Saudi Computer Society, King Saud University

Applied Computing and Informatics

(http://computer.org.sa)
www.ksu.edu.sa
www.sciencedirect.com

ORIGINAL ARTICLE

A Survey on HTTPS Implementation by Android Apps: Issues and Countermeasures

Xuetao Wei [*], **Michael Wolf**

University of Cincinnati, United States

Received 5 July 2016; revised 5 October 2016; accepted 31 October 2016
Available online 11 November 2016

KEYWORDS

HTTPS;
Android;
Mobile security;
TLS/SSL;
Mobile development

Abstract As more and more sensitive data is transferred from mobile applications across unsecured channels, it seems imperative that transport layer encryption should be used in any non-trivial instance. Yet, research indicates that many Android developers do not use HTTPS or violate rules which protect user data from man-in-the-middle attacks. This paper seeks to find a root cause of the disparities between theoretical HTTPS usage and in-the-wild implementation of the protocol by looking into Android applications, online resources, and papers published by HTTPS and Android security researchers. From these resources, we extract a set of barrier categories that exist in the path of proper TLS use. These barriers not only include improper developer practices, but also server misconfiguration, lacking documentation, flaws in libraries, the fundamentally complex TLS PKI system, and a lack of consumer understanding of the importance of HTTPS. Following this discussion, we compile a set of potential solutions and patches to better secure Android HTTPS and the TLS/SSL protocol in general. We conclude our survey with gaps in current understanding of the environment and suggestions for further research.

Contents

1. Introduction 102
2. Overview of Android HTTPS and current findings 103
3. In-depth cause analysis of HTTPS misuse 106
   3.1. Developer misuse of HTTPS 106

[*] Corresponding author.
E-mail address: weix2@ucmail.uc.edu (X. Wei).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.aci.2016.10.001

3.2. Server misconfigurations . . . 108
3.3. Lacking documentation on HTTPS . . . 109
3.4. Flaws in SSL/TLS libraries . . . 109
3.5. Issues in the HTTPS protocol . . . 110
3.6. Need for consumer awareness . . . 111
4. Proposed solutions . . . 111
4.1. Link SSL to DEBUGGABLE flag in the Android manifest . . . 112
4.2. Remove SSLErrorHandler. . . . 112
4.3. Android market and client side application validation . . . 112
4.4. Use SSL pinning. . . . 112
4.5. Improve documentation and API clarity . . . 112
4.6. Begin persistent Internet-wide SSL vulnerability scanning. . . . 113
4.7. Patch and increase oversight on SSL libraries . . . 113
4.8. Large mobile applications should use stronger HTTPS protections . . . 113
4.9. Revise the TLS protocol suite . . . 113
4.10. Increase consumer awareness . . . 114
5. Discussion and future work . . . 114
5.1. Gaps in research. . . . 114
5.2. Moving forward . . . 114
6. Conclusion . . . 115
References . . . 115

1. Introduction

The Android mobile platform was introduced in 2007 by Google and the Open Handset Alliance. Due to its open nature and support from both Google and third-party developers, it has become the most widespread mobile operating system as of 2014 [1]. The ease of entry to Android development has allowed the platform to expand to its current size; however, this free-for-all environment exacerbates issues in application security and user privacy. The security of Android is the burden which individual developers must bear and the standards for security are not always clear. Developers writing applications for Android must consider how their code will assure user safety while simultaneously calculating for minimal memory usage, battery life, and weak processing power. Their apps must comply to security protocols, launch as their own UID, sign their code, and minimize permissions [2]. Needless to say, lost in the innumerable tasks of application creation and deployment, security errors are undeniably frequent. In this survey, the primary focus will be on the insecure development of Internet-connected non-browser Android applications and the implementation of HTTPS, potential remedies, and suggestions for further research. The increased shift of consumer electronics to the mobile realm and the development of a wide range of applications that has followed has meant a steady increase in the amount of personal, critical and confidential information that flows in and out of mobile devices [3,4]. These handhelds use channels such as public WiFi which, even with modern protections, can be vulnerable [5]. Packets can be easily sniffed and manipulated when sent in plaintext HTTP messages over these networks [6]. There are many mechanisms which satisfy the goal of protecting packets, but the SSL and TLS protocol built into HTTPS has become the de facto suite, though it may not deserve the unchecked faith it receives [7]. Thus TLS and its implementations will be the system investigated as we continue. While web browsers are generally able to implement HTTPS connections securely since they are managed by enormous teams of engineers or contributors, Android applications do not have this sort of oversight. The widespread and mostly unsupervised creation of Android applications has allowed for security loopholes to appear in programs which use HTTPS calls [8]. According to the Bureau of Labor Statistics, software jobs in the US are set to grow by 30% by 2022 [9]. It is essential that both new and experienced developers are able to properly tackle loopholes in Android security. The Android platform has several encryption and security suites. It hosts a large Java encryption library and well-respected and versatile third-party implementations such as Bouncy Castle [10] and OpenSSL [11]. There are several different methods of implementing HTTPS built directly into the platform. These methods frequently require no custom code to function securely. In addition, the Android development training website hosts several walkthroughs on HTTPS [12]. Despite the need for transport-layer encryption and the ready availability of encryption mechanisms, many Android applications simply do not implement HTTPS when they should or their code alters the HTTPS implementation in a way that makes the application vulnerable. In these cases, user data are susceptible to Man in the Middle attacks. As shown in Fig. 1, MitM attacks allow for a malicious actor (E) to eavesdrop, intercept and insert itself into a conversation between two legitimate users (A and B). This has become one of the most pressing threats to wireless and cellular communications.

More unfortunately, there is frequently no warning to the user that these vulnerable connections are not secured by SSL/TLS. Issues remain in SSL libraries, the TLS and X.509 certificate validation protocol, and server-side configurations. As will be discussed in later sections, cleaning up the SSL universe to protect user data requires the cooperation of more parties than just Android application developers. It is imperative that proper encryption is used in all applications which process user data over the Internet. This paper will analyze why developers do not (or are unable to) implement secure HTTPS connections and present an idea for a solution to the

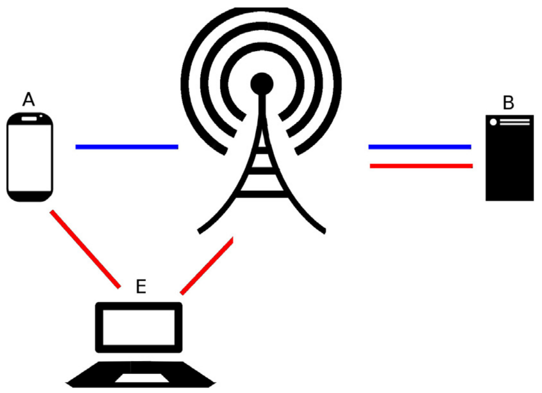


Figure 1 A classic Man In the Middle attack with the conversation between Alice (A) and Bob (B) being intercepted by Eve (E).

gap between theoretical security and implemented HTTPS security in Android. We will look at the state of the art research in fields beyond the mobile realm to detect trends in security and ascertain ways to harden the HTTPS environment on Android. The remainder of this paper is organized in the following way. Section 2 contains a summary of HTTPS, its proper usage on the Android platform, and the major relevant findings contributed by security researchers. Section 3 provides a deeper interpretation and grouping of these results including a listing and discussion of causes of HTTPS misuse. Section 4 provides a listing of potential solutions which have been suggested by security researchers. Section 5.1 gives the observed gaps in current Android HTTPS research. Section 5.2 contains concrete suggestions for future research which fulfill the some of the solutions suggested in Section 3 or bridge holes in current understanding noted in Section 5.1. The paper is concluded in Section 6.

2. Overview of Android HTTPS and current findings

Cryptography is difficult to implement even with modern software [13]. In order to create resistant keys, complex algorithms and programming mechanisms are needed. Hundreds of algorithms are used for different steps in the encryption process [2]. Adding to this, developers aren't always taught security best practices [14]. As computers increasingly grow in their processing capacity, so will the encryption systems grow in intricacy to maintain their defenses against brute force and man-in-the-middle (MITM) attacks. Internet systems have developed greater complexity with the influx of users, web stakeholders, and non-traditional server methodologies [15]. In order to secure user data and the integrity of Internet-connected applications, developers must be able to properly implement encryption technologies [4,16,17]. SSL/TLS is one such cryptographic system which requires a layer of abstraction in order to be usable to developers. SSL was developed to provide an end-to-end encrypted data channel for servers and clients on wireless systems. Given that wireless technology is prone on the physical level to eavesdropping attacks based on RF broadcast interception, this cryptographic protocol is vital for the secure transfer of any data to and from cell phones [18]. The cornerstone of SSL is the ability of the client to confirm without a doubt that the server contacted is the correct one. From here data can then be transferred with trust. To establish this state of trust, a complicated, mixed public-private key exchange takes place. This requires an extensive handshake and verification process to avoid sending the encryption key to any interceptors on the network who have latched on to the chain of communication.

In the lowest level, SSL functions in the following way depicted in Fig. 2. The client sends an HTTPS request to the server with its SSL version number and supported ciphers. The server responds with its SSL version number and ciphers as well as its certificate. This server certificate has been signed by a trusted certificate authority (CA) which has verified the servers authenticity. The client will compare the certificate's public key to its local key store and the field values to expected values. If the certificate passes, and the certificate has not been revoked by a CA (as determined by a query to a CA's certificate revocation list (CRL)), the handshake continues. The cipher suite is chosen from the algorithms which the client and server have in common. An example cipher suite could use ECDHE for key exchange, RSA for certificates, AES128-

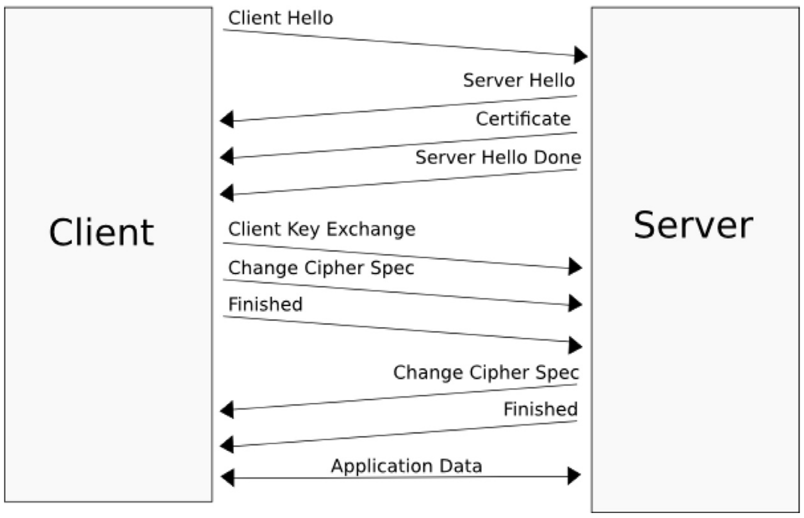


Figure 2 A simple TLS handshake.

