Graham Edgecombe

Detection of SSL-related security vulnerabilities in Android applications

Computer Science Tripos, Part II

Pembroke College

18th June 2014

Proforma

Name:	**Graham Edgecombe**
College:	**Pembroke College**
Project Title:	**Detection of SSL-related security vulnerabilities in Android applications**
Examination:	**Computer Science Tripos, Part II (2014)**
Word Count:	**11715**
Project Originator:	Graham Edgecombe
Supervisor:	Dr Alastair Beresford

Original Aims of the Project

The aim of the project was to develop two tools for detecting SSL-related vulnerabilities in Android applications which allow attackers to intercept network traffic with man-in-the-middle attacks. One tool detects vulnerabilities automatically by statically analysing the bytecode of Android applications, and the other attempts to exploit the vulnerabilities, allowing applications to be manually tested. Finally, I aimed to use both tools on a selection of real-world Android applications to discover how widespread the vulnerabilities were.

Work Completed

I successfully implemented both the static analysis and man-in-the-middle tools. The man-in-the-middle tool can also detect the certificate pinning, which reduces the amount of damage that the compromise of a certificate authority would cause. I also developed a number of vulnerable and secure applications using SSL to aid in the development and testing of both tools. Finally, I tested both tools on 177 applications from the Google Play Android store, identifying a number of vulnerable applications from which I could intercept a range of personal data.

Special Difficulties

None.

Declaration

I, Graham Edgecombe of Pembroke College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

Android is an operating system which is designed for use on mobile devices such as mobile phones and tablets. It is primarily developed by Google, with most of the code available under an open-source license. Android is installed on hundreds of millions of devices worldwide [1].

AppBrain has estimated that there are over 1 million applications available in the Google Play store [2], which allows users to buy applications or download them free of charge.

Mobile devices, such as those running Android, allow applications to access sensitive information such as the name, phone number or email address of the user and their contacts [3], the user's SMS messages, their calendar and their location [4]. Individual applications may also have data they wish to protect, such as passwords.

Some applications send personal information to remote servers. Whilst this in itself may not be desirable for users concerned about their privacy, another concern is that an attacker who can intercept network traffic may be able to access or modify the personal information.

Many applications use the SSL cryptographic protocol to protect sensitive network traffic. However, some applications do not use the SSL API correctly, leading to vulnerabilities which expose the applications to man-in-the-middle attacks. Man-in-the-middle attacks are typically easy to carry out on open Wi-Fi hotspots, which are commonly used by mobile devices to connect to the Internet.

Chapter 2 discusses SSL man-in-the-middle vulnerabilities and techniques for detecting and exploiting them in further detail. Chapter 3 discusses the implementation of two programs: a static analysis tool which attempts to find vulnerable Android applications automatically, and a man-in-the-middle tool which can be used to attempt to exploit vulnerable applications manually. I tested both

tools on a selection of popular applications from the Google Play store. The results are presented in Chapter 4 and my conclusions are presented in Chapter 5.

1.1 Previous work

In 2012, Georgiev et al. published a paper reviewing SSL certificate validation in non-browser software [5], identifying many applications and libraries, including some on Android, which do not perform certificate validation correctly.

Fahl et al. also published a paper in 2012 which discusses using static analysis techniques on Dalvik bytecode to check if Android applications contain vulnerable SSL validation code [6]. They ran their tool on a sample of 13,500 free applications from the Google Play store: 1,074 (8%) contained code that was potentially vulnerable to MITM attacks. They also manually checked 100 applications and were able to successfully launch MITM attacks against 41 of them.

In 2013, Egele et al. published a paper about using static analysis to discover applications which do not use the Android cryptography API correctly [7] – for example, using the electronic codebook (ECB) block cipher mode of operation.

Chapter 2

Preparation

2.1 Introduction

This chapter discusses low-level network protocols and vulnerabilities which allow attackers to intercept network traffic. The SSL protocol is commonly used to prevent such attacks, however, it can be used incorrectly by applications. The rest of the chapter discusses techniques for detecting vulnerabilities in Android applications which use SSL.

2.2 Ethernet

Ethernet is a family of computer networking standards for creating local area networks, which define the operation of both the physical layer and the data link layer.

Early versions of Ethernet connected computers with a bus topology: all computers were connected to a single cable – a shared medium. When an Ethernet frame (a packet) is sent, all other computers receive it. A media access control protocol (MAC) is used to address computers on the shared medium, so that they know if a frame they receive is destined for them. In Ethernet, each network interface card is programmed with a unique 48-bit MAC address by the manufacturer. Each frame contains a source and destination MAC address. If a network interface card receives a packet whose destination MAC address does not match the MAC address programmed into the card, it drops it.

Collisions are a problem that can arise in shared medium networks. They occur when two or more network interface cards try to transmit a frame simultaneously,

causing the signals on the cable to become garbled. Ethernet solves this problem with the use of an the carrier sense multiple access with collision detection (CSMA/CD) algorithm.

Before transmitting an Ethernet frame on a network using CSMA/CD, the network interface card will first wait for the medium to become idle, which means no other card is transmitting. The card will then start transmitting the frame. Whilst doing so, it continually listens to the signals on the wire to detect if another card has also attempted to transmit – if so, a collision has occurred. When a collision is detected, the card will send a jam signal to stop all other cards from transmitting. It waits for a random period of time defined by the binary exponentiated backoff algorithm,[1] then tries to transmit the packet again. After a maximum number of unsuccessful retries, the network card gives up.

More recent versions of Ethernet support hubs: devices which connect together multiple Ethernet cables. When a hub receives a frame on a port, it simply broadcasts it out on all other ports. This allows networks to be connected in a star or tree topology. However, hubs still have the disadvantage that frames are delivered to every computer on the network.

Switches fix this disadvantage. They are similar to hubs, however, they keep track of which MAC addresses can be reached through each port. This means an incoming frame does not need to be broadcast to every other port,[2] it only needs to be sent to a single port. New versions of Ethernet also support full duplex communication, allowing frames to be transmitted simultaneously in both directions between a network interface card and a switch. Switches store incoming frames in a buffer until the switching fabric is ready to forward them. This prevents collisions, however, frames can still be dropped if the buffer is full.

2.3 The Internet Protocol

The Internet Protocol (IP) is a protocol which operates at the network layer, on top of link layer protocols such as Ethernet. It connects multiple networks together

[1] For the nth collision, the binary exponentiated backoff algorithm picks a random number between 0 and $2^n - 1$ as the period of time to wait before transmitting again.

[2] Switches may still broadcast frames if they have not yet seen a MAC address. However, when the destination computer sends its own frame in reply, it will pass through the same switch. The switch reads the source MAC address for the reply frame and associates that with the port the frame arrived from. The next time a packet is sent to that address, a broadcast is not required, as the switch has associated the address and port.

with devices called routers. The networks IP runs on top of may have different link layer protocols and different transport and application layer protocols can run on top of IP. This is known as the 'hourglass' model and it means any application can be used with any network if they both support IP.

Each computer is given its own IP address. In IPv4, the most common version of the protocol used at the time of writing, the address is 32 bits long. However, as the Internet has grown rapidly, the pool of free IPv4 addresses has almost been exhausted. This has led to workarounds such as Network Address Translation (NAT) where a router exposes a single public IP address to the rest of the Internet, and translates packets passing through it to use a private IP address for each machine behind it, and vice-versa. IPv6, the latest version, has 128 bit addresses which should be sufficient for the foreseeable future.

When a computer sends an IP packet, it has to decide how to route the packet. On an end device, there are two possibilities: the packet is destined for another machine on the same local network, or, the packet is destined for a different network, in which case it must be delivered via the router.

On a network using Ethernet the computer must determine the MAC address of the destination computer if the destination computer is on the local network, or the MAC of the router otherwise.

2.4 The Address Resolution Protocol

The Address Resolution Protocol (ARP) is a request-response protocol used to discover the mapping between IP addresses and MAC addresses automatically. An ARP request contains an IP address and is broadcast to every computer on the local network. If a computer receives an ARP request which contains its own IP address, it sends an ARP response back to the sender, which contains the IP address and the associated MAC address.

The mappings from IP address to MAC address are cached for a period of time by each computer in an 'ARP table'. This avoids the need to send an ARP request for every outgoing packet – instead, ARP only needs to be once for each destination computer on the local network.

ARP is stateless, which means that responses can be received and processed by a computer without a corresponding request. This has some legitimate uses – for example, in a high-availability cluster of servers, if one server fails then an ARP response could be sent to update the ARP tables of all other machines so

that packets sent to the original server are sent to a backup server with a different MAC address instead.

However, as ARP responses are not authenticated, it is possible for a malicious user to send fake ARP responses, which they can use to direct packets to pass through their machine. This is known as ARP spoofing.

In an ARP spoofing attack, a spoofed ARP packet is sent periodically to the target machine, claiming that the IP address of the router resolves to the MAC address of the attacker's machine. Another spoofed ARP packet is sent periodically to the router, claiming that the IP address of the target machine resolves to the MAC address of the attacker's machine. The fake ARP packets must be sent periodically to ensure they override genuine ARP responses.

This causes the target machine to send all of its traffic destined outside the local network via the attacker's machine. It also causes the router to send all of its traffic destined to the target machine via the attacker's machine. This allows the attacker's machine to intercept and modify the traffic of any connections to/from the target machine.

IPv6 uses a different protocol for address resolution, analogous to ARP, called the Neighbor Discovery Protocol (NDP). It is also vulnerable to spoofing attacks. However, there is an NDP variant named the Secure Neighbor Discovery Protocol which uses public-key cryptography to prevent spoofing.

2.5 Wi-Fi

Wi-Fi is a standard for local area networks which operate wirelessly over an unlicensed band of the radio spectrum.

A Wi-Fi network is managed by an access point (AP). The access point periodically broadcasts its Service Set Identifier (SSID), which is the name of the network. All other devices in the network connect to the access point.

Wi-Fi networks can be open, which means no authentication or encryption is used. There are also two main schemes for authentication and encryption: WPA2-Personal, which uses a shared secret key for all users, and WPA2-Enterprise, in which each user has their own key. There are also some legacy encryption schemes such as WEP, which were found to have several security weaknesses.

Radio waves are a shared medium: packets are received by all nearby devices. Therefore, Wi-Fi uses a similar media access control protocol to Ethernet – carrier sense multiple access with collision avoidance (CSMA/CA), which is a modified version of Ethernet's CSMA/CD.

CSMA/CD cannot be used because of the hidden terminal problem, which occurs when two devices A and B are both within range of the access point, but not within range of each other. This stops the 'carrier sense' part of CSMA/CD from working. To fix this, in CSMA/CA a device must first send a Request to Send (RTS) packet and then wait to receive a Clear to Send (CTS) packet before it begins transmitting.

Another problem is trying to detect if a collision has occurred whilst transmitting a packet. This is not easy to achieve in a wireless network as a device's own transmission will drown out any transmissions it receives, due to the difference in proximity and signal strength dropping by an inverse-square relationship.

Therefore, in CSMA/CA a device must instead wait for an acknowledgement packet after transmitting. If it does not receive one within a certain length of time, it assumes a collision occurred and tries to transmit again after a delay.

