MITHYS: Mind The Hand You Shake

Mauro Conti[1], Nicola Dragoni[2], and Sebastiano Gottardo[1,2]

[1] University of Padua, IT
conti@math.unipd.it,sgottard@studenti.math.unipd.it
[2] Technical University of Denmark, DK
ndra@dtu.dk,s124645@student.dtu.dk

Abstract. Recent studies have shown that a significant number of mobile applications, often handling sensitive data such as bank accounts and login credentials, suffers from SSL vulnerabilities. Most of the time, these vulnerabilities are due to improper use of the SSL protocol (in particular, in its *handshake* phase), resulting in applications exposed to man-in-the-middle attacks. In this paper, we present MITHYS, a system able to: (i) detect applications vulnerable to man-in-the-middle attacks, and (ii) protect them against these attacks. We demonstrate the feasibility of our proposal by means of a prototype implementation in Android, named MITHYSApp. A thorough set of experiments assesses the validity of our solution in detecting and protecting mobile applications from man-in-the-middle attacks, without introducing significant overheads. Finally, MITHYSApp does not require any special permissions nor OS modifications, as it operates at the application level. These features make MITHYSApp immediately deployable on a large user base.

1 Introduction

The spread of mobile smartphones have led web service providers to pay attention to how the users could benefit from their services, while users are on the move. To this end, two main approaches have been adopted. At first, providers chose to offer a mobile-shaped version of their web service, which the users could access through a mobile web browser (acting as a “thin” client). As an alternative, providers started to offer their services by means of native applications for each specific mobile platform (also called “fat client” approach). This second approach rapidly became the most popular (interested readers can refer to [7] for a thorough comparison between the two approaches). Indeed, as the number of daily activated devices grows at a relentless rate, so does the number of applications which are downloaded and available to a huge end-user base.

An application that relies on a web service requires an active Internet connection. To gain this connection, a mobile device is typically equipped with two types of network interfaces: a 3G/4G module and a Wi-Fi module. The Wi-Fi module gives the user the opportunity of connecting a device to a wireless network created through a wireless access point. The Wi-Fi connection became more and more important, as many companies started offering free Internet access points, as an additional service for their customers. We can also find this scenario in many public infrastructures, such as libraries and universities. Unfortunately, this increasing popularity of free access points has led to new malicious attacks, based on the Man-In-The-Middle principle (from now, MITM attack). The *rogue access point attack* is a typical example of how dangerous the use of a free public access point might be [17]. As a consequence, protecting the communication in these open environments is crucial to keep user data private. This means that a mobile device must establish a secure connection with the remote server offering the needed web service. In a desktop environment, this connection lies between the web browser and the remote server. On the other hand, a mobile application is directly responsible of establishing the secure connection with the remote server, without relying on a web browser.

Technically speaking, the most common way of establishing a secure connection is by using Secure Sockets Layer (SSL) [1] and Transport Layer Security (TLS) [18], two cryptographic protocols that grant endpoint authentication and network data confidentiality over a TCP connection. These protocols were also designed to prevent malicious MITM attacks against two communicating entities. The problem is that, as recently pointed out [10], a significant number of mobile applications often do not perform the required steps to ensure a secure communication between the communicating parties. The flowing data between the application and the server, which is supposedly private, can be intercepted by a malicious third party by performing a MITM attack. This is a known problem that affects a huge number of mobile applications, mainly due to the respective developers that underestimate the importance of a proper use of the SSL/TLS protocols. Even if the problem has been raised more than one year ago, our recent test revealed that several applications (including widely used ones, such as PayPal and Facebook) are still vulnerable.

Example 1. *Let us assume a scenario where an attacker performs a rogue access point attack, with Starbucks' free Wi-Fi service as a target. The original Starbucks' access point (AP from now on) name is "Starbucks", while the attacker's AP name is "Starbucks Free". Let us suppose Alice visits Starbucks and notices the free Wi-Fi opportunity. She sees two open access points on her Android smartphone, so she chooses a random one, the attacker's "Starbucks Free" in this case. Alice wants to check her PayPal account, therefore she opens the PayPal Android application, which she had used before. Since the PayPal application suffer from the above SSL usage problem, the attacker is able to intercept Alice's PayPal account data, including her personal login information. What is more, she is not aware that she is a victim of a MITM attack.*

Again, given the huge number of vulnerable applications, the "wait-and-hope" approach is not appropriate, since it exposes the users to malicious MITM attacks until the developers release a security update. Instead, there is the need for an application-independent solution that: (i) detects the vulnerable applications; (ii) warns the user about the potential leak of sensitive data; and (iii) eventually compensates the lack of security by performing the adequate checks. Such a solution would not only secure the application-web server communication, it would also act as a security tool for mobile developers — who want to test the security level of their applications against SSL-based MITM attacks.

Contribution. In this paper we present MITHYS (Mind The Hand You Shake), a platform independent system architecture that:

- Detects mobile applications vulnerable to SSL-based MITM attacks, automatizing the detection of vulnerabilities pointed out in [10],[11])
- Protects mobile applications (especially, vulnerable ones) from SSL-based MITM attacks, by taking care of SSL certificate validation
- Gives the user full control on the vulnerable applications' behavior (e.g. the application can be blocked if vulnerable)

The MITHYS architecture is, to the best of our knowledge, the first solution that tackles the vulnerability of mobile applications to SSL-based MITM attacks [10],[11]. A fully-working, end-user-ready implementation of MITHYS, namely MITHYSApp, has been developed for the Android mobile platform, which represents one of the most flexible and popular mobile OS at the time being.

Being implemented at the application level, MITHYSApp does not require mobile OS alterations nor special permissions (i.e., root access). MITHYSApp just relies on a single manual configuration performed by the user. According to the selected configuration, MITHYSApp can operate in three modes:

- Automatic - detection of vulnerable applications and protection for all the installed applications, without requiring any user interaction;
- Selective - detection of vulnerable applications is automatic, but the user can decide whether to allow their execution or not;
- Manual - the user can manually select which applications must be analysed and which must be protected.

Finally, a set of experiments show the feasibility of our solution. In particular, we show that the current (non-optimized) version of MITHYSApp does not introduce a significant delay in network communication nor in the ordinary applications/OS behavior, while it effectively protects users from MITM attacks that can steal personal and sensible information.