GCM for message encryption, and SHA256 for message integrity checking. A pre-master secret is encrypted with server's public key using a cipher suite which is in common between the two machines and transmitted to the server. If the client has been asked to verify itself with a certificate, this will be included with the secret and transmitted to the server. If the authenticity is confirmed, each machine uses the pre-master secret to generate the master – a session ID that functions as the symmetric key for the SSL communication. Once the handshake has been completed and each device informs the other that all further communication will be encrypted with the session ID, the client encrypts its messages using the symmetric key and sends the data to the server. Once all data is sent, the connection is terminated [19,20]. Digital certificates, the core of the SSL system, are based on the X.509 protocol [21]. This protocol along with the Online Certificate Status Protocol (OCSP) [22] establish how certificates are to be developed, validated, and revoked [23]. The major components of certificate checking are issuer verification, hostname validation and revocation checking. Each of these steps assures that the server in question is still trusted by a certificate authority. Within the certificate validation process, issues have arisen with servers signing their own certificates and certificates using wildcard hostnames (for example, *.google.com). These discrepancies are easily spotted and flagged by properly implemented SSL clients or humans. However, in situations where the functionality of X.509 has been compromised by custom code, such as removed revocation checks, these invalid certificates can be accepted – rendering the SSL process useless [24]. Without proper validation checks, any rogue access point can break into the chain of communication, send a random certificate to the user, and forward the packets to the original server, decrypting and reading all data flowing between the ends. Much in the way that higher-level programming languages obscure memory management to make the developers job more straightforward, so do many encryption suites try to make encryption and decryption a standard, human-understandable process. The papers and communications which become the foundation of SSL, TLS, and the many improvements, revisions, and decisions on these topics, come from the Internet Engineering Task Force (IETF) [25]. These technical documents cannot be directly utilized by most developers. Thus, libraries and encryption suites take the technical documentation and develop a platform for applications to use. Libraries like OpenSSL [11] handle the TLS handshake for developers, leading to a more uniform and secure set of HTTPS connections. While these libraries have come under scrutiny due to security flaws [26,27], their role is vital in the Internet and they have existed for years. OpenSSL, founded in 1998, is used by servers which comprise 66% of the web servers [28]. SSL/TLS libraries are the 'physical' implementation of the IETF protocols. However, this code is not necessarily 'in the wild'. In this paper, we will use the term 'in the wild' to refer instead to consumer-facing applications. As libraries rely on protocols for guidance, consumer-facing implementations rely on libraries and, in effect, protocols, for guidance. For this paper, the primary 'wild' code investigated will be on the Android platform. The movement of cryptography to abstraction is especially important in Android which has a heavy focus on third-party development and ease-of-development. In the standard Android implementation of HTTPS, there are three parts in the creation of a secure connection with SSL/TLS. These three parts, setup, socket generation, and the certificate management, reflect the typical TLS handshake protocol [29]. The following is a possible SSL implementation. Setup involves customizing the HTTP packet headers. This can be done through HttpParams and ClientConnectionManager to transmit the proper headers and data. Cipher suites can be manually selected, but defaults will function for most calls. The socket is generated through an instance of the SSLSocketFactory class. Finally, the X509TrustManager which is an entity within the SSLSocketFactory will by default authenticate credentials and certificates. The stock Android trust manager has 134 root certificate authorities installed [30]. The library will attempt to trace the certificate trust chain back to one of these 134 root CAs. Once the client and server are certified, transmission commences. This can be an incredibly simple and black-boxed process. For instance, according to the Android Developer Training, valid HTTPS code can be written in four lines using HttpsURLConnection, part of the URLConnection library (see Listing 1) [12,31].

Assuming that the device had the proper certificates installed, this code in lst. 1 would be operational. The URLConnection API takes care of hostname verification and certificate management. Besides the URLConnection API, other libraries and middleware have been developed for application designers which manage these components. More customization is available in the Java Secure Socket Extension (JSSE) which comes packaged with Java [32]. Other common libraries include OpenSSL [11] and GnuTLS [33]: C-based frameworks for SSL/TLS implementation. Higher-level wrapper implementations of these SSL/TLS libraries include cURL [34] and Apache HttpClient [35]. Furthermore, certain industries have their own middleware such as Amazons Flexible Payment Service [36] which help abstract the HTTPS connection code away from the developer. While libraries are an attempt to make SSL/TLS implementation default, they can also leave the applications vulnerable. Since application security is completely tied to the libraries it uses, flaws in the libraries are in extension flaws in the applications which use them. Giorgiev et al. [37] found that SSL certificate validation is completely broken in many critical software applications and libraries. In one example, Chase Mobile Banking overrides X509TrustManager and doesn't check the server's certificate thus violating the most important aspect of HTTPS. Furthermore, Tendulkar et al. [8] found that during the investigation of 26 open-source applications, 10 were using SSL incorrectly. This is perhaps due to misreading library documentation or overriding important features of the suites. Even large-scale enterprises misuse HTTPS or don't fully secure their connections [24]. A vulnerability note released by CERT identifies applications by Fandango and Credit Karma which fail to validate SSL certificates [38]. The issues within Android are more complex than a lack of experience with application construction, they derive from issues in libraries, protocols, server configurations, and user comprehension of SSL and TLS. Fahl et al. [24] developed a tool called MalloDroid which targets application vulnerabilities dealing with MITM attacks. MalloDroid analyzed the API calls which applications made, checked the validity of certificates, and identified cases of custom HTTPS implementation. Of the applications tested, 8% were vulnerable. The main issues discovered in this investigation were symptoms of unnecessary customizations placed

```
URL url = new URL("https://wikipedia.org");
URLConnection urlConnection = url.openConnection();
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

Listing 1 Example of a standard Android HTTPS call.

over default SSL code. The use of customized code over SSL defaults is almost always detrimental [39]. Tendulkar et al. [8] found that 1613 out of 1742 implementations of SSL with custom code did not require anything beyond the defaults. In fact, in most cases, adding the single character ‘s’ would have allowed the application to securely use HTTPS. The primary customizations at fault were trust managers which accept all certificates, trust all hostnames, and ignore SSL errors [24,8]. Trust managers exist to validate certificates. When the certificate checking is turned off, security is compromised. Using user-defined trust managers that accept all certificates or self-signed certificates has been shown to be an issue in the Android community. It places user data in a vulnerable position and compromises the original intention of both SSL libraries and the SSL/TLS protocol. Unfortunately, trusting all hostnames is even simpler than implementing a custom trust manager. Using the org.apache.http.conn.ssl.AllowAll HostnameVerifier, developers are able to bypass checking the server for a certified hostname. Several applications investigated with MalloDroid contained custom classes which allowed all hostnames in the SSL connection [24]. This implementation subverts the fundamental trust process of SSL/TLS. Many mobile applications have been found to simply ignore the errors thrown by Android or a corresponding library which could not validate the HTTPS certificate.

As seen in Listing 2, messages are hidden from users and the application continues as though it has a secure connection[24,8]. Again, overriding errors thrown by the system defeats its purpose and mimics the insecure manner in which users click through SSL errors in the browser. However, unlike in desktop browsers, this case comes with the repercussion of never presenting the user with options for their own security.

One final issue that doesn’t revolve around the customization of default SSL code is that developers sometimes use hybrid HTTP/HTTPS or don’t use SSL/TLS at all. Fahl et al. [24] found an instant messenger application which sent login credentials over non-encrypted channels vulnerable to a replay attack. Other hybrid systems were vulnerable to stripping attacks or leaking data through broken SSL channels. Browsers and applications using Android’s WebView to connect to a server are particularly vulnerable in these cases. These instances warrant attention from both developers and server architects. Beyond application-level flaws, there are widespread server misconfigurations which lead to a large number of false positive SSL errors [40]. These false-positives take up user attention and lead to an unsafe dismissal of SSL error validity by developers and users. Certificate management is often a difficult and paperwork-intensive process for server operations teams. In addition, content delivery networks (CDNs), and more specifically CNAME routing, have complicated the certificate issuance and validation process. Since the CDN model is based off surrogate servers handling web traffic load from the customer’s server, using HTTPS properly, an intimate client-server model, requires less-than-ideal workarounds to maintain non-repudiation and trust. During the investigation of 20 CDN providers and 10,721 websites by Liang et al. [15], 15% raised invalid certificate errors. All 5 CDNs investigated had insecure HTTPS or HTTP communication on the back-end. Due to many Android applications reliance on servers which use CDNs, this issue needs to be resolved in order for efforts of client-side validation and error reporting to be accurate and attune to SSL errors. Each of these vulnerabilities identified are not just issues waiting to be exploited. Huang et al. [41] showed that 2% of certificates

```
public class IgnoreCertsSSLSocketFactory extends SSLSocketFactory {
        SSLContext sslContext = SSLContext.getInstance("TLS");

        public IgnoreCertsSSLSocketFactory(KeyStore truststore)
            throws NoSuchAlgorithmException, KeyManagementException,
            KeyStoreException, UnrecoverableKeyException {
            super(truststore);

