# iOS Pentesting

## iOS Basics

[PAGEiOS Basics](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/ios-basics)

## Testing Environment

In this page you can find information about the **iOS simulator**, **emulators** and **jailbreaking:**

[PAGEiOS Testing Environment](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/ios-testing-environment)

## Initial Analysis

### Basic iOS Testing Operations

During the testing **several operations are going to be suggested** (connect to the device, read/write/upload/download files, use some tools...). Therefore, if you don't know how to perform any of these actions please, **start reading the page**:

[PAGEiOS Basic Testing Operations](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/basic-ios-testing-operations)

For the following steps **the app should be installed** in the device and should have already obtained the **IPA file** of the application. Read the [Basic iOS Testing Operations](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/basic-ios-testing-operations) page to learn how to do this.

### Basic Static Analysis

It's recommended to use the tool [**MobSF**](https://github.com/MobSF/Mobile-Security-Framework-MobSF) to perform an automatic Static Analysis to the IPA file.

Identification of **protections are present in the binary**:

* **PIE (Position Independent Executable)**: When enabled, the application loads into a random memory address every-time it launches, making it harder to predict its initial memory address.

otool -hv <app-binary> | grep PIE # It should include the PIE flag

* **Stack Canaries**: To validate the integrity of the stack, a ‘canary’ value is placed on the stack before calling a function and is validated again once the function ends.

otool -I -v <app-binary> | grep stack\_chk # It should include the symbols: stack\_chk\_guard and stack\_chk\_fail

* **ARC (Automatic Reference Counting)**: To prevent common memory corruption flaws

otool -I -v <app-binary> | grep objc\_release # It should include the \_objc\_release symbol

* **Encrypted Binary**: The binary should be encrypted

otool -arch all -Vl <app-binary> | grep -A5 LC\_ENCRYPT # The cryptid should be 1

**Identification of Sensitive/Insecure Funcions**

* **Weak Hashing Algorithms**

# On the iOS device

otool -Iv <app> | grep -w "\_CC\_MD5"

otool -Iv <app> | grep -w "\_CC\_SHA1"

# On linux

grep -iER "\_CC\_MD5"

grep -iER "\_CC\_SHA1"

* **Insecure Random Functions**

# On the iOS device

otool -Iv <app> | grep -w "\_random"

otool -Iv <app> | grep -w "\_srand"

otool -Iv <app> | grep -w "\_rand"

# On linux

grep -iER "\_random"

grep -iER "\_srand"

grep -iER "\_rand"

* **Insecure ‘Malloc’ Function**

# On the iOS device

otool -Iv <app> | grep -w "\_malloc"

# On linux

grep -iER "\_malloc"

* **Insecure and Vulnerable Functions**

# On the iOS device

otool -Iv <app> | grep -w "\_gets"

otool -Iv <app> | grep -w "\_memcpy"

otool -Iv <app> | grep -w "\_strncpy"

otool -Iv <app> | grep -w "\_strlen"

otool -Iv <app> | grep -w "\_vsnprintf"

otool -Iv <app> | grep -w "\_sscanf"

otool -Iv <app> | grep -w "\_strtok"

otool -Iv <app> | grep -w "\_alloca"

otool -Iv <app> | grep -w "\_sprintf"

otool -Iv <app> | grep -w "\_printf"

otool -Iv <app> | grep -w "\_vsprintf"

# On linux

grep -R "\_gets"

grep -iER "\_memcpy"

grep -iER "\_strncpy"

grep -iER "\_strlen"

grep -iER "\_vsnprintf"

grep -iER "\_sscanf"

grep -iER "\_strtok"

grep -iER "\_alloca"

grep -iER "\_sprintf"

grep -iER "\_printf"

grep -iER "\_vsprintf"

### Basic Dynamic Analysis

Check out the dynamic analysis that [**MobSF**](https://github.com/MobSF/Mobile-Security-Framework-MobSF) perform. You will need to navigate through the different views and interact with them but it will be hooking several classes on doing other things and will prepare a report once you are done.

### Listing Installed Apps

Use the command frida-ps -Uai to determine the **bundle identifier** of the installed apps:

$ frida-ps -Uai

PID Name Identifier

---- ------------------- -----------------------------------------

6847 Calendar com.apple.mobilecal

6815 Mail com.apple.mobilemail

- App Store com.apple.AppStore

- Apple Store com.apple.store.Jolly

- Calculator com.apple.calculator

- Camera com.apple.camera

- iGoat-Swift OWASP.iGoat-Swift

### Basic Enumeration & Hooking

Learn how to **enumerate the components of the application** and how to easily **hook methods and classes** with objection:

[PAGEiOS Hooking With Objection](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/ios-hooking-with-objection)

### IPA Structure

The structure of an **IPA file** is essentially that of a **zipped package**. By renaming its extension to .zip, it can be **decompressed** to reveal its contents. Within this structure, a **Bundle** represents a fully packaged application ready for installation. Inside, you will find a directory named <NAME>.app, which encapsulates the application's resources.

* **Info.plist**: This file holds specific configuration details of the application.
* **\_CodeSignature/**: This directory includes a plist file that contains a signature, ensuring the integrity of all files in the bundle.
* **Assets.car**: A compressed archive that stores asset files like icons.
* **Frameworks/**: This folder houses the application's native libraries, which may be in the form of .dylib or .framework files.
* **PlugIns/**: This may include extensions to the application, known as .appex files, although they are not always present. \* [**Core Data**](https://developer.apple.com/documentation/coredata): It is used to save your application’s permanent data for offline use, to cache temporary data, and to add undo functionality to your app on a single device. To sync data across multiple devices in a single iCloud account, Core Data automatically mirrors your schema to a CloudKit container.
* [**PkgInfo**](https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPRuntimeConfig/Articles/ConfigApplications.html): The PkgInfo file is an alternate way to specify the type and creator codes of your application or bundle.
* **en.lproj, fr.proj, Base.lproj**: Are the language packs that contains resources for those specific languages, and a default resource in case a language isn' t supported.
* **Security**: The \_CodeSignature/ directory plays a critical role in the app's security by verifying the integrity of all bundled files through digital signatures.
* **Asset Management**: The Assets.car file uses compression to efficiently manage graphical assets, crucial for optimizing application performance and reducing its overall size.
* **Frameworks and PlugIns**: These directories underscore the modularity of iOS applications, allowing developers to include reusable code libraries (Frameworks/) and extend app functionality (PlugIns/).
* **Localization**: The structure supports multiple languages, facilitating global application reach by including resources for specific language packs.

**Info.plist**

The **Info.plist** serves as a cornerstone for iOS applications, encapsulating key configuration data in the form of **key-value** pairs. This file is a requisite for not only applications but also for app extensions and frameworks bundled within. It's structured in either XML or a binary format and holds critical information ranging from app permissions to security configurations. For a detailed exploration of available keys, one can refer to the [**Apple Developer Documentation**](https://developer.apple.com/documentation/bundleresources/information_property_list?language=objc).

