Security in IOT Devices

Final report

Submitted for the BSc in

Computer Science

April 2021

by

Kai Christopher Michael Tindall

Word count: 8648

Abstract

This project creates a secure environment for IoT devices to transmit their data to a central control server. This server was then supposed to publish that information to an API that the end user can access, however time constraints meant that it needed to be accessed through the command line with this server. This was done using the open-source Crypto++ library for the cryptological functionality. The project is one small step to becoming a fully working version of the aim, with the only change needing to be the device reporting real status data instead of test data.

Acknowledgements

*I’d firstly like to acknowledge my partner for her enormous support throughout my entire time at University and exhibiting enormous patience with me through the many late nights working.*

*I would also like to acknowledge the support of the Crypto++ Library contributors for their useful online resources and the free and open nature of their project. Without them this project would have taken significantly longer.*

Contents

[Abstract i](#_Toc70542636)

[Acknowledgements ii](#_Toc70542637)

[1 Introduction 3](#_Toc70542638)

[1.1 Background to the project 3](#_Toc70542639)

[1.2 Report Organisation 3](#_Toc70542640)

[1.3 Aims and objectives 3](#_Toc70542641)

[2 Literature review 5](#_Toc70542642)

[3 Requirements 6](#_Toc70542643)

[3.1 IoT Device Requirements 6](#_Toc70542644)

[3.2 Command and Control Server Requirements 6](#_Toc70542645)

[3.3 Client Requirements 7](#_Toc70542646)

[3.4 Security Requirements 8](#_Toc70542647)

[4 Design 9](#_Toc70542648)

[4.1 Software design 9](#_Toc70542649)

[4.1.1 System Design 9](#_Toc70542650)

[4.1.2 Security Library 9](#_Toc70542651)

[4.1.3 Message Library 1](#_Toc70542652)

[4.1.4 Command and Control Server 1](#_Toc70542653)

[4.1.5 Device Driver 3](#_Toc70542654)

[4.1.6 Public Information Registrar 4](#_Toc70542655)

[4.1.7 Client 5](#_Toc70542656)

[4.2 API Design 6](#_Toc70542657)

[4.2.1 GET Requests 6](#_Toc70542658)

[4.2.2 POST Requests 7](#_Toc70542659)

[5 Implementation and testing 8](#_Toc70542660)

[5.1 Implementation 8](#_Toc70542661)

[5.2 Testing 12](#_Toc70542662)

[6 Evaluation 16](#_Toc70542663)

[6.1 Evaluation Against Requirements 16](#_Toc70542664)

[6.1.1 IoT Device Requirements 16](#_Toc70542665)

[6.1.2 Command and Control Server Requirements 17](#_Toc70542666)

[6.1.3 Client Requirements 18](#_Toc70542667)

[6.1.4 Security Requirements 19](#_Toc70542668)

[7 Conclusion 20](#_Toc70542669)

[7.1 What has the project achieved? 20](#_Toc70542670)

[7.2 Methodology 20](#_Toc70542671)

[7.2.1 Work Management 20](#_Toc70542672)

[7.2.2 Time Planning 20](#_Toc70542673)

[7.3 Personal Reflection 21](#_Toc70542674)

[7.4 Further Work 21](#_Toc70542675)

[References 23](#_Toc70542676)

[Figure 1 System Design 10](#_Toc70542686)

[Figure 2: Security Library Class Diagram 1](#_Toc70542687)

[Figure 3 Message Library Class Diagram 1](#_Toc70542688)

[Figure 4 Command Server Class Diagram 2](#_Toc70542689)

[Figure 5 Command Server Message Sending Sequence Diagram 2](#_Toc70542690)

[Figure 6 Providing Authentication Through Signatures 3](#_Toc70542691)

[Figure 7 Device Driver Class Diagram 4](#_Toc70542692)

[Figure 8 Public Information Registrar Class Diagram 4](#_Toc70542693)

[Figure 9 Client Class Diagram 5](#_Toc70542694)

[Figure 10 Client User Interface Wireframe 6](#_Toc70542695)

[Figure 11 GitHub Boards Documentation Project 9](#_Toc70542696)

[Figure 12 GitHub Boards Closing Issues with Pull Requests 9](#_Toc70542697)

[Figure 13 Example CTest Definitions 12](#_Toc70542698)

[Figure 14 Arrange Act Assert 13](#_Toc70542699)

[Figure 15 Mock Definition 13](#_Toc70542700)

[Figure 16 Using the Mock as a FileIO Object 13](#_Toc70542701)

[Figure 17 Creating the Test Fixture 14](#_Toc70542702)

[Figure 18 Using the Test Fixture 14](#_Toc70542703)

# Introduction

## Background to the project

With the availability of cheap single board computers being at an all-time high, more people than ever are turning to them to create their own smart home devices. With these DIY smart home devices, most people have two options: Integrate the solution with an open-source platform like Home Assistant, or don’t and check on the status of the device manually. Both options aren’t particularly great. The average hobbyist will struggle to modify their solution to work with a solution like Home Assistant, and just leaving their project and checking the status manually becomes worse the more devices you need to check.

This project is supposed to help empower regular DIY-ers to be able to create as many devices as they see fit and not have to worry about having to check on them individually, or that someone could be listening in on the network traffic to gather some sensitive data.

## Report Organisation

This report will be organised in such a way that it will read chronologically, from first beginning the project, defining objectives, researching, defining requirements, all the way through to testing an evaluation.

## Aims and objectives

The project’s aim was to create a system to allow users to monitor the status of multiple IoT devices in an easy-to-use manner.

