KonvergeAI

CCT 310 - 3
Application Security

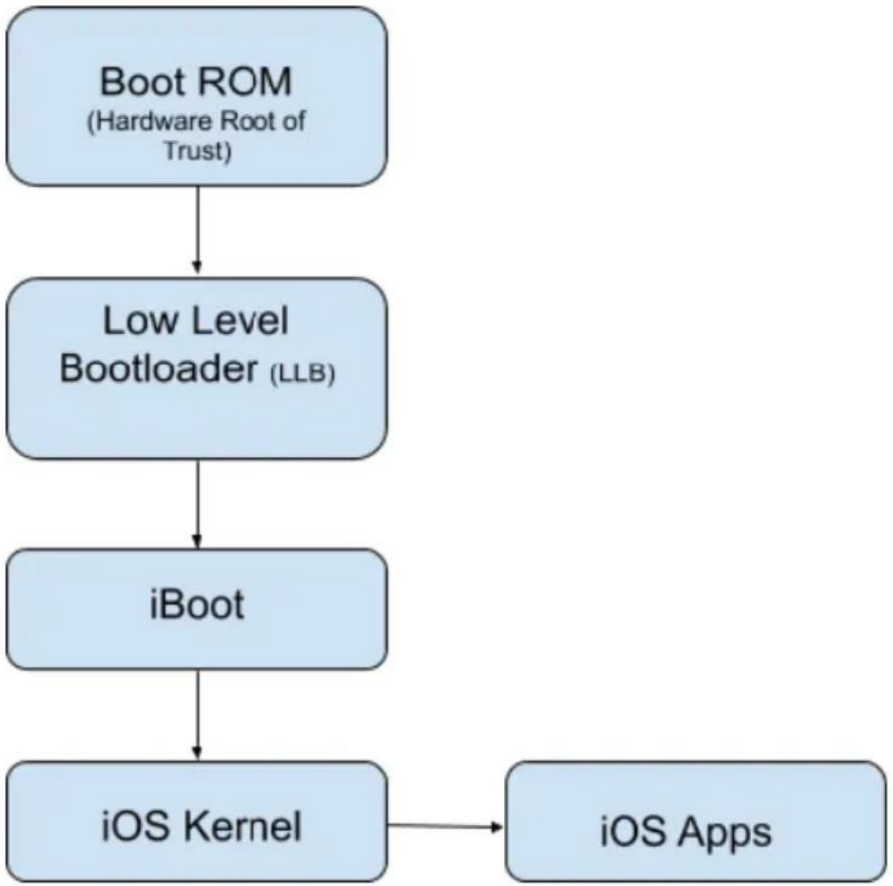
UNIT 6
iOS Application Security

iOS Security Model

1. System Security
2. Data Security
3. App Security

System Security

1. iOS Secure boot chain



when an iOS device is turned on, it immediately executes code from BOOT ROM which is a read-only memory, known as Hardware Root of trust, is laid down during chip fabrication, and is implicitly trusted. This also contains the Apple root certificate with public key and uses it to verify that the low-level boot loader is properly signed and has not been tampered before loading. LLB verifies the iBoot and iBoot verifies iOS kernel before starting it.

This process ensures lowest levels of software are not tampered and iOS running only on valid Apple devices.

2. System Software Authorization

Apple regularly releases software updates to address emerging security concerns and prevents devices being downgraded to older versions that lacks latest security updates.

3. SEP (Secure Enclave Processor)

Secure Enclave Processor is a co-processor fabricated within the system on chip. It runs its own OS, undergoes secure boot process separate from the rest of the device and receives its system updates independent of the other CPU components. The purpose of the SEP is to handle keys and other info such as bio-metrics and prevents main processor from gaining direct access to sensitive data.

4. Touch ID

Scans fingerprint and store mathematical representation of it in SEP.

