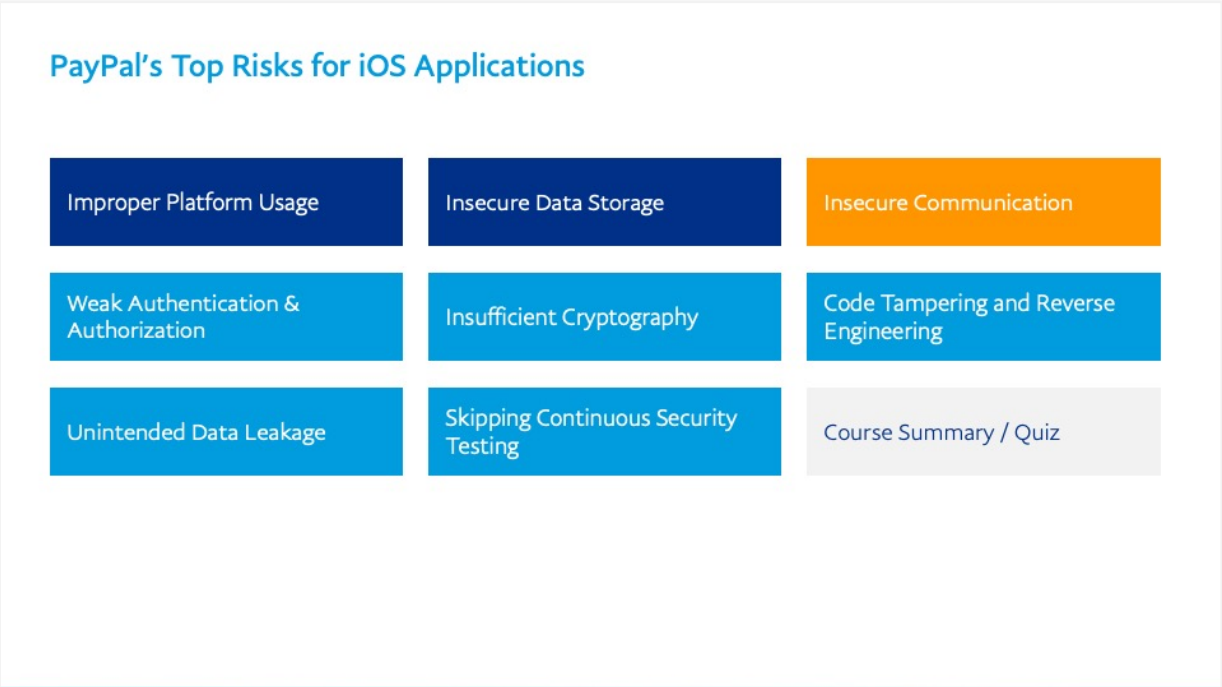
However, one problem does exist for this protection class. Tasks working in the background when the device is locked wouldn't be able to access data that is protected by this class. In this case, use the **NSFileProtectionCompleteUntilFirstUserAuthentication** class for specific files that need to be accessed while the device is locked.



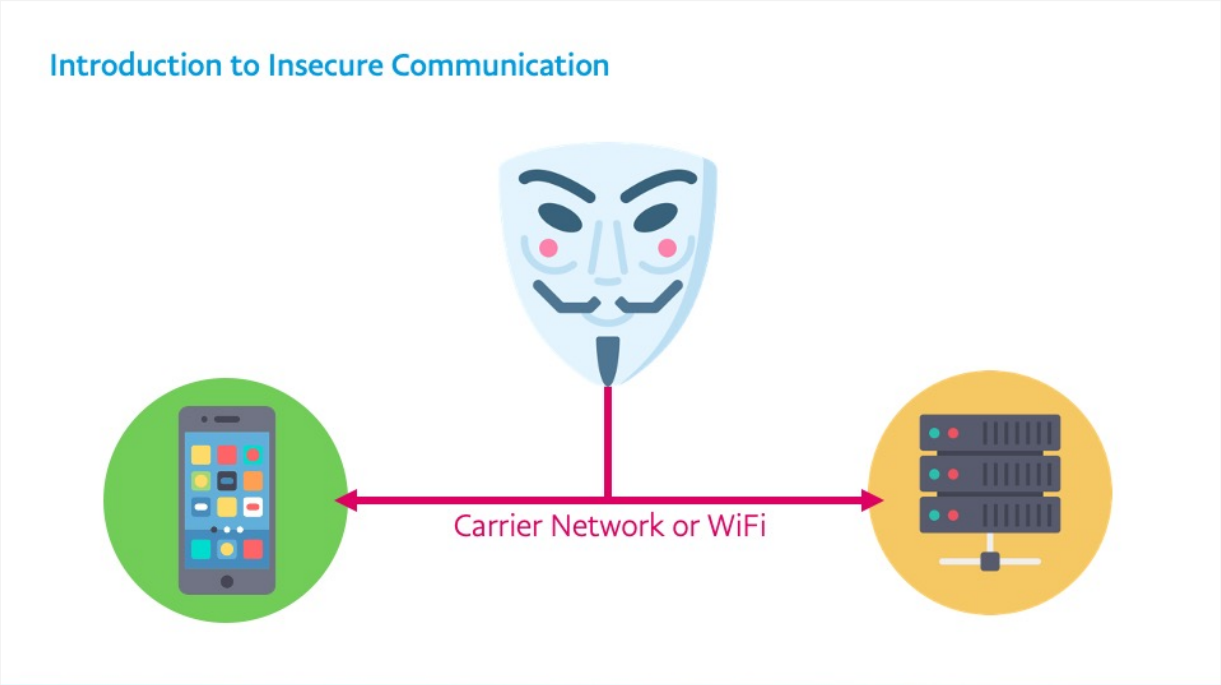
Developers should gain a deeper understanding of how to prevent insecure data storage. The Open Web Application Security Project (OWASP) recommends using purposefully vulnerable mobile apps, like iGoat, to threat model their apps and development frameworks.

This process allows developers to understand how APIs deal with information assets and app processes like URL caching, key-press caching, buffer caching, Keychain usage, application backgrounding, logging, data storage, browser cookie management, server communication, and traffic sent to third parties.

Using iGoat as a learning tool is a good way to learn these common security pitfalls and how to avoid them.



Now, let's look at Insecure Communication.



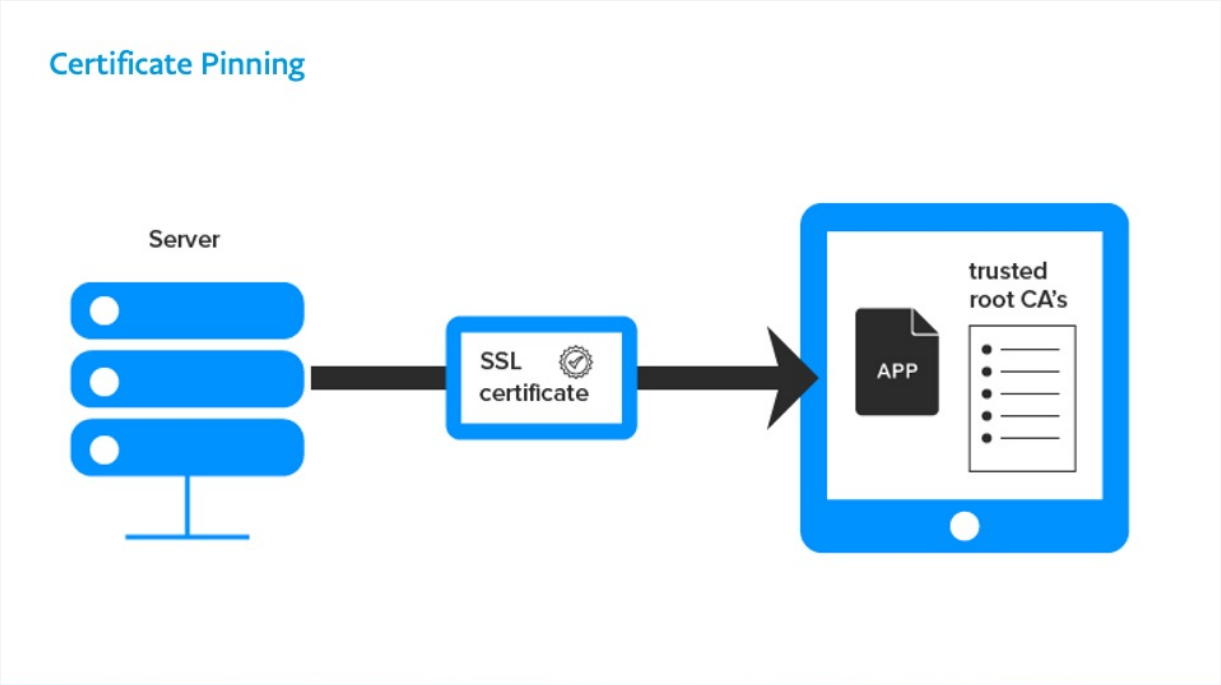
Almost all mobile apps developed in PayPal have a server-side component. Depending on the nature of the app, it may need to communicate with the server-side component hundreds to thousands of times daily, sometimes even more. Sensitive data such as usernames, passwords, and financial data are likely be transferred during these communications.

What if an attacker were to eavesdrop on communication between an app and the server-side component? What could they learn or manipulate? What if they were able to redirect all traffic from the app to a server they controlled?

Network eavesdropping attacks can happen in many ways. In some cases, attackers can trick users into using Wi-Fi hotspots that the attacker controls, giving them the ability to monitor and intercept any unencrypted mobile phone traffic.

Attackers also look for other weaknesses such as unencrypted network communication between clients and servers, or devices or software that are not updated or which malware installed from social engineering attacks.

To prevent network eavesdropping, developers need to confirm that our apps are only communicating with PayPal servers and that attackers do not have ways to redirect app traffic.



In certificate pinning, valid certificates are pre-populated onto the app. Whenever the app communicates with the app server, it will validate the certificate received from the server against an allowlist (formally whitelist) of certificates.

If the check passes, the app communicates with the server. If the check fails, the app safely assumes it is not communicating with a PayPal server.

There are two things to remember while implementing certificate pinning:

1. **Keep certificates updated** - Whenever the app server certificate is changed, an app should also be updated.

2. **Replace certificates before pushing code to production** - If you publish app code with test environment certificates, regular users won't be able to use them. Test environment certificates must be replaced with valid production environment certificates before publishing an app.

Certificates in Test Environments

PayPal Key Pinning Guide:
go/certpinning

Invalid certificate workarounds can also create unintended security issues. Occasionally, developers may turn off certificate validation during debugging. While useful when debugging, this can lead to unintended vulnerabilities if certificate validation is not turned on again prior to pushing code to production. Developers should ensure debug mode is disabled before publishing the code to the production.

Commonly, development environments use blocklist (formerly blacklist) exemptions to disable app transport security. This is not recommended.

Check the app's Info.plist to ensure the following blocklist exemptions are not used.

