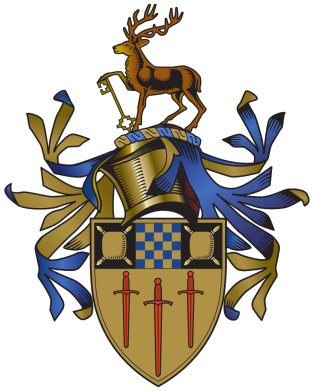


UNIVERSITY OF SURREY

Faculty of Engineering & Physical Sciences

Department of Computing

COM3001: Professional Project

A system to control the chain of trust on iOS.

Student: Ryan Edward Burke

URN: 6162710

Supervisor: Prof. Paul Krause

Paper submitted to the University of Surrey in partial fulfilment for the degree of BSc Computer Science

Declaration

This dissertation and the work to which it refers are the results of my own efforts. Any ideas, data, images or text resulting from the work of others (whether published or unpublished) are fully identified as such within the work and attributed to their originator in the text, bibliography or in footnotes. This dissertation has not been submitted in whole or in part for any other academic degree or professional qualification. I agree that the University has the right to submit my work to the plagiarism detection service TurnitinUK for originality checks. Whether or not drafts have been so-assessed, the University reserves the right to require an electronic version of the final document (as submitted) for assessment as above.

Ryan Burke

Abstract

This paper investigates the implementation of the certificate trust store on the Apple iOS mobile operating system, with the aim of improving security and privacy for users. At the time of writing, iOS offers no way in which users can chose the certificates they want trust for secure online communication and as such, introduce potential vulnerabilities to the confidentiality of their online data.

The basis of trust in certificate authorities is challenged and in a world where people are increasingly concerned about online privacy, the question is asked as to how users can control whom their devices should trust. Scenarios are identified whereby rogue agencies could perform man-in-the-middle attacks by forcing certificate authorities to issue fake digital certificates and how a client-side application could help protect against such attack.

Research is conducted into how the iOS system handles trust decisions and the role of the Security framework plays within this. From the research into the iOS security framework, a certificate management system has been created for the platform, consisting of a front-end management application and a system modification patch. Through the use of such system, users are able to assign trust properties to the certificates used for secure communication in order to block or allow network connections.

Acknowledgements

My family, for the encouragement to achieve my highest potential.

Lucie, for being so patient while I've studied these past years.

Prof. Paul Krause, for his support and feedback throughout the project.

Table of Contents

Declaration 2

Abstract 3

Acknowledgements 4

List of Acronyms 8

List of Figures 9

List of Tables 10

List of Code Examples 11

Chapter 1 - Introduction 12
- **1.1 Motivation** 12
- **1.2 Issue** 13
- **1.3 Project Aims** 14
- **1.4 Project Structure** 15
- **1.5 Report Structure** 15
- **1.6 Summary** 16

Chapter 2 – Technology Overview 17
- **2.1 TLS/SSL** 17
  - 2.1.1 X.509 Certificates 18
  - 2.1.2 Public Key Infrastructure 19
  - 2.1.3 Online Certificate Status Protocol 20
- **2.2 SSH** 21
- **2.3 OpenSSL** 21
- **2.4 Jailbreak** 22
  - 2.4.1 Cydia Substrate 22
  - 2.4.2 SpringBoard 23
- **2.5 Compelled Certificate Creation Attack** 23
- **2.6 Related Work** 25
  - 2.6.1 iOS SSL Kill Switch 25
  - 2.6.2 iSSLFix 26
  - 2.6.3 CACertMan 26
- **2.7 Summary** 26

Chapter 3 – Security Research 28
- **3.1 Trust Store** 28
- **3.2 Modifying Certificate Validity** 30
- **3.3 Analysing the Security Framework** 34
- **3.4 Summary** 36

Chapter 4 – System Development Strategy 38
- **4.1 System Description** 38
  - 4.1.1 CertManager 38
  - 4.1.2 CertHook 38
- **4.2 System Architecture** 39
- **4.3 Inter Process Communication** 40
- **4.4 User Interface Design** 41
- **4.5 Software Development Methodology** 43
  - 4.5.1 Milestones 45
  - 4.5.2 Git 45

4.6 Summary **46**

Chapter 5 – System Implementation: CertManager **47**
5.1 Accessing Certificate Store **48**
5.2 Additional Functionality **50**
5.2.1 Manual Blocking 50
5.2.2 Logging 51
5.2.3 CertBrowser 51
5.3 Writing to the Disk **53**
5.4 Wrappers and Utilities **53**
5.4.1 FileHandler 54
5.4.2 CertDataStore 54
5.4.3 X509Wrapper 54
5.5 External Resources **55**
5.5.1 Apple Security Framework 55
5.5.2 Categories 55
5.5.3 DDFileReader 56
5.5.4 Font Awesome 56
5.6 Summary **57**

Chapter 6 – System Implementation: CertHook **58**
6.1 Creating the Project **58**
6.2 Hooking Methods **59**
6.3 Loading User Preferences **59**
6.4 Blocking Connections **61**
6.5 Notifying the User **63**
6.6 Sandboxing **65**
6.7 Summary **66**

Chapter 7 – Development Environment **67**
7.1 Requirements **67**
7.2 Theos **68**
7.3 Storyboards and XIB **70**
7.4 iOS Framework Headers **70**
7.5 Remote Debugging **70**
7.6 Development Tools **71**
7.6.1 Xcode 72
7.6.2 Sublime Text 72
7.6.3 Hex Fiend 72
7.6.4 SourceTree 72
7.6.5 Terminal 72
7.7 Summary **73**

Chapter 8 – Testing and Effectiveness **74**
8.1 Manual Inspection **74**
8.2 Network Traffic Analysis **75**
8.3 Summary **77**

Chapter 9 – Conclusion **78**
9.1 Success Criteria **78**
9.2 Challenges **79**
9.2.1 Lack of documentation 79
9.2.2 Reliance on the Security Framework 80
9.3 Future Work **80**
9.4 Closing **81**

Bibliography.....**82**
Appendices **86**

List of Acronyms

API	Application Program Interface
APSL	Apple Public Source Licence
ARC	Automatic Reference Counting
BSD	Berkley Software Distribution
CA	Certificate Authority
CNNIC	China Internet Network Information Centre
CRL	Certificate Revocation List
DER	Distinguished Encoding Rules
DYLIB	Dynamic Library
GCHQ	Government Communications Headquarters
GNU	GNU's Not Unix!
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
IP	Internet Protocol
IPC	Inter Process Communication
IRC	Internet Relay Chat
MITM	Man-in-the-Middle attack
MVC	Model View Controller
NSA	National Security Agency
OCSP	Online Certificate Status Protocol
OS	Operating System
OTA	Over the Air
PKI	Public Key Infrastructure
RA	Registration Authority
SDK	Software Development Kit
SHA1	Secure Hash Algorithm 1
SKI	Subject Key Identifier
SSH	Secure Shell
SSL	Secure Socket Layer
SVN	Apache Subversion
TCP	Transmission Control Protocol
TIFF	Tagged Image File Format
TLS	Transport Layer Security
UDP	User Datagram Protocol
URL	Uniform Resource Identifier
USB	Universal Serial Bus

List of Figures

Figure 1 - The Keychain Access application for OS X. 13
Figure 2 - The certmgr.msc application for Windows. 14
Figure 3 - The Trusted Credentials application for Android. 14
Figure 4 - TSL Handshake Overview 18
Figure 5 - Partial contents of a X.509 certificate. 18
Figure 6 - Inheritance of trust. 19
Figure 7 - A normal certificate creation process. 24
Figure 8 - The compelled certificate creation attack. 25
Figure 9 - The contents of Security.framework on iOS 8.1 29
Figure 10 - The output of a single entry within certsIndex.data 29
Figure 11 - The raw certificate data stored inside the table data file. 30
Figure 12 - The certsTable.data file with the CNNIC public key zeroed out. 30
Figure 13 - iOS warning to the user. 31
Figure 14 - Google Chrome's warning to the user. 31
Figure 15 - The SHA1 values inside Blocked.plist 32
Figure 16 - The logic flow for checking the validity of a certificate. 33
Figure 17 - The different layers of networking in iOS 34
Figure 18 - The SecureTransport life-cycle 35
Figure 19 - Console output from traversing the chain of trust for Facebook.com 36
Figure 20 - System architecture overview diagram. 39
Figure 21 - The initial (left) and revised (right) wireframes for CertManager 41
Figure 22 - The wireframes for the manual page. 42
Figure 23 - The wireframe for the Log view. 42
Figure 24 - The CertBrowser wireframe, with additional blocking functionality. 43
Figure 25 - The Spiral development methodology as described by Boehm. [65] 44
Figure 26 - The custom Securityd framework. 48
Figure 27 - Console output of the CopyCertsFromIndices function. 49
Figure 28 - An early version of CertManager that listed the root certificates. 49
Figure 29 - The final implementation for the custom table cell. 49
Figure 30 - The view hierarchy for CertManager. 50
Figure 31 - The CertBrowser URL entry bar with green padlock icon. 52
Figure 32 - The different visual representations for viewing certificate data. 52
Figure 33 - Code to append a log to the existing file. 53
Figure 34 - The methods by the X509Wrapper class. 55
Figure 35 - A part of the license allowing modifications to code. 55
Figure 36 - An example of some of the icons available from the library. 57
Figure 37 - The CertHook SSLHandshake algorithm flow chart. 62
Figure 38 - A local notification shown when a certificate is blocked by CertHook 65
Figure 39 - The project dependencies defined in the control file. 67
Figure 40 - The variables for Theos set in the .bash_profile file. 68
Figure 41 - The contents of the CertHook MakeFile. 68
Figure 42 - The commands used to generate a root certificate. 75
Figure 43 - The contents of a sensitive message sent over HTTPS from the device. 76
Figure 44 - The certificate chain of the MiTM attack, shown by CertManager. 76
Figure 45 - Charles showing no data sent from the client. 77
Figure 46 - Potential detection mechanism for future work. 80

List of Tables

Table 1 - Market share of Certificate Authorities [24]. 20
Table 2 - Results from testing different certificates in the plist files. 33
Table 3 - The parameters of the SSLCreateContext method. 36
Table 4 - The parameters of the SSLSetConnection method. 36
Table 5 - The results of blocking different certificates in Safari. 74
Table 6 - The results of application testing. 74
Table 7 - Evaluation of project aims. 78

List of Code Examples

Code 1 - Method swizzling pseudocode. 22
Code 2 - The availability macro defined by Apple. 28
Code 3 - Detecting a change to the SHA1 field. 50
Code 4 - The regex for a SHA1 string representation. 51
Code 5 - The properties of the LogInformation class. 51
Code 6 - Obtaining trust information from an authentication challenge object. 51
Code 7 - Extracting certificate data from the SecTrustRef object. 52
Code 8 - An example of an Objective-C category. 56
Code 9 - First basic system modification code 58
Code 10 - Subproject code added to the CertManager MakeFile. 58
Code 11 - Original hooking method proof of concept. 59
Code 12 - The original way of reading user preferences. 60
Code 13 - Listening for a message from CertManager. 60
Code 14 - The code for stopping additional requests. 61
Code 15 - Method definition to get a reference to the messaging centre. 63
Code 16 - Method definition to register a listener to the messaging centre. 63
Code 17 - Method definition to send a message using the messaging centre. 64
Code 18 - The code to show a local notification from within a UIApplication. 64
Code 19 - The method definition used to show a local notification. 64
Code 20 - The code used to show a local notification. 65
Code 21 - Applying RocketBootstrap to the messaging centre. 66
Code 22 - The command to built the system. 68
Code 23 - The command to connect socat to the syslog. 71
Code 24 - Attaching debugserver to a process. 71
Code 25 - Connecting to the debugserver from a remote machine. 71

Chapter 1

Introduction

Over the last decade, the technology industry has seen a large shift from desktop computing towards mobile computing, with mobile now taking 60% of total time viewing digital media [1]. With this shift, new operating systems have emerged and become more abstract from their technical underlays in order to appeal to the public. Consumers are no longer expected to spend their time configuring the system; it should just work. This shift has lead to an environment where users have less control over the devices they use daily and options once controlled on the desktop ecosystem are now being hidden behind the scenes and controlled by the manufacturer.

The two largest forces behind the mobile movement have been Apple and Google, with the growth of the iOS [2] and Android [3] operating systems. While the two systems have a lot in common in terms of user interfaces and applications, the underlying systems are managed differently. Android is commonly known as an open platform, with the source code of the system being available for anyone to view and modify. There are many different branches of Android, with carriers and manufacturers able to create their own flavours to put on their devices [4]. iOS on the other hand is the opposite, Apple like to take advantage of vertical integration [5] in order to maintain control over the hardware, software and applications that can run on their devices. Their desktop operating system, OS X, only runs officially on Apple designed hardware and iOS is no different. The code is closed-source and only Apple can provide patches and updates to the system which appeals to the vast majority of consumers today who enjoy the simplicity of their products.

Helping further the increase in mobile usage is the availability of super-fast Internet [6], be it from the rollout of fibre optic cables or from the advancements in mobile Internet technology such as 4G [7]. This has lead to mobile devices being used for everyday Internet activities and with that comes the requirement that they able to process data securely. Underpinning this security is the TLS/SSL technology that has been reliably used on desktops since 1995 [8], using cryptography to securely transfer data between clients and servers across the Internet.

This project researches the implementation of TLS/SSL on the iOS platform and what sort of control, if any, a user has over the certificates it inherently trusts. Research techniques into the system include reading the official Apple system documentation, file system analysis, the use of private frameworks and overriding system functions to analyse data as it moves. Based on the results of this research, a mobile application has been created in order to allow users to assign trust decisions for the connections that are made from iOS devices.

1.1 Motivation

The motivation behind this project is due to the rising concern of digital security and privacy. Previously, there were rumours that an advanced persistent threat had the capabilities to intercept communications, but since the Global surveillance disclosures in 2013 [9], these have been confirmed as reality through projects run by government agencies such as the NSA and GCHQ [10]. As mobile computing progresses so does the emergence of new threats, in

this report the compelled certification attack is discussed and used as a threat example that could potentially be protected by the CertManager system.

As mentioned previously, there is a move towards more consumer friendly computing devices with a focus on user experience over configurability. Coupled with the rising trend of the Internet of Things [11] and more personal technology such as wearable computers, humanity is heading towards a future where everything will be generating and consuming data on a massive scale. The way in which this data is stored is also changing with increased usage of cloud based storage [12], we are seeing a move towards users not knowing where their personal data is and who has access to it.

While secure protocols have been implemented to help protect this data in transit and storage, global surveillance is a real threat. Governments are able to bypass protection by working in partnership with private companies that users already trust [13]. They can introduce or suggest changes to algorithms that at subtle weaknesses in them [14], so that they are easier to crack for those who know the vulnerability. Companies are forced to comply with requests with legal and economical sanctions against them if they do not comply, and bind them to secrecy with national security orders [15]. The author believes that from an information security context, a private company that resides in a nation who actively partakes in mass surveillance cannot be totally trusted as the basis of modern web security.

1.2 Issue

The focus on this project is on the implementation of TSL/SSL on iOS. On most desktop operating systems users are able to access their security settings and modify the certificates that their computer trusts but this is not the case for iOS. The main issue with this is the lack of being able to stop the system from making unwanted connections. Users may know for certain that there will be certain certificate authorities that they will never want to connect to, or be alerted to when they are used. Most operating systems provide an interface whereby the user is able to assign trust decisions to the root certificates that the system uses.

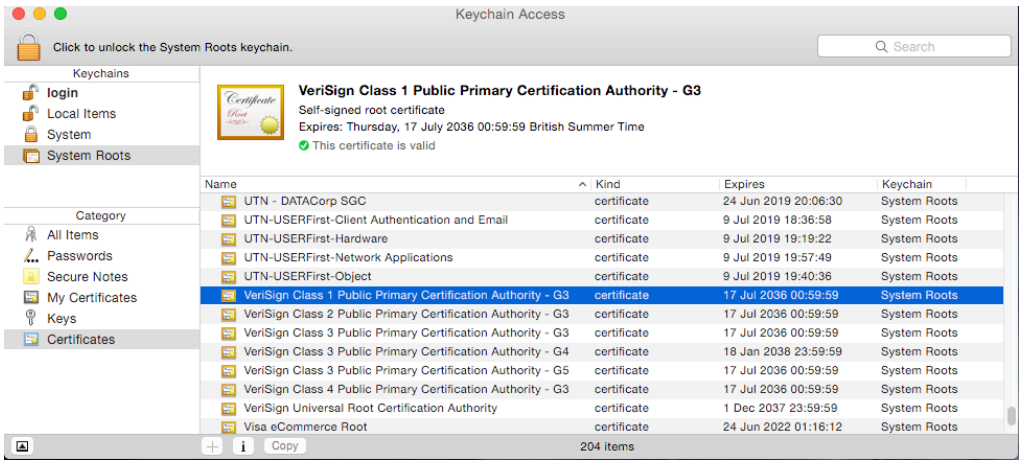


Figure 1 - The Keychain Access application for OS X.

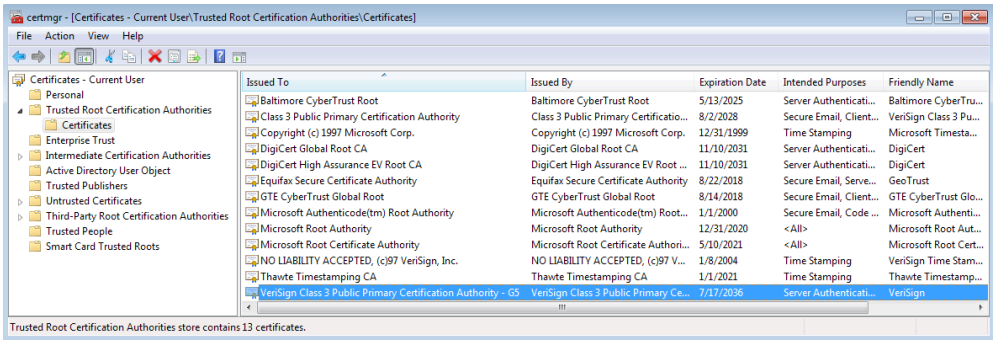


Figure 2 - The certmgr.msc application for Windows.

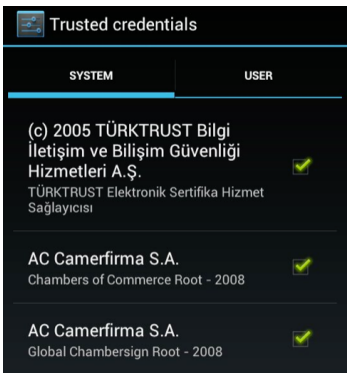


Figure 3 - The Trusted Credentials application for Android.

On iOS, there is no such management system in place; users must trust all root certificates pre-installed in software updates by Apple. This is an example of the level of control that Apple has over its operating system, as companies have to specifically apply to Apple in order to be included in their trust store [16]. One problem that arises is that Apple tends to discontinue software updates for older devices after around 4 years, this leaves many people without the latest root trust stores and therefore remain vulnerable. Meanwhile those running the latest software may have to wait weeks or even months before Apple removes blacklisted certificate authorities from their trust store. An example of this is the DigiNotar hack that was exposed on August 29th 2011 [17] but iOS was not patched until October 12th [18], leaving users vulnerable for over a month. Additionally some users may not wish to trust certain companies or government agencies that are in control of root certificates and are unable to remove that trust from their device. The problem in this instance is not one of security but more about giving users the personal freedom to choose whom they trust.

1.3 Project Aims

The aim of this project is to investigate how TSL/SSL is implemented on iOS, and by doing so, provide a way for users to control the certificates their devices can trust. In order to do this, the project has been split into sections in order to better structure the research and development process.

- Research TSL/SSL implementation on iOS.
- Identify how X.509 certificates can be extracted from an iOS network connection.
- Create a mobile application that can:
    - Show certificates pre-installed on the device.
    - Allow users to change the trust option for certificates.

- Provide a way in which a user can block all incoming and outgoing connections that do not match their trust preferences.
- Evaluate the effectiveness of the system in both a test environment and a real world scenario.

1.4 Project Structure

The project is split into two main sections. Firstly, conducting research into the iOS system itself through the use of a jailbroken system to gain access to private frameworks and learn how Apple implement SSL certificate checks at the system level. The second half of the project is using this obtained knowledge in order to create a mobile application and system level tweak that will allow users control over the certificates that are trusted on their device. Finally, this system is tested by performing a replicated attack on the device to evaluate the effectiveness of the system.

1.5 Report Structure

Chapter 1: This section introduces the project with reasoning as to why it was chosen. Aims are defined for the scope of the project and its structure is described.

Chapter 2: This is the literature review for the various technologies that will be discussed throughout the project.

Chapter 3: This section documents the steps taken to conduct research into the iOS system. It looks at different ways in which certificates can be extracted from the device, and how the system processes a TSL connection. It evaluates the best way in which to override a trust decision made by the system at current.

Chapter 4: This section defines how the system will work. It goes into detail into the architecture of the system and plans for the user interface of the system front-end.

Chapter 5: This section describes the implementation of the CertManager application. It is split into sub-sections based on different components of the overall system. In this section, the methods used to achieve key milestones in the project are described, with analysis on why those methods were chosen.

Chapter 6: This section describes the implementation of the CertHook system. This is the system that is responsible for implementing the custom certificate trust on the iOS system level. It is split into sections based on the key milestones that were completed, providing technical explanation and evaluation of methods.

Chapter 7: This section describes the development environment that was used for the project and the different issues that were overcome. This includes information about the build system, debugging and development tools that were utilised.

Chapter 8: This section contains results for testing the system after it was completed. It looks at various ways in which the system can be tested, and edge cases that may be found.

Chapter 9: The final section concludes the report and included an analysis of the system and its effectiveness. It looks at any issues or limitations of the system and describes how they could be addressed in future work. Finally, it looks at the knowledge and experience that was gained from the project.

1.6 Summary

This chapter has described an overview of the project and the aims that it sets out to achieve. It has introduced the background context that the system will be used within and highlights some of the issues that are currently being faced. The outline of the project has been defined by splitting it into two sections based on research to be conducted and time spent on software development of the system itself. By setting project aims, a clear success criteria has been defined that can be used for evaluation purposes at the end of the project.

Chapter 2

Technology Overview

This section aims to introduce the key technologies that are used within this project in order to provide context and background information that can be used later on in research. The different technologies first had to be researched so that an understanding could be obtained to act as the basis to further expand through research into the iOS system. The majority of this section explores the TLS and SSL protocols and the infrastructure supporting them. It also introduces the iOS jailbreak process and the libraries that were used in this project relating to that.

2.1 TLS/SSL

Underpinning the secure transport of data across the Internet is a series of cryptographic protocols known as TLS. The current version of TLS in use is 1.2 and it is an evolution of the earlier SSL protocols. The primary goal of the TLS protocol is to provide confidentiality and integrity to data while it travels over an unsecure network through the use of encryption [19]. The protocol can be split into two layers, the message layer and the handshake layer.

The handshake process begins by the client sending a request to the server to say that it wants to start a secure connection. Within this initiator message the client also includes a random number R_{client} to be used for later cryptographic functions. In response to this message, the server replies with its public certificate and another random number R_{server} to also be used later. The client creates another random number S known as the *pre-master secret* and encrypts it with the public key of the server, which is extracted from the server's certificate [20]. The encrypted pre-master secret is then sent to the server and a master key *K* is calculated from all three values where:

$$K = f(S, R_{client}, R_{server})$$

Equation 1 - Master key calculation function.

The master key is then hashed with the two R-values in order to generate 6 keys that are used for encryption, integrity and initialisation vectors for both ways of traffic. The server then uses its newly created encryption write key to create a hash of all the previous handshake messages and sends this back to the client. The client does the same and uses the encryption read key to make sure the two hashes are the same. If this is true then the connection can continue as the client has authenticated that it is speaking to the server. For the remainder of this session these keys are used in order to encrypt subsequent messages between the client and server.

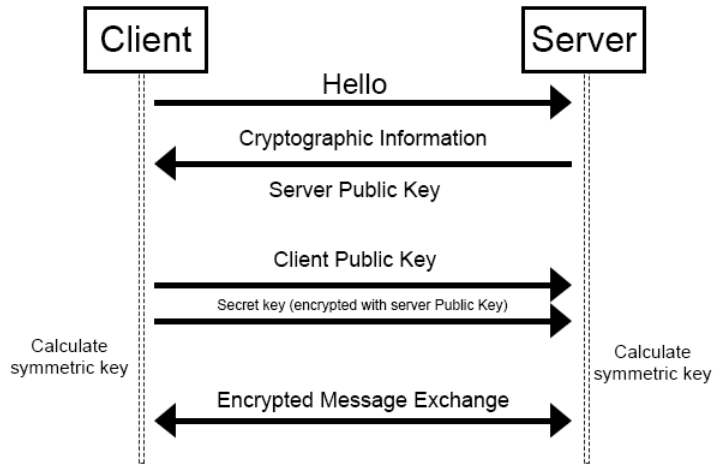


Figure 4 - TSL Handshake Overview

2.1.1 X.509 Certificates

As mentioned in 2.1 TLS/SSL, the initial information from the server to the client is sent in the form of a certificate. In the SSL/TLS protocol these certificates are in the X.509 format, a standard created by the IUT-T. The main purposes of the certificates are to prove the identity of an entity and for use in cryptographic functions. The certificate contains information relating to an entity and this is stored in a field referred to as a Distinguished Name [21]. This entity is defined within the subject field of the certificate that holds pieces of identifying information about the ownership of the certificate. There are various sub-fields that contain additional information, known as relative distinguished names or attributes, which can also be used to identify the owner.

