The Security Implications of Compiled vs Interpreted Code

Convenient yes, but you are engaging in risky behaviour: Subtle-but-significant differences

Author: Grant Goodes

ZIMPERIUM®

High-level Programming Languages: Compiled vs. Interpreted

Applications are generally created as source-code (text) written in a High-level programming language which is easily understood and maintained by human programmers. The vast majority of high-level programming languages can be divided into two classes: Compiled vs. Interpreted.

Compiled Languages

Compiled languages such as **C/C++** are designed to be processed by a toolchain (compilers, linkers, etc.) which converts the human-readable source-code into sequences of low-level machine-code instructions which can then be directly executed by the CPU. The low-level representation is sometimes referred to as a Binary Executable, since the machine-code instructions can be expressed as a sequence of binary- or hexadecimal-digits.

The compilation process can be quite complex, since the semantic distance between the concepts expressed in the High-level source-code and the primitive instructions supported by typical CPUs is so large. The compiler must be intimately tied to the specific CPU architecture (e.g. size of memory words, number and size of CPU registers, available low-level operations on data, etc.). Compiled code intended for a specific CPU cannot execute correctly on a different CPU, so is not portable at all.

Historically, compiled languages were the first to arrive on the scene (immediately after Assembly languages, which are effectively just a slightly more convenient way to program directly in machine-code). High-level programming languages, since they are intended to be compiled down to machine-code instructions, often expose some aspects of the underlying HW and CPU architecture, such as requiring programmers to directly manage memory allocations, utilize “raw” memory pointers, and understand the bit-level representation of data objects such as signed- vs unsigned-integers.

Interpreted Languages

In contrast, Interpreted languages such as **Java** are either processed by a much simpler toolchain which converts the human-readable source-code into a very high-level, abstract representation, typically referred to as Byte-code, or in some cases not processed at all, remaining as textual source-code. The intention of this approach is to permit the code to run on nearly any HW regardless of the underlying CPU architecture, which provides for great portability of the application code. This portability is achieved via provision of a so-called Virtual Machine (VM) that actually interprets the source-code or bytecode.

Interpreted languages, being newer creations, often strive to abstract away challenging concepts such as memory management and representation of data objects. Almost no interpreted language even has the concept of memory pointers, which makes them easier to use for the latest generation of programmers who have less experience with the underlying HW and CPU architecture.

Blurred Lines: JIT and AOT Optimization

The advantage of the **Java** bytecode representation is compactness and portability, but the downside is that the bytecode instructions must be interpreted one-by-one, with a single bytecode usually expanding to several if not hundreds of low-level machine code instructions, not to mention overhead due to the interpreter itself. As the popularity of **Java** was initially for small applications on desktop computers, this was not at first a big problem. With the increasing use of **Java** for very large applications (such as eBay) and then Google's selection of **Java** to develop applications for (relatively low-power) Android mobile devices, it became important to address this.

The first approach was JIT (Just In Time) optimization, which effectively cached the required machine code instructions that would result from traditional interpretation of individual bytecode snippets, and if those snippets occurred frequently (e.g. inside a loop) simply re-executed those already translated sequences. This saves the processing overhead of the interpreter, and also allows for improved speedup due to improved cache-locality and other low-level CPU effects, so can dramatically improve the performance of **Java** applications.

However, JITing suffered from one major problem which was that it required a "warm up" phase to populate the JIT cache every time a new application ran for the first time, which meant that initially the application would run no faster than an un-JITed application. The solution to this was AOT (Ahead Of Time) optimization/translation which as the name suggests is performed before the application is run (either at install-time, or even at build-time). AOT identifies suitable bytecode snippets to be pre-translated and cached, avoiding the startup latency of JIT.

With JIT or AOT, the Interpreted code is arguably now compiled code, so the line between compiler and interpreter is blurred. However, the JIT'ed or AOT'ed code is still easily mapped back to the original bytecode sequences so many of the same security concerns of traditional interpreted code still remain.

The Mobile Ecosystem: Two Paths

Apple based their Mobile platform on iOS (derived from macOS which in turn has roots in the UNIX-derived BSD) and is designed to run "native" applications written in compiled languages ranging from **C/C++** to **Objective-C**, and lately **Swift**.

Conversely, Google based their Mobile platform on Android (again, a customer variant of Linux) which of course at the system level executes binary code written in **C/C++**, but applications are expected to run inside the Dalvik Virtual Machine, which supports the **Java** programming language. With Android, **Java** is considered to be the "native" language, even though applications are not typically binary executables, but rather libraries of DEX bytecode created directly from the Java source-code and designed to be interpreted by the Dalvik VM. Android applications may also include truly native, compiled shared-libraries written in **C/C++**, but the fundamental basis of an application in Android is always DEX bytecode.

Why do Enterprises choose one over the other?

Ultimately, the choice of a HLL for development comes down to a couple of important considerations, both of which are related to cost.

