Keeping Data Secure in 2020

Ensuring data is kept safe

Jonah McElfatrick

CMP320: Ethical Hacking 3

BSc Ethical Hacking Year 3

2019/2020

Note that Information contained in this document is for educational purposes.

Executive Summary

This paper will demonstrate to the reader, how to keep data secure in the modern day. This will include how to transmit data securely through different methods as well as keeping data secure when being stored in a database or file system. It is important to understand the differences between secure transmission protocols and un-secure transmission protocols and how one can be more secure than the other. In this paper it can be seen the difference between secure and unsecure methods of transporting data. Storing data is just as important if not more important than transmitting data. Securing data when it is being stored is vital for good security measures. Stored data is usually the main focus for an attacker as it gives them more time to attempt to steal the data as the data is not been moved or transmitted.

A demonstration is carried out to show the difference between HTTP and HTTPS. A network protocol analyzer was used to capture traffic in transit over a network and therefore allows for a comparison between security methods used by both HTTP and HTTPS to be able to see which one would be considered more secure and why.

A piece of software called Kleopatra is used and demonstrated using both the command line interface and the graphical user interface in this paper as an alternative way to use the same secure encryption method that is used by HTTPS. This allows for a greater understanding of what happens when the data is being sent behind the scenes of a web browser.

As Kleopatra and HTTPS use the same encryption method, the mathematics behind the encryption method used by both methods is explored and explained to give the reader a more in depth understanding of how the data is being secured and how reliable and robust it is.

Storing data securely is also a large part of the data's life along with the transmission of data. Therefore in this paper, a comparison between different modern hashing algorithms has been carried out. This included researching any existing issues with the hashing algorithms, along with individual testing with an implemented password cracker to observe which hashing algorithms could possibly take longer and/or are harder to crack. It was observed that the MD5 hashing algorithm is no longer suitable for modern secure hashing of data.

+Contents

1 INTRODUCTION

1.1 BACKGROUND

In the 21st century, the world revolves around computers. From your average desktop computers, to smartphones to IOT (Internet of Things) devices such as remote-controlled lightbulbs, each one of these devices communicates with each other and transfers and stores data. In the modern day, there is data stored about almost everyone and everything on the planet. This can range from pricing of items on an online marketplace, what that odd-looking flower is in your back garden, who that actor was in your favorite movie, to more personal and sensitive information about people themselves. Taking online marketplaces as an example, most users that would be members of a marketplace website would have details such as their name, email, password and bank account details registered to that account. All of these details are sent across a network and then stored by the marketplace website company on their servers. In turn, this means that pressure is then placed onto the companies shoulders to keep this information safe from any malicious attacker, both in data transit and storage. One of the most common pieces of information that is considered sensitive information and is used for almost every kind of account is passwords. With passwords being used to access almost every form of modern account it is one of the main focuses of attackers to gain access to. This would in turn allow the attacker to try other accounts and repeat the process. Wired.com reported "Now, it seems, someone has cobbled together those breached databases and many more into a gargantuan, unprecedented collection of 2.2 *billion* unique usernames and associated passwords and is freely distributing them" (Greenberg, 2020) This indicates the need to ensure that sensitive information such as passwords are kept secure and are unreadable to any attacker that may gain access to them.

In previous years there was a lower required level of security, but as computers and technology have advanced and developed over the years, so has the increase in need for more complex and thorough security methods. Network traffic can now be captured with most modern devices and some free software. Lower levels of encryption and hashing methods are now outdated and can be cracked with consumer level hardware. With this possible, more in-depth and complex security methods have had to be developed and implemented to keep data secure from prying eyes.

There are many different ways to ensure that your data is kept safe when both being transported and stored. These methods range in difficulty to use and understand. Encryption can be used to obscure sensitive data being transported between devices. Encryption is the method of obscuring data by passing it through a mathematical algorithm. The most common method to encrypting data traffic is using HTTPS. This is due to it being the standard for transmitting data through a web browser to a web server. This is to keep the user's data safe. HTTPS uses the TLS (Transport Layer Security) protocol, formerly known as SSL (Secure Socket Layer), this protocol uses two different keys in an asymmetric key structure called RSA (Rivest-Shamir-Adleman) encryption. This method is also commonly known as public-key encryption. The two keys consist of a private key and a public key. The private key is linked to the owner, in this case the website, and is held on the webserver to decrypt incoming traffic to the server. The public key is distributed by the webserver to whoever is wanting to communicate with it. This allows the client to encrypt and send data securely to the server which will then decrypt the data.

