

What is OWASP Mobile Top 10?

OWASP Mobile Top 10 is a list that identifies types of security risks faced by mobile apps globally. This list, which was last updated in 2016, is an acting guide for developers to build secure applications and incorporate best coding practices. With almost 85 percent of apps tested by NowSecure found to be affected by at least one of the OWASP Top 10 risks, it becomes essential for developers to understand each one of them and adopt coding practices that nullify their occurrence as far as possible.

Enlisted below are the OWASP Mobile top 10 risks, which are marked from M1 to M10.

M1: Improper Platform Usage

This risk covers 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. Its occurrence is common, with average detectability, and can cause a severe impact on the affected apps.

Improper Platform Usage Risks

Data Leakage by Exploiting Android Intent

Android intents are messaging objects within the operating system that allow communication between different activities. These actions include communicating with background services, accessing data stored in the mobile device or another app's server, broadcasting messages during the change of events, starting or ending an activity like opening a browser or another app, etc. Since there are endless uses for intents, the possibility of data leakage during these message exchanges also becomes high.

Android Intent Sniffing

Many apps within the Android ecosystem are designed primarily to steal information from intents. These apps can study URL patterns or user information when it is in transit between the legitimate app and other Android components.

iOS Keychain Risk

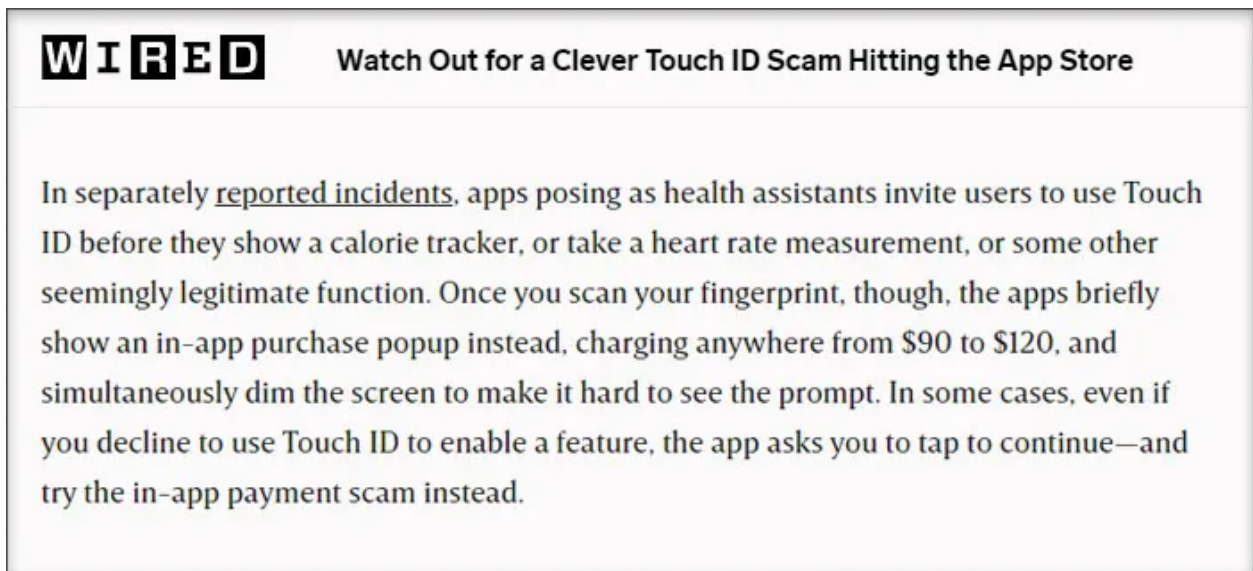
The Keychain is a secure storage facility that allows a mobile user to create hard-to-remember passwords, which are more difficult to crack, thus making third-party accounts – like bank and email accounts – accessed on mobile devices extra secure. iOS provides Keychain encryption out of the box so that the developer is not required to introduce their own encryption methods. Using access control lists and Keychain access groups, the developer can decide which apps and data require encryption and which

can be left open. If the user does not opt for the Keychain option, they may intuitively choose easy-to-remember passwords, which could be exploited by hackers.

iOS TouchID Risk

iOS allows the TouchID option to be used by developers for authenticating their mobile apps. Bypassing the TouchID option makes the authentication process vulnerable to hacking attempts.

