ZIMPERIUM

Implications of Low-Code Development on Mobile Application Security

Cars are getting safer, but we still need seatbelts.

Author: Grant Goodes

Executive Summary

Low-Code Development: A Growing Trend

Low-code development has emerged as a popular approach in software engineering. It offers rapid application development with minimal hand-coding. With smaller engineering teams, organizations can achieve satisfactory results faster, meeting the demands of today's fast-paced market.

However, the rise of low-code development also brings significant security implications. The inherently insecure nature of low-code approaches introduces vulnerabilities that cannot be overlooked. Neglecting code reviews in favour of development velocity poses a substantial risk, as the code still requires protection, despite its generative nature.

Addressing the Security Risk

To address the security risks associated with low-code development, organizations must implement injection-based protection as a first line of defense. This solution adds protections independent of the extent of generated code, ensuring security without compromising development speed. Additionally, detecting behaviour anomalies serves as a crucial last line of defense. By establishing a baseline for acceptable behaviour, organizations can identify and mitigate potential threats effectively.

Why Read Further

The white paper offers insight into low-code development's obvious benefits and its less understood challenges, which expand the application attack surface and make adopting traditional app protection techniques more difficult. It offers a security approach to help enterprise customers navigate the evolving low-code development landscape and its trade-offs to build secure, compliant, and resilient mobile applications.

Introduction

The creation of large software applications has always involved a significant amount of development cost, both in the form of skilled engineering resources, and the required investment of time to complete the normal development process all the way from architecture followed by iterative development, and finally testing and deployment. In the mobile world, these costs are exacerbated by the need to support both Apple/iOS & Google/Android, which have distinct programming languages and development environments. There have been many approaches to minimizing the development costs of mobile apps over the years, but most of them fall under the Low-Code Development category. Low-code Development brings tremendous benefits in terms of cost-savings and accelerated development timelines, but also introduces unexpected security challenges and risks.

What is Low-Code Development?

Simply put, Low-Code Development is the use of alternative programming languages or approaches to dramatically reduce the amount of code that must be written by specialized engineering resources. This, hopefully, reduces both the cost of the engineering team and the timelines to achieve a satisfactory result.

In Mobile Applications, Low-Code solutions were first offered as Hybrid Frameworks, which allowed writing applications in a single non-native language, then building and deploying them separately for iOS & Android. Using a hybrid framework effectively halved the amount of code needed, which is why they are called "low-code ".

What are the most popular frameworks?

One of the first Hybrid frameworks was Xamarin (now rebadged .NET MAUI) which attempted to take advantage of the skills of Windows desktop programmers in a Mobile world, and is based on Microsoft's **C#** Programming Language. Xamarin was followed by other frameworks based on **Javascript** (e.g. Apache/Cordova, React/Native, and Ionic), **Kotlin** (Kotlin Multi-Platform, or KMP), and **Dart** (Flutter). One common attribute of all Hybrid frameworks is the use of an interpreted programming language since this allows the use of a Virtual Machine (VM) for code execution which is ideally suited to abstracting the hardware and operating system differences between the two major mobile platforms, iOS and Android.

The Role of Open Source

The single most prevalent approach to reducing the amount of code that must be developed is the use of Open Source or other 3rd Party code. There is a long history of Open Source Software (OSS), and it has become so popular that it is estimated 97% of mobile apps use it in some form, and the average amount of open source-code in all apps is 80%. If we add in the use of commercial 3rd party SDKs, then the amount of application code actually written by application developers is even smaller. In addition, consider the abundance of online "question & answer" sites like Stack Overflow, which are hugely popular with inexperienced developers seeking answers to their programming problems, and whose code snippets are often embedded verbatim in commercial applications. The reality is that a significant percentage of application code is not written by in-house development teams, and in many cases (e.g. commercial 3rd party SDKs) there is not even source-code available for the 3rd party component that is used.

Schema-Based Approaches

Other Low-Code approaches have been tried, often involving the use of automatically generated code based on some high-level schema, hopefully reducing the amount of code that human programmers must write. Historically, Schema-based approaches (e.g. UML) have never been particularly successful, at least partly due to the requirement for some sort of Artificial Intelligence to interpret the schema and map that to real code.