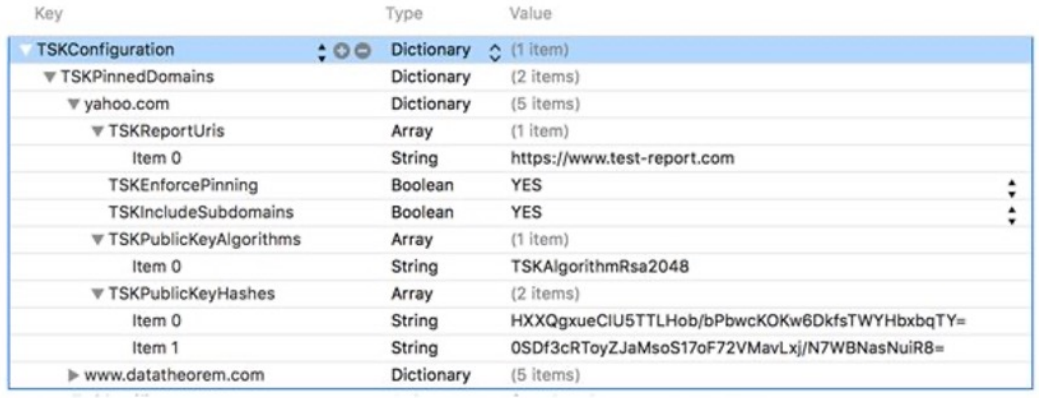
1. NSAllowsArbitraryLoads
2. NSExceptionAllowsInsecureHTTPLoads
3. NSExceptionMinimumTLSVersion
4. NSExceptionRequiresForwardSecrecy

iOS performs certificate verification automatically with App Transport Security (ATS), so any app with invalid certificates cannot communicate with servers. Be sure you follow good practices of certificate pinning and certificate validation is properly enabled before pushing to production.

Check out go/certpinning for documentation on key pinning.

TrustKit

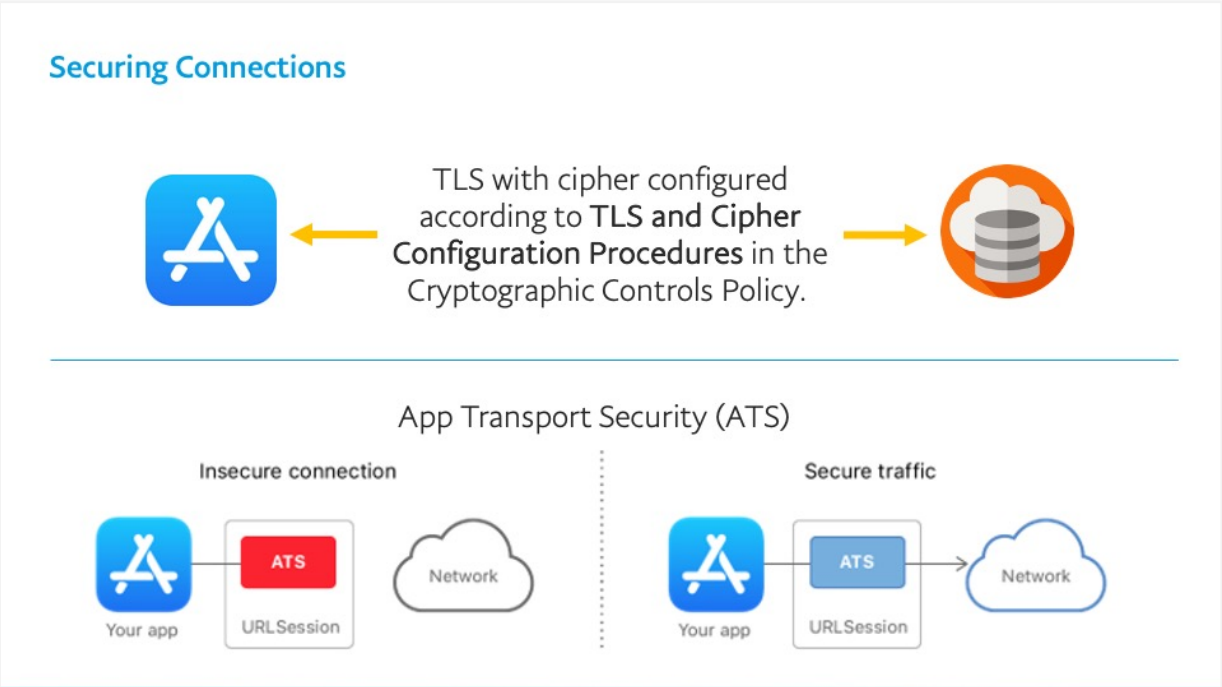
The policy can be configured within the App's `Info.plist` :



TrustKit is an open-source framework that makes it easy to deploy SSL public key pinning and reporting in any iOS 11+, macOS 10.13+, tvOS 11+ or watchOS 4+ App; it supports both Swift and Objective-C Apps.

TrustKit provides the following features:

- A simple API to configure an SSL pinning policy and enforce it within an app. The policy settings are heavily based on the HTTP Public Key Pinning specification.
- Sane implementation by pinning the certificate's Subject Public Key Info, as opposed to the certificate itself or the public key bits.
- A reporting mechanism to notify a server about pinning validation failures happening within the app, when an unexpected certificate chain is detected. This is like the report-uri directive described in the HPKP specification. The reporting mechanism can also be customized within the app by leveraging pin validation notifications sent by TrustKit.
- An auto-pinning functionality by swizzling the App's NSURLConnection and NSURLSession delegates in order to automatically add pinning validation to the App's HTTPS connections. This allows deploying TrustKit without modifying the app's source code.



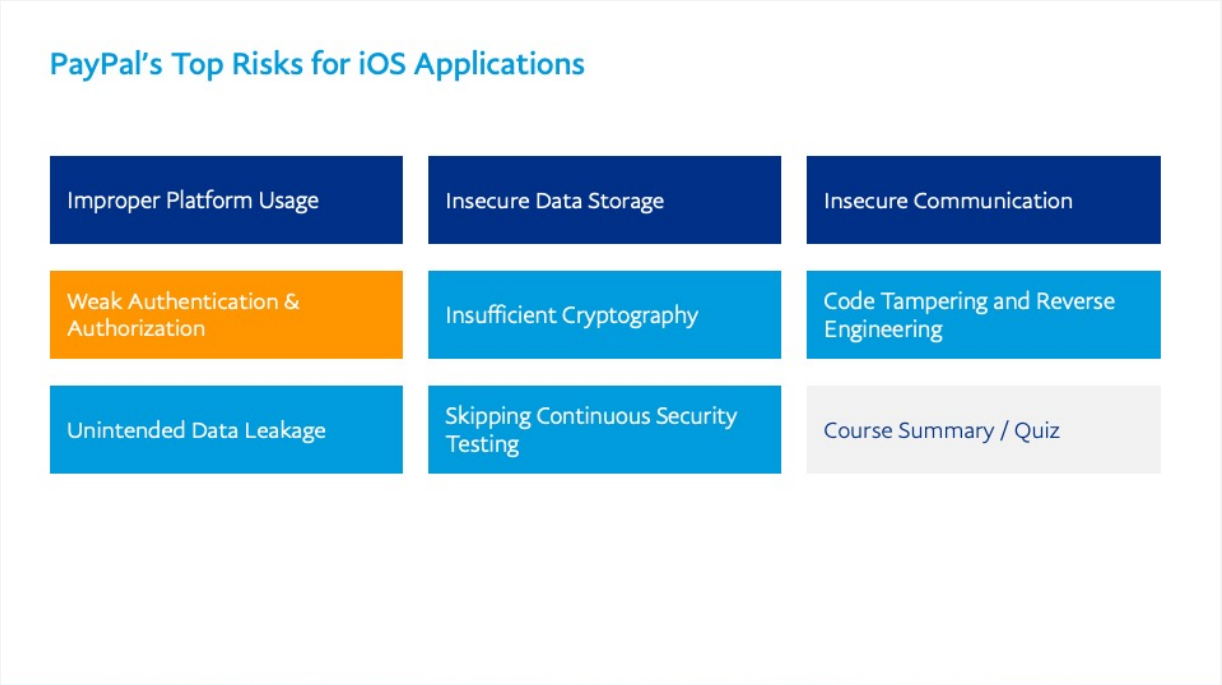
HTTPS and encryption are a must-have security requirement for all PayPal apps. Be sure you are familiar with certificate and cipher configurations as outlined by the PayPal Cryptographic Controls Standard Policy.

The goals of the TLS and Cipher include:

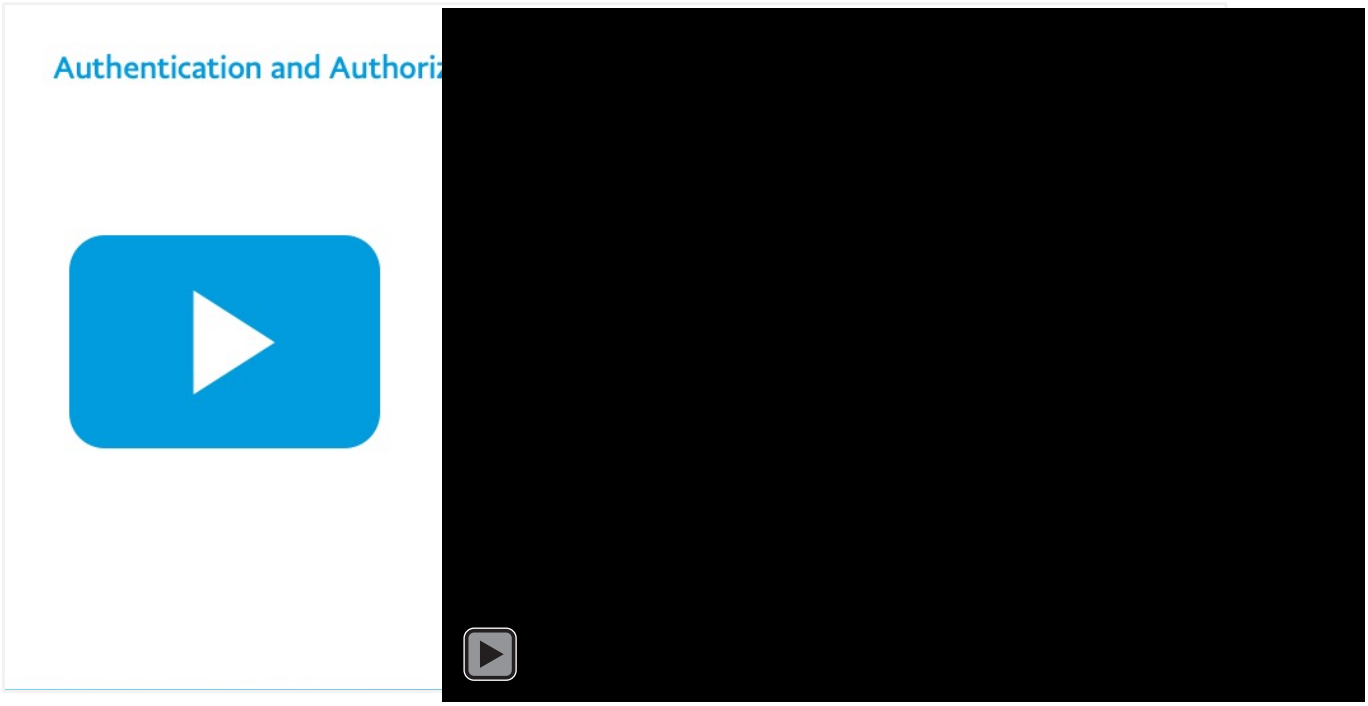
- Use the most secure configurations possible for each type of interface.
- Protect against known TLS and Cryptographic configuration vulnerabilities and exploits and address any outstanding Bug Bounty remediation issues.
- Leverage the latest standards once they are ready for prime time practical and identify the configurations that are required for all PayPal sites.