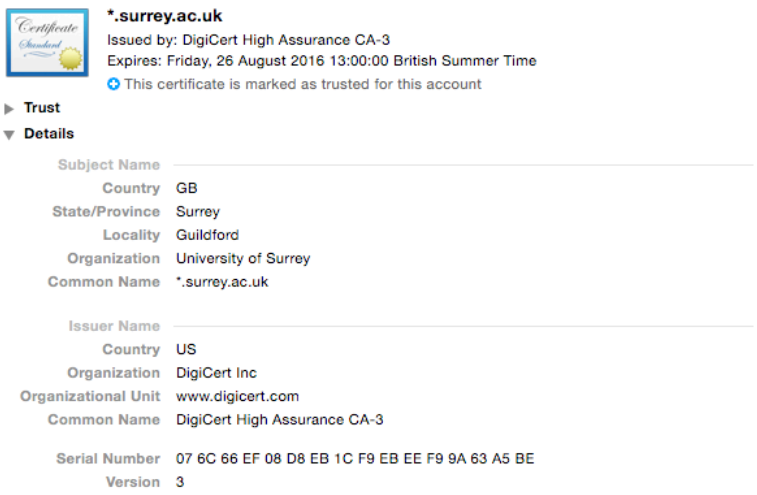


Figure 5 - Partial contents of a X.509 certificate.

As well as being able to confirm the ownership of a domain, the X.509 certificate provides a way in which an entity can store and share the public key they use for cryptography. A client is able to extract the public key from the certificate and use it to encrypt data that can only be decrypted using the corresponding private key, which is kept secret by the entity.

In addition to the standard X.509 certificate, a growing trend is for companies to use Extended Validation Certificates [22]. These are the same certificates but provide additional information that contains extended identity verification. They give no increase to the cryptographic security of the certificate's public key but are provided in order to tell users that the website owners

have had additional checks performed by the certificate authority to ensure their identity. This was seen as needed by the online industry due to the rise of 'domain only' certificates being issued by CAs for low prices. The impact of this low pricing strategy meant that the checks being made into the owners of websites were not very rigorous, meaning that there was no way to tell if a legal company was behind a website.

2.1.2 Public Key Infrastructure

The public key from the X.509 certificate being used to send sensitive information to a server to be decrypted by a private key is an example of asymmetric encryption. One of the issues that asymmetric encryption solves is the Key Distribution problem found in standard symmetric encryption. This is where each node n on a network is required to create a secret key for every node on the network that it want to communicate with. This would result in each node needing n^2 keys in order to talk to every node on the network, a figure that is not feasible on large networks such as the Internet [23]. By using asymmetric encryption each node has a public and private key pair that are cryptographically linked so that the private key can decrypt messages encrypted by the public key and the same in reverse. This means that a message can be signed with the public key and only the owner of the private key can decrypt it to read, providing confidentiality. Additionally, the message can be signed by the owner of the keys with his private key, anyone can now decrypt the message using the public key and will be able to confirm the identity of who sent the message as it could have only been signed with the private key.

The issue that remains is how a node can confirm that the certificate it received genuinely represents the entity that it claims to be. As mentioned in 2.1.1 X.509 Certificates, the certificate contains a distinguished name which could be for instance:

$$*.surrey.ac.uk$$

If the client made a request to the domain *surrey.ac.uk* and received back this certificate it could check that the common name matches the domain name and allow the connection. However, anyone can create a certificate and use any value for the name which means that just because a certificate says it represents a domain, it doesn't mean that it actually does. In order to fix this issue, PKI uses trusted authorities and inheritance of trust to verify the identity of a certificate. In the X.509 protocol these authorities are known as Certificate Authorities. If Alice trusts a certificate authority, who trusts an entity Bob, then Alice will trust Bob.

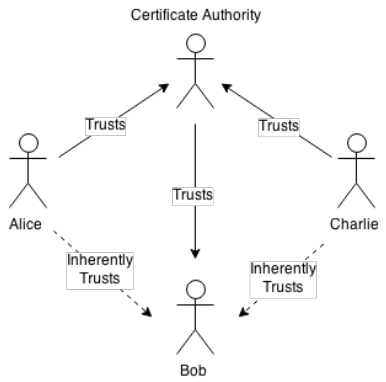


Figure 6 - Inheritance of trust.

A CA in its simplest form is an entity which issues digital certificates. The role of a CA within the PKI used for the world wide web is to validate that the entity that made a certificate request

for a domain name is the true owner. This is usually achieved by sending an email to an administrative account for that domain such as *admin@* or *webmaster@* asking the owner of the account to confirm that a request for a certificate was made. Once this confirmation has been made the CA will generate a new certificate with the public key and all the identifying information included, plus new information that says that the certificate was signed by the CA. In order to verify this, it generates a signature of the certificate using it's private key and stores this signature within the certificate.

When it comes to the Internet, and the world wide web in particular, there are millions of servers providing certificates to clients who do not trust them. On each client there is a list of certificate authorities that it trusts and these are usually provided by the operating system or by web browsers. When an SSL/TSL connection is made, the website's X.509 certificate is sent to the client and the client is able to check the issuer of the certificate to see if it is one of these trusted authorities. The client can then use the public key of the trusted authority to verify that the trusted authority signed the certificate.

The integrity of the entire system depends on all of the CA's within the client store being always trustworthy. There is no check in place for SSL/TSL that a certificate should come from a certain certificate authority, therefore a certificate for Alice signed by a trusted CA Bob would be accepted by the client. In addition to this, a certificate for Alice signed by trusted CA Dave would also be accepted because the client has no way of knowing which one is legitimate as it trusts both CAs not to issue certificates without first checking ownership. This is the foundation of the attack seen later in this report.

Most popular SSL certificate authorities

© W3Techs.com	usage	change since 1 April 2015	market share	change since 1 April 2015
1. Comodo	5.3%	+0.3%	35.6%	+0.9%
2. Symantec Group	4.7%		31.9%	-0.7%
3. Go Daddy Group	2.1%	+0.1%	14.0%	-0.1%
4. GlobalSign	1.5%	+0.1%	10.0%	
5. DigiCert	0.4%		2.7%	

percentages of sites

Table 1 - Market share of Certificate Authorities [24].

Looking at the distribution of market share of certificate authorities, the top 3 companies hold 81% of the global market share for issuing SSL certificates. What is also interesting is that 4 out of 5 of these companies are US based, meaning that they fall under US jurisdiction and law. This means that the vast majority of the web's secured traffic could potentially be intercepted by government authorities such as the NSA who could leverage these companies to issue false certificates.

2.1.3 Online Certificate Status Protocol

As part of running a secure PKI system, the ability to revoke certificates after they have been issued is needed. In order to do this, a protocol called OCSP is used to check the validity of a certificate against an online service, usually provided by the certificate issuer [25]. When the client receives a certificate it is able to send information about it to the online service and receive a response back indicating if the certificate has been revoked by the issuer or not. The URL of the OCSP server is provided within the X.509 certificate and is usually contacted via an unencrypted HTTP request because using HTTPS would require the OSCP server to present the client with a second certificate to verify, which could lead to a loop of OSCP requests. As a result, any attacker that is within the network and performing an attack on HTTPS using forged certificates could also intercept any OSCP requests and make them

return values indicating that the certificate presented is valid, even if it had been revoked. To stop this, the response is optionally cryptographically signed by the private key of the OSCP, although this again can lead to request loops.

With the growing number of websites adding SSL by default, the stress that this has put on the OCSP systems has also increased dramatically. In an ideal world, every time any connection from anywhere in the world is made to a secure website a request is sent to the OCSP server. This puts massive pressure on the servers, which have to perform a lookup for the requested certificate and then perform a cryptographic operation to make the response secured. An issue that has arisen from this is that when a request is made to the OCSP server, the original sender's IP address and the domain name is sent along with it. This adds an additional privacy risk to the SSL process and is not a risk that some applications will accept as it leaks sensitive information to a third party.

As a result, many applications such as web browsers do not implement OCSP checks at all, and when they do they do so with a fail soft option, meaning no response is a good response in case they receive no response back from the server. This means that an attacker could simply block any response from an OCSP request and the client will allow the certificate to be used, even if it has been revoked.

> *Currently all OCSP checking is "best attempt". That is, when you, or the system, evaluates a trust object (by calling SecTrustEvaluate) the trust object will attempt to contact the OCSP server. If it can contact the OCSP server and the server indicates that the certificate has been revoked, the trust evaluation will fail. If it can't contact the OCSP server, the trust evaluation will not fail (unless it fails for some other reason).* [26]

2.2 SSH

SSH is a secure networking protocol that allows remote access of a machine through a shell session, allowing the user to execute commands remotely. On iOS, both users mobile and root have the same default password – *alpine* – that gives access to the device once it has been jailbroken. While researching for this project, OpenSSH was used in order to gain access to the device as the root user to be able to navigate through the file system and discover useful files. It is also the method by which CertManager is installed onto the device, as Apple's Xcode application does not compile and install system modifications. Having SSH access to the device meant that the system log was accessible, which was essential for debugging during development and testing in the later stages of the project.

2.3 OpenSSL

OpenSSL is an open sourced implementation of the TSL/SSL protocol written in C. It provides developers with various utilities to wrap around the protocol and also contains cryptographic functions that allows for encryption and decryption operations [27]. Apple suggests that developers not wanting to use their SecureTransport for SSL operations should instead compile an alternative implementation to perform the same functions. For this project, OpenSSL was compiled for iOS [28] in order to access certificate information that is not available using the standard iOS frameworks.

2.4 Jailbreak

iOS is a UNIX based system, meaning that under the user interface there is a system running with different users and permissions. On iOS, the customer uses the phone as a user account called *mobile*, while the system runs as the user *root*, as is common to find on UNIX systems [29]. To jailbreak a device means to escalate the privileges of the user from mobile into root access, essentially to escape the 'jail' imposed on the mobile user on which files it can access and what processes it can run.

Apple actively block ways of gaining root access and so exploits are used in the software of both the iOS system and the device firmware to escalate privileges. Previous exploits used range from buffer overflows caused by opening a malformed TIFF image in the web browser [30], to exploiting leftover debug commands on the coprocessor to execute an unsigned payload over USB [31]. Apple usually patch these in software updates but those who want to keep their jailbreak will stay on old versions of software specifically for the ability to do so as iOS devices no longer support the downgrading of firmware.

This project utilises the jailbreak, as it needs to modify the iOS implementation of the SSL/TSL so that it applies system wide. Without a jailbreak the project would be limited to a single application running as *mobile*. While the system could still be implemented as a single application it doesn't have the same effect on increasing the security of the system. By creating a modification to the system using the jailbreak it means that any application or service on the device will be able to have its connection protected by CertManager.

2.4.1 Cydia Substrate

Cydia Substrate is a code insertion platform that allows developers to patch other processes running on a jailbroken iOS device [32]. It is the default framework used by developers to write system modifications and is developed and maintained by Saurik IT, who also run the popular Cydia package manager used by millions to install non-Apple approved applications and tweaks.

Substrate works by taking advantage of a feature of the Objective-C runtime known as method swizzling. Through the use of functions such as method_setImplementation and method_exchangeImplementations [33] developers are able to replace the original code of a function with their own implementation. A simple example would be:

```
function helloWorld {
    return "hello world"
}

function goodbyeWorld {
    return "goodbye world"
}

function main {
    method_setImplementation(helloWorld, goodbyeWorld)
    print(helloWorld)
}
```

Code 1 - Method swizzling pseudocode.

The code seen in Code 1 would print "*goodbye world"* instead of the text seen in the original function. The importance is that it is not just a temporary swap, any other classes that call the original function will also get the new implementation back. This is a very powerful feature of the Objective-C runtime and is used by jailbreak developers to add new features or modify behaviour of the system and applications. While this works well on its own, once multiple tweaks start modifying the same functions then issues can arise. Cydia Substrate solves these issues by providing its own API to swap functions, taking into account the inheritance hierarchy so that applications are more stable [34].

2.4.2 SpringBoard

SpringBoard.app is the main home screen for iOS, and is responsible for launching applications, showing the user icons and displaying notifications, amongst other things. The SpringBoard application is controlled by a singleton class, also called SpringBoard, which inherits from the UIApplication class so can be treated as an application in its own right. In this project, notifications were to be shown to the user alerting them when a certificate had been blocked and in order to do this the SpringBoard application needed to be patched in order to add a listening server into the process, awaiting messages from the blocking software in order to trigger a notification.

2.5 Compelled Certificate Creation Attack

As described by Soghoian and Stamm [35], the compelled certificate creation attack is a theoretical attack in which a government entity is able to force a certificate authority to generate a forged X.509 certificate. TSL/SSL is the protocol that gives the security of the network protocol HTTPS, which is used by millions of individuals and businesses to communicate sensitive information around the globe and is supported by a public key infrastructure. Due to the nature of PKI, any CA has the ability to issue a certificate for any domain and clients will accept this as legitimate because it trusts the CA that signed the certificate, regardless of if the certificate was the one given to the entity who owns the server.

In a normal SSL scenario, the website must first obtain a certificate from a certificate authority that it can use to verify its identity to customers. The certificate authority performs identity checks that the entity making the request is the owner of the website. Once the proof of ownership has been obtained, a certificate is generated for the requested domain and is signed by the authority's private key, establishing a chain of trust between the two certificates. The website is now able to use this certificate to serve HTTPS over a secure connection and the client makes a handshake request, whereby the certificate is returned back to it. The client checks the certificate to see if it is valid by building a certificate chain and checking that the root certificate is within it's own trust store.

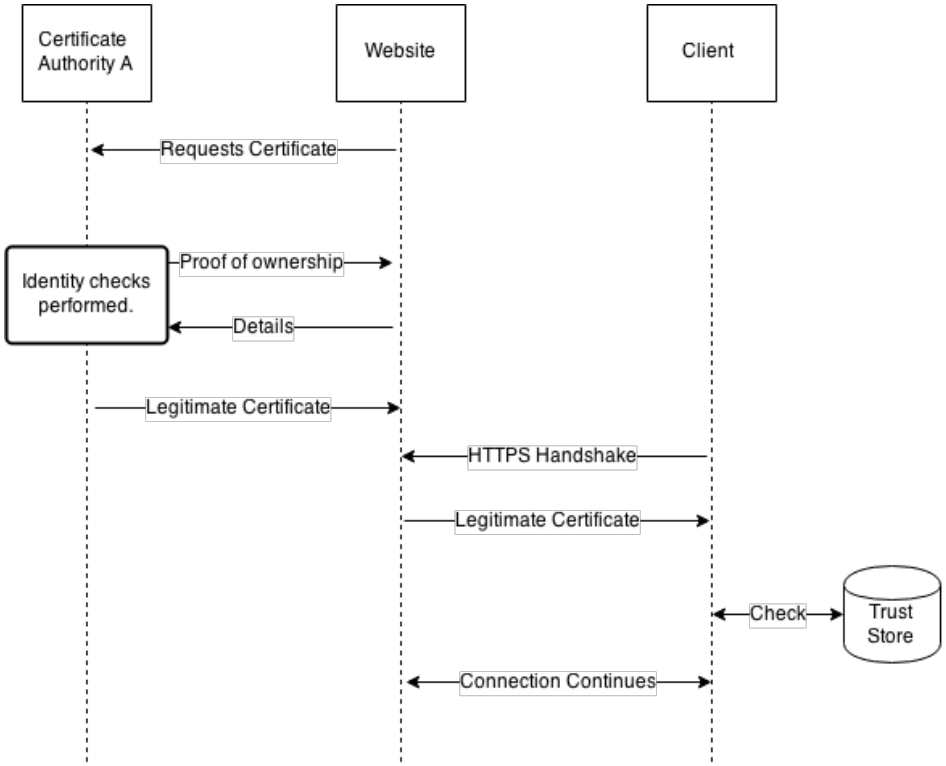


Figure 7 - A normal certificate creation process.

In the above example, the Certificate Authority A is a trusted entity and therefore the certificate received from the website is inherently trusted. The connection can now continue as normal, using the exchanged certificate for secure communication between the client and server.

An attacker is introduced to the scenario, in this case a government agency, although a malicious hacker obtaining access to the certificate authority can carry out the same attack. The agency in question makes a legal request to a certificate authority within it's own country who has no choice but to comply else face legal implications. The certificate authority generates a forged certificate for the domain name of the target website and returns it to the agency. The target makes a request to the website and the agency intercepts this connection by performing a MiTM attack. The response back to the client contains a certificate for the website as expected and the client checks the certificate to see if it is valid by building the certificate chain and checking that the root certificate is within it's own trust store. The certificate authority B is a trusted entity as far as the client is concerned and therefore the certificate received from the agency is inherently trusted. The HTTPS session is allowed to continue with the rogue agency able to intercept and read all communications without any warning triggered on the client side. All this time the client is never directly communicating with the website but instead the agency forwards all requests made and returns the responses.

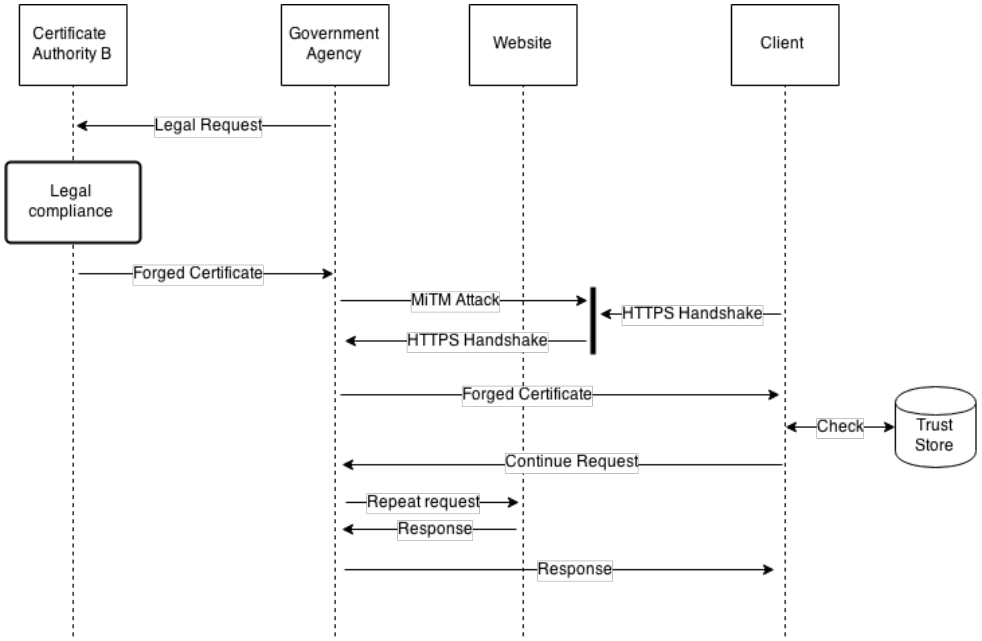


Figure 8 - The compelled certificate creation attack.

There exists an extension of this attack whereby the CA is forced to create an intermediate certificate for the rogue agency. This means that the agency can use this certificate to sign certificates for any domain they wish without needing to contact the CA each time, effectively extending the publishing permissions from the trusted CA to the agency. A certificate like this can be used in existing specialised hardware [36] in order to generate forged certificates on the fly for any secure connection request to perform a MiTM attack on all secure connections and not just those for a specified domain.

This attack is entirely plausible and while there are no documented cases of a government agency performing such an attack, there have been multiple instances of the same vulnerability being used. The certificates obtained from the DigiNotar attack were used to perform this exact attack, however instead of a legal request the certificate was gained by hacking their systems. Any security system is only as strong as its weakest link; it just so happens for public key infrastructure that its weakest link is the very foundation of its basis of trust, the certificate authorities.

2.6 Related Work

The scope of this project is quite unique in the fact that there is little research done into the iOS system as it is closed off. Without a working jailbreak, this method would be unable to function at all. While researching the iOS platform, several open source projects were found which used various methods to interact and modify the behaviour of the SSL implementation. These techniques were analysed and evaluated to see how useful they would be in the implementation of this project.

2.6.1 iOS SSL Kill Switch

The aim of this project is to remove all SSL protection on an iOS device in order for security researchers and application auditors to view the data being sent between applications and remote servers [37]. The kill switch uses a Substrate tweak in order to patch functions within the Secure Transport API, a part of the Security framework. This patch removes all certificate

validation performed by the system, by always reporting that the application calling the function is going to be performing its own certificate validation and then not implementing the validation.

Previous versions of this project patched the NSURLConnection class, a high level API provided by Apple for making connections to remote hosts. This had the drawback of not working with applications using custom networking APIs and by moving the patch to the Secure Transport API meant that it impacted the lowest level of SSL implementation found on the system [38]. This method of patching the system was better than it previously was and is useful to be aware of when implementing a system patch for this project. While Kill Switch does not have the same aims as this project, it does provide an interesting way of patching the system at the software level rather than by modifying values in files stored on the device.

2.6.2 iSSLFix

This project deals with providing an unofficial patch to iOS devices running iOS versions lower than 4.3.5 that were not patched after a bug was found in the way the system handled SSL requests [39]. Additionally this project provides a fix for devices running iOS versions lower than 4.3.2 against the DigiNotar and Comodo certificates that were used to perform MiTM attacks. This system does not use the substrate library and instead manually patches the *com.apple.securityd.plist* launch daemon file in order to inject the tweak as a dynamic library into the *securityd* binary, using code based on the open source substrate system.

iSSLFix is more in line with what this project sets out to achieve when compared with the Kill Switch project. With the addition of blacklisting certificate values it provides a way in which the system can reject certificates based on their values and this will be useful information to use when it comes to writing the code for the system modification of this project. The use of the manual injection code is down to needing to inject itself into the *securityd* daemon, something which at the time the code was written Substrate was unable to do. In iOS 8, Substrate is able to inject itself into daemons correctly and therefore this would be an unnecessary step to add in this project.

2.6.3 CACertMan

This application allows users on rooted Android devices to view information about the root authority certificates that come with the default Android trust store and block them. This project provides a look at how the front-end application for CertManager could be implemented. The way in which CACertMan works is by manually patching a file on the Android file system in order to remove the certificate from the file completely. This is quite a destructive method and requires a back up of the original store to be saved by the user in case they need to restore. In order to provide a safer user experience, CertManager would try and stay away from manually modifying the values from within system files, and look to implement the patch at the software level. The fact that this application can only modify existing certificates also stops users from adding additional certificates that they wish to block that may not already be pre-installed. An example of this would be if an intermediate certificate was compromised but the user didn't want to remove trust for the root CA, CACertMan would be unable to provide the means to do that.

2.7 Summary

This section has defined terminology that will be used throughout the remainder of the project and has researched the underlying technologies that will be used. The TSL/SSL protocol, public key infrastructure and X.509 certificates have been analysed in order to gain a greater

understanding of how they work together. This is essential for the success of the system as it builds upon the ideas defined here. In addition to the explanation of technologies, an attack scenario – the Compelled Certificate Creation Attack – was visualised based on the description provided by Soghoian and Stamm. This attack builds upon the ideas and opinions discussed within Chapter 1 that global mass surveillance is a realistic threat and that the weaknesses found in the TSL/SSL protocol are not weaknesses in the mathematics behind the cryptography but in the human elements of the protocol in the form of the Certificate Authorities. Finally, relevant work within the scope of this project is evaluated for its usefulness for the implementation process. Various applications are looked at to see how they implement changes to the default system SSL behaviour for both for iOS and Android and it was decided that a software based solution would be more flexible and provide a safer experience for a front-end application.

Chapter 3

Security Research

This chapter looks at the research conducted into the iOS system and the discoveries made in relation to certificate management and the secure transport of data. Findings that were directly related to this project were analysed for their usefulness in achieving the project aims defined in 1.3 Project Aims. The research looks at being able to extract certificates from the trust store that is stored on the device, modifying the trust settings for these certificates and then finally analysing the Security framework in order to establish the different methods that were called when an SSL connection occurred.

3.1 Trust Store

On OS X and iOS systems, the trusted Root Certificates are stored in a central location called the 'Trust Store'. The Apple Root Certificate Program is a service provided by Apple to help protect customers from security issues that can occur from the use of PKI certificates. For a Certification Authority to have its certificates added to the trust store they must meet the criteria set by Apple. This includes completing WebTrust audits [40] and conforming to various standards. Once they are approved they are added to the store, which is updated via iOS software updates to the devices.

This system works well on OS X, where the built in App Store application can install small security patches in the background. However on iOS, where updates are less frequent, bug fixes and new functionality are bundled together into major or minor releases, which take time to implement and deploy. On top of this, support for older devices is dropped with each major software release [41]. This results in devices still being used by people who will never receive updates to their trust store and therefore are always vulnerable to any future compromises in the certificate authority list.

The first stage of research was to gain access to the list of trusted certificates that are located on the device. In order to do this the source code of the Security framework was analysed and a function, *SecTrustCopyAnchorCertificates,* was discovered that allows the system to return an array of the default trusted root certificates that are used by the system [42]. The issue with this function was that it was designed to work for the OS X platform and not iOS, therefore when it was added to iOS code it wouldn't compile as it had been marked with a macro:

```
__OSX_AVAILABLE_STARTING(__MAC_10_3, __IPHONE_NA);
```

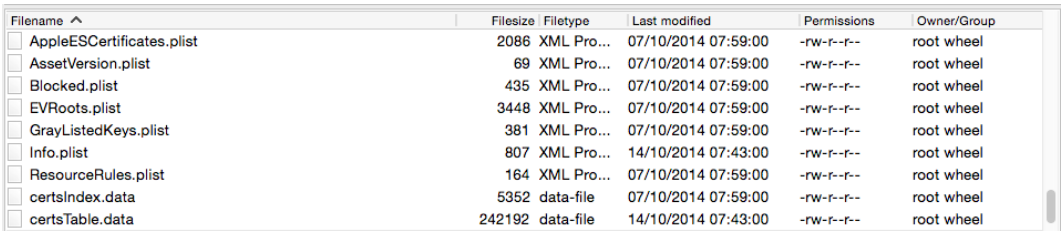
Code 2 - The availability macro defined by Apple.

This macro allows developers to specify the version at which this function first became available for use. In the above instance it began in OS X version 10.3 and was not applicable for iOS. This means that the function is currently not implemented on iOS and so can not be used to extract the certificate information from the system [43].

