Strathmore
UNIVERSITY

Strathmore University
SU+ @ Strathmore
University Library

Electronic Theses and Dissertations

2018

Secure plugin for automated software updates using Public Key Infrastructure for embedded systems

Victor M. Mbuvi
Faculty of Information Technology (FIT)
Strathmore University

Follow this and additional works at https://su-plus.strathmore.edu/handle/11071/5989

Recommended Citation

Mbuvi, V. M. (2018). *Secure plugin for automated software updates using Public Key Infrastructure for embedded systems* (Thesis). Strathmore University. Retrieved from https://su-plus.strathmore.edu/handle/11071/5989

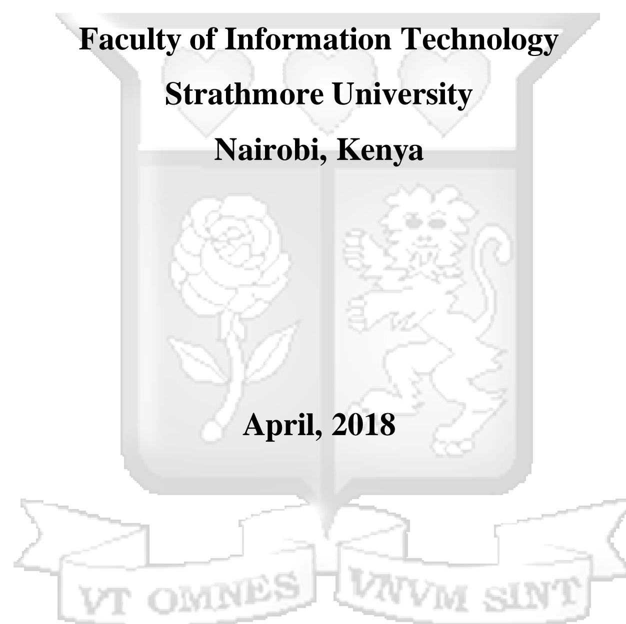
This Thesis - Open Access is brought to you for free and open access by DSpace @Strathmore University. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of DSpace @Strathmore University. For more information, please contact librarian@strathmore.edu

Secure Plugin for Automated Software Updates using Public Key Infrastructure for Embedded Systems

Mbuvi, Malombe Victor

A dissertation submitted in partial fulfillment of the requirements for the Degree of Master of Science in Information Systems Security at Strathmore University

Faculty of Information Technology
Strathmore University
Nairobi, Kenya



April, 2018

This dissertation is available for Library use on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Declaration

I declare that this work has not been previously submitted and approved for the award of a degree by this or any other University. To the best of my knowledge and belief, the dissertation contains no material previously published or written by another person except where due reference is made in the dissertation itself.

© No part of this dissertation may be reproduced without the permission of the author and Strathmore University.

Mbuvi, Malombe Victor

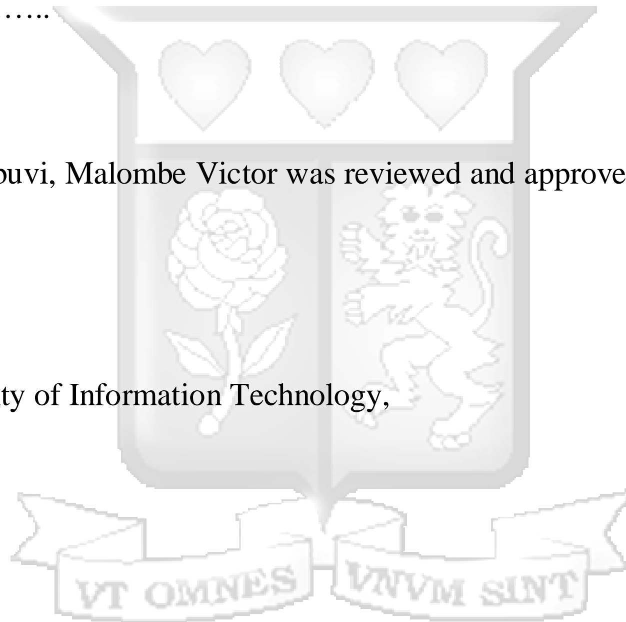
…………………………..

…………………………..

Approval

The dissertation of Mbuvi, Malombe Victor was reviewed and approved (for examination) by the following:

Dr. Vitalis Ozianyi
Senior Lecturer, Faculty of Information Technology,
Strathmore University



Abstract

Embedded systems are the driving force for technological development in many domains such as automotive, healthcare, and industrial control in the emerging post-PC era. As more and more computational and networked devices are integrated into all aspects of our lives in a pervasive and invisible way, security becomes critical for the dependability of all smart or intelligent systems built upon these embedded systems.

Most embedded device software is not updated after deployment. This is because chip manufacturers and system manufacturers (usually original device manufacturers) do not have any incentive, expertise, or even ability to patch the software once it is shipped. This leaves IoT developers to improvise their own ways of delivering software updates for embedded devices. These techniques do not have security in their design, and hence malicious updates from unauthorised sources may change the software leading to mass compromise.

This research reviewed previous work done using the Public Key Infrastructure in securing software updates in legacy systems and led to the development of a secure software updates plugin for embedded devices.

The prototype applies Experimental Research Design and Agile Development Methodology, for building of an evaluation platform. It provides opportunities to assess the dissertation progress and direction throughout the development lifecycle. This is achieved through iterations, coming up with a potentially stable product finally.

Key Words: Internet of Things (IoT), Secure Updates, Embedded Systems, Public Key Infrastructure, X.509

VT OMNES VNVM SINT

Table of Contents

Declaration i

Abstract ii

List of Figures vi

Abbreviations / Acronyms viii

Chapter 1 Introduction 1

1.1 Background of the Study 1

1.2 Problem Statement 2

1.3 Research Objectives 3

1.4 Research Questions 3

1.5 Justification of the Study 3

1.6 Scope 3

Chapter 2 Literature Review 4

2.1 Introduction 4

2.2 Embedded Systems Design and Requirements 4

2.2.1 Size and Cost 4

2.2.2 Power and Memory 5

2.2.3 Flexibility and Communication 6

2.2.4 Operating Systems and Programming Languages 6

2.2.5 Design Requirements and Limitations 7

2.3 Embedded Systems Security 7

2.3.1 Recent Attacks 8

2.3.2 IoT Vulnerabilities 9

2.4 Software Update Techniques 9

2.4.1 Secure Shell (SSH) 9

2.4.2 Port Forwarding 10

2.4.3 File Transfer Protocol (FTP) Clients 10

2.4.4 Local Access 10

2.5 Recommendations for Software Updates 10

2.6 Public Key Infrastructure 11

2.6.1 Overview 11

2.6.2 X.509 12

2.6.3 Public Key Infrastructure Operations 13

2.7 Secure Socket Layer / Transport Layer Security (SSL / TLS) 15

2.8 Cipher Suites 16

2.9 Systems Using PKI in Securing Updates in Other Cases 18

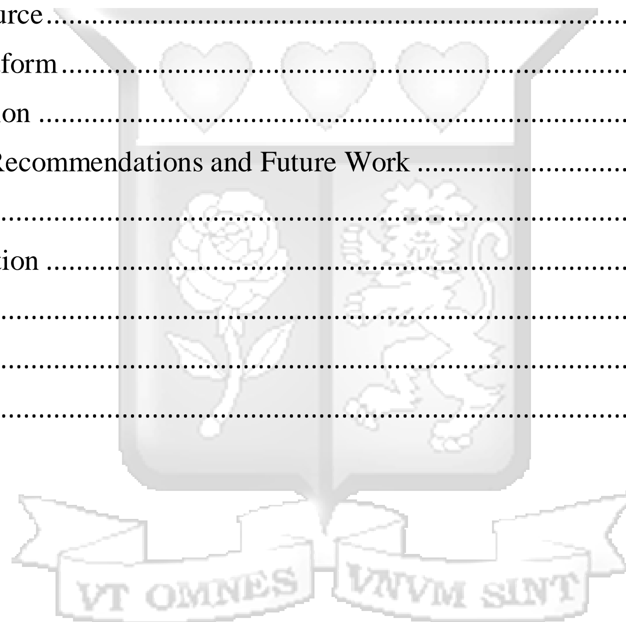
2.10 Conclusion 18

Chapter 3 Methodology 19

3.1 Introduction 19
3.2 Research Design 19
3.3 System Development Methodology 19
3.3.1 Planning 20
3.3.2 Requirements Analysis 20
3.3.3 System Design 20
3.3.4 Implementation 20
3.3.5 Testing 21
3.4 Conclusion 21
Chapter 4 System Design and Architecture 22
4.1 Overview 22
4.2 System Architecture 22
4.2.1 General Architecture 22
4.2.2 Authentication 23
4.2.3 Update Checking and Delivery 25
4.3 Use Case 26
4.3.1 Use Case 1 Run Update.sh script 27
4.3.2 Use Case 2 TLS Authentication 28
4.3.3 Use Case 3 Download Updates 29
4.4 Sequence Diagram 30
Chapter 5 System Implementation and Testing 31
5.1 Overview 31
5.2 Software Environment 31
5.2.1 Bash Programming 31
5.2.2 OpenSSL 31
5.2.3 Python Programming Language 31
5.2.4 Wireshark 31
5.2.5 Nmap 31
5.2.6 cURL 32
5.2.7 ShellCheck 32
5.3 Hardware Environment 32
5.4 Set-up and Configuration 32
5.5 Functions of the Prototype 33
5.5.1 Opening Secure TLS Connection 33
5.5.2 Checking and Downloading and Merging Updates from Remote Repository 35
5.5.3 Updating the Local Repository 35
5.6 System Testing 35
5.6.1 Test Plan 36

VT OMNES VNVM SINT

5.6.2 Functional Testing 39
5.6.3 Structural Testing 39
5.6.4 Testing for Sensitive Data Transmitted in Clear-Text 39
5.6.5 Testing for Weak SSL/TLS Ciphers/Protocols/Keys Vulnerabilities 39
5.6.6 Testing SSL/TLS Certificate Validity - Server 40
Chapter 6 Discussion of Results 41
6.1 Overview 41
6.2 Objective One 41
6.3 Objective Two 41
6.4 Objective Three 42
6.5 Objective Four 42
6.6 Advantages of the Developed Solution Compared to Existing Tools 43
6.6.1 Open Source 43
6.6.2 Multiplatform 43
6.6.3 Automation 43
Chapter 7 Conclusion, Recommendations and Future Work 44
7.1 Conclusion 44
7.2 Recommendation 44
7.3 Future Work 45
References 46
Appendices 51



List of Figures

Figure 2.1 Certificate Enrolment....13
Figure 2.2 Authentication Using Certificates....14
Figure 2.3 Certificate Revocation....15
Figure 2.4 Cryptographic architecture of SSL and TLS....16
Figure 2.5 PCAP listing the cipher suites that client supports....17
Figure 3.1 Agile Development Methodology....19
Figure 4.1 Auto Update Plugin System Architecture....22
Figure 4.2 TLS Authentication - Handshake Protocol....24
Figure 4.3 Update Checking and Delivery....26
Figure 4.4 Use Case Diagram....26
Figure 4.5 Sequence Diagram....30
Figure 5.1 Client Hello Packet....33
Figure 5.2 Server Hello Packet....34
Figure 5.3 Server Certificate Exchange Packet....34
Figure 6.1 Crontab....42

List of Tables

Table 2.1 Comparison of Platforms' Size, Weight and Cost 5
Table 2.2 Comparison of Platforms' CPU, Memory and Power 5
Table 2.3 Comparison of Platforms' Expansion Connectors and communication Interfaces 6
Table 2.4 Operating Systems and Programming Languages 6
Table 4.1 Run Update.sh Script Use Case 27
Table 4.2 TLS Authentication Use Case 28
Table 4.3 Download Updates Use Case 29
Table 5.1 Test Plan 36
Table 5.2 Test Case 37

Abbreviations / Acronyms

ADC	-	Analog to Digital Converter
AES	-	Advanced Encryption Standard
BIOS	-	Basic Input Output System
CA	-	Certificate Authority
CSR	-	Certificate Signing Request
FTP	-	File Transfer Protocol
IETF	-	Internet Engineering Task Force
IoT	-	Internet of Things
ITU-T	-	International Telecommunication Union Standardisation Sector
NAT	-	Network Address Translation
OWASP	-	Open Web Application Security Project
PKCS	-	Public-Key Cryptography Standards
PKI	-	Public Key Infrastructure
PRF	-	Pseudo Random Function
RSA	-	(Rivest–Shamir–Adleman), a public-key cryptosystem
RTC	-	Real Time Clock
SoC	-	System on a Chip
SSH	-	Secure Shell
SSL	-	Secure Socket Layer
TLS	-	Transport Layer Security
VPN	-	Virtual Private Network

Chapter 1 Introduction

1.1 Background of the Study

An embedded system is a computing system built into a larger system, designed for dedicated functions (Papp, Ma, and Buttyan, 2015). It consists of a combination of hardware, software, and optionally mechanical parts. Single board computers are often used to build embedded systems. Examples include Arduino, Raspberry Pi and Beaglebone (Jaziri, Charaabi & Jelassi, 2016).

The computing world is at a crisis point now regarding the security of embedded systems, where computing is embedded into the hardware itself, as with the Internet of Things. These embedded computers are shipped with vulnerabilities, and there is no good way to patch them (Schneier, 2014). With the growth of embedded systems market, more devices are connected to the Internet. The Internet of Things will put computers into all sorts of consumer devices.

Since IoT is a quickly developing and dynamic domain, the implementation of software components is liable to continuous changes addressing bug fixes, quality assurance or changed requirements. This has not been well addressed as would have been expected.

To understand this issue, it is critical to understand the embedded systems market. Companies such as Broadcom, Qualcomm, and Marvell make specialised computer chips that power embedded systems. These chips are cheap, and the profit margins slim. They typically put a version of the Linux operating system onto the chips, as well as a bunch of other open-source and proprietary components and drivers. They do as little engineering as possible before shipping, and there is little incentive to update their board support package until necessary (Schneier, 2014).

IoT devices have a reputation for being insecure at the time when they are manufactured. They are often expected to stay active in the field for more than ten years and operate unattended with Internet connectivity. Secure software updates are a critical capability for the success and growth of the IoT (Grau, 2016). IoT devices must support robust, secure, and scalable software and firmware update mechanisms. The historic approach of never upgrading devices in the field is not sustainable.

Public Key Infrastructure has been used to solve software updates security issues such as authentication and integrity. For instance, Microsoft Configuration Manager 2007 configures software updates to enable certificate revocation lists, which ensures better security against a certificate that has been revoked due to various reasons (Microsoft, 2015).

VT OMNES VNVM SINT

This dissertation focuses on using X.509 Public Key Infrastructure (Cooper, 2008) to ensure authentication, confidentiality and integrity of software updates for embedded devices.

Authentication ensures that updates are from authorised sources to authorised systems, confidentiality ensures privacy while integrity ensures updates are not altered over the communication channels (Kessler and Gary, 2003).

Initially, these cryptographic solutions have been avoided in IoT security due to limited processing capabilities of the embedded devices. However, a trend is developing whereby IoT hardware embarks security chips to speed up cryptographic computations, and that provide means to store keys securely (NXP, 2016).

Vast scale Internet of Things (IoT) service deployments keep running on a high number of distributed and interconnected computing nodes. These include sensors, actuators, gateways and cloud infrastructure. To guarantee the consistent monitoring and control of procedures, software updates must be conducted while the nodes are working without losing any sensed data or actuator guidelines (Weißbach et al., 2016).

1.2 Problem Statement

Most embedded device software is not updated after deployment. This is because chip and single board computer manufacturers do not have any incentive, expertise, or even ability to patch the software once it is shipped. The chip manufacturer is busy shipping the next version of the chip, and the original device manufacturers is busy upgrading its product to work with this next chip. Maintaining the older chips and products is not a priority.

In some cases, new devices have old software. For instance, one survey of common home routers found that the software components were four to five years older than the device (Heffner, 2010).

This leaves IoT developers to improvise their own ways of delivering software updates for embedded devices. These techniques do not have security in their design, and hence malicious updates from unauthorised sources may update the software leading to mass compromise.

Single board computers are not meant for standard desktop uses that require constant end user interaction. They are more suited for projects that require interaction with devices and sensors, hence minimal user interaction. This design limitation makes it inefficient for developers to keep their software up to date across multiple devices.

1.3 Research Objectives

1. To investigate how software updates are delivered and identify gaps and challenges in updating software in embedded systems,
2. To investigate the suitability of PKI in delivering software updates securely,
3. To design and implement an automated PKI based updates plugin for embedded systems, and
4. To validate the PKI based updates plugin for authenticity and integrity.

1.4 Research Questions

1. What methods do IoT developers use to deliver software updates, and what are the gaps and challenges in these methods?
2. How has PKI been used to ensure secure updates in other systems?
3. How can an automated PKI based updates plugin, if implemented, improve the security in delivering software updates in embedded systems?
4. How well does the based updates plugin address the gaps and challenges in delivering software updates in embedded systems?

1.5 Justification of the Study

This research has led to development of a plugin that enables embedded devices to receive updates automatically only from authentic sources, while ensuring that the delivered updates remain similar to those in the source.

This plugin can be integrated in different embedded system software running in single board computers.

1.6 Scope

This research focus only on network security, specifically on transport layer of the networks communications stack. This only applies to data in transit, but not protection of systems or stored data.

VT OMNES VNVM SINT

Chapter 2 Literature Review

2.1 Introduction

This chapter begins by reviewing the features and design requirements of embedded systems that use single board computers. The follow up sections look at different mechanisms that developers use to update their software in single board computers, including remote access and local access. The concept of Public Key Infrastructure is also reviewed, and ways in which it has been used to ensure secure software updates reviewed. The final section gives conclusions on the study based on the literature review by highlighting the research gap and need for an automated secure system for updating software in embedded systems.

2.2 Embedded Systems Design and Requirements

Embedded systems are important class of electronic systems which can be found everywhere. According to Liu et al. (2013), a device that includes a programmable computer but is not itself a general-purpose computer is an embedded system. Embedded systems are often built using single board computers.

Single board computers refer to computers complete with processor, RAM, graphics, and other peripherals on a single board (Isikdag, 2015). Examples include Raspberry Pi and Beaglebone boards. They run most popular embedded operating systems such as ARM architecture Linux flavours and Windows Embedded Compact (CE).

Characteristics of embedded systems include real-time operation, low manufacturing cost, low power and resource conscious.

Single board computers were originally designed for educational projects, embedded controllers or for use as development systems (Kruger & Hancke, 2014). Today, numerous collection of use single board computers to integrate their functions.