For those looking to work with this file in a more accessible format, the XML conversion can be achieved effortlessly through the use of plutil on macOS (available natively on versions 10.2 and later) or plistutil on Linux. The commands for conversion are as follows:

* **For macOS**:

$ plutil -convert xml1 Info.plist

* **For Linux**:

$ apt install libplist-utils

$ plistutil -i Info.plist -o Info\_xml.plist

Among the myriad of information that the **Info.plist** file can divulge, notable entries include app permission strings (UsageDescription), custom URL schemes (CFBundleURLTypes), and configurations for App Transport Security (NSAppTransportSecurity). These entries, along with others like exported/imported custom document types (UTExportedTypeDeclarations / UTImportedTypeDeclarations), can be effortlessly located by inspecting the file or employing a simple grep command:

$ grep -i <keyword> Info.plist

**Data Paths**

In the iOS environment, directories are designated specifically for **system applications** and **user-installed applications**. System applications reside in the /Applications directory, while user-installed apps are placed under /var/mobile/containers/Data/Application/. These applications are assigned a unique identifier known as a **128-bit UUID**, making the task of manually locating an app's folder challenging due to the randomness of the directory names.

As applications in iOS must be sandboxed, each app will have also a folder inside **$HOME/Library/Containers** with app's **CFBundleIdentifier** as the folder name.

However, both folders (data & container folders) have the file **.com.apple.mobile\_container\_manager.metadata.plist** that links both files in the key MCMetadataIdentifier).

To facilitate the discovery of a user-installed app's installation directory, the **objection tool** provides a useful command, env. This command reveals detailed directory information for the app in question. Below is an example of how to use this command:

OWASP.iGoat-Swift on (iPhone: 11.1.2) [usb] # env

Name Path

----------------- -------------------------------------------------------------------------------------------

BundlePath /var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app

CachesDirectory /var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693/Library/Caches

DocumentDirectory /var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693/Documents

LibraryDirectory /var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693/Library

Alternatively, the app name can be searched within the /private/var/containers using the find command:

find /private/var/containers -name "Progname\*"

Commands such as ps and lsof can also be utilized to identify the app's process and list open files, respectively, providing insights into the application's active directory paths:

ps -ef | grep -i <app-name>

lsof -p <pid> | grep -i "/containers" | head -n 1

**Bundle directory:**

* **AppName.app**
  + This is the Application Bundle as seen before in the IPA, it contains essential application data, static content as well as the application's compiled binary.
  + This directory is visible to users, but **users can't write to it**.
  + Content in this directory is **not backed up**.
  + The contents of this folder are used to **validate the code signature**.

**Data directory:**

* **Documents/**
  + Contains all the user-generated data. The application end user initiates the creation of this data.
  + Visible to users and **users can write to it**.
  + Content in this directory is **backed up**.
  + The app can disable paths by setting NSURLIsExcludedFromBackupKey.
* **Library/**
  + Contains all **files that aren't user-specific**, such as **caches**, **preferences**, **cookies**, and property list (plist) configuration files.
  + iOS apps usually use the Application Support and Caches subdirectories, but the app can create custom subdirectories.
* **Library/Caches/**
  + Contains **semi-persistent cached files.**
  + Invisible to users and **users can't write to it**.
  + Content in this directory is **not backed up**.
  + The OS may delete this directory's files automatically when the app is not running and storage space is running low.
* **Library/Application Support/**
  + Contains **persistent** **files** necessary for running the app.
  + **Invisible** **to** **users** and users can't write to it.
  + Content in this directory is **backed** **up**.
  + The app can disable paths by setting NSURLIsExcludedFromBackupKey.
* **Library/Preferences/**
  + Used for storing properties that can **persist even after an application is restarted**.
  + Information is saved, unencrypted, inside the application sandbox in a plist file called [BUNDLE\_ID].plist.
  + All the key/value pairs stored using NSUserDefaults can be found in this file.
* **tmp/**
  + Use this directory to write **temporary files** that do not need to persist between app launches.
  + Contains non-persistent cached files.
  + **Invisible** to users.
  + Content in this directory is not backed up.
  + The OS may delete this directory's files automatically when the app is not running and storage space is running low.

Let's take a closer look at iGoat-Swift's Application Bundle (.app) directory inside the Bundle directory (/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app):

OWASP.iGoat-Swift on (iPhone: 11.1.2) [usb] # ls

NSFileType Perms NSFileProtection ... Name

------------ ------- ------------------ ... --------------------------------------

Regular 420 None ... rutger.html

Regular 420 None ... mansi.html

Regular 420 None ... splash.html

Regular 420 None ... about.html

Regular 420 None ... LICENSE.txt

Regular 420 None ... Sentinel.txt

Regular 420 None ... README.txt

### Binary Reversing

Inside the <application-name>.app folder you will find a binary file called <application-name>. This is the file that will be **executed**. You can perform a basic inspection of the binary with the tool **otool**:

otool -Vh DVIA-v2 #Check some compilation attributes

magic cputype cpusubtype caps filetype ncmds sizeofcmds flags

MH\_MAGIC\_64 ARM64 ALL 0x00 EXECUTE 65 7112 NOUNDEFS DYLDLINK TWOLEVEL WEAK\_DEFINES BINDS\_TO\_WEAK PIE

otool -L DVIA-v2 #Get third party libraries

DVIA-v2:

/usr/lib/libc++.1.dylib (compatibility version 1.0.0, current version 400.9.1)

/usr/lib/libsqlite3.dylib (compatibility version 9.0.0, current version 274.6.0)

/usr/lib/libz.1.dylib (compatibility version 1.0.0, current version 1.2.11)

@rpath/Bolts.framework/Bolts (compatibility version 1.0.0, current version 1.0.0)

[...]

**Check if the app is encrypted**

See if there is any output for:

otool -l <app-binary> | grep -A 4 LC\_ENCRYPTION\_INFO

**Disassembling the binary**

Disassemble the text section:

otool -tV DVIA-v2

DVIA-v2:

(\_\_TEXT,\_\_text) section

+[DDLog initialize]:

0000000100004ab8 sub sp, sp, #0x60

0000000100004abc stp x29, x30, [sp, #0x50] ; Latency: 6

0000000100004ac0 add x29, sp, #0x50

0000000100004ac4 sub x8, x29, #0x10

0000000100004ac8 mov x9, #0x0

0000000100004acc adrp x10, 1098 ; 0x10044e000

0000000100004ad0 add x10, x10, #0x268

To print the **Objective-C segment** of the sample application one can use:

otool -oV DVIA-v2

DVIA-v2:

Contents of (\_\_DATA,\_\_objc\_classlist) section

00000001003dd5b8 0x1004423d0 \_OBJC\_CLASS\_$\_DDLog

isa 0x1004423a8 \_OBJC\_METACLASS\_$\_DDLog

superclass 0x0 \_OBJC\_CLASS\_$\_NSObject

cache 0x0 \_\_objc\_empty\_cache

vtable 0x0

data 0x1003de748

flags 0x80

instanceStart 8

In order to obtain a more compact Objective-C code you can use [**class-dump**](http://stevenygard.com/projects/class-dump/):

class-dump some-app

//

// Generated by class-dump 3.5 (64 bit).

//

// class-dump is right (C) 1997-1998, 2000-2001, 2004-2013 by Steve Nygard.

//

#pragma mark Named Structures

struct CGPoint {

double \_field1;

double \_field2;

};

struct CGRect {

struct CGPoint \_field1;

struct CGSize \_field2;

};

struct CGSize {

double \_field1;

double \_field2;

};

## Data Storage

To learn about how iOS stores data in the device read this page:

[PAGEiOS Basics](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/ios-basics)

The following places to store information should be checked **right after installing the application**, **after checking all the functionalities** of the application and even after **login out from one user and login into a different one**. The goal is to find **unprotected sensitive information** of the application (passwords, tokens), of the current user and of previously logged users.