On OS X, users can be admins – they can make changes at the system level and so can add and remove certificates from the Trust Store. On iOS there is no concept of an administrator

from the user's perspective as they are a standard user and Apple administrates the system side of the device via software updates. It therefore makes little sense for Apple to need to expose the root certificates on the device with a public API. Instead, they tell developers to call higher-level methods in the Security framework in order to interface with secure connection utilities. The idea being that no application should need to view what certificates are installed on the device to perform normal application functions.

It was at this point that the decision was made to move away from looking at public APIs and to jailbreak the device in order to gain access to lower levels of the system and security framework. It was assumed that the certificate trust store would be stored on the device in the form of a database or binary image and so the file system was analysed. On iOS, frameworks are stored in the file system under */System/Library/Frameworks* and so the Security framework files were found inside there. Within the Security folder there were several files that contained information relating to root certificate authorities.



Filename	Filesize	Filetype	Last modified	Permissions	Owner/Group
AppleESCertificates.plist	2086	XML Pro...	07/10/2014 07:59:00	-rw-r--r--	root wheel
AssetVersion.plist	69	XML Pro...	07/10/2014 07:59:00	-rw-r--r--	root wheel
Blocked.plist	435	XML Pro...	07/10/2014 07:59:00	-rw-r--r--	root wheel
EVRoots.plist	3448	XML Pro...	07/10/2014 07:59:00	-rw-r--r--	root wheel
GrayListedKeys.plist	381	XML Pro...	07/10/2014 07:59:00	-rw-r--r--	root wheel
Info.plist	807	XML Pro...	14/10/2014 07:43:00	-rw-r--r--	root wheel
ResourceRules.plist	164	XML Pro...	07/10/2014 07:59:00	-rw-r--r--	root wheel
certsIndex.data	5352	data-file	07/10/2014 07:59:00	-rw-r--r--	root wheel
certsTable.data	242192	data-file	14/10/2014 07:43:00	-rw-r--r--	root wheel

Figure 9 - The contents of Security.framework on iOS 8.1

The large file size of the *certsTable.data* file indicated that it could contain information relating to the certificate data. The corresponding *certsIndex.data* file also seemed like a relevant file as it shared the same prefix. A search on the Apple open source repository for the file names lead to a file *OTATrustUtilities.c* [44] which confirmed the use of the *certsTable.data* and *certsIndex.data* file for extracting certificates from the file system for use within the security framework.

Analysing *certsIndex.data* using a hex editor gave a collection of random characters that is usually associated with viewing binary data. Within Objective-C there is a way of saving data structures to disk by converting them into *NSData* objects. Using this knowledge, the *certsIndex.data* file was loaded into various data structures using the built in functions provided. The structure that loaded correctly was that of an *NSDictionary*, the Objective-C class representing a standard dictionary data structure [45]. This dictionary object was traversed, the number of keys counted and output to the developer console to give a value of 223. At the time of research, Apple listed 223 trusted certificates in the 2014081900 version of the trust store that came preinstalled with iOS 8.1.1 [46] and therefore it was confirmed that the *certsIndex.data* file must contain the references to these trusted certificates.

The dictionary created previously had its values output to the console in order to analyse the contents of the file further. The dictionary consisted of a 40-character key and then an incrementing value linked to it. The character set for the key was [0-9a-f], indicating the use of the hexadecimal numerical scheme. Each pair of characters represents a single byte value and therefore twenty bytes of data were being represented in total. When converted into bits this is 160-bits long, which is the standard length of a SHA1 hash.

$$< 5e211447\ e2b673b3\ c8ee9f3f\ 3725085e\ 10459577 > = \ (21216);$$

Figure 10 - The output of a single entry within certsIndex.data

This structure is common to find within a Database Index file, a data structure that is used to speed up the lookup speed of a database [47]. Assuming that the shorter integer value is an offset for the position of the start of the certificate data in the *certsTable.data* file, it was theorised that the Security framework generated the hash of a certificate and then checked to see if it exists in the index file. It would then be able use this value to access the data from the *certsTable.data* file to read data from the offset. In order to prove this theory, the *certsTable.data* file was examined with the use of a hex editor and one of the values from the index file was used as an offset, which showed the beginning of a human readable string containing information representing a root certificate authority.

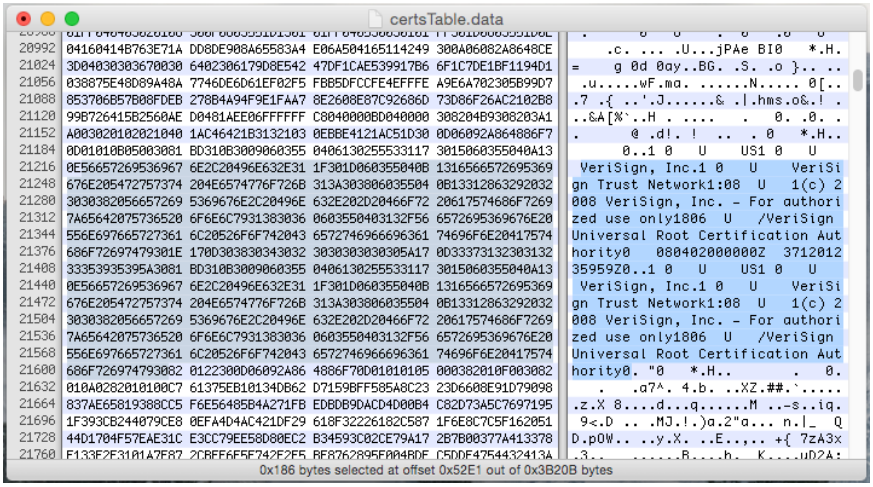


Figure 11 - The raw certificate data stored inside the table data file.

3.2 Modifying Certificate Validity

In order to confirm that the files were being used by the system they were modified in order to see the impact on the system. In order to do this, the public key for a certificate authority, CNNIC, was obtained by inspecting it via Google Chrome on a desktop machine. This value was then searched for within the *certsTable.data* file and was replaced with zeroes in order to remove it from the certificate store. The file was then uploaded back onto the device using SSH, overwriting the existing trust store.

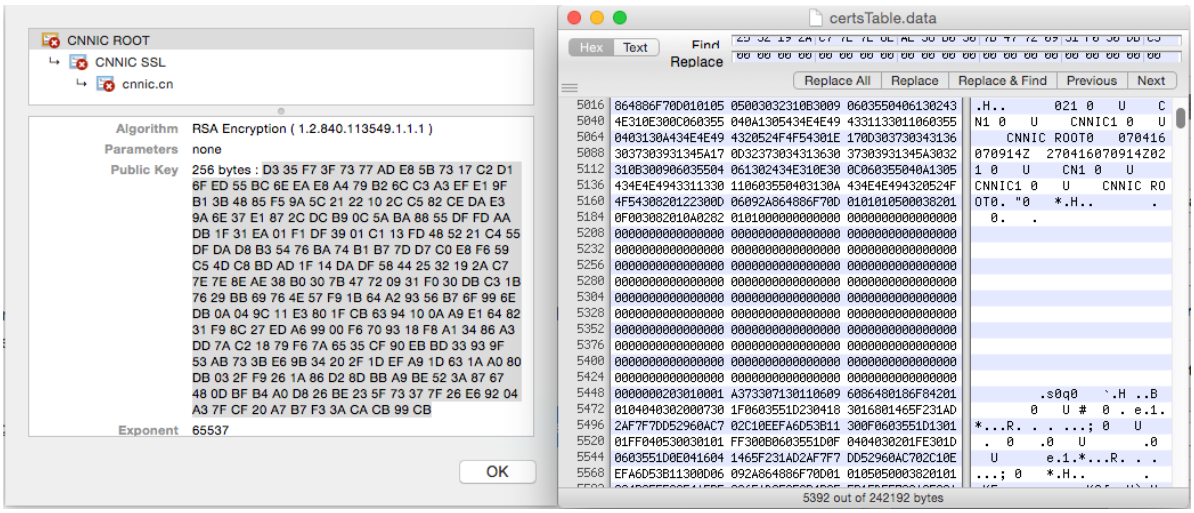


Figure 12 - The certsTable.data file with the CNNIC public key zeroed out.

After the file had been replaced the device was rebooted and a connection to *https://www1.cnnic.cn,* a website signed by the CNNIC Root certificate, was attempted. Previously, this website loaded without any problems however navigating to it after the certificate was removed gave a warning by the operating system that the server's identity could not be identified.

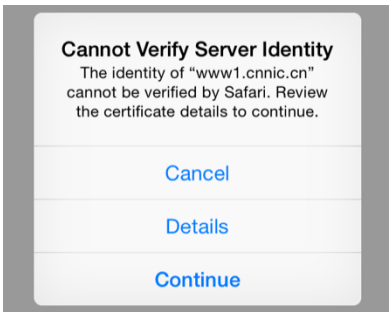


Figure 13 - iOS warning to the user.

It is the opinion of the author that this message does not give adequate information to the user that this could be a potential attack on the device. The vast majority of users would not understand the technical implications of this message and pressing *Details* provides more technical information about the certificate being presented. The most probable action from this message would be the user pressing *Continue* and allowing the connection to occur. When compared to the message shown by Google Chrome, the lack of severity shown by Apple's message is amplified.

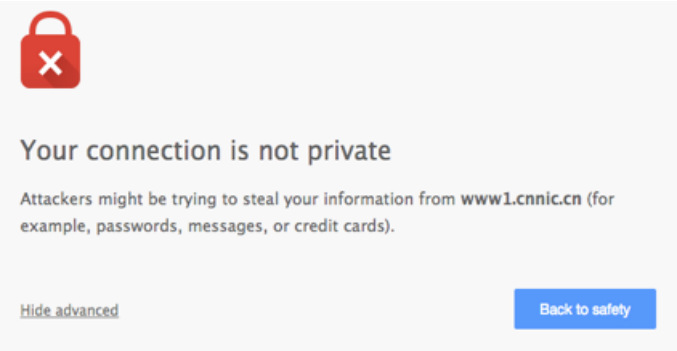


Figure 14 - Google Chrome's warning to the user.

The fact that trust was not given to a certificate provided by the CNNIC root certificate after it had been zeroed out manually at the file system level proved that these were the files used by the system for making trust decisions.

It is also worth noting at this stage that the system still booted and SSL connections worked when the trust store had been modified, this indicates that there is no integrity check performed on the store to make sure that it has not been modified. Potentially, certificates could be added to a device's system trust store so that they could not be uninstalled by the user and then be used to perform MiTM attacks.

The next stage of research was to find a way to dynamically add and remove trust for certain certificates. Modifying the database file worked fine in order to test a theory but would be a dangerous action to perform in an application, introducing the possibility of corruption to the files and meaning a backup of the trust store would be needed, as seen in the related work completed by the application in 2.6.3 CACertMan.

Within the Security framework folder extracted from the device, there existed a *Blocked.plist* file that contained an array of items and 40-character values. Based on the previous use of SHA1 values for representing certificates found in *certsIndex.data* it was assumed that these were also SHA1 representations of certificates used for dynamically blocking certificates that had been revoked, acting as Apple's own certificate revocation list [48].

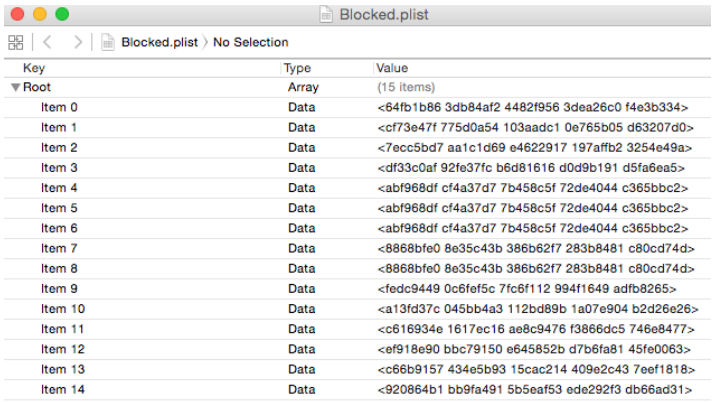


Figure 15 - The SHA1 values inside Blocked.plist

Searching online for these values found a match of a certificate hash [49] that had been wrongly issued by certificate authority TÜRKTRUST. Further reading found that TÜRKTRUST had issued two intermediate certificates by mistake to customers that could have been used to perform attacks on clients and that both these certificates had been revoked [50].

In order to test the use of this file within the system trust process, the trust store was reverted to its original state and the SHA1 of the CNNIC root certificate was added to the *Blocked.plist* file. After rebooting the device the same website as before was accessed using the Safari application without any issues. This suggests that root certificates that are in the plist files are not enforced by the system implementation.

According to Apple [46], there is a group of *always ask* certificates that trigger an alert to the user. There is an additional file *GrayListedKeys.plist* and comparing the first entry of this file with the entries on their website confirms this file contains those certificates. The certificates matched based on the X.509 Subject Key Identifier value, which at the time was interesting as the original tests had been using the SHA1 checksum as a means of identifying the certificates. As a result of this discovery, the previous test of adding the CNNIC root certificate to the *Blocked.plist* file was repeated, but using the SKI value instead. Unfortunately this had no impact to the outcome and the connection was still successful. In order to gain a greater understanding of these *plist* files, different certificates were added to them in order to see the impact that it had on the trust settings.

Certificate	**SKI**	**Property List File**	**Outcome**
cnnic.com	91 C4 ED 26 C5 25 EB FC D7 39 A2 6A AE C0 8A F3 B5 24 29 5F	Blocked.plist	Website blocked
CNNIC SSL (Intermediate)	45 00 BA 8A 18 90 51 C3 B1 CA F7 BC 65 39 2E 8C 56 90 44 30	Blocked.plist	Website blocked

CNNIC ROOT	65 F2 31 AD 2A F7 F7 DD 52 96 0A C7 02 C1 0E EF A6 D5 3B 11	Blocked.plist	Website loaded
cnnic.com	91 C4 ED 26 C5 25 EB FC D7 39 A2 6A AE C0 8A F3 B5 24 29 5F	GrayListedKeys.plist	Warning
CNNIC SSL (Intermediate)	45 00 BA 8A 18 90 51 C3 B1 CA F7 BC 65 39 2E 8C 56 90 44 30	GrayListedKeys.plist	Warning
CNNIC ROOT	65 F2 31 AD 2A F7 F7 DD 52 96 0A C7 02 C1 0E EF A6 D5 3B 11	GrayListedKeys.plist	No warning

Table 2 - Results from testing different certificates in the plist files.

The results from this test indicated that the property list files found within the Security framework folder only work for certificates that are not already in the trust store. Adding SKI values to *GrayListedKeys.plist* resulted in the system showing an alert to the user asking for confirmation that the user trusted the certificate, while adding them to *Blocked.plist* resulted in the connection failing gracefully. Based on this behaviour, it is assumed that the property lists do not have the ability to override the default root certificates from within the trust store. This is a logical design decision by Apple, if there were ever the need to block a root certificate then they would remove it from the trust store completely rather than override it with a property list value. Once the certificate is no longer trusted then there is no need to keep it stored as the trust store acts as a whitelist, only allowing specified values, whereas the property lists are blacklists, allowing everything except those defined within them.

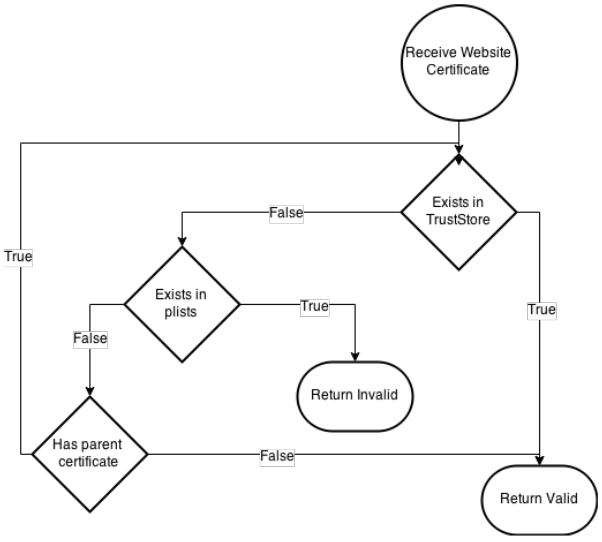


Figure 16 - The logic flow for checking the validity of a certificate.

It is worth noting that the security policies are only reloaded when the device is rebooted, using the property lists to enforce certificate checking would require the user to reboot the device every time a change is made. If this method were used for CertManager then the user would not be instantly protected when they changed their trust settings, perhaps providing a false sense of security unless a reboot was forced by the application after changing an option.

3.3 Analysing the Security Framework

The two previous attempts of blocking certificates both involved modifications of files used by the security framework itself in order to trigger the blocking logic and stop the connection. While this can be used effectively within a testing environment, it does not provide a clear way in which to build an application on top of. The next proposal for implementing the certificate block was to look at the next level up from the file system files and that was the Security framework itself. There is only one way to interface with the trust store on iOS and that is via this framework and so all secure connections must make calls to this process and as a result it provides the perfect location to implement additional checks on certificate validity.

In 2011 there was an attack on Dutch CA DigiNotar that required their root certificates to be removed from all trust stores. Modern browsers with their own trust stores were able to perform OTA updates but Apple had to wait until the next software update to include an update to the iOS trust store. The jailbreak community already had root access to the device and in response to the attack a developer created a project to patch the securityd library on iOS to make sure those certificates were not used [51]. Additionally, a piece of iOS malware patched the iOS Security Framework, sitting on top of the SSLWrite function to extract plaintext data before it was encrypted [52]. Both of these were achieved by using Substrate to replace the Security function implementations used by the system.

It was decided that the blocking code used by the CertManager system would use a similar method of using the Substrate library to implement the blocking logic. In order for the system modification to work, it needed access to the original methods that were called by the system. It was important at this stage to find methods that had access to the certificate data so that the information could be extracted and checked against the trust settings provided by the user. Additionally, the original method needed to be low enough within the system levels so that all connections used on the device would be affected by the blocking code, and not just those using certain implementations of accessing secure HTTP. This is because there are multiple ways in which a developer could create a secure connection. There are three main layers of networking API in iOS, NSURL, CFNetwork and BSD sockets, with each one being built on top of the other to provide extra functionality [53].

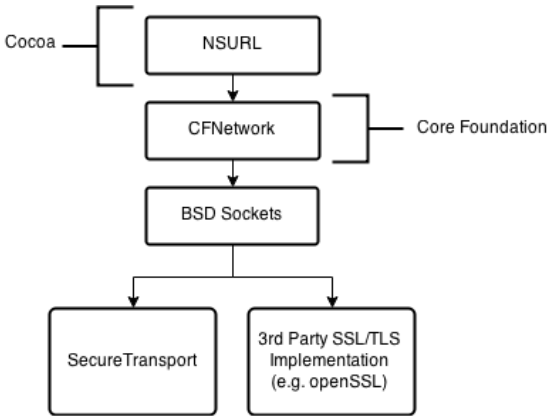


Figure 17 - The different layers of networking in iOS

The Foundation framework, which is a part of the Cocoa API (Cocoa Touch on iOS), provides basic Objective-C classes and data structures that are commonly used while developing applications for iOS and OS X. It has roots in NeXTSTEP, an operating system from the early 1990s that was acquired by Apple in 1996 [54]. The acquired system was merged with Apple's

existing OS, Mac OS 9, to create OS X and the “NS” prefix found in common class names is legacy of the **N**eXT**S**TEP name [55].

NSURL is a part of this Foundation framework and deals with high-level network connections [56]. It is usually the API of choice for developers looking to implement a connection to a domain or server due to its high level abstraction away from the underlying connection logic. This allows NSURL to be used to obtain a resource from a remote server or local file system without the developer needing to worry about dealing with any networking code. NSURL is built upon CFNetwork, which is a part of the Core Foundation API.

Core Foundation was written as a way in which existing OS 9 code (later Carbon), could interact with the NS code (later Cocoa) to act as an intermediate API that both would be able to use to talk to each other [57]. It is written in the C language and provides a lower level implementation of many functions found in Foundation. CFNetwork is based on top of BSD sockets, the lowest level of networking on the system, however it’s security functions are rooted in the SecureTransport layer.

Berkeley sockets, more commonly referred to as BSD sockets, is a low level networking API for interacting with internet and UNIX domain sockets. Use of BSD sockets on iOS is often discouraged because it does not activate the cellular radio or VPN services that higher level APIs do [58]. Therefore it is very rare to see BSD sockets being used in most applications, unless there is some legacy or performance reason to do so.

According to Apple, if a developer chooses to use BSD sockets directly they must implement the SSL or TLS manually by either using the Secure Transport API or by downloading an alternative SSL or TLS implementation such as OpenSSL [59]. Therefore the Secure Transport, part of the Security framework, was analysed to see if there were functions that could be overridden to implement the blocking logic. The Apple documentation for Secure Transport [60] lead to the creation a flow diagram to show the different stages that are executed when an SSL connection is requested within the framework.

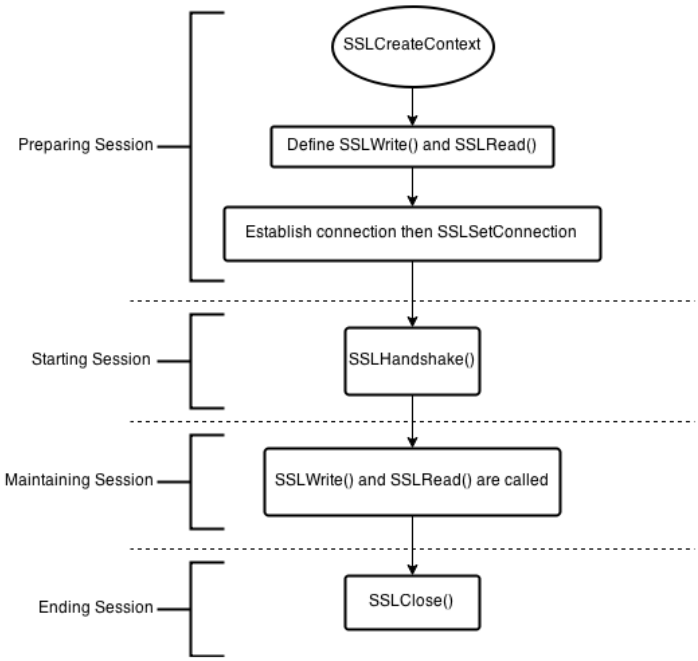


Figure 18 - The SecureTransport life-cycle

In order to access the data available to these different methods, a small logging utility was built using the Substrate system to override each function highlighted within the SecureTransfer class. The role of this modification was to log the input data and then return the result of the original function so that the system behaviour was unchanged. Starting at the lowest level, an override for the SSLCreateContext function was created. At this stage the parameters that the method had access to were:

Parameter	Usage
CFAllocatorRef	The allocator to use, used for allocating memory.
SSLProtocolSide	Which side of the protocol is being used, client or server.
SSLConnectionType	What type of connection, TCP or UDP.

Table 3 - The parameters of the SSLCreateContext method.

At this point there was no data that allowed access to certificate information, this was due to the connection not being set at this point. The next stage was the SSLSetConnection function, which was expected to contain more information than what was available before.

Parameter	Usage
SSLContextRef	An SSL session context reference.
SSLConnectionRef	An SSL session connection reference.

Table 4 - The parameters of the SSLSetConnection method.

Here there was a reference to the SSL context and a reference to the SSL connection object that is created by some higher-level framework such as CFNetwork, Sockets or Open Transport. However once again there is no reference to the certificates available. Referring back to 2.1 TLS/SSL, it isn't until the client requests the handshake with the server that the server then responds with it's public key certificate, therefore it makes sense that these wouldn't exist yet.

The next method that was called was SSLHandshake, the method that performs the SSL handshaking process. This method is provided with a single parameter *SSLContextRef*, that contains a reference to the SSL session that was created previously. When this object was given to another function within the Security framework, an array containing the certificates used for the session was returned. This confirmed that the location to implement the blocking code could be achieved from within the SSLHandshake method.

```
com.apple.WebKit.WebContent <Warning>: Summary: *.facebook.com
com.apple.WebKit.WebContent <Warning>: Summary: DigiCert High Assurance CA-3
com.apple.WebKit.WebContent <Warning>: Summary: DigiCert High Assurance EV Root CA
```

Figure 19 - Console output from traversing the chain of trust for Facebook.com

3.4 Summary

To conclude this chapter on iOS security research, a summary of the results is evaluated. The test device used for research was firstly jailbroken in order to gain full root access to the device. This meant that files used within the Apple framework folders on the file system were able to be browsed using SSH. Files were found within the Security framework folder and their contents were analysed using a hex editor to establish the location of the iOS trust store. At this stage, while it seemed likely that these were the files being used, it was not proven and so further research was needed to confirm this.

The contents of the files were modified to manually remove the trust for a root certificate. This was achieved by removing the public key of the certificate so that attempts to verify the validity

of any certificate signed by it would fail. This was the first test of the research and proved to be an effective way of blocking certificates on the device. It also confirmed that these files were the basis of the entire chain of trust used by the system for SSL.

Additional ways to implement a blocking mechanism were looked for, including the use of two property list files within the Security framework folder. Tests were conducted on the files to see the impact adding the SHA1 representations to them had on the trust of a certificate. These files were unable to block root certificates but did provide a way to remove trust for intermediate and normal certificates. Using the knowledge gained from the manipulation of the plist files the algorithm for how the system treats the certificates defined in the trust store and plist files was able to be visualised.

As a result of the tests, it was decided to look away from a file-system level implementation and to research into the software that was responsible for reading these files. By studying projects that had performed similar work it was decided that the best way to implement this would be by using Substrate in order to perform method swizzling on the existing Apple Security framework methods.

The Apple documentation was studied in order to gain an understanding of the SecureTransport layer of the Security Framework and a flow diagram was created to help understand the different stages of logic when creating a new SSL connection. By using this diagram, a tool was created using Substrate to hook into these stages to see what information was available at each step. The results of this testing showed that the SSLHandshake function was the lowest level in the Security Framework where access to certificate information was available and that this is where CertManager should implement the software block logic.