M1: Improper Platform Usage Example

A screenshot of a WIRED article snippet. The WIRED logo is in the top left. The title is "Watch Out for a Clever Touch ID Scam Hitting the App Store". The text describes how apps posing as health assistants use Touch ID to trick users into making in-app purchases.

WIRED Watch Out for a Clever Touch ID Scam Hitting the App Store

In separately reported incidents, apps posing as health assistants invite users to use Touch ID before they show a calorie tracker, or take a heart rate measurement, or some other seemingly legitimate function. Once you scan your fingerprint, though, the apps briefly show an in-app purchase popup instead, charging anywhere from \$90 to \$120, and simultaneously dim the screen to make it hard to see the prompt. In some cases, even if you decline to use Touch ID to enable a feature, the app asks you to tap to continue—and try the in-app payment scam instead.

Best Practices to Avoid Improper Platform Usage

iOS Keychain Best Practices

Do not allow Keychain encryptions through the server route and instead keep the encrypted keys in one device only, so that it cannot be exploited in other devices or the server.

Secure the app by using the Keychain to store the app's secret which should have a dedicated access control list. The user authentication policy of the access control list can be enforced by the OS.

iOS Android Intent Best Practices

Take the permission route to restrict which apps are allowed to communicate with their app, thus practically blocking all attempts from non-whitelisted traffic. Another option is to not allow an export option for intents for either or all of the activities, services and broadcast receivers with the Android framework so that Android components that have no reason to communicate with the app are kept at the outset.

Android Intent Sniffing Best Practices

This leakage can be controlled by defining explicit intents, where the intent object is clearly defined, thus blocking every other component to access the information contained in the intent. Also, check file permissions thoroughly before making the app public to ensure the required permissions are in place.

M2: Insecure Data Storage

The OWASP marks M2 exploitability as “easy”, prevalence “common”, detectability “average”, and impact “severe”. This risk in the OWASP list informs the developer community about easy ways in which an adversary can access insecure data in a mobile device. An adversary can either gain physical access to a stolen device or enter into it using malware or a repackaged app.

In the case of physical access to the device, the device’s file system can be accessed after attaching it to a computer. Many freely available software allows the adversary to access third-party application directories and the personally identifiable data contained in them.

Insecure Data Storage Risks

Compromised File System

While an obvious drawback of a compromised file system is the loss of personal information of the user, the app owner may stand to lose too due to the extraction of the app’s sensitive information via mobile malware, modified apps or forensic tools. From the user’s perspective, this kind of data compromise could lead to identity theft, privacy violation, and fraud for the individual user and reputation damage, external policy violation, and material loss in the case of business users.

An adversary looks at multiple places in a compromised device for unsecured data. They include SQL databases, log files, XML data stores, binary data stores, cookie stores, SD cards, and cloud-synced, which take place largely through vulnerabilities present in or happening due to the operating system, frameworks, compiler environments, new hardware, and rooted/jailbroken devices.

The exploitation of Unsecured Data

The exploitation of unsecured data becomes possible due to developers’ ignorance of how a device stores cache data, images, key presses, and buffers. Analysts have observed that a lack of proper technical documentation of these processes at the

operating system and development framework level allows developers to ignore these security processes and, thus, give a handle to hackers to manipulate data or processes in a device.

M2: Insecure Data Storage Example

FORTUNE

Popular [dating apps](#) such as [OkCupid](#), [Tinder](#), and [Bumble](#) have vulnerabilities that make users' personal information potentially accessible to stalkers, black mailers, and hackers.

The security lapses, which vary in terms of their severity and feasibility, could expose people's names, login information, location, message history, and other account activity, warned researchers at [Kaspersky Lab](#), a Moscow-based [cybersecurity firm](#) that's been the [subject](#) of recent [controversy](#) in the U.S., [in a new report](#).

Best Practices to Avoid Insecure Data Storage

iGoat iOS

For iOS devices, the OWASP recommends using purposefully vulnerable mobile apps, like iGoat, to threat model their apps and development frameworks. This process allows developers to understand how APIs deal with information assets and app processes, like URL caching, key-press caching, buffer caching, Keychain usage, application backgrounding, logging, data storage, browser cookie management, server communication, and traffic sent to third parties.