As Wi-Fi is based on Ethernet, it uses the ARP protocol which means it is also vulnerable to ARP spoofing.

Such attacks are particularly concerning in open Wi-Fi hotspots, which are typically unencrypted and can be used by any person with a device supporting Wi-Fi, such as a tablet or mobile phone, to connect to the Internet – including malicious users. They are typically found in public establishments such as train and bus stations, airports and coffee shops.

Wi-Fi hotspots which use encryption but use a key that is shared freely (for example, a coffee shop where you receive the key after ordering) are also a concern, as this does not provide much of an additional barrier against preventing malicious users from connecting.

It is therefore trivial for a malicious user to connect to the network: unlike in Ethernet, where the attacker would have to physically connect their device to the network, they can carry out their attack wirelessly by connecting in the same manner as the normal users, and then use ARP spoofing to intercept traffic.

Another possible attack is for an attacker to create their own Wi-Fi hotspot – for example, it could impersonate the name of a well-known provider of hotspots (such as the name of a mobile phone company), the name of a nearby shop or just be given an enticing name such as 'Free Wi-Fi'. As the attacker operates the hotspot, all the traffic will pass through their device.

Some laptop and desktop Wi-Fi cards can run in a special mode known as 'software AP mode' which allows them to run their own access point. Hostapd [8] is a Linux program which can be used in conjunction with a supported Wi-Fi card to create a Wi-Fi access point.

2.6 SSL

Transport Layer Security (TLS), which is more commonly known by the name of its predecessor, Secure Sockets Layer (SSL), is a cryptographic protocol used to provide a secure client-server communications channel which can be used on top of an insecure network.

When used correctly, SSL provides the following guarantees:

- **Authenticity:** the server which the client is communicating with is whom it claims to be. SSL also supports client authentication, which allows the server to check the client is whom it claims to be.
- **Confidentiality:** messages cannot be read by eavesdroppers.
- **Integrity:** messages cannot be tampered with.

This means even if an attacker is able to intercept and modify network traffic, they will only be able to read the ciphertext of SSL connections. If they do tamper with the traffic, the receiver will detect that this has occurred and close the connection.

Guarantees of confidentiality and integrity depend on authenticity: if an attacker can trick a client into thinking that they are the real server, the client will perform the handshake with the attacker – establishing a shared secret key with them, not with the real server. The attacker can use the shared key to decrypt messages sent by the client and encrypt their own messages to send to the client.

The attacker can also open their own connection to the real server. As servers do not typically authenticate the identity of the client, the server will not be able to tell the difference between a connection from the client or a connection from the attacker.

By relaying data between the two connections (the client to attacker connection, and the attacker to real server connection), an attacker is able to record and modify the plaintext traffic sent between the client and real server – despite the use of SSL. The client and real server will not be aware that this has taken place. This attack is known as a man-in-the-middle (MITM) attack.

2.7 SSL certificate validation

SSL uses X.509 certificates and public-key cryptography to allow a client to authenticate the identity of the server it is communicating with, or vice versa.

SSL's public-key infrastructure scheme uses certificate authorities (CAs) to authenticate the owner of a certificate. As a minimum, a CA will confirm the owner of a certificate controls the certificate's hostname (e.g. `www.example.com`).

Each certificate authority has a root certificate. At the time of writing, Android 4.4 trusts a set of 150 root certificates by default [9]. A certificate authority declares that the subject of a certificate owns the private key corresponding to the public key within the certificate by signing it. Certificate authorities may also sign certificates belonging to other certificate authorities, for example, the certificates of its resellers or subsidiaries. This leads to a chain of trust which starts at the root certificate (the trust anchor), followed by zero or more intermediate certificates, and finally ending at a leaf certificate – the certificate used to identify an individual server or client.

When a client opens an SSL connection, a procedure called certificate validation takes place which checks that the following conditions hold:

- A trust chain exists from the leaf certificate presented by the server to one of the trust anchors.
- The Common Name (CN) or a Subject Alternative Name (SAN) in the certificate matches the hostname of the server. This prevents a certificate being usable on any server.
- The current date/time is within the certificate's validity period.

Some applications implement additional checks – for example, web browsers typically check that the certificate has not been revoked by its certificate authority and that its public key is of a sufficient length.

Applications may also trust only specific certificates or public keys, rather than relying on the chain of trust from a certificate authority. This is known as certificate pinning. Certificate pinning prevents an application from trusting a certificate signed by a certificate authority which has been compromised.

If certificate validation fails, the application may ask the user if they wish to continue connecting to the server, ignoring the errors, or the connection will fail.

Applications which do not implement certificate validation correctly are vulnerable to man-in-the-middle (MITM) attacks – an attacker can present their own self-signed certificate to the client, impersonating the server. As SSL client authentication is rarely used, the server does not need to have a vulnerable certificate validation implementation.

2.8 SSL certificate validation in Android

Android performs SSL certificate validation using two classes. An `X509TrustManager` is used to check that a certificate is signed by a trusted certificate authority and is within its specified validity period. A `HostnameVerifier` is used to check that the hostname contained within the certificate matches the hostname of the server presenting that certificate to the client.

The default implementations of both of these classes are secure. However, Android allows applications to override the defaults and use custom implementations. This offers both advantages and disadvantages: for example, applications could override `X509TrustManager` to make use of certificate pinning rather than the traditional certificate authority model. However, applications can also override these classes with insecure implementations which do not carry out sufficient checking to prevent man-in-the-middle attacks.

Android exacerbates the problem by also providing insecure implementations of these classes within its standard library – for example, a `SSLCertificateSocketFactory` can optionally bypass all certificate validation and an `AllowAllHostnameVerifier` bypasses the hostname checking.

2.9 Review of open-source Android applications

In order to understand how Android applications use these classes in a vulnerable manner, I reviewed the source code of a small selection of open-source Android applications which use network communication to try to find SSL certificate validation vulnerabilities.

I found several vulnerable applications, such as a blogging application (with 1 to 5 million installations), a note taking application (with 10,000 to 50,000 installations), an unofficial Twitter client (with 100,000 to 500,000 installations) and a Bitcoin client (which is not available in the Google Play store). All use a custom `X509TrustManager` that never throws exceptions, meaning they trust all certificates with the correct hostname, regardless of the signer. Some of them also combined this with Android's `AllowAllHostnameVerifier`, meaning they consider any certificate at all to be valid. The following listing shows the vulnerable code from the blogging application:

```
public class TrustAllManager implements X509TrustManager {
  public void checkClientTrusted(X509Certificate[] cert, String authType)
```

```
    throws CertificateException { }
  public void checkServerTrusted(X509Certificate[] cert, String authType)
    throws CertificateException { }
  public X509Certificate[] getAcceptedIssuers() { return null; }
}
```

Some speculative reasons for why so many Android applications are vulnerable to man-in-the-middle attacks might include:

- There are several snippets of insecure Android SSL code on question and answer websites such as StackOverflow, which developers copy and paste into their applications.
- Some applications may disable the certificate validation procedure in order to assist in debugging – for example, if the developer was having a problem with network communication, disabling the validation allows them to carry out a man-in-the-middle attack on themselves to check the traffic the application sends and receives is correct.
- Some developers may initially use a self-signed certificate to avoid having to pay for a certificate signed by a trusted certificate authority during development. In order to allow the use of the self-signed certificate, they disable the validation procedure. However, after they have bought a certificate to use when application is released, they forget to re-enable the certificate validation procedure.

Possible ways of improving the situation include:

- The development of a tool which performs static analysis on Android applications to detect vulnerable patterns of code. For example, a static analysis tool could be integrated into the build system used by an Android application developer (e.g. Ant, Maven, Gradle, etc.), emitting warnings or even making the build fail if vulnerable code is found.
- The development of a tool which allows man-in-the-middle attacks to be performed. This would allow application developers to test their applications themselves, using techniques similar to the techniques used by a real attacker.
- The development of a simpler SSL API which does not allow the programmer to override the certificate validation procedure with their own so that it becomes 'foolproof'.

I decided to implement the first two of these for my project: a static analysis tool to detect vulnerable patterns of SSL certificate validation code, and a program for exploiting SSL certificate validation vulnerabilities by carrying out man-in-the-middle attacks.

2.10 Review of Dalvik static analysis libraries

2.10.1 Soot

Soot [10][11] is an open-source library for performing analysis and optimisation of Java bytecode. Soot converts Java bytecode to an intermediate three-address code representation named Jimple. Soot also contains a component called Dexpler [12], which converts Dalvik bytecode, the instruction set targeted by Android applications, to Jimple.

Dalvik has 238 instructions [13] and some Dalvik instructions do not have type annotations. Jimple is simpler: it has 15 instructions, all of which have type annotations, making it more suitable for static analysis. Listing 2.1 shows an example of Dalvik bytecode. Listing 2.2 shows the equivalent Jimple code.

```
1 const/4 v4, 0x1
2 new-array v3, v4, [Ljavax/net/ssl/TrustManager;
3 const/4 v4, 0x0
4 new-instance v5, Luk/ac/cam/gpe21/DumbX509TrustManager;
5 invoke-direct {v5}, Luk/ac/cam/gpe21/DumbX509TrustManager;-><init>()V
6 aput-object v5, v3, v4
```

Listing 2.1: Sample Dalvik bytecode.

```
1 $r5 = newarray (javax.net.ssl.TrustManager)[1];
2 $r6 = new uk.ac.cam.gpe21.DumbX509TrustManager;
3 specialinvoke $r6.<uk.ac.cam.gpe21.DumbX509TrustManager: void <init>()>();

4 $r5[0] = $r6;
```

Listing 2.2: Sample Jimple code.

In addition to supporting common data-flow analyses such as live variable analysis, Soot provides a framework for carrying out user-defined data-flow analyses.

Soot also has two separate points-to analysis implementations (SPARK and Paddle [14]). Points-to analysis is explained in further detail in Section 2.11.3.

Soot can also generate call graphs, split method bodies up into basic blocks and generate control flow graphs.

2.10.2 smali

smali [15] is a Dalvik bytecode assembler and disassembler. It has an internal library named dexlib2, which reads and decodes compiled Dalvik `.dex` files.

dexlib2 is a much lower-level library than Soot. It exposes Dalvik instructions directly, rather than using a simpler intermediate representation. It does not offer higher-level functionality provided by Soot.

2.10.3 ASMDEX

ASMDEX [16] is a low-level Dalvik bytecode manipulation library, similar to dexlib2. Therefore, it also exposes Dalvik directly and, like dexlib2, does not offer higher-level functionality.

ASMDEX does not appear to be actively developed – there has only been a single release, which dates back to early 2012.

2.10.4 Summary

I chose to use the Soot library to implement the static analysis tool, for the following reasons:

- Soot's internal representation, Jimple, is simpler and higher-level than the raw Dalvik instructions exposed by dexlib2.
- Soot contains built-in high-level functionality, such as the generation of control flow graphs, data-flow analysis and points-to analysis. dexlib2 lacks this functionality.
- There is an active mailing list and community surrounding Soot, and development work such as bug fixes and improvements are still being carried out.

