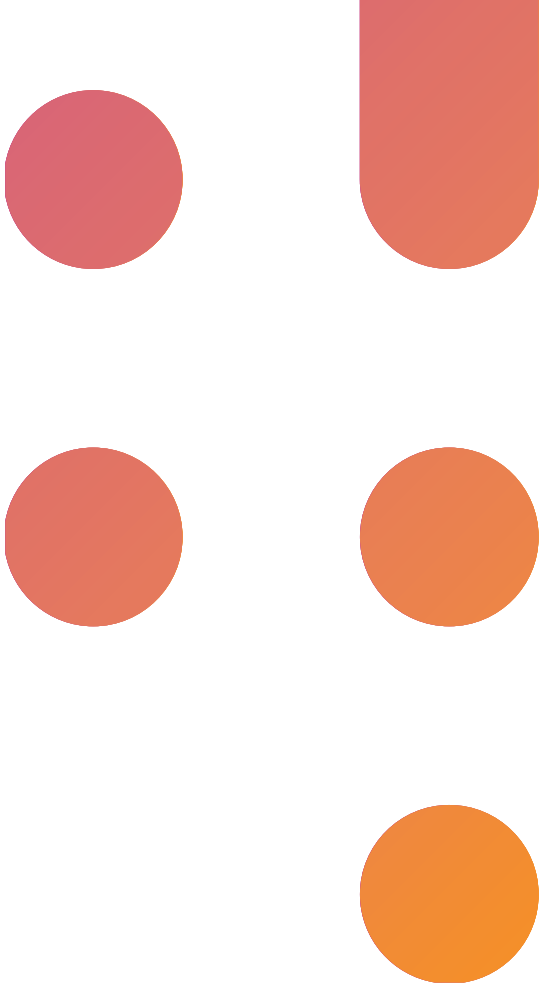


THE POWER AND RISK OF MOBILE

White paper

shield

TABLE OF CONTENTS

shield

EXECUTIVE SUMMARY

Mobile is fast becoming the preferred method for individuals to access critical online services. This increasingly means that sensitive personal information is being stored on mobile phones. Criminals are aware of the value of this data – which affects a wide range of industries and sectors. These criminals are intelligent and highlyb resourced so can exploit weaknesses in the mobile platforms, operating systems and applications.

Research suggests that half of mobile users will not take any steps to protect their devices, even when aware of the risks; and so-called operating system defenses are easily broken down. Mobile application developers need to be aware of this; and have to assume that the devices their applications are running on have been - or will be - compromised. This means that developers need to take responsibility of making their applications protect themselves.

Afterthought solutions to protect applications, such as malware and root detection, do not work. There are two reasons for this :

- Firstly, such detections work on a blacklist model - they search for known problems. This makes it an arms race and one the hackers are winning.
- Secondly, users react badly to applications that force restrictions on how they can use their phone - some users want to “root” their phone, others do not want to be made to install anti-virus software. These problems can be avoided if security is considered early in the development process.

To protect their services and their users, mobile application developers need to ensure that the following techniques have been applied to their applications to secure them against any potential leak of sensitive data :