2.2.1 Size and Cost

Single board computers are being designed to be smaller and cheaper as time goes by. Ease and cost of platform deployment is directly influenced by the physical size and cost of each platform (Maksimović et al., 2014). Ease and cost of platform deployment is directly influenced by the physical size and cost of each platform. A reduction in per-platform cost will result in the ability to purchase more of them, to deploy a collection network with higher density, and to collect more data (Beagle_Board, 2018). Table 2.1 below presents size, weight and cost of several embedded systems.

Table 2.1 Comparison of Platforms’ Size, Weight and Cost

Name	Size (mm)	Weight (g)	Cost per node US$
Raspberry Pi	85.6 x 53.98 x 17	45	25-35
Arduino (Uno)	75 x 53 x 15	~30	30
BeagleBone Black	86.3 x 53.3	39.68	45
Phidgets	81.3 x 53,3	60	50 - 200
Udoo	110 x 85	120-170	99-135

2.2.2 Power and Memory

The main goal of embedded systems is low power consumption to enable it run in remote or unattended environments (Hill, 2003). According to Richardson and Wallace (2012), microcontrollers can be powered from a range of sources including computer USB Port or powered USB hub (will depend on power output), special wall warts with USB ports, mobile phone backup battery (will depend on power output), solar charger for cell phone, and alkaline batteries.

Regarding the storage, the device should have sufficient memory to store the collected data (Upton & Halfacree, 2014). Table 2.2 below shows a comparative information on processor, random access memory and power rating of various single board computers.

Table 2.2 Comparison of Platforms’ CPU, Memory and Power

Name	Processor	RAM	Power
Raspberry Pi	ARM BCM2835	512 – 1000 MB	5V/ USB
Arduino	ATMEGA8, ATMEGA168, ATMEGA328, ATMEGA1280	16-32 KB	7-12V /USB
BeagleBone Black	AM335x 1GHz ARM® Cortex- A8	512 MB	5V
Phidgets	PhidgetSBC	64 MB	6-15V
Udoo (Quad)	Freescale i.MX6Quad, 4 x ARM® CortexTM-A9 core Atmel SAM3X8E ARM Cortex- M3 CPU	1 GB	6-15V

2.2.3 Flexibility and Communication

The architecture must be flexible and adaptive in order to be used in wide range of applications (Maksimović et al., 2014). Unlike a desktop personal computer, single board computers often do not rely on expansion slots for peripheral functions or expansion. They use general purpose input and output (GPIO) pins which can accept input and output commands and thus can be programmed (Schmidt, 2014). Some devices have a standard RJ45 Ethernet adapter and WiFi modules for communication (Vujovic & Maksimovic, 2014). Table 2.3 below analyses the connectors and communication interfaces of some devices.

Table 2.3 Comparison of Platforms' Expansion Connectors and communication Interfaces

Name	Analog inputs	Digital I/O pins	USB ports	LAN (Mbit)
Raspberry Pi	0	14	1-2	10/100/1000
Arduino	6	14	1	10/100
BeagleBone Black	6	14	1	10/100
Phidgets	8	8I+8O	6	-
Uddo (quad)	14	62+14	5	10/100/1000

2.2.4 Operating Systems and Programming Languages

The operating systems vary from traditional operating systems in terms of goals and technique and each system differs substantially in the approach to memory protection, dynamic reprogramming, thread model, real-time features, et cetera (Maksimović et al., 2014). Table 2.4 below indicates operating systems and programming languages across several devices.

Table 2.4 Operating Systems and Programming Languages

Name	Board operating system	Programming language
Raspberry Pi	Raspbian, Ubuntu, Android, ArchLinux, FreeBSD, Fedora, RISC OS	C, C++, Java, Phyton
Arduino	/	Arduino
BeagleBone Black	Linux Angstrom	Arduino, Python
Phidgets	Linux	Visual Basic, VB.NET, C#, C/C++, Flash/Flex, Java, Labview, Matlab, ActionScript 3.0, Cocoa
Udoo	Ubuntu, Android, Linux, ArchLinux	Arduino, C, C++, Java

2.2.5 Design Requirements and Limitations

According to Kuchcinski (2017), various design requirements embedded systems has to be taken into account. These include:

Software Design Requirements

Embedded systems' software should be flexible enough to adapt to possible or future changes in its requirements. This leads to the need to support reconfiguration whereby it can change its behaviour by loading different configware code. Also, embedded systems' software should be easy to update. Software update needs to be automatic and secure, without human intervention where possible. Finally, the most software needs to be open source or very cheap to gain the benefits of collaborative research and continuous improvement.

Hardware Design Requirements

Most microprocessors have been designed with limited processing speed capabilities, however, if this is increased it will enhance the ability to process more data.

Embedded systems need to work for long hours without human intervention, hence they need to be designed to use as less power as possible.

Limitations

Challenges facing embedded systems and IoT at large emanate from the design features themselves (Maksimović et al., 2014). These include limited processing speed and need to use minimum power. For instance, Raspberry Pi does not allow any connected device to draw more than 100mA from any of its USB ports (Richardson & Wallace, 2012).

Single board computers do not have Basic Input Output System (BIOS). It uses a firmware that is closed-source proprietary code programmed into the System on a Chip (SoC) processor, which cannot be modified. Upon power-up the firmware will initiate a bootloader on the SD card (Kuchcinski, 2017). This makes it very hard to troubleshoot boot problems.

Single board computers do not have built-in Analog to Digital Converter (ADC). External component must be used for analog-digital conversion. Finally, these systems do not have real-time clock (RTC) with a backup battery. They only depend on network time server to synchronise time.

2.3 Embedded Systems Security

Security is an important issue because of the roles of embedded systems in many mission and safety critical systems. Attacks on cyber systems are proved to cause physical damages (Marwedel, 2010). Comparing to conventional IT systems, security of embedded systems is no

better due to poor security design and implementation and the difficulty of continuous patching (Schneier, 2014).

According to Bertino (2016), Internet of Things (IoT) systems are vulnerable due to various reasons. These include not having well defined perimeters and continuously changing due to device and user mobility. IoT systems are highly heterogeneous with respect to communication medium and protocols, platforms, and devices. They could be autonomous entities that control other IoT devices and might include "things" not designed to be connected to the Internet. Also, IoT systems, or portions of them, could be physically unprotected and / or controlled by different parties. Unlike smartphone applications, which require permission for installation and many user interactions, granular permission requests might not be possible in IoT systems because of the large number of devices.

2.3.1 Recent Attacks

There is an alarming rise of hacks and breaches exploiting vulnerabilities within the IoT. As part of Internet of Things (IoT) infrastructure, embedded systems have suffered major attacks recently. These include:

Linux.Darlloz Worm: In November 2013, Symantec researchers discovered the Linux.Darlloz worm, which exploited a PHP vulnerability to propagate to IoT devices such as home routers, TV set-top boxes, security cameras, printers, and industrial control systems (Wang et al., 2017). The worm comprises a script preloaded with default usernames and passwords to crack into vulnerable systems. Once a device is infected, Darlloz starts a HTTP Web server on port 58455 in order to spread. Its primary objective was to mine cryptocurrency. Symantec estimates that at least 31,716 identified IP addresses in 139 regions were infected by Darlloz.

Mirai Malware: In September 2016, an IoT botnet built from the Mirai malware was discovered (Goodin, 2016). The malware uses a list of 62 common default usernames and passwords to gain access primarily to home routers, network-enabled cameras, and digital video recorders, which usually have less robust protection than other consumer IoT devices. According to Bellemans (2017), the botnet was responsible for a 600-Gbps attack targeting Brian Krebs's security blog (krebsonsecurity.com).

2.3.2 IoT Vulnerabilities

The Open Web Application Security Project (OWASP) IoT project (2016) lists top 10 vulnerabilities.

These include an insecure web interface that can be present when issues such as account enumeration, lack of account lockout or weak credentials are present.

IoT devices may have insufficient authentication/authorisation when weak passwords are used or are poorly protected; some default passwords may still be in use.

Also, IoT devices may run insecure network services that may be susceptible to buffer overflow attacks or attacks that create a denial of service condition leaving the device inaccessible to the user.

Lack of transport encryption/integrity verification is common in IoT, which allows data to be viewed as it travels over local networks or the Internet.

IoT devices may have privacy concerns due to the collection of personal data in addition to the lack of proper protection of that data is prevalent.

Insecure cloud interface is another vulnerability that is present when easy to guess credentials are used or account enumeration is possible.

Insecure mobile interface which is present when easy to guess credentials are used or account enumeration is possible.

Insufficient security configurability is present when users of the device have limited or no ability to alter its security controls.

The lack of ability for a device to be updated presents a security weakness on its own, leading to insecure software/firmware.

Finally, physical security weaknesses are present when an attacker can disassemble a device to easily access the storage medium and any data stored on that medium.

2.4 Software Update Techniques

2.4.1 Secure Shell (SSH)

Secure Shell (SSH) is a protocol for secure remote login and other secure network services over an insecure network (Ylonen & Lonvick, 2006). Software developers use SSH to connect to single board computer's command line from a Linux computer, a Mac, or another single board computer, without installing additional software.

To connect to a from a different computer (such as a server), a user types the command below in their terminal (replace <IP> with the IP address of the remote system):

```
ssh username@<IP>
```

VT OMNES VNVM SINT

If the username and IP address are correct, the system prompts for the password, which should be entered for a successful connection to the remote system terminal. Now the user can access and edit any software using powerful text editors such as VIM (Oualline, 2001).
However, SSH does not provide server authentication (Turner, 2014). It is the responsibility of the accessed node to verify that the public key is indeed authentic, which presents a challenge.

2.4.2 Port Forwarding

Port forwarding is an application of network address translation (NAT) that redirects a communication request from one address and port number combination to another while the packets are traversing a network gateway, such as a router or firewall (Park, 2011). Through port forwarding, developers can access a remote system on the Internet which are behind a router.
However, port forwarding exposes a network port on a private local area network (LAN) to the public Internet (Raspberry Pi Foundation, 2018). This is a known security vulnerability and must be managed carefully.

2.4.3 File Transfer Protocol (FTP) Clients

FTP is a protocol for exchanging files over any network supporting the TCP/IP protocol (Postel & Reynolds, 1985). FTP clients such as FileZilla (FileZilla® Project, 2018) are used to transfer data and edit remote files.
The FTP protocol is implemented on top of Telnet, which suffers from serious security issues that have limited its usefulness in environments where the network cannot be fully trusted (Boe & Altman, 2002).

2.4.4 Local Access

The developer physically accesses the system and logs in to the operating system to update any software (Raspberry Pi Foundation, 2018).

2.5 Recommendations for Software Updates

Software updates are needed to patch software that has bugs, which intruders any use to exploit remotely. Also, updates are necessary to deploy new features and improve performance. Although many approaches have been proposed in the past to secure embedded systems (Kleidermacher & Kleidermacher, 2012), various aspects such as cost consideration and deployment scale make it very difficult to secure them.
According to Simmonds (2016), requirements for software updates include being secure to prevents the device from being hijacked, being robust to prevent an update from rendering the

VT OMNES VNVM SINT

device unusable, being atomic such that an update must be installed completely or not at all, having a failsafe feature such that one can deploy fall-back mode if all else fails, and being able to preserve persistent state.

Grau (2016) outlined some mechanisms and issues to be addressed by a scalable, secure software standard. These include software signing and validation (including the role of certificates, keys or other authentication mechanisms), use of transport protocols for distribution of the software, encryption of the software while being distributed, and scheduling and distribution of the software updates. For example, a staged rollout of software updates may be desirable so that any issues discovered in the field that escaped lab testing can be found and reported before most of the devices are updated.

Another mechanism is to enable the user to have control over software updates. For many devices, the user should be allowed some discretion over when updates occur so as not to interrupt their use of the device, or the device needs to be aware of how the update will impact its operation. This needs to be balanced with the need to ensure that updates occur. Finally, the role of the gateway in caching and distributing software updates needs to be defined for IoT devices that connect via a gateway.

2.6 Public Key Infrastructure

2.6.1 Overview

A Public Key Infrastructure (PKI) is the service framework that is used to support large-scale public key-based technologies (Salomaa, 2013). According to Salomaa (2013), PKI provides the base for security services such as **encryption**, which refers to the process of enciphering data into cipher text in such a way that only authorised parties can access it and those who are not authorised cannot, **authentication**, which refers to the process of ensuring that any messages received were sent from the perceived origin, and **nonrepudiation,** which the process of ensuring that the original source of a secured message cannot deny having produced the message.

A PKI makes use of certificates issued by certificate authorities (CA) to operate. A CA refers to a trusted third party that signs the public keys of entities in a PKI-based system. A certificate refers to a document, which binds together the name of the entity and its public key, which has been signed by the CA (Cooper, 2008).

A PKI facilitates highly scalable trust relationships (Rea, 2015). PKIs can be further scaled using a hierarchy of CAs with a root CA signing the identity certificates of subordinate CAs.

According to w3techs web technology surveys (2018), many vendors offer CA servers as a managed service or as an end-user product. These include VeriSign, Entrust Technologies, and GoDaddy. Organisations may also implement private PKIs using Microsoft Server or Open SSL.

PKI has been standardised to allow interoperability across a wide variety of applications and vendors (Gigovic, 2014). In the early 1990s, RSA Security Inc. devised and published a set of standards that are known as Public-Key Cryptography Standards (PKCS). While not true industry standards, as they were specified and maintained by a single organisation, several of the standards have been accepted into the standards track processes of recognised standards organisations (Kaliski & Redwood, 1993). Some of the PKCSs include:

1. PKCS #1: RSA Cryptography Standard
2. PKCS #3: D-H Key Agreement Standard
3. PKCS #5: Password-Based Cryptography Standard
4. PKCS #6: Extended-Certificate Syntax Standard
5. PKCS #7: Cryptographic Message Syntax Standard
6. PKCS #8: Private-Key Information Syntax Standard
7. PKCS #10: Certification Request Syntax Standard
8. PKCS #12: Personal Information Exchange Syntax Standard
9. PKCS #13: Elliptic Curve Cryptography Standard
10. PKCS #15: Cryptographic Token Information Format Standard

2.6.2 X.509

X.509 is an International Telecommunication Union Standardisation Sector (ITU-T) standard for PKI which specifies, among other things, the formats for identity certificates and certificate validation algorithms (Cooper, 2008). The Internet Engineering Task Force (IETF) formed the PKIX working group to support standards development of X.509.

Currently, digital identity certificates use the X.509 version 3 structure (Slagell, Bonilla, & Yurcik, 2006) with the following fields: Version, serial number, algorithm ID, issuer, validity (not before and not after), subject, subject public key info (public key algorithm and subject public key), issuer unique identifier (optional), subject unique identifier (optional), extensions (optional), certificate signature algorithm and certificate signature.

2.6.3 Public Key Infrastructure Operations

1. Certificate Enrollment

To obtain an identity certificate, a system administrator will enroll with the PKI (Cooper, 2008). The first step is to obtain the CA's identity certificate. The next step is to create a certificate signing request (CSR), following PKCS #10. The CSR contains the identity information that is associated with the enrolling system, which can include data such as the system name, the organisation to which the system belongs, and location information (Vacca, 2013). Most importantly, the enrolling system's public key is included with the CSR. Depending on the circumstance, the CA administrator may need to contact the enroller and verify the data before the request can be approved. If the request is approved, the CA will take the identity data from the CSR, and add in the CA-specified data, such as the certificate serial number, the validity dates, and the signature algorithm, to complete the X.509 v3 certificate structure. It will then sign the certificate by hashing the certificate data and encrypting the hash with its private key. The signed certificate is then made available to the enrolling system. Figure 2.1 illustrates this process.

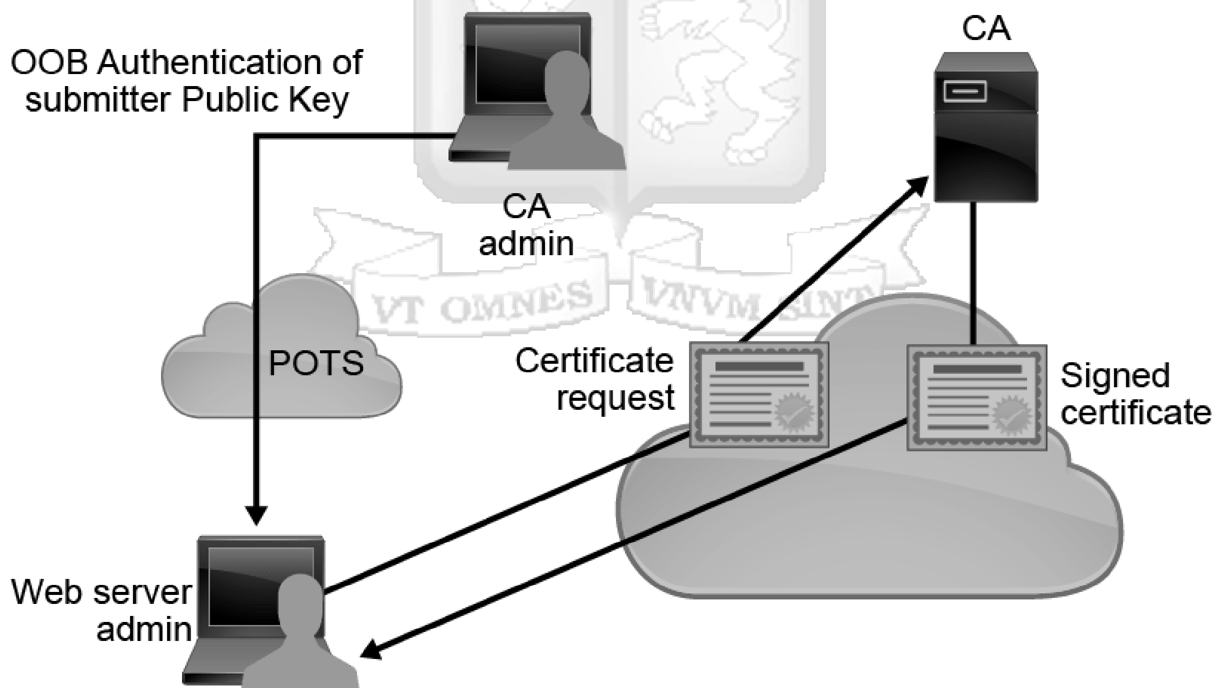


Figure 2.1 Certificate Enrolment

Source: Complementar, (2016)

2. Authentication Using Certificates

Systems that need to validate the identity certificate of other systems will have the root CA certificate (Ford, 1998). They will use the CA's public key to validate the signature on any certificate they receive. It is also important to understand that the certificate does not so much identify the entity of the peer. It only identifies the valid public key of the peer (Wang & Zhang, 2010). To be sure that the peer is the entity that is identified in the certificate, a system must challenge the peer to prove that it has the private key that is associated with the validated public key. For example, a message can be encrypted with the validated public key and sent to the peer. If the peer can successfully decrypt the message, then the peer must have the associated private key and is therefore the system that is identified by the digital certificate. Figure 2.2 illustrates this process.

