Exploring TLS Certificate Verification in Mobile Applications: Insights From and Beyond "Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications"

Marija Jovanovikj
Saarland University
Saarbrücken, Germany
majo00014@stud.uni-saarland.de

***Abstract*—Nowadays most mobile applications make use of Internet services to perform their functionalities. As such, establishing secure connections with the help of Transport Layer Security (TLS) is crucial, but not sufficient. The implementation of proper certificate validation logic is just as important and it is performed at the client side. Writing custom code, especially for implementing important security features such as certificate validation is usually not recommended, as it has been proven to be error-prone leading to open vulnerabilities. As a result, companies shift towards less code-writing solutions. For example, Google introduced *Network Security Configuration (NSC)*, an Extensible Markup Language (XML) configuration file that only requires configuring security parameters, instead of writing full code from scratch. Apple introduced the *App Transport Security (ATS)* which allows for similar network related configurations. While these solutions sound ideal on paper, we explore how they work in reality and the behavior of developers towards them. Additionally, Google implemented safeguards, which scan the applications before being published to find potential insecurities, including insecure certificate validation logic. We also show that these safeguards are not as efficient as expected and there is more room for improvement in this area.**

I. Introduction

As mobile applications started to advance over time, many of them started to make use of the Internet in order to successfully perform their functions. Internet connectivity became a necessity but it also increased the security risk and overall attack surface. As a result, the whole work and responsibility for establishing remote secure connections falls on the developer's shoulders. TLS is the de-facto standard today for encrypting communication channels between clients and servers on the Internet. However, the security guarantees of the protocol are in itself not enough. It is the client's job to perform correct TLS certificate verification. In other words, if the client side does not incorporate TLS correctly, then the mobile application will essentially be vulnerable to **Man In the Middle (MITM)** attacks.

Developing a mobile application is an iterative and time-consuming process. It often involves several phases, starting from gathering requirements and initial design, all the way to the actual hands-on coding and user testing. The end goal is to have a functional product. Often times, the application's functionality comes at an expense of its security and vice versa. This report addresses one of the major ongoing problems the developers of mobile applications persistently face, which is the implementation of proper **TLS certificate validation** for effective prevention of passive and active MITM attacks.

Poorly written and insufficient documentations, copy and pasting code from untrustworthy sources, and lack of support in the IDEs are one of the main reasons for this ongoing problem. Developers are given no other choice, but to interpret things on their own by writing their own custom code, potentially making mistakes, which become the root cause for publishing vulnerable applications.

Naturally, the question that may arise is what can be done to make the security implementation smoother and reduce the number of mistakes from the developers' side. From the discussion above, two key areas to work on can be detected: **1) Make the implementation of TLS certificate validation more usable** and **2) Have a mechanism to detect applications with vulnerable TLS certificate validation logic and prevent them from being published**.

In this report we focus on Android as it is the leading mobile market and we take a look at Google's countermeasures against such MITM attacks, their effectiveness, usability, and to what extent developers implement them properly for getting the best outcomes. The aim is to use the discoveries obtained from this report to help improve the efficacy of the introduced countermeasures and after that help developers to incorporate the solutions correctly in their mobile applications.

II. Related Work

In this section we list several papers that also address the importance of correctly implemented certificate validation logic in different Android applications.

In 2018, Alghamdi et al. [1] came up with an automated tool for verifying TLS X.509 certificate validations of *Internet of Things (IoT)* messaging protocols, specifically the **Message Queuing Telemetry Transport (MQTT)** broker-based protocol. Their results showed that 33.3% of the examined applications are vulnerable to MITM attacks.

In 2020, Wang et al. [2] designed a tool called **DCDroid** to detect certificate validation vulnerabilities using static and

dynamic analysis. With the static analysis they focus on locating potential vulnerable code in the applications, whereas with the dynamic analysis they focus on triggering User Interface (UI) components to confirm the vulnerabilities. They analyzed 2,213 applications from Google Play and 360app. Results showed that 457 (20.65%) of the applications had potential certificate validation vulnerability. After running DCDroid they confirmed that 245 (11.07%) of the total applications are vulnerable to MITM attacks.

In 2022, Pradeep et al. [3] analyzed 5,079 unique applications from the two official application stores (Android and iOS) and performed static and dynamic analysis to detect usage of **certificate pinning**. They found that the usage has increased four times compared to the previous studies. They found that 0.9% to 8% of Android applications and 2.5% to 11% of iOS applications use certificate pinning at run time. This is a big improvement compared to the previous 0.67% Android applications detected by Oltrogge et al. [4].

In 2024, Pourali et al. [5] found out that most Android applications use third-party libraries which may contain code for TLS validation. They designed an automated dynamic analysis tool named **Marvin** to identify TLS validation vulnerabilities and validation hijacking. Among 7,826 applications from Chinese application store and Google Play, 55.3% of the Chinese applications and 6.4% of the Google Play applications had insecure TLS certificate validation logic. Most of the vulnerabilities were due to third-party libraries used by the applications and not due to the developers' application code.

In 2024, Moon et al. [6] performed static analysis on 25 miscellaneous Android applications. They have identified various improper TLS implementations such as *insufficient verification of the TLS certificate chain, invalid TLS certificates, insufficient checking of the TLS certificate validity, and insecure WebViews usage*.

In 2025, Bandara et al. [7] analyzed the impact of **Original Equipment Manufacturer (OEM) customizations** on the Android TLS protocol stack, which may negatively impact the security of the applications. They found out that 20k Android applications from Google Play have TLS methods removed or tampered by OEMs. Over 75% of the applications use at least one of the removed functions, downgrading the security.