            TrustManager tm = new X509TrustManager() {
                public void checkClientTrusted(X509Certificate[] chain, String authType)
             throws CertificateException {
                }

                public void checkServerTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
                }

                public X509Certificate[] getAcceptedIssuers() {
                    return null;
                }
            };

            sslContext.init(null, new TrustManager[] { tm }, null);
        }
}
```

Listing 2 Overridden SSLSocketFactory found in the wild.

which users received when accessing Facebook were forged. These false certificates were invalid and should have been cast away, but users still followed them or the application which accessed the website did not throw an error, thus falling victim to a man-in-the-middle attack. As Moxie Marlinspike has shown with his tool sslsniff [42], automated MITM attacks are simple to carry out. The susceptibility of the physical layer of mobile communication to eavesdroppers only raises this risk. Android HTTPS development is in a bind. While developers want to have a secure system for their users, several factors within and without their control complicate proper implementation of end-to-end encrypted communication. Why aren't Android developers using HTTPS? Why do existing SSL implementations remain insecure? In the next section, we will analyze factors which have been identified in the current Android development process and SSL/TLS ecosystem which keep HTTPS from reaching the ideal security that it is often claimed to be.

3. In-depth cause analysis of HTTPS misuse

In this section, we will look into the issues which compromise the security of SSL/TLS implementations, in particular, on Android-based devices. First, we will look at the primary causes of SSL insecurities with current Android HTTPS implementations. Even though the only guidance afforded to developers hoping to secure their applications are a few weakly enforced frameworks in the Android system, security must be built by manually integrating custom security features into the project while retaining a functional and cohesive form. This process can be ad hoc, prone to error, repetitive, and inexact [43]. As shown in the previous section, vulnerabilities and holes are rampant and actively exploited. The first step in patching these flaws is determining their origin. There exist issues rooted in the mobile app development paradigm, server configuration, Android documentation, SSL/TLS libraries, the SSL/TLS protocol, and application consumers. Extensive research has gone into determining the root cause of each of these factors. This section will investigate each of these causes further.

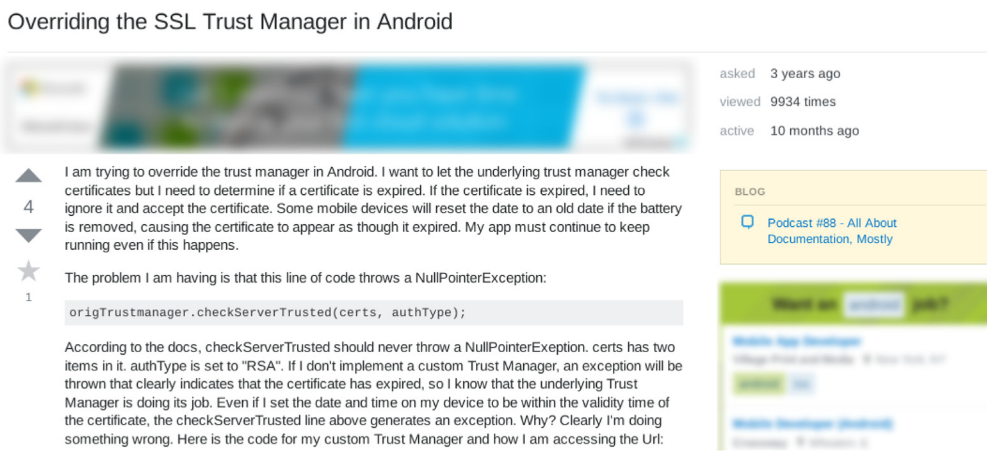
3.1. Developer misuse of HTTPS

Among the papers reviewed, the most commonly reported flaws in HTTPS configuration were due developer negligence. One such problem is debug code being left in production applications. this problem isn't new and it has been listed in the Common Weakness Enumeration [44]. Leftover code and snippets that bypass standard procedures to make the an app operational in development have a widespread effect on application security [8]. Ironically, leftover debug code can violate the protections which the system it models is supposed to afford. HTTPS is not immune to development glitches where the author of a program either leaves vulnerable code or places an intentional override in their application, especially on Android. This could come in the form of a situation where, in order for an application to populate and display data for the developer, the certificate validation must be set up to allow a stream of data from a mock server. This allows for the author to assure the other components of the application are properly functioning, but leaves the HTTPS connection vulnerable unless the certificate checking is turned back on. This happens with unfortunate frequency since there are simple mechanisms which we will later discuss to prevent unintentional remnant debug code from emerging in production applications. Developers want top level security, but also desire their product to function properly in development [24]. This creates an issue with a complex setup like SSL/TLS. As explained earlier, the most crucial part of an HTTPS communication paradigm is the existence of valid certificates and recognized certificate authorities. When running both the server and the application, developers may build their server with self-signed certificates for development and forget to change the application's validation process when they do get the proper CA-signed certificate or bypass this for alternative reasons [39,45]. Running an application without SSL/TLS protection while debugging is obviously harmless, but once these apps are open to the general public, there is extreme risk of data theft. Beyond inspecting the code itself, speaking to developers about their mistakes and security bugs yields a more thorough look into the cause of these developer-based flaws. A study conducted by Fahl et al. [24] showed a few trends among the developers surveyed.

(i) Developers make mistakes. Upon contacting the developers at fault, many took the advice and fixed their mistakes. Others, however, refused to admit that the flaw was an issue [39]. These mistakes are understandable. Android is a complex system and public-key cryptography is not a easily grasped even with high-level libraries. The startling rejection and denial made by developers in this survey may be a result of embarrassment at incorrectly implementing code. However, for applications made by developers both willing and unwilling to admit fault for SSL misconfiguration, it seems apparent that there was a failing in code coverage in the development process.

(ii) Another explanation may be apathy or simple ignorance on the topic of SSL/TLS security. A paper by Xie et al. [14] found that while many of the participants in their experiment had a general knowledge and awareness of software security, there were gaps between this knowledge and the actual practices and behaviors that their participants reported. Despite general knowledge of security, they were not able to give concrete examples of their personal security practices. In the same study, Xie et al. noted that there was a prevalence of the "it's not my responsibility" attitude. The developers often relied on other people, processes, or technology to handle the security side of the application. When these software authors are so busy with the pure functionality and viability of their product and the approaching deadlines, it is obvious that the security hat looks much better on another member of the team. Unfortunately, code review and quality assurance only go so far, especially when looking at an application retrospectively. In an ideal situation, security is considered in every step of the development process from the design through the deployment. As evidenced by this report, this is not the case in many development environments.

(iii) Online forums and user-to-user resources may not be the cause of developer misuse of SSL, but they allow developers to discover ways to bypass security measures in



Overriding the SSL Trust Manager in Android

asked 3 years ago
viewed 9934 times
active 10 months ago

4

1

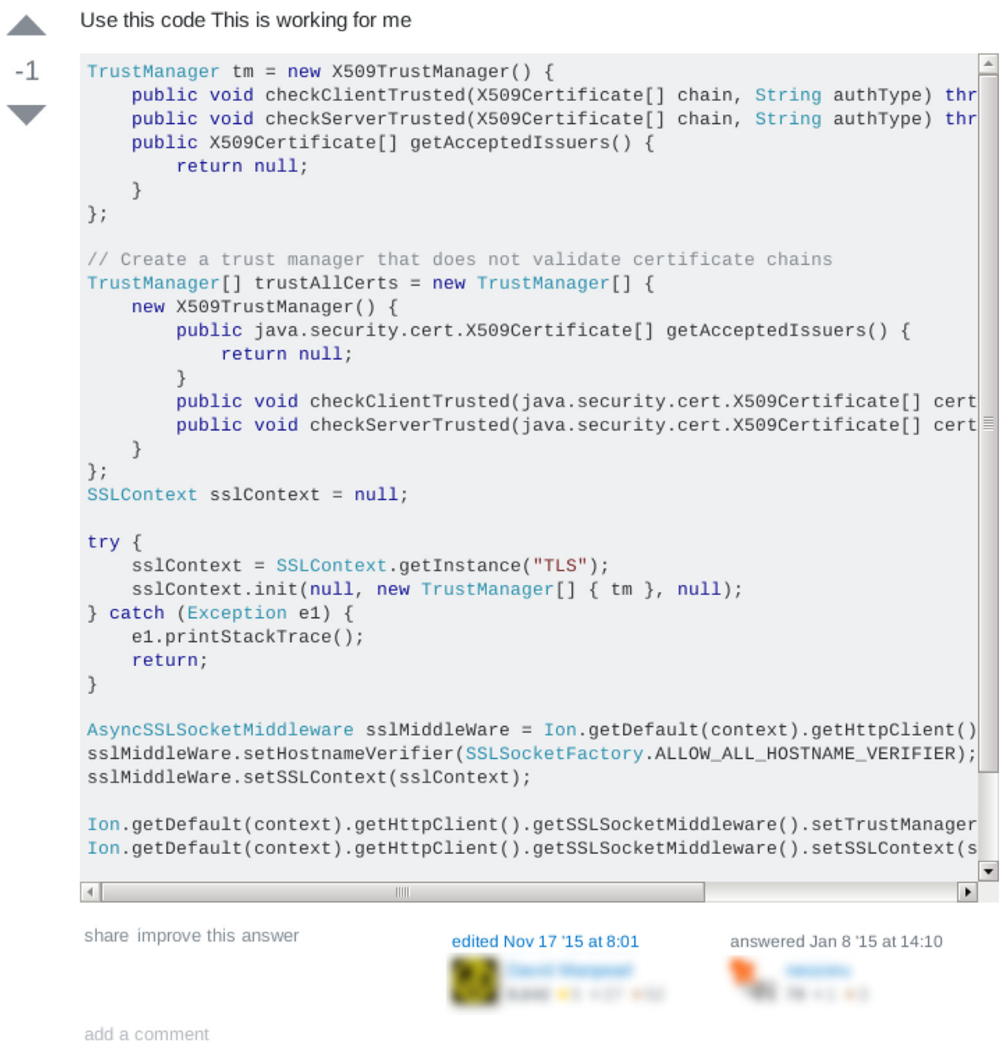
I am trying to override the trust manager in Android. I want to let the underlying trust manager check certificates but I need to determine if a certificate is expired. If the certificate is expired, I need to ignore it and accept the certificate. Some mobile devices will reset the date to an old date if the battery is removed, causing the certificate to appear as though it expired. My app must continue to keep running even if this happens.

The problem I am having is that this line of code throws a NullPointerException:

```
origTrustmanager.checkServerTrusted(certs, authType);
```

According to the docs, checkServerTrusted should never throw a NullPointerException. certs has two items in it. authType is set to "RSA". If I don't implement a custom Trust Manager, an exception will be thrown that clearly indicates that the certificate has expired, so I know that the underlying Trust Manager is doing its job. Even if I set the date and time on my device to be within the validity time of the certificate, the checkServerTrusted line above generates an exception. Why? Clearly I'm doing something wrong. Here is the code for my custom Trust Manager and how I am accessing the Url:

Figure 3 An example question seeking an SSL override on StackExchange.



Use this code This is working for me

-1