The project also had some primary and secondary objectives.

|  |  |
| --- | --- |
| **Primary Objective** | **Justification** |
| To create a driver software for the IoT devices to be able to talk to the command and control server. | The driver software allows the IoT devices to communicate to the command and control server securely using encryption. This is needed for data to not be usable if intercepted in transit by a hacker. |
| To create a command and control server to manage and communicates to the various IoT devices. | By having the command and control server do all the management of the IoT devices it lightens the load of the client, which means it will then be “dumber” and therefore easier to develop for end users. |
| To create and document an API that is exposed by the command and control server. | Using an API to transmit and receive commands and data between the client and the command and control server means that each implementation of the client doesn’t need to make sure it handles the business logic correctly and means that data will not be different between clients. |
| To create an example client that will communicate to the command and control server via the API. | This will allow users to be able to see and use the system without having to invest time developing a client for their chosen platform first. |

|  |  |
| --- | --- |
| **Secondary Objective** | **Justification** |
| To create three interchangeable secure communication methods that the user can choose from within the client. | This will then allow the user to pick and choose how they want their data to be encrypted depending on their individual circumstances. |
| Evaluate the best method of encryption for speed. | This will then allow me to recommend the chosen method to users who need a lot of data encrypted, but don’t necessarily need it to be the most secure. |
| Evaluate the best method of encryption for security. | This will then allow me to recommend the chosen method to users who need the highest level of security available for their data and don’t necessarily care if it makes it a bit slower. |

# Literature review

The main existing products within this field are “Home Assistants”. This includes products such as the Google Home, Amazon Alexa, the open-source alternative Home Assistant. The first two products mentioned share a common negative, they’re owned by big technology companies who’d like to keep their products private, which restricts users from easily integrating their DIY devices into these products.

These proprietary products can also leak your personal data. There have been incidents where these systems have been subject to data leaks (Burgess, n.d.), or they deceive you by keeping data it told you was deleted (Kelly, 2019). Big problems for anyone wanting to keep their private data private.

The open-source solution, Home Assistant is a better alternative than the proprietary products, however it’s still difficult to write in your own device without using some work around by buying other products.

These products also don’t offer the level of transparency that this project would be offering, in this project the user can alter the system to use any kind of encryption they’d like, using the security library and the configuration file. The user can also see where the centre of trust is within the system with the public information registrar, and check the keys given to the programs match up with what’s being received. This way manual checks can be done just in case the user suspects foul play. These features are simply not found on any of the other systems.

The ease of adding DIY devices needs to also be addressed as none of the other systems have easy ways for the devices to be added natively and report whatever status needs to be reported. Again this is just unheard of in these products.

# Requirements

## IoT Device Requirements

|  |  |
| --- | --- |
| **Requirement ID** | **Description** |
| SITD-DD-001 | The IoT device shall run the driver software on start-up. |
| SITD-DD-002 | The IoT device shall be able to transmit messages to the C&C server. |
| SITD-DD-003 | The IoT device shall be able to receive message from the C&C server. |
| SITD-DD-004 | The IoT device shall be able to decode encrypted messages. |
| SITD-DD-005 | The IoT device shall be able to encode messages. |
| SITD-DD-006 | The IoT device shall be able to sign messages. |
| SITD-DD-017 | The IoT device shall be able to check the authenticity of messages. |
| SITD-DD-007 | The IoT device shall be able to perform a key exchange with the C&C server. |
| SITD-DD-008 | The IoT device shall reject unauthorised communication. |
| SITD-DD-009 | The IoT device shall not crash if it encounters erroneous traffic. |
| SITD-DD-010 | The IoT device shall reply with a standard error report when it encounters erroneous traffic. |
| SITD-DD-011 | The IoT device shall reply to any authorised traffic with an acknowledgement message. |
| SITD-DD-012 | The IoT device shall wait for an acknowledgement message before sending repeat messages. |
| SITD-DD-013 | The IoT device should wait for 100 milliseconds before attempting repeat messages. |
| SITD-DD-014 | The IoT device shall be able to receive a workload to be deployed. |
| SITD-DD-015 | The IoT device shall be able to receive data to pass through to the running workload. |
| SITD-DD-016 | The IoT device shall be able to transmit data from the running workload back to the command and control server. |
| SITD-DD-017 | The IoT device shall receive the encryption method being used by the C&C server. |

## Command and Control Server Requirements

|  |  |
| --- | --- |
| **Requirement ID** | **Description** |
| SITD-CC-001 | The C&C server shall run on start-up. |
| SITD-CC-002 | The C&C server shall poll the network to discover new devices to connect to. |
| SITD-CC-003 | The C&C server should poll the network every 10 seconds. |
| SITD-CC-004 | The C&C server shall attempt to connect to a device when it receives the command to from the client. |
| SITD-CC-005 | The C&C server shall expose an API that the client can connect to. |
| SITD-CC-006 | The C&C server shall be able to encrypt traffic. |
| SITD-CC-007 | The C&C server shall be able to unencrypt traffic |
| SITD-CC-008 | The C&C server shall be able to perform a key exchange. |
| SITD-CC-009 | The C&C server shall be able to sign messages. |
| SITD-CC-010 | The C&C server shall be able to check the authenticity of messages. |
| SITD-CC-011 | The C&C server shall be able to transmit encrypted traffic. |
| SITD-CC-012 | The C&C server shall be able to receive encrypted traffic. |
| SITD-CC-013 | The C&C server shall reject unauthorised communication. |
| SITD-CC-014 | The C&C server shall not crash if it encounters erroneous traffic. |
| SITD-CC-015 | The C&C server shall serve API requests it receives from clients. |
| SITD-CC-016 | The C&C server may have the option to change the encryption method. |
| SITD-CC-017 | The C&C server shall act as a central authority for public keys. |

