Putting the "S" in HTTPS: Automatically Fixing Insecure HTTP and Flawed HTTPS Connections in Android

Vesko Stefanov

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2018

Abstract

The information that travels on the Internet is vulnerable to being read and modified. To solve this, cryptographic protocols, such as HTTPS, have been developed that achieve confidentiality, integrity, and authentication of such information. However, these protocols are not always implemented, and even when they are – they sometimes contain flaws that defy their purpose.

Of the available Android applications, 73.6% do not enforce that HTTPS, while 8% contain a flawed implementation of HTTPS.

I describe how I designed an Android application that would enforce the use of HTTPS and also fix its flawed implementations. I then start creating the application but its design requires more effort than anticipated. As a result, I decide to evaluate whether the application would be helpful before finishing it.

I conclude that the application would enforce HTTPS for 48% of the web services that did not use it, and fix flawed HTTPS for 86% of the web services that used it. Since I found that the application would be helpful, I decide to continue building it next year.

Acknowledgements

I am thankful to my supervisor, Kami Vaniea, for placing trust in me, nurturing my creativity, and mentoring me with undivided attention. She is an inspiring teacher.

I am also grateful to my family for their love and support, which have enabled me to work towards my degree; to Joe, my friends, and my classmates for their encouragement; to Wei Chen and Joseph Hallett for passing down their knowledge; and to Alexia Karageorgis for proofreading this work.

Table of Contents

Chapter 1

Introduction

Information that travels on the Internet is vulnerable to Man-in-the-Middle attacks that allow third-parties to read and modify it. This has been exploited by governments and internet access providers, who have been using it to their benefit without the users' consent [31, 24].

Cryptographic protocols have been created to protect information while it travels on the Internet. However, they are sometimes only theoretically sound. In practice, the way that the protocols are used is sometimes flawed. For example, the HTTPS protocol, which is used by web applications to transfer information, is used wrongly in 8% of Android and 9.7% of iOS applications [13, 16, 14]. The problem stems from developers who rely on online resources, such as Stack Overflow, that have been proved to cause the erroneous implementations [1]. In these cases, the travelling information is not protected by the protocols [13]. The aim of this project is to identify ways to help developers who do not correctly use cryptographic protocols.

1.1 Report Structure

The report is structured as follows.

Chapter 2 presents the work relevant to mine. That is, how in-transit information is protected with cryptographic protocols, the barriers to adopting these protocols, and the existing solutions to these barriers.

Chapter 3 describes the security properties that cryptographic protocols have to achieve and how they achieve them. It continues to describe TLS and HTTP(S).

Chapter 4 presents application that I developed in order to understand the barriers to implementing HTTPS, and the semi-structured interviews I conducted while gathering requirements. It also specifies the final solution – an Android application, and how

it would overcome the HTTPS adoption barriers. This solution is different from the tutorial which I was originally according to the project description.

Chapter 5 describes the detailed design of the Android application. Then it describes only a part of its implementation, and explains why I evaluated the application before finishing its implementation

Chapter 6 critically evaluates whether the application would be useful and whether I should continue building it. It establishes that the application would be able to achieve its goals.

Chapter 7 concludes my work and presents my further plans to continue building the application in the second year of my MInf project.

Chapter 2

Related Work

2.1 Threats to secure communication

On the Internet information travels between communicating devices, usually from a sender to a receiver. However, information does not travel directly between the sender and receiver, because there is normally no single cable that connects them. Instead, it hops through multiple intermediate devices and cables in order to reach the receiver, relying on each device to forward it further along its path. The path of such hops is non-deterministic and depends on which devices are available. This hopping matters because intermediate devices can read and modify the information in transit, that is, perform a Man-in-the-Middle attack.

The Chinese government provides for an example of a Man-in-the-Middle attack. It has been launching a massive-scale attack, targeting websites that are forbidden in China but are trying to circumvent this restriction. For example, the website GreatFire.org was targeted for being a "foreign anti-Chinese organization" as it offers software for circumventing the Chinese Internet censorship [31].

This attack is caused by software called the Great Cannon of China – an intermediate device located at the Chinese border. It manipulates the information that travels from the web services of the Chinese-based corporation Baidu to its users who are outside of China. The Great Cannon of China watches for in-transit information from Baidu and inserts JavaScript requests into it. This forces multiple users' devices to connect to to the targeted websites. The websites cannot handle the massive amount of connections, and are made extremely slow or inaccessible.

Another type of Man-in-the-Middle attacks affects users' privacy and are performed by Internet access services, such as the WiFi in Starbucks or one's home Internet provider. This is possible because Internet access providers are an intermediate device, which allows them to modify in-transit webpages that they do not own. In practice, Comcast uses JavaScript and its 3.5 million publicly accessible Wi-Fi hotspots across

the US to insert their advertisements into any webpage [28]; Verizon inserts permanent cookies that can be used by third parties to track the websites that a particular user is visiting [24]. These attacks are opposed by a movement for net neutrality, which says that Internet traffic should be treated equally regardless of its sender, recipient, or content. It also offers privacy protection such as limiting the ability of Internet access services to read in-transit information [8]. However, net neutrality is opposed by the Internet access services. In particular, last year, pressure from such services led to the United States repealing net neutrality a week before it came into action [32].

2.2 Cryptographic Protocols for the Internet

Cryptographic protocols for the Internet are mechanisms used to protect in-transit information. To do so, they rely on three fundamental properties: confidentiality, integrity, and authentication.

Confidentiality requires that in-transit information cannot be read or inferred. It is provided by symmetric encryption – a cryptographic protocol that changes the representation of the information before it leaves the sender and restores it when it reaches the receiver. Integrity requires that the in-transit information has not been modified by other devices. It is provided by Message Authentication Codes (MACs) – a tag of data sent with the information and used to confirm if the information has been modified. Authentication requires that in-transit information is coming from the sender and no one else. It is provided by requiring the sender to provide a certificate of identity that is consecutively verified by a third-party certificate authority. The workings of cryptographic protocols for the Internet are further detailed in Chapter 3.

2.3 Cryptographic Protocols for HTTP

In 1990 Tim Berners-Lee invented the first web browser, called WorldWideWeb. It was the first way to see the World Wide Web — a collection of webpages and other web resources [3]. The Hypertext Transfer Protocol (HTTP) — the technology that enabled Berners-Lee's browser to send and receive information — is still a pillar for today's web browsers and web applications. Information transferred with HTTP is not protected by cryptographic protocols and is vulnerable to Man-in-the-Middle attacks. A secure version of HTTP, called HTTP Secure (HTTPS), was later specified in 2000 [41]. HTTPS uses a cryptographic protocol, called Transport Layer Security (TLS), to protect sensitive online information.

2.4 Adoption of HTTPS

In the wake of the Snowden's revelations about government surveillance, Internet users became increasingly concerned about their online privacy. It became vital for compa-

nies that use the Internet to protect their users' information due to demand and political regulations. Since HTTP usage allows government agencies to perform Man-in-the-Middle attacks, HTTPS has been identified as a viable solution to surveillance. Thus, a decade after the definition of HTTPS, its widespread adoption began. Large companies like Twitter and Google started enforcing that information on their services travels through HTTPS [25, 43].

Web services can be classified according to their adoption of HTTPS: they do not support HTTPS, they support HTTPS but do not enforce its usage, and they support HTTPS and enforce its usage, the latter meaning that the web service only communicates with HTTPS and refuses HTTP communication. Nowadays, between 64% and 83% of the connections that desktop and mobile browsers make are HTTPS [17, 12]. Among the most used websites, 25% are not supporting HTTPS, while 30% are not enforcing HTTPS communication. Among Android applications 73.6% of these are not enforcing HTTPS communication [13].

Movements to support the adoption of HTTPS were born. The Federal HTTPS-Only Standard was introduced in 2015 — all publicly accessible Federal web services in the US should only use HTTPS [18]. Moreover, browsers have been reinventing the way users are informed about HTTP(S) websites. The Google Chrome web browser has been increasing the visibility of secure connections. It has a green padlock in the address bar indicating that a website is using HTTPS. After they realised some people think the padlock is a purse since it mainly appears in shopping websites, they explicitly wrote 'Secure' next to the padlock [19]. However, for websites that use HTTP there is no padlock nor an indication that the website is insecure — users are not informed about sending unprotected information. Fortunately, Chrome have announced plans to start adding a grey padlock and writing 'Insecure' for HTTP websites with password or credit card form fields from January 2017 [4].

Let's Encrypt is yet another important movement that was launched in 2016 by the Internet Security Research Group. It uses automation to address barriers to HTTPS adoption such as paying for certificates, determining ownership, and certificate renewal. Thus, developers can obtain a certificate faster and for free. They also do not need to manually create, configure, and renew it. [12].

2.5 Vulnerabilities of HTTPS

Despite the availability of an automatic method for certificate generation, it is still possible that developers hinder the security of the HTTPS protocol. For example, browser HTTPS connections result in an unnecessary warning 1.54% of the time due to the use of self-signed certificates or of certificates that do not match the server's hostname [2]. Such flawed connections trigger a high volume of false warnings, training users to ignore them but also ignore true ones. As a result, during an attack 68% of the users ignore the warnings.

Additionally, 8% of Android and 9.7% of iOS applications contain an HTTPS implementation that is vulnerable to a Man-in-the-Middle attack [13, 14, 16]. Fahl et al. [13] conducted a study and found that the most common reasons for a vulnerability are

- accepting any certificate even if it cannot be verified by a certificate authority whose certificate is installed on the application or on the device,
- accepting a certificate for any hostname as long as it can be verified by a certificate authority whose certificate is installed on the application or on the device, and
- not requiring the use of an HTTPS connection and allowing HTTP only connections.

These vulnerabilities were found to be introduced when developers

- forgot to add an '*s*' to the connection in order to use HTTPS; for example they used `http://developer.android.com` instead of `https://developer.android.com`.
- disabled certificate checks for debugging but forgot to restore them before releasing the application, and
- used their own class for checking certificates and hostnames because of the use of self-signed or pinned certificates.

To my surprise, of the applications that implement their class for verifying the server's hostname in Android, 98.9% do not actually verify the hostname [1], allowing for any certificate from the same certification authority to be accepted for that hostname.

Relying on the resources that are already available, developers report that 88.4% of them use Stack Overflow to learn how to code [21]. This is understandable since Stack Overflow provides answers to specific coding questions. Unfortunately, implementations based on Stack Overflow can be grossly insecure. Developers were asked to update an Android app to use HTTPS instead of HTTP connections [1]. One group was allowed to only use Stack Overflow; a second group — any available resource; and a third group — the official Android documentation. In each group approximately 17%, 50%, and 100% of the participants built a secure, functional system, respectively. The participants that used any resource accessed both Stack Overflow and the documentation in all but one case. However, the documentation is not a perfect resource either. More of the developers who used Stack Overflow produced functional code and found the resource more helpful than those who used the documentation. Additionally, a behaviour pattern emerged — developers would read the documentation until they feel pressed for time, then they copy code from Stack Overflow.

One approach to protect against such flaws is to increase developers' knowledge of cyber security. Amateurs, who rely on Stack Overflow, are in need of resources that educate them of the best security practices. Moreover, even the most security-conscious developers allow vulnerabilities and should be educated on the same topic. For example, such developers who are employed by banks whose regulators require that they handle sensitive information with great care, produce Android applications for general use that contain flawed HTTPS implementations [6]. These applications do not check

the hostname of the receiver that they communicate with and are vulnerable to Man-in-the-Middle attacks by intermediate devices that have a certificate from the same certificate authority [46].

2.6 Existing Solutions

Some approaches protect against HTTP vulnerabilities by taking the responsibility away from developers. These solutions include app store approvals, virtual private networks, onion routing, and browser plugins that enforce HTTPS.

App stores are public, digital collections of applications and are integrated into select operating systems. For example, the two biggest stores are Android's Google Play Store and Apple's App Store. Developers publish their apps on the app store if they want to make them available to the general public. However, each company also maintains their store and blocks applications that are deemed insecure. The common opinion is that Apple has stricter publication guidelines than Google, which is reinforced by the companies' approval process taking days and hours, respectively. During the approval process, app stores scan and run publications to determine if they contain insecure or malicious code. Unfortunately, code analysis has time and resource limitations – it cannot detect apps that run their insecure code only after some time or contain the insecure code in an encrypted format [35]. I have given examples in Section 2.5 that insecure applications still persist despite stores' efforts and regulation.

Virtual private networks (VPNs) are used by security-conscious users. If a client device is using a VPN and wants to send information to a web service over the Internet, the client sends the information to the VPN instead. The VPN then forwards the information to the web service. If the web service wants to respond with information, it sends it to the VPN service, which then forwards it to the client. Thus, the use of a VPN prevents the web service from knowing the identity of the client that it is communicating with. However, the travelling data is not encrypted and is vulnerable to Man-in-the-Middle attacks. The connection between the client and VPN is commonly encrypted, but does not have to be. Then the information that the client and the web service communicate travels encrypted between the client and the VPN and unencrypted between the VPN and the web service. Therefore, Man-in-the-Middle attacks are not possible between the client and the VPN but are possible between the VPN and the web service. Moreover, encrypted VPNs make it hard or theoretically impossible to establish the identity of the client that communicates with a web service when the unecnrypted communication between the VPN and the server is read via a Man-in-the-Middle attack. Therefore, when HTTP or flawed HTTPS is used in an application, an encrypted VPN cannot provide a cryptographic protocol between the application and the web service that protects against a Man-in-the-Middle attack.

The Tor open network[1] is another tool for users concerned about their security. It is a framework for anonymous communication developed by the Tor Project. If a client wants to send information to a web service, the client chooses a sequence of designated intermediate devices called onion routers. Then the client encrypts the information multiple times, once for each chosen onion router, and obtains a single encrypted representation. The encrypted representation is analogous to an onion because it has multiple layers of encryption. The client then sends the representation to the first onion router from the sequence, which, in turn, decrypts one layer of encryption and forwards the decrypted representation to the next router in the sequence. The next router decrypts another layer and sends the representation to the next router in the sequence; this step is repeated until all layers of encryption are decrypted. Then the last onion router decrypts the representation, obtains the original information, and sends the original information to the web service. When the web service communicates with the client, the reverse process is executed — the last onion router from the sequence receives information from the service, encrypts it with one layer of encryption and forwards the representation to the second but last onion router, and so forth. In the end, the client receives the service's response in a single representation that has multiple layers of encryption. Onion routing has the same vulnerabilities as VPNs. Therefore, when HTTP or flawed HTTPS is used in an application, neither onions networking nor an encrypted VPN can provide a cryptographic protocol between the application and the web service that protects against a Man-in-the-Middle attack.

HTTPS-Everywhere is a web browser plugin, developed by the Electronic Frontier Foundation (EFF)[2], is yet another tool for the security-aware users. It is a browser plugin that auto-upgrades browser connections from HTTP to HTTPS based on crowd-sourced ruleset, an Atlas[3]. Before an HTTP connection is going to be made to `http://example.com`, the ruleset is checked to see if the service supports HTTPS. If it does, then the plugin rewrites the request to `https://example.com` and thus requests HTTPS communication with the service. If the service responds with HTTP, the plugin assumes that this is a Man-in-the-Middle attack and blocks the communication to the browser.