### Plist

**plist** files are structured XML files that **contains key-value pairs**. It's a way to store persistent data, so sometimes you may find **sensitive information in these files**. It's recommended to check these files after installing the app and after using intensively it to see if new data is written.

The most common way to persist data in plist files is through the usage of **NSUserDefaults**. This plist file is saved inside the app sandbox in **Library/Preferences/<appBundleID>.plist**

The [NSUserDefaults](https://developer.apple.com/documentation/foundation/nsuserdefaults) class provides a programmatic interface for interacting with the default system. The default system allows an application to customize its behaviour according to **user preferences**. Data saved by NSUserDefaults can be viewed in the application bundle. This class stores **data** in a **plist** **file**, but it's meant to be used with small amounts of data.

This data cannot be longer accessed directly via a trusted computer, but can be accessed performing a **backup**.

You can **dump** the information saved using **NSUserDefaults** using objection's ios nsuserdefaults get

To find all the plist of used by the application you can access to /private/var/mobile/Containers/Data/Application/{APPID} and run:

find ./ -name "\*.plist"

To convert files from **XML or binary (bplist)** format to XML, various methods depending on your operating system are available:

**For macOS Users:** Utilize the plutil command. It's a built-in tool in macOS (10.2+), designed for this purpose:

$ plutil -convert xml1 Info.plist

**For Linux Users:** Install libplist-utils first, then use plistutil to convert your file:

$ apt install libplist-utils

$ plistutil -i Info.plist -o Info\_xml.plist

**Within an Objection Session:** For analyzing mobile applications, a specific command allows you to convert plist files directly:

ios plist cat /private/var/mobile/Containers/Data/Application/<Application-UUID>/Library/Preferences/com.some.package.app.plist

### Core Data

[Core Data](https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreData/nsfetchedresultscontroller.html#//apple_ref/doc/uid/TP40001075-CH8-SW1) is a framework for managing the model layer of objects in your application. [Core Data can use SQLite as its persistent store](https://cocoacasts.com/what-is-the-difference-between-core-data-and-sqlite/), but the framework itself is not a database. CoreData does not encrypt it's data by default. However, an additional encryption layer can be added to CoreData. See the [GitHub Repo](https://github.com/project-imas/encrypted-core-data) for more details.

You can find the SQLite Core Data information of an application in the path /private/var/mobile/Containers/Data/Application/{APPID}/Library/Application Support

**If you can open the SQLite and access sensitive information, then you found a miss-configuration.**

Code from iGoat

-(void)storeDetails {

AppDelegate \* appDelegate = (AppDelegate \*)(UIApplication.sharedApplication.delegate);

NSManagedObjectContext \*context =[appDelegate managedObjectContext];

User \*user = [self fetchUser];

if (user) {

return;

}

user = [NSEntityDescription insertNewObjectForEntityForName:@"User"

inManagedObjectContext:context];

user.email = CoreDataEmail;

user.password = CoreDataPassword;

NSError \*error;

if (![context save:&error]) {

NSLog(@"Error in saving data: %@", [error localizedDescription]);

}else{

NSLog(@"data stored in core data");

}

}

### YapDatabase

[YapDatabase](https://github.com/yapstudios/YapDatabase) is a key/value store built on top of SQLite. As the Yap databases are sqlite databases you can find them using the purposed commend in the previous section.

### Other SQLite Databases

It's common for applications to create their own sqlite database. They may be **storing** **sensitive** **data** on them and leaving it unencrypted. Therefore, it's always interesting to check every database inside the applications directory. Therefore go to the application directory where the data is saved (/private/var/mobile/Containers/Data/Application/{APPID})

find ./ -name "\*.sqlite" -or -name "\*.db"

### Firebase Real-Time Databases

Developers are enabled to **store and sync data** within a **NoSQL cloud-hosted database** through Firebase Real-Time Databases. Stored in JSON format, the data gets synchronized to all connected clients in real time.

You can find how to check for misconfigured Firebase databases here:

[PAGEFirebase Database](https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/buckets/firebase-database)

### Realm databases

[Realm Objective-C](https://realm.io/docs/objc/latest/) and [Realm Swift](https://realm.io/docs/swift/latest/) offer a powerful alternative for data storage, not provided by Apple. By default, they **store data unencrypted**, with encryption available through specific configuration.

The databases are located at: /private/var/mobile/Containers/Data/Application/{APPID}. To explore these files, one can utilize commands like:

iPhone:/private/var/mobile/Containers/Data/Application/A079DF84-726C-4AEA-A194-805B97B3684A/Documents root# ls

default.realm default.realm.lock default.realm.management/ default.realm.note|

$ find ./ -name "\*.realm\*"

For viewing these database files, the [**Realm Studio**](https://github.com/realm/realm-studio) tool is recommended.

To implement encryption within a Realm database, the following code snippet can be used:

// Open the encrypted Realm file where getKey() is a method to obtain a key from the Keychain or a server

let config = Realm.Configuration(encryptionKey: getKey())

do {

let realm = try Realm(configuration: config)

// Use the Realm as normal

} catch let error as NSError {

// If the encryption key is wrong, `error` will say that it's an invalid database

fatalError("Error opening realm: \(error)")

}

### Couchbase Lite Databases

[Couchbase Lite](https://github.com/couchbase/couchbase-lite-ios) is described as a **lightweight** and **embedded** database engine that follows the **document-oriented** (NoSQL) approach. Designed to be native to **iOS** and **macOS**, it offers the capability to sync data seamlessly.

To identify potential Couchbase databases on a device, the following directory should be inspected:

ls /private/var/mobile/Containers/Data/Application/{APPID}/Library/Application Support/

### Cookies

iOS store the cookies of the apps in the **Library/Cookies/cookies.binarycookies** inside each apps folder. However, developers sometimes decide to save them in the **keychain** as the mentioned **cookie file can be accessed in backups**.

To inspect the cookies file you can use [**this python script**](https://github.com/mdegrazia/Safari-Binary-Cookie-Parser) or use objection's **ios cookies get.** **You can also use objection to** convert these files to a JSON format and inspect the data.

...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios cookies get --json

[

{

"domain": "highaltitudehacks.com",

"expiresDate": "2051-09-15 07:46:43 +0000",

"isHTTPOnly": "false",

"isSecure": "false",

"name": "username",

"path": "/",

"value": "admin123",

"version": "0"

}

]

### Cache

By default NSURLSession stores data, such as **HTTP requests and responses in the Cache.db** database. This database can contain **sensitive data**, if tokens, usernames or any other sensitive information has been cached. To find the cached information open the data directory of the app (/var/mobile/Containers/Data/Application/<UUID>) and go to /Library/Caches/<Bundle Identifier>. The **WebKit cache is also being stored in the Cache.db** file. **Objection** can open and interact with the database with the command sqlite connect Cache.db, as it is a n**ormal SQLite database**.

It is **recommended to disable Caching this data**, as it may contain sensitive information in the request or response. The following list below shows different ways of achieving this:

1. It is recommended to remove Cached responses after logout. This can be done with the provided method by Apple called [removeAllCachedResponses](https://developer.apple.com/documentation/foundation/urlcache/1417802-removeallcachedresponses) You can call this method as follows:

URLCache.shared.removeAllCachedResponses()

This method will remove all cached requests and responses from Cache.db file.