2.11 Static analysis techniques

2.11.1 Control flow graphs

A control flow graph is a directed graph of all the possible execution paths through a procedure.

Each instruction in the procedure is a node in the control flow graph. An edge is added between a node x and a node y if, after executing the instruction x, the instruction y may be executed.

The predecessors of an instruction x are all instructions in the control flow graph which have an edge leading towards x. The successors of x are all instructions in the control flow graph which have an edge coming from x.

2.11.2 Data-flow analysis

Data-flow analysis is a static analysis technique which can determine properties of the values of the registers at each point in a program. It works by creating a set of data-flow equations for each node in the control flow graph of a procedure in the program. The equations can be solved by starting with an approximation of the value for each node and then applying them iteratively until a fixed point is reached.

In particular, I was interested in tracking the 'vulnerability' property as it flows through the program – for example, it takes several steps in Android to go from instantiating a vulnerable `X509TrustManager` to actually creating a vulnerable SSL socket which uses it.

The data-flow equations for each node in the control flow graph are as follows:

$$\text{in-vulnerable}(n) = \bigcup_{p \in \text{pred}(n)} \text{out-vulnerable}(p) \tag{2.1}$$

$$\text{out-vulnerable}(n) = (\text{in-vulnerable}(n) \setminus \text{safe}(n)) \cup \text{vulnerable}(n, \text{in-vulnerable}(n)) \tag{2.2}$$

Where:

- in-vulnerable(n) is the set of registers which may hold a vulnerable value before n is executed. It is defined to be the empty set for the entry point of the procedure, as none of the registers can be filled with a vulnerable value at this point.

- out-vulnerable(n) is the set of registers which may hold a vulnerable value after n is executed.
- pred(n) is the set of n's predecessors in the control flow graph.
- safe(n) is the set of registers which no longer hold a vulnerable value as a result of executing the instruction n.
- vulnerable(n, v) is the set of registers which hold a vulnerable value as a result of executing the instruction n, given the set of registers, v, which hold vulnerable values before execution of n.

This is a forward data-flow analysis: the vulnerability property flows forward in the direction of the execution of the program.

The use of a union to combine the out-vulnerable(n) values of the predecessors of a node means that a register is considered to hold a vulnerable value if any path through the control flow graph leads to that register holding a vulnerable value. In Listing 2.3, the `tms` array would be considered to be vulnerable as the path taken if `debug` is true leads to `tm` being assigned a vulnerable trust manager – even though the alternate path does not assign a vulnerable trust manager to `tm`. As a result, this technique may lead to false positives.

```
1 X509TrustManager tm;
2 if (debug) {
3   tm = new VulnerableTrustManager();
4 } else {
5   tm = new SafeTrustManager();
6 }
7 TrustManager[] tms = new TrustManager[] { tm };
```

Listing 2.3: Data-flow analysis across two control flow paths.

The following example shows how the safe(n) and vulnerable(n, v) functions are defined for one Jimple instruction:

$ra = $rb (register assignment)

The register assignment instruction copies the value of $rb into $ra.

As $ra may have previously held a vulnerable value, but has now been overwritten, $ra should be in the safe(n) set so that the vulnerability property is removed:

$$\text{safe}(n) = \{\$\text{ra}\} \tag{2.3}$$

If \$rb is in the set of registers, v, which were vulnerable before the execution of n, then \$ra will also contain the same vulnerable value. In this case \$ra should be in the vulnerable(n, v) set. Otherwise, the set should be empty:

$$\text{vulnerable}(n, v) = \begin{cases} \{\$\text{ra}\} & \text{if } \$\text{rb} \in v \\ \{\} & \text{otherwise} \end{cases} \tag{2.4}$$

Appendix A contains the full set of definitions for all Jimple instructions.

2.11.3 Points-to analysis

Points-to analysis is a static analysis technique which finds the set of storage locations a pointer may point to during runtime. From the storage locations, the possible types of object that are pointed to may also be determined.

In Java all objects are allocated on the heap and accessed with references, a restricted form of pointers. Unlike pointers in a language like C, the underlying memory addresses are not exposed, which prevents pointer arithmetic from being used. The use of garbage collection over manual memory management also prevents invalid references, other than a `null` reference, from being dereferenced.

Using points-to analysis is therefore useful for static analysis of Android applications. For example, it can be used to discover the possible storage locations of a `HostnameVerifier` reference. From the storage location, the concrete type of the object pointed to can also be determined – for example, whether the reference points to an `AllowAllHostnameVerifier`, which is vulnerable, or a `StrictHostnameVerifier`, which is secure.

2.12 Man-in-the-middle attack techniques

In order to carry out a man-in-the-middle attack, the attacker needs to be able to force packets sent by the client and server to pass through a computer they control. The operating system of the computer also needs to provide functionality for passing the intercepted packets to a user-space program, which can read and modify the packets, then send them onwards to their original destination.

Techniques an attacker could use to intercept traffic so that it passes through a computer they control include setting up a fake Wi-F hotspot, or using a technique

such as ARP spoofing, DHCP spoofing or DNS spoofing on an existing network. ARP spoofing is explained in detail in Section 2.4.

Linux contains a kernel module, Netfilter [17], which provides control over how network packets are processed as they pass through the network stack. Netfilter is managed by the user-space tool iptables, which can be used to pass the intercepted packets to a user-space program.

2.12.1 Transparent proxying

Transparent proxying [18] is a feature provided by iptables which allows network traffic to be intercepted and modified by a user-space program without the source or destination IP address or port being rewritten. This means the two endpoints of the connection are not aware that it has taken place.

The iptables `TPROXY` target is used to mark incoming packets which should be transparently proxied. The marked packets are passed through a different routing table, which is configured to deliver all packets locally.

A user-space man-in-the-middle program running with root permissions can set the `IP_TRANSPARENT` option on a TCP server socket which allows it to in intercept connections opened by the client.

For a socket which is accepted by the man-in-the-middle program in this manner, the remote address is set to the original source IP address and port (the address of the client), and the local address is set to the original destination IP address and port (the address of the server).

The man-in-the-middle program can then open a new TCP client socket. By setting the `IP_TRANSPARENT` option on this socket, it is allowed to bind the local address of the socket to the address of the client. It sets the remote address to the address of the server, and can then proceed to connect to the server.

The result of this process is that the man-in-the-middle program has now got two open sockets: one connected to the client and one connected to the server. It can forward and modify traffic in both directions between the two sockets.

As the local address of the socket connected to the client is set to the address of the server, and the local address of the socket connected to the server is set to the address of the client, both the client and server believe they are directly connected to each other and do not know that transparent proxying has taken place.

2.12.2 Network address translation

Network address translation (NAT) allows the IP addresses and port numbers in packets to be modified. In particular, the iptables `REDIRECT` target can be used to change the destination IP address and port number of incoming packets to redirect them to a man-in-the-middle program running on the local machine.

As this overwrites the destination address of the packets, Linux offers a non-standard argument to the `getsockopt()` system call to recover the original destination address – `SO_ORIGINAL_DST`.

2.13 Choice of programming language

The fact the Soot library is written in Java effectively restricts the languages the static analysis tool can be written in to Java or a JVM-based language, such as Scala or Groovy.

Android applications are written in a subset of Java and it is a language that I am familiar with, therefore I chose to write the static analysis tool in Java.

I also chose to write the man-in-the-middle tool in Java. Java's standard library includes an API for parsing X.509 certificates, and for writing SSL clients and servers. Writing both tools in Java also allows common code to be shared.

2.14 Software development practices

I decided to use the Git version control system to keep track of changes to the source code. I backed up the Git repository to an external disk once a week.

I used an iterative and incremental development model, in which there are many iterations of the complete development cycle (design, implementation and testing). Each iteration incrementally adds new features. For example, the first iteration of the man-in-the-middle tool was a simple proxy server. The second added SSL support using a fixed certificate. The third added on-the-fly certificate generation. The fourth added support for intercepting connections, and so on.

2.15 Summary

This chapter discussed network protocols such as Ethernet and how attacks such as ARP spoofing can be used to intercept network traffic. The SSL protocol is

commonly used to prevent such attacks, however, some Android applications use it incorrectly. Techniques for detecting SSL certificate validation vulnerabilities by performing static analysis and carrying out man-in-the-middle attacks were also discussed.

Chapter 3

Implementation

3.1 Introduction

This chapter discusses the implementation of two tools: one which analyses the bytecode of Android applications statically to find potential SSL certificate validation vulnerabilities. The other carries out man-in-the-middle attacks to attempt to exploit such vulnerabilities.

3.2 Static analysis tool

3.2.1 Overview

The library used for the static analysis tool, Soot, places various constraints on the structure of a program which uses it. Using the strategy pattern and the visitor pattern, Soot controls the entire process of analysing and transforming the bytecode, and calls back into your own program's code at the appropriate points.

Soot operates on the bytecode in a number of different phases. Each phase consists of a number of transformers. A transformer can perform either a whole program analysis or analyse single procedures at a time. There are various built-in transformers, and users of the Soot library can also define their own transformers and add them to a phase.

The most important phases, in the order they are executed, include:

- **Jimple Body Creation (jb)**, which loads Dalvik bytecode from an `.apk` file and converts it to Soot's Jimple intermediate representation. This phase also performs tasks such as the removal of unreachable code and dead assignments.

- **Whole Jimple Pre-processing Pack (wjpp)**, which does nothing by default but allows users to run their own inter-procedural (whole program) transformers before the call graph is created.
- **Call Graph Construction (cg)**, which uses the SPARK or Paddle library to perform points-to analysis and create a call graph.
- **Whole Jimple Transformation Pack (wjtp)**, which does nothing by default. Like the Whole Jimple Pre-processing Pack, it is intended for users to be able to run their own whole program transformers.
- **Whole Jimple Optimization Pack (wjop)**, which performs various whole program optimisations that are built into Soot.
- **Jimple Transformation Pack (jtp)**, which does nothing by default. It allows users to be able to run their own intra-procedural transformers.
- **Jimple Optimization Pack (jop)**, which performs various intra-procedural optimisations that are built into Soot, such as constant propagation and further removal of dead assignments and unreachable code.

Soot does not offer fine-grained control over the order in which custom transformers added to a pack are executed. Therefore, within the Whole Jimple Transformation Pack – the phase in which most of my analysis is run – I add a single transformer, which in turn calls the classes for different kinds of analysis the tool runs in a well-defined order.

3.2.2 Locating vulnerable `TrustManager` and `HostnameVerifier` implementations

Soot allows various objects it uses to be 'tagged' with auxiliary data. I make use of this functionality to tag `TrustManager` and `HostnameVerifier` implementations as one of:

- **Vulnerable**: if the class is definitely vulnerable.
- **Safe**: if the class is definitely not vulnerable.
- **Unknown**: if the tool doesn't know if the class is vulnerable or safe.

The implementations of these classes that are built into Android are trivial to identify by the package and class name, and are tagged with either vulnerable or safe.

A custom implementation of `HostnameVerifier` containing a `verify()` method which always returns true is tagged as vulnerable, as it considers any trusted certificate valid, regardless of if the hostname within the certificate is correct or not. The tool identifies such implementations by checking if every exit node in the control flow graph returns true.