```
TrustManager tm = new X509TrustManager() {
    public void checkClientTrusted(X509Certificate[] chain, String authType) thr
    public void checkServerTrusted(X509Certificate[] chain, String authType) thr
    public X509Certificate[] getAcceptedIssuers() {
        return null;
    }
};

// Create a trust manager that does not validate certificate chains
TrustManager[] trustAllCerts = new TrustManager[] {
    new X509TrustManager() {
        public java.security.cert.X509Certificate[] getAcceptedIssuers() {
            return null;
        }
        public void checkClientTrusted(java.security.cert.X509Certificate[] cert
        public void checkServerTrusted(java.security.cert.X509Certificate[] cert
    }
};
SSLContext sslContext = null;

try {
    sslContext = SSLContext.getInstance("TLS");
    sslContext.init(null, new TrustManager[] { tm }, null);
} catch (Exception e1) {
    e1.printStackTrace();
    return;
}

AsyncSSLSocketMiddleware sslMiddleWare = Ion.getDefault(context).getHttpClient()
sslMiddleWare.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
sslMiddleWare.setSSLContext(sslContext);

Ion.getDefault(context).getHttpClient().getSSLSocketMiddleware().setTrustManager
Ion.getDefault(context).getHttpClient().getSSLSocketMiddleware().setSSLContext(s
```

share improve this answer

edited Nov 17 '15 at 8:01

answered Jan 8 '15 at 14:10

add a comment

Figure 4 An unsafe suggestion for resolving the question on StackExchange.

order to solve errors. One such website is Stack Overflow [46]. Typically, errors solved are caused when the developer who has posed the question has incorrectly written a chunk of code. In these situations, Stack Overflow operates in an important, positive way. However, in the case of SSL errors, the most trivial way to stop the errors without configuring the server is to stop the application from throwing the errors. Figs. 3 and 4 show an example of an Android SSL certificate expiration override. While most respondents explain that these solutions should not be used in production environments before giving a sample override, some answers, such as the one shown in Fig. 4, do not provide that context. This answer has received negative feedback most likely for this reason. However, given almost 10,000 views, this solution has almost certainly ended up in a developer's production application. Thoughtful answers are often mixed with less security-oriented responses on these websites, allowing harmful programming paradigms to develop online.For instance, a developer may ask for a way to get past the UntrustedCertificate error in Apache's HttpClient and the answer may be to use a custom SSLSocketFactory to trust all hosts [47]. Of course, those who answered the question or other community members may stress how this should not run in production, but the solution is still presented in a fashion that a desperate developer can quickly find a work-around. Websites such as Stack Overflow don't encourage app designers to customize their SSL/TLS implementation to use self-signed certificates and accept all hosts in production, but they do show developers how to use them in testing [39]. Stackoverflow cannot be blamed nor can the open-door style of development which Android possesses. The fault in these situations are the developers who either forget to remove the work-around code or just ignore the warnings on using accept-all policies in production applications.

Despite the many flaws which can be found in Android development and production applications, there is no solid evidence that Android developers are more clumsy with SSL than others in a similar situation. Investigations of iOS applications has shown that the two platforms have a comparable number of SSL/TLS vulnerabilities [39]. The so-called walled garden approach doesn't seem to fix issues in developer misuse of HTTPS. While it may make sense to correlate a lack of developer knowledge with an incorrect SSL connection, it would be incorrect to say that the Android-specific development paradigm causes these errors. If anything can be found to be lacking, it is a lack of oversight on mobile applications. Another, more social, factor may contribute to these developer mistakes and in turn effect the security of HTTPS calls. Xie et al. [14] show that there are issues in developer environments (team members, support staff, managers, etc.) that can cause them to make mistakes. One such issue is misplaced trust in process. This involves believing that software security is only retrospective or investigated in the code review stages. Secondly, there is the feeling that a software engineer doesn't need to know about vulnerabilities if they aren't specifically working on them. This would be like designing a backend without paying any attention to the frontend. Software isn't contextual, and all components in the final project need to be designed, developed, and reviewed at every step in the process. Each member of a team should be aware of what the others are doing in order to create the most accurate and unified product. Finally, and most recognizably to developers, is the existence of external constraints which effect workflow and programming process on a human level. These include deadlines, client desires, government policy, and any sort of confining elements that would stop the developer from creating the product in the way he or she imagines. When the budget tightens or a deadline approaches, proper security can be an unfortunate sacrifice when a client's main focus is functionality and design [14]. Besides a missing understanding of HTTPS standards, these external constraints potentially hold the most sway over the correctness of a developer's solution. Developers are faced with pressure, deadlines, an imperfect support system, and the complexities of public key infrastructure and Android. Mistakes and misconfigurations are bound to arise in this system. When user data must rely on this stressed authorship, there are serious implications. While applications created by these developers are the breaking-point in this system, there are several more causes both for developer mistakes and general insecurities in Android SSL connections.

3.2. Server misconfigurations

On the opposite end of the TLS system is the HTTPS server. Setting up an Apache HTTPS server is not difficult [48]. In addition, security for these servers can be configured to be much higher with ease [49]. Despite this, only 45% of the Top 1 Million websites support HTTPS [50]. Furthermore, the systems which do operate on HTTPS can have flaws which can completely compromise the security of SSL. Korczynski et al. [51] discovered that even in a relatively small set of Internet services, certain elements of the TLS protocol were being ignored or misused. These heavily trafficked websites receive significant amounts of traffic and financial transactions, making it imperative for stronger end-to-end implementations of TLS. SSL server probing [52,53] has shown an upward trend in positive TLS implementation and healthy cipher use, however the growing reliance on encrypted data flows has made tight adherence to protocols on the server-side fundamental to effective security throughout the full Internet domain. A frequent mistake made by HTTPS servers is the use of self-signed certificates. Self-signed certificates, certificates which have no authority to back up their validity, work well in testing situations, but when a server needs accept requests from the public Internet, these false certificates are unsafe. In these cases, a signed certificate from a certificate authority must be acquired or purchased. These servers will treat Android traffic the same way as any traffic and cause the pitfalls for mobile traffic just as they do for desktop clients. Indeed, the most frequent issue with server configuration is the mishandling of certificate installation [40]. Certificate management isn't an automated process. After applying for a certificate from a certificate authority, that certificate is sent by email to the company which sent in the request. This certificate must then be manually installed in order for clients to believe that the server is in fact correct. When certificates expire following their two or three year lifespan, a smooth transition to a new certificate must be carried out in order to assure maximum uptime. Vratonjic et al. [54] found that among many other violations, 82.4% of servers investigated used expired or otherwise invalid certificates. Again, in the days following the Heartbleed bug, only 10% of vulnerable servers replaced their potentially compromised certificates. Of this 10%, another 14% reused the same private key which may have leaked [50,55]. These cases demonstrate the difficulty that system operators have with healthy use of certificates. Indeed, the prevalence of these incorrectly implemented certificates has a direct effect on developers and the services that rely on secure Internet connections. If developers can't connect to a server outside their control due to an SSL error, the only course of action would be to lower the validation parameters on their application. Both ends of the SSL connection need to maintain the highest level of security. In order to reach adoption by developers on all platforms, the system must display a reasonable level of consistent functionality. Certificate management is a tricky and complicated aspect of SSL which needs further research, tools, and perspectives to be introduced before it can reach a realistically reliable state. New technologies in the burgeoning operations world make certificate management even trickier. Content Delivery Networks (CDNs) are distributed server farms which spread out the load on large, public websites. The servers of the CDN act as surrogates for the main web server, stepping in the middle of a direct client-server relationship. This middle-man server must be trusted by the server, but any current method of doing this violates the SSL protocol [15]. Innovative methodologies must be contributed to the X.509 protocol and the certificate authority industry to meet the challenge of scaling websites and an ever-increasing pool of vital websites that require certificates to be properly installed. Until servers are properly secured, the security of all client applications will suffer. Developers will be wary of using the protocol and the default Internet connection methodology on Android will not be HTTPS until it is as easy to implement as cleartext HTTP.

3.3. Lacking documentation on HTTPS