5. Face ID

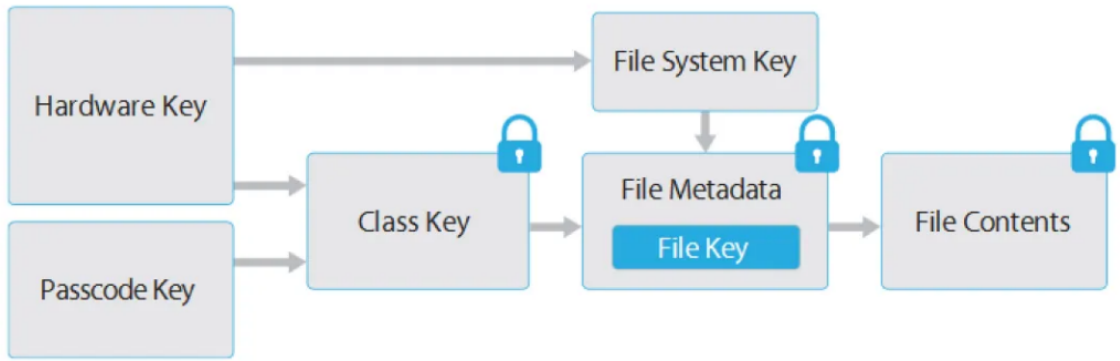
Uses True Depth camera system to accurately map the geometry of the face, use neural networks for determining attention, matching, and anti-spoofing. Data are digitally signed and sent to the SEP.

Data Security

1. Device ID and Group ID

Each device has its unique ID(UID) and a device group ID(GID) which are AES 256-bit keys compiled in to the application processor and SEP during manufacturing. No Software or hardware can access them directly. UID allows data to be tied to a particular device, hence if the memory chip is physically moved to another device, the encrypted files will not be accessible.

2. File Level Protection



iOS protects the file data by constructing and managing a hierarchy of keys in conjunction with hardware encryption engine. All keys are stored in SEP.

iOS protects the file data by constructing and managing a hierarchy of keys in conjunction with hardware encryption engine. All keys are stored in SEP.

3. Key chain data protection

The iOS Keychain can be used to securely store short, sensitive bits of data. eg: encryption keys and session tokens.

App Security

1. App code signing

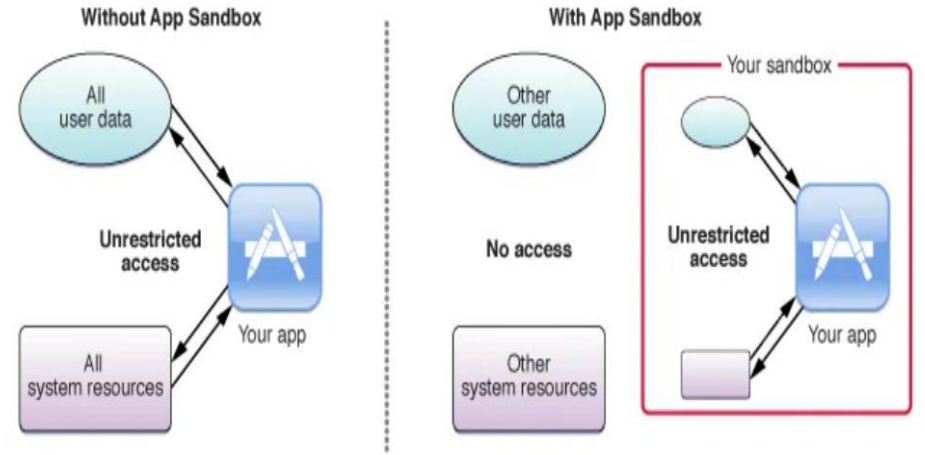
App code signing ensures that code is coming from a specific legitimate source/ developer (ensures ***authenticity***) and code has not been altered since it was signed.

2. App Updates

App updates are available to supported devices for security fixes and functionality enhancements.

3. App Sandbox

All third party apps are "***sandboxed***" and restricted from accessing files stored by other apps and making any changes to the device. Each app has got a "unique home directory" for its files and it is randomly assigned when app is installed.



Based on iOS 9.3, third party apps are located in

- */private/var/containers/Bundle/Application/<unique id>*
- */private/var/mobile/Containers/Data/Application/<unique id>*

Apple apps are located in */Application*

4. Run time process security

- **iOS Address Space Layout Randomization(ASLR)**

Primarily used to protect against buffer overflow attacks. Buffer overflows require an attacker to know where each part of the program is located in memory. ASLR randomizes the locations of different parts of the program in memory. Every time the program is run, components are moved to a different address in virtual memory. So attackers can no longer learn where their target is to inject malicious data in to the payload.

- **iOS Data Execution Prevention (DEP)**

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

- **Stack Smashing Protection**

Canary value is placed after the local variables to detect buffer overflows

- **Automatic Reference Counting**

ARC keeps track of class instances and decides when it's safe to deallocate the class instances it monitors. It does this by counting the references of each class instance.

5. App Store Review

Apple reviews apps before publish them in app store for users to download to ensure that apps are free of known malware and haven't been tampered with.