1. If you don't need to use the advantage of cookies it would be recommended to just use the [.ephemeral](https://developer.apple.com/documentation/foundation/urlsessionconfiguration/1410529-ephemeral) configuration property of URLSession, which will disable saving cookies and Caches.

[Apple documentation](https://developer.apple.com/documentation/foundation/urlsessionconfiguration/1410529-ephemeral):

An ephemeral session configuration object is similar to a default session configuration (see default), except that the corresponding session object doesn’t store caches, credential stores, or any session-related data to disk. Instead, session-related data is stored in RAM. The only time an ephemeral session writes data to disk is when you tell it to write the contents of a URL to a file.

1. Cache can be also disabled by setting the Cache Policy to [.notAllowed](https://developer.apple.com/documentation/foundation/urlcache/storagepolicy/notallowed). It will disable storing Cache in any fashion, either in memory or on disk.

### Snapshots

Whenever you press the home button, iOS **takes a snapshot of the current screen** to be able to do the transition to the application on a much smoother way. However, if **sensitive** **data** is present in the current screen, it will be **saved** in the **image** (which **persists** **across** **reboots**). These are the snapshots that you can also access double tapping the home screen to switch between apps.

Unless the iPhone is jailbroken, the **attacker** needs to have **access** to the **device** **unblocked** to see these screenshots. By default the last snapshot is stored in the application's sandbox in Library/Caches/Snapshots/ or Library/SplashBoard/Snapshots folder (the trusted computers can' t access the filesystem from iOX 7.0).

Once way to prevent this bad behaviour is to put a blank screen or remove the sensitive data before taking the snapshot using the ApplicationDidEnterBackground() function.

The following is a sample remediation method that will set a default screenshot.

Swift:

private var backgroundImage: UIImageView?

func applicationDidEnterBackground(\_ application: UIApplication) {

let myBanner = UIImageView(image: #imageLiteral(resourceName: "overlayImage"))

myBanner.frame = UIScreen.main.bounds

backgroundImage = myBanner

window?.addSubview(myBanner)

}

func applicationWillEnterForeground(\_ application: UIApplication) {

backgroundImage?.removeFromSuperview()

}

Objective-C:

@property (UIImageView \*)backgroundImage;

- (void)applicationDidEnterBackground:(UIApplication \*)application {

UIImageView \*myBanner = [[UIImageView alloc] initWithImage:@"overlayImage.png"];

self.backgroundImage = myBanner;

self.backgroundImage.bounds = UIScreen.mainScreen.bounds;

[self.window addSubview:myBanner];

}

- (void)applicationWillEnterForeground:(UIApplication \*)application {

[self.backgroundImage removeFromSuperview];

}

This sets the background image to overlayImage.png whenever the application is backgrounded. It prevents sensitive data leaks because overlayImage.png will always override the current view.

### Keychain

For accessing and managing the iOS keychain, tools like [**Keychain-Dumper**](https://github.com/ptoomey3/Keychain-Dumper) are available, suitable for jailbroken devices. Additionally, [**Objection**](https://github.com/sensepost/objection) provides the command ios keychain dump for similar purposes.

#### **Storing Credentials**

The **NSURLCredential** class is ideal for saving sensitive information directly in the keychain, bypassing the need for NSUserDefaults or other wrappers. To store credentials after login, the following Swift code is used:

NSURLCredential \*credential;

credential = [NSURLCredential credentialWithUser:username password:password persistence:NSURLCredentialPersistencePermanent];

[[NSURLCredentialStorage sharedCredentialStorage] setCredential:credential forProtectionSpace:self.loginProtectionSpace];

To extract these stored credentials, Objection's command ios nsurlcredentialstorage dump is utilized.

## **Custom Keyboards and Keyboard Cache**

With iOS 8.0 onwards, users can install custom keyboard extensions, which are manageable under **Settings > General > Keyboard > Keyboards**. While these keyboards offer extended functionality, they pose a risk of keystroke logging and transmitting data to external servers, though users are notified about keyboards requiring network access. Apps can, and should, restrict the use of custom keyboards for sensitive information entry.

**Security Recommendations:**

* It's advised to disable third-party keyboards for enhanced security.
* Be aware of the autocorrect and auto-suggestions features of the default iOS keyboard, which could store sensitive information in cache files located in Library/Keyboard/{locale}-dynamic-text.dat or /private/var/mobile/Library/Keyboard/dynamic-text.dat. These cache files should be regularly checked for sensitive data. Resetting the keyboard dictionary via **Settings > General > Reset > Reset Keyboard Dictionary** is recommended for clearing cached data.
* Intercepting network traffic can reveal whether a custom keyboard is transmitting keystrokes remotely.

### **Preventing Text Field Caching**

The [UITextInputTraits protocol](https://developer.apple.com/reference/uikit/uitextinputtraits) offers properties to manage autocorrection and secure text entry, essential for preventing sensitive information caching. For example, disabling autocorrection and enabling secure text entry can be achieved with:

textObject.autocorrectionType = UITextAutocorrectionTypeNo;

textObject.secureTextEntry = YES;

Additionally, developers should ensure that text fields, especially those for entering sensitive information like passwords and PINs, disable caching by setting autocorrectionType to UITextAutocorrectionTypeNo and secureTextEntry to YES.

UITextField \*textField = [[UITextField alloc] initWithFrame:frame];

textField.autocorrectionType = UITextAutocorrectionTypeNo;

## **Logs**

Debugging code often involves the use of **logging**. There's a risk involved as **logs may contain sensitive information**. Previously, in iOS 6 and earlier versions, logs were accessible to all apps, posing a risk of sensitive data leakage. **Now, applications are restricted to accessing only their logs**.

Despite these restrictions, an **attacker with physical access** to an unlocked device can still exploit this by connecting the device to a computer and **reading the logs**. It is important to note that logs remain on the disk even after the app's uninstallation.

To mitigate risks, it is advised to **thoroughly interact with the app**, exploring all its functionalities and inputs to ensure no sensitive information is being logged inadvertently.

When reviewing the app's source code for potential leaks, look for both **predefined** and **custom logging statements** using keywords such as NSLog, NSAssert, NSCAssert, fprintf for built-in functions, and any mentions of Logging or Logfile for custom implementations.

### **Monitoring System Logs**

Apps log various pieces of information which can be sensitive. To monitor these logs, tools and commands like:

idevice\_id --list # To find the device ID

idevicesyslog -u <id> (| grep <app>) # To capture the device logs

are useful. Additionally, **Xcode** provides a way to collect console logs:

1. Open Xcode.
2. Connect the iOS device.
3. Navigate to **Window** -> **Devices and Simulators**.
4. Select your device.
5. Trigger the issue you're investigating.
6. Use the **Open Console** button to view logs in a new window.

For more advanced logging, connecting to the device shell and using **socat** can provide real-time log monitoring:

iPhone:~ root# socat - UNIX-CONNECT:/var/run/lockdown/syslog.sock

Followed by commands to observe log activities, which can be invaluable for diagnosing issues or identifying potential data leakage in logs.

## Backups

**Auto-backup features** are integrated into iOS, facilitating the creation of device data copies through iTunes (up to macOS Catalina), Finder (from macOS Catalina onward), or iCloud. These backups encompass almost all device data, excluding highly sensitive elements like Apple Pay details and Touch ID configurations.