A known limitation of HTTPS-Everywhere is that it can only be used in cases where the website has been added to the ruleset. Another limitation is that the plugin only enforces HTTPS connections in a browser; the rest of the applications' installed on a device are unaffected. While the earlier solutions we discussed are available on any operating system, the HTTPS-Everywhere plugin is only available for desktop web browsers and the Firefox for Android web browser. However, users on mobile devices spend 85.7% of their time in applications rather than in the browser [11]. Assuming that usage time is proportional to the number of online connections that are made, most insecure connections that are made on Android cannot be affected by HTTPS-Everywhere.

[1] `https://www.torproject.org/`

[2] `https://www.eff.org/https-everywhere`

[3] `https://www.eff.org/https-everywhere/atlas/`

2.7 Summary

In-transit information is vulnerable to Man-in-the-Middle attacks that allow third-parties to read and modify it. HTTP is the protocol used by today's web applications to transfer information. It has a secure version – HTTPS – which protects against such attacks. Unfortunately, the protocol is not being used by all applications yet and some of the applications that use it, do not use it correctly. As a result, the protocol does not protect users from Man-in-the-Middle attacks. Corporations and users have been using alternative methods to protect themselves against these attacks, however, neither of them are complete solutions.

Chapter 3

Cryptographic Protocols for the Internet

3.1 Cryptography

Cryptography is the construction of communication protocols that prevent third parties from violating particular security properties [45]. The most important security properties are confidentiality, integrity, and authentication. I go on to present them and the theoretical cryptographic protocols that satisfy them. Then I explain how they come together to form TLS – the practical cryptographic protocol that is in the foundation of HTTPS.

3.1.1 Confidentiality

Confidentiality is the property that data in transit cannot be read or inferred. It is provided by encryption.

3.1.1.1 Encryption

Encryption is a cryptographic protocol that changes the representation of the original information before it leaves the sender and restores it after it reaches the receiver. Thus, only the receiver (and maybe the sender) but none of the intermediate devices are able to read or infer the original information.

Plaintext is the name for the original information, and ciphertext is the information's encrypted representation. Changing the representation of the plaintext is called encryption and restoring it is called decryption. The pair of corresponding encryption and decryption algorithms is called a cipher. The ciphers that are used in public software are usually known and standardised, such as the Advanced Encryption Standard (AES) and Rivest–Shamir–Adleman (RSA). Each algorithm takes a parameter called a

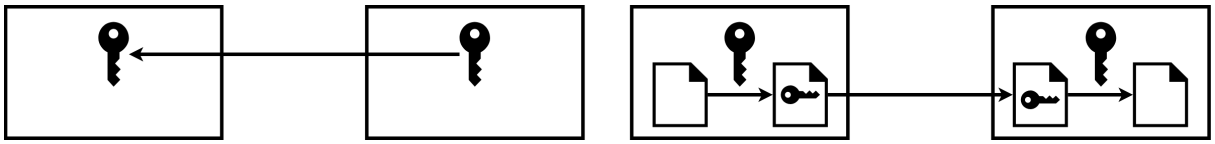

Figure 3.1: Symmetric encryption requires that the receiver sends its key to the sender. Then the sender can encrypt information with the key and send it to the receiver, who decrypts the ciphertext with the same key.

key – a randomly chosen piece of data, that is, a number or an array of bytes. The key used for encryption and decryption can be the same or be different.

3.1.1.2 Symmetric Encryption

Symmetric encryption is performed when the sender and receiver use the same key (Figure 3.1) [45]. Since everyone that has the key can decrypt the ciphertext, it is essential that the key is known only by the two communicating devices. Otherwise, any intermediate device that knows the key can decrypt the ciphertext too, defeating confidentiality. At the same time, it is required that both devices already know the key in order to communicate.

3.1.1.3 Limitations of Symmetric Encryption

It is infeasible in Internet communication to assume that both devices will already know a key they can use. On one hand, a developer cannot usually anticipate all web services that their software will communicate with and pre-equip both with a key. For example, a browser cannot practically be pre-equipped with the key of all encrypted pages it will load. On the other hand, the same key cannot be used by multiple devices because then all devices that have it can decrypt the information and some of them might not be trusted. Therefore, one of the devices needs to generate the key and send it to the other device in order to communicate with it using encryption. However, if we assume that the devices do not have another way to perform encrypted communication, the keys cannot be sent over the Internet without giving intermediate devices the ability to read them and then use them to decrypt the subsequent encrypted information in transit (Figure 3.2). Moreover, a deterministic algorithm that generates the same key on two different devices cannot simply be used – such an algorithm can be used by an intermediate device to also produce the same key.

3.1.1.4 Asymmetric Encryption

In contrast, asymmetric encryption, which is performed when the sender and receiver use a different key, allows two devices to create or communicate keys without giving the intermediate devices the ability to decrypt the information in transit (Figure 3.3) [45]. Each of the communicating devices generates a public and a private key, and

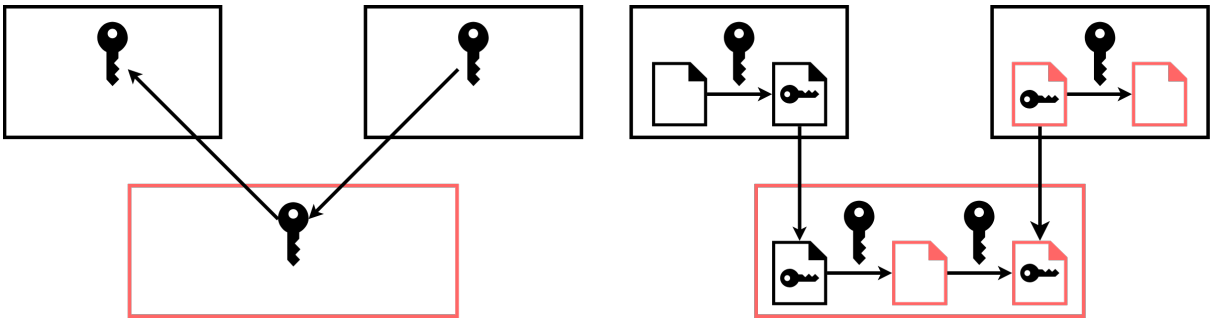

Figure 3.2: The key used for symmetric encryption is sent in plaintext. Hence, an intermediate device can read the key. The device can then decrypt the information from the sender, read and modifiy it, reecrypt it, and send it to the receiver.

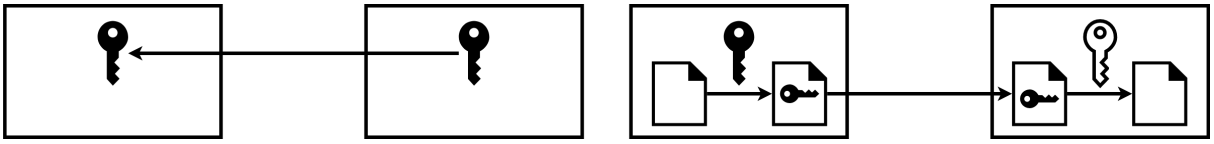

Figure 3.3: In asymmetric encryption the receiver sends its public key to the sender. Then the sender can encrypt information with the public key and send it to the receiver, who decrypts the ciphertext with another key – a private one.

sends the public key to the other device. The sender uses the receiver's public key to encrypt plaintext and then sends it. The receiver uses its private key to decrypt the arriving ciphertext. Intermediate devices could have read the public key while in transit; they can also send information to the receiver that is encrypted with it. The catch is that ciphertext can only be decrypted with the receiver's private key, which is known only by the receiver and has not been communicated. Therefore, intermediate devices cannot decrypt ciphertext in transit in order to read the original information.

3.1.1.5 Limitations of Symmetric Encryption

As shown in Figure 3.4, a Man-in-the-Middle attack is still possible. To set up asymmetric encryption, the receiver needs to tell the sender its public key. However, this key travels in plaintext. An intermediate device can read this key, and then forward its own public key to the sender. Thus, the information coming from the sender can be decrypted by the device, who can now read and modify it before it forwards it to the receiver.

Both symmetric and asymmetric encryption achieve confidentiality because data in transit cannot be read by intermediate devices since they do not have the (private) key. Encryption cannot be used between two devices that do not have the keys hard-coded. Furthermore, every ciphertext has a corresponding plaintext. The receiver will not know if an intermediate device has changed the ciphertext and, therefore, the plaintext.

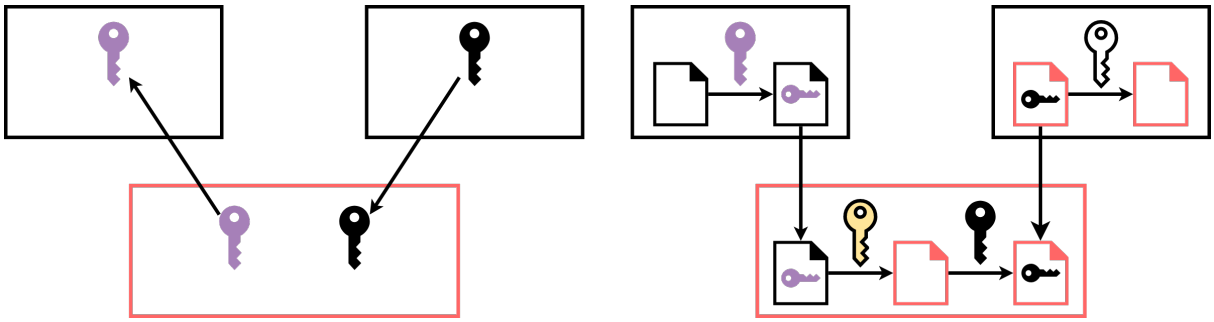

Figure 3.4: In asymmetric encryption the receiver sends its public key to the sender. However, the sender has no way to confirm that the received public key belongs to the receiver or to an intermediate device that is a Man-in-the-Middle. Such a device can negotiate separate keys with the sender and the receiver, and use the keys to read and modify the in-transit information.

3.1.2 Integrity

Integrity is the property that data in transit has not been modified by devices other than the sender.

3.1.2.1 Message Authentication Codes

Message Authentication Codes (MACs) are a cryptographic protocol used to provide integrity [45]. Before it sends ciphertext, the sender generates a small piece of data (a tag) from it using a signing algorithm and a symmetric key. Then both the ciphertext and the tag are sent to the receiver.

After the information arrives, the receiver splits it into the tag and the ciphertext, and confirms that the former corresponds to the latter using a verification algorithm and the symmetric key. If the data was modified by an intermediate device, then the verification check fails. This is reliable because the key is only known by the two communicating devices; intermediate devices cannot modify the in-transit information and generate a tag that verifies successfully. It is computationally infeasible for a tag to be computed without knowing the symmetric key. A limitation of MACs is that modified information cannot be used to obtain the original information; it can be obtained only from unmodified ciphertext.

MACs allow the receiver to confirm that the information that arrives has not been modified by anyone but the sender. However, they rely on encryption and suffer from its limitations. For MACs to work both communicating devices need to know the symmetric key. Asymmetric encryption can be used to confidentially exchange the symmetric key, however, such encryption does not ensure the integrity of the key. The ciphertext, and therefore the key, could still have been modified in transit. If the key that arrives at the receiver has been changed, none of the received information will pass the verification check.

3.1.3 Authentication

Authentication is the property that information is being received from the sender and not an intermediate device.

3.1.3.1 Digital Signatures

Digital signatures are a cryptographic protocol and the base for proving the authenticity of information [45]. They are similar to Message Authentication Codes as they say whether information in transit has been modified by anyone else but the sender. In contrast, digital signatures use an asymmetric key.

Before it sends ciphertext, the sender generates a small piece of data (a tag) from it using a signing algorithm and its private key. After the information arrives, the receiver splits it into the tag and the ciphertext, and confirms that the former corresponds to the latter using a verification algorithm and the sender's public key. In fact, everyone who knows the sender's public key can confirm that the information has not been modified by anyone else but the sender. Moreover, the sender is the only device that knows the private key and can create a corresponding tag for a piece of information. If a tag is generated with another private key, the verification algorithm (that uses the sender's private key) will fail.

3.1.3.2 Public-Key Certificates

A public-key certificate is a digital document that proves the identity of a web service according to an authority [45]. The certificate contains the server's identity (in the form of a hostname – a human-readable name such as `google.com`) and public key, an expiration date, and the authority's identity, among other fields. The certificate also contains a digital signature that can be used to prove that the fields have not been modified by anyone else but the authority.

3.1.3.3 Self-Signed Certificates

A self-signed certificate is one where the authority is also the server that certificate identifies [45]. If the sender provides such a certificate, it assumes that the receiver has its public key. The digital signature in the certificate has been signed with the private key of the authority (that is, the sender). It can only be used if the receiver already has the sender's public key, otherwise, as shown in Section 3.1.2.1, we do not have a secure way to exchange asymmetric keys yet.

3.1.3.4 Limitations of Self-Signed Certificates

Self-signed certificates do not provide authentication. The receiver needs the public key of the sender to be able to verify the digital signature that is part of the certificate.

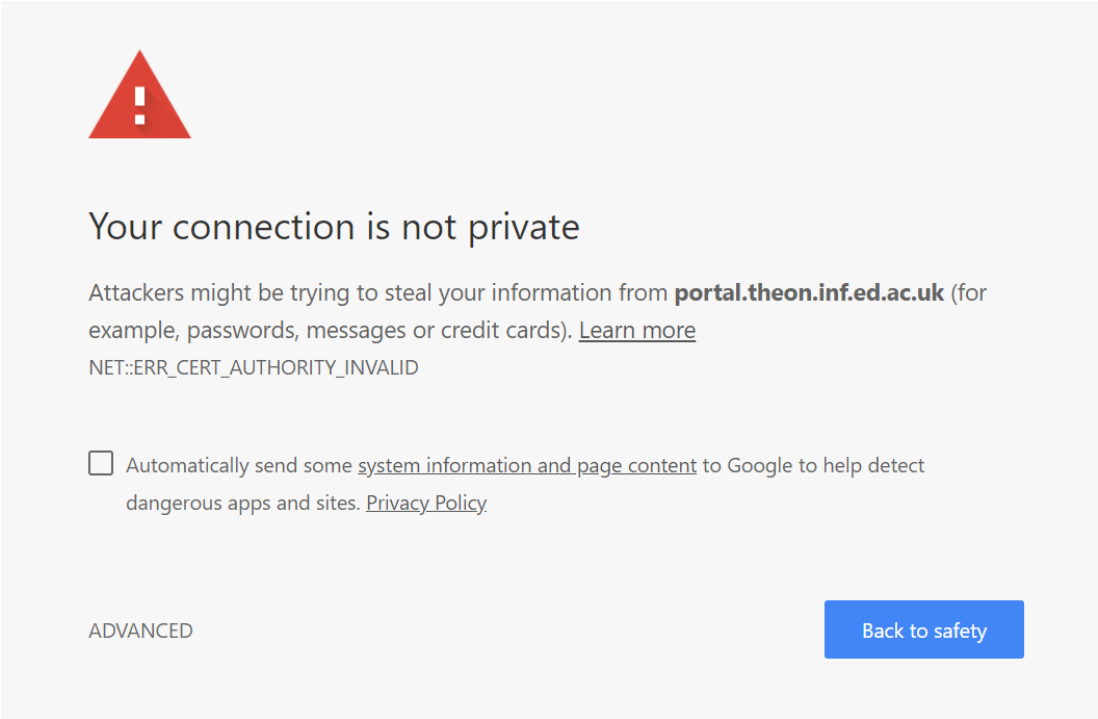

Figure 3.5: Self-signed certificate warning in the Chrome browser.