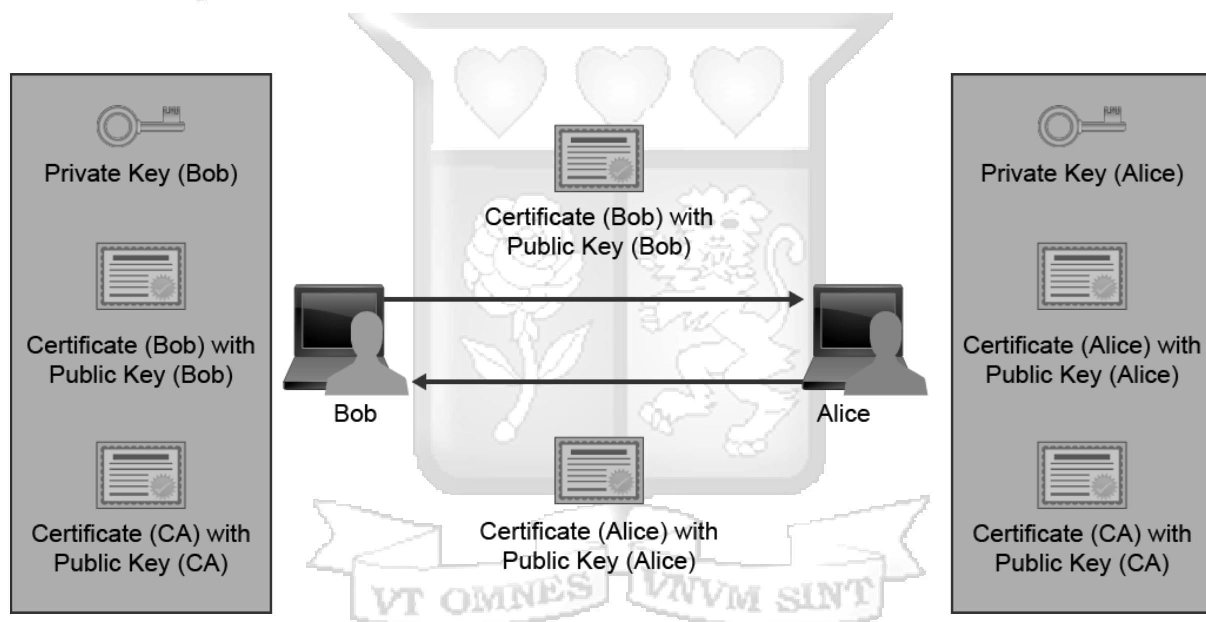


Figure 2.2 Authentication Using Certificates

Source: Complementar, (2016)

3. Certificate Revocation

Digital certificates can be revoked if keys are thought to be compromised, or if the business use of the certificate calls for revocation (Jayaprakash, 2015). If keys are thought to be compromised, generating new keys forces the creation of a new digital certificate, rendering the old certificate invalid and a candidate for revocation.

On the other hand, a consultant might obtain a digital certificate for VPN access into the corporate network only during the contract. During the SSL handshake a server invokes OCSP/CRL protocols to verify that the client's X509 Certificate is not revoked by its issuer.

OCSP and CRL must to make a http call to servers at CA to do the verification (Palihakkara, 2014). The CA servers respond with the revocation status of the certificates.

The SSL connection cannot be established any further if the response indicate that the certificates are revoked. Figure 2.3 illustrates this process.

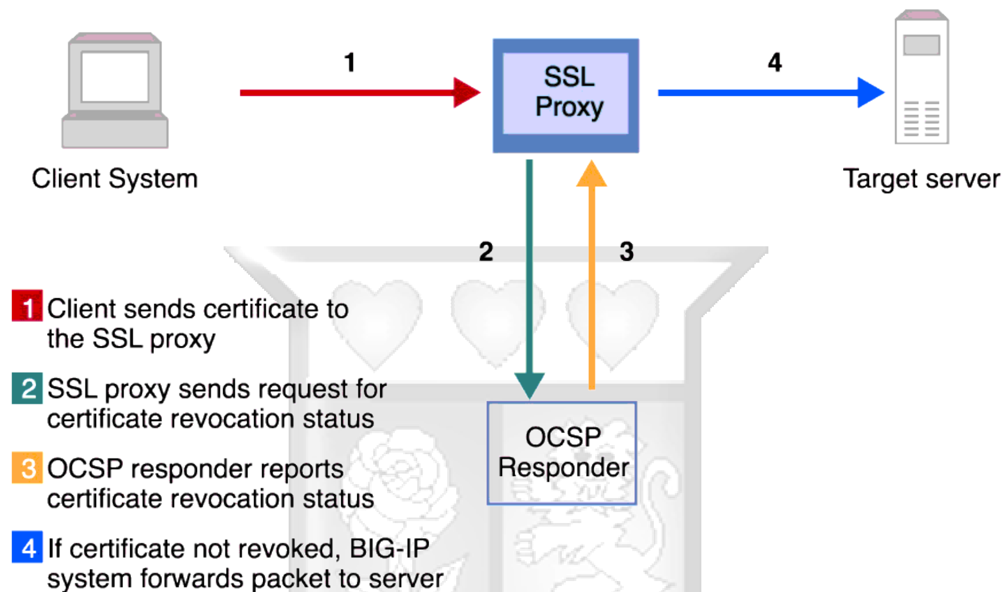


Figure 2.3 Certificate Revocation

Source: Palihakkara, (2014)

2.7 Secure Socket Layer / Transport Layer Security (SSL / TLS)

Secure Socket Layer (SSL) is an encryption technology for the web to provide secure transactions, such as the transmission of credit card numbers for e-commerce (Freier, Karlton, & Kocher, 1996). Transport Layer Security (TLS) is a successor to SSL. SSL was developed by Netscape in the 1990s to provide secure transactions between web browsers and web servers in support of commerce over the Internet. SSL became a de facto standard, but it has since been made obsolete by TLS which is standardised by the IETF. TLS version 1.0 was defined in RFC 2246 in 1999 and provided a standards-based upgrade to SSL version 3.0. TLS continues to evolve with TLS 1.3 in draft as of February 2015. Modern systems implement TLS.

TLS uses PKI to authenticate peer systems and public key cryptography to facilitate the exchange of session keys that are used to encrypt the SSL session (Turner, 2014). Many applications use TLS to provide authentication and encryption. The most widely used

application is HTTPS (Rescorla, 2000). Other well-known applications that were using poor authentication and no encryption were modified to be transported within TLS. Examples include SMTP (Hoffman, 2002), LDAP (Hodges, Morgan, & Wahl, 2000), and POP3 (Newman, 1999).

Figure 2.4 depicts the steps that are taken in the negotiation of a new TLS connection between a web browser and a web server.

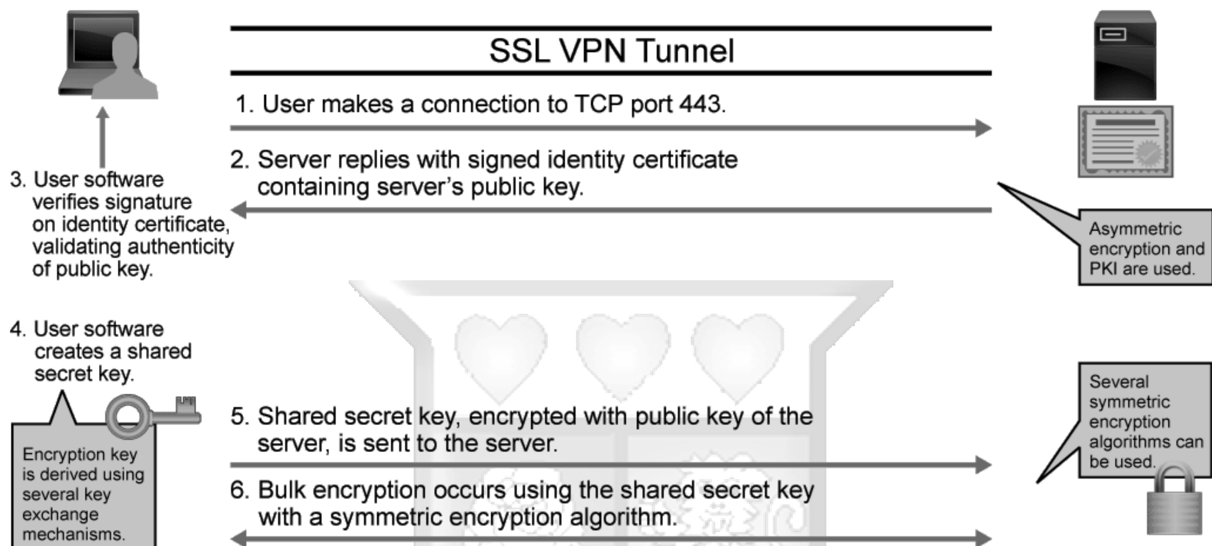


Figure 2.4 Cryptographic architecture of SSL and TLS

Source: Rescorla, (2000)

2.8 Cipher Suites

A cipher suite is a set of algorithms that help secure a network connection that uses SSL/TLS (Steiner et al., 2001). An SSL/TLS cipher suite is used to define a set of cryptographic algorithms including the authentication and key exchange algorithms (such as RSA), encryption algorithm (such as Advanced Encryption Standard - AES), message authentication code algorithm (such as SHA), and the PRF. The cipher suites are described in RFC 5288 and RFC 5289.

When a TLS connection is established, a TLS handshake occurs. Within the TLS handshake, a client hello and a server hello message are passed. First, the client sends a list of the cipher suites that it supports, in order of preference. Then the server replies with the cipher suite that it has selected from the client's list.

When a TLS connection is established, a TLS handshake occurs. Within the TLS handshake, a client hello and a server hello message are passed. First, the client sends a list of the cipher suites that it supports, in order of preference. Then the server replies with the cipher suite that

it has selected from the client's list (Polk, McKay, & Chokhani, 2014). Figure 2.5 below depicts part of a packet capture (PCAP) with a TLS client that is presenting the set of cipher suites that it can support to the server.

```
Cipher Suites (26 suites)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x009f)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x009e)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
```

Figure 2.5 PCAP listing the cipher suites that client supports

Source: Nath, (2015)

The structure and use of the cipher suite concept are defined in the documents that define the protocol (RFC 5246 for TLS version 1.2). This RFC defines mandatory cipher suites that must be implemented by all TLS-compliant applications. An example of cipher suite is TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384. It uses the Elliptic Curve Diffie-Hellman exchange (ECDHE) and Elliptic Curve Digital Signature Algorithms (ECDSAs) for authentication and key exchange, instead of using RSA.

CDHE_ECDSA is the authentication and key exchange algorithms. ECDHE_ECDSA is used to determine how the client and server will authenticate and establish the pre-master key during the TLS handshake. In this case, both the client and the server will derive the identical pre-master key using the Diffie-Hellman (DH) parameters (sent in the additional ServerKeyExchange message). The pre-master key is then used to derive the master key and the session-specific keys. With DH key exchanges, for the client to authenticate the server, the server will sign the DH parameters that are contained in ServerKeyExchange message with the server’s private key. The client verifies the signature with the server's public key in the server's certificate. Only if the signature is valid, the client will proceed with the TLS handshake.

AES_256_GCM is the bulk encryption algorithm. Galois Counter Mode (GCM) is a mode of operation for an authenticated symmetric key cryptographic block ciphers that has been widely adopted because of its efficiency and performance. GCM is an authenticated encryption algorithm that is designed to provide both data authenticity and confidentiality.

SHA-384 is used for the pseudorandom function. Since an authenticated encryption mode (GCM) is used, the messages neither have nor require a message authentication code.
The pseudorandom function is used to generate the keying materials that are used during the TLS session.

2.9 Systems Using PKI in Securing Updates in Other Cases

Modio

This is a software stack developed by Modio AB. It is based around embedded Linux and IoT, focusing on robustness, security, and integration tools. The stack consists of distribution and upgrades, certificate management, serial communications, data logging and frontend visualisations.
All Modio AB embedded devices run Modio Embedded OS, that incorporates the features ensuring secure, automatic and continuous updates of embedded and IoT systems (Modio, 2018). Caramel is Modio's CA system that manages TLS certificates, updates and refreshing.

2.10 Conclusion

Experts recommend certificate-based methods to secure IoT devices because certificates support proper security measures like the implementation of multi-factor authentication. More specifically, TLS supports two-way certificate-based authentication. For instance, device-to-server, or device-to-device authentication.
Existing solutions have been Operating System depended and not meant to integrate with custom applications in different single board computers (Modio, 2018).
Developers need a light weight tool that they can integrate with their application scripts for secure updates. This has been an ignored area and its implementation will help improve the security of IoT. This dissertation will make use of research by Grau (2016) and Gigovic (2014) as guidelines in coming up with the software updates plugin.

Chapter 3 Methodology

3.1 Introduction

This chapter explains the research methodology that was used. Focus falls on detailing the approach taken in doing the research that influenced the development of the solution.

3.2 Research Design

The selected approach is partly theoretical in nature through literature review and the development of a tool used to validate the proposed solution. A review of relevant research documents on embedded systems design and requirements, software update techniques and the Public Key Infrastructure (PKI) was sufficient to answer the first two objectives of this study. The theoretical research was intended to provide a deeper theoretical understanding of the research area. This formed a basis for the development and validation of the proposed solution.

3.3 System Development Methodology

This study sought to design, develop and test a tool that will deliver software updates to embedded devices in a secure way. The research used Agile Development Methodology, which provided opportunities to assess the dissertation progress and direction throughout the development lifecycle.

According to a Manifesto for Agile Software Development signed by Beck et al. (2001), agile development is a better way of developing software that promotes sustainable development. Scrum is the most widely used lightweight process framework for agile development in completing complex projects and innovative scope of work. This is achieved through iterations, coming up with a potentially stable product finally, as illustrated in figure 3.1 below:

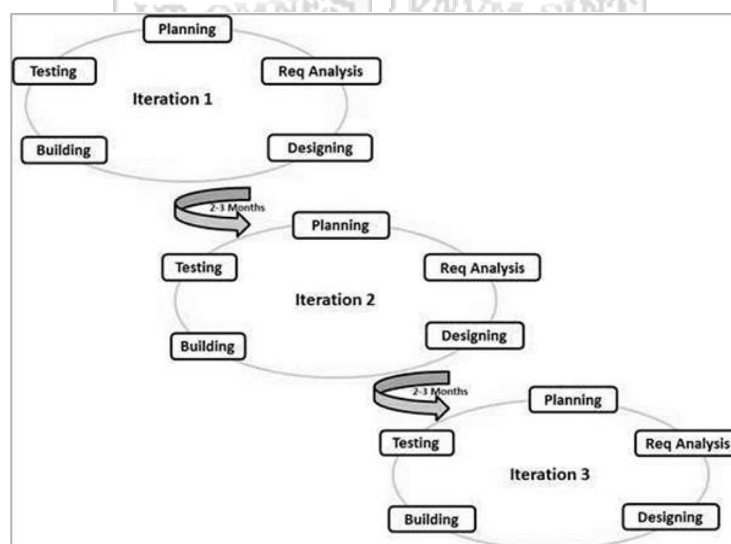


Figure 3.1 Agile Development Methodology

Source: Yuan, Han, & Hu, (2008)

3.3.1 Planning

The study took necessary steps to ensure that the processes to be undertaken are outlined clearly to achieve the research objectives. This involved how the problem was addressed in in different establishments and to come up with new ways of solving the problem.

3.3.2 Requirements Analysis

In this step, data collected through analysing systems in planning step was analysed. This helped in formulating the requirements of the systems that was built. The stage also included the creation of the system architecture. The design step involved coming up with the application designs based on the requirements gathered.

3.3.3 System Design

Functional Oriented Design (FOD) techniques were used to refine the functional requirements identified during system analysis and to decompose the design into sets of interacting units where each unit has clearly defined functions.

Unified Modelling Language (UML) was used to specify, visualise, construct, and document the artifacts of the system.

Use Case diagrams were used to show required usages of a system under design or analysis and to capture what the system is supposed to do.

System Sequence Diagrams were used to illustrate a scenario of a use case, the events that external actors generate, their order, and possible inter-system event.

These artifacts were developed by use of Draw.io tool.

3.3.4 Implementation

The development tools chosen for this study have different component which help in achieving certain functionalities. Python programming language was chosen because it provided several libraries which can be used to achieve low level functionalities and has many resources supported by a large community. Shell script was used to automate the tool functionalities, run cron jobs and execute Python scripts.

PKI module followed the National Institute of Standards and Technology's (NIST) guidelines for the selection, configuration, and use of TLS implementations (NIST Special Publication 800-52). Transport Layer Security (TLS) provides mechanisms to protect data during electronic dissemination across the Internet. This Special Publication provides guidance to the selection and configuration of TLS protocol implementations while making effective use of Federal Information Processing Standards (FIPS) and NIST-recommended cryptographic algorithms (Chernick et al., 2005).

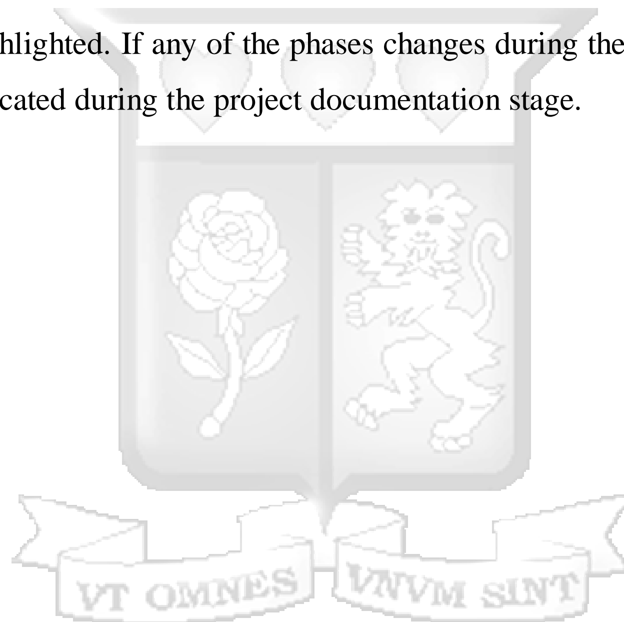
Git was used as the version control system (VCS) for recording and tracking changes to software code over time so that we can recall specific versions later (Chacon & Straub, 2014).