Large Language Models & AI

Recently, the advent of Large Language Models (LLM) has resulted in a lot of media attention about everything from the future of education to the meaning of creativity itself. In commercial software development, however, the implications of this burgeoning technology haven't been fully explored. Enter Generative AI (GenAI), in which very little if any source-code is written by human experts, but is rather generated automatically by AI technology.

With GenAI, an LLM-based AI engine is capable of understanding a natural language description of a problem and directly generating an implementation in source-code using any programming language of choice. With GenAI, the human programmer is effectively eliminated, at least for some components of the Mobile Application which can be easily described in words (e.g. Business Logic, User Interface, etc). In a major advancement in LLM-based development, GitHub has introduced Copilot Workspace, which gives developers access to a natural language coding assistant for architecting, building, testing, and running software. Even Stack Overflow is getting into the GenAI game with their newest offering, OverflowAI, which gives direct access to ChatGPT for coding suggestions.

GenAI is not “zero-code”, as the technology is in its early days, and there remain many programming tasks for which it is just not suited, but it is definitely attracting major attention in the media and in the boardrooms of major corporations. Of course, when technology and business models intersect, there is an expectation that GenAI will result in substantial cost-savings, resulting in higher profits seemingly “for free”. Nevertheless, the software development savings come at a cost, and that cost is reduced security.

The Advantages of Low-Code Development

The advantages of Low-Code development all arise from the reduction in the amount and complexity of hand-written source-code that must be maintained.

Write-once/Deploy-everywhere

Hybrid Frameworks are particularly advantageous in this regard since they allow users to create apps for both iOS & Android using a single set of source-code (written in a single language). Using the Hybrid approach, you can then incorporate the source-code into a minimal "stub app" for each mobile platform, which then executes transparently on both platforms. From the developer's perspective, they are building a single app that works on two mobile platforms, which is excellent.

Faster Development

If GenAI is used to write any portion of the application code, then the time to develop that code is dramatically reduced, effectively to how long it takes to describe that code in a natural language specification. With Hybrid development, since only one set of source-code must be written and maintained, development time can be reduced by up to a factor of two.

Smaller & Cheaper Development Teams

Traditional Mobile Application development requires engineers skilled in older programming languages that are increasingly uncommon. For example, iOS applications are written in a mixture ofC, C++, Swift, and Objective-C, while Android applications are written primarily in Java. In some universities, these programming languages are not even taught in the Computer Science curriculum, so senior, more experienced (and therefore more expensive) programmers must be hired.

Many Hybrid Frameworks such as React/Native are based on programming in Javascript, which due to its popularity in many domains, including Website development, has a very large pool of inexpensive talent. Additionally, with the use of only a single, common programming language for both mobile platforms, the development team can be shrunk by anything up to half. Finally, only a few engineers will have to be specialized in the individual platform aspects of iOS & Android, since Hybrid Frameworks largely abstract those platform details away.

Clearly, if GenAI is used to write any portion of the application code, the size of the programming team can be shrunk, which of course is a cost-saving.

Why Does Low-Code Development Potentially Increase Security Risk?

The primary purpose of Low-Code Development is to decrease engineering costs, but what about Application Security? Since application security is not the primary focus of Low-Code development, it is only natural that it might suffer. This section describes the sorts of Security Risks that are introduced when using Low-Code Development techniques.

Inherent Risk of Interpreted Languages

This risk primarily applies to Hybrid Frameworks which almost always involve the use of Interpreted Programming Languages such as Javascript as opposed to Compiled Programming Languages such as C/C++. Simply put, Interpreted Languages can dramatically increase the attack surface of the mobile application. It does this in several ways:

Vulnerable Source-Code

Interpreted Languages are usually incorporated into the application either directly as text-based source-code (e.g. Javascript) or as a form of abstract bytecode that is trivially mapped back to source-code (e.gC#). The source-code or bytecode is usually in the form of a “blob” of code that is present as one of the application’s resources and then interpreted on the fly by the Virtual Machine (VM) it runs on. This provides the attacker with an extremely easy way to attack the security of the application by simply modifying the “blob”, which allows for application logic to be modified or removed to inject malicious code.

Compiled Languages such as C/C++ result in assembly/machine-level code in the form of binary/object code, which is very low-level and typically very difficult to map back to the original source-code. Binary code modification and injection can be performed, but they are much more challenging than source-code modification. In addition, advanced HW security features available on the two standard Mobile platforms are designed to protect binary-code from modification or exploitation, and are generally not effective for source-code/bytecode.

