Pin it!
Improving Android Network Security At Runtime

Damjan Buhov*, Markus Huber†, Georg Merzdovnik*, and Edgar Weippl*
*SBA Research, Vienna, Austria
E-mail: {dbuhov, gmerzdovnik, eweippl}@sba-research.org
† St. Pölten University of Applied Sciences, St. Pölten, Austria
E-mail: markus.huber@fhstp.ac.at

***Abstract*—Smartphones are increasingly used worldwide and are now an essential tool for our everyday tasks. These tasks are supported by smartphone applications (apps) which commonly rely on network communication to provide a certain utility such as online banking. From a security and privacy point of view a properly secured (encrypted) communication channel is important in order to protect sensitive information against passive and active attacks. Previous research outlined that developers often fail to implement proper certificate validation in their custom SSL/TLS implementations and thus fail to secure the network communication. Previous research however proposed solutions for developers and not for the affected users. This global growth introduced drastic changes to the network utilization.**

In this paper we discuss this issue on the basis of Android apps. We analyzed over 50,000 Android apps, collected during two consecutive years, regarding the correct use of SSL/TLS protocols. Furthermore, we discuss the current situation. We propose dynamic certificate pinning, a device-based solution that overcomes the problem of broken SSL/TLS implementations in Android apps. To the best of our knowledge, we are the first to solve this problem by combining established techniques such as certificate pinning with dynamic instrumentation techniques to tackle one of the major security challenges in the network communication of smartphone applications.

I. Introduction

The increased use of smartphones also means more risks. People tend to part with the traditional desktop computers, even notebooks, and satisfy their requirements and needs with smartphones. This is possible, because all web services essential for personal and business use are rapidly transforming to meet the requirements of the mobile domain. The vast majority of those applications rely on network communication for data transfer, implying that applications that deal with sensitive user information are required to provide secure (*encrypted*) communication. Although there are multiple concepts that provide secure communication, Android applications most commonly use SSL (*Secure Sockets Layer*) and its successor TLS (*Transport Layer Security*) [16]. These two protocols are used to securely connect the client with a legitimate server. In Android, the security of SSL/TLS is in close interdependence with the client application. This means that the client (application) should employ proper verification methods for the server certificate, the hostname and deal with the SSL/TLS errors correctly. Hence, the verification logic is completely controlled by the application, or in other words the application developers are responsible for implementing the SSL/TLS certificate validation correctly. If such a validation scheme is not correctly employed, users face the risk of a *Man-in-the-Middle (MITM)* attack [28]. Such attacks can have significant implications especially in the financial sector, which increases the importance in the mobile domain. By default, the Android applications trust only certificates validated against the internal trust store, however, there are certain cases in which the developers need to implement custom validation of the certificates. Whether developers add custom implementation due to the need of applying additional protection measures such as *certificate pinning* or just because of problems they encounter with their self-signed certificates, the networking part of the application could easily become a vulnerable spot as a result of mistakes in these implementations. Since Android is the most dominant market share holder [15], such vulnerability will affect millions of users.

Recent research outlined that particular measures have to be taken in order to solve this issue. There have been many attempts, however, the situation has not changed. Tendulkar et al. [37] proposed that all the SSL/TLS configuration including pinned certificates etc. should be stated in the manifest file as a part of the application package. Back in 2012, Fahl et al. [26] examined the state of more than 13,000 applications regarding their SSL implementation. Using their script Mallodroid [12] they discovered that more than 1,000 applications from the data set were vulnerable to MITM attacks. All of the previous research efforts, however, propose instructions and solutions that are aimed towards the developers and not the affected users. This implies that even if there is a solution for the problem, developers might ignore this fact for various reasons. Another major drawback of using certificate pinning is the update interval of the applications, since it has been proven that from the time the update has been released by the vendor, until it reaches the users, makes the application totally unsuitable from a security point of view [31]. Thus, the goal of this paper is to detect and resolve this problem on a large scale and in real time, focusing on the affected users instead of the developers. For a single application or a very small set of applications it is possible to detect and fix this issue using static instrumentation. However, we aim for a dynamic device-based solution that overcomes the SSL/TLS issue and reaches

ISBN 978-3-901882-83-8 © 2016 IFIP

a large number of users. In summary, the contributions of this paper are:

- We analyzed the most popular 50,000 Android applications gathered over the period of two years and discovered that the SSL/TLS issue still exists and, more importantly, we found an increase of nearly 6% in terms of vulnerable applications from 2014 compared to 2013.
- We present a novel approach for patching Android applications during runtime that solves the SSL/TLS problem without any user interaction. Our solution employs established techniques such as *certificate pinning* combined with dynamic instrumentation techniques and provides a tool that can be installed on every Android device.
- To the best of our knowledge, we are the first to actually provide a solution that combines these techniques and directly affects the user, unlike most of the research effort that is aimed towards the developers.