3.3.5 Testing

System testing was guided by open web application security project (OWASP) testing guide version 4. The tests included testing for functional requirements, sensitive data transmitted in clear-text, testing for weak SSL/TLS ciphers / protocols / keys vulnerabilities, and testing server SSL certificate validity.

3.4 Conclusion

The chapter highlighted the methodology that was adopted in undertaking the study. The methods for data collection and analysis are proposed to guide the researcher in making deductions from the evidence to be collected and thus answer the research questions. The tools employed are also highlighted. If any of the phases changes during the implementation stage, the details will be indicated during the project documentation stage.



Chapter 4 System Design and Architecture

4.1 Overview

This chapter presents a detailed view of how the auto update plugin was designed. The researcher used the use case diagram, data flow diagrams, system sequence diagram and data modelling to illustrate different aspects of the auto update plugin.

4.2 System Architecture

The system is based on the following architecture:

4.2.1 General Architecture

The general architecture of the system is a client server model is ideal in a situation where multiple single board devices need to be updated regularly. A trust model is used whereby the server and client devices register with Certificate Authority (CA) server, which verifies the device identity. The trust model follows the Public Key Infrastructure x.509 standard.

An overview of how the application works is shown in figure 4.1. The client establishes a secure connection to the server using Transport Layer Security (TLS) to avoid security breaches such as Man in the Middle Attack (MITM). It checks if the server has software updates using sha256 hashes at predefined time intervals. If there are updates, the client downloads the updates (not the whole application). This ensures efficient bandwidth usage by avoiding downloading the whole application code.

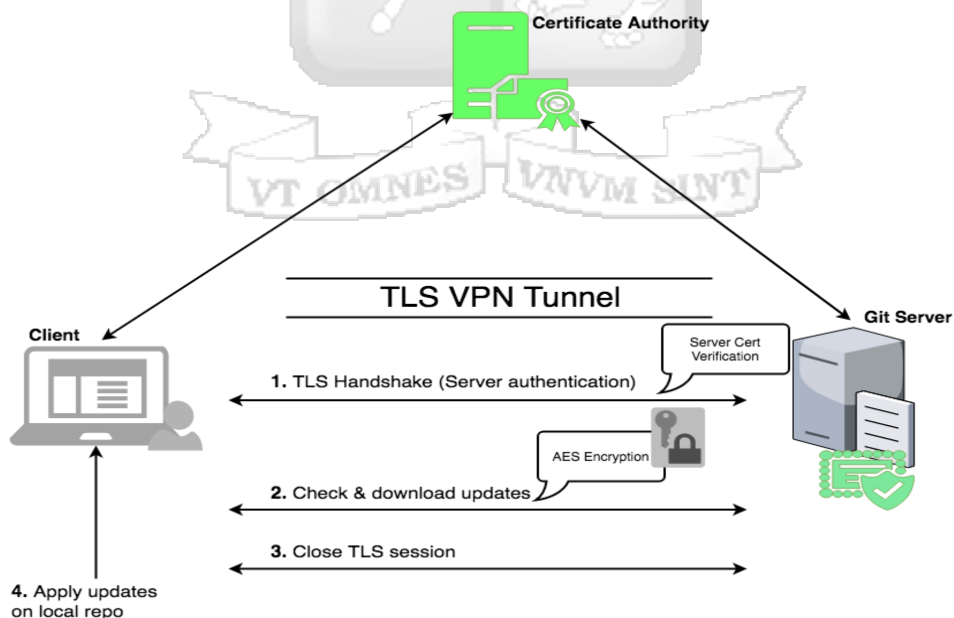


Figure 4.1 Auto Update Plugin System Architecture

Step 1: TLS Handshake (Server Authentication)

This is the first step that takes place after the update.sh script is triggered by crontab daemon. The main goal of this step is to authenticate the remote server and open a secure TLS connection between the client and the server. Subsection 4.2.2 discusses this process in detail.

Step 2: Check and Download Updates

The second step involves two processes, client checking if there are updates in the server, and if available the server transmits the updates only to the client. A version control system is used to ensure that only the updates are transmitted to ensure efficient use of bandwidth. Subsection 4.2.3 discusses this process in detail.

Step 3: Close TLS Session

The third step involves termination of the client – server connection. This is done immediately after the updates are successfully delivered the client. The client sends a close_notify message to server notifying it that it will not send any more requests during that session. Any data received after a closure alert is ignored.

Step 4: Apply Updates on Local Repository

The fourth step is to update the intended software. The fetched updates are merged with the application code using git merge utility.

4.2.2 Authentication

The client authenticates the server prior to checking and downloading the updates. This is indented to ensure that rogue servers cannot update the client devices. Figure 4.2 illustrates the TLS handshake process that authenticates the server. In this model, only the server certificate is verified. We do not need to authenticate the client since the server can only be accessed by trusted devices in a local area network (LAN) or a VPN. This is a good model for the resource constrained single board computers.

The certificate authority (CA) is locally implemented to sign server certificate, that is provided for verification during the TLS handshake. The devices negotiate supported cipher suites and agrees on which one to use. The process ends after the devices calculate a symmetric key that will be used to encrypt further communication between the client and the server.

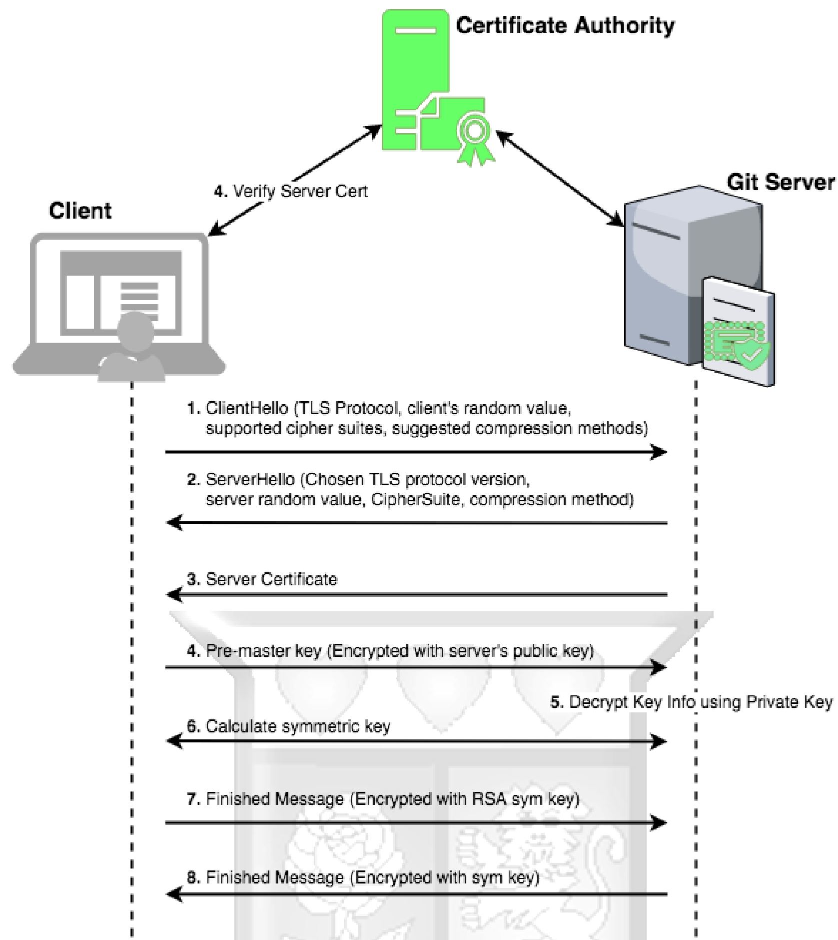


Figure 4.2 TLS Authentication - Handshake Protocol

Step 1: ClientHello Message

The client first sends ClientHello Message when it first connects to a server. The ClientHello message includes a variable-length session identifier, cipher suite list, and a list of compression algorithms supported by the client, ordered according to the client's preference. Structure of this message is indicated in appendix A.

After sending the ClientHello message, the client waits for a ServerHello message. Any handshake message returned by the server, except for a HelloRequest, is treated as a fatal error.

Step 2: ServerHello Message

The server will send this message in response to a ClientHello message when it was able to find an acceptable set of algorithms. The ServerHello Message contains server_version which is the lower of that suggested by the client in the client hello and the highest supported by the server, server random, session_id which is the identity of the session corresponding to this connection, cipher_suite which is the single cipher suite selected by the server from the list in ClientHello.cipher_suites, compression_method which is the single compression algorithm

selected by the server from the list in ClientHello.compression_methods and a list of extensions. Structure of this message is indicated in appendix A.

Step 3: Server Certificate

This message conveys the server's certificate chain to the client. The certificate must be appropriate for the negotiated cipher suite's key exchange algorithm and any negotiated extensions. Structure of this message is indicated in appendix A.

Step 4: Pre-master Key

In this step, the client sends the Client Key Exchange Message that sets the premaster secret, either by direct transmission of the RSA-encrypted secret or by the transmission of Diffie-Hellman parameters that will allow each side to agree upon the same premaster secret. Structure of this message is indicated in appendix A.

Step 5: Pre-master Key Decryption

If RSA is being used for key agreement and authentication, the server uses its private key to decrypt the 48-byte premaster secret.

Step 6: Calculating Symmetric Key

The symmetric key is the master secret that will be used to encrypt and decrypt communication between the client and the server. For all key exchange methods, the same algorithm is used to convert the pre_master_secret into the master_secret. Structure of this operation is indicated in appendix A.

Step 7: Exchanging Finished Messages

Handshake Finalization Messages are exchanged between the client and server to indicate successful TLS connection. Structure of this message is indicated in appendix A.

4.2.3 Update Checking and Delivery

After a predefined period, the client checks if there are any updates at the server. The updates are implemented in a git server. Git is a version control framework that ensures changes to code are tracked and efficiently coordinate work on the code amongst multiple developers. The git server has a remote repository that is a copy of the client software. It is updated by the developers if need be, and such changes are expected to reflect on client devices. To support this, the client device has a local repository that is a replica of the remote repository. The client periodically checks if there are any changes in the remote repository and downloads them by using the git fetch command. It then applies the updates to the running application through git merge command. Figure 4.3 illustrates this model.

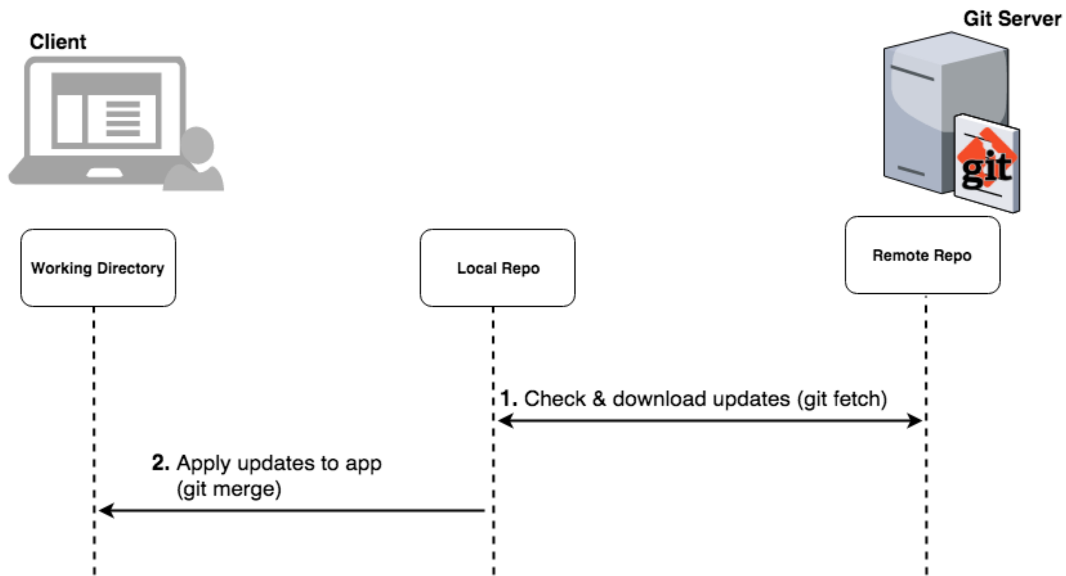


Figure 4.3 Update Checking and Delivery

4.3 Use Case

Figure 4.4 is a use case diagram that illustrates the major interactions taking place between the various subsystems and actors in the developed prototype.

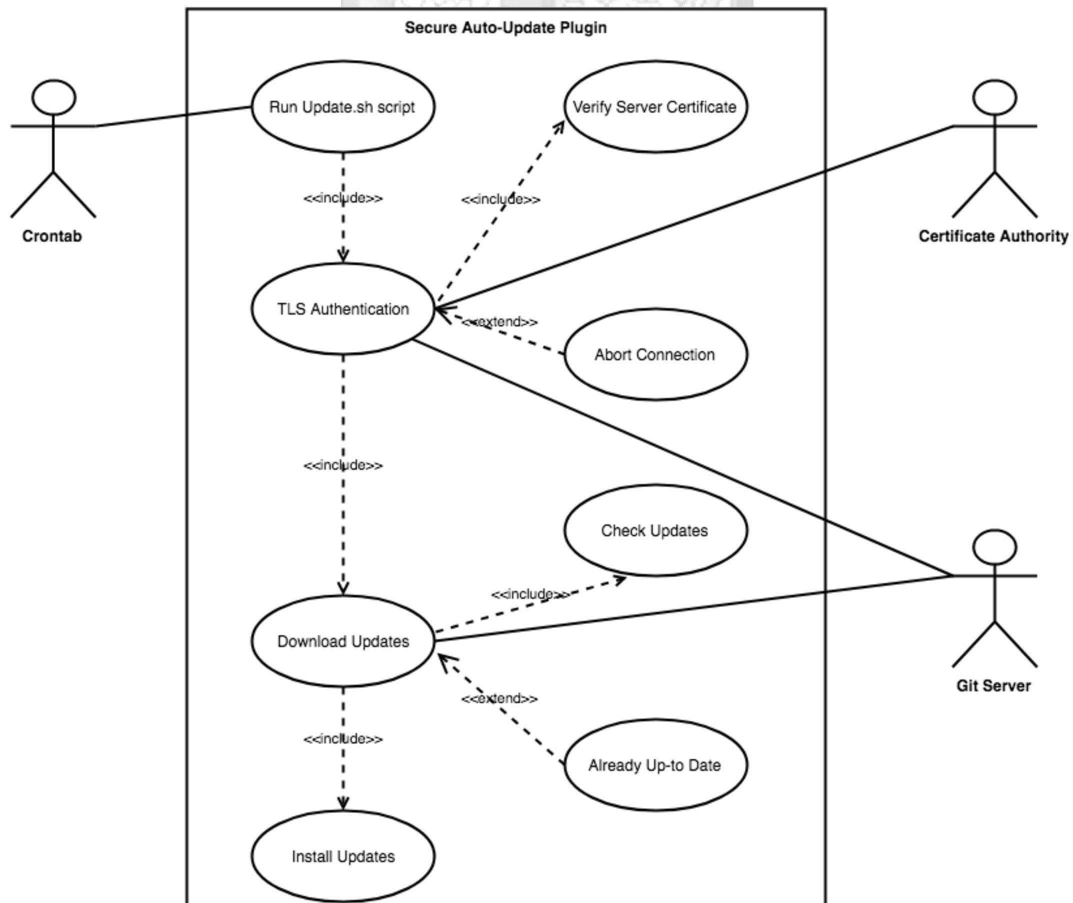


Figure 4.4 Use Case Diagram

4.3.1 Use Case 1 Run Update.sh script

Table 4.1 Run Update.sh Script Use Case

Use Case Name:	Run Update.sh script
Description:	This use case contains instructions to execute TLS authentication scripts, updates checking, download and installation to the local software.
Primary Actor:	Crontab
Secondary Actor:	None
Include Use Cases:	TLS Authentication Download Updates Install Updates
Extend Use Cases:	None
Preconditions:	Predefined time interval must have lapsed
Post Conditions:	The system establishes a secure connection to the git server
Main Flow:	The system triggers TLS Authentication use case and waits for it to complete The system triggers Download Updates use case and waits for updates to be downloaded The system merges updates with the software
Alternative Flows:	Failed authentication The system restarts TLS Authentication Maximum 3 attempts allowed The use case ends

VT OMNES VNVM SINT

4.3.2 Use Case 2 TLS Authentication

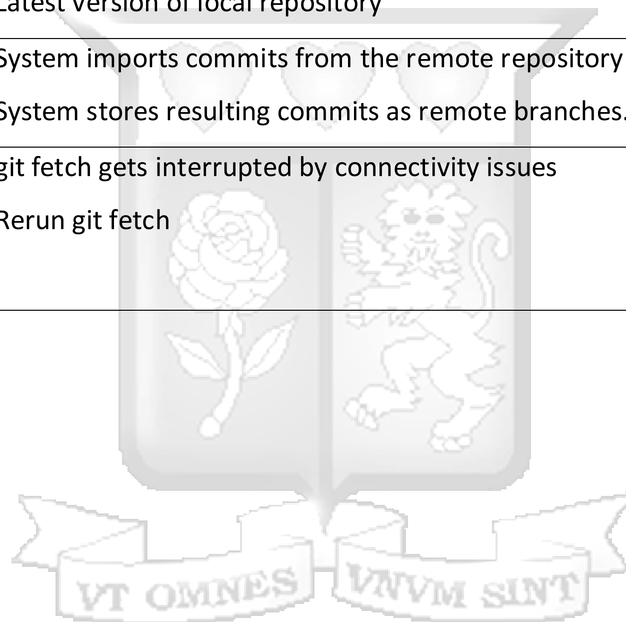
Table 4.2 TLS Authentication Use Case

Use Case Name:	TLS Authentication
Description:	This use case contains procedures for the system to establish a secure connection to the git server using Transport Layer Security (TLS). The certificate authority (CA) signs server certificate, that is provided for verification during the TLS handshake.
Primary Actor:	None
Secondary Actor:	Certificate Authority Git Server
Include Use Cases:	Verify Server Certificate
Extend Use Cases:	Abort Connection
Preconditions:	Server must have a certificate
Post Conditions:	The system establishes a secure TLS connection to the git server
Main Flow:	System sends ClientHello message to server that includes TLS Protocol, client's random value, supported cipher suites, suggested compression methods. Server replies with ServerHello message that includes chosen TLS protocol version, server random value, CipherSuite, and compression method. Server sends its certificate to system (client). Client verifies server certificate using the certificate authority. System encrypts pre-master key with server's public key and sends it to server. Server decrypts the pre-master key using its private key. Both system and server calculate a symmetric key using the pre-master key. Both system and server exchange Finished Messages encrypted with symmetric key.
Alternative Flows:	Invalid certificate Trigger Abort Connection use case