A custom implementation of `TrustManager` containing a `checkServerTrusted()` method that never throws a `CertificateException` is tagged as vulnerable, as it considers any certificate valid, regardless of if it was signed by a trusted certificate authority or not.

Any remaining untagged implementations of either class are tagged as unknown.

3.2.3 Data-flow analysis in Soot

Soot's data-flow analysis framework contains several different types of 'flow set' – the data structure used to store sets of registers at each point in the program. In my tool, the set contains all registers pointing to a vulnerable object. The most suitable data structure depends on whether you expect the set to be sparse (arrays) or dense (bit sets). The complement operation is only available for bit sets.

As relatively few registers will be marked as vulnerable (most methods in an application will have no vulnerable registers at all because they do not contain any SSL-related code), and as the flow equations I used do not require the complement operation, I used the sparse array flow set implementation.

Soot's library carries out parts of the data-flow analysis – such as the fixed point iteration; however, the user must implement methods for:

- Computing the set of registers possessing the desired property at the entry node of the control flow graph.
- Merging the sets of registers for an instruction with multiple predecessors.
- Calculating the changes to the set of registers caused by the execution of an instruction.

The first two are simple: at the entry point the analysis assumes that no registers point to a vulnerable object, so the sets are initialized as the empty set. This does limit the tool to only detecting vulnerable code when it is contained

entirely within a single method and may lead to false negatives. Merging the registers is performed with a union operation, in order to consider an application vulnerable if only one possible path of execution is vulnerable.

Calculating the changes to the sets of registers is more complicated, and involves translating the data-flow equations into code. Soot supports the visitor design pattern on instructions, which allows different decisions based on the type of instruction (such as assignment, method invocation, etc.) to be made – in this case, deciding which registers to add and remove from the set of vulnerable registers.

Appendix B shows an example of how a data-flow analysis equation has been converted to the Java code for Soot's data-flow analysis framework.

3.2.4 Points-to analysis in Soot

The API Soot exposes for points-to analysis is simple: given a reference to a local variable or field, it will return a points-to set, which represents the set of objects that variable or field might point to a runtime. However, Soot does not provide a user-facing API for iterating over the objects in the set. Instead it provides a limited number of operations, such as checking if it is empty, checking if two points-to sets have a non-empty intersection and returning a list of the possible types of the objects in the set.

I used points-to analysis, instead of data-flow analysis, for simple cases where only a single method call is required for code to be vulnerable. For example:

```
HostnameVerifier hv = ...;
HttpsURLConnection.setDefaultHostnameVerifier(hv);
```

The points-to analysis can be used to find the possible types the `hv` variable may point to at runtime to determine if the code uses a vulnerable `HostnameVerifier` implementation.

3.3 Man-in-the-middle tool

3.3.1 Overview

At a high level, the man-in-the-middle tool loads an RSA key pair and certificate from the disk, which are later used to sign the spoofed certificates. It creates a server-side SSL socket and then listens in an infinite loop for incoming connections.

Connections are intercepted by either setting up a Wi-Fi hotspot with the Hostapd [8] software or with ARP spoofing using the `arpspoof` program from the Dsniff [19] suite of network tools. iptables rules are used to divert the intercepted connections to the man-in-the-middle application.

When a connection is accepted, a task which performs the SSL handshake is run on a dynamically-sized thread pool – threads which have finished running a task are re-used, or, if there are no spare threads, a new thread is spawned and added to the pool.

Performing the SSL handshake involves finding the IP address and port of the server that the client intended to connect to. The man-in-the-middle tool makes a connection to the server and attempts to perform an SSL handshake to detect if the server uses SSL. If so, it retrieves a copy of the real X.509 certificate, generates and signs a spoofed X.509 certificate based on the real one, switches the socket between the client and man-in-the-middle tool to using SSL and presents the fake certificate to the client. If the server uses plaintext these steps are skipped.

After performing the handshake, it runs two more tasks on the thread pool: one which relays data from the client to the server, and one which relays data from the server to the client. Both of these tasks also log the data they intercept. The handshake task terminates at this point. The subsequent two tasks terminate when the real client or server closes the socket.

The rest of this section describes the implementation of each of these steps in further detail.

3.3.2 Finding the IP address of the client's intended destination

As discussed in Section 2.12, there are two different techniques for using iptables to redirect traffic to the man-in-the-middle tool: transparent proxying and network address translation. The tool implements support for both intercept techniques and allows the user to choose between them at runtime as they are both useful in different configurations.

As transparent proxying does not rewrite addresses, the local address of the socket is the original address the client intended to connect to.

Network address translation is more complicated as it requires reading the `SO_ORIGINAL_DST` option of the socket with the `getsockopt()` C library function. Java's standard library only supports a small subset of cross-platform socket options, therefore the program must make the call into native code itself.

I used the Java Native Access (JNA) [20] library to do this. Unlike the Java Native Interface (Java's built-in functionality for interfacing between Java and native code), JNA does not require C or C++ code to be written. Instead, the programmer translates the declarations of the functions and structures in the C header files into Java declarations and calls them as if they are normal Java functions.

For example, Listing 3.1 shows the declaration of the C `getsockopt()` function on my machine.

The `int` type in C is mapped directly to the `int` type in Java by JNA. Pointers, such as the `void *` pointer, are mapped to JNA's `Pointer` class.

The `socklen_t *` pointer can be treated in a simpler manner. `socklen_t` is an alias for the `int` type on my system – therefore, the `socklen_t *` pointer behaves exactly as if it was an `int *` pointer.

JNA has an `IntByReference` class which contains methods for reading and writing an integer to the memory location pointed to by an `int *` pointer – the read method automatically takes care of reading the 4 bytes of memory at that location, and performing bit shifting and bitwise OR operations to convert the 4 bytes to an `int`. Similarly, the write method automatically takes care of splitting an `int` into 4 bytes with bit shifting and bitwise AND operations, and then writes the 4 bytes to the location in memory pointed to by the pointer.

The `Pointer` class could be used in place of `IntByReference`, but this would be more cumbersome as the operations for reading and writing an integer would have to be performed manually.

The function name, return type and argument types must match. However, the argument names may be different.

Listing 3.2 shows how to map the C `getsockopt()` declaration to a Java declaration suitable for use with JNA:

The `Native.loadLibrary()` call takes the name of the shared library the function is contained within. JNA appends the prefix 'lib' and the suffix '.so' to the name of the library to find the name of the file containing the library – producing `libc.so` in this case. It returns a class implementing the `CLibrary` interface. Calling the `getsockopt()` method on this class will make JNA make a native call to the equivalent function in `libc.so`, automatically translating the arguments from their Java representation to their C representation, and doing the converse for the return value.

C structs are converted to a Java class extending JNA's `Structure` class in a analogous manner to converting C functions. For example, Listing 3.3 shows the

```
1 #define SOL_IP 0
2 #define SO_ORIGINAL_DST 80
3
4 extern int getsockopt (int __fd, int __level, int __optname,
5                         void *__restrict __optval,
6                         socklen_t *__restrict __optlen) __THROW;
```

Listing 3.1: C getsockopt() declaration.

```
1 public interface CLibrary extends Library {
2   public final CLibrary INSTANCE = (CLibrary)
3     Native.loadLibrary("c", CLibrary.class);
4
5   public final int SOL_IP = 0;
6   public final int SO_ORIGINAL_DST = 80;
7
8   public int getsockopt(int socket, int level, int option_name,
9                         Pointer option_value, IntByReference option_len)
10                        throws LastErrorException;
11 }
```

Listing 3.2: Java getsockopt() declaration.

```
1 public final class sockaddr_in extends Structure {
2   public short sin_family;
3   public byte[] sin_port = new byte[2];
4   public byte[] sin_addr = new byte[4];
5   public byte[] sin_zero = new byte[8];
6
7   @Override
8   protected List getFieldOrder() {
9     return Arrays.asList("sin_family","sin_port","sin_addr","sin_zero");
10  }
11 }
```

Listing 3.3: Java sockaddr_in declaration.

translation of the `sockaddr_in` struct to Java.

The field order must be specified manually as the Java reflection API does not guarantee that it will return fields in the order in which they were declared.

The `getsockopt()` call takes the socket's file descriptor as one of its arguments. Java does not directly expose the underlying file descriptor of a high-level `Socket` or `SSLSocket` object, therefore I used Java's reflection API to find the reference to the socket's `FileDescriptor` and then read the value of its `fd` field, which contains the underlying file descriptor as an integer.

Finally, the tool converts the fields of the `sockaddr_in` struct into an `InetSocketAddress`, the class Java uses to represent the combination of an IP address and port number in its standard library.

This required particular care with regards to the endianness of integer fields in the C structure and in the Java object: C fields are big or little endian, depending on the CPU architecture, however, Java fields are always big endian regardless of the CPU.

JNA will automatically convert arguments and struct fields from big endian to little endian and vice-versa on little endian systems, such as the x86 machine I used. However, this automatic conversion actually produces the incorrect result on such a machine for the port number in the `sockaddr_in` structure, as the structure is defined to always use 'network' (big endian) byte order.

To work around this problem I represented the port number in the Java `Structure` as a 2 byte field, and combine the bytes into a short integer manually. This produces the correct result on all CPUs.

3.3.3 Spoofed certificate generation

The cryptography API in Java's standard library does not support the generation of X.509 certificates. Therefore I used the open-source Bouncy Castle cryptography library [21] to generate X.509 certificates and RSA key pairs.

Generating a 2048-bit RSA key pair takes around half a second on my desktop machine. This delay can be perceived by an application whose traffic is being intercepted, therefore the program generates a single key pair on startup and uses the same key pair for every spoofed certificate.

To generate the spoofed certificate, the man-in-the-middle program first opens an SSL socket to the IP address and port of the real destination server and performs the SSL handshake with the server. The real server presents its certificate chain to the man-in-the-middle program as part of the handshake.

The program reads the information it needs to spoof from the leaf certificate of the certificate chain, which is the certificate identifying the individual server – the others belong to certificate authorities.

The subject of the certificate is an X.500 distinguished name (DN). Each distinguished name contains several relative distinguished names (RDNs). An RDN consists of a type and a value. Examples of RDN types include: C (the subject's country), O (the subject's organization) and CN (the subject's common name).

For example, the subject of the SSL certificate used on `https://www.cl.cam.ac.uk/` is:

```
CN=www.cl.cam.ac.uk,
OU=Computer Laboratory,
O=University of Cambridge,
C=GB
```

In X.509 certificates used for SSL, the common name RDN contains the hostname of the server that is associated with the public key in the certificate. This is not the intended use of the common name field as defined by the X.500 standard – storing a hostname inside it has been deprecated by RFC 2818 [22] and the CA/Browser Forum's Baseline Requirements [23]. However, it is still present in many certificates for compatibility with older SSL clients.

The replacement for the common name field is the subjectAltName extension, which allows several different hostnames to be associated with a single certificate, as well as other types of identifier (such as IP addresses and e-mail addresses). Each identifier in the subjectAltName field is known as a SAN (subject alternative name).

This permits a single certificate to be used on multiple hostnames – for example both the www and non-www variant of a domain name (`www.example.com` and `example.com`).

