PROMON

A beginner’s guide to code obfuscation

TECHNICAL GUIDE

Contents

02 Introduction

03 What is code obfuscation?

03 Advantages of code obfuscation

04 Examples of code obfuscation techniques

06 Obfuscation methods

06 Minification

07 Intermediate-level obfuscation and its demise

07 Binary-level obfuscation

09 Obfuscating iOS apps

10 Obfuscating Android apps

11 Obfuscating JavaScript apps

12 Is code obfuscation alone enough to protect an app?

13 Combine code obfuscation with runtime protection

Introduction

Mobile app-based cybercrime is ever-evolving and hackers are always finding new methods to reverse engineer apps to identify weaknesses, uncover secrets, and get hold of sensitive information.

Code obfuscation has become a standard method used by developers to challenge cyber criminals from analyzing app code, thus protecting apps against intellectual property theft, loss of revenue, and reputational damage.

This beginner's guide provides an introduction to code obfuscation, along with a few examples of essential obfuscation techniques you should consider. You will also learn why it's important to implement in Android, iOS, and JavaScript-based applications, how some specific obfuscation approaches have now become obsolete, and the latest innovations in code obfuscation.

What is code obfuscation?

At its core, obfuscation describes the act of obscuring or making something harder to understand. Thus, code obfuscation is a method of modifying an app's code to make it difficult for attackers to read and understand.

Obfuscation will help conceal the logic and purpose of an app's code while keeping its functionality. Using obfuscation methods will make it more difficult for an attacker to perform reverse engineering, analyze the code, and retrieve sensitive information.

Advantages of code obfuscation

The threshold for an attacker to carry out a reverse engineer attack is significantly heightened if your app's code is obfuscated, as it will often be too costly and time-consuming for them to succeed in their attempt. By implementing effective obfuscation methods and making the app hard to reverse engineer, you will:

- **Protect intellectual property** by securing your app's unique features and functionality, so you can maintain a competitive advantage in the market
- **Make your app's functional logic and algorithms less exposed** on the client side
- **Make it more challenging for attackers to read and understand your app's unique features and functionalities**
- **Help safeguard sensitive data and key components of your app** from being easily found

Examples of code obfuscation techniques

Renaming

This technique will rename, for example, classes and method names in a randomized fashion. It is often used to obfuscate application code developed in Java, .NET, and Android platforms.

String obfuscation

String obfuscation hides strings by scrambling them in various ways. It hides and replaces strings with a nonsensical representation of the string for the human eye. The CPU will be able to de-scramble the string dynamically during execution, while an attacker would make little meaning out of the string from static analysis.

Code splitting

To make the app's execution flow harder to follow, developers can use code splitting to split large sections of code into multiple smaller components.

Code flattening

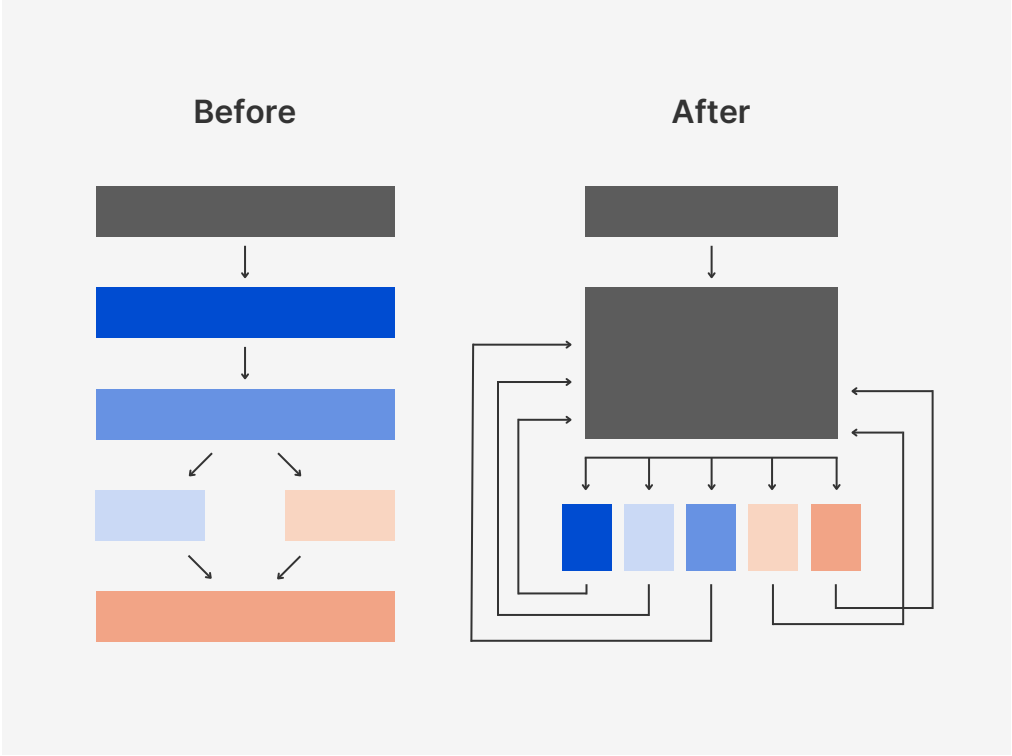
Developers can restructure the code to remove the usual modular and organized indentation, hierarchy, and inlining functions by making it look like a flat list of instructions, destroying the app's readable code logic.

