ZIMPERIUM®

The practical guide to application hardening

What is application hardening?

Application hardening, also known as application shielding or in-app protection, is the process of modifying an existing application to make it more resistant to hacking attempts such as reverse-engineering, tampering, and monitoring.

Application hardening protects both the app itself and the data it uses, and is a required application security method under various regulations and standards. There are multiple application hardening techniques; companies may choose to employ all of them or a subset depending on their exposure, risk tolerance, and performance requirements.

Why it's important

As long as there are applications, there will be application vulnerabilities that can be exploited by attackers. One of the most common ways vulnerabilities are introduced is through ordinary coding errors. The industry standard sits at 15 to 50 errors per 1,000 lines of code. With modern applications containing tens of thousands to millions of lines of code, thousands of potential flaws of varying levels of severity may exist within any application.

Hackers can exploit these vulnerabilities to steal intellectual property and sensitive data, obtain cryptographic keys, or hijack the application for malicious purposes.

Application hardening shields these vulnerabilities from attack. Depending on the degree of hardening employed, it can protect partially or fully against static analysis of your source code, dynamic analysis of your application at run time, attacks that attempt to bypass application or system controls, and even code tampering.

Best practices to harden applications

As with any aspect of application development, you need to weigh risk, cost, and reward. If your application has no or low value IP, does not store or access sensitive information, does not connect with other systems, and does not use cryptographic keys, then basic application hardening methods may be enough.

If you want to protect against piracy, data theft, or code tampering, or if your application could be used as an entry point to compromise other systems, then more sophisticated application hardening is recommended.

1 Basic application protections

Renaming

This rudimentary form of code obfuscation changes identifiable variable and method names like “user_name” or “grant_access” into meaningless character strings to make them confusing to a hacker. The program execution behavior remains unchanged. Renaming is not effective, however, against deobfuscators, debuggers, or monitoring tools.

Dummy code insertion

With this approach, extraneous code is added into the application that does not affect program execution or logic, but does make reverse-engineered code somewhat more difficult to analyze. Many of the free obfuscators that perform renaming also insert dummy code.

Unused code and metadata removal

By eliminating dead code, debug information, and non-essential metadata from applications, it reduces the information an attacker can obtain and use.

2 Advanced obfuscation

Code obfuscation is a key anti-reverse engineering technique. The goal is to make the code as confusing as possible while keeping functionality the same. Unlike the simpler types of code obfuscation described previously, advanced obfuscation is more difficult to break, especially when combined with other advanced hardening techniques.

Control flow obfuscation

With control flow obfuscation, the basic structure of how subroutines are called is modified to make code more difficult to trace. It stops decompilers from reconstructing source code by removing the patterns that they look for. For example, inlining functions (replacing method calls with the actual method body), replacing calls to subroutines with computed jumps, and converting tree-like conditional constructs into flat switch statements. This latter method, called control flow flattening, uses a dispatcher to control the flow instead of calling routines directly from other routines as illustrated in Figure 1.

Objective-C message call and metadata obfuscation

In Objective-C, messages to object instances are resolved only at run time which means that message calls are stored in the binary code in plain form. Hackers can use this as an attack vector to manipulate the execution logic. Objective-C code can be protected by obfuscating plain text message calls contained within the source code so that they are not easily readable and editable. It's also important to encrypt Objective-C metadata, such as names of classes, methods, protocols, as well as method arguments and their types, to conceal this useful information from static analysis tools. The encrypted data is only decrypted at run time when the obfuscated application is loaded.