### Security Risks

The inclusion of **installed apps and their data** in backups raises the issue of potential **data leakage** and the risk that **backup modifications could alter app functionality**. It's advised to **not store sensitive information in plaintext** within any app's directory or its subdirectories to mitigate these risks.

### Excluding Files from Backups

Files in Documents/ and Library/Application Support/ are backed up by default. Developers can exclude specific files or directories from backups using NSURL setResourceValue:forKey:error: with the NSURLIsExcludedFromBackupKey. This practice is crucial for protecting sensitive data from being included in backups.

### Testing for Vulnerabilities

To assess an app's backup security, start by **creating a backup** using Finder, then locate it using guidance from [Apple's official documentation](https://support.apple.com/en-us/HT204215). Analyze the backup for sensitive data or configurations that could be altered to affect app behavior.

Sensitive information can be sought out using command-line tools or applications like [iMazing](https://imazing.com/). For encrypted backups, the presence of encryption can be confirmed by checking the "IsEncrypted" key in the "Manifest.plist" file at the backup's root.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

...

<key>Date</key>

<date>2021-03-12T17:43:33Z</date>

<key>IsEncrypted</key>

<true/>

...

</plist>

For dealing with encrypted backups, Python scripts available in [DinoSec's GitHub repo](https://github.com/dinosec/iphone-dataprotection/tree/master/python_scripts), like **backup\_tool.py** and **backup\_passwd.py**, may be useful, albeit potentially requiring adjustments for compatibility with the latest iTunes/Finder versions. The [**iOSbackup** tool](https://pypi.org/project/iOSbackup/) is another option for accessing files within password-protected backups.

### Modifying App Behavior

An example of altering app behavior through backup modifications is demonstrated in the [Bither bitcoin wallet app](https://github.com/bither/bither-ios), where the UI lock PIN is stored within net.bither.plist under the **pin\_code** key. Removing this key from the plist and restoring the backup removes the PIN requirement, providing unrestricted access.

## Summary on Memory Testing for Sensitive Data

When dealing with sensitive information stored in an application's memory, it is crucial to limit the exposure time of this data. There are two primary approaches to investigate memory content: **creating a memory dump** and **analyzing the memory in real time**. Both methods have their challenges, including the potential to miss critical data during the dump process or analysis.

## **Retrieving and Analyzing a Memory Dump**

For both jailbroken and non-jailbroken devices, tools like [objection](https://github.com/sensepost/objection) and [Fridump](https://github.com/Nightbringer21/fridump) allow for the dumping of an app's process memory. Once dumped, analyzing this data requires various tools, depending on the nature of the information you're searching for.

To extract strings from a memory dump, commands such as strings or rabin2 -zz can be used:

# Extracting strings using strings command

$ strings memory > strings.txt

# Extracting strings using rabin2

$ rabin2 -ZZ memory > strings.txt

For more detailed analysis, including searching for specific data types or patterns, **radare2** offers extensive search capabilities:

$ r2 <name\_of\_your\_dump\_file>

[0x00000000]> /?

...

## **Runtime Memory Analysis**

**r2frida** provides a powerful alternative for inspecting an app's memory in real time, without needing a memory dump. This tool enables the execution of search commands directly on the running application's memory:

$ r2 frida://usb//<name\_of\_your\_app>

[0x00000000]> /\ <search\_command>

## Broken Cryptography

### Poor Key Management Processes

Some developers save sensitive data in the local storage and encrypt it with a key hardcoded/predictable in the code. This shouldn't be done as some reversing could allow attackers to extract the confidential information.

### Use of Insecure and/or Deprecated Algorithms

Developers shouldn't use **deprecated algorithms** to perform authorisation **checks**, **store** or **send** data. Some of these algorithms are: RC4, MD4, MD5, SHA1... If **hashes** are used to store passwords for example, hashes brute-force **resistant** should be used with salt.

### Check

The main checks to perform if to find if you can find **hardcoded** passwords/secrets in the code, or if those are **predictable**, and if the code is using some king of **weak** **cryptography** algorithms.

It's interesting to know that you can **monitor** some **crypto** **libraries** automatically using **objection** with:

ios monitor crypt

For **more information** about iOS cryptographic APIs and libraries access <https://mobile-security.gitbook.io/mobile-security-testing-guide/ios-testing-guide/0x06e-testing-cryptography>

## Local Authentication

**Local authentication** plays a crucial role, especially when it concerns safeguarding access at a remote endpoint through cryptographic methods. The essence here is that without proper implementation, local authentication mechanisms can be circumvented.

Apple's [**Local Authentication framework**](https://developer.apple.com/documentation/localauthentication) and the [**keychain**](https://developer.apple.com/library/content/documentation/Security/Conceptual/keychainServConcepts/01introduction/introduction.html) provide robust APIs for developers to facilitate user authentication dialogs and securely handle secret data, respectively. The Secure Enclave secures fingerprint ID for Touch ID, whereas Face ID relies on facial recognition without compromising biometric data.

To integrate Touch ID/Face ID, developers have two API choices:

* **LocalAuthentication.framework** for high-level user authentication without access to biometric data.
* **Security.framework** for lower-level keychain services access, securing secret data with biometric authentication. Various [open-source wrappers](https://www.raywenderlich.com/147308/secure-ios-user-data-keychain-touch-id) make keychain access simpler.

However, both LocalAuthentication.framework and Security.framework present vulnerabilities, as they primarily return boolean values without transmitting data for authentication processes, making them susceptible to bypassing (refer to [Don't touch me that way, by David Lindner et al](https://www.youtube.com/watch?v=XhXIHVGCFFM)).

### Implementing Local Authentication

To prompt users for authentication, developers should utilize the **evaluatePolicy** method within the **LAContext** class, choosing between:

* **deviceOwnerAuthentication**: Prompts for Touch ID or device passcode, failing if neither is enabled.
* **deviceOwnerAuthenticationWithBiometrics**: Exclusively prompts for Touch ID.

A successful authentication is indicated by a boolean return value from **evaluatePolicy**, highlighting a potential security flaw.

### Local Authentication using Keychain

Implementing **local authentication** in iOS apps involves the use of **keychain APIs** to securely store secret data such as authentication tokens. This process ensures that the data can only be accessed by the user, using their device passcode or biometric authentication like Touch ID.

The keychain offers the capability to set items with the SecAccessControl attribute, which restricts access to the item until the user successfully authenticates via Touch ID or device passcode. This feature is crucial for enhancing security.

Below are code examples in Swift and Objective-C demonstrating how to save and retrieve a string to/from the keychain, leveraging these security features. The examples specifically show how to set up access control to require Touch ID authentication and ensure the data is accessible only on the device it was set up on, under the condition that a device passcode is configured.