Beyond the physical limitations of an SSL connection, one of the problems which developers face is a lack of proper documentation and a foundation in the importance of application security. There is very little research of interactive support to developers for secure software development [14]. This information is critical to expose developers to correct methodologies and point them in the way of secure Internet connection creation. While the Android platform prides itself on ease of use, it can be surprisingly confusing. For instance, manual analysis of financial applications with vulnerabilities in their inter-app communication yielded the conclusion that several flaws were caused due to developer confusion over Android's complexities [56]. The authors stated that they believe that these errors, made by security-conscious developers, are indicative of the fact that Android's Intent system is confusing and tricky to use securely. This subject, completely separate from SSL/TLS in terms of purpose and architecture, has shown that Android is at its core a complex system that is difficult to comprehend from a front-end developer standpoint. Existing documentation and tutorials are not reaching their audiences effectively. Approaching the issue of 'complexity' isn't an endeavor that can happen with a single update. However, in order to further the security and proper development practices of Android applications, the maintainers of the operating system must work toward abstracting the complexities or putting out better documentation. Further research must go into the psychology behind technical documentation comprehension, particularly for Android. One such example of inadequate training is also the most critical. The Android developer training on SSL/TLS is sorely lacking in proper examples and implementation. The training on security is near the bottom of the screen and listed below trainings on user interface and performance [12]. There is a minimal explanation of the protocol or public key cryptography in general. A lack of solid documentation in popular SSL/TLS libraries also presents an issue [37]. The OpenSSL library documentation [11] is a meaty webpage that can be rather intimidating. It may be that the quick code snippets of StackOverflow are much more appealing. In several prominent libraries, there are examples of generally confusing APIs. This will be discussed in the next subsection. In order to fulfill their role in the implementation of SSL, libraries must create documentation for developers who are not cryptography experts. Major security-breaching methods like AllowAllHostnameVerifier should be documented as being for testing purposes only [57]. Finally, there are general barriers in coding that need to be broken down in order to allow developers to properly build secure programming principles into their products. Research by Ko et al. [58] has presented findings on the elements of programming environments which prevent problem solving. The primary takeaway from this study is that there is a minimal error reporting infrastructure in many major IDEs and programming language compilers. There are invisible rules that seem to exist without much documentation and differences in programming interfaces interfere with the natural flow of problem solving. Not only do libraries need to be more informative, but application development tools should be smart enough to identify security flaws or inform developers of best practices. A solid documentation source would be responsive to user confusion and effective in communicating the most simple, but secure solution.

3.4. Flaws in SSL/TLS libraries

The ideal Android HTTPS library would enable developers to use SSL correctly without coding effort and prevent them from breaking certificate validation through customization [39]. This would be a model where socket generation and administration of certification authorities are the only responsibilities assigned to programmer. It would bridge the gap between control facilities needed to establish HTTPS connections, making it unnecessary to involve programmers in the development of every essential interface in the already complex HTTPS environment [29]. Furthermore, the API should allow certain relaxed certificate validation when the application is being testing. Dozens of libraries and SSL/TLS abstraction frameworks exist to make HTTPS easier to use. Despite the goal of making the system more approachable, Cairns et al. and others have shown that major SSL/TLS libraries remain too complicated and low-level [17,59]. Fahl et al. claim that there is no solid library which provides easy SSL usage [39]. Indeed, it seems that frustration with APIs is the guiding factor behind developers resorting to StackOverflow to find work-arounds. Georgiev et al. conducted an investigation into critical applications which were compromised due to these flawed or poorly-written libraries [37]. The cURL library is one such confusing library. For example, Amazons Flexible Payments Service PHP library attempts to enable hostname verification by setting cURLs CURLOPT_SSL_VERIFYHOST parameter to true. Unfortunately, this is the wrong boolean to turn on hostname verification and thus the middleware and all applications using it are compromised. PayPals Payment library makes the same mistake. Not only cURL, but GnuTLS has a misleading gnutls_certificate_verify_peers2 which leaves the Lynx text-based web browser vulnerable. Poorly worded APIs defeat the goal of libraries to make SSL easier to correctly implement. Combined with poor documentation, these libraries can be detrimental to a healthy public key infrastructure. Other problems were pointed out in the study by Georgiev et al. Validation was lacking and documentation was so scarce that users were led to misuse the suite. Error handling was different for each library. This sort of miscommunication between systems has lead developers to frequently use the incorrect SSL/TLS libraries for their specific problem. For instance, python libraries urllib2 and httplib which do not support certificate checking, were used in applications hooking into PayPal and Twitter. The disconnect between end users and libraries can be bridged with better communication, documentation, and standards across libraries. Not only are SSL/TLS APIs found to be often confusing, but some contain their own programmatic holes. Apache Axis, which is used by big-name applications from PayPal and Amazon, implements Apache's HTTPClient. Axis uses the standard SSLSocketFactory, but omits hostname verification. Using the independent nature of various SSL libraries to compare reactions to certificates, Brubaker et al. found several holes in major open source libraries and browsers [60]. While efforts in finding flaws have encouraged library developers to patch their software, more oversight needs to go into these libraries which provide the backbone (and reputation) of the HTTPS ecosystem. Making APIs easier goes hand-in-hand with documentation clarification and developer education on security [17]. As technology progresses and more of the Internet supports HTTPS connections, libraries will be forced to become more user friendly and standard. Issues like Heartbleed, which allowed attackers to sniff protected memory from approximately 25–50% of Alexa top 1 million HTTPS sites, while frightening, will encourage more scrutinizing eyes to fall on open source SSL libraries and the infrastructure which supports it [50]. Security researchers have called for more development on these critical open-source projects in order to protect the entirety of the HTTPS infrastructure. Android developers rely on these libraries and they must be firmly in place for developers to use.

3.5. Issues in the HTTPS protocol

Below the lowest-level SSL libraries, the TLS/x.509 protocols are set. Even in this foundation of the HTTPS world, there are flaws. As Fahl et al. express, the SSL/TLS protocol isn't forceful enough [24]. Validation checks are not a central part of the SSL/TLS and X.509 standards [22,61]. Recommendations are given in these IETF papers, but the actual implementation is left to the application developer. IETF RFC 2818, Section 3.1 states that if the client has external information as to the expected identity of the server, the hostname check may be omitted. Both OpenSSL and cURL have issues with the proper implementation of certificate validation. This leniency in the protocol shifts focus away from a vital part of SSL security. While understandable for developing applications with a limited budget, this guiding document of HTTPS must be more definite on the vital subject of hostname verification especially in production applications. Beyond its weak enforcement of certificate validation, there are several issues with the TLS protocol that leave it vulnerable. A couple issues mentioned in a study by Bhargavan et al. [59] are that the protocol allows cipher suites which have been ruled unsafe and allows reused identities on resumption of sessions which can potentially break through the process of the TLS handshake by bypassing it. As security researchers continue to look into the TLS protocol and its shortcomings, more hardening mechanisms will be determined. Server-side policies have been presented by the IETF to curb the use of HTTP instead of HTTPS such as HTTP Strict Transport Security (HSTS) [62]. Described in Fig. 5, they allow for a server to always redirect traffic to HTTPS through the use of an additional header. This effectively counteracts Moxie Marlinspike's SSLSniff [42]. Consumers must manually override occurrences of failed HSTS in their browser. However, these fixes are not as effective as client-side HTTPS Everywhere [63] which similarly forces the server to provide the HTTPS version of the service.

Unfortunately, no such implementation of HTTPS Everywhere for communication libraries exists at the time this paper was written. The promise of client certification validity can

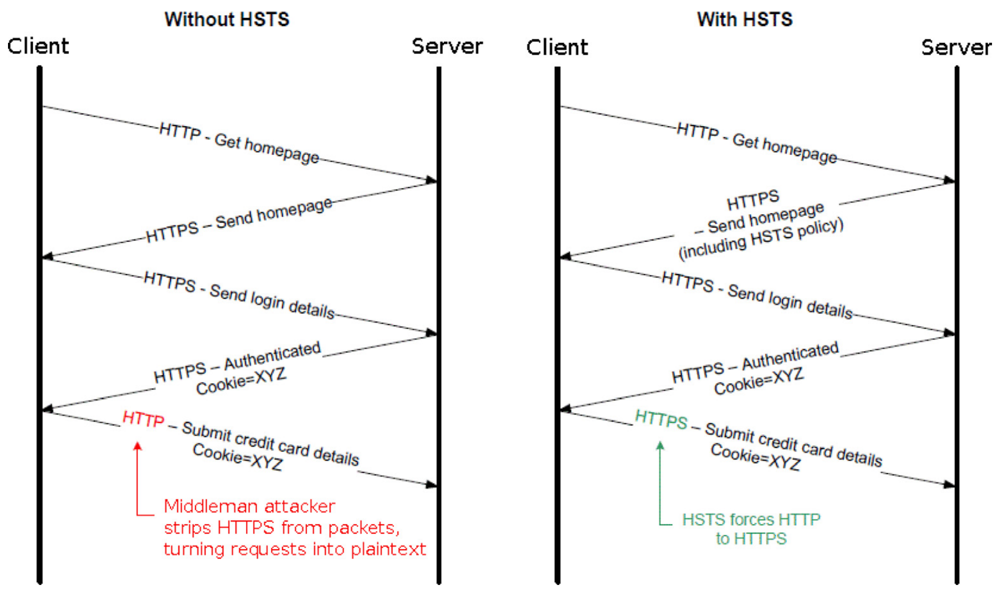

Without HSTS
Client
Server
HTTP - Get homepage
HTTPS – Send homepage
HTTPS - Send login details
HTTPS – Authenticated Cookie=XYZ
HTTP – Submit credit card details Cookie=XYZ
Middleman attacker strips HTTPS from packets, turning requests into plaintext
With HSTS
Client
Server
HTTP - Get homepage
HTTPS – Send homepage (including HSTS policy)
HTTPS - Send login details
HTTPS – Authenticated Cookie=XYZ
HTTPS – Submit credit card details Cookie=XYZ
HSTS forces HTTP to HTTPS

Figure 5 Overview of the role of HSTS [64].

double the prevention efforts of man-in-the-middle attacks. As HTTPS becomes more of a universal standard, it is clear that communication libraries must offer an HTTPS-only style method to protect user data from plaintext servers and HTTPS stripping attacks. Another critical protocol in SSL security, X.509 [65], is extremely general and flexible. It has too many complexity related security features of which few are used. Parsing X.509 certificates isn't simple [66]. Indeed, it also bears the pressure of new technologies for which it has no solution. The X.509 Protocol leaves no room for CDNs which have become ubiquitous on the Internet [15]. Like the TLS protocol, the X.509 certificate protocol needs an update which narrows down its reach, provides rigidity and standard security mechanisms, and is able to adapt to changes in the make-up of Internet routing infrastructure. In the current SSL environment, system operators and developers are effected by the corruption of certificate authorities. As shown in the work of Amann et al. [67] and Bates et al. [68], the entire CA system is convoluted, unreliable, and overflowing with too many CAs [7]. When web certificates rely on the authenticity of a CA's web of trust, this web should be as small as possible. Several high profile cases of rogue certificate issuance in recent years [69,70] have raised questions about the security of these trusted servers [54]. Nearly every major CA has had a large leak of some sort. Certificate authorities can be socially engineered into surrendering certs to malicious actors. Their systems can be compromised or incorrectly configured and the system of CAs is liable to fall into capitalistic tendencies which may not be conducive for healthy certificate validation. Alternatives have been presented such as Convergence [71] which would replace certificate authorities with notaries which would ping destination servers to verify the validity of the desired path. Further discussion of the future validation methodologies for SSL is beyond the purview of this paper. While the CA infrastructure has dire consequences for SSL security, it isn't necessarily applicable to developer misuse of SSL. Along with the risky manual installation process which server administrators must carry out, the primary flaw in the current CA system which effects developers is the process of simply getting a certificate. Nearly all certificate authorities require a fee to receive a certificate [40]. This is not conducive to the widespread acceptance of HTTPS. Again, as noted by Zhang et al. following the fallout of Heartbleed [55], a majority of system administrators failed to revoke the certificates which had been reissued. This leaves these vulnerable certificates out in the wild to be used against hosts. The certificate authority industry needs to adopt standards which aid the easy, free access to certificates and a simple installation process. The security community must work with system operators to build a more intuitive revocation process. In the best case scenario, the introduction of a simpler method of certificate deployment and reset would secure the systems of critical Internet applications. Complications and vulnerabilities at the protocol level drip down to the designs of libraries and applications. In order to assure developers properly use SSL, the system must be no-more difficult to implement than a standard HTTP call.

3.6. Need for consumer awareness