4.3.3 Use Case 3 Download Updates

Table 4.3 Download Updates Use Case

Use Case Name:	Download Updates
Description:	This use case downloads the changes from remote repository in authenticated git server to the local repository.
Primary Actor:	None
Secondary Actor:	Git Server
Include Use Cases:	Check Updates
Extend Use Cases:	Already up-to Date
Preconditions:	Established secure connection between system/client and server
Post Conditions:	Latest version of local repository
Main Flow:	System imports commits from the remote repository into local repository. System stores resulting commits as remote branches.
Alternative Flows:	git fetch gets interrupted by connectivity issues Rerun git fetch



4.4 Sequence Diagram

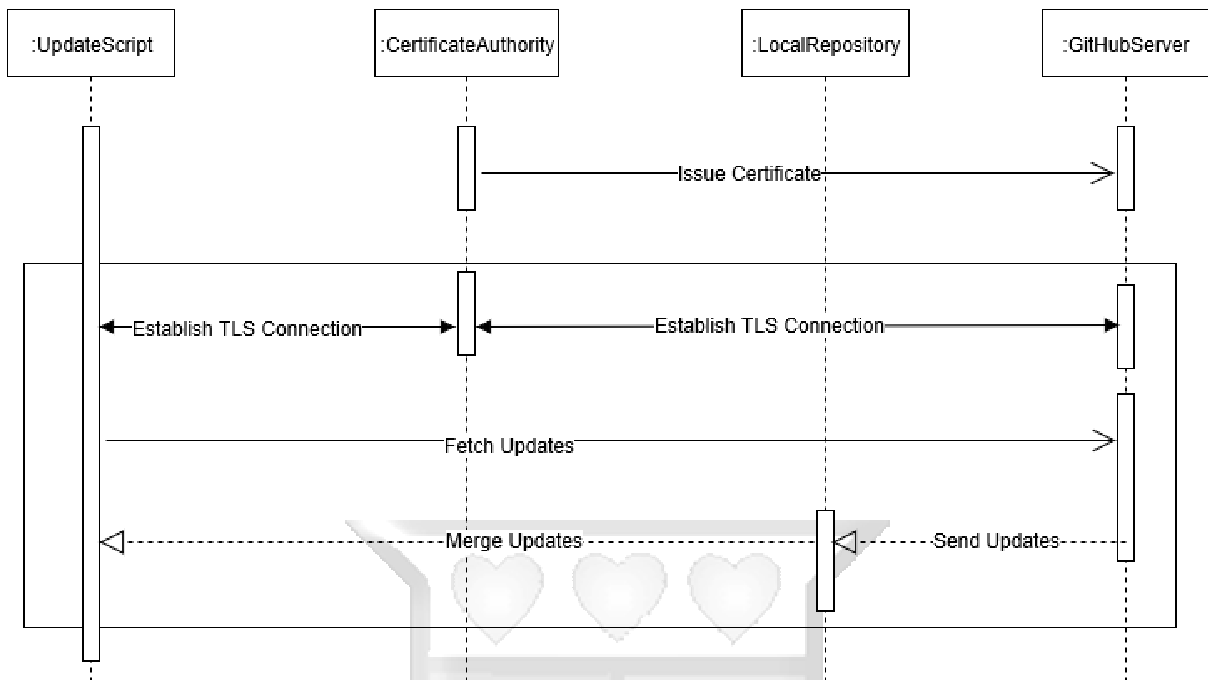


Figure 4.5 Sequence Diagram

Issue Certificate: A trusted global Certificate Authority signs and issues a public certificate to the GitHub server. The client already knows this CA since it is preinstalled in the operating system by the manufacturer. The client will validate this certificate every time during the TLS handshake.

Establish TLS Connection: The UpdateScript starts by establishing a secure communication channel between the system and the GitHub server. OpenSSL commands are used. This is achieved through TLS handshake discussed in section 4.2.2.

Fetch Updates: The UpdateScript checks for updates in the Github server. If there are no updates, the connection is closed up-to the next scheduled update check time.

Send Updates: If updates are available, the GitHub server transmits only the changes to the local repository. These are send in encrypted format such that privacy and integrity are maintained.

Merge Updates: The local repository is checked for consistencies, and if everything matches the remote repository, changes are merged to the indented code.

Chapter 5 System Implementation and Testing

5.1 Overview

This chapter discusses the system implementation procedures. This includes the implementation environment, main functions of the prototype and system testing which describes the kind of tests carried out and the intended results. The relevant screenshots of the prototype are provided.

5.2 Software Environment

The system is a network application using the client-server model. It is implemented on Raspbian GNU/Linux 9 (stretch) which is a Debian distribution. The system was built in different phases using several programming / scripting languages and software that include:

5.2.1 Bash Programming

Bash Programming is the art of creating shell scripts designed to be run by the Unix shell, a command-line interpreter. The system uses scripts to set up the environment, run processes, and audit the system through log maintenance.

5.2.2 OpenSSL

OpenSSL is an open source implementation of the SSL/TLS protocol. It provides a command line application to perform a wide variety of cryptography tasks, such as creating and handling certificates and related files.

5.2.3 Python Programming Language

Python is an interpreted high-level programming language for general-purpose programming. It has low-level capabilities which makes it faster than other high-level languages (Monk, 2015). Python was selected because it works well in small devices that have limited computational power and memory.

5.2.4 Wireshark

Wireshark is a free and open source packet analyser that enables visibility of network activity at packet level. In this research, it is used to capture traffic between the Raspberry Pi and the GitHub Server and inspect the packets' Secure Sockets Layer header.

5.2.5 Nmap

Nmap free security scanner, port scanner, and network exploration tool that was originally written by Gordon Lyon. It is compatible with all major operating systems. It has inbuilt scripts used for specialised scanning and vulnerability analysis. In this research, it is used to test cryptographic suites and TLS implementation.

VT OMNES VNVM SINT

5.2.6 cURL

cURL is a command line tool and library for transferring data with URLs using various protocols. In this research, it is used to test if the server supports unsecure communication via HTTP.

5.2.7 ShellCheck

ShellCheck is a GPLv3 static analysis tool that gives warnings and suggestions for bash/sh shell scripts. In this research, it is used for code analysis to point out and clarify typical syntax issues that cause a shell to give cryptic error messages, to point out and clarify typical intermediate level semantic problems that cause a shell to behave strangely and counter-intuitively and to point out subtle caveats, corner cases and pitfalls that may cause an advanced user's otherwise working script to fail under future circumstances.

5.3 Hardware Environment

Central Processing Unit:	4× ARM Cortex-A53, 1.2GHz
Random Access Memory:	1GB LPDDR2 (900 MHz)
Network Interface:	10/100 Ethernet or 2.4GHz 802.11n wireless

5.4 Set-up and Configuration

This is the only task where the end user interacts with the system. Several shell scripts are used to configure the auto-update plugin. To make this user friendly, Whiptail, a program that allows shell scripts to display dialog boxes to the user for informational purposes is used to get input from the user in a friendly way. This information includes the URL to remote repository, the local software to update, and the frequency of updates. The screenshots below illustrate this process:

The welcome message shown in appendix B is displayed upon running the installer. The user should press return button to proceed.

The user is prompted to input the local directory of the software to be updated as sown in appendix B. If it is not available, it will be created and cloned from the cloud.

The user is prompted to select the frequency of update checking and downloading as shown in appendix B.

Finally, the user is prompted to enter the URL to the remote git repository from which the updated will be pulled from as shown in appendix B.

Appendix B indicates the installation and configuration summary of the auto update plugin.

VT OMNES VNVM SINT

5.5 Functions of the Prototype

5.5.1 Opening Secure TLS Connection

The software is designed to communicate with the remote server through a secure TLS tunnel. This is done before update checking and downloading. The client, a Raspberry Pi single board computer (ipv4 address: 10.9.41.47) exchanges TLS handshake messages with the server, a git server (ipv4 address: 192.30.41.47) to establish a secure connection.

Line 7 in code snippet in appendix C shows the openssl command to create a TLS connection to github server via the standard port 443.

After the TCP three-way handshake, the TLS handshake begins by Client Hello message from the Raspberry Pi specifying, among other information, 57 cipher suites that it can support. Figure 5.1 shows a Wireshark packet capture with the Client Hello TLS packet from the client (IP address 10.9.41.47) to the GitHub server (IP address 192.30.253.113). The Secure Socket Layer record shows the information about the handshake protocol Client Hello message. The client supports TLS 1.2 and can accept the listed 57 cipher suites.

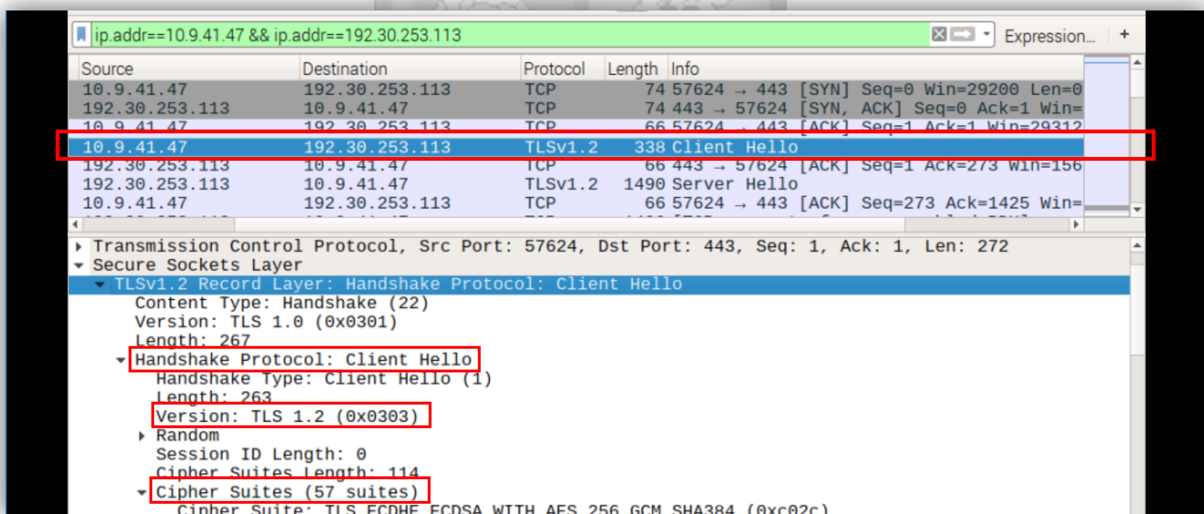


Figure 5.1 Client Hello Packet

The next step of the TLSv1.2 Record Layer Handshake Protocol is the Server Hello from the server (IP address 192.30.253.113) to the client (IP address 10.9.41.47) as indicated in the Wireshark packet capture shown in figure 5.2 below. Some notable items in the Server Hello message include the TLS version (TLS 1.2), Cipher Suite that will be used for the data exchange (TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256) and some extensions.

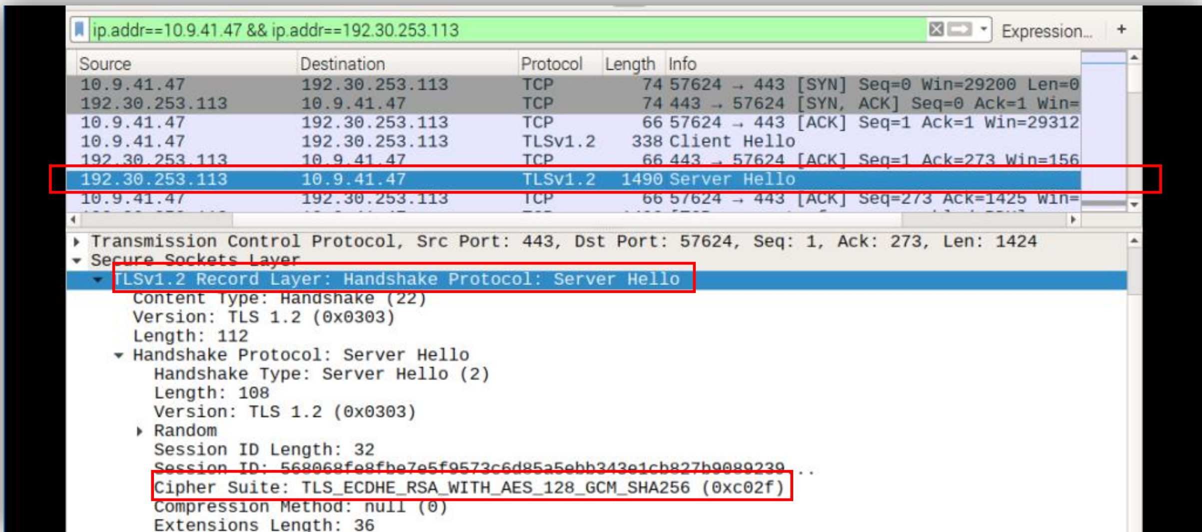


Figure 5.2 Server Hello Packet

The server (IP address 192.30.253.113) then sends its certificate signed by a trusted Certificate Authority to the client (IP address 10.9.41.47) as indicated in the Wireshark packet capture shown in figure 5.3 below. The TLS record layer indicates a certificate. The handshake protocol supports EC Diffie-Hellman server key exchange algorithm. The certificate is signed by sha256 algorithm with RSA encryption. The certificate also contains the Public Key of the server that will be used to encrypt keying material.

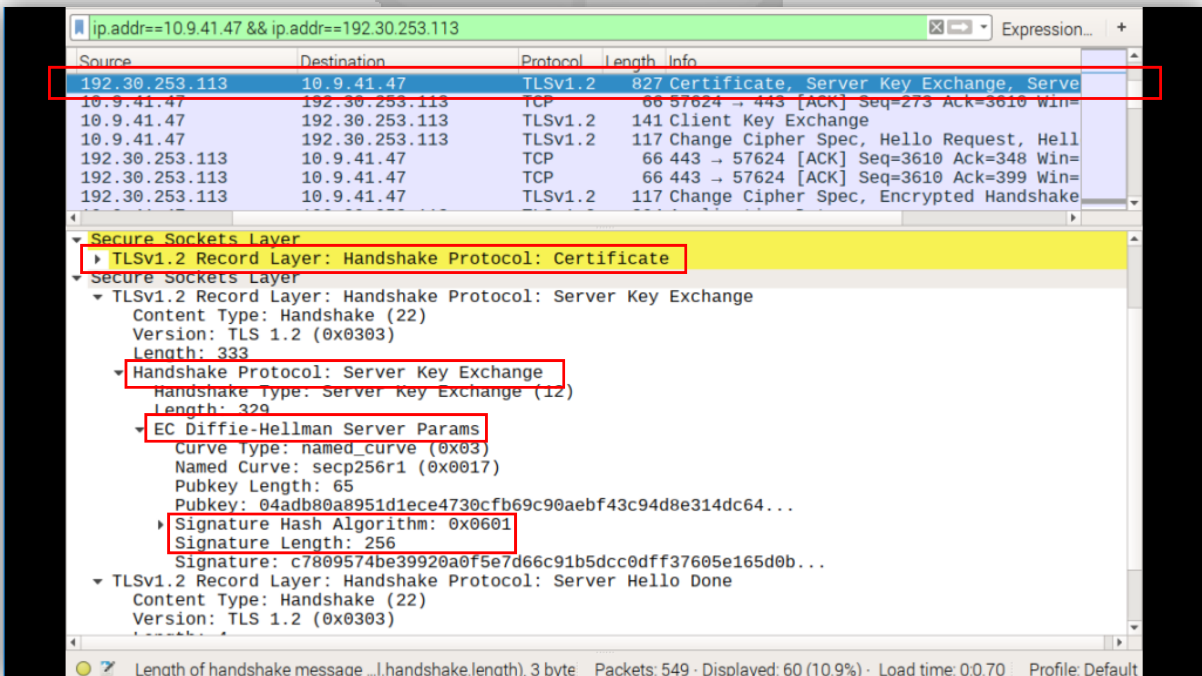


Figure 5.3 Server Certificate Exchange Packet

This certificate not only ensures the identity of the server, but also proves it. Its issued to github.com by DigiCert SHA2 Extended Validation Server CA (DigiCert Inc) as shown in appendix C. The Raspberry Pi is configured with trusted certificates which are globally maintained. These are in "/etc/ssl/cert" directory. Appendix C illustrates this.

All application data exchanged is encrypted (HTTP via TLS), including alerts. After the Raspberry Pi fetches updates, the connection is closed using TCP FIN messages as shown in appendix D.

5.5.2 Checking and Downloading and Merging Updates from Remote Repository

Updates are fetched from a remote repository and integrated with a local branch. The utility 'git pull' is shorthand for 'git fetch' followed by 'git merge FETCH_HEAD' utilities. Therefore, 'git pull' runs 'git fetch' with the given parameters and calls 'git merge' to merge the retrieved branch heads into the current branch.

Line 7 of the code snippet in Appendix D is a 'git pull' command that includes a fetch utility to check for and download updates from the remote repository.

The updates are send from server in encrypted data packets, as illustrated in Appendix D. If there are no updates, nothing is downloaded. This is a bandwidth saving feature of the git pull command. Appendix D summarises these operations.

5.5.3 Updating the Local Repository

The git pull command contains a merge utility that incorporates the downloaded updates into the local software repository so that it is in the latest version.

5.6 System Testing

Several tests were done on the auto-update plugin to ensure that it is meets its functional requirements and is secure. Section 2.3.2 outlines the top 10 IoT vulnerabilities which must be addressed for IoT applications and equipment. Due the headless nature of the auto-update plugin, some vulnerabilities such as those relating to user interface and stored data are not applicable.

The auto update plugin is a network application that transmits sensitive data through the network. This data must be protected. Despite the availability of high grade ciphers, some misconfigurations in the server can be exploited to force the use of weak ciphers, creating a vulnerability that attackers can exploit to gain access to the intended secure communication channel or cause a Denial of Service (DoS). The testing procedures focused on system

VT OMNES VNVM SINT

functionalities, TLS ciphers and transport layer protection. Table 5.1 shows in detail the manual test plan.