## Client Requirements

|  |  |
| --- | --- |
| **Requirement ID** | **Description** |
| SITD-CL-001 | The Client shall be able to perform a key exchange with the C&C server. |
| SITD-CL-002 | The Client shall be able to display information received from the C&C server. |
| SITD-CL-003 | The Client shall allow the user to input data to send to an IoT device. |
| SITD-CL-004 | The Client shall allow the user to send a workload to an IoT device. |
| SITD-CL-005 | The Client shall have a GUI. |
| SITD-CL-006 | The Client shall be able to sign encrypted traffic. |
| SITD-CL-007 | The Client shall be able to encrypt traffic. |
| SITD-CL-008 | The Client shall be able to decrypt traffic. |
| SITD-CL-009 | The Client shall be able to send encrypted traffic over an API. |
| SITD-CL-010 | The Client shall be able to receive encrypted traffic over an API. |
| SITD-CL-011 | The Client shall request data from the C&C server. |
| SITD-CL-012 | The Client should request data from the C&C server every five seconds. |
| SITD-CL-013 | The Client should give the user an option to change the amount of time the client waits to request data from the C&C server. |
| SITD-CL-014 | The Client shall reject unauthorised communication. |
| SITD-CL-015 | The Client shall not crash if it encounters erroneous traffic. |
|  |  |

## Security Requirements

|  |  |
| --- | --- |
| **Requirement ID** | **Description** |
| SITD-SEC-001 | All traffic shall be encrypted using a reputable method. |
| SITD-SEC-002 | All encryption methods shall be written by qualified reputable external authors. |
| SITD-SEC-003 | Each message shall be signed by the author. |
| SITD-SEC-004 | Encryption functionality shall be written as a library to enable reuse. |

# Design

## Software design

### System Design

Diagram

Description automatically generated

Figure System Design

The overall system design shows the flow of data through the system and how it’s getting there. Most of the time internally (behind the API) all communication will be through encrypted TCP communications. The only time the communications will not be encrypted is when programs are exchanging keys to set up an encrypted line.

The API is the main method of communication into the system for the client. This will be in the form of an HTTP API, this allows the client to take almost any form, including a website.

### Security Library

The pivotal component to the project was the security library. This library needed to be designed with interfaces in mind to ensure that other encryption methods could be implemented and used quickly where the users needed it.

In order to achieve this without having to change other software to adapt to these changes, interfaces were designed along with a factor so that a configuration could be passed in and the appropriately configured service could be received regardless of what methods the user wished to use.

This library is intended to be used as a wrapper to the Crypto++ library, making it easier for the other programs to use security features without having to alter the code base to switch the type of method it uses.

A configuration generator was also included with file input/output options so that the user will be able to write their own configuration in whatever file format there is a matching file input/output method for. The file input/output class is also based on an interface so can be swapped for a different method very easily, should it need to be.

Diagram

Description automatically generated

Figure : Security Library Class Diagram

### Message Library

The message library needs to exist so that each program has a reference to how the data coming through is organised. There also needs to be some generic messages within the library otherwise it will result in a lot of bloat for every single use case of a message.

The StartTrans message is a special message sent from the device driver to start the transaction, this gives the server a chance to send back any special initial values that it may need to.

The rest of the key exchange procedure is done using the request and response messages. This is agreeing a symmetrical key to be able to use EncryptedMessages. Once a symmetrical key is agreed, then EncryptedMessages can start being used. The device driver uses these messages to send regular status updates mainly, and the server uses these messages to send commands like sending over a new workload.

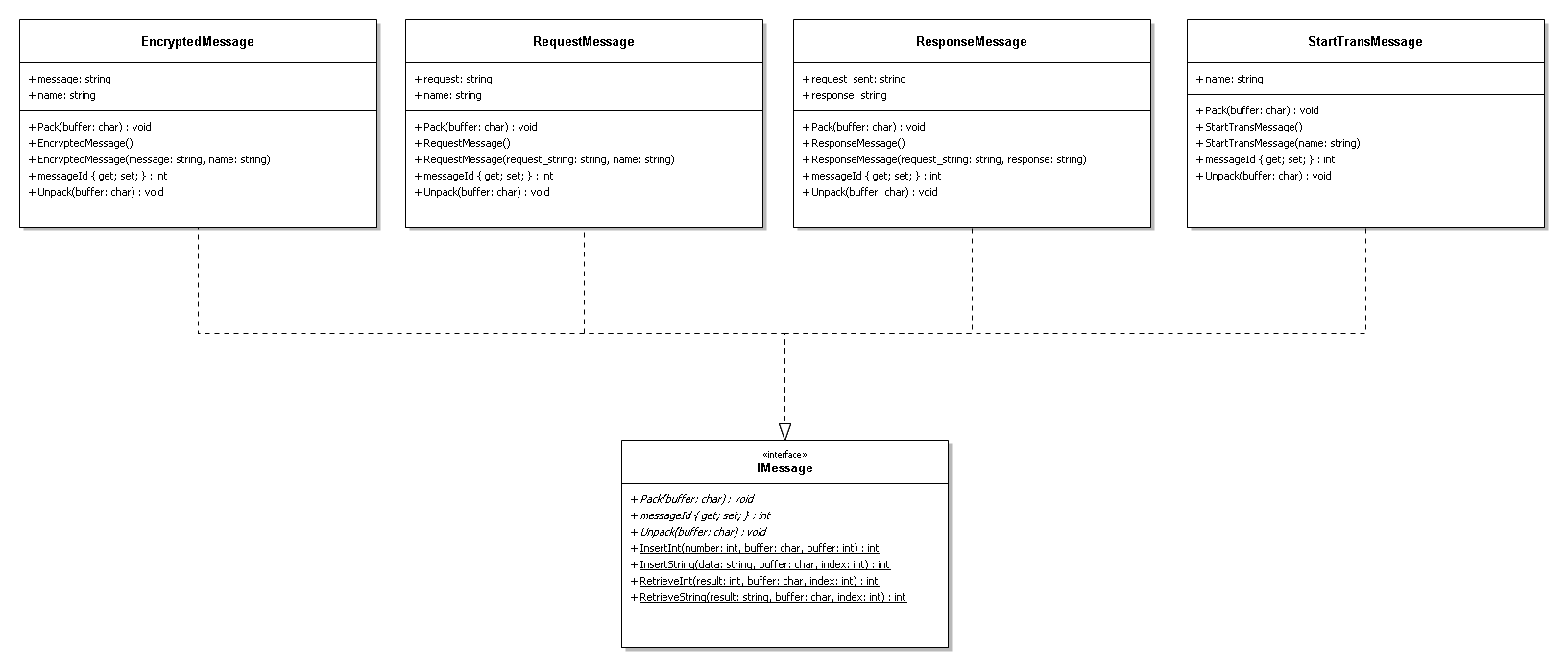


Figure Message Library Class Diagram

### Command and Control Server

The command server binds everything together and so is very important. However, it is also quite simple in concept. It only needs to handle connections coming through to it, which the actual connection can be handled with a separate class, and then just provide responses to requests.