Finally, among all of the guilty parties of HTTPS vulnerabilities, the most under-noted one is the end user. As the work of Felt et al. has shown with regards to web browsers, users frequently disregard warnings about SSL/TLS [72]. This issue is just as pressing on the Android platform. Another paper by Felt [73] explores Android user attention when shown messages on application permissions. Granting applications permissions is a critical responsibility which users must bear in order to protect their privacy and security. Unfortunately, when 308 users were interviewed, only 17% of participants paid attention to permissions during installation and only 3% of respondents correctly answered all three permission comprehension questions. What this means is that the populace views their applications as black boxes. Few users are critical of their applications enough to even read the warnings. This is worrisome since the pace of change is set by consumers. Unless end-users desire security, it will not be implemented in a widespread manner. When investigating user comprehension of HTTPS on Android, the numbers are equally bleak. An online survey by Fahl et al. shows that half of the 700 users questioned could not determine if they were using HTTP or HTTPS [24]. Many of the participants failed to read the entire warning message. Participants were mostly college-aged and included students majoring in IT-related and non-IT-related fields. Results showed that even this age-and-major group didn't have a sufficient understanding of data security. Despite the difficulty which comes with informing users of the risks of insecure TLS connections, it seems imperative that no group can push developers to properly implement HTTPS more than their end users. One subset of the lack of user comprehension is that Android does not offer any default warning for SSL errors. This forces developers to provide one for themselves if they wish to inform users about failed certificate validations [39]. Furthermore, error reporting in libraries and browsers is broken [60]. In a study by Brubaker et al. found that during an investigation of major browsers, many only reported one error even if there were more. It can be presumed that Android applications which have much less oversight than these browsers have even worse error reporting. Furthermore, the messages produced by libraries aren't always human readable and the application frequently does little to clarify the message. This leads to an uninformed end-user who clicks through warnings that seem unimportant. One notable mistake which hinders the reputability of SSL errors is the high number of false flags. A study conducted by Akhawe et al. [40] found that when analyzing SSL errors on a mass-scale that 1.54% were false warnings due to misreading the messages from the SSL API. In order for SSL to secure Internet communications, the end-user must remain vigilant of the state of their connection's integrity and the state of their application's security. Developers must work to insure that users are informed and able to check the security of the application itself. In the next section, we will present a list of solutions proposed by researchers and industry leaders and their ideas on combating broken SSL channels.

4. Proposed solutions

Several solutions to flaws in Android HTTPS security have been proposed in the aforementioned papers, the IETF, and security community as a whole, but a more extensive overhaul is needed to bring this burgeoning platform into the traditional level IND-CPA compliance [13]. Changing the way a developer works on a psychological level is not a feasible solution and remains out of the purview of this paper. Those seeking to make developers more astute at securing their SSL implementations must focus on support tools and resources. In this section, using ideas from other platforms and contexts, we will discuss a few of the ideas and subject areas which have seen solutions proposed.

4.1. Link SSL to DEBUGGABLE flag in the Android manifest

Based on research by Fahl et al. [24] and Georgiev et al. [37], Tendulkar et al. [8] suggest changes in the Android manifest to increase secure development practices. A single link between the DEBUGGABLE flag in the Android manifest and SSL/TLS verification would allow developers to build their apps using mock certificates while still eventually forcing them to make their SSL/TLS connections functional in production. Certificate checks would have to be intact if debugging was off, but the app could accept self-signed certificates if it was on. Applications submitted to the Market with the DEBUGGABLE flag would be rejected. This solution would directly counteract many of the issues with developers forgetting their debug code in their applications. This is a simple solution that should be implemented in the Android manifest. It is the most direct and effective solution to allow Android developers to write safe SSL communications enumerated in this paper. While it may require some coordination between the Android Market maintainers and developers who already have their applications live, the change would create a sensible workflow for all developers to follow.

4.2. Remove SSLErrorHandler

Tendulkar et al. also suggest removing the ability for developers to override SSLErrorHandler [8]. This prevents developers from hiding SSL error messages from the end user, forcing them to fix their code rather than obfuscate incorrect implementations. Coupled with disabling of SSL checks when the DEBUGGABLE flag is triggered, half of SSL vulnerabilities could be prevented [8]. This does, however, restrict the programmatic capability of developers and would make debugging much more challenging. Instead, warnings or errors could better announce the danger of leaving certificate checking out of the application.

4.3. Android market and client side application validation

Applications submitted to the Android market could be required to undergo scrutiny by MalloDroid [24] or the automated fuzzing framework noted in recent work by Malek et al. [74]. Fuzzing, also talked about in other SSL testing research [37], would test the number of accepted certificates from randomly generated data in a way like Frankencerts [60]. Applications with AllowAllHostnameVerifier [57] would be flagged. Another solution proposed by Enck et al. suggests Kirin [75] as a service within individual devices which would check applications for dangerous permissions and malicious code. This could be refitted into a service which also verifies that applications downloaded properly use HTTPS, flagging applications that use unsafe certificate verification methods or custom root stores. This device-based solution would not only protect a phone from the Android Market, but also apps in open-source repositories like F-Droid [76]. This solution would not prevent developers from writing non-HTTPS code, but would stop these applications from reaching production markets. Difficulties of implementing this include the setup and oversight required by market operators and the added restrictions placed on applications which may not deal in sensitive data.

4.4. Use SSL pinning

Another solution suggested by many researchers is the increased use of SSL Pinning. A pinning strategy using Trust on First Use (TOFU) retrieves the certificate from a website when it is first accessed. Every further connection compares the cached cert to the one which has been sent. This tactic is already used with success in SSH as Key Continuity Management. The user must trust the server on the first access, but all future sessions are more secure. The technology of HTTPS Strict Transport Security with pinning (HSTS) [77] forces all clients which wish to connect to a server to do so by HTTPS. Any situations where there is a mismatch will force a new session to be held to renegotiate certificates. This may be a hefty burden on bandwidth and may break services already in place. However, this technology would prevent attacks from rogue CAs, SSL stripping, session hijacking, and developers who do not use secure HTTP from connecting to secure servers. Coupled with other SSL protocol improvements, pinning could round out HTTPS, forming an effective foundation for Android applications that developers can rely on. SSL pinning relies on the validity of an initial connection and may become burdensome on the end user, leading to further manual overrides and apathy toward web security.

4.5. Improve documentation and API clarity

Developer misuse of SSL can be mitigated can also be mitigated through better documentation and clearer APIs. Cases of unclear APIs are specific and should be continually analyzed and reported by the community. Consistent error reporting by APIs can aid in a better understanding of how developers should solve for SSL instead of against it [37]. Code analysis techniques and intelligent suggestions could be adopted by IDEs to check SSL implementations in real-time [58]. Furthermore, Android platform maintainers with the help of the development and security communities need to create a more comprehensive and educational source of information on correct implementations of HTTPS in both testing and production systems. One small example is that the Android "Preparing to Release" checklist should include information about removing debug code in applications and ensuring the application does not accept self-signed certificates [65]. Improving documentation is as close to improving the developer paradigm as is possible. Updating documentation is not a trivial task and may take years to update across the entire ecosystem. However, new tools which aid in the development of SSL session creation should be sure not only to explain how to successful connection, but also the importance of properly approaching the debugging vs production certificate management situation.

4.6. Begin persistent Internet-wide SSL vulnerability scanning

Research conducted by the EFF, Durumeric et al. [50], Zhang et al. [55], Levillain et al. [52], and others following Heartbleed have shown that widespread scanning of the Internet for holes in the security of SSL are possible. These scans identify exactly how safe some hosts and servers are and potentially where attacks are originating from. As shown in the work of Durumeric et al. has presented, these scans also allow for researchers to notify server operators whose systems may be vulnerable to attack. This type of notification has proven effective in improving the safety of the Internet. A movement toward Internet-wide vulnerability scanning has positive implications for patching the security of the HTTPS. As new paradigms for SSL analysis are developed [51], the capacity for SSL adherence analysis becomes greater. Yet, the growing number of network flows across the internet will make tracing individual applications and versions more challenging. Any observatory must accurately and transparently identify applications that may be at risk and be able to notify the developers at once.

4.7. Patch and increase oversight on SSL libraries

In order to secure SSL implementations on every platform, SSL libraries and middleware with flaws in their validation and revocation checks mentioned in the previous section need to be patched. Design refactoring of current libraries should focus on hiding low-level code. A system similar to HTTPS Everywhere should be used in Android communication APIs [63,24]. As libraries are more and more utilized, their code needs to be scrutinized and brought into a state of security that is both forward-looking and all-encompassing. Large companies which handle critical data should be contributing efforts toward the improvement of open-source SSL libraries and protocols. Changes to libraries may roll out slowly across the SSL/TLS ecosystem, but they would become the cornerstone of safe Android web development.

4.8. Large mobile applications should use stronger HTTPS protections

In a more specific sense, large mobile applications made by Facebook, Amazon, and Google have the ability to verify their own certificates in mobile applications and detect MITM attacks [41]. Certain companies which can spare the bandwidth and application space should look into origin bound certs (OBCs) [41] and the use of HSTS [15]. These fixes are more of a server-side change than anything, but their use can secure popular mobile applications. Depending on the success of these systems, APIs for third-party apps which hook into company servers could also require clients to have certificates. This may prove challenging in scenarios where the application is closed source. However, even black box approaches [51] are able to identify certain patterns in SSL traffic that indicate unsafe SSL usage. The technology industry seems to understand the importance of SSL, but implementations of the strongest and arguably the most complex security measures in reality are rarely ideal.