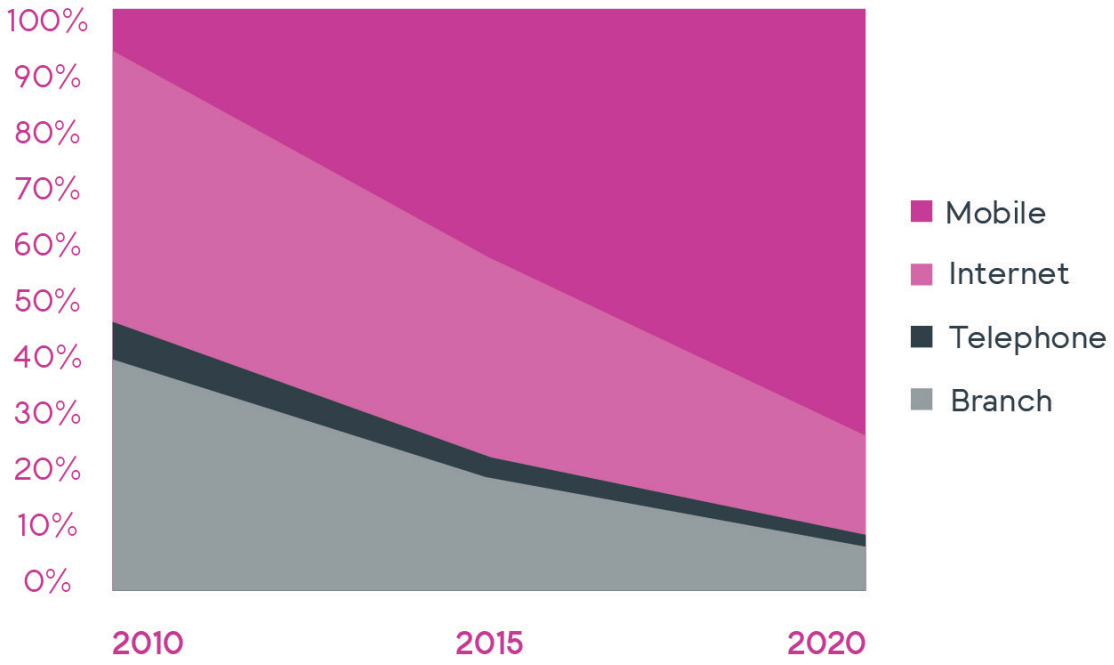
- Code Integrity Checks: Prevents any unauthorized changes to the mobile app.
- Code Obfuscation: Hides the critical and sensitive portions of a mobile app.
- WhiteBox Cryptography: Enables secure encryption and data storage on any platform.
- Processor Native Code: Provides the solid foundations to build the protection layers.

Inside Secure’s MatrixSSE combines these techniques into a comprehensive package that has been deployed in more than 400 million mobile applications to secure financial, entertainment and gaming services. These applications have successfully gone through extensive penetration and attack testing by external security labs.

INTRODUCTION

The Power and Risk of Mobile

Mobile is fast becoming the preferred method for individuals to access online services. Increasingly, this means that sensitive information is being stored on mobile phones. The change is highlighted by looking at the banking industry where over the last five years mobile banking has moved from being only a small percentage of total interactions to being the dominant channel (figure 1).



Criminals are fast becoming aware of the value of data stored on mobile devices - not just financial data but across a wide range of industry and sectors. These criminals are intelligent and highly resourced so can exploit weaknesses in mobile operating system and application security.

It is not just the data held within mobile applications that is valuable to criminals; mobile applications provide a path straight through perimeter defenses of IT systems. This means that a weak mobile application will be the weak point in IT security systems. If an attacker can control a mobile application, they can use it make apparently legitimate requests of the supporting IT systems – comprising not just the mobile application but also the whole IT system.

