**Mobile Application security testing**

**Contents**

1. [OWASP Mobile Top 10-2016](#Owasp_Mobile)--------------------------------------------------1

Android application Security testing

1. [Static analysis - android](#Static_Android)--------------------------------------------------------3
2. [Dynamic analysis- android](#Dynamic_android)-----------------------------------------------------8
3. [References](#References_android)-----------------------------------------------------------------------13

IOS application Security testing

1. [Static analysis- IOS](#Static_IOS)-------------------------------------------------------------14

1. [Dynamic Analysis- IOS](#Dynamic_IOS)---------------------------------------------------------19
2. References-----------------------------------------------------------------------25

**Mobile Application Security testing**

Security has always been a major concern for businesses, and this concern is even greater when it comes to mobile applications. Today, almost every leading brand or product has a mobile app to connect with their customers more easily.

Security testing is aimed to assess the security controls of the android applications as per the Open web application Security (OWASP) Mobile security Top 10.

The [OWASP Mobile Security Project](https://owasp.org/www-project-mobile-security/) includes a list of the top ten security risks that mobile applications face today. Each of the top ten mobile security risks is ranked by its threat level and further investigated.

* [M1: Improper Platform Usage](https://owasp.org/www-project-mobile-top-10/2016-risks/m1-improper-platform-usage)
* [M2: Insecure Data Storage](https://owasp.org/www-project-mobile-top-10/2016-risks/m2-insecure-data-storage)
* [M3: Insecure Communication](https://owasp.org/www-project-mobile-top-10/2016-risks/m3-insecure-communication)
* [M4: Insecure Authentication](https://owasp.org/www-project-mobile-top-10/2016-risks/m4-insecure-authentication)
* [M5: Insufficient Cryptography](https://owasp.org/www-project-mobile-top-10/2016-risks/m5-insufficient-cryptography)
* [M6: Insecure Authorization](https://owasp.org/www-project-mobile-top-10/2016-risks/m6-insecure-authorization)
* [M7: Client Code Quality](https://owasp.org/www-project-mobile-top-10/2016-risks/m7-client-code-quality)
* [M8: Code Tampering](https://owasp.org/www-project-mobile-top-10/2016-risks/m8-code-tampering)
* [M9: Reverse Engineering](https://owasp.org/www-project-mobile-top-10/2016-risks/m9-reverse-engineering)
* [M10: Extraneous Functionality](https://owasp.org/www-project-mobile-top-10/2016-risks/m10-extraneous-functionality)

**M1: Improper Platform Usage**

Improper Platform Usage is a risk that is very important to identify. This is because it can have a significant impact on your data or devices. This risk involves the misuse of an operating system feature or a failure to use platform security controls properly.

This may include Android intents, platform permissions, the Keychain, or other security controls that are part of the platform.

**M2: Insecure Data Storage**

Data security can be defined as the security surrounding any data that is stored or transmitted. Data of android applications are stored in various locations like servers, mobile devices, and cloud storage. All of these locations are susceptible to attacks by hackers. To protect the data from these attacks, the data needs to be stored securely.

**M3: Insecure Communication**

 Insecure communication is sending sensitive data over non-secure channels. When sending data over non-secure channels, it can be intercepted by anyone who has access to this channel, which is everyone on the same network.

This means that if you are sending sensitive data, the data can easily be copied. This is very common in public WiFi access points. When using public WiFi access points, you should always assume that your data is being intercepted.

**M4: Insecure Authentication**

Authentication is a mechanism to prove a user’s identity to a system. It is also a process of initializing and maintaining a “state” on the system (e.g. a session or a login state), which can be used to determine the user’s identity.

Weak authentication is one of the root causes of many security risks. Attack vectors such as authentication bypass, information disclosure via debug messages, session invalidation are typical examples of insecure authentication.

**M5: Insufficient Cryptography**

While cryptography is a fundamental part of any app that stores user data, there is a common misconception that cryptography can solve all security problems. Cryptography is just a tool that helps to protect the data from attackers.

If any weak point is found in the cryptographic implementation, an adversary can still access sensitive information. In this blog post, we will walk you through the most common cryptography mistakes and how to avoid them.