Android Debug Bridge

Android developers can use Android Debug Bridge (ADB) shell to check file permissions of the targeted app and a database management system, like sqlite3, to check database encryption. ADB also offers commands like 'logcat' to allow developers to check error logs contained in Android logs, which can leak sensitive user or security information to a malware. While it is common knowledge that extensive error logs help developers during the development process, but if left unattended, these logs can lead to a compromised app or data. An Android developer should also use tools like Android Device Monitor and Memory Analysis Tool to ensure that the device's memory does not

have any unintended data for an unspecified duration, which could be exploited by a hacker or unauthorized person who can obtain physical access to the device.

M3: Insecure Communication

Data transmission to and from a mobile app generally takes place through a telecom carrier and/or over the internet. Hackers intercept data either as an adversary sitting in the local area network of users through a compromised Wi-Fi network, tapping into the network through routers, cellular towers, proxy servers, or exploiting the infected app through malware.

Insecure Communication Risks

Stealing of Information

Out of these categories, monitoring traffic through compromised or unsecured Wi-Fi networks is the easiest way for an adversary to steal information. However, the OWASP expects developers to watch all outbound and inbound traffic to a mobile device including TCP/IP, Wi-Fi, Bluetooth/Bluetooth-LE, NFC, audio, infrared, GSM, 3G, SMS, etc.

Man in The Middle (MITM) Attacks

While mobile developers are generally aware of using SSL/TLS for authentication, they do not validate these certificates properly, leaving scope for network attackers to mount man-in-the-middle (MITM) attacks. Such attacks allow the adversary to view and modify traffic sent between an app and its server and intercept session IDs. Since security certificates are domain-specific, they are not available on testing servers. Developers tend to accept self-signed certificates on production servers when they test codes. This leaves a gap for MITM attacks, as a self-signed certificate is as good as an unencrypted or plaintext connection. In cases where developers disallow self-signed certificates, attackers tend to exploit the permissive hostname verifier option pushed by developers. If all hostnames are allowed, attackers can simply use any valid certificate issued by a certificate authority and use it to gain control over app-server traffic.

Admin Account Compromise

The real danger of a MITM attack is not when an adversary steals user data, but when insecure communication allows for data theft of the admin account. This can lead to hacking of the entire website and all its sensitive data. Such an attack can also impact or steal encryption keys, passwords, private user information, account details, session tokens, documents, metadata, and binaries.

M3: Insecure Communication Example

Penetration testers from Rapid7 conducted a hacking exercise to identify potential loopholes in children's GPS watches.

RAPID7 Blog

Exploitation and mitigation

Given an unchanged default password and a lack of SMS filtering, it is possible that an attacker with knowledge of the smart watch phone number could assume total control of the device, and therefore use the tracking and voice chat functionality with the same permissions as the legitimate user (typically, a parent).

Best Practices to Avoid Insecure Communication

Developers should incorporate the following suggestions from the OWASP to address insecure communication:

- Assume that the network layer is not secure and is susceptible to eavesdropping
- Watch out for leakages over the traffic communicated between an app and the server. Also check the device that holds the app and another local device or a local network, including wired networks
- Apply SSL/TLS to transport channels that the mobile app will use to transmit sensitive information, session tokens, or other sensitive data to a backend API or web service
- Account for outside entities like third-party analytics companies, social networks, etc. by using their SSL versions when an application runs a routine via the browser/webkit.
- Avoid mixed SSL sessions as they may expose the user's session ID
- Use strong, industry-standard cipher suites with appropriate key lengths
- Use certificates signed by a trusted CA provider
- Establish a secure connection only after verifying the identity of the endpoint server using trusted certificates in the key chain
- Alert users through the UI if the mobile app detects an invalid certificate
- Do not send sensitive data over alternate channels (e.g, SMS, MMS, or notifications)

- If possible, apply a separate layer of encryption to any sensitive data before it is given to the SSL channel. In the event that future vulnerabilities are discovered in the SSL implementation, the encrypted data will provide a secondary defense against confidentiality violation

M4: Insecure Authentication

This problem occurs when a mobile device fails to recognize the user correctly and allows an adversary to log into the app with default credentials. This typically happens when an attacker fakes or bypasses the authentication protocols, which are either missing or poorly implemented, and interacts directly with the server using either malware which sits in the mobile device or botnets, thus establishing no direct communication with the app.

