Android Insider Attacks

Extended Abstract

David Goeth
University Passau
Passau, Germany
goeth@fim.uni-passau.de

ABSTRACT

Android apps can execute native code by a shared library or an executable. The native code is executed in the same memory address space as the Java byte code. The communication between Java and the native code is done by the Java Native Interface (JNI), which allows the native code to read and write into the process memory address space and thus changing the Java code, as well. Malicious code potentially can use this to abuse vulnerabilities of an benign app.
In this paper, we will look in depth at a possible scenario for attacking a benign app from malicious native code and analyze the reasons making this attack possible.

KEYWORDS

Android Native Code, JNI, Code patching, Trampoline Hooks

ACM Reference Format:
David Goeth. 2017. Android Insider Attacks: Extended Abstract. In *Proceedings of Advanced seminar: Real Life Security*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Android allows developers to write native code that is able to interact with the Java code. This can be accomplished with the Native Development Kit (NDK). In the documentation of the NDK it is stated, that the NDK can be useful in the following cases [3]

- Squeeze extra performance out of a device to achieve low latency or run computationally intensive applications, such as games or physics simulations.
- Reuse your own or other developers' C or C++ libraries.

In [6] it was showed, that the use native code can speed up Java applications considerable.

As stated in [3] , the communication between Java and the native code is done via Java Native Interfaces(JNI) [1]. JNI is a native programing interface, so that Java code running in the JVM can interoperate with applications and libraries written in other programming languages [8].

JNI wasn't designed with security in mind and doesn't do any security checks. This fact can be derived from the design specification document from JNI (section 'Reporting Programming Errors')[7]. The native code has direct access to the memory of the running process and as the Java and native code are running in the same memory address space, the native code is able to read and write arbitrary memory from the JVM [1, p. 2].

It can be registered, that JNI is a powerful tool that doesn't restrict the developer. On the one side this speeds up execution of native code and is convenient for developers, but on the other side it is also very dangerous seen from a security perspective. If an attacker is able to execute code in a native library the executed code can abuse the power of JNI, too.

A selection of possible attacks exploiting JNI are:

- As native libraries reside in the same address space as a JVM, bugs in native libraries can enable attackers to read and write the JVM's memory [12, p. 3].
- Similar to the Reflection API, JNI allows to retrieve and set the content of private fields. This enables an attacker to steal confidential information [12, p. 3].
- JNI allows to set the destination of an object pointer without type-checking. Thus type-confusion attacks are possible [12, p. 4].
- OS system calls can be invoked that may violate the security policy that the JVM imposes on a Java application [12, p. 4].
- Changing the behavior of the Java application by replacing functions by manipulating or injecting byte

Advanced seminar: Real Life Security, WS 2017/2018, Passau, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html

code using code patching methods. We will present later in this paper an example for this kind of attack.

It should be highlighted that quite a few Android apps use native code: In a survey of Google Play apps [2, p. 1] , it was found out, that 86% of the top 50 apps have been shipped with native code. It was also stated, that there were also apps (like Skype), that use native code for a common core being shared among multiple platforms. If an app would be compromised through a flaw in the native code, possible thousands or even millions of systems would be affected, thus these apps could be an interesting target for attackers.

In this paper we will analyze how the security vulnerabilities exposed by JNI can be abused by an attacker, if he/she is able to execute code in a native library, and how the attacker can compromise the benign target app using native code.

The following sections of this paper are structured as follows: In Section 2 we will look at other works dealing with security concerns of native code, in Section 3 we will discuss techniques and methods we will need for Section 4, where finally the aforementioned attack scenario is presented.

2 RELATED WORK

Just a few years ago, the security threats coming from native libraries weren't paid that much attention since it was mistakenly believed that native code would be as secure as Java code since Android regulates all components above the kernel [13, p. 1].
But in recent years, more effort was done in finding ways to make native library calls more secure.

In [1] the authors analyzed dynamically around of 446 thousand Android apps that use native code and developed a Native-Code policy that limits many malicious behaviours while still allowing the correct execution of 99.77% of the benign apps.

The authors from [13] present a security framework called NativeGuard, that isolates and executes code from native libraries in a separated non-privileged application. The communication with the source app is done via interprocess communication. Through NativeGuard, native libraries don't inherit privileges from the Android app and they cannot access the memory from it anymore. Also a well known software sandboxing project is Robusta [11].