Debugging with LLDB

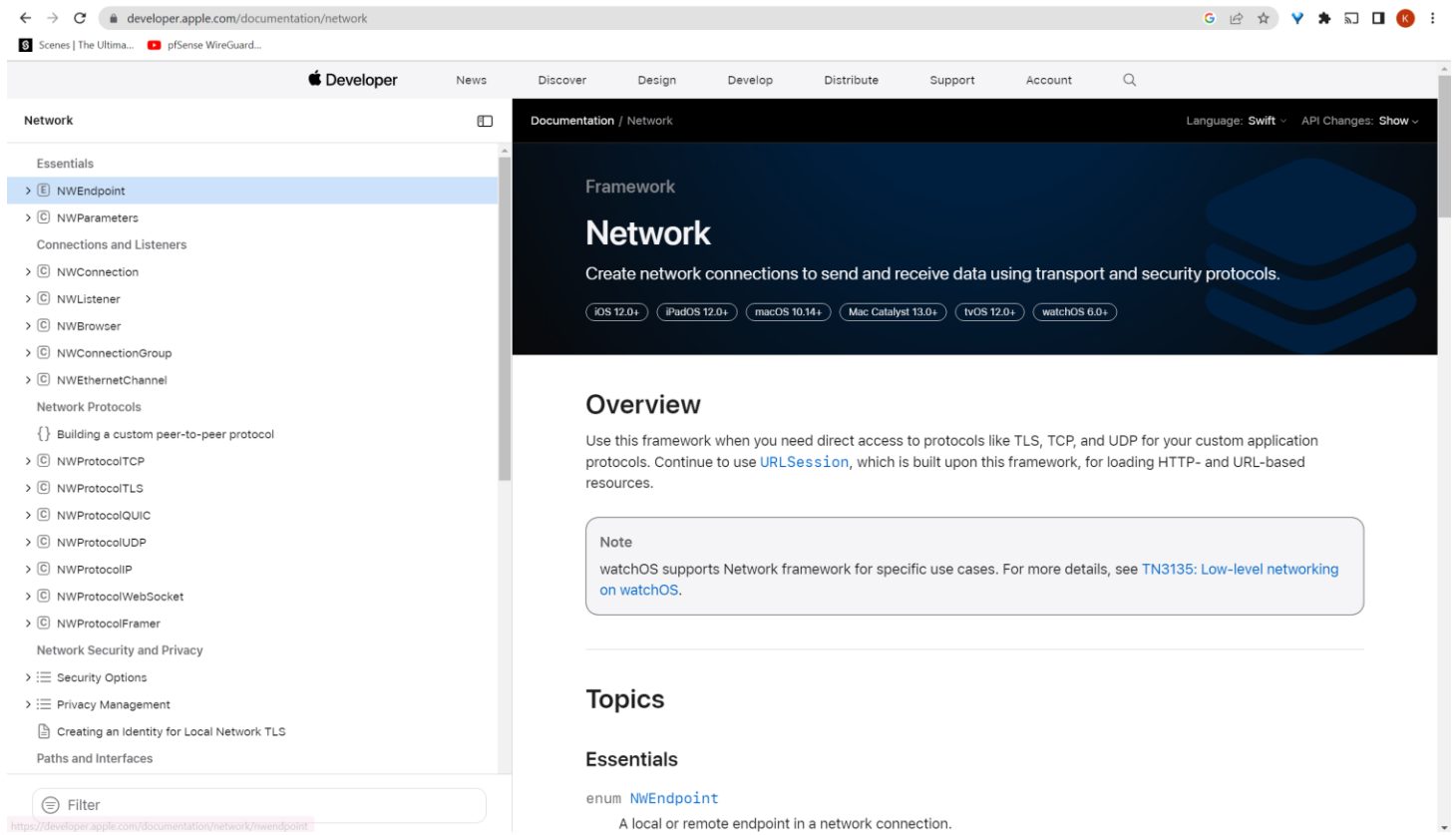
https://lldb.llvm.org/use/tutorial.html

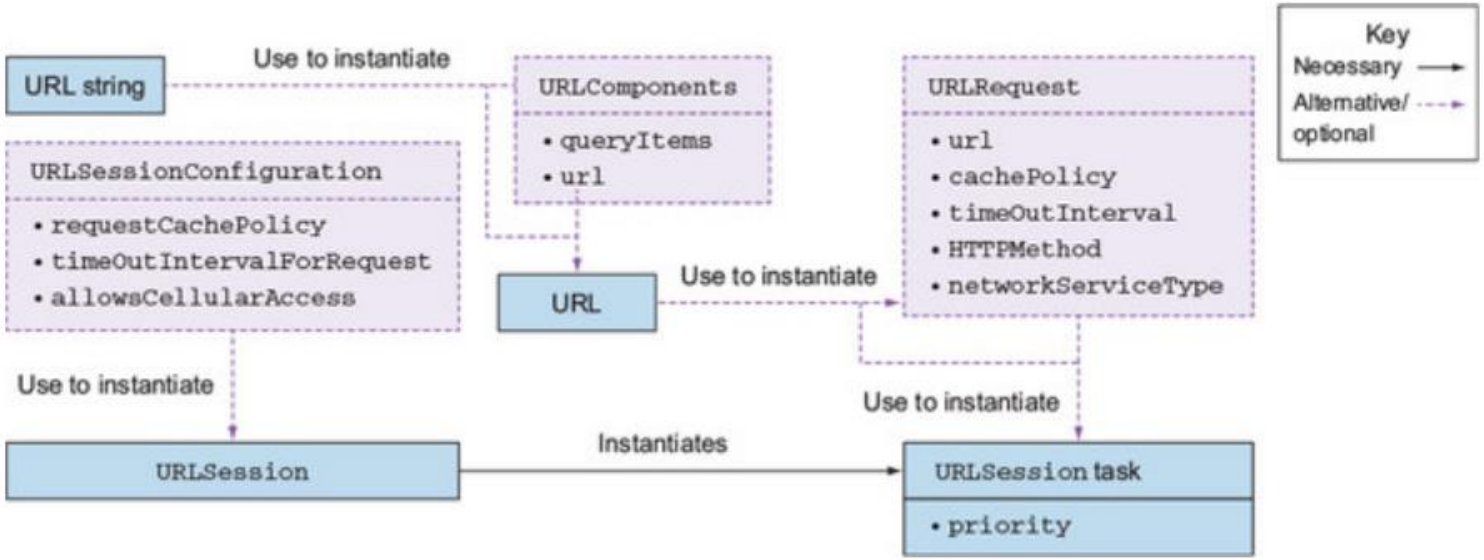
https://opensource.apple.com/source/lldb/lldb-310.2.36/www/tutorial.html