Vulnerabilities and Risks Related to the Abstract Virtual Machine

The other component of an Interpreted Language, beyond the source-code/bytecode, is the abstract Virtual Machine (VM) that interprets that code. The VM implements the abstract processor that the Interpreted Language is designed to execute on.

A compiled language's binary code encapsulates all its intended semantics, primarily machine instructions, directly from its source-code. This limits the attack surface to the assembly level. In contrast, interpreted languages rely on bytecode executed within a virtual machine (VM), expanding the attack surface to include both the bytecode and the VM implementation, as the VM significantly shapes the application's behaviour and semantics.

The application logic, its storage for important data, and many virtual services and utilities are all managed directly by the VM. The attacker can effectively bypass any logic or security elements in the application code by exploiting the VM code. Since the VM is typically highly focused on maximizing performance and minimizing size, it is rarely designed with these sorts of attacks in mind. Additionally, the virtual machine (VM) often includes convenience features like logging and debugging, which assist developers but can also be used by attackers for reverse-engineering and exploitation during runtime. Essentially, any tool or feature designed for developer convenience can potentially be exploited by attackers.

Weakened Application Sandbox

Both iOS and Android run application code in what is termed a "Sandbox", which is designed to limit the application to a carefully chosen subset of the powerful features of the operating system (so-called User vs. Root features) and also to prevent one application from accessing resources of other applications. The application sandbox is provided and enforced by the Operating System itself, and its security is generally pretty good, as Apple and Google have spent a lot of effort enhancing that security over the years. However, with Interpreted Languages, the VM plays the role of the Operating System for the application code, and as mentioned above, the primary design goal of the VM is maximizing performance and minimizing size, not security. Due to this, almost any VM is likely to have a less secure application sandbox enforcement, which increases the attack surface.

GenAI & Code-Reviews

An important aspect of application security is code reviews. In addition to their traditional purpose of weeding out obvious bugs and generally ensuring code quality, they are a way for security leads can ensure that the software is not full of classic source-level vulnerabilities or other fundamental security flaws (e.g. insecure cryptography, leakage of sensitive data, etc.). Normally, security code reviews are performed by senior programmers with extensive experience in writing reliable and secure code.

A Low-Code Development shop may eventually see almost all the source-code created automatically by GenAI, allowing a potentially dramatic reduction in the size of the development team. However, this results in a double whammy when it comes to performing code reviews: Not only is almost no one left to review the code, but those that remain didn't even write the code, so it can be much harder to understand it. When it comes to security, this is a recipe for disaster.

Applying Code Protection Technology is a Challenge

Various code protection technologies can be applied to mobile applications, typically using open-source or commercial tooling. The security these technologies provide comes at a cost (e.g. size and performance penalties), and that cost must be managed with care. So these technologies are typically extensively configured, as they must strike a balance between application performance, size and security. In the best case, the task of such security configuration falls to dedicated security professionals, but often in smaller development shops, the more senior/experienced application programmers will perform this configuration.

The very benefits of GenAI-generated code make configuring code protection technologies a challenge. As with code reviews, when GenAI is adopted, developers will have difficulty configuring code protection for code they didn't write and don't fully understand. Furthermore, due to the reduction in the developer team size, fewer engineers are available to perform configuration tasks. The result is almost certainly improperly configured code protection, which can result in unacceptable size, performance issues, and performance costs. In the worst case, a GenAI development shop may have no choice but to completely eliminate code protection since they just can't get it to work.

Finally, note that one of the fundamental code protection technologies, Code Obfuscation, is much less effective when applied to Interpreted Languages typically used in Hybrid frameworks. Code obfuscation is possible for languages like **Javascript,** but due to the very high-level nature of the code in interpreted languages, these techniques are easily detected and often trivial to bypass. For example, **Javascript** obfuscators can turn the entire program "blob" into what looks to the human eye like whitespace (by taking advantage of the large number of whitespace equivalents in the Unicode character-set), but as impressive as this is in a visual sense, the original program-logic is easily restored by open-source tooling.

