Mobile Application Security: Detecting Improper SSL Implementations, Permission Misuse, and Vulnerable Interfaces

Elise Mai, Tiffany Tseng
[edmai,tatseng]@email.wm.edu

Abstract

Our research focuses on identifying key security vulnerabilities in a data set of 100 apps. We conducted four experiments to detect (1) permission over-privilege, (2) unprotected exported components that can lead to capability leaks, (3) vulnerable JavaScript-to-Java interfaces on Android WebViews, (4) improper SSL validation. Our methodology involved developing a Python script that employs Androguard to search the disassembled app files for certain classes, methods, and strings. Our findings reveal that a vast majority of apps are over-privileged and have unprotected exported components. Over 30% of apps are potentially vulnerable to cross-site scripting attacks on Android WebView on outdated devices and 15% are always vulnerable on any device. Nearly 50% of the total apps improperly performed certificate checks and over 30% improperly performed hostname checks. Our results indicate that developers need to adhere more closely to security principles and stop overriding secure Android defaults.

1 Introduction

This paper addresses five research questions related to 3 main research goals: permission misuse, SSL misuse, and interface vulnerabilities in Android applications. The research questions we addressed were:

1. Do apps ask for more permissions than they need, or do they follow to the principle of least privilege?

2. Do apps ignore certificate hostnames?

3. Do apps improperly implement custom SSL validation (i.e. TrustManagers and SocketFactories that trust all certificates)?

4. Do apps often have exported components?

5. Are apps vulnerable to attacks via JavaScript-to-Java interfaces for their WebViews?

Question 1 addresses permission misuse, questions 2 and 3 address SSL misuse, and questions 4 and 5 address interface vulnerabilities. More specifically, this paper addresses permission usage violating the principle of least privilege, vulnerable exported components, and SSL misuse by improperly verifying certificates and hostnames.

Permission misuse is a prevalent problem, with many applications violating the principle of least privilege [1]. Over 50% of apps are over-privileged or do not perform the appropriate permission checks before calling an API [4]. Exported components, especially components that run in the background of the app such as services, broadcast receivers, and content providers, leave the app open to attacks through intent hijacking and collusion. Exported components are probably prevalent because the default behavior in dealing with components is to set them as exported if intent filters are present and it takes more effort on the app developer's part to make sure intents are sent to their intended destination rather than avoid using implicit intents. The same reason applies to SSL misuse, where it is up to the app developer to validate certificates. It is easier to trust all certificates or allow all hostnames than to deal with an SSLError that hinders the function of the app.

In this paper, we will address over-privileged apps, exported components, and SSL misuse with four different experiments using AndroGuard, a Python tool used to reverse engineer and analyse vulnerabilities in Android apk files.

The remainder of this paper proceeds as follows. Section 2 gives some basic background information about the topics that we want to address in this paper. Section 3 overviews the high-level approach we took to addressing our research questions listed above. Section 4 describes in detail the design and implementation of our experiments. Section 5 evaluates our solution and describes the limitations of our approach. Section 6 discusses assumptions we made in our approach, limitations of our approach and additional topics for future work. Section 7 concludes with the implications of our results.

2 Background

In Android, phones rely on the permission model to protect the user's resources and other apps. The least-privilege principle is the idea that phone applications should only be allowed to access sensitive user data on the phone that they truly need to serve their function and components should try to obtain information through other means than requesting permissions to sensitive data. Failing to abide by the principle of least privilege can lead to security vulnerabilities, as attackers can leverage extra permissions to carry out malicious intents[1].