The common name and subject alternative names which contain hostnames are copied into the spoofed certificate. Several extensions are added to the spoofed certificate which are required to specify how it may be used:

- **basicConstraints** with CA bit set to false, which means the certificate is intended to be a leaf certificate – not a certificate authority.
- **subjectKeyIdentifier** and **authorityKeyIdentifier** which contain the SHA-1 hash of the certificate's public key and the SHA-1 hash of the public key used to sign the certificate respectively.

- **keyUsage**, which specifies how the certificate's public key may be used.

  If Diffie-Hellman key exchange is used for the handshake, the public key is used to sign the Diffie-Hellman messages sent from the server to the client. This requires the 'signing' usage to be set.

  If Diffie-Hellman is not used for the handshake, the client generates the shared secret key, encrypts it with the server's public key and sends it to the server. This requires the 'key encipherment' usage to be set.

- **extendedKeyUsage**, which specifies the certificate's public key may be used for TLS (SSL) web server authentication.

Once the fields of spoofed certificate have been populated, it is signed by a certificate authority certificate. I created two of my own CA certificates: one is untrusted by the client, which allows `TrustManager` vulnerabilities to be exploited. The other is trusted by the client, by installing it in Android's list of certificate authorities. This allows vulnerabilities which do not require a vulnerable `TrustManager` to be exploited, such as `HostnameVerifier` vulnerabilities. A command line flag passed to the man-in-the-middle program is used to select which CA certificate should be used.

3.3.4 Server Name Indication

Due to the exhaustion of IPv4 addresses, HTTP virtual hosting was developed to allow several websites to be hosted on a server using a single IP address (previously, each website would need its own IP address for the web server to distinguish between them).

SSL has a similar scheme to HTTP virtual hosting known as Server Name Indication (SNI). SNI sends the hostname the client is connecting to as part of the SSL handshake.

This cannot be performed in the application layer protocol running on top of SSL – SNI is required because the server needs to know the hostname during the SSL handshake, before the application layer protocol has started, so that the correct SSL certificate with a matching hostname can be selected.

Android's API has client-side SNI support. Therefore, the man-in-the-middle tool needs to be able to forward the SNI hostname sent by the client to the real server. This allows the real server to send back the correct certificate, if it uses SNI, which in turn allows the man-in-the-middle tool to spoof the correct certificate.

Java has offered client-side SNI support since Java 7. Server-side SNI, which is required for the man-in-the-middle tool, is only present in Java 8. The SNI hostname can be fetched from the SSL socket's 'handshake session' object.

3.3.5 Graphical User Interface

Whilst I originally intended for the man-in-the-middle tool to be a command-line program, it became clear during development that it was hard to distinguish the traffic intercepted from several simultaneous connections. It was also hard to distinguish which packets were sent from the client to server, and which packets were sent from the server to client.

Therefore, I developed a graphical user interface using Java's Swing GUI library. Figure 3.1 shows a screenshot of the GUI. The left hand side contains a list of all the connections that have been intercepted, colour-coded by their current state. The right hand side contains three tabs: one containing information about the selected connection (such as the IP address of the client and server, the SSL version and negotiated cipher suite and the hostname within the SSL certificate), one containing the intercepted data that was sent by the client to the server, and the final one containing intercepted data that was sent by the server to the client.

Command-line operation is still supported, as it is required for running the automated test suite.

3.4 Summary

This chapter discussed how vulnerable classes are identified by the static analysis tool, and how Soot's data-flow and points-to analyses can be used to discover if vulnerable classes are used. It also discussed how the man-in-the-middle tool intercepts spoofed traffic and generates fake SSL certificates on the fly.

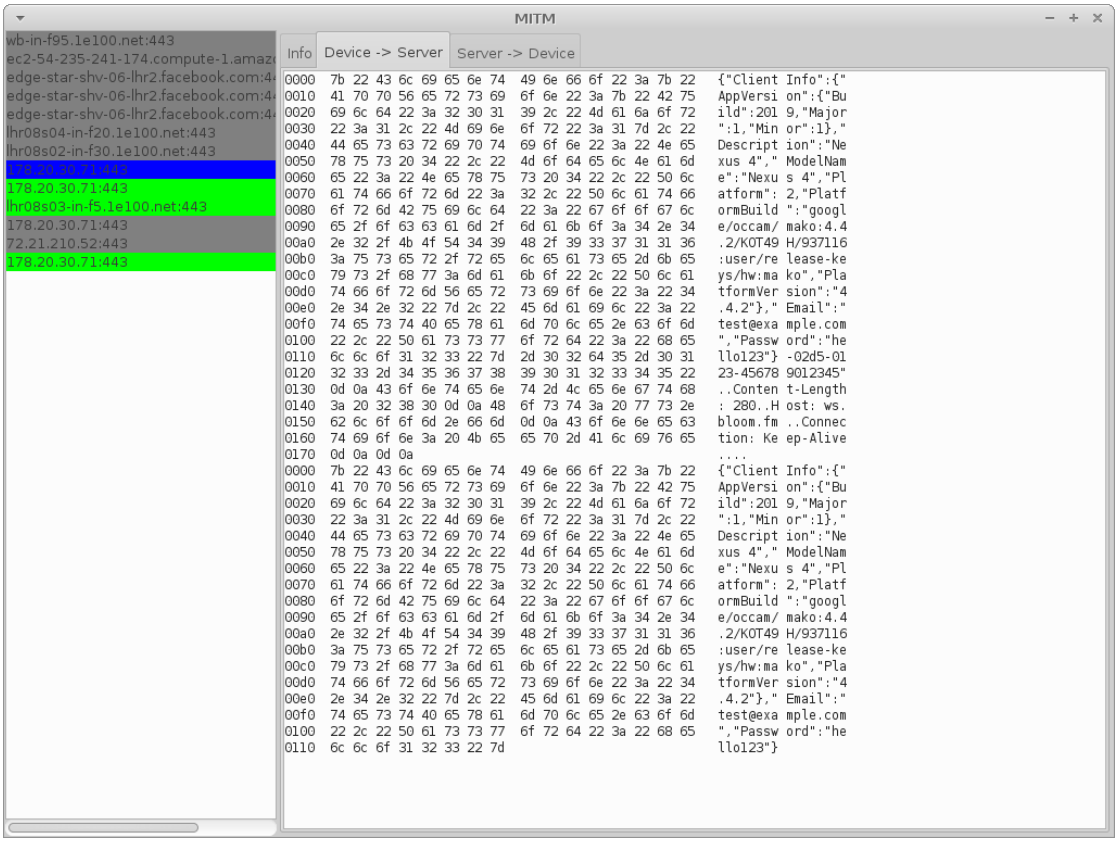

Figure 3.1: Screenshot of the tool's graphical user interface.

Chapter 4

Evaluation

4.1 Introduction

This chapter discusses how I performed automated testing during development. It also presents the results of testing both of the tools I developed against 177 real-world Android applications.

4.2 Automated testing

I wrote a small number of unit tests to test parts of both tools. However, Soot uses lots of global state held in singleton classes, which made writing unit tests for the static analysis tool difficult.

As the man-in-the-middle tool has several command line options that can be set in a large number of combinations for exploiting different kinds of vulnerability, I wrote a shell script to run a suite of black-box integration tests to ensure it operates correctly for each combination of options.

The shell script uses two 'echo' servers which both accept connections, read packets received from the client and then send the same packets back to the client. One used SSL, and the other used plaintext.

The script runs several clients: all of them connect to the server, send a single byte to the server and then wait for the server to reply with the same byte to check the connection is working. One client uses plaintext, one uses SSL and checks if the Server Name Indication (SNI) hostname is forwarded correctly by the man-in-the-middle tool. The final client uses SSL and can be configured to run in various secure and vulnerable modes: it can verify the certificate is trusted

using the standard certificate authority model, use certificate pinning or not verify the certificate is trusted at all. It can also be configured to verify if the hostname matches or skip the hostname verification.

The script checks if the clients and servers can successfully communicate with each other when the man-in-the-middle tool is not running. All of them are expected to succeed in this case.

It also checks if the clients and servers are able to communicate with each other, with the appropriate iptables rules in place to direct traffic via the man-in-the-middle tool. It exhaustively tests every combination of client vulnerability and man-in-the-middle mode. Some cases are expected to fail: for example, a client which performs SSL validation correctly being presented a spoofed certificate by the man-in-the-middle tool should reject the connection. Other cases are expected to succeed: for example, a client which doesn't perform hostname verification should accept a trusted certificate with an incorrect hostname.

There are 38 different test cases which take 70 seconds to run. All the test cases pass in the final version of the program.

4.3 Corpus of real-world test applications

I used a selection of 177 Android applications which were downloaded from the Google Play Store to test my project on real-world applications.

The applications were downloaded by my supervisor over a 24 hour period starting on the 23rd October 2013. Due to the Google Play Store's rate limiting, the applications were downloaded over several sessions.

Free applications were selected from the following categories:

- All applications in the top charts (top free apps, top grossing apps, top free games and top grossing games).
- All new releases (top free apps, top grossing apps, top free games and top grossing games).
- All 40 applications in the editors' choice.
- The top five applications from each of the 34 categories.

The applications were downloaded from an IP address within the University of Cambridge, therefore the applications are ones which Google displays to users in the UK.

Result	Number of applications	Percentage
Vulnerable	96	62.7%
Safe	46	30.1%
Unsure	11	7.2%

Table 4.1: The number of applications for each static analysis result, excluding the 24 applications which Soot failed to analyse.

The applications were installed onto a Nexus 4 phone, which was factory reset before downloading the applications with a brand new Google account.

4.4 Static analysis tool

I ran the static analysis tool against all 177 applications downloaded from the Google Play Store. The process is completely automated: a shell script iterates over each application's `.apk` file, passes that to the static analysis tool and records the output of the tool in another file. Appendix C shows a sample of the output for a vulnerable application. Finally, the shell script aggregates all the data into a format suitable for loading into a spreadsheet application for further analysis.

When using Soot's default points-to analysis implementation, SPARK, the tool fails to analyse 80 applications (45%) due to exceptions thrown within Soot – at the time of writing there is no stable Soot release with Dalvik bytecode support, and it appears that there are still some remaining bugs.

I therefore switched to using Paddle, an alternative points-to analysis implementation for Soot. When using Paddle, the static analysis fails on fewer applications – only 24 (14%).

Automatically testing an application can lead to three results:

- **Vulnerable:** the application definitely contains vulnerable code.
- **Safe:** the application definitely does not contain vulnerable code – either as a result of using the Android SSL API safely, or by not using SSL at all.
- **Unsure:** the application contains SSL-related code, but the tool does not know if it is vulnerable or not.

Table 4.1 shows the number of applications that were classified as vulnerable, safe and unsure.

Vulnerability	Number of applications	Percentage
`HostnameVerifier` only	6	6.3%
`TrustManager` only	68	70.8%
`HostnameVerifier` and `TrustManager`	22	22.9%

Table 4.2: The 96 applications classified as vulnerable by static analysis, split by the type of vulnerability.