As we can see, applications with vulnerable certificate validation logic are still being published. Researchers mostly work on coming up with *tools* for better detection of application vulnerabilities, including certificate validation vulnerabilities. This shows the importance of successful on-time vulnerability detection mechanisms and prevention from publishing, apart from the introduction of easier configuration methods like NSC. We can recommend such tools to be integrated by Google Play to significantly improve their safeguards.

III. TLS Foundations and History

TLS which stands for *Transport Layer Security* is considered as the worldwide standard for encrypting communication channels between clients and servers on the Internet. The protocol consists of two ordered subprotocols: 1) **Handshake protocol** and 2) **Record protocol**. First, the handshake protocol is executed, and with that several actions take place, such as negotiating the version of the protocol and the set of cryptographic algorithms that will be used. These negotiations are crucial to ensure interoperability between different implementations and TLS versions. Furthermore, during the handshake phase, *authentication* of one or both parties is involved. Usually, it is the client that authenticates the server and very rarely does the server authenticate the client. Authentication is performed using *digital certificates* and they are crucial for learning the public keys and verifying identities. It is important to note that the digital certificates are issued and signed by trusted *certificate authorities (CAs)*, which allows the parties to confirm that the presented public keys indeed belong to the parties that claim to own them. The last part of the handshake phase, which comes after the authentication action is the establishment of a shared secret, which will be used for securing all further communication for the given session. The client creates a session key for the given communication session and encrypts it using the server's public key, which the client already obtained with the authentication step from the digitial certificate. Once the server receives the encrypted secret, it can decrypt it using its private key and from that moment both parties will have obtained the secret session key. From this moment, the record protocol can begin which is responsible for encrypting and decrypting data between both parties, as well as validating the integrity of the data. Overall, TLS ensures *confidentiality*, *integrity*, and *authenticity*. It can further prevent *active* and *passive* MITM attacks. Despite all that, what must be taken as a fact is that *the security guarantees of TLS solely depend on the correct certificate verification by the client* [8].

A. TLS Implementations in Android - How It Initially Started

In the beginning, Android developers had the option to implement TLS using Android's native TLS libraries or other third-party TLS libraries. Additionally, Android gave the developers an option to implement TLS using their own proprietary TLS libraries. One key problem to note here is that many Android devices are running outdated versions of Android with equally outdated TLS APIs. That in return poses a direct risk on applications that make use of default OS provided TLS libraries. Regardless of how new the application is, it may be left with no other choice but to use old broken cipher suites and protocol versions which makes it an easy target for well-known vulnerabilities. One way to prevent this is to use a pre-built TLS library with the application package for better control over the TLS configurations. However, using TLS sockets is of no guarantee that the connection will be secure. It is important for mobile developers to properly configure their TLS connection parameters. Once the client receives the server's certificate, it is crucial for the developer to perform correct verification. That includes correctly verifying whether the hostname in the server's certificate matches the hostname of the server that the client is trying to connect to (implemented via an interface called HostnameVerifier), verifying the server's certificate chain, ensuring the certificate is issued by a trusted CA, performing checks on certificate expiry and revocation (implemented via an interface called TrustManager). As Android gives the developers the freedom to customize their code, if at least one step is incorrectly or insufficiently implemented by the developer, the application

will become vulnerable to MITM attacks. For example, developers can customize certain interfaces such as **TrustManager** and **HostnameVerifier**, instead of using the system-defined defaults. Creating a custom TrustManager means the developer has the freedom to opt to trust all certificates, including those that are expired, self-signed or malicious, making the application vulnerable to MITM attacks. On the other hand, a custom HostnameVerifier can perform insufficient hostname verification of the server with which the application is communicating. This can allow an attacker to impersonate a legitimate server and trick the application into sending sensitive data to the attacker. The vulnerable HostnameVerifier implementation can skip the hostname validation and instead only verify the hash of the certificate. As we can see, there are many parameters to be taken into consideration when implementing TLS in mobile applications, thus customization can actually bring more harm than good. In such cases, it is a better practice to just stick to using the provided safe defaults [9].

Many modern operating systems nowadays come with a pre-installed list of certificates belonging to trusted root CAs. The number of entries in the list for Android depends on the Android version and device. Android owners can find their exact number by navigating to the Trusted Credentials section in settings. In general, the list easily includes *over 100 trusted root CAs*. Apart from the pre-installed list, users and developers have the freedom to use their own custom CAs and include them in the trusted list. From a developer's point of view, working with a *custom CA* meant creating custom certificate validation logic through Android APIs. While having an option for customization is often considered a benefit, it can also bring a lot of drawbacks. Writing custom code significantly increases the chance of error-prone implementations leading to the creation of vulnerabilities which often times go unnoticed. Additionally, working with the Android APIs brings limitations in a sense that there may be no APIs that perform certain necessary actions. As an example taken from [4, Background on TLS and Android, p. 2], before Android 4.2, there existed no APIs that applications could use to ask for the certificate chain validated by the system. In other words, there was no way to find out what the system validated. As a consequence, attackers could easily manipulate the certificate list presented by the server.

IV. Google's Introduced Countermeasures

In 2016 and 2017, Google introduced two countermeasures to tackle the prior MITM attack vulnerabilities that came with the TLS libraries and overall code customization. The two novelties introduced were the *Network Security Configuration (NSC)* and the *Safeguards in Google Play*.