**M6: Insecure Authorization**

Authorization is a process that ensures that only authorized individuals who are allowed to access the data are performing the access operation. Authorization is a crucial aspect of the CIA triad. Many mobile applications have improper authorization implemented due to which low-level users can access information of any high privileged user.

**M7: Client Code Quality**

Application code quality is the essential factor in ensuring the quality of the final product. As a developer, you should have several goals for your application. Many security flaws can occur in a mobile application, but the most common ones are SQL Injection, Cross-Site Scripting, and Buffer Overflows. The reason why these security flaws occur is because of the poor quality of the client code.

**M8: Code Tampering**

Code tampering is a process in which hackers or attackers exploit the existing source code of an application by modifying it with malicious payloads, which can lead to business disruption, financial loss, and loss of intellectual property.

The issue is usually found in the mobile apps that are downloaded from third-party app stores. These app stores are not associated with the official mobile application developers and usually distribute pirated apps.

**M9: Reverse Engineering**

Reverse Engineering is a process to decompile the mobile application to understand the application logic. Code obfuscation is done to prevent attackers from reading the application code and understanding the logic.

**M10: Extraneous Functionality**

Bad actors such as cyber-criminals or hackers try to understand the mobile application’s extraneous functionality. The main goal is to understand and explore hidden functionalities of the backend framework.

**Android Application Security Testing Methodology**

**What is android pentesting:**

Android penetration testing is the process of finding security vulnerabilities in the android application. It is a systematic approach to searching for weaknesses in an Android app, verifying the app’s security, and making sure it abides by the security policies. It includes trying to attack the android application by using various methods and tools.

Below is the security testing process of the android application pentesting.

Android application pentesting methodology can be classified into two categories

**Static Analysis**

**Dynamic Analysis**

**STATIC ANALYSIS**

Android applications are written primarily in Java, Kotlin, and C++. When distributed, they use the **.apk** extension which stands for **Android Package**. An APK is a ZIP file containing a predefined structure that is used to store different components of an application, like assets and bytecode, such as the one mentioned below.

Static analysis is used for studying an already packaged application and detecting code weaknesses without having direct access to the source code. With static analysis, we don’t look at the application’s behaviour at runtime, as we do during dynamic analysis. Hackers may use static analysis to detect the use of a weak encryption algorithm, for instance.

Automated static analysis of the mobile applications can be done by MobSF (Mobile Security Framework)