Swift

// From https://github.com/mufambisi/owasp-mstg/blob/master/Document/0x06f-Testing-Local-Authentication.md

// 1. create AccessControl object that will represent authentication settings

var error: Unmanaged<CFError>?

guard let accessControl = SecAccessControlCreateWithFlags(kCFAllocatorDefault,

kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,

SecAccessControlCreateFlags.biometryCurrentSet,

&error) else {

// failed to create AccessControl object

return

}

// 2. define keychain services query. Pay attention that kSecAttrAccessControl is mutually exclusive with kSecAttrAccessible attribute

var query: [String: Any] = [:]

query[kSecClass as String] = kSecClassGenericPassword

query[kSecAttrLabel as String] = "com.me.myapp.password" as CFString

query[kSecAttrAccount as String] = "OWASP Account" as CFString

query[kSecValueData as String] = "test\_strong\_password".data(using: .utf8)! as CFData

query[kSecAttrAccessControl as String] = accessControl

// 3. save item

let status = SecItemAdd(query as CFDictionary, nil)

if status == noErr {

// successfully saved

} else {

// error while saving

}

Objective-C

// 1. create AccessControl object that will represent authentication settings

CFErrorRef \*err = nil;

SecAccessControlRef sacRef = SecAccessControlCreateWithFlags(kCFAllocatorDefault,

kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,

kSecAccessControlUserPresence,

err);

// 2. define keychain services query. Pay attention that kSecAttrAccessControl is mutually exclusive with kSecAttrAccessible attribute

NSDictionary\* query = @{

(\_ \_bridge id)kSecClass: (\_\_bridge id)kSecClassGenericPassword,

(\_\_bridge id)kSecAttrLabel: @"com.me.myapp.password",

(\_\_bridge id)kSecAttrAccount: @"OWASP Account",

(\_\_bridge id)kSecValueData: [@"test\_strong\_password" dataUsingEncoding:NSUTF8StringEncoding],

(\_\_bridge id)kSecAttrAccessControl: (\_\_bridge\_transfer id)sacRef

};

// 3. save item

OSStatus status = SecItemAdd((\_\_bridge CFDictionaryRef)query, nil);

if (status == noErr) {

// successfully saved

} else {

// error while saving

}

Now we can request the saved item from the keychain. Keychain services will present the authentication dialog to the user and return data or nil depending on whether a suitable fingerprint was provided or not.

Swift

// 1. define query

var query = [String: Any]()

query[kSecClass as String] = kSecClassGenericPassword

query[kSecReturnData as String] = kCFBooleanTrue

query[kSecAttrAccount as String] = "My Name" as CFString

query[kSecAttrLabel as String] = "com.me.myapp.password" as CFString

query[kSecUseOperationPrompt as String] = "Please, pass authorisation to enter this area" as CFString

// 2. get item

var queryResult: AnyObject?

let status = withUnsafeMutablePointer(to: &queryResult) {

SecItem Matching(query as CFDictionary, UnsafeMutablePointer($0))

}

if status == noErr {

let password = String(data: queryResult as! Data, encoding: .utf8)!

// successfully received password

} else {

// authorization not passed

}

Objective-C

// 1. define query

NSDictionary \*query = @{(\_\_bridge id)kSecClass: (\_\_bridge id)kSecClassGenericPassword,

(\_\_bridge id)kSecReturnData: @YES,

(\_\_bridge id)kSecAttrAccount: @"My Name1",

(\_\_bridge id)kSecAttrLabel: @"com.me.myapp.password",

(\_\_bridge id)kSecUseOperationPrompt: @"Please, pass authorisation to enter this area" };

// 2. get item

CFTypeRef queryResult = NULL;

OSStatus status = SecItem Matching((\_\_bridge CFDictionaryRef)query, &queryResult);

if (status == noErr){

NSData\* resultData = ( \_\_bridge\_transfer NSData\* )queryResult;

NSString\* password = [[NSString alloc] initWithData:resultData encoding:NSUTF8StringEncoding];

NSLog(@"%@", password);

} else {

NSLog(@"Something went wrong");

}

### Detection

Usage of frameworks in an app can also be detected by analyzing the app binary's list of shared dynamic libraries. This can be done by using otool:

$ otool -L <AppName>.app/<AppName>

If LocalAuthentication.framework is used in an app, the output will contain both of the following lines (remember that LocalAuthentication.framework uses Security.framework under the hood):

/System/Library/Frameworks/LocalAuthentication.framework/LocalAuthentication

/System/Library/Frameworks/Security.framework/Security

If Security.framework is used, only the second one will be shown.

### Local Authentication Framework Bypass

#### **Objection**

Through the **Objection Biometrics Bypass**, located at [this GitHub page](https://github.com/sensepost/objection/wiki/Understanding-the-iOS-Biometrics-Bypass), a technique is available for overcoming the **LocalAuthentication** mechanism. The core of this approach involves leveraging **Frida** to manipulate the evaluatePolicy function, ensuring it consistently yields a True outcome, irrespective of the actual authentication success. This is particularly useful for circumventing flawed biometric authentication processes.

To activate this bypass, the following command is employed:

...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios ui biometrics\_bypass

(agent) Registering job 3mhtws9x47q. Type: ios-biometrics-disable

...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # (agent) [3mhtws9x47q] Localized Reason for auth requirement: Please authenticate yourself

(agent) [3mhtws9x47q] OS authentication response: false

(agent) [3mhtws9x47q] Marking OS response as True instead

(agent) [3mhtws9x47q] Biometrics bypass hook complete

This command sets off a sequence where Objection registers a task that effectively alters the outcome of the evaluatePolicy check to True.

#### Frida

An example of a use of **evaluatePolicy** from [DVIA-v2 application](https://github.com/prateek147/DVIA-v2):

+(void)authenticateWithTouchID {

LAContext \*myContext = [[LAContext alloc] init];

NSError \*authError = nil;

NSString \*myLocalizedReasonString = @"Please authenticate yourself";

if ([myContext canEvaluatePolicy:LAPolicyDeviceOwnerAuthenticationWithBiometrics error:&authError]) {

[myContext evaluatePolicy:LAPolicyDeviceOwnerAuthenticationWithBiometrics

localizedReason:myLocalizedReasonString

reply:^(BOOL success, NSError \*error) {

if (success) {

dispatch\_async(dispatch\_get\_main\_queue(), ^{

[TouchIDAuthentication showAlert:@"Authentication Successful" withTitle:@"Success"];

});

} else {

dispatch\_async(dispatch\_get\_main\_queue(), ^{

[TouchIDAuthentication showAlert:@"Authentication Failed !" withTitle:@"Error"];

});

}

}];

} else {

dispatch\_async(dispatch\_get\_main\_queue(), ^{

[TouchIDAuthentication showAlert:@"Your device doesn't support Touch ID or you haven't configured Touch ID authentication on your device" withTitle:@"Error"];

});

}

}

To achieve the **bypass** of Local Authentication, a Frida script is written. This script targets the **evaluatePolicy** check, intercepting its callback to ensure it returns **success=1**. By altering the callback's behavior, the authentication check is effectively bypassed.

The script below is injected to modify the result of the **evaluatePolicy** method. It changes the callback's result to always indicate success.