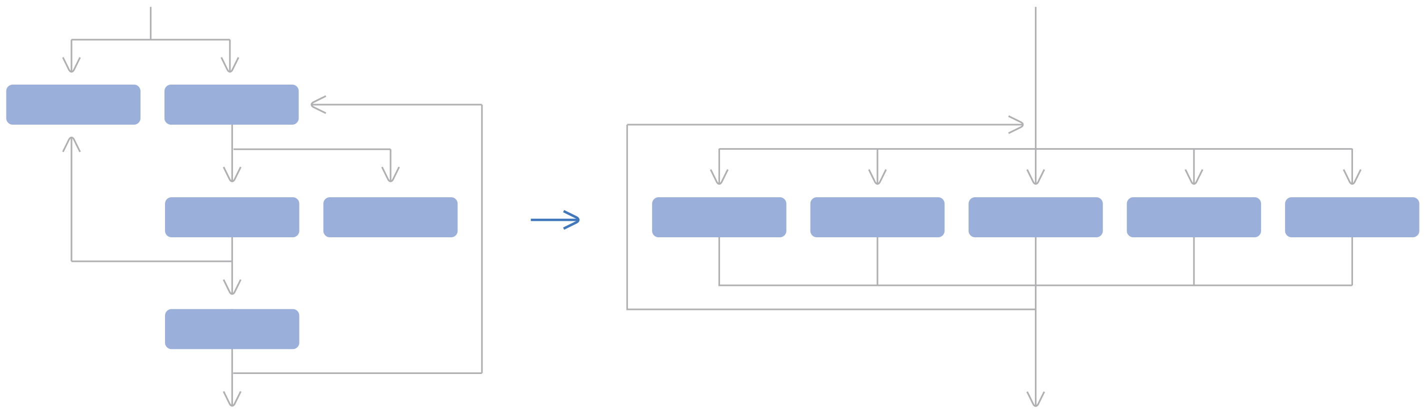


Figure 1.

Flattening the control flow to prevent reverse-engineering

3 Anti-debugging

Debuggers are used by legitimate software engineers to find coding errors but in the hands of hackers are a powerful tool to reverse engineer code. They inspect the state of an application, extracting information such as which functions are being called, the values of variables, and data contained in arbitrary memory locations. Generally, they work by setting breakpoints in the application—when that point is reached, the program stops execution and the user inspects the state at that point. Some debuggers also perform disassembly and decompilation.

A very basic anti-debugging method is to insert API calls to query process and system information to check for the existence or operation of a debugger. For example, IsDebuggerPresent, CheckRemoteDebuggerPresent, or by using debugger detaching and CloseHandle checks. These are fairly easy to add to existing application code. However, they are also fairly easy for a hacker to bypass.

One of the most effective methods to prevent debugging, called “modified code anti-debugging,” inserts code in the application that detects the presence of a debugger and takes appropriate defensive actions when present. Typically, it analyzes the application process, comparing it to the expected norm to discover any debugger-inserted breakpoints. Kernel syscalls can be used to bypass hacker-inserted user-mode hooks, which forces the attacker to modify their kernel, thereby increasing the skill and effort required for a successful attack.

4 Binary packing

Packers encrypt and compress code, adding a stub that unpacks it at run time. Hackers use this trick to hide malware from antivirus scanners as the packed code does not contain recognizable patterns to detect.

Packing compiled application code makes it difficult for hackers to reverse engineer as they can’t run the code through a disassembler or decompiler. Instead, they must capture the code after it has been decrypted into memory, and only then reverse engineer it. This essentially stops all static analysis, forcing the attacker to apply more complex and time-consuming dynamic analysis techniques.

5 Diversification

Once hackers successfully break one instance of an application, they can potentially create an automated tool to break any other instance of the application. This is known as a class break or the break once, run everywhere (BORE) problem.

Diversification alters code to create instances of the same software that are functionally identical, but where the surface of the code is uniquely different in shape and structure.

This effectively thwarts an attacker's attempts to exploit information gained from one deployment to compromise other deployments. It is much harder to develop a universal cracking scheme for software instances that are diversified. Instead, each software instance must be cracked individually.

Diversification can be performed across a population of applications, so that conceivably every application instance is different. But it is also a useful technique for diversification of versions of code. For example, if a hacker successfully reverse engineers Version 2.3 of an application, they will have to start all over again once Version 2.4 is released.

Don't forget about cryptographic key protection

A critical part of application shielding involves protecting cryptographic keys. Too often, keys are hard-coded into the application where hackers can easily extract them or are exposed in memory as they are being used in cryptographic operations. If your keys are compromised, it's like having no encryption at all.

Hardware-backed security

Hardware-based protections such as hardware security modules (HSM), trusted platform modules (TPM), and trusted execution environments (TEE) can provide strong protection for cryptographic keys but are complex to implement across device platforms and can become vulnerable if the owner has gained root privileges to the device—for example on a jailbroken phone. They also are susceptible to some types of side-channel attacks.

Keystores

Many platforms and OSes offer keystores to securely store and use cryptographic keys (Android Keystore, Java Keystore, Apple Secure Enclave, Windows Keystore). These should suffice for applications without high-value or sensitive information. However, their supported cryptographic algorithms and operations are often limited and cryptographic operations must be reimplemented on each platform.

White-box cryptography

With white-box cryptography, the keys are always hidden whether in use, in transit, or stored. Generally, white-box cryptography is not considered as safe as purpose-built security hardware and computations are slower. However, white-box software algorithms can be deployed on devices that lack hardware support and they function uniformly across platforms. Moreover, keys remain protected even if an adversary gets root access to the device. Some white-box cryptography libraries offer excellent protection against side channel attacks.

