Imperial College London

Department of Computing

Encryption is Hard: Android's TLS Misadventures

by

Oliver Grubin

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science of Imperial College London

September 2015

Abstract

The security and confidentiality of user transmitted data on the Internet is at a dangerous stage. More and more, users are under attack from those who would use their data for financial gain or to eavesdrop on their communications. Transport Layer Security (TLS) is the mechanism that has been developed to secure user privacy across the open Internet. After significant vulnerabilities were identified recent years, desktop browsers such as Mozilla Firefox and Google Chrome have become hardened against attacks and many implementation errors have been fixed. The younger mobile ecosystem is not so secure. Users connecting to the Internet on their mobile phones using one of the myriad of apps available have no indication that their data is being transmitted securely, and no confidence that implementations are safe and bug-free.

This study demonstrates the failings in the implementation of TLS security on the Android operating system by examining the failings of apps developed by major corporations, analysing the Android ecosystem as a whole through the automated analysis of over 2500 apps, demonstrating the ease with which a man-in-the-middle attack can be conducted, and finally by developing an app entitled Sift, intended to protect and warn users when they are under attack.

Acknowledgements

Thank you to my supervisor, Jeremy Bradley, for his help and support with this project and to HG for careful proof reading and helpful suggestions (although any remaining errors are my own). Additional thanks should go to Netcraft for supplying the idea for Sift. Final thanks go to Mohamed ‘Hexene’ Naufal for providing a starting point for the app, and to StackOverflow and the open source community in general, without which coding would be made infinitely more difficult.

Contents

Chapter 1

Introduction

This study reveals the widespread misuse of the Transport Layer Security (TLS) cryptographic protocol in many major Android applications. It demonstrates how easy it is for users to have important information stolen and how simple it is for an attacker to orchestrate a widespread attack. Apps from across the spectrum are affected, including those that deal with finance, healthcare and online shopping.

TLS is designed to secure data transmitted over computer networks. It is used in all areas of the Internet, including web browsing, email and mobile applications. It has four main goals: cryptographic security (providing confidentiality, authenticity and integrity), interoperability, extensibility and efficiency. Unfortunately, despite its widespread use, the protocol and its implementations have suffered, and continue to suffer, from serious flaws and vulnerabilities. These are examined in detail, looking at issues that have affected the protocol itself as well specific implementations both in general and those particular to Android applications.

1.1 Objectives and Contributions

An app, entitled Sift, is produced, which protects users when an app is behaving insecurely, alerting them to the potential leaking of credentials and private data. In addition to the protection offered, attention will be drawn to this serious issue among the general public, spurring developers into making sure their apps are coded safely and securely, but safeguarding user data until they do.

Although many users don't pay any attention while browsing the internet on desktops and laptops, more technical users have learned to be cautious when connecting to the Internet from unfamiliar locations. In recent years there has been an increasing level of awareness of the ability for passive interceptors to recover plaintext HTTP traffic, but TLS offers users an amount of trust that their personal information (usernames, passwords, credit card numbers, etc.) will be transmitted without interception, and users have been trained to 'look for the padlock' (Figure 1.1) when making secure transactions.

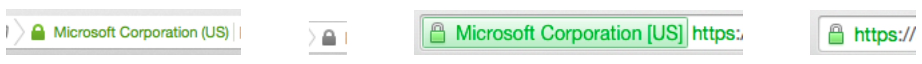
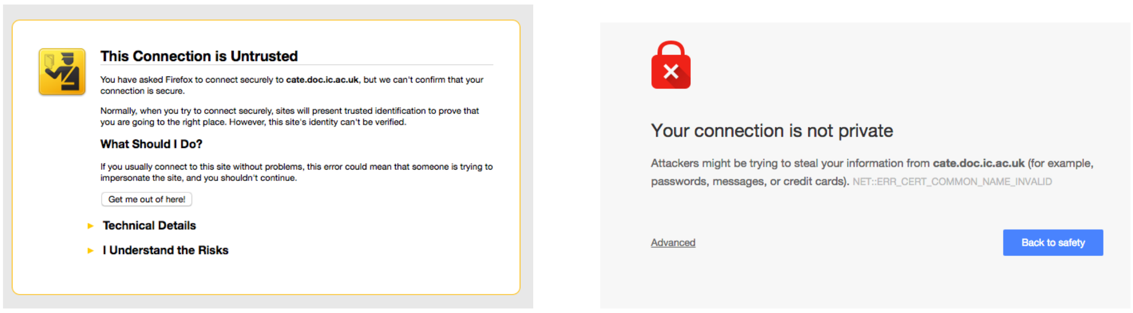


Figure 1.1: TLS security indications from Mozilla Firefox and Google Chrome

When faced with a potentially compromised TLS connection, desktop browsers such as Mozilla Firefox and Google Chrome have evolved to display warning screens to users (Figure 1.2) alerting them to the potential for data to be leaked to a malicious third party. However, users increasingly access secure information from their mobile phones, but mobile operating systems, and the applications that run on them, have been slower to alert users to problems in TLS connections. This leaves them vulnerable to providing a third-party man in the middle with information that can be used in fraud and identity theft.

This study will also demonstrate the ease with which a malicious wireless access point can be set up with cheap, off the shelf hardware, and used to harvest users' information.



This Connection is Untrusted

You have asked Firefox to connect securely to **cate.doc.ic.ac.uk**, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

What Should I Do?

If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

Get me out of here!

Technical Details

I Understand the Risks

Your connection is not private

Attackers might be trying to steal your information from **cate.doc.ic.ac.uk** (for example, passwords, messages, or credit cards). NET::ERR_CERT_COMMON_NAME_INVALID

Advanced

Back to safety

Figure 1.2: Mozilla Firefox and Google Chrome Certificate Warnings

An analysis of over 2500 apps has been performed, and several apps developed by major corporations, with downloads totalling in the tens of millions have been identified as leaking users' personal information, including important financial and medical details. Some of these apps allow users to login using their account credentials from Google or Facebook. A whole wealth of user data is available in these accounts, and this data is put at risk even when the apps produced by Google and Facebook themselves are secure. The developers have been informed of the vulnerabilities and in some cases fixed the issues very quickly, but users can be left vulnerable to apps where the bugs have gone unreported, or when using those which the developers have not yet fixed. Sift can protect users in these cases, and so takes the decision-making power away from developers and places it back in the hands of the users.

1.2 Report Outline

The report begins with an in depth analysis of the TLS protocol and issues it and its various implementations have faced. It continues by looking specifically at issues with TLS in the Android operating system, exploring approaches already taken to analysis of vulnerable apps and examining specific causes of insecurity.

The next section describes development of an app, Sift, to protect users against these vulnerabilities.

This is followed by a section detailing the construction of testing infrastructure, to test both Sift and individual Android apps and to evaluate the Android ecosystem as a whole.

The results of this investigation are then examined and evaluated. The report concludes with recommendations for securing the Android API and with ideas for future work.

Chapter 2

Background

The TLS protocol began life as the Secure Socket Layer (SSL) and was developed by Netscape in 1994 [1]. SSL 1.0 was never publicly released. SSL 2.0 was included in Netscape Navigator 1.1 but it was quickly discovered to have some serious vulnerabilities. SSL 3.0 was released in 1995. Following that, there was an effort to standardize the protocol, which included changing its name to Transport Layer Security (which was mostly a political push by Microsoft) [2]. TLS 1.0 was released in 1999 [3]. This remained in use for several years, even as TLS 1.1 and 1.2 were released in 2006 and 2008 respectively [4, 5]. However, following the release of TLS 1.2 and a greater awareness of the need for strong cryptography over the Internet, there was a concerted push for all clients and servers to move to TLS 1.2. SSL 3.0 was finally deprecated in June 2015 [6]. This section will give a brief overview of TLS 1.2 (with reference to older versions of the protocol where relevant). This information mostly comes from the TLS 1.2 Request for Comments (RFC) 5246 [5], and the excellent summary by Ristić in *Bulletproof SSL and TLS* [7], with cryptographic background from Schneier's classic *Applied Cryptography* [8].

It will be followed with a brief overview of several major vulnerabilities and issues that have been discovered in the protocol and its implementations, to which a significant proportion of clients and servers remain vulnerable.

2.1 The TLS Protocol

TLS is implemented with a high level Record Protocol, which encapsulates a variety of lower level protocols, specifically four core subprotocols: handshake protocol, change cipher spec protocol, application data protocol and alert protocol. It is designed to be extensible, easily allowing other subprotocols to be added.

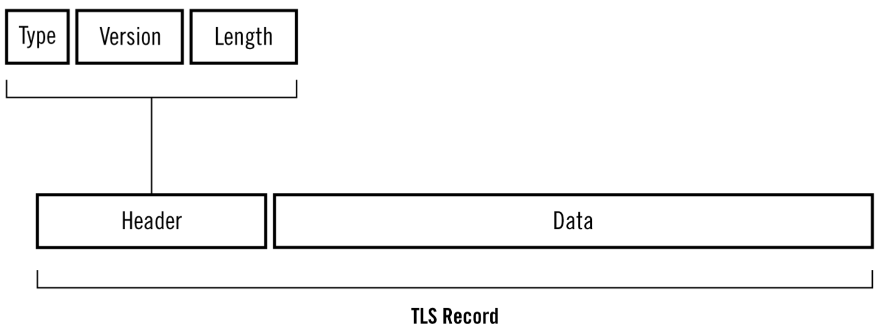


Figure 2.1: Struture of a TLS record packet. Source: [7, p24]

Each TLS record has a header followed by message data (Figure 2.1). The record layer handles message transport (splitting large buffers into smaller chunks), encryption, integrity validation and compression, although this is no longer used due to the CRIME vulnerability (detailed in Section 2.4.3).

2.1.1 Handshake Protocol

The handshake protocol is the most complex part of the protocol. It is conducted in plain text and there are several variations. The most common handshake when using the Internet is a full handshake with server authentication, but without client authentication. When a client is recognised by a server, an abbreviated handshake can be performed, allowing a session to be resumed with fewer messages.

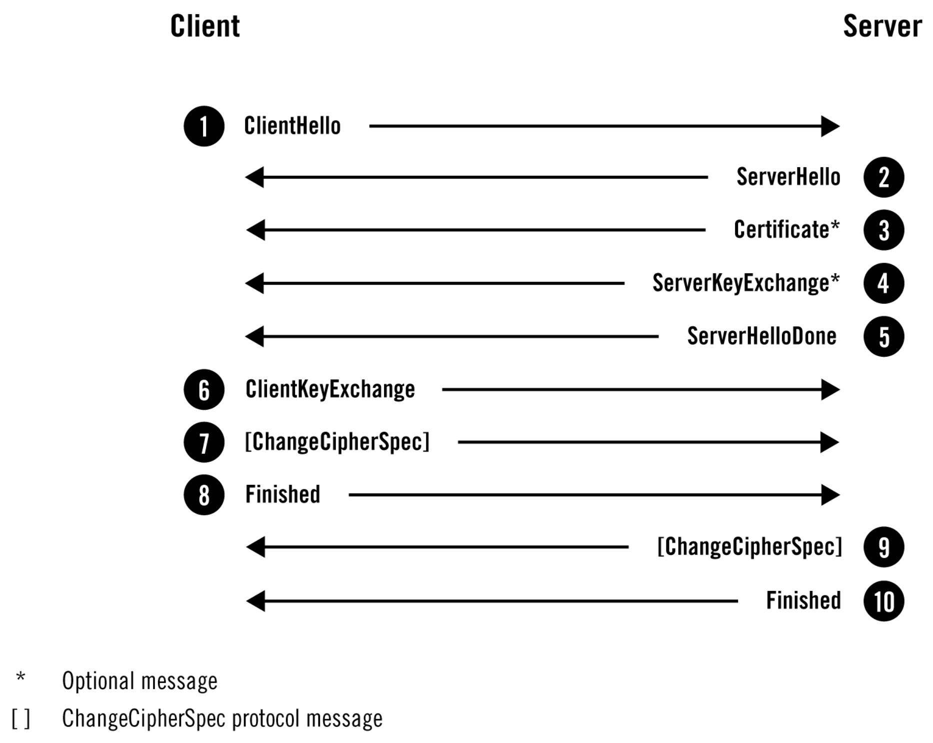


Figure 2.2: The TLS handshake. Source: [7, p27]

Figure 2.2 shows the messages exchanged for a full TLS handshake. There are four main activities that must be accomplished in the handshake:

1. Exchange capabilities and agree on desired connection parameters.
2. Validate the presented certificate(s) or authenticate using other means.
3. Agree on a shared master secret that will be used to protect the session.
4. Verify that the handshake messages haven't been modified by a third party [7, p26].

The handshake can be easily observed using a tool such as Wireshark[1] to observe network traffic. The purpose of each message becomes more clear with a specific example. Here the client is host 129.31.232.71 which is negotiating a connection to `https://cpp.doc.ic.ac.uk`, host 146.169.13.85.

2.1.1.1 Client Hello

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

```
SSL Record Layer: Handshake Protocol: Client Hello
        Content Type: Handshake (22)
        Version: TLS 1.0 (0x0301)
        Length: 231
        Handshake Protocol: Client Hello
            Handshake Type: Client Hello (1)
            Length: 227
            Version: TLS 1.2 (0x0303)
            Random
                GMT Unix Time: Feb 16, 1997 00:02:25.000000000 GMT
                Random Bytes:
                   f74af2d4fe86fb3364ede90ec9fe994d94a6b449c026d0d4...
            Session ID Length: 32
            Session ID: a09a7312d65259738801337c0229f39aca51d9457158210d
               ...
            Cipher Suites Length: 22
            Cipher Suites (11 suites)
                Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0
                   xc02b)
                Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0
                   xc02f)
                Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a
                   )
                Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009
                   )
                Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
                Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
                Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
                Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
                Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
                Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
                Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
            Compression Methods Length: 1
            Compression Methods (1 method)
                Compression Method: null (0)
            Extensions Length: 132
            Extension: server_name
                Type: server_name (0x0000)
                Length: 21
                Server Name Indication extension
                    Server Name list length: 19
                    Server Name Type: host_name (0)
                    Server Name length: 16
                    Server Name: cpp.doc.ic.ac.uk
            Extension: renegotiation_info
                Type: renegotiation_info (0xff01)
                Length: 1
                Renegotiation Info extension
                    Renegotiation info extension length: 0
            Extension: elliptic_curves
                Type: elliptic_curves (0x000a)
                Length: 8
                Elliptic Curves Length: 6
                Elliptic curves (3 curves)
                    Elliptic curve: secp256r1 (0x0017)
                    Elliptic curve: secp384r1 (0x0018)
                    Elliptic curve: secp521r1 (0x0019)
            Extension: ec_point_formats
                Type: ec_point_formats (0x000b)
                Length: 2
        EC point formats Length: 1
        Elliptic curves point formats (1)
            EC point format: uncompressed (0)
    Extension: SessionTicket TLS
        Type: SessionTicket TLS (0x0023)
        Length: 0
        Data (0 bytes)
    Extension: next_protocol_negotiation
        Type: next_protocol_negotiation (0x3374)
        Length: 0
    Extension: Application Layer Protocol Negotiation
        Type: Application Layer Protocol Negotiation (0x0010)
        Length: 41
        ALPN Extension Length: 39
        ALPN Protocol
            ALPN string length: 5
            ALPN Next Protocol: h2-16
            ALPN string length: 5
            ALPN Next Protocol: h2-15
            ALPN string length: 5
            ALPN Next Protocol: h2-14
            ALPN string length: 2
            ALPN Next Protocol: h2
            ALPN string length: 8
            ALPN Next Protocol: spdy/3.1
            ALPN string length: 8
            ALPN Next Protocol: http/1.1
    Extension: status_request
        Type: status_request (0x0005)
        Length: 5
        Certificate Status Type: OCSP (1)
        Responder ID list Length: 0
        Request Extensions Length: 0
    Extension: signature_algorithms
        Type: signature_algorithms (0x000d)
        Length: 18
        Signature Hash Algorithms Length: 16
        Signature Hash Algorithms (8 algorithms)
            Signature Hash Algorithm: 0x0401
                Signature Hash Algorithm Hash: SHA256 (4)
                Signature Hash Algorithm Signature: RSA (1)
            Signature Hash Algorithm: 0x0501
                Signature Hash Algorithm Hash: SHA384 (5)
                Signature Hash Algorithm Signature: RSA (1)
            Signature Hash Algorithm: 0x0201
                Signature Hash Algorithm Hash: SHA1 (2)
                Signature Hash Algorithm Signature: RSA (1)
            Signature Hash Algorithm: 0x0403
                Signature Hash Algorithm Hash: SHA256 (4)
                Signature Hash Algorithm Signature: ECDSA (3)
            Signature Hash Algorithm: 0x0503
                Signature Hash Algorithm Hash: SHA384 (5)
                Signature Hash Algorithm Signature: ECDSA (3)
            Signature Hash Algorithm: 0x0203
                Signature Hash Algorithm Hash: SHA1 (2)
                Signature Hash Algorithm Signature: ECDSA (3)
            Signature Hash Algorithm: 0x0402
                Signature Hash Algorithm Hash: SHA256 (4)
                Signature Hash Algorithm Signature: DSA (2)
            Signature Hash Algorithm: 0x0202
                Signature Hash Algorithm Hash: SHA1 (2)
```

```
                    Signature Hash Algorithm Signature: DSA (2)
```
Listing 2.1: Client Hello

Listing 2.1 is a full listing of the `Client Hello` protocol message, in which the client initiates a handshake and informs the server of its capabilities. This message can also be sent when a client wishes to renegotiate a connection, or when a server has requested a renegotiation.

Some of the key elements in this message are the protocol version, indicating the highest version the client supports, here listed as TLS 1.2. This is followed by the `random` field, of which 28 bytes are randomly generated with the following four bytes containing information from the client's clock. In this session, it is clear that the client is not sending an accurate time, listing Feb 16, 1997: an accurate time is not required in the protocol, and in fact, because of fears that the clock could be used to fingerprint clients [9], this is now a random time. This is a common feature when examining the TLS protocol: elements which were once thought to be necessary, or crucial are realised to contain flaws, and so remain in label only, with random data in their place.

The `session ID` field can be used to resume an existing session. The `cipher suite` block indicates the ciphers supported by the client in order of preference. The `compression` field indicates the supported compression methods. As mentioned above, this is no longer used, and so the method is listed as `null`. Finally, the `extensions` block is used for arbitrary extensions to the protocol. Some important extensions are discussed in Section 2.1.7.

2.1.1.2 Server Hello

```
Secure Sockets Layer
    TLSv1.2 Record Layer: Handshake Protocol: Server Hello
        Content Type: Handshake (22)
        Version: TLS 1.2 (0x0303)
        Length: 57
        Handshake Protocol: Server Hello
            Handshake Type: Server Hello (2)
            Length: 53
            Version: TLS 1.2 (0x0303)
            Random
                GMT Unix Time: May 29, 2015 17:03:50.000000000 BST
                Random Bytes: 1
                   c5842d1b3d7c0d2f46df7a9c1af422d9f74be76b6f44519...
            Session ID Length: 0
            Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
            Compression Method: null (0)
            Extensions Length: 13
            Extension: server_name
                Type: server_name (0x0000)
                Length: 0
            Extension: renegotiation_info
                Type: renegotiation_info (0xff01)
                Length: 1
                Renegotiation Info extension
                    Renegotiation info extension length: 0
            Extension: SessionTicket TLS
                Type: SessionTicket TLS (0x0023)
                Length: 0
                Data (0 bytes)
```

The Server Hello message is similar to Client Hello and allows the server to communicate back to the the client the elements of the protocol it supports; thus there is only one option per field. The server can also offer a version that the client did not list as its best version; the client can accept or reject the offer.

2.1.1.3 Certificate

```
Secure Sockets Layer
    TLSv1.2 Record Layer: Handshake Protocol: Certificate
        Content Type: Handshake (22)
        Version: TLS 1.2 (0x0303)
        Length: 2691
        Handshake Protocol: Certificate
            Handshake Type: Certificate (11)
            Length: 2687
            Certificates Length: 2684
            Certificates (2684 bytes)
                Certificate Length: 1318
                Certificate: 308205223082040
                   aa00302010202141dc339fcc65bbcbb58... (id-at-commonName=
                   cpp.doc.ic.ac.uk,id-at-organizationalUnitName=Computing
                    Department,id-at-organizationName=Imperial College of
                   Science, Technology ,id-at-localityName=LONDON
                    signedCertificate
                        version: v3 (2)
                        serialNumber : 0
                           x1dc339fcc65bbcbb58ad3e971939114870469c0f
                        signature (sha256WithRSAEncryption)
                            Algorithm Id: 1.2.840.113549.1.1.11 (
                               sha256WithRSAEncryption)
    ....
```

Listing 2.2: Certificate

The optional certificate message contains the server's X.509 certificate. It must match the offered cipher suite.

2.1.1.4 Server Key Exchange

```
Secure Sockets Layer
    TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
        Content Type: Handshake (22)
        Version: TLS 1.2 (0x0303)
        Length: 527
        Handshake Protocol: Server Key Exchange
            Handshake Type: Server Key Exchange (12)
            Length: 523
            Diffie-Hellman Server Params
                p Length: 128
                p: d67de440cbbbdc1936d693d34afd0ad50c84d239a45f520b...
                g Length: 1
                g: 02
                Pubkey Length: 128
                Pubkey: 3dc99e8e511da11023061d0285674ea4065fcefdaf1607cb
                   ...
                Signature Hash Algorithm: 0x0401
                    Signature Hash Algorithm Hash: SHA256 (4)
                    Signature Hash Algorithm Signature: RSA (1)
                Signature Length: 256
                Signature: 5
                   ea3e0e62fbde554fb005daab0d9fa11be8f57dd378ef757...
```

This is another optional message that contains any additional data that may be needed for the key exchange.

2.1.1.5 Server Hello Done

```
Secure Sockets Layer
    TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
        Content Type: Handshake (22)
        Version: TLS 1.2 (0x0303)
        Length: 4
        Handshake Protocol: Server Hello Done
            Handshake Type: Server Hello Done (14)
            Length: 0
```

This indicates that the server has sent all the necessary handshake messages and is waiting for the client.

2.1.1.6 Client Key Exchange

```
Secure Sockets Layer
    TLSv1.2 Record Layer: Handshake Protocol: Client Key Exchange
        Content Type: Handshake (22)
        Version: TLS 1.2 (0x0303)
        Length: 134
        Handshake Protocol: Client Key Exchange
            Handshake Type: Client Key Exchange (16)
            Length: 130
            Diffie-Hellman Client Params
                Pubkey Length: 128
                Pubkey: 3b80967adcfbbae616582a841c20d3ce8049ceda2151b6f5
                   ...
```

Similar to `Server Key Exchange`, this carries the required client data for the key exchange. This is a mandatory message.

2.1.1.7 Change Cipher Spec

```
TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
    Content Type: Change Cipher Spec (20)
    Version: TLS 1.2 (0x0303)
    Length: 1
    Change Cipher Spec Message
```

Both client and server send this message when there is enough information to switch to encryption. All messages following this will be encrypted. Change Cipher Spec is actually a separate subprotocol to the Handshake Protocol, which has been a cause of issues [7, p31]: it is supposed to be the penultimate message in the handshake, but implementations allowed it at other points in the sequence [10, p4].

2.1.1.8 Finished

```
TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 64
    Handshake Protocol: Encrypted Handshake Message
```

This message indicates that the handshake is complete and is encrypted. It contains the `verify data` field which is a hash of all handshake messages combined with the new master secret.

2.1.1.9 Session Resumption

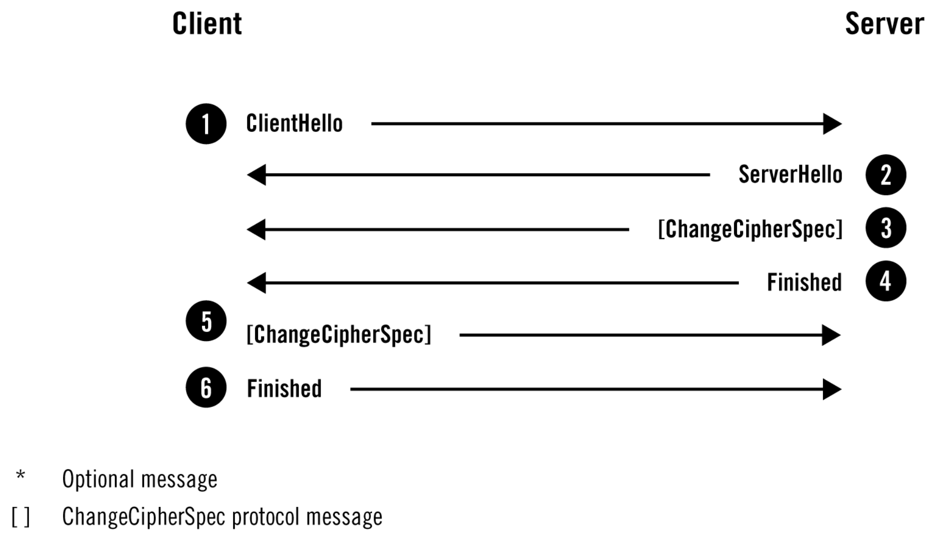


Figure 2.3: TLS session resumption. Source: [7, p35]

As shown in Figure 2.3, if a connection has previously been established between the server and client, an abbreviated handshake can be performed, requiring only one network round trip. The `session ID` field is used, and a new set of keys is generated using the already generated master secret. The server can either cache the previous session, or the client can use a session ticket, defined in RFCs 4507 [11] and 5077 [12].

2.1.1.10 Renegotiation

TLS supports renegotiation to generate new session keys or switch cipher suites. If a client wants to initiate a renegotiation it simply needs to send another `Client Hello` message. If a server wants to renegotiate the connection it needs to send `Hello Request`. The initial renegotiation protocol has been found to be insecure and is no longer supported: the `renegotiation_info` extension must be used instead (see Section 2.4.1).

2.1.2 Key Exchange

In the key exchange, the client and server agree on a premaster secret which is then used to generate the master secret which secures the session. A variety of algorithms can be used for this, but the most common are RSA, Ephemeral Diffie-Hellman (DHE_RSA) which provides forward secrecy and Ephemeral Elliptic Curve Diffie-Hellman (ECDHE_RSA and ECDHE_ECDSA) which is based on elliptic curve cryptography.

The `Server Key Exchange` message allows the server to select an algorithm if it wishes. In the example message, the server selected Diffie-Hellman.

RSA key exchange is very simple: the client generates a premaster secret, encrypts it with the server's public key and sends it to the server. While straightforward, and easy to implement, this suffers from a lack of forward secrecy: every message encrypted with the server's public key can be decrypted with the server's private key, at any point in time. If an adversary stores all messages from the client and at some point in the future discovers the server's private key, they will be able to decrypt all messages.

Diffie-Hellman avoids this problem, as the client and server each generate their own parameters, meaning a different encryption key is used for each session. When ephemeral Diffie-Hellman is used, no parameters are stored and reused between sessions, providing forward secrecy. The server's parameters are encrypted with its private key, allowing the client to decrypt them with the server's public key and verify that they were issued by the server.

Elliptic Curve Diffie-Hellman uses the same principles as regular Diffie-Hellman, but uses elliptic curves (chosen by the server) as the 'hard' problem, rather than modular arithmetic.

The premaster secret is eventually used to create the master secret using a Pseudorandom Function (PRF). This takes a the premaster secret, a seed (taken from both the client and server) and a label and generates the master secret. The master secret can then be further processed with the PRF to generate any required keys, specifically MAC keys, encryption keys and IVs.

2.1.3 Encryption

When it comes to actually encrypting the data, TLS supports many ciphers, which break down into three types of encryption: stream, block and authenticated.

2.1.3.1 Stream Encryption

In stream encryption, the plaintext is XORed with a keystream byte by byte to produce a ciphertext. In TLS, a message authentication code (MAC) is calculated of the sequence number (to avoid replay attacks), header (to protect the unencrypted data) and plaintext, combined with the plaintext and encrypted to form the ciphertext. The most common stream encryption algorithm is RC4, which is now considered insecure (see Section 2.4.6), and so this method is rarely used.

2.1.3.2 Block Encryption

In block encryption, the plaintext is encrypted a block at a time. In modern algorithms a block is usually 16 bytes. If a block is smaller than 16 bytes then padding must be added to make it the correct size. This has been a cause of weakness in TLS (for example the POODLE exploit, described in Section 2.4.5). A second issue is that block ciphers are deterministic: they produce the same output for the same input. This is a major issue for TLS because an attacker can attempt to encrypt a block of plaintext any number of times. If the ciphertext matches some captured ciphertext, she now has the encryption key and can decrypt the entire session. Because HTTP has predictable messages, attempting to guess the plaintext for a specific block becomes a lot easier. Electronic Codebook Mode (ECB) works in this way. To counter this, Cipher Block Chaining (CBC) is introduced, which uses an Initialization Vector (IV) to produce a different encrypted block each time.

Generation of IVs has also proved to be an issue in TLS: in TLS 1.0 the previous encrypted block was used as the IV, but that was found to be insecure in 2011 (see Section 2.4.2) and so explicit IVs are now included in the record. The most popular Block Encryption algorithm is AES and it is heavily used in TLS.

2.1.3.3 Authenticated Encryption

Authenticated encryption is currently the most favoured encryption mode in TLS [7, p44]. It provides encryption and integrity validation in the same algorithm. It can be seen as a combination of stream encryption and block encryption. Galois/Counter Mode (GCM) is the mode of operation used in TLS. In the example `Client Hello`, it can be seen as part of the most preferable cipher suite supported by the client.

2.1.4 Certificates

X.509 certificates are defined in RFCs 2818 [13] and 5280 [14]. They have become the primary method of verifying trust within the TLS protocol. A certificate essentially contains a public key, information about who it represents, and a signature from a trusted authority. Listing 2.2 shows some key elements of a certificate. A certificate firstly contains its version number; either 1, 2 or 3. Most certificates are v3, although some v1 root certificates still exist - these have been a cause of some issues within implementations of TLS in certain libraries. Certificates have a serial number, which was initially sequentially issued by the issuing authority, but now for security reasons are a random number. They also detail the algorithm used for the signature (`sha256WithRSAEncryption` in the example), the issuer, validity (both a start time and end time) and subject.