The remainder of this paper is organized as follows. Section II provides the background and threat model of our work. Section III briefly describes our methodological approach. In Section IV we present a detailed explanation of the implementation of the proposed tool while Section V presents the results from the static analysis as well as evaluation of our tool. Section VI is reserved for discussion about the advantages, limitation and future work for the proposed solution. In Section VII we revise existing approaches that tackle this problem. Finally, Section VIII concludes the findings of our paper.

II. Background and Motivation

In this section we provide a brief overview on the security of Android with an emphasis on providing secure network communication. We furthermore discuss our threat model.

A. Android Security

Smartphones and tablets continue to replace the traditional desktop computer; Android has the biggest share of the overall smartphone market. The ongoing transition from desktop operating systems to mobile operating systems such as Android brings along a number of security improvements for the average user. Android relies on the concept of multilevel security [30] and, compared to traditional desktop operating systems, each Android application is executed in an isolated sandbox. In combination with Google's firm control over available applications with their Play Store[1], the impact of common security threats such as malware has been limited. While the overall security of Android outlines a number of security improvements, users still face security and privacy risks. These risks emerge from the ever growing amount of third-party applications. The development of third-party applications relies on the capability and knowledge of the developer. Their lack of knowledge or ignorance of security issues introduces grave implementation bugs in applications, which have an overall negative impact on the security of mobile users. Security of these applications is achieved through the use of the Android permission model and sandboxing which defines a particular memory space for the application to execute. This way the system ensures that only allowed resources will be available to the particular application. In most of the cases the requested resources by applications include permission to access the network/Internet. Hereby, the actual implementation within the application dictates the level of security for the network communications. In Android, SSL/TLS are the standards that enable secure communication and are widely used among the Android applications. Because of the fact that the security of SSL/TLS relies on certificates, proper implementation of the validation procedure for the server certificate is essential. By default, every Android device comes pre-shipped with 150+ root certificates.[2] These 150+ root certificates are used to ensure that applications can verify that they are communicating with legitimate servers. This basic network security model requires that application developers buy legitimate certificates from Certification Authorities (CAs). Nowadays, there are numerous Certificate Authorities that issue verified certificates, however, just 15 of them hold more than 95% of the market shares [2], [17]. There are cases in which the developers are using self-signed certificates. The use of self-signed certificates puts an obligation on the developer to ensure that proper security mechanisms are put in place to achieve the same level of security as with officially signed certificates.

B. SSL in Android

The implementation of SSL in Android is achieved through certain packages provided by the Android SDK [1]. Usually, developers make use of the *javax.net.*, java.net.*, android.net.*, java.security,*, org.apache.** modules which provide them with all the important interfaces such as *TrustManager* and *HostnameVerifier*. Furthermore, the TrustManager interface contains a method called *checkServerTrusted* through which the validation of the certificates is performed. Developers can choose whether to use the default configuration of the SSL/TLS or to implement their own custom version. Usually, the default configuration is used when the application is using trusted certificates, whereas custom implementation is required for any other case. In both cases, they must ensure that the validation of the certificates is properly implemented. This procedure can be described as follows:

- **Certificate verification** The server sends the chain of certificates to the application. At this point the application tries to validate the chain using the bottom-up approach, i.e. starting from the end certificate (also known as leaf certificate) and continue to the intermediate and root certificate. The validation of the certificates includes checks for the expiration date of the certificate and whether it is signed from its successor in the list or from a trusted root certificate. In this setup the last certificate is usually signed by one of the certificates that came with the device.

[1]https://play.google.com

[2]Location of the Root CAs: Settings → Security → Trusted credentials

If the validation succeeds, the connection is established; otherwise it is immediately terminated.

- **Hostname Verification** Another very important check is the hostname verification. Every certificate has its designated destination so the application has to check whether the certificate is issued for the desired destination. This information is usually found in the Common Name (CN) field or the subjectAltName. According to the newer standard [14], the subjectAltName should be checked first and if it exists, the CN field should not be checked at all.

Although this validation procedure works for certificates that are signed by some of the root certificates that are preshipped with Android, there are certain cases in which the developers need to implement their own logic. The most common reason behind this is the fact that most of the Android developers make use of self-signed certificates for various reasons, such as testing the product before official release, or simply because of financial reasons. When using this kind of certificates, developers are obligated to perform custom implementation of the validation procedure to make the application immune to the most common threat described in *II-D Threat Model*.

C. Certificate Pinning

Among all available advanced protection measures, *certificate pinning* [10], [11], [24], [29], [33] stands out as the most recommended one. With proper implementation, it reduces the risk of Man-in-the-Middle attacks to a minimum. This technique is the most common representative of the advanced concepts with respect to the custom use of SSL/TLS protocols that was previously mentioned. There are a lot of publicly available solutions that could be directly applied in order to secure the network communication and in particular Android applications. Certificate pinning works by bundling the server certificates with the application. Hereby, the application verifies the security of the network communication based on its included *Pins* (server certificates). Therefore, developers do not need to buy third-party certificates from CAs, and applications can, moreover, detect attacks in which trusted certificates are forged.