The static analysis tool detects two different types of vulnerability: the implementation or usage of a `HostnameVerifier` which does not perform hostname validation correctly, and the implementation or usage of a `TrustManager` which does not check if the certificate has been signed by a trusted certificate authority. An application can also be vulnerable to both vulnerabilities, which means it does no SSL certificate validation at all and accepts any certificate. Table 4.2 shows the number of applications which were classified as vulnerable, split by the type of vulnerability.

It takes 3 hours and 20 minutes for the tool to analyse the complete set of 177 applications on a machine with a 4 core 3.3 GHz Intel i5-2500 processor and 8 GB of RAM. The mean time to analyse an individual application is 89.97 seconds. Figure 4.1 shows the distribution of analysis times as a histogram.

The adjusted Fisher-Pearson standardized moment coefficient, which is a measure of skewness, is 3.28. This indicates that there is a positive skew: most applications take a short amount of time to analyse and there is a long tail of applications which take a long time to analyse.

The majority of the time spent performing the analysis is within Paddle's points-to analysis, which uses the Zhu/Calman/Whaley/Lam algorithm [24].

Figure 4.2 plots the logarithm of the static analysis runtime against the size of an application's bytecode, which is estimated by the size of the `classes.dex` file. The coefficient of determination, R^2, is 0.78. This shows that there appears to be an exponential relationship between the size and analysis runtime.

4.5 Manual testing

I also manually tested the set of 177 applications by running them with the man-in-the-middle program configured to intercept their traffic. I spent a few minutes using each application, performing actions to try to trigger SSL network communication.

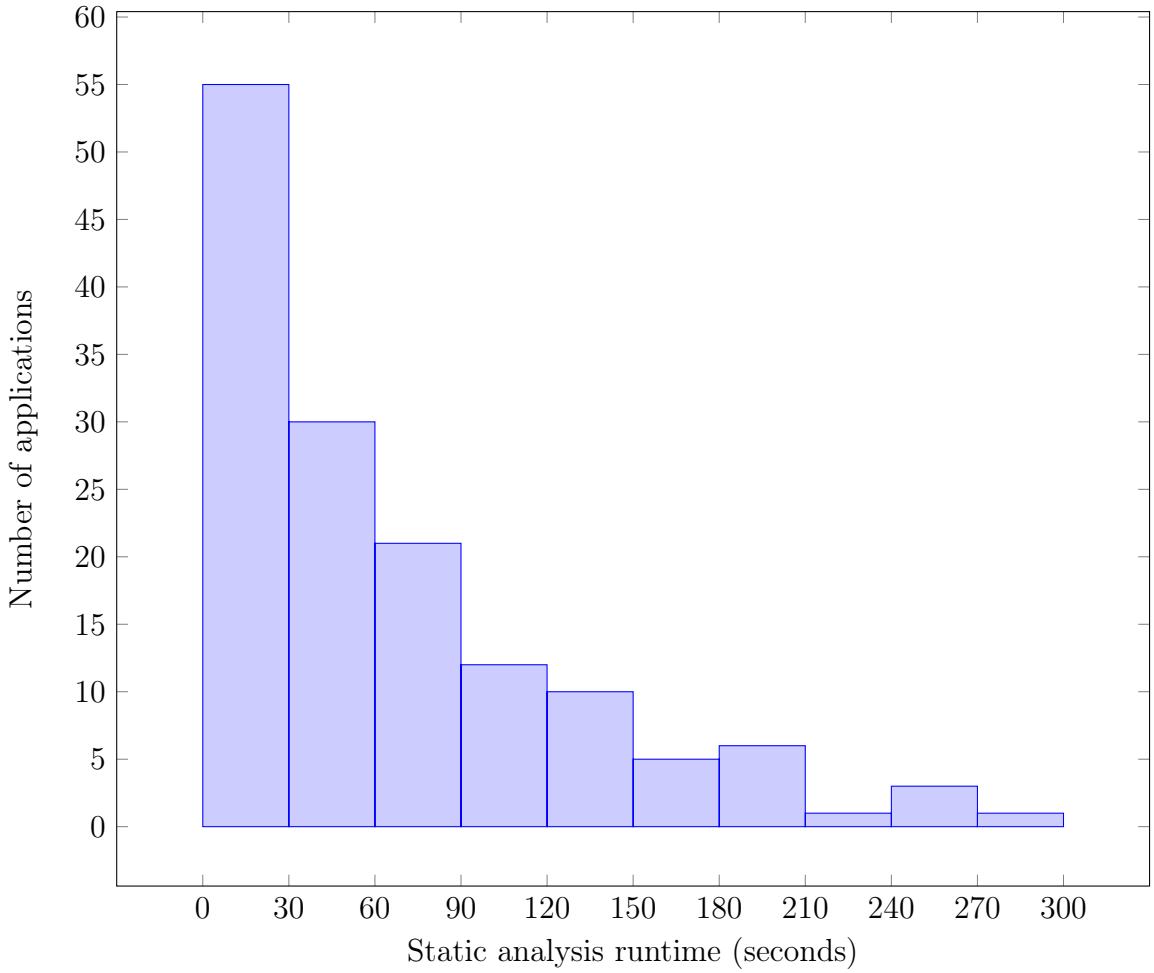

Figure 4.1: Histogram showing time taken to run static analysis on each application, split into bins of 30 seconds each. (9 applications took over 300 seconds to analyse and are not included in the histogram.)

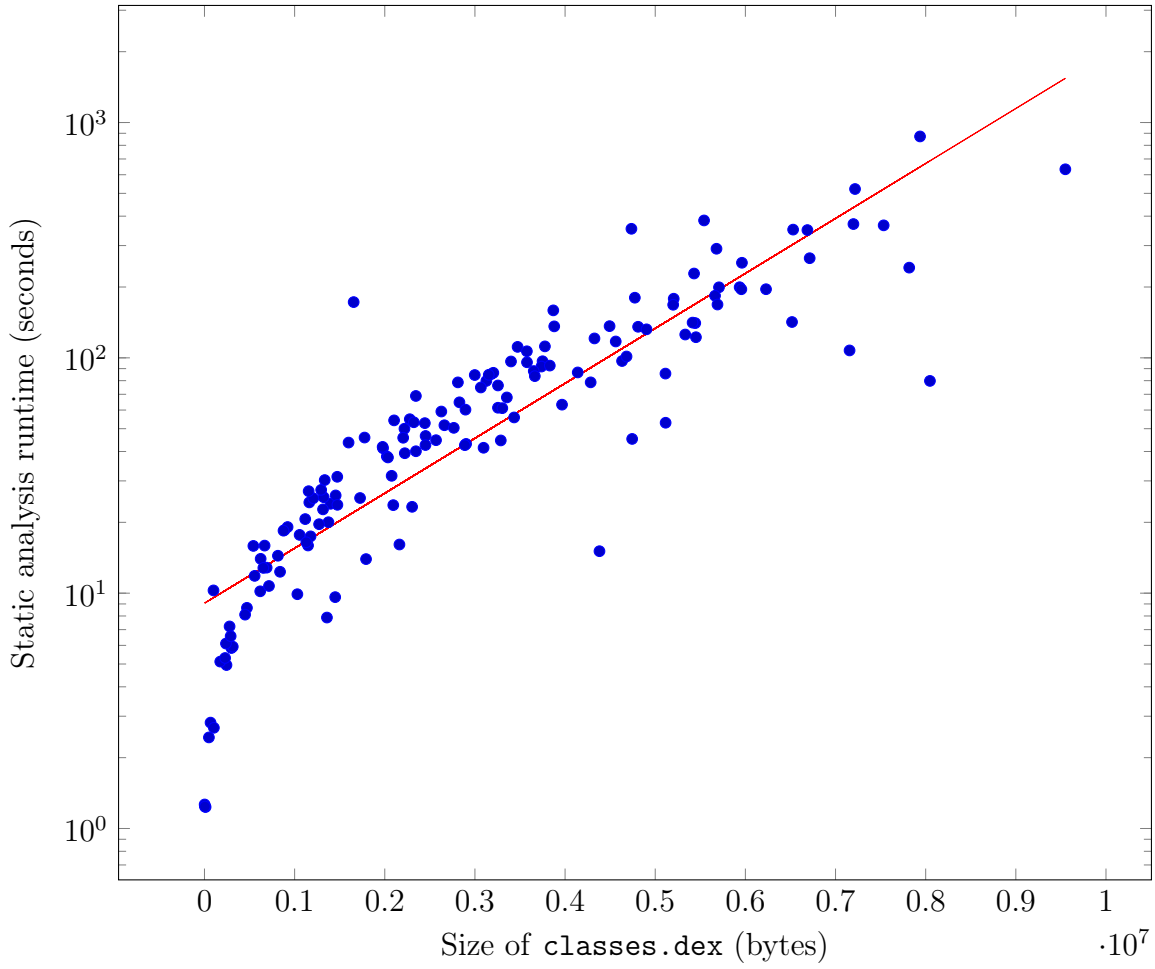

Figure 4.2: Plot of static analysis runtime against size of the application's bytecode. The regression line has the equation $y = \exp(5.38x \times 10^{-7} + 2.20)$.

Result	Number of applications	Percentage
Vulnerable	28	15.8 %
Unsure	149	84.2 %

Table 4.3: The number of applications for each manual testing result.

Vulnerability	Number of applications	Percentage
`HostnameVerifier` only	9	32.1 %
`TrustManager` only	1	3.6 %
`HostnameVerifier` and `TrustManager`	18	64.3 %

Table 4.4: The 28 vulnerable applications found by manual testing, split by the type of vulnerability.

I tested 97 applications from within the Android emulator, which was configured to run the x86 version of Android 4.4.2. Each application was tested in a fresh instance of the emulator to ensure they were isolated from each other.

The remaining 78 applications did not work correctly in the emulator – some applications use native code and thus require the ARM version of Android, and some use OpenGL ES, which was not supported by the video card drivers on my computer. I tested the remaining applications on my Google Nexus 4 phone. I formatted the phone and installed a clean copy of Android 4.4.2 before using it for testing. After testing each application, I uninstalled it before installing the next one.

Manually testing an application can lead to two results:

- **Vulnerable:** the application is definitely vulnerable as it was successfully exploited with the man-in-the-middle tool.
- **Unsure:** the application could not be successfully exploited with the man-in-the-middle tool. However, the application may be vulnerable as it is unlikely that the manual testing ends up executing all possible paths of execution in the application under test.

Table 4.3 shows the number of applications that were classified as vulnerable and unsure in manual testing. Table 4.4 contains only the applications that were classified as vulnerable during manual testing, split by the types of vulnerability previously described in Section 4.4.

	Vulnerable	**Unsure**
Unpopular	22	115
Popular	6	34

Table 4.5: 2×2 contingency table of manual testing result against application popularity.

4.6 Certificate pinning

By serving a certificate signed by a trusted certificate authority and with the correct hostname to the client, the man-in-the-middle tool can be used to check if application under test uses certificate pinning. An application which uses certificate pinning is more secure if a certificate authority is compromised, as it only trusts a single certificate and thus will not trust certificates signed by the compromised CA.

I only found three applications out of 177 which used certificate pinning: the Barclays mobile banking app, Twitter (a social network) and WhatsApp (an SMS replacement).