Certificates also contain extensions, allowing additions to be made to them without needing to define a whole new version. They can be marked `critical`, meaning an implementation must understand the extension or reject the certificate. There are many extensions defined. Some key ones are `Subject Alternative Name` which replaces the `Subject` field and allows a certificate to support multiple identities. `Name Constraints` can limit the domains a certificate is able to issue certificates for. `Basic Constraints` contains more limitations on the certificate, including `Path Length` which limits the depth of a certificate chain. There is also the `CRL Distribution Points` extension which details the location of the Certificate Revocation List to allow a client to determine if the certificate has been revoked. This has mostly been replaced by OCSP, the Online Certificate Status Protocol, detailed in `Authority Information Access` to allow faster, real-time verification of certificate validity.

2.1.4.1 Certificate Chains

In practice, certificate verification works based on a chain of trust: a relatively small subset of root certificates sign a larger group of intermediate certificates, which can then be used to validate individual leaf certificates for domains. Root certificates are extremely valuable as they are inherently trusted by browsers and operating systems. All signing must take place manually, and on an offline computer. Certification Authorities (CAs) are in control of these root certificates. The certificate chain for the example connection is shown in Listing 2.3.

```
1 cpp.doc.ic.ac.uk
Fingerprint: 23d2911ffec2b355a8c137afdad609abe9c33e37
RSA 2048 bits (e 65537) / SHA256withRSA

2 QuoVadis Global SSL ICA G2
Fingerprint: 6036330e1643a0cee19c8af780e0f3e8f59ca1a3
RSA 2048 bits (e 65537) / SHA256withRSA

3 QuoVadis Root CA 2   Self-signed
Fingerprint: ca3afbcf1240364b44b216208880483919937cf7
RSA 4096 bits (e 65537) / SHA1withRSA
```

Listing 2.3: A Certificate Chain

2.1.4.2 Certificate Pinning

A relatively new innovation is the concept of certificate pinning. In this implementation, a browser or application ‘pins’ a certificate or CA, meaning that any other certificate will be rejected. This is designed to get around the fact that any CA can issue a certificate for any domain. By enforcing white-listing on the client side, a developer can be more certain that any certificate used will be valid and correct.

2.1.5 Application Data Protocol

There is little to say about the Application Data Protocol. It carries the encrypted data that the client and server actually want to send. The record layer manages fragmentation, encryption and packaging of this data.

2.1.6 Alert Protocol

The alert protocol is used for transmitting short status messages, usually errors, to the other party. They contain a level and a description, with no ability to specify an arbitrary error message. If a `fatal` alert is received, the connection is terminated and the session is destroyed. A `warning` leaves the client the discretion as to whether to continue the session.

One alert that is not an error is the `close notify` alert. This is used to close a TLS connection. This is needed to avoid truncation attacks, where a malicious third party closes the connection before it is intended to be complete.

2.1.7 Extensions

TLS supports extensions, which can be used to add functionality without changing the protocol itself. Some of these are defined in RFC 6066 [15]. As can be seen in the example packets, extensions can be specified in the `Extensions` block in `Client Hello` and `Server Hello`. There are too many extensions to discuss in detail here, but some important ones are `elliptic_curves` which indicates support for Elliptic Curve cryptography and `heartbeat` which allows for keep-alive functionality (and was the source of the bug in OpenSSL that was the cause of Heartbleed, discussed in Section 2.3.1). The `renegotiation_info` extension is used to indicate support for secure renegotiation, and `server_name` is used for virtual host support for certificates.

2.2 Certificate Vulnerabilities

Certificates are a key element in the TLS protocol, yet are arguably one of the weakest elements, irrespective of implementation issues. The problem of decentralized trust has not been solved; there is no way to determine whether an unknown third party is trustworthy without asking a trusted party. Trusted parties are identified in the TLS ecosystem as Certification Authorities (CAs): a client has a set of root certificates (Android 5.1.1 has 162 root certificates, for example). Any valid certificate that is signed by one of these CAs will be trusted and CAs can issue certificates for any domain without obtaining permission. A certificate is obtained by verifying ownership of a domain and with a monetary payment.

Immediately, one can see the weakness in this approach: if one of the root CAs is compromised, or otherwise acts in a manner detrimental to the client (for example issuing certificates in exchange for large payments, regardless of whether the requester controls that domain), it is both difficult to detect, and hard to deal with once detected. This is not a theoretical problem. In 2011 DigiNotar was compromised. This was detected when an Iranian Gmail user reported problems accessing his account [16]: Chrome's use of certificate pinning had detected the problem. DigiNotar had been fully compromised, and hundreds of fraudulent certificates had been issued, including for some well known companies [17].

Even without a compromised CA there have been many incidents where attackers have been able to obtain certificates they were not entitled to. Due to weaknesses with the MD5 algorithm [18], researchers were able to obtain a CA certificate valid for signing any other certificates. They used the Chosen-Prefix Collision Attack to trick RapidSSL into issuing this certificate. After this incident, CAs moved to using the SHA-1 algorithm to sign certificates. In 2014 however, Google announced that Chrome would start warning about certificates signed with SHA-1, and encourage a move to SHA-2, because of severe weaknesses in the SHA-1 algorithm [19].

In 2008, poor email verification resulted in Mike Zusman being able to obtain a certifiate for `login.live.com`, Microsoft's authentication hub [20]. He was able to register the email address `sslcertificates@live.com`, which Thawte's systems accepted as an administrator account. In 2015 Microsoft faced the same issue with their Finnish `live.fi` domain [21].

Other CAs have been found issuing weak keys (for example DigiCert in 2011 [22]) or to have bugs in their systems allowing certificates to be issued for any domain names (StartCom [23] and CertStar [24] in 2008). Clearly the mix of a commercial ecosystem with the need to verify millions of websites in a hostile environment is not going to guarantee security.

2.2.1 Validating Certificates

In addition to issues with CAs, verifying certificates as valid is not a trivial task. The RFCs describing X.509 certificates (2818 [13] and 5280 [14]) are long, complex and difficult to interpret. Brubaker et al. generated eight million certificates (which they called Frankencerts) and ran them against multiple SSL libraries [25]. They used the varying acceptance of these certificates as a verification oracle: if one implementation accepts a certificate as valid and another rejects it then they cannot both be adhering to the specification.

There have been many issues in clients failing to validate certificates correctly. Perhaps the most well known of these is Apple's 'goto fail' bug, in which one misplaced line of code led to easy hijacking of DHE and ECDHE connections [26].

In July 2015 a serious vulnerability in OpenSSL was announced [27] which would allow clients to treat invalid certificates as legitimate. It was only in the code for a few months, but as one of the most widely used tools for encrypting connections, this was still a serious issue.

There have also been several studies looking specifically at certificate verification within the Android ecosystem. While the effectiveness of warnings to users about the security of a connection has been examined by Egelman et al. [28] and Sunshine et al. [29], when running an Android application, there is not even a security indicator that one can ignore: 'developers currently have the power to define SSL policies for an app without those being transparent for the user' [30, p54]. Fahl et al. [31] conducted a static analysis of 13,500 Android applications, discovering that the majority did not provide adequate security to user data. This is explored in more detail in Section 3.

2.2.2 Certificate Revocation

There are also issues with revoking certificates [32]. The initial solution was to use Certificate Revocation Lists (CRLs), but they have become unwieldy due to their large size [7, p146]. Additionally, propagation time for revocation can be up to ten days, which is a serious problem if a certificate needs to be immediately revoked. The Online Certificate Service Protocol, OCSP allows real-time checking, but unfortunately most browsers are configured to ignore all failures. It's also possible for an active man-in-the-middle to block all OCSP requests easily, leaving the client unaware that a certificate has been revoked. Google's Chrome browser now uses CRLSets, which are proprietary.

2.2.3 Superfish

Vulnerabilities have also been introduced by trusted third-parties. In February 2015 it was discovered that Lenovo had been pre-installing software on its computers called Superfish [33]. This installed a trusted root certificate in the operating system, allowing the software to examine all encrypted connections and inject advertisements into the stream. By using the same root certificate for every installation, Superfish rendered all clients vulnerable to man-in-the-middle attacks using certificates signed by this certificate. This was made worse by the fact that the root certificate had been secured with the incredibly weak password 'komodia', allowing attackers to easily sign new certificates [34].

2.3 Implementation Issues

Implementing the complex TLS protocol is not an easy task. Subtle mistakes can be easily introduced and remain undiscovered for a long period of time. This section will examine some of the most significant implementation issues that have been discovered in TLS, including perhaps the most notorious, Heartbleed.

2.3.1 Heartbleed

Heartbleed [35] exploited a flaw in the OpenSSL implementation of the Heartbeat TLS extension. It was disclosed in April 2014. The flaw itself was very simple: OpenSSL failed to check the length field in the heartbeat message, allowing an attacker to send a small amount of data but pretend they had sent a lot, therefore receiving up to 64KB of memory back from the server. This could happen an unlimited amount of times. Crucial information is stored in memory, including private RSA keys, session tokens and passwords; with enough work all of these became visible to the attacker.

The issue was easily fixed, and remarkably many servers were patched very quickly. According to one study, one month later only 1.36% of sites were vulnerable [36]. Despite the easy fix, the bug had been in OpenSSL from version 1.0.1 and left unnoticed for many years.

2.3.2 FREAK

In the 1990s, the United States limited the export of cipher keys to 40 bits. Although this rule was relaxed in January 2000, many libraries still contain code for negotiating encryption with these weak keys. FREAK, which stands for Factoring RSA Export Keys [37], enables a man-in-the-middle attacker to force a server to supply a weak, 512-bit RSA key to the client by forcing the `Server Key Exchange` message, which is not expected to be sent. These keys are weak enough to be cracked in real-time by powerful computers. Even when using a regular computer, some libraries reuse the weak key for performance reasons, allowing the key to be cracked offline, and then used to break the security of all future sessions between the server and its clients.

2.3.3 Logjam

Logjam was announced in May 2015 [38]. Similarly to FREAK, it downgrades TLS connections to exploit export grade encryption and negotiate weak, 512-bit, Diffie-Hellman keys. In addition to that, it was observed that many servers used the same prime numbers for Diffie-Hellman key exchange, and the researchers posit that some of the most common 1024 bit primes have been broken by state-level agencies such as the NSA, breaking this level of encryption.

2.3.4 Downgrade Attacks

In a downgrade attack, an attacker tries to force the connection protocol to be downgraded to a more vulnerable one. The ability to downgrade was introduced because of the lack of interoperability between clients and servers that supported different protocol versions. While there have been attempts to block forced downgrades, most modern browsers can still be forced to downgrade their connection to at least TLS 1.0, although the `TLS_FALLBACK_SCSV` extension has been introduced to prevent protocol downgrade [39].

2.3.5 Opportunistic Encryption

In recent years there has been an effort to extend encryption to connections even when servers do not explicitly support it. HTTP Alternative Services [40] seems to be gaining acceptance as the accepted standard. This feature was introduced in Firefox 37 in April 2015, however it was quickly removed in version 37.0.1 [41] due to a critical bug that allowed malicious websites to bypass HTTPS security. This highlights the dangers of introducing new extensions to the protocol, even when they attempt to solve an existing problem.

2.4 Protocol Attacks

Weaknesses in the TLS protocol itself are another difficult issue. Unlike implementation flaws such as buffer overflows, and insecure bounds checking, which can be easily fixed, faults in the protocol itself are a lot harder to resolve quickly. Changes to the protocol can break interopability between clients and servers, and there are many clients which may never be updated. This section will look at some of the most serious protocol attacks so far.

2.4.1 Insecure Renegotiation

Insecure renegotiation was discovered in 2009 [42]. A man-in-the-middle attacker could sit between a client and server and act as a transparent proxy. It catches the `Client Hello` from the client, and holds it, sending its own `Client Hello` to the server. It will then submit an HTTP request to the server that will have a negative consequence on the client. It will then quickly let through the client's `Client Hello`, which the server will treat as a renegotiation attempt, process, and then send back to the client the results of the bad HTTP request. This technique was used by Anil Kurmos to retrieve encrypted data in 2009 [43].

2.4.2 BEAST

BEAST, or Browser Exploit Against SSL/TLS, was announced in the summer of 2011 [44]. It exploited an issue that had been noted many years before but not considered serious. The issue had even been fixed in TLS 1.1, but at the time many clients and servers still only supported TLS 1.0. It exploits a weakness in the Initialization Vector (IV), allowing an attacker to decrypt much of the traffic.

BEAST attacks the CBC mode of encryption (described in Section 2.1.3.2): by making the IVs predictable the encryption is effectively reduced to ECB, which is easy to break. The problem with TLS 1.0 was that it used a random IV to encrypt the first block, but then used subsequent encrypted blocks as IVs for the next block. This would not be an issue unless the attacker is able to see the encrypted blocks and thus discover the IV: of course, by watching the traffic on the wire, she can. This is resolved in TLS 1.1 and 1.2 which use per-record IVs. The practicalities of exploiting this issue are complex, involving crafting specific HTTP messages to enable easy guessing of the plaintext. BEAST is a client-side attack, and at the time of writing all modern browsers are no longer vulnerable, but BEAST remains an issue with older clients and servers.

2.4.3 Compression Attacks: CRIME, TIME and BREACH

As mentioned in Section 2.1.1, the TCP protocol supports compression. However a series of attacks have demonstrated that using compression is insecure in every case, and modern servers all disable compression. The issue comes because an attacker is able to submit arbitrary data for compression. Combining this with secret data, she is able to create a 'compression oracle' allowing the data to be revealed.

CRIME, Compression Ratio Info-leak Made Easy, was disclosed in 2012 [45] and uses Javascript malware to extract cookies in an active man-in-the-middle attack. TIME, Timing Info-leak Made Easy, advanced CRIME to no longer require an attacker to be on the same network as the victim [46]. It measures I/O differences using Javascript to detect timing differences in the server responses. It cleverly exploits the TCP slow-start algorithm to generate time differences large enough to be exploitable.

BREACH, Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext, followed CRIME and TIME in August 2013 [47]. It is similar to CRIME but focuses on the HTTP response compression. They were able to 'attack Outlook Web Access (OWA), [and] retrieve CSRF tokens with 95% reliability and often in under 30 seconds' [7, p213].

2.4.4 Lucky 13

Lucky 13 exploits the fact that padding (used in CBC mode) is not protected by the TLS protocol, allowing the attacker to modify the padding and observe how the server reacts [48]. 'Using JavaScript malware injected into a victims browser, the attack needs about 8,192 HTTP requests to discover one byte of plaintext' [7, p220]. This technique uses a 'padding oracle' to guess which combination of bytes decrypt to valid padding.

2.4.5 POODLE

POODLE, Padding Oracle On Downgraded Legacy Encryption, [49] is another attack that uses a padding oracle. It exploits a weakness in SSL 3 (and has since been shown to be effective with TLS 1.0 [50]) and allows an attacker to discover one byte of plaintext at a time.

2.4.6 Bar Mitzvah Attack

RC4 is a common stream encryption algorithm used with TLS. At one point it was recommended to be used in favour of CBC ciphers with TLS 1.0 to avoid the BEAST and POODLE attacks. However, RC4 has now been found to be completely broken. The Bar Mitzvah attack exploits this, allowing 'partial plaintext recovery attacks on SSL-protected data, when RC4 is the cipher of choice' [51]. RFC 7465 [52] now prevents the further use of RC4 in TLS and support will be removed from all major browsers starting in January and February 2016 [53].

2.4.7 Further RC4 Attacks

In March 2015 Garman et al. released a paper reducing the number of encryptions required to break a TLS connection from 2^{34} [54] to 2^{26} [55], although each encryption required a separate TLS connection, making the attack difficult to execute. In July 2015 the RC4NoMore attack [56] reduced this further and the researchers were able to break a session cookie in 75 hours, requiring 9×2^{27} requests.

2.4.8 Triple Handshake Attack

The triple handshake attack breaks secure renegotiation in TLS (which had been introduced to fix problems with the original implementation of renegotiation), allowing an attacker to learn the premaster secret and decrypt all traffic in a connection. It requires client authentication to be in use and allows an attacker to impersonate the client. The attacker directs a client to a malicious server, and then mirrors the random parameters to create a connection with the intended server. It then uses session resumption to resume the original connection, and then attempts to renegotiate the connection to take control of both connections. For more details, see Ristić [7, pp230-237].

2.5 Further Vulnerabilities

As should hopefully have become clear, these issues have been coming to light regularly. There are certainly further issues to be found. Even over the course of this project from May 2015 to August 2015, three or four major issues have been announced: the TLS protocol and its implementation is not fixed and complete. This emphasizes the importance of both careful development techniques, and the need for layered security, as the possibility for encryption to be compromised, possibly through no fault of one's own, is high.

2.6 State Actors

State intervention is also an issue. As the Snowden revelations have shown, organisations such as the NSA and GCHQ are not averse to interfering with global internet traffic. Programs such as that revealed by the Brazilian television show Fantastico [57] peform man-in-the-middle attacks against adversaries. A leaked document 'illustrates with a diagram how one of the agencies appears to have hacked into a target's Internet router and covertly redirected targeted Google traffic using a fake security certificate so it could intercept the information in unencrypted format' [58]. This, combined with the XKEYSCORE program (in which all available internet traffic is collected and stored for a number of days) [59] allow these agencies the ability to examine a large amount of internet traffic. Unfortunately, not much can be done to combat this. With the authority of government, they can demand access to root keys, allowing valid certificates to be generated and traffic to be decrypted with a user being none the wiser. Their unrivalled computing resources allow them to take advantage of any slip in server configuration, for example it has been speculated that they are able to break the 1024-bit primes used in Diffie-Hellman key exchange.

The NSA has also tried to attack elements of the TLS protocol as part of its Bullrun program. The Dual Elliptic Curve Deterministic Random Bit Generator (Dual EC DRBG) was adopted as a standard in 2005 and 2006. The Snowden revelations showed that this was a deliberately weak random number generator planted by the NSA [60]. When this is used as the random number generator in the TLS handshake, with knowledge of the backdoor a TLS connection becomes 65,000 times easier to break [61, 62]. Since the exposure of this program this random number generator is no longer used, but the potential remains for some other element of the protocol to contain a backdoor unknown to the majority of users.

2.7 The Human Element

Even if the protocol is secure, and all implementations are correct, security still relies on end users understanding any warnings and not overriding any security that is in place. There have been many studies examining adherence to certificate warnings, with one study showing a 70% click through rate (i.e. users ignoring the warnings they are given) on Google's Chrome browser [63]. For any security solution to be effective, a user must be informed as to the meaning and severity of any warnings and must not be overwhelmed by alerts when they do not perceive themselves to be doing anything that requires encryption.

A solution could be introduced that attempts to analyse a connection and only warn a user when the risk of interception and value of data loss is deemed to be severe. This would avoid the user growing accustomed to bypassing warnings and would perhaps lead to greater security of truly sensitive information. Any attempt at this would have to be carefully examined however, both in regards to classification of sensitive data and because of the tendency of users to reuse passwords in different contexts.

Chapter 3

TLS on Android

In the past few years there have been several studies examining TLS security in general [64, 10, 25, 65] and security on Android smartphones in particular [66]. Edgecombe developed a static analysis tool to analyse application use of certificates, as well as a tool to conduct man-in-the-middle attacks [67]. Fahl et al. examined SSL security and also developed a static analysis tool [30, 31], and Egele et al. examined the use of cryptographic libraries, finding some severe vulnerabilities [68].

Netcraft[1] examined the source code of several applications and discovered some serious issues in implementing application security, finding one particularly egregious example where the `verify` method of the `HostnameVerifier` had been overridden to always return true [69].

Georgiev et al. discovered flaws in widely used applications from major companies, including the banking app for Chase, one of the largest banks in the United States [64].

Fahl et al. suggest extending the Android API to enforce appropriate usage of cryptographic primitives [30], however at the time of writing this remains a suggestion and has not been integrated into the Android operating system. Even if this approach was integrated into the Android API, the issue of existing apps failing to provide security to users would still exist.

In their separate study of 11,748 Android applications, Egele et al. found that '10,327 programs - 88% in total - use cryptography inappropriately' [68]: clearly developers cannot be trusted to implement security in a way that guarantees to a user that their data is secure. In the same way that antivirus aids the user in discriminating malicious applications, and firewalls block unwanted network traffic, Fahl et al. [31, p59] suggest that a mechanism should be in place that shows the user in real time the security of their connection (verifying the validity of certificates and the strength of the TLS protocol), allowing them to decide whether to proceed with inputting their confidential information.

In September 2014, researchers at the CERT Division of the Software Engineering Institute of Carnegie Mellon University analysed over a million Android apps, finding over 23,000 that failed to properly secure TLS traffic [70]. Sam Bowne at City College San Francisco found his own set of vulnerable apps [71] and a separate company, AppBugs, that analyses apps installed on users' handsets found another set of apps that failed to protect users' information [72]. Since the development of apps moves at such a fast pace, and as old apps get updated, and new ones are released, maintaining a blacklist of insecure apps becomes very difficult.

Poorly coded apps can even have real-world implications. In August 2015, security researcher Samy Kamkar revealed vulnerabilities in the remote connection apps for General Motors, BMW, Mercedes-Benz and Chrysler, allowing an attacker to gain access to vehicles [73].

This is also a challenge for regulators. In March 2014, the Federal Trade Commission (FTC) in the United States settled with two companies, Fandango and Credit Karma, over their failure 'to properly implement SSL encryption' [74]. While this action is to be commended, the fact that only two companies have been fined despite the much larger number of apps with this vulnerability highlights the difficulty of relying on government regulators to protect unwitting users.

[1] `http://www.netcraft.com/`

3.1 Reasons for Insecurity

These apps are insecure not through any fault of the TLS protocol, but because of implementation issues. As mentioned earlier, when negotiating a TLS connection, the client is presented with a certificate by the server. The client must properly verify this certificate before proceeding with the handshake. Specifically, a chain of trust must be established, linking the presented certificate to a trusted root certificate in the trust store (Figure 3.1) and the hostname in the certificate must match the hostname the client is trying to connect to (Figure 3.2).

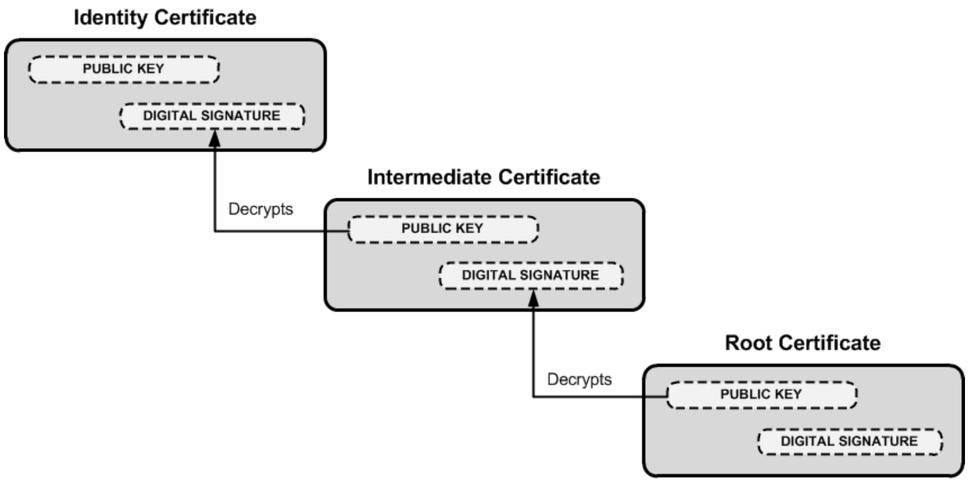
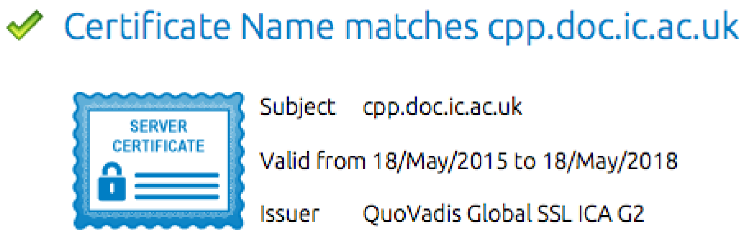


Figure 3.1: Source: `http://security.stackexchange.com/a/56393`


Certificate Name matches cpp.doc.ic.ac.uk

Subject cpp.doc.ic.ac.uk

Valid from 18/May/2015 to 18/May/2018

Issuer QuoVadis Global SSL ICA G2

Figure 3.2: Source: `https://www.digicert.com`

Android provides an Application Programming Interface (API) to do this, although with some caveats. Development of the Android operating system has moved extremely rapidly over the past few years: it is currently on version 5.1 (Lollipop). However, the ecosystem has been slower to update; as of August 3, 2015, 87.1% of devices are on version 4.4 (Jelly Bean) or below (Figure 3.3).

Google has tried to mitigate this by providing support libraries, allowing developers to use the most up to date API and still maintain compatibility with the oldest devices, but this is not a complete solution. As such, there are a number of APIs developers may use when attempting to create a secure connection to a server.

There are even two interfaces for establishing a connection, Apache HTTP Client, used by Google in Android Gingerbread 2.3 and below but as of 2011 no longer being actively developed, and HttpURLConnection, which is now the recommended API [75]. Apache HTTP Client was in fact deprecated in API version 22, but can still be used. Of course, if one provides developers with multiple options, there will be many that do not research the best option, and use whatever they first come across.

To initiate a TLS connection, Google's documentation [76] suggests code as simple as Listing 3.1, although if one wants to do anything more complicated, the process becomes more involved.

One may want to use a certificate signed by a root that is not in the default Android trust store, or even a self-signed certificate. One may want to establish a TLS connection with a protocol other

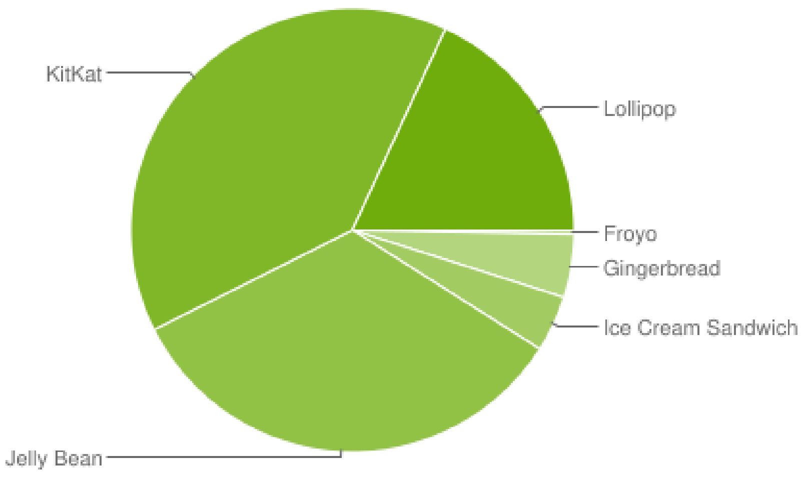


Figure 3.3: Android Operating System Fragmentation. Source: `https://developer.android.com/about/dashboards/index.html`

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

Listing 3.1: Source: [76]

than HTTP, or server issues may require workarounds in client code.

3.1.1 Custom TrustManagers

Establishing a custom trust store requires implementing a custom TrustManager. This puts certificate verification in a developer's hands, and unsurprisingly, many developers have made simple errors. Modern development relies heavily on searching the internet for already solved solutions to problems, with StackOverflow[2] a common source of information. Therefore, if a highly rated, but incorrect solution exists, many developers will end up blindly copying that into their code. Just such a situation exists here. When searching for a solution to their client not validating what seems to be a correct certificate, common answers on StackOverflow encourage developers to use code similar to Listing 3.2.

```java
    public MySSLSocketFactory(KeyStore truststore) throws
       NoSuchAlgorithmException, KeyManagementException,
       KeyStoreException, UnrecoverableKeyException {
        super(truststore);

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

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

            public X509Certificate[] getAcceptedIssuers() {
```

[2] `http://stackoverflow.com/`

```
                    return null;
                }
            };
```

Listing 3.2: An ineffective `TrustManager`. Source: `http://stackoverflow.com/a/4837230`

At a glance, one can see that the crucial `checkServerTrusted` method contains no code, and so will trust any server it encounters. There are many warnings surrounding this, and Google's own documentation cautions:

> Many web sites describe a poor alternative solution which is to install a TrustManager that does nothing. If you do this you might as well not be encrypting your communication, because anyone can attack your users at a public Wi-Fi hotspot by using DNS tricks to send your users' traffic through a proxy of their own that pretends to be your server. The attacker can then record passwords and other personal data. This works because the attacker can generate a certificate and - without a TrustManager that actually validates that the certificate comes from a trusted source - your app could be talking to anyone. So don't do this, not even temporarily. You can always make your app trust the issuer of the server's certificate, so just do it.[76]

Nevertheless, code like this still makes it into widely downloaded apps published on the Play Store.

3.1.2 Custom HostnameVerifiers

The second thing that must occur during certificate verification is hostname validation: that is, making sure the hostname provided in the certificate matches the hostname of the server the client is trying to connect to. This can be complicated by the use of virtual hosting, as many hostnames are located on one server, but the use of the Server Name (SNI) extension (as described in Section 2.1.7 and the RFC [15, Sec. 3]) can alleviate this. Unfortunately Apache HTTP Client does not support SNI, so any developer that chooses to use this API will be stuck. Section 6.3.1 examines the prevalence of the SNI extension among Android apps.