https://www.youtube.com/watch?v=v_C1cvo1bil

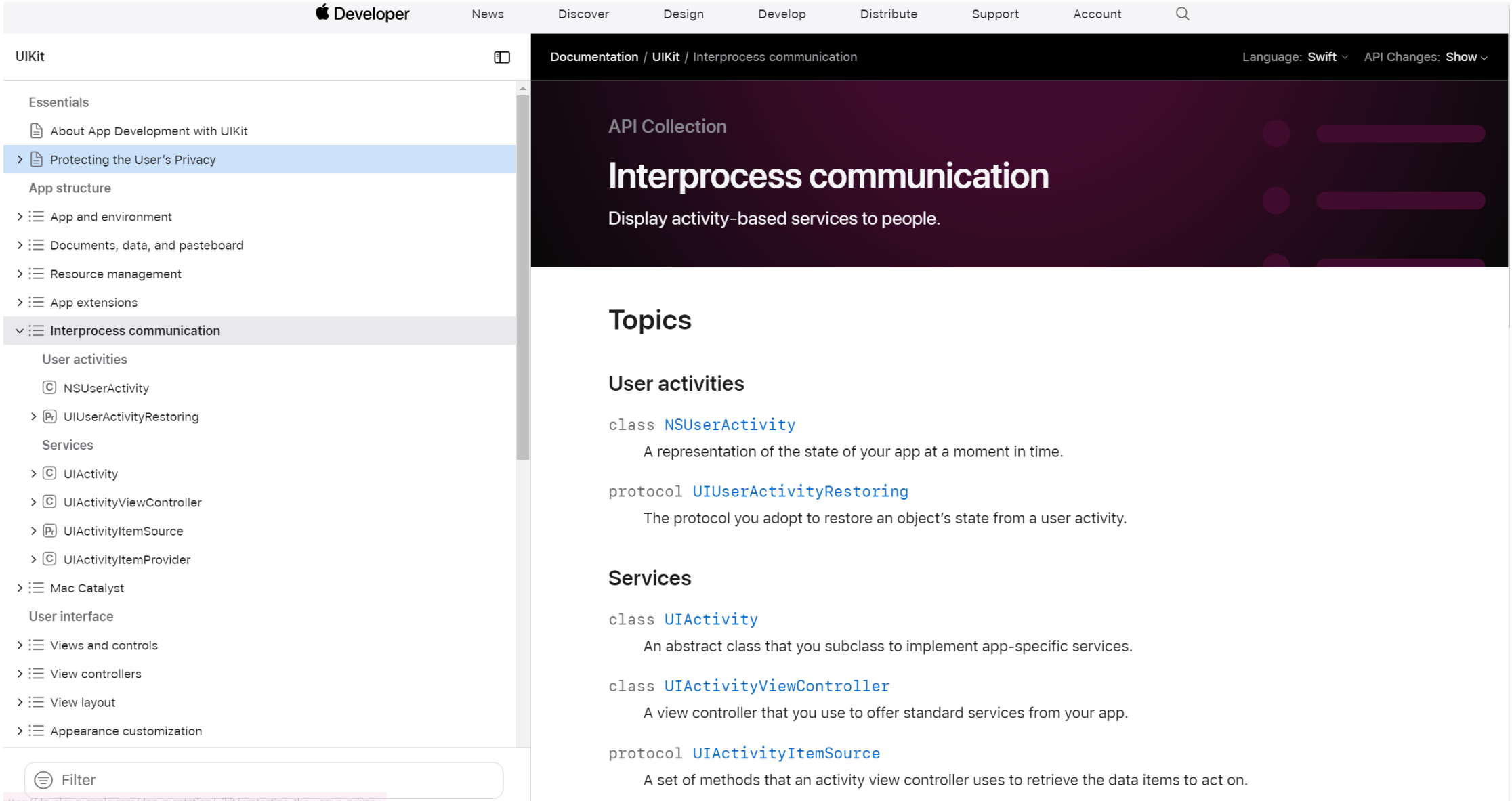
Networking & Interprocess Communication

- Connecting to web services
- Requesting data from a web service
- Downloading from a web service
- Parsing JSON data
- Using dependency managers





Communicating to a to a Web Service Example


Developer
News
Discover
Design
Develop
Distribute
Support
Account
UIKit
Documentation / UIKit / Interprocess communication
Language: Swift
API Changes: Show
Essentials
About App Development with UIKit
Protecting the User's Privacy
App structure
App and environment
Documents, data, and pasteboard
Resource management
App extensions
Interprocess communication
User activities
NSUserActivity
UIUserActivityRestoring
Services
UIActivity
UIActivityViewController
UIActivityItemSource
UIActivityItemProvider
Mac Catalyst
User interface
Views and controls
View controllers
View layout
Appearance customization
Filter
API Collection
Interprocess communication
Display activity-based services to people.
Topics
User activities
class NSUserActivity
A representation of the state of your app at a moment in time.
protocol UIUserActivityRestoring
The protocol you adopt to restore an object's state from a user activity.
Services
class UIActivity
An abstract class that you subclass to implement app-specific services.
class UIActivityViewController
A view controller that you use to offer standard services from your app.
protocol UIActivityItemSource
A set of methods that an activity view controller uses to retrieve the data items to act on.

A Failed IPC Hack: The Pasteboard