There also needs to be some control functions for the main function to utilise. These are such as the IsRunning function which allows the main function to create a while loop and while the command server doesn’t receive a stop order, that while loop shall continue to receive incoming sockets and pass them through to the command server to handle in a separate thread.

Graphical user interface

Description automatically generated with medium confidence

Figure Command Server Class Diagram

When the command server sends out an encrypted message, the command server will encrypt and sign the data. This provides the trust that it has came from the command server with the signature, and the security that nobody else has seen it through using that encryption.

Diagram

Description automatically generated

Figure Command Server Message Sending Sequence Diagram

Diagram

Description automatically generated

Figure Providing Authentication Through Signatures

### Device Driver

The device driver is the software that will run on the IoT device. For the device to remain as responsive as possible, the device includes a message queue that different threads on the device driver can add to that will be processed continually by a sending thread. The message queue will also need a mutual exclusion lock to ensure synchronisation between threads, but this was left from the design for brevity.

Table

Description automatically generated

Figure Device Driver Class Diagram

### Public Information Registrar

The public information registrar acts as a central authority, similar to a certificate authority that’s used in web technologies. This tries to prevent an attack where the malicious actor tries to pass off their own public key as someone else’s. In the event they are successful, it means they can act as a man-in-the-middle and reveal information very easily.

Any program can connect to this registrar, and the registrar handles their request to either register a public key or request the public key with some filter parameters. Each program should have the public key of the registrar given to them so that the same attack could not happen where a malicious actor doesn’t pretend to be the registrar. This way the project can ensure that connections are genuinely between the correct parties.

A picture containing table

Description automatically generated

Figure Public Information Registrar Class Diagram

### Client

The client is the users main access point into the system and needs to gather all the information it can from the command server. Because the client is a graphical application too there is a graphics handler class that will deal with everything that needs to be displayed. The graphics handler has a data store object too that stores all the information that the graphics handler uses when it’s displaying the screen.

The API handler is a class specifically used to talk to the command server. When it gets the information back from the command server, it will parse it into the device data structure that represents the appropriate devices and update the datastore, which will in turn update what gets displayed onto the screen.

Diagram

Description automatically generated

Figure Client Class Diagram

This mock-up wireframe of the client illustrates how the client will look. This will be handled by the graphics handler class and all the status, workload, and defined commands will come from the datastore.

The aim of this design was to be quite minimal, not trying to add too much for the user, as to not over complicate it. All the devices the command server knows about is listed on the left, and if the user was to click on one of them it brings up the information panel about it. The information panel tells the user the devices current status, shows what workload the device is running and a button to upload a new one, and a section to send off predefined commands.

A picture containing diagram

Description automatically generated

Figure Client User Interface Wireframe

## API Design

### GET Requests

GET requests are HTTP requests that aim to get data from the server.

|  |  |  |
| --- | --- | --- |
| **Request Name** | **Parameters** | **Description** |
| GetServerPubKey | N/A | Gets the server’s public key from the public information registrar. |
| GetDeviceNames | Encyption Key | Gets the server’s list of devices connected. The client sends over their symmetrical encryption key, encrypted with the server’s public key. And the server sends back the data encrypted with the client’s key. |
| GetDeviceStatus | Encryption Key Device Name | Gets the status of a single device from the server. Request is encrypted with server’s public key and server encrypts response with encryption key provided. |

### POST Requests

POST requests are HTTP requests that aim to give data to the server.

|  |  |  |
| --- | --- | --- |
| **Request Name** | **Parameters** | **Description** |
| PostDeviceCommand | Device Name Device Command Command Parameters | Tells the server to execute a predefined command on the device. |

# Implementation and testing

## Implementation

First, C++ was chosen as the language the project was to be written in. C++ was chosen because being cross-compatible on as many platforms as possible was very desirable due to not knowing what kind of platform the user would want to use the project on. Therefore, C++ was chosen over something like C#, which is a more modern language, but it’s only viable on Windows platforms. C++ was chosen over C because it has a lot more modern features like strings, classes, and namespaces. C++ can also use C functions very easily too so if needed be, C could be mixed in.

Now the language was decided, the project needed to be held in source control. Git was chosen to be the source control method used because it’s the most popular method (Slant, n.d.) and the project developers have extensive experience using it. GitHub was used as a remote repository because it’s free and has an extensive list of features like pull requests and actions. Competitors like GitLab were not considered because they’re a paid service.

Visual Studio code was chosen as the text editor to write the project in, this was chosen mainly for its lightweight nature. This was important since development was done in a Linux/GNU virtual machine, in order to be as close to the final target (Raspberry Pi) as possible. The other main option, Eclipse, was ruled out for this reason, because Eclipse is famous for it being very resource intensive.

Because one of the major reasons for picking C++ was to try to get the project to be very platform independent, CMake was chosen as a build configuration manager. CMake allows the project to define certain build parameters and when the user runs CMake it will compile the project using what’s best for the user, meaning the developers of the project don’t need to write complex make files.