A. NSC Overview

NSC is an XML configuration file that lets developers implement custom certificate validation logic without having to write custom code. The whole idea was to reduce the room for errors by just having to tweak certain configuration parameters in the XML file, instead of having to actually code. The file structure of the XML is shown in Figure 1, adapted from [4, Figure 1, p. 3].

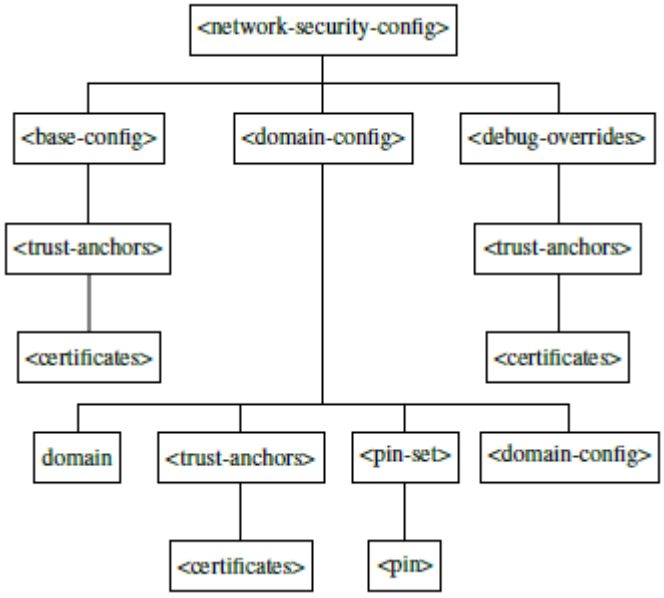

Fig. 1: NSC file structure adapted from [4, Figure 1, p. 3].

The file begins with the root tag <network-security-config> which encloses all other settings. It then branches to <base-config>, <domain-config> and <debug-overrides>.

The <**base-config**> sets default security settings for all network connections, unless overridden by a specific <domain-config> setting. Trusted certificates can be listed inside a <**trust-anchors**> tag, where we specify the sources where the application should look for trusted root CA certificates. For example, by setting the source to *"system"*, we limit our trust to just pre-installed system root CA certificates, as shown in Listing 1.

Listing 1: Trusting pre-installed system root CA certificates.

```
<base-config
cleartextTrafficPermitted="false">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

Alternatively, if the same code snippet is reused, but the source is set to *"user"*, then we allow to trust user-installed certificates from settings. We can also define to trust custom CA certificate stored somewhere in the application's folder, for example in the *res/raw* folder and setting the source to *"@raw/my_ca"*. Starting from Android 7, user-installed CA certificates are **not trusted** by default. Instead, trust is limited to pre-installed system root CA certificates. However, developers have the option to re-enable user-installed certificates and thus downgrading the secure default configurations. The *cleartextTrafficPermitted* flag enables unencrypted HTTP traffic, if set to true. Since Android 9, it is **not permitted** by default. However, as with the certificates, developers can still change the secure default setting and set the flag to allow HTTP. If no NSC file is used by the developer, setting the attribute *android:usesCleartextTraffic* in the application's

Manifest allows HTTP. If an NSC file exists, then this attribute is ignored.

Moving on to the <**domain-config**>, its usage is for specifying domain-specific policies. It is crucial to note that these policies override the base configurations for that specific domain. For example, Listing 2 shows overriding of the base configuration for the *example.com* domain and allowing HTTP traffic for it. Furthermore, this policy applies to *all subdomains* of example.com.

Listing 2: Overriding the base configuration for example.com and allowing HTTP traffic for it.

```
<domain-config
cleartextTrafficPermitted="true">
    <domain includeSubdomains="true">
    example.com
    </domain>
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</domain-config>
```

The <**pin-set**> tag implements *certificate pinning* to restrict which certificates should be accepted for a domain. A connection will be established, if at least one certificate from the server's certificate chain matches any of the registered pins. For example, Listing 3 shows how certificate pinning can be enforced, by setting the *expiration date* for the pinning enforcement, as well as the *Base64* encoded hash of the certificate's public key.

Listing 3: Enforcement of certificate pinning.

```
<pin-set expiration="2026-12-31">
    <pin digest="SHA-256">
    BASE64_ENCODED_HASH
    </pin>
</pin-set>
```

Finally, the <**debug-overrides**> tag allows the definition of different certificate trust settings while the application is in *debug mode*. For example, Listing 4 allows self-signed or user-installed CA certificates during development phase, which is not possible in production when publishing applications in Google Play.

Listing 4: Enabling trust for user-installed CA certificates during development.