4.9. Revise the TLS protocol suite

Beyond issues with developers, the TLS protocol needs to revised to allow for further progress in technological security. IETF RFC 2818, Section 3.1 [61], which deals with HTTP over TLS, needs to be revised to be more strict on validation guidelines. The protocol must require hostname and certificate validation and the community must adopt the strongest standard possible and implement it correctly. From there, applications which deal with user data can be built. The IETF [78] gives the following recommendations to certificate authorities and client developers:

- Move away from including and checking strings that look like domain names in the subjects Common Name.
- Move toward including and checking DNS domain names via the subject AlternativeName extension designed for that purpose: dNSName.
- Move toward including and checking even more specific subjectAlternativeName extensions where appropriate for using the protocol (e.g., uniformResourceIdentifier and the otherName form SRVName).
- Move away from the issuance of so-called wildcard certificates (e.g., a certificate containing an identifier for "*.example.com").

Furthermore, the X.509 needs revision [21]. In order to make way for CDNs, several amendments have been suggested, such as DNS-Based Authentication of Named Entities (DANE) [79,15]. In order to defend the protocol from resumption attacks, the suggestion made by Bhargavan et al. [59] is to create a new channel binding that would serve as a unique session hash. Master secrets thus benefit from this nonce. Also, secure resumption indicator which forces connections to check previous sessions is recommended. Stricter name constraints can define exactly who is receiving a certificate and make social engineering more difficult. However, certificate transparency (CT) represents a more promising proposal. It compiles a list of all existing certificates on the Internet. This allows for the public to view and investigate fraudulent certs issued in preparation for MITM attacks. This protocol, of course, relies on interested parties, like the EFF's SSL observatory to pay attention [66]. This could also help CAs determine the validity of cert requests. Combining CT and pinning would greatly increase security [67]. The openness of these systems will both spread awareness of SSL security, but hopefully spur further educational materials and human-friendly implementations. One other addendum to the protocol would establish a system that rather than allowing certificates to expire and throw a fatal error immediately, have certs warn the administrator for a week before throwing the error and more relaxed warnings [40]. CAs can use more specific revocation lists– some for normal expirations and some for blacklists [40]. These two solutions would stop a large number of false positive warnings which undermine the social comprehension of SSL. Finally, Android specifically could benefit from implementing a device-wide web security policy [80] which would guide its specific implementations of SSL to a strong standard. The TLS/X.509 protocol can benefit from dozens of new additions and specifications which will meet the needs of the applications which use it. The main limitation remains what direction would be the most sustainable solution moving forward, garnering both industry and academic support. While these increases in strictness will make the line drawn between secure and insecure architectures clear, it may pose an issue to developers and users who are looking for performance and availability over security. Suggestions for a warning escalation system may alleviate the pressure on developers and system administrators, much innovation and discussion remains before a proper certificate architecture that solves for both security and usability can be put in place.

4.10. Increase consumer awareness

Gaining user pressure on developers seems to be possible at this point to in only a few ways. If platform developers were to implement an effective non-HTTPS warning system in Android, the hands of developers would be pushed [24]. This would not alert all users, but it would alert those who are security conscious. Going further and preventing users from going to sites with misconfigured SSL/TLS forces developers to fix their authentication issues though it may inconvenience a user [72]. Less frequent and more accurate warnings may stop the end user from ignoring the errors since a user will obviously click through messages if they are bombarded with them [40]. The end user is most familiar with SSL when it gives them an error. These interactions need to be more meaningful and human-understandable. As with any security education, placing proper importance on SSL/TLS will require concrete examples and explanations about why sites with broken certificates should be avoided almost completely.

5. Discussion and future work

5.1. Gaps in research

This survey contains data from a wide range of academic papers and recent events; however, the topic of Android developer misuse of HTTPS is not solved. There are no interviews which answer the 'why' question or truly get into the developers' heads. Why don't developers fix the holes in their code noted in the work by Fahl et al. [39]? Why do developers use HTTP when using HTTPS may require only changing the URL in the call? The answers to these questions are currently conjectures. While steps are made to implement the solutions above, a deeper look into the psychological and potentially sociological underpinnings of SSL implementation is needed. Research into developer understanding of SSL documentation is lacking. While it is clear that their knowledge of cryptography may not be strong [17], there has been no research into the specific tools which developers need to create a secure HTTPS connection. There have been no tools, as of this writing, which check the security of HTTPS calls as they are written in an IDE. Much of existing research revolves around static code analysis of applications. Codebases that are not publicly available have generally not been included. Research is still wanting in how strongly closed source Android apps follow the TLS protocol and how these implementations compare to those seen in open source repositories. While the prevalence of MITM attacks in the wild has been investigated, the prevalence of MITM attacks against Android phones has not been studied on such a scale. Another area of SSL ecosystem research which is not clear is the effect of implementing HSTS and OBCs on a modern cellular or wireless network. Finally, there is little research in the field of an Android implementation of the DANE protocol and similar tools. Determining the effectiveness of these tools on the Android platform or as an extension in an Android browser is an important tool in deciding how to better secure HTTPS on Android and make it the standard protocol of the platform.

5.2. Moving forward

The previous sections opens up several ideas for next steps in research to prevent Android developer misuse of HTTPS. In this section, we will present some recommendations for future work. A productive solution to the issue of misinformation and SSL ignorance would be the creation of an online resource which exists as a single, accurate reference for the growing number of Android developers seeking to implement HTTPS in their applications. This solution would work with existing parties such as Android Developer Training and Stack Overflow to present credible and understandable information. A project in this field would include a primer on public key infrastructure, the proper usage of HTTPS, current attacks on SSL, a presentation of the most popular ways of implementing TLS on Android, and directions on how to acquire a server certificate. The presentation would be easy to read and include links to resources for further study and more specific problem solutions. A plugin to an IDE which would provide real-time feedback on the legitimacy of HTTPS calls could be developed in order to point out mistakes to developers. Similar to warnings which arise when using C's vulnerable strcpy, this plugin could then be tested for effectiveness at properly informing developers of their mistakes and the proper way to implement SSL. This plugin would need to return human readable and specific errors. An experimental plugin, emphaSSL [45] has been developed pursuant of this idea. The review of 75 open-source applications showed that 40% of the applications had significant violations of TLS protocol. This is concerning given the popularity of some of these applications and the sensitive data they transmit. Further research is needed to determine how effective feedback within the IDE is for developers and how to best present security suggestions during the product creation lifecycle. Fahl et al. [24] mention the implementation of their service MalloDroid as part of the Android Market or as a web application. In order to bring the benefits of Mallodroid to end users, an experimental service based off Mallodroid could be developed which would detect applications with vulnerable SSL connections and flag the program operator. This would fit into a model of an Android Market app or an end user device. This experiment would present either of these models with static code checking at its core and predict success rates. Furthermore, the work of Yao et al. [81] could be used by market administrators and network watchdogs to identify insecure traffic and identify the applications and specific software versions which are vulnerable. Individual flows could be analyzed for weak ciphers, expired and self-signed certificates, as well as completely plain-text packets. This would give greater oversight on existing network traffic at a more practical and automated level than static code analysis and would open up oversight to closed-source applications. Implementations of traffic fingerprinting and analysis have been conducted [51,52], which give great insights to the way that various Internet services handle SSL overall. As mentioned in the work of Georgiev et al. [37], several open source libraries could benefit from a reworded API and stronger documentation. Another project which could originate from this survey would be an effort to present clear method names and contributions to these open source libraries. This would require strong collaboration with security experts in the community and further research into psychological implications of programming syntax. One of the more specific solutions for Android which could come from work existing on a desktop scale would be the implementation of CDNSEC [82], a Firefox add-on that demonstrates the DANE protocol, as an Android service. While this would serve a very specific purpose based deeply in the work of Liang et al. [15] and not so much on the SSL comprehension for developers, it would be the first step toward adoption of DANE and in extension, forward-thinking SSL security, on multiple platforms. Furthermore, the development of Convergence [71], a CA-free certificate validation system on the Android platform would allow for the promising protocol to expand and test the implications of the overhead on mobile phones.

A less technical research project could be conducted in a similar manner to that of Xie et al.'s survey of developers [14], but focus on asking developers what their major challenges were in implementing HTTPS. Following the survey, the experimenters could look into the applications made by these developers to see how the SSL was implemented. Conclusions drawn from this would go into refining documentation, educational materials, and SSL libraries. Furthermore, developers could be presented with a situation which requires an HTTP call in their chosen language. The experimenters would then record the comprehension of the developer, whether or not web resources were used, and how well this implementation would withstand a MITM attack.