4.7 Popularity of applications

In order to test if popular applications are less likely to be vulnerable I used the Fisher's exact statistical significance test.

I split the applications into two categories: unpopular (installed on less than 50 million devices) and popular (installed on 50 million or more devices). Table 4.5 shows the manual testing results split by the two popularity categories.

The null hypothesis is that there is no relationship between popularity and chance of an application being vulnerable. The alternative hypothesis is that popular applications are less likely to be vulnerable – i.e. more applications would lie in the top left and bottom right cells of Table 4.5.

The one-tailed p-value obtained by Fisher's exact test for this data is $p = 0.65$. This is greater than 5% and therefore is not sufficient to reject the null hypothesis.

		Static Analysis			
		Vulnerable	**Unsure**	**Safe**	**Failed**
Manual Testing	**Vulnerable**	21	1	0	6
	Unsure	75	10	46	18

Table 4.6: Number of applications split by both their static analysis result and manual testing result.

4.8 Comparison between static analysis and manual testing

Table 4.6 shows the number of applications for each possible combination of static analysis and manual testing result.

Only a single application which was determined to be definitely vulnerable by manual testing was classified as 'unsure' by the static analysis tool – the rest are classified as vulnerable by the static analysis tool. None of them were classified as safe, which shows the results from the two tools are consistent with each other.

However, there are 75 applications which are classified as vulnerable by the static analysis tool, but which could not be exploited by the man-in-the-middle tool and are therefore classified as 'unsure'. It is possible that some of these applications are not actually vulnerable during normal usage, but unsafe SSL code exists for development purposes and is used only when a debugging flag is set. One potential improvement to the static analysis tool would be to try to detect if vulnerable code can never be called in normal usage, to try to reduce the overestimation of the number of vulnerable applications.

4.9 Responsible disclosure of security vulnerabilities

Out of the 28 applications identified as vulnerable by manual testing, I was able to intercept a range of sensitive and personal data from 9 of them – such as the user's e-mail address, username, password, postal code, gender and date of birth. One of the 9 applications used plaintext. The remaining applications used SSL but did not perform SSL certificate validation correctly.

In particular, two applications used an embedded web browser, which did not perform SSL certificate validation correctly. They used OAuth within the browser

to allow the user to sign into their application with the username and password of a selection of popular third-party services such as Google, Twitter and Instagram – allowing the user's password for the third-party authentication service to be intercepted. This is a rather serious vulnerability as a user's Google password is particularly useful to the attacker if they use Google Mail – as an attacker could use e-mail recovery to gain access to accounts for many other services used by the user.

The three most popular of these applications have between 10 million and 50 million installs each.

The security community has largely adopted the responsible disclosure model for disclosing security vulnerabilities. In accordance with this model, I sent emails to developers of each of these applications, with detailed information about the security vulnerability, its impact and how it can be fixed. I also promised to not reveal details publicly for 3 months to give them time to develop and roll out a fix.

At the time of writing, two applications have been fixed and rolled out to users already, and fixes for four others are being developed. I have not received responses from the remaining three developers despite several attempts to contact them.

4.10 Summary

This chapter discussed the automated unit and integration testing I performed. It also presented the results of testing both the static analysis tool and the man-in-the-middle tool against 177 real-word Android applications from the Google Play. Finally, it discussed some particularly interesting vulnerabilities in more detail and my approach to disclosing vulnerabilities to application vendors.

Chapter 5

Conclusion

In conclusion, a significant number of Android applications I tested – 96 out of 177 – contained vulnerable SSL certificate validation code. I was able to successfully execute a man-in-the-middle attack against 28 applications. Out of those 28 applications, I identified 9 which sent personal and sensitive data with the vulnerable code.

Tools like the ones I have developed could be used by developers and publishers of Android applications to help identify potential vulnerabilities before the application is released to users. For example, developers could run the static analysis tool as part of an automated build process, or a publisher could run it on applications submitted to their store. The man-in-the-middle tool can be used to analyse potentially vulnerable applications manually, to check if they can be exploited, and if so, what kinds of data can be intercepted by an attacker.

The Android API does not lend itself to implementing SSL certificate validation securely. For example, it provides an insecure `HostnameVerifier` implementation, and an `SSLSocketFactory` class which bypasses all certificate validation checks. An application can also change the default `HostnameVerifier` and `SSLSocketFactory` used for all new HTTPS connections, which can cause libraries the application uses to become vulnerable, despite them normally being secure. The situation could be improved by removing the insecure classes provided by the API and removing the ability to change the defaults globally in an application.

I only found three applications which used certificate pinning. Certificate pinning is an alternative to the certificate authority model. Instead, the application only trusts a single certificate. If any of the 150 certificate authorities trusted by Android were compromised, the compromised certificate authority would not be able to issue certificates that would be trusted by an application using certificate

pinning. However, the certificates would be trusted by the remaining applications which do not use pinning, allowing the attacker to carry out a man-in-the-middle attack against any of them.

Finally, the Android documentation should perhaps highlight the consequences of disabling certificate validation more explicitly to try to educate more developers about the risks of doing so.

Further extensions to the tools I have developed could include:

- Checking if applications implement certificate expiration correctly.
- Checking if applications implement the certificate path validation algorithm correctly. For example, checking that they do not trust certificates signed by a certificate which does not have the 'CA' bit set.
- Checking if applications obey the 'keyUsage' and 'extendedKeyUsage' certificate fields.
- Checking if any applications implement certificate revocation checking – for example, with the Online Certificate Status Protocol (OCSP) or Certificate Revocation Lists (CRLs). Certificates whose private key has been compromised may be revoked by a certificate authority so that they can no longer be used.

Bibliography

[1] (2014). Android, the world's most popular mobile platform, [Online]. Available: https://developer.android.com/about/index.html.

[2] (2014). Number of available Android applications, [Online]. Available: http://www.appbrain.com/stats/number-of-android-apps.

[3] (2014). Accessing Contacts Data, [Online]. Available: https://developer.android.com/training/contacts-provider/index.html.

[4] (2014). Making Your App Location-Aware, [Online]. Available: https://developer.android.com/training/location/index.html.

[5] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software", in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ACM, 2012, pp. 38–49.

[6] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben and M. Smith, "Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security", in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ACM, 2012, pp. 50–61.

[7] M. Egele, D. Brumley, Y. Fratantonio and C. Kruegel, "An Empirical Study of Cryptographic Misuse in Android Applications", in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2013, pp. 73–84.

[8] (2013). hostapd: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator, [Online]. Available: http://hostap.epitest.fi/hostapd/.

[9] (2014). Android Open Source Project: CA Certificates, [Online]. Available: https://android.googlesource.com/platform/libcore/+/master/luni/src/main/files/cacerts/.

[10] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework", in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, IBM Press, 1999, pp. 13–.

[11] P. Lam, E. Bodden, O. Lhoták and L. Hendren, "The Soot framework for Java program analysis: a retrospective", in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[12] A. Bartel, J. Klein, M. Monperrus and Y. Le Traon, "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot", in *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*, Beijing, China, 2012, ISBN: 978-1-4503-1490-9.

[13] (2014). Opcodes, [Online]. Available: `https://developer.android.com/reference/dalvik/bytecode/Opcodes.html`.

[14] (2008). Paddle: BDD-based context-sensitive interprocedural analysis of Java, [Online]. Available: `http://www.sable.mcgill.ca/paddle/`.

[15] (2014). smali - An assembler/disassembler for Android's dex format, [Online]. Available: `https://code.google.com/p/smali/`.

[16] (2012). ASMDEX, [Online]. Available: `http://asm.ow2.org/asmdex-index.html`.

[17] (2014). netfilter/iptables project homepage, [Online]. Available: `http://www.netfilter.org/`.

[18] (2008). Transparent proxy support, [Online]. Available: `https://www.kernel.org/doc/Documentation/networking/tproxy.txt`.

[19] (2000). dsniff, [Online]. Available: `http://www.monkey.org/~dugsong/dsniff/`.

[20] (2014). Java Native Access, [Online]. Available: `https://github.com/twall/jna`.

[21] (2013). Legion of the Bouncy Castle Java Cryptography APIs, [Online]. Available: `https://www.bouncycastle.org/java.html`.

[22] E. Rescorla, *HTTP Over TLS*, RFC 2818 (Informational), Updated by RFC 5785, Internet Engineering Task Force, May 2000. [Online]. Available: `http://www.ietf.org/rfc/rfc2818.txt`.

[23] *Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates Version 1.1.6*, Jul. 2013. [Online]. Available: `https://cabforum.org/wp-content/uploads/Baseline_Requirements_V1_1_6.pdf`.

[24] J. Zhu and S. Calman, "Symbolic pointer analysis revisited", in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04, Washington DC, USA: ACM, 2004, pp. 145–157, ISBN: 1-58113-807-5. DOI: `10.1145/996841.996860`. [Online]. Available: `http://doi.acm.org/10.1145/996841.996860`.

[25] (2012). Soot: a Java Optimization Framework, [Online]. Available: `http://www.sable.mcgill.ca/soot/`.

[26] (2014). Security Tips | Android Developers, [Online]. Available: `https://developer.android.com/training/articles/security-tips.html`.

Appendix A

Data-flow equations for `X509TrustManager`

The following equations are the data-flow equations used for each Jimple instruction of interest during the `X509TrustManager` data-flow analysis. For the remaining 'uninteresting' Jimple instructions not listed, safe(n) is defined to be the set of all registers overwritten by the instruction and vulnerable(n, v) is defined to be the empty set.

$ra = $rb (register assignment)

$$\text{safe}(n) = \{\$\text{ra}\} \tag{A.1}$$

$$\text{vulnerable}(n, v) = \begin{cases} \{\$\text{ra}\} & \text{if } \$\text{rb} \in v \\ \{\} & \text{otherwise} \end{cases} \tag{A.2}$$

$ra = new X (object instantiation)

$$\text{safe}(n) = \{\$\text{ra}\} \tag{A.3}$$

$$\text{vulnerable}(n, v) = \begin{cases} \{\$\text{ra}\} & \text{if X is vulnerable} \\ \{\} & \text{otherwise} \end{cases} \tag{A.4}$$

$ra[$rb] = $rc (array assignment)

$$\text{safe}(n) = \{\} \tag{A.5}$$

$$\text{vulnerable}(n, v) = \begin{cases} \{\$\text{ra}\} & \text{if } \$\text{rc} \in v \\ \{\} & \text{otherwise} \end{cases} \tag{A.6}$$

$ra = virtualinvoke $rb.getSocketFactory() (`SSLContext.getSocketFactory()` **call)**

$$\text{safe}(n) = \{\$\text{ra}\} \tag{A.7}$$

$$\text{vulnerable}(n, v) = \begin{cases} \{\$\text{ra}\} & \text{if } \$\text{rb} \in v \\ \{\} & \text{otherwise} \end{cases} \tag{A.8}$$

virtualinvoke $ra.init($rb) (`SSLContext.init()` **call)**

$$\text{safe}(n) = \{\} \tag{A.9}$$

$$\text{vulnerable}(n, v) = \begin{cases} \{\$\text{ra}\} & \text{if } \$\text{rb} \in v \\ \{\} & \text{otherwise} \end{cases} \tag{A.10}$$

