M7: Insufficient Binary Protection

Threat Agents

Application Specific

Attackers who target app binaries are motivated by various reasons.

The binary could contain valuable secrets, such as commercial API keys or hardcoded cryptographic secrets that an attacker could misuse. In addition, the code in the binary could be valuable on its own, for example, because it contains critical business logic or pre-trained AI models. Some attackers might also not target the app itself but use it to explore potential weaknesses of the corresponding backend to prepare for an attack.

Besides collecting information, attackers could also manipulate app binaries to access paid features for free or to bypass other security checks. In the worst case, popular apps could be modified to contain malicious code and be distributed via third-party app stores or under a new name to exploit unsuspecting users. One common attack example is reconfiguring the payment identifiers in an app, repackaging it, and distributing it via app stores. Then, when users

download this unauthorized copy from the app store, the attacker receives the payments instead of the original provider.

Attack Vectors

Exploitability EASY

App binaries usually can be downloaded from the app stores or copied from mobile devices, so binary attacks are easy to set up.

An app binary could be subject to two types of attacks:

- Reverse engineering: The app binary is decompiled and scanned for valuable information, like secret keys, algorithms, or vulnerabilities.
- Code tampering: The app binary is manipulated, e.g., to remove license checks, circumvent paywalls or obtain other benefits as a user. Alternatively, the app can be manipulated to contain malicious code.

Security Weakness

Prevalence COMMON

Detectability EASY

All apps are vulnerable to binary attacks and many will end up the subject of some form of an attack at some time. Those apps that *have* sensitive data or algorithms hardcoded into their binary are particularly vulnerable to binary attacks. These apps should employ countermeasures to fend off potential attackers long enough so that an attacker will give up because the cost of successfully breaking the protection would be more expensive than the gain

from that success. Oftentimes, e.g., in case of copy protection, it is sufficient to prolongate the cracking process until the targeted revenue from app sales has been reached.

In general, fully compiled apps like iOS apps are less susceptible to reverse engineering and code tampering than higher-level bytecode found in Android apps (note that this might not hold for apps developed with cross-platform technologies, like PWA or Flutter).

Especially popular apps are likely to be manipulated and redistributed through app stores. Detecting and removing these manipulated copies is offered by specialized companies but is also possible with certain detection and reporting mechanisms within the apps themselves.

Note that there are no fully reliable mechanisms to prevent binary attacks. Defending against them is an arms race between the developers investing in countermeasures and attackers who break these measures. So, the question to be answered for each app is: How much effort should be put into measures against binary attacks?

Technical Impacts

Impact MODERATE

As stated before, a binary attack could either happen as reverse engineering and leak information from the app binary or as code tampering and alter how the app works.

If secrets leak, they must be replaced quickly throughout the system, which is difficult if the secrets are hardcoded in the app. Information

leakage from the binary also has the potential to reveal security vulnerabilities in the backend.

Yet, manipulation has even more impact on the technical soundness of a system. By manipulation of the binaries, attackers could change how apps work arbitrarily, for example to their own benefit or to disturb the backends, if they are insufficiently hardened against such malicious requests.

Business Impacts

Impact MODERATE

Leakage of API keys for commercial APIs or similar can cause significant costs if they are misused on a large scale. The same holds for apps that are tampered with to remove license checks or to publish their functionality with a competing app. In both cases, individuals cracking an app or stealing an API key for personal use will likely go unnoticed. However at scale, for example when API keys or even functionality is systematically used with other apps, malicious competitors might get a significant advantage because they have significantly lower costs.

The business model of the app developers may be threatened even more if intellectual property, like algorithms or AI models that have been developed with great effort, becomes public or is stolen by a malicious competitor.

Great reputational damage could arise in particular for popular apps that get redistributed with malicious code. Even though the app provider can hardly prevent redistribution of a tampered copy of its app, the negative publicity will likely be directed at the original provider. Hence, redistribution of

unauthorized copies should be made as difficult as possible for an attacker to reduce the probability of this risk.

Am I Vulnerable To 'Insufficient Binary Protection'?

All apps are vulnerable to binary attacks. Binary attacks can become particularly harmful if the app has sensitive data or algorithms hardcoded in its binary or if it is very popular. If there are additional protective measures, like obfuscation, encoding of secrets in native code (for Android) or similar, successful attacks become harder to achieve but never impossible.

Whether the app is sufficiently secure depends on the business impact that different binary attacks could have. The more motivating it is for attackers and the greater the impact would be, the more effort should be put into protection. Hence, "vulnerability" to binary attacks is highly specific to the given app.

