

# Rethinking SSL Development in an Appified World

Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, Matthew Smith

Distributed Computing & Security Group  
Leibniz University Hannover, Germany  
Hannover, Germany

{fahl,harbach,perl,koetter,smith}@dcsec.uni-hannover.de

## ABSTRACT

The Secure Sockets Layer (SSL) is widely used to secure data transfers on the Internet. Previous studies have shown that the state of non-browser SSL code is catastrophic across a large variety of desktop applications and libraries as well as a large selection of Android apps, leaving users vulnerable to Man-in-the-Middle attacks (MITMAs). To determine possible causes of SSL problems on all major appified platforms, we extended the analysis to the walled-garden ecosystem of iOS, analyzed software developer forums and conducted interviews with developers of vulnerable apps. Our results show that the root causes are not simply careless developers, but also limitations and issues of the current SSL development paradigm. Based on our findings, we derive a proposal to rethink the handling of SSL in the appified world and present a set of countermeasures to improve the handling of SSL using Android as a blueprint for other platforms. Our countermeasures prevent developers from willfully or accidentally breaking SSL certificate validation, offer support for extended features such as SSL Pinning and different SSL validation infrastructures, and protect users. We evaluated our solution against 13,500 popular Android apps and conducted developer interviews to judge the acceptance of our approach and found that our solution works well for all investigated apps and developers.

## Categories and Subject Descriptors

D.4.4 [Software]: Communications Management—*Network communication*; H.3.5 [Information Storage and Retrieval]: Online Information Services—*Data Sharing*

## General Terms

Security, Human Factors

## Keywords

Android, Apps, iOS, MITMA, Security, SSL

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [Permissions@acm.org](mailto:Permissions@acm.org).

CCS '13, November 4–8, 2013, Berlin, Germany.

Copyright 2013 ACM 978-1-4503-2477-9/13/11 ...\$15.00.

<http://dx.doi.org/10.1145/2508859.2516655>.

## 1. INTRODUCTION

The proliferation of smartphones and tablet devices is changing the way software is being developed. Both the Android Play Market and the Apple App Store offer over half a million apps and more than one billion apps are installed from these markets every year. The “*There is an app for that*” attitude and the possibility to quickly reach a huge global market has inspired thousands of small software companies and hobby developers to create software on an unprecedented scale. These join big businesses in vying for the lucrative app market. One issue many developers seem to have in common is that they often have problems when working with SSL. In previous work, we conducted an in-depth study of 13,500 Android apps which showed that a large number of apps did not use SSL correctly [5], thus making them vulnerable to Man-In-The-Middle attacks. The affected applications ranged from home-brew and open source apps to those developed by large corporations and security specialists, suggesting that SSL problems are not just a matter of untrained developers getting it wrong.

While our previous work and a related study by Georgiev et al. [8] discuss possible reasons for why so many apps across such a wide range of developers are affected and make recommendations on how to prevent these problems in the future, the actual causes of the problems have not yet been identified nor have the potential countermeasures been evaluated. In this paper, we continue where this work left off and evaluate the root causes of SSL coding problems. Based on these findings, we argue that the way developers work with SSL needs to be changed significantly. We designed a framework for SSL development to demonstrate our proposal, implemented it for Android, evaluated it against the set of 13,500 popular apps we analyzed in our previous study and conducted developer interviews to show its functionality and feasibility.

Our solution makes several changes to the status quo: First, it removes the need for developers to write actual SSL code. Instead an app developer can turn on and configure SSL using only configuration options in case they intend to deviate from the standard use case. Second, unlike in the current system, all use-cases we found in apps and those described by developers are supported securely, removing the need for dangerous customization. Third, the framework can make a distinction between developer devices and end-user devices. This allows developers to rapidly prototype applications and tinker without much effort, but it also automatically and properly protects end-users. Fourth, in the remaining rare cases that there is a problem with SSL,

the end-user is reliably informed about the nature of the problem which means that apps can no longer silently ignore these warnings and invisibly make users vulnerable to MITM attacks. Finally, our approach allows for new SSL validation strategies and infrastructures, such as Certificate Transparency [11], Convergence [13], and AKI [10], to be deployed centrally instead of potentially requiring tens of thousands of app developers to make adjustments to their code, thus significantly easing the deployment of new SSL validation strategies<sup>1</sup>.

Our contributions can be summarized as follows:

- We conducted the first analysis of iOS SSL security to ascertain whether the walled garden approach and stricter code auditing process of Apple could prevent the kind of problems previously seen on Android. We manually examined 1,009 iOS apps. The results show that, similar to Android, 14% of apps using SSL do not implement SSL validation correctly and are thus vulnerable to active MITMAs.
- We conducted the first in-depth study of the reasons behind the widespread problems with SSL on the two major app platforms Android and iOS, including both technical aspects as well as an in-depth study with affected developers.
- Based on the above research, we designed and implemented countermeasures for handling SSL on Android. This solution can also serve as a blueprint for SSL handling on other appified platforms.
- We conducted an extensive evaluation of our approach by auditing 13,500 apps and showing that our solution covers all use-cases present in these apps. We also conducted a follow-up developer study to ensure our approach does not break any development needs and would find acceptance within the target community.

## 2. RELATED WORK

### 2.1 SSL Validation

The main body of work on SSL has centered around finding a replacement for the current weakest link, the CA-based validation of certificates. Many new proposals have been made that promise to strengthen the security of certificate validation, including Perspectives [18], Convergence [13], Certificate Transparency [11], Sovereign Keys [3], TACK [12], and DANE [9]. However, none of these systems has achieved widespread adoption yet. While this is certainly at least in part due to the fact that some have only recently been proposed, a major issue faced by all systems is that they require modification of client code handling SSL connections. While on desktop systems updating the handful of browsers would already cover a large amount of SSL real estate, the situation on appified platforms is bleak. Here thousands of apps contain their own SSL validation implementation and thus must potentially update their code to support a new certificate validation strategy.

<sup>1</sup>We believe this last feature could be useful in the light of the many new solutions suggested in this domain that are currently facing adoption/deployment problems.

### 2.2 SSL on Appified Platforms

Research has shown that the appification and app market trend has caused new security and privacy challenges for users, developers and researchers [15, 2, 4, 19, 5, 1, 17, 7, 6]. This research focused mainly on the threats posed by malicious entities and their apps. The problems with SSL we [5] and Georgiev et al. [8] discovered are different in that they are not caused by malicious intent, but nonetheless pose a serious threat.

While the bulk of SSL connections on desktop systems occurs in browsers which validate SSL certificates correctly, there are also applications that use SSL to protect their communication. Georgiev et al. [8] analyzed the security of SSL certificate validation in a wide range of SSL libraries and programming frameworks. They conclude that many popular libraries fail when it comes to SSL certificate validation and thus endanger the applications which are based on these libraries.