However, even though this secures the data when being transmitted to the web server, when the data is being stored in the web server then a different method is required to secure the data. Hashing algorithms are a method of storing sensitive information inside of database's without being in a plain text form that is easy to read if accessed. This is carried out by passing in a value through the one-way mathematical algorithm. This means that the hash is very difficult, almost impossible to reverse. There are varying levels of complexity when it comes to hashing algorithms. Some of the lower level complexity algorithms, MD4, MD5 and a mid-range algorithm SHA-256 are among the most commonly known algorithms. MD4 and MD5 are commonly known to be used to verifying the integrity of downloaded files from the internet to ensure that the file has not been altered in transit. SHA-256 is the basic standard of what should be used to store sensitive information. There are more complex algorithms such as SHA-384 and SHA-512, these are more complex versions of SHA-256 and are considered more secure to use. These are analyzed and discussed later in the paper.

1.2 Aim

The aims of this project are:

- To allow the reader to gain an understanding of why keeping data hidden and stored correctly is imperative for secure data handling.
- To demonstrate different methods on how to securely transmit data over a network.
- To understand the mathematics behind RSA encryption used by HTTPS and Kleopatra.
- To demonstrate how to store sensitive data, such as passwords, securely and how attackers may try to break any attempt to obscure the data.

2 Procedure & Results

2.1 Overview of Procedure

In the following procedure, it will explain different methods in keeping data safe while being transmitted or being stored on a computer. This will include transmitting data through a web browser, transmitting data privately and then storing sensitive data such as passwords on a computer or database. The procedure contains:

- A demonstration of the difference between using HTTP and HTTPS when transmitting data over a web browser and how one is better than the other.
- A demonstration on how to use the Kleopatra software in conjunction with Gpg4win to send encrypted data through other methods than through a web browser.
- Both HTTPS and Kleopatra use RSA encryption, following the demonstrations on HTTPS and Kleopatra, the mathematics behind RSA encryption is explored and explained to give an understanding of how it is secure to encrypt data using this method.
- A comparison and demonstration between different hashing algorithms for securely storing data effectively and correctly. A password cracker implementation to demonstrate which hashes are easier to crack than others.

2.2 HTTP & HTTPS

HTTPS uses SSL certificates along with RSA encryption to ensure that the users' data is kept secure when being transmitted over a network. HTTP does not have any of these traits. HTTP uses no encryption to request and send data. This means that the user's data is sent over a network without being encrypted.

To show the difference between captured traffic using the two different protocols, HTTP and HTTPS, a piece of software called Wireshark was used. This allows for the packets on a network to be captured and possibly viewed depending on the protocol being used.

2.2.1 Using HTTP

The first step in analyzing HTTP packets is to navigate to a website using HTTP. In this case a local webserver and website were being used to demonstrate this. The website used can be seen in figure 1 below.