Since the application receives the whole chain of certificates from the server side, developers are left with the choice of which certificate to pin. Accordingly, pinning different certificates from the entire trust chain brings its own advantages as well as disadvantages:

- *Pinning the end certificate (leaf)* reduces the attack surface to a minimum since there are no certificates that are or could be signed from it; however, it is potentially subjected to a major drawback when it comes to change. These types of certificates are subjected to a change more often than the intermediate certificate, which implies that with every change of the leaf certificate the application has to be updated with the new pins, otherwise it will not be usable in terms of network connectivity.
- *Pinning the intermediate certificate* has a reasonably larger attack surface in comparison with the previous category, but it requires less updates since this certificate is not changed very often.
- *Pinning the root certificate* leaves the biggest attack surface compared to the previous two categories, however, in this case the update frequency is the lowest.

Furthermore, this technique could be applied both to the whole certificate as well as just to the public key of the certificate. Although it is the easier solution and in general it seems natural to pin the whole certificate, it is not the optimal one. The reason behind this is the fact that certificates can be reissued multiple times. This means that we can encounter multiple certificates with the same public key, but with different attributes, e.g. expiration date. Based on this assumption, we can conclude that it is more convenient to pin the public key of the certificate. Although the implementation of this approach is more difficult due to some extra steps regarding the key extraction, in the end, this approach significantly reduces the need for frequent updates of the application.

D. Threat Model

Our threat model regarding the network security of Android applications focuses on *Man-In-The-Middle (MITM)* attacks. MITM attacks describe a category of network-based attacks during which an adversary places himself between a client and a server. The adversary can then perform either passive or active attacks on the observed network traffic. Active attacks include hijacking active user session to perform malicious actions on behalf on the targeted users. Passive attacks include the collection of sensitive information such as account credentials or personal information. Proper use of certificates can prevent such attacks. If applications do not protect the communication between mobile devices and their backend servers, attackers can easily perform active/passive MITM attacks by e.g. monitoring users on public Wi-Fi hotspots. Our particular threat model focuses on applications that aim to protect their users with a secure communication channel (SSL/TLS), but fail to implement this protocol properly. In particular, our threat model accounts for the following network security challenges:

- **Broken Certificate Verification:** If an Android application uses certificates issued from one of those certificate authorities, which are shipped in with the device, it relies on the standard implementation of the SSL/TLS protocol and provides therefore basic security. The problem arises with the use of self-signed certificates. In this case, the developers should implement proper validation procedures in order to secure the connection. These custom implementations tend to leave the application insecure by implementing a broken certificate validation. The issue of a broken custom certificate validation remains a major problem of current Android applications and leaves applications as vulnerable to MITM attacks as if no encryption was used at all.
- **Compromised Certification Authorities:** Even in cases in which developers rely on certificates issued by trusted

certification authorities and the default verification mechanisms of Android, powerful adversaries might still perform MITM attacks. The Diginotar case [9] clearly showed the risk of trusted certificate authorities (CAs) being compromised by adversaries. If an attacker is able to compromise one of the 150 CAs trusted by Android, he can perform MITM attacks on applications with the standard SSL/TLS protection.

To account for the attack vectors of our threat model, we propose a solution to dynamically pin application certificates. Hereby, we rely on the *Trust on First Use (TOFU)* principle. Since we do not have any previous information about the certificate that is going to be pinned, we use this approach to get the first certificate that the application will receive during the establishing of the SSL/TLS connection. Therefore, our threat model assumes that the first connection between a given mobile application and their corresponding servers is not compromised. Based on our threat model we aim to overcome the following challenges regarding the network security of Android applications:

- **Dynamically upgrade applications to use certificate pinning.** If the implementation of the pinning is not correct, the user faces grave security and privacy consequences. Users become an easy target for adversaries to steal sensitive information such as banking credentials, social security numbers, etc. We attribute the lack of proper SSL/TLS implementations to a knowledge gap of the developers. Previous research showed e.g. that the vast majority of developers are not familiar with the concept of certificate pinning [31]. Most of the time they are guided by random forum posts and discussions on popular online forums such as stackoverflow.[3] It so happens that a number of posts related to the use of SSL/TLS on Android actually advise developers to handle the SSL/TLS errors by accepting all certificates. By doing this, they actually remove any security on the network level, even the (secure) default setting, because most of the posted solutions suggest to use custom implementations of the *TrustManager* [18] which overrides the default one. Therefore, we aim at overcoming this knowledge gap by proposing a solution that focuses on the affected users instead of developers.
- **Providing the users with detailed information for every certificate change.** In order not to significantly lower the usability of the Android applications by immediately terminating the connection when a certificate change occurs, we have provided the users with a notification containing detailed description for the change that just occurred. At this point, users are left with the possibility to accept this change and continue to use the application, or to reject it, which would imply that the connection will be terminated immediately.