A developer unaware, or unable, to use SNI may, for ease of use, simply decide to disable hostname verification:

```
MySSLSocketFactory.setHostnameVerifier(SSLSocketFactory.
    ALLOW_ALL_HOSTNAME_VERIFIER);
```

This will solve a developer's problem, but will leave the client app trusting any correctly signed certificate, no matter the hostname. Alternatively, they may decide to create their own HostnameVerifier, and as with the TrustManager, try and look for a solution on the Internet. They may find code like that in Listing 3.3.

```
DO_NOT_VERIFY = new HostnameVerifier() {
                @Override
                public boolean verify(final String hostname, SSLSession
                    session) {
                    return true;
                }
            };
```

Listing 3.3: An ineffective `HostnameVerifier`

Again, at a glance, it is clear that this method performs no verification whatsoever and so offers no protection, but code like this can be found in popular apps on the Play Store.

A more insidious issue is when a developer has decided to use a custom SSLSocket implementation. This does not include any hostname verification whatsoever, and so a developer must remember to include it. If she forgets, there will be no warning or obvious notification; the app will just become silently vulnerable.

3.1.3 onReceivedSslError

Another source of error in Android apps comes when a developer wants to implement a WebView. A WebView is an interface in an app which essentially just displays a webpage. If that webpage loads over a TLS connection, it is possible to override the `onReceivedSslError` method as in Listing 3.4.

```java
public void onReceivedSslError(WebView view, SslErrorHandler handler
    , SslError error)
{
    handler.proceed();
}
```

Listing 3.4: An ineffective WebView SSL error handler. Source: `http://stackoverflow.com/a/5978391`

This instructs the app to completely ignore any SSL errors and simply proceed as normal, completely removing any security that the connection may provide. Patrick Mutchler at Stanford University examined this particular vulnerability in 2014 and found 5% of apps with this code [77]. As the cited StackOverflow answer demonstrates, developers face an issue with their app and are desperate to get it working with the easiest method possible. If that entails removing security from the connection then some developers will happily do that.

3.1.4 Summary

	Correct hostname Signed by trusted root	Any hostname Signed by trusted root	Correct hostname Signed by anyone	Any hostname Signed by anyone
Secure Implementation	Allowed	Rejected	Rejected	Rejected
No hostname verification	Allowed	Allowed	Rejected	Rejected
No trust chain verification	Allowed	Rejected	Allowed	Rejected
No hostname verification No trust chain verification	Allowed	Allowed	Allowed	Allowed

Table 3.1: Summary of certificate validation errors and their consequences

Table 3.1 shows how an app coded with these vulnerabilities will behave when faced with certain certificates. This suggests the strategies that a man-in-the-middle attacker can attempt when trying to gain information from the app. Any app that follows anything but the first row of the table is insecure, and liable to leak confidential information.

3.2 A Solution

Various mitigations for these weaknesses have been proposed, including updating the Android API to enforce secure connections [30] and scanning apps for security weakness upon uploading to application stores [31], but these have not been incorporated into the Android ecosystem so far. Even if these solutions were to be implemented, a user, even if she were technically able, is unable to examine the security of a connection: unlike with a desktop web browser, one cannot examine the certificate, or check the version of the TLS protocol being used (if indeed it is being used at all).

Therefore, an Android application was proposed (as suggested by Fahl et al. [31, p59]), designed to sit between an application and its connection to the internet, and inform a user as to the security of their connection, including any relevant protocols and the validity of any certificates. This would provide protection to users against poorly coded apps, as well as alerting them if they were connected to a malicious WiFi hotspot - a notification could appear warning them that any certificates presented to any app was invalid and that they should disconnect immediately.

Chapter 4

Implementing the App

The app aims to replicate the warnings given to users by desktop browsers, allowing normal internet use to continue, but warning if weaknesses arise, and completely blocking the connection if it cannot be deemed trustworthy. This section will detail the architecture of the app, the design choices that had to be made, and some implementation challenges that had to be overcome.

4.1 Design

4.1.1 Goals

For the app to be successful it had to achieve two primary goals.

- It must be easy to use and run, not just for technical users but for all users. This means a minimum of configuration and the ability to install the app, press one button and let the app do the rest. Of course, it may offer some configuration ability for more advanced users. It should also run without any user modification to the phone: no root access to the operating system should be required.

- It must be as correct as possible. That is, it must be clear when an issue is discovered, and crucially, not permit insecure traffic without explicit user approval. It is better to be more, rather than less, restrictive. It must have as few false positives and negatives as possible.

4.1.2 Development Environment and Tools

As an Android app, the tool was developed in Java. The integrated development environment (IDE) used was Android Studio[1] running on an Apple MacBook Air. Development occurred mostly on a Samsung Galaxy S6 running Android Lollipop 5.0.2, although other phones were used for testing. Git was used for version control and code was synchronized with the Imperial College Department of Computing GitLab server for secure backup. Development was modelled on the Agile methodology, in which rapid prototypes and iterations were developed and worked on in two week sprints.

4.1.3 Issues and Challenges

There were several issues to be aware of when developing this application.

- If applications use custom protocols to generate a secure connection, this application will be unable to verify the integrity of the connection. However, a survey of the Android ecosystem suggests that using the in-built SSL API is the most common way that apps attempt to secure a connection between client and server.

- Issues may be faced when trying to assign connections to applications: a modern smartphone will have many concurrent connections to the internet at once, and so the application must be able to distinguish relevant connections for the foreground app.

[1] `http://developer.android.com/tools/studio/index.html`

- As pointed out by Fahl et al. [30], in general using self-signed certificates creates a vulnerability, as an application will then trust a certificate signed by an attacker. However, if an app successfully implements certificate pinning and pins a self-signed certificate, this connection will still be secure. The application may be unable to verify this. It may be possible to present a forged certificate to an application to determine acceptance, but this solution will need to be examined.

- As certificate verification is hard to implement correctly, there is a risk that this application contains a subtle bug that verifies a certificate as valid even if it is not. This would lead to a false sense of security for the user and would be worse than providing no indication at all.

- Developing on a mobile phone introduces efficiency challenges. Battery life becomes a key concern, as does the importance of not adding any additional latency to a connection. This application will attempt to minimize both battery usage and latency, but this may prove demanding.

4.2 Initial Decisions

The most important decision to be made for the structure of the app was to decide the method of traffic interception. The decision was made to analyse all traffic on the phone. This avoids having to maintain a third-party server to which a user's phone must be constantly connected. While maintaining a third-party server allows insight into connections from many users, allowing heuristics to be developed as to what constitutes a compromised connection, there are several downsides to this approach. With an app of this nature, user privacy is very important, and the possibility that traffic may be stored on a system outside of a user's control would be hard for some to accept. In addition, routing traffic through another server would add significant latency to the connection. The financial and practical implications of hosting a server of this nature would also not be insignificant.

With the decision thus made to analyse traffic on the user's device, the method of doing so had to be determined. The Android operating system provides three methods for doing this; running a local proxy server, running a local virtual private network (VPN), or observing the traffic directly as it passes through the kernel.

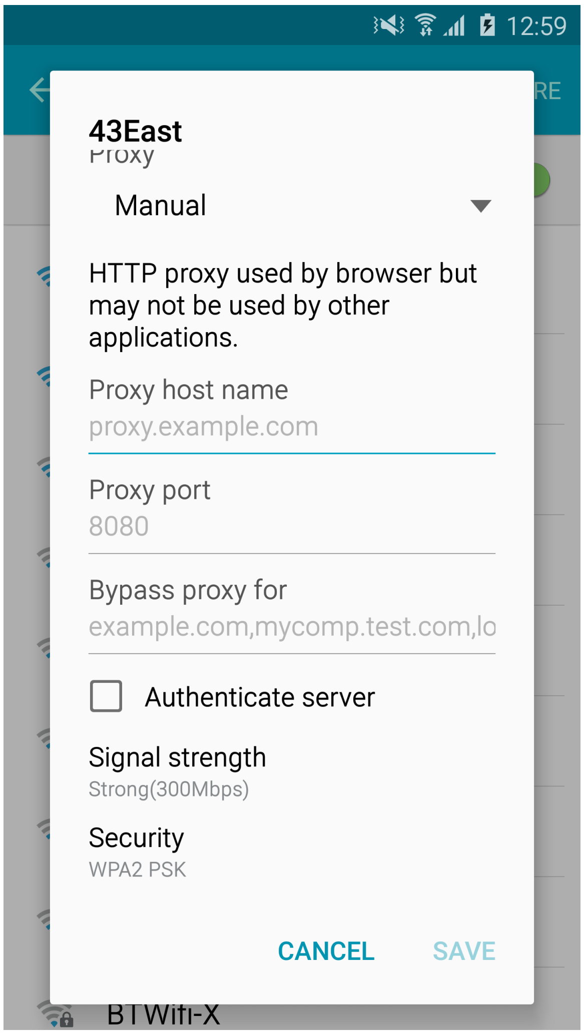
4.2.1 Local Proxy Server

Using a local proxy server is perhaps the easiest method to implement. It does not require root access to the phone and it operates at the HTTP layer, thus requiring little additional work to reconstruct the network traffic once intercepted. According to RFC 2817, SSL connections are tunnelled over an HTTP proxy server using the CONNECT method [78], which also allows easy comprehension of the server an app is attempting to connect to, which is important for performing hostname verification.

There are some downsides to this approach, however. It is the most difficult for a user to configure, as, since version 4.1.2, the Android system does not let apps automatically configure a proxy for security reasons. This means that for each WiFi connection, a user must manually point traffic to the proxying app (Figure 4.1) - a step that will quickly be forgotten. This also restricts screening to WiFi connections, as 3G and 4G mobile data connections cannot be proxied. Additionally, apps can be coded (either accidentally or deliberately) to avoid a configured proxy altogether, and so any app implementing a proxy server would not be able to analyse and secure any connections such an app makes.

Existing apps that use this approach include Adblock Plus for Android[2] which allows users to block advertising on their phones.

[2]https://adblockplus.org/en/android-about


12:59

RE

43East

Proxy

Manual

HTTP proxy used by browser but may not be used by other applications.

Proxy host name

proxy.example.com

Proxy port

8080

Bypass proxy for

example.com,mycomp.test.com,lo

☐ Authenticate server

Signal strength

Strong(300Mbps)

Security

WPA2 PSK

Figure 4.1: Complicated Android Proxy Configuration

4.2.2 Local VPN

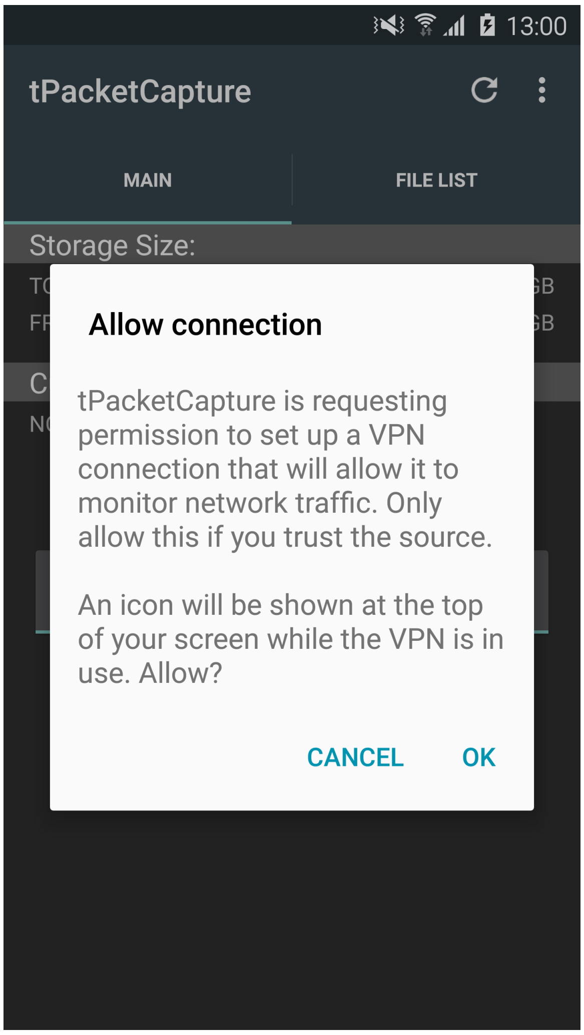
The Android API supports a VpnService,[3] primarily designed for establishing tunnels to third-party VPN servers, and like the local proxy method, does not require root access. This exposes a TUN interface that operates on the Internet layer. This means that packet interception will include an IP header, a TCP or UDP header, and an application layer header (e.g. DNS or HTTP). Ideally these packets could be forwarded as is, but unfortunately this would require raw sockets, which is not available on Android without root (and in fact there is no raw sockets API in Java).

This means that layer translation would have to take place: the VpnService has to strip the Internet and Transport layer headers and forward the Application layer data over a new socket. When data is received back, the Internet and Transport layer headers must be added back on. While this is relatively straightforward for UDP packets, for TCP packets, the TCP connection (the handshake, acknowledgements, etc) must be managed, requiring a TCP/IP stack. If this approach is taken, a lot of work must go into this.

On the positive side, such an approach is more straightforward for a user. They must simply accept one system dialogue (Figure 4.2) and from then on all traffic will go through the VPN.

Unfortunately, this approach limits the use of proxy servers, as Android does not support both proxying a connection and running a VPN. Additionally, this may have a larger impact on battery life than the local proxy approach. Because Android also requires explicit user approval to start the VPN (again for security reasons), the app cannot be set to run automatically when the phone is turned on. Additionally, Android's VPN does not currently support IPv6. This is not currently

[3]`http://developer.android.com/reference/android/net/VpnService.html`



Allow connection

tPacketCapture is requesting permission to set up a VPN connection that will allow it to monitor network traffic. Only allow this if you trust the source.

An icon will be shown at the top of your screen while the VPN is in use. Allow?

CANCEL OK

Figure 4.2: Android VPN Approval Dialogue

an issue for WiFi traffic, as there is very little IPv6 support on the Internet, but could become significant in a few years time, and indeed some carriers' 4G implementations rely solely on IPv6 already.

There are several existing apps that take this approach either to block all traffic, block specific traffic, or monitor traffic, suggesting that this approach is achievable and that there is an existing market of users happy to follow these steps. These apps include:

- AdGuard Android:
`http://adguard.com/en/adguard-android/overview.html`

- Mobiwol: NoRoot Firewall (100,000-500,000 installs):
`https://play.google.com/store/apps/details?id=com.netspark.firewall`

- NoRoot Firewall (500,000-1,000,000 installs):
`https://play.google.com/store/apps/details?id=app.greyshirts.firewall`

- tPacketCapture (100,000-500,000 installs):
`https://play.google.com/store/apps/details?id=jp.co.taosoftware.android.packetcapture`

- Packet Capture (10,000-50,000 installs):
`https://play.google.com/store/apps/details?id=app.greyshirts.sslcapture`

4.2.3 Examining Traffic Directly

Taking this approach requires root access to the phone, so was almost immediately ruled out, but it is worth looking at the possibilities offered to see what might be lost by not following it. This would either use libpcap, the Linux packet capture library, or iptables (which would require writing a kernel module). As this approach is more low-level than both a proxy or VPN, the impact on battery life would be more limited. Analysis is also likely to be quicker and more efficient. The AdAway app[4] uses this approach.

4.2.4 Decision

Based on these considerations, the choice was made to take the VPN approach. This required the least work on the users' part, and had the chance of capturing the most traffic from apps, making the additional work required worth it. Additionally, a basic existing implementation had been created by Mohamed 'Hexene' Naufal[5] and licensed as Apache v2.0, allowing free use and modification. This would save a lot of the initial work and meant connection analysis could begin much more quickly.

4.3 Building on LocalVPN

Nauful's LocalVPN app is extremely basic. As seen in Figure 4.3, the user interface is a single button. The app merely passes traffic through the VPN without doing any manipulation or analysis. This is a good proof of concept, but requires a lot of work on top.

4.3.1 VPN Architecture

The VPN runs as a Service, which on the Android operating system means it can continue to run in the background, even while the main app is closed. It extends Android's VpnService which provides basic scaffolding for setting up a VPN. The service runs a thread which continually receives and send traffic to and from the network. Data is buffered in directly allocated byte buffers for performance as they are allocated separately from the Java virtual machine (JVM) heap. A basic TCP/IP stack is implemented. This handles connection set-up and tear-down and deals with the most important TCP flags. TCP connections are cached in a Transmission Control Block implemented as a Least Recently Used cache of 50 entries.

Each packet is initialized as a Java object, allowing inspection of the IP and UDP/TCP headers. This also allows the headers to be recalculated as required for the layer translation mentioned above.

4.3.2 Packet Capture

The first step to turning this pass-through app into something useful was being able to understand what was happening in it. This required being able to observe the traffic with a tool such as Wireshark. The pcap file format, which Wireshark uses, is relatively simple [79]. The file requires a global header (which marks if the packets are stored as little endian or big endian, the version number and other key information), and each packet is proceeded by a header which includes a timestamp and the length of the packet.

With some experimentation, the header in Listing 4.1 was found to be sufficient.

```
0xA1B2C3D40004000200000000000000000000FFFF0000000C
```

Listing 4.1: Wireshark Header

The Packet class was also extended to include the required packet header and the requisite timestamp. Writing these bytes out to a file on the Android external file system produced files which can be opened with Wireshark and easily analysed.

[4]https://f-droid.org/repository/browse/?fdid=org.adaway
[5]https://github.com/hexene/LocalVPN

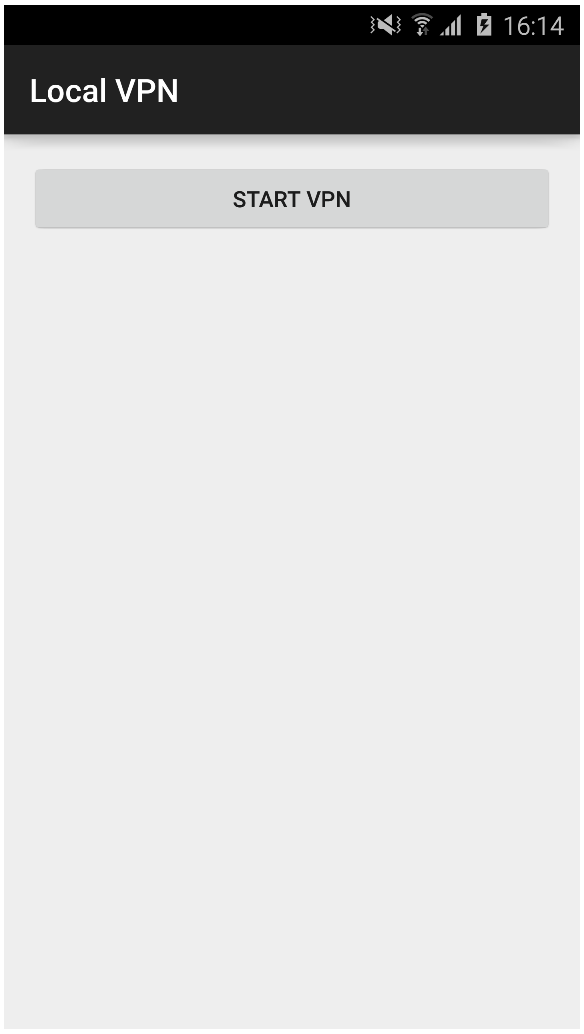


Figure 4.3: LocalVPN's Basic UI

4.4 Finding and Analysing TLS Packets

Now that it is possible to capture packets and examine them in more detail, focus can move to TLS packets in particular. A simple heuristic for determining if traffic is TLS is if it is being transmitted over port 443. A future expansion could dynamically examine packets on other ports to determine if they are encrypted too, but for now just looking at all traffic on port 443 will suffice.

Looking into the packet header it is easy to determine if the source or destination port is 443 and so select the packet for further examination. This occurs on the TCPInput and TCPOutput threads; whenever they detect an appropriate packet, they alert they instruct the transmission control block (TCB) object for that connection to analyse the packet. This method takes the bytes given to it, and constructs a TLSPacket object (similar to the generic Packet object). Following the specification in the TLS RFCs [3, 4, 5], a header can be constructed (Listing 4.2), and further analysis can be performed.

```
public static class TLSHeader
    {
        public byte majorVersion;
        public byte minorVersion;
        public ContentType type;
        public short length;
        public int payload;

        public enum ContentType
        {
            change_cipher_spec(20),
            alert(21),
            handshake(22),
            application_data(23),
            Other(0xFF);
            ...
        }

    ...
    }
```

Listing 4.2: The TLS Record Header as a Java Object

Specifically, we are most interested in the handshake packets. We can select the packets of this type, and again create a Java object for them (Listing 4.3).

```
public static class TLSHandshake
    {
        public static final int TLS_HANDSHAKE_HEADER_SIZE = 4;

        public HandshakeType msg_type;
        public int length;
        ...

        public enum HandshakeType
        {
            hello_request(0),
            client_hello(1),
            server_hello(2),
            certificate(11),
            server_key_exchange (12),
            certificate_request(13),
            server_hello_done(14),
            certificate_verify(15),
            client_key_exchange(16),
            finished(20),
            Other(0xFF);
            ...
         }

    ...
    }
```

Listing 4.3: The TLS Handshake Header as a Java Object

Drilling down even further, headers can be constructed for the `client_hello`, `server_hello`, `certificate`, and other key handshake packets. They are omitted here for brevity but can be found in the `TLSPacket.java` source file. This allows us to extract key information about the connection before it is encrypted and we can no longer read it. From the `server_hello` packet we can determine the TLS Version and Cipher Suite chosen by the server, and from `client_hello` we can see the Cipher Suites that the client supports - this will be very useful in analysing the state of the ecosystem and searching for possible vulnerabilities in servers and clients (Section 6.3.1).

Crucially, the `certificate` packet gives us the certificate chain used in the connection. This stream of bytes can be constructed as an X509Certificate from the Android API.[6] Now we are able

[6]`http://developer.android.com/reference/java/security/cert/X509Certificate.html`

to attempt to verify the hostname and chain of trust, and determine if a man-in-the-middle attack is being staged against the user. We can also write the certificate to a file on the internal storage for later manual analysis.

4.4.1 Caveats

This technique allows us to examine the handshake as it passes across the network, without slowing down the connection. To avoid issues, it is important to be aware of some lower level details. As the RFC states,

> The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2ˆ14 bytes or less. Client message boundaries are not preserved in the record layer (i.e., multiple client messages of the same ContentType MAY be coalesced into a single TLSPlaintext record, or a single message MAY be fragmented across several records) [5, Sec 6.2.1].

Because we are looking at the connection packet-by-packet, this is crucial information. It is important not to throw away data by considering a packet just to contain one TLS record, and similarly, it is important not to miss information by reading only half a packet. Careful use of the buffer is required in this instance - packets may even span more than two TCP packets depending on their size (although this will mainly be the case for Application Data, in which we are not interested).

It is also important to note when the client and server have agreed on an encryption key and the connection is encrypted. This will be signified by a `ChangeCipherSpec` packet. Any further data received after this point may seem to be a regular handshake packet [5, Sec 7.1], but this will be coincidence, and any attempt to read further will cause an error.

4.5 Certificate Verification

As discussed in Section 3.1, the reason many apps fail to verify certificates correctly is because they have overridden the default Android API. Therefore, to enforce correct verification, it is enough to simply pass the obtained certificates to the Android API. Exceptions will be thrown if there is an issue. Verifying the chain of trust is relatively simple, as Listing 4.4 demonstrates, but hostname verification throws up some challenges.

```
private void validateCertChain(X509Certificate[] chain) {
    try {
        TrustManagerFactory trustManagerFactory =
           TrustManagerFactory.getInstance(TrustManagerFactory.
           getDefaultAlgorithm());
        trustManagerFactory.init((KeyStore) null);
        for (TrustManager trustManager : trustManagerFactory.
           getTrustManagers()) {
            X509TrustManager x509TrustManager = (X509TrustManager)
               trustManager;
            x509TrustManager.checkServerTrusted(chain,
               TrustManagerFactory.getDefaultAlgorithm());
        }
    } catch (CertificateException e) {
        ...
    }
}
```

Listing 4.4: Validating the Certificate Chain

4.5.1 Finding the Hostname

As mentioned in Section 4.2.1, if using the Local Proxy approach, finding the hostname is just a case of looking at what server was named in the HTTP CONNECT request. Since this approach was not taken, an alternate approach must be found.

4.5.1.1 SNI

The easiest way of achieving this is to use the Server Name Indication extension in the Client Hello packet. As described in Section 2.1.7 and the RFC [15, Sec. 3], the packet contains a field that describes the host that it is attempting to connect to. Listing 4.5 shows an example of this. SNI is intended to provide support for HTTPS over virtual servers, that is, where one physical server hosts multiple websites all with different hostnames. Without a field in the packet identifying the intended site, either a certificate would have to be valid for all sites on the physical host, or certification would not work at all. We can take advantage of this field and verify the hostname in any received certificate against the hostname in the SNI field (Listing 4.6).

```
Extension: server_name
                Type: server_name (0x0000)
                Length: 19
                Server Name Indication extension
                    Server Name list length: 17
                    Server Name Type: host_name (0)
                    Server Name length: 14
                    Server Name: www.google.com
```

Listing 4.5: The SNI extension

```
private void validateSNIName(X509Certificate cert) {
    MyHostnameVerifier hv = new MyHostnameVerifier();
    try {
        hv.verify(sni_name, cert);
    } catch (SSLException e) {
        ...
    }
}
```

Listing 4.6: Verifying SNI

4.5.1.2 DNS

Unfortunately however, not every TLS connection includes the SNI extension. For one thing, SSL3 does not have the ability to include extensions, so this technique will not work with any connections using SSL3. In addition, as mentioned in Section 3.1, Apache HTTP Client does not support SNI, so any apps using this library will also fail when using this technique. There are also various other apps whose TLS connections do not include the SNI field. For this reason a secondary technique must be found.

An initial approach would be to take the IP address of the connection and do a reverse DNS lookup using the `getHostName()` method of `InetAddress`.[7] Of course, this does not solve the problem of virtual hosting, as the hostname returned, if any, will be that of the physical server. One could also take the reverse approach, and lookup the IP addresses of the hostnames stored in the certificate with the `getAllByName()` method. If any of these IP addresses match the source IP address then we can be reasonably certain that the certificate's hostname is valid. This approach however is vulnerable to a DNS spoofing attack. If an attacker can control the DNS responses a

[7]`http://developer.android.com/reference/java/net/InetAddress.html`

client receives (which is certainly possible), she can simply configure her own hostname presented in the certificate to match that of the intended connection, and trick the app into believing the certificate is valid. Thus, while this technique may offer a small amount of verification, it is subject to both false positives and false negatives and so cannot be seen as completely reliable.

An alternative approach is to try and emulate the HTTP CONNECT message that would be received if running a local proxy. This can be achieved by using a DNS cache. The VpnService can be configured so that all DNS traffic passes through the VPN. Then, by observing all UDP traffic over port 53 (in a similar manner to the way TLS traffic is analysed) it is possible to see all the DNS queries and responses that the user's phone is sending. By turning these messages into a DNSPacket Java object (following the DNS RFC, 1035 [80]), we can learn the intended hostnames an app is intending to connect to, and the associated IP addresses. By storing these in a cache, when attempting to verify a hostname in a certificate, we can lookup the IP address that the app is trying to connect to, and find the associated hostname or hostnames. We can then verify that the received certificate matches one of these hostnames (Listing 4.7).

```
private void validateHostname(X509Certificate cert, InetAddress
    source) {
    MyHostnameVerifier hv = new MyHostnameVerifier();
    try {
        List<String> l = DNSCache.getDomain(source.getHostAddress())
            ;
        if (l != null) {
            boolean verified = false;
            SSLException exception = null;
            for (String domain : l) {
                try {
                    hv.verify(domain, cert);
                    verified = true;
                    break;
                } catch (SSLException e) {
                    exception = e;
                }
            }
            if (!verified)
                throw exception;
        }
        else {
            ...
        }
    } catch (SSLException e) {
        ...
    }
}
```

Listing 4.7: Verifying hostnames through a DNS cache

4.5.1.3 Android Bugs

This technique works very well, and would solve this problem completely, were it not for one issue. There is a bug in the Android operating system that means all DNS requests routed through the VPN are not rewritten with the correct IP address, but instead use the internal VPN address.[8] This means that no DNS replies are received, as the router does not know the correct IP address of the device. This of course, completely kills internet connectivity. It appears that the bug has been

[8] https://code.google.com/p/android/issues/detail?id=64819

fixed in the latest version of Android, but whether specific device manufacturers have included the fix is unclear: the bug is still present on a Samsung Galaxy S6 running Android 5.0.2 and a Google Nexus 4 running Android 5.1.

After much investigation, a workaround was found for this issue. Using a rooted phone, `iptables` was configured to log all traffic it handled:

```
iptables -t nat -A OUTPUT -j LOG --log-uid --log-level debug --log-ip-
    options
```

Viewing the logs with `dmesg` revealed that the source IP address of certain UDP packets wasn't being rewritten to the IP of the outgoing interface. This could be resolved on a rooted phone by adding an `iptables` rule:

```
iptables -t nat -A POSTROUTING -p udp -j MASQUERADE
```

Most phones of course are not rooted, so another solution needed to be found. After exploring many options, it was discovered that when a UDP socket is opened by the VPN, if it is explicitly bound to the IP address of the outgoing interface, then the source IP addresses of the packets are correct. This was achieved with the code in Listing 4.8. This was a significant discovery, as without it hostname verification would not work on many large networks.

```java
InetSocketAddress sa = new InetSocketAddress(localIP, sourcePort);
outputChannel.socket().setReuseAddress(true);
outputChannel.socket().bind(sa);
```

Listing 4.8: Workaround for Google's VPN DNS bug

Another DNS issue that arose was that some large networks (including that of Imperial College London) block DNS requests to external servers, and force all traffic to go through their local DNS server. The app was initially configured to send all DNS traffic through Google's DNS server, `8.8.8.8`, but code had to be added to also use the local DNS server provided. This of course is vulnerable to DNS spoofing, where the returned IP address of a domain points to a server controlled by an attacker, and so the option is available to disable this server and use solely Google's external one.