Similarly to symmetric encryption, it is infeasible in Internet communication to assume that the receiver will know that public key. A developer cannot usually anticipate all web services that their software will communicate with and pre-equip it with their public key. For example, a browser cannot practically be pre-equipped with the public key of all encrypted pages it will load.

Self-signed certificates work in some situations such as the internal webpages of the School of Informatics. The webpage `https://portal.theon.inf.ed.ac.uk/` uses a self-signed certificate that is installed on the computers of all staff members. Thus, the staff members' browser can automatically verify that it is talking to this webpage and not an intermediate device. However, students who have not explicitly installed this certificate on their computers and who access the page are not protected against a Man-in-the-Middle attack by an intermediate device (see Figure 3.5).

Self-signed certificates can achieve authentication by proving the identity of the sender. However, they do not work when the two devices do not have a hard-coded asymmetric key.

3.1.3.5 Certificate Authorities

Certificate authorities (CAs) are third-party authorities (CAs) who issue public-key certificates to web services [45]. Thus, there can be few trusted CAs and developers can pre-install their public keys into their software. Nowadays, a set of CAs is pre-installed into operating systems and web browsers

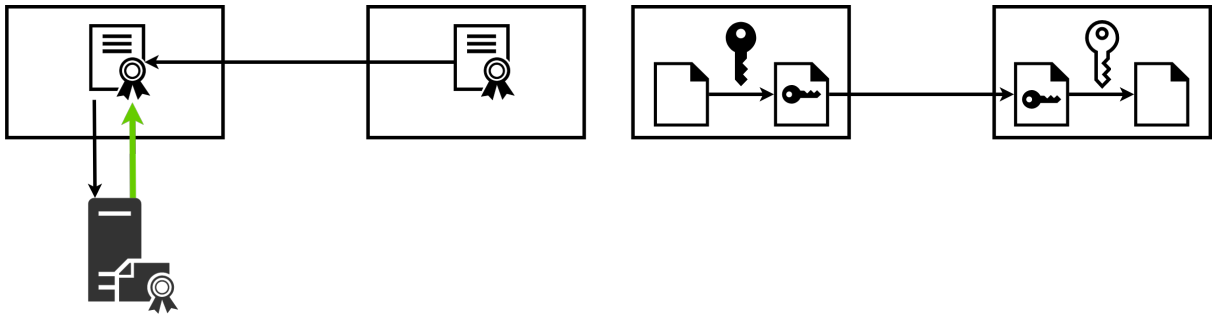

Figure 3.6: The receiver provides its certificate and public key to the sender. The sender checks the validity of the key by using the digital signature in the certificate. It also connects to the CA to check if the certificate has not been revoked. If the certificate is valid, the information is sent with asymmetric encryption.

When an application communicates with a web service and wants to authenticate the web service, it can request from the service to provide its certificate. While it is possible, it is not common that the web services request an application to also provide a certificate. Upon receiving the service's certificate, the application can use the pre-installed public key of the authority that created the certificate to verify the integrity of the certificate. If the verification has been successful, the application is certain that the public key in the certificate belongs to the web service (Figure 3.6). Thus, if the application encrypts in-transit information with that public key only the web server can decrypt it.

It is important to note that the contents of the certificate, even when verified, might not be valid. For example, the certificate is also checked to see if

- its hostname matches the hostname of the web service.
- it has expired, and
- it has been revoked.

Each of these checks is essential and a failure in any of them should result in the certificate not being valid (Figure 3.6).

Certification via certificate authorities achieves authentication even if the communicating devices do not have a hard coded asymmetric key.

3.1.3.6 Certificate and Public Key Pinning

Pinning the certificate and the public key of a web service means that the certificate, and the corresponding public key, of a web service have been hard-coded [27]. Therefore, when an application communicates with a web service and wants to authenticate the web service, it still requests that the service send its certificate to the application. However, the application does not use the private key of the CA to verify the received certificate, but instead compares the received certificate to the hard-coded one. Pinning

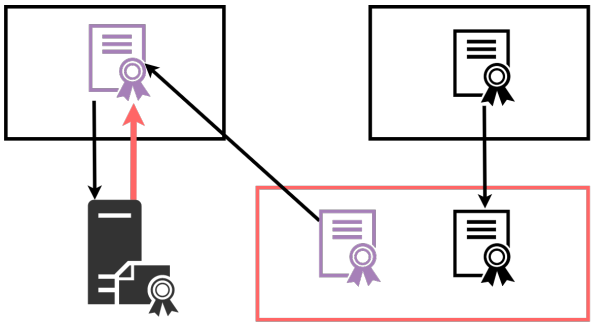

Figure 3.7: The receiver provides its certificate and public key to the sender. The sender checks the validity of the key by using the digital signature in the certificate. However, one of the checks has failed – the CA has revoked this certificate. The sender does not forward information to the receiver because it cannot confirm its identity.

removes the need for verification by CAs. This introduces an additional layer of security because CAs are not always trusted either – according to Google, the Symantec Corporation issued questionable website certificates [9].

3.2 Transport Layer Security (TLS)

Transport Layer Security (TLS) is a practical cryptographic protocol that provides the security properties from the previous section through their corresponding theoretical cryptographic protocols [41]. It is the successor of Secure Sockets Layer (SSL), a protocol prohibited by Internet Engineering Task Force [42].

TLS combines the cryptographic protocols described in the previous section to achieve all security properties – confidentiality, integrity, and authentication. At a high level, the protocol requires a handshake to create a channel for secure communication and then uses the channel to securely transfer information.

1. The application that requires communication with a web service sends a *Hello* message to it.

2. The web service responds with its certificate, which is checked for validity by the application, and specifies a preferred symmetric encryption algorithm, among other information.

3. The application then sends information, encrypted with the service's public key, that is used to create a symmetric key.

4. The encrypted communication between the application and the service starts, using the symmetric key that has been established and MACs. (Symmetric encryption is used, rather than asymmetric encryption, because it is less computationally demanding [10].)

3.3 Hypertext Transfer Protocol Secure (HTTPS)

The Hypertext Transfer Protocol (HTTP) is a protocol that allows applications, such as web browsers and Android applications, to send and receive information over the Internet [15]. An example of an HTTP request and the corresponding response are shown in Listing 3.1 and Listing 3.2. In the request, the first line specifies the type of the request, the location of the requested resource, and the version of the HTTP protocol; in the response, the first line specifies the version of the protocol, and the status of the response, denoted by a status code and a textual representation. The following lines contain header fields and values, an empty line, and an optional message body.

Listing 3.1: An HTTP request.

```
POST /goform/setDateTime HTTP/1.1
Content-Type: text/html
Content-Length: 24
Host: 10.168.168.1:8080
Connection: Keep-Alive
User-Agent: android-async-http/1.4.5
Accept-Encoding: gzip

[Full request URI: http://10.168.168.1:8080/goform/setDateTime]
[HTTP request 1/1]
[Response in frame: 606]
File Data: 24 bytes
```

Listing 3.2: An HTTP response.

```
HTTP/1.0 200 OK
Server: GoAhead-Webs/2.5.0
Pragma: no-cache
Cache-control: no-cache
Content-Type: text/html

[HTTP response 1/1]
[Time since request: 2.637808000 seconds]
[Request in frame: 572]
File Data: 13 bytes
```

The information transferred with HTTP is not protected by cryptographic protocols and is vulnerable to being read and modified by intermediate devices. For this purpose, a secure version of HTTP has been created. Called HTTP Secure (HTTPS), it uses the TLS cryptographic protocol to create a channel for secure communication and then uses the channel to securely transfer information. However, not all web services support HTTPS.

The HTTP and HTTPS protocols are interchangeable – they only differ in that the latter uses an encrypted channel while in transit. A special HTTP request can be used by the application or the web service to upgrade from HTTP to HTTPS. It hence follows

that HTTP connections can be upgraded to HTTPS when the web service supports it. However, it is not that simple – a request asking for an upgrade and a response saying a web service does not support the upgrade are both sent via an HTTP message, which can be modified by intermediate devices. Therefore, either of these messages cannot be reliably delivered to their destination.

3.4 Summary

I described the security properties of confidentiality, integrity, and authentication, and the cryptographical protocols that provide them – encryption, MACs, and public-key certificates, respectively. Together these protocols compose TLS. Lastly, I introduced HTTP, the most used protocol by applications that transfer data over the Internet. It is insecure unless paired with TLS. Unfortunately, HTTP cannot be used to reliably determine whether a web service supports TLS.

Chapter 4

Requirements Gathering

The original purpose of the project was to help developers correctly use TLS when developing mobile applications. As observed in Chapter 2, developers are currently making many errors from simply not including the '*s*' in HTTPS to not checking the validity of certificates correctly. To understand the problem better, I decided to start by creating a small sample Android application which correctly made HTTP(S) connections. Doing so highlighted several existing problems with the current support for developers at my level. I then conducted semi-structured interviews with several mobile developers where I asked them about their experiences and discussed several of my ideas. As an outcome, I realised that asking developers to do all the necessary steps to implement HTTPS (correctly) is unrealistic, and a better solution would be to take responsibility away from them through automation.

In this chapter I describe the various requirements gathering activities I conducted and what I learned from each one.

4.1 Implementing HTTP(S) Connections in Android

The application that I developed was based on the official Android Developers tutorial [29] and was created in Android Studio, the official integrated development environment (IDE) for Android applications. The application was simple and did not have an interface; instead it made a connection to an HTTP(S) service and printed the response in Android Studio's logging tool. Later I went back to my implementation and confirmed that it did not contain any of the previously mentioned flaws.

4.1.1 Limitations of the Android Developers tutorial

After implementing the application, I identified that

1. Despite it being enough to just add an '*s*' to the URL in order to make an HTTPS

Listing 4.1: A Java code snippet from Android Developers that aims to show how simple it is to implement an HTTPS connection.

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

connection, I was not sure if was enough and if the connections that I implemented were secure.

2. The code from the Android Developers tutorial contains a couple of semantic errors – one in an overridden generic method and another in a class definition that lacks a parameter. They did not allow me to compile the code after copying it into my application. I used a Google search and *(ironically)* Stack Overflow to fix them.
3. There is an alternative – an example application that works without modifications is provided in addition to the tutorial. It contains minor semantic differences from the tutorial that should not affect the security of the final implementation. Consistency is important in the case that someone decides to build their application based on the example one.
4. The tutorial gives cautionary warnings to avoid the misuse of HTTPS that we saw in Chapter 2.
5. I found that HTTPS configuration in Android can be achieved in two different coding languages: Java and XML.

The first three observations suggest that creating coding tutorials is hard even for corporations and that there is significant complexity when maintaining tutorials in several media. The fourth observation means the Android Developers have recognised the research that I have outlined in Chapter 2 and are actively trying to prevent HTTPS misuse. I continue by focusing on the first and the fifth observations.

The first observation suggests that there might be missing information in the Android Developers tutorial about HTTP(S). After creating an application based on the tutorial, I was still questioning which features of the implementation make a connection be an HTTPS one. In fact, the tutorial tried to address that by saying "Assuming you have a web server with a certificate issued by a well known CA, you can make a secure request with code as simple as this:" and showing the code sample from Listing 4.1.

However, the tutorial does not explain that it is the '*s*' in the URL that enforces that a connection is HTTPS, nor does it warn developers that omitting the '*s*' makes the connection an HTTP one. I suggest that the tutorial should explicitly mention the part of the code that enforce that an HTTPS connection is made. For example, it could say "In the code snippet it is the '*s*' in `https://developer.android.com` that makes

the connection secure. If you remove it and write `http://developer.android.com` instead, then you are making an insecure connection."

The last observation shows that such HTTPS configuration in Android can be achieved in two different languages: Java and XML. In Chapter 2 I recognised why and how HTTPS misuse occurs - customisation due to debugging and self-signed and pinned certificates leads to lack of certificate validation. For example, Listing 4.2 programmers have to write 23 lines of Java code to pin a self-signed certificate and make an HTTPS connections. Alternatively, the same is achieved in Listing 4.3, occupying only 9 lines of XML code.

The XML approach seems simpler and safer – an opinion that is also backed by the tutorial, which says the approach "lets apps customize their network security settings in a safe, declarative configuration file without modifying app code." However, in the tutorial the XML approach is listed after the Java approach, as seen in Figure 4.1. On one hand, if I assume that developers implement the tutorial sequentially, they will be less likely to delete the Java customisation they just created and rewrite it in XML. On the other hand, if I assume that developers only read selected pages of the tutorial, they will pick the Java approach because it is listed first. In both cases they will not try the simpler, safer XML approach.

I suggest that the order of the tutorial is modified so that the XML approach is presented first. The Java approach could even be removed from the hamburger menu of the tutorial, and could be linked from within the webpage of the XML approach. The Java approach cannot be simply removed because it offers a greater degree of customisation compared to the XML one.

4.2 Possible Solutions

After reviewing related literature in Chapter 2 and implementing HTTP(S) connections in an Android application, I selected four possible solutions to the problem

1. A tutorial that explains what encryption is, how to implement it in an Android application, and how to test the implementation. This solution aims to educate developers in order to give them the ability to think critically about their own implementation.

2. Create a wrapper library for implementing HTTP(S) connections in Android that has two requirements. The first requirement is that the library has secure settings by default unless the developer has specified otherwise. For example, it uses HTTPS by default instead of HTTP. The second requirement is that developers who are using the library are less prone to HTTPS misuse. This solution aims to reduce the ability of developers to create flawed implementations.