[3]http://stackoverflow.com

III. Methodology

A. Number of vulnerable applications

Generally speaking, Android can be divided into two main parts: The first part is the Android operating system, and the second one are the Android applications. We do not focus on the overall security of the Android operating system, but rather on applications and the not-so-obvious threats presented by them. Nowadays, there are more than 1.5 million available applications in the official Android market place [13]. Taking in consideration the popularity of the Android OS, we firstly analyzed the top 50,000 applications from all categories over two consecutive years. This allowed us to determine to what extend SSL/TLS issues are present in these two sets of applications. These analyses are focusing directly on the correct implementation of the HTTPS protocol, namely the *TrustManager*. The cases in which the applications are using pure HTTP are not taken into consideration and are immediately classified as applications that do not have any issues. To perform our analysis, we rely on the *Mallodroid* tool which explicitly targets the implementation of the *TrustManager*. In detail, we performed an indicative experiment in order to determine if the SSL/TLS errors still exist. Detection of other network flaws or tracking the evolution of particular applications is out of the scope of this work.

The applications were already crawled by *Playdrone* [39] and are publicly available at archive.org. We selected the top 25,000 applications from late 2013 and resp. from late 2014. According to [39], crawled applications originate from different categories. Finally, this experiment should help to understand if SSL/TLS implementation errors continue to put user data at risk or if the situation improved.

B. Fixing a broken trust manager

Since Android applications are packages stored across servers (market place), we do not have access to nor are we permitted to perform any changes to them. The situation is however different when the applications are downloaded and installed on a certain device, since the user has already agreed on all of the previously presented terms, known as Android permissions. In general, our approach does not interact or change the code of the application as static instrumentation would. We tackle the problem of broken SSL/TLS implementations dynamically, leaving the apk[4] intact. We are able to achieve this by leveraging the functionality of the Cydia Substrate framework [35]. We chose to use Cydia because of two reasons:

- It is available for other smartphone operating systems which increases the chances of a widespread use of our approach.
- It is the only framework that supports the Android permission model. This means that even though it requires ROOT access, its use must be explicitly specified in the Android manifest file.

[4]Android application package

The overall goal of our approach is to evaluate a proof-of-concept solution of our dynamic approach to fix the SSL/TLS issue from the users' perspective.

IV. Design

Our proof-of-concept implementation is based on the dynamic instrumentation of mobile applications [19], [35]. These dynamic instrumentation frameworks are especially popular among the users of custom ROMs such as *CyanogenMod* [4]. Today, the most common use of these frameworks consists in the creation of customized widgets and other GUI elements. The only requirement that has to be met for proper use of these frameworks is the available ROOT access to the device. This unleashes the full power of the frameworks, expressed through the possibility of interception, hooking and modification of functions, system calls and class loading, interpreted both through Java and native code. The fact that they operate at runtime enables us to intercept and modify all networking calls. Furthermore, the wide use of these frameworks in communities that rely on custom ROMs renders our tool as a promising candidate for securing the network communication. Recent statistics [5] [3] show that there are currently more than 50 million people using CyanogenMod on their smartphones. The vast majority of those users are strongly focused on privacy and security enhancements, especially after the Edward Snowden revelations.

In our previous research [22] we identified the most suitable candidates to achieve our goal. We thus decided to use the Cydia Substrate framework [6], [35] for our dynamic approach. We chose this framework because it is the only framework that is available for the two leading smartphone operating systems – Android and iOS. In contrast to previous research – which is mostly focused on the developers – our focus group are the users. Furthermore, our solution presents itself as an OS extension, so it could be easily included as a factory feature. Cydia Substrate (formerly known as Mobile Substrate) serves as a base for the development of particular tools/modules. It provides a set of different APIs which can be adapted according to the specific needs. In general, these APIs make it possible to get a reference to a particular class of the applications with the *MSJavaHookClassLoad* and then search inside that class for the desired method or function that should be hooked with the *MSJavaHookMethod*. The last step would be to replace the code of the hooked function or method with our custom implementation. This procedure could be easily applied to all generic calls, however, in some cases static instrumentation of the code might be needed in order to detect the hooking point. Our implementation consists of two classes, one for the implementation of the hooking functionality, and the other for the pinning trust manager. Since SSL/TLS implementations in Android apps are dependent on certain API calls that are provided in the tutorial itself [20], we do not have to perform any static instrumentation to distinguish these calls. Instead, we can directly interact with those API calls. Using the previously mentioned *MSJavaHookClassLoad* function, we wait for the *javax.net.ssl.TrustManagerFactory* to load and search for the *getTrustManagers* method to be hooked. Upon hooking, the current implementation of the *getTrustManagers* is overridden by our custom implementation. This set of instructions directly influences the current implementation by substituting it with our version. Furthermore, additional changes have to be made for the application to work properly. We then override the *setSSLSocketFactory* method upon loading of the *javax.net.ssl.HttpsURLConnection* and setting it to use our implementation of the trust manager. Last but not least, we override the *init* method from the *javax.net.ssl.SSLContext* class to use our *TrustManager*. This way we ensure that every established connection will be pinned. Although there are different ways to verify the hostname, we are using strict verification. This means that every pin is associated with its designated host. By using the *TOFU* principle, we pin every connection upon the first encounter. Therefore, every pin is associated with the designated host, and when the connection is trying to be established later on, the hostname along with the pin for that particular connection is checked. Whenever there is a mismatch in any of the fields, whether it would be the hostname or the pin itself, a notification will alert the user immediately. Instead of terminating the connection if the certificate changes, which can happen without any malicious intent, we enable the user to decide whether to approve the change and pin the new certificate or to reject the change and terminate the connection. The users are included in this process, because if we directly terminate the connection, it will render the applications unusable.