4.5.2 False Positives

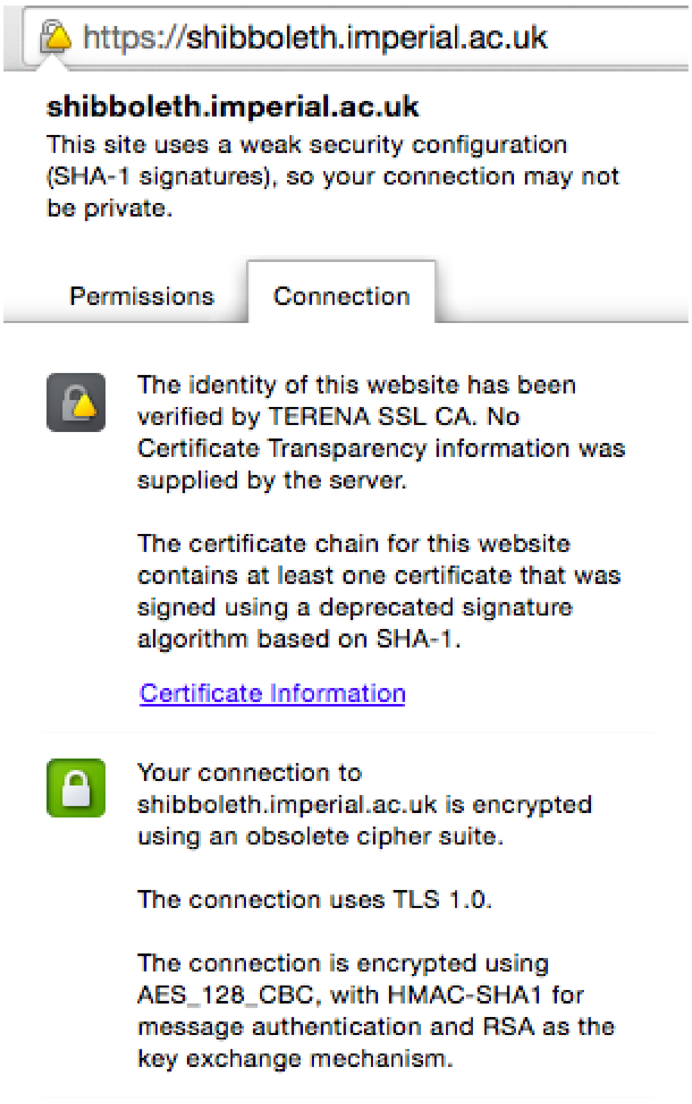
These techniques prove to be effective in verifying both the chain of trust and hostnames for standard certificates. There are one or two cases where false positives are thrown and that users must verify manually to ensure protection. Normally an app relying on a self-signed certificate would be vulnerable to attack, because an attacker could present their own self-signed certificate for the same hostname and be trusted. In this situation the app will alert the user to a potential attack. If however, the self-signed certificate is 'pinned' in the app, that is, the app is set to trust that certificate, and *only* that certificate, then security is still achieved. There is no way for a third party app to verify that an app is using certificate pinning, and so manual verification must be used.

In addition, certain apps hardcode the specific hostnames that they trust, in addition to the hostname they are attempting to connect to. That is, if they try and connect to `domain-one.com` and receive a certificate valid for `domain-two.com`, the app is configured to see this as a valid certificate. Again, a third-party app is not able to know this, and so will alert the user to a certificate error, which must be manually inspected and approved (or not).

Specific examples of both of these cases are in Section 6.2.

4.6 Grading Connections

Now that it is possible to inspect the TLS handshake, and verify that the certificates are valid or not, we can decide if a connection should be allowed to proceed or not. However, with awareness of the various protocol and implementation vulnerabilities discussed in Sections 2.3 and 2.4, we can also take a more fine-grained approach. Even if the connection's certificate is valid, an old version of the TLS protocol may be in use, or a weak Cipher Suite may be in effect, which are both conditions that may affect the security of a connection. Blocking a connection may be a little excessive in this case as users may quickly become frustrated and disable the app, but it is useful to give a warning to the user that the connection is not as secure as it could be (in the same manner as web browsers do on the desktop: Figure 4.4).



https://shibboleth.imperial.ac.uk

shibboleth.imperial.ac.uk

This site uses a weak security configuration (SHA-1 signatures), so your connection may not be private.

The identity of this website has been verified by TERENA SSL CA. No Certificate Transparency information was supplied by the server.

The certificate chain for this website contains at least one certificate that was signed using a deprecated signature algorithm based on SHA-1.

Certificate Information

Your connection to shibboleth.imperial.ac.uk is encrypted using an obsolete cipher suite.

The connection uses TLS 1.0.

The connection is encrypted using AES_128_CBC, with HMAC-SHA1 for message authentication and RSA as the key exchange mechanism.

Figure 4.4: A TLS security warning from Google Chrome

A grade from A to F can be computed for each connection in a similar manner to Ristić's SSL Labs. This allows users to make an informed decision about whether to proceed with using an app without blocking it outright. After some consideration, the grading criteria outlined in Table 4.1 was implemented in the app. This grade is displayed on the connection information page and is colour-coded to highlight the meaning of each letter without a less technical user having to fully understand. The list of TLS Cipher Suites is defined by the Internet Assigned Numbers Authority (IANA) [81], and with reference to the literature, can be marked as secure, insecure or obsolete. See Appendix C for the definitions used in this app.

In addition, each app can be given a grade, based on the connections it makes. Initially, this is a simple algorithm, and is simply the lowest grade of the previous ten connections. This is sufficient to highlight any active poor connections in an app, but allows a grade to develop over time as an app is updated or a phone connects to a different WiFi hotspot.

Grade	Criteria
A	Using TLS version 1.2 and a strong cipher suite
B	Using a version of TLS below 1.2 or a weak cipher suite or a certificate with a weak signature
C	Invalid certificate presented: attempted attack on connection
F	FAIL: Invalid certificate presented and further traffic allowed to proceed

Table 4.1: Grading Criteria

4.7 Linking Connections to Apps

Although the local VPN allows us to examine all the connections that the phone is making to the internet, connections are not explicitly marked with the name of the app that is making the connection. This is an issue, because we need to identify which apps are making which connections in order to assess their security. Luckily, because of Android's roots in the Linux operating system, a solution is at hand. The `/proc/net/tcp` and `/proc/net/tcp6` interfaces provide information about the TCP connections the operating system has made, and are readable without requiring root access. The format is as in Listing 4.9. IP addresses are given in hexadecimal format and the UID column gives the ID of the user that owns a particular connection. Because each app on Android runs as a different user this means we can link connections to apps.

```
$ cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when
     retrnsmt   uid  timeout inode
   0: 1D00A8C0:9007 03D23AD8:01BB 01 00000000:00000000 00:00000000
      00000000 10091        0 440101 1 0000000000000000 23 4 1 10 -1

$ cat /proc/net/tcp6
  sl  local_address                                remote_address
                             st tx_queue rx_queue tr tm->when retrnsmt
     uid  timeout inode
   0: 0000000000000000FFFF00001C00A8C0:E4DE 0000000000000000
      FFFF000003D23AD8:01BB 01 00000000:00000000 00:00000000 00000000
      10219        0 445576 1 0000000000000000 25 4 11 10 -1
```

Listing 4.9: Output of /proc/net/tcp and /proc/net/tcp6

The AndroidTCPSourceApp library[9] helps accomplish this. It loads these interfaces into a `BufferedReader` and searches for a matching IP address, before looking up the UID (via the Android API, Listing 4.10), and returning an object with information about the appropriate app. This library required a little rewriting, as the regular expressions it used to search were not complete, and to increase efficiency, but was useful in saving the initial work. With a package name it is easy to use the Android API to get the app's user-friendly name and icon to present to a user in the user interface (Listing 4.10).

```
PackageManager manager = context.getPackageManager();
String[] packagesForUid = manager.getPackagesForUid(pidEntry);
String packageName = packagesForUid[0];
```

Listing 4.10: Getting a Package Name from a UID

4.8 Data Persistence

Without some way to store this information on the phone, every time the app closes (or is forced to close by the operating system), the historical data of connections made is lost, affecting one's

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

ability to look back at an app's previous behaviour and perform data analysis. Thus, some way is needed to store connections for easy recall. Storing this information in a database would be the most obvious solution, and indeed, Android provides an easy way for apps to create and use a SQLite database. However, this still requires writing SQL statements, which adds complexity. For this reason it would be ideal to use an object relational mapper (ORM). This allows one to map Java objects to database tables, without having to write any SQL. Active Android[10] is just such a library, and is used heavily in this app. An ER diagram for the database schema used by the app is shown in Figure 4.5.

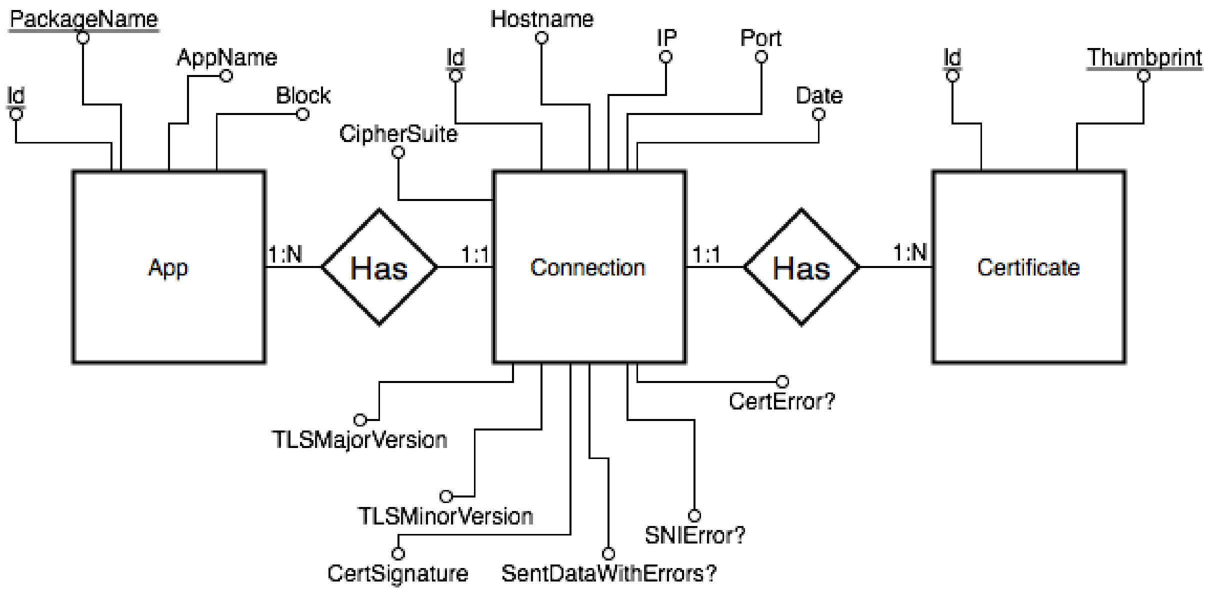


Figure 4.5: ER Diagram

Conveniently, it also provides an easy representation of the key elements the app needs to deal with. Whenever the app interfaces with Connection, App or Certificate objects, that data is directly backed by the database, with no intermediate layer required. Certificates are written to files on the filesystem, indexed by their fingerprint. With Active Android, reading rows from tables is made very easy, as regular Java syntax can be used. To update or create a row, all that is needed is to call the `save()` method.

4.9 The User Interface

As this app is intended for use by non-technical users, some time was dedicated to creating a simple, easy to use interface. The design language is influenced by Google's Material Design, which is intended to promote a common experience among all Android Apps. On the main screen (Figure 4.6), apps are represented as 'cards' which can be swiped and tapped. A large message is displayed to indicate to the user whether protection is enabled or not (recall that protection cannot be enabled automatically, because user interaction is required to start the VpnService).

The cards are implemented using a `CardView` from Android's support library, and the list is created using a `RecyclerView` (which is more memory-efficient than the older `ListView`).

On the bottom right corner one will also notice the large button. This is a 'Floating Action Button' which is designed to highlight the primary action of an app and provide easy and consistent access to it. In this app it is used to start and stop the VpnService. It is implemented using the `FloatingActionButton` widget from the Android Design Library.

Figure 4.7 shows the page that is displayed when an app card is selected. A large, colour-coded grade is displayed. The user is also able to decide whether to always block an app, never block an app, or let blocking be decided based on analysis (the default behaviour). The list of historic

[10]`https://github.com/pardom/ActiveAndroid`

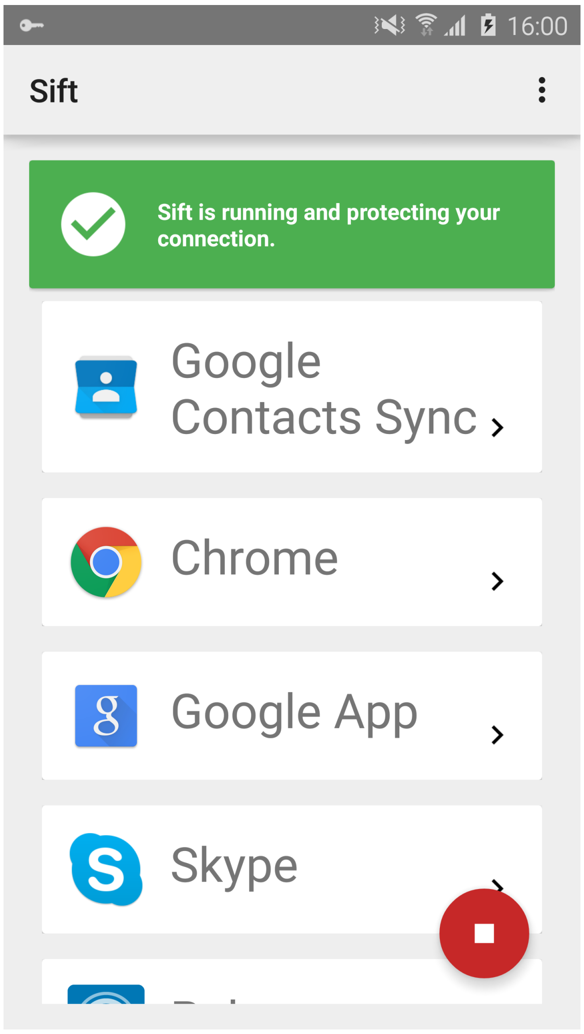


Figure 4.6: Main Screen

connections is displayed below. Each connection can be chosen to display further detail. If a connection is problematic, it is highlighted in red.

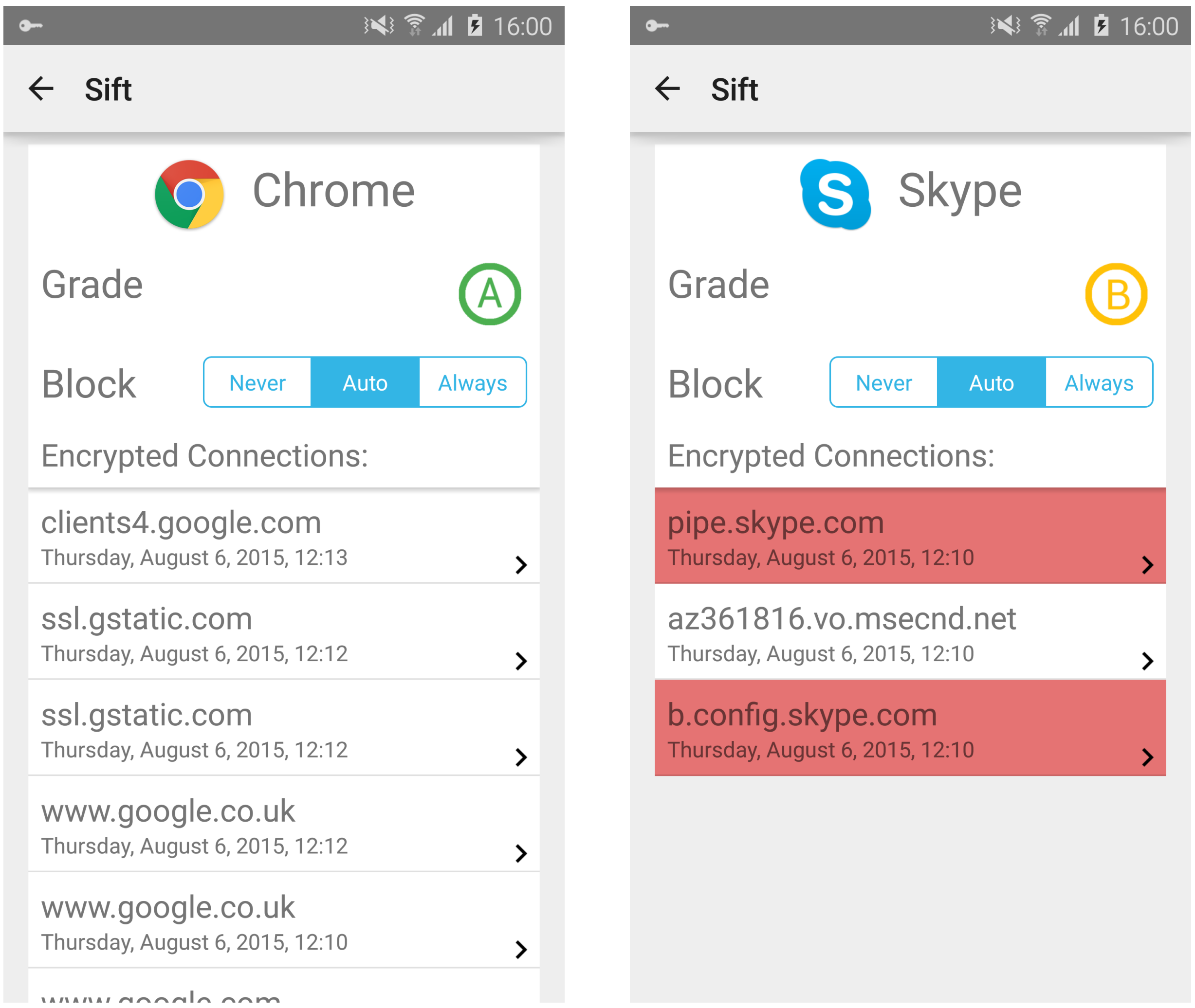
The connection screen (Figure 4.8) provides more information about an individual connection. Its grade is displayed, along with the IP address, date and time. The version of the TLS protocol used is highlighted, as is the cipher suite that was negotiated. Any provided certificate is displayed below.

Figure 4.9 shows an app that has been under attack, and leaks data. It is marked clearly with an F grade, and notifies the user that the app has been blocked.

Of course, one cannot rely on a user constantly opening the app to examine what has been happening. If problematic events are encountered, Android notifications are generated to alert the user about the situation (Figure 4.10). These can be enabled or disabled in the app's preferences, which also allows packet capture to be enabled or disabled.

4.9.1 Identity

The app also needs an identity: a name and a logo so that it can be identified on users' phones and on the Google Play Store. The name Sift was chosen, as it represents the app's ability to filter connections, and allow only the appropriate ones through. A logo was also created, again following the Material design language, and is shown in Figure 4.11. With these elements in place, the app can be submitted to the Google Play Store for general use, allowing anyone to download and use the app.


16:00

Sift

Chrome

Grade A

Block: Never [✓ Auto] Always

Encrypted Connections:

clients4.google.com
Thursday, August 6, 2015, 12:13

ssl.gstatic.com
Thursday, August 6, 2015, 12:12

ssl.gstatic.com
Thursday, August 6, 2015, 12:12

www.google.co.uk
Thursday, August 6, 2015, 12:12

www.google.co.uk
Thursday, August 6, 2015, 12:10

www.google.com

(a)

16:00

Sift

Skype

Grade B

Block: Never [✓ Auto] Always

Encrypted Connections:

pipe.skype.com
Thursday, August 6, 2015, 12:10

az361816.vo.msecnd.net
Thursday, August 6, 2015, 12:10

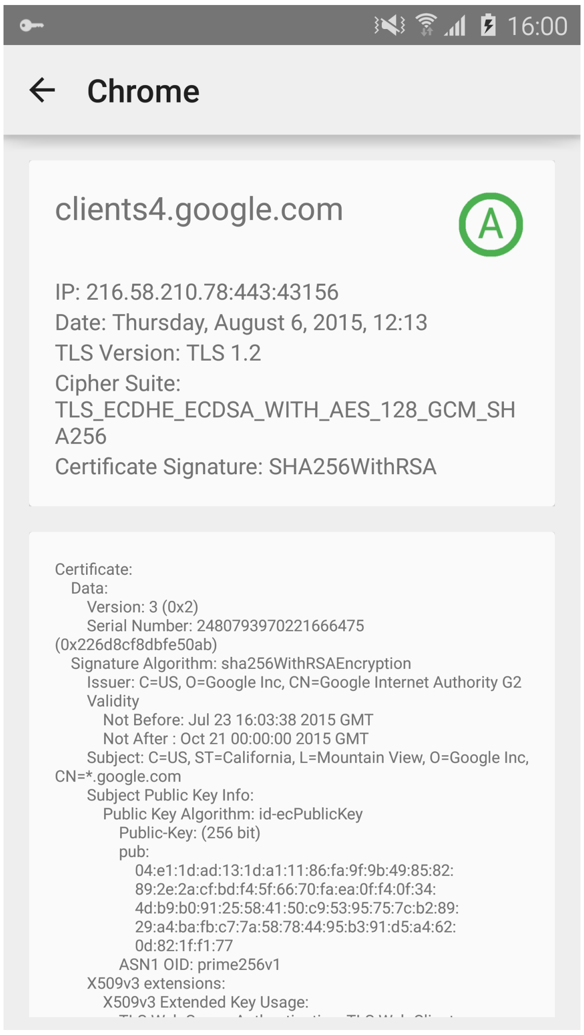
b.config.skype.com
Thursday, August 6, 2015, 12:10

(b)

Figure 4.7: App Pages

4.10 Summary

Sift was created to provide users awareness of the TLS connections their apps make, and to protect them from poorly coded, vulnerable apps. It operates as a local VPN, tunnelling all traffic through itself before passing it on to the network. It implements a simple interface making it easy for non-technical users, and can run in the background with minimal user interaction. Sift can be downloaded from the Google Play Store at this address: `https://play.google.com/store/apps/details?id=com.imhotepisinvisible.sslsift`. It proves to be effective, and a further examination of its operation can be found in Section 6.2.

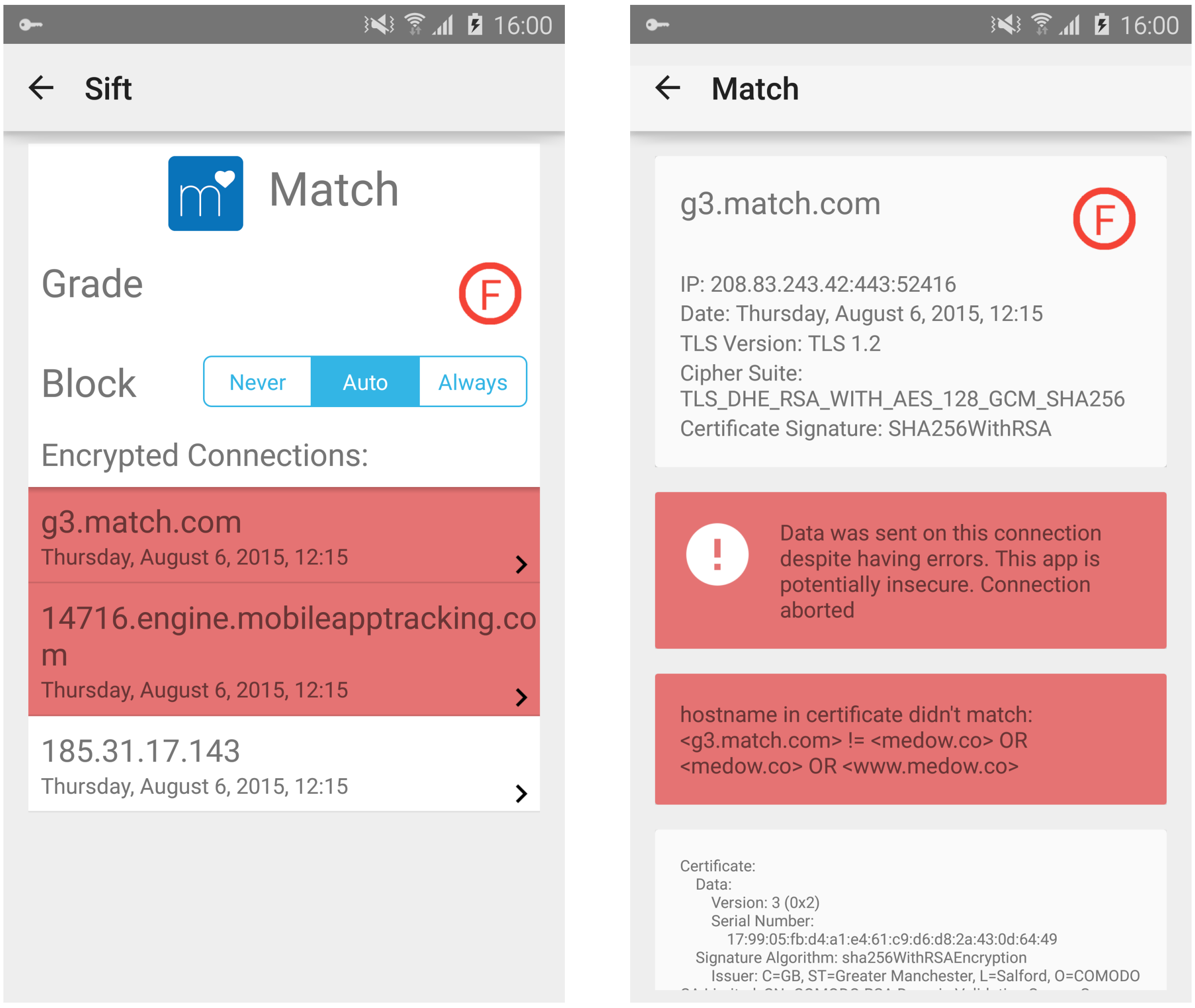


clients4.google.com

IP: 216.58.210.78:443:43156
Date: Thursday, August 6, 2015, 12:13
TLS Version: TLS 1.2
Cipher Suite:
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SH
A256
Certificate Signature: SHA256WithRSA

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 2480793970221666475
(0x226d8cf8dbfe50ab)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=Google Inc, CN=Google Internet Authority G2
        Validity
            Not Before: Jul 23 16:03:38 2015 GMT
            Not After : Oct 21 00:00:00 2015 GMT
        Subject: C=US, ST=California, L=Mountain View, O=Google Inc,
CN=*.google.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:e1:1d:ad:13:1d:a1:11:86:fa:9f:9b:49:85:82:
                    89:2e:2a:cf:bd:f4:5f:66:70:fa:ea:0f:f4:0f:34:
                    4d:b9:b0:91:25:58:41:50:c9:53:95:75:7c:b2:89:
                    29:a4:ba:fb:c7:7a:58:78:44:95:b3:91:d5:a4:62:
                    0d:82:1f:f1:77
                ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Extended Key Usage:
```

Figure 4.8: The Connection Screen


16:00

Sift

Match

Grade F

Block Never [✓ Auto] Always

Encrypted Connections:

g3.match.com
Thursday, August 6, 2015, 12:15

14716.engine.mobileapptracking.com
Thursday, August 6, 2015, 12:15

185.31.17.143
Thursday, August 6, 2015, 12:15

16:00

Match

g3.match.com F

IP: 208.83.243.42:443:52416
Date: Thursday, August 6, 2015, 12:15
TLS Version: TLS 1.2
Cipher Suite:
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
Certificate Signature: SHA256WithRSA

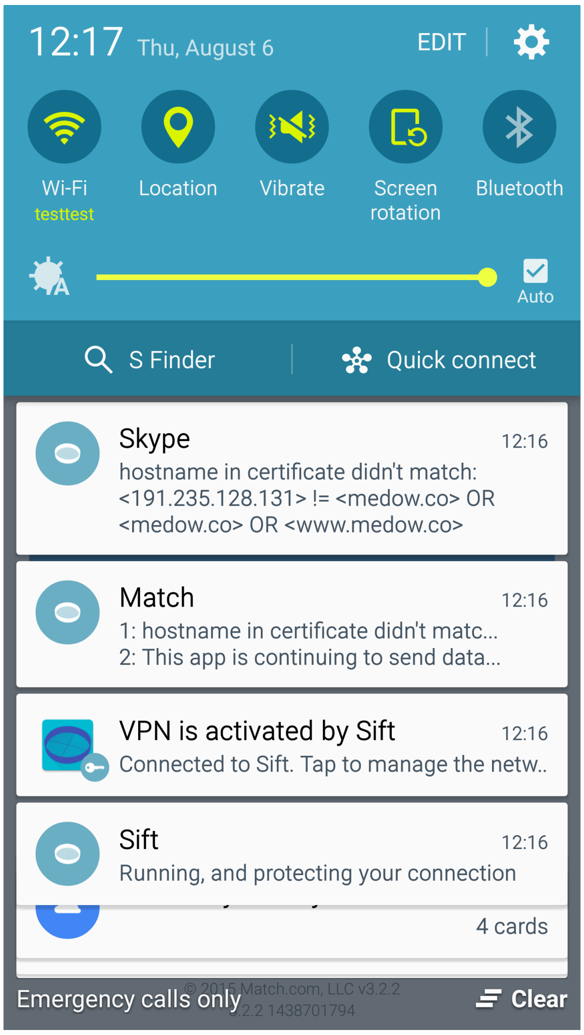
Data was sent on this connection despite having errors. This app is potentially insecure. Connection aborted

hostname in certificate didn't match: <g3.match.com> != <medow.co> OR <medow.co> OR <www.medow.co>

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            17:99:05:fb:d4:a1:e4:61:c9:d6:d8:2a:43:0d:64:49
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=GB, ST=Greater Manchester, L=Salford, O=COMODO
```

(a) (b)

Figure 4.9: A vulnerable app


12:17 Thu, August 6 EDIT

Skype 12:16
hostname in certificate didn't match: <191.235.128.131> != <medow.co> OR <medow.co> OR <www.medow.co>

Match 12:16
1: hostname in certificate didn't matc...
2: This app is continuing to send data...