3. A tool for testing whether the HTTPS connections in Android applications are secure. For example, a developer has created an Android application with HTTPS

Listing 4.2: Adding a custom certificate with Java in Android takes 23 lines of code (counting the semicolons after each statement).

```
    // Load CAs from an InputStream
1.  CertificateFactory cf =
        CertificateFactory.getInstance("X.509");
2.  InputStream caInput = new BufferedInputStream(
        new FileInputStream("load-der.crt"));
3.  Certificate ca;
4.  try {
5.      ca = cf.generateCertificate(caInput);
6.      System.out.println("ca=" +
            ((X509Certificate) ca).getSubjectDN());
7.  } finally {
8.      caInput.close();
9.  }

    // Create a KeyStore containing our trusted CAs
10. String keyStoreType = KeyStore.getDefaultType();
11. KeyStore keyStore = KeyStore.getInstance(keyStoreType);
12. keyStore.load(null, null);
13. keyStore.setCertificateEntry("ca", ca);

    // Create a TrustManager that trusts the CAs
    // in our KeyStore
14. String tmfAlgorithm =
        TrustManagerFactory.getDefaultAlgorithm();
15. TrustManagerFactory tmf =
        TrustManagerFactory.getInstance(tmfAlgorithm);
16. tmf.init(keyStore);

    // Create an SSLContext that uses our TrustManager
17. SSLContext context = SSLContext.getInstance("TLS");
18. context.init(null, tmf.getTrustManagers(), null);

    // Tell the URLConnection to use a SocketFactory
    // from our SSLContext
19. URL url = new URL("https://example.com");
20. HttpsURLConnection urlConnection =
        (HttpsURLConnection)url.openConnection();
21. urlConnection.setSSLSocketFactory(
        context.getSocketFactory());
22. InputStream in = urlConnection.getInputStream();
23. copyInputStreamToOutputStream(in, System.out);
```

Listing 4.3: Adding a custom certificate with XML in Android takes 9 lines of code.

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <network-security-config>
3.     <domain-config>
4.         <domain includeSubdomains="true">example.com</domain>
5.         <trust-anchors>
6.             <certificates src="@raw/my_ca"/>
7.         </trust-anchors>
8.     </domain-config>
9. </network-security-config>
```

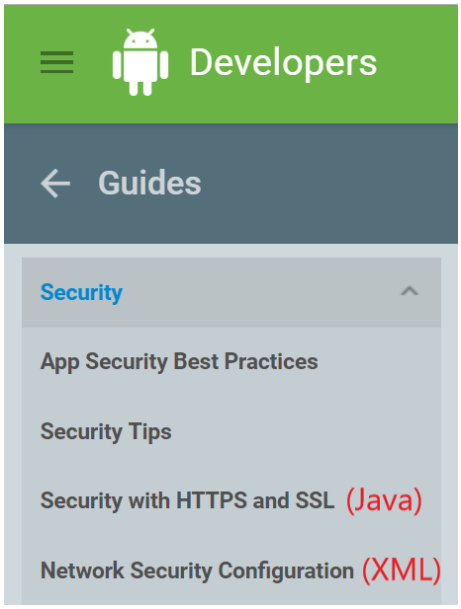

Figure 4.1: The hamburger menu of the Android Developers tutorial where the Java configuration of HTTPS is presented before the XML one.

connections and wants to test whether they are secure before the tool is released. This could be achieved through static or dynamic analysis. This solution aims to give a widely-applicable, public tool for catching flawed implementations.

4. Fixing the advice on Stack Overflow that leads to HTTPS misuse, posting the fixed code in the same thread, and explaining how the other solutions lead to misuse. This solution aims to decrease the number of security flaws introduced through wrong Stack Overflow advice.

Since I had gained an understanding of the misuse of HTTPS, I could come up with possible solutions but could not objectively decide which one is the most suitable.

4.3 Semi-Structured Interviews

Interviews are a fundamental research method for collecting data about people's opinions by asking them questions. In particular, semi-structured interviews require a script of open-ended questions that will be asked, and also allow for new question to be asked during the interview [22]. I decided to conduct semi-structured interviews in order to learn which solution is most preferred.

4.3.1 Preparation

To prepare for the interviews, I created a script that will be read aloud to each participant, followed by a list of questions that will be asked. In the interviews I decided to use the word 'encryption' instead of 'cryptographical protocols' because the former term is synonymous to the latter in informal speech. The full script is included in Appendix A. After obtaining consent from each participant, I went on to ask questions that would reveal

- whether the participant has implemented any cryptographic protocols,
- how much they know about cryptographic protocols,
- what the barriers to implementing cryptographic protocols are,
- how they would order the solutions I have suggested in terms of helpfulness, and
- what resource for learning cryptographic protocols they prefer.

People are more likely to choose the first solutions in a list, rather than the later ones. This phenomenon is called order bias [38]. I presented the possible solutions in a random order each time to avoid order bias.

4.3.2 Results

I interviewed five UG4 students who volunteered to participate in the interviews. Out of the five participants, three we female and two were male; only two were brought

up in the same country but all were European; and three had created an Android application in the past. The participants were aged between 21 and 23 years. One of the participants had also done an internship in cyber security for a military defence company.

Ensuring that their connections are encrypted was not done by any of the participants in the software they built. This could mean that a considerable number of students that are graduating from the University of Edinburgh do not have practical experience with cryptographic protocols. In fact, it is common knowledge that university students in computer science lack cyber security knowledge, and universities are being forced to make cyber security a mandatory part of their computer science degrees [23].

Connecting to a Uniform Resource Locator (URL) is something that all participants had done. A URL is a web address used by a communication protocol (such as HTTP) to locate a web resource. Most of these protocols support the use of TLS or another cryptographic protocol to secure in-transit data. Therefore, all participants had subconsciously made the decision whether to implement secure or insecure connections in the past.

The reasons given by the participants for not ensuring that their connections are encrypted are

- They were designing for a company or for university and the use of any cryptographic protocols was not a requirement.
- They were designing for a company and the application was going to be used only within the company.
- They assumed the API uses cryptographic protocols or said it should have.
- They did not have enough time to learn how to implement cryptographic protocols.
- The data that the application handled was not sensitive or behavioural.
- While the technology that they used can use cryptographic protocols if it is turned on, the tutorial that they used did not explain how to do this.

These barriers indicate that the reasons for not implementing HTTPS are multifaceted and unlikely to go away on their own.

The order of priority of the solutions according to the participants is the same as I presented them above, overall saying that a tutorial would be most helpful. An overview of the tutorial structure could be formed from their opinions. It would use a modular format where several topics are presented, each on its own. The topics would be ordered but also allow advanced users to skip. The topics would ideally be presented in textual and in video format because the participants strongly preferred

one or the other. The tutorial would be interactive, asking the user to implement some functionality. It could even contain its own IDE that the user interacts with – a code box for the user's implementation that can also test the quality of the implementation automatically.

4.4 Final Solution

Around the same time I was finishing the interviews, I came up with another solution inspired by the browser plugin, HTTPS-Everywhere. The idea was that I would design an application that achieves two goals.

- The suggested application enforces that all applications on the device use only HTTPS to communicate with a web service if the service supports HTTPS. In doing so, there is the potential to upgrade the connections of 73.6% of Android applications that use HTTP to communicate with services that support HTTPS [13].
- The suggested application protects all applications on the device that contain flawed HTTPS implementations, such as lack of certificate or hostname verification. In doing so, the communication of the 8% Android application with HTTPS flaws can be fixed [13].

Compared to HTTPS-Everywhere that works only for web browsers, this solution targets all applications on a device because mobile devices spend 85.7% of their time in applications rather than in the browser [11]. Moreover, the solution uses an Android application to achieve its goals through automation. In this way, it is superior to the previously suggested ones. They depend on the manual effort of the developers and assume that they are trusted. Lastly, I believe that the suggested application will generate considerable interest — HTTPS-Everywhere has a combined 3.3 million users between the Chrome, Firefox, and Opera web browsers.

I detail the design of the suggested application in the next chapter.

4.5 Summary

I placed myself in the developers' shoes to learn how they might implement HTTPS connections wrongly. I used my knowledge to come up with possible solutions to help developers and evaluated them objectively through semi-structured interviews. I eventually discovered that an Android application can be designed that automatically enforces HTTP communication when possible and fixes wrongly implemented HTTPS communication.

Chapter 5

Design and Implementation

I have developed two goals whose intention is to automatically upgrade HTTP and fix flawed HTTPS connections in Android. In this chapter, I designed an Android application that would accomplish these goals. To do so, it intercepts the in-transit information of all other applications on the device; these applications are referred to as "clients". It then creates additional connections to forward the information to its destination. Intercepting all connections allows the application to control the HTTP(S) ones. After completing the design, I created a simple, test application to help me with debugging during the implementation phase. I then started working on the application that I had designed. During its implementation, I realised that the application is more complicated than I had anticipated. Since this was the first part of an MInf project, I decided to evaluate whether the current design would be useful rather than completing the application.

5.1 Design

5.1.1 Platform

The application modifies the HTTP(S) connections that all clients are making. I considered several platform options when designing the system – intermediate devices such as routers and Android.

An application for intermediate devices such as routers can be installed on devices that are under the user's control. For example, the user can install the application on their home router, however, they cannot install it on their work router. There is a variety of software like *mitmproxy*[1] that can be used to build the application. However, this solution is impractical for two reasons.

[1] `https://mitmproxy.org/`

Firstly, the user has to be able to understand how to install software on their router. To solve this, a guide could be created that teaches users the basics of their router and the installation process.

Secondly, the user cannot be protected by the application when they connect to the Internet in public places such as cafés or their work. The user does not have an intermediate device that they control there as the router is owned by the public company. To solve this, portable devices have been designed such as InvizBox 2[2] and Betterspot[3]. They act as the intermediate device between the user's devices and the public company's router. They protect the user's connections by forwarding them to a VPN service. However, neither of these devices offers the HTTP(S) protection goals that I have suggested. They only encrypt the in-transit information between the device and the VPN service.

An application for Android devices removes the setup time and the need of additional hardware, associated with the prior suggestion. However, the application would only be able to protect HTTP(S) connections on the Android device it has been installed on. In contrast, the prior suggestion affects any device that is connected to the router.

I chose to build the Android application. In my opinion, it is more likely to be used since it does not require additional user knowledge or hardware. It also works when users are not in control of their routers.

5.1.2 Intercepting the HTTP(S) Connections

To achieve the goals, the application needs to gain control over clients' HTTP(S) connections.

Controlling the clients in order to control the HTTP(S) connections that they make could not be easily achieved. Such behaviour is moderated by the Android operating system, which controls applications with process isolation and a user-based permissions model. For example, if one application tries to modify another, the operating system will block its attempt unless it has appropriate privileges. These privileges cannot be obtained by a third-party application (such as mine) in Android unless the device has been rooted.

Rooting is modifying the Android operating system in order to be able to gain privileged control of the operating system. It usually voids manufacturers' warranty and may render the device non-functional. Hence, it is not widespread and only 27.44% of Android users have rooted their device [30]. I did not assume that the device I am designing for is rooted – I wanted the application to work on any Android device.

[2] https://www.kickstarter.com/projects/683682172/invizbox-2-online-privacy-and-security-simplified

[3] https://www.kickstarter.com/projects/betterspot/betterspot-a-vpn-router-for-all-devices-and-platfo

Intercepting clients' HTTP(S) connections was an alternative, easier approach. I could uses the base class `VPNService` provided by the Android operating system to create a virtual network interface from and to the clients. Thus, it can be configured to intercept all HTTP(S) connections and make them accessible through the interface.

The class is designed to allow developers to create such Android applications that are VPN clients. Once installed on a device, a VPN client would intercept some or all connections that the device is making, and forward them to a VPN server. The VPN server then forwards them to their destination. However, the `VPNService` does not enforce that its implementation follows this scenario. It simply provides a virtual network interface, and the developer is responsible for creating the information flow between this interface and the VPN server. I did not follow the scenario because my application does not control any intermediate devices. Instead it intercepts connections, processes them on the device to achieve the HTTP(S) goals, and then forwards them to their destination.

5.1.3 VPNService within the TCP/IP Architecture

The interface provided by `VPNService` operates on the Internet Protocol (IP). To explain what this means, I am using the TCP/IP architecture – a conceptual model that describes the communication protocols used on the Internet.

The TCP/IP architecture contains four layers: the application, host-to-host transport, Internet, and network interface layers. Each layer corresponds to multiple networking protocols. In particular, HTTP(S) is one of the communication protocols that sit on the application layer and IP is the protocol that sits on the internet layer.

Figure 5.1 shows the path of information that is communicated between a sender and receiver over the Internet; each device is represented by a stack of layers. Information is passed between layers only where they are touching. For example, the application and the Internet layer cannot communicate directly, their communication has to pass through the host-to-host transport layer. Thus, each layer can only function if the layers below it are working correctly.

Since `VPNService`'s virtual interface operates on the IP, it follows that it intercepts the communication of all protocols that sit above it: the protocols of the application and host-to-host transport layer. Thus, the connections I have intercepted use other communication protocols apart from HTTP(S). The class does not provide methods to intercept only HTTP(S), nor to only intercept on particular ports, which could have been useful since HTTP(S) normally uses ports 80 and 443. It does provide a method to only intercept the communication of particular clients. Alas, they can use multiple communication protocols at once and I could not simply intercept from clients that make HTTP(S) connections. However, in Android 91.7% of all networking API calls are related to HTTP(S) [13]. Therefore, not much overhead is caused by intercepting and forwarding all protocols, instead of doing so for HTTP(S).

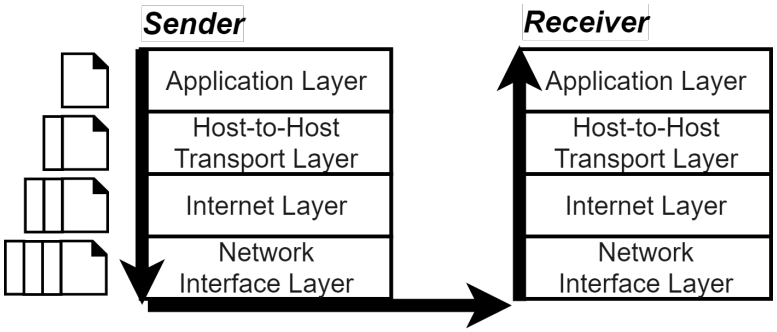

Figure 5.1: Information flow on the Internet according to the TCP/IP architecture. The sender and receiver are each represented by a stack of layers. To the very left is shown that each layer adds or removes a header to or from the beginning of in-transit information. While it is not shown, the network interface layer protocols also add a trailer to the end of the information.

5.1.4 Forwarding the Intercepted Connections

So far I could intercept all clients' connections – this includes HTTP(S) but also other protocols. However, I had not yet forwarded the intercepted information to its destination, and restored the communication between the sender and the receiver. Information is intercepted just before it is forwarded by the IP to the next layer, the network interface layer. Therefore, it was easiest to forward the intercepted information to the network interface layer, and delegate the further forwarding to it. Additionally, using another protocol would have required that I maintained the protocol headers.

5.1.4.1 Limitations of Headers

Figure 5.1 shows that when information has been communicated to a layer, the corresponding protocol adds or removes a header to or from the information. For example, when information reaches the Internet layer in the sender, the IP adds its header to it and then sends it to the next lower layer. After the information reaches the receiver's Internet layer, the IP removes the header from it and forwards it to the layer above.

Therefore, if it forwarded the intercepted information to a layer different from the network interface one, the application would have to add and remove headers. Using another protocol is impractical since adding and removing headers requires additional computation and since devices may process more than tens of thousands of headers per second [7].

5.1.4.2 Review of Libraries

I conducted a review of the available libraries for forwarding in-transit information and of available applications that use the `VPNService`. In both cases I was searching

for approaches to forward the intercepted information with or without modifying the headers. I tested three possible approaches but the first two were unsuccessful.

A connection between the application layer and the network interface layer was the first approach that I considered. If the application sends the intercepted information to the network interface it can delegate to it the responsibility to forward the information further. Moreover, the IP packets (that is, information with an IP header) that are obtained from the intercepting virtual interface can be sent to the network interface layer without modifying headers.

The *pcap* API is a powerful API for intercepting, crafting, and sending packets that supports forwarding to the network interface layer. Its functionality is provided by *libpcap*[4], a wrapper library written in C. Moreover, the functionality of libpcap can be accessed with the wrapper libraries witten in Java *pcap4j*[5], *jpcap*[6], and *jnetpcap*[7].

I created a simple Android application. Since it was a prerequisite for all *pcap*-based Java libraries, I added the C-library *libpcap* to the application and compiled it with the Android Native Development Kit. I found it easiest to work directly with *libpcap* rather than installing and satisfying the dependencies of the Java libraries. Since *libpcap* is in C and the application was in Java, I created a Java Native Interface (JNI) that facilitates the communication between the two.

To establish a connection between the application and the network interface layer I used *libpcap*'s method `pcap_t *pcap_open_live(const char *device, int snaplen, int promisc, int to_ms, char *errbuf)` with the name of the device's physical Wifi interface, '`wlan0`'. However, the application and library did not have sufficient permissions to establish the connection and caused an error. The reason for this is explained later.

A connection between the IP of the application and the web service is the second approach that I considered. It is less convenient to use than the previous one because it requires that the IP headers of the intercepted information be modified, hence, additional computation is required. I created a simple Android application that used the class `Os`, which belongs to the Android system. I created a socket with an arbitrary IP address and connected to it, using the methods `socket(int domain, int type, int protocol)` and `connect(FileDescriptor fd, InetAddress address, int port)`, respectively. However, the application did not have sufficient permissions to establish the connection and caused an `EPERM` error.

An expert in Android security helped me to better understand the role of the permissions in making connections at the different layers of the TCP/IP architecture. To

[4] `http://www.tcpdump.org/`
[5] `https://www.pcap4j.org/`
[6] `https://github.com/jpcap/jpcap`
[7] `http://jnetpcap.com/`

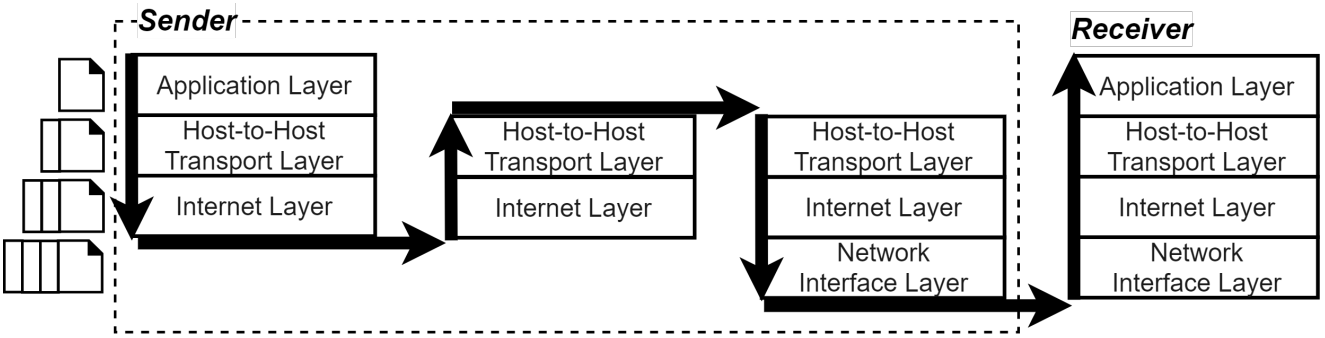

Figure 5.2: Information flow on the Internet when *LocalVPN* is used, according to the TCP/IP architecture. From left to right, the stacks of layers represent the client, the *LocalVPN* application, the Android operating system, and a web service. All stacks that are within the dashed-line rectangle are executed on a single device. Hence, the dashed-line rectangle represents the sender and the right-most stack, the receiver.

establish a connection on the Internet and network interface layers, the Android application requires a networking permission such as CAP_NET_ADMIN. The permission is granted by the Android operating system and can be obtained through a modification of it [37]. Therefore, establishing a connection on the Internet and network interface layers was not useful since I was designing an application that works on any Android device and not just for users who have modified their operating system.

A connection between the host-to-host transport layer of the application and the web service is the third and final approach that I considered. It is the least convenient so far because it requires that both the IP and the transport layer headers of the intercepted information be modified. The approach is used by an application *LocalVPN*[8], that is available on GitHub.

Figure 5.2 shows how the TCP/IP architecture is modified by *LocalVPN*. Information originates in the client (left-most in the figure), where it is intercepted by the `VPNService` at the Internet layer, in particular the IP. The information is then retrieved by *LocalVPN* (middle left) via the `VPNService`'s virtual interface. It contains the headers of all layers down to, and including, the IP. *LocalVPN* removes two of the headers – the IP and transport layer ones – and sends the remaining information to the web service (right-most). It uses the contents of the removed headers to determine the identity of the web service. However, the information is not sent directly, instead the Android OS (middle right) replaces the two headers that were removed and adds a new one – the network interface header. It then forwards the information to the web service.

I could use this approach to restore the flow of the intercepted information between the clients and the web services. In fact, I can use *LocalVPN* as a base for my implementation. However, the *LocalVPN* does not function correctly all the time. For example, the application crashes or all clients do not have access to the Internet. Therefore, I did not reuse *LocalVPN*, instead I decided to implement it anew.

[8] `https://github.com/hexene/LocalVPN`

Listing 5.1: A rule from the *HTTPS-Everywhere Atlas*. It says that the web service that has the hostname "example.com" or "www.example.com" supports HTTPS, and HTTP communication with that service should be upgraded to HTTPS.