5.6.1 Test Plan

Table 5.1 Test Plan

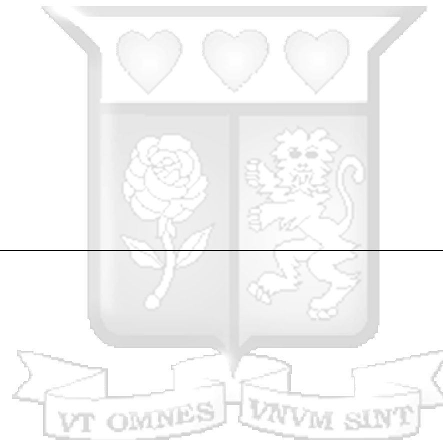
Item	Value
Objectives	To define the tools to be used throughout the testing process. To communicate to the responsible parties the items to be tested, set expectations around schedule. To define environmental needs. To define how the tests will be conducted.
Test Items	Functional Testing Structural Testing The Transport Layer Security Algorithms. The GitHub server connection security.
Features to be Tested	Data encryption. SSL/TLS ciphers. SSL/TLS certificates. Source code.
Approach	Manual tests will be carried out on all the modules of the application by different users. Test Cases were the developed with step by step procedures for testing the features. The sample target application to be updated by the system will be availed to the testers via a public GitHub account. Users participating in the testing will fill in prepared manual test cases with the results of the testing. Each user is required to repeat testing each module at least three times using the same datasets to ensure the results are repeatable.
Item Pass/Fail Criteria	All core functionality of the systems should function as expected and outlined in the individual test cases. There must be no critical defects found and an end user must be able to complete a purchase cycle successfully and initiate a refund without any errors. 95% of all test cases should pass and no failed cases should be crucial to the end-user's ability to use the application.
Suspension Criteria	Testing should be paused immediately if either part of the system is found vulnerable to a security breach.
Test Deliverables	Test Plan (this document itself) Test Cases Test Scripts Defect/Enhancement Logs Test Reports
Test Environment	Raspberry Pi 3 device with Raspbian Stretch version 4.14 operating system Internet access Testing tools include Nmap, cURL, ShellCheck

Table 5.2 shows the test use case summary results filled in by the users testing the tool. From the actual results obtained, it is evident that all test cases succeeded and that the tool is functioning properly as intended.

Table 5.2 Test Case

ID	Test Case and Procedure	Expected Results	Actual Results	Pass /Fail
1	Test Name: Data encryption Test Procedure: Open the CLI Type command 'curl -kis http://github.com' to check if the site allows unencrypted traffic Press 'return' button	HTTP/1.1 301 Moved Permanently Content-length: 0 Location: https://github.com/	HTTP/1.1 301 Moved Permanently Content-length: 0 Location: https://github.com/	Pass
2	Test Name: SSL/TLS ciphers Test Procedure: Open the CLI Install Nmap 'apt-get install nmap' Type command 'nmap -sV --reason -PN -n --top-ports 100 www.github.com' to confirm if ssl/https services are running. Press 'return' button Type command 'nmap --script ssl-cert,ssl-enum-ciphers -p 443 www.github.com' to check certificate information, weak ciphers and SSL/TLS. Press 'return' button	Port: 443, Service: ssl/https Status: Open Reason: syn-ack ssl-cert: Subject: commonName=github.com/organizationName=GitHub, Inc./stateOrProvinceName=California/countryName=US Subject Alternative Name: DNS:github.com, DNS:www.github.com Issuer: commonName=DigiCert SHA2 Extended Validation Server CA/organizationName=DigiCert Inc/countryName=US Public Key type: rsa Public Key bits: 2048 Signature Algorithm: sha256WithRSAEncryption Not valid before: 2016-03-10T00:00:00 Not valid after: 2018-05-17T12:00:00 MD5: b890 fabe 8bb6 3625 899e 1e00 4981 4797 SHA-1: d79f 0761 10b3 9293 e349 ac89 845b 0380 c19e 2f8b	Port: 443, Service: ssl/https Status: Open Reason: syn-ack ssl-cert: Subject: commonName=github.com/organizationName=GitHub, Inc./stateOrProvinceName=California/countryName=US Subject Alternative Name: DNS:github.com, DNS:www.github.com Issuer: commonName=DigiCert SHA2 Extended Validation Server CA/organizationName=DigiCert Inc/countryName=US Public Key type: rsa Public Key bits: 2048 Signature Algorithm: sha256WithRSAEncryption Not valid before: 2016-03-10T00:00:00 Not valid after: 2018-05-17T12:00:00 MD5: b890 fabe 8bb6 3625 899e 1e00 4981 4797 SHA-1: d79f 0761 10b3 9293 e349 ac89 845b 0380 c19e 2f8b	Pass

3	Test Name: SSL/TLS certificates Test Procedure: Open the CLI Type command ‘curl --insecure -v https://www.github.com 2>&1 | awk 'BEGIN { cert=0 } /^* Server certificate:/ { cert=1 } /^*/ { if (cert) print }'’ to validate ssl certificate. Press ‘return’ button	SSL certificate verify ok	SSL certificate verify ok	Pass
4	Test Name: Source code Test Procedure: Open a web browser Go to URL ‘https://www.shellcheck.net/’ Paste the script code in the Ace editor to find bugs Check the ShellCheck Output	$ shellcheck .\update.sh No issues detected	$ shellcheck .\update.sh No issues detected	Pass



5.6.2 Functional Testing

Functional testing checks if core requirements and functionalities are met.

1. Automate update process: This is achieved by setting up crontab that will be running at a predefined time interval. An administrator can increase or reduce the time interval depending on the organisation requirements. The cron code below implements this feature:
   ```
   00 09-18 * * * sh /home/pi/AutoUpdatePlugin/update.sh 2>/home/pi/logs/cronlog
   ```
2. Create a secure TLS connection: The proper implementation of TLS 1.2 ensures that this functional requirement is met. The tool instantiates a secure TLS connection using the following openssl command:
   ```
   echo | openssl s_client -connect www.github.com:443 2>&1 | sed --quiet '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > github.crt
   ```
3. Check and download updates: This is achieved through the following git code:
   ```
   git pull https://github.com/vmalombe/Pi_LED_Blinking.git
   ```

5.6.3 Structural Testing

In structural testing, tests are derived from the knowledge of the software's structure or internal implementation. Structural testing is critical because it is meant to secure updates relating to embedded systems, that usually control critical systems. The use of an existing cryptography framework helped in reducing the number of errors because the bugs had been identified and improved on over time by a large community of developers. The development process involved constant peer review to counter check the logic of the code.

5.6.4 Testing for Sensitive Data Transmitted in Clear-Text

The cURL tool was used to attempt an HTTP connection to the server using the command below:

```
curl -kis http://github.com
```

The server HTTP/1.1 301 message indicates that the server is not reachable through unencrypted channel, and gives the encrypted link that cab be used. Appendix Eillustrates this manual test.

5.6.5 Testing for Weak SSL/TLS Ciphers/Protocols/Keys Vulnerabilities

The first step was to identify ports which have SSL/TLS wrapped services using Nmap using the command below:

```
nmap -sV --reason -PN -n --top-ports 100 www.github.com
```

Scan results indicate that port 443 (SSL/HTTPS) is open, as shown in Appendix E.

The next step was to check certificate information, weak ciphers and SSL/TLS using Nmap scripts (ssl-cert, ssl-enum-ciphers) with the command below:

nmap --script ssl-cert,ssl-enum-ciphers -p 443 www.github.com

Appendix E illustrates this test whereby the GitHub certificate is issued by DigiCert Inc, a trusted CA, an RSA public key of 2048 bit is used, sha256 is the signature algorithm and a list of strong cipher suites are supported.

5.6.6 Testing SSL/TLS Certificate Validity - Server

The cURL command below checks if the certificate presented by the server is valid.

curl --insecure -v https://www.github.com 2>&1 | awk 'BEGIN { cert=0 } /^* Server certificate:/ { cert=1 } /^*/ { if (cert) print }'

Appendix E indicates a successful ssl certificate validation process.

Chapter 6 Discussion of Results

6.1 Overview

Findings obtained during the study formed the basis on which the auto-update plugin prototype was developed. The prototype was tested to ascertain that it met all its requirements. This chapter analyses the findings in relation to the research objectives and extent to which the findings agree with the literature review.

6.2 Objective One

The first objective was to investigate how software updates are delivered and identify gaps and challenges in updating software in embedded systems. IoT developers often require updating their software in single board computers. This may be due to need to change software functionalities, fix security bugs, among other reasons. Section 2.4 identified the main three techniques that are used to update software in single board computers.

Secure Shell (SSH) is the most common technique that provides integrity. However, SSH does not provide server authentication. The client device must verify the authenticity of the server by verifying its public key on its own. This is dangerous since another server with a valid certificate can update the client.

The developed secure auto update plugin runs on Transport Layer Security (TLS), which provides server authentication through Public Key Infrastructure (PKI). Server certificate is signed by a trusted Certificate Authority. Also, the update plugin is configured to connect only to one update source controlled by the developer.

6.3 Objective Two

The second objective was to investigate the suitability of PKI in delivering software updates securely. Public Key Infrastructure has been used in other system implementations to provide secure data transfer between client and server devices. This includes application carrying sensitive updates. Section 2.6 discusses the structure and functions of Public Key Infrastructure to provide data encryption, authentication and nonrepudiation. Public Key Infrastructure has been standardised to allow interoperability across a wide variety of applications and vendors through Public-Key Cryptography Standards (PKCS).

The use of identity certificates and certificate validation algorithms has been defined by ITU-T's X.509 standards. Today, there are many vendors that offer Certificate Authority servers as a service. This creates a chain of trust amongst communication devices. Modern operating systems ship with a list of the trusted SSL vendors which the client devices can trust. This

VT OMNES VNVM SINT

research leverages on this model to ensure and prove server's identity, as opposed to self-signed certificates which cannot be trusted in public networks.

Transport Layer Security (TLS) is an implementation framework for PKI for web services to provide secure transactions. Section 2.7 outlines its features. It uses cipher suites (section 2.8) which define the authentication and key exchange algorithms (such as RSA), encryption algorithm (such as Advanced Encryption Standard - AES), message authentication code algorithm (such as SHA), and the PRF.

6.4 Objective Three

The third objective was to design and implement an automated PKI based updates plugin for embedded systems. The update.sh script is run by cron, which refers to a daemon to execute scheduled commands. The cron utility is launched by launchd (Unix system wide and per-user daemon/agent manager) when it sees the existence of /etc/crontab or files in /usr/lib/cron/tabs. The crontab shown in figure 6.1 was created to checks the status of the script daily during the working hours 9 a.m. - 6 p.m.

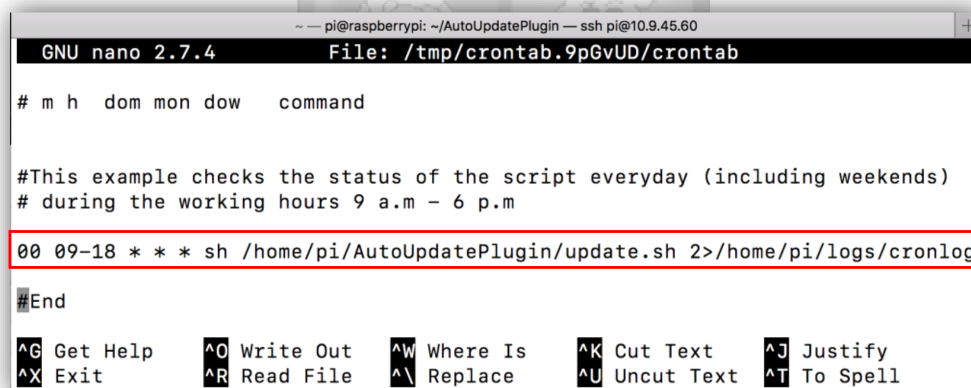


Figure 6.1 Crontab

The cron utility then wakes up every minute, examining all stored crontabs, checking each command to see if it should be run in the current minute.

6.5 Objective Four

The fourth objective was to validate the PKI based updates plugin for authenticity and integrity. Section 5.6 addressed the system security testing to ensure that it is secure. Test on HTTP traffic revealed that updates can only be transmitted in an encrypted TLS tunnel, and not in plain text. Test on certificate information and weak ciphers revealed that the certificate is valid and signed by a trusted CA (DigiCert Inc). A strong cipher suite TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 is used for communication.

Therefore, the automated tool is able to update custom applications for embedded systems in a secure way.

6.6 Advantages of the Developed Solution Compared to Existing Tools

6.6.1 Open Source

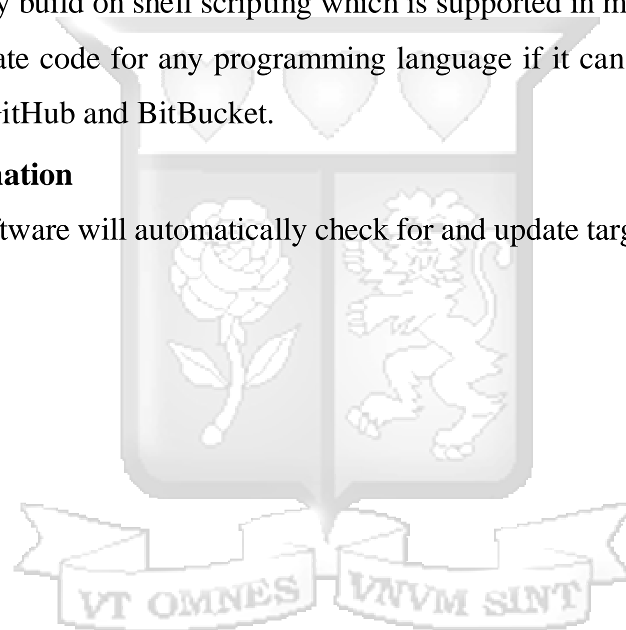
The secure update plugin is an open source software that can be downloaded via https://github.com/vmalombe/Auto-Update-Plugin-for-Single-Board-Computer-Software.git. It is distributed under the MIT License whose conditions only require preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

6.6.2 Multiplatform

The software is largely build on shell scripting which is supported in many operating systems. It can be used to update code for any programming language if it can be hosted in a version control system such GitHub and BitBucket.

6.6.3 Automation

Once installed, the software will automatically check for and update targeted code without user intervention.



Chapter 7 Conclusion, Recommendations and Future Work

7.1 Conclusion

The study reveals that IoT developers use unsecure methods to update custom software in single board computers. Such update methods increase the attack surface of the embedded system, which may be controlling a critical system. Some other ways are not efficient in an environment with many single board computers or having many updates to the custom software. They take a lot of administration time and effort just to ensure software is up to date. The study shows existence of some tools that try to provide an update framework, but they are platform depended and not free or open source to be used by upcoming developers or small organisations who do not have a budget for such licenses. Using the development tools discussed in chapter 4, the automated secure update plugin prototype was developed. During system development agile methodology was used. This allowed for more frequent release with subsequent user feedback which led to development of a usable and reliable prototype. Usability testing and system validation was performed, and respondents generally found the tool valuable and satisfying.

The aim of the developed system is to automate the update process of any custom software of single board computers in a secure way.

7.2 Recommendation

The auto-update secure update plugin was of great importance to IoT developers. The researcher noted however, that there was still a lot more that can be done. The following recommendations are thus given:

There is need for public awareness to the existence of this open source tool. The tool can be advanced by collaborating with private and public organisations to encourage IoT developers to use it and avoid the issues identified with the identified usual ways of IoT software. Its wide use would improve it with time to increase its efficiency. The open source community is encouraged to evaluate, critique and give recommendations to improve this tool.

Developers using private Certificate Authorities and self-signed certificates should install git server locally and point to auto-update tool to it. However, the disadvantages of this model should be noted and addressed using extra security measures such as having a very secured private network perimeter.

Finally, integration with the single board computer default software base would ensure that it is widely used.

7.3 Future Work

The scope of this study was to secure data in transit over the network. Future work will concern ensuring that the downloaded updates are secured, and that the single board computers are well configured to reduce the attack surface. Security intrusion detection and alert capabilities will also be added to the tool to offer more comprehensive security service.

References

Beagle Board, (2018). Retrived on April 2018 from http://beagleboard.org/black

Beck, K., Beedle, M., Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., & Kern, J. (2001). Manifesto for agile software development. Retrieved January 11, 2017, from https://abacus.abo.fi/ro.nsf/141b8735bd22ff31c225700600473a01/b71e893af50d2744c22 57ada0033bf94/$FILE/Bilaga1.pdf

Bellemans, J. (2017). IoT Botnets: A look under the hood of Mirai. Retrieved on March 2018 from http://www.blackholetec.com/main/article/iot-botnets-look-under-hood-mirai

Bertino, E. (2016, March). Data Security and Privacy in the IoT. In EDBT (Vol. 2016, pp. 1-3).

Bertino, E., & Islam, N. (2017). Botnets and Internet of things security. Computer, 50(2), 76-79.

Boe, M., & Altman, J. (2002). TLS-based Telnet Security. draft-ietftn3270e-telnet-tls-06 (work in progress).

Chacon, S., & Straub, B. (2014). Pro git. Apress.

Chernick, C. M., Edington III, C., Fanto, M. J., & Rosenthal, R. (2005). Guidelines for the selection and use of transport layer security (TLS) implementations (No. Special Publication (NIST SP)-800-52).

Complementar, B. (2016). CISCO CyberSecurity course–Netacad• LONG, Johnny. Google Hacking Para Pentest, Novatec.

Cooper, D. (2008). Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile.

FileZilla® Project, 2018. Overview. Retrieved on march 2018 from https://filezilla-project.org/

Ford, W. (1998). Public Key Infrastructure Interoperation. IEEE Aerospace Conference, Snowmass at Aspen, Vol. 4, 329-333.

Freier, A. O., Karlton, P., & Kocher, P. C. (1996). Secure socket layer 3.0. IETF draft, November.

Gigovic, B. (2014): Fundamentals of the PKI Infrastructure. Global Knowledge Training LLC. Retrieved on March 2017 from http://losnuevosguerreros.org/pluginfile.php/16/mod_glossary/attachment/555/WP_SI_PKI_Infrastructure.pdf

Goodin, D. (2016). Record-breaking DDoS reportedly delivered by> 145k hacked cameras. Ars Technica, 28.

Grau, A. (2016): Secure Software Updates for IoT Devices. Icon Labs. Retrieved on March 2017 from https://down.dsg.cs.tcd.ie/iotsu/subs/IoTSU_2016_paper_2.pdf

Grau, A. (2016). Internet of Things Software Update Workshop (IoTSU): Secure Software Updates for IoT Devices.

Heffner, C. 2010. How to Hack Millions of Routers.Seismic LLC. Retrieved on March 2017 from https://www.defcon.org/images/defcon-18/dc-18-presentations/Heffner/DEFCON-18-Heffner-Routers.pdf

Hill, J. L. (2003). System architecture for wireless sensor networks (Doctoral dissertation, University of California, Berkeley).