VPN is activated by Sift 12:16
Connected to Sift. Tap to manage the netw..

Sift 12:16
Running, and protecting your connection

4 cards

Emergency calls only © 2015 Match.com, LLC v3.2.2 3.2.2 1438701794 Clear

Figure 4.10: Notifications

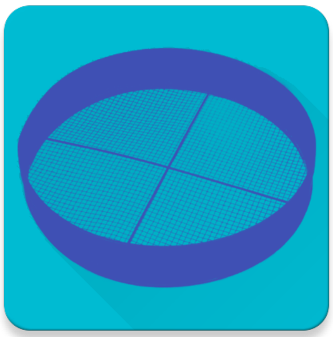


Figure 4.11: Sift's Logo

Chapter 5

Testing App Vulnerability

In order to both test Sift, and to perform a survey of encryption in the Android ecosystem, a testing suite needed to be created. This would sit between a phone and the internet and attempt to perform a man-in-the-middle attack against any TLS connections, presenting forged certificates and seeing if apps permitted connections to proceed. This system could also be used for packet capture, useful when surveying many apps in one go.

5.1 Man-in-the-Middle Techniques

In the real world, an attacker will generally use one of a small subset of techniques to force a victim's traffic through her system. When setting up a test system, one of these approaches must be chosen.

5.1.1 Malicious Hotspot

A malicious hotspot is simply a WiFi access point over which an attacker has complete control. This means that she can run any transparent proxy or other traffic manipulation software she wants, without the user being aware. It can take the form of a hacked router, a laptop with two network interfaces, or even a small system such as a Raspberry Pi. There is no control over access point naming, so given a convincing name such as 'Free WiFi' or 'BTWiFi' users can be convinced to connect. This can be especially problematic with mobile devices, as once a network has been connected to, devices will attempt to connect to any networks with the same name (or SSID). Thus, if a user has *ever* connected to a hotspot run by BT, their device will always try and connect to a hotspot called 'BTWiFi' if it is available.

Technical users may disable this feature, but some situations can prohibit this. For example, AT&T, one of the largest cell phone operators in the United States configures all iPhones sold to automatically connect to networks with the SSID 'attwifi' [82], and this behaviour is very hard to disable (and will be re-enabled every time an AT&T sim card is inserted into the phone). Therefore, in the US, an attacker merely needs to name an access point 'attwifi' and a large amount of traffic will automatically start flowing through their system.

5.1.2 ARP Spoofing

ARP spoofing takes advantage of the lack of security in the Address Resolution Protocol. This protocol is a basic component of networking, and is how devices identify themselves and others on a network, connecting physical (MAC) addresses to network (IP) addresses. Simply, a device broadcasts asking which device has a certain IP address. If a listening device indeed has that address it will reply with its MAC address. This communication occurs frequently on a network. Therefore, a device looking for the address of a local gateway (frequently a router) will send a message asking for the router's MAC address, and will receive a reply in turn. However, there is nothing stopping an attacker listening to these broadcasts and replying that in fact *they* are the router, and therefore to send all traffic through them. The reverse is done with the router, making it think that the attacker's address is the victim's address. This can easily be done with the `arpspoof` tool.[1] Now the attacker can view and manipulate the traffic between the victim and the router.

This can be mitigated somewhat, but most networks (especially public WiFi networks) take no precautionary measures. The ARP protocol was designed in 1982 [83] and security was not on the minds of the creators. While ARP remains a key element of networking, this attack will always be a viable option.

5.1.3 Other techniques

Other techniques also exist, such as DNS spoofing and DHCP spoofing, which allow the attacker to either direct traffic intended for one server to a different server of their choice or to confuse the victim into thinking that the gateway machine is the attacker's [84], but become increasingly difficult to achieve.

5.1.4 Decision

Since no deception is involved in this experiment, and running a hotspot provides the most control, that was the option chosen. An access point configured to perform man-in-the-middle attacks could be created with a certain SSID and a phone could be configured to connect to that network.

5.2 Setting up the Malicious Access Point

5.2.1 Hardware

The following hardware was used:

- Raspberry Pi Model B
- Kingston 8GB SD card
- Edimax EW7811Un WiFi USB adapter
- Ethernet cable

A Raspberry Pi was chosen both for its extremely cheap cost (approximately £25) and small size, allowing it to be used discreetly. Any laptop or desktop computer supported by Kali Linux can also easily be used. The key component to the set-up is having two network interfaces: one to create the malicious network, and one for Internet traffic. In this set-up Ethernet was used to connect to the Internet, but for portability a second WiFi adapter could be used, or even a 3G or 4G mobile connection.

5.2.2 Software

Kali Linux[2] is the operating system used. Kali is a Linux distribution based on Debian Linux, which is designed specifically for network monitoring, security auditing and penetration testing. It can run on x86 or ARM, among other architectures, making it suitable for the ARM-powered Raspberry Pi.

Additional software:

- TightVNCServer[3] and SSH are used to access the Raspberry Pi remotely.
- Dnsmasq[4] is used as a DNS cache and relay.
- Hostapd[5] is used to create the access point.

[1] `http://su2.info/doc/arpspoof.php`
[2] `https://www.kali.org/`
[3] `http://www.tightvnc.com/`
[4] `http://www.thekelleys.org.uk/dnsmasq/doc.html`
[5] `https://w1.fi/hostapd/`

- Mitmproxy[6] is used to create the transparent proxy that is used to attack the TLS connections.

5.2.3 Installation

The Kali Linux ARM distribution was downloaded and imaged onto an SD card. Once booted up it is easy to SSH into the Raspberry Pi to continue setting up. After expanding the OS partition to fill the SD card,[7]) the full distribution of Kali was installed:

```
apt-get install kali-linux-full
```

The additional software was then installed:

```
apt-get install tightvncserver dnsmasq hostapd
```

Configuration files were created for dnsmasq (Listing 5.1) and hostapd (Listing 5.2) and a signed certificate was downloaded to the system to enable the invalid hostname attack.

```
interface=wlan0
dhcp-range=10.0.0.10,10.0.0.250,12h
dhcp-option=3,10.0.0.1
dhcp-option=6,10.0.0.1
server=8.8.8.8
log-queries
log-dhcp
```

Listing 5.1: dnsmasq.conf

```
interface=wlan0
driver=nl80211
ssid=badguy
channel=1
```

Listing 5.2: hostapd.conf

5.2.4 Installation Issues

The version of hostapd in the Kali software repository is not compatible with the Edimax WiFi adapter used. An appropriate version can be downloaded from Realtek's website[8] and compiled.

The network-manager service gets in the way of the manual network configuration and must be stopped with

```
service network-manager stop
```

5.2.5 Running the Software

These commands assume that the Ethernet adapter is at eth0 and the WiFi adapter is at wlan0 and that the configuration files are stored in the working directory.

Bring up the WiFi adapter:

```
ifconfig wlan0 10.0.0.1/24 up
```

[6]https://mitmproxy.org/

[7]Following the instructions here:
https://linhost.info/2015/05/expand-the-root-partition-in-kali-linux-for-the-raspberry-pi/

[8]http://www.realtek.com.tw/downloads/downloadsView.aspx?Langid=1&PNid=21&PFid=48&Level=5&Conn=4&DownTypeID=3&GetDown=false

Start dnsmasq:

```
dnsmasq -C dnsmasq.conf -d
```

Start hostapd:

```
hostapd ./hostapd.conf
```

Enable IP forwarding:

```
sysctl -w net.ipv4.ip_forward=1
iptables -P FORWARD ACCEPT
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

At this point one can connect to the WiFi hotspot and it will simply pass on traffic to the Ethernet interface. Wireshark can be used to inspect the traffic flow.

Now mitmproxy can be started. It can be run in two different configurations depending on the attack to be run: presenting the user with a self-signed certificate, or presenting the user with a valid certificate signed for an invalid host.

First configure IPTables to route all TLS traffic to mitmproxy:

```
iptables -t nat -A PREROUTING -i wlan0 -p tcp --dport 443 -j REDIRECT --to
    -port 8080
```

Unencrypted HTTP traffic can also be routed through the proxy:

```
iptables -t nat -A PREROUTING -i wlan0 -p tcp --dport 80 -j REDIRECT --to-
    port 8080
```

And run mitmproxy with one of the two commands:

```
mitmproxy -T --host
mitmproxy -T --host --cert=cert.pem
```

The `-T` flag runs mitmproxy transparently (i.e. invisible to the user) and the `--host` flag helps with displaying the connections to the user. The `--cert` flag points to the location of a custom certificate. Mitmproxy will run on port 8080 by default.

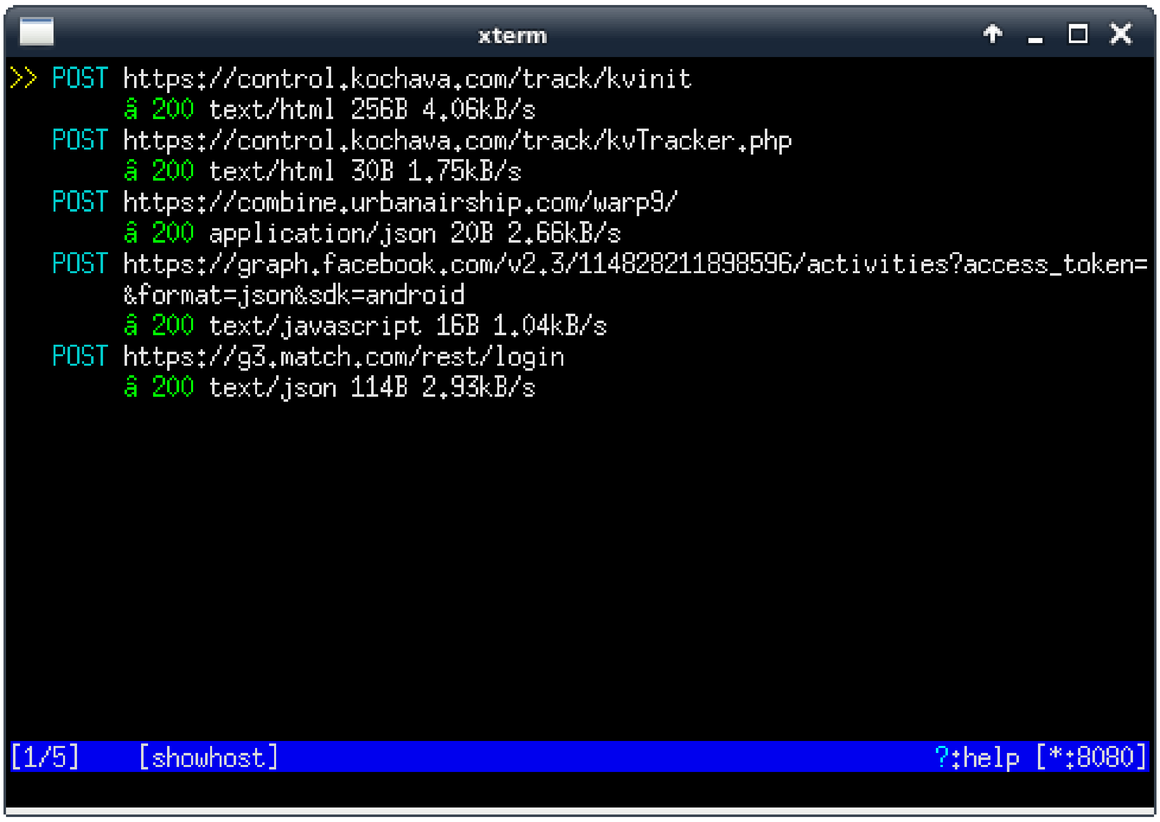
Everything is now set up and waiting for connections from an unwitting user (Figure 5.1).

5.3 Manually Testing Vulnerable Apps

Apps were tested by configuring an Android phone to connect to the attacking access point. `Mitmproxy` was configured to run, and the app was opened. Any available login box was filled in with a dummy username and password. If any traffic appeared in the `mitmproxy` window then the app was marked as vulnerable and further examination was conducted. Apps chosen for testing included those of financial institutions, large chains, and other apps with a high number of downloads on the Google Play Store.

5.4 Further Examination

When a vulnerable app is discovered, it may prove useful to examine it further to discover which error has been made when creating a TLS connection. Fahl et al. introduced the Mallodroid static analysis tool [31], which works with the AndroGuard decompiler to statically analyse Android applications. This can provide useful guidance as to where to look for vulnerabilities, but as is typical with static analysis, can miss certain issues, or highlight issues with code that is in fact never called.



```
xterm
>> POST https://control.kochava.com/track/kvinit
        â 200 text/html 256B 4.06kB/s
   POST https://control.kochava.com/track/kvTracker.php
        â 200 text/html 30B 1.75kB/s
   POST https://combine.urbanairship.com/warp9/
        â 200 application/json 20B 2.66kB/s
   POST https://graph.facebook.com/v2.3/114828211898596/activities?access_token=
        &format=json&sdk=android
        â 200 text/javascript 16B 1.04kB/s
   POST https://g3.match.com/rest/login
        â 200 text/json 114B 2.93kB/s

[1/5]    [showhost]                                        ?:help [*:8080]
```

Figure 5.1: List of captured https connections

Indeed, Dormann, when testing apps at CERT [85] highlights the same issue: the static analysis provides useful information, but cannot be relied on completely. Therefore, a small number of apps that have been found to be vulnerable can be manually decompiled and inspected, finding the exact causes of an issue.

The decompilation process will be described here. Specific results and analysis can be found in Section 6.1.

5.4.1 Static Analysis: Mallodroid

Running the Mallodroid tool is relatively straightforward. An installation of AndroGuard[9] (including all relevant dependencies) must be in place, and then the tool can be run:

```
./mallodroid.py -f /path/to/app.apk -x
```

Mallodroid will output XML highlighting any issues it has found.

5.4.2 Decompilation

Because Android apps are written in Java (although native code can be included for performance reasons), decompilation is a relatively straightforward task. There is not space for an exploration of the architecture of Android apps here, but it is sufficient to know that the Java code is compiled to Dalvik bytecode, which is stored in an APK file along with the app's resources. An APK is essentially a zip file, which can be opened and extracted. As one might expect, there are a number of tools designed to convert this Dalvik bytecode back into readable Java. Dex2Jar[10] takes an APK file or DEX file as input and outputs CLASS files zipped as a JAR file. This can be read with any number of Java decompilers, including JD[11] and Luyten[12] (which is a GUI frontend for the

[9]https://code.google.com/p/androguard/
[10]https://github.com/pxb1988/dex2jar
[11]http://jd.benow.ca/
[12]https://github.com/deathmarine/Luyten

Procyon decompiler). Because different Java decompilers perform differently it is useful to analyse the JAR files with a few of them.

Other tools designed specifically for Android can also be used. Bytecode Viewer[13] wraps all of the above tools into one package, making things more convenient. APKStudio[14] is also useful as it incorporates Smali, which allows editing of the Dalvik bytecode, which is needed when a decompiler is unable to process a particular method, or to edit and recompile applications (to patch out an app checking if a phone has root access or not, for example).

When a JAR file created with Dex2Jar has been loaded into one of these decompilers, a search can be performed looking for the specific API calls that we know are required when implementing TLS in apps: instantiation of an `X509TrustManager` or `HostnameVerifier`. Mallodroid can also be useful in directing us to the relevant classes. At this point it becomes easy to see what errors have been made, whether it is overriding a `TrustManager` to provide no protection at all, forgetting to include a `HostnameVerifier` with a custom `SocketFactory`, setting a `HostnameVerifier` to pass all hostnames, or allowing a `WebView` to ignore SSL errors.

5.4.2.1 Obfuscation

Some apps use the ProGuard tool to 'shrink, optimize and obfuscate' their code.[15] This renames all classes, fields and methods with short, obscure names, making understanding the decompiled Java code harder. This can provide an obstacle to inspecting apps, although any calls to the Android API cannot be obfuscated (as the operating system needs to understand the calls), so searches for `X509TrustManager` and so on will still succeed.

5.5 Automated Testing

As intimated in the previous section, manual testing and static analysis can only reveal so much. To get a broader view of the Android ecosystem and to find a range of apps with security vulnerabilities, widescale, automated, dynamic testing is required. A process was devised to download and test a large number of Android applications from the Google Play Store, with minimal user interaction required.

5.5.1 Finding the Apps

Before downloading the apps, criteria were needed to decide which ones to evaluate. Apps on the Google Play Store are divided into 45 categories, 20 for games and 25 for other applications (Listing 5.3). Ranking charts are produced, globally and by category, divided by free and paid apps, ordered by number of downloads, to help users decide which apps they may want to install. Since apps have a higher likelihood of containing sensitive user information, it was decided to focus on those, rather than games, and to analyse the top 100 of all free apps, and the top 100 free apps in each of the application categories, a total of 2500 apps.

```
BUSINESS
COMICS
COMMUNICATION
EDUCATION
ENTERTAINMENT
FINANCE
HEALTH_AND_FITNESS
LIBRARIES_AND_DEMO
LIFESTYLE
APP_WALLPAPER
MEDIA_AND_VIDEO
```

[13]https://github.com/Konloch/bytecode-viewer

[14]https://apkstudio.codeplex.com/

[15]http://developer.android.com/tools/help/proguard.html

```
MEDICAL
MUSIC_AND_AUDIO
NEWS_AND_MAGAZINES
PERSONALIZATION
PHOTOGRAPHY
PRODUCTIVITY
SHOPPING
SOCIAL
SPORTS
TOOLS
TRANSPORTATION
TRAVEL_AND_LOCAL
WEATHER
APP_WIDGETS
```

Listing 5.3: Google Play Store App Categories

Gaining programmatic access to this list of apps is not made easy by Google's Play Store interface, so `google-play-scraper`,[16] a Node.js[17] based scraper was used. A script was written in JavaScript to iterate over the relevant categories and generate a CSV file of the app names, their package identifier and a market link, producing a list of 2500 apps. The script and list of apps can be seen in Appendices B.1 and D.

5.5.2 Downloading the Apps

Downloading 2500 apps in one go is also not made straightforward by Google. There is no easy interface in the Play Store to download multiple apps, and all downloads are sent directly to a phone, which will not have enough storage for that number of apps in one go. Therefore, a tool called `Raccoon`[18] was used, which is able to take a list of Market URLs and download the APKs to a PC. It is a Java app that has both a GUI and a command line interface. Bulk download from a provided list can be initiated with the command:

```
java -jar raccoon-3.5.jar -a /path/to/configuration/ -u -i list.txt
```

5.5.3 Running the Apps

`Monkeyrunner`[19] is a tool included in the Android SDK that allows developers to create Python scripts to test their apps. They can configure certain button presses to be made, allowing them to determine if their apps are behaving in the desired manner.

Using this tool, a Python script was created to loop over the entire set of 2500 downloaded apps, extract the package and activity names (required by `monkeyrunner` to launch an app) and attempt to log in (if a log in page was encountered at the first screen). This script can be seen in Appendix B.2.

While this script was running, the phone was connected to the Raspberry Pi access point, which was configured to log all SSL and DNS traffic to a packet capture file with `tcpdump`:[20]

```
tcpdump -i wlan0 -w cap.pcap
```

Once execution was complete this app could be analysed to determine the various parameters apps used to connect to their backend servers.

In another pass of the `monkeyrunner` script, `mitmproxy` could be configured to run. Any connections that appeared in the log would be from a vulnerable app, and these apps could then be examined further.

[16] `https://github.com/facundoolano/google-play-scraper`
[17] `https://nodejs.org`
[18] `http://www.onyxbits.de/raccoon`
[19] `http://developer.android.com/tools/help/monkeyrunner_concepts.html`
[20] `http://www.tcpdump.org`

Testing 2500 apps takes about 12 hours, and runs without any user interaction required. This allows multiple tests to be run and enables useful analysis. The results of this process can be seen in Section 6.3.

5.6 Summary

A system was developed to perform a man-in-the-middle attack against an Android phone. Apps were then tested manually and automatically, with both static and dynamic analysis. Apps that seemed vulnerable were inspected further by decompiling and manually inspecting them. This revealed a wide range of apps that either do not use the most up-to-date version of TLS available, do not use sufficiently secure cipher suits, or in the worst cases, do not offer users any protection against a man-in-the-middle eavesdropper.

Chapter 6

Results and Evaluation

The testing detailed in the previous section revealed several apps that fail to properly secure communication between the client and the server. This section will begin with case studies of several of these apps, investigating the ways in which security is broken. It will continue by looking at the performance of Sift when running these apps, followed by an examination of the Android ecosystem by looking at the results of automated testing. It will conclude with recommendations for the Android operating system and app developers to help ensure proper TLS security in the future.

6.1 Case Studies

6.1.1 Online Shopping App

An app developed by a major online retailer was identified.[1] It has been downloaded from the Play Store between 100,000 and 500,000 times. The app allows customers to purchase goods, and so has the ability to store and transmit payment information. On testing, it were found to leak user information when attacked with a correctly signed certificate with an invalid hostname (Figure 6.1).

6.1.1.1 The Cause

Without access to the full source code for the app the cause cannot be determined with 100% accuracy, but using the decompilation techniques detailed in Section 5.4.2, an estimation can be made.

A custom `SSLSocketFactory` is defined in the `AdditionalKeyStoresSSLSocketFactory` class, allowing a custom keystore to be used. This keystore is intended to hold certificates not otherwise available to the app through the Android operating system.

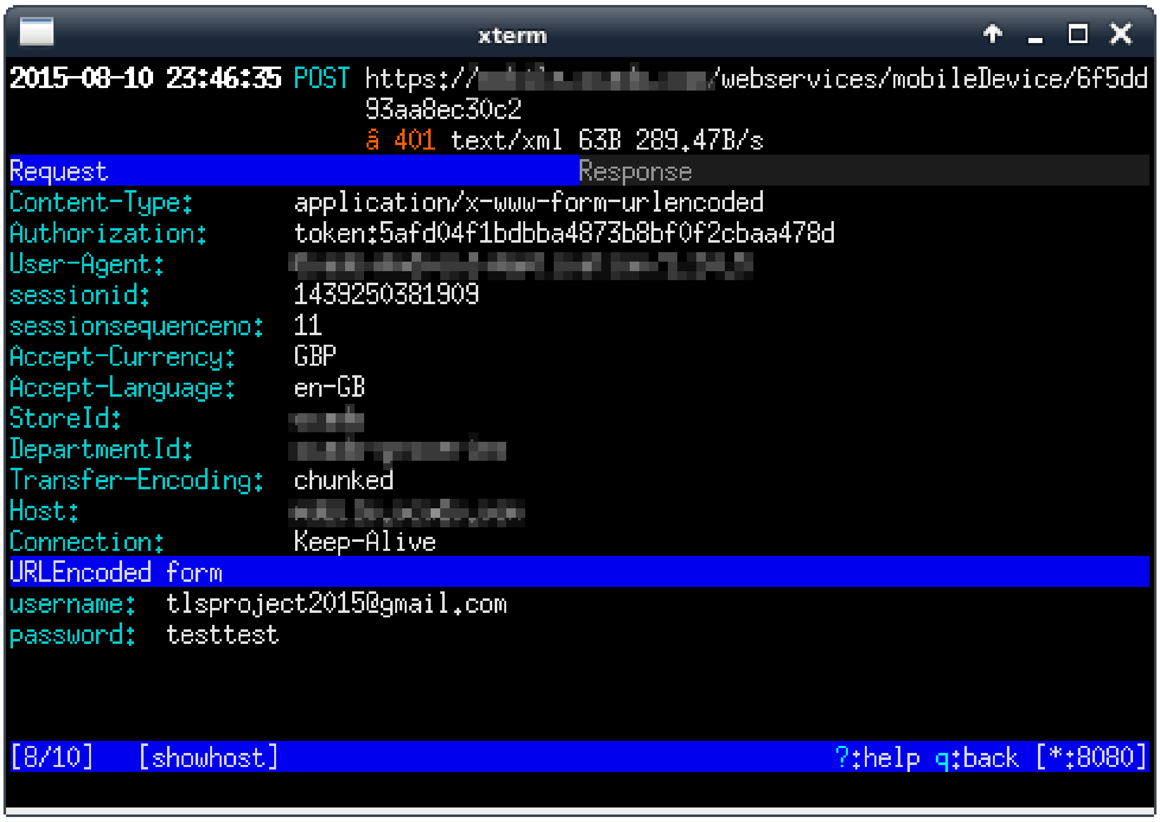
The purpose for this in these apps is not clear, as the keystore (stored in the `res/raw/-keystore.bks` file with the password 'hel10s') only contains a Verisign root and intermediate certificate. The root certificate is already present in both a Samsung Galaxy S6 and a Google Nexus 4, and the intermediate certificate can be provided by the server, but perhaps this is needed to support older Android devices (although in fact the server the app connects to present certificates signed by different roots).

In any case, the code in the app seems to follow an answer available from the StackOverflow website.[2] As noted in the comments, this code accidentally fails to verify the hostname of the presented certificate.

In addition, `activity.views.WebViewStack` implements a WebView. It contains the following code:

[1]We have been asked to withhold the name of this app to allow time for users to update to the latest secured version.

[2]`http://stackoverflow.com/a/6378872`



```
2015-08-10 23:46:35 POST https://[redacted]/webservices/mobileDevice/6f5dd
                         93aa8ec30c2
                         â 401 text/xml 63B 289.47B/s
Request                                  Response
Content-Type:          application/x-www-form-urlencoded
Authorization:         token:5afd04f1bdbba4873b8bf0f2cbaa478d
User-Agent:            [redacted]
sessionid:             1439250381909
sessionsequenceno:     11
Accept-Currency:       GBP
Accept-Language:       en-GB
StoreId:               [redacted]
DepartmentId:          [redacted]
Transfer-Encoding:     chunked
Host:                  [redacted]
Connection:            Keep-Alive
URLEncoded form
username:  tlsproject2015@gmail.com
password:  testtest



[8/10]   [showhost]                                    ?:help q:back [*:8080]
```

Figure 6.1: Leaked information from the online shopping app

```
public void onReceivedSslError(final WebView webView, final
    SslErrorHandler sslErrorHandler, final SslError sslError) {
      sslErrorHandler.proceed();
}
```

This instructs the app to continue with a request when an SSL error is received, rather than displaying an error.

6.1.1.2 The Solution

The solution that would require the fewest changes would be to make sure a `HostnameValidator` is instantiated (as recommended by Google in their documentation.[3]) Alternatively, `Additional-KeyStoresSSLSocketFactory` could extend `SSLCertificateSocketFactory`, Google's implementation of `SSLSocketFactory` which provides hostname verification, or hostname verification could be directly coded into the `AdditionalKeyStoresSSLSocketFactory` class.

Of course, if using a custom keystore is no longer necessary, this class could be removed completely and the standard API could be used.

One could also go in the opposite direction and implement certificate pinning, which would mean that the app would only trust a specific certificate, providing protection from malicious trusted roots.

The fix for the WebView is simple. The code should instead read

```
public void onReceivedSslError(final WebView webView, final
    SslErrorHandler sslErrorHandler, final SslError sslError) {
      sslErrorHandler.cancel();
}
```

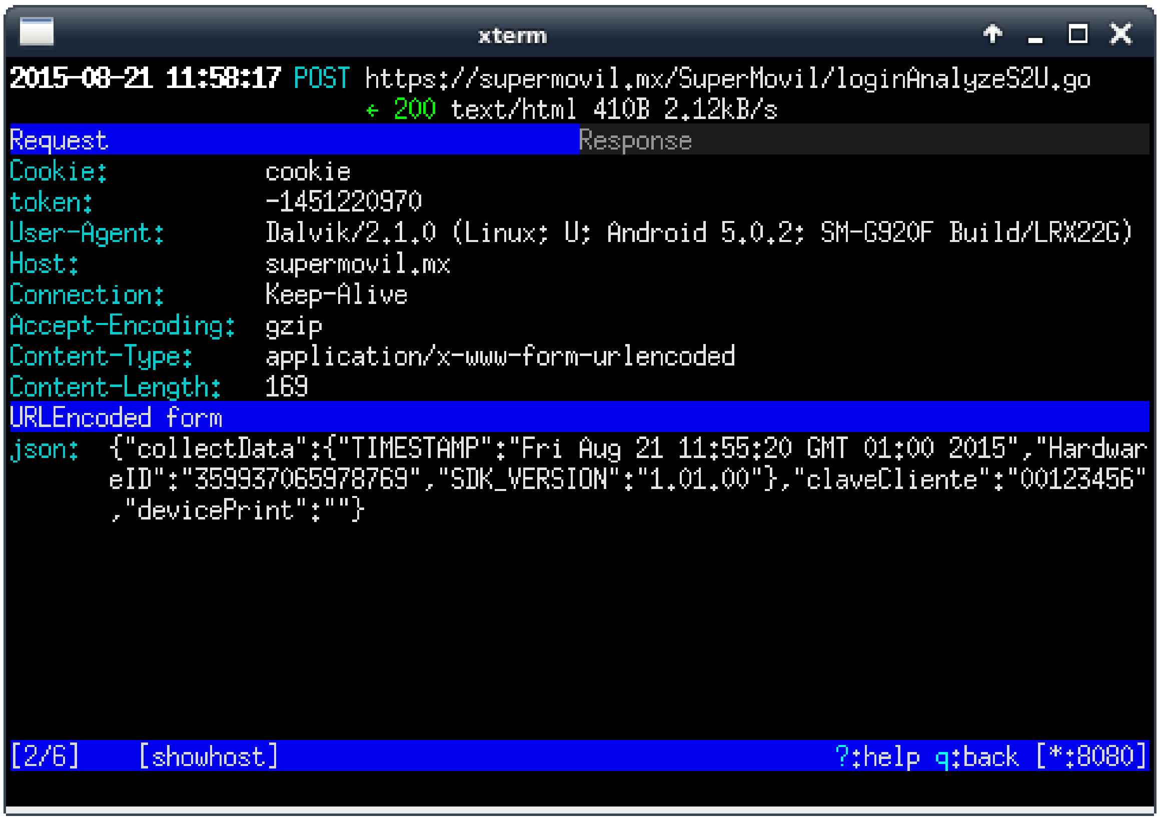
[3]https://developer.android.com/training/articles/security-ssl.html#WarningsSslSocket

6.1.1.3 Resolution

This issue was reported to the retailer on August 12 2015. They were very receptive to the report and were able to release a fixed version of the app on August 14 2015. The custom `SocketFactory` and keystore were removed and the WebView code was corrected.

6.1.2 Santander Mexico

Santander is one of the largest banks in the world. Their Mexican division has an app called SuperMóvil[4] which has been downloaded between 500,000 and 1,000,000 times on the Google Play Store. The app can be used for mobile banking, and so the information it transmits is highly confidential. Unfortunately, it fails to properly verify hostnames, and so easily falls victim to a man-in-the-middle attack (Figure 6.2). The versions tested were 2.5.10, 2.7 and 2.8.3, the latest currently available at the time of writing.