In [5] we examined the state of SSL on Android. We analyzed the SSL security of 13,500 popular free apps from Google's Play Market. The results showed that 1,074 apps contained SSL code that either accepted all certificates or all hostnames for a certificate and thus leaves the users potentially vulnerable to MITMAs. The cumulative install base of the apps with confirmed vulnerabilities against MITMAs ranged between 40 and 185 million users.

Georgiev et al. [8] make recommendations for what could be done to alleviate these widespread problems. They recommend that app developers should use fuzzing and adversarial testing. Developers should not modify application code and disable certificate validation for testing with self-signed and/or untrusted certificates, but create a temporary keystore with the untrusted CA's public key in it. They also recommend that app developers should not rely on SSL libraries to do things correctly for them. Instead they should set all necessary security parameters themselves explicitly. For library developers, they recommend that SSL libraries be made more explicit about the semantics of their APIs. They also recommend that libraries use safe defaults as much as possible. Furthermore, they should not silently skip important functionalities such as hostname verification and should remain consistent in using return values or flags for reporting purposes.

Our central suggestion was to drastically limit custom SSL handling in apps. In an alternative approach we suggested that a static code analysis be performed by the app store or the app installer to then inform developers/users about potentially unsafe code.

Both papers only briefly describe their potential countermeasures as part of their recommendations for future work. No implementations or evaluations were presented. Interestingly, the countermeasures suggested by the two parties diverge somewhat in their direction. Georgiev et al. call for better APIs for developers, emphasize that developers need to check their apps themselves instead of relying on libraries to do things correctly and to work around API restrictions using custom keystores instead of modifying validation for development purposes, while we suggest almost the opposite approach by recommending to limit the developers' capabilities for customizing SSL and adding checks to prevent broken apps from entering the market or the device. In the next two sections of this paper, we will examine the root causes of the problems facing developers when using SSL to better

judge which countermeasures are likely to achieve the best results.

### 2.3 SSL Development Paradigm

There are countless SSL libraries and TrustManagers that aim to make the integration of SSL into apps easier. However, as the studies by Georgiev et al. [8] and us [5] have shown, many of these are either broken or so error-prone that developers break their apps using them. To the best of the authors' knowledge there is no related work evaluating the root causes of these SSL coding problems nor any work suggesting changes to the development paradigm. Our paper makes important contributions to both these areas.

## 3. SSL ON IOS

To examine whether or not the widespread SSL problems of apps and libraries introduced above are endemic to the Android and open source ecosystems, we conducted the first in-depth analysis of iOS apps to see if the more restrictive and curated Apple App Store would prevent apps with broken SSL code from entering the iOS ecosystem. While iOS does not offer developers as many options as Android, it still offers similar freedom concerning the implementation and use of SSL. Developers can decide if they want to use SSL or not and just like on Android they can use SSL but can turn SSL certificate validation off. They are also left alone with the challenge to find an appropriate way to handle certification validation errors and inform the user: In terms of SSL APIs, Android and iOS are fairly similar. However, Apple performs a code analysis on all apps, in order to prevent apps that do not conform to their policies from entering the store.

We did not have the means to automatically crawl Apple's app store and use static code analysis on tens of thousands of apps as we did with Android, so we opted for a manual approach. Initially, we downloaded 150 cherry-picked apps for analysis. Based on the findings in these apps we conducted developer interviews as described in section 4.2. We then extended our study to include 1,009 apps for a more robust evaluation. We downloaded the 1,009 apps by selecting the most popular free apps. Since iOS does not work with permissions the way Android does, it was not possible to see which apps have access to sensitive information and can connect to the Internet before installation. We therefore installed all of the apps on an iPhone 4S running iOS 5 to study them in action. We then mounted active Man-In-The-Middle attacks against SSL connections using a transparent proxy to see how the apps react and what kind of sensitive information we could gather.

We captured network traffic from 884 apps and SSL-protected network traffic from 697 of these 884 apps. Of these 697 apps, 98 (14%, 9.7% of all 1,009 apps) were vulnerable to our MITMA and leaked sensitive information. Of the remaining 599 apps that were not vulnerable to our active MITMA, 312 apps presented the user with a warning message; 58 apps presented a warning message indicating that there were problems with the SSL certificate. 254 presented warning messages that did not give an appropriate description of what was going on, for instance stating that the login credentials were wrong or that there was no Internet connection available. Finally, 287 apps simply did not connect to the attacked host either doing nothing, hanging indefinitely or crashing. Additionally, 82 (9.27%) of the 884 apps used

plain HTTP connections to transfer sensitive information. Two apps were vulnerable to SSL stripping attacks. One of these was an online banking app that loaded the bank's website via plain HTTP, while the other app connected to the HTTP version of Facebook. Thus we were able to gather sensitive information from 182 apps, i.e. 20.5% of apps from which we captured network traffic. The information included the usual suspects of login credentials, banking accounts, data stored on cloud storage services, emails, or chat messages.

This shows that the SSL problems on iOS are similar to those on Android and that Apple's more restrictive and curated app development model and App Store do not prevent SSL-related security issues. Just like on Android, app developers turn off SSL certificate validation, write apps that are vulnerable to SSL stripping and even in cases where apps apply SSL certificate validation correctly, they often do not present sensible feedback when validation fails.

While there are many more details worth discussing in the context of iOS, for the purpose of this work, the fact that iOS apps suffer from a similar number of SSL problems shows that these problems must have underlying causes which are not specific to a platform or app store model. For more details on the iOS analysis the reader is referred to AppendixA.

## 4. CAUSE ANALYSIS

The iOS study shows that the trouble developers have using SSL correctly is common to the major platforms and across all applications and platform paradigms. To develop an effective countermeasure, we wanted to identify the root causes of these problems. Therefore, we studied entries about SSL development for both Android and iOS in online forums and conducted interviews with developers who had produced broken SSL code.

### 4.1 Online Forums

We used `stackoverflow.com`, a popular online forum for software developers, to search for results that contained text such as *"android/ios allow all certificates"* and *"android/ios trust all certificates"* first. These kinds of results were found in threads where developers asked how they could *"work with self-signed certificates"* or *"make 'java.net.ssl.SSLException' go away"*. Answers to these and similar questions mainly contained explanations on how to turn off SSL certificate validation or hostname verification without mentioning that this would create serious security issues. Studying these discussions confirmed our impression that many developers on both platforms – iOS as well as Android – lack an adequate understanding of how SSL works and were frustrated with the complexity of customizing SSL code and thus willing to use the quick fixes offered by the forums – potentially without understanding the consequences. These threads had more than 30,000 views. To put this number into perspective, searching for *"android ssl pinning"* or *"ios ssl pinning"* returns only one result for Android and two results for iOS with less than 600 views in total.