There have been occasional reports of people abusing the `UIPasteboard` mechanism as a kind of IPC channel. For example, some try using it to transfer a user's data from a free version of an application to a "pro" version, since there's no way the newly installed application can read the old application's data. Don't do that!

An OAuth library designed to work with Twitter[3] uses the general paste-board as a mechanism to shuffle authentication information from a web view to the main part of the app, as in this example:

```
- (void) pasteboardChanged: (NSNotification *) note {
❶   UIPasteboard *pb = [UIPasteboard generalPasteboard];

    if ([note.userInfo objectForKey:UIPasteboardChangedTypesAddedKey] == nil)
     return;
    NSString *copied = pb.string;

    if (copied.length != 7 || !copied.oauthtwitter_isNumeric) return;
❷   [self gotPin:copied];
}
```

After reading data from the general pasteboard at ❶, this library validates the data and sends it to the `gotPin` method at ❷.

But the general pasteboard is shared among all applications and can be read by any process on the device. This makes the pasteboard a particularly bad place to store anything even resembling private data.

Data Leakage & Injection Attacks

M4: Unintended Data Leakage

Threat Agents

Application Specific

Agents that may exploit this vulnerability include the following: mobile malware, modified versions of legitimate apps, or an adversary that has physical access to the victim's mobile device.

Attack Vectors

Exploitability EASY

An agent that has physical access to the device will use freely available forensic tools to conduct the attack. An agent that has access to the device via malicious code will use fully permissible and documented API calls to conduct this attack.