SSL is a cryptographic protocol that is used for network communication to protect messages from eavesdropping and tampering. The server obtains a certificate that contains its public key and is signed by a trusted Certificate Authority (CA). When the client connects to the server, the certificate is transferred to the client, who must validate the certificate. The implementation of a proper validation method is left up to the app developer, who often overlooks or ignores several things that make the validation process obsolete. Two common errors are trusting all certificates regardless of who signed them and what they were issued for, and allowing all hostnames without checking whether the certificate was issued for the address or not. Trusting all certificates can be done by implementing the TrustManager interface to trust all certificates or using Trust Managers and Socket Factories that trust all certificates. Allowing all hostnames can be done by using available Hostname Verifiers that already do so or building a custom hostname verifier that verifies all certificates. [3]

App components, such as activities, services, content providers, and broadcast receivers, can be internal or exported. Internal components receive requests from within the app only and external components can receive requests from other apps. In the AndroidManifest.xml file, the "exported" attribute is usually set to "false" by default, except in certain cases, such as the launcher activity, since it is expected that an app can be started from anywhere. [5] Specifically, if a component has an intent filter specified, its "exported" attribute is set to "true" by default, with the rationale that the absence of intent filters means that the component is intended to be invoked only by specifying its exact class name while the presence of intent filters means the component is intended for external use. [5] In the case that an intent filter is present, the "exported" attribute should be manually set to "false" because exported components can be accessed by any 3rd-party application that can potentially obtain confidential user information. The presence of exported components poses a possible vulnerability and opens the app to malicious attackers gaining access to confidential information.

Exported components can make an app vulnerable to intent hijacking via implicit intents. An explicit specifies a message and which application or component will satisfy the intent by providing the app's package name or the component's full class name. An implicit intent specifies a message, but does not specify any specific component that can satisfy the intent, so it is broadcast to any available component and the Android system finds the appropriate component to fulfill the intent based on intent filters declared in the Android Manifest. Intent hijacking occurs when the ActivityManager is tricked to send the implicit intent to a malicious component. Public components such as services, broadcast receivers, and content providers are more susceptible to intent hijacking than activities because they operate in the background. Also, when there is more than one activity that can accept an implicit intent, the ActivityManager allows the user to choose which activity to accept it, but if there is more than one service that can accept it, the ActivityManager randomly chooses a service to do so. with broadcast theft, intent hijacking is not even necessary because anyone who registers for a broadcast can receive an implicit intent. To specify who receives the broadcast, a permission must be created and the sendBroadcast API can be used to send the broadcast to the component with the permission.

JavaScript-to-Java interfaces are when a Java object instance is bound to a JavaScript variable name so that the JavaScript code can call any public methods on the Java object. This enables powerful features by allowing JavaScript code rendered in a WebView to call Java methods implemented in the app, allowing the web page in the WebView to interact with features such as the camera, microphone, or SMS manager. Though powerful, this is a huge security vulnerability that allows an attacker to remotely run malicious code in the app using the JavaScript-to-Java interface. Since WebViews are commonly from untrusted sources or over an unauthenticated HTTP rather than HTTPS connection, an attacker can inject malicious JavaScript into the HTTP stream, which is executed in the JavaScript sandbox and can use the JavaScript-to-Java interface to break out into the app context. Although Google has made attempts to fix the vulnerability, any app with a WebView and a JavaScript-to-Java interface is always vulnerable to the attack if they have a target API of 16 or older. An app with a JavaScript-to-Java interface is vulnerable only on outdated devices if it has a target API level of 17 or newer. [6]

3 Overview

Our approach was to use static analysis with Androguard to assess vulnerabilities in the applications. We analyzed 100 apps in total, 50 of which were smart/IoT apps and the other 50 were randomly sampled apps. We based our