### 4.2 Interviews

While online forums give a good indication of potential issues, a more reliable way to confirm whether or not these issues are the actual reasons causing the problems in the thousands of real world apps is to talk to developers. We

contacted 78 developers of the 82 vulnerable apps<sup>2</sup> from the 100 Android and 150 iOS apps we studied in detail. We informed the developers of the discovered vulnerabilities via email and kindly asked them to contact us for further information and assistance in fixing their problems. We received responses from 39 of the 78 developers. We disclosed the respective vulnerability, offered further assistance and asked whether the developers were willing to discuss the details of the security bug either via telephone or email. We promised the developers to anonymize all information they provided and to make neither their names nor the apps' names publicly available. 14 developers agreed to an interview while the rest was not willing to discuss the topic in further detail – often with regard to “constraints” dictated by their legal departments. The interviews were conducted in German or English, depending on the developer's origin. However, not all developers were native speakers. Statements were translated by the authors, grammatical errors were not corrected.

#### 4.2.1 Results

One of the main causes for the problems we found, was that developers wanted to use self-signed certificates during development. 5 of the 14 developers reported that they needed to turn off SSL certificate validation during development because they were working with test servers that used self-signed SSL certificates. To avoid certificate validation exceptions, they implemented their own SSL certificate validation strategies that accept all certificates, or copied code from the online forums mentioned above that promised to help with getting out of the “self-signed certificate dilemma”. While it is understandable that developers turn off SSL certificate validation in the development phase, these developers basically forgot to remove their accept-all code when they released their apps. Three of these five developers realized that this was a serious security threat and stated that they should fix this security issue as soon as possible. The other two did not see the problem and even after our explanations stated the following:

*“You said that an attacker with access to the network traffic can see the data in cleartext. I tried that and I connected my phone to a Wi-Fi hotspot on my laptop. When I used Wireshark to look at the traffic, Wireshark said that this is a proper SSL protected data stream and I could not see any cleartext information when I manually inspected the packets. So I really cannot see what the problem is here.”*

This supports the hypothesis stated by Georgiev et al. [8] that too little adversarial testing is conducted by app developers. However, it also raises the issue that there are developers, who, while being technically adept enough to use Wireshark to check if their app's traffic is really encrypted, do not understand the nature of the threat and thus take no precautions to counter it.

Apart from developers wanting to use self-signed certificates during development, we also talked to developers who actually wanted to use them in their production environment but were unaware of the security implications of accepting any certificate:

*“I was using a self-signed certificate for my app because it is free and CA-signed certificates cost a lot. But, ac-*

<sup>2</sup>Some developers were responsible for both an Android and an iOS app; thus there were only 78 developers for 82 apps.

*tually, I had no idea that working with self-signed certificates could have resulted in such a security issue. I think the online forum where I found the code snippet only said that it makes self-signed certificates work.”*

*“We added this piece of code because our client uses an SSL certificate for his web-service which was signed by a certificate authority that is not pre-installed on Android and actually we did not realize that this would cause such trouble.”*

Sometimes, the broken SSL code was added because developers had difficulties understanding the problem and just went for the first solution that seemingly made the problem disappear:

*“This app was one of our first mobile apps and when we noticed that there were problems with the SSL certificate, we just implemented the first working solution we found on the Internet. [...] We usually build Java backend software for large-scale web services.”*

However, there were also developers who even after being informed about the problems and the threat scenario did not properly understand the problem and their countermeasures did not address the threat arising from a MITMA:

*“We hadn't realized that it would cause such an issue by using self-signed certificates in the past time, and we just verified if the certificate was expired. But after noticing this issue, we strengthened the security check like verifying host name. We believe this improvement can ensure users' security. So we still stick to trust self-signed certificates right now for its smaller size and lower bandwidth cost.”*

So while they added hostname verification after we informed them about the issue, they still accept all self-signed certificates thus defeating hostname verification entirely. In another case, a development company of a vulnerable online banking app needed two iterations to fix their app correctly, even though we had sent them the necessary code snippets. There were also cases where developers thought using broken SSL was adequate to protect information that they deemed to be not that valuable:

*“We checked into this. Only the [...] feature is using a weak SSL certificate and that connection only sends the device models and IMEI, but that's not a security concern.”*

Some developers knew that their code could cause security problems but saw no other option but to work with self-signed certificates by turning off certificate validation entirely, since their customers wanted to use self-signed certificates.

*“This issue exists because many of our customers use self-signed certificates for SSO (single sign on). Some time back, a fix was implemented to allow this to work.”*

One of those developers raised the interesting point that Android does not offer any default warning, forcing developers to provide one for themselves if they wish to inform users about failed certificate validations:

*“The app accepts all SSL certificates because some users wanted to connect to their blogs with self-signed certs*

and [...] because Android does not provide an easy-to-use SSL certificate warning message, it was a lot easier to simply accept all self-signed certificates.”

The consequence of this design decision was that all users of this app were at risk because some wanted to use self-signed certificates.

The iOS developers in our study tended to rely on frameworks and libraries during development and were understandably startled and upset when told that their apps were endangered because of faulty code generated by the framework (cf. Appendix A):

*“I am using the MKNetworkToolkit as a network wrapping library and its SSL features for HTTPS. After you informed me of the issue I checked the library’s code and found that by default SSL certificate validation is off. But, when I used the library in my app, I trusted it and did not check for the SSL MITMA vulnerability because it is a widely used library.”*

*“When I was starting to build apps for iOS, I had a strong background in coding web applications. When I came up with the Titanium framework that allows developers to build native mobile apps by writing HTML and Javascript, I decided to use this framework just because it was easier for me. [...] I never thought that the framework would produce broken code for SSL encryption and [...], although we conduct security audits for our apps, we did not include SSL certificate validation checks into the audit process.”*

The feedback from app developers confirms that developers struggle to implement SSL correctly when they have a need to deviate from the standard use-case. They also rely on the implementations of frameworks and libraries to protect their apps without thoroughly testing either the frameworks’ or their app’s security. However, our investigation also provides some new insights: developers modified certificate validation code for internal testing purposes, e. g. working with arbitrary self-signed certificates on test servers, but forgot about that and thus did not remove the code for the production environment. So even those developers who understood the current need for signed and trusted certificates put their customers at risk. Also, the developers’ problems did not only lie in the complexity of the code, but were based on a lack of understanding of how SSL works. There were also some cases where developers turned SSL validation off because of a customer’s request, either accepting or not realizing the implications. Even when we explained what could go wrong and how to correct it, developers struggled when trying to fix their app (cf. Section 4.4). Altogether, our results imply that code-level customization of SSL-handling is an overwhelming problem for many developers and that there is a fairly high level of frustration with the complexity of adapting code to their use-cases.

### 4.3 Summary

After studying code snippets and advice in developer forums, as well as talking to app developers that use broken SSL certificate validation, we believe that in most cases when Android and iOS developers deviate from default SSL certificate validation strategies – that are secure on both platforms by default – they apply customization features in a way that weakens security significantly (c.f. Section 6.2 for

a quantitative confirmation). Many developers of affected apps seem to have only a partial understanding of what SSL does and how it works. Yet, there are also developers who complain about the bad support for self-signed certificates and the lack of easy-to-use SSL warning messages.