Roadmap. Section 2 discusses related work. Section 3 introduces the details of the security problem we solve. Section 4 presents MITHYS, our solution for protecting mobile applications vulnerable to MITM attacks. Section 5 focuses on the implemetation of MITHYSApp. Section 6 evaluates our solution in terms of effectiveness and network delay. Finally, Section 7 concludes the paper.

2 Related Work

Today's smartphones are capable of handling different types of personal data, which most of the times can be considered sensible. As a result, smartphones security is becoming more and more a key topic in the security research community, generating a lot of studies about dangerous threats and possible solutions (as shown by the proceedings of recent top conferences on security, such as ESORICS, POLICY and CCS). Considering only the Android case and to mention only a few papers, Davi et al. [9] presented an analysis of the privilege escalation attacks, together with some possible approaches to the problem [5], [6]. Becher et al. [3] gave a more general security overview about the mobile smartphones environment, whereas Shabtai et al. [16] focused more deeply in an Android security assessment. Other works focused on the direction of extending Android security features: e.g. considering Context-based access control [8] and enforcing different modes of uses based on security profiles [15]. To mention all the papers aiming at securing Android is out of the scope of this paper. What we consider instead important to point out is that, although this increasing research effort, a significant work has still to be done in order to secure smartphone platforms. This is proved by the huge vulnerability recently discovered regarding the use of the SSL cryptographic protocol.

Various misuses of the SSL protocol are spread both in the desktop environment and in the mobile environment, exposing private data (potentially sensible) to malicious attacks. In particular, Georgiev et al. [11] analysed the SSL usage across various environments, only to find out that this protocol's implementation is "completely broken in many security-critical applications and libraries". Meanwhile, Fahl et al. [10] analysed the SSL usage on 13,500 Android applications, and found out that a large percentage of them suffer from SSL vulnerabilities, which expose them to dangerous man-in-the-middle attacks. To add it up, some of these applications (such as PayPal and Facebook) are very popular, covering up to 185 million users. Both studies just gave some advices to developers, but did not mention any solution to the SSL usage problem.

SSL misuse vulnerabilities have been also considered in the literature. For example, the work in [4] shows an approach to detect SSL-based man-in-the-middle-attacks. However, this approach is designed for desktop web browsers, so it is not suitable for the setting of mobile applications that we are considering in this work. Furthermore, a simple MITM attack towards the third-party server proposed in [4] completely invalidates their protection mechanism. This problem is also acknowledged by the authors in their work.

Despite the size of the problem, the SSL usage vulnerability problem for mobile applications is still out there, threatening millions of users and their private data. We will focus on this problem in the next Section.

3 The Problem: Validating SSL Certificates

Nowadays Internet browsers, electronic mail clients, instant messaging clients, and nearly every entity that needs a secure communication to a remote service are using SSL and TLS, two standard cryptographic protocols that perform network data encryption and endpoint authentication over a TCP connection. An SSL secure communication begins with an operation called *handshake*, in which the server is authenticated by the client (and viceversa, eventually). After that, these two entities agree on a common cryptographic material, used to begin the encrypted communication. This flow can be roughly summarised as follows (we are not considering the client authentication steps, which are optional):

1. The client contacts the server, and they exchange some preliminary parameters, among which the certificates (the client's certificate is optional, therefore often missing); the exchanged parameters are called context of a SSL session.
2. The client authenticates the server by using the information obtained in the previous step, especially the server's certificate; for a secure session to be established, the server must be successfully authenticated by the client (either implicitly or explicitely).
3. The client, thanks to the previous information exchange, creates a pre-master secret, encrypted with the server's public key obtained from the server's certificate, and sends it to the server.
4. The server decrypts the message and uses the pre-master secret to compute the master secret while the client does the same.
5. Using the master secret, both the client and the server generate the so called session keys, that will be used to communicate securely.
6. The communication starts as the client sends the first encrypted message.

There is a slight problem on the second point of the above flow. The client must authenticate the server in order to be sure that it is communicating with the right server and not with, for instance, a malicious one which is faking its identity (a typical MITM situation). This is mostly done by thoroughly checking the server's SSL certificate fields (e.g., expiration date, issuer, signature).

Example 2. *Continuing the scenario described in Example 1, let us suppose Alice is using PayPal's Android application (PayPalApp), which needs to communicate with PayPal's remote server (PayPalServer). However, the attacker (MITM) is able to intercept the ingoing and outgoing traffic of PayPalApp. The following steps are performed as part of the SSL handshaking process:*

1. *PayPalApp queries PayPalServer for its X.509 certificate (which contains PayPalServers's public key).*
2. *MITM intercepts PayPalApp's request and asks PayPalServer for its certificate pretending she is PayPalApp; PayPalServer sends its certificate to MITM.*
3. *MITM now generates a fake X.509 certificate containing MITM's public key instead of the PayPalServer's one; MITM also makes this fake certificate look like PayPalServer's one, then sending it back to PayPalApp.*
4. *Depending on how strict are PayPalApp's checks against MITM's certificate, PayPalApp will eventually think that she's talking to PayPalServer.*
5. *At this point, MITM can intercept the plain text of every message (i.e., MITM can easily decrypt the messages) PayPalApp sends to PayPalServer and viceversa, but she is undetected.*

In Example 2, *PayPalApp* performs very poor checks against MITM's certificate (e.g., it might not check the issuer name of the certificate, therefore not recognizing a MITM attack). As a result, Alice is not able to detect that the communication with *PayPalServer* is not secure at all, allowing MITM to intercept all the available data. It is important to stress that this is not just a toy example, we have actually developed a demo implementing this specific attack.

It is clear by now that the key point of this procedure consists in validating the server's certificate in a proper way. Since many mobile applications do not perform this step correctly, exposing the end-user to dangerous MITM attacks, our solution focuses on solving this specific problem.

4 MITHYS: Mind The Hand You Shake

In this section, we present MITHYS (Mind The Hand You Shake), a system designed to detect potentially MITM-vulnerable applications, and to compensate the lack of security by protecting applications from MITM attacks. To the best of our knowledge, MITHYS represents the first solution that tackles the MITM vulnerability of mobile applications by taking on the security checks required to establish a proper secure connection. For space limitation, we omit details on MITHYS user interface and configuration. Instead, we focus on the core of MITHYS and we describe it from a system point of view, focusing on its architecture, its implementation (Section 5) and its evaluation (Section 6).