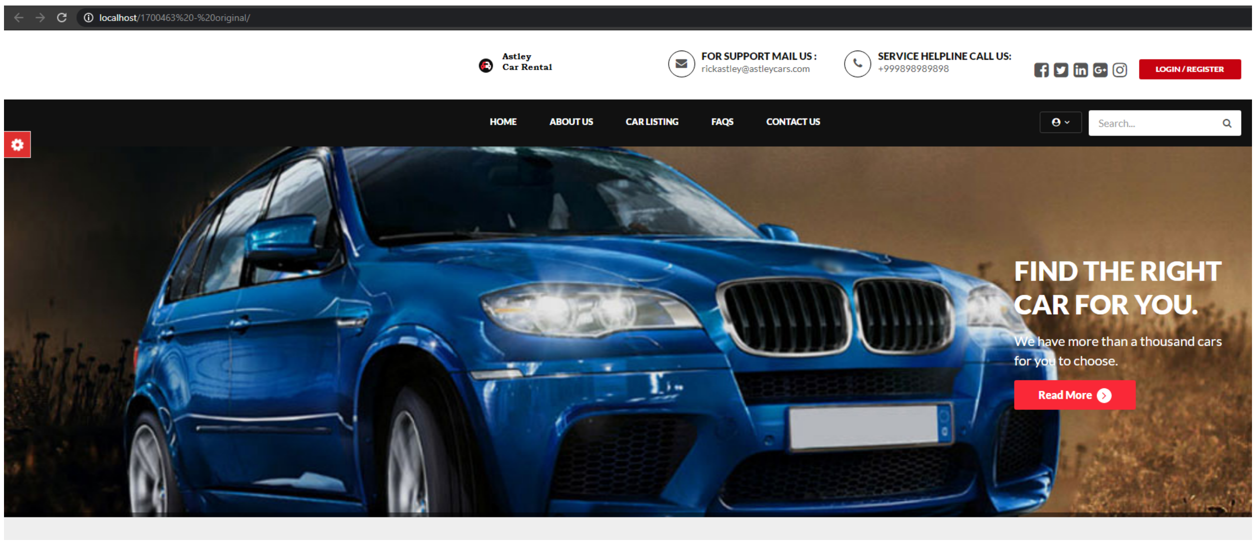
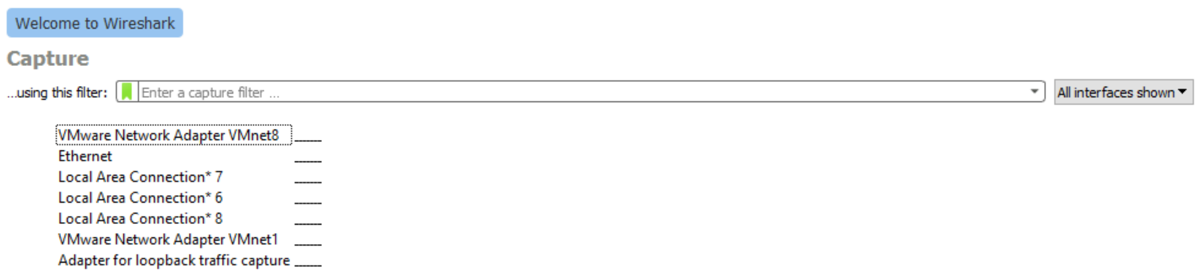

localhost/1700463%20-%20original/
Astley Car Rental
FOR SUPPORT MAIL US :
rickastley@astleycars.com
SERVICE HELPLINE CALL US:
+999898989898
LOGIN / REGISTER
HOME
ABOUT US
CAR LISTING
FAQS
CONTACT US
Search...
FIND THE RIGHT CAR FOR YOU.
We have more than a thousand cars for you to choose.
Read More

Figure 1: HTTP website

From here, Wireshark was launched. A list of connections can be seen on the home screen of Wireshark. Since the website that is being monitored is on the local machine, the loopback connection is the one we are interested in. This can be seen I figure 2 below.



Welcome to Wireshark

Capture

...using this filter: Enter a capture filter ... All interfaces shown

VMware Network Adapter VMnet8
Ethernet
Local Area Connection* 7
Local Area Connection* 6
Local Area Connection* 8
VMware Network Adapter VMnet1
Adapter for loopback traffic capture

Figure 2: Wireshark Home Page

Once the loopback traffic capture has been selected, then navigating back to the website. Login details are entered into the login portal. This can be seen in figure 3 below.