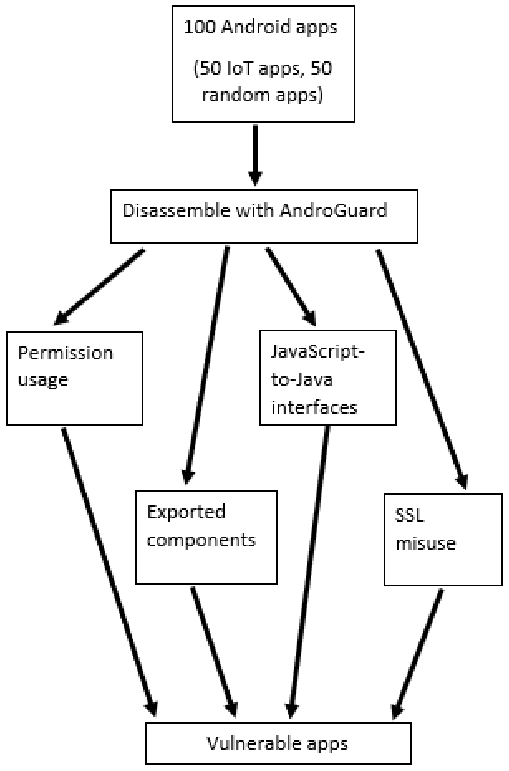
100 Android apps
(50 IoT apps, 50 random apps)
Disassemble with AndroGuard
Permission usage
JavaScript-to-Java interfaces
Exported components
SSL misuse
Vulnerable apps

Figure 1: A high-level architecture of our approach

hypotheses on what other papers on the topics found. Our general approach for each experiment was to use Androguard to disassemble the app, then search the app's code for strings that were relevant to our analysis or use Androguard's built-in functions to obtain information about the apps that were relevant, such as the app's SDK level or manifest file. The script we wrote printed counts of vulnerable apps to the terminal. Our high-level figure is shown in Figure 1.

Our design goal for this architecture is to identify all possible vulnerabilities related to apps having too many permissions, using exported components, using JavaScript-to-Java interfaces, and incorrectly checking hostnames and certificates. The main challenge is to try to limit the number of false positives, since static analysis is prone to lacking refinement and requiring manual verification to confirm results. We try to do this by prioritizing the most common ways that these vulnerabilities occur, like looking for common classes that are used to construct broken TrustManagers and HostnameVerifiers and looking for exported components specifically in services, providers, and receivers instead of in activities, where they are less likely to be exploited.

4 Design

The first experiment we designed answered a research question regarding permission misuse: "Do apps ask for more permissions than they need, or do they follow to the principle of least privilege?" We conducted this experiment by looking at what permissions the app requests and comparing them to what permissions it actually uses. This was done by keeping a list of all the permissions the app requested in its AndroidManifest.xml file, then parsing through the app's disassembled code and finding instances where the app actually checked for the permission. We hypothesized that apps are typically over-privileged, with over 50% of apps requesting permissions that they do not need [4].

The second experiment answered the question: "Do apps often have exported components?" Our script extracts the app's AndroidManifest.xml file and analyzes it for instances of a component being exported, specifically by searching for the string "exported="true"". We excluded instances where this statement was being specified for a launcher activity as well as instances where the exported component was permission-protected. We also looked through the manifest files for where intent filters were created, in which case we checked specifically to see if the the "exported" component was specifically set to "false." Of the unprotected exported components, we removed exported Activities because they are not as prone to intent hijacking and focused on components that operate in the background: services, broadcast receivers, and content providers. Of these most vulnerable apps with exported components, we sorted them by what permissions they had to give an indication of the potential impact of a capability leak. Our script outputs the number of apps with unprotected exported components, which is a potential vulnerability, and the number of potential confused deputies with dangerous permissions relating to the Internet, phone number, sensitive APIs (location, camera, microphone), and external storage.

The third experiment we designed answered a research question regarding interface vulnerabilities: "Do apps have vulnerable JavaScript-to-Java interfaces for their WebViews?" We looked to see if the applications had WebViews, JavaScript Interfaces, and enabled JavaScript, which are the three characteristics that make an app vulnerable to attacks via JavaScript-to-Java interfaces. If the application had all three of these attributes and a target SDK level of 16 or lower, it was marked as being always vulnerable for attacks on any Android device. If the application had all three attributes but a target SDK higher than 16, it was marked as being vulnerable on outdated devices, and if it had all three attributes but no target SDK level it was marked as being potentially vulnerable. We expected that most apps are not vulnerable to these kinds of attacks and some may be vulnerable only on outdated devices, with at least 60% of apps not being vulnerable.