Again, a device-wide security policy could be proposed or discussed in further research which would encourage Android developers to adopt a standard set of security procedures and set a benchmark for SSL usage. This exists in diverse formats, but the presentation of a unified system would fulfill the call issued by Jeff Hodges and Andy Steingruebl for a web security policy framework [80], but with a particularly mobile lean.

The development of a sustainable Internet-scanning service for security researchers would allow for further research into the shortcomings that still exist within the HTTPS protocol in its current form. This tool would be available to researchers, commercial entities, and security organizations in order to find holes to patch. The outcome would be much like the work of Durumeric et al. following Heartbleed [50], extensive notification of vulnerable entities with the hopes that these systems would be patched quickly. Further solutions will certainly arise for the Android platform as research into new protocols, languages, and programming paradigms continues.

6. Conclusion

This paper has compiled current research on vulnerable HTTPS implementations and lacking protocols. We looked at the current shortcomings in the real-world application of end-to-end transport security, listed the areas in SSL which need improvement, and presented the current proposals of solutions to these areas. It is clear that Android developers remain unable to properly use the protocol in their applications due to reasons within and without their reach. Even in situations where the implementation is syntactically correct, there can be flaws along the chain of communication or in the CAs which back up the trust web of SSL. User-facing applications, SSL libraries, protocols, and infrastructure have limitations which must be further investigated. Most notably, the areas of education and support tools for developers require research and resolution. Evaluation of the solutions presented by the papers was brought together in this survey and including them into the next set of TLS/X.509 protocols and versions of SSL libraries will ensure user data integrity and security in a mobile environment that continues to see growth in the amount of confidential data transmitted. From these solutions, developers will benefit and this will trickle down to end users.

References

[1] IDC, Worldwide smartphone market grows 28.6% year over year in the first quarter of 2014, according to idc, Tech. rep., IDC, 2014. <http://www.idc.com/getdoc.jsp?containerId=prUS24823414>.

[2] J. Burns, Mobile application security on android, in: Black Hat USA, 2009.

[3] H. Wilcox, Press release: Juniper research forecasts total mobile payments to grow nearly ten fold by 2013, Tech. rep., Juniper Research, 2008. <http://www.juniperresearch.com/viewpressrelease.php?pr=106>.

[4] P. Ruggiero, J. Foote, Cyber threats to mobile phones, Tech. rep., 2011. <https://www.us-cert.gov/sites/default/files/publications/cyber_threats-to_mobile_phones.pdf>.

[5] H. Hwang, G. Jung, K. Sohn, S. Park, A study on mitm (man in the middle) vulnerability in wireless network using 802.1x and eap, in: IEEE ICISS, 2008.

[6] T. King, Packet sniffing in a switched environment, SANS Reading Room. <https://www.sans.org/reading-room/white-papers/networkdevs/packet-sniffing-switched-environment-244>.

[7] Revisiting ssl: a large scale study of the internet's most trusted protocol, Tech. rep., 2012.

[8] V. Tendulkar, Mitigating android application ssl vulnerabilities using configuration policies, North Carolina State University, 2013. http://repository.lib.ncsu.edu/ir/bitstream/1840.16/8840/1/etd.pdf.

[9] BLS, Occupational outlook: Computer programmer, Tech. rep., US BLS, 2012. <http://stats.bls.gov/ooh/computer-and-information-technology/computer-programmers.htm>.

[10] The Legion of Bouncy Castle, The Legion of Bouncy Castle. <http://bouncycastle.org>.

[11] OpenSSL Project, OpenSSL. <http://www.openssl.org/>.

[12] Security with HTTPS and SSL, android Developer Training. <https://developer.android.com/training/articles/security-ssl.html>.

[13] M. Egele, D. Brumley, Y. Fratantonio, C. Kruegel, An empirical study of cryptographic misuse in android applications, in: ACM CCS, 2013.

[14] J. Xie, H.R. Lipford, B. Chu, Why do programmers make security errors?, in: IEEE VL/HCC, 2011

[15] J. Liang, J. Jiang, H. Duan, K. Li, Kang, T. Wan, J. Wu, When https meets cdn: a case of authentication in delegated service, in: IEEE S &P, 2014.

[16] M. Egele, D. Brumley, Y. Fratantonio, C. Kruegel, An empirical study of cryptographic misuse in android applications, in: ACM CCS, 2013.
[17] K. Cairns, G. Steel, Developer-resistant cryptography, in: W3C/IAB STRINT, 2014.
[18] Y. Zou, X. Wang, L. Hanzo, A survey on wireless security: Technical challenges, recent advances and future trends, CoRR abs/1505.07919. < http://arxiv.org/abs/1505.07919 > .
[19] Microsoft, SSL/TLS in Detail. < http://technet.microsoft.com/en-us/library/cc785811.aspx > .
[20] The transport layer security (tls) protocol, rFC 5246. < http://tools.ietf.org/html/rfc5246 > .
[21] Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile, rFC 5280. < http://tools.ietf.org/html/rfc5280 > .
[22] X.509 internet public key infrastructure online certificate status protocol – ocsp, rFC 6960. < http://tools.ietf.org/html/rfc6960 > .
[23] Microsoft, How Certificates Work. < http://technet.microsoft.com/en-us/library/cc776447%28v = WS.10%29.aspx > .
[24] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, M. Smith, Why eve and mallory love android: an analysis of android ssl (in)security, in: ACM CCS, 2012.
[25] The internet engineering task force. < http://www.ietf.org/ > .
[26] Heartbleed bug. < http://heartbleed.com/ > .
[27] Debian security advisory 1571. < http://heartbleed.com/ > .
[28] Netcraft, April 2014 web server survey, Tech. rep., Netcraft, 2014. < http://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html > .
[29] I. Buitron-Damaso, G. Morales-Luna, Https connections over android, in: IEEE CCE, 2011.
[30] Trusted roots. < http://www.setupmobile.se/wp-content/uploads/2011/11/trusted_roots_ICS.txt > .
[31] Android, URLConnection. < https://developer.android.com/reference/java/net/URLConnection.html > .
[32] Oracle, JSSE. < http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html > .
[33] GnuTLS, GnuTLS. < http://gnutls.org/ > .
[34] Haxx, cURL. < http://curl.haxx.se/ > .
[35] Apache, HTTPClient – SSL Guide. < https://hc.apache.org/httpclient-3.x/sslguide.html > .
[36] Amazon, Amazon Flexible Payment Service. < https://payments.amazon.com/developer > .
[37] R. Anubhai, D. Boneh, M. Georgiev, S. Iyengar, S. Jana, V. Shmatikov, The most dangerous code in the world: validating ssl certificates in non-browser software, in: ACM CCS, 2012.
[38] Vulnerability note vu#582497: Multiple android applications fail to properly validate ssl certificates. < http://www.kb.cert.org/vuls/id/582497 > .
[39] S. Fahl, M. Harbach, H. Perl, M. Koetter, M. Smith, Rethinking ssl development in an appified world, in: ACM CCS, 2013.
[40] D. Akhawe, B. Amann, M. Vallentin, R. Sommer, Here's my cert, so trust me, maybe?: understanding tls errors on the web, in: WWW, 2013.
[41] L.S. Huang, A. Rice, E. Ellingsen, C. Jackson, Analyzing forged ssl certificates in the wild, in: IEEE S&P, 2014.
[42] Moxie Marlinspike, SSLSniff. < http://www.thoughtcrime.org/software/sslsniff/ > .
[43] M. Ongtang, S. McLaughlin, W. Enck, P. McDaniel, Semantically rich application-centric security in android, in: ACSAC, 2009.
[44] Cwe-489: Leftover debug code. < http://cwe.mitre.org/data/index.html > .
[45] X. Wei, M. Wolf, L. Geo, K.H. Lee, M. Huang, N. Niu, emphassl: towards emphasis as a mechanism to harden network security in android apps, in: IEEE GLOBECOM, 2016.
[46] Stack overflow. < http://stackoverflow.com/ > .
[47] Stack overflow: Ssl – untrusted certificate error. < http://stackoverflow.com/questions/2642777/trusting-all-certificates-using-httpclient-over-https > .
[48] Setting up Apache HTTP Server with SSL support on Ubuntu/Debian. < http://softwareinabottle.wordpress.com/2011/12/18/setting-up-apache-http-server-with-ssl-support-on-ubuntudebian/ > .
[49] Apache, SSL/TLS Strong Encryption: How-To. < http://httpd.apache.org/docs/2.4/ssl/ssl_howto.html > .
[50] Z. Durumeric, J. Kasten, D. Adrian, J.A. Halderman, M. Bailey, The matter of heartbleed, in: ACM IMC, 2014.
[51] M. Korczynski, A. Duda, Markov chain fingerprinting to classify encrypted traffic, in: IEEE INFOCOM, 2014.
[52] O. Levillain, A. Ebalard, B. Morin, H. Debar, One year of ssl internet measurement, in: ACM ACSAC, 2012.
[53] H. Lee, T. Malkin, E. Nahum, Cryptographic strength of ssl/tls servers: current and recent practice, in: ACM IMC, 2007.
[54] N. Vratonjic, J. Freudiger, V. Bindschaedler, J. Hubaux, The inconvenient truth about web certificates, in: WEIS, 2011.
[55] L. Zhang, D. Choffnes, D. Levin, T. Dumitras, A. Mislove, A. Schulman, C. Wilson, Analysis of ssl certificate reissues and revocations in the wake of heartbleed, in: ACM IMC, 2014.
[56] E. Chin, A. Felt, K. Greenwood, D. Wagner, Analyzing inter-application communication in android, in: ACM MobiSys, 2011.
[57] Apache, AllowAllHostnameVerifier. < http://developer.android.com/reference/org/apache/http/conn/ssl/AllowAllHostnameVerifier.html > .
[58] A.J. Ko, B.A. Myers, H.H. Aung, Six learning barriers in end-user programming systems, in: IEEE VL/HCC, 2004.
[59] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, P. Strub, Triple handshakes and cookie cutters: breaking and fixing authentication over tls, in: IEEE S&P, 2014.
[60] C. Brubaker, S. Jana, B. Ray, S. Khurshid, V. Shmatikov, Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations, in: IEEE S&P, 2014.
[61] Http over tls, rFC 2818. < http://tools.ietf.org/html/rfc2818 > .
[62] Http strict transport security (hsts), rFC 6797. < http://tools.ietf.org/html/rfc6797 > .
[63] EFF, HTTPS Everywhere. < https://www.eff.org/https-everywhere > .
[64] Http strict transport security hsts. < http://scratchingsecurity.blogspot.com/2013/06/http-strict-transport-security-hsts.html > .
[65] Android, Preparing for Release. < http://developer.android.com/tools/publishing/preparing.html > .
[66] An observatory for the ssliverse. < https://www.eff.org/files/DefconSSLiverse.pdf > .
[67] Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations, in: ACSAC, 2013.
[68] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, K.R.B. Butler, Forced perspectives, in: ACM IMC, 2014.
[69] P. Hallam-Baker, Comodo ssl affiliate the recent ra compromise, Comodo Blog. < https://blogs.comodo.com/uncategorized/the-recent-ra-compromise/ > .
[70] Microsoft security advisory 2607712. < https://technet.microsoft.com/en-us/library/security/2607712.aspx > .
[71] Convergence. < http://convergence.io/ > .
[72] A. Felt, R. Reeder, H. Almuhimedi, S. Consolvo, Experimenting at scale with google chrome's ssl warning, in: ACM CHI, 2014.
[73] A. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, D. Wagner, Android permissions: user attention, comprehension and behavior, in: ACM SOUPS, 2012.
[74] S. Malek, N. Esfahani, T. Kacem, R. Mahmood, N. Mirzaei, A. Stavrou, Packet sniffing in a switched environment (2012). < http://mason.gmu.edu/nesfaha2/Publications/SERE2012.pdf > .

[75] W. Enck, M. Ongtang, P. McDaniel, On lightweight mobile phone application certification, in: ACM CCS, 2009.
[76] F-Droid, F-Droid. < https://f-droid.org/ > .
[77] Certificate pinning extension for hsts, updated http://tools.ietf.org/html/rfc6797. < http://www.ietf.org/mail-archive/web/websec/current/pdfnSTRd9kYcY.pdf > .
[78] 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), rFC 6125. < http://tools.ietf.org/pdf/rfc6125.pdf > .
[79] The dns-based authentication of named entities (dane) transport layer security (tls) protocol: Tlsa, rFC 6698. < http://tools.ietf.org/html/rfc6698 > .
[80] The need for a coherent web security policy framework, 2010. < http://www.w2spconf.com/2010/papers/p11.pdf > .
[81] Y. Hongyi, R. Gyan, A. Tongaonkar, L. Yong, M. Zhuoqing Morley, Samples: self adaptive mining of persistent lexical snippets for classifying mobile application traffic, in: ACM MobiCom, 2015.
[82] GitHub, DANE4CDN. < https://github.com/cdnsec/dane4cdn > .