Chapter 4

System Development Strategy

Designing the system is an important stage of the project whereby the system aims defined within 1.3 Project Aims are turned into a realistic design. In this section, the design process for the application in order to meet the requirements is discussed from both a technical and user perspective. The architecture of the system is defined and provides an overview of how the system will work to provide an interface to the user while also being able to make modifications to the underlying system. Lastly, the software development strategy used throughout the project is outlined and evaluated in its efficiency.

4.1 System Description

The system is composed of two separate projects, a front-end application named CertManager and a system modification named CertHook. The design for the system is that the user will interact with the front-end application and the technical implementation of blocking the system certificates will be achieved within CertHook.

4.1.1 CertManager

CertManager is the front-end application for the system and is where all interaction between the user and the system occurs. Within CertManager, the user is able to view the certificates that are pre-installed on the device in the form of the trust store. From here, users are able to select if they want to trust or block certificates on an individual basis. For example, the user may not expect that he will be making any connections using the CNNIC root certificate, he is also aware that the certificate may have been used to perform MiTM attacks, therefore he chooses to block the certificate on his device.

Additionally, the user can choose to manually add their own certificates to block. This is done by providing the SHA1 hash of the certificate and this will be added to the list of blocked certificates that is checked for when a secure connection is attempted. A web browser has been implemented into the application that provides a way for users to quickly find out the chain of trust the website is using. Clicking on a certificate in the chain provides a fast way for users to block the certificate by adding it to their manual block list.

While it is useful to have the connections blocked in the background, it is also important that the user knows when a connection has been attempted. In order to do this, a logging feature has been added to the application that reads attempted connections from a log file and provides a list of them to the user, showing important information such as which process made the attempt, what certificate was used and what time the attempt occurred.

4.1.2 CertHook

A jailbreak system modification was created using the Theos pre-processor and build system. The system modification used Substrate to implement a check on all incoming and outgoing TSL/SSL connections performed on the device. Based on previous research, it was decided

that the functions that it would override would be the SSLHandshake method as it is suitably low in the system level and because it occurs before any sensitive data would be potentially sent over the network.

Once CertHook has been injected into a process it will replace the SSLHandshake function in the security framework that has been loaded. This is where the additional check occurs for the trust settings based on the user preferences saved in a data store. Once the check has finished, a response is sent back to the original entity that called the function, replacing the original handshake method if it was found to be not trusted. If it trusts the certificate then the original function is called and the value created by that is returned as normal. This is a design decision that is important to the overall security of the device as it is unknown what additional checks the SSLHandshake method performs. It could be possible that the certificate has been revoked or expired and by calling the original method the normal trust decision performed by the system can continue.

In order for the two systems to work together they share a common storage location for user preferences and a notification system so that the application can alert the modification that the trust of a certificate has changed so that blocks occur immediately.

4.2 System Architecture

This section of the report will define an overview of the system architecture.

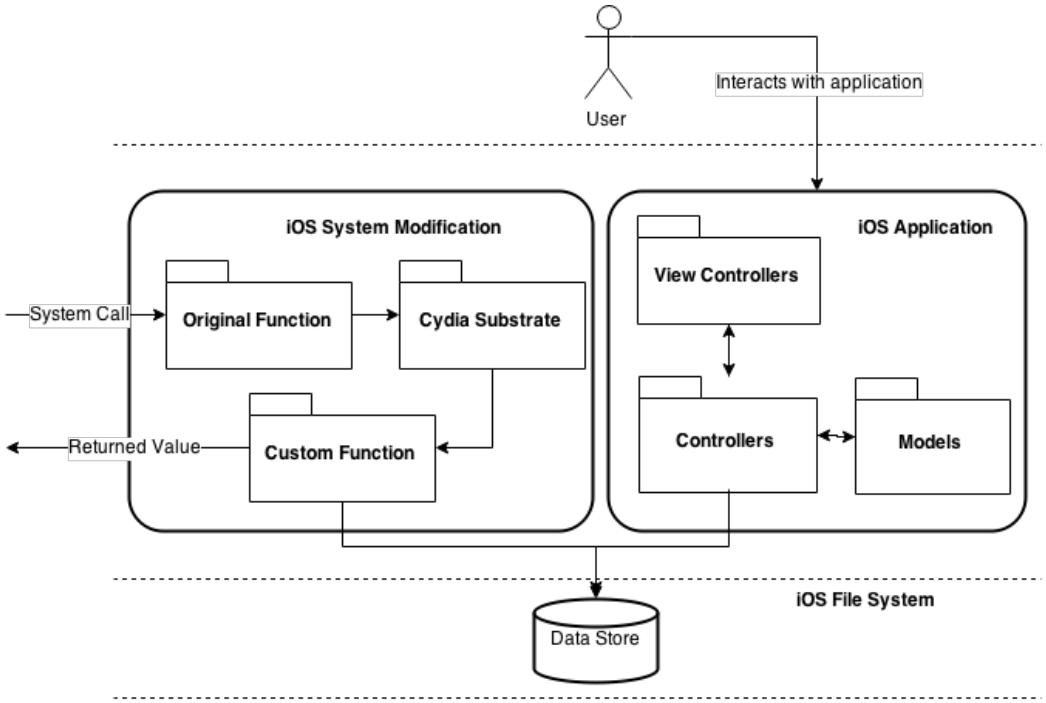


Figure 20 - System architecture overview diagram.

As discussed previously, the system is split into two sections. CertManager follows the standard pattern for iOS application development through the use of MVC architecture in order to keep code separated logically and in a way that is maintainable. When developing for iOS there is a frequent use of View Controllers, which is a slight variation on the traditional MVC model. Views shown to the user are owned exclusively by a corresponding subclass of

UIViewController which contains the logic needed by the system to display a view to the user. In addition to the standard UIViewController, CertManager takes advantage of UITableViewController, a more specialised type of view that displays data in a table.

When a user adds or removes trust for a certificate, it is persisted on the disk in a location where it can also be accessed by CertHook. By doing this, a link between the two applications can be established without coupling them into the same application. There are no dependencies between the application and the tweak as they are able to be built and run without the other. The data store is required for the functionality of the tweak to work but potentially the CertManager application could be replaced by another front-end and as long as it writes to the data store in the same format then CertHook will still work.

The iOS system modification, CertHook, follows the design pattern found in all Substrate modifications. The Substrate process runs in the background and is in control of detecting what processes are started on the device. Once the process is loaded then substrate checks its own data store for which system modifications should be loaded into that process, if any. This can be defined by setting a filter in the modification to say which processes it is built for. In the case of CertHook this filter is left intentionally blank so that it is injected into all processes that are started. This is vital to the success of the system as all network communications need to be protected, not just those provided by front-end applications.

4.3 Inter Process Communication

Due to the split nature of the system, it is essential that both the main application and Substrate modification are able to communicate and share data.

In order to achieve this, a common messaging scheme was used between both systems and data was stored in a location accessible to both. To create a way of sending messages between the systems *CPDistributedMessagingCenter* was used. This is a wrapper around the Darwin kernel notification system that is able to broadcast between processes. The messaging centre is a part of the private AppSupport framework provided by Apple, and works using a client and server interface. Clients can register against the messaging centre using an identifier in order to listen for messages, while the Server creates messages containing the same identifier and an optional dictionary of values. The messaging centre then handles the transfer of the data between the two processes meaning that a message from CertHook could be sent whenever an SSL handshake failed.

There are two uses of IPC within the system design, the first is to communicate between separate instances of the CertHook system modification. This is so that notifications can be sent to a single process and displayed as a local notification to the user. The second use of IPC is when the CertManager application detects a change in the trust settings of the certificates. In order to keep the CertHook application up to date, a notification is posted across the system to trigger a refresh of the files used by CertHook.

As the messaging centre is part of a private framework, it is not publically documented. Therefore in order to know what functions were available to use, a header dumper tool was used in order to extract the header file from the compiled Objective-C code [61]. Once a copy of the header file had been obtained functions were searched for those that looked likely to create a server and client and these were inserted into the code in order to send and receive messages between different instances of CertHook.

4.4 User Interface Design

The barrier to entry of the system for a user is the requirement to have a jailbroken iOS device, something that will most likely correspond to having technical knowledge regarding computing. Technically competent people are most likely to use the application and therefore it was felt that the interface would not have to be abstracted from the technical level. As a result, technical jargon has been used within the application text and the user is expected to understand what a SHA1 hash is for manual insertion of certificates. This system is treated as a utility for advanced management of the operating system and not a simple front-end application to be used by non-technical people.

One of the important parts of user interface design is the creation of wireframes in order to provide a visual plan of what the basic layout of the application will be and how different elements of the view will interact with others. Having wireframes was a good place to start developing from, as iOS development is very View Controller centric, meaning that it is harder to write logic without first having a way to interface with the application. Being able to plan the code structure around the design of each page rather than around the system logic meant that working user interfaces could be created quickly with mocked data behind it to get a feel for the application. It also meant that time was not consumed writing business logic before it was fully known what the outcome of the design would be as there may have been parts of the logic that was not needed. Having a clear design strategy lead to less redundant code and saved valuable time throughout the project.

The application was designed to be as minimal as possible and to conform to the iOS Human Interface Guidelines [62]. In order to achieve this, list views were utilised in order to show data to users as this is a common way of showing large amounts of data in a user friendly way. Originally, the ability to toggle the trust of a certificate was by swiping each row of the table to the left to display two buttons. However after user testing the application, the feedback was that the ability to change multiple values was tedious as it took too many swipe events. As a result of this feedback a toggle switch was added to each row instead and the swipe ability removed, this meant that the status of each certificate was clearly visible and it was faster to modify multiple values.

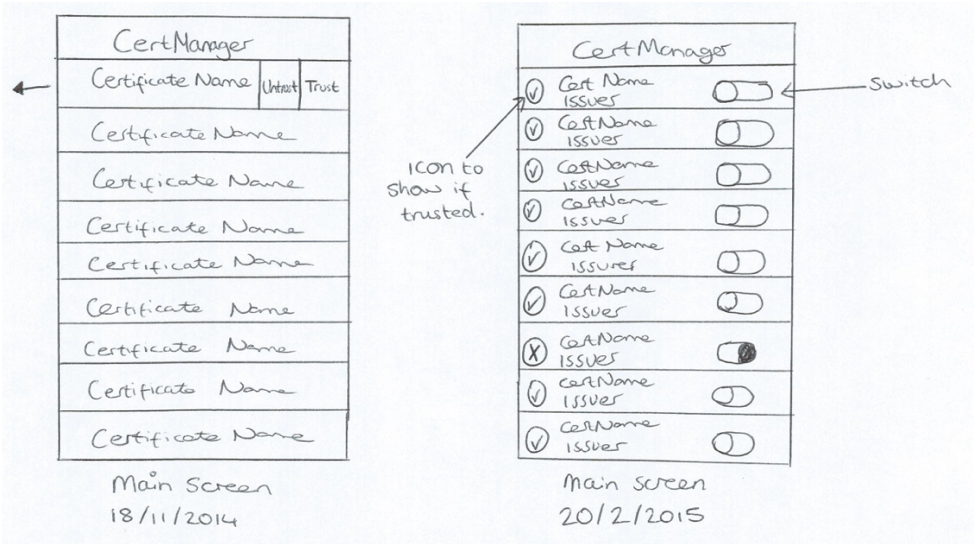


Figure 21 - The initial (left) and revised (right) wireframes for CertManager

In the first iteration of design, the application had a single table view listing the root certificates installed on the device. Due to the iterative nature of the project, once the back-end code had caught up with the front-end after more research had been conducted, it was decided that additional features would be added to the application in order to make it more useful for users. A tabular design was chosen in order to display to the user the multiple view choices in a clear and concise way, with the use of open source icons [63].

It was at this stage that the idea to allow users to add their own certificates into the application was realised. As the foundation of the blocking code had been established, additional view controllers were added and designed in a similar way to the original screen. Pressing the add button causes an alert to show the user with two text fields where they can choose to either block or cancel the action. Adding a new certificate to block requires the SHA1 of the certificate and this is validated by the system to make sure it is a valid SHA1 value.

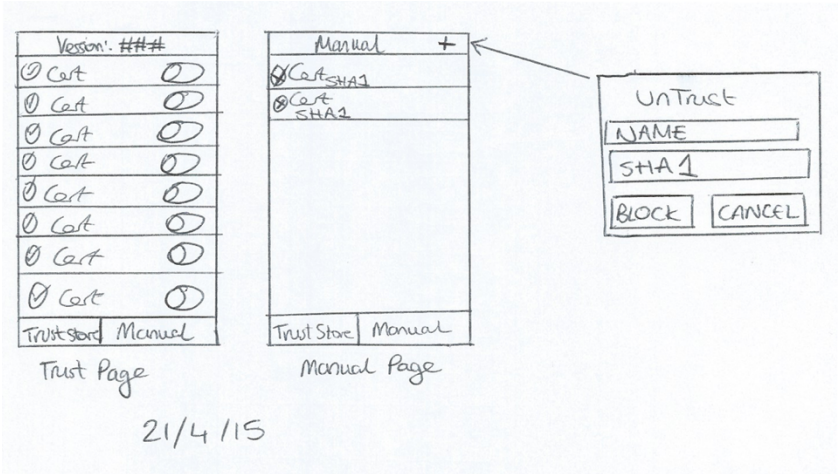


Figure 22 - The wireframes for the manual page.

The next design change was the addition of the logging system, a place in the application where users could see what certificates had been blocked and by what process. This was shown again in a list view due to the large number of logs that could potentially be shown. Important information about the certificate, mirrored from the existing information shown by a notification, was added into the cell so that users could view the details.

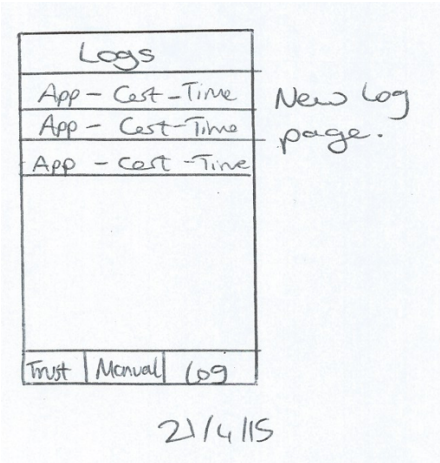


Figure 23 - The wireframe for the Log view.

In the early stage of the project, there was to be a second application in the system named CertBrowser, which would allow users to view the certificates that a website was using. The code was written for a basic web browser as its own project but, after the tabular design of the

main application was added, it made more sense to add it as a new view to the main application. As a result, the existing code and design for the browser was merged into the main CertManager project.

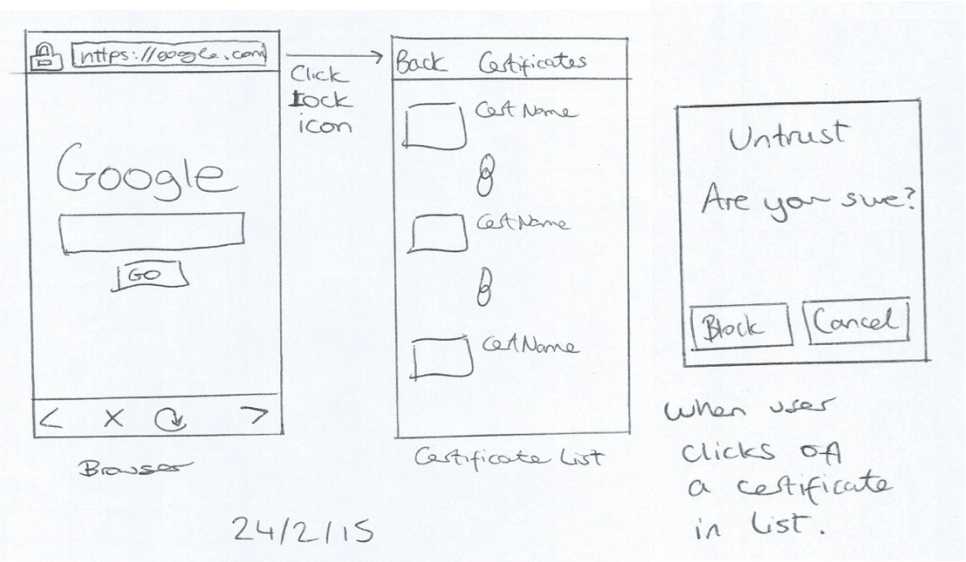


Figure 24 - The CertBrowser wireframe, with additional blocking functionality.

By changing the design of the application to include the browser it allowed for new functionality to be added which would not have been possible before. As the browser had access to the existing shared code used to talk to the file system it was possible to have the user tap on a certificate that was currently being used in a connection and provide the means to add it to their manual block list.

Designing before coding was an important part of the development lifecycle, which allowed for the expansion of ideas to improve the application without being held back by the technical details. Once an idea had been developed then it was easier to envision how to achieve it in code. Over the course of the development lifecycle, the design and functionality of the CertManager application changed from being a way to simply turn on and off the root certificates, into an application that can add custom certificate blocking to the system and also includes a web browser that allows the user to see the chain of trust of the current website, a feature prevalent in modern desktop browsers but something that has not caught on in the mobile world.

4.5 Software Development Methodology

In software engineering a software development methodology is a framework that is followed in order to help plan and control how a project will be executed, and ultimately how the product will be realised [64]. When this project began it required a large amount of research into the underlying system before any development could begin. As a result of this, normal software development methodologies did not apply well to this project as the requirements were in a constant state of change.

The way in which this project was mostly developed was through an incremental development process whereby prototypes were created after the required research had been completed [64]. If it were to be compared to any standard methodology it would most likely fall towards the Spiral methodology.

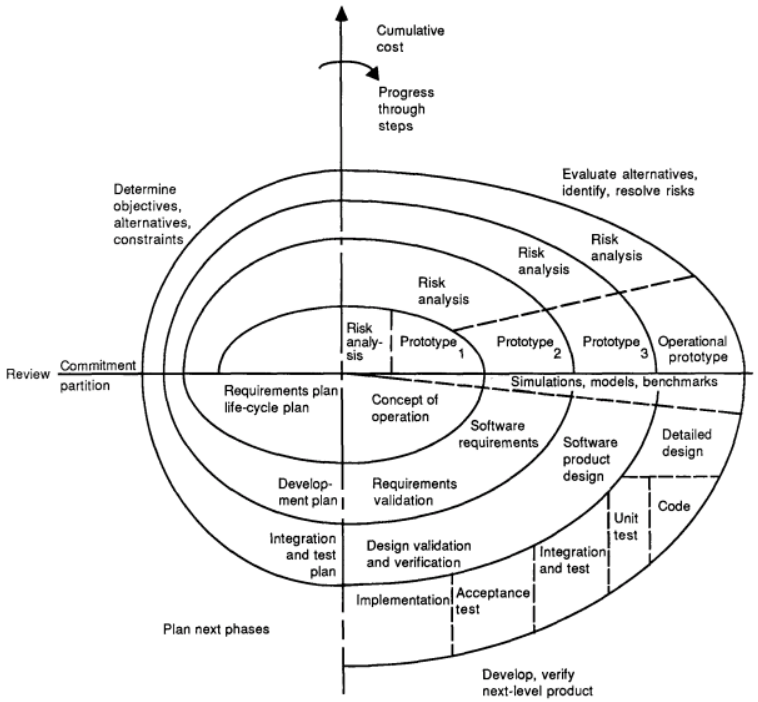


Figure 25 - The Spiral development methodology as described by Boehm. [65]

With Spiral, the project is split into cycles that contain repeating actions:

- Planning phase.
- Risk analysis phase.
- Software development phase.
- Evaluation phase.

In the case of this project, the risk analysis phase became more of a research stage, whereby the iOS system was analysed and the technical ways of implementing the system were discovered. Sometimes the research phase took longer than the development phase, but this was to be expected as the time it was going to take to reach a conclusion was an unknown variable when each cycle began. At the end of each cycle a prototype was developed which was also linked to a milestone, this meant that each cycle had a goal to achieve by the end with something to be delivered.

One of the benefits of using the Spiral methodology was that work was able to be broken down into smaller segments that were easier to manage. Each section built on top of the last until the final product was completed. A downside of this was that quite early on in the project there was an application that looked like it was working, however there was still the back-end development to complete. This was especially the case for the CertManager application because of how easy it was to create the interfaces using mocked data in the back end. This produced what is known as technical debt, where code still needs to be written to consider a milestone of the project complete. Leaving this technical debt would mean that more work would need to be completed in the future [66]. In order to combat this, the front-end view code was treated more as a design process whereas the technical coding was viewed as the actual task needed to be completed. When a new milestone was started, it would not be considered complete until the business logic behind the task had been completed too.

4.5.1 Milestones

As work was completed in iterations, there were several milestones that were being worked towards from the beginning of the project to the end. After each milestone was reached, a working prototype of the application was available for testing which allowed for modifications and additions to the system that would not have been realised if the software had been written all at once. The milestones that were set were linked to the project aims defined within 1.3 Project Aims and were:

- Creating a working iOS application that could be installed on a device.
- Get a list of certificates from the trust store on the device.
- Allow the user to toggle the trust of the certificates.
- Save the user preferences to the disk.
- Create a working iOS substrate tweak that could be installed on a device.
- Load the user preferences from the disk.
- Override functions from within the Security framework.
- Block untrusted connections from occurring.
- Alert the user that a certificate was blocked.

4.5.2 Git

When developing a piece of software it is important to be able to track progression and revert code back to previous states. Version control systems provide a way in which this is possible by tracking changes made to files as they are modified. These changes can be collated into commits that are then saved into a repository against a reference number. The commits can be given titles and messages in order to track changes made to code in a readable way and entire file structures can be reset to points within its history. Version control also adds a layer of accountability to code by logging who made the changes to individual lines of code so they can be traced back to individual developers at a later point.

Git [67] provides advantages over other version control systems such as SVN, including the fact that each developer holds a full copy of the repository on their system, and this means that a constant network connection to a central repository is not needed. The Git version control system was used throughout the development of this project in order to provide a way in which different features could be tested on new branches before being merged into the main code base. Doing this meant that issues and bugs were easier to find and reverting code back to previously stable states was trivial. In addition to these benefits, it also enabled the integration of the development process into a smooth workflow whereby milestones defined in 4.5.1 Milestones could be broken down into individual pieces of work and then as individual commits. Through the use of small and frequent commits, it meant that code changes were traceable and bugs that appeared later on in the development process could be identified in a clear and cohesive way.

Git also addresses the issue of losing code due to hardware failure or from accidently deleting work on a local machine. This project was developed using the popular online service GitHub in order to provide a remote Git repository to push code to. This meant that after each commit was made locally, it was also pushed to a remote server. The GitHub website provides a useful interface to interact with different versions of the code and allows for additional contributors to provide patches to the project in the form of pull requests. It is the aim of the project to become an open source piece of software available for developers to improve and contribute to, GitHub provides the perfect platform to enable this and is often the platform chosen by many individuals and companies working on open source projects.

4.6 Summary

Within this section the system has been formally defined and the way in which the project has been completed has been explained. The system has been broken into two main sections, CertManager, a front-end user interface whereby the user can interact with the system and CertHook, a system modification using the Substrate library which is where the certificate checking logic is implemented. These two components for the application are joined by a common file storage location in order to interact with each other. The decision to structure the application architecture this way was due to research into existing similar systems who also use the Substrate library in order to interact with the Security framework.

The user interface was designed before writing the code in order to provide structure to the development process. By designing views first it meant that less redundant code was written and less time was wasted on designing while coding. The application interface was tested with users and feedback given was reflected in iterative changes to the design.

The way in which the application was developed was discussed in the form of the software development methodology. While being partially a research based project, the system was designed once research had been completed within a cycle based timescale. This most reflected the Spiral methodology which focuses on iterative design and development with a working prototype at the end of each cycle. By structuring the project around this approach it meant that the project was split into smaller sections and became more manageable. The importance of source control technology was discussed within the context of the project and its usefulness evaluated.

This section built upon the project aims in Chapter 1 in order to turn the idea of the system and its aims, into a design that could later be executed within the implementation stage. By defining a clear design strategy for the system it meant that less time was wasted later on when developing the code, when coupled with the cycles of the Spiral methodology it meant that there were smaller feedback loops when it came to making improvements to the software. As it becomes more expensive, in terms of time, to make changes later on in the life time of the project [68] this method proved to be effective in completing the project to the time scale given.

Chapter 5

System Implementation: CertManager

This section of the report details the steps that were taken to implement the system design into a working application. It is split into sections based on the major milestone events that were needed to be completed in order for the system to function correctly.

CertManager is a typical front-end iOS application that uses the UIKit framework in order to present different views to the user. Based on the previous research and designs, the vast majority of the application's functionality was to present data about the system to the user. The implementation class diagram is described using UML and can be viewed in the Appendices

Appendix A - CertManager class diagram. The final user interface of the application can be viewed in Appendix B - CertManager user interface.

5.1 Accessing Certificate Store

The first stage of building the CertManager application was to implement the research knowledge gained previously on extracting the certificate information from the system trust store. The user interface had previously been defined from 4.4 User Interface Design and so a template project was created with a *UITableViewController* as it's initial view. This view needed a data store delegate defined in order to obtain the information required for it to draw the table cells and so the next stage of development looked at obtaining this data.

Based on the research conducted in 3.1 Trust Store, it was shown that the certificate data was stored in two files, *certsIndex.data and certsTable.data*, within the Security framework folder on the file system. In order to extract this data, the Apple Open Source directory for the Security framework built for OS X was searched. Code that composed the Security Server daemon was found and a file called *OTATrustUtilities.c* contained references to the two files on the disk.

OTATrustUtilities [69] is a part of the security server (securityd), a daemon that runs on both OS X and iOS, that provides implementations of encryption, decryption and authorization management. There is no public API for securityd, instead it listens for messages from other services and API's that come included in the frameworks provided by Apple. These include Keychain, Certificate, Key and Trust services. On OS X, there is an additional Authorisation Service which is what is used to prompt the user for administrative passwords [70].

The OTATrustUtilities.c file contained the source code that could read the index data from the disk but it was not able to be called by the application code because it is part of the securityd code and therefore abstracted behind the Security framework, which as discovered earlier does not allow access to information about the default certificates installed on the device. In order to bypass the Security framework and call this code directly, the source code was imported into the project as an extra utility file. Initially this did not work because it had missing imports and dependencies that were also part of the securityd project and so these were also imported and a custom implementation of securityd was built so that it would compile and was imported as a framework that could be used by the application.