Not explicitly designed for Android, but nevertheless an interesting project is CHERI JNI, that is a hardware-assisted implementation of JNI extending the guarantees required for Java's security model to native code [2].

In [10] the security implications of loading additional code from external sources at runtime have been analyzed. Attackers can abuse the code loading to hide malicious code from security services like Google Bouncer. But benign applications can also have vulnerabilities, if the code loading is done insecurely. It was found out, that Android doesn't enforce enough security checks on external code and developers of benign apps often don't implement right appropriate protection mechanisms or are unwitting of the threats. The authors have done a large-scale analysis of popular Google Play apps and have laid bare that 9.25% of them are vulnerable to code injection attacks. As a sample attack they did a HTTP Man-In-The-Middle attack on a benign application.

With the results of the analysis they modified the Dalvik VM in such a way that attacks through external code loading can be blocked. They did this by adding missing security checks for external sources.

3 BACKGROUND

In this section we want to examine how an attacker could manage it so that malicious (native) code gets executed. Of course the bandwidth of possible attacks is much too large, so we want pick out some interesting attacks shining out.

A general attack canning be done would a HTTP Man-In-The-Middle (MITM) attack how the authors from [10, p.9] did it: In general it is the exploitation of insecure communication channels. The stated app has implemented an update process where the update is downloaded from a remote HTTP server. The update is then loaded through a Classloader at runtime. But by using the insecure HTTP connection instead of HTTPS, the application is vulnerable to a MITM attack, through which again an attacker could execute arbitrary code that is allegedly considered to be an update.

Most of the apps out there are free apps, as shown in [15, p. 164], that 15% of all Android apps are paid apps, however they only account for 0.2% of the total app installs. Thus, it is not surprising that the developers of free apps like to use advertising libraries: A survey of 2012 showed that 49% of all apps and 50% of the top free apps used libraries for monetarized advertising [9, p.7]. Advertising libraries have a large user base and thus security plays even a major role. If an advertising library gets compromised through a malicious insider or a flaw in the native code, this could guide to fatal consequences. An actually well intentioned C code could lead through memory safety vulnerabilities to arbitrary code injection as elaborated in [14].

4 ATTACK SCENARIO

In this section we will present and discuss a possible attack from native code, that leverages the power of memory insecurity in C++ to compromise an Android Java application. The content of this section is structured as follows: In Subsection ***Model*** we present the theoretical concept of the attack,

some design decisions and the demo application. In Subsection ***Implementation*** we go into more implementation specific details and validate our former created attack model.

4.1 Model

In our scenario we have a benign application that uses native code. The reason for the use of native code are secondary. It could be used for advertising, 3D graphics acceleration, high performance computations, but for our experiment it doesn't really matter, since we use the native library primarily as the entrance point for our attack.

The app connects itself to a remote server and establishes an encrypted communication channel using asymmetric cryptography.

Our goal is to act on the attacker's view: The goal is to monitor the communication between the client (benign app) and the remote server. Cracking the asymmetric encryption directly isn't an option. We rather want intercept function calls that handle the sending and receiving of messages.

The key component for our attack scenario here is *(function) hooking*. By changing the machine code in certain ways (e.g. jumps to another address) it is possible to change the execution flow. In this way it is possible to execute before a function is called or after it has been called. But one could also only call the new created code skipping and thus replacing the original function completely.

It is possible to copy the code of the original function and store it on a different (new) memory location. Through that it is possible to execute custom code instead of the original function, but if wished the code of the original function can still be accessed (and be executed).

This technique of first saving the original function and than hooking the old location is called trampoline hooking and is coined by Microsoft's Win32 API hooking library Detour [5].

Using C++ we can create machine code using some assembly and converting that to bytes. Of course the created code has to be made executable. But this is no problem (e.g. through using the Linux functions *mmap*[2] or *mprotect*[3] function)).

Till now we have a foundation to modify machine code (*not* Java Byte Code!). So the next step is to get the function in memory. Luckily , here helps JNI, as it allows to get a class instance with a function call of *FindClass*[4] and *GetMethodID*[5] for retrieving a jmethod from a method by its name.