```
<ruleset name="Example.com">

        <target host="example.com" />
        <target host="www.example.com" />

        <rule from="^http:" to="https:" />

</ruleset>
```

5.1.5 Controlling the HTTP(S) connections

The resulting design has not yet achieved the main goals of the system

1. The suggested application enforces that all applications on the device use only HTTPS to communicate with a web service if the service supports HTTPS.
2. The suggested application protects all applications on the device that contain flawed HTTPS implementations.

5.1.5.1 Goal 1: Enforcing HTTPS

To satisfy the first goal, I decided that the application will monitor the intercepted connections for HTTP communication. The application will not forward such communication immediately. Instead it will connect to the *HTTPS-Everywhere Atlas*[9] ruleset and check whether the destination of the communication, that is a web service, supports HTTPS. An example of a rule from the ruleset is shown in Listing 5.1. If the web service supports HTTPS then the application rewrites the URL from the intercepted connection. For example, `http://example.com` would be rewritten to `https://example.com`. Then the application makes an HTTPS connection to the web service, requesting the web service to respond with the HTTPS protocol and not the HTTP one. When the service's response is received, headers are added to it, and it is forwarded to the client via `VPNService`'s interface.

On one hand, if an HTTPS response is received back, the second goal will ensure that it does not use flawed HTTPS. Thus, a Man-in-the-Middle attack of the connection is (theoretically) not possible. On the other hand, if an HTTP response is received back this might be due to two reasons: an intermediate device is causing a Man-in-the-Middle attack or the service that responded has been misconfigured. In both cases the application stops the connection because it should be an HTTPS one. As a result, the application enforces that all clients use only HTTPS to communicate with a web service if the service supports HTTPS.

[9] `https://www.eff.org/https-everywhere/atlas/`

5.1.5.2 Consideration of Alternative Approaches

An HTTP message (called an upgrade message) can be used by the sender or the receiver to request that HTTP communication is upgraded to HTTPS. The current design rewrites HTTP URLs and makes an HTTPS connection to the web service. Instead, it is possible that the application creates an upgrade message and sends it to the client. According to HTTP's specification, the client should then reestablish the original connection but in HTTPS.

I chose not to use this approach. The Android class used for making HTTP(S) connections does not support automatic upgrades. For example, if the developer creates an HTTP connection to `http://example.com` and the web service (or my application) responds with an upgrade message, the class object does not redirect – it raises an exception. Developers have the ability to handle the exception and manually create redirecting functionality. Since Android applications are created by novice and professional developers alike, I assume that a large number of applications will not support HTTP upgrades.

5.1.5.3 Goal 2: Protecting against Flawed HTTPS

To satisfy the second goal, I decided that the application will monitor the intercepted connections for HTTPS communication. As mentioned in Chapter 2, these connections can be flawed due to two common reasons

- accepting any certificate even if it cannot be verified by a certificate authority that is installed on the application or on the device, and
- accepting a certificate for any hostname as long as it can be verified by a certificate authority that is installed on the application or on the device.

HTTPS communication starts with a TLS handshake. To successfully complete the handshake the web service provides its certificate, which is checked for validity by the client. However, 8% of Android applications do not correctly check the validity of the certificate due to the flaws above [13].

The certificate is communicated in plaintext. Therefore, certificates can be read from the intercepted connections and checked for validity. If a certificate is not valid, then the application stops the HTTPS connection.

5.1.5.4 Consideration of Alternative Approaches

The application cannot read or modify the in-transit information once the TLS handshake has been completed. Such information travels encrypted and the application does not know the keys that decrypt it. Provided that it has a certificate installed on the device, the application can cause a Man-in-the-Middle attack. Thus, the client and the web service do not communicate directly but each communicates with the application, which can now read and modify the information.

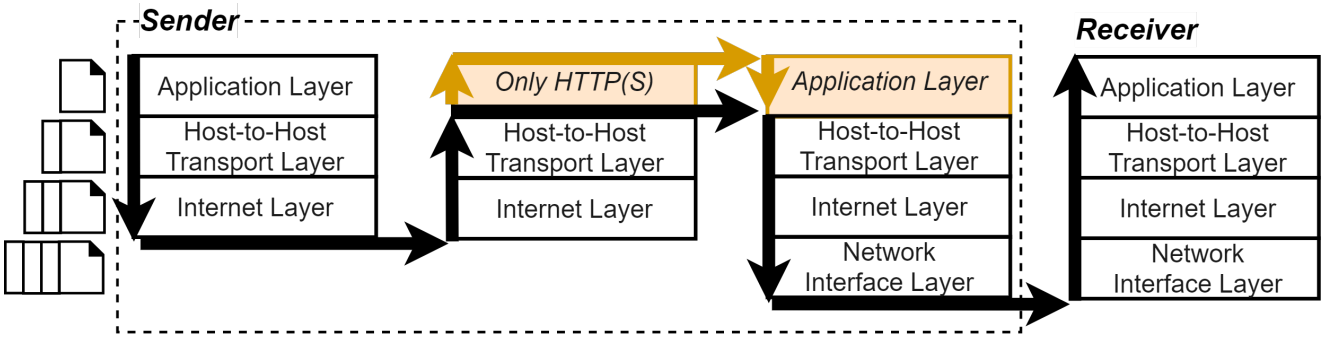

Figure 5.3: Information flow on the Internet under the final design, according to the TCP/IP architecture. From left to right, the stacks of layers represent the client, the *LocalVPN* application, the Android operating system, and a web service. All stacks that are within the dashed-line rectangle are executed on a single device. Hence, the dashed-line rectangle represents the sender and the right-most stack, the receiver.

I did not choose this approach due to two important limitations. Firstly, it encrypts the information twice and decrypts it once, whereas information is otherwise only encrypted once. Encryption and decryption are computationally costly operations [34]. Secondly, the other 92% of Android applications do not use flawed HTTPS. If the suggested application has a security flaw then by decrypting these applications' connections I am also making them insecure. Furthermore, no flaws have been observed in other aspects of the TLS handshake besides certificate validation, hence, I can focus only on it.

5.1.6 Final Design

The final design of the suggested application is illustrated in Figure 5.3. It uses the `VPNService` class to intercept all applications' connections. Then all connections, but HTTP(S) ones, are forwarded to their destination by the host-to-host transport layer. HTTP(S) connections are further moderated – some HTTP ones are upgraded and the validity of the certificates is checked in HTTPS ones.

There are several limitations of this approach.

- Clients that use pinned certificates may not be able to make HTTPS connections. The application only has access to the certificates that have been installed on the device. Thus, it will not allow HTTPS connections that rely on pinned certificates that are not installed on the device, because these certificates' validity fails when checked by the application.
- The HTTPS-Everywhere ruleset has been designed to enforce HTTPS for webpages. As such, it may not contain rules for most of the web services that Android applications communicate with. Then the application would not be able to enforce HTTPS for many services. This can be solved by manually creating a list of such services that support HTTPS. It can be used in conjunction with the

ruleset.

- The suggested application may require a significant amount of resources, such as CPU time and battery power. It more than doubles the number of layers in the TCP/IP architecture and relies on header modification.

5.2 Implementation

5.2.1 Creating a Test Application

I started the implementation phase by creating a simple application. It can be used to test the implementation of the suggested application. I reused my Android application that creates HTTP connections from Chapter 4. Then I modified it to create such connections that support my design goals directly

- An HTTP connection to a server that does not support HTTPS. This connection should not be upgraded, and if it is then the connection does not work.
- An HTTP connection to a server that supports HTTPS. This connection should be upgraded.
- An HTTPS connection to a server that supports HTTPS. This connection should continue using HTTPS.

Each type of connection can be made at the press of a button and the server's response is printed inside the user interface. From the response, it can be determined if the connection was successful or an error occurred instead.

For the first type of connection I decided to use a university service, making connection to `http://www.drps.ed.ac.uk/`. This is a university owned webpage that does not support HTTPS and raises an error instead of trying to downgrade to HTTP. For the latter two, I decided to use the *Informatics homepages web service*[10] since it supports HTTPS. It also allows me to create my own webpage and control its responses. I used PHP, a server-side scripting language, to create a webpage that responds in the same protocol that it received a request in. Thus, I can test whether the HTTP request was upgraded by the application that I am designing.

5.2.2 Creating the Application

I then started implementing the suggested application according to the design that I presented above. I created a placeholder Android application in Java. I used the `VPNService` class to intercept the connections of all clients. I used its method `addRoute("0.0.0.0", "0")` to intercept the connections that have the destination IP address `"0.0.0.0/0"`, that is, any IP address. The class returns a virtual network interface, which has an underlying `FileInputStream` and `FileOutputStream`. The

[10] `http://computing.help.inf.ed.ac.uk/homepages`

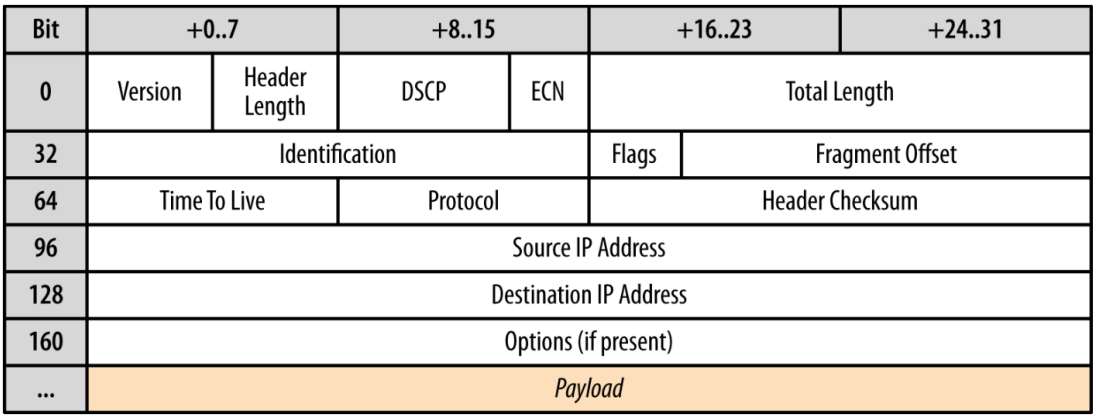

Figure 5.4: The IP version 4 (IPv4) header. The figure was taken from Grigorik, 2013 [20].

input stream provides access to the IP packets sent by the clients; the output stream is used to send IP packets to the clients.

As previously established, the IP packets may contain headers for any application and transport protocol. Moreover, the application would remove the IP and transport header from each packet. It would then forward the remaining information to its destination with the transport protocol specified in the headers. In Java there are two main transport protocols – the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP) [36]. Since they are not interchangeable, I needed to support both transport protocols. I decided to first implement functionality for forwarding information with UDP. Compared to TCP, UDP uses a simpler handshake.

I created two concurrent threads, each associated with one of the streams. The input thread intercepts the IP packets sent from the clients, removes its headers, and forwards it to the web services. The output thread receives information from the web services, adds headers to construct an IP packet, and forwards it to the clients.

5.2.2.1 Removing Headers

The intercepted packets all have the same IP header (Figure 5.4) and UDP header (Figure 5.5). When the application retrieves a packet from the input thread, it uses the *(header) length* field to determine the size of the headers and remove them. In addition, the header fields are saved. They will be used to forward information and construct an IP packet later. Bit operations[11] are used to read these fields since they vary in bit size.

The IP has two versions – 4 and 6. The application only supports IPv4. As I have outlined in Chapter 7, it will be extended to support IPv6 too. IPv6 is IP's latest version and only became an Internet standard on 14 July 2017 [44].

[11] `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html`