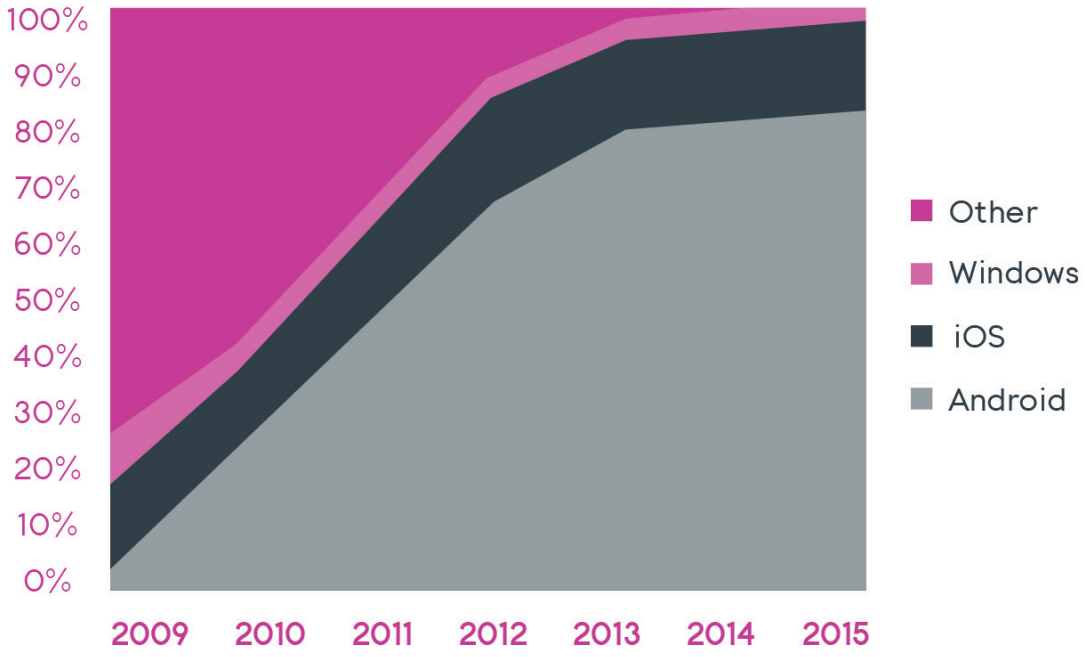
Research suggests that half of mobile users will not take any steps to protect their

devices2; and so-called operating system defenses are easily broken down. This means that mobile application developers need to assume that the devices their applications are running on have been - or will be - compromised. It therefore falls on the developers to take responsibility of making their applications protect themselves.

This paper will help developers, product managers and risk professionals to understand the steps required to secure mobile applications.

Growing Dominance of Android

Between 2009 and 2015, one mobile operating system enjoyed explosive rates of growth (see figure 2). Over those six years, Android has gone from being a minor player with less than 10% market share; to being by far the most widely used operating system with 80% market share globally.



During this time, Android's main competitor - Apple's iOS - has kept a fairly constant market share of around 15%. While it is certainly true that Android's figures are being fueled by heavy adoption rates in emerging markets, where the price-point gives it huge appeal; this does not take away from its impressive levels of growth, especially given the drops seen for Windows Phone, Symbian and BlackBerry. At a global level, Android is now the dominant OS, and by quite some distance.

Every software platform presents its own set of security challenges. One of the challenges when it comes to protecting Android applications is that the standard development language on the platform is Java. While Java has many advantages in terms of development speed and flexibility, it is an inherently insecure language to

develop applications in. It presents a serious challenge to anyone looking to secure their application.

BEST PRACTICES TO DEVELOP SECURE MOBILE APPLICATIONS

Security can not be an Afterthought

Mobile application developers should build self-defending programs so that deployed instances can protect themselves from hackers, pirates, targeted malware, insider betrayal and even hardware errors. The best way to achieve this is to consider security at the start of any development rather than try to add it at the end.

A recent study by Ponemon Institute4 on mobile applications revealed that 77% of organizations cite "rush to release" pressures as a primary reason why their mobile applications contain vulnerable code. The trouble with this approach is that they are trying to add security at the end of the development. How sensitive data is protected needs to be an architecture question; it is not something that can be successfully applied at release time.