Achieving Application Security in a Low-Code World

Application Security is best achieved by Defence In Depth: Layers of security that reinforce one another and force an attacker to operate on more than one front at once. But how do we achieve that when some of those layers (Security Code Reviews, Software Obfuscation, etc.) are either ineffective or completely impractical in the Low-Code world?

Low-code development still has two security layers that can be applied and remain effective barriers to attack:

First Line of Defence: Runtime Protection

As described above, conventional code protection tooling may be challenging or even impractical to apply with low-code development. The programming language either does not allow them or they cannot be configured and applied by the smaller and less skilled development teams supported by Low-Code Development.

However, Runtime Application Self-Protection (RASP) is commonly available in SDK form, and can thus be added to the application with trivial effort, even in a Low-Code scenario such as a Hybrid app. RASP effectively adds a set of passive and active defenses against reverse-engineering and tampering techniques that malicious actors normally apply when trying to inspect and exploit an application. For example, RASP can either detect or prevent: Attaching a debugger; examining and modifying memory; intercepting and modifying function calls (hooking), etc.

RASP verifies that a target application is running in a safe & uncompromised environment, including detecting if the operating system has been modified to enable insecure behaviour (e.g. Rooting or Jailbreaking). It serves as a security suit-of-armour for your application, and even when other security best practices are off the table (e.g. Code Reviews), RASP can serve as your first and most effective line of defense.

Last Line of Defence: Active Threat Monitoring

While RASP serves to make active reverse-engineering and exploitation of your app difficult, that does not mean your application is without vulnerabilities. Active Threat Monitoring is another protection technology that can help provide a last line of defense, even if the security design of your application is compromised. Active Threat Monitoring, like RASP, can be added to the application easily in the form of an SDK, and as the name suggests, continuously monitors the execution of your application, looking for anomalous behaviour that may be a sign that an active exploit is being attempted. Whenever the security of the runtime environment or the application is threatened, a security alert with forensics is sent to the security team. Additionally, application developers can use this security telemetry to model threats and even design dynamic on-device threat responses such as limiting the application's access to sensitive data or services, and if desired crashing the application entirely.

Conclusion

The software industry is already struggling to ensure that applications are secure against malicious exploitation and can protect sensitive user data, even when human developers write all the software. By using Low-Code development to reduce development costs, new security risks are introduced, effectively resulting in hidden security costs.

Larger Attack-Surface

Low-code development approaches result in a larger attack surface, particularly in the case of Interpreted Languages that are usually required by Hybrid Frameworks. Interpreted Languages dramatically reduce the effectiveness of standard Code Protection approaches including Software Obfuscation and RASP, exposing source-code to increased risk of analysis and modification. Additionally, they require the use of a Virtual Machine to act as the interpreter and implement some of the program semantics, adding to the attack surface.

Resource Crunch

A Low-Code Development shop may eventually see almost all the source-code created automatically by GenAI allowing a potentially dramatic reduction in the size of the development team. But this will most likely result in a double-whammy when it comes to performing code reviews, configuring application security tooling, and evaluating the security result, since the skilled resources are either absent or unfamiliar with the code since they didn't even write it.

Solving the Problem

To provide the "Sec" in SecDevOps, automated security tooling must be used more heavily. All automatically generated code should be scanned for vulnerabilities at a minimum. In addition, Runtime Application Self-Protection (RASP) and runtime Threat Monitoring must be integrated into the application code. RASP ensures that the integrity of the platform where the application runs, and the code itself, are not vulnerable to attack. Finally, Threat Monitoring ensures that undiscovered vulnerabilities can be detected as they arise, and allows the developers to make changes to the application to address them in the next release. Even when the GenAI code is not fully understood by a human developer, Security Testing, RASP, and Threat Monitoring will act as much-needed security seatbelts.

To learn more about how Zimperium can help, contact us today at

www.zimperium.com/contact-us/

Grant Goodes

About the Author

Grant is Zimperium's Innovation Architect. Prior to joining Zimperium, Grant held senior positions driving mobile application security innovations at organizations, including Guardsquare, Cloakware, Arxan Technologies (now Digital.ai), and Irdeto. In his recent role as Guardsquare's Chief Scientist, Grant was responsible for defining the future direction of their mobile application security solutions. During his tenure at Cloakware, Grant led the development of the kernel-based Android Secure Platform technology. At Arxan, he developed the company's next-gen white-box cryptography and encryption product.

ZIMPERIUM