The main idea behind MITHYS is to act as a friendly MITM on the mobile device. Every time a "new" application (an application which has not been tested yet) requests a resource via the HTTP over the SSL protocol (from now on, HTTPS requests), the MITHYS system tries to act as a man-in-the-middle, forging a fake ad-hoc SSL certificate for the application. If the application is not vulnerable, it will immediately block the communication; otherwise (the application is vulnerable), the communication will proceed normally, as if there is no third party between the application and the remote server. In both scenarios, MITHYS is able to protect the application from potentially malicious MITM attacks by performing additional checks on the SSL connection (Section 4.3).

An high-level overview of the MITHYS architecture is shown in Figure 1. At a macroscopic level, there are two main components, highlighted in the figure by thicker borders. The first one is called *MITHYS Proxy*, a proxy-based mobile application that runs on the mobile device. The second one is called *MITHYS WebServer*, a remote web server hosted and reachable through the Internet.

We now describe the two key components of MITHYS: MITHYS WebServer (Section 4.1) and MITHYS Proxy (Section 4.2). Then, in Section 4.3 we describe how the overall system works.

4.1 MITHYS WebServer

This component acts as a trusted party for the solution. It features only one servlet, whose purpose is to retrieve the SSL certificates chain (typically in the

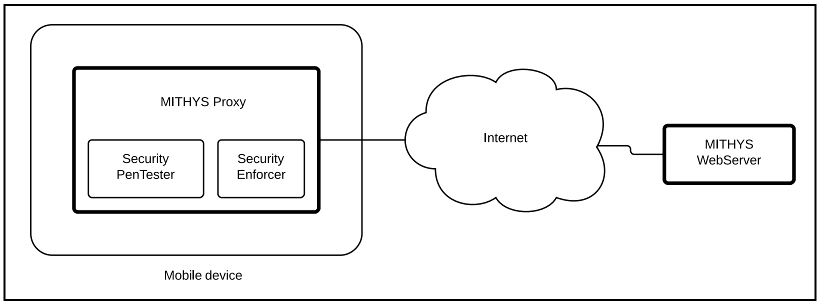


Fig. 1. The MITHYS high-level architecture

X.509 standard) of the URL passed as an argument; then, it serializes the chain in a proper way and returns it as a result. This servlet is only reachable via HTTPS, meaning that it has a SSL certificate associated to it. This is a key point of the whole architecture. This SSL certificate is self-signed, i.e. generated from the root certificate of our private Certification Authority (i.e., *MITHYS CA*). Since we have access to the original certificate, we can use its information to add an extra layer of security against MITM attacks, as we discuss in Section 4.2. Finally, we underline that we do not consider this component as a possible target for attacks, mainly because (i) it can be hosted on highly secure cloud services (e.g., Google Compute Engine) and (ii) it is easier to protect this single component rather than protecting millions of user devices with an highly variable set of installed applications. However, in order to prevent Denial-of-Service (DoS) attacks, we recommend the redundancy approach, by means of a MITHYS WebServer pool.

4.2 MITHYS Proxy

This represents the main component of the architecture. Its main purpose is to receive all the HTTPS requests coming from the applications installed on the mobile device, and to pass the information back and forth between the application and its associated web server. It can also strengthen the applications' security by performing additional checks (as detailed later in this section) on the SSL connection. In order to fulfill its tasks, it features two independent modules (see Figure 1): *Security PenTester* and *Security Enforcer*.

Security PenTester. This module is the component which represents the actual MITM. It impersonates the original remote server by forging a fake SSL certificate for the mobile application. It also contacts the original remote server, pretending to be the application itself. If Security PenTester is able to establish a secure connection with the application (that is to say, the application accepts the fake SSL certificate), it acknowledges that the application is vulnerable. Otherwise, we can only have some degree of confidence that the application is not vulnerable, while it could be actually vulnerable in other circumstances. This module runs continuously, so every application is basically tested every time it

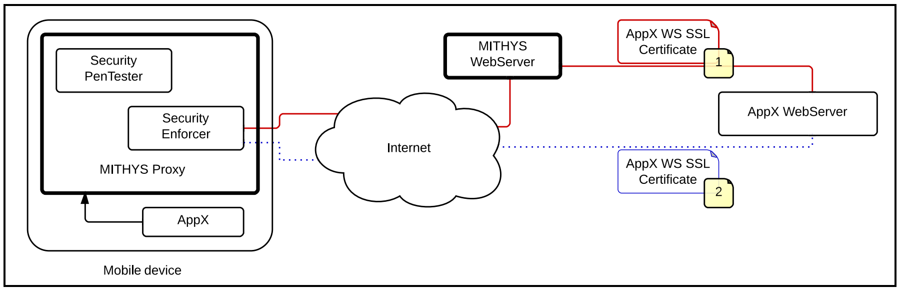


Fig. 2. The MITHYS Security Enforcer interaction scheme.

issues an HTTPS request. Since we want *"PenProof"* applications (i.e., applications that are not vulnerable to the PenTester) to be excluded from further security tests, an effective approach consists in adding them to a whitelist: every application on that list avoids the Security PenTester module, but may still be strengthened by the Security Enforcer module.

We want to point out that the use of a whitelist is actually mandatory. A PenProof application that receives a fake SSL certificate for an HTTPS request will terminate the connection immediately, therefore not working correctly. As a consequence, the MITHYS system needs to be aware of the already (successfully) tested applications, so that we do not hinder their normal operations.

Security Enforcer. This module performs additional checks on the SSL connection to the remote server in place of the mobile application. More specifically, given the HTTPS request issued by AppX (an installed application), this module performs the following operations (illustrated in Figure 2):

- Issues an HTTPS request to the MITHYS WebServer, in order to retrieve the SSL certificates chain associated to the URL of the application's HTTPS request (Step 1 in the figure);
- Retrieves the SSL certificates chain associated to the URL of the HTTPS request (Step 2 in the figure);
- Compares the two certificates chains. Each certificate of one chain is compared to the respective certificate of the other chain. This is done by checking if the signatures of the two certificates correspond.