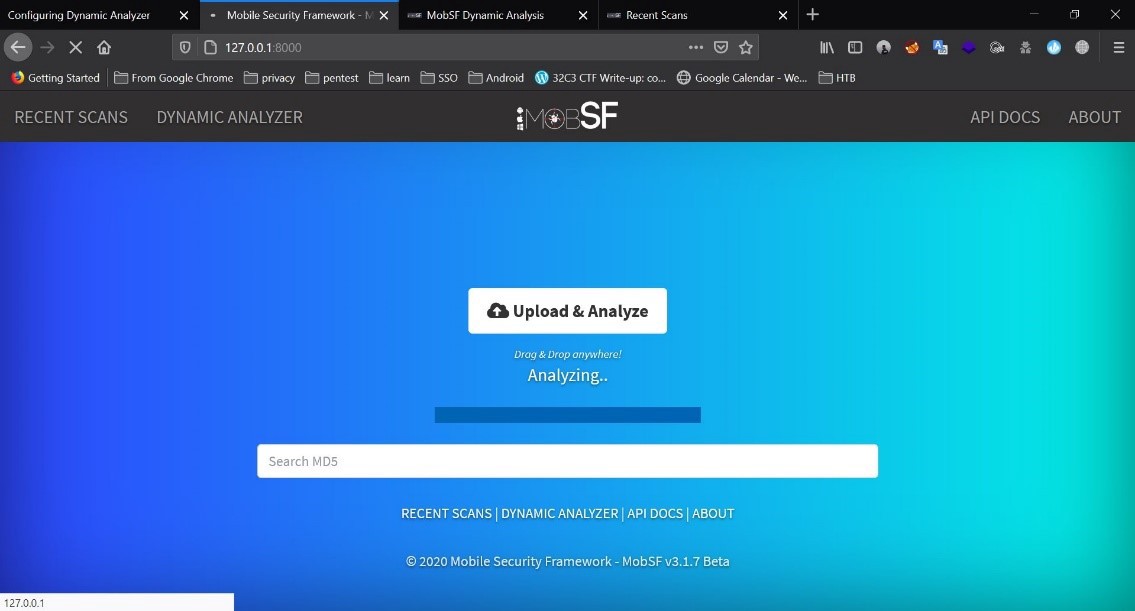


**MobSF Mobile Security Framework)**

Install the MobSF from [Getting Started (mobsf.github.io)](https://mobsf.github.io/docs/#/)

Go to browser and open localhost:8000

Upload and scan the given target apk.



Below is the screenshot of the MobSF result page after the static analysis.

After the static analysis of the android application in the MobSF, analyze the vulnerabilities given by the tool and perform the false positive analysis.

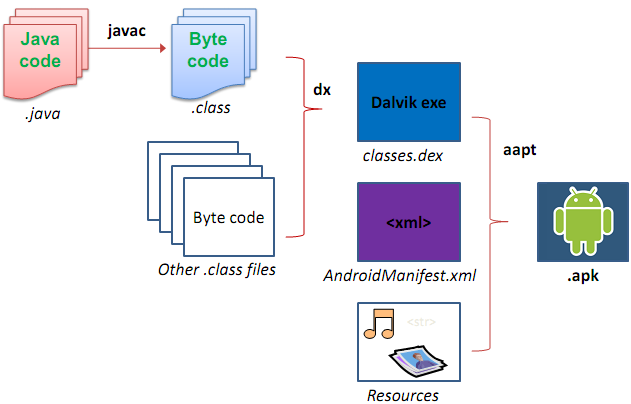
**Reverse Engineering: (OWASP M9: Reverse engineering)**

The first step of reverse engineering the android app is to decompile the app using the below tools

**Decompiling APKs:**

1. Unzipping Apk
2. APK tool [Apktool - How to Install (ibotpeaches.github.io)](https://ibotpeaches.github.io/Apktool/install/)
3. JADX
4. JD-GUI

After decompiling the Apks, below are the folders that we see after decompiling the application.. Browse and analyze the files in the apk manually and execute the static analysis test cases from the **OWASP Mobile checklist** v1.2 or 1.4.



**AndroidManifest.xml:** The AndroidManifest.xml file is the control file that tells the system what to do with all the top-level components (specifically activities, services, broadcast receivers, and content providers described below) in an application. This also specifies which permissions are required. This file may be in Android binary XML that can be converted into human-readable plaintext XML with tools such as [android-apktool](https://code.google.com/p/android-apktool/), or [Androguard](https://code.google.com/p/androguard/wiki/Usage" \l "Androaxml) which we will cover in the upcoming post.

**Android manifest file**

**META-INF directory:**

* + MANIFEST.MF: the Manifest File
  + CERT.RSA: The certificate of the application.
  + CERT.SF: The list of resources and SHA-1 digest of the corresponding lines in the MANIFEST.MF file.

**Lib**: The directory containing the compiled code that is specific to a software layer of a processor, the directory is split into more directories within it:

* + armeabi: compiled code for all ARM based processors only
  + armeabi-v7a: compiled code for all ARMv7 and above based processors only
  + x86: compiled code for X86
  + mips: compiled code for MIPS processors only

**Res**: the directory containing resources not compiled into resources.arsc

**Assets:** A directory containing applications assets, which can be retrieved by AssetManager.

**Classes.dex**: The classes compiled in the dex file format understandable by the Dalvik virtual machine

**Resources.arsc:** A file containing precompiled resources, such as binary XML for example.

* Open the android manifest file.XML in any text editor and analyse the file.
* Extract the java classes and read and analyse the java classes.
* Check for the hard coded sensitive information in the application package.
* And execute all the static analysis test cases according to the OWASP mobile checklist 1.2 or 1.4

**Dynamic analysis**

Dynamic analysis is used to find possible ways to manipulate application data when the application is running. Dynamic analysis can also be automated as well as manual.