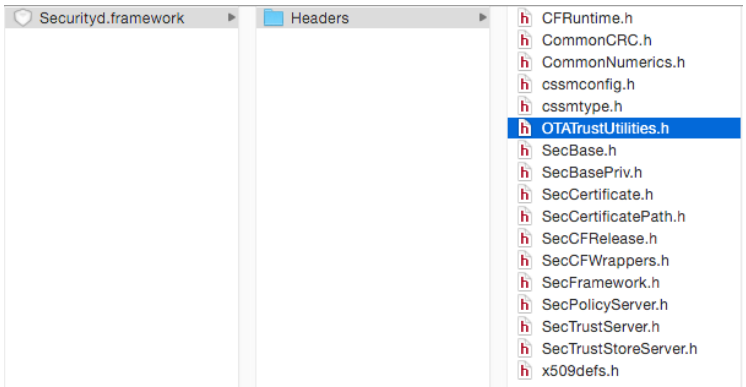


Figure 26 - The custom Securityd framework.

Using this newly imported class, the certificate index values were able to be extracted from the *certsIndex.data* file to provide a *CFArrayRef* object containing the offset values. In order to obtain this two functions were used, firstly *SecOTAPKICopyCurrentOTAPKIRef* gave a

reference object to be used later on which behind the scenes calls the function *SecOTACreate* that is responsible for reading the index file from the file system and loading it into a look-up table. Secondly, by passing the reference back to *SecOTAPKICopyAnchorLookupTable*, the look up table is returned as a dictionary structure.

Another class that was part of the securityd application was the *SecTrustServer.* This class contained the functions required to extract the certificate data from the *certsTable.data* file after giving it the anchor lookup table data structure. The function *CopyCertsFromIndices* provided this functionality and it returned an array containing the *SecCertificateRef* representations of the certificates. After these were obtained they could be added to a local array list in order to use within the application. In the early testing phase of CertManager, the application took this array list, iterated over it and extracted the certificate issuer to then print to the console.

```
2014-11-13 17:43:37.185 CertManager[3457:130988] certs: (
    "<cert(0x15fe164d0) s: Izenpe.com i: Izenpe.com>",
    "<cert(0x15fe1a840) s: Izenpe.com i: Izenpe.com>",
    "<cert(0x15fe1aa60) s: ApplicationCA i: ApplicationCA>",
    "<cert(0x15fe1ac80) s: Certum Trusted Network CA i: Certum Trusted Network CA>",
    "<cert(0x15fe1b3b0) s: CNNIC ROOT i: CNNIC ROOT>",
```

Figure 27 - Console output of the CopyCertsFromIndices function.

After this information had been obtained it was then a case of displaying this to the user in the form of a table instead of outputting the data to the console. In order to achieve this, a custom table cell was created to show the name of the certificate.

Figure 28 - An early version of CertManager that listed the root certificates.

In order to provide the user with a way of blocking the certificate, a toggle button was added to the cell and a call-back function linked to the change event. When the function is triggered it will either add or remove the SHA1 value of the certificate from a property list containing the list of blocked certificates. There was an issue in the original implementation of this code because the toggle button that was added to the cell had no way of knowing which certificate it corresponded to. In order to work around this issue, a new class, *TableCellSwitch*, was created that was a sub-class of the UISwitch. The only addition this class made to it's super-class was a new property that was able to hold a reference to an *NSIndexPath.* This index path is the way in which the UITableViewCell knows which section and row it is in within the context of the table. By providing this value to the switch object when it was created, it meant that when the function was triggered when it changed, a reference to the switch that called it could be obtained and then the index path property from within it could be accessed.

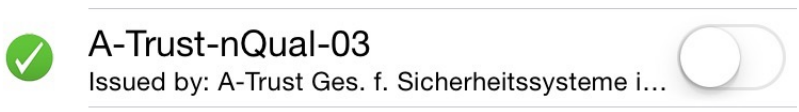


Figure 29 - The final implementation for the custom table cell.

5.2 Additional Functionality

Once the basic functionality of the application had been completed, it was decided that extra features could be added into the project. In order to do this the application view structure needed to be changed as the application was built around a single controller and needed support for showing multiple views. From the design phase, a tabular system needed to be implemented, which meant adding an extra layer to the view hierarchy so that all views were sub-views of a single tab bar controller.

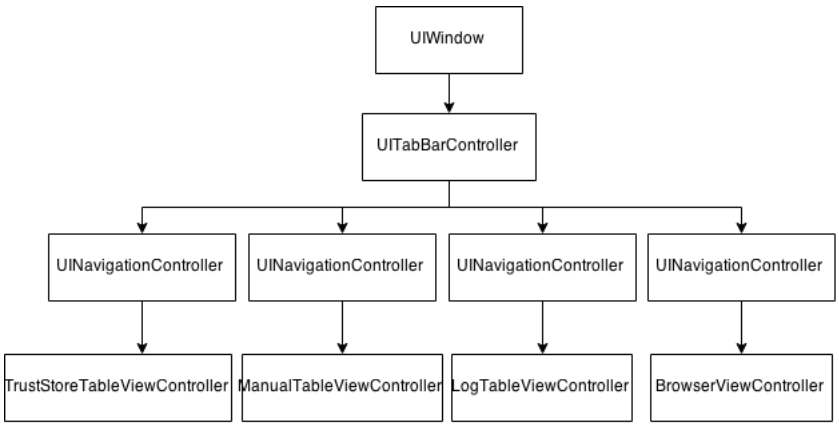


Figure 30 - The view hierarchy for CertManager.

This *UITabBarController* was then set as the root view controller for the UIWindow object, a representation of the screen in an Objective-C object, by using the *setRootViewController* method. The navigation controllers were not used for their ability to switch between views because this application did not need that functionality, the only reason that they were used was for the ability to show a navigation bar to the user and to provide additional buttons for the user to interact with.

5.2.1 Manual Blocking

Now that the ability to add additional views to the application had been completed, it meant that additional functionality could be implemented. One of the features that had been designed was for the ability for users to add their own SHA1 values in order to block certificates that were not found in the trust store. These values would be stored separately from the block list for the root certificates so the methods that were used to write NSDictionary and NSArray objects to the disk were made generic with the ability to pass different file names and objects.

In order to provide a way in which the user can add new certificates, a *UIAlertController* was used in order to show an alert view to the user which contained two text fields, asking the user for a title of the certificate and the SHA1 representation for it. As this data was going to be used within CertHook, it was important that the SHA1 value was checked to make sure it was valid as it could potentially cause issues elsewhere. In order to do this an action was attached to the event that is triggered whenever the value of the text field changes.

$$[textField\ addTarget{:}self\ action{:}@selector(textChanged{:})\ forControlEvents{:}UIControlEventEditingChanged];$$

Code 3 - Detecting a change to the SHA1 field.

The *textChanged* function then takes the value from within the text field and checks that it is a valid SHA1 value using a regular expression:

$$\text{^}[a-fA-F0-9]\{40\}\$$

Code 4 - The regex for a SHA1 string representation.

Which make sure that the string only contains values that are either upper or lowercase A to F, or numerical values 0 to 9. The additional check is that the length of the string is 40 characters long as this is the expected length of a SHA1 hash value.

The main difference between this view and the existing root certificate table view is that the cells did not have the toggle on them. This is because if the certificate is shown in the manual list then it is automatically blocked, therefore having a toggle does not make sense as it would not be shown anyway. In order to allow a user to still have the ability to remove the certificate from the list, a swipe mechanism was added to each cell which provided the user with a button to delete.

5.2.2 Logging

In order to encapsulate the data used for the logging system a model object, *LogInformation*, was created by creating a new class that extended *NSObject*. This class contained properties that represented the data needed for a log to be added into the system.

```
@property (strong, atomic) NSString* application;
@property (strong, atomic) NSString* certificateName;
@property (strong, atomic) NSString* time;
```

Code 5 - The properties of the LogInformation class.

In order to save the logs to the file system a helper method was created which provided a way in which a Log can be represented as a single string value. Additionally, a second initialisation method was created in order to convert a string representation back into a valid *LogInformation* object. This class could then be used by the *LogTableViewController* to gain information regarding a log and display it back to the user in the form of a table.

5.2.3 CertBrowser

CertBrowser, originally designed to be its own application, was an addition to the CertManager application in order to provide extended functionality to users. CertBrowser is a standard web browser based on the *UIWebView* class with the added ability to view the certificates that had been used for a connection to a HTTPS website. In order to access certificate information, the *BrowserViewController* class that controlled the web view was as a *NSURLConnectionDelegate* [71]. This delegate is a protocol which allows custom authentication handling and allowed the browser to access the certificate information from a request via the *willSendRequestForAuthenticationChallenge* method, which takes an NSURLAuthenticationChallenge object as a parameter. Using this challenge object, a reference to the SSL security trust reference could be obtained.

$$SecTrustRef\ trustRef\ =\ challenge.protectionSpace.serverTrust;$$

Code 6 - Obtaining trust information from an authentication challenge object.

This trust reference was then saved for later use when a user clicks the green lock button found next to the URL entry text field. When this was pressed, they are taken to another view and this *SecTrustRef* object was passed along as well.

Figure 31 - The CertBrowser URL entry bar with green padlock icon.

The view that is then shown is the *CertificateTableViewController* and in here the *SecTrustRef* had the certificates contained within it extracted by using functions found within the Security framework, similar to the logic used to obtain certificate information within the CertHook application.

```
NSMutableArray *certs = [[NSMutableArray alloc] init];
CFIndex count = SecTrustGetCertificateCount(trustRef);

//For each certificate in the certificate chain.
for (CFIndex i = count - 1; i >= 0; i--)
{
   //Get a reference to the certificate.
   SecCertificateRef certRef = SecTrustGetCertificateAtIndex(trustRef, i);
   [certs addObject:(__bridge id) certRef];
}
```

Code 7 - Extracting certificate data from the SecTrustRef object.

By starting at the end of the certificate chain it meant that the certificates were added into the NSMutableArray in the correct order, with the root certificate being first and then subsequent certificates coming after. The table that was used for this view used the *certs* array as it's data source and custom cells were created in order to display the information held within the object to users and in order to access this information the *X509Wrapper* utility was utilised. In order to show the difference between a root certificate authority and a regular certificate, different icons were used based on the images used in the Mac OS X Keychain Access application.

Figure 32 - The different visual representations for viewing certificate data.

By selecting any of the icons, the user was presented with an option to add the certificate to their manual block list. This meant that users were able to block certificates they didn't have the SHA1 value for and could block through any stage of the certificate chain. From an attack perspective, this isn't really prevention but more as a reaction. If a user noticed abnormal activity in their browsing habits then it provides a way in which they could check the certificates in use and block them if they seemed suspicious, something that is not currently available to do on iOS.

5.3 Writing to the Disk

In order for the system to work, CertManager needed to be able to store the trust preferences of the user onto the file system so that it could be read later by the CertHook application. There are various ways in which data can be written to the file system including Core Data [72], an object relational mapping framework which abstracts data storage for objects away from their technical implementation and NSCoder [73], a way in which objects can be converted between different formats, including binary that can be stored on disk.

For this application, those implementations seemed excessive as the data that needed storing were simple NSString values and those frameworks are built more for use with complex data structures. Throughout the application the primary source of information is stored within NSArray and NSDictionary objects. These data structures include high level functions in which the contents of the object can be written to a file on the disk and also the ability to recreate the object using the contents of a file. The data is converted to a property list representation and so the files that the data is stored to is the *.plist* file type, which has an XML-like structure. By using these built in functions, data could be saved and loaded without needing to worry about the underlying file system logic.

While this worked well for storing the SHA1 string values for the certificate representations, when the logging system was introduced it became apparent that a new method of writing to the disk would be needed. The main issue was that with the previous method of storing data the entire data structure had to be loaded, updated, and then written back again afterwards to replace the existing file. With the logging system it was not in the design of the system for the entirety of the logs to be loaded whenever a new log was going to be created. Therefore a way in which data could be appended to the existing file was needed.

```
NSFileHandle *fileHandle = [NSFileHandle fileHandleForWritingAtPath:fullPath];
[fileHandle seekToEndOfFile];
[fileHandle writeData:[infoString dataUsingEncoding:NSUTF8StringEncoding]];
[fileHandle closeFile];
```

Figure 33 - Code to append a log to the existing file.

In this implementation, an NSFileHandler object is created with the path to the log file given. The function *seekToEndOfFile* is called so that the pointer of the writing function is now pointing at the end of the file. Write data is called using the information from the LogInformation object and then the file is closed again. This means that the original file is still there but new information has been added to the end of it.

When it came to reading back the objects from the file system to show within the Logging page on CertManager, each line of the file needed to be read into the application and converted into a *LogInformation* model object. An external library was used in order to read the file a line at a time as this functionality is not currently possible by default Objective-C classes. After each line was read, it was split into separate strings based on the location of the comma character and these strings were used to re-create the *LogInformation* object.

5.4 Wrappers and Utilities

A utility class is a class where all functions and fields are as static, and as such the class is as being stateless. A utility can not be instantiated by another object as it has no constructor and so it is entirely dependent on data being passed to convenience functions.

While the project was under development there were several times where the same methods were being called repeatedly. These included operations such as reading and writing data structures to the disk and reading data from certificate objects. Some of these methods were also part of other libraries and frameworks that were written in C code and not the Objective-C code that most of the application was using. While C works within Objective-C without any issues, for consistency across code it was decided to move these functions out into utility classes.

5.4.1 FileHandler

As the application was always writing and storing data to the same location, it made sense to put these functions into a utility that provided a simple API to load and save information to the disk. The benefit of having all file system code within one class also meant that once new certificates had been saved to the disk, a notification could be sent out to alert a change to CertHook. This could have been done individually but by making it a part of the write function and having all other code calling that function, it meant that the essential notification code would always be called.

Additionally, the application was using custom model objects in order to store log information as its own data structure. This meant that it wasn't as simple to output into a file, where the *writeToFile* method could be used for NSArray and NSDictionary, and so custom read and write code was needed to be able to output the object to a file in a human readable form and then read back in and constructed again as a LogInformation object.

5.4.2 CertDataStore

The front-end application is backed by the use of View Controllers, when an action such as toggling a certificate trust is performed, this calls a method with the view controller. With a UITableView, delegates are required in order to provide the view with the information that it needs to show data to the user. Originally, the view controller was the owner of an NSArray that was used to hold the certificates that had been read from the disk. This worked nicely until other view controllers needed to access the same data and then code started to be repeated. In order to abstract the view controller away from the file system a new type of object was created to act as a data store. It provided an interface for the view controller to access the data it needed to build the front end but without exposing the original NSArray or the file system. This meant that the table view could create a new CertDataStore object and then make reference to that instead of needing to import the FileHandler code and manage its own data store.

5.4.3 X509Wrapper

In order to fully interact with X.509 certificates on iOS, an external library was needed because the built in Security framework does not allow access to certain fields. In order to do this the OpenSSL library was compiled for iOS and included within the project. The project is written in C and so a lot of the functions and variables were hidden behind obscure names and to access data it was split across several functions that had to be called. Additionally the data had to be accessed several times in different places within the application code so the decision

was made to move the logic into its own utility class. This meant that common functions could be for accessing data within a certificate in an Objective-C way.

```
+(NSString *) issuerForCertificate:(SecCertificateRef)cert;
+(NSString *) sha1ForCertificate:(SecCertificateRef)cert;
+(NSDate *)   expiryDateForCertificate:(SecCertificateRef) cert;
```

Figure 34 - The methods by the X509Wrapper class.

5.5 External Resources

In any software development project there are times where an additional feature will be needed that is outside the scope of the project or expertise of the developer. The rise in open source software and the sharing of code between developers is a great way to reduce the time spent on trivial tasks so more time can be focused on the development of the system functionality. This project was no different and throughout there were times when solutions to problems were available from online resources such as *GitHub* [74] and *Stack Overflow* [75] where code and projects had been shared in order to help others. As mentioned previously, this project will also become open sourced once complete so that others can use it to learn themselves.

5.5.1 Apple Security Framework

As part of Apples attempt to integrate with the open source community, and the fact that several key components of its operating system were open source previously, they have made a vast amount of the source code for OS X available for the public. In order to help during research into the Security framework for iOS, a lot of time was spent reading through the source code for the OS X version that had been open sourced. As both iOS and OS X are based around the same core Darwin foundation, it was assumed that key components such as Security would most likely be shared across platforms. After the *certsIndex.data* and *certsTable.data* files had been found within the iOS file system, the Security framework and *securityd* daemon were downloaded in order to search the code for any references to these files. This lead the research towards two files used for extracting data from the files: *OTATrustUtilities.c* [44] and *SecTrustServer.c* [76].

These files are licenced under the Apple Public Source Licence [77] and therefore it was important that the project conformed to its licence agreement.

> 2.2 Modified Code. You may modify Covered Code and use, reproduce, display, perform, internally distribute within Your organization, and Externally Deploy Your Modifications and Covered Code, for commercial or non-commercial purposes

Figure 35 - A part of the license allowing modifications to code.

The licence is very flexible and essentially removes all liabilities Apple may have and means that the source code for any project using the code must also be made available, as this project will be doing. A full copy of this licence can be found within the project source code as part of the licence requirements.

5.5.2 Categories

One feature of Objective-C is a design pattern called Categories. A category is a way of adding additional functionality to existing classes that you may not have the original source code to. For example this is the contents of *NSString+Logger.m*

```
@implementation NSString (Logger)
- (void) customLogger {
        NSLog(@"Custom Log %@", self);
}
@end
```

Code 8 - An example of an Objective-C category.

Any class is able to import this file and it means that every NSString now has a new function that can be called to perform a custom action. Note that NSString is a class written by Apple and part of the Foundation framework yet functions were able to be added to it without modifying the original source code. This method is preferred when adding new functionality compared to sub-classing, which is usually used to replace existing functionality.

This project used Categories that provided useful means of transforming data between different representations. For example, at one point there is an NSData object that contains a SHA1 value, in order to represent this as a string to show the user, a category was added to the NSData class in the form of the *hexStringValue* function [78] that was provided to a request made online on how to read data from a certificate object.

5.5.3 DDFileReader

When it came to building the logging system to read logs from the file system it was expected that it could be achieved in a similar way to reading in a potentially large file in Java, by using a buffered reader. As it turned out, Objective-C does not have a way in which it can read files line by line, which means that the entire file has to be loaded into memory. Log files could reach potentially large sizes and therefore it was chosen to look for an alternative method to load them into the application.

An answer to a similar question was found on the *StackOverflow* website, with a developer providing a helper class that provided the solution [79]. After the class was added into the application, it showed errors as it was written for non-ARC compliant code where memory management was handled manually. On the same website another developer [80] had rewrote the entire class to work with ARC, and so this version was used instead.

5.5.4 Font Awesome

Font Awesome [81] is an open source font that provides scalable vector icons, primarily used in web development to show aesthetically pleasing icons to users across all browsers. The font is distributed under the SIL Open Font Licence [82] that makes it free for both personal and commercial use. As CertManager needed to show various icons to the user for the web browser functionality, it was decided that this font set would be used instead of using images so that the icons could respond nicely to presses and could be coloured based on variables within the application. Using images for doing this would have meant designing different colours for all three types of resolutions. Through the use of fonts to display icons it meant that they would scale across devices too.

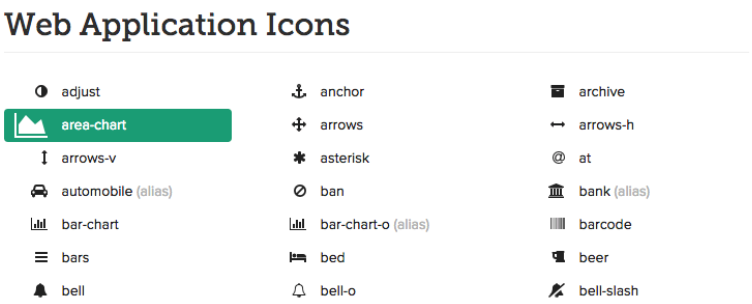


Figure 36 - An example of some of the icons available from the library.

As mentioned, the fonts were designed around being used for the web but a project was encountered which provided a Category for NSString which provided a wrapper for the Font Awesome project to work inside Objective-C code [83]. Once again, the open source community was able to provide a simple way of implementing a simple feature that would have taken up valuable development time if it had to be written manually.

5.6 Summary

This chapter has outlined the different stages of the development process that were undertaken in order to create the CertManager application. As the application is split into different tabs, the functionality of each tab has been described and the way in which it achieved its aim has been described from a technical perspective.

In order to access the certificates from the system trust store, code from the *securityd* project was taken from the Apple open source directory. This was adapted in order to work on the iOS platform as apposed to OS X and inserted into the project. Once this had been implemented, functions that were able to read the content of the files from the disk were called by the wrapper classes that were created and this data displayed to the user.

Additional functionality was added to the application in the form of users being able to add manual values of certificates in order to block them from being used. A logging system was also implemented using custom Log object representations that were read from a log file on the disk. Finally a web browser was created which provides the functionality for users to access the certificates that have been used within a HTTPS request and optionally add them to their blocked list.

In order to bring together common code and functionality, utility classes were created which abstracted away repetitive technical code into a simple to use API that could be called from anywhere in the application. Functionality within utilities included reading and writing from the file system, accessing the application representation of the certificate store and performing trust actions on them.

Finally, external resources that were used in the creation of the application were , showing how they were used and the sources of the information. These resources were used mainly as helper functions which provided fast ways of implementing logic that the application needed but was not within the scope of the project and so was not worth spending the time implementing custom solutions.

Chapter 6

System Implementation: CertHook

CertHook is the Substrate system modification that is used to implement the logic that blocks connections based on the certificates in an SSL request. Based on previous system design and research, it was decided that the block should be implemented in the SecureTransport layer of the system Security framework. This provided a low enough level to provide the block system wide without resorting to modification of files on the file system. This chapter looks at how CertHook was developed and the techniques used to implement the block.

6.1 Creating the Project

The first stage of CertHook was creating an environment in which the modification could be developed. In order to do this, a new sub-project was added to the main CertManager project and this included a new MakeFile to be used by the Theos build system along with a simple class that imported the Substrate header file.

```
/**
 * CertHook.xm
*/
#import <Foundation/Foundation.h>
#include <substrate.h>

%ctor {
    %init;
}
```

Code 9 - First basic system modification code

The file extension of *.xm* is used by the Logos Processor that is part of the Theos build tool. These files are converted into standard *.mm* files to be used by the Clang compiler [84]. The *%ctor* macro is provided by Logos and creates a constructor method for the class, inside there the %init macro is called which sets up any other code within the file. The main purpose for making this class was to establish that the build environment was set up correctly. Within the MakeFile for CertManager the following lines were added:

```
SUBPROJECTS += CertHook
include theos/makefiles/aggregate.mk
```

Code 10 - Subproject code added to the CertManager MakeFile.

By adding this code to the main MakeFile it meant that the CertHook was now described as a sub-project. This gave the benefit of being able to run the *make* command in the main project file and have both systems being built at once and bundled into a single .deb file for installation. The aggregate.mk file is provided by Theos and provides the imported information needed to merge the two projects together.

6.2 Hooking Methods

The next stage of building CertHook was to implement the method swizzling functionality that was researched previously. By using the Substrate framework that had already been imported, the code to set up the method was added into the constructor of the class.

```
#import <Security/SecureTransport.h>
#import <substrate.h>

/**
 *  A reference to the original SSLHandshake method.
*/
static OSStatus (* original_SSLHandshake)(SSLContextRef context);

/**
 *  The override SSLHandshake method.
*/
static OSStatus replaced_SSLHandshake(SSLContextRef context) {
    NSLog(@"Inside the replacement SSLHandshake method”);
    return original_SSLHandshake(context);
}

%ctor {
    %init;
    MSHookFunction((void *) SSLHandshake,
      (void *)  hooked_SSLHandshake,
     (void **) &original_SSLHandshake);
}
```

Code 11 - Original hooking method proof of concept.

Code 11 shows the code that was used by CertHook in order to override the original SSLHandshake function within the SecureTransport class. The class is compiled into a dylib which is then injected into the SecureTransport class and the constructor method is called. This then calls the MSHookFunction method provided by the Substrate library, taking three parameters. The first parameter is an address to the location of the original function which will be replaced, in this instance it is the SSLHandshake method from the imported SecureTransport.h file. The second parameter is a pointer to the replacement function that will be called instead of the original function, in this case the replacement function is *replaced_SSLHandshake*. The third parameter is a pointer to another pointer which points to the original function. This is used so that the original implementation can still be called once the original function has been replaced [85].

When this code was compiled it was successfully injected into all processes and by watching the console output of the device it was possible to see whenever an SSL handshake occurred by being able to see the contents of the *NSLog* being printed.

6.3 Loading User Preferences

Now that the tweak had the ability to override the SSLHandshake function, the next stage of implementation was to be able to gain access to the list of blocked certificates that had been generated based on the preferences of the user in the CertManager application. Originally,

this was achieved by having a constant string which gave the path to where the preferences had been stored and then when the replacement SSLHandshake method was called, this file would be read into an NSArray.

```
#define KEYS @"/private/var/mobile/Library/Preferences/CertManagerTrustedRoots.plist"

static OSStatus replaced_SSLHandshake(SSLContextRef context) {
    NSArray *arr = [[NSArray alloc] initWithContentsOfFile:KEYS];
    NSArray *trustedRoots = [[NSArray alloc] initWithArray:arr];
    NSLog(@"Trusted: %@", trustedRoots);
    return original_SSLHandshake(context);
}
```

Code 12 - The original way of reading user preferences.

This was tested on the device and the list of blocked certificates was printed to the console. After evaluating the code, it seemed like a lot of pressure on the file system as the array was being read into memory every time the SSLHandshake function was called, which could potentially be hundreds of times in an application like a web browser. In order to help reduce the memory footprint of the tweak and increase performance, the code to load the property list was moved into the constructor method. This meant that the blocked certificate list was now loaded when the process is first loaded and stored in a static variable accessible to the replacement SSLHandshake method.