Enforcing secure network links in your app can be done using App Transport Security. On Apple platforms, a networking security feature called App Transport Security (ATS) improves privacy and data integrity for all apps and app extensions. It does this by requiring that network connections made by your app are secured by the Transport Layer Security (TLS) protocol using reliable certificates and ciphers. ATS blocks connections that don't meet minimum security requirements.

ATS operates by default for apps linked against the iOS 9.0 or macOS 10.11 SDKs or later. In cases where you need to connect to a server that isn't fully secure—and that you can't reconfigure to make more secure—you can add exceptions to loosen some of the ATS requirements.



Now, let's look at Weak Authentication and Authorization.

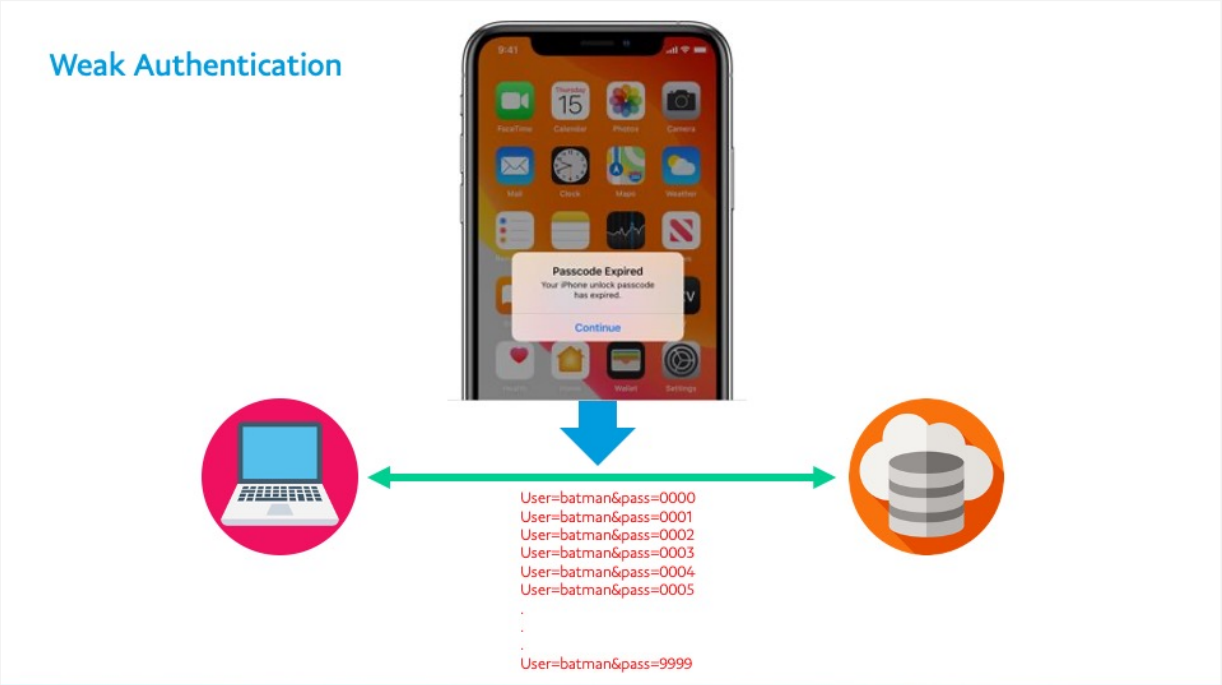


Authentication and authorization are critical security considerations across applications in the PayPal infrastructure. Therefore, it is very important to ensure that all the applications created adhere to strong authentication and authorization controls. All externally accessible PayPal endpoints SHOULD perform both authentication and authorization checks before processing the request.

What are the differences between authentication and authorization?

Authentication is the process of verifying who you are. When you log on to a service with a username and password, you are authenticating into the service.

Authorization is the process of verifying that you have access to something. When you try to access a resource or perform an action, authorization determines if you are permitted to do so. Authorization should only occur after a successful authentication.



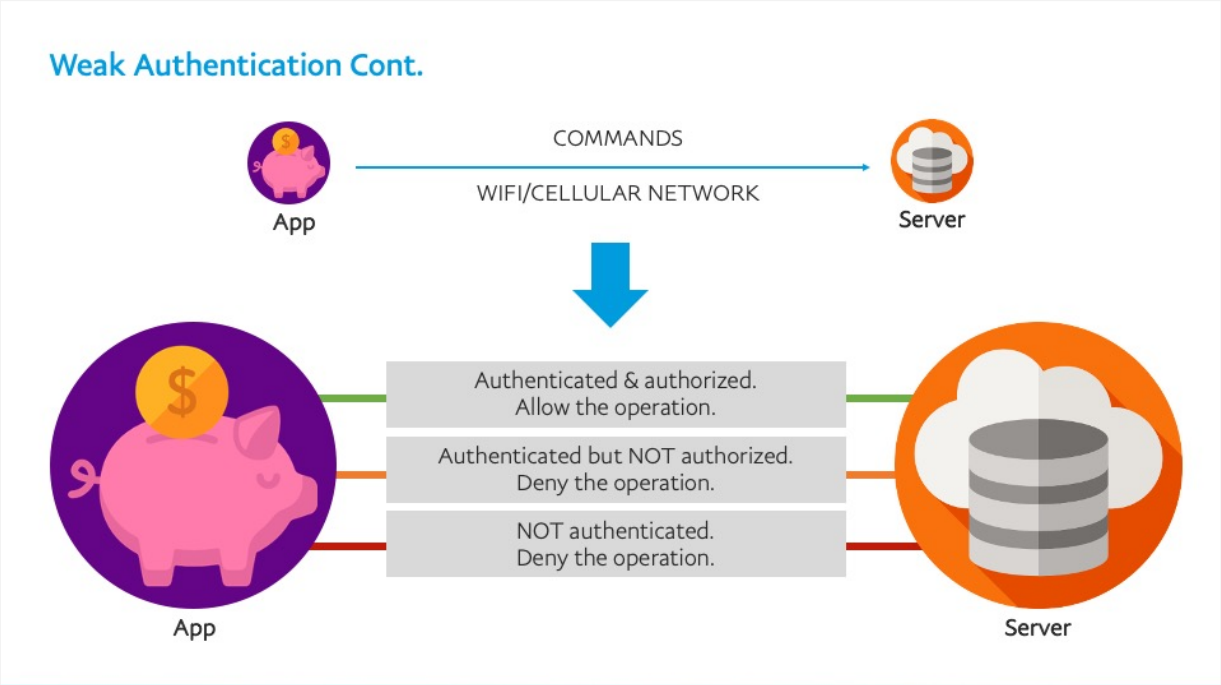
Consider the following scenario: PayPal is launching a new account authentication mechanism for Android app users. This would require users to authenticate with a 4-digit PIN. What are some possible misuse cases for this new authentication mechanism?

Shoulder surfing, where an attacker looks over a user's shoulder while they type in their PIN, is one possible misuse case. Of course, some users may try to safeguard their PIN by covering their phone while they enter it. However, this is not a reliable safeguard.

Attackers will always choose the path of least resistance. The easiest way for an attacker to exploit the example authentication mechanism would be to target the app server instead of the app itself. Attackers know that usernames and PINs are saved on a server. They don't need to interact with an app to take over a user's account.

They can instead observe the communication between the app and app server using their own device. Once familiar with the protocol and other communication mechanisms, they will bypass the app and start generating the communication on their own. Using automated scripts, which tests all possible 4-digit PIN combinations against the app server, can be done very quickly.

Even if the password is not a 4-digit PIN, the attacker could follow the same steps for similar authentication. Understanding misuse cases such as this one is crucial to preventing authentication security issues.



Consider this next scenario. The PayPal app has been redesigned so authentication and authorization decisions are done on the app side. The app is responsible for deciding:

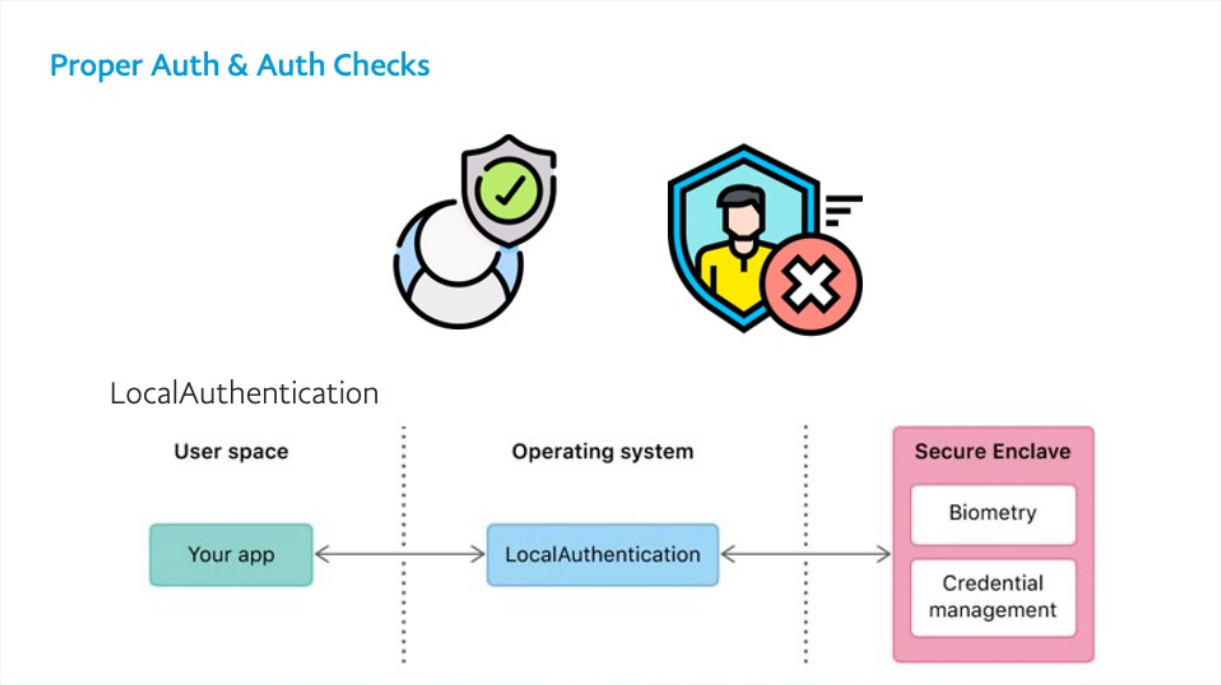
1. If the user is permitted to complete transactions
2. If user A can view personal information of user B

Once the app has made the decision locally, it will call the app server to execute the transactions or fetch the requested personal information.