For Dynamic analysis of the android application, We need to install the emulators such as Genemotion and Android studio.

Install the APK on the emulator or a physical android device.

**Android Debug Bridge (adb)** is a versatile command-line tool that lets you communicate with a device. Install the ADB tools on the system. ADB can be used to access the internal files and folders of the android device and the installed applications.

Below are the some of the commands of the adb:

* Adb
* Adb devices
* Adb install sample.apk
* Adb pull
* Adb push
* Adb forward
* Adb shell

**Installation link**: [Android Debug Bridge (adb)  |  Android Developers](https://developer.android.com/studio/command-line/adb)

**(M1: Improper Platform Usage)**

Check for the dangerous permissions and the other android components in the android manifest file.

Android components like exported activities, exported services, exported content providers.

Check for the insecure broadcast receivers, intents redirections, Web view vulnerabilities, Deep linking vulnerabilities.

**Dangerous permissions**

**Insecure data storage: (M2: Insecure data storage)**

Login into the android application and verify the xml files in the shared preferences folder.

Check for the keyboard cache in the /data/ folders. SQLite can be used to read the DB files.

Check for the file permissions such as MODE\_WORLD\_READABLE or MODE\_WORLD\_WRITABLE.

Check for the keys and sensitive information in the files strings.xml, build config etc.

Sensitive information in the build config file

**Application issues:**

**Root detection (M2: Insecure data storage)**

Root or jailbreak the android device and install the android application on the rooted phone. And launch the target application in the rooted phone. If the application is launching in a rooted android device, it is vulnerable. If the application is implementing root detection mechanism, try to bypass the root detection mechanism using the Frida.

>frida -l rootbeer.js -U -f com.apk.app --no-pause

**Screenshot protection**

Check and verify if the application is protected against a screenshot of the sensitive information in the target android application.

**Network Analysis: (M3: Insecure Communication, M5: Insufficient Cryptography)**

Setup the proxy in the phone in the wifi options. Go to Burpsuite to generate the CA certificate and import the certificate into the emulator and add the certificate into the trusted root certificate.

Capture the traffic of the android application using the emulator.

If SSL pinning is enabled in the application, we need to disable the SSL pinning using Frida or objection.

Check for the Cryptographic algorithms used and also the SSL protocols.

Weak SSL Ciphers

**Access control issues: (M6: Insecure Authroization)**

Check for the broken access control issues in the target android application.

Check for the Insecure direct object reference vulnerabilities.

If there are different roles with in the same android application, try for the horizontal access controls, vertical access control issues.

**Input Validation issues: (M7: Client Code Quality)**

Browse and crawl the entire android application and make a note of all the input points and test for the input injection vulnerabilities and other vulnerabilities.

For eg: On the login page, try for injection vulnerabilities such as SQL injection vulnerabilities.

Check for URI schemes and URL redirections.

Unvalidated input

**Insecure Logging: (M10: Extraneous Functionality)**

Use Pidcat and Logcat to check the sensitive information in the application logs.

Check if the android application is debuggable in the android manifest file.

Sensitive information in the logs of the android application

**References and Guides:**

Owasp MSTG Guide v1.4

Mobile checklist [https://mobexler.com/checklist.htm](https://mobexler.com/checklist.htm%20)

Owasp Mobile testing checklist v 1.4

Hackers Handbook -Android

**IOS Application Pentesting**

**Static Analysis**

Static analysis can help detect a number of storage security issues to name few hardcoded sensitive info, Stealing App data in backup, Leaking sensitive data inside device log with NS log etc.