If the certificates contained in the two chains do not match, it means that a MITM attack might be in place. On the other hand, if the two chains have a 1:1 match, we can be sure that no SSL-based MITM attack is being held at that time. This assumption is based on the fact that the HTTPS request to the MITHYS WebServer is MITM-proof. To achieve such requirement, since the SSL certificate of our MITHYS WebServer is known a priori, we can store it on a keystore and embed it in our MITHYS Proxy mobile application. So, when the HTTPS request to the MITHYS WebServer is issued, the obtained SSL certificate is matched against our keystore: any failure will invalidate the certificates chains comparison, indicating an ongoing MITM attack of some kind. It is worth pointing out that an application which has passed the Security PenTester's controls might still be monitored by the Security Enforcer (e.g., as an extra security measure for the user). What is more, Security Enforcer only sends to MITHYS WebServer the URL of the original HTTPS request, without transmitting any sensitive information of the user.

4.3 MITHYS Workflow

In order to better understand how the overall MITHYS system works, Figure 3 shows a simplified workflow of a generic scenario where the mobile application AppX issues an HTTPS request (e.g., to *https://www.appx.com/api/login*). The request is intercepted by our MITHYS Proxy, that checks whether the application has ever been whitelisted. If not, Security PenTester tries to act as a MITM and determines if AppX is aware of a third entity between AppX's remote server and itself. If the application is aware of the MITM, it is whitelisted: each subsequent HTTPS request coming from that application will be executed as is, without any interception. Otherwise, Security Enforcer is activated in order to prevent any malicious MITM attacks. Again, note that even a whitelisted application might take advantage of the latter module, if specified by the user.

Example 3. *Back to our running example, let us consider Example 2 to show the workflow of MITHYS with PayPal's Android application. The key assumption is that Alice is using a MITHYS implementation on her smartphone. Alice starts the PayPalApp, which in turn issues HTTPS requests to the PayPalServer. These requests are intercepted by MITHYS' Security PenTester (PenTester from now on). PenTester retrieves the list of whitelisted applications to check if PayPalApp is among those. The whitelist is initially empty, so PenTester acts as a SSL MITM and forges a fake SSL certificate. PayPalApp, as we show in Section 6.1, is vulnerable to this attack, so it accepts the certificate. Now that PenTester has acknowledged that PayPalApp is vulnerable, it reports this information to the MITHYS' Security Enforcer module (Enforcer from now on). Enforcer must now protect PayPalApp from actual MITM attacks by performing the steps described in Section 4.2. What is more, Enforcer will protect all the future PayPalApp's HTTPS requests.*

Example 4. *We reconsider Example 3, but we assume that this time Alice wants to use the Twitter application, which is not vulnerable to SSL MITM attacks (Section 6.1). Again, Alice is using a MITHYS implementation. Alice starts TwitterApp, PenTester intercepts the HTTPS requests to TwitterServer and tries to act as a SSL MITM for TwitterApp. The latter is not vulnerable, so it will reject the fake SSL certificate and abort the current operation. Now PenTester knows that the application is secure, so it adds TwitterApp as a new whitelist entry. TwitterApp can operate without the Enforce protection, but the user might want to be protected anyway. If this is the case, Enforcer will protect all the future TwitterApp's HTTPS requests. Otherwise, it will simply forward the HTTPS requests/responses between TwitterApp and TwitterServer.*

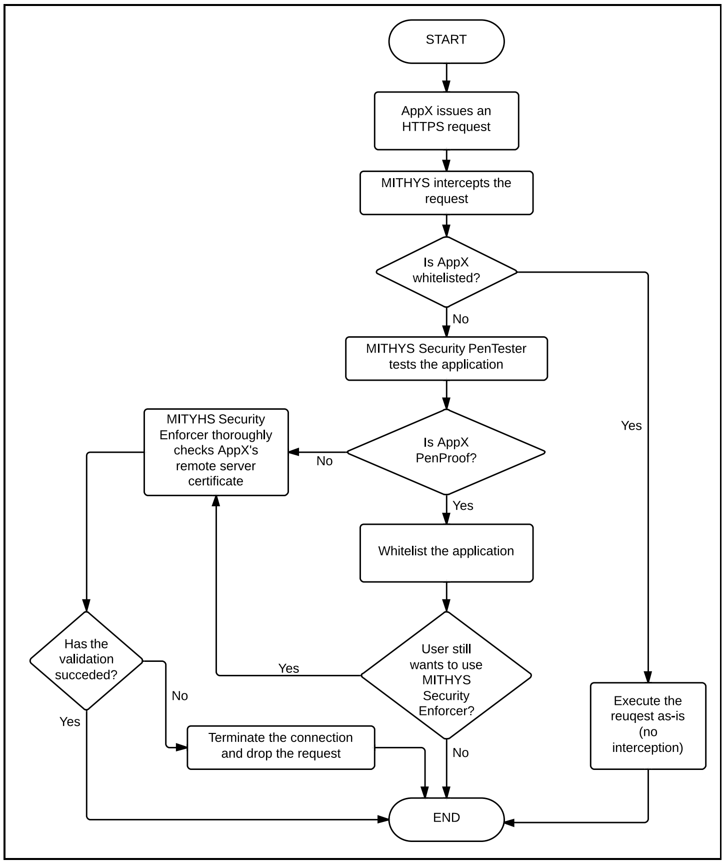

START
AppX issues an HTTPS request
MITHYS intercepts the request
Is AppX whitelisted?
No
Yes
MITHYS Security PenTester tests the application
MITYHS Security Enforcer thoroughly checks AppX's remote server certificate
No
Is AppX PenProof?
Yes
Whitelist the application
Has the validation succeded?
Yes
No
User still wants to use MITHYS Security Enforcer?
Yes
No
Terminate the connection and drop the request
Execute the reuqest as-is (no interception)
END

Fig. 3. Workflow of the MITHYS architecture with the AppX mobile application.

5 Implementation of MITHYS: MITHYSApp

This section discusses our implementation of MITHYS, namely the MITHYSApp Android application which acts as the MITHYS Proxy component. The MITHYS WebServer consists in a Micro Instance of Amazon's Elastic Compute Cloud Web Services (AWS EC2) [2]: a continuously running Apache Tomcat instance serves an HTTPS-only Java servlet called `GetSSLCertificate`.