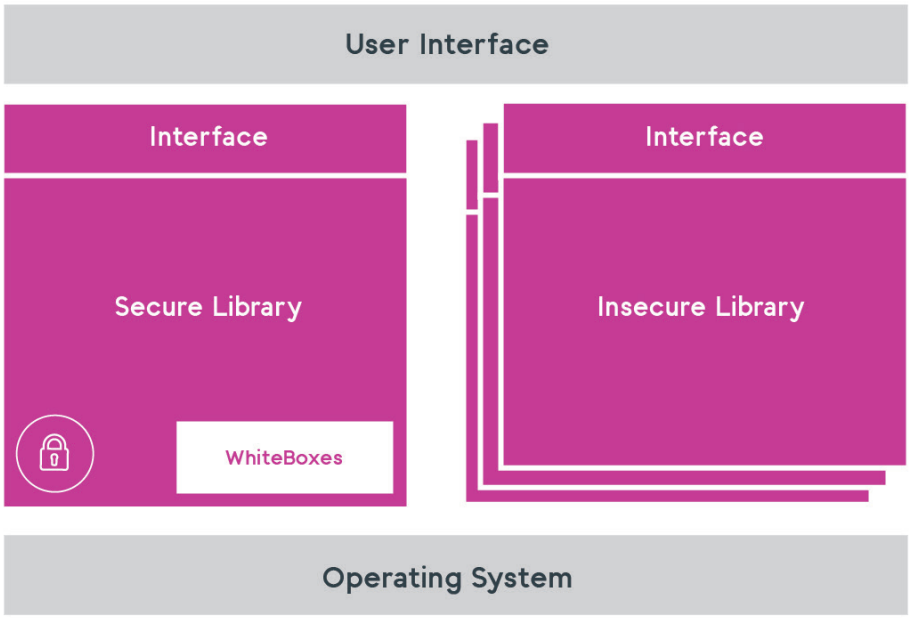
Afterthought solutions such as malware and root detection do not work. There are two reasons for this. Firstly, such detections work on a blacklist model - they search for known problems - this makes it an arms race and one the hackers are winning. Secondly, users react badly to applications that force restrictions on how they can use their phone - some users want to "root" their phone, others do not want to have anti-virus software forced upon them.

These problems can be avoided if security is considered early in the development process.

Typical Architecture of a Secure Mobile Application

Mobile applications are built out of multiple components (often called libraries). When architecting an application, it is important to identify which of those components are "secure" and which are "insecure". Sensitive data should never leave the secure components in an unencrypted form.

The secure components can contain code and data that is not necessarily sensitive - it just happens to have been secured. In fact, it is important to put the majority of the business logic into the secure components. This stops a hacker from lifting the secured components and replacing them with their own implementations - thus circumventing the protections.



Prefer a fully static language (C, C++)

Static languages (that compile to processor native code) implement a rigid program that does not expect changes at runtime. This foundation permits tamper-resistance as a natural extension. Partially and fully dynamic languages make a poor foundation for tamper-resistance because they permit changes to application behavior through standard language features, making attacks difficult to differentiate from legitimate runtime operations.

Popular languages and environments in order of preference as a basis for tamper-resistance :

- C and C++ are fully static, and optimal for tamper-resistance.
- Objective-C is risky because while it does permit the same static functionality as C, any Objective-C methods and types may be legitimately modified by an external component. Inside Secure has technology to detect these intrusions but the detection strategy rests upon good tamper-resistance as a foundation and therefore the static portion of Objective-C.
- Swift is considered risky because on iOS devices, it is usually compiled to an intermediate code called bitcode. Bitcode is then converted to native machine

code on Apple's AppStore. This means that, like Java, it is difficult to know the resultant execution code at development time.

- C# should also be considered risky as it is typically used to develop software that runs in the .NET environment.
- Java and .NET environments are the least capable of supporting tamper-resistance, having no static form and no reflective visibility of its own material. Applications that run in these environments can neither be obfuscated nor secured with a high degree of confidence without a custom, obfuscated runtime (effectively replacing the virtual machine on the client). It is therefore important to minimize the amount of Java involved in any secure application - at least in the areas where sensitive data may flow.
- HTML5 and its associated technologies (such as JavaScript) are impossible to secure. The delivered code is more-or-less plain text that can be easily read and modified. While it is possible encrypt the code, it has to be decrypted prior to execution and so is easily intercepted.

On some platforms (e.g. Android) there will be little option but to use Java for certain things - particularly User Interfaces (UIs) and some other operating system services. It is important to ensure that these layers are carefully separated from anything dealing with secrets and sensitive data. Java components that must operate with secrets must be designed to operate on encrypted secrets only. Never in the clear, ever.

In most cases this will mean Java can only transport secrets around and not operate on their values. If at any point this 'law' is breached in the application architecture, this is where the focus of an attack will be - it is the weak link.