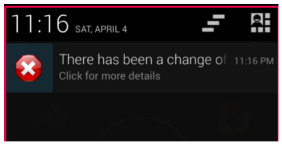
V. Results

We conducted a static analysis of 50,000 Android applications. The static analysis is solely focused on the *TrustManager* implementation within the applications. The applications dated from two consecutive years. One set of top 25,000 applications was crawled in late 2013 and the second set in late 2014. For our analysis we used the *Mallodroid* script [12] and the results are categorized according to the following criteria: *Broken TrustManager*, *Possibly Broken TrustManager*, *Broken hostnameVerifier*, *Possibly Broken hostnameVerifier*, *Broken SSLError Handling* and *Possibly Broken SSLError Handling*. The results confirmed that the applications rely more and more on network communication. The results are presented in Table I and Table II.

	Trust Manager	Hostname Verifier	SSL Error
Broken	17%	7%	0.08%
Possibly Broken	6%	1%	15%
No issues	54%		

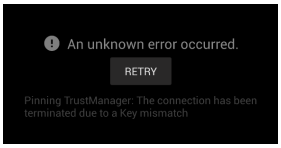
TABLE I: Classification of applications that contain Broken and Possibly broken TrustManager, Hostname Verifier and SSL Errors for the set of 25,000 applications from late 2013

It is evident that the situation is just getting worse. The top 25,000 applications from late 2013 contained 3,834 applications or nearly 17% that had no implementation or a broken custom implementation of the validation procedure.

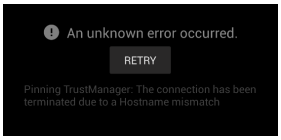


(a) Screenshot from the notification in the Notification Bar

(b) Screenshot from the body of the notification showing the change of the certificate that occurred



(c) Example of terminated network connection due to key mismatch



(d) Example of terminated network connection due to hostname mismatch

Fig. 1: Screenshots of the notification and the error messages produced by a mismatch of the public keys and the hostnames – simulation of *MITM* attack

	Trust Manager	Hostname Verifier	SSL Error
Broken	23%	13%	0.05%
Possibly Broken	10%	4%	29%
No issues	21%		

TABLE II: Classification of applications that contain Broken and Possibly broken TrustManager, Hostname Verifier and SSL Errors for the set of 25,000 applications from late 2014

```
public TrustManager(){
  return;
  }
public void
    checkServerTrusted(java.security.cert.
X509Certificate[]s1, String s2){
  return;
  }
```

Listing 1: Example code for Broken TrustManager that would accept all certificates

Usually, such a set of applications also contains the other two categories, i.e. broken hostname and no handling of SSL/TLS errors. Rather surprisingly for us, the applications from 2014 turned out to have more broken SSL/TLS implementations. As seen in Table II, 23% resp. 4,804 applications have a broken SSL/TLS implementation or are set to accept every certificate that is presented to them. This increase of nearly 6% is a clear indication that the problem still exists among the Android applications that make use of the SSL/TLS protocols for securing their network communication. Although it is evident that the problem still exists, moreover we notice an increase in the applications that contain broken SSL/TLS implementation from the set of 2014, it could be an indication that the awareness towards this issue has finally raised. This is because of the functionality of the *Mallodroid* script which targets just the applications that are using SSL/TLS, specifically the *TrustManager* implementation, while the applications that use just HTTP are immediately classified as applications that contain no issues. Due to the design of this script, additional network flaws are also not registered. This indicates that all of the applications that use just HTTP or do not use Internet are classified under the *No Issues* category. Moreover, this increase could be classified as mixture of fast adoption of the concepts providing additional SSL/TLS security combined with the lack of knowledge with regard to the actual implementation of these concepts. The lack of a centralized body (such as Google Play is for testing the applications regarding all additional threats) that is capable of testing the actual implementation of the networking part of the application could easily introduce such increase in the results, since Google itself has a quite open approach towards the process of becoming a developer without assessing their actual qualifications. This implies that even with increased awareness, there is no guarantee that the implementation will be correct. Finally, we handpicked a very small set of already identified applications with broken SSL/TLS implementation for further analysis. In this set of apps, we encountered classes named *FakeTrustManager*, *AcceptAllTrust* etc. and found copied chunks of code directly from the forums that advice users to trust all certificates in order to solve the SSL/TLS errors in their applications. An example of a detected broken implementation is outlined in Listing 1.