Our fourth experiment answered the research questions regarding SSL misuse, which were ”Do apps ignore certificate hostnames?” and ”Do apps improperly implement custom SSL validation (i.e. TrustManagers and SocketFactories that trust all certificates)?” To check if apps ignore certificate hostnames, we obtained the names of several classes that were known to allow all certificate hostnames [3] and flagged apps that used any of them. We also checked for improper hostname verification for apps that implemented their own HostnameVerifiers by checking to see if the verify method of the SSL session was instructed to always return True.

To answer the research question relating to custom SSL validation, we obtained the names of several Trust Managers and Socket Factories that trusted all certificates [3] and flagged any apps that used these known broken classes. We also flagged apps that called proceed() when Android's SslErrorHandler was called, which is equivalent to ignoring all SSL errors. For apps that implemented their own Trust Manager, we checked if ”checkClientTrusted” methods were written to return void, which showed that the custom Trust Manager was trusting all certificates. We expected a fair number of apps to have issues with certificate verification, with around 20% of apps either bypassing SSL verification altogether by allowing all certificates or containing code that accepts all host names [3].

5 Evaluation

For the first experiment, we found that out of the 50 randomly sampled apps, only five apps used the exact number of permissions that they requested. The other 45 apps all requested more permissions than they used. Thirteen apps used at least three-fourths of the permissions they requested, 31 apps used at least half, and 44 used at least a quarter.

Out of the 50 IoT apps, only 2 apps used the exact number of permissions they requested, while the other 48 requested more permissions than they used. Seven apps used at least three-fourths of the permissions they requested, 21 used at least half, and 33 used at least a quarter.

For both datasets, our hypothesis that most apps are over-privileged was strongly supported. Ninety-three out of the 100 apps asked for permissions that they did not need. Of those 93, most of the apps (77) used at least a quarter of the permissions they requested and 20 used at least 3/4 of the permissions they requested, which shows that relatively few apps use more than 3/4 of the permissions they request. Most of the apps (around 56%) used at least half of the permissions they requested. There doesn't seem to be a significant difference between the random apps and the IoT apps in terms of which violated the rule of least permissions most.

For the second experiment on exported components (excluding exported activities), out of 50 random apps, 43 in total had unprotected exported components. Forty-three apps had internet permissions, 35 had READ_PHONE_STATE permissions, which allows the app access to the device's phone number, 7 had sensitive API permissions like location, camera, or microphone access, 32 had external storage permissions, and none of the apps had SMS permissions.

Of the 50 IoT apps, 40 in total had unprotected exported components. Of the 40 apps, all 40 of them had internet permissions, 9 of them had the READ_PHONE_STATE permission, 25 had permissions for sensitive API (location, camera, or microphone access), 3 of them had SMS permissions, and 30 had external storage permissions.

The vast majority of apps (83/100) in both the random and IoT datasets had unprotected exported components. All apps with unprotected exported components also had internet permissions. Most of the random apps also had READ_PHONE_STATE and external storage permissions, which makes the apps particularly vulnerable to 3rd party apps that might use collusion to obtain a person's phone number or any unprotected information they might store on their phone's external storage. The IoT apps had less apps with READ_PHONE_STATE permissions, but half the apps had permissions for sensitive API such as location, camera, or microphone, which can be vulnerable due to the exported components, and similarly to the random apps, more than half the IoT apps had external storage permissions.

The third experiment used the criteria mentioned in Section 4 to sort out apps that were always vulnerable to a JavaScript-to-Java attack, would be vulnerable on an outdated device, or are potentially vulnerable since they have all three characteristics of the vulnerability but do not specify a target SDK. The three characteristics that make an app vulnerable to this attack is having WebViews, JavaScript interfaces, and enabling JavaScript. If the applications with these characteristics had a target SDK higher of 16 or lower, they are always vulnerable to a JavaScript-to-Java attack. The experiment found that for the 50 random apps, 14 were always vulnerable to this sort of attack, 14 would be vulnerable on outdated devices, and 10 are potentially vulnerable but do not specify a target SDK level. Twelve apps were not vulnerable to this kind of attack at all.