One interesting aspect we found in the interviews was that developers in general seem to be interested in providing a high level of security for their users. We offered all developers to help them with their SSL problems and most of them took the offer. After giving background information on the security model of SSL and certificate validation, 10 of the 14 interviewed developers accepted our assistance. 7 of these 10 developers decided to strengthen their app’s security by implementing SSL pinning. They did this because it gave them full control over the SSL certificates trusted by their app and they found that this was the most secure, flexible and cheapest way to provide a high level of security. We provided the developers with code to integrate SSL pinning based on Moxie Marlinspike’s github page<sup>3</sup>. All developers agreed that being able to control exactly which certificates their apps trust is a great way to increase security, but that they would not have known how to do this without our help.

Our results imply that allowing app developers to customize SSL handling on source-code level overburdens many developers and leads to insecure apps. While it is easy to weaken app security by removing the default SSL certificate validation code, it is hard to strengthen it by implementing pinning or other security-strengthening strategies. Only one developer stated that using insecure SSL should not be taken too seriously, which makes us believe that in most cases insecure SSL connections are unintentional and that users must be protected against careless developers and developers who usually are no security experts.

### 4.4 Follow-up Analysis

Our developer study showed that many developers were unaware of the dangers facing their SSL connections and interested in fixing the issues. Some of the developers who accepted our help were capable of fixing the SSL problems in their apps. To analyze how developers cope with this situation without direct help, we ran a follow-up analysis. All affected developers (iOS & Android) were informed about the vulnerabilities and the possible security consequences for their users at the time of discovery of the vulnerability. They were given the recommendation that they should fix the identified issues as soon as possible. Three months after the respective notifications, we downloaded the apps again to check if they had fixed the SSL vulnerabilities. We found that 51 of the 78 developers did not fix the SSL issues. Six apps were not available any more which means we could not test them a second time. Only 21 (26.9%) apps were fixed and implemented correct SSL certificate validation in their current versions. Of these 21 apps, 9 belonged to developers we had helped directly during the interview process. However, 5 of the 14 developers we interviewed did not fix the SSL issues in their apps. Furthermore, developers of an app that included vulnerable SSL certificate validation on both Android and iOS only fixed it for Android while the iOS app was still vulnerable in the second test.

The results of this follow-up analysis indicate that even after informing and educating developers about vulnerabilities in their SSL code, problems in correcting these mistakes and

<sup>3</sup><https://github.com/moxie0/AndroidPinning>

deploying a safe solution still remain. Finding that 73.1% of all informed developers did not fix the reported SSL issues demonstrates even more that the current SSL mechanisms on appified platforms need rethinking.

## 5. A NEW APPROACH TO SSL SECURITY ON APPIFIED PLATFORMS

In the previous sections, we showed that incorrect SSL validation is a widespread problem on appified platforms and analyzed the causes of these issues. In a follow-up study, we found that only a small part of previously vulnerable apps had fixed their app’s SSL vulnerabilities, even after we informed the developers about the problems. In the following, we propose a major change in how app developers use SSL to address these issues. While we implemented our ideas for the Android platform, they can serve as a blueprint for increasing SSL security on other platforms such as iOS, but also OS X and Windows 8 which are moving towards the app paradigm as well.

Based on our analysis described above, we believe that simplifying SSL libraries or trying to educate developers in the context of SSL security will not solve the problem. For most developers, network security is not a primary concern and they just want to “make it work”. An ideal solution would enable developers to use SSL correctly without coding effort and prevent them from breaking SSL validation through customization. However, it is also important not to restrict their capabilities to produce functional (and secure) applications: If our solution does not offer the needed functionality, developers will be tempted to break it just like they are breaking the built-in SSL code at the moment.

Before going into detail, we summarize the desired features for SSL validation identified during analysis:

**Self-Signed Certificates – Development.** Developers commonly wish to use self-signed certificates for testing purposes and hence want to turn off certificate validation during testing.

**Self-Signed Certificates – Production.** A few developers wanted to use self-signed certificates in their production app for cost and effort reasons.

**Certificate Pinning / Trusted Roots.** Developers liked the idea of having an easy way to limit the number of trusted certificates and/or certificate authorities.

**Global Warning Message.** Developers requested global SSL warning messages since they described building their own warning messages as too challenging.

**Code Complexity.** Developers described the code-level customization features of SSL as too complex and requiring too much effort.

In addition to these developer requirements, we add a user requirement to our list of desired changes on how SSL is handled in apps. We base this goal on the related area of SSL handling in browsers. While a website can choose not to offer SSL at all, it cannot prevent the browser from warning the user about an unsafe connection. The website also cannot turn off SSL validation for the user. In the world of apps, developers currently have the power to define SSL policies for an app without those being transparent for the user.

While there are well known usability problems with SSL warning messages in browsers [16], we believe that allowing developers to for instance silently ignore SSL errors and put the user at risk is worse. We thus define an additional goal:

**User Protection.** The capabilities of a developer should be limited in a way that prevents them from invisibly putting user information at risk.

To achieve all these features, several changes to the way SSL is used on appified platforms are necessary. First and foremost, we propose the following paradigm-shift: instead of letting all developers implement their own SSL code (and potentially break SSL in the process, with no chance for the user to notice), the main SSL usage patterns should be provided by the OS as a service that can be added to apps via configuration instead of implementation. This is a fairly radical shift in responsibility, however, we believe there now is enough evidence to warrant this move. Furthermore, the evaluation of our system presented in Section 6 shows that it is both technically possible and acceptable from the developer’s standpoint. Configuration instead of implementation also lends itself well to offer other requested features, such as allowing developers to turn off SSL certificate validation for their app on their device in the settings during development. This would allow the use of self-signed certificates during development, but would not affect the installation of an app on a user’s device. Surprisingly, none of the major mobile or desktop operating systems provide this feature, although we believe it would offer significant benefits to all parties.

The platform should offer configurable options for the new SSL service so that developers cannot and need not circumvent security features on the code level. The simple removal of the need to tinker with SSL security aspects for testing purposes will already reduce the amount of vulnerable apps significantly. It will also protect users from developers who do not understand how SSL works and who therefore make honest mistakes during implementation.

Table 1 gives an overview of our proposed modifications compared to the traditional code-level approach.

### 5.1 Implementation on Android

Figure 1 gives a high-level overview of the modifications we implemented to create the proposed SSL service on Android. The white boxes contain classes we modified or created for our solution. The dashed lines show Android components that are now circumvented since they proved to be insecure. The grey boxes are comments on what the different components do. The start arrow shows the entry point where app code passes control over to the central SSL system. The features offered by our solution are presented in the following sections.