Security Weakness

Prevalence COMMON

Detectability EASY

Unintended data leakage occurs when a developer inadvertently places sensitive information or data in a location on the mobile device that is easily accessible by other apps on the device. First, a developer's code processes sensitive information supplied by the user or the backend. During that processing, a side-effect (that is unknown to the developer) results in that information being placed into an insecure location on the mobile device that other apps on the device may have open access to. Typically, these side-effects originate from the underlying mobile device's operating system (OS). This will be a very prevalent vulnerability for code produced by a developer that does not have intimate knowledge of how that information can be stored or processed by the underlying OS. It is easy to detect data leakage by inspecting all mobile device locations that are accessible to all apps for the app's sensitive information.

Technical Impacts

Impact SEVERE

This vulnerability may result in the following technical impacts: extraction of the app's sensitive information via mobile malware, modified apps, or forensic tools.

Business Impacts

Application / Business Specific

The nature of the business impact is highly dependent upon the nature of the information stolen. Sensitive information theft may result in the following business impacts:

- Privacy Violations
- PCI Violations
- Reputational Damage; or
- Fraud.

Am I Vulnerable To ‘Unintended Data Leakage’?

Unintended data leakage (formerly side-channel data leakage) includes vulnerabilities from the OS, frameworks, compiler environment, new hardware, etc. without a developers knowledge.

In mobile development, this is most seen in undocumented (or under-documented) internal processes such as:

- The way the OS caches data, images, key-presses, logging, and buffers.
- The way the development framework caches data, images, key-presses, logging, and buffers.
- The way or amount of data ad, analytic, social, or enablement frameworks cache data, images, key-presses, logging, and buffers.

How Do I Prevent ‘Unintended Data Leakage’?

It is important to threat model your OS, platforms, and frameworks, to see how they handle the following types of features:

- URL Caching (Both request and response)
- Keyboard Press Caching
- Copy/Paste buffer Caching
- Application backgrounding
- Logging
- HTML5 data storage
- Browser cookie objects
- Analytics data sent to 3rd parties

It is especially important to discern what a given OS or framework does by default. By identifying defaults and applying mitigating controls, you can avoid unintended data leakage.

Example Attack Scenarios

iOS

- URL Caching (Both request and response)
- Keyboard Press Caching
- Copy/Paste buffer Caching
- Application backgrounding
- Logging
- HTML5 data storage
- Browser cookie objects
- Analytics data sent to 3rd parties

SQL Injection

Client-side SQL injection results from parsing externally supplied data that injects valid SQL into a badly formed SQL statement. Statements that are constructed dynamically on execution, using unsanitized, externally supplied input, are vulnerable to SQL injection. Malicious input will contain SQL metacharacters and statements that subvert the intent of the original query.

For example, imagine a simple status message is posted to a website by a user. It then gets downloaded and added to a local data store. If the user posting the original content has basic security knowledge and malicious intent, the user could embed SQL into the message, which will be executed when parsed by the SQL engine. This malicious SQL could destroy or modify existing data in the data store.

On iOS, the most commonly used SQL API is SQLite. **Listing 12-1** shows an example of an incorrectly formed, dynamically constructed SQL statement for SQLite.

```
NSString *uid = [myHTTPConnection getUID];
NSString *statement = [NSString StringWithFormat:@"SELECT username FROM
users where
    uid = '%@'",uid];
const char *sql = [statement UTF8String];
```

Listing 12-1: An unparameterized SQL statement vulnerable to SQL injection

The problem here is that the `uid` value is being taken from user-supplied input and inserted as is into a SQL statement using a format string. Any SQL in the user-supplied parameter will then become part of that statement when it ultimately gets executed.

To prevent SQL injection, simply use parameterized statements to avoid the dynamic construction of SQL statements in the first place. Instead of constructing the statement dynamically and passing it to the SQL parser, a parameterized statement causes the statement to be evaluated and compiled independently of the parameters. The parameters themselves are supplied to the compiled statement upon execution.

```
  static sqlite3_stmt *selectUid = nil;
❶ const char *sql = "SELECT username FROM users where uid = ?";
❷ sqlite3_prepare_v2(db, sql, -1, &selectUid, NULL);
❸ sqlite3_bind_int(selectUid, 1, uid);
  int status = sqlite3_step(selectUid);
```

Listing 12-2: A properly parameterized SQL statement

The SQL statement is constructed with the `?` placeholder at ❶. The code then compiles the SQL statement with `sqlite3_prepare_v2` at ❷ and lastly binds the user-supplied `uid` using `sqlite3_bind_int` at ❸. Since the SQL statement has already been constructed, no additional SQL provided in the `uid` parameter will be added to the SQL itself; it's simply passed in by value.