To exploit these authentication and authorization functionalities, an attacker will start by observing the communication between the app and app server. During their analysis, they determine the service does not require a valid account for the above requests, and they only need to collect the endpoint URL and the request/command structure.

After collecting information on the necessary operations, they bypass the app and start communicating with the app server directly. The attacker is now able to perform transactions between any account or view personal information of any user just by requesting it from the server.

This example illustrates why authentication and authorization decisions should not be decided by the app. Doing so could create very large security risks.



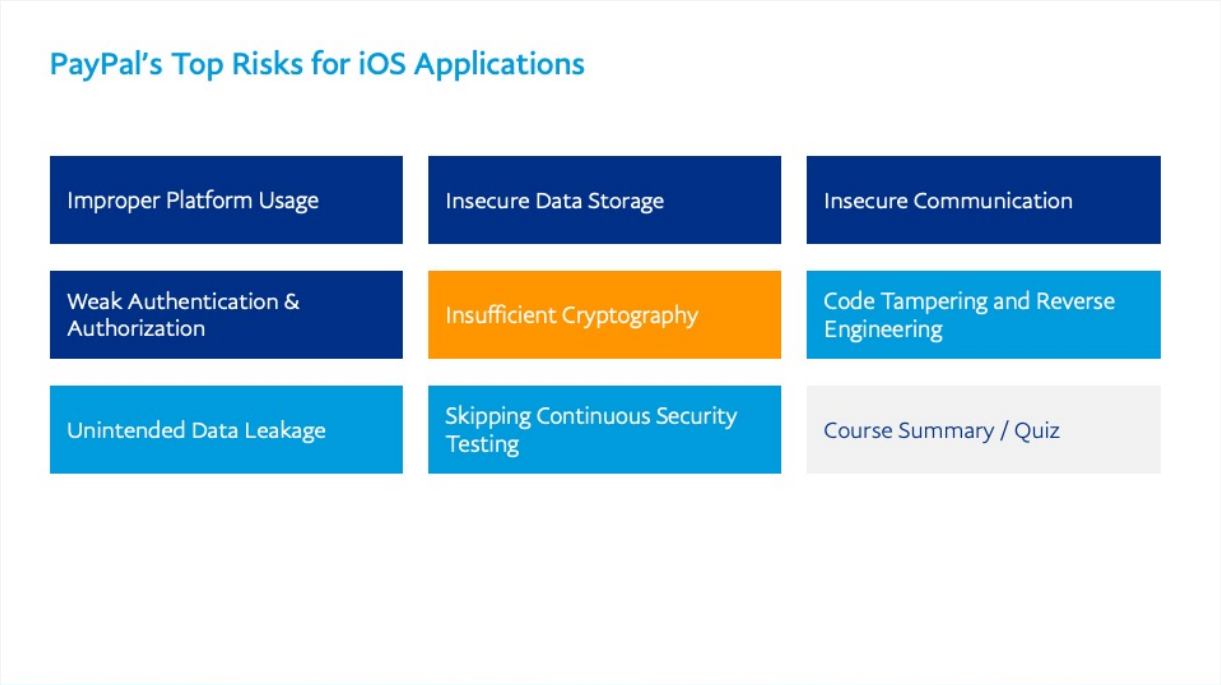
Here are some recommended ways to perform authentication and authorization checks.

1. Differentiate between functionality available to authenticated and non-authenticated users. For all authenticated functions, check if the user has necessary permissions to complete that function. If the user is not logged in, deny the request and invite the user to login. The use of existing PayPal frameworks will also add layers of security.
2. During local authentication, an app authenticates the user against credentials stored locally on the device. The user is able to "unlock" the app or some inner layer of functionality by providing a valid PIN, password, or biometric characteristics such as face or fingerprint, which is verified by referencing local data. This allows users to conveniently resume an existing session with a remote service or as a means of step-up authentication to protect some critical function.
3. Many users rely on biometric authentication like Face ID or Touch ID to enable secure, effortless access to their devices. As a fallback option, and for devices without biometry, a passcode or password serves the same purpose. Use the LocalAuthentication framework to leverage these mechanisms in your app and extend authentication procedures that your app already implements.

Developers have two options for incorporating Touch ID/Face ID authentication:

1. LocalAuthentication.framework: a high-level API that can be used to authenticate the user via Touch ID. The app can't access any data associated with the enrolled fingerprint and is notified only whether authentication was successful. Developers can display and utilize an authentication prompt by utilizing the function evaluatePolicy of the LAContext class. Two available policies define acceptable forms of authentication:
   - deviceOwnerAuthentication(Swift) or LAPolicyDeviceOwnerAuthentication(Objective-C): The user is prompted to perform Touch ID authentication. If Touch ID is not activated, the device passcode is requested instead. If the device passcode is not enabled, policy evaluation fails.
   - deviceOwnerAuthenticationWithBiometrics (Swift) or LAPolicyDeviceOwnerAuthenticationWithBiometrics(Objective-C): Authentication is restricted to biometrics where the user is prompted for Touch ID.
2. Security.framework: A lower-level API to access keychain services. This is a secure option if your app needs to protect some secret data with biometric authentication, since the access control is managed on a system-level and cannot easily be bypassed. Security.framework has a C API, but there are several open-source wrappers available. The framework underlies the LocalAuthentication.framework. Apple recommends using the higher-level APIs whenever possible.

Be aware that using either the LocalAuthentication.framework or the Security.framework will be a control that can be bypassed by an attacker as it only returns a boolean and no other data. The Apple Developer website offers code samples for both Swift and Objective-C.



Next, we'll look at Insufficient Cryptography.

Data Encryption & Decryption

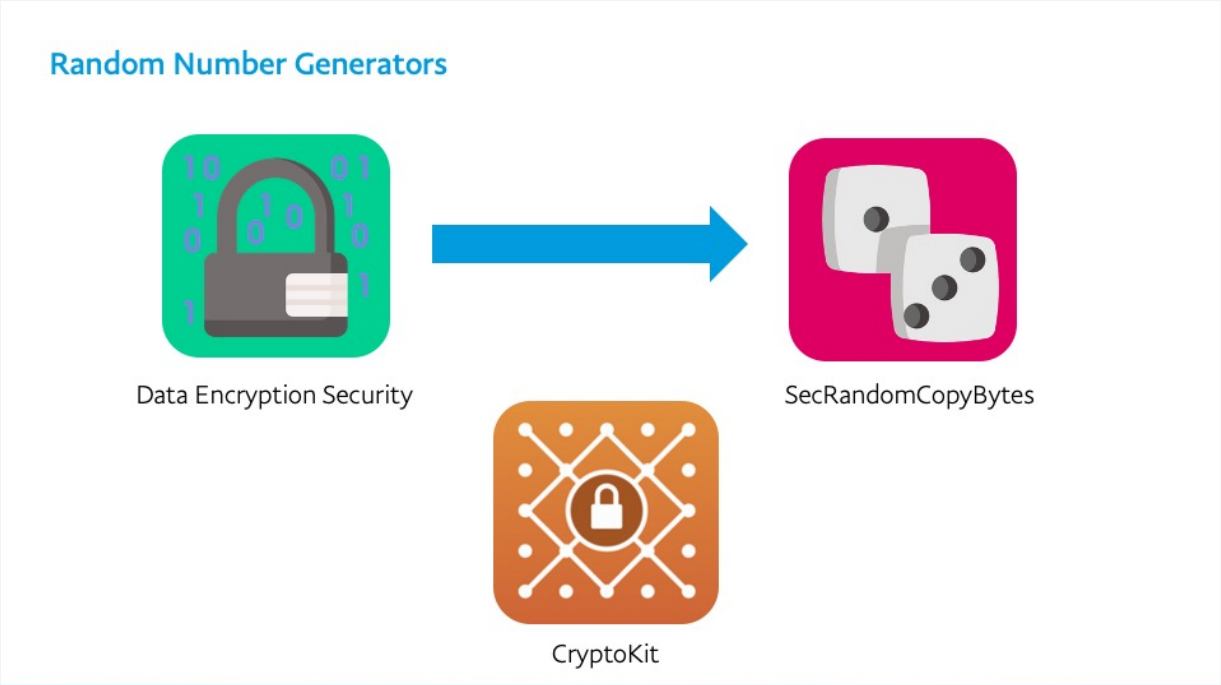
Encoding	Encryption	Hashing	Obfuscation	Compression
ASCII	AES	SHA-256	JavaScript Obfuscator	LZMA
Unicode	Blowfish	MD5	Progaurd	Zip
Base64	RSA	SHA-512	Dotfuscator	Rar
URL Encoding	Triple DES	SHA-3	XOR	

Sensitive Information Disclosure is the most significant risk faced by PayPal applications. As an online payments system provider, it is vital PayPal handles customer data with the utmost care.

Data Classification Standards help determine the type of data at hand, as well as how it should be handled. According to the Data Classification Standards, specific controls are in place that determine where and how data can be stored, transmitted, or processed. Developers should understand the differences between encoding, encryption, hashing, obfuscation, and compression, and how to use them.

While all these techniques convert plain text to something that is not immediately human readable, not all are suitable for protecting information. For example, information that is encoded, obfuscated, or compressed can be restored to human-readable form just by knowing the corresponding algorithm and hence are NOT SECURE.

Hashing transforms data and makes data retrieval irreversible (i.e., one-way encryption). Hashing is irreversible and should not be used in cases where you would need to retrieve the data from the hash.



Encryption plays a big role in protecting data handled by PayPal apps. Developers must follow Cryptographic Control standards when handing sensitive data and implementing encryption. When deciding on which encryption algorithms to encrypt apps, it is recommended that you use modern algorithms from a trusted source that is tested frequently by the security community.

Additionally, The National Institute of Standards and Technology of the US government publishes cryptographic standards from time to time and recommends encryption algorithms. You can read this document to learn about emerging threats.

For iOS, use Apple's **CryptoKit** to perform cryptographic operations securely and efficiently.