The first is the industry-wide reality that there are simply far more developers skilled in Interpreted Languages than there are for Compiled Languages, and that the relative scarcity of **C/C++** programmers mean that they demand higher salaries. Thus, it is far easier to staff a development team of Interpreted Language programmers.

Second, there is the very attractive possibility of covering both iOS & Android with one code-base when choosing one of the Hybrid development frameworks such as React/Native and Cordova which are almost always based on Interpreted Languages. This gives the double benefit of saving development costs on two sets of source-code while using a lower-cost development team.

Inherent Risks of Interpreted Languages

The risks described in this section apply to Android (using Java for Application development) but also 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 increase (sometimes dramatically) the Attack Surface of the mobile application. It does this in several ways:

Vulnerable and Exposed 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 high-level/abstract Bytecode that is trivially mapped back to source-code (e.g. **C#**). 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). 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 and for injection of malicious code.

Vulnerable 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. Unlike with binary code, which represents the entirety of the implementation (and thus all attacks must involve that code), with interpreted code the VM itself is part of the attack surface. 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. In fact, in many cases, the VM incorporates logging and debugging facilities which aid the developer, but also arm the attacker with reverse-engineering and exploitation tools that are freely usable at runtime.

Weakened Application Sandbox

Both iOS and modern Android run application code in what is termed a "Sandbox", which is designed to limit the application to a 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 is in the role of the Operating System to the application code, and as mentioned above, the primary design goal of the VM is maximizing performance and minimizing size, not security. Thus, almost any VM is likely to have less secure enforcement of the Application Sandbox, and this again represents an increase in the attack surface.

Software Obfuscation for the purpose of preventing reverse engineering is a venerable technique used to protect IP and to safeguard applications from exploitation, fraud, etc. Software Obfuscation may be applied at the Binary (native instruction) level, on some form of Intermediate Representation (e.g. LLVM bitcode), or on the original Source-Code itself. Which of these approaches is chosen is generally a tradeoff between convenience and sophistication of the resulting obfuscation.

Inherent Security of Compiled Languages

In this section, we will highlight some of the inherent security advantages of Compiled Languages such as **C/C++**.

Reduced Exposure of Semantics

The purpose of High-Level Languages (HLL) such as **Javascript** is to permit the human programmer to write down the application logic (also called semantics) in a relatively straight-forward manner, at least somewhat close to written natural languages such as English. The advantage of using an HLL is that the resulting code is easy to understand, even by those who are seeing it for the first time. The benefit is code that is easier to maintain, with fewer bugs, and also making it easier to modify and extend.

The source-code of compiled Languages such as **C/C++** is only an intermediate representation of the application since that source-code must first be compiled, resulting in assembly/machine-level code (binary/object-code) which is very low-level, and can execute directly on the CPU. Compared with the original source-code, the Binary code is a dramatically reduced lens through which to view the application code: Many high-level concepts such as Data Structures, Object-oriented methods, and even simple constructs such as iteration, are essentially eliminated. Contrast this with the original HLL source-code which, by design, richly expresses those semantics. We call this effect Semantic Lowering, and once the semantics have been lowered to the low-level machine-code instructions, it is effectively impossible to completely recover the original semantics of the high-level source-code, and only informed guesses may be made.

Now consider the viewpoint of an attacker, wishing to understand the application logic with the goal of searching for security vulnerabilities or even modifying application behaviour to achieve some malicious intent. If the attacker could obtain a copy of the original source-code, they would have tremendous insight into the business logic, security architecture, and other key aspects of the application's design, all of which would make finding or exploiting vulnerabilities much easier. Binary code analysis, modification and injection can be performed, but is much more challenging than performing the same actions on the HLL source-code.

Increased Difficulty of Reverse Engineering

As alluded to in the previous section, the Semantic Lower effect of Compiled Code introduces challenges to any attempts at Reverse Engineering the binary code, and thus makes identification of vulnerabilities and creation of exploits much more difficult. There are actually two reasons for this: The shortage of skilled attackers, and the relative scarcity of attack tooling.

First consider the expertise required for the attacker. The pool of experts for high-level source-code is effectively all programmers familiar with that language, so is very large. Conversely, the low-level machine-code instructions that make up binary executables are typically only understood by Assembly programmers, compiler-writers and other quite exotic sorts of engineers. Thus, the number of attackers capable of reverse engineering applications written in Compiled Languages is dramatically lower than the corresponding number of attackers for Interpreted Languages.

Second, consider the tooling and frameworks required to perform the reverse engineering. The tooling to analyze and manipulate HLL source-code are extremely common and readily available, as they are effectively the same tools needed by all programmers, typically making up the traditional IDE (Integrated Development Environment): Source code visualizers, code-structure analyzers, debuggers, etc. Contrast this with the tooling to analyze and manipulate low-level machine code. Binary analysis and debugging tools are much more specialized and require a great deal of skill to understand and use.