```
<debug-overrides>
    <trust-anchors>
        <certificates src="user" />
    </trust-anchors>
</debug-overrides>
```

B. NSC Efficacy and Limitations - Discussion

Unfortunately, the introduction of NSC did not completely remove the risk of MITM attacks. Developers can still make errors when configuring the XML file. The errors can be due to *insufficient and unclear documentation* leading to developers' own interpretations, *copy and pasting* insecure code from external sources, *lack of proper support* for the Android Studio IDE etc. On the other hand, some errors may be done on purpose, such as making the application more functional which otherwise would be hindered due to the security measures. Such example, taken from [4, Introduction, p. 1], of an erroneous NSC setting that affected 43% of the Android ecosystem in 2019 is the **Google's official Gmail** application for Android that was vulnerable to MITM attacks due to user-installed CAs. Another problem that still pertains is that developers can still choose to work with Android APIs instead of NSC for their certificate validation logic, as in earlier Android versions. Even worse is that in some cases the custom certificate validation code *can override* the NSC settings, essentially making NSC configurations useless. For example, a vulnerable TrustManager implementation overriding the NSC certificate pinning.

C. Analysis of NSC Usage Among Developers

The research conducted by Oltrogge et al. [4] on 1,335,322 free Android applications available on Google Play, that have received an update since the introduction of NSC, showed that 99,212 of those applications implemented custom NSC configurations. The rest of the research was conducted on the remaining 96,400 applications, after all obfuscated applications were removed. A trend has been spotted that since the enforcement of HTTPS as default web protocol with the release of Android 9, the adoption of custom NSC settings greatly increased, as can be seen in Figure 2, adapted from [4, Figure 2, p. 5]. In the remaining parts of the NSC discussion, we analyze the usage of specific NSC configuration parameters and their impact on the overall application security.

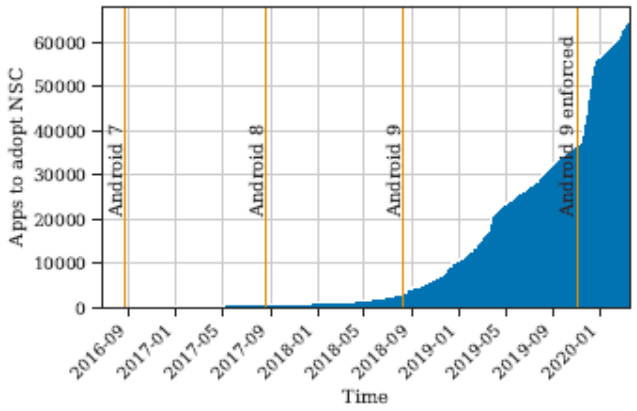

Fig. 2: Following the trend of NSC adoption adapted from [4, Figure 2, p. 5].

1) Usage of cleartext traffic

As already mentioned, the **cleartextTrafficPermitted** flag is used for enabling (if set to true) or disabling (if set to false) HTTP connections for all or only specific domains. Since Android 9, the flag is *set to false* by default. From the conducted research by Oltrogge et al. [4], there were 89,686 applications found that use the flag in their code. Surprisingly, 98.98% of them use the flag to re-enable HTTP, and only 4.56% use it to enforce HTTPS. From this, we can conclude that the majority of the applications deliberately choose to *downgrade* their

security. Concerning this, further work was conducted to check whether the domains for which HTTP was enabled, could also be accessed via HTTPS. The results showed that for 8,935 applications, HTTPS *was possible*. Despite that, developers proceeded with HTTP connections. Figure 3, adapted from [4, Appendix, Table 8], displays the domains for which downgrades were mostly performed. We can detect unusual domains such as *"127.0.0.1"* and *"localhost"*, which can hint that the developers have copied and pasted code from other resources or they have left previous debugging configurations. Similar pattern of invalid domains were detected for those with HTTPS upgrade. Another interesting point is that some applications targeting Android 9 or higher, *explicitly* set the flag to false. However, this essentially has no security benefit as it is already the default configuration, but it shines light on potential *misunderstanding* from the developer's side. Some applications that do not implement NSC can still configure cleartext traffic by using a similar flag named **"usesCleartextTraffic"**, declared in the Manifest file. If an NSC is used and it is properly configured, then the flag in the Manifest *cannot* override the NSC settings. Interestingly, among those applications that do not make use of NSC, the *usesCleartextTraffic* flag is vastly used to explicitly enforce HTTPS, which is *contrary* to the usage behavior of the cleartextTrafficPermitted flag. In general, the findings have shown that applications opt for allowing HTTP traffic for *all domains*, whereas HTTPS is enforced for *specific domains* only. Overall, allowing HTTP and relying on possible HTTPS redirects is very dangerous. *HTTP to HTTPS redirects* expose the initial HTTP request and the server's response to a MITM attack, as they are not encrypted and are susceptible to modifications. As a result, an attacker can control the redirect by *overtaking* it and serving *malicious content* instead of the originally intented benign one.

# Apps	HTTPS	Domain Value
11,689		127.0.0.1
4,290		localhost
740		10.0.2.2
449		localdev.cc
392		amazon-adsystem.com
376		virenter.com
366		10.0.3.2
366	✓	securenetsystems.net
293	✓	renweb.com
290	✓	getfitivity.com

✓ HTTPS would be possible

Fig. 3: Common domains with HTTP downgrades adapted from [4, Appendix, Table 8].

2) Usage of certificate pinning and its overall efficacy

Findings from [4] have shown that only 663 applications implement *certificate pinning* through NSC. That is only 0.67% out of all applications, a remarkably low number. The applications that make use of this feature are mostly in the *finance domain* such as banking or mobile money applications. Most of the times, the certificates that were pinned were *intermediate* CA certificates (542), followed by *leaf* certificates (483), and finally *root* CA certificates (289). The pinned CA certificates were mostly pre-installed system CAs. As a security and functional measure, Android Studio's **LINT** feature promotes the usage of *backup pins* by creating warnings if none are registered. Developers often bypass the warnings by simply using identical or invalid backup pins, as the LINT feature *does not* check for the actual correctness of the pins.

Nowadays, certificate pinning is considered as impractical in management sense and an outdated practice. More often, CAs periodically *rotate* certificates and their intermediates to improve security. When a certificate is rotated, the new certificate will no longer match the pinned certificate. Therefore, the pinned certificate needs to be *updated* to reflect the contents of the rotated certificate, otherwise the client will *reject* the new certificate and no secure connection will be established. This can cause the specific domain or overall the application to *break* and be *inaccessible* [10].

As an alternative approach to certificate pinning comes the **Certificate Transparency (CT)** concept. It was developed by Google and was launched in 2013. CT's goal is to publicly display certificates issued by CAs in **public logs**. This promotes responsibility and accountability for CAs, and any misconduct will be publicly available, hurting the CAs' reputations. CT helps in *detecting* the issuance of rogue certificates, but it does *not actually prevent* that from happening. Owners of domains should *regularly* check the public CT logs to see if a CA has issued a certificate for their domain without their consent, so that owners can further trigger *certificate revocation* [11]. As of Android 16 (API level 36), developers can opt in to *certificate transparency in NSC*, as shown in Listing 5. The default behavior is that all certificates are accepted no matter if they are logged in a CT log or not. To ensure the application connects only to destinations with certificates logged in a CT log, developers have the option to opt in CT either *globally* or on a *per-domain* basis.

Listing 5: Configuring certificate transparency in NSC.