After having presented the design of our tool, we assess its effectiveness. In order to be able to test our tool, we did a setup that includes Android devices with root access, Cydia Substrate installed and our tool, which in turn was installed

Fig. 2: Screenshot from the file that contains the public key pins and the hostnames

as an extension to the framework. As specified by the Cydia Substrate framework, after installation of a new module the phone has to be rebooted. From the point when the phone is booted, the public key of every certificate for every connection is automatically pinned. Figure 2 shows the pinned public keys.

Our implementation of the pinning is based on the suggestions from the OWASP guide [33]. Although it might require frequent updates, we decided to pin the end certificate in order to reduce the attack surface as much as possible. By pinning the public key of the certificate, we introduced the user with a little bit more flexibility compared to pinning the whole certificate. This means that the applications that use our approach will eliminate the need for an update when the certificate is reissued, since the public key will be the same.

We manually verified our solution against a number of applications that had a completely broken use of SSL/TLS protocols. Upon applying our solution, all of the tested applications were functioning as expected and their public keys were successfully pinned. In order to simulate a MITM attack, we manually changed the pinned keys in the file. This means that next time when the application tries to establish a network connection, our solution should send an alert to the user. After successfully changing the key and reopening the application, we received an alert that the key had been changed (shown in Figure 1a and Figure 1b). Here the user can decide whether he accepts this change and pin the new certificate or just reject it. If the user pins the new key, the application will continue to work normally, since in our case the certificate that is received is the valid one from the application, because we performed the change directly in the file that contain the pins. In general, if the user rejects the change of the certificate, the connection will be immediately terminated. An example of this case can be seen in Figure 1c and Figure 1d. Furthermore, the tests performed against the set of applications showed no decrease in the overall performance of the applications.

VI. Discussion

Our approach improves the network security of broken/default SSL implementations in Android applications by dynamically pinning server certificates. Hereby, we directly improve the security of affected users instead of focusing on the developers of the applications. We also managed to shift the pinning strategy from an application-based approach to a broader device-based solution. Furthermore, we do not have any limitations regarding the number of applications, because our approach works with all Android applications that are using network communication.

A. Security Challenges

Besides all the benefits presented throughout this paper, our current proof-of-concept implementation has open challenges. Since we are relying on the *TOFU* principle, our tool works only if the first connection is benign. Although there is still a risk that the first connection might be malicious, it is definitely significantly lower in comparison to the risk of MITM attacks on a totally insecure application. Furthermore, our approach could be easily adapted to overcome the TOFU issue in future work. As proposed by Wendlandt [41], we plan to implement a third-party notary service to ensure that applications use the correct pin, even when the first connection has already been tampered with. This solution enables the possibility to provide pins in advance and, more importantly, offers a distributed infrastructure to detect MITM attacks. By implementing such an extension, the application will directly receive the pins while the risk of a first malicious connection is eliminated, because the provided pins are verified against our third-party notary service. Furthermore, it also overcomes the limitation presented by the certificate updates, because new certificates will be verified and the user will be accordingly informed whether the certificates are malicious. An other alternative approach to overcome the limitation presented with the use of the *TOFU* approach is to use the DNS-based Authentication of Named Entities *(DANE)* [7]. This alternative would, however, also require the support of DANE by applications developers.

Our approach requires root privileges to work. Therefor, an attacker would also need root privileges to subvert our security improvements by e.g. tampering with the pin storage of our tool. In scenarios where attackers are able to gain root access to a device, they can already access any information stored on the device directly and would likely not focus on subverting our protection mechanisms. It is also important to state that our approach is not directly exposing the system to any additional risks apart from the ones already presented with the use of the root mode in Android operating system. Finally, to address the cases in which applications already have a correct implementation of the SSL/TLS protocols, we will introduce application whitelisting. This means that applications with a correct implementation of SSL/TLS could be excluded and not obligated to use our implementation of the *TrustManager*.