These two effects are essentially a double whammy, since both the pool of experts in binary-code and the number of tools to analyze and manipulate binary-code are small, and the few tools that do exist cannot be used by non-experts.

More Possibilities for Code Protection

Application Code Protection can be achieved using a number of different approaches and technologies, but it is clear that there are more and stronger options that apply only to Compiled Languages.

First, consider the common approach of Source-level Obfuscation. The purpose of Obfuscating the source-code is to reduce the ability of attackers to analyze and exploit the semantics (business logic, security architecture, etc.) exposed by the HLL code. Features such as Control-Flow Flattening, Constant Obfuscation, etc. have been well studied and are easily applied by either open-source or commercial tools. Obfuscation can be performed for both Interpreted and Compiled languages, but is relatively ineffective for Interpreted Languages such as **Javascript**. This is because the source-code (or at least the high-level bytecode) of Interpreted Languages is in the hands of the attacker, allowing them to apply open-source de-obfuscation tools and recover the original, human-readable code. With Compiled Languages, only the resulting binary (machine code) is available for inspection, and the process of compilation lowers the semantic content so much that automatic deobfuscation is very difficult or even impossible.

Second, with a Compiled Language binary, the option of direct obfuscation at binary-level exists, permitting application of very powerful and hard-to-break transformations such as the creation of non-standard machine-code: E.g. violations of standard Binary ABIs, non-standard calling-conventions and register usage, etc. Traditional binary-analysis tools and decompilers can be broken entirely but such binary-level obfuscation, making it a very powerful technique, and one that is ONLY available for Compiled Languages.

Finally, effective code-integrity solutions exist for binary-code, allowing the detection or even prevention of binary-code modifications. This stops many dynamic analysis techniques such as debugging, hooking, etc., very challenging for Compiled Language Applications. Notably, such code-integrity technology is either unavailable or not very effective for Interpreted Languages.

Recommended Best Practices

The overall message about Interpreted vs. Compiled HLL Programming Languages for development is clear: Compiled Languages involve both reduced attack surface and distinct advantages when it comes to applying Code Protection technologies. Thus, it is recommended to use a Compiled Language solution code where possible (note that for iOS this is the default). Where Interpreted solutions are required (e.g. a Hybrid development framework is being used), attempt to isolate security-sensitive code into a separate module and implement that in Compiled code (note that all Interpreted code solutions provide facilities for interfacing to native code libraries).

Regardless of your choice, here are 5 Security Best Practices to keep in mind:

1. Design with a good security architecture from the beginning
    - Especially concentrate on appropriate use of cryptography, sensitive data isolation, etc.
2. Perform Code Reviews, with Security as one of the important criteria for accepting code submissions
3. Use Code Vulnerability Scanners to detect Security problems in the resulting application
    - Not all interpreted languages can be scanned at source-level, but Binary/Runtime scanners do exist which can support even those languages
4. Always include a RASP solution, as almost all analysis and exploitation approaches rely on degraded platform integrity and code- or data-modification, which RASP is designed to prevent
5. Use a Runtime Threat Detection (SDK-based) solution to detect anomalous or concerning behaviour at runtime

But I'm on Android...

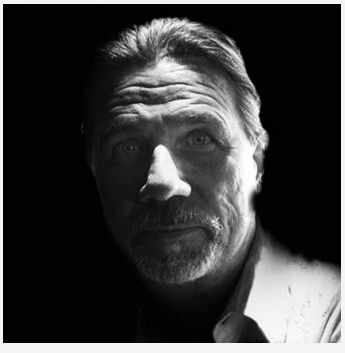
Since Android requires Java for at least some of the application code, it is impossible to completely avoid use of Interpreted code. However, Android apps can make use of pure binary shared-libraries via the Java Native Interface (JNI). Thus the security best-practice for Android apps is to move important business logic and security-related code to shared-libraries written in C/C++. It should be noted that the JNI interface is itself vulnerable to analysis and exploitation so care must be taken to protect the JNI interface, via such techniques as authenticated APIs, etc.

Conclusion: The Security vs. Convenience Tradeoff

Fundamentally, Interpreted Programming Languages have many significant advantages. Interpreted application code is more portable, easier to write, with the additional benefit that there are far more programmers available when staffing your development team. Additionally, Hybrid frameworks (e.g. React/Native, Cordova, etc.) using Interpreted languages can allow a "write once, run everywhere" approach to application development, avoiding the need for duplication of the programming effort to support both iOS and Android platforms. Thus, Interpreted languages are often preferred, but it must be recognized that lower cost and convenience almost always have security implications. As we have demonstrated, the reduced attack surface of binary-code and the superior code-protection technologies available for use at the binary-level gives a clear advantage to development in Compiled Languages. Despite the increased costs of Compiled Languages due to the relative scarcity of skilled programmers and requirement for separate code-bases to support iOS & Android, the message is clear: When it comes to security, the cost is worth it!

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.