```
<certificateTransparency enabled=
["true" | "false"]/>
```

If *enabled*, CT logs will be used to validate certificates. If the application uses custom certificate or from the user store, such certificates usually *do not exist* publicly and cannot be verified with CT. Therefore, for such cases CT verification is *disabled* by default. However, CT checks can *still be forced* by explicitly enabling CT in a <domain-config> configuration, but may not succeed if the certificate chain is not present in the CT logs [12].

3) Usage of custom certificate authorities

The usage of *custom CA* was found in 38,628 applications, by Oltrogge et al. [4]. Out of those applications, 37,562 of them used the custom CA *globally*, whereas 1,781 used for *domains*. 759 applications worked with the already pre-installed CAs and added their own custom CA certificates (30 for global purposes and 744 for domains). Some applications, specifically 123 of them *do not* accept all of the pre-installed CAs. 8.67% of the applications that target Android 7 and higher, *re-enable* trust for user-installed certificates and as such 8,001 for global purposes and 707 for domains. This poses a significant security downgrade and instead it is recommended to use such certificates only within the <debug-overrides> tag.

4) Usage of debug overrides

The number of detected applications, from the conducted study by Oltrogge et al. [4], that made use of *debug overrides* settings is 10,085. These applications are mostly popular with high download rate. Out of them, 318 applications use *custom certificates* and 170 *certificates of MITM* attack tools. The number of applications that *enable* user-installed certificates is 9,904. Even though the NSC <debug-overrides> tag provides various configurations for debugging purposes, some applications still opt to *misuse* it and set debugging configurations outside of it. For example, 41 applications set custom CA configurations so they can use certificates of MITM attack tools for debugging TLS connections. Using these certificates in production means anyone that can access the network can *intercept and decrypt* the traffic without needing to compromise the client.

D. NSC Keypoints and Suggestions

To sum up the whole discussion about NSC, we take a look at Figure 4, adapted from [4, Table 3, p. 6], which shows the different NSC settings and how their usage impacts the overall security of the application. We can conclude that setting the flag *cleartextTrafficPermitted* to true, working with *user trusted* CAs, overriding the *certificate pinning* and setting short expiration of pinning will lead to an insecure application and such usage should be avoided.

- As already mentioned, even though NSC was introduced, the alternative way of performing certificate validation via custom code is *not removed*. One may wonder why is that so? The answer is, this alternative way may be necessary for some cases, especially for **legacy reasons**. Even if we remove the possibility for writing custom code, developers can still go *levels down in the hierarchy* and customize things in the lower levels to implement the same vulnerable logic. The point to take is, developers will go out of their way to make something work the way they want to. If Google goes on to block all possibilities of customization, even in the lower levels of the hierarchy, then that will essentially become *dictatorship*.
- The usage of the cleartext traffic configurations is of a problematic nature among developers. One may wonder, why not always enforce HTTPS then? That cannot be so easily done. In some cases we *cannot use HTTPS*, such as in **Internet of Things (IoT)** devices residing in a local network. Such devices usually *do not have* certificates issued by public CAs. If they had to run HTTPS locally, it would mean they would need self-signed certificates, which are hard to manage and can downgrade the security if not properly validated. Additionally, some **libraries for advertisements** demand HTTP because making thousands of small requests over HTTPS is inefficient (TLS overhead greater than actual data). One possible solution regarding the libraries problem could be to *bundle* all those small requests into **one https request**, achieving better efficiency.
- Developers often **copy and paste code** without being aware of the potential security vulnerabilities that may arise. One way to solve this is by maintaining a *database* which will keep vulnerable sources from which developers may copy and paste code. One could perform such checks even if the code is not fully copied, but *partially* (such as a subset of it like few lines etc).
- The *correctness* of the NSC implementation depends on the *library that supports NSC* (for example OkHTTP). Even if NSC was configured correctly in the Android application, its actual enforcement depends on how the application makes network requests and what libraries it uses.
- The incorporation of **Large Language Models (LLMs)** can be beneficial. For example, a developer can obtain a custom certificate from *LetsEncrypt!* and an LLM can *fill in* the XML configurations for the developer. We should note that developers are not always malicious, they just want things to be easier to configure and be functional in the end.
- Creating detailed *safe defaults* that do not break the application's functionality can be a good countermeasure for the bigger mass. Most developers will not bother to change them. This way the application's security and functionality will both be satisfied.
- Better **documentation**, **support** in IDE, **warnings** in IDE and further **validity checks** (for example checking if a valid domain is actually entered and it is not just a URL) can help improve NSC and its correct usage.

E. Apple's App Transport Security (ATS)