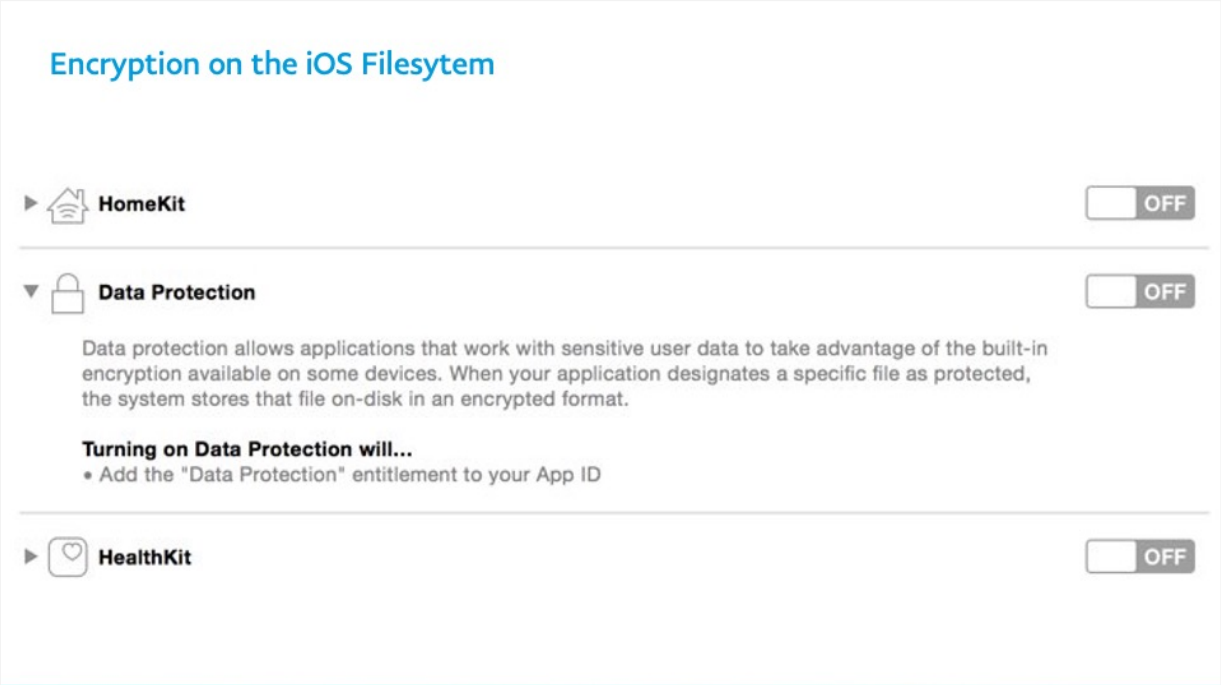
Common cryptographic operations include:

- Computing and comparing cryptographically secure digests.
- Using public-key cryptography to create and evaluate digital signatures and to perform key exchange. In addition to working with keys stored in memory, you can also use private keys stored in and managed by the Secure Enclave.
- Generating symmetric keys and using them in operations like message authentication and encryption.

Choose CryptoKit over lower-level interfaces. CryptoKit frees your app from managing raw pointers, and automatically handles tasks that make your app more secure, like overwriting sensitive data during memory deallocation.

Random Number Generators
Random number generators are used by apps to create random numbers. Cryptographically strong random number generators should be used when generating security-sensitive features such as unique IDs, passwords or secret keys. Random number generators should use strong entropy so that number correlation attackers are mitigated; in other words, so true randomness is ensured.

The **SecRandomCopyBytes** class provides a cryptographically strong random number generator.

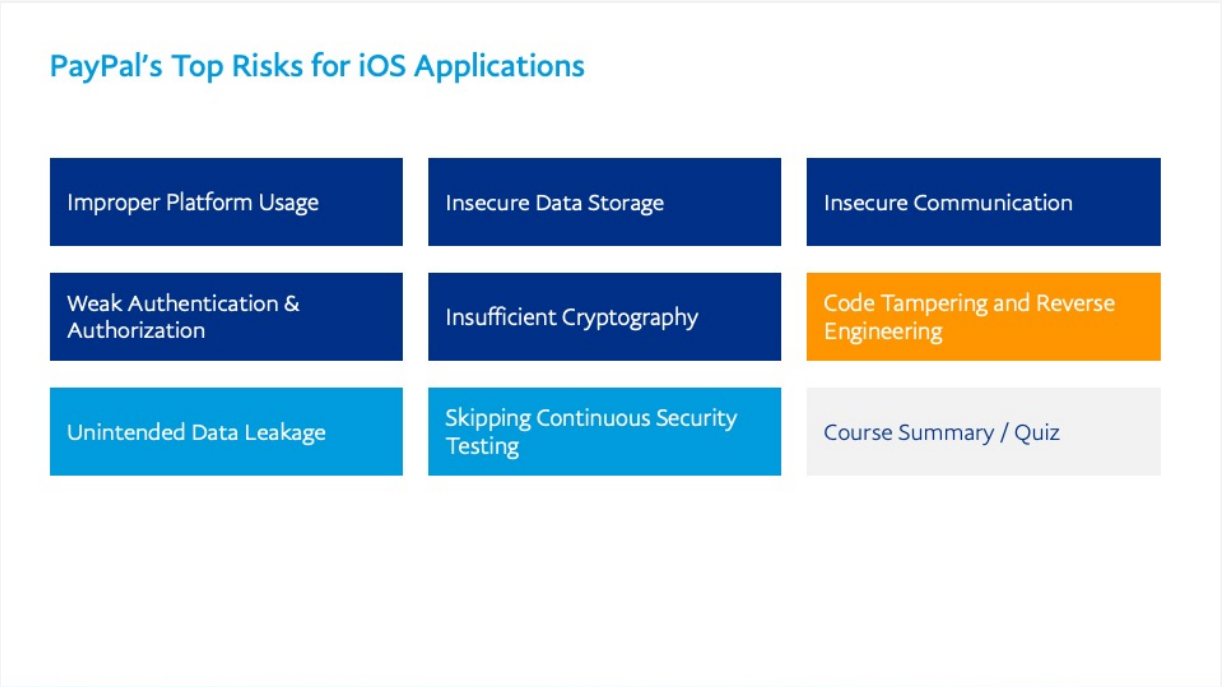


To enable encryption on your iOS filesystem, turn on the Data Protection entitlement in the XCode project configuration of the App. As shown in the screenshot, this setting is available in the **Capabilities** tab within XCode.

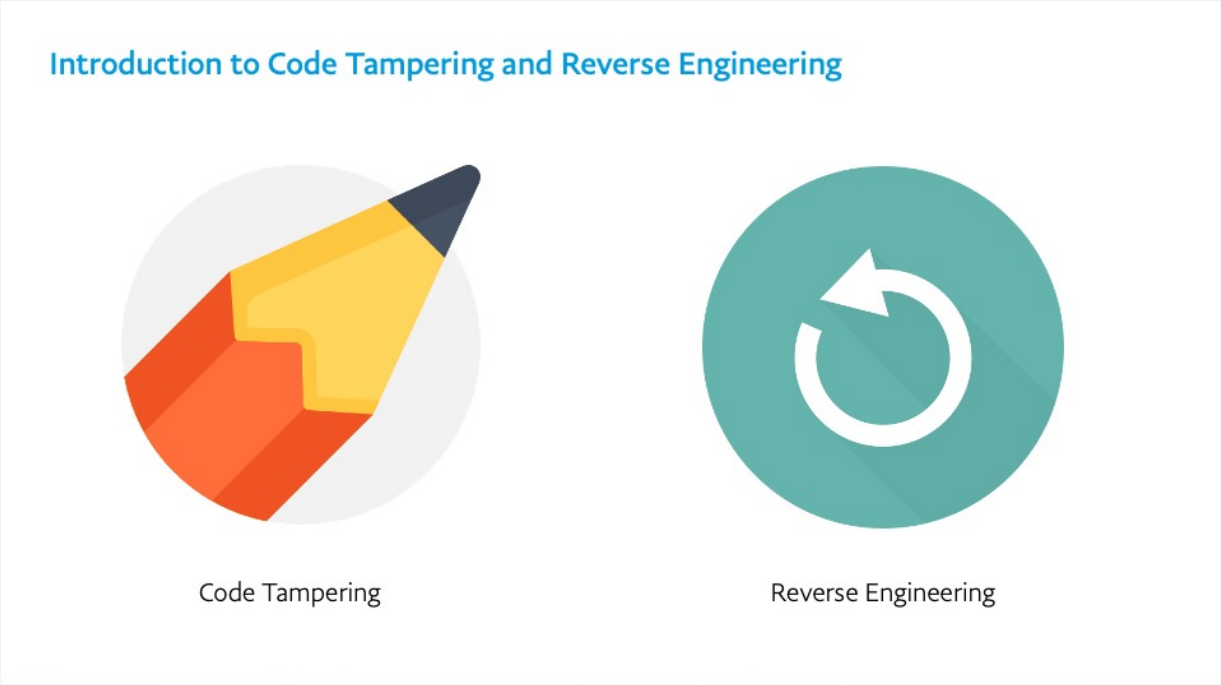
Because the entitlement will cause iOS to encrypt all the App's files when the device is locked, it might cause problems if the App tries to access its files while running in the background (for example, when the device might be locked).

You can lower the protection level to **NSFileProtectionCompleteUntilFirstUserAuthentication** for the specific files that need to be accessed while in the background.

Lastly, the applicationProtectedDataWillBecomeUnavailable: and applicationProtectedDataDidBecomeAvailable: App delegate methods can be used to manage the App's access to protected files.



Next, we will review Code Tampering and Reverse Engineering.

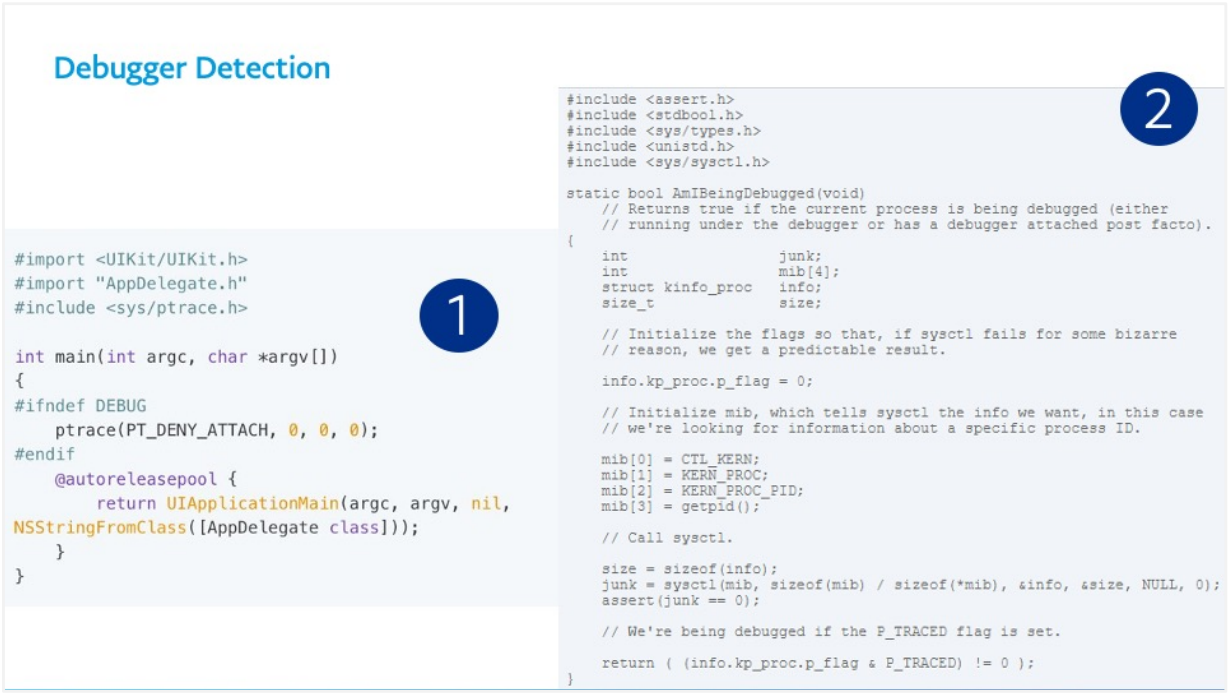


Two attack methods often overlooked by app developers are **code tampering** and **reverse engineering**.