Now each SSLHandshake request would read from memory instead of the disk, resulting in faster processing and lower overheads. However this introduced another problem as a user could load an application, then open CertManager and choose an additional certificate to block. When they returned to the original application the constructor code for CertHook will not be called because the process was already loaded but just in the background. This meant that the most up to date list of blocked certificates was not available to CertHook and the user would not be aware of this.

As a workaround a global notification system was implemented into both CertHook and CertManager. When the user changes the trust value for a certificate in the application a notification is posted across the system. In the constructor for CertHook, an observer was added to an *CFNotificationCenterRef* which meant that whenever the message with the corresponding identifier was received, a method would be called.

```
CFNotificationCenterRef reload = CFNotificationCenterGetDarwinNotifyCenter();
CFNotificationCenterAddObserver(reload, NULL, &updateRootsNotification,
CFSTR("uk.ac.surrey.rb00166.CertManager/reload"), NULL, 0);
```

Code 13 - Listening for a message from CertManager.

When this occurred, the *updateRootsNotification* method was called which reloaded the user preferences from the disk into memory again. This meant that CertHook always had the most up to date version of the list of blocked certificates without needing to manually check the file system each time.

6.4 Blocking Connections

At the start of the project a function named *SecTrustSetAnchorCertificates* was mentioned which sets the list of current anchor certificates that are accepted for a given connection. It was believed that this function could be use within the SSLHandshake method to always override the list of trusted certificates to only include the ones that had not been included in the users blacklist. When this is used with the *SecTrustSetAnchorCertificatesOnly* function, this disables the trust of a certificate being evaluated against the root certificates stored on the device. This would have meant that instead of storing a list of blocked certificate values, a whitelist of trusted certificates would need to be kept instead. After evaluating the solution it was deemed inappropriate to use these functions as it could potentially create additional security issues. Some applications use certificate pinning [86] to make sure that connections to their applications only use certificates signed by a certain root certificate. This function is often called with a single certificate passed to it and if CertHook then overrode that with a full list of default system certificates it could impact the security of that application. Additionally, keeping a whitelist would require more memory to store the data of all certificates and is less flexible to use. It also means that users would not be able to manually add certificates outside of the scope of the root certificate list.

In order to block connections using invalid certificates it was decided that an algorithm would be written in order to implement it manually. From testing previously it was shown that the SSLHandshake method can gain access to a list of the certificates being used in the connection, from the base certificate up to its root. The general overview of the algorithm is to take that list of certificates and iterate over each one. The SHA1 hash for the certificate would be calculated and compared to the list of untrusted certificates by the user. If the certificate is untrusted then the value that is returned indicates a failed connection, otherwise the original SSLHandshake method is called so that additional checks can be performed by the system implementation.

Once is algorithm had been implemented, it was tested on the device. This resulted in Safari infinitely trying to make a request to the server. There were subsequent handshake attempts after the initial attempt and those don't fail as there were no certificates in those requests. The handshake was being declined because it contained untrusted certificates but then Safari was performing retry logic to see if the error was temporary, these empty requests didn't contain any certificates and therefore the original handshake method was being called and returning a successful connection. This caused Safari to try the handshake with the certificates again, which were blocked, and this caused an infinite loop.

As a result, logic was added so that if an untrusted certificate is discovered, a flag is set. If there are subsequent requests to the same domain name or IP address but without any certificates then they are declined too. When the user then visits another website using SSL the new request will contain certificates, and therefore skip this logic.

```
if(BLOCKED_PEER) {
        BLOCKED_PEER = NO;
        //We just blocked this peer, fail again.
        return errSSLClosedAbort;
}
```

Code 14 - The code for stopping additional requests.

Returning the value of errSSLClosedAbort indicates that the connection was closed due to an error. This stops the system from trying to repeat the connection again. There are various different error messages that can be returned from SSLHandshake which indicate the trust of the certificate chain. The message that was used for untrusted root certificates was *errSSLUnknownRootCert*, this indicates to the caller that the certificate chain is valid but the root certificate is not trusted [60].

At this stage, the CertHook system was able to block connections based on the root certificate used for the connection. An additional feature was added to the system in which the user was able to add their own certificates to block that were not limited to the root certificate store. This meant that if an intermediate certificate issued by a root certificate was compromised and this certificate was discovered, then the user would have the option to block the certificate manually on the device instead of waiting for Apple to patch the system and include the certificate in the *Blocked.plist* files mentioned previously. In order to support this additional change, another property list had to be read from the disk when CertHook was initialised. This new list contained manually blocked certificates and so for each certificate that was being read by the tweak an additional check was added to search this new array and set the invalid return value if it existed.

The source code of the algorithm has been included within this report and can be viewed in Appendix C - Blocking algorithm source code. Additionally, the algorithm has been displayed as a flow diagram in order to display the different stages taken when overriding the SSLHandshake method and abstract the functionality from the code.

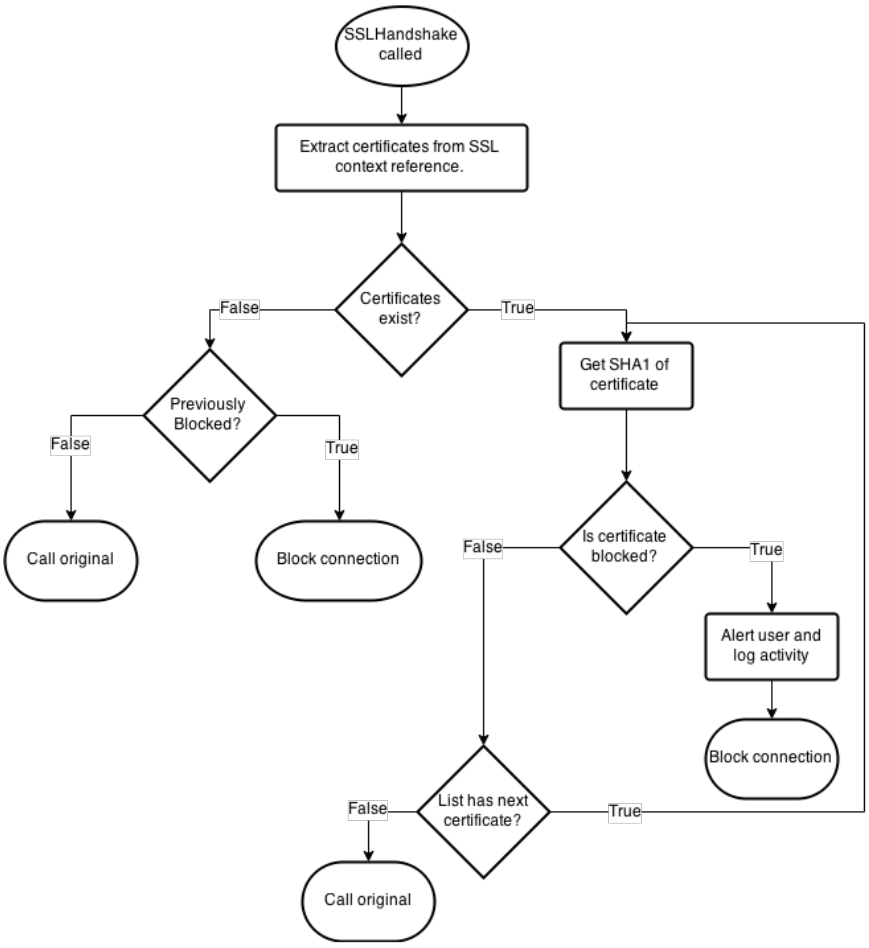


Figure 37 - The CertHook SSLHandshake algorithm flow chart.

6.5 Notifying the User

At this stage, CertHook was able to block the connection from occurring and this was viewable by information provided by the console logs as a developer. However from a user perspective, it was unsure if there had been a genuine network fault or a block occurring on the connection. Without a explicit message to the user that a certificate had been blocked, there is no way in which that user would be able to tell the difference.

Originally it was believed that the code to generate notifications would be able to come from the same code that blocks the system calls within the SecureTransport class. However, it was discovered that in order to show messages to the user they needed to be sent from within the SpringBoard process as that is the only process that has access to showing UI overlays.

Due to the fact that each process that is started has the CertHook code injected into it, there needed to be a way in which these sandboxed application could send a message to the Springboard process telling it to show a notification to the user, while also providing relevant information to it about the certificate that was blocked and the process that caused it so that this could be displayed to the user.

There are various ways in which inter-process communication can be achieved on iOS however thy are usually reserved for system applications and not commonly seen or approved for applications on the App Store. The first method that was looked at was CFNotificationCenter. This class is part of the Core Foundation library and is used to post notifications using the Darwin based notification system. This centre provides system wide notifications that any application can listen to without being impacted by the Sandbox. The problem was that the version on iOS is limited, as it cannot take parameters, so no data can be passed between processes using this method. As a result, alternative methods of inter-process communication needed to be researched.

Another class was found called CPDistributedMessagingCenter, that is able to broadcast additional data between processes. The messaging centre is a part of the private AppSupport framework provided by Apple, and works using a client and server interface. Clients can register against the messaging centre using an identifier in order to listen for messages, while the Server creates messages containing the message identifier and an optional dictionary of values. It uses the Bootstrap system to link the messaging services together [87] and is usually sandboxed, however this was bypassed to make the messaging work (see 6.6 Sandboxing). The CPDistributedMessagingCenter works using a server/client structure. When the application is loaded it injects itself into SpringBoard and waits for the “init” method to be called. When this happens, it creates a new CPDistributedMessagingCenter server with a unique identifier and then starts the server on the thread that SpringBoard is running on.

As the messaging centre is part of a private framework it is not publically documented, therefore in order to know what the functions were available to use a header dumper tool *class-dump-z* [88] was used in order to extract the header file from the compiled Objective-C code. Once a copy of the header file had been obtained, the functions that looked likely to create a server and client were discovered:

$$CPDistributedMessagingCenter\ centerNamed:$$

Code 15 - Method definition to get a reference to the messaging centre.

$$CPDistributedMessagingCenter\ registerForMessageName:target:selector:$$

Code 16 - Method definition to register a listener to the messaging centre.

$$CPDistributedMessagingCenter\ sendMessageName{:}\ userInfo{:}$$

Code 17 - Method definition to send a message using the messaging centre.

Using these functions, an additional file was created within CertHook named *SpringBoardHook.xm* and this contained an additional Substrate hooking function that injected itself into the SpringBoard process. Within the constructor, it runs a listening server for the messaging centre using the *runServerOnCurrentThread* method and then registers for the messaging centre using the *registerForMessageName* method.

Meanwhile in the SecureTransport hook, a function was so that when a certificate is blocked, a reference to the *CPDistributedMessagingCenter* with the same unique identifier is created and then a message is posted to it alerting any listeners that a certificate has been blocked. Along with this message is a dictionary object that contains the name of the certificate and the process that made the connection attempt. This data is then picked up by the servers listening for the notification, which in this case is running on the SpringBoard process.

At this stage the Springboard process was now receiving system notifications whenever a process blocked a certificate, bringing the whole system together in one location. A way to display this information to users was needed and so research was conducted into the way in which notifications are shown to the users in the form of a banner. In a regular application, a local notification can be shown to the user by calling a function in the UIApplication class.

$$[[UIApplication\ sharedApplication]\ scheduleLocalNotification{:}notification]$$

Code 18 - The code to show a local notification from within a UIApplication.

Recalling the fact that CertHook has no filter to it, it means that it can be injected into processes that are not always applications. It is possible that a background daemon could notify the SpringBoard that a certificate had been blocked, and because daemons are not subclasses of UIApplication they will not be able to schedule a local notification as calling the *[UIApplication sharedApplication]* would return a nil value.

As a result, it was decided to look into the implementation of the *scheduleLocalNotification* method to see how notifications were being shown to the users. This lead to the discovery of a private framework called *BulletinBoard*, which controls local and push notifications. The headers for this framework had been dumped by someone else using the *class-dump* tool and posted online to GitHub [89]. From analysing those header files and looking in other frameworks for references to those classes a controller class within the SpringBoard framework, *SBBulletinBannerController*, was found that provided means to show notifications to the user.

$$-(void)observer{:}(id)observer\ addBulletin{:}(id)bulletin\ forFeed{:}(unsigned)feed;$$

Code 19 - The method definition used to show a local notification.

This *addBulletin* function expected an object of type *BBBulletinRequest* which is a reference back to the BulletinBoard framework.

```
//Create a bulletin request.
BBBulletinRequest *bulletin = [[BBBulletinRequest alloc] init];
bulletin.sectionID = @"uk.ac.surrey.rb00166.CertManager";
bulletin.title =  @"Certificate Blocked";
bulletin.message = [NSString stringWithFormat:@"%@ attempted to make a connection
using certificate: %@", process, summary];
bulletin.date = [NSDate date];
SBBulletinBannerController *ctrl = [%c(SBBulletinBannerController) sharedInstance];
 [ctrl observer:nil addBulletin:bulletin forFeed:2];
```

Code 20 - The code used to show a local notification.

The BBBulletinRequest header files were analysed [90] in order to see the values that could be set in order to show information to the user. Once this had been created and posted to the SBBulletinBannerController, local notifications were displayed to the user without needing any reference to the UIApplication context. A reference to the CertManager application was added using the *sectionID* property which meant that users could tap on the notification when it appeared to automatically open the CertManager application.

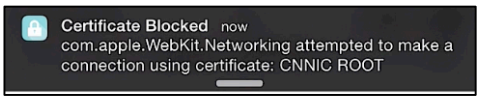


Figure 38 - A local notification shown when a certificate is blocked by CertHook

6.6 Sandboxing

As part of the security built into the iOS system, applications and the data they own are sandboxed. This means that they are unable to access any information that is outside of their own storage, or interfere with any processes not owned by themselves in order to protect the system from potentially malicious applications [91]. The way in which Substrate works means that tweaks are compiled into *.dylib* files and then injected into the application process at runtime. If the process is sandboxed, then the tweak is too, as it adds no way of escaping that.

In the initialisation code for CertHook, it was required to read a property list file from the disk in order to find out which certificates the user had chosen to block. When this was initially attempted the following error occurred:

kernel[0] <Notice>: Sandbox: deny file-read-data /private/var/mobile/Documents/blacklist.plist

This is because the application is not allowed to access files located in the Documents folder, as by the sandboxing kernel extension [92]. In order to work around this, the file was moved to a location on the file system that was outside of the sandbox for all applications. These locations are fairly limited and consist of the following [93]:

- The application container itself
- /private/var/mobile (not including the Media and Library folders)
- /private/var/mobile/Media/DCIM
- /private/var/mobile/Media/Photos
- /private/var/mobile/Library/AddressBook
- /private/var/mobile/Library/Keyboard
- /private/var/mobile/Library/Preferences

These locations are accessible but submitting an application that used one of these locations would most likely result in the application not being accepted by Apple to be in the App Store. Fortunately this system will be able to use these locations as it will be distributed using the Cydia store, which allow system modifications. It was decided that the file would be stored within *~/Library/Preferences* due to the fact that it is holding preferences for the application and so seemed a logical place to store it. After this was completed, CertHook was able to read the list of blocked certificates from any process that it was injected in.

In addition to having shared access to the file system, CertHook also needed to be able to talk between different instances of itself in order to display notifications to the user. As mentioned previously, sandboxed applications are unable to talk to other processes since iOS 7. The original use of CPDistributedMessagingCenter in order to send messages between processes was unable to do this because it was blocked by the system sandbox. CPDistributedMessagingCenter is built on top of a UNIX based messaging system called *mach ports* which is managed by a central system called *bootstrap*. Bootstrap allows ports to be registered by providing it with a unique identifier however processes are restricted by the system itself as to which unique identifiers they are allowed to call. In order to bypass this restriction a library called RocketBootstrap [94] was used, which adds a secondary central system that doesn't restrict the processes that can be called.

```
CPDistributedMessagingCenter *center = [%c(CPDistributedMessagingCenter)
centerNamed:MESSAGING_CENTER];
rocketbootstrap_distributedmessagingcenter_apply(center);
```

Code 21 - Applying RocketBootstrap to the messaging centre.

After the default messaging centre had the rocketbootstrap function applied to it, it was able to send messages between processes without any restrictions applied by the system. This meant that the sandbox no longer applied and notifications could be displayed to the user whenever a connection was blocked.

6.7 Summary

This chapter has explained in detail how the CertHook system modification was created. Based on previous research, the Substrate library was used in order to inject a custom SSLHandshake implementation into the Security framework. More specifically, the MSHookFunction was used in order to replace the original functionality with the blocking algorithm and then calling the original function.

The blocking algorithm was implemented by taking the list of certificates being used for the SSL handshake and iterating over them. The SHA1 checksum was then calculated for each certificate and this was compared to the list of blocked certificates defined by the user. If any of the certificates were blocked then the connection was aborted.

In order to alert the user to the fact that a certificate has been blocked, a notification system was implemented to display a message to them. This notification needed to be created from within the SpringBoard process and so an additional hooking function was created in order to start a notification listening server within SpringBoard that waited for other instances of CertHook to post messages to it containing information about the certificate that was blocked.

Chapter 7

Development Environment

This chapter provides an explanation into the development environment that was used in order to create the CertManager system. The requirements to build the system are defined and given explanation as to why they are needed. The Theos build system is formally introduced and configuration code explained step-by-step. Additionally, the steps needed to debug the system remotely are documented and the third party applications that were used throughout the project are listed.

7.1 Requirements

The project was developed on an Apple MacBook running OS X, which is a requirement in order to build the project. The restrictions created by Apple meant that a developer certificate was needed in order to code-sign the application so that it would run on a non-jailbroken iOS device. Once the device was jailbroken this restriction was lifted through the use of the AppSync [95] package to patch the installation daemon on the device and remove the need for code signing.

Originally, the application was built using the Xcode IDE and was signed with a developer certificate being installed as an application owned by the *mobile* user. While this works, the application should be running as a system application stored and managed using the Debian package manager for integration into the Cydia front-end system. Early on in the development process the project was moved out of Xcode and into a new project environment which uses the Theos build tool to compile the application, package it into a .deb file and then install it on the device. The downside of this is that the debugging utilities and code-completion found within Xcode were lost for when the front-end was being developed. As a result of this, a hybrid set-up was created whereby the Application could be built in Xcode and signed as a normal AppStore application while it could also be built using Theos to run it as a system application. This meant that front-end development could be completed with the use of the Apple development tools as intended and when it was required to test interactivity with CertHook it could be built in Theos.

The additional requirement for running this project is a jailbroken iOS device with mobile substrate and dpkg [96] installed. Every public jailbreak method installs the Cydia front-end package manager, which installs dpkg automatically. In addition to this, OpenSSH is needed on the device in order to remotely install the application on a local network. The *control* file in the application project defines the requirements formally, and any attempt to install the compiled .deb file on the system will result in these additional dependencies being installed automatically.

```
Depends: mobilesubstrate (>= 0.9.5001),
com.rpetrich.rocketbootstrap (>= 1.0.2),
firmware (>= 8.0)
```

Figure 39 - The project dependencies defined in the control file.

7.2 Theos

Theos is a suite of tools for building iOS software without the need of Xcode. The vast majority of jailbreak tweak developers use Theos in order to build their projects and deploy them to devices [97]. Supporting Theos are three main components:

- Project template system (NIC), providing a way to quickly build project boilerplate code.
- Build system that uses GNU Make and DPKG to package software into .deb files.
- A set of pre-processor directives, Logos, that provides shorthand functions when writing tweaks.

For this project, Theos was mainly used for its build system that allowed for the jailbreak tweak to be built and deployed over the air using SSH. Theos is configured using system variables defined in the *.bash profile* file found in the current user's home directory.

```
export THEOS=/opt/theos
export IPAD_IP=192.168.1.4
export IPHONE_IP=192.168.1.2
export THEOS_DEVICE_IP=$IPHONE_IP
export THEOS_DEVICE_PORT=22
```

Figure 40 - The variables for Theos set in the .bash_profile file.

Once these values had been set the application can be built and installed on the device using the command:

$$make\ package\ install$$

Code 22 - The command to built the system.

This then uses the *Makefile* found in the root of the project directory. Within this file there are various imports and declarations of variables in order to call the Theos code.

```
ARCHS := armv7 arm64
TARGET := iphone:8.1

include ../theos/makefiles/common.mk

TWEAK_NAME = CertHook
CertHook_FRAMEWORKS = Security, UIKit
CertHook_PRIVATE_FRAMEWORKS = BulletinBoard
CertHook_LIBRARIES = substrate rocketbootstrap
SHARED_CFLAGS = -fobjc-arc

include ../theos/makefiles/tweak.mk
```

Figure 41 - The contents of the CertHook MakeFile.

The *ARCHS* variable sets the architecture that the tweak should be built for. When building for OS X and iOS, applications are compiled into Mach-O [98] files that contain information regarding the symbol locations for that particular architecture. One of the features found is that multiple architectures can be combined together into a single binary file which allows it to run across multiple instruction sets. For example on iOS the instruction set is ARM, but there have been various different versions as processors have improved so to build a single application

that runs on the first generation devices using ARMv6 and latest generation that use ARM64, the binary can contain the instruction sets for all the different versions.

The *TARGET* variable defines the version of the iOS SDK that the system should be built against. In the case of this project, the version of iOS that the application was built against was 8.1. This is mainly due to the requirement of the jailbreak, which has been patched on versions greater than this. One issue that was encountered while developing was that Xcode was updated regularly to include the latest SDK's to build against but would not keep the previous ones installed, this meant that when Theos tried to build the project, the SDK it needed was missing. In order to fix this an old version of Xcode was downloaded and the 8.1 SDK extracted from it to build against.

The first include that is seen is for the *common.mk* file within the Theos framework that provides the basic information needed for Theos to know how to compile and build the application. It is essentially loading the common code that will be used later on.

The *TWEAK_NAME* variable defines the name of the tweak and tells Theos what prefix it should use to discover the next variables.

The *CertHook_FRAMEWORKS* variable can be a single value or a set of multiple values that defines which public frameworks the project is using. This is important information that is needed so that Theos is able to pull these frameworks into the build process to link against. Without this section, many of the function calls made within the tweak would fail.

The *CertHook_PRIVATE_FRAMEWORKS* variable is similar to the previous variable except it defines private frameworks that are not included within the public API documentation. Applications that are being submitted to the App Store are not allowed to use these frameworks, as Apple reserves them for their use only. They usually interact with the system level and as such as not suitable for third party developers to have access to. In the case of this project, the private framework *BulletinBoard* was imported because this is the framework that is used by the system to show notifications to the user. As this is not a public API, many of the private frameworks need to be created manually by using the header-dumping tool mentioned previously to generate a set of header files.

The *CertHook_LIBRARIES* variable defines the different third party libraries used within the application. Libraries are conceptually different from frameworks as functions defined in libraries are usually called by the application, whereas a framework usually calls your application. Frameworks usually encapsulate some sort of abstract design that can be implemented by an application, whereas a library already has that defined. In the example of CertHook, libraries used included the substrate library for overriding functions and rocketbootstrap that was used to escape sandboxing issues when messaging between processes.

SHARED_CFLAGS are extra parameters passed to the C compiler. In this instance the *fobjc-arc* argument is passed. This indicates to the compiler that ARC should be enabled when compiling this code.

Lastly there is a final import of another make file provided by the Theos system, *tweak.mk*. As Theos can build many different types of project, the specific type of project to be built needs to be defined and this is done by importing it's particular make file.

7.3 Storyboards and XIB

To create user interfaces for iOS, the Xcode IDE provides an interface builder tool in which components can be drag and dropped onto a canvas. Behind the scenes, this generates a complex XML-like file called a *xib*, containing all the information the compiler needs to generate the view in a .nib file. This process of compiling the interface is a feature that is supported in Xcode but not in the Theos build tool [68].

As a result of this, all application view design was moved into the code rather than built using the graphical interface builder tool. This meant that instead of drag and dropping, UI objects were initialised and added as sub-views to the current view from within the view controller. For a simple interface like the one found in CertManager this was a good alternative to using IB however could become a problem if a more complex project was developed for Theos.

One way to get around this was to compile the .xib files manually using the ibtool utility provided by Apple that is able to compile the xib into a nib, which could then be loaded dynamically in code. This is the approach that was chosen in order to design the splash screen to look like the application before data had been loaded. However as the design of the application was being changed and tested so often it was easier to design it using code rather than having to keep compiling the nib files each time a minor change was made.

7.4 iOS Framework Headers

As mentioned previously in the Theos build set up, iOS frameworks are not always available for public use and so Apple does not release public API's for them in the form of header files. One of the features of the Objective-C language and runtime is that no functions are truly private, messages are sent to classes and the class either has a handler to respond to the message or it doesn't. As a result of this, while a function may not be publically documented, any other object can still call it and this allows any application to use the private frameworks provided by Apple.

In order to find the names of functions within private frameworks, a utility called *class-dump-z* was used [88]. This is a command line utility which examines the Objective-C runtime information that is stored in the binary file containing compiled code. This particular project is a re-write of an existing project and is specifically aimed towards iOS binaries. This allowed for the exploration of the private frameworks by running them through the tool.

The frameworks required for the project to run were added into the Git repository as separate sub-module that can be downloaded by going into the *theos* folder and running *git pull* from within it.

7.5 Remote Debugging

While Xcode provides a way in which running applications can be debugged, applications written and built by Theos are unable to use this debugging utility. Throughout the development of CertHook it was required to see where bugs were in the code and what state data was in during the life time of the tweak. In order to do this there needed to be a way to view the output of the system log file (syslog).

The syslog daemon runs on the device and by using standard UNIX tools that had been compiled for iOS a way to interact with it was provided. The tool is called *socat* and stands for

Socket CAT, providing a way to establish streams of data between sources [99]. The syslog daemon runs a socket connection and it can be accessed using the command:

$$socat - UNIX-CONNECT:/var/run/lockdown/syslog.sock$$

Code 23 - The command to connect socat to the syslog.

This results in a shell being created in which commands can be sent to the daemon. Using the *watch* command creates a *tail* [100] like output whereby new information generated by the console is immediately printed into the console as they arrive [101].