### 5.2 Features

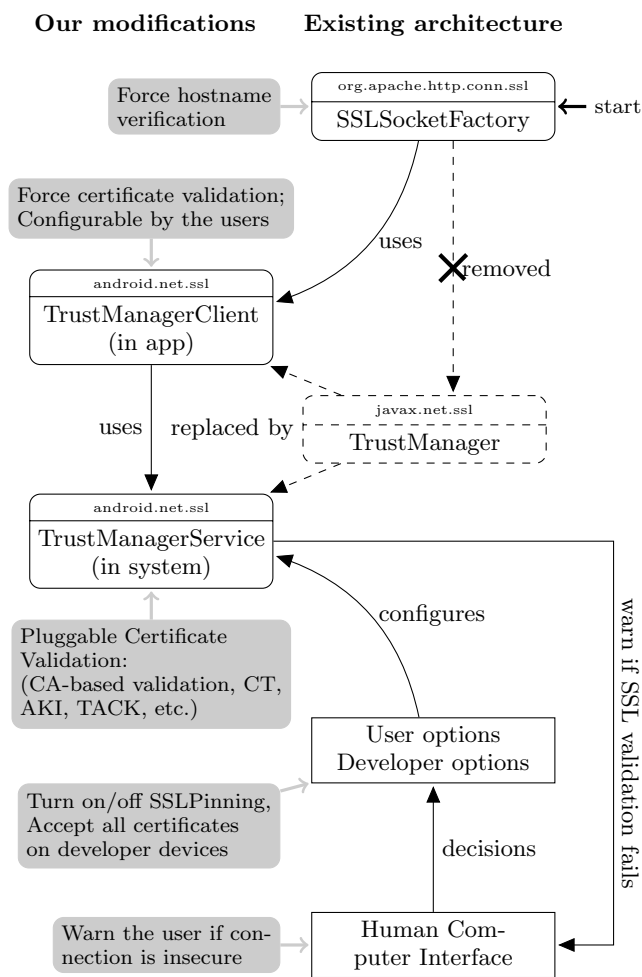
#### 5.2.1 Mandatory SSL Validation

As stated above, we propose that the capability and need to customize SSL certificate validation and hostname verification on source code-level is removed. Instead, SSL certificate validation should be enforced for every SSL handshake automatically, while taking into account the different usage scenarios such as development vs. production.

	CA Validation	CA Pinning	Certificate Pinning	Development Mode	Logging	Validation Strategies
Standard	✓	—	—	—	—	—
Our approach	✓	✓	✓	✓	✓	<i>P</i>

**Table 1: A comparison between the status quo and our approach concerning validation features.**

✓ = supported out of the box;  
 — = custom code required;  
*P* = pluggable.



**Figure 1: This figure illustrates the process of creating an SSL protected network connection. The grey boxes comment on our contributions.**

To this end, we provide the `TrustManagerClient` and `TrustManagerService` that replace the capabilities of Android’s default `TrustManager` (cf. Figure 1). We only modify meth-

ods which are private and final, thus binary compatibility is given and we do not break modularity. More information on the compatibility of our approach can be found in Section 6.2 and Appendix B. Both the client and service part of our SSL validation implementation prevent Android apps from using broken certificate validation. Upon creation of a socket, the newly developed `TrustManagerClient` automatically requests SSL certificate validation from the service counterpart. App developers cannot circumvent secure validation anymore, since customized `TrustManager` implementations are prevented by our modification. The `TrustManagerService` enforces SSL certificate validation against the trusted root CAs and can drop the connection or present the user with a warning message in case validation fails (more on this in Section 5.2.4).

To mandate secure hostname verification, we patched all stock hostname verifiers to enforce browser compatible hostname verification. We also added hostname verification to the central `SSLSocketFactory` (cf. Figure 1). Hostname verification is conventionally delegated to the application layer: With HTTPS for example, the hostname for verification is extracted from the requested URL. In contrast, Android’s `SSLURLConnection` implementation does not check the hostname, even though it may have been provided in the method call. Our patch improves this behavior by verifying hostnames with the parameters provided during connection establishment for any SSL connection.

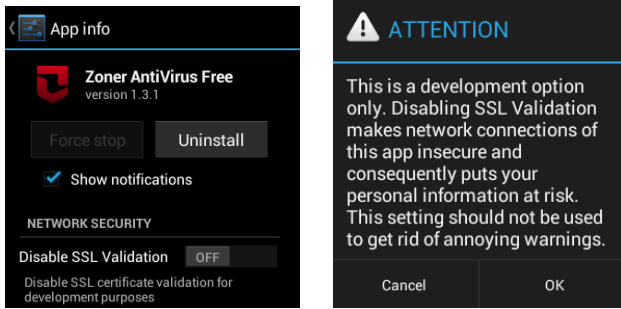
This strict enforcement could cause developer issues in some usage scenarios described by our study participants, so several configuration options are described in the following in order to adapt our solution to different situations. Additionally, we discuss potential pathological cases in the appendix (see App. B.1).

### 5.2.2 Self-Signed Certificates

To allow developers to use self-signed certificates for testing purposes, we add a new option (cf. Figure 2) to the *Developer* settings, allowing app developers to turn off SSL certificate validation for specific apps installed on their device without needing to modify the code of their app. This option is monitored by the `TrustManagerService` and skips certificate validation for this app only. These settings only affect the specific app on the developer device, not the apps deployed onto users’ devices or other apps on the developer’s device. Thus, even if developers forget to turn on certificate validation again, this has no effect on apps on user devices. This feature effectively protects users from forgetful developers and solves many of the problems we discovered during code analysis and interviews.

We only allow this option on devices that have developer settings enabled. Thus, app developers have a simple way to work with self-signed certificates during development while preventing careless users from turning off SSL certificate validation for their apps.<sup>4</sup> Nonetheless, we show a warning message using strong wording that advises against abuse (cf. Fig. 2(b)) when this option is toggled.

<sup>4</sup>While it is conceivable that users annoyed by warning messages could find information online on how to activate developer options and then turn off certificate validation for a specific app, we believe this risk is fairly low compared to the huge benefit this option brings. Additionally, we recommend limiting this option to devices that are registered with Google developer accounts to prevent normal users from



(a) Developer settings to turn off certificate validation for an app. This option is not displayed for normal users. (b) On disabling validation, a message warns against security threats.

**Figure 2:** Instead of writing code, SSL parameters can be changed with via Developer settings options.

### 5.2.3 SSL Pinning

SSL public key pinning can be configured by app developers to limit the number of certificates an app trusts. It can either be used to specify exactly which CA certificates are trusted to sign SSL certificates for this app or directly specify which individual SSL certificates are to be used. The standard Android approach to use certificate pinning requires developers to implement it individually in their source code, resulting in only very few apps implementing pinning at all.<sup>5</sup> The standard approach for limiting the number of trusted issuers is using a custom keystore, which is also complex and requires custom code to load the keystore. Using our extension, it is possible to configure SSL certificate pinning in an Android app's `Manifest.xml` file. This allows developers who know which endpoints their app connects to (this should be most apps) to easily and securely add SSL pinning without having to write any code. Figure 3 shows the `Manifest.xml` for an Android app with SSL pinning.