Apple's App Transport Security (ATS) is a security feature that makes sure all network connections initiated by iOS applications are secured with the Transport Layer Security (TLS). It is set by *default* starting from **iOS 9.0 and macOS 10.11 SDKs or later**, and as such ATS will block connections that do not meet the minimum requirements. In cases where you want to connect to a specific server and that server is not fully secure, **exceptions** can be added to make ATS more accepting and flexible. ATS is configured via the **Info.plist (property) file** in the application. ATS exceptions can be added by creating a dictionary value for the **NSAppTransportSecurity** key in the property file. Developers need to add *justification* when adding exceptions, as they can trigger additional App Store reviews. ATS allows configurations for both global and domain-specific cases, [13]. ATS allows for enabling additional features such as *certificate transparency and certificate pinning* via defining the corresponding keys. The keys in the ATS dictionary are all optional as the default values are suitable enough for most applications, [14].

F. Google Play Safeguards Overview

Starting from 2014, Google introduced a service to the Google Play application developers called *App Security Improvement Program* (ASI). The goal of this service is to scan applications that are being uploaded for the first time to Google Play, or applications that roll out an update for

base-config	domain-config	element	attribute	secure	insecure	reason
✓	✓		cleartextTrafficPermitted	-, false	true	allows HTTP without TLS
✓	✓	<certificates>	src	-, system	user	allows user trusted CAs
			overridePins	-, false	true	disables pinning
	✓	<pin>		always		adds a pinned certificate
	✓	<pin-set>	expiration	>10 days[a]	<10 days[a]	pinning not enforced after expiration date

Fig. 4: Overview of different NSC settings and their impact on security adapted from [4, Table 3, p. 6].

potential vulnerabilities. The checks performed include checks for *insecure certificate validation logic*. If the scan indicates that a vulnerability exists, the application is *blocked* from being published and the developer gets *alerts* both via email and the Google Play Console with additional support on how to improve the application. The ASI program initially, in 2014, was used for scanning embedded AWS credentials and embedded Keystore files. As years passed, Google updated its list of vulnerabilities to check. As of 2015 and 2016, Google added to its list checks for **Webview SSLErrorHandler** (July 17, 2015), **GnuTLS** (October 13, 2015), **TrustManager** (February 17, 2016), **OpenSSL** ("logjam" and CVE-2015-3194, CVE-2014-0224) (March 31, 2016), and **Insecure Hostname Verification** (November 29, 2016) [15]. The research paper [4], aimed to test the effectiveness of these safeguards against insecure usage of **TrustManagers** (TM), **HostnameVerifiers** (HV), **WebViewClients** (WV), and insecure **third party libraries** (LB). The tests were performed by creating and publishing an application with such insecure interfaces and checking the things Google Play detected. Furthermore, code that was *reachable* is denoted with R, *hidden behind debug options* with D, and *unreachable* with U.

G. Google Play Safeguards Efficacy Against Insecure Certificate Validation

Figure 5, adapted from [4, Table 6, p. 10], shows the four different implementations that the Google Play Safeguards were tested on (TM, HV, WV, LB). All experiments were conducted by Oltrogge et al. [4].

1) Insecure TrustManager

- **Empty TrustManager**: Firstly, the researchers tested the safeguards on different variations of empty TrustManager. They tested on a TM that can be toggled with debug flag (denoted as TM-D), a TM that is always used (TM-R), and a TM whose code is not reachable (TM-U). To make things more interesting, for the TM-R implementation, the researchers changed its name to a very controversial one which is *TrustAllTrustManager*. Results show that none of these insecure implementations were blocked by Google Play. **Conclusion**: The safeguards do not check for empty TrustManagers at all.
- **Non-empty but insecure TrustManager that is always reachable**: Tested on TM that only checks for the server's certificate expiration date (TM-R-expired, TM-R-chainexpired, TM-R-selfsigned). Furthermore, tested on TM that only checks whether the server sends a certificate chain (TM-R-chain). Results show that these vulnerable implementations were also *accepted* by Google Play.

Experiment	Reachability Passed	Validation Logic
TrustManager		
TM-U	○✓	No Validation at All
TM-R	●✓	No Validation at All
TM-D	◐✓	No Validation at All
TM-R-renamed	●✓	No Validation at All, Renamed
TM-R-expired	●✓	Cert Is Not Expired
TM-R-selfsigned	●✓	Cert Is selfsigned and Not Expired
TM-R-chain	●✓	Cert Has a Chain
TM-R-chainexpired	●✓	Cert Has a Chain or Is Not Expired
HostnameVerifier		
HV-R	●✓	No Validation at All
HV-D	◐✓	No Validation at All, Debug switch
HV-R-global	●✓	No Validation at All, Used by Default
HV-R-contains	●✓	Verify Hostname Using "string.contains"
WebViewClient		
WV-R	●X	always proceed
WV-D	◐✓	always proceed, Debug switch
WV-wrapped	●✓	always proceed, Depend on invariant condition
Library		
LB-U-acra	○X	Acra with Insecure TM
LB-U-jsoup	○✓	JSoup with Insecure TM and HV
LB-U-asynchttp	○✓	async-http with insecure TM

● Always (R)eachable; ◐ Hidden Using a Debug Flag; ○ (U)nreachable
✓ App was accepted by Google Play; X App was blocked by Google Play

Fig. 5: Four categories of experiments performed for testing the efficacy of Google Play's safeguards adapted from [4, Table 6, p. 10].

2) Insecure HostnameVerifier

- **Always true HostnameVerifier**: Tested on a reachable (HV-R) and protected by Boolean debug flag (HV-D) HostnameVerifiers that accept any hostnames. Additionally, experimented with a reachable global HostnameVerifier for all TLS connections (HV-R-global). Results show that Google Play *accepts* all of them.