Code Integrity and Protection

Anti-Tampering

Anti-Tampering protects applications against hackers who have unlimited access to deployed versions of the program. Unlimited access is not difficult for a hacker to achieve as they can simply lift the application from a compromised device. Once a hacker has unlimited access to the application they can perform a wide range of attacks on the application to understand its operation and weaknesses. On their own, these attacks may be of limited risk to the application provider but if they reveal how to perform a mass attack across the installed user base, then the risk could be substantial.

To provide maximum protection but limited performance impact, an anti-tampering solution should intelligently analyze your code both statically and dynamically (by

watching it in operation), then automatically inject an optimized network of various software "checks" into a copy of your source, which is then compiled and built as normal. This network of checks forms a nervous system that reacts when the software is tampered with. Developers should never have to interact with the nervous system; it should all be handled automatically as part of the build process.

Intelligent analysis and optimization enables the system to distribute hundreds and thousands of checks throughout the code, checking the program and each other, while typically having a negligible impact on performance. These checks are resistant to detection and automated removal techniques. If any change is made to the executable, multiple checks detect the change and respond.

Anti-tamper technology performs tamper-detection at the granularity of functions. Too few functions mean reduced security. It does not matter that the functions are related to security, it only matters that they are actively used. The more weight the code being anti-tampered has, the more effective the anti-tampering will be.

Code Obfuscation

Obfuscation on its own is not application protection; but it can be a useful layer to have as part of wider defenses. Application developers should hide sensitive data in software and obfuscate sensitive code.

With the move from the web towards native mobile applications, very significant quantities of sensitive logic and key Intellectual Property (IP) has shifted from backend servers behind corporate firewalls to millions of mobile devices. At the same time many new features deployed on such devices contain commercially sensitive algorithms, security-sensitive features that seek to fingerprint or authenticate devices and users, or those that decrypt streams of sensitive content are now included in applications.

Within applications, these sorts of routines can be routinely examined using a wide array of freely available debugging and hacking tools, creating significant risk of hacking and IP theft.

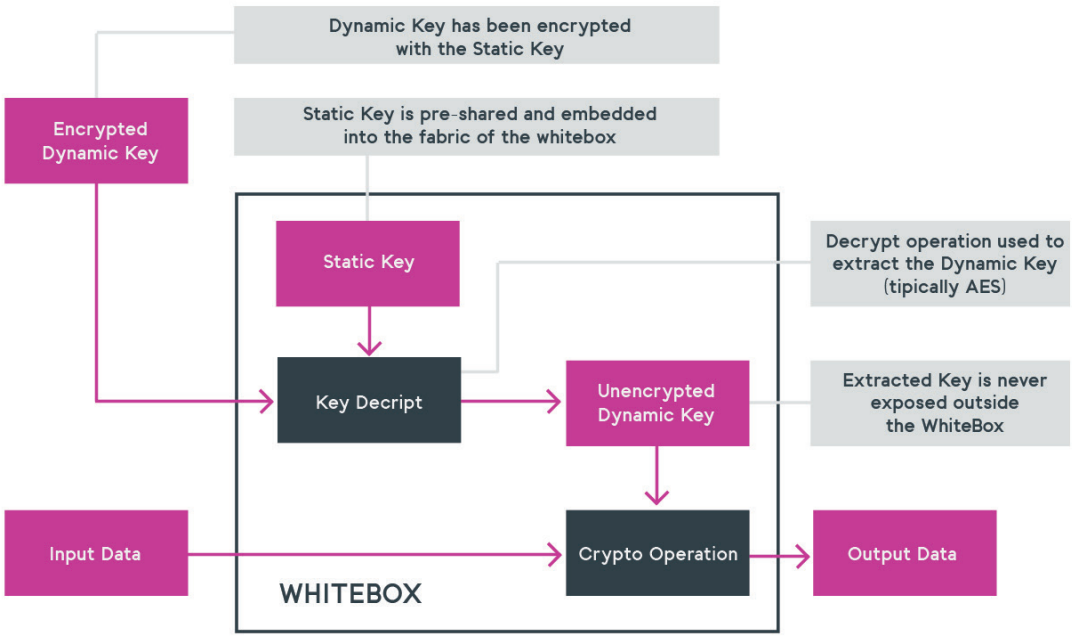
Powerful obfuscation dramatically increases the complexity of reverse engineering an application's sensitive functions, significantly hampering attempts to statically or dynamically analyze their operation, making analysis impractical for all but the most skilled attacker and ensuring that even elite hackers will move on to softer, less frustrating targets.