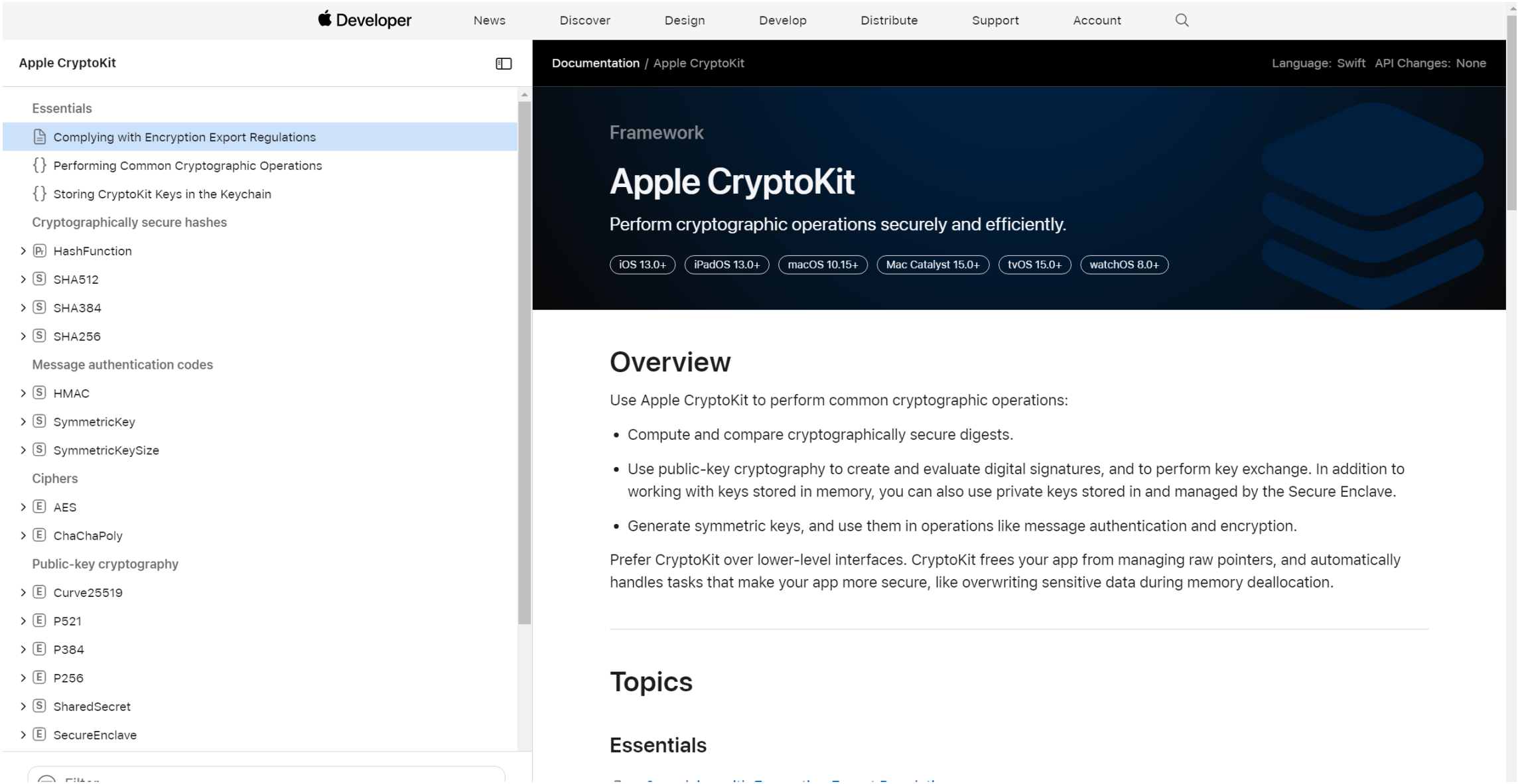
In addition to preventing SQL injection, using parameterized, prepared statements will improve application performance under most circumstances. You should use them for all SQL statements, even if a statement isn't taking input from untrusted sources.

Encryption & Authentication

Data protection is an iOS feature that you use to secure your app's files and prevent unauthorized access to them. Data protection is enabled automatically when the user sets an active passcode for the device. You read and write your files normally, but the system encrypts and decrypts your content behind the scenes. The encryption and decryption processes are automatic and hardware accelerated.

You specify the level of data protection that you want to apply to each of your files. There are four levels available, each of which determines when you may access the file. If you do not specify a protection level when creating a file, iOS applies the default protection level automatically.