Encryption was of course, a big part of the project. Because security is famously very difficult to write, and the best practice is to use a library (Cogswell, 2015), the project needed a library to use. Crypto++ was chosen as that library because of the extensive list of other projects that use Crypto++ (Crypto++, 2019).

Unit testing was decided to be used too, which also needed a library as it would take too much effort to write a custom framework specifically for this project. The googletest framework was chosen to be the unit testing framework used in the project. This framework was chosen over other frameworks such as the boost unit testing framework because the developers have had unpleasant times using the boost framework in the past, and the googletest framework was easy to use as it integrated with CMake well using the CTest feature, and was also easy to write.

Development was then started on the project. GitHub boards were used to assign work items, each time work started on a new work item a new branch was created to work on that specific item, and then when that work item was completed, a pull request was created that closed that specific work item.

Graphical user interface, text, application

Description automatically generated

Figure GitHub Boards Documentation Project

A picture containing text, monitor, screenshot, black

Description automatically generated

Figure GitHub Boards Closing Issues with Pull Requests

PugiXML was chosen to be the file IO library, this was since it seemed easy to use and was recommended by another developer who’d used it in a previous C++ project. The documentation was clear on how to use it, so it was implemented very easily and quickly.

The security library was implemented very close to the design and so was therefore very quick to write, although it did take longer than expected due to the complexity of the Crypto++ library. If this was to be done again, more time should be allocated when working with complex libraries.

The security library was thoroughly tested using automated unit testing through the google test framework. Every functional part of the library like the services had their own test suite dedicated to them, which could be easily expanded upon by just adding to the appropriate file.

By the time the security library was finished, the project was already quite far behind schedule. This was due to a few reasons, first designing the project was a lot more difficult than imagined, so a lot of time was used designing. This was due to the developer not having much experience designing software properly, so it was a big learning opportunity however used a lot more time than budgeted. Secondly, the schedule was very optimistic when it came to writing the implementation for the components, it took a lot longer than the week estimated per component. This should have been more obvious when planning due to the developer having limited experience with C++.

This led to the need to rescope the project. The decision was made to rescope the project to include a command server and device driver, with the public information registrar not being implemented currently but with it still in mind when writing the project. The client program was also left out unless there would be sufficient time to write an adequate one, with the replacement user interaction being a command line interface. The device driver would also for the time being only report fake statuses however would be written in such a fashion as to easily allow the device driver to report real statuses from a running workload.

The command server development began after the security library, with the decision made to write the message library as development went on as the programs needed the individual messages. A mix of reading documentation both from online sources, blogs, online tutorials, and command line ‘man’ (manual) pages on the relevant TCP functions led the development of the TCP server aspect of the command server with fast results. After that it was on to writing the command server class which utilised the TCP server class to receive incoming clients and pass them onto a command server function running within a separate thread. Threading was utilised within the server as without it only one connection could be held at one time, which is obviously not very good for a server.

A separate function was written in the command server to handle an incoming message from the connected socket. This function would use the static function ReceiveInt from the message library to extract the message ID of the incoming data, which could then be used within a switch statement and then unpack that raw data into the appropriate message structure.

The message library used the IMessage static functions to pack and unpack all the raw data. Everything consisted of integers and strings, and each message had a specific data order defined within the message class itself which put structure into the raw data. Integers were just stored as four ordinary bytes, but strings on the other hand needed to be put in a special sub-data-structure. The length of the string was first given as four bytes, which could then be used to read only the correct number of bytes which the string was within. This meant that the developer could easily insert and retrieve strings in raw byte arrays using two simple static functions.

When the command server successfully extracted a message, it would take an action defined in the handle message function, which, if it were a status message, would put the device’s status within an internal store that was shared across all threads. The message store was a map of strings to other maps, with the sub-map being a map of strings to strings which were the statuses of the device with the values. There was a single mutual exclusion lock for the entire data store, this means that the data store used coarse grain locking, which is fine for a few different threads trying to access it but will not scale well.

A better solution for this could be a finer grain of locking, which could be that each device’s status map could also be linked to a mutual exclusion lock, this would mean that if two separate devices are trying to update their status at the same time it wouldn’t matter. This improvement could massively improve individual status message turn around time in a bigger scale, say there’re ten thousand devices connected, each one of them in the current system needs to wait for the same mutual exclusion lock, whereas with the improvement they’d all have their own lock and would therefore dramatically reduce wait times.

The command line interface mentioned in the rescoping was implemented in its own thread, as not to disturb the command server in its purpose. The command line interface was given two commands, ‘List’ and ‘Show {Device}’, which would list the devices the command server currently had statuses for and would show the status of the device chosen. Thus, giving the user quick access to all device’s status from a single point.

The command server was tested throughout development using a telnet console. This allowed connections to be made and data to be easily sent before the device driver was created. Any issues with the command server after development had begun and progressed to a state where TCP packets were being transferred then was debugged using the prototype device driver.

The device driver software was to be written next and was frequently tested using the command server written. Some of the code from the TCP server written for the command server was recycled and used in the device driver, with the main difference being that instead of listening for connections it was creating them. The server was created with threading in mind, this allowed it to have multiple things happening at once that could each communicate with the command server. This also meant that a lot of synchronisation techniques were required to keep everything in line and prevent data races.

The most common, of course, were mutual exclusion locks. These are the most basic of synchronisation methods and a few were utilised to place critical sections where important data was accessed. Atomics were utilised extensively as flags for while loops in threads, as the client would need to be stopped gracefully if a premature exit were required. Atomics provided an easy way to achieve this as no locking of waiting needed to be used.

As mentioned above, when stopping prematurely the device driver would need to be stopped gracefully as to not cause long lasting port and IP problems for the device. Therefore, the SIGINT signal was intercepted in the code to instead of its normal behaviour, call stop on the device driver object.

The stop function would set the atomic flag to the false value to allow the threads to exit and then the main thread would wait for a join for all threads until closing the socket and then exiting the main program.