Pinning individual leaf certificates also allows developers to use self-signed certificates in a production environment in a secure way, which is a requirement several app developers articulated (cf. Section 4). In case developers wish to restrict the number of trusted issuers instead of pinning individual leaf certificates, pins for certificate issuers can be added to the `Manifest.xml` file in the same way. The app then accepts only certificates issued by the pinned certificate issuers. To simplify the process of creating certificate pins and adding them to the `Manifest.xml` file, we extended Eclipse's Android Development Tools. This way pins for given certificates can be generated and added to an app's `Manifest.xml` file automatically.

Since SSL public key pinning can be problematic in some rare cases, for instance if a company mandates the use of an SSL proxy<sup>6</sup>, we allow for SSL pinning to be disabled globally using the (enterprise) Device Administration API policies. While careless users cannot unintentionally turn off SSL pinning since they do not have access to the De-

carelessly breaking their apps' SSL security.

<sup>5</sup>One of these apps is the Twitter app by Twitter Inc., for whom Moxie Marlinspike developed the SSL pinning code.

<sup>6</sup>Some companies use SSL proxy servers to monitor the network traffic of their employees.

```

<!-- SSLConfiguration for my App -->
<ssl>
  <pinning>
    <!-- Only trust my self-signed certificate. -->
    <pin type="leaf" val="18:DA:D1:9E:26:7D..." />
    <!-- And certificates signed by Verisign. -->
    <pin type="issuer" val="8F:57:5A:C8:5B..." />
  </pinning>
  <logging level="INFO" />
  <handle-validation-fails action="Do-Not-Connect" />
</ssl>

```

**Figure 3:** The new SSL configuration options for an Android app's `Manifest.xml` file allow developers to easily configure different options for handling SSL. Developers can pin either leaf or issuer certificates, determine if their app should log SSL-relevant information and observe how their apps react to failed SSL certificate validations. By default no pin is set, logging is turned off and apps refuse to connect to hosts for which certificate validation failed.

vice Administration API, enterprises can configure devices to respect company policies.

### 5.2.4 User Protection

Currently it is entirely up to the developers to implement the UI to interact with the user when something goes wrong with SSL. This has led to a large number of apps silently accepting invalid certificates, crashing or displaying unintelligible warning messages such as: *"Reset your local time to the current time"* when faced with a certificate validation error. The lack of a ready-to-use warning message was also an issue criticized by developers in our study. It should also be impossible for app developers to invisibly accept untrusted certificates without the users' consent. We offer a system-triggered, standardized warning that gives app users the chance to recognize security threats originating from insecure SSL connections, thus preventing app developers from silently accepting invalid certificates. This capability is needed for apps that connect to endpoints outside of the control of developers and thus might not have trusted certificates, such as mobile browsers, news readers, blog aggregators, etc.<sup>7</sup>. In these cases, the users are allowed to decide if they want to connect anyway after being shown a warning message.

Usability studies on SSL warning messages for browsers [16] show that designing meaningful and effective SSL warning messages is a challenging task. Designing such a system is outside the scope of this paper which is why we use Android's stock browser warning message for now. Still, we think that having a standardized warning message that app developers can use to let the user decide what to do with untrusted certificates is a good starting point for such future work. Issues such as habituation need to be taken into account at that point, but showing any warning message is better than allowing apps to silently accept all invalid certificates due to developers' negligence.

<sup>7</sup>One of the developers we interviewed explicitly stated he turned off certificate validation for his app entirely because his customers wanted to connect to blogs with self-signed certificates.



While we no longer allow developers to silently accept connections for which validation fails, we do allow developers to be more restrictive and drop connections for which validation fails without allowing the user to override. This is the correct (and default) behavior for most apps where the developer knows which endpoints the app communicates with and these endpoints have valid certificates, such as online banking, social networking and most other single purpose apps. As long as developers exercise due diligence with their server certificates, the only validation errors would be in the presence of a real MITMA.<sup>8</sup> In these cases, users would be effectively protected from themselves.

The decision of whether a warning message should be displayed or connections should be dropped was added as a configuration parameter to an app’s manifest (cf. Fig. 3).

Thus, our framework can protect users of multi-purpose apps from developers who would hide warnings and accept untrusted certificates as well as enabling developers to protect their users from accidentally accepting connections from MITMAs for apps where the endpoints are known in advance.

### 5.2.5 Alternate SSL Validation Strategies

One feature which offers promising future potential is the capability of our system to plug new validation strategies into the system and thus protect both new and existing apps without requiring a large number of app developers to update their code. This could significantly speed up the adoption of alternatives to the current weakest-link CA based system. We have created a plugin infrastructure for this purpose and are in the process of evaluating Certificate Transparency (CT) [11] and AKI [10] as new approaches to validate certificates. This feature is still in the experimental phase and is currently configured into an app’s manifest. It is a matter for future work to research how and by whom this feature should be configured and of course to create plugins to improve certificate validation.

## 6. EVALUATION

As Section 4 showed, all broken SSL implementations on Android and iOS came about because developers wanted to customize the way their app uses SSL and failed to do so safely. While our new approach to SSL-development in apps closes all the security holes we found, it will only be adopted by developers if all their needs are met and they feel comfortable with configuration instead of coding. To evaluate our approach, we conducted two evaluation studies.

We ran interviews with the 14 developers of our previous developer study to discuss how they perceived our proposed solution and if they would be comfortable using it. Since the number of developers available for interview was fairly small, we also did an extensive code analysis of all custom SSL-handling code in the set of 13,500 Android apps to ensure that all use-cases could be covered by our solutions and thus remove the need for dangerous code-level customization.

### 6.1 Developer Evaluation

<sup>8</sup>We realize not all developers practice due diligence with their certificates, however we still believe this to be the right default setting. With this setting developers would quickly realize that there is a problem with their certificate and would be forced to update it.

We conducted a pre-test study in which novice developers added SSL to their apps and used pinning and self-signed certificates. Since our framework made all these tasks trivial, we decided not to deploy this study on a large scale, since it would not have led to any valuable insights. Instead, we focused on whether the proposed paradigm shift would find the developers’ approval and make them feel comfortable with configuring SSL instead of implementing it. We presented our approach to the 14 developers from Section 4.2 and queried them if these features would fulfill their requirements and remove the need to customize the way their app uses SSL. We also asked if they would feel comfortable using configuration instead of coding to add advanced features such as SSL pinning to their apps.

All use-cases of these developers were met with our new design and the reaction of the developers was very positive. They confirmed their previous statements – that SSL development is too complex – and that they very much appreciated anything which would ease the burden. They were particularly positive about the option to use self-signed certificates during development and to use pinning in such an easy way for their production apps. None of the interviewed developers were concerned that they could not fulfill their certificate validation tasks with the provided configuration options. Due to the small number of developers willing to discuss the problems they have with SSL development, we decided to follow up this qualitative study with a quantitative study.