6 Tampering protections

Often attacks aim to modify your application code to hijack it for their own purposes. They may install rootkits and backdoors, disable security monitoring, subvert authentication, and inject malicious code that logs keystrokes, steals data, escalates user privileges, or performs other malicious actions. Anti-tampering protections, also referred to as runtime application self-protection (RASP), detect and prevent attacks that alter your application.

Integrity checking

Integrity checking hardens applications by inserting thousands of small, overlapping pieces of code called checkers as shown in Figure 2. During runtime, each of these checkers tests whether a particular segment of the executable has been tampered with. If any tampering has occurred, actions can be triggered to protect the application's integrity such as notifying the user, calling a custom response function, generating a log message, or even shutting down the program.

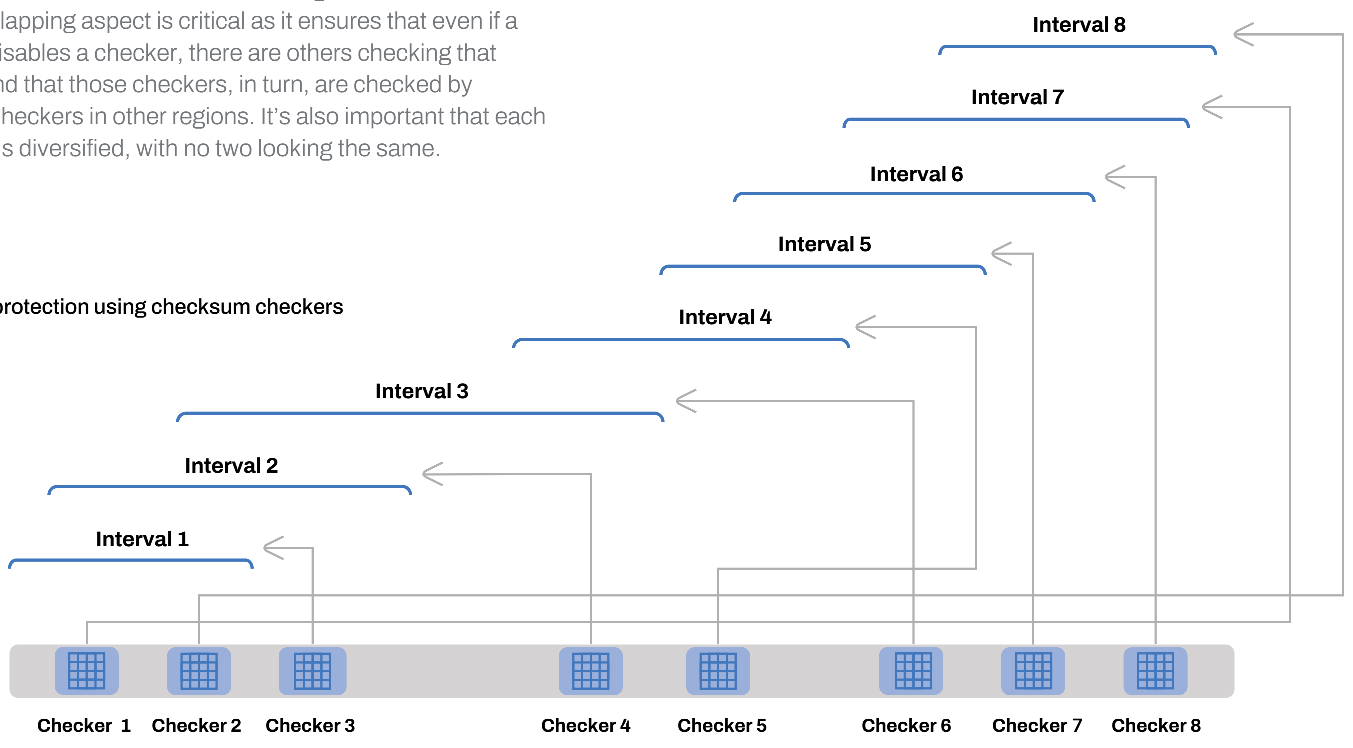
For example, in Figure 2, checker 1 is doing a checksum of interval 7. If a hacker attempts to modify checker 1, then checkers 3 and 4 will detect that a change has been made. The overlapping aspect is critical as it ensures that even if a hacker disables a checker, there are others checking that region and that those checkers, in turn, are checked by several checkers in other regions. It's also important that each checker is diversified, with no two looking the same.

Anti-method swizzling

Method swizzling is a feature in Objective-C language—commonly used on Apple platforms—that has been co-opted by hackers to attack and change the behavior of applications. Method swizzling modifies the executable by mapping a class method name to a different method implementation, changing its behavior at run time. It is legitimately used to replace or extend class methods in binaries for which the source code is not available. In the hands of an attacker, it can be used to redirect storage of credit card information to another server, extract credentials, capture customer information, or other ill intent.