```
2015-08-21 11:58:17 POST https://supermovil.mx/SuperMovil/loginAnalyzeS2U.go
                                ← 200 text/html 410B 2.12kB/s
Request                                  Response
Cookie:             cookie
token:              -1451220970
User-Agent:         Dalvik/2.1.0 (Linux; U; Android 5.0.2; SM-G920F Build/LRX22G)
Host:               supermovil.mx
Connection:         Keep-Alive
Accept-Encoding:    gzip
Content-Type:       application/x-www-form-urlencoded
Content-Length:     169
URLEncoded form
json:  {"collectData":{"TIMESTAMP":"Fri Aug 21 11:55:20 GMT 01:00 2015","Hardwar
       eID":"359937065978769","SDK_VERSION":"1.01.00"},"claveCliente":"00123456"
       ,"devicePrint":""}

[2/6]    [showhost]                                     ?:help q:back [*:8080]
```

Figure 6.2: Leaked information from the Santander Mexico app

6.1.2.1 The Cause

The output from Mallodroid (Listing 6.1) reveals a series of errors.

```
<result package="mx.bancosantander.supermovil">
        <trustmanagers>
                <trustmanager broken="Maybe" class="com.twincoders.
                    twinpush.sdk.communications.security.
                    TwinPushTrustManager">
                         <xref class="com.twincoders.twinpush.sdk.
                            communications.DefaultRequestLauncher" method="
                            executeRequest"/>
                </trustmanager>
```

[4]https://play.google.com/store/apps/details?id=mx.bancosantander.supermovil

```
                <trustmanager broken="True" class="mx.bancosantander.
                   realidadaumentada.ContextoRealidad$5">
                        <xref class="mx.bancosantander.realidadaumentada.
                           ContextoRealidad" method="getHttpGETInputStream
                           "/>
                </trustmanager>
                <trustmanager broken="True" class="mx.bancosantander.
                   supermovil.comunicaciones.HttpsJSONRequest$1">
                        <xref class="mx.bancosantander.supermovil.
                           comunicaciones.HttpsJSONRequest" method="&lt;
                           init&gt;"/>
                </trustmanager>
                <trustmanager broken="True" class="mx.bancosantander.
                   supermovil.util.Utils$2">
                        <xref class="mx.bancosantander.supermovil.util.
                           Utils" method="ObtenerImagen"/>
                </trustmanager>
                <insecuresslsocket/>
        </trustmanagers>
        <hostnameverifiers>
                <hostnameverifier broken="True" class="mx.bancosantander.
                   realidadaumentada.ContextoRealidad$4">
                        <xref class="mx.bancosantander.realidadaumentada.
                           ContextoRealidad" method="getHttpGETInputStream
                           "/>
                </hostnameverifier>
                <hostnameverifier broken="True" class="mx.bancosantander.
                   supermovil.comunicaciones.HttpsJSONRequest$2">
                        <xref class="mx.bancosantander.supermovil.
                           comunicaciones.HttpsJSONRequest" method="&lt;
                           init&gt;"/>
                </hostnameverifier>
                <hostnameverifier broken="True" class="mx.bancosantander.
                   supermovil.util.Utils$3">
                        <xref class="mx.bancosantander.supermovil.util.
                           Utils" method="ObtenerImagen"/>
                </hostnameverifier>
                <allowhostnames/>
        </hostnameverifiers>
        <onreceivedsslerrors/>
</result>
```

Listing 6.1: Santander Mexico Mallodroid Output

Further inspection shows completely empty TrustManagers and HostnameVerifiers (Listing 6.2 shows just one, from `mx.bancosantander.realidadaumentada.ContextoRealidad`), a textbook example of what not to do. For a major financial institution, this is a rather shocking result.

```
    if (paramString.startsWith("https://"))
    {
      HttpsURLConnection.setDefaultHostnameVerifier(new
         HostnameVerifier()
      {
        public boolean verify(String paramAnonymousString,
           SSLSession paramAnonymousSSLSession)
        {
          return true;
        }
      });
      SSLContext localSSLContext = SSLContext.getInstance("TLS");
```

```
        X509TrustManager local5 = new X509TrustManager()
        {
          public void checkClientTrusted(X509Certificate[]
             paramAnonymousArrayOfX509Certificate, String
             paramAnonymousString)
            throws CertificateException
          {}

          public void checkServerTrusted(X509Certificate[]
             paramAnonymousArrayOfX509Certificate, String
             paramAnonymousString)
            throws CertificateException
          {}

          public X509Certificate[] getAcceptedIssuers()
          {
            return new X509Certificate[0];
          }
        };
        SecureRandom localSecureRandom = new SecureRandom();
        localSSLContext.init(null, new X509TrustManager[] { local5 },
           localSecureRandom);
        HttpsURLConnection.setDefaultSSLSocketFactory(localSSLContext.
           getSocketFactory());
      }
```

Listing 6.2: Insecure Code from the Santander Mexico App

6.1.2.2 The Solution

Unlike the online shopping app, this is not a complicated error. The app simply needs to correctly implement TrustManagers and HostnameVerifiers, which can be easily done using the standard Android API.

6.1.2.3 Resolution

The issue was reported to Santander Mexico on August 17 2015. At the time of writing no reply has been received.

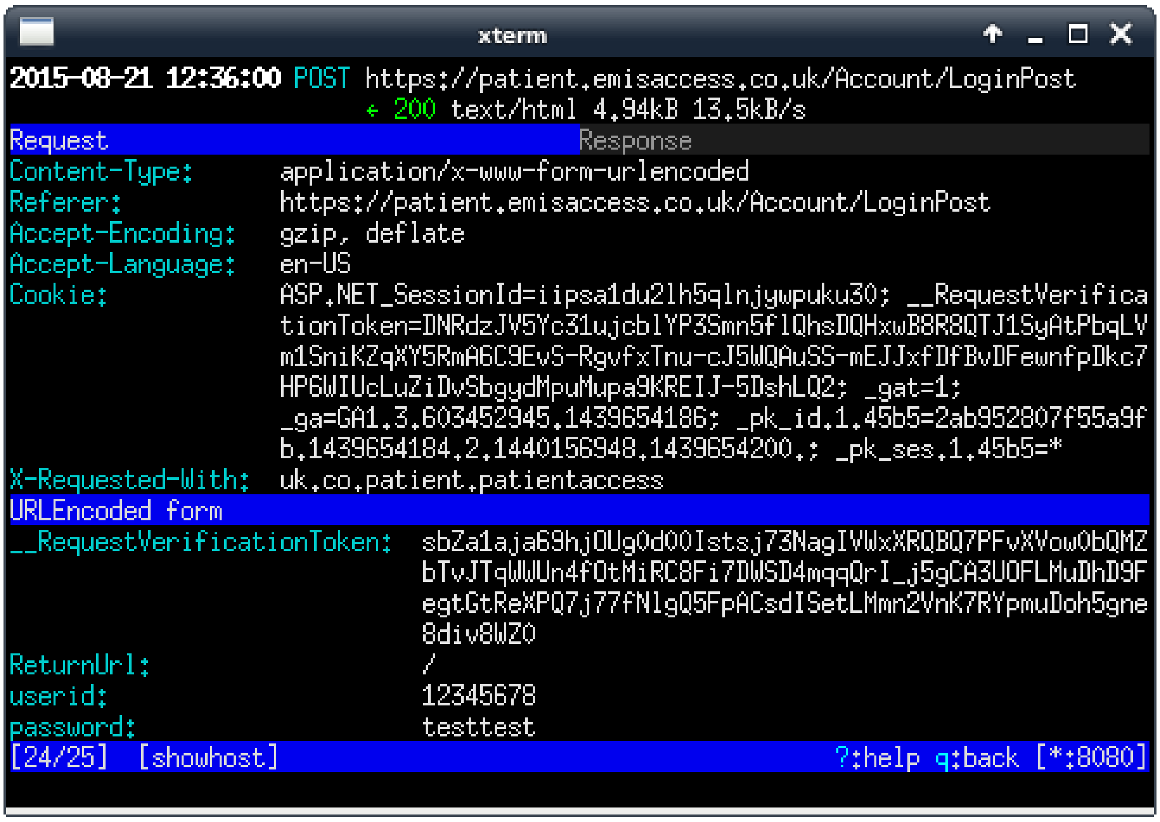
6.1.3 Patient Access

Patient Access[5] is an app that lets patients access local GP services, including viewing medical records and booking appointments. It has been downloaded between 100,000 and 500,000 times on the Google Play Store. One would think security would be a priority when viewing medical records are an issue, and failing to keep such information private is a serious offence under the UK's Data Protection Act 1998. However, login information is easily visible when a man-in-the-middle attack is conducted (Figure 6.3). The version tested was 1.1, the latest currently available at the time of writing.

6.1.3.1 The Cause

The app implements a WebView, and fails to deal with any SSL errors, instead instructing the app to proceed if any issue is encountered (Listing 6.3).

[5]`https://play.google.com/store/apps/details?id=uk.co.patient.patientaccess`



```
xterm
2015-08-21 12:36:00 POST https://patient.emisaccess.co.uk/Account/LoginPost
                         ← 200 text/html 4.94kB 13.5kB/s
Request                                   Response
Content-Type:        application/x-www-form-urlencoded
Referer:             https://patient.emisaccess.co.uk/Account/LoginPost
Accept-Encoding:     gzip, deflate
Accept-Language:     en-US
Cookie:              ASP.NET_SessionId=iipsa1du2lh5qlnjywpuku30; __RequestVerifica
                     tionToken=DNRdzJV5Yc31ujcblYP3Smn5flQhsDQHxwB8R8QTJ1SyAtPbqLV
                     m1SniKZqXY5RmA6C9EvS-RgvfxTnu-cJ5WQAuSS-mEJJxfDfBvDFewnfpDkc7
                     HP6WIUcLuZiDvSbgydMpuMupa9KREIJ-5DshLQ2; _gat=1;
                     _ga=GA1.3.603452945.1439654186; _pk_id.1.45b5=2ab952807f55a9f
                     b.1439654184.2.1440156948.1439654200.; _pk_ses.1.45b5=*
X-Requested-With:    uk.co.patient.patientaccess
URLEncoded form
__RequestVerificationToken:  sbZa1aja69hjOUg0d00Istsj73NagIVWxXRQBQ7PFvXVow0bQMZ
                             bTvJTqWWUn4f0tMiRC8Fi7DWSD4mqqQrI_j5gCA3UOFLMuDhD9F
                             egtGtReXPQ7j77fNlgQ5FpACsdISetLMmn2VnK7RYpmuDoh5gne
                             8div8WZ0
ReturnUrl:                   /
userid:                      12345678
password:                    testtest
[24/25]  [showhost]                                   ?:help q:back [*:8080]
```

Figure 6.3: Leaked information from the Patient Access app

```java
public void onReceivedSslError(WebView paramWebView,
    SslErrorHandler paramSslErrorHandler, SslError paramSslError)
{
  paramSslErrorHandler.proceed();
}
```

Listing 6.3: Insecure code from the Patient Access app

6.1.3.2 The Solution

Again, this is a very simple fix. The `proceed()` method call can simply be replaced with `cancel()` and the app will refuse to initiate a connection when faced with an invalid certificate.

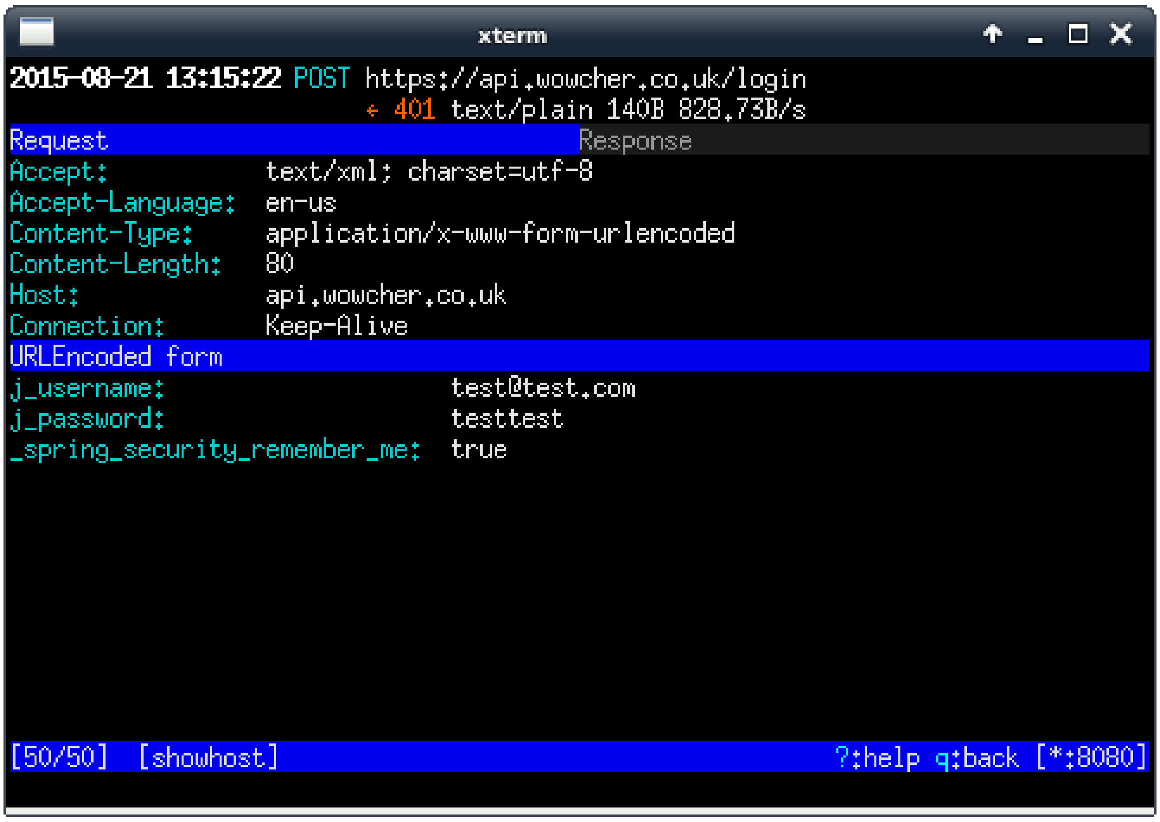
6.1.3.3 Resolution

The issue was reported to EMIS Health on August 17 2015. At the time of writing no reply has been received.

6.1.4 Wowcher

Wowcher, owned by Associated Newspapers Ltd., is an app that allows users to purchase discounts to restaurants, shops, and more.[6] It takes payment information, and so data security is a top priority. It has been downloaded between 1,000,000 and 5,000,000 times on the Play Store. Unfortunately, it is susceptible to a man-in-the-middle attack (Figure 6.4). The version tested was 3.4.1, the latest currently available at the time of writing.

[6] `https://play.google.com/store/apps/details?id=com.anmedia.wowcher.ui`



```
xterm
2015-08-21 13:15:22 POST https://api.wowcher.co.uk/login
                          ← 401 text/plain 140B 828.73B/s
Request                                  Response
Accept:            text/xml; charset=utf-8
Accept-Language:   en-us
Content-Type:      application/x-www-form-urlencoded
Content-Length:    80
Host:              api.wowcher.co.uk
Connection:        Keep-Alive
URLEncoded form
j_username:                    test@test.com
j_password:                    testtest
_spring_security_remember_me:  true

[50/50]  [showhost]                          ?:help q:back [*:8080]
```

Figure 6.4: Leaked information from the Wowcher app

6.1.4.1 The Cause

This app makes a different error. It manages to set up a Trust Manager correctly, but when instantiating a `HostnameVerifier` uses the code in Listing 6.4. This explicitly tells the app to allow any hostname encountered in a certificate, and essentially disables security.

```
((SSLSocketFactory)localObject).setHostnameVerifier(SSLSocketFactory
    .ALLOW_ALL_HOSTNAME_VERIFIER);
```

Listing 6.4: Insecure code from the Wowcher app

6.1.4.2 The Solution

This is very easy to fix. A functioning `HostnameVerifier` must be used.

6.1.4.3 Resolution

The issue was reported to Wowcher on August 17 2015. At the time of writing no reply has been received.

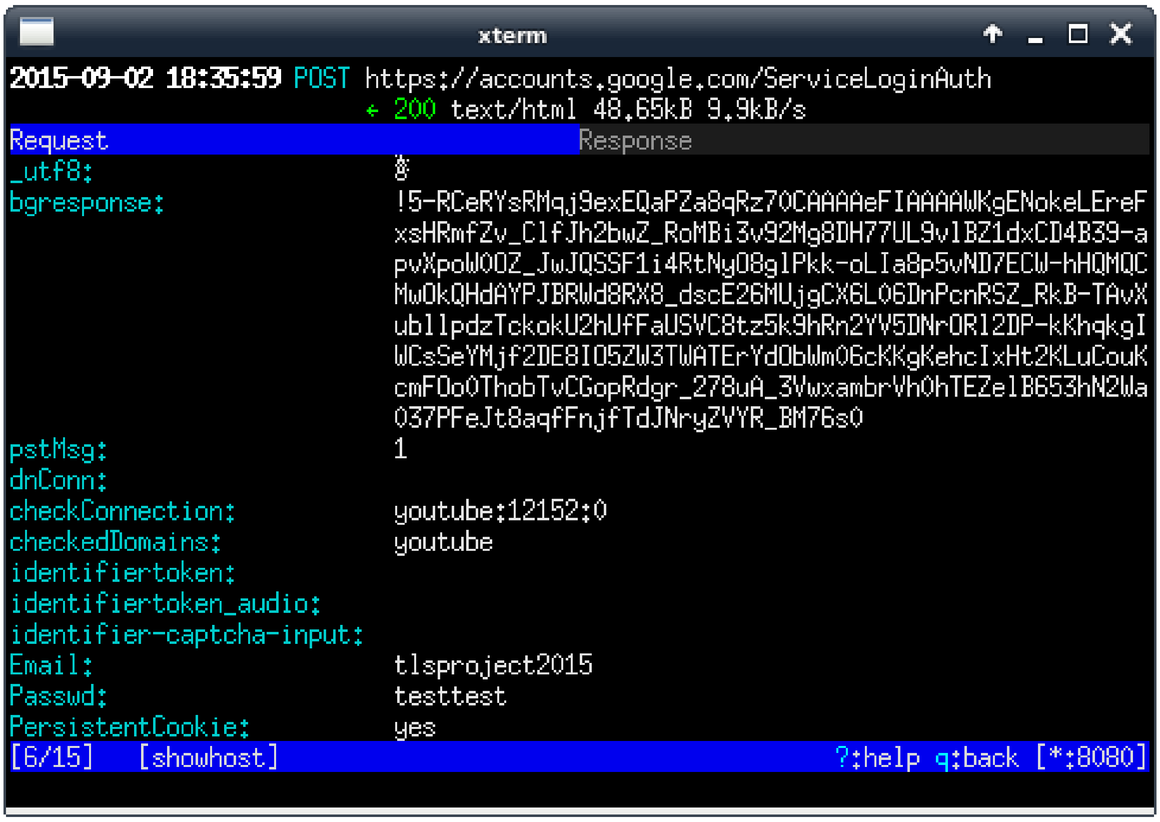
6.1.5 Other Apps

These four apps all leak user information, but the errors they make are all different. The online shopping app fails to instantiate a `HostnameVerifier`, Santander uses a custom, non-existent Trust Manager and `HostnameVerifier`, Patient Access does not protect its WebView, and Wowcher uses a useless `HostnameVerifier`. These errors can be found repeatedly in many more apps. A brief selection is detailed in Table 6.1.

One app it might be worth highlighting here is gReader. gReader is a news app that lets users read news from multiple sources in an easy way.[7] It has been downloaded between 1,000,000 and

[7] `https://play.google.com/store/apps/details?id=com.noinnion.android.greader.reader`

5,000,000 times on the Play Store. One might consider losing credentials to a news app worrisome, but not that important. However, users don't use a gReader account to authenticate: instead they are able to sign in with an account from one of Google, Facebook, Twitter, Microsoft or Evernote. As Figure 6.5 shows, this results in the compromise of these account details. This is potentially catastrophic, because an attacker can leverage a user's Google account to compromise many other accounts: they just need to send password reset emails to the user's GMail address. This highlights the dangers in the increasing push for using one set of credentials to sign into accounts all over the Internet.



```
2015-09-02 18:35:59 POST https://accounts.google.com/ServiceLoginAuth
                         ← 200 text/html 48.65kB 9.9kB/s
Request                                    Response
_utf8:                       ☃
bgresponse:                  !5-RCeRYsRMqj9exEQaPZa8qRz70CAAAAeFIAAAAWKgENokeLEreF
                             xsHRmfZv_ClfJh2bwZ_RoMBi3v92Mg8DH77UL9v1BZ1dxCD4B39-a
                             pvXpoW00Z_JwJQSSF1i4RtNy08g1Pkk-oLIa8p5vND7ECW-hHQMQC
                             MwOkQHdAYPJBRWd8RX8_dscE26MUjgCX6L06DnPcnRSZ_RkB-TAvX
                             ubllpdzTckokU2hUfFaUSVC8tz5k9hRn2YV5DNrOR12DP-kKhqkgI
                             WCsSeYMjf2DE8I05ZW3TWATErYdObWm06cKKgKehcIxHt2KLuCouK
                             cmFOoOThobTvCGopRdgr_278uA_3VwxambrVhOhTEZelB653hN2Wa
                             037PFeJt8aqfFnjfTdJNryZVYR_BM76s0
pstMsg:                      1
dnConn:
checkConnection:             youtube:12152:0
checkedDomains:              youtube
identifiertoken:
identifiertoken_audio:
identifier-captcha-input:
Email:                       tlsproject2015
Passwd:                      testtest
PersistentCookie:            yes
[6/15]   [showhost]                                    ?:help q:back [*:8080]
```

Figure 6.5: Leaked information from the gReader app

All of these vulnerabilities are still current as of the time of writing, even as updated versions have been released that claim to have 'bug fixes'. The vulnerabilities were reported on August 17 2015 but at the time of writing no reply has been received from any developer.

6.1.6 Safe Apps

Of course, this analysis highlights apps that fail to protect users' information. However, the majority of apps do get the TLS implementation right, and refuse to establish a connection when under attack. Those written by the large technology companies such as Google, Twitter and Facebook perform the best, but many other apps do behave properly. However, even in these cases, the error message displayed to a user is not clear. An error message may not be displayed, with the app just refusing to load (as is the case with the GMail app), or if there is an error message it is often a generic one complaining about the lack of internet connectivity, as in Figure 6.6, which may leave users confused; they are after all connected to the internet.

6.2 Sift

The vulnerable apps were tested using the same Raspberry Pi access point, but with the Sift app running. The app performed well, blocking connections in every case, and displaying the reason why. It also became easy to tell that the phone was connected to a malicious access point, because

App Name	Number of Downloads	Description	Version	Vulnerability
Match.com US	5,000,000 - 10,000,000	Online dating	3.2.0, 3.2.2, 3.2.3	TrustManager and Hostname
gReader (Google and Facebook login)	1,000,000 - 5,000,000	News reader	4.1.2, 4.2.0	TrustManager and Hostname
State Bank Anywhere (India)	1,000,000 - 5,000,000	Internet Banking	4.1.1, 4.1.2	TrustManager and Hostname
Pizza Hut US	5,000,000 - 10,000,000	Food ordering	2.0	TrustManager and Hostname
Last.fm	1,000,000 - 5,000,000	Music tracking	2.0.0.3	TrustManager and Hostname
Airelive	100,000 - 500,000	Video chat	1.2.3	TrustManager and Hostname
MyTube (Google login)	500,000 - 1,000,000	Playlist Creator	2.06	TrustManager and Hostname

Table 6.1: More Vulnerable Apps

a notification was generated for every app that attempted to make a TLS connection, whether it was vulnerable or not. The app thus performs two uses: it protects a user from poorly coded apps, and alerts the user that the access point they are connected to is not secure and they should consider disconnecting. Figures 4.9 and 4.10 show the app responding to this situation.

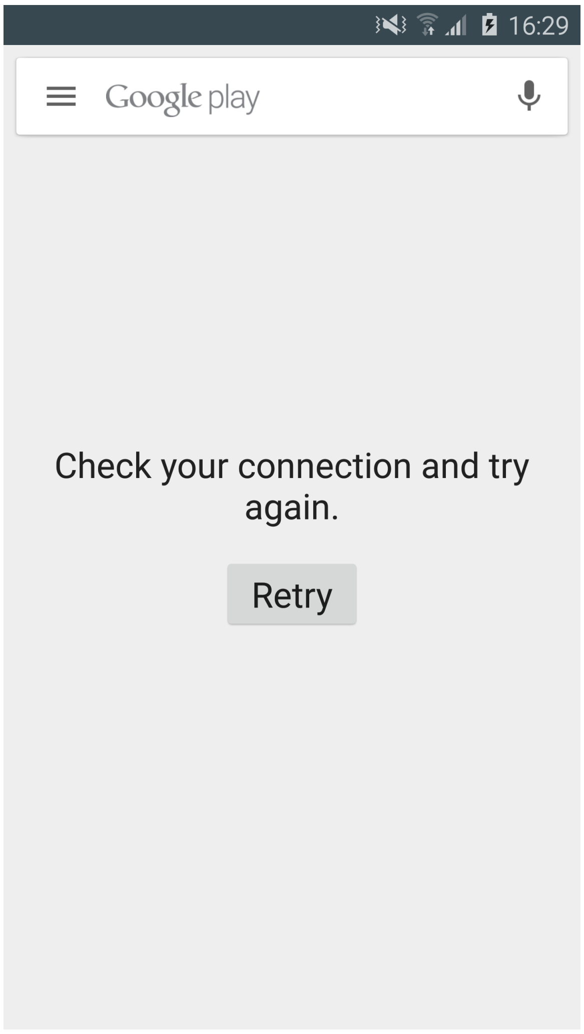
The app also did well at identifying apps that were not using the latest verion of TLS, or up-to-date cipher suites. For some reason, the Skype app[8] consistently connected to `pipe.skype.com` using TLS 1.0 even though the server supports TLS 1.2. It also warns that this server presents a SHA-1 certificate, which is dangerously weak [19]. The app also highlighted apps that used self-signed certificates, and those that had custom hostname verification algorithms.

The Netflix app[9] makes connections to several domains, including `ichnaea.netflix.com`, `api-global.netflix.com` and `android.nccp.netflix.com`. The first two of these domains use certificates signed by Verisign, however for some reason, `android.nccp.netflix.com` uses a self-signed certificate, and causes Sift to create a notification (the custom Netflix certificate authority is stored in `res/raw/ca.bks` with the password 'spyder' as well as `assets/ca.pem`). This is implemented in the classes `com.netflix.mediaclient.nccp.NccpKeyStore` and `com.netflix.mediaclient.nccp.NccpRequestTask`. `NccpRequestTask` instantiates an `SSLSocketFactory` using only the custom keystore, and then permits all hostnames to be used (Listing 6.5).