For a quick check, developers can inspect their own app binaries using similar tools as attackers would use. There are many free or affordable tools, like MobSF, otool, apktool and Ghidra that are also quite easy to use and well documented.

How Do I Prevent 'Insufficient Binary Protection'?

For each app, it should be assessed whether any critical content is contained in the binary or whether its popularity mandates binary protection. If yes, a threat modeling analysis helps to identify the highest risks and their expected financial impact in case they

occur. For the most relevant risks, countermeasures should be taken.

Apps always run in untrusted execution environments and should only get the least necessary information they need to work, as this information is always at risk of being leaked or manipulated. But assuming that certain secrets, algorithms, security checks, and similar must be within the app's binary, different attacks can be fended off by different means:

Reverse engineering: To prevent reverse engineering, the app binary should be made incomprehensible. This is supported by many free and commercial obfuscation tools. Compiling part of apps natively (iOS and Android) or using interpreters or nested virtual machines makes reverse engineering even harder, as many decompiling tools only support one language and binary format. This kind of obfuscation is a tradeoff between the complexity of the code and robustness against reverse engineering, as many libraries that rely on certain strings or symbols in the code will not work with full obfuscation. Developers could check the quality of their obfuscation by using the tools from the previous section.

Breaking security mechanisms: Obfuscation also helps against manipulation, as an attacker must understand the control flow in order to skip security checks and like. In addition, local security checks should also be enforced by the backend. For example, required resources for a protected feature should only be downloaded if a check succeeds locally and in the backend. Finally, integrity checks could detect code tampering and render the app installation unusable, e.g., by deleting some resources. However, such an integrity check could

also be found and deactivated as any other local security check.

Redistribution (with malicious code): Integrity checks, e.g., on startup, could also detect redistribution and modification of app binaries. These violations could automatically be reported to find and remove the unauthorized copies of the app from the app stores before they become widespread. There are also specialized companies that support this use case.

Example Attack Scenarios

Scenario #1 Hardcoded API keys: Assume an app uses a commercial API where it must pay a small fee for each call. These calls would be easily paid for by the subscription fee the users pay for that app. However, the API key used for access and billing is hardcoded in the app's unprotected binary code. An attacker who wants access could reverse engineer the app with free tools and get access to the secret string. Since API access is only protected with the API key and no additional user authentication, the attacker can freely work on the API or even sell the API key. In the worst case, the API keys could be misused a lot, causing substantial financial damage to the provider of the app, or at least blocking legitimate users of the app if the API access is rate-limited.

Scenario #2 Disabling payment and license checks: A mobile game might publish its app and the first levels for free. If the users like the game, they pay for full access. All the resources for the later levels are shipped with the app. They are only protected by a license check, where the license is downloaded when the user pays. An attacker could reverse engineer the

app and try to understand how the verification of the payment happens. If the app binary is not sufficiently protected, it is easy to locate the license check and just replace it with a static success statement. The attacker can then recompile the app and play it for free or even sell it under another name in the app stores.

Scenario #3 Hardcoded AI models: Assume a medical app that features an AI to answer user requests given as speech or free text inputs needs. This app includes its specialized and quality-assured AI model in its source code to enable offline access and avoid hosting own download servers. This AI model is the most valuable asset of this app and took many person-years in development. An attacker might try to extract this model from the source code and sell it to competitors. If the app binary is insufficiently protected, the attacker could not only access the AI model, but also learn how it is used, selling this information along with the AI training parameters.

References

- OWASP
  - Tampering and Reverse Engineering (MASTG)
  - Tampering and Reverse Engineering iOS (MASTG)
  - Tampering and Reverse Engineering Android (MASTG)
  - OWASP Reverse Engineering and Code Modification Prevention Project
  - OWASP Top 10 for Large Language Models: LLM10: Model Theft
- External

- External References

Edit on GitHub

Spotlight: 7ASecurity

7ASecurity is an ISO27001 and SOC2 certified cybersecurity consultancy and OWASP Platinum Supporter specializing in manual, high-quality penetration tests and secure code audits. Trusted since 2011 by the Linux Foundation, Mozilla Foundation, Tor Project, The Guardian, and many others (7asecurity.com/publications). Every engagement is tailored to the client priorities and threat model, and guided through a dedicated channel from scoping to free fix verification. With researcher-led testing that outperforms automation, 7ASecurity consistently uncovers vulnerabilities others miss — delivering real results with proof and replication steps, clear guidance, and measurable security improvements that stand up to real-world threats.

Corporate Supporters