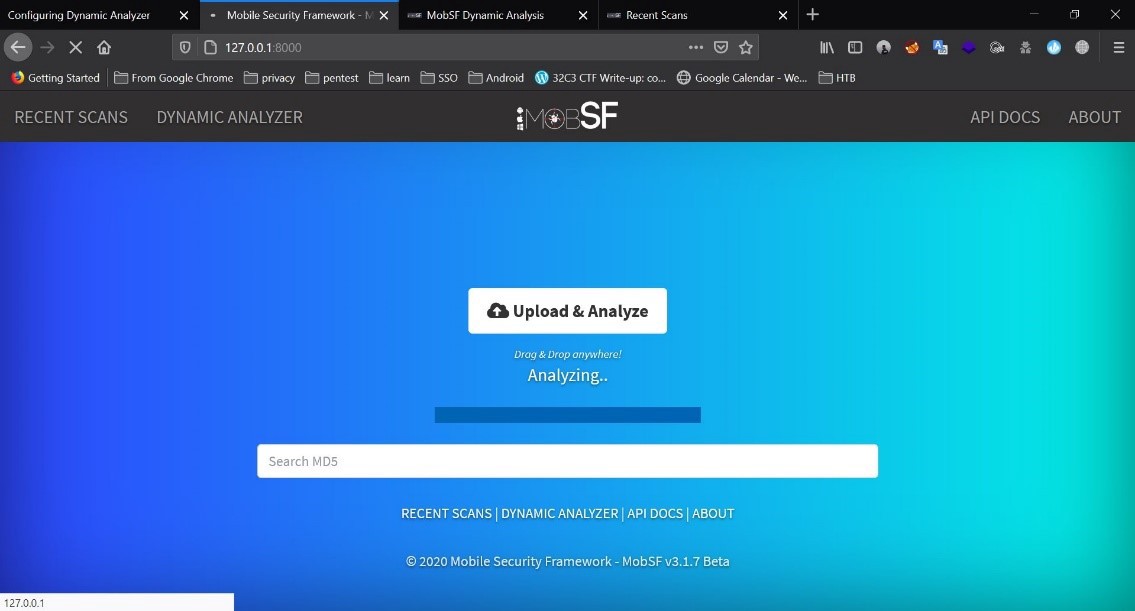
MobSF can be used for the automated static analysis of the IOS application.

**MobSF (Mobile Security Framework)**

Install the MobSF from [Getting Started (mobsf.github.io)](https://mobsf.github.io/docs/#/)

Go to browser and open localhost:8000

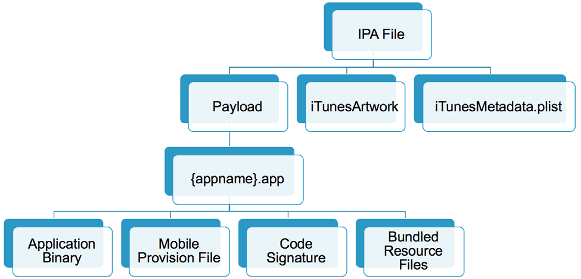
Upload and scan the given target ipa file.



Below is the screenshot of the MobSF result page after the static analysis in the MobSF

After the static analysis of the IOS application in the MobSF, analyze the vulnerabilities given by the tool and perform the false positive analysis.

**IPA architecture**



iOS applications are bundled in a single file of type ipa (short for iOS App-store Package).

Although they come with an ipa suffix they are just regular ZIP files, you can try modifying the suffix to zip and then extracting them.  
Since IPA files all have a well-defined internal structure, What you’ll find inside is a single folder called, inside of which will be another folder called .app. Besides, the Payload folder, you may also find other folders and files, usually for use of iTunes service.

**Plist in the App directory:**

plist (Property List) is a flexible and convenient format for storing application data. It was originally defined by Apple, for use in iPhone devices. It decides what icon to use for a bundle, what document types an app can support, and many other behaviours that have an impact outside the bundle itself. These files may contain sensitive data like Gmaps API key, etc.

**Info.plist from Health first IOS App**

**Contents of plist file:**

* **Version information:** The current version of the application is shown in the plist file.
* **Display name of the application**: The name that is displayed in the iphone. It is stored as ‘CF bundle Display name’ in plist file.
* **Application permissions:** The description given by the application for usage of user’s sensitive permissions. It is specified in the keys ‘NS<requirement>UsageDescription’.
* **Supported iOS version and devices** — The specification of the iOS version and device models supported by the application. Devices are specified by the key ‘UISupportedDevices’ and the minimum iOS version by ‘MinimumOSVersion’.
* **Application Transport Security** — The information regarding URLs/domains the application is allowed or disallowed to load. It is specified in the key ‘NSAppTransportSecurity’

**NsUserdefault:**

The user defaults is a .plist file in your app’s package and you can use it to set and get simple pieces of data. It’s structure is very similar to that of a dictionary and the user defaults are often regarded as a key-value store.