5.1 The MITHYSApp WebServer

MITHYSApp WebServer implements the MITHYS WebServer component. It is hosted on Amazon Elastic Compute Cloud (Amazon EC2) [2] as part of the Amazon Web Services. A Micro Instance of the EC2 cloud, which we can consider as a proper Virtual Private Server (VPS), runs the Apache Tomcat web server and servlet container. There is only one servlet, called `GetSSLCertificateServlet` that takes in input two arguments: the first one is the target URL, the second one is the HTTP method that should be used to invoke that URL. This servlet simply issues an HTTPS request to the target URL (accordingly to the HTTP method) and retrieves the SSL certificates chain associated to that URL. The Base64 serialization of the chain is returned as a JSON-formatted result. Please note that this servlet is only available via HTTPS, and it uses an SSL certificate generated from our MITHYS Certification Authority (MITHYS CA) in order to prevent MITM attacks against our MITHYSApp application.

5.2 The MITHYSApp Android Application

MITHYSApp is an Android app that implements the MITHYS Proxy component. It relies on the open source Android library SandroProxyLib[3], which is based in turn on the OWASP WebScarab project, that offers a working-out-of-the-box proxy for Android. What is more, it behaves as the MITHYS Security PenTester by default due to the fact that, every time it receives a new HTTPS request, it acts as a MITM and forges ad-hoc fake certificates. These certificates are generated from the MITHYS CA, and their hostname matches the hostname of the target server, looking similar to the original ones. From now on we will use also the term “proxy” to refer to the proxy part of this library. While not requiring any special permission or OS modifications, MITHYSApp requires the installation of the MITHYS CA certificate and the setup of the proxy address for the current Wi-Fi connection. MITHYS guides the user in both these steps, both performed only once at installation time.

Security PenTester. We had to modify and to extend the SandroProxyLib library in order to implement the above component correctly. First of all, given an intercepted HTTPS request, we need to know which application generated it: in terms of Java objects, we only have a `Socket` instance that represents the connection between the application and the proxy, of which we only know the port. But, since Android is a Linux-based OS, we can read the content of the */proc/net/tcp* (or */proc/net/tcp6* if an IPv6 address is available) file that maps all the active sockets to their Unix processes: in this way we know which port is being used, so we can obtain the UID of the process which is using that port. This information, together with the `PackageManager.getPackagesForUid(uid)` method provided by Android, offers us the possibility of knowing which application issued the HTTPS request given just the port of its `Socket` object. To the best of our knowledge, this is the only technique available at the time being, so we created a small and useful Android library[4] which eases this process for the developer. Another modification to the proxy library consisted in introducing the whitelisting mechanism, so that each time an installed application refuses to establish a secure connection with the proxy (that is, the SSL handshake phase between our proxy and the application cannot be completed) it communicates

[3] `https://github.com/SandroB/sandrop/tree/master/projects/SandroProxyLib`

[4] `https://github.com/dextorer/AndroidTCPSourceApp`

the non-vulnerable application to MITHYSApp. To do so, an `AppDescriptor` object containing package name, application version and requested URL is created and sent to the running instance of MITHYSApp. The latter receives the `AppDescriptor` object and inserts its values on a local SQLite database. This database must be encrypted in order to prevent manual tampering, so we used a custom Android library called SQLCipher[5] to provide "transparent 256-bit AES encryption of database files". In addition, for each new HTTPS request the proxy checks if the application who issued it has been whitelisted before, by querying the SQLite database: if so, no interception is made and the proxy simply passes the data back and forth between the whitelisted application and the remote server. In addition, in order to prevent alterations to the local MITHYS keystore, we invoke a JNI-compiled library that checks the current Java package name and the keystore size. Thanks to this approach, any attempt to (i) replace the native library, to (ii) modify the Java code of MITHYSApp or even to (iii) replace the keystore will lead to a non working application.

Security Enforcer. In order to implement the Security Enforcer module, we had to extend the SandroProxyLib library so that, every time a vulnerable application issues an HTTPS request, the proxy performs the following steps:

1. Retrieves the SSL certificates chain associated to the URL of the HTTPS request.
2. Issues an HTTPS request to the MITHYSApp WebServer, in order to retrieve the SSL certificates chain associated to the URL of the application's HTTPS request.
3. Compares the two certificates chains, as described in Section 4.2.

If no MITM attack is in place, the comparison will succeed and the HTTPS request will be issued without further ado. If a MITM attack is in place, the HTTPS request issued towards the MITHYSApp WebServer will simply fail (as we explained in Section 4.2). A smarter attacker might decide not to intercept the HTTPS requests addressed to our MITHYSApp WebServer: but this won't prevent our Security Enforcer module from detecting a MITM attack, since the two certificates chains are still compared one against the other.

6 System Evaluation

In this section, we present a set of tests that assess the performance impact of the MITHYS approach and determine its ability to successfully detect vulnerable applications. More specifically, we want to show that, although MITHYS requires additional HTTPS requests in order to protect the mobile device from MITM attacks, the user is not dramatically affected by this overhead. First, we will analyse the effectiveness of MITHYSApp's vulnerability detection in Section 6.1. Then, in order to determine the additional overhead, we will discuss our test method in Section 6.2 and the results in Section 6.3.

[5] `https://guardianproject.info/code/sqlcipher/`

6.1 Vulnerability Detection

In their analysis, Fahl et al. [10] manually audited some of the most popular Android applications, in order to test their vulnerability to SSL-based MITM attacks. We manually tested the same set of applications (that, in the meantime, could have been updated, fixing this MITM vulnerability) against MITHYSApp, therefore evaluating the capability and the accuracy of detecting vulnerable applications. We show our results in Table 1. The results show that MITHYSApp is able to successfully detect vulnerable applications (according to Fahl et al.'s findings). MITHYSApp is also consistent with the results in [10] in detecting *Twitter* and *Voxie Walkie Talkie* as non vulnerable.

Application	Test result	Application	Test result
Amazon MP3	×	Google Play Store	×
Chrome	×	Google+	×
Dolphin Browser HD	×	Hotmail	×
Dropbox	×	Instagram	×
Ebay	×	OfficeSuite Pro 6	×
Expedia Bookings	×	PayPal	×
Facebook Messenger	×	Twitter	✓
Facebook	×	Voxie Walkie Talkie	✓
Foursquare	×	Yahoo! Messenger	×
GMail	×	Yahoo! Mail	×

Table 1. MITHYSApp results in detecting apps safe from SSL-based MITM attacks. (✓) indicates that the app is safe; (×) means that the app is vulnerable.