Insecure Authentication Risks

Input Form Factor

Insecure input form factors are a common source of manipulation in mobile devices since app makers and mobile platforms encourage easily accessible four- or six-digit passwords for ease of access. Apart from a weak input form factor, unreliable internet access on mobile devices forces developers to adopt an offline-online approach to authenticate sessions.

Insecure User Credentials

A technical impact of insecure authentication is that when an app cannot determine the user credentials properly, it is also unable to record user activity correctly. If such a user exploits the data or code from or transmissions to and from the device, the security team cannot correctly establish the source and nature of the attack. Moreover, insecure authentication also plays havoc with user permissions in the device, since the operating system will not know exactly what role to assign to the user who has not been properly authenticated.

M4: Insecure Authentication Example

FBI warns about attacks that bypass multi-factor authentication (MFA)

- In 2019 a US banking institution was targeted by a cyber attacker who was able to take advantage of a flaw in the bank's website to circumvent the two-factor authentication implemented to protect accounts. The cyber attacker logged in with stolen victim credentials and, when reaching the secondary page where the customer would normally need to enter a PIN and answer a security question, the attacker entered a manipulated string into the Web URL setting the computer as one recognized on the account. This allowed him to bypass the PIN and security question pages and initiate wire transfers from the victims' accounts.

Best Practices to Avoid Insecure Authentication

Carefully study the authentication scheme of the app and test it with binary attacks in offline mode to determine if it can be potentially exploited. The security team should check if the app is letting a POST/GET command be executed to establish a connection with the server without an access token. A successful connection establishes vulnerabilities in the app. The developer should ensure that passwords and security keys are not stored locally on the mobile device. Such data is extremely prone to manipulation.

The security team should try to implement as many of the following methods to secure the mobile app from insecure authentication:

- The security protocols of the web app should match those of the mobile app in terms of complexity and authentication methods;
- Use online authentication methods, just as in a web browser. If the ease of access takes precedence in a mobile device, try to guard against binary attacks, which are described in detail in M10 risk;
- Do not allow the loading of app data unless the server has authenticated a user session. Storing app data locally can create faster loading, but it can compromise on user security;
- Wherever local storage of data becomes an eventuality, make sure that it is encrypted with an encrypted key derived from the login credentials of the user, thus forcing the app to authenticate the app data at least once;

- Persistent authentication requests should be stored on the server and not locally. Always remember to caution the user when they opt-in for the remember-me option;
- The security team needs to be careful with using device-centric authorization tokens in the app since a stolen device app becomes vulnerable. A device-centric authorization token for an app ensures that the app cannot be accessed if the new owner of the device changes the device access password;
- Since unauthorized physical access of mobile devices is common, the security team should enforce periodic authentication of user credentials and logouts from the server-side;
- Make sure that the user is forced to choose alphanumeric characters for passwords. They are more secure than simple letter combinations. Alongside this, the developer can force the user to identify an image or word before allowing access to the app. The two-factor authentication method is fast gaining currency.

M5: Insufficient Cryptography

Data in mobile apps become vulnerable due to weak encryption/decryption processes or infirmities in the algorithms that trigger encryption/decryption processes. Hackers can gain physical access to the mobile device, spy on network traffic, or use malicious apps in the device to access encrypted data. Using flaws in the encryption process, its aim is to decrypt data to its original form in order to steal it or encrypt it using an adversarial process and thus render it useless for the legitimate user.

Insufficient Cryptography Risks

Stealing App and User Data

Both Android and iOS force encryption of app codes using certificates issued by trusted sources, which they decrypt in the device memory after verification of the encryption signature when an app is called by the user. However, many commonly available tools allow bypassing of this method. These tools can be used to download an app in a jailbroken device, decrypt it there, and take a snapshot of the decrypted app back to the original device's memory before the app's execution by the user. Once the app executes in this compromised state, the hacker can further analyze the app to conduct binary attacks or steal user and app data. Any developer who relies on the default encryption process provided by the operating system runs the risk of code manipulation.