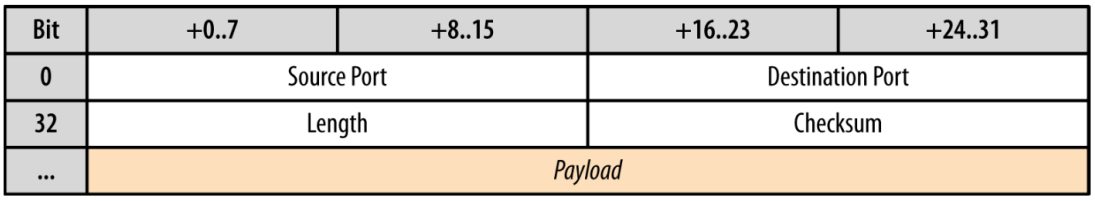

Figure 5.5: The UDP header. The figure was taken from Grigorik, 2013 [20].

5.2.2.2 UDP forwarding

The application uses UDP connections to forward the remaining information to the web service. I used a `DatagramChannel` to open these connections. The *destination IP address* and the *destination port* fields are used to uniquely identify the web service. It is important to note that I explicitly protect the forwarding channels from the `VPNService`. Thus, they are not intercepted by it and their connections can reach the service.

A UDP connection is persistent, meaning that once it is established the connection can be used to send information multiple times. It is only destroyed when one of the communicating devices closes it. I used the Google Guava library[12] to create a least-recently-used cache for UDP connections. As a result, the connection to a web service is not redone if it already exists. The cache has capacity of 2000 connections and also closes connections that have not been used for more than 5 minutes. These parameters can easily be optimised when the application is evaluated.

5.2.2.3 Receiving information from web services

When a UDP connection is opened, it is also registered to a `Selector`. The `Selector`[13] examines all UDP channels and determines when a web service has responded. It then asks the output thread to handle the response.

5.2.2.4 Restoring Headers

The output thread receives a response that does not have UDP or IP headers. However, the `VPNService`'s output stream needs them to deliver the in-transit information to the correct client. The application creates an empty `ByteBuffer` that will become an IP packet. The received information is inserted into the buffer at an offset that leaves enough space where the UDP and IP headers will be created.

To create the IP and UDP header, the application reuses some fields. For example, *source port* and *destination port* are taken from the original packet but swapped. Other

[12] `https://google.github.io/guava/releases/17.0/api/docs/com/google/common/cache/package-summary.html`

[13] `https://docs.oracle.com/javase/7/docs/api/java/nio/channels/Selector.html`

fields such as *checksum* are calculated. I created the checksum method because it did not exist in Java. It calculates a 16bit one's complement sum [39]. Additionally, importing a third-party library, such as *libpcap*, only for its checksum method is likely to cause unnecessary consumption of device resources.

The response from a service can be of arbitrary length. However, the IP protocol is only able to carry information of size up to 65,535 bytes [26]. Thus, the application will have to also fragment big responses and send multiple IP packets back to the client. Fragmentation required that I maintain additional sessions between the application and the client. For example, fields like *identification* and *fragment offset* need to be used in order to identify each block of information that has been fragmented and each fragment, respectively. As a result, the complexity of building the application increased.

5.2.2.5 Progression to Evaluation

I had planned to evaluate the usefulness of the application after building it. However, during the implementation phase the application's complexity increased. I observed that to restore the information flow between the clients and services, support for multiple transport and Internet protocols has to be implemented. That is, the UDP and TCP transport protocols and the IPv4 and IPv6 Internet protocols. I started adding support for UDP and IPv4. However, the protocols required that not only headers are added or removed, but that the application maintains different sessions for single fields in the headers. I did not want to spend more time than planned on a design that might not be useful. For this reason, I decided to instead use an alternative method to evaluate its usefulness. Moreover, this was the first part of an MInf project. Provided that the evaluation is positive, I have another year to finish the implementation.

5.3 Summary

I designed an Android application that would accomplish the HTTP(S) goals from the previous chapter. It is responsible for intercepting the communication of all clients and forwarding it to the corresponding web services. This allows the application to upgrade HTTP and fix flawed HTTPS connections. I created a simple, test application to help me with debugging during the implementation phase. I then started implementing the suggested application. However, the complexity of building it kept increasing. Since this was the first part of an MInf project, I decided to evaluate whether the current design would be useful rather than completing the application.

Chapter 6

Evaluation

As discussed in Chapter 5, the application has three main limitations. One of them is that the application might consume a considerable amount of the device's resources. I was not able to evaluate it because the application was not finished. However, I was able to evaluate the two other limitations – whether the application would break some HTTP and some HTTPS connections. Since I need test data, I obtain a pre-existing dataset that contains the HTTP(S) connections made by an Android device. I extract these connections and recreate them to evaluate the limitations. In the end, I conclude whether the application would be helpful, or whether it would rather break many connections and make the client applications unusable.

6.1 Limitations of the Application

Pinned certificates that are not installed on the device are sometimes used in clients. One of the limitations of the application is that it will break the communication that relies on such certificates. The device requires that a web service sends a certificate before an HTTPS connection is established. If the service sends a certificate, the device cannot check its validity unless the same certificate has been stored on it. The current design says the application stops HTTPS connections if their certificates are not valid. Thus, if clients rely on pinned certificates that are not installed on the device, then they will not be able to make connections. If many clients do so, then the application might deteriorate users' experience – they will not be able to use the clients and might find the application unhelpful and unusable.

I decided to create a list of HTTPS connections made by clients by recording the HTTPS requests that lots of Android applications are making. These requests would be in the form of a URL, which identifies a web service. I would then use the URLs to recreate HTTPS connections. Due to the use of HTTPS, the web services are required to send me their certificates. I could check the validity of these certificates. If the certificate is valid according to the device, then that connection could be verified by the suggested application too – it relies on the device's installed certificates. However, if the certificate was not valid, the client must use a pinned certificate that has

not been installed on the device. Thus, I have obtained an objective measure of how many connections use such pinned certificates and would be rendered unusable by the application.

The HTTPS-Everywhere ruleset was designed for web pages and might not contain rules for the web services that Android devices use. However, the application is not able to determine whether it should enforce HTTPS to any services that are not in the ruleset. I would have to manually add to the ruleset such services, if I want their connections that use HTTP to be upgraded to HTTPS. Alternatively, I could create a separate list that is used by the application that complements the ruleset.

I decided to create a list of HTTP connections made by also recording the HTTPS requests that lots of Android applications are making. I would then modify their URLs to create HTTPS connections and determine how many services support responding in HTTPS. The result can be used to later create the ruleset or list of servers for which HTTPS should be enforced. Additionally, this work can be used to establish how many connections might be broken if the suggested application simply enforced HTTPS on all HTTP communication.

6.2 Evaluation

6.2.1 Obtaining a Dataset

I identified the need for a list of HTTP and HTTPS connections made by Android applications. To create it I used a dataset[1] provided by my supervisor, Kami Vaniea. The dataset was originally generated by intercepting all the packets that were sent by an Android device when it was connected to the Internet. The Android applications that complement 14 Internet of Things (IoT) devices were also installed on the device. A *Samsung Galaxy Tab A (SM-T550)* tablet was used, running *Android 5.0.2* with *Google Play Services* disabled. It was also rooted to allow running *tcpdump*, the software that intercepts the connections.

During the generation all the information that was sent from the tablet to the Internet was captured into 28 *pcap* files. There are multiple files because multiple captures were recorded – one for each IoT device's application. Each *pcap* file contains the information of all communication that the device sent and received, split into packets. Each packet includes the headers of all protocols that were used, such as HTTP(S).

[1] `https://groups.inf.ed.ac.uk/tulips/projects/iotlab/`

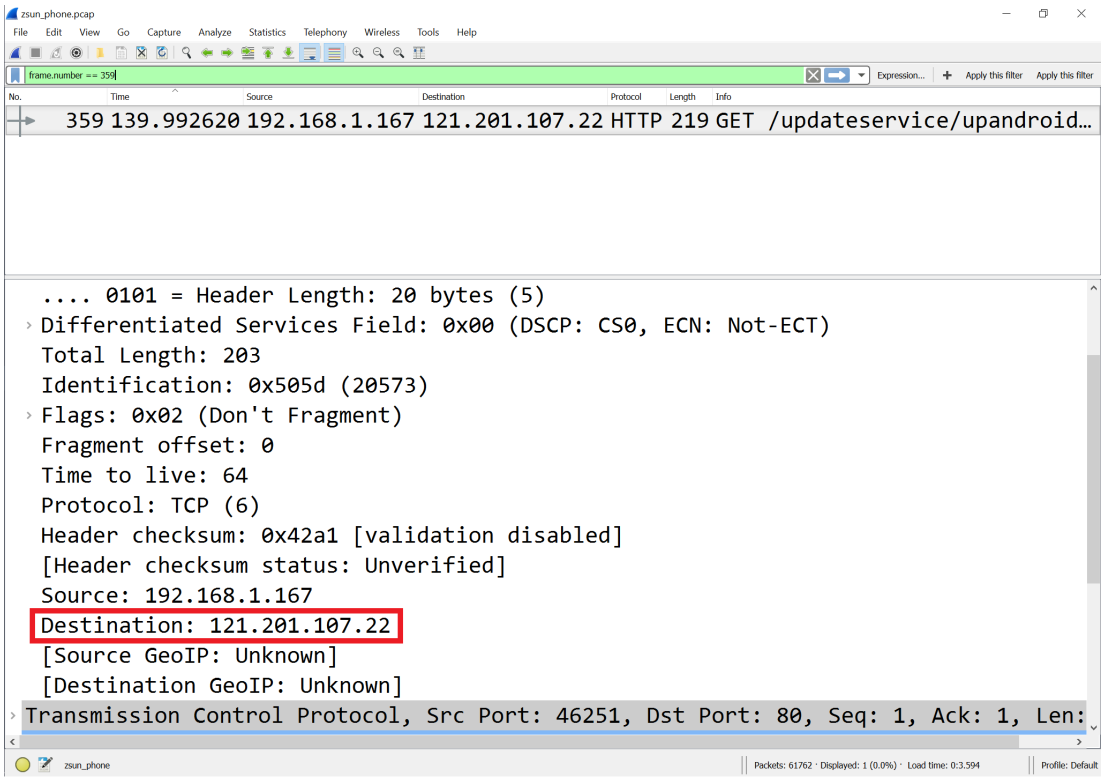

Figure 6.1: The resolved destination of an HTTP connection. It was taken from the IP header, using Wireshark.

6.2.2 Extracting HTTP(S) Connections

I started processing the dataset by using *Wireshark*[2], a free, open-source analyser for such communication. However, the packets contained the IP addresses of the web services, rather than their hostnames in URLs. This was a problem because multiple web services can correspond to one IP address. The number of existing web services is greater than the number of available IP addresses. Hence, a Domain Name System (DNS) is used that resolves the hostname of the web service into an IP address [33]. The device then uses the address to connect to the web service. An example from the dataset can be seen in Figure 6.1.

I could not simply connect to the resolved IP address as the hostname that it points to changes across time and location. Therefore, if I did so I would have connected to an arbitrary web service. Thus, I would first have to restore the original hostname from the IP address, that is, unresolve it. The original hostname is recorded in the messages that were used during DNS resolution, which are also contained in the *pcap* files. Such a message is shown in Figure 6.2. I used Wireshark to automatically unresolve the IP addresses into the hostnames that they represent. I obtained the result in Figure 6.3.

I proceeded to extract all HTTP and HTTPS connections that were made with the command-line version of *Wireshark* called *tshark*. For HTTP connections I extracted

[2] https://www.wireshark.org/

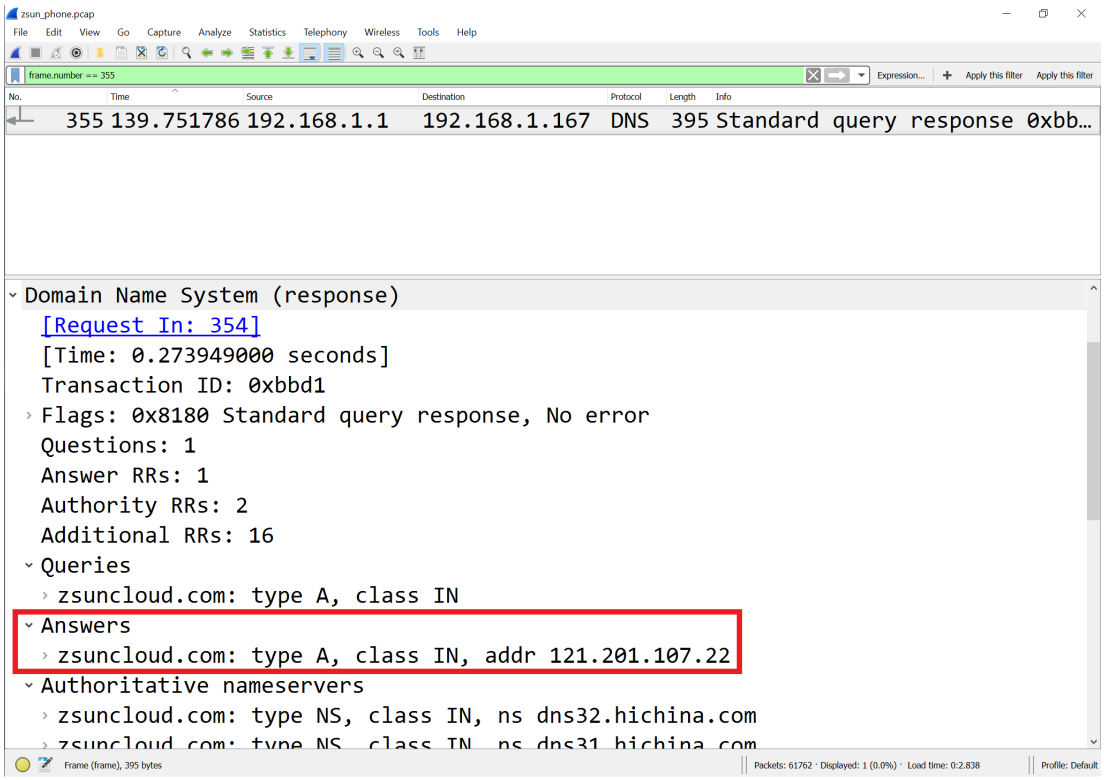

Figure 6.2: The DNS message that contains the IP address and the hostname of the web service.

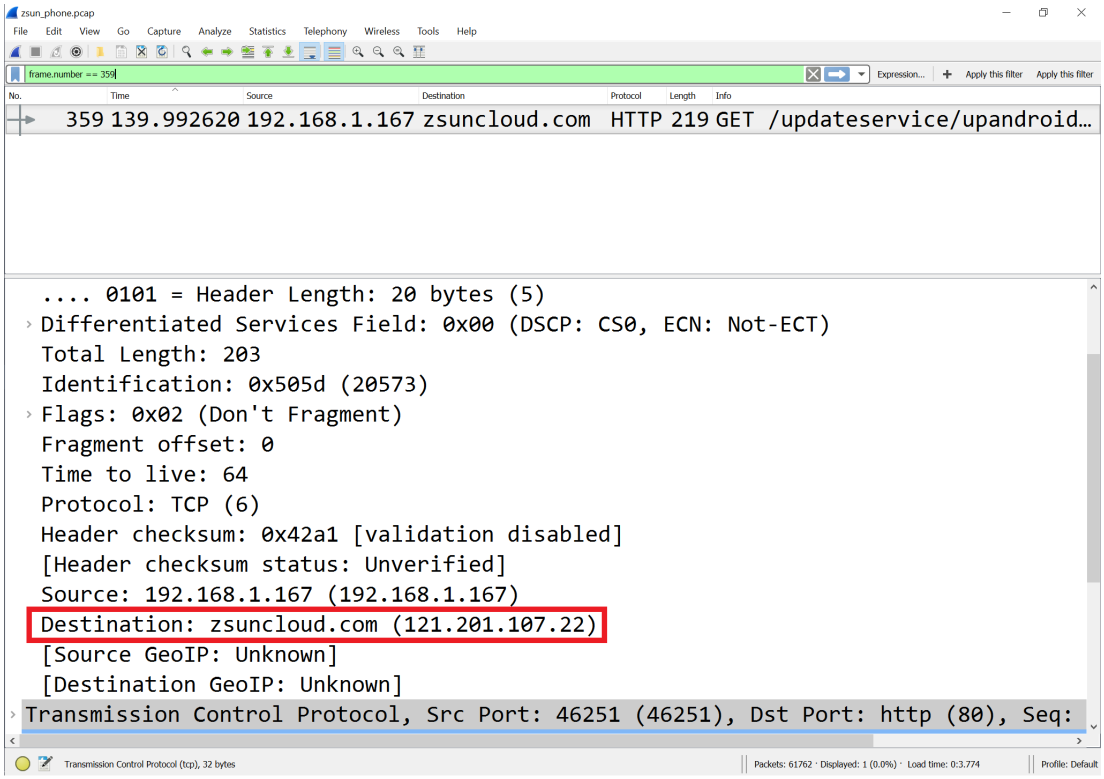

Figure 6.3: The unresolved destination of an HTTP connection.

Listing 6.1: An HTTP request that contains the unresolved hostname and service's port, and the full URL, which has the GET parameters.