6.2 Experimental Setting

We have tested MITHYSApp with three of the most popular Android applications. These application belong to different categories of Google's Play Store, and represent three different important aspects that a typical mobile user is interested to: social networking, finance checking, cloud storage access. In particular, the applications we considered are: Facebook[6] (social networking service), PayPal[7] (global e-commerce business allowing online payments and money transfers), and Dropbox[8] (web-based file hosting service).

In our tests we considered two operations common to all the applications listed above: *login* and *logout*. These operations are very network-intensive, hence representing a perfect test scenario for MITHYSApp. As main tool for testing, we used `monkeyrunner` [13]. This tool allows interacting (e.g., pressing buttons, typing text) with an Android device by writing a simple Python script and running it via Android Debug Bridge (`adb`[9]). We wrote three scripts, one for each considered application. Each script basically performs these operations:

[6] `https://play.google.com/store/apps/details?id=com.facebook.katana`

[7] `https://play.google.com/store/apps/details?id=com.paypal.android.p2pmobile`

[8] `https://play.google.com/store/apps/details?id=com.dropbox.android`

[9] `http://developer.android.com/tools/help/adb.html`

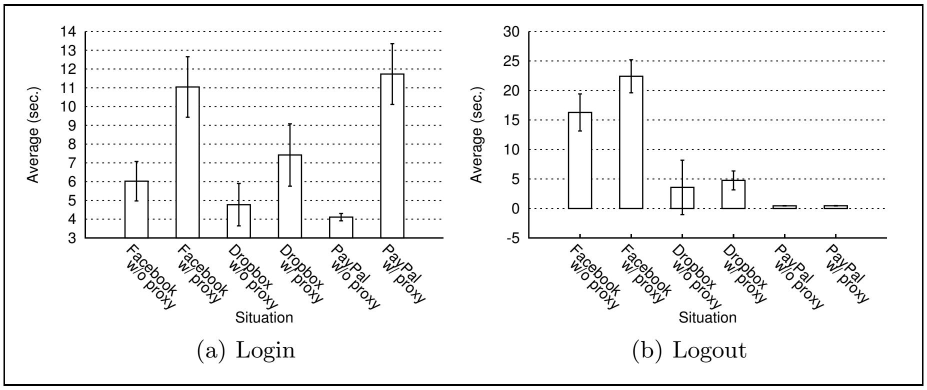
1. Connects to the Android device;
2. Opens the Android logcat in a subprocess (more on this later);
3. Starts the application's login activity;
4. Enters the credentials for a valid account;
5. Presses the login button and saves the current time on a variable called `LoginStartTime`;
6. Monitors the logcat in order to see when the main activity of the application is displayed - as soon as this happens, it saves the current time on the `LoginEndTime` variable;
7. Calculates the login time as (`LoginEndTime` - `LoginStartTime`);
8. Executes a number of actions in order to start the logout procedure; as soon as the logout button is pressed, it saves the current time on `LogoutStartTime`;
9. Monitors the logcat in order to see when the login activity of the application is displayed - as soon as this happens, it saves the current time on `LogoutEndTime`;
10. Calculates the logout time as (`LogoutEndTime` - `LogoutStartTime`);
11. Prints the two results.

We want to focus for a moment on the use of the `logcat` [12]. This tool allows the developer to collect and view the log messages, both coming from the Android OS and from the installed applications. We used specific `logcat` messages to determine the end of each operation (login and logout). Every time that the system displays a particular activity of the application (i.e., the main activity after the login, the login activity after the logout), we are sure that the considered operation has ended. This approach leads to reliable and repeatable tests, whereas it does not pollute the tests results at all.

6.3 Network Overhead

The results of our experiments are reported in Figure 4. In particular, Figure 4(a) and Figure 4(b) represent the overhead for the login and logout operation, respectively. We can observe that the average delay added by using MITHYSApp is approximately five seconds. Since this value is almost constant for each of the considered situations, the delay is more likely to be noticed by the user for shorter operations. The two figures show a higher delay in using MITHYSApp for both the login and the logout operations. This overhead is not suprising though, becuase MITHYSApp needs to issue additional network requests in order to protect mobile applications from MITM attacks. If we consider Facebook, the introduced delay for the login operation is about 55%, whereas for the logout operation it is about 33%.

There is an important point here we want to stress. While the current version of MITHYSApp is a fully-working implementation, we need to consider that it has not yet been optimised, both in terms of certificate caching and in terms of network performances. As a consequence, the values that emerged from the tests can be considered as an upper bound for the additional delay, which in some situations may be indeed noticeable by the user. We believe that, by properly



(a) Login

(b) Logout

Fig. 4. MITHYS: time overhead for representative applications.

optimising our implementation, we can reduce the five seconds average delay to a value of three or even two seconds. Another aspect that we have to take into account is that MITHYSApp is able to prevent MITM attacks that usually are performed nearby free Internet access points. Therefore, the user should take advantage of it while she is connected to a wireless access point, whereas it could be deactivated in other less attack-prone circumstances.