Dead code insertion

This technique adds non-functional code snippets to mislead hackers from trying to understand and capture the app's logic.

Control flow obfuscation

When determining the intent of an application, it's important to understand its control flow. Control flow obfuscation is to alter the logical execution flow of the application by controlling the dynamic flow of application logic dynamic in a different and controlled manner. By complicating the control flow by, for example, control flow flattening, it takes a lot longer to understand where the code is going or why it takes a certain direction.



This diagram shows control flow flattening. By complicating the control flow, it makes it harder to understand where the code is going or why it takes a certain direction.

Obfuscation methods

Without obfuscation, your app's security code is like an open book—easy to read and alter. There are various ways to obfuscate code, each with its own advantages and disadvantages.

For example, a standard compiler like Clang has three main parts: a frontend that processes the source code, an intermediate level for initial compiling, and a backend that finalizes the code into a runnable binary format. As such, there are three approaches to code obfuscation:

1. **Source Level**
   Alter the source code before the frontend compiles it
2. **Intermediate Level**
   Manipulate the intermediate code before the backend turns it into binary
3. **Binary Level**
   Change the binary code right after compilation, before distribution

Minification (source level)

Minification is the process of removing all unnecessary characters from the source code without changing its functionality. Characters removed often include whitespace, comments, and sometimes even shortening variable names. It's common for Android Java and Kotlin code, as well as for JavaScript.

Minification is often mistaken for code obfuscation, as it does make the app smaller and the code harder to read. However, it's generally easy to reverse with commonly accessible tools.

Intermediate-level obfuscation and its demise

Low-Level Virtual Machine (LLVM) is a standout example of this intermediate approach. For app developers, LLVM bitcode has been a staple of Apple's toolchain and the Android Native Development Kit for the past seven years. It's a toolkit for building compilers that can work with any programming language and target any machine architecture. Compilers like Clang or Rustc use it to support various systems, from Linux on X86_64 to iOS. Obfuscation at this level is easier to manage since it's not tied to a specific language or architecture.

One catch: this method often depends on the toolchain.

For example, iOS and macOS apps that use intermediate-level obfuscation can be affected by updates to Apple's development software, like the new Xcode 15 version, which has effectively deprecated the ability to build bitcode apps.

Binary-level obfuscation

Binary-level obfuscators are quite a rarity. They typically must cater to multiple machine code instruction sets, e.g. ARM64, Intel X86_64, Arm V7 (32-bit), etc. They also must be able to accurately read and rewrite the various binary formats supported, e.g. ELF on Android and Mach-O on iOS/MacOS. There are many complexities in this process that somebody using LLVM simply does not have to worry about.

However, binary-level obfuscators offer some tremendous benefits:

- They are not tied to the toolchain, and so, in theory, they can protect binaries from any compiler, e.g. Swift, C/C++, Rust, Golang, Dart, Xamarin, Unity, etc.
- They are also not dependent on any development tools, meaning protection can be run entirely independently of the development process, e.g. by an IT Security function rather than developers.

- Because they are dealing with actual machine code instructions, they can achieve finer-grained obfuscation than is possible at the intermediate level and certainly way beyond what is possible at the source level.

An excellent example of such a next-generation approach is the Promon Jigsaw engine, the first cross-platform, post-compile, native obfuscation technology.

Jigsaw is toolchain agnostic, which allows developers to freely choose the tools they prefer without compromising security.

iOS

Obfuscating iOS apps

Although both Android and iOS apps are a constant target for hacking and reverse engineering, many think that iOS apps are not as exposed to such attacks as apps for Android.