The most complex synchronisation method used by far was a conditional variable. A conditional variable can best be described as kind of a ‘smart lock’. It allows a thread to wait until a variable meets a certain criterion and the conditional variable has been notified. This was especially useful when a thread would send a specific request to the server through the message queue and need to wait on the response. An example of where this was used was within the key exchange thread where the thread needed to wait on the server sending over its public part of the key.

This is where the current implementation of the project currently stands. The device driver is a very strong base and only really requires the feature of getting statuses from a real program, which should be easy enough to achieve. The command server has a basic version of the API, albeit only in the command line form, so this would need expanding to instead of being commands inputted from the console be commands inputted over a HTTP connection. The public information registrar was never implemented however can be easily inserted into the current state of the project. The client was never implemented but due to the open nature of the client a suitable client could be whipped up quickly using a framework like C#’s WPF. However the security library is fully operational and able to be expanded with different methods incredibly easy without changing the source code in any other program.

## Testing

Throughout development of the security library and the message library, automated unit testing using the GoogleTest framework. Each functional component got its own test suite and was defined in the CMake CTest functionality as a test executable.

Text

Description automatically generated

Figure Example CTest Definitions

The “Arrange, Act, Assert” methodology was used to structure the unit tests, this is a widely used framework that helps to write good unit tests (Andy Knight, 2020). It helps to make sure that the test sets up, tests something, and then reports on that test, and nothing else.

Text

Description automatically generated

Figure Arrange Act Assert

Mocks were also used in some tests where other functionality was needed, but not needing to test too much at once. This can lead to misleading tests, because if the depended-on functionality breaks, then the test will break. This would then look like the tested functionality is breaking, even though it’s not, it’s just the test failing because of functionality unrelated to what the test should be testing.

Mocks fix this because mocks are supposed to look like the depended-on functionality, but instead of having actual logic, it is given a result to return. For example, if you have a validator service that checks certain conditions, then the mock version will not check anything, it will just return true or false, whatever the test needs it to return.

Text

Description automatically generated

Figure Mock Definition

Text

Description automatically generated

Figure Using the Mock as a FileIO Object

In the above unit test, the mock FileIO object is configured to repeatedly return the expected\_result structure when the ReadConfiguration method is called with “FakePath” as the argument.

Test fixtures were also utilised within some unit tests. Test fixtures allow for a SetUp and TearDown method to be called before the test begins and after the test finishes, respectively. This allows for repeated code to be eliminated from all tests that use that test fixture. This means that all tests are guaranteed to start from a common ground and cuts down on copying mistakes from one test to another.

Test fixtures in GoogleTest are easy to define, and the only change needed to use them is instead of using the TEST macro when defining a test, the TEST\_F macro is used. The name of the test fixture class also needs to be the same as the test suite name, which is how GoogleTest knows to use that test fixture for that test.

Text

Description automatically generated

Figure Creating the Test Fixture



Figure Using the Test Fixture

Manual testing was used during the development of the command server and the device driver. When writing the TCP server component of the command server, a telnet console was used to connect to the server and send data. This was possible all the way up to encrypted communication needed to be used. Telnet was perfect for this early testing because it’s unencrypted data being sent and it’s very easy to set up a console. The distribution of GNU/Linux used in development already came with a telnet command and the only arguments it needed were the port and IP address of the server, and then data could be easily sent from then on.

Test phrases were used during testing, to check how the logic was parsing the data. When the server received a certain test phrase, it would respond with a defined response that would prove that the server is sending and receiving TCP data correctly. This testing was done manually because it’s very difficult to test network traffic automatically.

When encryption became necessary in the development a prototype device driver needed to be created to use in testing. This prototype client was very simple and was just one file with a main function that did the entire testing in that function. This allowed the prototype client to test key exchanges and sending and receiving encrypted messages with the command server.

When the command server was working correctly with the network traffic and message passing, then the real device driver development could start. Manual testing was also conducted when developing the device driver as with it also being primarily network traffic it could not be automatically tested.

The already written command server was used to test the device driver when writing the device driver. When a new feature was added into the device driver, it would be tested against the command server to make sure the feature was working correctly and didn’t crash either the device driver or the command server.

When it came to testing the command server’s command line commands, manual testing was done to first ensure it works as it should with correct data, and then erroneous data was also given to the server to make sure nothing broke.

# Evaluation

The main aim of this project was to provide the users with a way to monitor the status of their IoT devices, which the final implementation of the project comes very close to doing that. The only thing that would need to change for that aim to be fully satisfied would be to have the device driver send over real status requests rather than predefined test ones.

This feature should be easy enough to implement if given the time. All that would need to happen is the device driver would need to read in from a pipe that the workload had written into, and then populate its status update with that data. This is estimated to take a week and a half to go from researching how to use pipes correctly, to implementing, to testing, to pulling it into the master branch of the project.

|  |  |
| --- | --- |
| **Primary Objective** | **Achieved?** |
| To create a driver software for the IoT devices to be able to talk to the command and control server. | Yes. Driver software was created that can communicate with the command |
| To create a command and control server to manage and communicates to the various IoT devices. | Yes. A command and control server was created that can communicate with the IoT devices. |
| To create and document an API that is exposed by the command and control server. | Half achieved, API was created (see 4.2), however extensive documentation detailing how to use it was never achieved. |
| To create an example client that will communicate to the command and control server via the API. | No. Only implementation is the command line interface on the command server. |

|  |  |
| --- | --- |
| **Secondary Objective** | **Achieved?** |
| To create three interchangeable secure communication methods that the user can choose from within the client. | No. However, because of how the security service was designed this would be very easy to implement. |
| Evaluate the best method of encryption for speed. | No. |
| Evaluate the best method of encryption for security. | No. |