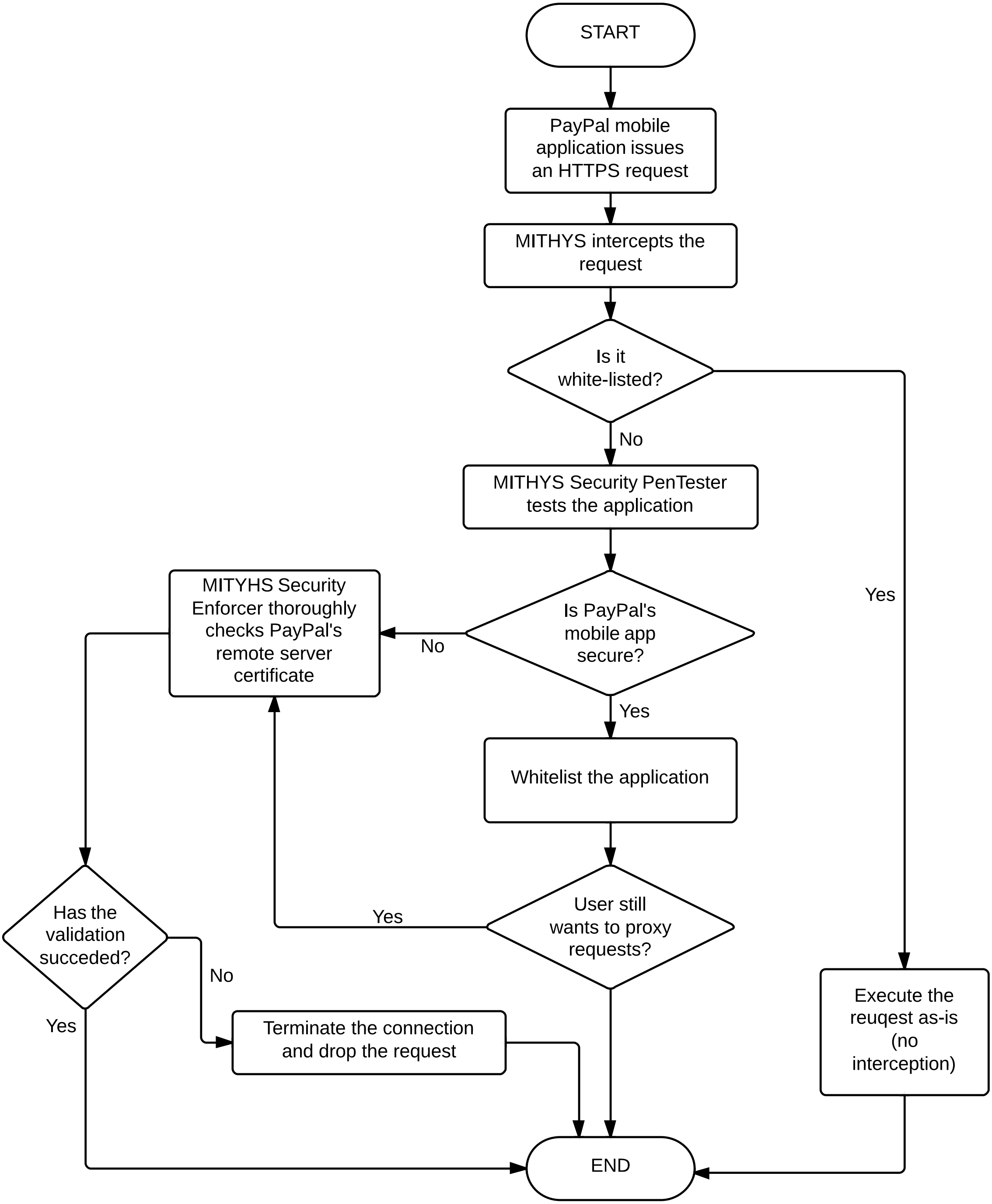
7 Conclusion

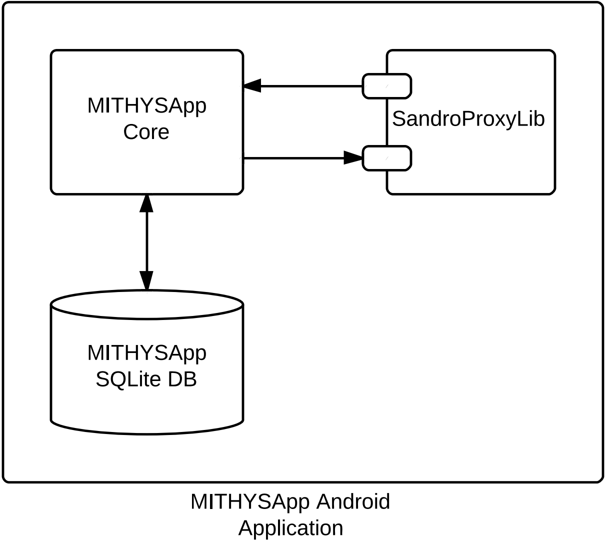
In this paper we have addressed a SSL vulnerability that has been recently shown affecting a base of many millions of users of mobile devices. To solve this problem, we have proposed MITHYS, a system for mobile devices which is able to protect mobile applications from SSL vulnerabilities. The architecture of MITHYS is light and feasible for several mobile platforms. To support this claim, we implemented MITHYSApp, i.e., MITHYS for Android. In particular, we implemented MITHYSApp at the application level, thus facilitating the spread of our solution and its installation on Android-powered mobile devices. We decided to focus on the Android platform mostly due to its popularity and flexibility. However, we have reasons to believe that mobile applications for Apple devices (e.g., iPhone, iPad) are just as vulnerable as the ones available for Android. For example, Thampi [19] was able to perform an SSL-based MITM attack to analyse the Path iOS application, discovering an illegitimate upload of the user's contacts to Path's servers. As a consequence, Path released a security update to its application, acknowledging the problem [14].