**Insecure Transport layer**

App Transport Security (ATS) is disabled on the domain {‘NSAllowsArbitraryLoads’: True}’. Disabling ATS can allow insecure communication with particular servers or allow insecure loads for web views or for media, while maintaining ATS protections elsewhere in your app.  
The NSAllowArbitraryLoads key is set to NO by default. Setting the key to YES will opt-out of ATS and its associated security benefits.

**Insecure version of OS installation allowed:**

Allowing application installation on devices with insecure versions of iOS may expose the application to a large number of vulnerabilities and attacks and also pose a significant threat to user’s data.

**Hardcoded Api keys:**

Almost all iOS apps need private values, such as API keys, secrets and passwords. Some of these may need to be used in the source code, to setup third-party SDKs or backend APIs. Some secrets may be needed during the build process or to use developer tools, such as communicating with an Apple Developer account.

**Username, passwords in the health first IOS application**

IOS Application dynamic analysis can be done using OSX or physical IOS device or any emulator.

**Dynamic analysis**

Jailbreaking is the privilege escalation of an Apple device for the purpose of removing software restrictions imposed, With a jailbroken device, you can install apps and tweaks that aren’t authorized by Apple, but you also remove the tough security protections that Apple has built into iOS.  
You can perform Tethered or Untethered jailbreak.

**1. Tethered:** This is a temporary jailbreak type. Once the device is rebooted, the device no longer remains in the jailbreak state.

**2. UnTethered Jailbreak:** The untethered jailbreak is a permanent type of jailbreak where even after rebooting the device, it will be in jailbreak state only.

Corellium can also be used for the dynamic analysis of the IOS application. Corellium has the option to select the jailbroken iphone while configuring.

Graphical user interface, application

Description automatically generated

**Setting up the proxy:**

Enable manual proxy in the wireless setting and add proxy details.

In the Burp Suite, go to “Proxy Settings” and set the listener to All Interfaces.

Now, on the iPhone, open “http://burp” and download the CA certificate.

Install the CA certificate.

Now, Navigate to Settings → General → About → Certificate Trust Settings and enable Portswigger CA Certificate.

**Jailbreak detection bypass:**

Install the IOS application on a jail broken device and check if the application is showing any alert or blocking the application to run. If the application is not blocking, then it is not protected with jailbreak detection.

If the application is implemented with jailbreak detection, try to bypass the jailbreak detection using Frida and objection.

**Fingerprint bypass:**

**SSL Pinning bypass:**

**Sensitive data in keychain**

**M1: Improper Platform Usage**

The first category in the list of OWASP Mobile Top 10 covers the aspects where an attacker can abuse the security control features provided by the iOS platform. In addition, sometimes due to failure in the proper implementation of platform features, multiple security issues could arise.

There are multiple components that are required to be analyzed under this category such as Biometric Authentication, Keychain, Platform Permission, etc.

Some of the general risks that occur under this category are:

* Bypassing Local Biometric (Touch/Face ID) Authentication
* Excessive Permissions
* Poorly Configured/Weak Permissions

Permissions in info.plist file

**M2: Insecure Data Storage**

The iOS applications often store some data locally in different components. There is a chance of getting sensitive data stored in the local storage that could be exploited if an attacker has access to the physical device of the user.

Some of the general risks that occur under this category are:

* Sensitive Data Stored in Plist

* Sensitive Data Stored in UserDefaults
* Sensitive Data Stored in Keychain
* Sensitive Data Stored in Core Data
* Sensitive Data Stored in Webkit Caching
* Sensitive Data Stored in Realm
* Sensitive Data Stored in Couchbase Lite
* Sensitive Data Stored in YapDatabase
* URL caching (both request and response);
* Keyboard press caching
* Copy/Paste buffer caching
* Insecure Logging
* Background Screen Caching
* Analytics data sent to 3rd parties.