## Evaluation Against Requirements

### IoT Device Requirements

|  |  |  |
| --- | --- | --- |
| **Requirement ID** | **Description** | **Achieved?** |
| SITD-DD-001 | The IoT device shall run the driver software on start-up. | No. |
| SITD-DD-002 | The IoT device shall be able to transmit messages to the C&C server. | Yes. |
| SITD-DD-003 | The IoT device shall be able to receive message from the C&C server. | Yes. |
| SITD-DD-004 | The IoT device shall be able to decode encrypted messages. | Yes. |
| SITD-DD-005 | The IoT device shall be able to encode messages. | Yes. |
| SITD-DD-006 | The IoT device shall be able to sign messages. | Half, it can very easily using the security service but doesn’t currently. |
| SITD-DD-017 | The IoT device shall be able to check the authenticity of messages. | Half, it can very easily using the security service but doesn’t currently. |
| SITD-DD-007 | The IoT device shall be able to perform a key exchange with the C&C server. | Yes. |
| SITD-DD-008 | The IoT device shall reject unauthorised communication. | Half, it can very easily using the security service but doesn’t currently. |
| SITD-DD-009 | The IoT device shall not crash if it encounters erroneous traffic. | Yes. |
| SITD-DD-010 | The IoT device shall reply with a standard error report when it encounters erroneous traffic. | Yes. |
| SITD-DD-011 | The IoT device shall reply to any authorised traffic with an acknowledgement message. | No. |
| SITD-DD-012 | The IoT device shall wait for an acknowledgement message before sending repeat messages. | No. |
| SITD-DD-013 | The IoT device should wait for 100 milliseconds before attempting repeat messages. | No. |
| SITD-DD-014 | The IoT device shall be able to receive a workload to be deployed. | No. |
| SITD-DD-015 | The IoT device shall be able to receive data to pass through to the running workload. | Yes. It can receive data to pass through but doesn’t currently pass it through to a workload. |
| SITD-DD-016 | The IoT device shall be able to transmit data from the running workload back to the command and control server. | No. |
| SITD-DD-017 | The IoT device shall receive the encryption method being used by the C&C server. | No. |

### Command and Control Server Requirements

|  |  |  |
| --- | --- | --- |
| **Requirement ID** | **Description** | **Achieved?** |
| SITD-CC-001 | The C&C server shall run on start-up. | No. |
| SITD-CC-002 | The C&C server shall poll the network to discover new devices to connect to. | No. Redundant requirement. Changed to device connecting to the command and control server. |
| SITD-CC-003 | The C&C server should poll the network every 10 seconds. | No. |
| SITD-CC-004 | The C&C server shall attempt to connect to a device when it receives the command to from the client. | No. |
| SITD-CC-005 | The C&C server shall expose an API that the client can connect to. | No. |
| SITD-CC-006 | The C&C server shall be able to encrypt traffic. | Yes. |
| SITD-CC-007 | The C&C server shall be able to unencrypt traffic | Yes. |
| SITD-CC-008 | The C&C server shall be able to perform a key exchange. | Yes. |
| SITD-CC-009 | The C&C server shall be able to sign messages. | Yes. |
| SITD-CC-010 | The C&C server shall be able to check the authenticity of messages. | Yes. |
| SITD-CC-011 | The C&C server shall be able to transmit encrypted traffic. | Yes. |
| SITD-CC-012 | The C&C server shall be able to receive encrypted traffic. | Yes. |
| SITD-CC-013 | The C&C server shall reject unauthorised communication. | No. |
| SITD-CC-014 | The C&C server shall not crash if it encounters erroneous traffic. | Yes. |
| SITD-CC-015 | The C&C server shall serve API requests it receives from clients. | No. |
| SITD-CC-016 | The C&C server may have the option to change the encryption method. | Yes. |
| SITD-CC-017 | The C&C server shall act as a central authority for public keys. | No. |

### Client Requirements

|  |  |  |
| --- | --- | --- |
| **Requirement ID** | **Description** | **Achieved?** |
| SITD-CL-001 | The Client shall be able to perform a key exchange with the C&C server. | None of the client requirements were achieved due to a client not being produced. |
| SITD-CL-002 | The Client shall be able to display information received from the C&C server. |
| SITD-CL-003 | The Client shall allow the user to input data to send to an IoT device. |
| SITD-CL-004 | The Client shall allow the user to send a workload to an IoT device. |
| SITD-CL-005 | The Client shall have a GUI. |
| SITD-CL-006 | The Client shall be able to sign encrypted traffic. |
| SITD-CL-007 | The Client shall be able to encrypt traffic. |
| SITD-CL-008 | The Client shall be able to decrypt traffic. |
| SITD-CL-009 | The Client shall be able to send encrypted traffic over an API. |
| SITD-CL-010 | The Client shall be able to receive encrypted traffic over an API. |
| SITD-CL-011 | The Client shall request data from the C&C server. |
| SITD-CL-012 | The Client should request data from the C&C server every five seconds. |
| SITD-CL-013 | The Client should give the user an option to change the amount of time the client waits to request data from the C&C server. |
| SITD-CL-014 | The Client shall reject unauthorised communication. |
| SITD-CL-015 | The Client shall not crash if it encounters erroneous traffic. |

### Security Requirements

|  |  |  |
| --- | --- | --- |
| **Requirement ID** | **Description** | **Achieved?** |
| SITD-SEC-001 | All traffic shall be encrypted using a reputable method. | Yes. |
| SITD-SEC-002 | All encryption methods shall be written by qualified reputable external authors. | Yes. |
| SITD-SEC-003 | Each message shall be signed by the author. | No. |
| SITD-SEC-004 | Encryption functionality shall be written as a library to enable reuse. | Yes. |

As illustrated above, although the project hasn’t hit all the requirements yet, it has hit a lot of them and was close to being in a state where all primary objectives would be completed, and all requirements satisfied.

# Conclusion

## What has the project achieved?

The project has achieved two of the primary goals, and half of another. Although the project isn’t currently in the position that it should have been in by now, it’s still in a good position. A lot of the major work has been done and to progress towards finishing it completely it wouldn’t take a lot of work.