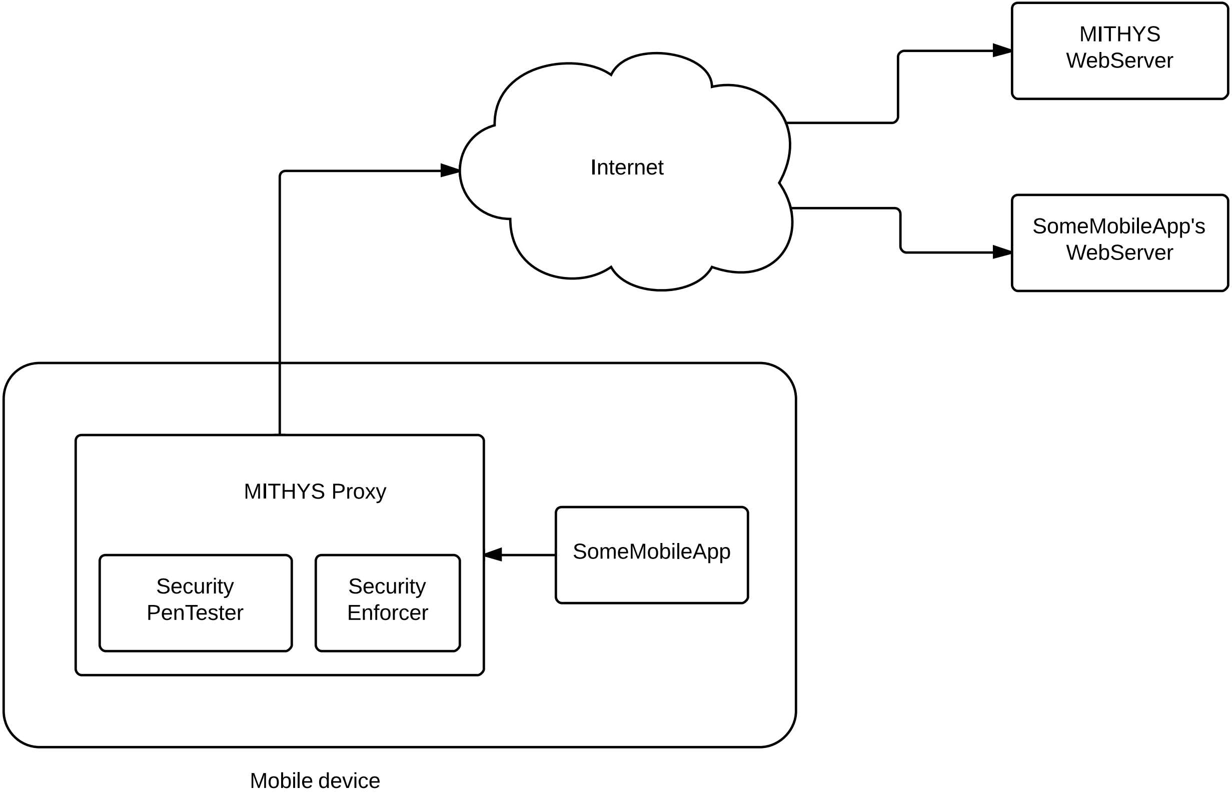
The results of our experiments showed that MITHYSApp has a limited overhead that even if noticeable, we believe being accepted by users when effectively protecting them from man-in-the-middle attacks aiming at stealing personal and sensible information. MITHYSApp represents a first (though fully working) implementation of the MITHYS system. Therefore, its performances can be vastly improved by adding advanced caching mechanisms. While the delay introduced by using MITHYSApp is still acceptable, we estimate that it can be further reduced by at least two seconds.

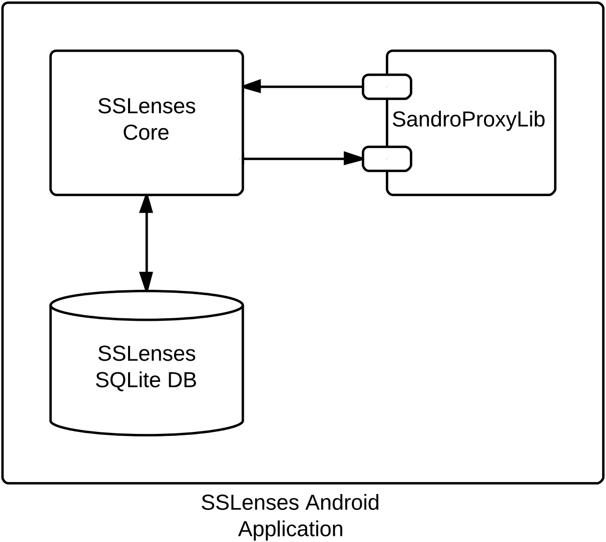
References

1. P. K. A. Freier, P. Karlton. The Secure Sockets Layer (SSL) Protocol Version 3.0. http://tools.ietf.org/html/rfc6101, 2001.
2. Amazon.com, Inc. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.
3. M. Becher, F. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 96–111, 2011.
4. K. Benton, J. Jo, and Y. Kim. Signaturecheck: a protocol to detect man-in-the-middle attack in ssl. In *Proceedings of CSIIRW'11*. ACM.
5. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.
6. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *Proceedings of NDSS'12*.
7. A. Charland and B. Leroux. Mobile application development: web vs. native. *Commun. ACM*, 54(5):49–53, May 2011.
8. M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In *Information Security*, pages 331–345. Springer, 2011.
9. L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Information Security*, pages 346–360. Springer, 2011.
10. S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: an analysis of android ssl (in)security. In *Proceedings of CCS'12*, pages 50–61, New York, NY, USA, 2012. ACM.
11. M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of CCS'12*, pages 38–49, New York, NY, USA. ACM.
12. Google Inc. logcat. http://developer.android.com/tools/help/logcat.html.
13. Google Inc. monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.
14. Path Inc. Path - We are sorry. http://blog.path.com/post/17274932484/we-are-sorry.
15. G. Russello, M. Conti, B. Crispo, and E. Fernandes. Moses: supporting operation modes on smartphones. In *Proceedings of SACMAT '12*, pages 3–12. ACM.
16. A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *Security Privacy, IEEE*, 8(2):35–44, 2010.
17. S. Shetty, M. Song, and L. Ma. Rogue access point detection by analyzing network traffic characteristics. In *MILCOM 2007. IEEE*, pages 1–7, 2007.
18. C. A. T. Dierks. The TLS Protocol Version 1.0. http://www.ietf.org/rfc/rfc2246.txt, 1999.
19. A. Thampi. Path uploads your entire iPhone address book to its servers. http://mclov.in/2012/02/08/path-uploads-your-entire-address-book-to-their-servers.html.


START
PayPal mobile application issues an HTTPS request
MITHYS intercepts the request
Is it white-listed?
No
Yes
MITHYS Security PenTester tests the application
Is PayPal's mobile app secure?
No
MITYHS Security Enforcer thoroughly checks PayPal's remote server certificate
Yes
Whitelist the application
User still wants to proxy requests?
Yes
Has the validation succeded?
No
Yes
Terminate the connection and drop the request
Execute the reuqest as-is (no interception)
END


MITHYSApp
Core
SandroProxyLib
MITHYSApp
SQLite DB
MITHYSApp Android
Application


MITHYS WebServer
Internet
SomeMobileApp's WebServer
MITHYS Proxy
SomeMobileApp
Security PenTester
Security Enforcer
Mobile device


SSLenses
Core
SandroProxyLib
SSLenses
SQLite DB
SSLenses Android
Application