B. Usability Challenges

Our proof-of-concept implementation requires user interaction and can thus be compared with the current implementation of common web browser warnings, during which users are explicitly asked to decide whether to proceed or terminate the connection to the desired web service. Due to the specific nature of real-world usability testing of our module, a large-scale long-term usability test would provide additional insights regarding the user acceptance of our approach. In addition, an adaption of the previously mentioned notary-based or DANE-based pinning approach would also minimize the requirement for users to decide if a given certificate is valid or not. Our work touches upon another important issue related to secure network communication that is still present in the Android OS: the lack of visual indicators. Unlike browsers, where the user is notified with the lock in the address bar when HTTPS is used, in Android there is no way for the user to distinguish whether the user is using a secure channel to transmit sensitive data or not. Our approach could therefor be used to inform the user when secure network communication is used for a specific app. Finally, we plan to make our approach even more accessible by porting our proof-of-concept implementation to iOS.

VII. Related Work

Taking into consideration the market share of Android, it is obvious that any vulnerability would affect a large number of users. From the start, researchers put a lot of effort in discovering bugs and proposing solutions. It is the same with the networking part of the applications that are currently on the official market. Trummer et al. [38] and Onwuzurike et al. [32] recently showed that some of the most popular applications currently available are still vulnerable to MITM attacks. This underlines that the problem still exists and according to [27], lack of knowledge is one of the reasons for this issue. While not being able to directly influence this matter, researchers turn to proposing tools for static analysis that could help developers and researchers to detect broken SSL/TLS implementations. Sounthiraraj et al. [36] proposed SMV-Hunter, a tool that combines static and dynamic analysis to detect incorrect use of SSL/TLS protocols. Zuo et al. [42] presented a hybrid approach to discover these vulnerabilities. They analyzed 13,820 applications and found out that 1,360 are potentially vulnerable. The drawback of all those solutions is that they are focused on the developers. In contrary, our approach is aiming at a more scalable solution: we developed a module for dynamic certificate pinning which scales the common application-based approach to a broader device-based and user-focused solution. Instead of limiting the certificate pinning to just one app, our module is able to implement this approach for every single application installed on the device. Furthermore, all of the past research is performed over a fixed set of applications, whereas our focus is put on the users, i.e. without having a limitation for the application set. This means that we are able to apply our solution to any application that is utilizing the network. This way we directly influence the user instead of the developers.

Network security is just a part of the whole Android security model, therefore we refer the interested user to Enck et al. [25] for a detailed explanation of the Android operating system as well as its overall security concept. Currently a number of researchers focus on discovering applicable attack vectors for the Android operating system. Bugiel et al. [21] and Davi et al. [23] present privilege escalation attack vectors that underline weaknesses in the Android operating system. Finally, the Android permission has received considerable attention from the research community. Information regarding the evolution and effectiveness of the permission system as well as detailed studies regarding over-privileged applications can be found in [34], [40].

VIII. Conclusion

In this paper we discuss a major security and privacy issue of Android applications: weak protection of the network communication between devices and backend servers. To this end we performed a static analysis on 50,000 Android applications gathered over a period of two years. Our analysis showed that the broken implementation of SSL/TLS communication remains a serious issue for popular Android applications. Our analysis suggests that this issue did not improve over time. We furthermore present a novel approach to overcome the issue of broken SSL/TLS use in Android applications. Hereby, we proposed a tool that provides dynamic pinning of certificates during runtime. To the best of our knowledge, we are the first to tackle this major security challenge from the users' perspective. Our approach is based on a popular dynamic instrumentation framework which is available for the great majority of Android devices and, thus, makes our proposed implementation a suitable candidate for future custom ROMs. Therefore, we made the source code of our proof-of-concept implementation publicly available [8] to spur adaption of our approach in popular custom Android ROMs such as CyanogenMod.

Acknowledgment

This research was funded by COMET K1, FFG – Austrian Research Promotion Agency. Moreover, this work has been carried out within the scope of "u'smile", the Josef Ressel Center for User-Friendly Secure Mobile Environments, funded by the Christian Doppler Gesellschaft, A1 Telekom Austria AG, Drei-Banken-EDV GmbH, LG Nexera Business Solutions AG, NXP Semiconductors Austria GmbH, and Österreichische Staatsdruckerei GmbH.

References

[1] Android sdk. [Online]. Available: http://developer.android.com/sdk/index.html
[2] Certificate authority. [Online]. Available: https://en.wikipedia.org/wiki/Certificate_authority
[3] Cyanogen usage statistics. [Online]. Available: http://www.androidcentral.com/cyanogen-now-has-more-users-windows-mobile-and-blackberry-combined
[4] Cyanogenmod. [Online]. Available: http://www.cyanogenmod.org
[5] Cyanogenmod statistics. [Online]. Available: http://www.digitaltrends.com/mobile/does-cyanogen-really-have-more-users-than-windows-mobile-and-blackberry-combined/

[6] Cydia substrate apk. [Online]. Available: https://play.google.com/store/apps/details?id=com.saurik.substrate

[7] Dns-based authentication of named entities (dane). [Online]. Available: https://tools.ietf.org/html/rfc6698

[8] "Dynamic pinning solution." [Online]. Available: https://github.com/dbuhov/pinningTrustManager