```
eu10.vimtag.com 3080
http://149.202.201.87:3080/ccm/cacs_reg_req.js?
```

Listing 6.2: An HTTPS request that contains the unresolved hostname and the service's port, and the value of the SNI field.

```
l11083s.cdngc.net 443 uk-odc.samsungapps.com
```

the unresolved hostname and the service's port into a list. They uniquely identify a web service. I also saved the full URL, which contains the *GET* parameters that say what resource is being requested [15]. A line from the list can be seen in Listing 6.1. For HTTPS connections I also extracted the unresolved hostname and the service's port into another list. Additionally, when the field was present, I saved the Server Name Identification (SNI) field from the TLS header. I could not store the *GET* parameters from the HTTPS communication, because they travel encrypted and the connections were not decrypted during their recording. A line from the second list can be seen in Listing 6.2.

I stored the SNI field because the hostname may not actually point to the web service that an application connects to. Content delivery networks[3] (CDNs) are servers that are located strategically, so that they provide fast delivery of Internet contents. Web services use CDNs, which cache some of their information. When the cache is located closer to the user than the service, then the cache is used to deliver the information instead of the web service. For example, an application requests information from the web service `uk-odc.samsungapps.com`; however, the CDN has identified that its service `l11083s.cdngc.net` can deliver the content faster. As seen in Listing 6.2, the original web service is recorded in the SNI field, while the CDN service in the unresolved hostname.

6.2.3 Recreating the HTTP(S) Connections

To recreate the HTTP(S) connections from the extracted URLs I used the python package *Requests*[4] on a *Surface Pro 3* with *Windows 10 Pro* version 1709. In particular, I found the `get()` method useful to send an HTTP *GET* request to a URL. I used a timeout of 5 seconds. It means that the method sends a request to the server, and then only waits 5 seconds for the service's response. After this time has passed the method will raise a timeout error.

[3] `https://httpd.apache.org/docs/2.4/vhosts/`

[4] `http://docs.python-requests.org/en/master/`

The HTTP connections I saved were used to approximate how many of all HTTP connections are broken if the application rewrites them to HTTPS ones. Additionally, the list of all connections that can be rewritten can be added to the application's ruleset.

The information that I extracted from the *pcap* files contains the unresolved hostnames, service's ports, and full URLs with the *GET* parameters. It can be used to construct two URLs – one that makes an HTTP connection and another that makes an HTTPS connection. Thus, I could first test whether the former URL is still alive and if yes, then check if the latter works.

I took the full URL, such as `http://149.202.201.87:3080/ccm/cacs_reg_req.js?` and replaced the IP address with the hostname – `http://`*eu*10.*vimtag.com*`:3080/-ccm/cacs_reg_req.js?`. If the full URL does not already contain the port, I added it too. Thus, I constructed the final version of the HTTP URL. I then created the HTTPS URL. I modified what I had so far by adding an '*s*' into `http://`. Then I also changed the port to 443, the default port for HTTPS [40], I obtain the second URL too – `https://`*eu*10.*vimtag.com*`:443/ccm/cacs_reg_req.js?`.

I use `get()` with each URL to request the resource from the web service. During the connection I record

- Whether the URL is alive. If it is not, an exception will be raised by the library. These are exceptions due to invalid certificates, timeouts, or invalid URLs.
- The final URL in case the service responded with any redirects and the library followed them.

A service might send a redirect if it wants to change the HTTP(S) protocol that is used. For example, the library requests `https://example.com`. However, the service does not support HTTPS and sends back a redirect to the library with the URL `http://example.com`, enforcing the library to use HTTP.

I later chose all alive HTTP connections and evaluated whether they can be upgraded to HTTPS. I determined this using the final URL. If it containsd the '*s*' in `https://`, then the web service supported the upgrade. Otherwise, the service did not support the upgrade and redirected to HTTP.

The HTTPS connections that I extracted can be used to determine how many of all HTTPS connections rely on a certificate that has been pinned in the client but has not been installed on the device. The information that I extracted from the *pcap* files contains the unresolved hostnames, service's ports, and the SNI field if it was specified. I use the SNI field when present, or the hostname otherwise, in combination with the port to create an HTTPS URL. For example, `http://uk-odc.samsungapps.com:443/`.

To create the connections I use the `get()` method again and record whether the URL is alive and what the final URL is.

I later determine that all URLs that are alive do not use a certificate that has been pinned in the client but has not been installed on the device. The rest of the connections result either in errors that mean they use such certificates, or other unrelated errors.

The device's installed certificates are used to check the validity of the certificate that the server sent. However, the certificates on the *Surface Pro 3*, which I am using to recreate the connections, are different from these in the *Samsung Galaxy Tab A*, which was used to generate the dataset. Hence, if a certificate check says the certificate is invalid, this might not mean that the connection relies on a certificate that has been pinned in the client but has not been installed on the *Samsung* device. Instead, it might mean that the certificate is not installed on the *Surface* device but was installed on the *Samsung* one. Thus, running the experiment with a different set of certificates makes it non-comparable. As a result, I extracted the certificates from the *Samsung* device and installed them onto the *Surface* one. Only then did I proceed to recreate all the connections.

6.3 Results

6.3.1 Overview of Extracted Connections

From the *pcap* files I extracted all 3293 HTTP connections (that is, URLs) that were made by various Android applications on the *Samsung* device. In particular, I extracted all *GET* and *POST* requests. These connections were made to 62 web services, as determined by the number of unique hostnames in the URLs. On average 54.8 connections were made to each web service, with a standard deviation of $\sim$135. During the recreation of the connections, 41 (2% of all HTTP connections) of them, which were made to two services (2% of the HTTP services), failed because the service did not respond.

I also extracted 1885 HTTPS connections. They were determined by extracting the first *Hello* message of the TLS protocol. These connections were made to 105 web services, as by the unique hostnames in the URLs. On average $\sim$18 connections were made to each service, with a standard deviation of $\sim$28. Only 773 of the URLs were alive, which correspond to 45 web services. From the remaining 2519 connections – the URLs of 983 connections that correspond to two services were invalid, and 1437 connections that correspond to 15 services failed because the service did not answer.

6.3.2 Fixing Flawed HTTPS connections

I previously observed that some developers hinder the security of HTTPS – their implementations accept any certificate or any hostname, even if they are not valid. The suggested application fixes these flaws by forcing that certificates and hostnames are checked for validity. However, these checks are not always helpful.

Pinned certificates that are not installed on the device are sometimes used by clients. Connections that rely on such certificates will be stopped by the application because their validation check will always fail – the application does not have access to the

Table 6.1: This table shows how many recreated HTTPS connections were successful, both per connection and per web service. Unsuccessful connections due to an invalid certificate or hostname are also reported. Overall, 1885 HTTPS connections were made to 105 web services.

	Successful	**Invalid Certificate**	**Invalid Hostname**
Connections	1613 (86%)	134 (7%)	97 (5%)
Web Services	77 (73%)	21 (20%)	5 (5%)

Table 6.2: The table compares the number of successful connections when *Samsung* certificates are either checked by the app or not.

	Check validity	**Don't check validity**
Connections	1613 (86%)	1715 (91%)
Web Services	77 (73%)	89 (85%)

certificate that is pinned in the client and can be used for successful validation. There is no mechanism for the application to request the certificate from the client due to the operating system's process isolation. Stopping all connections that rely on such certificates may negatively effect the clients' functionality and user experience.

Table 6.1 says that the certificate validation check failed for 7% of all 3293 HTTPS connections, which were made to 20% of all services. When these connections were made originally, while the *pcap* files were being captured, they could have only worked if these certificates were pinned to the clients. Therefore, I assume that 7% of all HTTPS connections and 20% of all web services use pinned certificates that are not installed on the device.

Out of the connections that result in invalid certificates, 76% were made to different *Samsung* services, and correspond to 60% of the services that use pinned certificates that are not installed on the device. A possible solution is for the application to be disabled for certain services. For example, if the application does not check the validity of *Samsung*'s certificates, 5% more connections become successful to 12% more services (Table 6.2).

Hostnames different from the hostname on the certificate are another reason for failed verification checks. For example, when an HTTPS connection request was sent to `https://api-diagmon.samsungdmroute.com`, the service responded with a certificate as a part of the TLS handshake. However, the certificate was for two hostnames that are different from the requested service's hostname – `*.samsungdm.com` and `samsungdm.com` are both different from `api-diagmon.samsungdmroute.com`. When the service's hostname is different from certificate's hostname the certificate is also not valid.

Of all HTTP connections, 5%, which correspond to 5% of the services, would be stopped by the suggested application due to such invalid hostnames (Table 6.1).

Out of the connections that result in invalid hostnames, 78% were communicating to different *Google* services, which correspond to 40% of the services that were contacted and sent an invalid certificate. Similarly to invalid certificates, the application could be disabled for certain services. For example, if the application does not check both *Samsung*'s certificates and the hostnames of *Google*'s certificates, only 5% of all HTTPS connections are stopped by the application. This is a considerable decrease since 14% of all HTTPS connections are stopped otherwise.

6.3.3 Upgrading HTTP Connections

I noted that the *HTTPS-Everywhere* ruleset was designed for web pages and might not contain the rules for many web services. Without these rules, the application cannot enforce HTTPS for web services – it can only enforce the protocol for services that support it. I decided to create a list of the services that use HTTP but also support HTTPS. It could also be used to report how many HTTP connections are upgradable. I extracted 3293 HTTP URLs from the IoT dataset, which point to 62 unique web services. Then I recreated each connection, but I also recreated an HTTPS version of it. Thus, I obtain a list of services that could be added or used in addition to the ruleset.

Of the alive HTTP connections that were recreated in HTTPS

- less than 1%, which corresponds to 6% of the alive services, did not support the upgrade to HTTPS. This means that most services support HTTPS, however, they did not enforce that it is used.
- 37%, which correspond to 67% of the alive services, were successful. These are the connections that could be upgraded by the suggested application.
- 43%, which correspond to 9% of the alive services, did not work due to the service sending an invalid certificate. These are the connections that rely on pinned certificates that have not been installed on the device. They would be stopped by the application because their certificates do not pass the validation check. Since only four services rely on such certificates, an exception could be easily added to the application. The exception means that the certificates for these services are not checked for validity.
- 20%, which correspond to 18% of the alive services, failed due to an invalid hostname. Of these connections 97% communicated with *Samsung* services. As mentioned earlier, the solution to this is adding an exception to the application such that it does not check the hostname for *Samsung* services.

As a result, I identified 30 web services that support HTTPS out of all 62 services that made HTTP connections. These services can be added to the *HTTPS-Everywhere* ruleset or a custom ruleset in the application. Then the connections that they make can be automatically enforced to use HTTPS, and therefore, be protected by the cryptographic protocols.

Table 6.3: This table shows how many recreated HTTPS connections were successful, both per connection and per web service. Unsuccessful connections due to an invalid certificate or hostname, or a lack of response are also reported. Overall, 773 HTTPS connections were made to 45 web services – they were a modified version of the set of alive HTTP connections.

	Successful	**No response**	**Invalid Certificate**	**Invalid Hostname**
Connections	286 (37%)	3 (<1%)	329 (43%)	155 (20%)
Web Services	30 (67%)	3 (6%)	4 (9%)	8 (18%)

6.4 Discussion

As a result of the evaluation, I established that the application is capable of upgrading the connections that are made to 48% of the clients that did not use HTTPS before. Moreover, it would fix the possible flaws in the connections that are made to 86% of the services that use HTTPS. To my mind, this is significant enough to justify that I should build the application. However, I observed that allowing the user of the application to add exceptions for certain web services would be essential to the design.

Web services use pinned certificates that are not installed on the device. They also have hostnames different from the ones included in the certificates that they send. Thus, without the option to add an exception 12% of the HTTPS connections and 63% of the upgraded HTTP connections would be stopped by the application. This is likely to have a large, adverse effect on the functionality of the clients, and even make them unusable. In my opinion, users are very likely to not use the application if it breaks the other applications on their device. However, if the users have the ability to add exceptions, the communication of all connections that are stopped by the application but are essential can be restored. Therefore, design of the application will have to give users the ability to add exceptions to the rules.