The obfuscation technology used should allow flexibility to target specific functions and provide the ability for complexity to be tuned to maximize efficacy while meeting performance and code size requirements. Obfuscation techniques that can be employed include control flow obfuscation, symbol name obfuscation, loop flattening, code mutation, decoy code, decoy functions, and literal substitution. Opaque predicates and expressions ensure that the code must be dynamically analyzed if any attempt at understanding is to be made.

Obfuscation should be applied to the code blocks, functions and key data before compilation to ensure, at a bare minimum, there are multiple barriers to compromise.

WhiteBox Cryptography

For critical secret processing, WhiteBox technology should be employed to dissolve secrets into the code and to obscure algorithms.



WhiteBox solutions come in two models: those that provide a pre-created WhiteBox and those that provide the tools for developers to create their own WhiteBoxes. The former has advantages that the effort to create the WhiteBox is pushed to the solution provider. The latter gives a great degree of flexibility and freedom; any algorithm (not just the crypto the solution provider supports) could be WhiteBoxed and algorithms can be chained together to increase the security of an operation.

The other point to remember is that whoever creates the WhiteBox controls the cryptographic keys that unlock the WhiteBox. If the application developer creates the WhiteBox then they are the sole entity to have those keys. If the solution provider creates the WhiteBox, then they have the keys.

As with obfuscation, WhiteBox technology is only one layer in the defenses and to be truly effective it needs to be used with a powerful anti-tamper solution. The highest protection comes when the anti-tamper solution can fully integrate the WhiteBox with

the rest of the application. To fully anchor the WhiteBox into a secure component, it is important to have the WhiteBox's source code so it can be compiled along with the rest of the secure component.

It is possible to use a WhiteBox from Java (or another insecure component), but it is strongly discouraged. Even though the WhiteBox can be designed to ensure secrets are encrypted before they leave the WhiteBox and decrypted as they enter; allowing direct calls to the WhiteBox from an insecure component will place the WhiteBox at significant risk of 'lifting', and subsequent illicit control (performing work for Application and Hacker alike). A WhiteBox is only as strong as it is 'anchored' to a secure component that bears tamper-resistance. This is a very important point and must be considered in design of any mixed-language application intending to process secrets in software.

Data Storage

Sensitive information must be protected in use, at rest and in transit. "In use" is covered by the techniques discussed above. "In transit" requires secure communication protocols, the mobile end-point of which is a secure component.

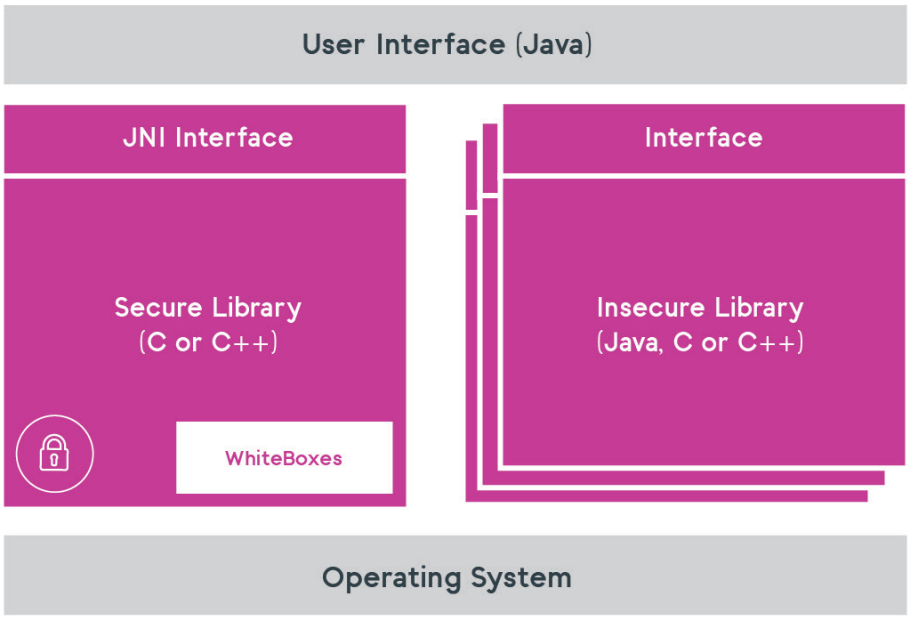
When it comes to "at rest", it is important to design the storage of an application in such a way that the critical information (e.g. passwords) do not reside directly on a device. If they do, they must be stored securely. They should reside within encrypted storage in the internal app data directory, and the app should be marked to disallow any backups to cloud services.