// from https://securitycafe.ro/2022/09/05/mobile-pentesting-101-bypassing-biometric-authentication/

if(ObjC.available) {

console.log("Injecting...");

var hook = ObjC.classes.LAContext["- evaluatePolicy:localizedReason:reply:"];

Interceptor.attach(hook.implementation, {

onEnter: function(args) {

var block = new ObjC.Block(args[4]);

const callback = block.implementation;

block.implementation = function (error, value) {

console.log("Changing the result value to true")

const result = callback(1, null);

return result;

};

},

});

} else {

console.log("Objective-C Runtime is not available!");

}

To inject the Frida script and bypass the biometric authentication, the following command is used:

frida -U -f com.highaltitudehacks.DVIAswiftv2 --no-pause -l fingerprint-bypass-ios.js

## Sensitive Functionality Exposure Through IPC

### Custom URI Handlers / Deeplinks / Custom Schemes

[PAGEiOS Custom URI Handlers / Deeplinks / Custom Schemes](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/ios-custom-uri-handlers-deeplinks-custom-schemes)

### Universal Links

[PAGEiOS Universal Links](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/ios-universal-links)

### UIActivity Sharing

[PAGEiOS UIActivity Sharing](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/ios-uiactivity-sharing)

### UIPasteboard

[PAGEiOS UIPasteboard](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/ios-uipasteboard)

### App Extensions

[PAGEiOS App Extensions](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/ios-app-extensions)

### WebViews

[PAGEiOS WebViews](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/ios-webviews)

### Serialisation and Encoding

[PAGEiOS Serialisation and Encoding](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/ios-serialisation-and-encoding)

## Network Communication

It's important to check that no communication is occurring **without encryption** and also that the application is correctly **validating the TLS certificate** of the server. To check these kind of issues you can use a proxy like **Burp**:

[PAGEiOS Burp Suite Configuration](https://book.hacktricks.xyz/mobile-pentesting/ios-pentesting/burp-configuration-for-ios)

### Hostname check

One common issue validating the TLS certificate is to check that the certificate was signed by a **trusted** **CA**, but **not check** if **the hostname** of the certificate is the hostname being accessed. In order to check this issue using Burp, after trusting Burp CA in the iPhone, you can **create a new certificate with Burp for a different hostname** and use it. If the application still works, then, something it's vulnerable.

### Certificate Pinning

If an application is correctly using SSL Pinning, then the application will only works if the certificate is the once expected to be. When testing an application **this might be a problem as Burp will serve it's own certificate.** In order to bypass this protection inside a jailbroken device, you can install the application [**SSL Kill Switch**](https://github.com/nabla-c0d3/ssl-kill-switch2) or install [**Burp Mobile Assistant**](https://portswigger.net/burp/documentation/desktop/mobile/config-ios-device)

You can also use **objection's** ios sslpinning disable

## Misc

* In **/System/Library** you can find the frameworks installed in the phone used by system applications
* The applications installed by the user from the App Store are located inside **/User/Applications**
* And the **/User/Library** contains data saved by the user level applications
* You can access **/User/Library/Notes/notes.sqlite** to read the notes saved inside the application.
* Inside the folder of an installed application (**/User/Applications/<APP ID>/**) you can find some interesting files:
  + **iTunesArtwork**: The icon used by the app
  + **iTunesMetadata.plist**: Info of the app used in the App Store
  + **/Library/\***: Contains the preferences and cache. In **/Library/Cache/Snapshots/\*** you can find the snapshot performed to the application before sending it to the background.

### Hot Patching/Enforced Updateing

The developers can remotely **patch all installations of their app instantly** without having to resubmit the application to the App store and wait until it's approved. For this purpose it's usually use [**JSPatch**](https://github.com/bang590/JSPatch)**.** But there are other options also such as [Siren](https://github.com/ArtSabintsev/Siren) and [react-native-appstore-version-checker](https://www.npmjs.com/package/react-native-appstore-version-checker). **This is a dangerous mechanism that could be abused by malicious third party SDKs therefore it's recommended to check which method is used to automatic updating (if any) and test it.** You could try to download a previous version of the app for this purpose.

### Third Parties

A significant challenge with **3rd party SDKs** is the **lack of granular control** over their functionalities. Developers are faced with a choice: either integrate the SDK and accept all its features, including potential security vulnerabilities and privacy concerns, or forego its benefits entirely. Often, developers are unable to patch vulnerabilities within these SDKs themselves. Furthermore, as SDKs gain trust within the community, some may start to contain malware.

The services provided by third-party SDKs may include user behavior tracking, advertisement displays, or user experience enhancements. However, this introduces a risk as developers may not be fully aware of the code executed by these libraries, leading to potential privacy and security risks. It's crucial to limit the information shared with third-party services to what is necessary and ensure that no sensitive data is exposed.

Implementation of third-party services usually comes in two forms: a standalone library or a full SDK. To protect user privacy, any data shared with these services should be **anonymized** to prevent the disclosure of Personal Identifiable Information (PII).

To identify the libraries an application uses, the **otool** command can be employed. This tool should be run against the application and each shared library it uses to discover additional libraries.

otool -L <application\_path>

## **References & More Resources**

* <https://mobile-security.gitbook.io/mobile-security-testing-guide/ios-testing-guide/0x06b-basic-security-testing#information-gathering>
* [iOS & Mobile App Pentesting - INE](https://my.ine.com/CyberSecurity/courses/089d060b/ios-mobile-app-pentesting)
* <https://mas.owasp.org/MASTG/techniques/ios/MASTG-TECH-0057/>
* <https://mas.owasp.org/MASTG/techniques/ios/MASTG-TECH-0058/>
* <https://mas.owasp.org/MASTG/techniques/ios/MASTG-TECH-0059/>
* <https://mas.owasp.org/MASTG/iOS/0x06d-Testing-Data-Storage>
* <https://coderwall.com/p/kjb3lw/storing-password-in-keychain-the-smart-way>
* <https://mas.owasp.org/MASTG/tests/ios/MASVS-STORAGE/MASTG-TEST-0055/>
* <https://mas.owasp.org/MASTG/tests/ios/MASVS-STORAGE/MASTG-TEST-0053>
* <https://mas.owasp.org/MASTG/techniques/ios/MASTG-TECH-0060/>
* <https://mas.owasp.org/MASTG/tests/ios/MASVS-STORAGE/MASTG-TEST-0058>
* <https://mas.owasp.org/MASTG/tests/ios/MASVS-STORAGE/MASTG-TEST-0060>
* <https://mas.owasp.org/MASTG/Android/0x05f-Testing-Local-Authentication/>
* <https://mas.owasp.org/MASTG/tests/ios/MASVS-AUTH/MASTG-TEST-0064>
* <https://medium.com/securing/bypassing-your-apps-biometric-checks-on-ios-c2555c81a2dc>
* <https://mas.owasp.org/MASTG/tests/ios/MASVS-STORAGE/MASTG-TEST-0054>
* <https://github.com/ivRodriguezCA/RE-iOS-Apps/> IOS free course(<https://syrion.me/blog/ios-swift-antijailbreak-bypass-frida/>)
* <https://www.sans.org/reading-room/whitepapers/testing/ipwn-apps-pentesting-ios-applications-34577>
* <https://www.slideshare.net/RyanISI/ios-appsecurityminicourse>
* <https://github.com/prateek147/DVIA>
* <https://github.com/prateek147/DVIA-v2>
* [https://github.com/OWASP/MSTG-Hacking-Playground%20](https://github.com/OWASP/MSTG-Hacking-Playground)
* OWASP iGoat [*https://github.com/OWASP/igoat*](https://github.com/OWASP/igoat) <<< Objective-C version [*https://github.com/OWASP/iGoat-Swift*](https://github.com/OWASP/iGoat-Swift) <<< Swift version
* <https://github.com/authenticationfailure/WheresMyBrowser.iOS>
* <https://github.com/nabla-c0d3/ssl-kill-switch2>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

iOS Matrix

Below are the tactics and techniques representing the two MITRE ATT&CK® Matrices for Mobile. The Matrices cover techniques involving device access and network-based effects that can be used by adversaries without device access. The Matrix contains information for the iOS platform.