Hodges, J., Morgan, R., & Wahl, M. (2000). Lightweight Directory Access Protocol (v3): Extension for Transport Layer Security (No. RFC 2830).

Hoffman, P. (2002). SMTP service extension for secure SMTP over transport layer security.

Isikdag, U. (2015). Internet of Things: Single-board computers. In Enhanced Building Information Models (pp. 43-53). Springer, Cham.

Jayaprakash, K. (2015). Public Key Infrastructure: A Survey. Scientific Research Publishing Inc.

Jaziri, I., Charaabi, L., & Jelassi, K. (2016, October). A closed Loop DC Motor Control using low cost single-board microcontroller based on embedded Linux. In Electrical Sciences and Technologies in Maghreb (CISTEM), 2016 International Conference on (pp. 1-5). IEEE.

Kleidermacher, D., & Kleidermacher, M. (2012). Embedded systems security: practical methods for safe and secure software and systems development. Elsevier.

Kruger, C. P., & Hancke, G. P. (2014, July). Benchmarking Internet of things devices. In Industrial Informatics (INDIN), 2014 12th IEEE International Conference on (pp. 611-616). IEEE.

Kuchcinski, K. (2017). Design of Embedded Systems. Lund Institute of Technology Sweden.

Kumar, N., Madhuri, J., & ChanneGowda, M. (2017). Review on security and privacy concerns in Internet of things. In IoT and Application (ICIOT), 2017 International Conference on (pp. 1-5). IEEE.

Liu, Y., Guo, L., Liu, J., Yue, Y., Maple, C., & Crabbe, C. (2013). An Application-Oriented Top-Down Scheme for FPGA-Based Embedded System Design with 3D Graphics Applications. International Conference on Computer Sciences and Applications, Wuhan, pp. 756-764. doi: 10.1109/CSA.2013.182

Maksimović, M., Vujović, V., Davidović, N., Milošević, V., & Perišić, B. (2014). Raspberry Pi as Internet of things hardware: performances and constraints. design issues, 3, 8.

Marwedel, P. (2010). Embedded system design: Embedded systems foundations of cyber-physical systems. Springer Science & Business Media.

Microsoft. (2015). How to Enable CRL Checking for Software Updates. Retrieved on April 2017 from https://technet.microsoft.com/en-us/library/hh508767.aspx

Modio. (2018). How it works. Retrieved on March 2018 from https://www.modio.se/technology.html

Monk, S. (2015). Programming the Raspberry Pi: getting started with Python. TAB Electronics.

Nath, A. (2015). Packet Analysis with Wireshark. Packt Publishing Ltd.

Newman, C. (1999). Using tls with imap, pop3 and acap.

NXP Semiconductors, (2016). A710x family: Secure authentication microcontroller. Retrieved on April 2017 from http://www.nxp.com/products/identification-and-security/authentication/secure-authentication-microcontroller:A710X_FAMILY

Open Web Application Security Project. (2014). Top 10 IoT Vulnerabilities (2014) Project. Retrieved on March 2018 from https://www.owasp.org/index.php/Top_IoT_Vulnerabilities

Oualline, S. (2001). Vi iMproved. New Riders Publishing.

Palihakkara, S. (2014). WSO2 ESB OCSP/CRL Verification implementation. Retrived on April 2018 from http://problemsolvedweb.blogspot.co.ke/2014/03/wso2-esb-ocspcrl-verification.html

Papp, D., Ma, Z., & Buttyan, L. (2015). Embedded systems security: Threats, vulnerabilities, and attack taxonomy. 13th Annual Conference on Privacy, Security and Trust (PST), Izmir, pp. 145-152. doi: 10.1109/PST.2015.7232966

Park, S. D. (2011).Port forwarding configuration system and method for wire and wireless networks. U.S. Patent No. 8,050,192. Washington, DC: U.S. Patent and Trademark Office.

Perry, D. (1998). Vhdl (Vol. 2). McGraw-Hill.

Pi, R. (2013). Raspberry Pi. Raspberry Pi, 1, 1. Retrieved on April 2017 from http://micklord.com/foru/Raspberry%20Pi%20Pages%20from%20Computer%20Shopper%202015-02.pdf

Polk, T., McKay, K., & Chokhani, S. (2014). Guidelines for the selection, configuration, and use of transport layer security (TLS) implementations. NIST special publication, 800(52), 32.

Postel, J., & Reynolds, J. (1985). File transfer protocol.

Python 3.6.1 documentation. (2017). Retrieved on April 2017, from https://docs.python.org/3/

Raspberry Pi Foundation. (2018). Access your Raspberry Pi over the Internet - Raspberry Pi Documentation. Retrieved on March 2018 from https://www.raspberrypi.org/documentation/remote-access/access-over-Internet/README.md

Rea, S. (2015). AllSeen Summit. Taking PKI Where No PKI Has Gone Before.

Rescorla, E. (2001). SSL and TLS: designing and building secure systems (Vol. 1). Reading: Addison-Wesley.

Richardson, M., & Wallace, S. (2012). Getting started with raspberry PI. " O'Reilly Media, Inc.".

Ross, M., & Morrison, R. (1996). Experimental research methods. Handbook of research for educational communications and technology: A project of the association for educational communications and technology, 1148-1170.

Salomaa, A. (2013). Public-key cryptography. Springer Science & Business Media.

Schmidt, M. (2014). Raspberry Pi: A Quick-Start Guide. Pragmatic Bookshelf.

Schneier, B. (2014). The Internet of Things Is Wildly Insecure And Often Unpatchable. Retrieved on March 2017 from https://www.schneier.com/essays/archives/2014/01/the_Internet_of_thin.html

Simmonds, C. (2016). Software update for IoT. 2net Ltd.

Slagell, A., Bonilla, R. & Yurcik, W. (2006). A Survey of PKI Components and Scalability Issues. 25th IEEE Performance, Computing, and Communications Conference, Phoenix.

Steiner, M., Buhler, P., Eirich, T., & Waidner, M. (2001). Secure password-based cipher suite for TLS. ACM Transactions on Information and System Security, 4(2), 134-157.

Turner, S. (2014). Transport layer security. IEEE Internet Computing, 18(6), 60-63.

Upton, E., & Halfacree, G. (2014). Raspberry Pi user guide. John Wiley & Sons.

Vacca, J. (2013). Public Key Infrastructure. In: Cyber Security and IT Infrastructure Protection, Steven Elliot, Waltham, pg 75-107.

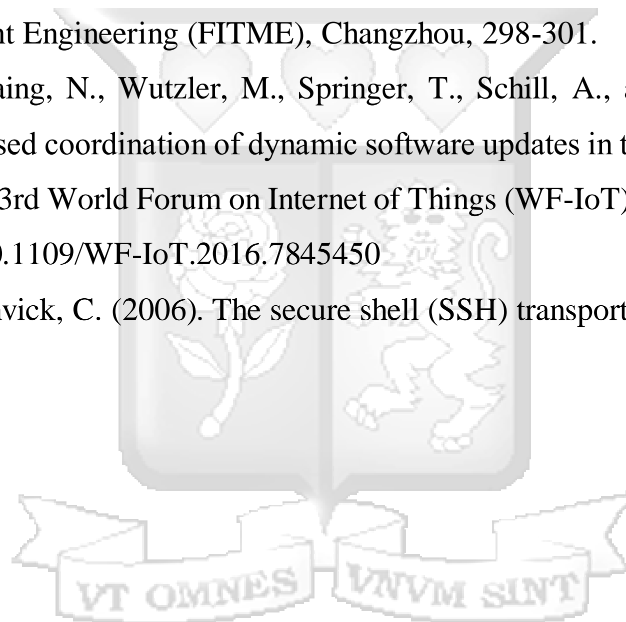
Vujovic, V., & Maksimovic, M. (2014). Raspberry Pi as a Wireless Sensor node: Performances and constraints. In Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on (pp. 1013-1018). IEEE.

W3Techs, (2018). Usage of SSL certificate authorities for websites. Retrieved on March 2018 from https://w3techs.com/technologies/overview/ssl_certificate/all

Wang, K., & Zhang, Z. (2010). Design and Implementation of a Safe Public Key Infrastructure. International Conference on Future Information Technology and Management Engineering (FITME), Changzhou, 298-301.

Weißbach, M., Taing, N., Wutzler, M., Springer, T., Schill, A., and Clarke, S. (2016). “Decentralised coordination of dynamic software updates in the Internet of Things,” 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), Reston, VA, pp. 171-176. doi: 10.1109/WF-IoT.2016.7845450

Ylonen, T., & Lonvick, C. (2006). The secure shell (SSH) transport layer protocol.



Appendices

Appendix A TLS Handshake Message Structure

```
struct {
        ProtocolVersion client_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suites<2..2^16-2>;
        CompressionMethod compression_methods<1..2^8-1>;
        select (extensions_present) {
            case false:
                struct {};
            case true:
                Extension extensions<0..2^16-1>;
        };
    } ClientHello;
```

Figure A.1 ClientHello Message Structure

```
struct {
        ProtocolVersion server_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suite;
        CompressionMethod compression_method;
        select (extensions_present) {
            case false:
                struct {};
            case true:
                Extension extensions<0..2^16-1>;
        };
    } ServerHello;
```

Figure A.2 ServerHello Message Structure

```
opaque ASN.1Cert<1..2^24-1>;

struct {
    ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
```

Figure A.3 Server Certificate Message Structure

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa:
            EncryptedPreMasterSecret;
        case dhe_dss:
        case dhe_rsa:
        case dh_dss:
        case dh_rsa:
        case dh_anon:
            ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

Figure A.4 Client Key Exchange Message Structure

```
master_secret = PRF(pre_master_secret, "master secret",
                    ClientHello.random + ServerHello.random)
                    [0..47];
```

Figure A.5 Calculating master_secret

```
struct {
    opaque verify_data[verify_data_length];
} Finished;
```