For the 50 IoT apps, only 1 app was always vulnerable to JavaScript-to-Java attack, 18 apps were vulnerable on outdated devices, and 3 apps are potentially vulnerable but do not specify a target SDK level. Twenty-eight apps are not vulnerable at all.

Only 15 apps out of the 100 apps were always vulnerable to a JavaScript-to-Java attack (around 15%), which matched with our hypothesis that most apps are not vulnerable to these attacks and is an improvement from the paper published in 2014 that sampled 102,174 apk files and found 21.8% were always vulnerable [6]. Thirty-two apps in total are vulnerable on an outdated device with SDK 16 or lower, which is almost a third of the apps. Thirteen apps are potentially vulnerable but do not specify a target SDK level. Since this is a vulnerability associated with older apps and an older SDK that Google already started addressing in the past, it is expected that less apps are vulnerable to it as more apps change their target SDK to something above 16. This can also be seen with the IoT apps, which have noticeably less apps that are vulnerable to the attack than random apps; only 1 IoT app is always vulnerable while 14 random apps are always vulnerable. It is reasonable to assume that the IoT apps were developed more recently than the random apps, since smart apps have emerged only in recent years and probably on average use a higher SDK level than other apps, so are more resistant towards vulnerabilities associated with older SDKs.

The final experiment for detecting SSL vulnerabilities found that of the 50 random apps, 8 implement broken TrustManager/SSLSocketFactory classes and 20 implement custom TrustManagers that accept all certificates. This means 28 in total out of the 50 apps implement improper certificate verification that trusts all certificates. For hostname verifiers, of the 50 random apps, 11 implemented broken Hostname Verifier classes and 4 implemented improper custom HostnameVerifiers. In total, 15 out of 50 of the apps implemented improper hostname verification that accepted all hostnames.

Of the 50 IoT apps, 8 implemented broken TrustManager/SSLSocketFactory classes and 13 implemented improper custom TrustManagers. Twenty-one out of the 50 apps implemented improper certificate verification so that all certificates were trusted. Out of the 50 IoT apps, 12 implemented broken HostnameVerifier classes and 5 implemented improper custom HostnameVerifiers. Seventeen apps in total implemented improper hostname verification so that they accepted all hostnames.

These findings for SSL vulnerabilities confirm our hypothesis that a fair number of apps have issues with certificate verification. Forty-nine out of 100 apps, or almost 50%, of the apps trusted all certificates and 32 out of 100 apps, or almost a third of all the apps, accepted all hostnames. The random apps and the IoT apps did not differ significantly in number of SSL errors, which is worrying considering the IoT apps may be newer than the random ones but still do not seem to implement better SSL verification practices.

6 Discussion

For the experiment checking for over-privileged apps that ask for more permissions than they need, we only checked the manifest files for permissions that the app requested and then checked if permission checks for those specific permissions were performed throughout the app code. With this approach, if an app does not use all the privileges it asked for in the manifest, this does not necessarily guarantee the app uses a least privilege architecture. A least privilege architecture is when an app uses all possible ways to avoid asking for permissions as possible and in order to completely establish a least privilege architecture, all conceivable channels through which a component can interact with other components within and outside of the app have to be explored and mediated. The DelDroid tool [1] actually establishes a least privilege architecture and extracts the exact permissions that components need to provide basic functionality. Another better approach that could be used is constructing a permission map that keeps track of the permissions needed for an API call, which can more accurately determine the set of permissions needed by the app.

Our experiment checking for exported components only checks for strings that explicitly state "exported="true"" in instances where there are intent filters in the manifest. In our approach we assumed that any exported component is a security vulnerability, though in reality to ensure this we would have to manually verify all the instances of exported components and track the data flow through the app to make sure that it is actually a vulnerability. Our experiment can identify possible vulnerabilities, but may have many false positives since we assume the apps have a very simple data flow logic. For example, exported components may be permission protected, even if the manifest has the "exported="true"" attribute. A better approach would be to identify apps with more dangerous permissions and public components, then find an execution path that uses the permission.