All iOS apps in the App Store are encrypted. An attacker needs to break the encryption in order to debug an application. However, iOS devices can decrypt this code. In order to decrypt a target application, an attacker will need some ingenuity and time.

For example, an attacker works with a jailbroken device and installs a debugger. When they launch the target app, iOS will decrypt the app code and place it in memory. That attacker can now dump the memory and get the decrypted app code. At this stage, the attacker can use other tools to analyze the app further.

While there is no foolproof mechanism to fight against code tampering and reverse engineering attacks, there are some defense mechanisms which can be implemented to prevent these kinds of attacks. We will also cover how to stop app debugging, how to distribute app logic for secure usage, code obfuscation, and jailbreak detection.



There are a few different anti-debugging techniques applicable to iOS which can help prevent a debugger from attaching to an app.

First, iOS XNU kernel implements a ptrace system call which can be used to detect a debugger, and if a debugger is found will terminate processes. (See image 1) The example code can help detect if an app is being actively traced by a debugger. This code should run in the main.m file of the app.

The second example code comes from the Apple Documentation Archive. Here the code inspects the CPU kernel's process list for a ptrace flag. The kernel will watch for any app being traced. (See Image 2)

We recommend saving the above code as a header file and calling the function when the app handles sensitive information. This will prevent any overloading the kernel with frequent requests.

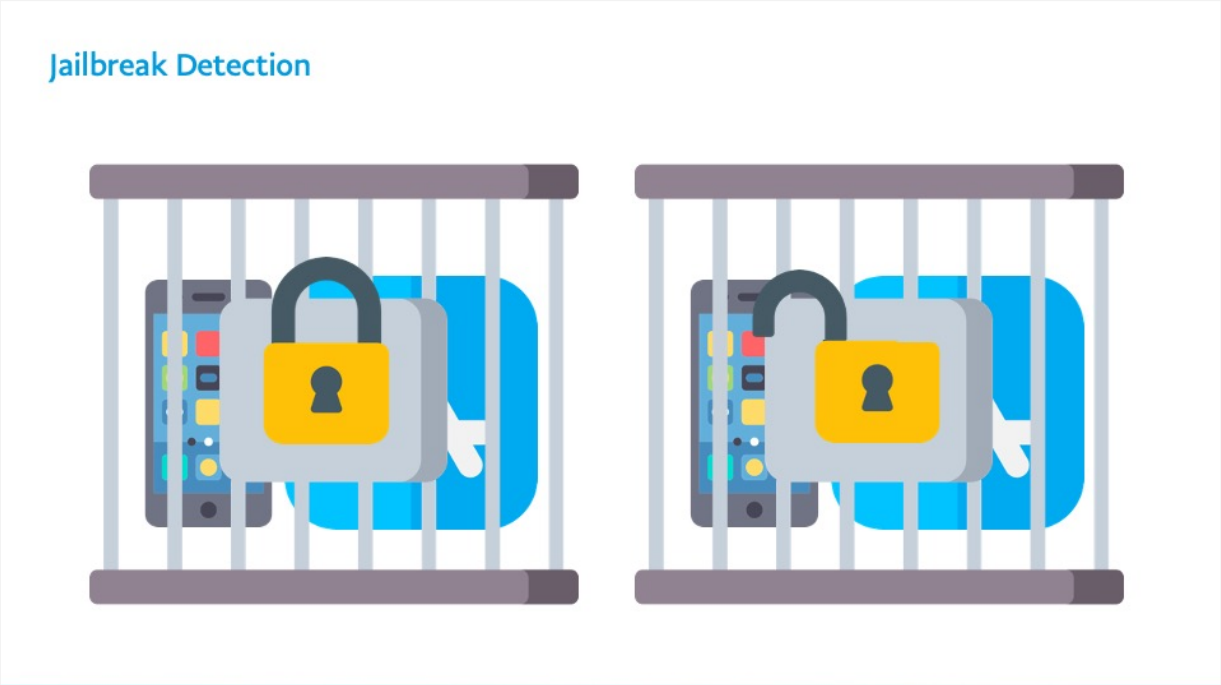
Fighting Code Tampering

Another method to protect against code tampering is to use **code obfuscation techniques** to make source code difficult for humans to understand. Employing a technique such as naming obfuscation where variable names are changed can mislead attackers and create confusion.

Other forms of obfuscation such as instruction substitution, control flow flattening, dead code injection, and string encryption can make your code obfuscation more complex and in-depth, making the reverse engineering process more time consuming.

There are many obfuscators available for Swift including SwiftShield.

Another method is **to move business decision logic and intellectual property code to server-side code**. This can be an extra proactive step to protecting the app's codebase.



Jailbreak detection mechanisms, when added to additional reverse engineering defenses, can increase the difficulty of running an app on a jailbroken device. Like other defenses mentioned in this section, jailbreak detection is not very effective on its own, but implementing checks throughout the app's source code can improve the effectiveness of anti-tampering defenses overall.

Jailbreak detection techniques for iOS include:

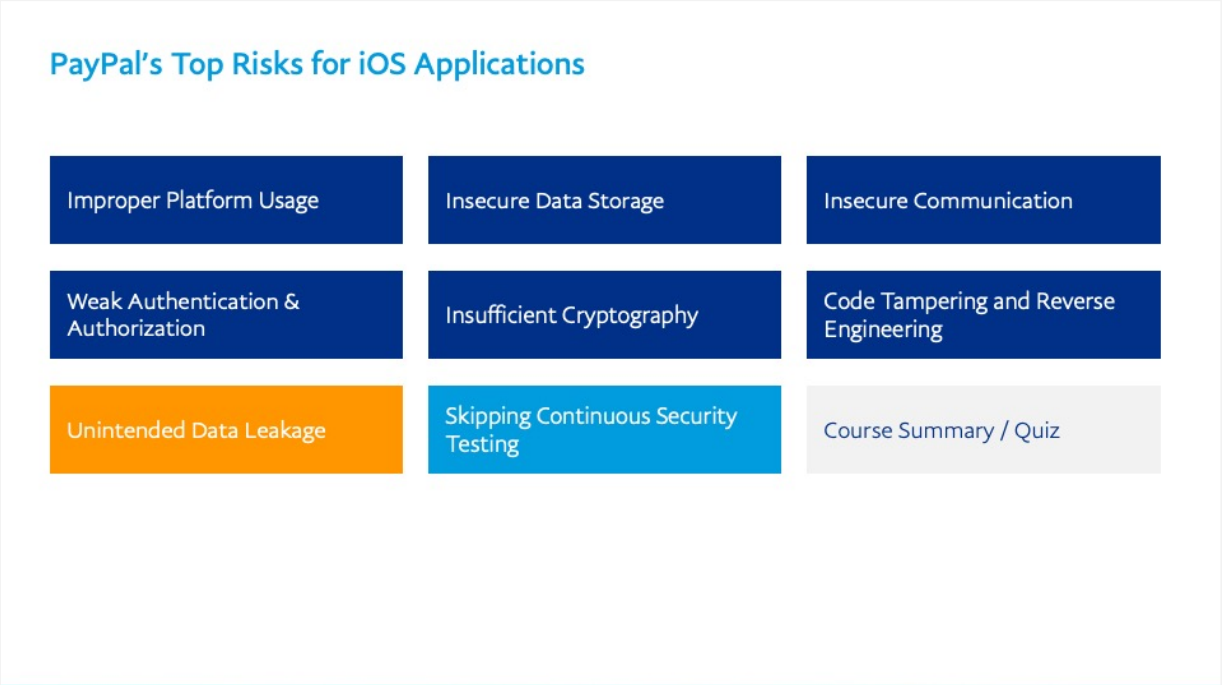
- File-based checks
- File permissions checks
- Protocol handler checks

In a file-based check, you can see if any new files or directories were added on the device.

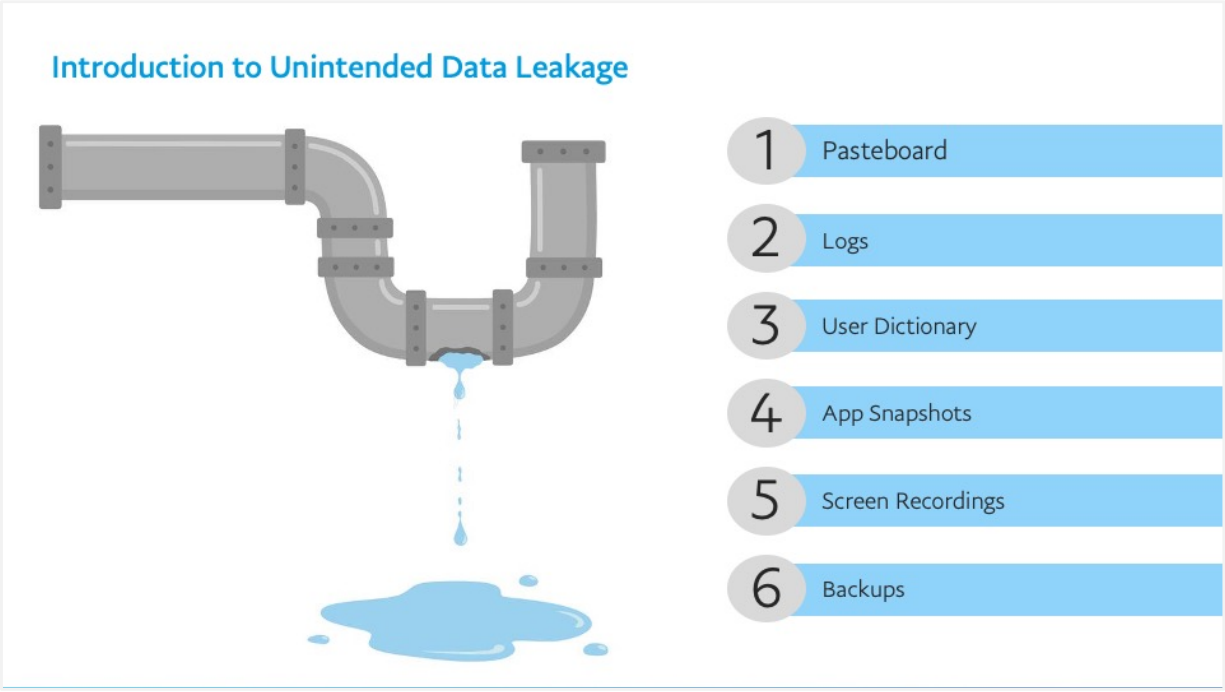
In a file permission check, if the app can modify files outside its sandbox this indicate the device is jailbroken. Many jailbroken devices will have Cydia installed, the unofficial App Store. During a protocol handler check, you can attempt to open a Cydia URL. If it is successful, the device is jailbroken.

In general, the more complicated the jailbreak detection is, the more difficult it is to detect and bypass. The most common mistake when implementing jailbreak detection often lies in the implementation itself.