SECURITY CHALLENGES IN ANDROID APPLICATIONS

Architecture of Secure Android Application

A recent study5 found more than 97% of the paid-for Android applications were vulnerable with many security issues. This is in part down to a lack of consideration given to security issues and in part down to the Java environment that Android encourages developers to work in.

The generic architecture of a secure mobile application described above also applies to Android (figure 5). It is acceptable to develop the insecure components in Java; but the secure ones need to be developed in C++ to allow strong anti-tamper to be applied to them. Remember, the more "weight" of the application in the secure components, the better.



Security Considerations

If security and protection of the application has not been properly considered, the lack of binary protection in Android applications exposes the application to a large variety of technical and business risks.

To fully consider the security implementations, it is important to understand the vulnerabilities that come from a lack of binary protection. The main weaknesses can be attributed to three aspects :

- Application software that is deployed into an untrusted environment will often turn into prey for attackers through reverse engineering, modification, analysis and exploitation.
- Application software that is easy to access will allow attackers to gain entry into the application binary in ways that allow compromising acts to proceed very easily through damaging its integrity and technique.
- Application software that is easy to modify through binary compromise can result in many negative impacts to the application, its provider and its user.

Given that a lack of binary protection is a highly dangerous position for application developers to be in and greatly increases the opportunity for a mass-attack against

all instances of an application, the application developer needs to take responsibility to protect their applications.

The best practices to adopt are :

- The application should implement but not solely rely on :
  - Root detection to assess whether the operating system's defenses are still in place.
  - Debugger detection to assess whether someone is spying on the application. Note that legacy techniques like root and debugger detection should only be considered an indicator as they can be easily fooled.
  - Its own implementation of secure services (e.g. a SSL client) as operating system services can easily be replaced by compromised implementations.
  - Certificate pinning to provide mutual authentication when communicating over secure channels.
- There should be protection in the application against adversarial reverse engineering.
- The sensitive parts of the application should have prevention against unauthorized code modification.
- There should be a strategy and procedure to detect code modification at runtime.

Avoid Java Code

Given the dominance of Android, the majority of mobile applications are written in Java. The problems that Java, and other managed code languages, present to secure application development are significant.

The Anti-Tamper Challenge

Successful application security must rely upon layers, with anti-tampering of application code the most effective foundation to build the other layers.

A crucial part of anti-tampering is code integrity checking, which verifies that genuine application code is being executed and that a hacker has not modified it. This can be achieved with compiled languages like C and C++ because the code that executes is identical to the code that was originally compiled.

This is difficult with "runtime compiled" languages, like Java, because the code shipped is in bytecode form. This bytecode is not executed directly when the application runs. When an application runs, the language runtime compiles the bytecode into machine code and executes that. Given that there is no way to distinguish this intended modification from malicious tampering, it is impossible to deliver strong runtime

antitamper protection of Java code.

The best that can be achieved with languages like Java is to verify the integrity of the bytecode, but this will not detect any changes to the compiled code that the runtime is actually executing. A hacker can easily intercept the compiled code and modify that instead, thus evading the bytecode verification.