It has achieved the aim of providing a simple way for the user to access the status of the IoT devices, even if for the moment those statuses are just test ones. However, currently signatures do not take place which means the authenticity of the messages isn’t validated. The security service does have all the functionality to sign and verify messages though so it would be very easily implemented.

Speaking of security, the project has achieved writing a library that can security implementation behind interfaces, so that all the user needs to do is load a different config and the security library will do the rest. This means that the code that uses the library doesn’t need to change in order to update it to use new encryption methods. This is great for particularly large code bases where it may not be clear where would need changing if the encryption method was to need changing, and could very easily lead to a mismatch where one part of the program uses method A where another part uses method B.

## Methodology

### Work Management

In the software development plan, it mentions that the work will be split into sprints using the SCRUM methodology. This did not work very well with only one developer, so the decision was made to work with a more Kanban approach of putting work on the project board and just working through them one by one. This worked great for this project because it requires a lot less management than the scrum methodology, which when considering time was always short, was a big bonus.

The weekly and monthly reports also seemed like a good idea, but in the end they faded away because they just weren’t providing enough value to the project to make the time investment worth it.

### Time Planning

Time planning was done through a Gantt chart at the beginning of the project. This kind of time planning wasn’t very accurate because of the limited experience in time planning before, which lead to task times getting mis-judged and falling behind schedule.

In the future, with this experience time planning in this manner should go better and allow more time to be allocated to a work item. This experience will ensure that suitable time is allowed for research, implementation, testing, and reviewing; instead of just implementation as the plan for this project did.

Regular check ups on the time schedule should also be incorporated in the future, with a chance to consult the plan, change it if necessary, and progress. Appropriate time should be allocated for doing just this as well, because it isn’t a trivial job, it takes time to properly reflect on and assess the situation, and work on an action plan to put into place to help.

Extra time was added into the project to allow for setbacks, however this did little to combat the glaringly bad estimations, so this extra time dried up very quickly.

## Personal Reflection

This project has really progressed me as a software engineers and has allowed me to get my hands dirty in a lot of valuable skills, such as automated unit testing (including test fixtures and mocks), working with a Kanban board to track work, using a configuration tool like CMake to configure how the project should be built cross-compatibly.

I also managed to really practise my C++ skills as before this project I’d only ever worked on maintaining C++ projects, never actually written one from scratch myself which has really helped cement the knowledge for later use.

My knowledge of good security practices has also increased as a result of this project, having to work closely with a security library and implement classed that used these methods.

My estimation of how long work will take has also improved dramatically. As this was the first time I’ve had to actually estimate timeframes for work I missed out planning for some important aspects of software engineering. I only actually budgeted the implementation stage of development, and completely neglected the research, testing, and review stages of development, which are in their own right very important and require time to be dedicated to them.

This project has exposed me to working professionally and conducting myself in an efficient manner by exposing me to a lot of things I’ve been able to ignore in the past due to repeatably “hacking” solutions together and relying on my skills as a programmer to carry me through without any regard to proper planning, designing, testing, or reviewing.

This has been a pivotal moment for me in my software engineering career and going forward has inspired me to manage my projects and work properly.

## Further Work

Further work for this project is very well defined, it’s the work that hasn’t been done yet to finish the project. This includes altering the messages to require signatures, this shouldn’t take long at all as it is just adding to existing code. It is estimated this would take around a week and a half, which includes implementing, testing, and reviewing. The research for this functionality has already been done.

The public information registrar also needs writing from scratch, luckily though it’s a very simple component. It’s estimated this would take around three weeks to write comfortably, this includes writing the component, testing it, reviewing it, and integrating it into the current system, which would take a little bit of work but nothing too serious.

The API needs to be exposed from the command server, this will take a bit of work to write the component, however integrating it into the current system would be simple. It’s estimated this feature would take around four weeks. This would take this long for a couple of reasons, firstly a C++ HTTP library would need to be found and understood, and secondly, it’s a big component to write.

The client also needs to be written, which if done in C++ is estimated to take around eight weeks, because creating a graphical user interface is a brand-new concept to understand and then the logic, testing, and reviews on top of that. However, if the client were to be written in C# WPF, then the estimate would drop down to four weeks due to experience in that area.

The final work that would need to be completed would be getting real statuses from workloads on the device. This is estimated at two weeks as it’s not a massive feature and all the supporting code is already there.

To get the project to be fully complete then, it is estimated it would take around eighteen and a half weeks to write in pure C++, with that figure falling to fourteen and a half if C# WPF was chosen as the client interface.

References

Andy Knight, 2020. Arrange-Act-Assert: A Pattern for Writing Good Tests. Automation Panda. URL <https://automationpanda.com/2020/07/07/arrange-act-assert-a-pattern-for-writing-good-tests/> (accessed 4.27.21).

Burgess, M., n.d. Google Home’s data leak proves the IoT is still deeply flawed. Wired UK.

Cogswell, J., 2015. 5 Best Practices for Security in Open Source Development. Linux.com. URL <https://www.linux.com/news/5-best-practices-security-open-source-development/> (accessed 4.25.21).

Crypto++, 2019. Related Links - Crypto++ Wiki [WWW Document]. Crypto++ Website. URL <https://www.cryptopp.com/wiki/Related_Links#Commercial_Products_Using_Crypto.2B.2B> (accessed 1.20.21).

Kelly, M., 2019. Amazon confirms it holds on to Alexa data even if you delete audio files [WWW Document]. The Verge. URL <https://www.theverge.com/2019/7/3/20681423/amazon-alexa-echo-chris-coons-data-transcripts-recording-privacy> (accessed 4.28.21).

Slant, n.d. Slant - 14 Best version control systems as of 2021 [WWW Document]. Slant. URL <https://www.slant.co/topics/370/~best-version-control-systems> (accessed 4.25.21).