### 6.2 Compatibility Evaluation

To evaluate whether our proposed solution really covers all relevant use-cases, we ran another static code analysis on the set of 13,500 Android apps [5]. We extracted all customized TrustManager implementations and manually analyzed the semantics of the `checkServerTrusted` methods. Unlike in the previous study, we also analyzed the 2.04% of implementations that customized SSL handling without breaking SSL. While there are not many apps in this category, it is still imperative that these good apps also continue to work as expected with our new approach. In total, we found 3,464 classes that implement customized TrustManagers. We categorized them based on their handling of SSL validation compared to the default procedure. Table 2 gives an overview of the customizations we found. We denote a customization that weakens certificate validation with a “-”, a customization that strengthens validation with a “+” and customizations that do not affect validation security with a “=”.

Customized Implementations	3,464	Security Impact
Accept All Certificates	3,098	-
Expiry-Only Check	263	-
Leaf-Cert Pinning	47	+
Add Logging to Default Validation	32	=
Add Hostname Pinning to Default Validation	16	+
Limit Trusted Issuers	8	+

**Table 2: Distribution of Customized TrustManager Implementations in Android Apps**

In the 97.02% of cases where TrustManagers accept all certificates or only check certificates' expiry dates, our approach protects app users from careless developers by enforcing secure certificate validation. As we previously showed [5], 97.1% of the endpoints in the 1,074 vulnerable apps had valid certificates. In these cases, our modification fixes the apps without any development effort or negative side-effects for the developer or the user. In the remaining 2.9% of cases, the endpoints used by the apps do not have valid certificates. In these cases, our system would prevent the connection, unless the developer installs a valid certificate, updates their application to pin the current certificate or sets the `handle-validation-fails` option in the manifest (cf. Fig. 3) to show warning messages. Since all three modifications a developer would have to perform are very easy in our framework, we believe this to be a good trade-off for the broken apps.

While the majority of all implementations turns effective SSL certificate validation off entirely, a small number of developers created beneficial customizations. 0.9% of TrustManagers add logging to the default certificate validation process. While this does not strengthen certificate validation itself, it still is a potentially positive feature that should not be made impossible. We therefore added a configuration option (cf. Fig. 3) that allows developers to get log output from the framework's validation process.

0.5% of implementations add hostname verification directly to the certificate validation process. This strengthens an app's security since default SSL certificate validation does not cover checking the hostname during the SSL handshake outside of an `HttpsURLConnection`. This feature is covered in our framework (cf. Section 5.2.1 above).

In the 1.6% of cases where TrustManagers were used to improve SSL validation (i.e. through pinning a leaf certificate or a CA), the functionality added by the custom code is available as a configurable option in our solution. Thus, we found no custom SSL code which implements a use-case that is not covered by our solution with significantly less effort. We conducted field trials that confirmed this analysis.

### 6.3 Deployability

All our modifications are implemented as part of Android's Java Framework. A system update would thus be the most convenient way to make the new features of our system available to developers and users. All apps built from then on would use this update by default and would benefit from our framework's features. Existing apps' binaries do not need to be modified to benefit. However, it is possible for power users with root access to their device to install our modifications on their devices without having to wait for an official update or having to make major changes to their device, such as flashing the device.

A noteworthy feature of our solution is that it does not break binary compatibility and could in theory be used to protect existing apps as well. Our modifications only affect private final methods of the Java Framework and thus do not break modularity or collide with developer code. This would instantly fix all the vulnerable apps we discovered in our studies. However there are some rare pathological cases which would need to be considered if our system were to be applied to all apps blindly. We discuss these issues and the possible trade-offs in Appendix B.1.

## 7. LIMITATIONS

We contacted 78 developers and asked them to participate in interviews about the identified SSL issues in their apps. Only 14 developers agreed to participate in an interview while all others either did not respond at all or turned down our request with reference to company guidelines forbidding them to talk about confidential information. Despite the relatively small number of participants, we were still able to identify a wide range of causes which lead to serious security issues in the context of app development and SSL. After the developer study, we conducted a large scale code analysis and did not find any indication that there were further issues that would warrant more interviews.

## 8. CONCLUSION

In this paper, we argue for a new way of handling SSL connections on appified platforms, since previous work discovered severe problems in this area. To discover the root causes of these problems, we conducted a study of 1,009 iOS apps to ascertain whether iOS suffers from the same problems as Android. We also surveyed developer forums and conducted a developer study. Based on our findings, we proposed to rethink how developers interact with SSL: instead of requiring developers to work with SSL on the code level, we designed and implemented a framework that allows them to protect their network connections via configuration. Our solution prevents developers from willfully or accidentally breaking SSL, while at the same time giving them easy access to additional features, such as pinning and the secure use of self-signed certificates. We evaluated our proposal with existing Android apps and showed that all use-cases we found can be implemented more easily and securely with our approach. The feedback we gathered from developers was also very positive.

There are several important areas of future work: Our framework offers all necessary features to include alternative certificate validation strategies such as Certificate Transparency and AKI just to name two. We are currently working on plugins for these proposals to implement these promising solutions. Since currently there is a great amount of research being conducted on these alternative validation approaches, we believe our framework can make a valuable contribution by enabling this research to be deployed with greater ease and thus not only to be studied better but also hopefully to help with real world adoption.

Furthermore, while we currently recommend our solution to be adopted by new or updated apps, it is in theory possible to apply our changes to all existing apps, thus fixing all broken apps with one update. In future work, we will investigate how to discover and avoid possible side-effects in apps that applied benign code modifications to the SSL validation process. For more on this avenue of research, the reader is referred to Appendix B.

## Acknowledgments

We would like to thank all the developers that agreed to support our research, openly discussed security issues in their apps and provided us with the background needed to design our solution.

## References

- [1] P. P. F. Chan, L. C. K. Hui, and S. M. Yiu. Droid-Checker: Analyzing Android Applications for Capability Leaks. In *WISEC '12: Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM Press, Apr. 2012.
- [2] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 13th International Conference on Information Security*, pages 346–360, 2011.
- [3] P. Eckersley. Sovereign Key Cryptography for Internet Domains. <https://git.eff.org/?p=sovereign-keys.git;a=blob;f=sovereign-key-design.txt;hb=master>.
- [4] A. Egners, B. Marschollek, and U. Meyer. Messing with Android’s Permission Model. In *Proceedings of the IEEE TrustCom*, pages 1–22, Apr. 2012.
- [5] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*. ACM Press, Oct. 2012.
- [6] A. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM Press, 2012.
- [7] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM Press, Oct. 2011.
- [8] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications security*. ACM Press, Oct. 2012.
- [9] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Protocol for Transport Layer Security (TLS): TLSA . <https://tools.ietf.org/html/rfc6698>, Aug. 2012.
- [10] T. Hyun-Jin Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In *Proceedings of the 2013 Conference on World Wide Web, to appear*, 2013.
- [11] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. Network Working Group Internet-Draft, v12, work in progress. <http://tools.ietf.org/html/draft-laurie-pki-sunlight-12>, Apr. 2013.
- [12] M. Marlinspike. TACK: Trust Assertions for Certificate Keys. <http://tack.io/draft.html>.
- [13] M. Marlinspike. SSL And The Future Of Authenticity. In *BlackHat USA 2011*, 2011.
- [14] P. Saint-Andre and J. Hodges. RFC 6125, Mar. 2011.
- [15] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, and e. al. Soundcomber: A Stealthy and Context-aware Sound Trojan for Smartphones. *Proceedings of the Network and Distributed System Security Symposium*, 2011.
- [16] J. Sunshine, S. Egelman, H. Almuhiemedi, N. Atri, and L. Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proceedings of the 18th USENIX Security Symposium*, pages 399–416, 2009.
- [17] T. Vidas, D. Votipka, and N. Christin. All Your Droid Are Belong To Us: A Survey Of Current Android Attacks. In *Proceedings of the 5th USENIX Workshop on Offensive Technologies*, pages 10–10, 2011.
- [18] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: improving ssh-style host authentication with multi-path probing. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC’08, pages 321–334, Berkeley, CA, USA, 2008. USENIX Association.
- [19] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, 2012.