[9] Final report on diginotar hack shows total compromise of ca servers. [Online]. Available: https://threatpost.com/final-report-diginotar-hack-shows-total-compromise-ca-servers-103112/77170/

[10] M. marlinspike. tack - trust assertions for certificate keys. [Online]. Available: http://tack.io/draft.html

[11] M. marlinspike. your app shouldn't suffer ssl's problems. [Online]. Available: http://www.thoughtcrime.org/blog/authenticity-is-broken-in-ssl-but-your-app-ha/

[12] Mallodroid script - https://github.com/sfahl/mallodroid. [Online]. Available: https://github.com/sfahl/mallodroid

[13] Number of available applications in the google play store. [Online]. Available: http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[14] Representation and verification of domain-based application service identity within internet public key infrastructure using x.509 (pkix) certificates in the context of transport layer security (tls). [Online]. Available: https://tools.ietf.org/html/rfc6125

[15] Smartphone os statistics 2015. [Online]. Available: https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1&qpsp=2015&qpnp=1&qptimeframe=Y

[16] The transport layer security (tls) protocol version 1.2. [Online]. Available: https://tools.ietf.org/html/rfc5246

[17] Usage of ssl certificate authorities. [Online]. Available: http://w3techs.com/technologies/overview/ssl_certificate/all

[18] X509trustmanager. [Online]. Available: http://developer.android.com/reference/javax/net/ssl/X509TrustManager.html

[19] Xposed framework. [Online]. Available: http://repo.xposed.info

[20] Android. Security with https and ssl. [Online]. Available: http://developer.android.com/training/articles/security-ssl.html

[21] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. [Online]. Available: http://www.internetsociety.org/towards-taming-privilege-escalation-attacks-android

[22] D. Buhov, M. Huber, G. Merzdovnik, E. Weippl, and V. Dimitrova, "Network security challenges in android applications," in *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, Aug 2015, pp. 327–332.

[23] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Proceedings of the 13th International Conference on Information Security*, ser. ISC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 346–360. [Online]. Available: http://dl.acm.org/citation.cfm?id=1949317.1949356

[24] N. Elenkov. Certificate pinning in android 4.2. [Online]. Available: https://github.com/nelenkov/cert-pinner

[25] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *Security Privacy, IEEE*, vol. 7, no. 1, pp. 50–57, Jan 2009.

[26] 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 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 50–61. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382205

[27] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking ssl development in an appified world," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516655

[28] 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*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382204

[29] M. Marlinspike. Android pinning. [Online]. Available: https://github.com/moxie0/AndroidPinning

[30] J.-S. Oh, M.-W. Park, and T.-M. Chung, *The Multi-level Security for the Android OS*, B. Murgante, S. Misra, A. Rocha, C. Torre, J. Rocha, M. Falcão, D. Taniar, B. Apduhan, and O. Gervasi, Eds. Springer International Publishing, 2014, vol. 8582.

[31] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl, "To pin or not to pin—helping app developers bullet proof their tls connections," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 239–254. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/oltrogge

[32] L. Onwuzurike and E. De Cristofaro, "Danger is my middle name: Experimenting with ssl vulnerabilities in android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '15. New York, NY, USA: ACM, 2015, pp. 15:1–15:6. [Online]. Available: http://doi.acm.org/10.1145/2766498.2766522

[33] OWASP. Certificate and public key pinning. [Online]. Available: https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning

[34] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: A perspective combining risks and benefits," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '12. New York, NY, USA: ACM, 2012, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/2295136.2295141

[35] Saurik. Cydia substrate. [Online]. Available: http://www.cydiasubstrate.com

[36] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *Proceedings of the 19th Network and Distributed System Security Symposium*, 2014.

[37] V. Tendulkar and W. Enck, "An application package configuration approach to mitigating android SSL vulnerabilities," *CoRR*, vol. abs/1410.7745, 2014. [Online]. Available: http://arxiv.org/abs/1410.7745

[38] T. Trummer and T. Dalvi. The savage curtain: Mobile ssl failures, black hat - https://www.blackhat.com/docs/ldn-15/materials/london-15-trummer-dalvi-the-savage-curtain-mobile-ssl-failures-wp.pdf.

[39] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," *SIGMETRICS Perform. Eval. Rev.*, vol. 42, no. 1, pp. 221–233, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2637364.2592003

[40] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/2420950.2420956

[41] D. Wendlandt, D. G. Andersen, and A. Perrig, "Perspectives: Improving ssh-style host authentication with multi-path probing," in *USENIX 2008 Annual Technical Conference*, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 321–334. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404041

[42] C. Zuo, J. Wu, and S. Guo, "Automatically detecting ssl error-handling vulnerabilities in hybrid mobile web apps," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: ACM, 2015, pp. 591–596. [Online]. Available: http://doi.acm.org/10.1145/2714576.2714583