**M3: Insecure Communication**

The iOS application communications on a client-server architecture. Due to improper/weak implementation of the communication standards such as communication over HTTP, an attacker can attempt to steal sensitive information over unencrypted checks, perform Man-in-the-Middle or attempt to analyze the request/response by capturing them with proxy tools such as Burp Suite.

Some of the general risks that occur under this category are:

* Communication over HTTP
* Plain-Text Submission of Sensitive Information
* Logging Sensitive information in the Logs
* **Lack/Bypass-able SSL Pinning**

SSL Certificate Pinning, or pinning for short, is the process of associating a host with its certificate or public key. Once you know a host’s certificate or public key, you pin it to that host. In other words, you configure the app to reject all but one or a few predefined certificates or public keys.

**Using Objection and Frida.**

**Frida** it’s a dynamic code instrumentation toolkit. It lets you inject snippets of JavaScript or your own library into native apps on Windows, macOS, GNU/Linux, iOS, Android, and QNX. Frida also provides you with some simple tools built on top of the Frida API.

**Objection** is a runtime mobile exploration toolkit, powered by Frida, built to help you assess the security posture of your mobile applications, without needing a jailbreak.

**M4: Insecure Authentication**

When an application fails to properly perform the authentication checks or allows an attacker to manipulate the login/authentication requests to gain access to the victim user’s account, it is generally considered under the Insecure Authentication category.

Some of the general risks that occur under this category are:

* Guessable/Weak/Default Credentials
* Authentication Bypass using attack such as Injection Attacks

**M5: Insufficient Cryptography**

The iOS applications tend to store the user data or utilize the user data in different client-server requests. Due to a lack of strong cryptography, an attacker may attempt to access the data that is encrypted using the weak cryptographic method and gain hold of sensitive information.

Some of the general risks that occur under this category are:

* Stealing App and User Data
* Accessing Encrypted Files
* Accessing Encrypted Client-Server Request Endpoints

**M6: Insecure Authorization**

The insecure authentication and insecure authorization are usually two checks which cause confusion. This attack talks about abusing the weak implementation of user authorization checks to perform attacks out of the user’s privilege and access level.

Some of the general risks that occur under this category are:

* IDOR
* Privilege Escalation
* Direct Request

**M7: Client Code Quality**

Due to the poor code quality, often, an attacker may attempt to pass the crafted inputs to function calls made within an app in an attempt to execute them or observe the application’s behaviour. It may lead to some mal-function and exploitable scenarios in the application.

Some of the general risks that occur under this category are:

* Format String Vulnerabilities
* Buffer Overflows
* Remote Code Execution
* Memory Exhaustion

**M8: Code Tampering**

If the application does not implement the code tampering checks, an attacker may attempt to modify the application code and inject malicious code such as a backdoor and distribute it using App Store and other methods that could lead to stealing sensitive information and comprising the user’s device as well.

Some of the general risks that occur under this category are:

* Malware Injection
* Stealing Sensitive Data
* Persistent Backdoor

**M9: Reverse Engineering**

The reverse engineering approach involves analyzing the binary and it’s code using tools such as Hopper, otool, IDA Pro in order to understand the application’s code patterns, various function implementation and performing bypass/attacks using runtime & dynamic instrumentation approaches.

One of the common examples is to analyze the implementation of SSL Pinning logic and understand the functions that are responsible for pinning checks and using tools such as Frida to bypass it by writing a custom logic script.

**M10: Extraneous Functionality**

In general, before pushing an application to production, the development team keeps code to have easy access to the backend server, create logs to analyze errors or carry staging information and testing details. This code is extraneous to the functioning of the app. Basically, It has no use for the intended use once the app is in production and it is required only during the development cycle.

In certain cases, this code can carry information related to databases, user details, user permissions, API endpoints, etc. or disable functionalities like two-factor authentication.

**References and Guides:**

Owasp MSTG Guide v1.4.0

Mobile checklist [https://mobexler.com/checklist.htm](https://mobexler.com/checklist.htm%20)

Owasp Mobile testing checklist v 1.4