- **Insufficient HostnameVerifier**: Tested on a reachable (HV-R) that does not always return true but it also does not perform full hostname verification logic check. For example, instead of checking for the entire hostname, only a substring of it is checked. Results show that Google Play *accepted* this too.

3) Insecure WebViewClient

We already briefly looked at the TrustManagers and HostnameVerifiers interfaces, and now we will shortly introduce **WebViewClients**. A WebView in Android is a *UI component* that can display web content such as HTML, CSS, JavaScript inside an application, mimicking a mini web browser. A **WebViewClient** is a class that controls how a WebView loads and interacts with the web content. As such, it is important to take a look how Android applications handle TLS errors within the WebViewClient implementation. Specifically, we are interested in the *onReceivedSslError* method, whether it is customized and does not use the default secure behavior. If the application overrides it and for example calls **handler.proceed()**, essentially it bypasses the certificate validation checks, thus accepting invalid certificates as well. With the secure default, the application would either do nothing or will call **handler.cancel()** to block a page with an invalid certificate from loading, [16].

- **WebViewClient with no error handling**: Tested on reachable (WV-R) and protected by Boolean debug flag (WV-D) WebViewClients that ignore certificate validation errors and always proceed with establishing a TLS connection. At last, Google Play detected the *WV-R vulnerable* implementation and *blocked* the application from being published. However, the more complex, *WV-D* implementation successfully *passed* the checks.
- **WebViewClient with obfuscated error handling**: Tested on a WV that obfuscates all insecure error handlings. Precisely, the experiment was performed in a way that the call to the proceed method is hidden behind a Boolean expression based on an invariant check (WV-wrapped). Results show that Google Play *did not detect* this vulnerability.

4) Vulnerable Android Libraries

- **Acra 4.2.3**: Library that provides developers with application crash reports, with this specific version being rejected by Google Play (LB-U-acra).
- **JSoup 1.11.1**: Library that provides developers with HTML parsing, with claims that this specific version is being rejected by Google Play due to insecure usage of a TrustManager (LB-U-jsoup). Reproduction results show that despite such complaints from developers, Google Play *accepted* the usage of the library's version.
- **android-sync-http 1.4.9**: Library that provides developers with interfaces for HTTP connections (LB-U-asynchttp) and includes an insecure TrustManager in this specific version (1.4.9). Despite that, results show that Google Play *accepted* the usage of the library's version.

H. Google Play Safeguards Keypoints

We can conclude from the performed experiments that Google Play's checks for vulnerable certificate validation logic are **inefficient**. It blocks really basic vulnerabilities and does not detect more complex ones. This leaves room for developers to become creative in hiding their vulnerable code on purpose, which will most likely not be detected, for ensuring the application is functional and runs smoothly. However, not all developers do such things on purpose. Some make genuine mistakes and are unaware of their consequences. That is why, in both cases, it is important to catch those vulnerabilities effectively before they are published for mass usage. Since the Google Play's implemented safeguards are insufficient, researchers have proposed that it would be better if Google uses tools such as **CryptoGuard** or **LibScout** to better detect vulnerabilities in applications.

I. Google Play Improvements

- **Dynamic update of device's security provider against SSL exploits**: Google Play services now allow applications to receive dynamic updates for the device's security provider, such as fixes for TLS libraries, using the ProviderInstaller API. Prior to this, if a vulnerability was found, the patch could be applied *after an OS update*, which can take a significant amount of time. Applications can simply call the API to make sure the device is running a patched version of the TLS provider before any TLS connections are opened, all **independently** of OS updates [17].
- **Updatable root store**: Introduced with Android 14, Android's root store has become updatable via Google Play. Prior to this, the root store could only be updated with an *OS update*, which again takes time for changes to be effective. This new feature allows trusted CAs to be added or removed **outside** of full OS updates [18].

J. Apple's App Store Review

During the application review process, among the plenty of checks, applications are also checked whether they contain vulnerable settings that allow insecure network connections. Specifically, Apple looks at how developers use and configure **ATS** (what kind of exceptions they declare to customize the default settings). However, unlike Android, Apple *does not provide* an official documentation as to how these checks are performed. Researchers at *Sudo Security Group Inc*. performed bulk static analysis of application binaries from Apple's App Store. They were able to identify 76 iOS apps on the App Store vulnerable to MITM attacks due to improper certificate validation. Even though Apple pushes developers to use ATS, still ATS *does not ensure* that applications accurately verify certificates, it just ensures the developers use TLS [19].

V. Discussion and Conclusion

TLS usage by mobile applications is *not enough* to ensure the establishment of proper secure connections. The key to success is to implement *full and proper certificate validation logic* on the client side. However, it has been shown that this is a rather difficult goal to achieve. It all started from writing