```java
private HttpClient getNewHttpClient()
{
  try
  {
    Object localObject = new SSLSocketFactory(NccpKeyStore.
       getInstance());
    ((SSLSocketFactory)localObject).setHostnameVerifier(
       SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
    ...
  }
  ...
}
```

Listing 6.5: Extract from Netflix's custom KeyStore

[8]https://play.google.com/store/apps/details?id=com.skype.raider
[9]https://play.google.com/store/apps/details?id=com.netflix.mediaclient



Check your connection and try again.

Retry

Figure 6.6: An incorrect error from the Google Play Store app when under a man-in-the-middle-attack.

Even though all hostnames are permitted, security is provided by the custom keystore: only Netflix is able to sign certificates with the provided CA, and so an attacker is unlikely to have a certificate that will allow her to decode the traffic. This is not a usual arrangement and the Sift app highlights this. An SSLLabs scan of the domain (Figure 6.7) also reveals vulnerability to the POODLE attack, vulnerability to denial of service attacks, and other issues.

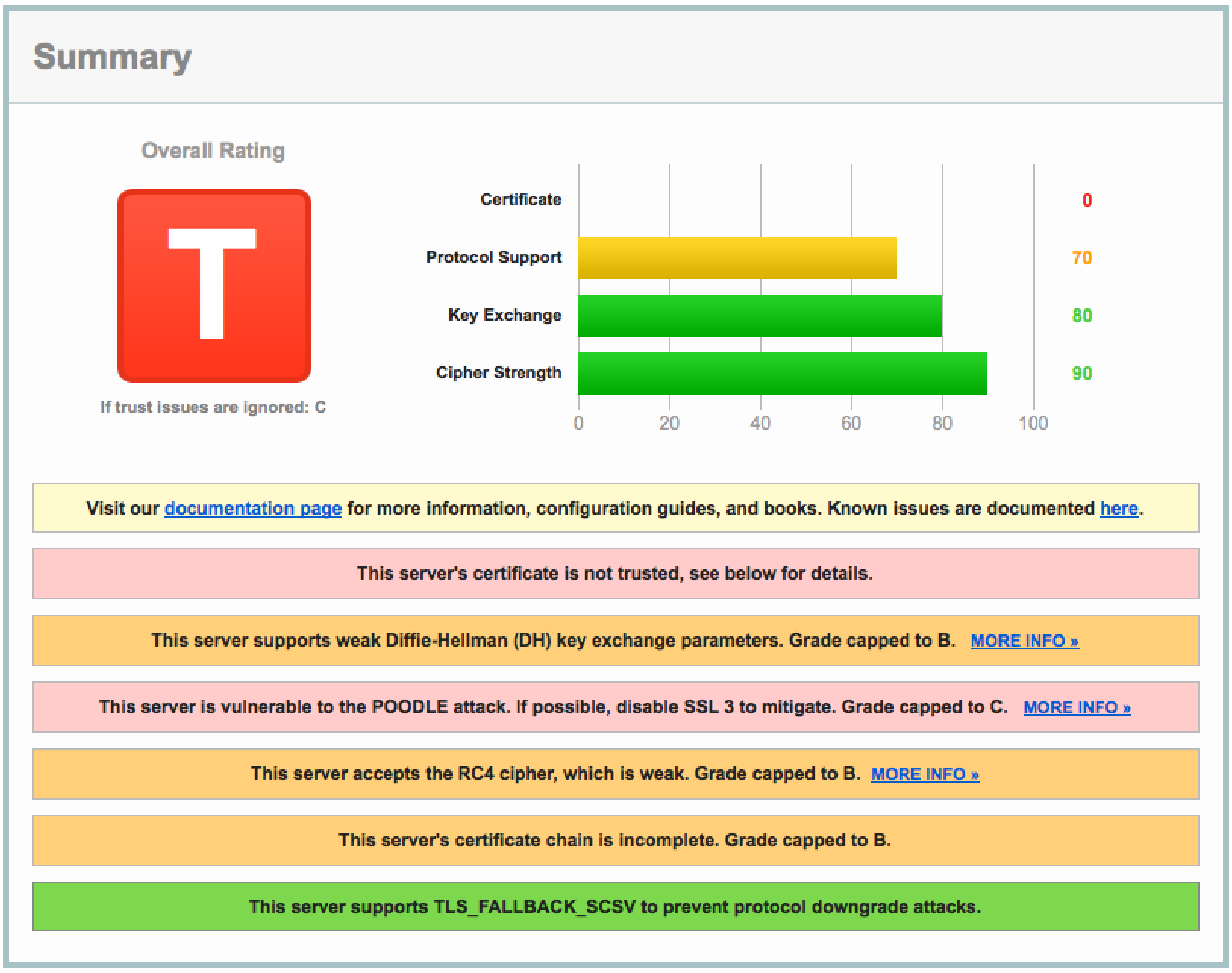
The Snapchat app[10] hardcodes additional hostnames that certificates can validate to, even if the hostname of the server does not match. It marks the domains `feelinsonice.com` and `www.feelinsonice.com` as permanently trusted (Listing 6.6). This does not cause an immediate vulnerability, as Snapchat is in control of the `feelinsonice.com` domain, but this custom logic has the potential to cause issues in the future.

In all, Sift performs well and can be said to be meeting its goals. It alerts a user when a phone is under attack, highlighting potential issues. It also draws attention to custom app configurations raising awareness of workarounds developers have made to get their apps working.

6.3 Automated Testing Results

The automated testing also revealed some interesting results. Six additional apps vulnerable to man-in-the-middle attacks were found. This number could be even higher with modifications to the testing script. Currently, the script attempts to login if the login screen is the first page, but

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



Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

This server's certificate is not trusted, see below for details.

This server supports weak Diffie-Hellman (DH) key exchange parameters. Grade capped to B. MORE INFO »

This server is vulnerable to the POODLE attack. If possible, disable SSL 3 to mitigate. Grade capped to C. MORE INFO »

This server accepts the RC4 cipher, which is weak. Grade capped to B. MORE INFO »

This server's certificate chain is incomplete. Grade capped to B.

This server supports TLS_FALLBACK_SCSV to prevent protocol downgrade attacks.

Figure 6.7: SSLLabs results for `android.nccp.netflix.com`

a number of apps have several screens before the login screen is reached. With a more sophisticated script, or more manual intervention, more TLS connections could be triggered, potentially discovering more vulnerable apps.

6.3.1 The Android Ecosystem

Analysing the TLS connections made also gave some useful insight into the Android ecosystem as a whole. Using the generated packet capture file, `tshark` could be used to provide useful information. Various parameters can be extracted and counted:

- The total number of connections (`ClientHello` is `ssl.handshake.type` 1 and `ServerHello` is `ssl.handshake.type` 2):

```
tshark -r cap.pcap -Y 'ssl.handshake.type == 1' | wc -l
```

```
tshark -r cap.pcap -Y 'ssl.handshake.type == 2' | wc -l
```

- Cipher suites used by servers:

```
tshark -r cap.pcap -VY 'ssl.handshake.type == 2' | grep 'Cipher Suite
    :' | sort | uniq -c | sort -n
```

- Cipher suites supported by clients:

```java
public final class bcb implements X509TrustManager
{
    ...
    static {
        TRUSTED_SUBJECT_ALTERNATIVE_DNS_NAMES = new String[] { "
           feelinsonice.com", "www.feelinsonice.com" };
        sTrustedSubjectAlternativeDNSNames = new HashSet<String>(
           Arrays.asList(bcb.TRUSTED_SUBJECT_ALTERNATIVE_DNS_NAMES))
           ;
    }
    ...
}
```

Listing 6.6: Extract from Snapchat's custom TrustManager

```
tshark -r cap.pcap -VY 'ssl.handshake.type == 1' | grep 'Cipher Suite
   :' | sort | uniq -c | sort -n
```

- The version of TLS used:

```
tshark -r cap.pcap -VY 'ssl.handshake.type == 2' | grep 'Version:' |
   sort | uniq -c | sort -n
```

- Whether the SNI extension is supported and used:

```
tshark -r cap.pcap -Y 'ssl.handshake.type == 2 && ssl.handshake.
   extension.type == 0x0000' | wc -l
```

```
tshark -r cap.pcap -Y 'ssl.handshake.type == 1 && ssl.handshake.
   extension.type == 0x0000' | wc -l
```

- The algorithms used to sign certificates:

```
tshark -r cap.pcap -VY 'ssl.handshake.certificate' | grep '
   algorithmIdentifier' | sort | uniq -c | sort -n
```

6.3.1.1 Results from the overall top 100 free apps

From the 100 top free apps on August 6 2015, 928 `ClientHello`s were sent, and 919 `ServerHello`s were received in reply. Of these, 891 were TLS version 1.2 and the remaining 28 were TLS 1.0. This is a fairly good result, but the fact that 3% of servers still respond with TLS 1.0 7 years after TLS 1.2 was standardized is still considerable, especially considering the fact that it is unlikely for any of the servers to have been running for longer than 7 years: the first Android phone was released in 2008.

Support for the SNI extension is considerably lower. It was offered by 583 of the client connections (63%), but only accepted by 318 of the servers (35%). There is a long way to go before support for this extension is complete, even though it is now completely supported by the Android API, and offers a layer of security and convenience for developers.

Looking at the algorithms used to sign the certificates offered by servers is also interesting. In total 1634 certificates were presented. Of these, 128 (7.8%) were signed with the `sha384With-RSAEncryption` algorithm, 595 (36.4%) with `sha256WithRSAEncryption` and 911 (55.8%) with `shaWithRSAEncryption`. Google Chrome marks certificates due to expire during or after 2016 and signed with the weak SHA-1 algorithm as insecure. With over half of the certificates signed with this algorithm, there is a long way to go for the transition to SHA-256 and above to be even close to complete.

Cipher suite support is similarly enlightening. The cipher suites negotiated by the servers are shown in Table 6.2. The levels of security displayed are the same as used in Sift, and can be found in Appendix C. Figure 6.8 shows in graph form the proportion of secure to obsolete cipher suites. While the majority (94.6%) of suites are secure, those that are not leave an app vulnerable to attack. RC4 has been shown not to be secure (Section 2.4.6). 3DES, used by two connections has theoretical attacks against it and is considered problematic: there is no reason for a server to be using this cipher. Using CBC mode with the SHA algorithm is also insecure, and vulnerable to the BEAST attack (Section 2.4.2).

Cipher Suite	**Count**	**Security**
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	1	Secure
TLS_ECDHE_RSA_WITH_RC4_128_SHA	1	Insecure
TLS_RSA_WITH_AES_128_GCM_SHA256	1	Secure but no forward secrecy
TLS_RSA_WITH_3DES_EDE_CBC_SHA	2	Insecure
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	3	Secure
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	6	Secure
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	7	Secure
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	8	Secure
TLS_RSA_WITH_AES_256_CBC_SHA	8	Insecure
TLS_RSA_WITH_AES_128_CBC_SHA	10	Insecure
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	12	Secure
TLS_RSA_WITH_RC4_128_SHA	29	Insecure
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	69	Secure
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	112	Secure
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	650	Secure

Table 6.2: Cipher suites negotiated by servers of the top 100 free Android apps

Looking at the cipher suites offered by clients also raises some questions. The full list is too long to be included here, but some of the interesting rows can be found in Table 6.3. Interestingly, the only two suites offered by all of the clients are insecure.

Cipher Suite	**Count**	**Security**
TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA	9	Insecure
TLS_RSA_EXPORT1024_WITH_RC2_CBC_56_MD5	9	Insecure
TLS_RSA_EXPORT1024_WITH_RC4_56_MD5	9	Insecure
TLS_RSA_EXPORT1024_WITH_RC4_56_SHA	9	Insecure
TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA	33	Insecure
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	36	Insecure
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	36	Insecure
TLS_DHE_DSS_WITH_DES_CBC_SHA	37	Insecure
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256	201	Secure
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256	201	Secure
TLS_RSA_WITH_AES_128_CBC_SHA	928	Insecure
TLS_RSA_WITH_AES_256_CBC_SHA	928	Insecure

Table 6.3: Cipher suites offered by clients of the top 100 free Android apps

A significant number of clients offer support for export cipher suites, which are almost designed

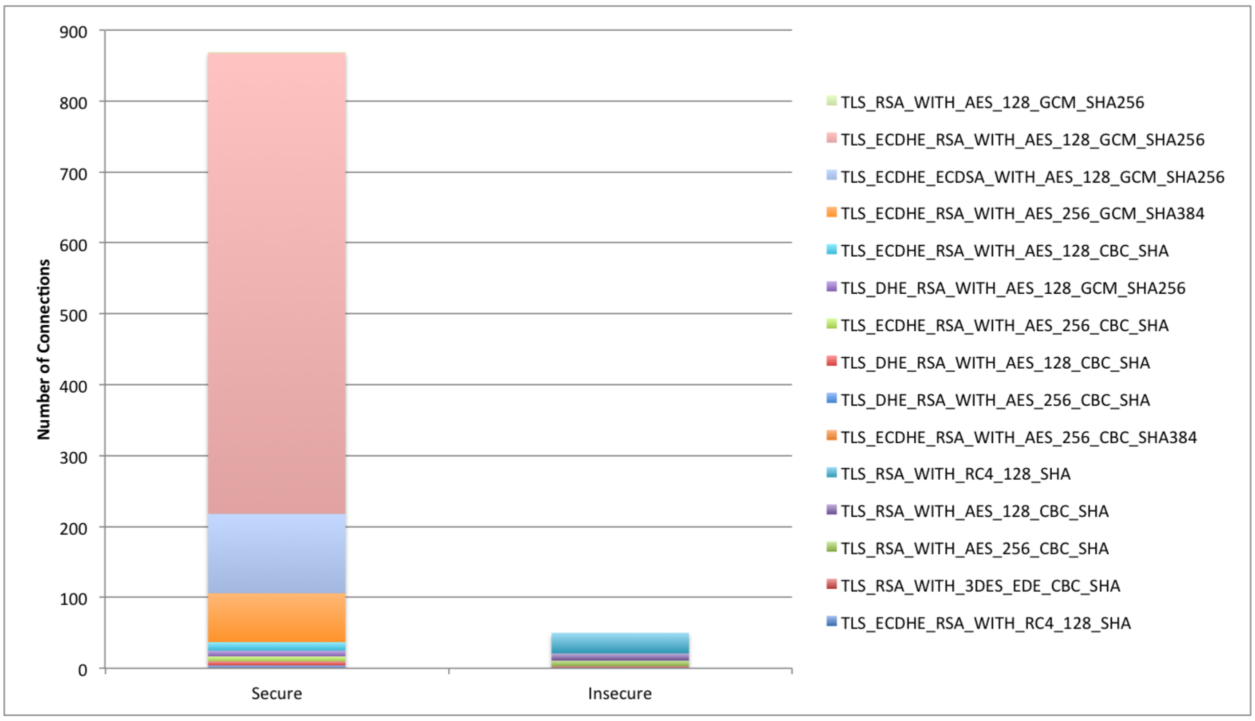


Figure 6.8: Graph of negotiated cipher suites by servers of the top 100 free Android apps