By using *socat* the results of NSLog methods where able to be analysed and this helped greatly when debugging and working out what values were being passed. The issue with the system console is that it did not always provide information about why an application had crashed due to the injection of the code. This meant that looking at a stack trace was not possible and so it was sometimes hard to pinpoint where issues were occurring from. When Xcode runs an application it attaches an lldb debugger [102] to the process and acts as a wrapper for its output. In order to achieve similar results for debugging other applications, instructions were followed from an online source onto how to attach a console application called debugserver [103] to a process.

By default, debugserver is only able to attach itself to applications signed by Xcode, so in order to attach to system applications the original debugserver had its property list modified and then it was code signed again to apply the changes. This modified version was then loaded onto the device via SSH and then run using the command:

$$./debugserver *:1234 - a "Application Name"$$

Code 24 - Attaching debugserver to a process.

This would start a server on the device and attach itself to the application defined in “Application Name”. On the Mac, the *lldb* tool was run from inside the Terminal and then the remote server running on the iOS device was connected to it by running:

$$platform\ select\ remote - ios$$
$$process\ connect\ connect://IP\_ADDRESS:1234$$

Code 25 - Connecting to the debugserver from a remote machine.

Which then spawned a connection to the debugger in order for the process to be debugged. Using commands for lldb breakpoints were added to functions and then it was possible to wait for the code in CertHook to cause the application to crash. When this happened, more detailed information about what caused the crash was output to the console which allowed for issues to be resolved much quicker than previously.

7.6 Development Tools

Throughout the development of the system, various development tools were utilised. Different IDEs were needed in order to write code in different languages and build systems. Additionally third party applications were used in order to help with the research phase and to provide source control management. This section details the applications that were used in the project and what they were used for.

7.6.1 Xcode

Xcode is the industry standard for developing applications for both iOS and OS X [104]. The application contains a suite of different development tools that allow for fast creation of interfaces, built in code signing and deployment to either a physical device or an iOS emulator. Xcode provides full support for debugging applications with the ability to set breakpoints and analyse the state of the application memory at run time. Xcode was used to create the CertManager application as the built in autocomplete of method names proved to speed up development time dramatically. The CertHook application was not originally created in Xcode due to its lack of support for the Logos pre-processing language, however by creating aliases on the disk and renaming the files to .mm, Xcode could be tricked into processing the language as Objective-C++ and allow for autocomplete to work.

7.6.2 Sublime Text

Sublime Text is a powerful cross-platform text-editing tool with support for extensions to add further functionality [105]. Sublime offers syntax highlighting for hundreds of programming languages and, for building the CertHook application, made it easier to format and structure the code. Especially useful was the Make plugin that allowed the application to be build using from within the text editor without needing to keep switching to a terminal window.

7.6.3 Hex Fiend

Hex Fiend is an open source hex editor for OS X [106] that was used throughout the research stages of the project in order to analyse files within the iOS Security framework. The software allowed for analysis of the hex values of the *certsIndex.data* and *certsTable.data* files and this lead to the discovery of these files containing the root certificate information.

7.6.4 SourceTree

SourceTree is a GUI application for Mercurial and Git that runs on both Windows and OS X [107] and provides an easy way to manage a project that is under source control. In the case of this project, Git was chosen as the source control system. SourceTree proved valuable when it came to merging different feature branches together and when deciding what sections of code to add into individual commits. While the terminal application was used for general Git use, SourceTree was used when more advanced operations needed to be carried out such as merging and cherry-picking commits.

7.6.5 Terminal

Terminal.app is the default terminal emulator built into OS X. It provides a way in which the user can interact directly with the system using text-based commands in a bash UNIX shell. Terminal was used throughout the project, in order to build the project via command line and install remotely onto the test device, the Make program was invoked using terminal. The Git command was also heavily used for daily operations when making commits and pushing to remote servers. In addition to this, the terminal provided a means to debug the iOS code remotely without the need for Xcode. This was essential to the development of the CertHook tweak as it could not be attached to the Xcode debugger. In order to do this the SSH command was used in order to log in to the device remotely and then the system log could be watched and replayed back into the terminal.

7.7 Summary

This chapter has defined how the environment used for development was set up and provided information relating to the building of the project. Requirements of having a machine running OS X and a jailbroken iOS device have been explained and the use of the Theos build system has been demonstrated. In order to provide a means of debugging the application, two different solutions were discussed and their usefulness evaluated. Connecting to the system log daemon provided basic logging information whereas the *lldb* debugger allowed for custom breakpoints into applications that were not able to be viewed by the Xcode debugger. Finally, a list of applications that were used to help create the system is shown in order to present how they were used and the benefits each one had.

Chapter 8

Testing and Effectiveness

At this stage of the project it is important that the system is tested so that it's performance can be evaluated and so that there is evidence to support the claims that the project has completed the aims defined at the beginning. In order to do this two main testing criteria were defined, firstly a manual inspection of the behaviour seen within applications and secondly a more technical test to perform inspection on the network traffic coming to and from the device once the system had been activated.

8.1 Manual Inspection

In order to test the system from the application level, manual inspection was used to navigate to known websites over a HTTPS connection in the Safari browser pre-installed on the device. A variety of websites were accessed, each of which had a different known root certificate, in order to test across a range of connections.

URL	Root Certificate	Result
https://cnnic.cn	CNNIC ROOT	Blocked
https://www.facebook.com	DigiCert High Assurance EV Root CA	Blocked
https://www.google.com	GeoTrust Global CA	Blocked
https://m.twitter.com	VeriSign Class 3 Public Primary CA – G5	Blocked

Table 5 - The results of blocking different certificates in Safari.

Additionally, different iOS applications were used for testing to see the response of each one to the user. The expected output was that an error message should be shown to the user indicating some sort of invalid connection attempt or certificate occurred.

Application	Result
Mobile Safari	Blocked
AppStore	Blocked
Google Chrome	Allowed
locationd	Blocked
Twitter	Blocked
Facebook	Blocked

Table 6 - The results of application testing.

As shown in the table above, Google Chrome was unaffected by CertManager. This issue is addressed later in 9.2.2 Reliance on the Security Framework.

There was visual confirmation to the user that the connection was not successful in the form of the notifications from CertManager and messages from Safari indicating that the connection was not made. However this alone is not a good enough test to ensure that CertManager works as expected as a tool to increase security and privacy, as it does not ensure that any sensitive data left the device.

8.2 Network Traffic Analysis

In order to prove that no sensitive data was leaked to an attacker, the attack was replicated on a test iOS device with CertManager installed. A tool named *Charles* [108] was used, which is an application to proxy Internet traffic on a computer in order to extract information. One feature of Charles is that it can also be used as a remote proxy for devices on the same network. This allowed for the test iOS device to be connected to a computer by modifying the HTTP proxy from within the Wi-Fi settings. Once this was set up, all Internet traffic on the iOS device was routed via the laptop and intercepted by Charles. At this stage, only HTTP traffic could be read, as it is unencrypted, and so a way to intercept HTTPS traffic was needed.

The aim of this test was to see if CertManager provided protection against the compelled certificate creation attack described previously. This would not be possible to test because a trusted certificate authority would not generate a certificate for a well known website without proof of ownership or a legal request, neither of which was possible in this situation. In order to replicate the attack a fake certificate authority was created that was able to sign certificates for any domain. The fake certificate authority was then added to the device and this way the test device would treat the fake certificate authority the same as any other root certificate when it came to making trust decisions. The OpenSSL package found on OS X was used in order to create the self-signed certificate.

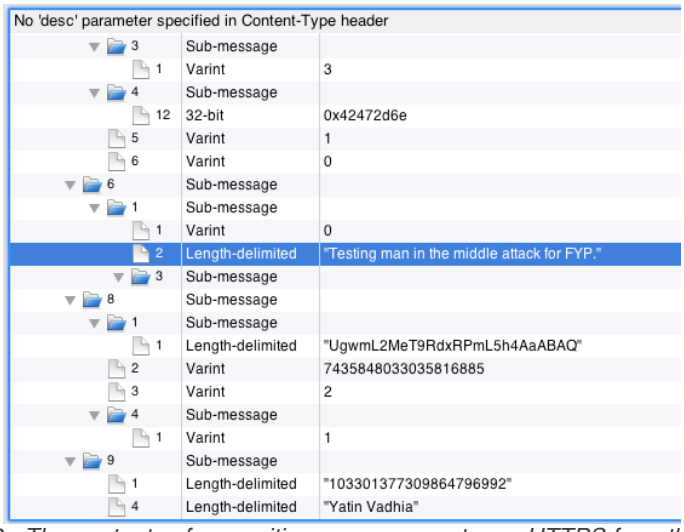
$$openssl\ req\ -x509\ -newkey\ rsa:1024\ -keyout\ fakeCA.key\ -out\ fakeCA.crt\ -days\ 3650\ -nodes$$

$$openssl\ pkcs12\ -export\ -out\ fakeCA.pfx\ -inkey\ fakeCA.key\ -in\ fakeCA.crt$$

Figure 42 - The commands used to generate a root certificate.

Once generated, the fake CA was added to the Charles proxy application that was then able to intercept the HTTPS handshake request coming from the phone and reply using a certificate for the requested domain signed with the fakeCA certificate. Initially the iOS device rejected the attack due to the fact that it did not trust the fakeCA certificate, which was to be expected. This is the security of the public key infrastructure working correctly as the generated root certificate does not exist in the device trust store and therefore certificates generated from it are not trusted either. In order to bypass this protection the certificate was emailed to the target device and then opened by the user to show an interface whereby the certificate was able to be trusted [Appendix D]. Apple have provided reasonable protection against adding root certificates to the device, with several confirmation messages the user had to accept and also an entry of the device passcode before it was added. It is worth noting at this stage that the certificate is not added to the system trust store, but a separate trust store owned by the user, as a result it will not show up in CertManager as part of the trusted system root certificates.

At this stage, the proxy could now decrypt all secure traffic giving the attacker the ability to view all sensitive data going to and from the device. This is an example of whereby the attacker would have access to an intermediate signing certificate in which they could sign their own certificates for all requests no matter what the domain. If the attacker only had access to a forged certificate for a single domain, then *Charles* has an option to target on a per-domain basis and allow all other SSL requests through as normal.



No 'desc' parameter specified in Content-Type header		
3	Sub-message	
1	Varint	3
4	Sub-message	
12	32-bit	0x42472d6e
5	Varint	1
6	Varint	0
6	Sub-message	
1	Sub-message	
1	Varint	0
2	Length-delimited	"Testing man in the middle attack for FYP."
3	Sub-message	
8	Sub-message	
1	Sub-message	
1	Length-delimited	"UgwmL2MeT9RdxRPmL5h4AaABAQ"
2	Varint	7435848033035816885
3	Varint	2
4	Sub-message	
1	Varint	1
9	Sub-message	
1	Length-delimited	"10330137730986479 6992"
4	Length-delimited	"Yatin Vadhia"

Figure 43 - The contents of a sensitive message sent over HTTPS from the device.

On the device, a connection was made to an example website, *facebook.com*, using the browser feature of CertManager. The connection went through without any warnings to the user and the green padlock indicating a secure connection was shown. When the padlock icon was clicked it showed the certificate chain for the current connection, the certificate provided for the Facebook website was not the original but one signed by the faked CA root certificate.

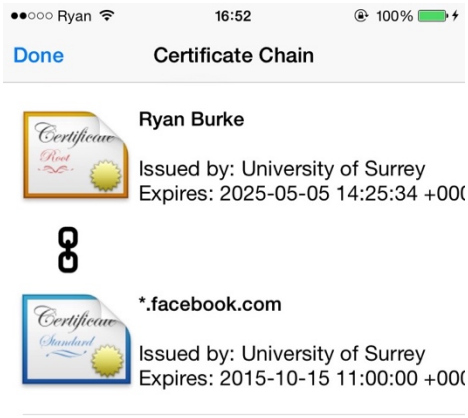


Figure 44 - The certificate chain of the MiTM attack, shown by CertManager.

From using the CertManager browser, this provides the user with a way in which they can manually inspect the certificates in use. Even without blocking the connection this is a useful tool to have and is a feature available on most modern browsers. In comparison, visiting the website in Safari gives no indication of who the server is talking to and what certificates are in use. If a victim viewed the certificate chain and noticed that the root certificate signing the Facebook certificate was say CNNIC, they would be able to jump to the conclusion that they were being MiTM attacked and add the certificate to their blocked list.

In order to replicate this scenario the *Ryan Burke* root certificate was added to the manual blocked list within CertManager. This immediately updated the CertHook system tweak settings and notifications appeared to the user indicating that many connections had been blocked that had been using that root certificate. In order to see the impact this had on the attackers ability to view sensitive information the output was recorded in Charles again.

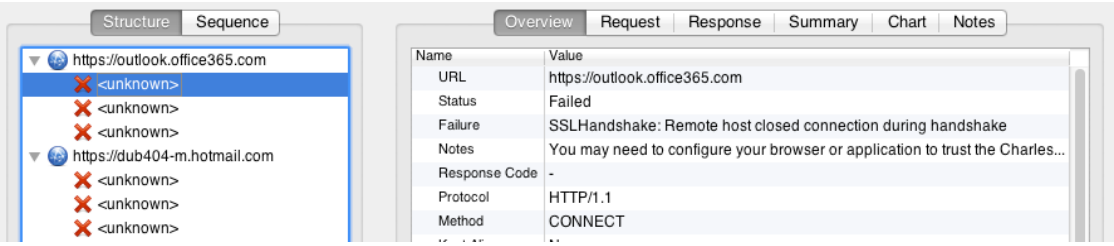


Figure 45 - Charles showing no data sent from the client.

This output was repeated for all connection attempts over HTTPS. The SSLHandshake method is reported to have failed when attempting to make a connection to the remote server. This means that no data is sent because this handshake happens before data is transmitted. By blocking the handshake the client defaults to failing the entire session, it is seen above that several subsequent attempts are made at reconnecting and then it gives up.

8.3 Summary

This section has defined testing criteria that were performed in order to test the effectiveness of the system within a test environment. Firstly, different certificates were added to the list of untrusted certificates and connections were made using the default iOS web browser, Safari. The expected behaviour was that the connections would fail and this proved to be the case for the range of certificates that were tested.

The second test was to see that the system worked across a range of different applications so that it was a system wide block, one of the requirements of the project. Different applications were used to connect while the root certificates that the applications used were blocked. This showed that certain applications would still be susceptible to attacks if they are not using the system implementation of the Security framework. This includes applications that use third-party security libraries such as OpenSSL to perform SSL related functionality.

The conclusion of these tests is that CertManager is effective in blocking a replicated MiTM attack, providing the user is able to block the certificate in use. No sensitive data is sent from the phone due to the SSL handshake failing as a result of CertHook performing additional checks on the validity of the certificate chain. This proves to stop a MiTM attack from gaining sensitive data but at the expense of the user being unable to make any connections from the device until the attack is stopped.

Chapter 9

Conclusion

This chapter contains a summary and evaluation of the project in general. It looks at the success criteria of the project that were defined in Chapter 1 and how far the system came in achieving them. The challenges that were faced throughout research and development are explored and the future of the project discussed. The report finishes with a final closing statement summarising the personal and technological achievements of the project.

9.1 Success Criteria

In order to gauge the level of success of this project I refer back to the project aims defined within the introduction.

Aim	Achieved
Research TSL/SSL implementation on iOS.	Yes
Identify how a TSL/SSL certificate can be extracted from an iOS network connection.	Yes
Create a mobile application that can: - Show certificates pre-installed on the device. - Allow users to change the trust option for certificates.	Yes
Provide a way in which a user can block all incoming and outgoing connections that do not match their trust preferences.	Yes
Evaluate the effectiveness of the system in both a test environment and a real world scenario.	Yes

Table 7 - Evaluation of project aims.

The first aim of this project was to research into how iOS handles SSL/TSL connections at the system level. This objective made up a large majority of the research that was conducted into the system and was achieved by using various sources of information including the Apple documentation, using Substrate to hook system level functions and by dumping header files to gain information about what data each function had access to.

The second aim was to identify a way in which certificate information could be extracted when a network connection occurred. This built on top of the research achieved in the first objective and using the Substrate library within the SSL handshake method. Having already researched and used functions within the Security framework, this made implementing certificate extraction within the web browser added to CertManager easier.

The third aim was to create a mobile application that was able to show users the certificates that were preinstalled on the device in the form of the trust store and allow them to choose whether or not they trusted these certificates. This was realised through the CertManager application and by performing research into how certificates could be extracted from the trust store. This lead to porting the *securityd* open source project provided by Apple onto iOS and building a small sub-set of the project in order to compile the code. Once this had been completed the certificates could be accessed and displayed to the user with the ability to toggle trust being saved onto the disk.

The fourth aim was achieved through the development of the CertHook application. At the beginning of the project it was unsure how this blocking mechanism would be implemented and research was conducted into how certificates could be blocked at various levels of the system. Initially the file system was analysed and the raw trust store data was modified in order to make the certificate chain invalid. While this worked, it was decided that it was not suitable for a consumer based application and that a safer method would be to use the Substrate library. As an extension to this aim, the added functionality of being able to block certificates outside the scope of the trust store was added. One improvement to the blocking system could be that there might be a more efficient place to implement the block that has not been found by the research conducted in this project. The SSLHandshake is called quite a few times every time a request is made and therefore the custom code is also called often.

The last aim was to evaluate the effectiveness of the system and this was achieved through the testing conducted after the system had been built. By using the *Charles* tool, all network traffic could be viewed and replicating the compelled certificate creation attack gave a practical real world example of the attack. Admittedly, a determined attacker could use alternative methods of bypassing the protection provided by the system, such as detecting when their certificate had been blocked and switching to another one.

9.2 Challenges

A range of challenges have been overcome in the project. These are discussed here, together with some ideas on how to further enhance the project's usefulness and technical foundation.

9.2.1 Lack of documentation

One challenge that was faced throughout the entirety of this project was the lack of documentation that was available. Due to the nature of the project, documenting the inner workings of the iOS private frameworks is not something that Apple provide. This is in contrast to the large amount of documentation and example code available for writing the front end application, which was vastly easier to develop for than the undocumented sections. In order to work around this, various sources including the *class-dump-z* [88] tool and GitHub header dumps [109] were used in to obtain the header files of classes. Luckily, Objective-C encourages the use of self-describing method names which usually provided enough context about what a function's objective was that looking for individual functions was simplified.

In addition to the Apple private frameworks, the CertHook project is developed in an environment which is very niche. There are not many developers who create content for jailbroken iOS devices and even less who write documentation as projects are usually built in leisure time and not as professional projects. Fortunately many of these developers are still active within IRC chat rooms and provide support for other developers using their code. Many times when documentation could not be found online on how to do something, asking the question in the *irc.saurik.com* server resulted in a response from the original developer. The generosity of the jailbreak development community can not be understated as they were a vital resource in helping debug niche issues that were sometimes unique to this project.

One thing that the author tried to do was to give back information discovered through research by updating the website "The iPhone Wiki" when new functionality was found. It is hoped that through helping to enhance the small amount of documentation on the iOS system that other developers will be able to take advantage of this information. In addition to this, the project has been developed with the aim of becoming an open sourced project available through

GitHub. This meant that Git commits contained useful information about changes being made to the project and code was commented throughout so that it can be used to teach others.

9.2.2 Reliance on the Security Framework

The limitation of CertHook is that it only works when applications use the Security framework for SSL connections. Fortunately, this covers the majority of applications and networking libraries that are commonly found. One example of an application that does not use the Security Framework is Google Chrome for iOS, which was seen previously when the system was being tested in different applications.

On OS X, Google Chrome continues to obtain trust decisions by the system's Keychain Services, meaning that changes made by the user will impact certificate validation within the browser. However for iOS this is not the case, instead it uses its own certificate path-building engine, backed by OpenSSL, which it uses against its own version of a trust store [110]. As a result of this, CertHook is unable to stop requests because Chrome is not calling the functions in the Security Framework and therefore the Substrate override is never called. A similar patch for Chrome could developed using the same blocking algorithm but this would need to be done for each application implementing its own custom SSL functionality, or by looking at the common libraries used like OpenSSL and writing extra patches for those.

9.3 Future Work

While the system has proven to be effective and meets the targets set at the start of the project, there is room for improvement in the future. At current, the system is a reactive form of defence, with the user needing to chose which certificates they want to block. Now that the foundation of the project has been implemented, future work could include improving the algorithm that blocks the certificates to become a proactive function. In order to do this, an algorithm similar to that used by the Certlook Firefox add-on described by Soghoian and Stamm [35] could be implemented. This would work by using the properties of the X.509 certificates within the chain of trust in order to calculate how likely a certificate is fraudulent or not. Factors that would impact this would be the certificate hash, country and name of the CA and the country and name of the website. Each time a website is visited using HTTPS, these factors are stored by the system and then when the page is visited again it is able to perform a trust analysis. A certificate that looks suspicious could prompt the user to chose if they want to allow the connection or not.

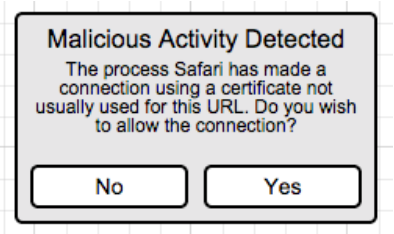


Figure 46 - Potential detection mechanism for future work.

An exciting project related to this area of research is Certificate Transparency by Google [111], which looks to fix the structural flaws found within the PKI and SSL. Certificate Transparency adds new components to the existing SSL system which allows for logging and auditing of the certificates which CA's issue. Whenever a new certificate is created by the CA a record of this is sent to external logging servers who keep a record and return a *signed certificate timestamp* that is then added into the certificate. When the client receives the certificate it can extract the

SCT and make sure that it is valid. In addition to this, third party auditors can view the entire log files and check it for malicious activity. This creates an element of auditable trust in CA's instead of blind trust based on the fact that the issuer is a large company. If this system ever becomes common and the infrastructure is set up but not supported in iOS, it is possible that this sort of additional check could be implemented by the CertHook application to make sure that certificates are within this log file and to block any that are not.

Ideally, the future of this project would be limited if Apple were able to implement a similar system to CertManager into iOS as a native feature of the operating system. If users were given the choice about which CA's they wanted to trust like they are on most other platforms, then the need for this project would be nullified. In order to help drive future work towards this system, it will be released on GitHub as an open sourced project and shared with the jailbreak developer community. Quite often it is the case that developers will branch the projects to add their own features or to fix bugs and then request to merge it back into the main project once it has been peer reviewed.

9.4 Closing

Throughout the project, the CA that was often referenced as potentially performing rogue actions was the CNNIC. In the process of completing this project an attack was identified by Google on their services through the use of unauthorized certificates. These certificates were issued by an intermediate certificate held by a company called MCS Holdings who had obtained their certificate from the CNNIC authority [112]. In response to this, Google removed trust for the root CNNIC certificate that ships with Google Chrome, as did Mozilla for Firefox [113]. This situation highlights the exact reason for the need of this system on the iOS platform as at the time of writing Apple have still not removed the certificate from their own trust store for iOS and OS X and users remain vulnerable.

In conclusion, this project was able to achieve the goals that it set out to achieve. The project aims had been met and exceeded through the additional functionality that was added to the application. Evaluating the real world practicality of the project, one of the biggest drawbacks is that it requires a jailbroken iOS device which at the time of writing accounts for around 2 million devices [114]. Interestingly, 60% of jailbroken devices are from China which when looking at the recent MiTM activity by CNNIC certificate authority [115], could prove that a tool such as CertManager has potential to be used within that customer segment as a form of protection.

As far as I am aware, there is no other system that provides anything like the functionality of CertManager available for the iOS platform and being the first of its kind made it an exciting project to work on. Personally, a lot was learnt in regards to developing for the iOS platform and the Objective-C programming language in order to develop the system. It provided an insightful look into the underlying system that powers devices used by millions every day and satisfied a personal interest into security and privacy for users. It is hoped that this project proves to be useful in the real world and its future development is continued by myself and others.

Bibliography

[1] comScore. (2014, August) ComScore. [Online]. http://www.comscore.com/layout/set/popup/content/download/26291/1346569/version/1/file/The+US+Mobile+App+Report.pdf

[2] Apple Inc. iOS 8. [Online]. https://www.apple.com/uk/ios/

[3] Google Inc. Android. [Online]. https://www.android.com/

[4] Cyanogenmod. Cyanogenmod | Android Community Operating System. [Online]. http://www.cyanogenmod.org/

[5] Ben Bajarin. (2011, July) Why Competing with Apple Is So Difficult. [Online]. http://techland.time.com/2011/07/01/why-competing-with-apple-is-so-difficult/

[6] Akamai. (2014) The Akamai State of the Internet Report. [Online]. http://www.akamai.com/dl/content/q4-2014-soti-a4.pdf

[7] Priyadarshan Patil Piyush Gupta. 4G- A NEW ERA IN WIRELESS TELECOMMUNICATION. [Online]. http://www.idt.mdh.se/kurser/ct3340/ht09/ADMINISTRATION/IRCSE09-submissions/ircse09_submission_13.pdf

[8] Kipp E.B. Hickman. (1995, February) The SSL Protocol. [Online]. http://ssllib.sourceforge.net/SSLv2.spec.html

[9] Louise Leenan Jannie Zaaiman, *ICCWS 2015 - The Proceedings of the 10th International Conference on Cyber Warfare and Security.*, 2015.

[10] Ewen MacAskill Glenn Greenwald. (2013, June) Boundless Informant: the NSA's secret tool to track global surveillance data. [Online]. http://www.pulitzer.org/files/2014/public-service/guardianus/05guardianus2014.pdf

[11] Hermann Kopetz, *Real-Time Systems.*: Springer, 2011.