6.4.1 Limitations of the Evaluation

After the evaluation I considered different aspects of it and challenged my assumptions. Doing so, I discovered two limitations of my approach.

The invalid certificates and hostnames were assumed to be caused by pinned certificates not on the device and misconfigured servers, respectively. However, it is possible that they were caused by a Man-in-the-Middle attacker who was active while I was running the experiments. In the beginning of each HTTPS connection, the service is asked to provide a certificate. The certificate is checked for validity and used to make sure that the device is communicating with the service and not someone else. An attacker who is trying to impersonate the service could be sending invalid certificates or valid certificates with a different hostname from the service's. Their action would have achieved the same outcomes as the ones I am observing.

Android applications for IoT devices were used to create the dataset that I used. The number of connections that are upgradable could be different in applications from domains. It is important to note that the information that IoT devices record is considered sensitive [5]. Therefore, more services from the IoT domain might support HTTPS than would be in other domains.

6.5 Summary

I extracted the HTTP(S) connections from a dataset of Android Internet traffic. My goal was to evaluate whether many of these connections would be stopped by the suggested application due to the use of pinned certificates. Furthermore, I was unsure if the *HTTPS-Everywhere ruleset* would contain rules for the services that Android clients connect to. I processed a a pre-existing dataset, recreating the HTTP(S) connections from it. I concluded the application can enforce HTTP communication with 48% of all services that normally use HTTP, which is enough to me to justify building it further. I also observed that design of the application will have to give users the ability to add exceptions to its rules. Otherwise, the application may break the correct functionality of some clients and deteriorate the user experience.

Chapter 7

Conclusion

The (correct) usage of cryptographic protocols, in particular HTTPS, is essential for ensuring confidential, integral, and authenticated communication over the Internet. As we have seen, developers that work with HTTPS are currently making many errors from simply not including an '*s*' the URL to not checking the validity of certificates correctly. The purpose of my project was to

> "Help application developers correctly use encryption libraries."

During the requirements gathering stage I realised that expecting developers to do all the necessary steps to implement HTTPS (correctly) is unrealistic. I then designed an Android application that would take responsibility away from developers through automation. It aims to achieve two goals

- The suggested application enforces that all applications on the device use only HTTPS to communicate with a web service if the service supports HTTPS. In doing so, there is a potential to upgrade the connections of 73.6% of Android applications that use HTTP to communicate with services that support HTTPS [13].
- The suggested application protects all applications on the device that contain flawed HTTPS implementations, such as lack of certificate or hostname verification. In doing so, the communication of 8% Android application with HTTPS flaws can be fixed [13].

I started implementing the application, which relies on Android's base class `VPNService` to intercept the communication between all clients and web services. It required that I use the UDP and TCP transport protocols and IPv4 and IPv6 Internet protocols to restore the information flow between the communicating devices. However, the protocols required that not only headers are added or removed, but that the application maintains different sessions for single fields in the headers. This increases the complexity of the design.

I had planned to evaluate the usefulness of the application after building it. However, since the complexity increased and time resources for finishing and evaluating the application were limited, I decided to use an alternative method to evaluate its usefulness.

Thus, if it is not useful I should not spend a considerable amount of time implementing it.

I extracted the HTTP(S) connections from a dataset of Android connections, and recreated them to evaluate if the application would help or break them. I conclude that the system would enforce HTTPS for 48% of the web services that did not use it, and fix flawed HTTPS for 86% of the web services that used it. In my opinion, the results of the evaluation phase show that the suggested application would be useful. Since this work is the first part of an MInf project, I have decided to continue implementing the application during the second part.

7.1 Further Work

During the second part of the project I will finish the planned implementation. That is, I still need to finish restoring the information flow by supporting all the protocols – TCP, UDP, IPv4, and IPv6. I will then add support for the HTTP and HTTPS protocols in order to achieve the two goals from above. As a result, I would have built an application that intercepts all clients' communication and forwards it to the corresponding web services, but also enforces HTTPS and checks HTTPS certificates for validity. In the end, I will build a user interface that allows the users to interact with the application and to add exceptions to the rules that the application uses for upgrading HTTP and validating HTTPS.

Bibliography

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 289–305. IEEE, 2016.

[2] Devdatta Akhawe, Bernhard Amann, Matthias Vallentin, and Robin Sommer. Here's my cert, so trust me, maybe?: understanding tls errors on the web. In *Proceedings of the 22nd international conference on World Wide Web*, pages 59–70. ACM, 2013.

[3] Tim Berners-Lee. The worldwideweb browser. `https://www.w3.org/People/Berners-Lee/WorldWideWeb.html`. (Accessed: 02 February 2018).

[4] Google Security Blog. Moving towards a more secure web. `https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html`. (Accessed: 02 February 2018).

[5] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick D. McDaniel, and A. Selcuk Uluagac. Sensitive information tracking in commodity iot. *CoRR*, abs/1802.08307, 2018.

[6] Tom Chothia, Flavio D Garcia, Chris Heppel, and Chris McMahon Stone. Why banker bob (still) cant get tls right: A security analysis of tls in leading uk banking apps. In *International Conference on Financial Cryptography and Data Security*, pages 579–597. Springer, 2017.

[7] Cisco. Bandwidth, packets per second, and other network performance metrics. `https://www.cisco.com/c/en/us/about/security-center/network-performance-metrics.html`. (Accessed: 5 April 2018).

[8] Global Net Neutrality Coalition. Net neutrality. `https://www.thisisnetneutrality.org/`. (Accessed: 02 February 2018).

[9] Andrew Whalley Devon OBrien, Ryan Sleevi. Chromes plan to distrust symantec certificates. `https://security.googleblog.com/2017/09/chromes-plan-to-distrust-symantec.html`. (Accessed: 24 August 2017).

[10] Diaa Salama Abd Elminaam, Hatem Mohamed Abdual-Kader, and Mohiy Mohamed Hadhoud. Evaluating the performance of symmetric encryption algorithms. *IJ Network Security*, 10(3):216–222, 2010.

[11] eMarketer. Smartphone apps crushing mobile web time. https://www.emarketer.com/Article/Smartphone-Apps-Crushing-Mobile-Web-Time/1014498. (Accessed: 02 February 2018).

[12] Let's Encrypt. Let's encrypt stats. https://letsencrypt.org/stats/#percent-pageloads. (Accessed: 02 February 2018).

[13] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[14] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking ssl development in an appified world. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 49–60. ACM, 2013.

[15] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol – http/1.1. https://tools.ietf.org/html/rfc2616. (Accessed: 2 March 2018).

[16] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[17] Google. Transparency report. https://transparencyreport.google.com/. (Accessed: 02 February 2018).

[18] Federal government of the United States. The https-only standard. https://https.cio.gov/. (Accessed: 02 February 2018).

[19] Andy Greenberg. Google's chrome hackers are about to upend your idea of web. https://www.wired.com/2016/11/googles-chrome-hackers-flip-webs-security-model/. (Accessed: 02 February 2018).

[20] Ilya Grigorik. *High Performance Browser Networking: What every web developer should know about networking and web performance*. " O'Reilly Media, Inc.", 2013.

[21] HackerRank. 2018 developer skills report. https://research.hackerrank.com/developer-skills/2018/. (Accessed: 02 February 2018).

[22] Bruce Hanington and Bella Martin. *Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers, 2012.

[23] Chris Havergal. Teach all computing students about cybersecurity, universities told. https://www.timeshighereducation.com/news/teach-all-computing-students-about-cybersecurity-universities-told. (Accessed: 5 April 2018).

[24] Jacob Hoffman-Andrews. Verizon injecting perma-cookies to track mobile customers, bypassing privacy controls. https://www.eff.org/deeplinks/2014/11/verizon-x-uidh. (Accessed: 02 February 2018).

[25] Twitter Inc. Securing your twitter experience with https. https://blog.twitter.com/official/en_us/a/2012/securing-your-twitter-experience-with-https.html. (Accessed: 02 February 2018).

[26] University of Southern California Information Sciences Institute. Internet protocol. https://tools.ietf.org/html/rfc791. (Accessed: 12 January 2018).

[27] Jim Manico Kevin Wall Ricardo Iramar Jeffrey Walton, John Steven. Certificate and public key pinning. https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning. (Accessed: 24 August 2017).

[28] David Kravets. Comcast wi-fi serving self-promotional ads via javascript injection. https://arstechnica.com/tech-policy/2014/09/why-comcasts-javascript-ad-injections-threaten-security-net-neutrality/. (Accessed: 02 February 2018).

[29] Google LLC. Connecting to the network. https://developer.android.com/training/basics/network-ops/connecting.html. (Accessed: 30 August 2017).

[30] Kristijan Lucic. Over 27.44% users root their phone(s) in order to remove built-in apps, are you one of them? https://www.androidheadlines.com/2014/11/50-users-root-phones-order-remove-built-apps-one.html. (Accessed: 5 April 2018).

[31] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah McKune, Arn Rey, John Scott-Railton, Ronald Deibert, and Vern Paxson. Chinas great cannon. *Citizen Lab*, 10, 2015.

[32] Kieren McCarthy. Pai, pai, mr american spy: Fcc supremo rips up privacy protections for broadband punters. https://www.theregister.co.uk/2017/02/24/fcc_kills_off_isp_customer_privacy_rules/. (Accessed: 02 February 2018).

[33] Paul Mockapetris. Domain names – concepts and facilities. https://tools.ietf.org/html/rfc1034. (Accessed: 2 March 2018).

[34] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The cost of the s in https. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 133–140. ACM, 2014.

[35] Lily Hay Newman. How malware keeps sneaking past google play's defences. https://www.wired.com/story/google-play-store-malware/. (Accessed: 31 March 2018).

[36] Oracle. Networking basics. https://docs.oracle.com/javase/tutorial/networking/overview/networking.html. (Accessed: 3 September 2017).

[37] Stack Overflow. How do i get an android process running with the cap_net_admin capability. https://stackoverflow.com/questions/13042117/how-do-i-get-an-android-process-running-with-the-cap-net-admin-capability. (Accessed: 20 January 2018).

[38] William D Perreault. Controlling order-effect bias. *The Public Opinion Quarterly*, 39(4):544–551, 1975.

[39] Jon Postel. User datagram protocol. https://www.ietf.org/rfc/rfc768. (Accessed: 14 January 2018).

[40] E. Rescorla. Http over tls. https://tools.ietf.org/html/rfc2818. (Accessed: 3 March 2018).

[41] Eric Rescorla. Http over tls. https://tools.ietf.org/html/rfc2818/. (Accessed: 02 February 2018).

[42] T. Polk S. Turner. Prohibiting secure sockets layer (ssl) version 2.0. https://tools.ietf.org/html/rfc6176. (Accessed: 10 September 2017).

[43] Sam Schillace. Default https access for gmail. https://gmail.googleblog.com/2010/01/default-https-access-for-gmail.html. (Accessed: 02 February 2018).

[44] Aftab Siddiqui. Rfc 8200 ipv6 has been standardized. https://www.internetsociety.org/blog/2017/07/rfc-8200-ipv6-has-been-standardized/. (Accessed: 15 February 2018).

[45] William Stallings. *Cryptography and network security: principles and practice*. Pearson Education India, 2003.

[46] Chris McMahon Stone, Tom Chothia, and Flavio D Garcia. Spinner: Semi-automatic detection of pinning without hostname verification. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 176–188. ACM, 2017.

Appendix A

Semi-Structured Interview Script

Hello, my name is Vesko. I am a fourth-year student and today I will be asking you questions for my honours project. My project is about encryption in mobile apps and helping app developers use it.

I will be taking notes while we speak. I will use the information you give me for my project, but I will not use your name or any other identifying information. Please remember that your goal today is to tell me about encryption in mobile apps. There are no wrong answers and you have the right to stop at any time. Do you have any questions?

Do you agree?

Thank you for agreeing, let's start with the first question.

- Do you know about any Android encryption flaws?
- How about encryption flaws in other mobile platforms?
- Have you ever ensured that the connections that you are making in an app are encrypted?
- (Ask if participant answered no to the last question.) How about in any app?
- (Ask if participant answered yes to either of the last two questions.)
  - What issues did you encounter when implementing encryption?
  - What could be different that would help you to not have these issues?
- (Else ask.)
  - Why did you choose not to implement encryption in the apps you have created?
  - What would need to be different for you to have implemented encryption?
- Have you ever connected to a URL in your software?
- Which one of the suggested solutions would help you most to use encryption libraries? (Present solutions in a shuffled order.)
  - An Android library with safe defaults and minimal setup.
  - A guide that explains what encryption is, how to implement it in Android apps, and how to test your implementation.
  - Some users assume that they have implemented encryption correctly because the information they see looks like gibberish. However, this is not always correct - there is safe gibberish and unsafe gibberish. This option is a test tool for Android that decrypts unsafe gibberish and shows it to you.
  - Many encryption solutions on StackOverflow are unsafe. This option is posting a corrected solution in the same thread as the unsafe solution and explaining why the other solution is unsafe.
- Where do you imagine a guide to be hosted and what format would it have?