Figure A.6 Handshake Finalization Message Structure

VT OMNES VNVM SINT

Appendix B Setup and Configuration

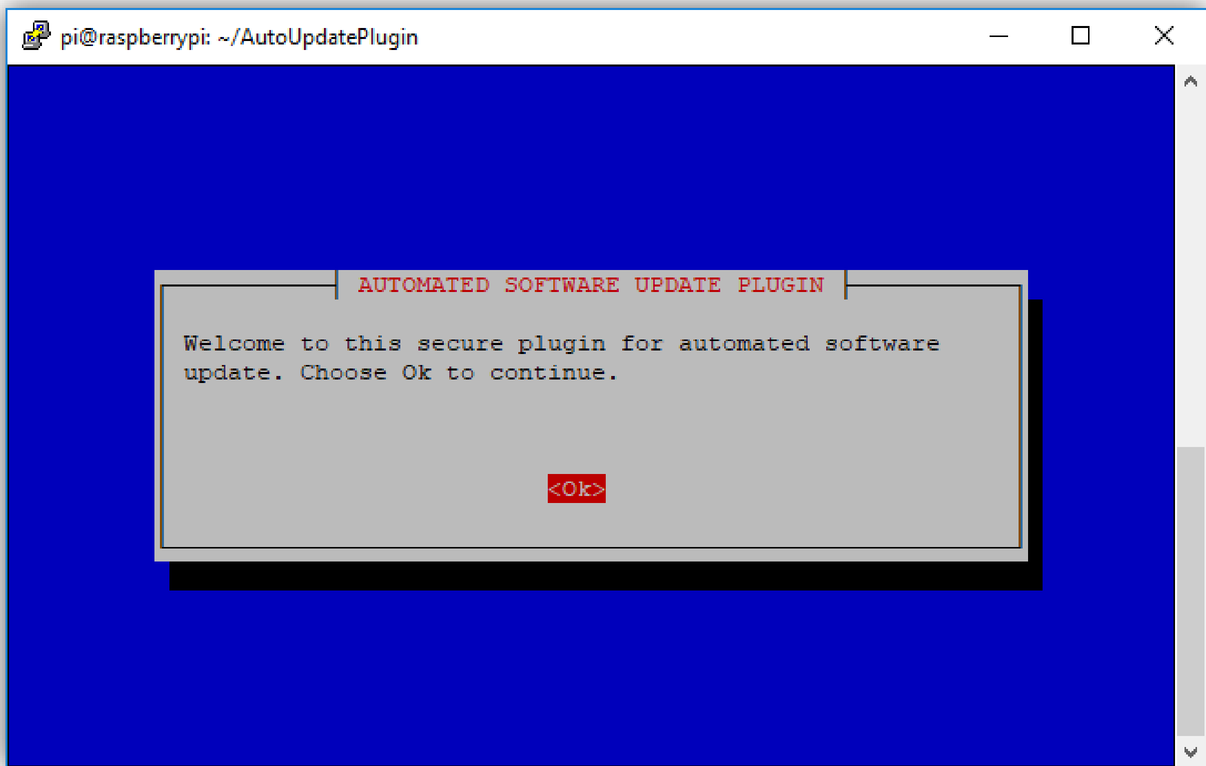


Figure B.1 Configuration Welcome Message

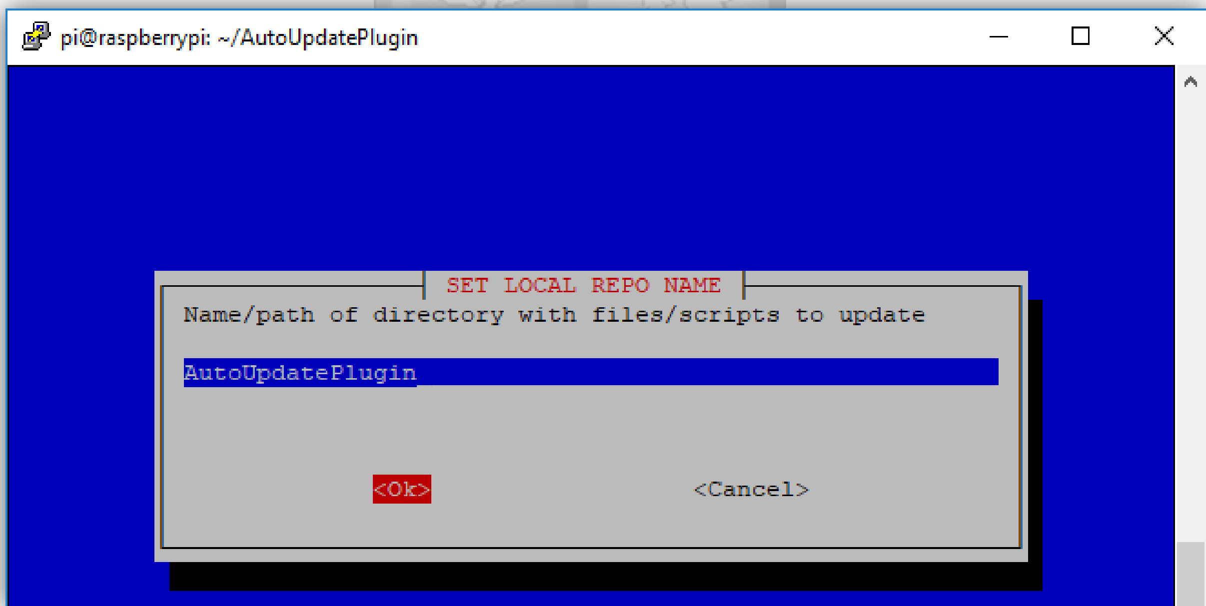


Figure B.2 Set Path to Local Repository

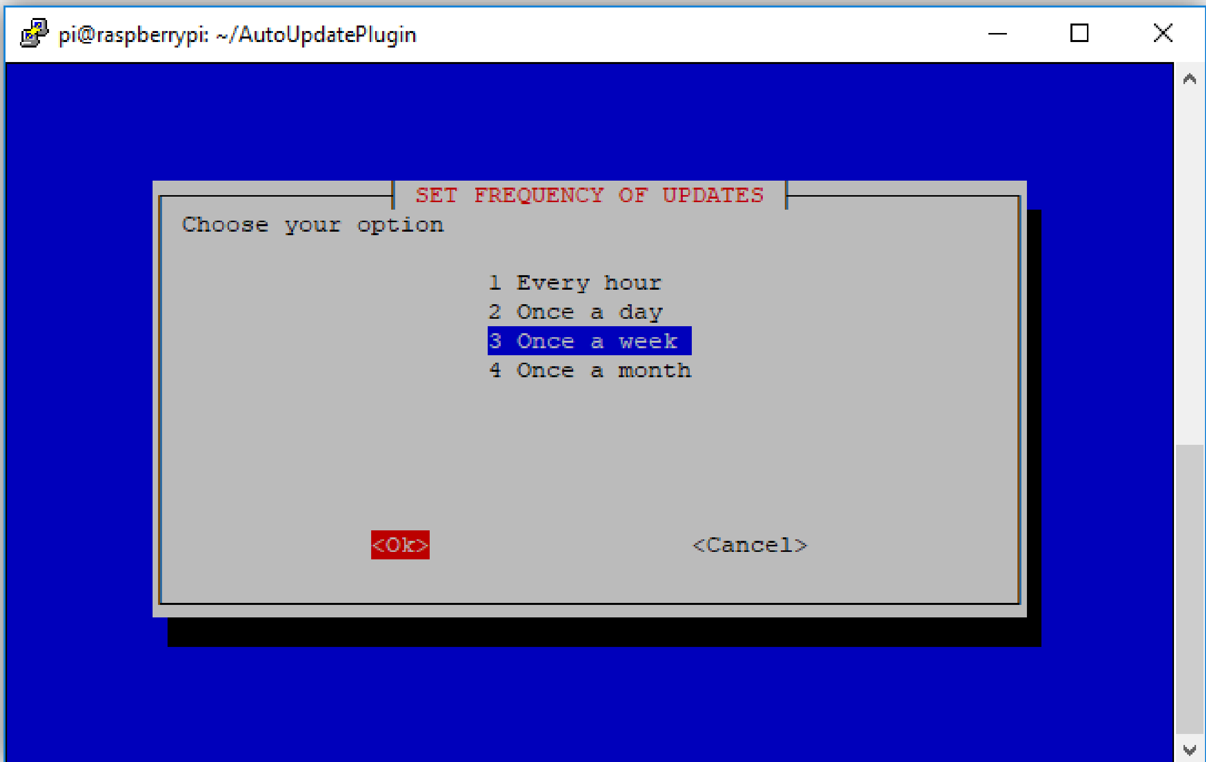


Figure B.3 Set Frequency of Update

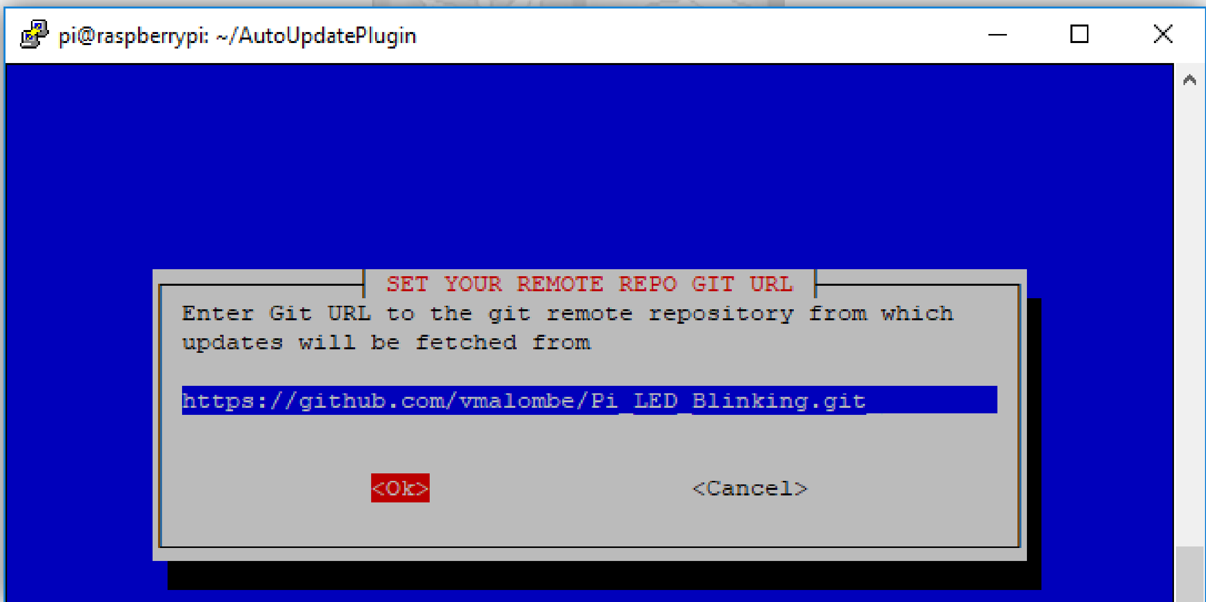


Figure B.4 Remote Repo URL

```
~ — pi@raspberrypi: ~/AutoUpdatePlugin — ssh pi@10.9.45.60
pi@raspberrypi:~/AutoUpdatePlugin $ ./installation.sh
Script name: AutoUpdatePlugin
Your chosen option: 2
Git URL to remote git repo: https://github.com/vmalombe/Pi_LED_Blinking.git
Cloning into 'Pi_LED_Blinking'...
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (39/39), done.
remote: Total 43 (delta 21), reused 11 (delta 4), pack-reused 0
Unpacking objects: 100% (43/43), done.
pi@raspberrypi:~/AutoUpdatePlugin $
```

Figure B.5 Installation Summary

Appendix C Server Connection

```bash
#!/bin/bash
URL=$(whiptail --title "SET YOUR REMOTE REPO GIT URL" --inputbox "Enter Git URL to the git remote repository from which updates will be fetched from" 10 60 https://github.com/vmalombe/Pi_LED_Blinking.git 3>&1 1>&2 2>&3)

exitstatus=$?
if [ $exitstatus = 0 ]; then
    echo "Git URL to remote git repo:" $URL
    echo | openssl s_client -connect www.github.com:443 2>&1 | sed --quiet '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > github.crt
    git clone $URL
else
    echo "You chose Cancel."
fi

```

Figure C.1 Code Connect to Server

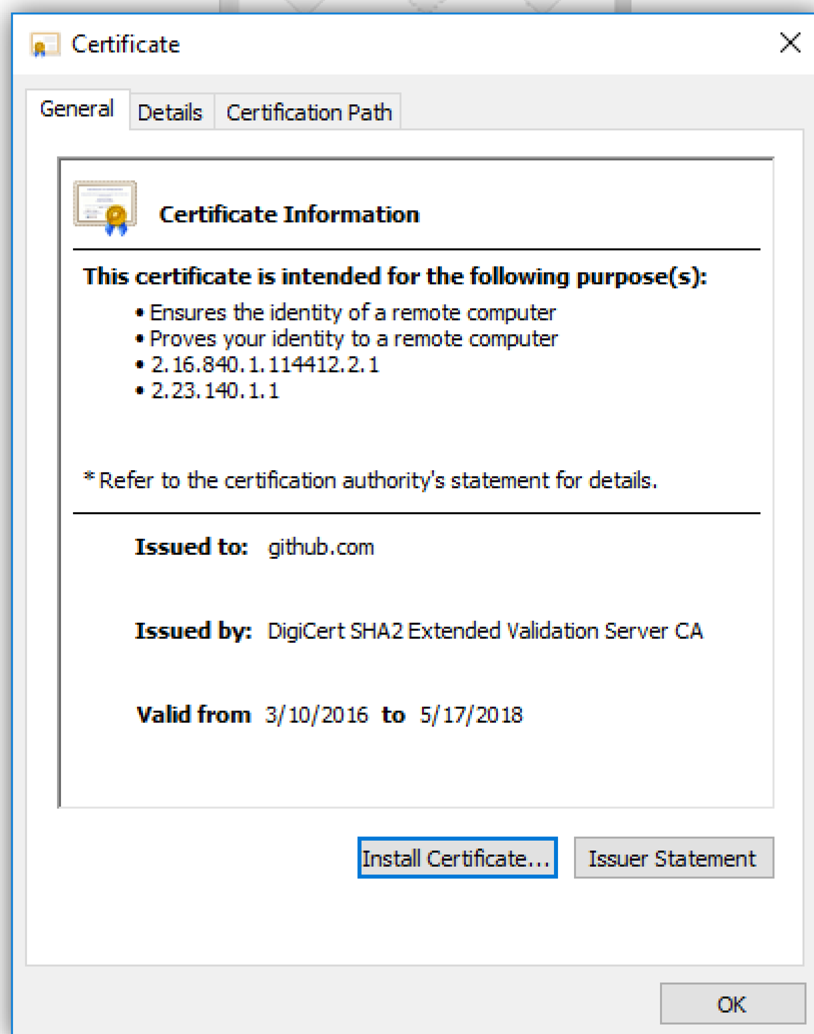


Figure C.2 Server Certificate Details

```
~ — pi@raspberrypi: ~/AutoUpdatePlugin — ssh pi@10.9.45.60
pi@raspberrypi:~/AutoUpdatePlugin $ ls /etc/ssl/certs/ | grep DigiCert
DigiCert_Assured_ID_Root_CA.pem
DigiCert_Assured_ID_Root_G2.pem
DigiCert_Assured_ID_Root_G3.pem
DigiCert_Global_Root_CA.pem
DigiCert_Global_Root_G2.pem
DigiCert_Global_Root_G3.pem
DigiCert_High_Assurance_EV_Root_CA.pem
DigiCert_Trusted_Root_G4.pem
pi@raspberrypi:~/AutoUpdatePlugin $
```

Figure C.3 Raspbian Operating System Trusted Certificates

Appendix D Packet Captures & Update Checking

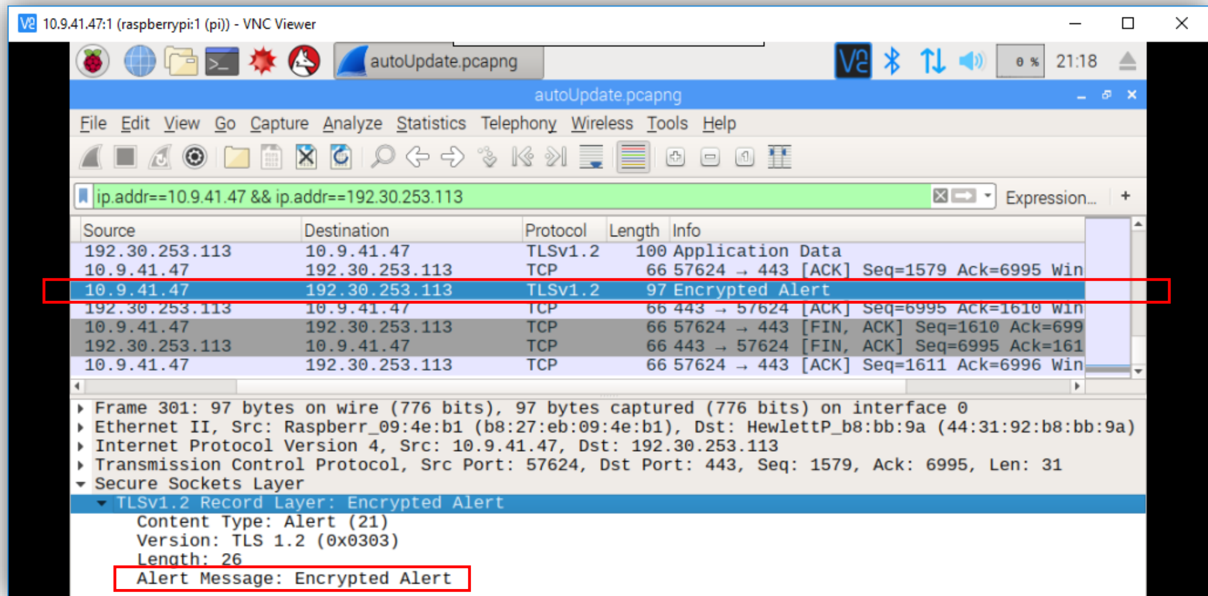


Figure D.1 Encrypted Application & Alert Packets

```
1   #!/bin/sh
2
3   #Go into directory
4   cd Pi_LED_Blinking
5
6   #Then fetch and merge remote git to clone
7   git pull https://github.com/vmalombe/Pi_LED_Blinking.git
8
9   #And then run it afresh
10  sudo python PiFreq-Auto.py
11
12  #End
13
```

Figure D.2 Updates Fetching and Merging Code

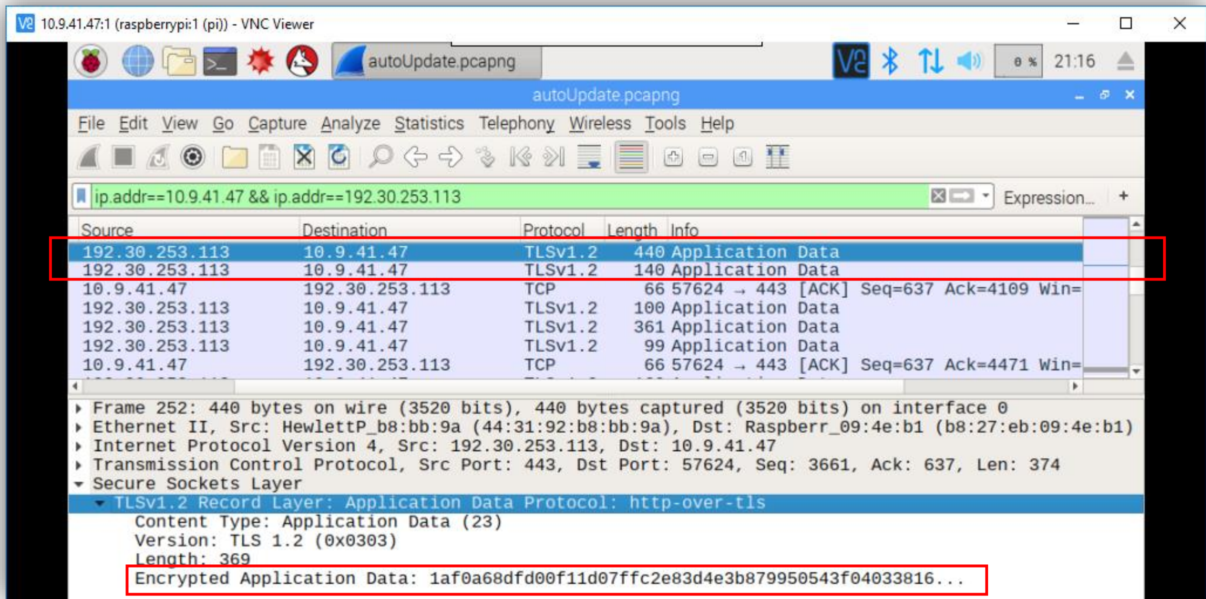


Figure D.3 Encrypted Update Packet

```
~ — pi@raspberrypi: ~/AutoUpdatePlugin — ssh pi@10.9.45.60
pi@raspberrypi:~/AutoUpdatePlugin $ ./update.sh
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 4), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/vmalombe/Pi_LED_Blinking
 * branch            HEAD       -> FETCH_HEAD
Updating 4486238..d5823a4
Fast-forward
 LICENSE           | 2 +-
 PiFreq-Manual.py  | 2 +-
 2 files changed, 2 insertions(+), 2 deletions(-)
pi@raspberrypi:~/AutoUpdatePlugin $ ./update.sh
From https://github.com/vmalombe/Pi_LED_Blinking
 * branch            HEAD       -> FETCH_HEAD
Already up-to-date.
pi@raspberrypi:~/AutoUpdatePlugin $
```

Figure D.4 Encrypted Update Packet

Appendix E Manual Testing

```
~ — pi@raspberrypi: ~/AutoUpdatePlugin — ssh pi@10.9.45.60
pi@raspberrypi:~/AutoUpdatePlugin $ curl -kis http://github.com
HTTP/1.1 301 Moved Permanently
Date: Fri, 27 Apr 2018 15:04:39 GMT
Content-length: 0
Location: https://github.com/

pi@raspberrypi:~/AutoUpdatePlugin $
```

Figure E.1 Testing for Sensitive Data Transmitted in Clear-Text

```
~ — pi@raspberrypi: ~/AutoUpdatePlugin — ssh pi@10.9.45.60
pi@raspberrypi:~/AutoUpdatePlugin $ nmap -sV --reason -PN -n --top-ports 1
00 www.github.com

Starting Nmap 7.40 ( https://nmap.org ) at 2018-04-16 20:32 UTC
Nmap scan report for www.github.com (192.30.253.113)
Host is up, received user-set (0.00074s latency).
Other addresses for www.github.com (not scanned): 192.30.253.112
Not shown: 93 filtered ports
Reason: 93 no-responses
PORT     STATE  SERVICE   REASON       VERSION
21/tcp   open   ftp?      syn-ack
80/tcp   open   http      syn-ack
110/tcp  open   pop3?     syn-ack
113/tcp  closed ident     conn-refused
143/tcp  open   imap?     syn-ack
443/tcp  open   ssl/https syn-ack
8008/tcp open   http      syn-ack      Fortinet FortiGuard block page
```

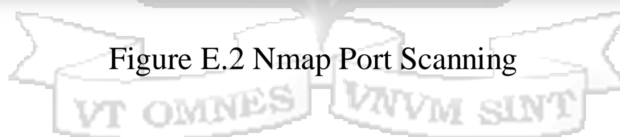


Figure E.2 Nmap Port Scanning

```
~ — pi@raspberrypi: ~/AutoUpdatePlugin — ssh pi@10.9.45.60
pi@raspberrypi:~/AutoUpdatePlugin $ nmap --script ssl-cert,ssl-enum-ciphers
-p 443 www.github.com

Starting Nmap 7.40 ( https://nmap.org ) at 2018-04-16 20:28 UTC
Nmap scan report for www.github.com (192.30.253.112)
Host is up (0.00086s latency).
Other addresses for www.github.com (not scanned): 192.30.253.113
rDNS record for 192.30.253.112: lb-192-30-253-112-iad.github.com
PORT    STATE SERVICE
443/tcp open  https
| ssl-cert: Subject: commonName=github.com/organizationName=GitHub, Inc./sta
teOrProvinceName=California/countryName=US
| Subject Alternative Name: DNS:github.com, DNS:www.github.com
| Issuer: commonName=DigiCert SHA2 Extended Validation Server CA/organizatio
nName=DigiCert Inc/countryName=US
| Public Key type: rsa
| Public Key bits: 2048
| Signature Algorithm: sha256WithRSAEncryption
| Not valid before: 2016-03-10T00:00:00
| Not valid after:  2018-05-17T12:00:00
| MD5:   b890 fabe 8bb6 3625 899e 1e00 4981 4797
|_SHA-1: d79f 0761 10b3 9293 e349 ac89 845b 0380 c19e 2f8b
| ssl-enum-ciphers:
|   TLSv1.2:
|     ciphers:
|       TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (secp256r1) - A
|       TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (secp256r1) - A
|       TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (secp256r1) - A
|       TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (secp256r1) - A
|       TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (secp256r1) - A
|       TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (secp256r1) - A
|       TLS_RSA_WITH_AES_128_GCM_SHA256 (rsa 2048) - A
|       TLS_RSA_WITH_AES_256_GCM_SHA384 (rsa 2048) - A
|       TLS_RSA_WITH_AES_128_CBC_SHA256 (rsa 2048) - A
|       TLS_RSA_WITH_AES_128_CBC_SHA (rsa 2048) - A
|       TLS_RSA_WITH_AES_256_CBC_SHA256 (rsa 2048) - A
|       TLS_RSA_WITH_AES_256_CBC_SHA (rsa 2048) - A
|     compressors:
|       NULL
|     cipher preference: server
|_  least strength: A

Nmap done: 1 IP address (1 host up) scanned in 22.73 seconds
pi@raspberrypi:~/AutoUpdatePlugin $
```

Figure E.3 Nmap Check Certificate Information, Weak Ciphers and SSL/TLS

```
~ — pi@raspberrypi: ~/AutoUpdatePlugin — ssh pi@10.9.45.60
pi@raspberrypi:~/AutoUpdatePlugin $ curl --insecure -v https://www.github.
com 2>&1 | awk 'BEGIN { cert=0 } /^\* Server certificate:/ { cert=1 } /^\*
/ { if (cert) print }'
* Server certificate:
*  subject: businessCategory=Private Organization; jurisdictionC=US; juris
dictionST=Delaware; serialNumber=5157550; street=88 Colin P Kelly, Jr Stre
et; postalCode=94107; C=US; ST=California; L=San Francisco; O=GitHub, Inc.
; CN=github.com
*  start date: Mar 10 00:00:00 2016 GMT
*  expire date: May 17 12:00:00 2018 GMT
*  issuer: C=US; O=DigiCert Inc; OU=www.digicert.com; CN=DigiCert SHA2 Ext
ended Validation Server CA
*  SSL certificate verify ok.
* Curl_http_done: called premature == 0
* Connection #0 to host www.github.com left intact
pi@raspberrypi:~/AutoUpdatePlugin $
```

Figure E.4 Testing SSL/TLS Certificate Validity - Server