[4]

[2]http://man7.org/linux/man-pages/man2/mmap.2.html

[3]http://man7.org/linux/man-pages/man2/mprotect.2.html

[4]https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions.html#FindClass

[5]https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions.html#GetMethodID

4.2 Implementation

As we'll need to write some assembly in machine code in this scenario we will constrain ourselves to one specific Android version and one specific machine instruction set. As the instruction set we chose *i386 (x86)* and we chose *Android 8.0 (Oreo)* as the target platform.

-Until now only a proof of concept implementation is done that is rather rudimentary. Besides feature completion the final implementation will be furnished with a nicer logging Activity and fully documented source code.

- it could be implemented entirely in C++, but it would be (unnecessarily) labor intensive as one have to have deep knowledge of the internals of ART. But it would also be very fragile, since the internals of ART would have to be re-implemented. The behavior is therefore not only dependent from the machine instruction set, but also from the concrete implementation of interfaces and private functions, that could change from commit to commit. This are the reasons to favor a not entirely in C++ code written attack, but instead also use Java code, that cooperates with the native code.

- Disadvantages of Java loading part: Read and Write Permission for external Storage in the Android app manifest. An attacker could compile a function to native code and than extract the compiled malicious code and integrate it in the source code as a char array. Than the attacker just needs to make the code executable (e.g. using the *mprotect*. This executable code can then act as replacement hook for any function that has the same method signature (arguments and return value). This way, an attacker could entirely use native code for the attack and he/she further needs no additional permissions.

5 CONCLUSION

REFERENCES

[1] Vitor Monte Afonso, Paulo Lício de Geus, Antonio Bianchi, Yanick Fratantonio, Christopher Krügel, Giovanni Vigna, Adam Doupé, and Mario Polino. 2016. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy. In *NDSS*.

[2] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson. 2017. CHERI JNI: Sinking the Java Security Model into the C. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 569–583. DOI: https://doi.org/10.1145/3093337.3037725

[3] Google. 2017. Android NDK Intro. https://developer.android.com/ndk/guides/index.html. (2017). https://developer.android.com/ndk/guides/index.html Last accessed on 11/10/2017.

[4] Google. 2017. ART and Dalvik. https://source.android.com/devices/tech/dalvik/. (2017). https://source.android.com/devices/tech/dalvik/ Last accessed on 12/08/2017.

[5] Galen Hunt and Doug Brubacher. 1999. Detours: Binary Interception of Win32 Functions. In *Third USENIX Windows NT Symposium*. USENIX, 8. https://www.microsoft.com/en-us/research/publication/detours-binary-interception-of-win32-functions/

[6] S. Lee and J. W. Jeon. 2010. Evaluating performance of Android platform using native C for embedded systems. In *ICCAS 2010*. 1160–1163. DOI: https://doi.org/10.1109/ICCAS.2010.5669738

[7] Oracle. 2017. JNI Spec Chapter 2: Design Overview. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/design.html. (2017). https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/intro.html Last accessed on 11/13/2017.

[8] Oracle. 2017. JNI Specification Chapter 1. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/intro.html. (2017). https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/intro.html Last accessed on 11/13/2017.

[9] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)*. ACM, New York, NY, USA, 71–72. DOI: https://doi.org/10.1145/2414456.2414498

[10] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. (01 2014).

[11] Joseph Siefers, Gang Tan, and Greg Morrisett. 2010. Robusta: Taming the Native Beast of the JVM. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, New York, NY, USA, 201–211. DOI: https://doi.org/10.1145/1866307.1866331

[12] Mengtao Sun and Gang Tan. 2012. *JVM-Portable Sandboxing of Java's Native Libraries*. Springer Berlin Heidelberg, Berlin, Heidelberg, 842–858. DOI: https://doi.org/10.1007/978-3-642-33167-1_48

[13] Mengtao Sun and Gang Tan. 2014. NativeGuard: Protecting Android Applications from Third-party Native Libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks (WiSec '14)*. ACM, New York, NY, USA, 165–176. DOI: https://doi.org/10.1145/2627393.2627396

[14] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 48–62. DOI: https://doi.org/10.1109/SP.2013.13

[15] Haoyu Wang, Zhe Liu, Yao Guo, Xiangqun Chen, Miao Zhang, Guoai Xu, and Jason Hong. 2017. An Explorative Study of the Mobile App Ecosystem from App Developers' Perspective. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 163–172. DOI: https://doi.org/10.1145/3038912.3052712