Access Encrypted Files

Many developers mishandle encryption keys, which allows adversaries to gain control of the encrypted files even if they have been secured using the best possible algorithms. Developers also tend to place encryption keys in the same directories as the encrypted data. This makes it easier for hackers to access the keys and use them for decryption.

M5: Insufficient Cryptography Example



Best Practices to Avoid Insufficient Cryptography

- Choose modern encryption algorithms to encrypt apps. The choice of the algorithm takes care of this vulnerability to a large extent, as an encryption algorithm from a trusted source generally is tested by the security community frequently
- The National Institute of Standards and Technology of the US government publishes cryptographic standards from time to time and recommends encryption algorithms. The developer should keep an eye on this document to watch out for emerging threats.

M6: Insecure Authorization

Many people confuse the M4 risk with M6 risk since both of them are about user credentials. Developers should keep in mind that insecure authorization involves the

adversary taking advantage of vulnerabilities in the authorization process to log in as a legitimate user, unlike insecure authentication, in which the adversary tries to bypass the authentication process by logging in as an anonymous user.

Insecure Authorization Risks

Unregulated Access to Admin Endpoints

Under the M6 risk, once the attacker gains access to an app as a legitimate user, the next task for them is to gain administrative access by force-browsing to an endpoint where it can execute admin commands. Attackers typically use botnets or malicious programs in the mobile device to exploit authorization vulnerabilities. The result of this security compromise is that the attacker can mount binary attacks on the device in the offline mode.

IDOR Access

In some cases, the authorization scheme lets an adversary run insecure direct object references, commonly known by its acronym IDOR, where it can obtain access to an object, like databases or files, simply by providing user-supplied inputs. Such leakages can destabilize the whole operating system or lead to loss of data and reputation.

M6: Insecure Authorization Example

Hacking smart car alarm systems

Systems designed to guard against car theft can be used to track, immobilize, and steal vehicles.

Having hijacked your account and logged into the app in your name, a cybercriminal gains access to a mass of data and all smart alarm functions. A simple change of password will lock you out of the system. The attacker will be able to:

- Track all vehicle movements,
- Enable and disable the alarm system,
- Lock and unlock the car doors,
- Enable and disable the immobilizer, an antitheft tool that prevents the engine from starting,
- Cut the engine — in some cases even while the car is moving.

Best Practices to Avoid Insecure Authorization

- Continuously test user privileges by running low-privilege session tokens for sensitive commands, which are reserved for high-privilege users. If commands can be executed successfully, immediately check the authorization scheme of the app
- Developers should keep in mind that the user authorization scheme usually goes wrong in offline mode. However, in certain cases developers allow user permissions and roles to be sent to the server, which can also cause vulnerability in the authorization scheme.
- Run authorization checks for roles and permissions of an authenticated user at the server rather than from the mobile device. The chances of authentic users exploiting high-privilege functionalities reduce in the backend verified user management schemes rather than when they are verified within the mobile device.

M7: Poor Code Quality

The M7 risk emerges from poor or inconsistent coding practices, where each member of the development team follows a different coding practice and creates inconsistencies in the final code or does not create enough documentation for others to follow. The saving grace for developers here is that even if the prevalence of this risk is common, its detectability is low. Hackers cannot easily study the patterns of poor coding and often require manual analysis, which is not easy to do. Automatic tools, which are employed to identify memory leaks or buffer overflows using fuzz testing, can help access information but do not easily allow the execution of foreign code in the mobile device.

Poor Code Quality Risks

Safe Web Code, Compromised in Mobiles

Mobile code can compromise an otherwise secure app that performs well in web browsers by letting a threat agent push untrusted inputs whenever code subsets are called within a mobile device. Perse, such a mobile code may not be malicious, but by allowing the execution of untrusted code in the device, it can severely compromise the user information. Typical vulnerabilities in this category include memory leaks and buffer overflows.