To minimize method swizzling, work mostly in C++ and use ObjC classes as little as possible. Application hardening solutions, such as Zimperium's zShield, often include mechanisms to detect swizzling and execute defensive actions.

Figure 2.
Integrity protection using checksum checkers



iOS jailbreak and Android rooting detection

Jailbreaking an iOS device involves removing the software and hardware limitations established by Apple by gaining root access to the device. Similarly, rooting gives privileged operating system access on Android devices, overriding limitations established by carriers and hardware manufacturers. Once a device is jailbroken or rooted, the installed security controls are breached and a rogue app could access your application, its data, and credentials and keys. A critical part of mobile application hardening is giving your app the ability to detect a jailbroken or rooted device and take defensive actions accordingly.

Additional tampering detections

The more tampering methods you detect, the more robust your hardening. Best practices dictate that your application includes detection and response mechanisms against specific relevant attacks. Below are some commonly used protections.

Function caller verification

An executable application file contains a number of functions. Normally, there is a predefined logic how and when these functions are called at run time. However, a skilled hacker can analyze the binary code, find vulnerabilities in the execution logic, and alter the original flow of the program by calling some functions in an unexpected way, for example on Windows, by using DLL injection.

To guard against such manipulation of function calls, create a whitelist of modules (*.dll or *.exe files) that are allowed to call certain sensitive functions of the application code. Store the signatures of these authorized modules within the application binary and use at run time to verify function caller modules.

Shared library cross-checking

One attack path used by hackers is to replace or modify the shared libraries called by an application. To decrease risk from this type of attack, limit the use of shared libraries as much as possible. For the shared libraries your application does use, implement cross-checks to detect library tampering. For example, you can create cryptographic signatures for each library and perform random checks during run time to make sure they match.

Mach-O binary signature verification

All macOS, iOS, and tvOS applications distributed via the App Store are signed with Apple's private key, which prevents piracy and unauthorized distribution. However, members of Apple's Developer Program can re-sign any application with their own private key included in the development certificate, allowing the application to be run on corresponding development devices. Several services on the Internet illegally re-sign apps to distribute paid apps for free. You can insert safeguards into your application to protect against unauthorized re-signing and distribution of apps in the Mach-O file format (used by macOS, iOS, and tvOS apps).

Google Play licensing protection

Application piracy remains a primary concern for Android developers. While Android provides an anti-piracy library to verify and enforce licenses at run time, this Java-based library can be easily cracked. An effective protection against piracy is to replace the Google Play license verification library with a hardened implementation that is more difficult to reverse engineer and modify.

7 In-app defense actions

Once your application identifies a tampering attempt, it should trigger a defensive response. The simplest is to shut down the application. However, some types of attacks pose less risk and may not warrant such extreme action. Or you may want to trick the hacker into thinking they were successful so they stop attacking your app. For example, if you are protecting a game, you can secretly corrupt the game map instead of crashing the application so that it appears cracked to the hacker but actually has been transformed into an unplayable state.

Common defense actions include:

- Blocking account access
- Generating a log message to be sent to administrator
- Halting the execution of commands
- Corrupting elements of the application so a hacker believes they have been successful, but they only have minimal access
- Deletion of sensitive data
- Shutting down the application entirely

Real-time, automatic defense actions provide a clear advantage over generating alerts that need to be investigated and manually remediated. This type of complex detection and response generally requires an advanced in-app protection solution.

In-app protection using Zimperium's zShield

Zimperium's zShield injects self-defending capabilities into your application. It prevents tampering, reverse engineering and other techniques used by cyber-criminals to gain access to sensitive information and your app code. zShield uses multiple defense methods including code obfuscation, code flattening, and real-time intrusion detection, strengthening your app security in minutes.

zShield is one component of Zimperium’s Mobile Application Protection Suite (MAPS), the only unified solution that combines comprehensive in-app protection with centralized threat visibility. MAPS also includes:

zKeyBox: A state-of-the-art white-box cryptography tool that keeps secret cryptographic keys well hidden within the app code, even during runtime.

zDefend: An advanced in-app protection solution that enables mobile apps to immediately determine when a user’s device is compromised, any network attacks are occurring and even if malicious apps are installed. App developers can configure appropriate remedial actions when a given threat is detected.

zScan: A mobile app testing solution that identifies privacy, security and compliance risks in the development process, before apps are released to the public.

To learn more, visit zimperium.com.

ZIMPERIUM®

Learn more at: zimperium.com
Contact us at: 844.601.6760 | info@zimperium.com

Zimperium, Inc
4055 Valley View, Dallas, TX 75244