The experiment checking for SSL vulnerabilities may miss some vulnerabilities due to the fact that we used a pre-existing list of known broken Trust Managers, Socket Factories, and Hostname Verifiers [3] when there could be more that have been used that we do not know of. Our approach also doesn't account for more complicated implementations of SSL verification and only accounts for the most common ways of improper SSL use, like using known broken classes, creating a TrustManager that trusts all certificates, calling proceed() to handle an SSL error, and implementing Java's HostnameVerifier.verify() function to only return True for all hostnames.

7 Conclusion

Our findings reveal that a shocking amount of apps in our data set have potentially exploitable security vulnerabilities. From our static analysis, we learned that a vast majority of apps are over-privileged and have unprotected exported components, a combination that might result in dangerous confused deputies and capability leaks. Over 30% of apps are potentially vulnerable to cross-site scripting attacks via the JavaScript-to-Java interface for WebViews on outdated Android devices and 15% are always vulnerable on any Android device. Nearly 50% of the total apps improperly performed certificate validation (either they implemented a known broken TrustManager or a custom one that bypasses all checks) and over 30% improperly performed hostname checks (either they implemented a known broken HostnameVerifier or a custom one that bypasses all checks). Though our data set is comprised of only 100 apps, a limitation that we acknowledge, these results nevertheless suggest that many Android apps in the Google Play Store have similar potentially exploitable vulnerabilities.

We can point to a few takeaways from our experiments and make recommendations for developers.

First, developers often fail to adhere to the principle of least privilege. Apps overwhelmingly ask for more permissions than they actually require, which increases the risk level if these apps are compromised. To minimize the attack surface, developers should stick to the minimal set of permissions needed [4].

Second, developers should be cautious about exporting components. Some developers might not be aware that using intent filters will automatically export the component. Communication within the app should use explicit intents to avoid intent hijacking. Data returned from components should be sanitized and verified. Developers can protect an exported component by requiring strong permissions; for example, Signature permissions can limit an exponent's exposure to only trusted apps [2].

Third, developers should understand that adding a JavaScript interface to a WebView can expose their app to injection attacks, leading to data leaks and even arbitrary code execution. Developers can prevent the JavaScript-to-Java interface vulnerability by ensuring that the WebView does not load content over an unencrypted network connection. Interestingly, this vulnerability might no longer be a problem in the future, if all Android devices run API version 17 or greater and all apps have a target API version of 17 or greater [6].

Lastly, developers should strive to implement proper SSL verification. Many SSL flaws occur because developers use known broken SSL classes or implement a custom TrustManager or Hostname Verifier that does not perform actual checks, putting the app at risk to man-in-the-middle attacks. Developers should adhere to Android's default SSL implementations and abstain from copying and pasting broken TrustManagers and HostnameVerifiers found on Stack Overflow.

With the rise of IoT apps and mobile apps in general, we need to be able to detect security vulnerabilities and evaluate the potential for exploitation in future apps. As smart phones, smart homes, and smart apps increasingly become intimately connected to our personal lives and have greater access to our personal information, these issues are more relevant than ever.

References

[1] H. Bagheri, M. Hammad, and S. Malek. Determination and Enforcement of Least-Privilege Architecture in Android. In *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017.

[2] E. Chin, A. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *MobiSys '11: Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011.

[3] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory Love Android:An Analysis of Android SSL (In)Security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[4] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou. Analysis of Android Applications' Permissions. In *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, 2012.

[5] J. Six. Chapter 4. Component Security and Permissions. In *Application Security for the Android Platform*, 2011.

[6] D. R. Thomas, A. R. Beresford, T. Coudray, T. Sutcliffe, and A. Taylor. The Lifetime of Android API vulnerabilities: case study on the JavaScript-to-Java interface. In *Cambridge International Workshop on Security Protocols*, 2015.