- **No protection.** The file is always accessible.
- **Complete until first user authentication.** (Default) The file is inaccessible until the first time the user unlocks the device. After the first unlocking of the device, the file remains accessible until the device shuts down or reboots.
- **Complete unless open.** You can open existing files only when the device is unlocked. If you have a file already open, you may continue to access that file even after the user locks the device. You can also create new files and access them while the device is locked or unlocked.
- **Complete.** The file is accessible only when the device is unlocked.


Developer
News
Discover
Design
Develop
Distribute
Support
Account
Apple CryptoKit
Documentation / Apple CryptoKit
Language: Swift API Changes: None
Essentials
Complying with Encryption Export Regulations
Performing Common Cryptographic Operations
Storing CryptoKit Keys in the Keychain
Cryptographically secure hashes
HashFunction
SHA512
SHA384
SHA256
Message authentication codes
HMAC
SymmetricKey
SymmetricKeySize
Ciphers
AES
ChaChaPoly
Public-key cryptography
Curve25519
P521
P384
P256
SharedSecret
SecureEnclave
Framework
Apple CryptoKit
Perform cryptographic operations securely and efficiently.
iOS 13.0+
iPadOS 13.0+
macOS 10.15+
Mac Catalyst 15.0+
tvOS 15.0+
watchOS 8.0+
Overview
Use Apple CryptoKit to perform common cryptographic operations:
Compute and compare cryptographically secure digests.
Use 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.
Generate symmetric keys, and use them in operations like message authentication and encryption.
Prefer 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.
Topics
Essentials

Framework

Authentication Services

Make it easy for users to log into apps and services.

iOS 12.0+ iPadOS 12.0+ macOS 10.15+ Mac Catalyst 13.0+ tvOS 13.0+ watchOS 6.0+

Overview

Use the Authentication Services framework to improve the experience of users when they enter credentials to establish their identity.

- Give users the ability to sign into your services with their Apple ID.
- Enable users to look up their stored passwords from within the sign-in flow of an app.
- Provide a passwordless registration and authentication workflow for apps and websites using iCloud Keychain or a physical security key.
- Perform automatic security upgrades from weak to strong passwords, or upgrade to using Sign in with Apple.
- Share data between an app and a web browser using technologies like OAuth to leverage existing web-based logins in the app.
- Create a single sign-on (SSO) experience in an enterprise app.

Simple and straightforward sign-up and sign-in flows reduce the burden on the user to remember passwords, which improves security.

Mobile Privacy Concerns

Dangers of Unique Device Identifiers

iOS's *unique device identifiers (UDIDs)* stand as something of a cautionary tale. For most of iOS's history, the UDID was used to uniquely identify an individual iOS device, which many applications then used to track user activity or associate a user ID with particular hardware. Some companies used these identifiers as access tokens to remote services, which turned out to be a spectacularly bad idea.

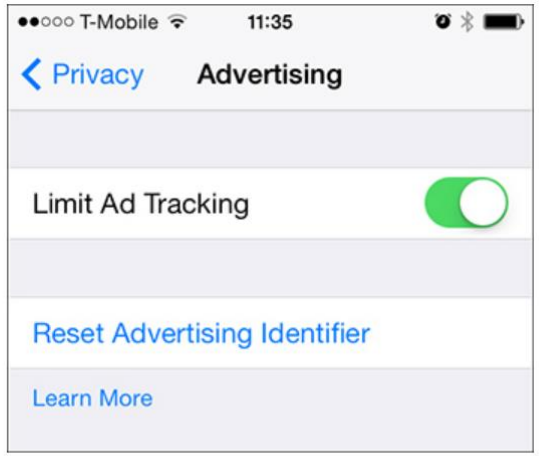


Figure 14-1: The user interface for indicating that the `advertisingIdentifier` *should be used for limited purposes*

The Risks of Storing Location Data

Few aspects of mobile privacy have generated as much negative press as tracking users via geolocation data. While useful for an array of location-aware services, a number of issues arise when location data is recorded and stored over time. Most obvious are privacy concerns: users may object to their location data being stored long-term and correlated with other personal information.[4] Aside from PR concerns, some European countries have strict privacy and data protection laws, which must be taken into account.

Managing Health and Motion Information

Some of the most sensitive information that applications can handle is health information about the user. On iOS, this data can be retrieved using the HealthKit API and the APIs provided by the device's M7 motion processor, if it has one. You'll take a brief look at how to read and write this data and how to request the minimum privileges necessary for an app to function.