```
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
Cipher Suite: TLS_RSA_WITH_DES_CBC_SHA (0x0009)
Cipher Suite: TLS_RSA_EXPORT_WITH_RC4_40_MD5 (0x0003)
Cipher Suite: TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5 (0x0006)
Cipher Suite: TLS_RSA_EXPORT_WITH_DES40_CBC_SHA (0x0008)
Cipher Suite: TLS_RSA_EXPORT1024_WITH_RC4_56_MD5 (0x0060)
Cipher Suite: TLS_RSA_EXPORT1024_WITH_RC2_CBC_56_MD5 (0x0061)
Cipher Suite: TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA (0x0062)
Cipher Suite: TLS_RSA_EXPORT1024_WITH_RC4_56_SHA (0x0064)
```

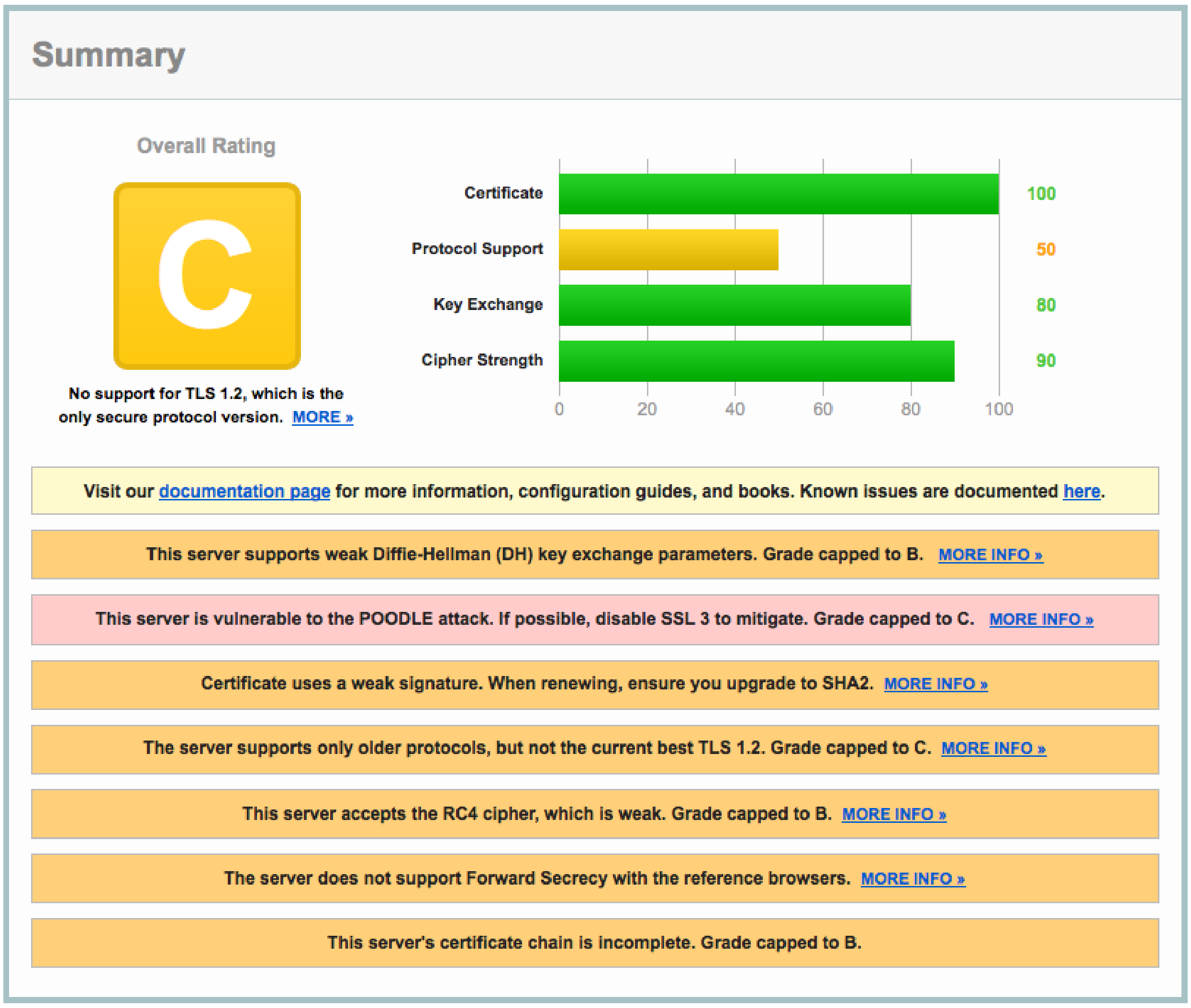
Listing 6.7: Cipher Suites supported by the Subway Surfers app

to be breakable and have been successfully attacked with the FREAK and Logjam attacks (Sections 2.3.2 and 2.3.3). While the servers eventually negotiate a more secure suite this clearly exposes the client to some form of attack. Additionally, this also suggests other configuration issues. Using the packet capture file, the hostnames of the servers that the clients supporting these weak cipher suites were attempting to connect to were extracted. Universally, a scan with Ristić's SSLLabs tool highlighted issues with the servers, ranging from vulnerability to denial of service attacks to configurations that could lead to other attacks. As an example, the app connecting to `subwaysurfers.kiloo-games.com` supports 13 cipher suites (Listing 6.7).

Just seeing the words `EXPORT`, `MD5`, `RC2` and `DES` in this list should be enough to arouse suspicion. This connection is generated by the Subway Surfers game downloaded by between 500,000,000 and 1,000,000,000 users.[11] The results of a scan using the SSLLabs tool are shown in Figure 6.9. It is vulnerable to the POODLE attack because it still supports SSL3. It does not support TLS 1.2 at all, the certificate is signed with the weak SHA-1 algorithm, and its certificate chain is incomplete. This server could be considered vulnerable to a host of attacks.

[11]`https://play.google.com/store/apps/details?id=com.kiloo.subwaysurf`

No server or client should offer support for export protocols and the fact that they do suggests poor coding or lack of consideration for security. Client support for weak protocols can be used as a signifier of weak servers and vulnerable clients, and serve as an excellent place for an attacker to start looking.



Summary

Overall Rating

C

No support for TLS 1.2, which is the only secure protocol version. MORE »

Certificate	100
Protocol Support	50
Key Exchange	80
Cipher Strength	90

Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

This server supports weak Diffie-Hellman (DH) key exchange parameters. Grade capped to B. MORE INFO »

This server is vulnerable to the POODLE attack. If possible, disable SSL 3 to mitigate. Grade capped to C. MORE INFO »

Certificate uses a weak signature. When renewing, ensure you upgrade to SHA2. MORE INFO »

The server supports only older protocols, but not the current best TLS 1.2. Grade capped to C. MORE INFO »

This server accepts the RC4 cipher, which is weak. Grade capped to B. MORE INFO »

The server does not support Forward Secrecy with the reference browsers. MORE INFO »

This server's certificate chain is incomplete. Grade capped to B.

Figure 6.9: SSLLabs results for `subwaysurfers.kiloo-games.com`

6.3.1.2 Results from the top 100 free apps from each category

The results from the larger scan, performed on August 10 2015, were similar. 15806 `ClientHellos` were sent, and 15779 `ServerHellos` were received in reply. Of these, an impressive 15393 were TLS version 1.2, 26 were TLS version 1.1, 356 were TLS version 1.0, 2 were marked as unknown, and amazingly, there were 2 SSL 3 connections. One of these belonged to a connection for the Flurry analytics library, and the other for the MagicJack app.[12] to `prov1.talk4free.com` Overall, the fact that 97.5% of connections used TLS 1.2 is reassuring, but the fact that even one app (downloaded by between 10,000,000 and 50,000,000 people) used the obsolete SSL 3 protocol is amazing.

SNI support mirrored that of the smaller sample. It was offered by 9591 of the client connections (61%) and accepted by 5577 of the servers (35%).

The algorithms used to sign the certificates were also similar. Of a total of 28280 presented certificates, 2653 (9.4%) were signed with the `sha384WithRSAEncryption` algorithm, 10899 (38.5%)

[12]`https://play.google.com/store/apps/details?id=com.magicjack`

with `sha256WithRSAEncryption`, 14701 (52.0%) with `shaWithRSAEncryption`, and 27 (0.1%) with other algorithms.

There is some interest in looking at the other algorithms presented, although they represent such a small percentage of the total. One certificate, used on a domain owned by Salesforce, `shopitize.desk.com`, used the ultra-strong `sha512WithRSAEncryption`. We can expect to see more certificates signed with this in the future. 7 certificates were signed with `ECDSAWithSHA256` and 7 with `ECDSAWithSHA384`; secure, if a little esoteric. Interestingly, 12 certificates were signed with the incredibly insecure `md2WithRSAEncryption` algorithm. Looking into this futher, in every case the certificate was a root Verisign certificate. Ignoring the fact that servers shouldn't be sending root certificates in the first place (they should be in the client's trust store already), the signature algorithm for root certificates doesn't matter, as they are trusted by default, and so this is not in fact a security risk. The apps that these certificates were presented to were Nationwide Mobile Banking,[13] Amex UK,[14] HSBC Mobile Banking[15] and eBay.[16] Three out of four of these apps are finance related: perhaps they all share a common developer that has misconfigured the servers in the same way.

The cipher suites negotiated by the servers are shown in Table 6.4. One connection was made to `chronicle.comparethemarket.com` with the `TLS_RSA_WITH_RC4_128_MD5` suite, which is incredibly insecure. As with the smaller sample (and visualised in Figure 6.10), 95.4% of suites used were secure, still leaving one in twenty connections using insecure cipher suites.

Cipher Suite	**Count**	**Security**
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	1	Secure
TLS_RSA_WITH_RC4_128_MD5	1	Insecure
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	1	Secure
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	13	Secure
TLS_ECDHE_RSA_WITH_RC4_128_SHA	21	Insecure
TLS_RSA_WITH_3DES_EDE_CBC_SHA	21	Insecure
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	32	Secure
TLS_RSA_WITH_AES_128_GCM_SHA256	33	Secure but no forward secrecy
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	34	Secure
TLS_RSA_WITH_AES_256_GCM_SHA384	38	Secure but no forward secrecy
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256	40	Secure
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	86	Secure
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	155	Secure
TLS_RSA_WITH_RC4_128_SHA	191	Insecure
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	205	Secure
TLS_RSA_WITH_AES_128_CBC_SHA	244	Insecure
TLS_RSA_WITH_AES_256_CBC_SHA	253	Insecure
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	750	Secure
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	2875	Secure
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	10783	Secure

Table 6.4: Cipher suites negotiated by servers of the top 100 free Android apps in each category

The cipher suites offered by the clients also offer a similar picture. 129 different suites were offered by the clients, ranging from the insecure `TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA` offered by one connection (the MagicJack app again), to the very secure `TLS_ECDHE_ECDSA_WITH_CHACHA20_-POLY1305_SHA256` offered by 4239, to the again insecure `TLS_RSA_WITH_AES_128_CBC_SHA` offered by

[13]`https://play.google.com/store/apps/details?id=co.uk.Nationwide.Mobile`
[14]`https://play.google.com/store/apps/details?id=com.americanexpress.android.acctsvcs.uk`
[15]`https://play.google.com/store/apps/details?id=com.htsu.hsbcpersonalbanking`
[16]`https://play.google.com/store/apps/details?id=com.ebay.mobile`

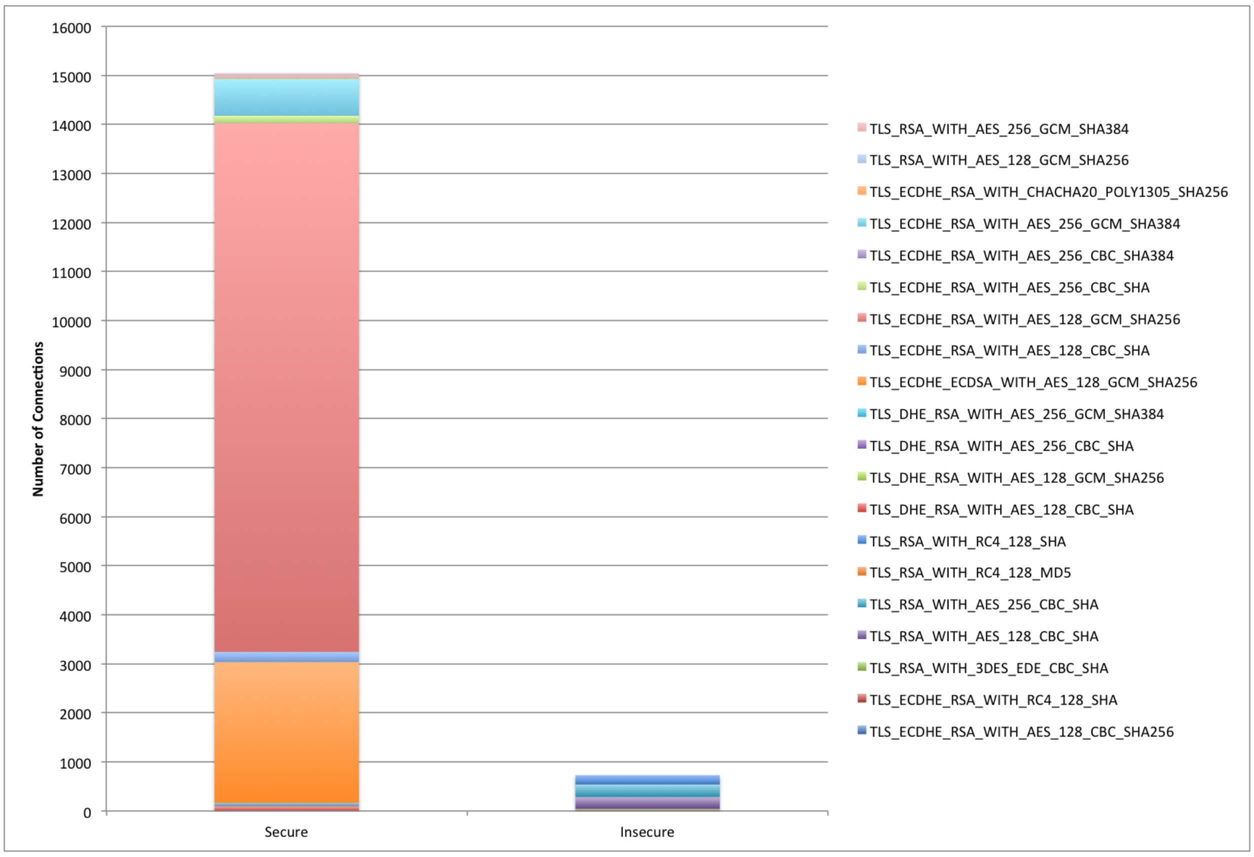


Figure 6.10: Graph of negotiated cipher suites by servers of the top 100 free Android apps in each category

nearly all (15800) clients.

6.4 Summary

Apps were tested manually and automatically, and run against the Sift app. Some serious issues were found. Apps made by large corporations failed to properly secure customer data, leaving information at risk. The increasing use of cross-site authentication means that apps failing to secure user information can be leveraged to provide access to higher-value accounts such as those controlled by Google and Facebook. A survey of the Android ecosystem was performed, showing that around one in forty connections do not use the most up-to-date version of TLS and one in twenty negotiating a connection with insecure cipher suites. There is much room for improvement.

Chapter 7

Conclusion

This has been a wide-ranging study, encompassing both the development of a new app for users and an examination of the Android ecosystem. This section will reiterate the contributions made by this project, put forward some recommendations to improve security on Android, offer areas for future work, and make a final conclusion.

7.1 Achievements

- An app, Sift, has been developed that alerts users when a man-in-the-middle attack is taking place. It protects them from poorly coded apps that fail to properly secure the TLS communication channel, preventing the loss of important personal information.
- Several apps have been identified that do not protect user information, despite their claims. These include apps that deal with personal, financial and medical information. In depth analysis has been performed, pinpointing the exact cause of vulnerability. The app developers have been notified, with one, a major UK-based online retailer, rapidly fixing their app in response to the report.
- A straightforward process for setting up and performing man-in-the-middle attacks with cheap, off-the-shelf hardware has been detailed, demonstrating the ease and practicality of this attack.
- An automated analysis of over 2500 apps has been performed, highlighting the state of the Android ecosystem and demonstrating that many apps, and the servers that they communicate with, may be susceptible to attacks based on known weaknesses in the TLS protocol, despite having properly coded certificate validation.

7.2 Recommendations for Android

With the results of this investigation, several recommendations can be made to improve the security of TLS connections on the Android operating system:

- As suggested by Fahl et al. [30], the Android operating system must enforce stronger security through the API. Disabling of crucial certification checks cannot be permitted, or should require many more steps to make sure that a developer is fully aware of the consequences. Currently, it is too easy to forget to use a `HostnameValidator` when using a custom `SSLSocketFactory` - this is a fault of the API, and while it may have logical reasoning behind it[1], the API should be changed to make things clearer. Developers should be given as little

[1] 'The rationale for that decision was probably that the TLS/SSL handshake happens at the socket layer, whereas the hostname verification depends on the application-level protocol (HTTPS at that time). Therefore, the `X509TrustManager` class for certificate trust checks was integrated into the low-level `SSLSocket` and `SSLEngine` classes, whereas the `HostnameVerifier` API was only incorporated into the `HttpsURLConnection`.' [86]

freedom as possible when security is at stake: too many mistakes can be made, as the results of this study show. The longer developers are able to shoot themselves in the foot, the higher the likelihood of a significant data breach, causing users loss of important data and bringing negative publicity to Android.

- Google, as stewards of the Android operating system could also do more. The vast majority of app installs by users come from Google's Play Store. The Play Store already scans uploaded apps for malware and other issues; it would be a simple step to add static analysis similar to Fahl's Mallodroid tool. This would help catch the most egregious of errors by developers.

- Certificate pinning should be made easier to implement and use for developers. If this is integrated correctly it provides protection both against man-in-the-middle attacks and attacks using certificates signed by trust certificate authorities (perhaps conducted by state agencies). If certificate pinning is made easy and error free, a lot of issues can be prevented. Of course, if configuration remains similar to now, many developers may get this process wrong and accidentally leave their apps vulnerable to attack: care must be taken in insuring that as few configuration errors as possible can be made.

- Users should also be made more aware that they are under attack. Sift alerts users that they are potentially connected to a malicious access point and allows them to take steps to disconnect and avoid any data breach. Without Sift running, a user will not know that they are on an untrustworthy connection. When a correctly coded app fails certificate validation, often the error given is a generic one relating to a lack of internet connectivity, leaving the user unaware of the true cause of failure. While increased awareness cannot help every user, the more information technical users have the better.

- There could also be a stronger response from government regulators. The FTC in the United States fined two companies in 2014 [74], but stronger enforcement could persuade large companies to take security more seriously. One could argue that failing to secure personal data over the internet is a breach of the UK's Data Protection Act 1998, which contains strong penalties for violations. This study has shown that personal, financial and medical information can all be at risk, and government regulators would do well to protect users' information.

7.3 Future Work

There remains space for significant future work in extending the Sift app to offer more protection to users:

- As Google improves the state of the Android VpnService, IPv6 support can be added to the app, to offer protection on networks of the future, and on 4G mobile phone networks.

- Non-HTTPS traffic can also be analysed, including secure email (perhaps by using dynamic TLS detection through protocol sensing). Google is also increasingly experimenting with the QUIC protocol as a faster alternative to TCP: work will be needed to analyse these connections.

- Certificates in the app can be cached, and the user alerted if an unexpected certificate is used on the app's next connection. Care will have to be taken to avoid false positives.

- The certification validation algorithms could also be extended, using multiple techniques and implementations (perhaps using those developed by Mersinjak et al. [10] or Fahl et al. [30]). This would also help verify the validation code in the manner of Brubaker et al. [25].

- Additionally, the app could check the revocation lists in the certificates, assuming that issues with the reliability of these can be resolved.

- Apps could be categorized by risk, providing stronger alerts to a user if an app in a higher risk category (for example the financial category) has issues.
- Vulnerabilities could be reported to a central server, allowing potential attacks to be quickly found and reported to authorities and other users.
- Some apps use self-signed certificates securely. A process could be developed to have a whitelist of these certificates (either user or developer driven) to avoid false positives.
- A testing mode could be implemented where the app itself attempts to conduct a man-in-the-middle attack against the apps on the phone. If any succeeds an app could be marked as vulnerable. This feature would also be useful for developers testing their own apps as they could avoid setting up a malicious access point of their own.
- To improve speed and battery life, a future version of the app could be coded using the Android Native Development Kit (NDK) which allows code to be written directly in native-code such as C or C++.
- To improve the testing suite, more sophisticated scripts could be written to ensure all apps make a secure connection when being tested.
- The `mitmproxy` software could also be extended to attempt both the self-signed certificate attack and incorrect hostname attack in the same handshake, rather than having to test connections twice.
- If one was designing `mitmproxy` to attack users, it may also be worthwhile to extend it to attack probabilistically: perhaps only every tenth connection. This would allow most apps to behave normally, with the user unconcerned, but with any vulnerable apps still leaking important user information.
- A further investigation could also investigate the possibility of attacking apps that advertise support for weak cipher suites (include the Export suites). There is a likelihood that these apps could be tricked into negotiating a weak TLS connection with the server causing the connection to be compromised.
- Regular scanning of the Android ecosystem could also be performed and published, perhaps on a monthly basis. This would both track the (hopefully) improving level of TLS support and security in apps, as well as provide information to users seeking to learn about the security of the apps they use.

7.4 Final Conclusion

The implementation of TLS in many Android apps is, to put it simply, broken. Google, as steward of the operating system must take some blame, both for allowing a confusing API to mislead developers, and for not taking steps to scan apps for simple vulnerabilities. The ultimate fault though, rests with developers who claim to implement security, yet fail to do so. Users are left in the middle, with no awareness that their personal data can be stolen, and with no protection from this theft. Sift helps defend users against poorly coded apps, while the naming and analysis of the broken apps developed by major corporations should help to improve the general state of the Android ecosystem. Until the day that all apps implement TLS correctly, users should follow a simple mantra: ‘Love all, trust a few’ [87, I.i.61].

Bibliography

[1] Eric Rescorla. *SSL and TLS: designing and building secure systems.* Boston, Mass; London: Addison-Wesley, 2001.

[2] Tim Dierks. *Security Standards and Name Changes in the Browser Wars.* May 2014. URL: http://tim.dierks.org/2014/05/security-standards-and-name-changes-in.html (visited on 08/12/2015).

[3] Tim Dierks and Christopher Allen. *The TLS Protocol Version 1.0.* RFC 2246. RFC Editor; RFC Editor, Jan. 1999. URL: http://www.rfc-editor.org/rfc/rfc2246.txt.

[4] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.1.* RFC 4346. RFC Editor; RFC Editor, Apr. 2006. URL: http://www.rfc-editor.org/rfc/rfc4346.txt.

[5] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2.* RFC 5246. RFC Editor; RFC Editor, Aug. 2008. URL: http://www.rfc-editor.org/rfc/rfc5246.txt.

[6] R. Barnes et al. *Deprecating Secure Sockets Layer Version 3.0.* RFC 7568. RFC Editor; RFC Editor, June 2015. URL: http://www.rfc-editor.org/rfc/rfc7568.txt.

[7] Ivan Ristic. *Bulletproof SSL and TLS.* London: Feisty Duck, 2014.

[8] Bruce Schneier. *Applied cryptography: protocols, algorithms and source code in C.* 2nd. New York; Chichester: Wiley, 1996.

[9] Nick Mathewson and Ben Laurie. *Deprecating gmt_unix_time in TLS.* Internet Draft. IETF Secretariat, Dec. 2013. URL: http://www.ietf.org/internet-drafts/draft-mathewson-no-gmtunixtime-00.txt.

[10] David Kaloper Merinjak et al. "Not-Quite-So-Broken TLS: Lessons in Re-Engineering a Security Protocol Specification and Implementation". In: *24th USENIX Security Symposium (USENIX Security 15).* Washington, D.C.: USENIX Association, Aug. 2015. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/mersinjak.

[11] J. Salowey et al. *Transport Layer Security (TLS) Session Resumption without Server-Side State.* RFC 4507. RFC Editor; RFC Editor, May 2006. URL: http://www.rfc-editor.org/rfc/rfc4507.txt.

[12] J. Salowey et al. *Transport Layer Security (TLS) Session Resumption without Server-Side State.* RFC 5077. RFC Editor; RFC Editor, Jan. 2008. URL: http://www.rfc-editor.org/rfc/rfc5077.txt.

[13] E. Rescorla. *HTTP Over TLS.* RFC 2818. RFC Editor; RFC Editor, May 2000. URL: http://www.rfc-editor.org/rfc/rfc2818.txt.

[14] D. Cooper et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* RFC 5280. RFC Editor; RFC Editor, May 2008. URL: http://www.rfc-editor.org/rfc/rfc5280.txt.

[15] D. Eastlake. *Transport Layer Security (TLS) Extensions: Extension Definitions.* RFC 6066. RFC Editor; RFC Editor, Jan. 2011. URL: http://www.rfc-editor.org/rfc/rfc6066.txt.

[16] Heather Adkins. *An update on attempted man-in-the-middle attacks.* Aug. 2011. URL: http://googleonlinesecurity.blogspot.co.uk/2011/08/update-on-attempted-man-in-middle.html (visited on 08/12/2015).

[17] J. R. Prins. *DigiNotar Certificate Authority breach: "Operation Black Tulip".* Tech. rep. FOX-IT, Sept. 2011. URL: http://www.rijksoverheid.nl/ministeries/bzk/documenten-en-publicaties/rapporten/2011/09/05/diginotar-public-report-version-1.html.

[18] Alexander Sotirov et al. *MD5 considered harmful today.* Tech. rep. Dec. 2008. URL: http://www.win.tue.nl/hashclash/rogue-ca/ (visited on 08/12/2015).

[19] Eric Mill. *Why Google is Hurrying the Web to Kill SHA-1.* Sept. 2014. URL: https://konklone.com/post/why-google-is-hurrying-the-web-to-kill-sha-1 (visited on 08/14/2015).

[20] Mike Zusman. *DNS vuln + SSL cert = FAIL.* July 2008. URL: https://intrepidusgroup.com/insight/2008/07/dns-vuln-ssl-cert-fail/ (visited on 08/12/2015).

[21] Olli Vanska. *A Finnish man created this simple email account - and received Microsoft's security certificate.* Mar. 2015. URL: http://www.tivi.fi/Kaikki_uutiset/2015-03-18/A-Finnish-man-created-this-simple-email-account---and-received-Microsofts-security-certificate-3217662.html (visited on 08/12/2015).

[22] Johnathan Nightingale. *Revoking Trust in DigiCert Sdn. Bhd Intermediate Certificate Authority.* Nov. 2011. URL: https://blog.mozilla.org/security/2011/11/03/revoking-trust-in-digicert-sdn-bhd-intermediate-certificate-authority/ (visited on 08/12/2015).

[23] Mike Zusman. *Nobody is perfect.* Jan. 2008. URL: http://schmoil.blogspot.co.uk/2009/01/nobody-is-perfect.html (visited on 08/12/2015).

[24] Eddy Nigg. *Untrusted Certificates.* Dec. 2008. URL: https://blog.startcom.org/?p=145 (visited on 08/12/2015).

[25] C. Brubaker et al. "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations". In: *Security and Privacy (SP), 2014 IEEE Symposium on.* 2014, pp. 114–129. ISBN: 1081-6011.

[26] Adam Langley. *Apple's SSL/TLS bug.* Feb. 2014. URL: https://www.imperialviolet.org/2014/02/22/applebug.html (visited on 08/12/2015).

[27] *CVE-2015-1793.* Available from MITRE, CVE-ID CVE-2015-1793. Feb. 2015. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1793.

[28] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. "You'Ve Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* CHI '08. Florence, Italy: ACM, 2008, pp. 1065–1074. ISBN: 978-1-60558-011-1. URL: http://doi.acm.org/10.1145/1357054.1357219.

[29] Joshua Sunshine et al. "Crying Wolf: An Empirical Study of SSL Warning Effectiveness". In: *Proceedings of the 18th Conference on USENIX Security Symposium.* SSYM'09. Montreal, Canada: USENIX Association, 2009, pp. 399–416. URL: http://dl.acm.org/citation.cfm?id=1855768.1855793.

[30] Sascha Fahl et al. "Rethinking SSL Development in an Appified World". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security.* CCS '13. Berlin, Germany: ACM, 2013, pp. 49–60. ISBN: 978-1-4503-2477-9. URL: http://doi.acm.org/10.1145/2508859.2516655.

[31] Sascha Fahl et al. "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.* CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 50–61. ISBN: 978-1-4503-1651-4. URL: http://doi.acm.org/10.1145/2382196.2382205.

[32] *Certificate revocation: Why browsers remain affected by Heartbleed.* Apr. 2014. URL: http://news.netcraft.com/archives/2014/04/24/certificate-revocation-why-browsers-remain-affected-by-heartbleed.html (visited on 08/12/2015).

[33] Dan Goodin. *Lenovo PCs ship with man-in-the-middle adware that breaks HTTPS connections.* Feb. 2015. URL: http://arstechnica.com/security/2015/02/lenovo-pcs-ship-with-man-in-the-middle-adware-that-breaks-https-connections/ (visited on 08/12/2015).

[34] Robert Graham. *Extracting the SuperFish certificate.* Feb. 2015. URL: http://blog.erratasec.com/2015/02/extracting-superfish-certificate.html (visited on 08/12/2015).

[35] *The Heartbleed Bug.* Apr. 2014. URL: http://heartbleed.com/ (visited on 08/12/2015).

[36] Robert Graham. *300k servers vulnerable to Heartbleed one month later.* May 2014. URL: http://blog.erratasec.com/2014/05/300k-servers-vulnerable-to-heartbleed.html (visited on 08/12/2015).

[37] Beurdouche et al. *FREAK: Factoring RSA Export Keys.* Mar. 2015. URL: https://www.smacktls.com/#freak (visited on 08/12/2015).

[38] David Adrian et al. *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice.* Tech. rep. May 2015. URL: https://weakdh.org/imperfect-forward-secrecy.pdf (visited on 08/12/2015).

[39] Bodo Moeller and Adam Langley. *TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks.* Internet Draft. IETF Secretariat, July 2014. URL: http://www.ietf.org/internet-drafts/draft-ietf-tls-downgrade-scsv-00.txt.

[40] Mark Nottingham, Patrick McManus, and Julian Reschke. *HTTP Alternative Services.* Internet Draft. IETF Secretariat, Oct. 2014. URL: http://www.ietf.org/internet-drafts/draft-ietf-httpbis-alt-svc-04.txt.

[41] Paul Ducklin. *Firefox issues brand new update to fix HTTPS security hole in new update.* Apr. 2015. URL: https://nakedsecurity.sophos.com/2015/04/07/firefox-issues-brand-new-update-to-fix-https-security-hole-in-new-update/ (visited on 08/12/2015).

[42] Ray Marsh. *Renegotiating TLS.* Tech. rep. PhoneFactor, Inc., Nov. 2009. URL: http://ww1.prweb.com/prfiles/2009/11/05/104435/RenegotiatingTLS.pdf (visited on 08/12/2015).

[43] Amil Kurmus. *TLS renegotiation vulnerability (CVE-2009-3555).* Nov. 2009. URL: http://www.securegoose.org/2009/11/tls-renegotiation-vulnerability-cve.html (visited on 08/12/2015).

[44] Thai Duong and Juliano Rizzo. "Here Come The XOR Ninjas". In: (May 2011). URL: http://www.hit.bme.hu/~buttyan/courses/EIT-SEC/abib/04-TLS/BEAST.pdf (visited on 08/12/2015).

[45] Juliano Rizzo and Thai Duong. *The CRIME attack.* Tech. rep. 2012. URL: www.ekoparty.org/archive/2012/CRIME_ekoparty2012.pdf (visited on 08/12/2015).

[46] Tal Be'ery and Amichai Shulman. *A Perfect CRIME? Only TIME Will Tell.* Mar. 2013. URL: https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf (visited on 08/12/2015).

[47] Yoel Gluck, Neal Harris, and Angelo Prado. "BREACH: Reviving the CRIME Attack". In: (July 2013). URL: http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf (visited on 08/12/2015).

[48] Nadhem J. AlFardan and Kenneth G. Paterson. *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols.* Feb. 2013. URL: http://www.isg.rhul.ac.uk/tls/Lucky13.html (visited on 08/12/2015).

[49] Bodo Moller, Thai Duong, and Krzysztof Kotowicz. "This POODLE Bites: Exploiting The SSL 3.0 Fallback". In: (Sept. 2014). URL: https://www.openssl.org/~bodo/ssl-poodle.pdf (visited on 08/12/2015).

[50] Adam Langley. *The POODLE bites again.* Dec. 2014. URL: https://www.imperialviolet.org/2014/12/08/poodleagain.html.

[51] Itsik Mantin. "Bar Mitzvah Attack; Breaking SSL with a 13-year old RC4 Weakness". In: (2015). URL: https://www.blackhat.com/docs/asia-15/materials/asia-15-Mantin-Bar-Mitzvah-Attack-Breaking-SSL-With-13-Year-Old-RC4-Weakness-wp.pdf (visited on 08/12/2015).

[52] A. Popov. *Prohibiting RC4 Cipher Suites.* RFC 7465. RFC Editor; RFC Editor, Feb. 2015. URL: http://www.rfc-editor.org/rfc/rfc7465.txt.

[53] Michael Mimoso. *Google, Mozilla, Microsoft to Sever RC4 Support in Early 2016.* Sept. 2015. URL: https://threatpost.com/google-mozilla-microsoft-to-sever-rc4-support-in-early-2016/114498 (visited on 09/01/2015).

[54] Nadhem AlFardan et al. "On the Security of RC4 in TLS". In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13).* Washington, D.C.: USENIX, 2013, pp. 305–320. ISBN: 978-1-931971-03-4. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/alFardan.

[55] Christina Garman, Kenneth G. Paterson, and Thyla Van der Merwe. "Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS". In: *24th USENIX Security Symposium (USENIX Security 15).* Washington, D.C.: USENIX Association, Aug. 2015, pp. 113–128. ISBN: 978-1-931971-232. URL: http://blogs.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/garman.

[56] Mathy Vanhoef and Frank Piessens. "All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS". In: *24th USENIX Security Symposium (USENIX Security 15).* Washington, D.C.: USENIX Association, Aug. 2015, pp. 97–112. ISBN: 978-1-931971-232. URL: http://blogs.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/vanhoef.

[57] Ryan Gallagher. *New Snowden Documents Show NSA Deemed Google Networks a "Target".* Sept. 2013. URL: http://www.slate.com/blogs/future_tense/2013/09/09/shifting_shadow_stormbrew_flying_pig_new_snowden_documents_show_nsa_deemed.html (visited on 08/14/2015).

[58] Ryan Gallagher. *Google MITM attack.* Sept. 2014. URL: http://www.scribd.com/doc/166819124/mitm-Google (visited on 08/14/2015).

[59] Glenn Greenwald. *XKeyscore: NSA tool collects 'nearly everything a user does on the internet'.* July 2013. URL: http://www.theguardian.com/world/2013/jul/31/nsa-top-secret-program-online-data (visited on 08/14/2015).

[60] Nicole Perlroth, Jeff Larson, and Scott Shane. *N.S.A. Able to Foil Basic Safeguards of Privacy on Web.* Sept. 2013. URL: http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html?pagewanted=all (visited on 08/14/2015).

[61] Stephen Checkoway et al. "On the Practical Exploitability of Dual EC in TLS Implementations". In: *23rd USENIX Security Symposium (USENIX Security 14).* San Diego, CA: USENIX Association, Aug. 2014, pp. 319–335. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/checkoway.

[62] Daniel J. Bernstein, Tanja Lange, and Ruben Niederhagen. *Dual EC: A Standardized Back Door.* Cryptology ePrint Archive, Report 2015/767. 2015. URL: https://eprint.iacr.org/2015/767.pdf.

[63] Devdatta Akhawe and Adrienne Porter Felt. "Alice in Warningland: A Large-scale Field Study of Browser Security Warning Effectiveness". In: *Proceedings of the 22nd USENIX Conference on Security.* SEC'13. Washington, D.C.: USENIX Association, 2013, pp. 257–272. ISBN: 978-1-931971-03-4. URL: http://dl.acm.org/citation.cfm?id=2534766.2534789.

[64] Martin Georgiev et al. “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*. CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pp. 38–49. ISBN: 978-1-4503-1651-4. URL: `http://doi.acm.org/10.1145/2382196.2382204`.

[65] J. Clark and P. C. van Oorschot. “SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements”. In: *Security and Privacy (SP), 2013 IEEE Symposium on*. May 2013, pp. 511–525. ISBN: 1081-6011.

[66] Georgios Portokalidis et al. “Paranoid Android: Versatile Protection for Smartphones”. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC ’10. Austin, Texas, USA: ACM, 2010, pp. 347–356. ISBN: 978-1-4503-0133-6. URL: `http://doi.acm.org/10.1145/1920261.1920313`.

[67] Graham Edgecombe. *Detection of SSL-related security vulnerabilites in Android applications*. Tech. rep. June 2014.

[68] Manuel Egele et al. “An Empirical Study of Cryptographic Misuse in Android Applications”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS ’13. Berlin, Germany: ACM, 2013, pp. 73–84. ISBN: 978-1-4503-2477-9. URL: `http://doi.acm.org/10.1145/2508859.2516693`.

[69] Graham Edgecombe. *Android SSL security vulnerabilities*. Tech. rep. Netcraft, 2014.

[70] *Vulnerability Note VU#582497*. Available from CERT. Sept. 2014. URL: `http://www.kb.cert.org/vuls/id/582497`.

[71] Sam Bowne. *Popular Android Apps with SSL Certificate Validation Failure*. 2015. URL: `https://samsclass.info/128/proj/popular-ssl.htm` (visited on 08/14/2015).

[72] Dan Goodin. *Game-over HTTPS defects in dozens of Android apps expose user passwords*. June 2015. URL: `http://arstechnica.com/security/2015/06/game-over-https-defects-in-dozens-of-android-apps-expose-user-passwords/` (visited on 08/12/2015).

[73] Sean Gallagher. *Simple Wi-Fi attack grabs BMW, Mercedes, and Chrysler cars virtual keys*. Aug. 2015. URL: `http://arstechnica.com/security/2015/08/simple-wi-fi-attack-grabs-bmw-mercedes-and-chrysler-cars-virtual-keys/` (visited on 08/14/2015).

[74] *Fandango, Credit Karma Settle FTC Charges that They Deceived Consumers By Failing to Securely Transmit Sensitive Personal Information*. Mar. 2014. URL: `https://www.ftc.gov/news-events/press-releases/2014/03/fandango-credit-karma-settle-ftc-charges-they-deceived-consumers` (visited on 08/12/2015).

[75] Jesse Wilson. *Androids HTTP Clients*. Sept. 2011. URL: `http://android-developers.blogspot.co.uk/2011/09/androids-http-clients.html` (visited on 08/14/2015).

[76] *Security with HTTPS and SSL*. URL: `http://android-developers.blogspot.co.uk/2011/09/androids-http-clients.html` (visited on 08/14/2015).

[77] Patrick Mutchler. *Who Needs SSL Anyway?: Ignoring SSL Errors in Android WebView*. Sept. 2014. URL: `http://stanford.edu/~pcm2d/blog/ssl.html` (visited on 08/14/2015).

[78] R. Khare and S. Lawrence. *Upgrading to TLS Within HTTP/1.1*. RFC 2817. RFC Editor; RFC Editor, May 2000. URL: `http://www.rfc-editor.org/rfc/rfc2817.txt`.

[79] *Libpcap File Format*. URL: `https://wiki.wireshark.org/Development/LibpcapFileFormat` (visited on 08/14/2015).

[80] P. Mockapetris. *Domain names - implementation and specification*. RFC 1035. RFC Editor; RFC Editor, Nov. 1987. URL: `http://www.rfc-editor.org/rfc/rfc1035.txt`.

[81] Eric Rescorla. *TLS Cipher Suite Registry*. Available from IANA. URL: `https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4`.

[82] Adi Sharabani. *WiFiGate How Mobile Carriers Expose Us to Wi-Fi Attacks*. June 2013. URL: `https://www.skycure.com/blog/wifigate-how-mobile-carriers-expose-us-to-wi-fi-attacks/` (visited on 08/14/2015).

[83] Smoot Carl-Mitchell and John S. Quarterman. *Using ARP to implement transparent subnet gateways*. RFC 1027. RFC Editor; RFC Editor, Oct. 1987. URL: `http://www.rfc-editor.org/rfc/rfc1027.txt`.

[84] Marco Valleri and Alberto Ornagi. *Man in the middle attacks*. 2003. URL: `https://www.blackhat.com/presentations/bh-europe-03/bh-europe-03-valleri.pdf` (visited on 08/14/2015).

[85] Will Dormann. *Finding Android SSL Vulnerabilities with CERT Tapioca*. Sept. 2014. URL: `https://insights.sei.cmu.edu/cert/2014/09/-finding-android-ssl-vulnerabilities-with-cert-tapioca.html` (visited on 08/14/2015).

[86] Georg Lukas. *Java/Android SSLSocket Vulnerable to MitM Attacks*. Aug. 2014. URL: `http://op-co.de/blog/posts/java_sslsocket_mitm/` (visited on 08/14/2015).

[87] William Shakespeare. *All's Well That Ends Well*. London: Arden Shakespeare, 2007.

Appendix A

Sift User Manual

Sift can be downloaded and installed from the Google Play Store from `https://play.google.com/store/apps/details?id=com.imhotepisinvisible.sslsift`.

Using Sift is straightforward. After installing and opening the app, simply press the large button on the bottom right. The VPN will now be running and monitoring TLS traffic in the background. Simply use other apps as normal. One can return to Sift to see more information about the connections apps make, but if anything requiring the user's attention arises, a notification will be generated.

Settings can be adjusted through the menu, allowing notifications to be turned off and on and packet capture to be enabled. The entire history can also be cleared through the menu.

Appendix B

Testing Scripts

B.1 Google Play Store Scraper Script (JavaScript)

```
var gplay = require('google-play-scraper');

myCats = {
    BUSINESS: 'BUSINESS',
    COMICS: 'COMICS',
    COMMUNICATION: 'COMMUNICATION',
    EDUCATION: 'EDUCATION',
    ENTERTAINMENT: 'ENTERTAINMENT',
    FINANCE: 'FINANCE',
    HEALTH_AND_FITNESS: 'HEALTH_AND_FITNESS',
    LIBRARIES_AND_DEMO: 'LIBRARIES_AND_DEMO',
    LIFESTYLE: 'LIFESTYLE',
    APP_WALLPAPER: 'APP_WALLPAPER',
    MEDIA_AND_VIDEO: 'MEDIA_AND_VIDEO',
    MEDICAL: 'MEDICAL',
    MUSIC_AND_AUDIO: 'MUSIC_AND_AUDIO',
    NEWS_AND_MAGAZINES: 'NEWS_AND_MAGAZINES',
    PERSONALIZATION: 'PERSONALIZATION',
    PHOTOGRAPHY: 'PHOTOGRAPHY',
    PRODUCTIVITY: 'PRODUCTIVITY',
    SHOPPING: 'SHOPPING',
    SOCIAL: 'SOCIAL',
    SPORTS: 'SPORTS',
    TOOLS: 'TOOLS',
    TRANSPORTATION: 'TRANSPORTATION',
    TRAVEL_AND_LOCAL: 'TRAVEL_AND_LOCAL',
    WEATHER: 'WEATHER',
    APP_WIDGETS: 'APP_WIDGETS'
};

Object.keys(myCats).forEach(function(cat) {
  gplay.list({
    category: cat,
    collection: gplay.collection.TOP_FREE,
    num: 100,
    country: 'gb'
  })
  .then(function(apps){
    apps.forEach(function(app) {
      console.log('"' + app.title + '",' + app.appId + ',market://details?id=' + app.appId);
    })
  })
  .catch(function(e){
    console.log('There was an error fetching the list!');
  });
})
```

B.2 MonkeyRunner Script (Python)

```
import glob
import subprocess
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice
import logging
logging.basicConfig(filename='log.log',level=logging.INFO,format='%(asctime)s %(message)s')

build_tools = '/path/to/android/sdk/build-tools/21.1.1/'
root = '/path/to/apk_storage/'

def has_internet_permission(apk):
    command = "aapt dump permissions " + apk + " | sed 1d | awk -F\"'\" '/name=/ {print $2}'"
    process = subprocess.Popen(build_tools + command, stdout=subprocess.PIPE, stderr=None, shell=True)
    permissions = process.communicate()[0]
    #print permissions
    if "INTERNET" in permissions:
        return True
    else:
        return False

def get_package_name(apk):
    command = "aapt dump badging " + apk + " | awk '/package/ {print $2}' | awk -F\"'\" '/name=/ {print $2}'"
    process = subprocess.Popen(build_tools + command, stdout=subprocess.PIPE, stderr=None, shell=True)
    package = process.communicate()[0]
    return package

def get_activity_name(apk):
    command = "aapt dump badging " + file + " | awk '/launchable-activity/ {print $2}' | awk -F\"'\" '/name=/ {print $2;exit;}'"
    process = subprocess.Popen(build_tools + command, stdout=subprocess.PIPE, stderr=None, shell=True)
    activity = process.communicate()[0]
    return activity

device = MonkeyRunner.waitForConnection()
```

```
files = glob.glob(root + '*/*.apk')
for file in files:
    print file
    if has_internet_permission(file):
        device.installPackage(file)
        package = get_package_name(file)
        activity = get_activity_name(file)
        runComponent = package.strip() + '/' + activity.strip()
        print runComponent
        logging.info(package)
        device.startActivity(component=runComponent)
        MonkeyRunner.sleep(10)
        device.press('KEYCODE_DPAD_CENTER', MonkeyDevice.DOWN_AND_UP
           )
        device.press('KEYCODE_DPAD_DOWN', MonkeyDevice.DOWN_AND_UP)
        device.type('asdf1234-user')
        device.press('KEYCODE_DPAD_CENTER', MonkeyDevice.DOWN_AND_UP
           )
        device.press('KEYCODE_DPAD_DOWN', MonkeyDevice.DOWN_AND_UP)
        device.type('qwer5678-pass')
        device.press('KEYCODE_DPAD_ENTER', MonkeyDevice.DOWN_AND_UP)
        device.removePackage(package)
```

Appendix C

Cipher Suite Security

The full list of cipher suites is too long to be included here. Below is a list of cipher suites deemed to be secure, created with reference to Mozilla's recommended configuration.[1] Any suites that use insecure algorithms (Export keys, MD5, DES, RC4, PSK) can be considered to offer no to very little security. The remaining cipher suites offer some security but using one of the recommended cipher suites would be highly recommended.

```
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
TLS_DHE_DSS_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256
```

[1] https://wiki.mozilla.org/Security/Server_Side_TLS#Recommended_configurations

Appendix D

Evaluated Apps

The list of apps is too long to be included in text form. A CSV of the 100 apps tested can be downloaded from `http://www.doc.ic.ac.uk/~og514/top100-080615.csv` and a CSV of the 2500 apps tested can be downloaded from `http://www.doc.ic.ac.uk/~og514/top2500-081015.csv`.