In practice, the best jailbreak detection combines multiple techniques and integrates them into other functions so that they cannot easily be bypassed. Testing jailbreak detection to determine if it can be bypassed is another way to check on the effectiveness of these techniques.



Next up, we will review Unintended Data Leakage.

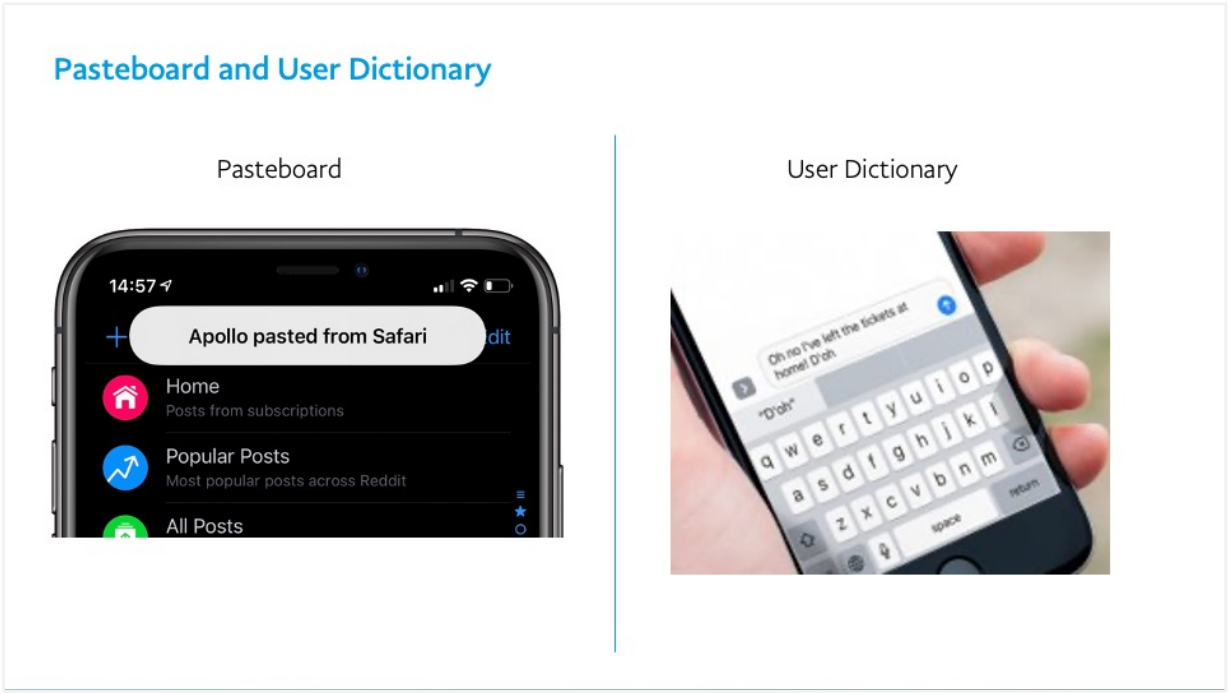


Compared to traditional web and desktop apps, mobile apps have unique data leakage behavior. Often the data leaking is unintentional and results from platform features available to users.

Developers must be aware of how these features can be misused, and must actively prevent the leaking of any sensitive data.

There are several main features to secure against data leakage:

1. Pasteboard
2. Logs
3. User dictionary
4. Background snapshots
5. Screen Recording
6. Backups



Pasteboard: This feature allows users to copy paste text across apps. However, if a user selects and copies sensitive information while using an app, like a credit card number, it will be saved to the pasteboard and become accessible to pasteboard monitoring apps installed on the phone. It can be a legitimate monitoring app or a malicious app that monitors the pasteboard without the user's knowledge. As of iOS 9, the pasteboard content is accessible to apps in the foreground only, which reduces the attack surface of password sniffing from the clipboard dramatically but is not a guaranteed solution.

Starting in iOS 10, the Find pasteboard (UIPasteboardNameFind constant) became unavailable, and persistent named pasteboards were deprecated. iOS sets the persistence of all pasteboards automatically, with named pasteboards marked as nonpersistent and the systemwide general pasteboard marked as persistent. If you try to set the setPersistent(_:) property on a pasteboard, Xcode issues a deprecation warning. Instead of persistent named pasteboards, developers should use app groups to create shared containers. In this case, a random string is generated to define the pasteboard. The ConcealedType data identifier is later added to instruct other apps to treat the content as confidential. You should also use the setItems(_:) method to set the expiration time and date for any copied data.

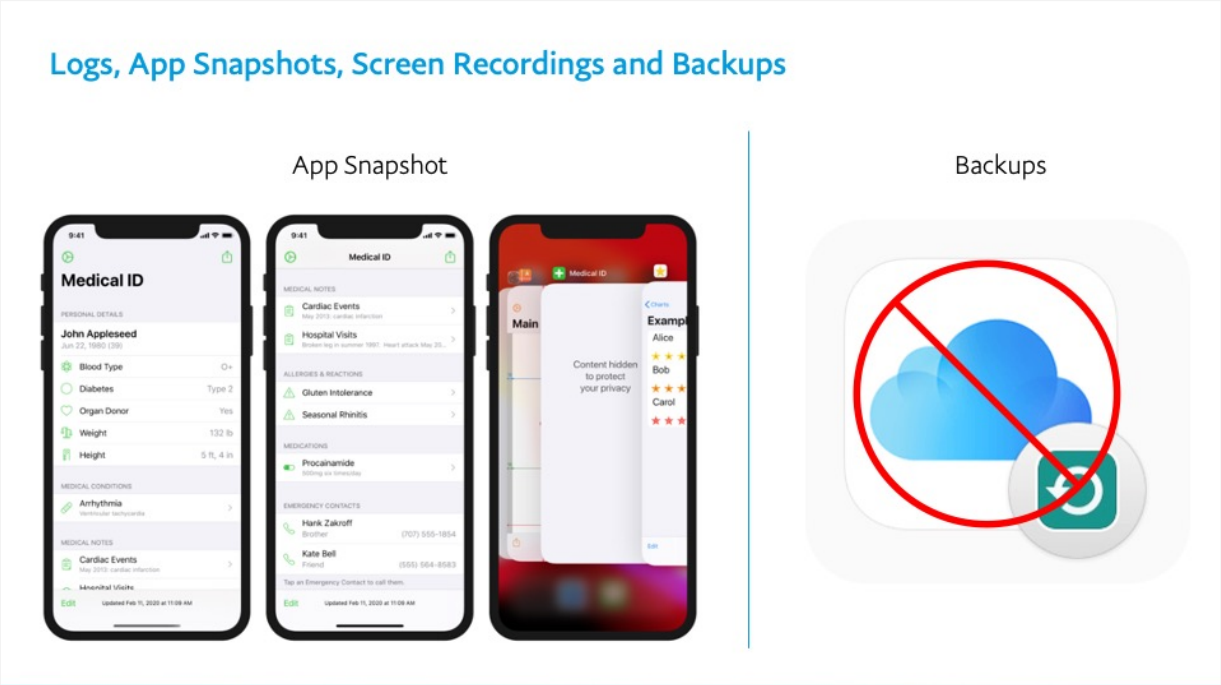
It is recommended that any item saved into the pasteboard be given an expiration date. Starting with iOS 14, iOS will display a notification every time an app accesses the device's pasteboard, bringing visibility to users about which apps could be collecting their data. Minimize the App's usage of the pasteboard. If the App uses the pasteboard to detect user-copied content such as website links, only check once per app foregrounding.

Additionally, make use of UIPasteboard's data type properties before reading its contents:

- hasURLs
- hasImages
- hasColors
- hasStrings

Accessing these properties does not cause an alert to be shown to the user.

User Dictionary: iOS tracks words typed by the user for improving the auto-correction feature of the phone and to provide the user with frequently used words. If a user types sensitive information in a text field, it can also get stored onto the user dictionary. This default iOS behavior can be turned off for individual input text field fields dealing with sensitive information by setting the autocorrectionType to UITextAutocorrectionTypeNo. As an additional defense, the keyboard selection can be restricted. Disable the use of any third-party keyboard completely.

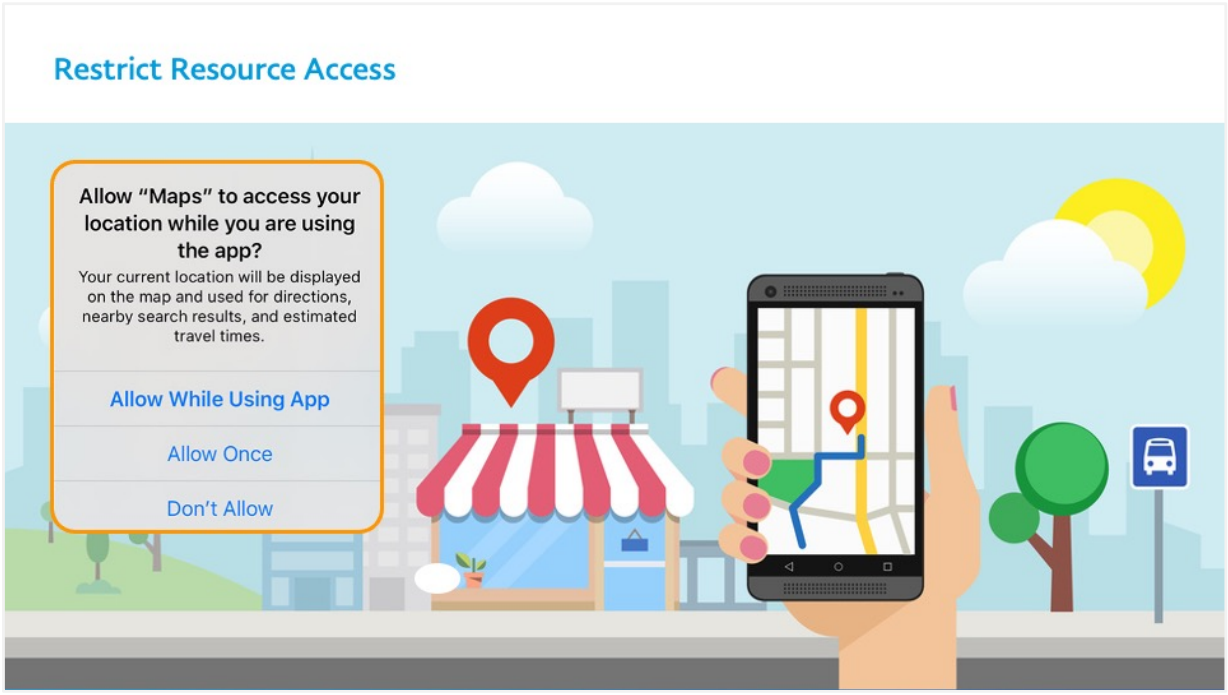


Logs: An iOS app may use a logging feature to make its debugging easier. NSLog in Objective-C and print/println in Swift may print information to application logs at the time of crash or any other event. Production-ready applications should make sure that no sensitive information is being logged into the application's logs

App Snapshot: iOS saves a screenshot of each app when it is backgrounded by the user. This screenshot is later used when the user is shown the app switching menu. If a PayPal app user presses the home button when the app is showing any sensitive information on the screen, it will get saved as a picture file. Setting the app snapshot as a white background can protect any sensitive information from being stored in the cache.
Use sceneDidEnterBackground method (for iOS13 and later) or applicationDidEnterBackground method (for iOS 12 and earlier) to set app snapshot to white background. In iOS, the screenshot gets saved in 'Library/Caches/Snapshots/<Bundle_Identifier>' directory.