"Integrity protection makes it difficult to modify the WhiteBoxed native library code but it is comparatively easy for an attacker to patch the Java bytecode or hook/intercept the JNI calls and bypass anti-rooting"

- A Leading Security Lab.

It is also possible to modify the language runtime to effectively neutralize any bytecode verification methods, although this is less common because modifying the runtime-compiled code is so simple.

The bottom line here is that Java applications are easy for hackers to modify, and if they can modify the code, developers cannot expect the application to be secure for very long.

A Formidable Hacking Toolkit

As well as evading anti-tampering, a hacker can modify the language runtime to transform all of the code in a Java application, right down to the instruction level.

These transformations can be used for a wide range of purposes, including :

- Making large-scale behavior changes to the application;
- Injecting runtime analysis mechanisms into every part of the code;
- And even adding features to the application.

The danger is that a hacker can perform these transformations anywhere from the bytecode through to the compiled machine code. Modifying the bytecode is very simple and powerful enough for most tasks (and the hacker's additions are helpfully optimized by the runtime), whereas modifying the machine code allows the hacker to do almost anything.

The real issue this causes from a security perspective is that it allows hackers to implement extremely efficient emulation-level analysis tools, which they can use against any security measures the application contains. This significantly raises the bar on the security measures that the developer needs to employ, if they are to remain effective - defenses essentially have nowhere to hide from this type of attack.

Method Swizzling

The Objective-C method swizzling attack takes advantage of Objective-C's dynamic typing to hook method calls, allowing a hacker to control the behavior of an application at the method level. This approach has obvious limitations but hackers have used it successfully for a number of purposes, the most well known being to disable jailbreak detection code in iOS applications.

Although Java is statically-typed, the loose relationship between an application's bytecode and the code that actually executes makes it possible to implement a similar attack to Objective-C's method swizzling; not to mention even more powerful variants There are at least two frameworks already available for Android that provide this functionality, the most popular one being the Xposed framework.

The main issue with this type of attack is that it allows less-skilled hackers to make significant changes to an application's behavior with very little effort. The most obvious attack (on Android) would be to disable root detection code - the detection code itself might be native code but if it has to be called from Java code, basic hacking skills are often enough to disable the call.

CONCLUSION

Mobile applications are already central in most people's life i.e., the primary method for accessing banking, healthcare, entertainment and increasingly for payment services. As a result, sensitive data and functionality are increasingly exposed to theft and subversion. We have seen from the Ponemon Institute study how organizations "rush to release", putting their users and consumers sensitive data vulnerable for attacks. This attitude needs to be changed, as security of mobile applications should be weighed equally, if not more, than the data center itself.

To protect their employers and users, mobile application developers need to ensure that the following techniques have been applied to their applications to secure them from any potential leak of sensitive data :

- Code Integrity Checks: Prevents any unauthorized changes to the mobile app.

- Code Obfuscation: Hiding the critical and sensitive portions of a mobile app.
- WhiteBox Cryptography: Enables secure encryption and data storage on any platform.
- Processor Native Code: Provides a solid foundation from which to build the security layers.

Inside Secure's MatrixSSE combines these techniques into a comprehensive package that has been deployed in more than 400 million mobile applications to secure financial, entertainment and gaming services. These applications have successfully gone through extensive penetration and attack testing by external security labs.

MatrixSSE allows mobile applications to securely process and store sensitive data in a hostile mobile environment. It simplifies the integration of security into mobile applications and allows them to defend against malicious attacks. Application providers in financial, banking, retail, healthcare and enterprise industries can benefit from the MatrixSSE security solution.

For further details about MatrixSSE, please visit

http://www.insidesecure.com/Markets-solutions/Payment-and-Mobile-Banking/MatrixSSE2

ABOUT INSIDE SECURE

Inside Secure provides comprehensive embedded security solutions. World-leading companies rely on Inside Secure's mobile security and secure transaction offerings to protect critical assets including connected devices, content, services, identity and transactions. Unmatched security expertise combined with a comprehensive range of IP, semiconductors, software and associated services gives Inside Secure customers a single source for advanced solutions and superior investment protection. For more information, visit www.insidesecure.com.