## APPENDIX

### A. IOS SURVEY DETAILS

Several of the vulnerable apps we found were created using popular programming frameworks. Since any bug introduced by such a framework could potentially affect a large number of apps, we decided to take a closer look at these frameworks. During our analysis, we identified two cross-platform mobile application SDKs and an iOS networking wrapper library that all create code that contains vulnerable SSL certificate validation.

#### *MKNetworkKit.*

The MKNetworkKit<sup>9</sup> is a networking wrapper library for iOS with the aim to be easy to use and to simplify the iOS default networking stack. In online forums, such as [stackoverflow.com](http://stackoverflow.com), developers often complain about problems with iOS’s built-in networking APIs, accounting for the popularity of libraries such as MKNetworkKit, which provides lightweight methods for standard networking tasks. The library can also handle HTTPS requests, but fails when it comes to SSL server certificate validation.

#### *Titanium Framework.*

The Titanium cross platform mobile application SDK<sup>10</sup> is a JavaScript-based platform which enables developers to write mobile apps in JavaScript and automatically translates them into native mobile apps. Titanium targets iOS, Android and HTML5, making it particularly attractive for web developers who want to create mobile apps. While the

<sup>9</sup><http://blog.mugunthkumar.com/products/ios-framework-introducing-mknetworkkit/>

<sup>10</sup><http://www.appcelerator.com/platform/titanium-sdk/>

Titanium framework generates secure SSL code for Android, SSL certificate validation for iOS apps is turned off. Based on app creation statistics posted on their website, this could affect more than 30,000 apps built with the framework.

### *PhoneGap.*

PhoneGap<sup>11</sup> is a free open source framework for developing mobile apps for seven platforms including Android and iOS, using HTML, CSS and JavaScript. It contains dedicated classes for data transfer that include customized SSL verification code. For Android and iOS, the framework produces code that effectively turns SSL certificate validation off. If developers do not manually check the generated code, they will not see the comment and thus not be aware of the problem. According to PhoneGap<sup>12</sup>, more than 23,000 apps could be affected.

## **B. TRANSITION PERIOD/COLD-TURKEY UPDATE**

With such a high number of users at risk and such a slow/non-existent response time by developers when fixing the vulnerabilities, it might be worth considering activating our framework not only for new apps but for all existing apps as well. This would instantly fix all instances of apps with broken SSL we could find. There are two ways in which our framework could be deployed in such a case. The first approach would simply override custom SSL code. This would work fine for 98% of the 13,500 apps we analyzed and fix all the broken ones. However, the remaining 2% would lose important functionality (such as custom pinning) until the developers update their apps. They would still have standard certificate validation but their custom improvements would be disabled. So while applying our framework for all apps in this way is simple and helps most apps, it would be good to avoid this undesirable side-effect. A second approach could combine our framework's validation code with the app's custom SSL validation code. In this case, the simple rule would be only if both validation methods accept a certificate, then the connection is established. If one of the two validation processes rejects a certificate, a warning message is shown to the user by our framework.<sup>13</sup> The main issue with this is that users could potentially see two warning messages for the same connection: In those cases where validation legitimately fails and the developers of the app followed best practices and warns the user and the users accepts the invalid certificate, our framework would also warn the user and the user would have to accept again. While this is not a security threat per se, it is a highly undesirable characteristic to have in a system. However, our code analysis has shown that these cases should be very rare when no MITM attacker is present. Thus this could be considered an acceptable trade-off to fix the many vulnerabilities which otherwise will remain unfixed for an unknown amount of time. It might also be possible to get the developers of the "good" apps to update their apps for the greater good by

<sup>11</sup><http://phonegap.com/>

<sup>12</sup><http://www.slideshare.net/AndreCharland/phone-gap-stats-growth>

<sup>13</sup>For the cold-turkey approach, the default warning mechanism would be switched from "drop connection" to "warning" since we cannot automatically tell in advance if the app has a legitimate reason to connect to untrusted hosts.

contacting them before the cold-turkey roll-out. Ironically getting the few "good" developers to react is probably easier than getting all the developers with vulnerable apps to react. However, evaluating and discussing the ramifications of this kind of roll-out is beyond the scope of this paper and would need to be discussed in the community.

## **B.1 Pathological Cases**

There are some pathological cases that need to be considered when activating our framework for existing apps:

### *B.1.1 IP Addresses Instead of Hostnames*

The current Android API allows URLs and socket connections to be established with IP addresses instead of hostnames. Using an IP address has the drawback that hostname verification might not work properly. The concept of virtual hosts for HTTP(S) servers hinders effective hostname verification when an IP address is used to establish a connection instead of a hostname, since common names for SSL certificates typically are fully qualified domain names [14]. Yet, there are certificate authorities<sup>14</sup> that issue certificates with IP addresses as the common name. So, while creating a secure connection using an IP address can cause problems because hostname verification fails, there is also a valid use-case for this practice. Our SSL API treats IP addresses as normal hostnames during hostname verification.

We analyzed the set of Android 13,500 apps to find those that include URLs using IP addresses instead of hostnames to estimate the scope of this practice in Android apps. In all 13,500 apps, we found 163 apps (1.21%) that include IP address-based URLs, pointing to 118 different hosts. 88 of these IP addresses did not support SSL. Of the 30 which supported SSL, only one of the remaining used a certificate for which hostname verification did not fail. However, this certificate was self-signed, so none of the IP address-based apps used SSL correctly.

### *B.1.2 Custom Sockets*

If an app implements a custom application layer socket that resolves hostnames by itself, it may rely on the SSLSocketFactory to create an SSL-secured socket based on only an IP, but cater for hostname verification itself at a later stage. Given the modifications of our framework, this implementation would break, as our modified SSLSocketFactory would attempt to verify the hostname (in this case the IP address) during the handshake and fail because the hostname in the certificate presented by the server is unknown to the SSLSocketFactory. While it is uncommon to not delegate hostname resolution to the operating system, we acknowledge that such implementations would need to be updated to work with our modifications.

<sup>14</sup>e.g. <https://www.globalsign.com/>