Screen Recording: Screen recording also has the potential of exposing sensitive information. As developers for payment apps, we need to ensure any PCI or PII is not captured from the app.
Using UIScreen.capturedDidChangeNotification will send a notification when the captured status of the screen changes. The screen can then be hidden, and a warning can be created to let users know that screen recording is not permitted.

Backups: iTunes and iCloud backups will include keychain items unless they are explicitly excluded. This exclusion can be done with the help of the kSecAttrAccessible accessibility class.
Using kSecAttrAccessibleWhenUnlockedThisDeviceOnly will exclude data in this Keychain item from an iCloud or local backup. Valet, an open-source library, can also be used for storing and retrieving data from the Keychain.



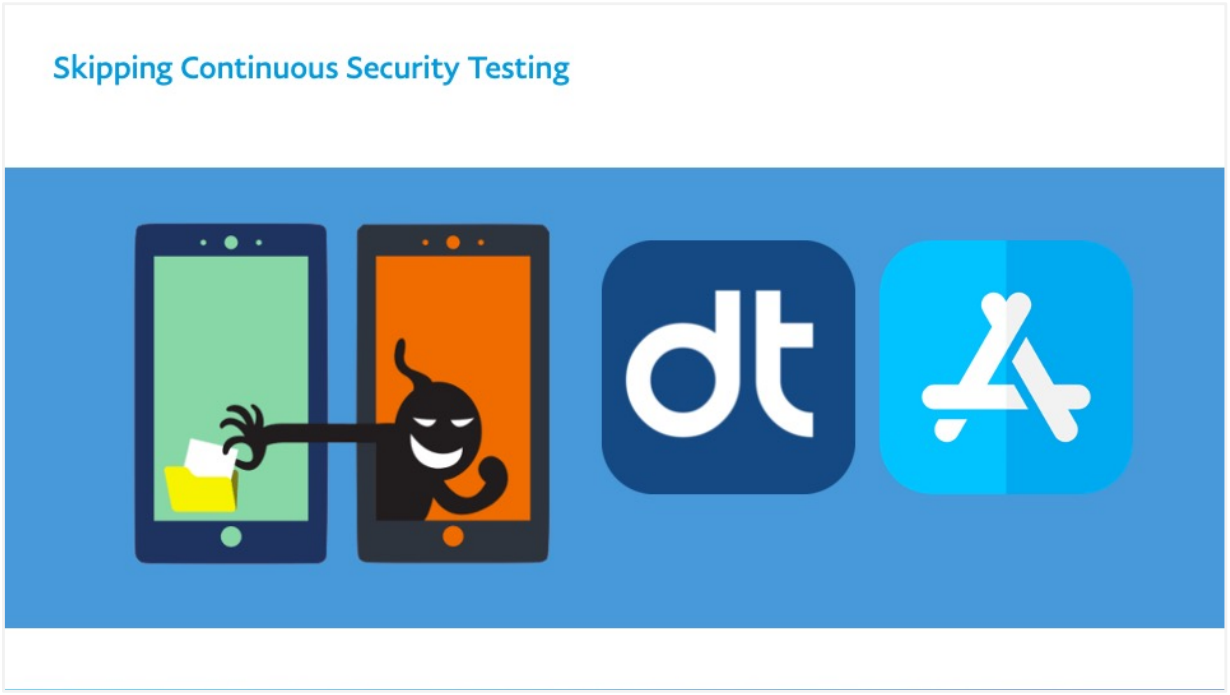
An important takeaway for reducing the risk of unintended data leakage is to also restrict what data the app accesses. Not all apps need access to all data collected by a mobile device. By limiting the scope of data accessed, the smaller the potential for exposure.

Apple's operating systems restrict access to protected data and has apps request access by prompting users with a permission request.

Developers can add a "purpose string" or usage description to explain why the app needs access to sensitive information which is added to the app's Information Property list (info.plist).

Additional authorization checks should also be enabled to ensure a user is authorized to receive the requested permission.

Next, we will look at Skipping Continuous Security Testing.



PayPal performs mobile security assessment using Data Theorem. It is used to scan iOS and Android applications on a continuous basis in search of security flaws and data privacy gaps.

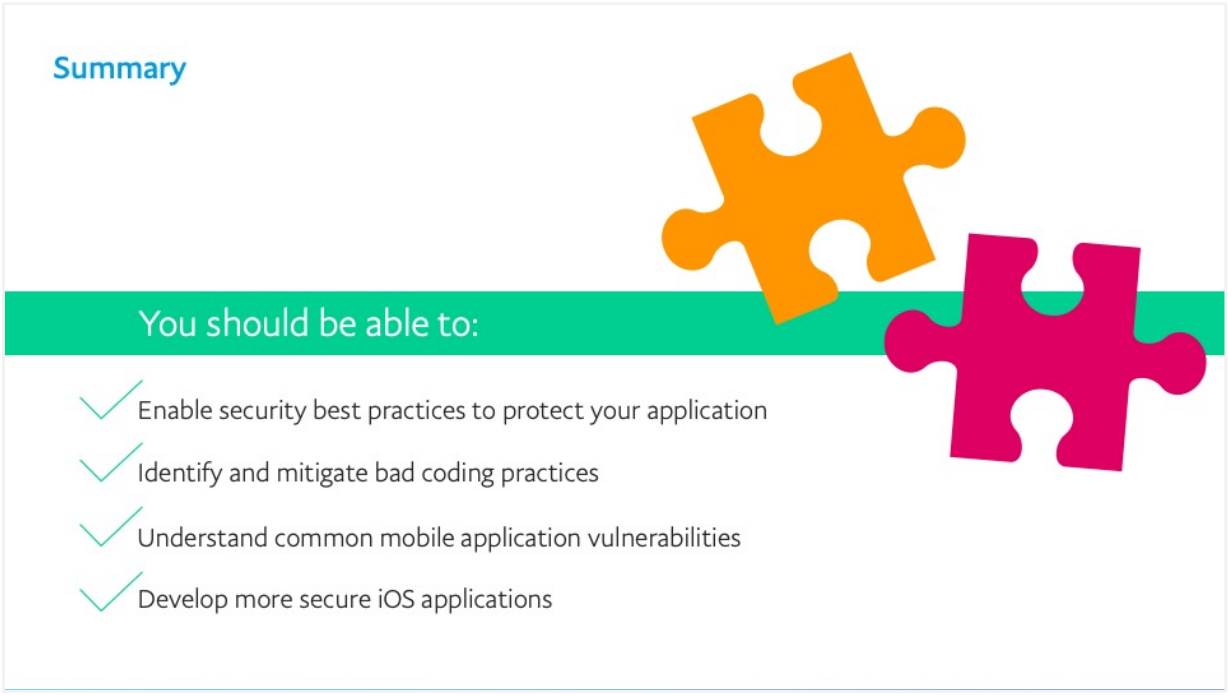
The scans include static analysis, dynamic analysis, and application logic analysis to uncover security flaws and data privacy gaps. Furthermore, the technology generates Objective-C, Java, and C# code that will help lock down the high-risk areas of our mobile application.

Scans should always be completed on Pre-Prod apps and features. This will help identify vulnerabilities in the app before its released to live.

Data Theorem also scans apps in the App Store on a daily basis to identify any newly introduced vulnerabilities. Development teams can login into the Data Theorem portal to validate the vulnerabilities that are reported.

All Security P1s need to be considered as top priority and remediation should be completed quickly. High priority issues determined by the App Store should be remediated as soon as possible to ensure your app is not blocked by Apple for violating policies.

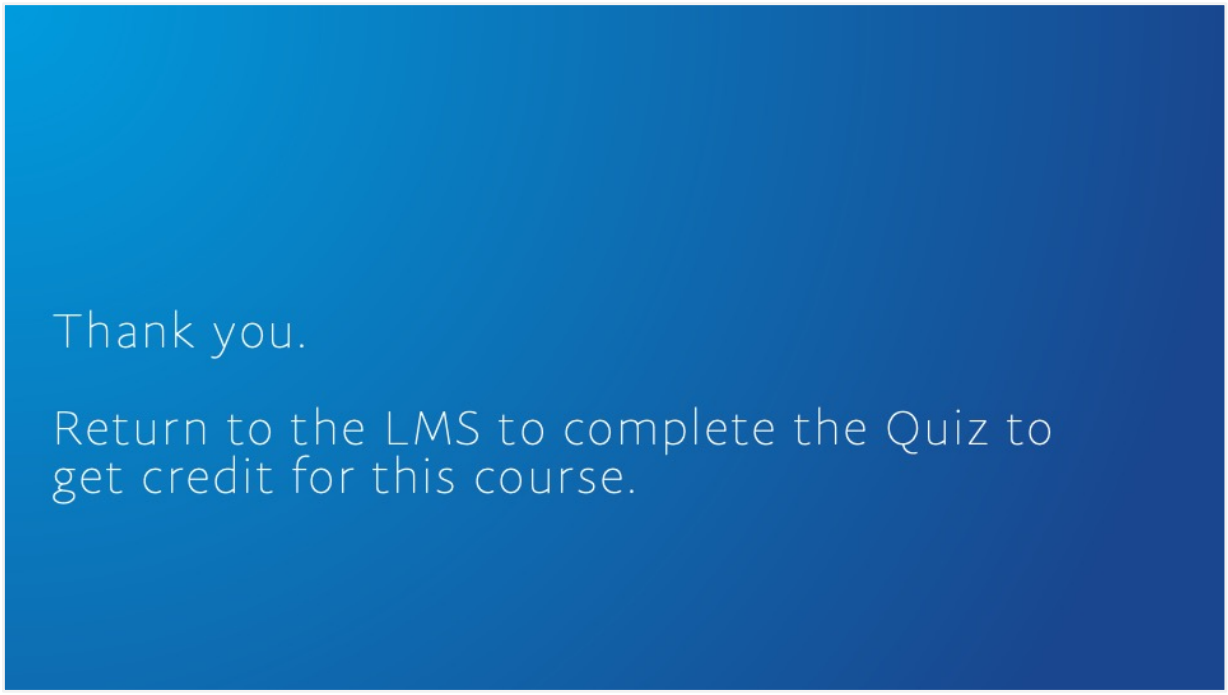
To onboard new apps into Data Theorem, please reach out help-appsec-scans@paypal.com.



Congratulations! You have reached the end of this training.

You should now be able to:

- Enable security best practices to protect your application
- Identify and mitigate bad coding practices
- Understand common mobile application vulnerabilities
- Develop more secure iOS applications



Thank you for taking the time to participate in this training.

To get credit for this course, you must return to the LMS and launch the quiz.