Appendix B

Soot data-flow analysis code for `X509TrustManager`

The following listing shows an example of how one of the data-flow equations from Appendix A was converted to Java code with Soot:

```
1  @Override
2  protected void flowThrough(final FlowSet in, Unit node, final FlowSet out)
       {
3    node.apply(new AbstractStmtSwitch() {
4      @Override
5      public void caseInvokeStmt(InvokeStmt stmt) {
6        /* change the out-vulnerable set to equal the in-vulnerable set */
7        in.copy(out);
8
9        /* stop here if it invokes a constructor or static method,
10        * this case only cares about instance method invocation */
11       InvokeExpr expr = stmt.getInvokeExpr();
12       if (!(expr instanceof InstanceInvokeExpr))
13         return;
14
15       InstanceInvokeExpr instanceExpr = (InstanceInvokeExpr) expr;
16       SootMethod targetMethod = stmt.getInvokeExpr().getMethod();
17
18       /* check if the target method signature matches SSLContext.init() */
19       if (Signatures.methodSignatureMatches(targetMethod,
20           Types.SSL_CONTEXT, /* class containing the init() method */
21           VoidType.v(), /* return type */
```

```
22          "init", /* method name */
23          Types.KEY_MANAGER_ARRAY, /* first argumnet */
24          Types.TRUST_MANAGER_ARRAY, /* second argument */
25          Types.SECURE_RANDOM)) { /* third argument */
26        Value context = instanceExpr.getBase(); /* the SSLContext */
27        Value trustManagerArray = instanceExpr.getArg(1); /* the
              TrustManager array */
28
29        /* if SSLContext.init() is called with a vulnerable TrustManager,
30         * then mark the SSLContext as vulnerable */
31        if (out.contains(trustManagerArray)) {
32          out.add(context);
33        }
34      }
35    }
36
37    /* additional cases (such as register assignment) here... */
38  });
39 }
```

Listing B.1: Data-flow analysis code for the `SSLContext.init()` call.

Appendix C

Example static analysis output

The static analysis tool outputs vulnerabilities in tab-separated value format. The first column is the vulnerable class (or method), the second column contains the type of vulnerability, and the third column contains `VULNERABLE`, `SAFE` or `UNKNOWN`.

```
1 com.zubhium.utils.ZubhiumTrustManager PERMISSIVE_TRUST_MANAGER VULNERABLE
2
3 fm.bloom.framework.EasyX509TrustManager PERMISSIVE_TRUST_MANAGER
    VULNERABLE
4
5 com.zubhium.utils.ZubhiumNetworkUtils$MySSLSocketFactory$1
    PERMISSIVE_TRUST_MANAGER VULNERABLE
```

Listing C.1: Static analysis output from the vulnerable Bloom.fm music streaming app. I reported these vulnerabilities to Bloom.fm, who have since fixed their application.

Appendix D

Project Proposal

Computer Science Project Proposal

Detection of SSL-related security vulnerabilities in Android applications

Graham Edgecombe, Pembroke College

Originator: Graham Edgecombe

22nd October 2013

Project Supervisor: Dr Alastair Beresford

Director of Studies: Chris Hadley

Project Overseers: Dr Stephen Clark & Dr Pietro Lió

Introduction

Transport Layer Security (TLS), more commonly known as Secure Sockets Layer (SSL) – the name of its predecessor, is a cryptographic protocol used to provide a client-server communication channel which prevents eavesdropping and tampering of messages sent through it.

SSL uses public-key cryptography to allow the client to verify the identity of the server (or vice versa). Each server has a certificate which contains information such as the hostname of the server, a public key and the address of the organisation operating the server. A number of organisations, known as certificate authorities (CAs), are trusted by web browsers and SSL libraries to sign certificates. The act of signing a certificate implies that the CA has verified that the owner of the certificate is whom they claim to be.

Upon opening an SSL connection, an SSL library will usually check if the certificate presented by the server meets all of the following conditions:

- The certificate is signed by a CA on the list of trusted CAs.
- The certificate's hostname matches the hostname of the server the client believes it is connected to.
- The certificate is within its validity period.

This procedure is called certificate validation. If a certificate is deemed to be invalid, the user will typically be asked if they wish to continue connecting to the server or the connection will fail outright.

Certificate validation is typically implemented very thoroughly in web browsers, with some even going beyond the procedure described above and also checking that certificates have not been revoked and that the public key is of a sufficient length.

However, research suggests that certificate validation is not implemented as thoroughly outside of web browsers [5] - for example, in mobile phone and tablet applications. If self-signed certificates or certificates with any hostname are considered valid it is possible to execute a man-in-the-middle (MITM) attack and thus eavesdrop or tamper with the communication between the two parties. Mobile phone applications are particularly interesting targets for attackers because of the prevalence of open, unencrypted Wi-Fi hotspots which could be used to launch MITM attacks.

Automated detection of common SSL-related security vulnerabilities could be useful for programmers writing mobile phone applications which perform network communication.

Starting Point

Over the summer I worked at Netcraft on their SSL Server Survey **ssl-survey** so I am already familiar with SSL terminology and I have a basic grasp of how it works internally.

The Security I course from Part IB of the Computer Science Tripos is also relevant, particularly the section about asymmetric cryptography.

Substance and Structure of the Project

The aim of the project is to write a program which performs static analysis of Dalvik bytecode to find common SSL-related vulnerabilities, which might allow a MITM attack to be performed, and to write a second program which attempts to carry out MITM attacks against SSL connections on an unencrypted WiFi network. Both parts would be an implementation of techniques described in *Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security* [6].

The Soot [25] Java bytecode bytecode analysis library is likely to be used to perform static analysis. Soot converts Dalvik bytecode to Jimple – a typed three-address code representation which is more amenable to static analysis. Soot also implements a variety of useful algorithms such as control flow graph generation and decomposition of a method into basic blocks.

The following vulnerabilities could be detected:

- **Use of a `HostnameVerifier` that disables hostname verification.** This allows an attacker to use a signed certificate from a trusted CA with any hostname to carry out a MITM attack.
- **Use of `SSLSocket` directly without a `HostnameVerifier`.** Unlike `HttpsURLConnection`, when using `SSLSocket` directly the programmer must implement hostname verification themselves.
- **Use of a custom `TrustManager` that trusts any certificate.** This allows an attacker to use a self-signed certificate to carry out a MITM attack.

- **Use of plaintext TCP/IP communication.** For example, the use of the `Socket` class directly indicates that plaintext communication may be used by the application. SSL also has some "null" cipher suites which do not perform authentication and/or encryption.

The program which performs MITM attacks would use a technique such as ARP spoofing to intercept and modify the traffic. It would be used to confirm if an application is vulnerable to possible certificate validation problems detected by the static analysis. It could also be used to check properties of the handshake as it takes place – such as the use of old versions of SSL or undesirable cipher suites (e.g. cipher suites which do not provide perfect forward secrecy, the deliberately weakened export cipher suites, "null" cipher suites, etc.)

I will write a number of small Android applications to use as test cases for both the static analysis and the MITM program – some of which will contain the vulnerabilities listed above (to confirm they can be detected), and others which will not (to confirm that there are no bugs causing false positives). The complexity of the test cases could be gradually increased during the project to try to ensure the static analysis is capable of detecting variations of the vulnerabilities that are present in real applications, rather than just simple cases.

Success Criterion

To be considered a success, the project should be able to:

1. Detect hostname verification and trust manager vulnerabilities, and the use of plain text TCP/IP communication, to a reasonable extent using static analysis (obfuscation, reflection etc. would prevent this being possible in all cases.)

2. Launch a MITM attack against applications with certificate validation vulnerabilities on an unencrypted WiFi network.

3. Use an intercepted handshake to detect the use of old SSL versions/undesirable cipher suites.

Both static analysis and an attempted MITM attack should be performed on a selection of real applications (e.g. 100 hand picked applications or a larger number automatically downloaded from the Google Play application store) to discover

how widespread SSL-related vulnerabilities are, if popular applications are safer than less popular applications and to compare the safety of different categories of application (e.g. games, business, productivity, shopping, etc.)

Possible Extensions

Possible extensions include:

- Investigating the use of certificate pinning, which allows an application to trust a limited number of CAs or a single certificate, which limits the damage of CA compromise.
- Perform static analysis of other classes of vulnerability. The Android documentation lists common vulnerabilities [26] which could be used as a starting point.

Plan of Work

1. **Michaelmas weeks 2–4 (19/10/2013–06/11/2013):** Research how to use the Soot library and its Jimple representation. Write some test programs to familiarise myself with using it.
2. **Michaelmas weeks 5–6 (07/11/2013–20/11/2013):** Start writing the static analysis program – loading `.apk` files, converting Dalvik to Jimple with Soot and add detection of hostname verification vulnerabilities. Write some simple Android applications which do not perform hostname verification to use as test cases.
3. **Michaelmas weeks 7–8 (21/11/2013–08/12/2013):** Add detection of trust manager vulnerabilities and write applications to use as test cases.
4. **Michaelmas vacation (09/12/2013–23/12/2013):** Add detection of possible plaintext communication along with test cases. Add code to generate a report with a list of possible vulnerabilities and their location (the class and method).
5. **Michaelmas vacation (27/12/2013–12/01/2014):** Test the static analysis program with some real applications. Make amendments to improve

detection, if possible within the time frame. Research techniques to perform a MITM on unencrypted WiFi networks (e.g. ARP spoofing).

6. **Lent weeks 0–2 (13/01/2014–29/01/2014):** Write progress report. Start writing the MITM program, aiming for it to intercept plaintext communication.

7. **Lent weeks 3–5 (30/01/2014–12/02/2014):** Add support for SSL interception to the MITM program, which should generate certificates on the fly to exploit certificate validation vulnerabilities. Test using the same test cases used for the static analysis program.

8. **Lent weeks 6–8 (13/02/2014–16/03/2014):** Add detection of old versions of SSL/undesirable cipher suites to the MITM program. Start evaluation – testing on some real Android applications.

9. **Easter vacation (17/03/2014–06/04/2013):** Complete evaluation and fix any remaining bugs, start writing the first draft of the dissertation.

10. **Easter vacation (07/04/2014–20/04/2014):** Complete first draft of the dissertation.

11. **Easter weeks 0–2 (21/04/2014–7/05/2014):** Hand copies of full draft to supervisor and director of studies for feedback on 22/04/2014. Amend dissertation as required to produce the final version.

12. **Easter week 3 (8/05/2014–16/05/2014):** Print, bind and hand in dissertation.

Resources Declaration

I have already obtained copies of the Android SDK and the Soot library.

I have a Google Nexus 4 Android phone which will be useful for testing. In case of loss or failure, the emulator in the Android SDK should be sufficient for many tasks until I buy a replacement phone. I also have a wireless router which can be used to test the MITM program.

I will use the Git version control system and perform development on my own desktop computer, which runs the stable release of Debian. It has a quad core CPU and 8 GB of RAM. I back up all files to an external disk once a week with `rdiff-backup`. In case of failure, I have a laptop which could be used instead.

I accept full responsibility for this machine and I have made contigency plans to protect myself against hardware and/or software failure.