| [**Initial Access**](https://attack.mitre.org/tactics/TA0027) | [**Execution**](https://attack.mitre.org/tactics/TA0041) | [**Persistence**](https://attack.mitre.org/tactics/TA0028) | [**Privilege Escalation**](https://attack.mitre.org/tactics/TA0029) | [**Defense Evasion**](https://attack.mitre.org/tactics/TA0030) | [**Credential Access**](https://attack.mitre.org/tactics/TA0031) | [**Discovery**](https://attack.mitre.org/tactics/TA0032) | [**Lateral Movement**](https://attack.mitre.org/tactics/TA0033) | [**Collection**](https://attack.mitre.org/tactics/TA0035) | [**Command and Control**](https://attack.mitre.org/tactics/TA0037) | [**Exfiltration**](https://attack.mitre.org/tactics/TA0036) | [**Impact**](https://attack.mitre.org/tactics/TA0034) |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 techniques | 3 techniques | 3 techniques | 2 techniques | 9 techniques | 4 techniques | 7 techniques | 2 techniques | 10 techniques | 8 techniques | 2 techniques | 3 techniques |
| |  |  | | --- | --- | | [Application Versioning](https://attack.mitre.org/techniques/T1661) |  | | [Drive-By Compromise](https://attack.mitre.org/techniques/T1456) |  | | [Exploitation for Initial Access](https://attack.mitre.org/techniques/T1664) |  | | [Lockscreen Bypass](https://attack.mitre.org/techniques/T1461) |  | | [Phishing](https://attack.mitre.org/techniques/T1660) |  | | [Replication Through Removable Media](https://attack.mitre.org/techniques/T1458) |  | | |  | | --- | | [Supply Chain Compromise (3)](https://attack.mitre.org/techniques/T1474) | | = |  | | |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | [Command and Scripting Interpreter (1)](https://attack.mitre.org/techniques/T1623) | | = |  | | [Exploitation for Client Execution](https://attack.mitre.org/techniques/T1658) |  |  | | [Scheduled Task/Job](https://attack.mitre.org/techniques/T1603) |  |  | | |  |  | | --- | --- | | [Boot or Logon Initialization Scripts](https://attack.mitre.org/techniques/T1398) |  | | [Compromise Client Software Binary](https://attack.mitre.org/techniques/T1645) |  | | [Scheduled Task/Job](https://attack.mitre.org/techniques/T1603) |  | | |  |  | | --- | --- | | [Exploitation for Privilege Escalation](https://attack.mitre.org/techniques/T1404) |  | | |  | | --- | | [Process Injection (1)](https://attack.mitre.org/techniques/T1631) | | = |  | | |  |  | | --- | --- | | [Application Versioning](https://attack.mitre.org/techniques/T1661) |  | | [Download New Code at Runtime](https://attack.mitre.org/techniques/T1407) |  | | |  | | --- | | [Execution Guardrails (1)](https://attack.mitre.org/techniques/T1627) | | = |  | | |  | | --- | | [Indicator Removal on Host (1)](https://attack.mitre.org/techniques/T1630) | | = |  | | |  | | --- | | [Masquerading (1)](https://attack.mitre.org/techniques/T1655) | | = |  | | |  | | --- | | [Obfuscated Files or Information (1)](https://attack.mitre.org/techniques/T1406) | | = |  | | |  | | --- | | [Process Injection (1)](https://attack.mitre.org/techniques/T1631) | | = |  | | |  | | --- | | [Subvert Trust Controls (1)](https://attack.mitre.org/techniques/T1632) | | = |  | | |  | | --- | | [Virtualization/Sandbox Evasion (1)](https://attack.mitre.org/techniques/T1633) | | = |  | | |  |  | | --- | --- | | [Clipboard Data](https://attack.mitre.org/techniques/T1414) |  | | |  | | --- | | [Credentials from Password Store (1)](https://attack.mitre.org/techniques/T1634) | | = |  | | |  | | --- | | [Input Capture (2)](https://attack.mitre.org/techniques/T1417) | | = |  | | |  | | --- | | [Steal Application Access Token (1)](https://attack.mitre.org/techniques/T1635) | | = |  | | |  |  | | --- | --- | | [File and Directory Discovery](https://attack.mitre.org/techniques/T1420) |  | | |  | | --- | | [Location Tracking (2)](https://attack.mitre.org/techniques/T1430) | | = |  | | [Network Service Scanning](https://attack.mitre.org/techniques/T1423) |  |  | | [Process Discovery](https://attack.mitre.org/techniques/T1424) |  |  | | |  | | --- | | [Software Discovery (1)](https://attack.mitre.org/techniques/T1418) | | = |  | | [System Information Discovery](https://attack.mitre.org/techniques/T1426) |  |  | | |  | | --- | | [System Network Configuration Discovery (2)](https://attack.mitre.org/techniques/T1422) | | = |  | | |  |  | | --- | --- | | [Exploitation of Remote Services](https://attack.mitre.org/techniques/T1428) |  | | [Replication Through Removable Media](https://attack.mitre.org/techniques/T1458) |  | | |  |  | | --- | --- | | [Adversary-in-the-Middle](https://attack.mitre.org/techniques/T1638) |  | | [Archive Collected Data](https://attack.mitre.org/techniques/T1532) |  | | [Audio Capture](https://attack.mitre.org/techniques/T1429) |  | | [Clipboard Data](https://attack.mitre.org/techniques/T1414) |  | | [Data from Local System](https://attack.mitre.org/techniques/T1533) |  | | |  | | --- | | [Input Capture (2)](https://attack.mitre.org/techniques/T1417) | | = |  | | |  | | --- | | [Location Tracking (2)](https://attack.mitre.org/techniques/T1430) | | = |  | | |  | | --- | | [Protected User Data (4)](https://attack.mitre.org/techniques/T1636) | | = |  | | [Stored Application Data](https://attack.mitre.org/techniques/T1409) |  |  | | [Video Capture](https://attack.mitre.org/techniques/T1512) |  |  | | |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | [Application Layer Protocol (1)](https://attack.mitre.org/techniques/T1437) | | = |  | | |  | | --- | | [Dynamic Resolution (1)](https://attack.mitre.org/techniques/T1637) | | = |  | | |  | | --- | | [Encrypted Channel (3)](https://attack.mitre.org/techniques/T1521) | | = |  | | [Ingress Tool Transfer](https://attack.mitre.org/techniques/T1544) |  |  | | [Non-Standard Port](https://attack.mitre.org/techniques/T1509) |  |  | | [Out of Band Data](https://attack.mitre.org/techniques/T1644) |  |  | | [Remote Access Software](https://attack.mitre.org/techniques/T1663) |  |  | | |  | | --- | | [Web Service (3)](https://attack.mitre.org/techniques/T1481) | | = |  | | |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | [Exfiltration Over Alternative Protocol (1)](https://attack.mitre.org/techniques/T1639) | | = |  | | [Exfiltration Over C2 Channel](https://attack.mitre.org/techniques/T1646) |  |  | | |  |  | | --- | --- | | [Endpoint Denial of Service](https://attack.mitre.org/techniques/T1642) |  | | [Generate Traffic from Victim](https://attack.mitre.org/techniques/T1643) |  | | [Network Denial of Service](https://attack.mitre.org/techniques/T1464) |  | |