Lacunae in Third-Party Libraries

Developers should be careful while integrating popular libraries in their apps. Even established players inadvertently offer compromised libraries, which creates a security problem for app owners. Often, developers do not keep a track of newer versions of third-party libraries where the library developer may have corrected the bad code of earlier versions, thus letting adversaries exploit an app that could be easily secured.

Client Input Insecurity

In apps created for specific clients, developers write code to accept all input as safe. This practice can lead to content provider attacks since a content provider call can contain sensitive information. An attacker can also call a content provider and obtain access to unsecured information.

M7: Poor Code Quality Examples

The Hacker News

Hackers Used WhatsApp 0-Day Flaw to Secretly Install Spyware On Phones

Apparently, the vulnerability, identified as CVE-2019-3568, can successfully be exploited to install the spyware and steal data from a targeted Android phone or iPhone by merely placing a WhatsApp call, even when the call is not answered.

Also, the victim would not be able to find out about the intrusion afterward as the spyware [erases](#) the incoming call information from the logs to operate stealthily.

Best Practices to Avoid Poor Code Quality

Mobile-Specific Code

The simplest solution to fix this issue is to rewrite code in the mobile device rather than looking to fix problems at the server-side. Developers should keep in mind that poor coding at the server level is different from the one at the client-side. A code issue at the server side will reflect in the web view of the app too, but the mobile-side bad coding will impact only the mobile user.

Static Analysis

The developer should consistently use third-party tools for static analysis to identify memory leaks and buffer overflows. The development team should try to eradicate the mismatch between the length of incoming buffer data and the target buffer.

Code Logic

The developer should avoid simple logic in codes, as it is known to be hackers' favorite both in iOS and Android devices. Hackers can change a value in the code with simple logic and circumvent the whole security apparatus. Such codes can be attacked at assembly and runtime levels. The developer could control this leakage by stopping untrusted sessions from enforcing privileges at the device level and instead activate them at the server. It is also recommended to withhold the permissions until a session has been authenticated using OTP, secret questions, or challenges.

Library Version

The development team should create a list of all third-party libraries used in the app and check for their newer versions periodically, even if it has used libraries only from trusted sources.

Content Provider

Developers should consider all client input to be untrusted and validate it irrespective of whether it comes from the app or the user. They should carefully set permission flags on content provider inputs to stop all unauthorized access.

M8: Code Tampering

Hackers prefer code tampering of apps over other forms of manipulations, as it allows them to gain unbridled access to the app, the user behavior, or even the whole mobile device. They tend to push users to download tampered versions of popular apps from third-party app stores through phishing attacks and misleading advertisements.

Code Tampering Risks

Malware Infusion

Once a user has been successfully prompted to download a tampered app, they have either downloaded and installed the app whose core binary has been changed or resource package altered. Tampered apps allow hackers to change system APIs to allow the execution of malicious foreign code in the mobile device. Hackers, then, tend to indulge in binary patching, modification of resident code in the device, memory modification, and data theft among others.

Data Theft

Often, extra features are offered in modified apps that do not exist in original apps, thus it becomes an incentive for users to adopt them. The prevalence of tampered apps is so common that companies invest huge amounts of resources in detecting and removing duplicate apps from app stores and educating users about possible data theft scenarios.

However, their code has to reside in an environment that is beyond their watch. Hackers can also exploit lacunae in the operating system to impact the code of a legitimate app. Moreover, where users allow rooting or jailbreaking of devices, they knowingly create options for third parties to manipulate resident code in the device.

Thus, developers should not conclude that all tampering is undesirable. Some cases may be inconsequential, while others could be deliberate on the part of users. But, in the cases of financial or gaming apps, developers need to be extra careful. In gaming apps, for example, tampered features allow users to bypass freemium options and jump

to the next stage for which they would have to pay otherwise. Through such a lucrative offer for users, the hacker could easily insert spyware in their devices and steal their user information.

M8: Code Tampering Example



Best Practices to Avoid Code Tampering

Runtime Detection

The developer should ensure that the app can detect code change at runtime. If a tampered app wants to run in a rooted or jailbroken device and the developer does not wish to allow this kind of execution, it is best to report this compromise to the server at runtime itself. RASP is one such technology that developers can use to detect and deter attack vectors in real-time.