[12] Saugatuck Technology. (2014, November) Digital Business Rethinking Fundamentals. [Online]. http://cbs2014.saugatucktechnology.com/images/Documents/Presentations/CBS14%20McNee%20Keynote%20-%20Cloud%20and%20Digital%20Business-12Nov2014.pdf

[13] Ewen MacAskill, Laura Poitras, Spencer Ackerman and Dominic Rushe Glenn Greenwald. The Guardian. [Online]. http://www.theguardian.com/world/2013/jul/11/microsoft-nsa-collaboration-user-data

[14] Matthew Fredrikson, Tadayoshi Kohno, Thomas Ristenpart Bruce Schneier. (2015, February) Surreptitiously Weakening Cryptographic Systems. [Online]. https://eprint.iacr.org/2015/097.pdf

[15] Google Inc. (2013, June) Asking the U.S. government to allow Google to publish more national security request data. [Online]. http://googleblog.blogspot.co.uk/2013/06/asking-us-government-to-allow-google-to.html

[16] Apple Inc. Apple Root Certificate Program. [Online]. https://www.apple.com/certificateauthority/ca_program.html

[17] Google Inc. (2011, August) Google Online Security Blog. [Online]. http://googleonlinesecurity.blogspot.co.uk/2011/08/update-on-attempted-man-in-middle.html

[18] Apple Inc. (2011, October) Apple Support. [Online]. http://support.apple.com/kb/HT4999

[19] (2008, August) IETF. [Online]. https://tools.ietf.org/html/rfc5246#section-1

[20] Radia Perlman, Mike Speciner Charlie Kaufman, *Network Security: Private Communication in a Public World*, 2nd ed.: Pearson, 2002.

[21] IETF. (2006, June) IETF - RFC 4514. [Online]. https://www.ietf.org/rfc/rfc4514.txt

[22] The CA/Browser Forum. Guidelines For The Issuance And Management Of Extended Validation Certificates. [Online]. https://cabforum.org/wp-content/uploads/EV-V1_5_5.pdf

[23] Carl Mummert Gary L. Mullen, *Finite Fields and Applications.*, 2007. [Online]. Finite Fields and Applications

[24] Q-Success. W3Techs - World Wide Web Technology Surveys. [Online]. http://w3techs.com/

[25] OpenSSL. Online Certificate Status Protocol utility. [Online]. https://www.openssl.org/docs/apps/ocsp.html

[26] Petr Dvořák. (2012, March) Web Archive - Details on SSL/TLS certificate revocation mechanisms on iOS. [Online]. https://web.archive.org/web/20121026094830/http://www.inmite.eu/en/blog/20120302-details-certificate-revocation-mechanisms-on-ios-iphone

[27] OpenSSL. OpenSSL. [Online]. https://www.openssl.org/docs/apps/openssl.html

[28] x2on. GitHub. [Online]. https://github.com/x2on/OpenSSL-for-iPhone

[29] iPhone Dev Wiki. iPhone Dev Wiki - Su. [Online]. https://www.theiphonewiki.com/wiki/Su

[30] The iPhone Wiki. [Online]. https://theiphonewiki.com/wiki/LibTiff

[31] Will Strafach and iPhone Dev Team. The iPhone Wiki. [Online]. https://theiphonewiki.com/wiki/ARM7_Go

[32] Jay Freeman. Cydia Substrate. [Online]. http://www.cydiasubstrate.com/

[33] Apple Inc. Objective-C Runtime Reference. [Online]. https://developer.apple.com/library/mac/documentation/Cocoa/Reference/ObjCRuntimeRef/index.html#//apple_ref/c/func/method_exchangeImplementations

[34] Saurik. Cydia Substrate. [Online]. http://www.cydiasubstrate.com/api/c/MSHookMessageEx/

[35] Sid Stamm Christopher Soghoian, "Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL," 2010. [Online]. https://s3.amazonaws.com/files.cloudprivacy.net/ssl-mitm.pdf

[36] PACKET FORENSICS. (2011) WikiLeaks. [Online]. https://www.wikileaks.org/spyfiles/docs/PACKETFORENSICS-2011-1UModuRack-en.pdf

[37] iSECPartners. GitHub - iOS SSL Kill Switch. [Online]. https://github.com/iSECPartners/ios-ssl-kill-switch

[38] Apple Inc. iOS Developer Library. [Online]. https://developer.apple.com/library/ios/technotes/tn2232/_index.html#//apple_ref/doc/uid/DTS40012884-CH1-SECSECURETRANSPORT

[39] National Institute of Standards and Technology. (2011, Aug.) National Vulnerability Database. [Online]. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-0228

[40] WebTrust. WebTrust Program for Certification Authorities. [Online]. http://www.webtrust.org/homepage-documents/item27839.aspx

[41] Empirical Magic Ltd. iOS Support Matrix. [Online]. http://iossupportmatrix.com/s/iOS_Support_Matrix_v3_1_2-October2014.pdf

[42] Apple Inc. Certificate, Key, and Trust Services Reference. [Online]. https://developer.apple.com/library/mac/documentation/Security/Reference/certifkeytrustservices/index.html#//apple_ref/c/func/SecTrustCopyAnchorCertificates

[43] Apple Inc. Availability.h. [Online]. http://www.opensource.apple.com/source/CarbonHeaders/CarbonHeaders-18.1/Availability.h

[44] Apple Inc. (2010) Apple Open Source. [Online]. http://opensource.apple.com/source/Security/Security-55471/sec/securityd/OTATrustUtilities.c

[45] Apple Inc. NSDictionary Class Reference. [Online]. https://developer.apple.com/library/prerelease/ios/documentation/Cocoa/Reference/Foundation/Classes/NSDictionary_Class/index.html

[46] Apple Inc. List of available trusted root certificates in iOS 8. [Online]. https://support.apple.com/en-us/HT204132

[47] RedHat. Using Indexes to Improve Database Performance. [Online]. https://access.redhat.com/documentation/en-US/Red_Hat_Directory_Server/8.1/html/Deployment_Guide/Deployment_Guide-Designing_the_Directory_Topology-Using_Indexes_to_Improve_Database_Performance.html

[48] IETF. (2002, April) Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. [Online]. https://tools.ietf.org/html/rfc3280#appendix-C.4

[49] Christian Ratzenhofer. Gitorious. [Online]. https://gitorious.org/community-ssu/maemo-security-certman/commit/0be038825a98dae2d80fd411a02cb4c86ed1b36a

[50] Mozilla. (2012, Dec.) Bug 825022 - Deal with TURKTRUST mis-issued *.google.com certificate. [Online]. https://bugzilla.mozilla.org/show_bug.cgi?id=825022

[51] jan0. GitHub. [Online]. https://github.com/jan0/isslfix

[52] SektionEins. (2014, April) SekitonEins. [Online]. https://www.sektioneins.de/en/blog/14-04-18-iOS-malware-campaign-unflod-baby-panda.html

[53] Nathan Jones, John Szumski Jack Cox, *Professional iOS Network Programming: Connecting the Enterprise to the iPhone and iPad.*: Wiley, 2012.

[54] Apple Inc. (1996, December) Web Archive. [Online]. https://web.archive.org/web/19970301172356/http://live.apple.com/next/961220.pr.rel.next.html

[55] Aaron Hillegass and Joe Conway Christian Keur, "Page 63," in *iOS Programming: The Big Nerd Ranch Guide*, 4th ed., 2014.

[56] Apple Inc. NSURL Class Reference. [Online]. https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSURL_Class/index.html

[57] RidiculousFish. Bridge. [Online]. http://ridiculousfish.com/blog/posts/bridge.html

[58] Apple Inc. Using Sockets and Socket Streams. [Online]. https://developer.apple.com/library/mac/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/SocketsAndStreams/SocketsAndStreams.html

[59] Apple Inc. iOS Developer Library. [Online]. https://developer.apple.com/library/ios/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/SecureNetworking/SecureNetworking.html

[60] Apple Inc. Mac Developer Library. [Online]. https://developer.apple.com/library/mac/documentation/Security/Reference/secureTransportRef/

[61] nygard. GitHub. [Online]. https://github.com/nygard/class-dump

[62] Apple Inc. iOS Human Interface Guidelines. [Online]. https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/

[63] PixelLove. PixelLove. [Online]. http://www.pixellove.com/

[64] Centers for Medicare & Medicaid Services. (2008, March) SELECTING A DEVELOPMENT APPROACH. [Online]. http://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/SelectingDevelopmentApproach.pdf

[65] Barry W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, May 1988.

[66] Girish Suryanarayana, *Refactoring for Software Design Smells: Managing Technical Debt.*, 2014.

[67] Linus Torvalds. Git. [Online]. https://git-scm.com/

[68] Agile Modeling. Examining the Agile Cost of Change Curve. [Online]. http://www.agilemodeling.com/essays/costOfChange.htm

[69] Apple Inc. OTATrustUtilities.c. [Online]. http://opensource.apple.com/source/Security/Security-55471/sec/securityd/OTATrustUtilities.c

[70] Apple Inc. Security Server and Security Agent. [Online]. https://developer.apple.com/library/ios/documentation/Security/Conceptual/Security_Overview/Architecture/Architecture.html

[71] Apple Inc. NSURLConnectionDelegate Protocol Reference. [Online]. https://developer.apple.com/library/mac/documentation/Foundation/Reference/NSURLConnectionDelegate_Protocol/index.html

[72] Apple Inc. Core Data Programming Guide. [Online]. http://developer.apple.com/library/mac/#documentation/cocoa/Conceptual/CoreData/cdProgrammingGuide.html

[73] Apple Inc. NSCoder Class Reference. [Online]. https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSCoder_Class/index.html

[74] GitHub. GitHub. [Online]. https://github.com

[75] Stack Overflow. [Online]. https://stackoveflow.com

[76] Apple Inc. (2010) Apple Open Source. [Online]. http://www.opensource.apple.com/source/Security/Security-55471/sec/securityd/SecTrustServer.c

[77] Apple Inc. (2003, August) Apple Open Source. [Online]. http://www.opensource.apple.com/license/apsl/

[78] Dirk-Willem van Gulik. (2012, April) Stack Overflow. [Online]. https://stackoverflow.com/questions/9749560/how-to-calculate-x-509-certificates-sha-1-fingerprint-in-c-c-objective-c

[79] Dave DeLong. Stack Overflow - How to read data from NSFileHandle line by line? [Online]. http://stackoverflow.com/a/3711079

[80] johnrubythecat. Stack Overflow - How to read data from NSFileHandle line by line? [Online]. http://stackoverflow.com/a/8027618

[81] Dave Gandy. Font Awesome. [Online]. https://fortawesome.github.io/Font-Awesome/

[82] Nicolas Spalinger & Victor Gaultney. (2007, Feburary) SIL Open Font License (OFL). [Online]. http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=OFL

[83] Alex Drone. GitHub. [Online]. https://github.com/alexdrone/ios-fontawesome

[84] LLVM Developer Group. clang: a C language family frontend for LLVM. [Online]. http://clang.llvm.org/

[85] Saurik IT. Cydia Substrate. [Online]. http://www.cydiasubstrate.com/api/c/MSHookFunction/

[86] iSECPartners. (2012) When Security Gets in the Way. [Online]. https://media.blackhat.com/bh-us-

12/Turbo/Diquet/BH_US_12_Diqut_Osborne_Mobile_Certificate_Pinning_Slides.pdf

[87] KennyTM, Uroboro Dustin Howett. iPhone Dev Wiki. [Online]. http://iphonedevwiki.net/index.php/CPDistributedMessagingCenter

[88] KennyTM. (2009, December) class-dump-z — Extracting class interface for Objective-C version 2 ABI.. [Online]. https://code.google.com/p/networkpx/wiki/class_dump_z

[89] Apple Inc. GitHub - BulletinBoard. [Online]. https://github.com/nst/iOS-Runtime-Headers/tree/master/PrivateFrameworks/BulletinBoard.framework

[90] Apple Inc. GitHub - BBBulletinRequest. [Online]. https://github.com/masbog/PrivateFrameworkHeader-iOS-iPhone-5.-/blob/master/BulletinBoard.framework/BBBulletinRequest.h

[91] Dino A. Dai Zov. Apple iOS Security Evaluation. [Online]. http://hakim.ws/BHUS2011/materials/DaiZovi/BH_US_11_DaiZovi_iOS_Security_WP.pdf

[92] Dionysus Blazakis. (2011, January) The Apple Sandbox. [Online]. https://reverse.put.as/wp-content/uploads/2011/06/The-Apple-Sandbox-BHDC2011-Paper.pdf

[93] iPhone Dev Wiki. [Online]. http://iphonedevwiki.net/index.php/Seatbelt#Working_around_the_sandbox

[94] Ryan Petrich. (2014, January) GitHub. [Online]. https://github.com/rpetrich/rocketbootstrap

[95] Karen Tsai. GitHub. [Online]. https://github.com/angelXwind/AppSync

[96] Linux. dpkg(1) - Linux man page. [Online]. http://linux.die.net/man/1/dpkg

[97] iPhone Dev Wiki. iPhone Dev Wiki - Theos. [Online]. http://iphonedevwiki.net/index.php/Theos

[98] Apple Inc. OS X ABI Mach-O File Format Reference. [Online]. https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/index.html

[99] Gerhard Rieger. socat - Multipurpose relay. [Online]. http://www.dest-unreach.org/socat/

[100] David MacKenzie, Ian Lance Taylor, and Jim Meyering Paul Rubin. (2012, March) TAIL. [Online]. http://unixhelp.ed.ac.uk/CGI/man-cgi?tail

[101] The iPhone Wiki. System Log. [Online]. https://www.theiphonewiki.com/wiki/System_Log

[102] LLVM Developer Group. The LLDB Debugger. [Online]. http://lldb.llvm.org/

[103] The iPhone Wiki. debugserver. [Online]. http://iphonedevwiki.net/index.php/Debugserver

[104] Apple Inc. Apple. [Online]. https://developer.apple.com/xcode/

[105] Jon Skinner. Sublime Text. [Online]. https://www.sublimetext.com/

[106] ridiculous_fish. RidiculousFish. [Online]. http://ridiculousfish.com/hexfiend/

[107] Atlassian. Atlassian. [Online]. https://www.atlassian.com/software/sourcetree/overview

[108] Karl von Randow. Charles Web Debugging Proxy. [Online]. www.charlesproxy.com

[109] Nicolas Seriot. iOS Runtime Headers. [Online]. https://github.com/nst/iOS-Runtime-Headers

[110] Ryan Sleevi. Chrome: From NSS to OpenSSL. [Online]. https://docs.google.com/document/d/1ML11ZyyMpnAr6cIIAwWrXD53pQgNR-DppMYwt9XvE6s/edit?pli=1#

[111] Google Inc. Certificate Transparency. [Online]. http://www.certificate-transparency.org/

[112] Adam Langley. (2015, March) Maintaining digital certificate security. [Online]. http://googleonlinesecurity.blogspot.co.nz/2015/03/maintaining-digital-certificate-security.html

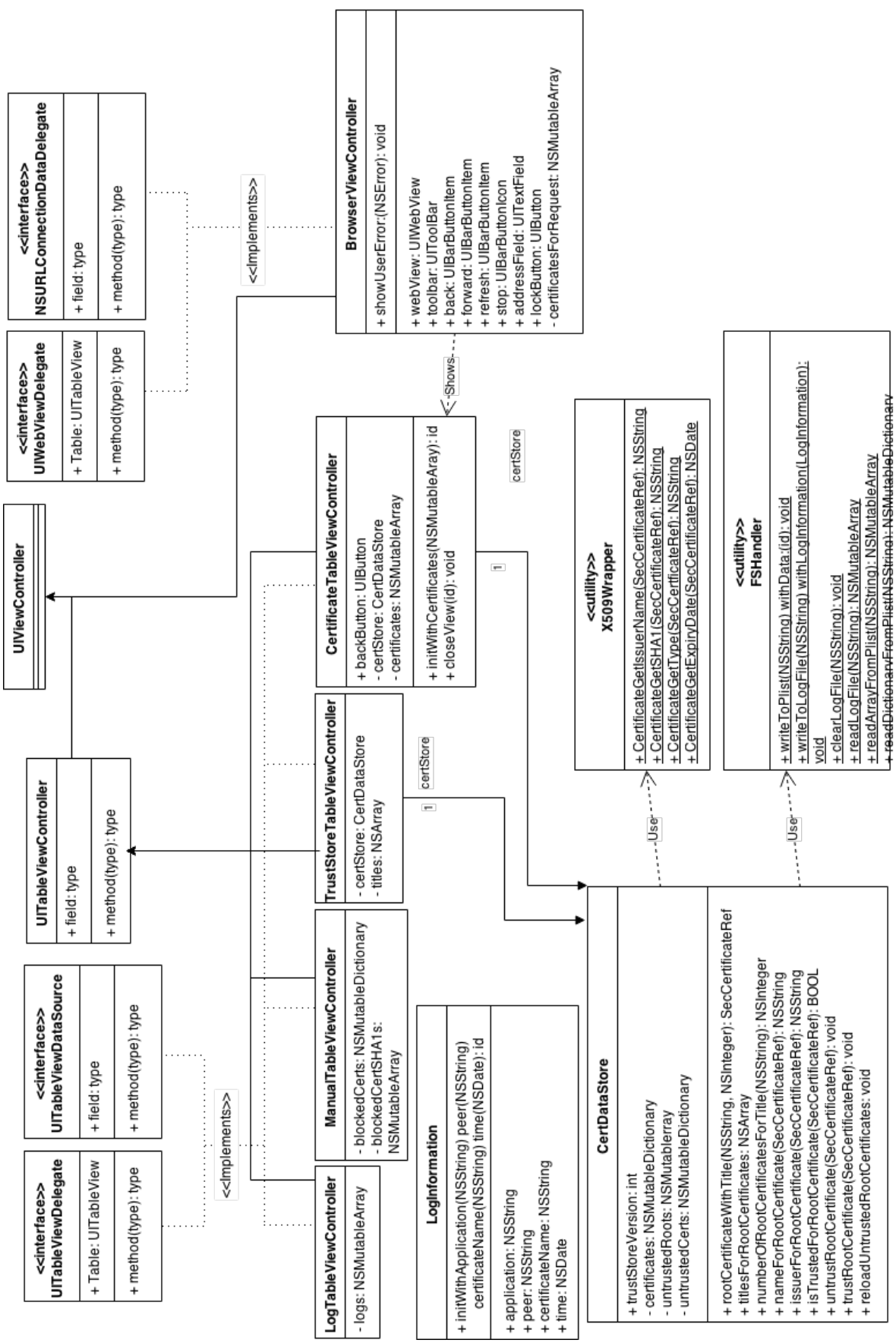
[113] Mozilla. The MCS Incident and Its Consequences for CNNIC. [Online]. https://blog.mozilla.org/security/files/2015/04/CNNIC-MCS.pdf

[114] Pangu Boot. (2014, July) Pangu Jailbreak Statistic: Almost 2 Million Jailbroken iOS Devices, 3 Million Unique Visitors, Total 7 Million Jailbreak request. [Online]. http://panguboot.com/pangu-jailbreak-statistic/

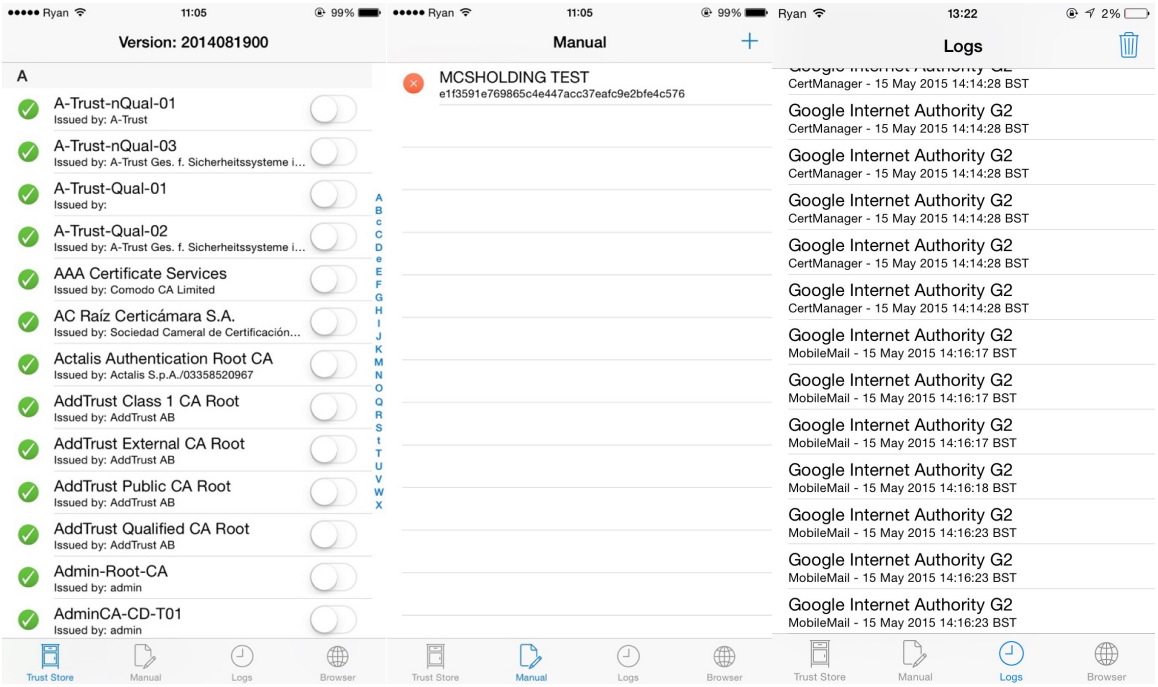
[115] Mozilla. Revoking Trust in one CNNIC Intermediate Certificate. [Online]. https://blog.mozilla.org/security/2015/03/23/revoking-trust-in-one-cnnic-intermediate-certificate/

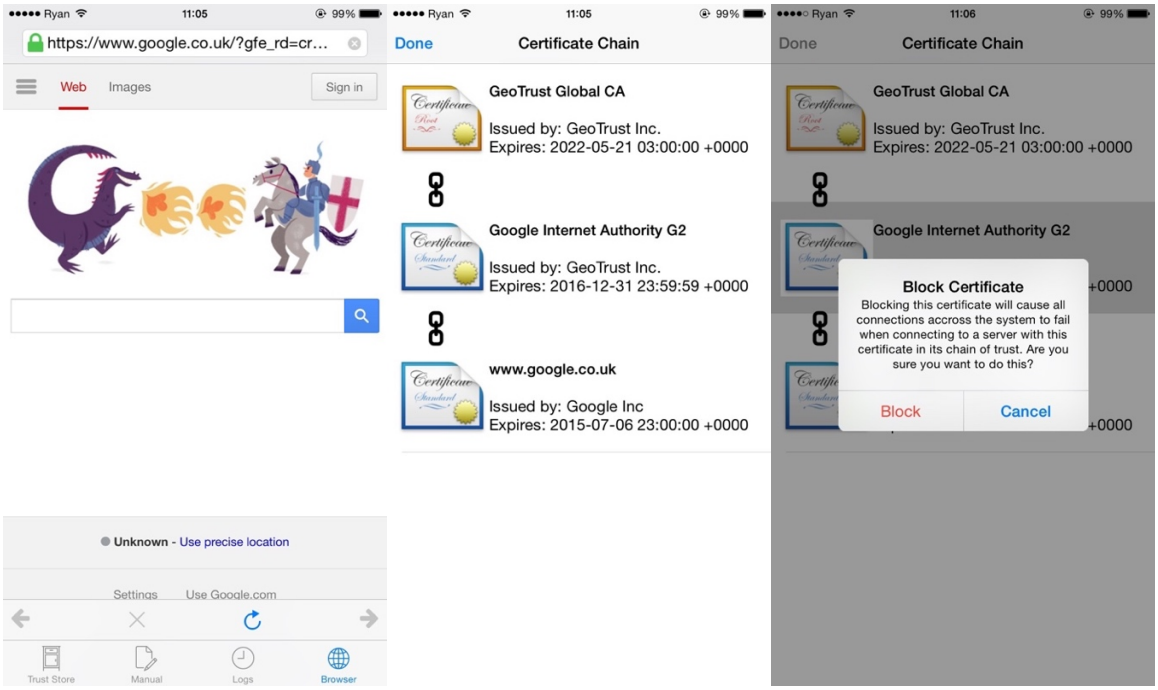
Appendices

Appendix A - CertManager class diagram.



Appendix B - CertManager user interface.





```
static OSStatus hooked_SSLHandshake(SSLContextRef context) {

    SecTrustRef trustRef = NULL;
    SSLCopyPeerTrust(context, &trustRef);
    size_t len;
    SSLGetPeerDomainNameLength(context, &len);
    char peerName[len];
    SSLGetPeerDomainName(context, peerName, &len);
    NSString *peer = [[NSString alloc] initWithCString:peerName
encoding:NSUTF8StringEncoding];
    CFIndex count = SecTrustGetCertificateCount(trustRef);
    //Does request contain certificates?
    if(count > 0) {
        //For each certificate in the certificate chain.
        for (CFIndex i = 0; i < count; i++)
        {
            //Get a reference to the certificate.
            SecCertificateRef certRef = SecTrustGetCertificateAtIndex(trustRef, i);
            //Convert the certificate to a data object.
            CFDataRef certData = SecCertificateCopyData(certRef);
            //Convert the CFData to NSData and get the SHA1.
            NSData * out = [[NSData dataWithBytes:CFDataGetBytePtr(certData)
length:CFDataGetLength(certData)] sha1Digest];
            //Convert the SHA1 data object to a hex String.
            NSString *sha1 = [[out hexStringValue] lowercaseString];
            OSStatus value = -1;
            //If the SHA1 of this certificate is in our lists.
            if([untrustedRoots containsObject:sha1]) { value = errSSLUnknownRootCert; }
            else if([untrustedCerts containsObject:sha1]) { value = errSSLClosedAbort; }
            if(value != -1) {
                NSString *summary = (__bridge NSString *)
SecCertificateCopySubjectSummary(certRef);
                BLOCKED_PEER = YES;
                certificateWasBlocked(certRef);
                return value;
            }
        }
    }
    else {
        if(BLOCKED_PEER) {
            BLOCKED_PEER = NO;
            return errSSLClosedAbort;
        }
    }
    BLOCKED_PEER = NO;
    return original_SSLHandshake(context);
}
```

Appendix D - Testing certificate installation.