custom code via APIs and progressed to more contemporary solutions such as customizing dedicated *configuration files*. Even though the focus was to shift towards simpler and less code-writing style, the adoption of such solutions by developers still *remains low*. Main reason is due to insufficient documentations and support in the IDE. Despite the introduction of the newer solutions, developers can still work with the prior ways via APIs due to *legacy reasons*. Thus, even though the dedicated configuration files exist to make developers' lives easier, they are not a sufficient mechanism to rely on for security enforcement. As a *reinforcement strategy*, it is important to scan applications for possible vulnerabilities and improper certificate validation implementations before they are actually published. However, it has been shown that both iOS and Android applications are *still being published* with such vulnerabilities. Thus, it is important to strengthen the detection mechanism and possibly integrate already well established tools that can detect such cases more effectively. Conducting measurement studies and analyses in such cases is of a crucial importance, as they can effectively reflect the real world behavior. This report aims to shine light on the efficacy of the current solutions and depict the developers' behavior and mental model in order to effectively enhance the solutions to achieve better adoption and in the end have more secure applications.

REFERENCES

[1] K. Alghamdi, A. Alqazzaz, A. Liu, and H. Ming, "Iotverif: An automated tool to verify ssl/tls certificate validation in android mqtt client applications," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 95–102. [Online]. Available: https://doi.org/10.1145/3176258.3176334

[2] Y. Wang, G. Xu, X. Liu, W. Mao, C. Si, W. Pedrycz, and W. Wang, "Identifying vulnerabilities of ssl/tls certificate verification in android apps with static and dynamic analysis," *Journal of Systems and Software*, vol. 167, p. 110609, 04 2020.

[3] A. Pradeep, M. T. Paracha, P. Bhowmick, A. Davanian, A. Razaghpanah, T. Chung, M. Lindorfer, N. Vallina-Rodriguez, D. Levin, and D. Choffnes, "A comparative analysis of certificate pinning in android & ios," in *Proceedings of the 22nd ACM Internet Measurement Conference*, ser. IMC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 605–618. [Online]. Available: https://doi.org/10.1145/3517745.3561439

[4] M. Oltrogge, N. Huaman, S. Klivan, Y. Acar, M. Backes, and S. Fahl, "Why eve and mallory still love android: Revisiting TLS (In)Security in android applications," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 4347–4364. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/oltrogge

[5] S. Pourali, X. Yu, L. Zhao, M. Mannan, and A. Youssef, "Racing for TLS certificate validation: A hijacker's guide to the android TLS galaxy," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 683–700. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/pourali

[6] I. T. Moon, A. Mimi, and M. M. Rahman Mridha, "Improper implementation of transport layer security: Impact on android applications and man-in-the-middle attack," in *2024 IEEE International Conference on Power, Electrical, Electronics and Industrial Applications (PEEIACON)*, 2024, pp. 1–6.

[7] V. Bandara, S. Pletinckx, I. Grishchenko, C. Kruegel, G. Vigna, J. Tapiador, and N. Vallina-Rodriguez, "Beneath the surface: An analysis of oem customizations on the android tls protocol stack," in *2025 IEEE 10th European Symposium on Security and Privacy (EuroSP)*, 2025, pp. 677–700.

[8] N. O. Tippenhauer, "Tls," Security Core Lecture, Saarland University, 2024.

[9] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill, "Studying tls usage in android apps," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 350–362. [Online]. Available: https://doi.org/10.1145/3143361.3143400

[10] D. Kozlov, "Avoiding downtime: modern alternatives to outdated certificate pinning practices," [Online]. Available: https://blog.cloudflare.com/why-certificate-pinning-is-outdated/, July 29, 2024, accessed: September 03, 2025.

[11] M. Oltrogge, "Tls on android – evolution over the last decade," SULB, Saarland University, 2021.

[12] Android Developers, "Network security configuration," [Online]. Available: https://developer.android.com/privacy-and-security/security-config#ConfigInheritance, May 02, 2025, accessed: September 03, 2025.

[13] Apple Developer, "Preventing insecure network connections," [Online]. Available: https://developer.apple.com/documentation/security/preventing-insecure-network-connections, accessed: September 13, 2025.

[14] A. Developer, "Nsapptransportsecurity," [Online]. Available: https://developer.apple.com/documentation/bundleresources/information-property-list/nsapptransportsecurity, accessed: September 13, 2025.

[15] Android Developers, "App security improvement program," [Online]. Available: https://developer.android.com/privacy-and-security/googleplay-asi, September 24, 2024, accessed: September 08, 2025.

[16] S. Bugiel, "Network security: Webviews, deeplinks, and app links," Mobile Security Lecture, Saarland University, 2024.

[17] Android Developers, "Update your security provider to protect against ssl exploits," [Online]. Available: https://developer.android.com/privacy-and-security/security-gms-provider, September 24, 2024, accessed: September 09, 2025.

[18] M. R. xda developers, "Android 14 makes root certificates updatable via google play to protect users from malicious cas," [Online]. Available: https://www.xda-developers.com/android-14-root-certificates-updatable/, February 08, 2023, accessed: September 09, 2025.

[19] W. Strafach, "76 popular apps confirmed vulnerable to silent interception of tls-protected data," [Online]. Available: https://medium.com/@chronic_9612/76-popular-apps-confirmed-vulnerable-to-silent-interception-of-tls-protected-data-2c9a2409dd1#.ueusaai6f, February 06, 2017, accessed: September 15, 2025.