Checksum Changes

The developer should use checksums and evaluate digital signatures to see if file tampering has taken place. Since code and file tampering almost always changes the checksum value, it is the easiest way to determine adversarial action.

Data Erasure

Ensure that the app code, keys, and data are erased once tampering has been detected. Such a provision will destroy the very rationale of tampering and discourage hackers from targeting the same app again.

M9: Reverse Engineering

Reverse engineering of mobile code is a commonly exploitable occurrence. Hackers tend to use external, commonly available binary inspection tools, like IDA Pro, Hopper, otool, etc., to study the original app's code patterns and its links with server processes.

Reverse Engineering Risks

Dynamic Inspection at Runtime

Some languages – like Java, .NET, Objective C, Swift – are more susceptible to reverse engineering than others, as they allow dynamic inspection at runtime. Among other damages, reverse engineered code can impact the security of servers, data contained in mobile devices, and the server's ability to detect jailbroken or rooted devices.

Code Stealing

Reverse engineering can be used by app competitors to see the app's functionalities threadbare and even copy some features stealthily. This way, the cost of developing new code is reduced.

Premium Features

Hackers may use this technique to access the premium features of the app by bypassing the authentication process. Game cheats can gain an unfair advantage over their competing peers through this method.

M9: Reverse Engineering Examples



How Pokémon Go Fans Hacked ‘Em All: And How to Prevent Similar Reverse-Engineering

Reverse engineering the geolocation service and serving it **false geodata**, players were able to “control” their character in-game, ignoring the app’s basic premise of capturing Pokémon in the player’s immediate vicinity and instead allowing players to travel across the world, hatching eggs in minutes rather than hours.

Additionally, players were able to reverse engineer the way the application handled the **Pokedex**, revealing all the current Pokémon and their spawn locations (and specifically which Pokémon were locked to specific regions).

Best Practices to Avoid Reverse Engineering

Use Similar Tools

The best way to secure an app against reverse engineering is to use the same tools that hackers use to attempt reverse engineering. If these tools can easily analyze the app’s string tables, control flow path, interactions with servers, cryptographic constants and ciphers, metadata, etc., the code stands compromised. Developers can also use a tool like [AppSealing](#) to detect attempts at reverse engineering in real-time.

Code Obfuscation

The obfuscation process should entail targeting specific segments of the source code, string tables, and methods that least impact code performance. The developer should ensure that the level of obfuscation they employ should not be easily reversed with de-obfuscation tools, like IDA Pro and Hopper.

Use C Languages

Consider using C and C++, which can help in runtime manipulation to a great extent. Many libraries of these two languages easily integrate with Objective C. A similar approach for Android apps will be to use its Java Native Interface. The purpose of using C and C++ libraries is to protect the runtime or reverse-engineering tools, like class-dump, class-dump-z, Cycrypt, or Frida.

M10: Extraneous Functionality

Before an app is ready for production, the development team often keeps code in it 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, i.e. it has no use for the intended user once the app is in production and is required only during the development cycle.

Extraneous Functionality Risks

In most cases, a benign code offers no extra advantage to an adversary who gains access. However, 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.

M10: Extraneous Functionality Example



An Obscure App Flaw Creates Backdoors In Millions of Smartphones

A group of researchers from the University of Michigan identified hundreds of applications in Google Play that perform an unexpected trick: By essentially turning a phone into a server, they allow the owner to connect to that phone directly from their PC, just as they would to a web site or another internet service. But dozens of these apps leave open insecure ports on those smartphones. That could allow attackers to steal data, including contacts or photos, or even to install malware.

Best Practices to Avoid Extraneous Functionality

The developer should know that automated tools cannot always detect the presence of the M10 risk. More often than not, it requires manual intervention before the app is pushed into app stores. The developer should take the following steps before the app is released:

- Ensure that no test code is present in the final build;
- Ensure that no hidden switches are present in the configuration settings;
- Logs should not contain any description of backend server processes, administrative privileges, etc.;
- In general, logs should not be descriptive at all;
- Ensure that full system logs are not exposed to apps by OEMs;
- Use ProGuard or DexGuard to stop interaction between method calls and log classes;
- Ensure that an adversary cannot set the app's debug flag to true; and
- API endpoints accessed by the app should be well documented.