Objective-C and Swift are the most common programming languages for iOS apps. Both are compiled into machine code, which makes it more challenging to translate the code back to the original source code. This has created a thought that iOS apps are hard to reverse engineer. However, the interest in analyzing machine code is nothing new, and there is a mature technology in place for reverse engineering machine code. In addition, when Objective-C and Swift are compiled to machine code, there is a lot of metadata in the binary that is required for these languages, which makes it much easier to understand the code than if you would, for example, write C code.

iOS apps are, in other words, not hard to reverse engineer and analyze, and you must take the necessary steps to prevent these threats. Moreover, the recent deprecation of LLVM bitcode in Xcode has made it even harder to protect iOS apps' code, and few vendors have viable code obfuscation tools that aren't based on bitcode.

Obfuscating Android apps

Many Android apps don't have a sufficient level of protection and often limit their obfuscation methods to code minification alone.

Android app developers have several programming languages to choose from, each with its characteristics. Java has long been a go-to language, offering robust libraries and wide community support. However, Kotlin, a modern language fully interoperable with Java, has quickly gained traction because it uses a cleaner syntax and more advanced features. For more computationally intensive tasks, developers use C/C++ and the Android NDK (Native Development Kit) to write performance-optimized native code. Additionally, for building cross-platform apps, a viable option is using the Flutter framework with its Dart language, which allows developers to write code once and deploy it on both Android and iOS.

Getting back to the Android app development process, DexCode is another key component. When developers compile their Android app, the Java or Kotlin code is first converted into Java bytecode. Next, this code is further compiled into DEX (Dalvik Executable). DexCode is a bytecode format designed specifically for Android's runtime environment.

Unprotected Android apps increase the risk of exposing your businesses to IP theft, loss of revenue, or reputation damage. Android developers should choose security software that applies advanced and multiple obfuscation techniques.

Obfuscation techniques for Android include renaming of class, function, and method names, namespace flattening, and code shuffling.

Obfuscating JavaScript apps

JavaScript is currently the most popular programming language and has the highest number of contributors and repositories, outpacing other alternative programming frameworks. And it's more important than ever to consider JavaScript obfuscation.

Developing a single hybrid app is quicker and may be more cost-effective than developing native Android and iOS individually. However, hybrid apps can be more vulnerable to attacks than apps written using native languages because JavaScript is easier to reverse engineer and modify as it is not compiled into a more abstract representation in the published app.

The main objective of obfuscating your JavaScript code is to hide the parts of the code that attackers could target by hiding things like strings, objects, and variables. Essentially, obfuscation makes it hard to analyze your code to conceal the meaning of the data.

In general, JavaScript obfuscation revolves predominantly around adding entropy (or complexity) to the JavaScript to make it more difficult to understand.

Is code obfuscation alone enough to protect an app?

Code obfuscation is a very effective method, as it will force an attacker to spend more time, effort, and resources reverse engineering an app to visualize and understand its logic—and it should be implemented as part of your security.

But while obfuscation makes your business less prone to reverse engineering and intellectual property theft, you should not solely rely upon obfuscation as it does not protect your apps and their code or features from being changed or manipulated by malware or real-world attack scenarios.

Therefore, complete code protection combined with comprehensive runtime protection is essential to protect apps fully. Choose a security product that applies advanced and strong obfuscation techniques and other protection mechanisms, such as encryption.

> “...complete code protection combined with comprehensive runtime protection is essential to protect apps fully.”

Combine code obfuscation with runtime protection

Promon offers the most comprehensive in-app protection solution on the market.

In addition to applying robust obfuscation to your Android, iOS, and JavaScript apps, Promon's range of mobile app security products will monitor your app's runtime behavior and detect if your app executes in an insecure environment, e.g. hooking frameworks, emulators, and debuggers.

They also detect the presence of code hooks, block injection of malicious code into the app, and enable your app to modify its behavior in real time to interrupt potential malware attacks.

PROMON

About Promon

Promon leads the way in proactive mobile app security. For 19 years, we've been making the world a safer place by securing any app, on any device—in no time at all. Today, we protect over 2 billion users, secure 13 billion monthly transactions, and safeguard $2.5 trillion in market cap. Promon is headquartered in Oslo, Norway, with offices in more than 15 countries around the world.

Would you like to talk to an expert?

Mobile app security is crucial to preserve and improve your business reputation. Request pricing or talk to an expert to learn more today.

Book a meeting »

promon.io

Promon AS
Cort Adelers Gate 30
0251 Oslo
Norway