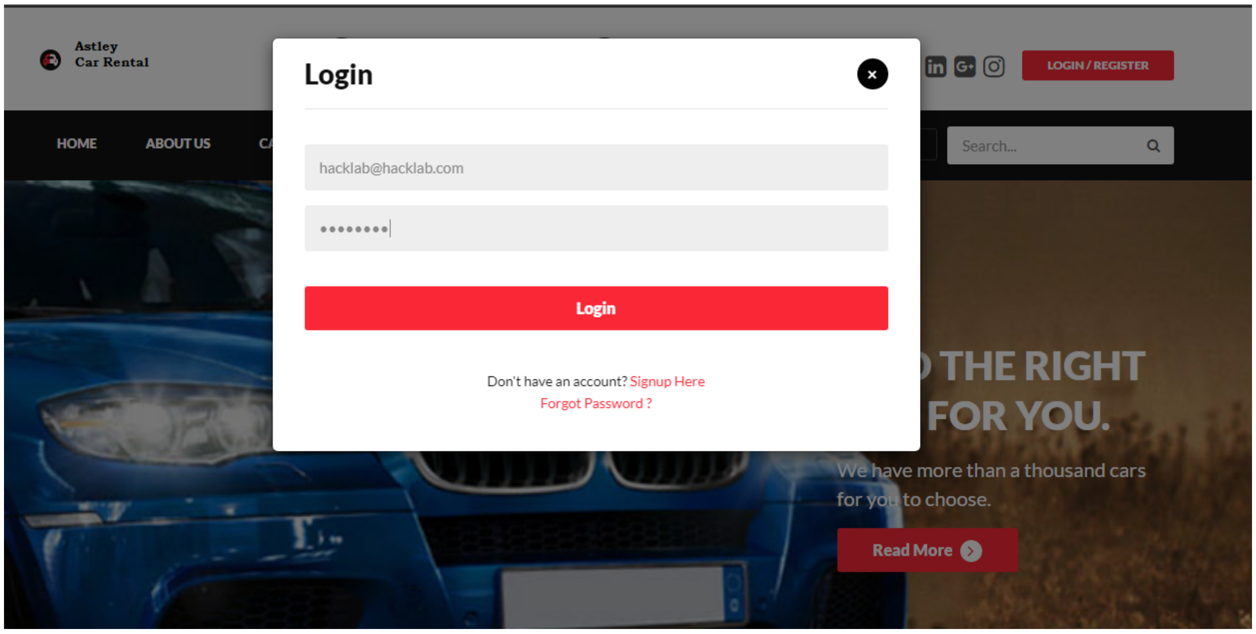
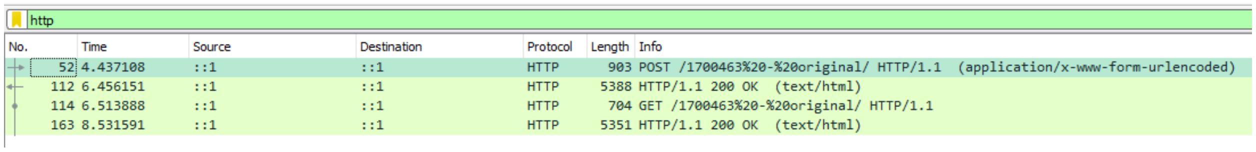


Figure 3: Login Portal

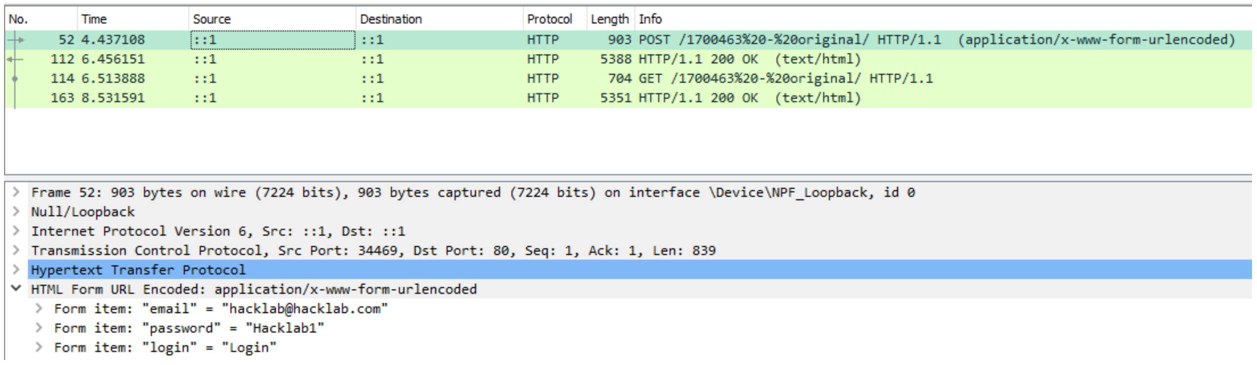
Going back to Wireshark, the traffic we are interested in is HTTP traffic. A filter can be placed on the captured packets. This can be seen in figure 4 below where the filter of 'http' was used.



```
http
No.   Time      Source  Destination  Protocol  Length  Info
  52  4.437108  ::1     ::1          HTTP       903 POST /1700463%20-%20original/ HTTP/1.1  (application/x-www-form-urlencoded)
 112  6.456151  ::1     ::1          HTTP      5388 HTTP/1.1 200 OK  (text/html)
 114  6.513888  ::1     ::1          HTTP       704 GET /1700463%20-%20original/ HTTP/1.1
 163  8.531591  ::1     ::1          HTTP      5351 HTTP/1.1 200 OK  (text/html)
```

Figure 4: Filter HTTP

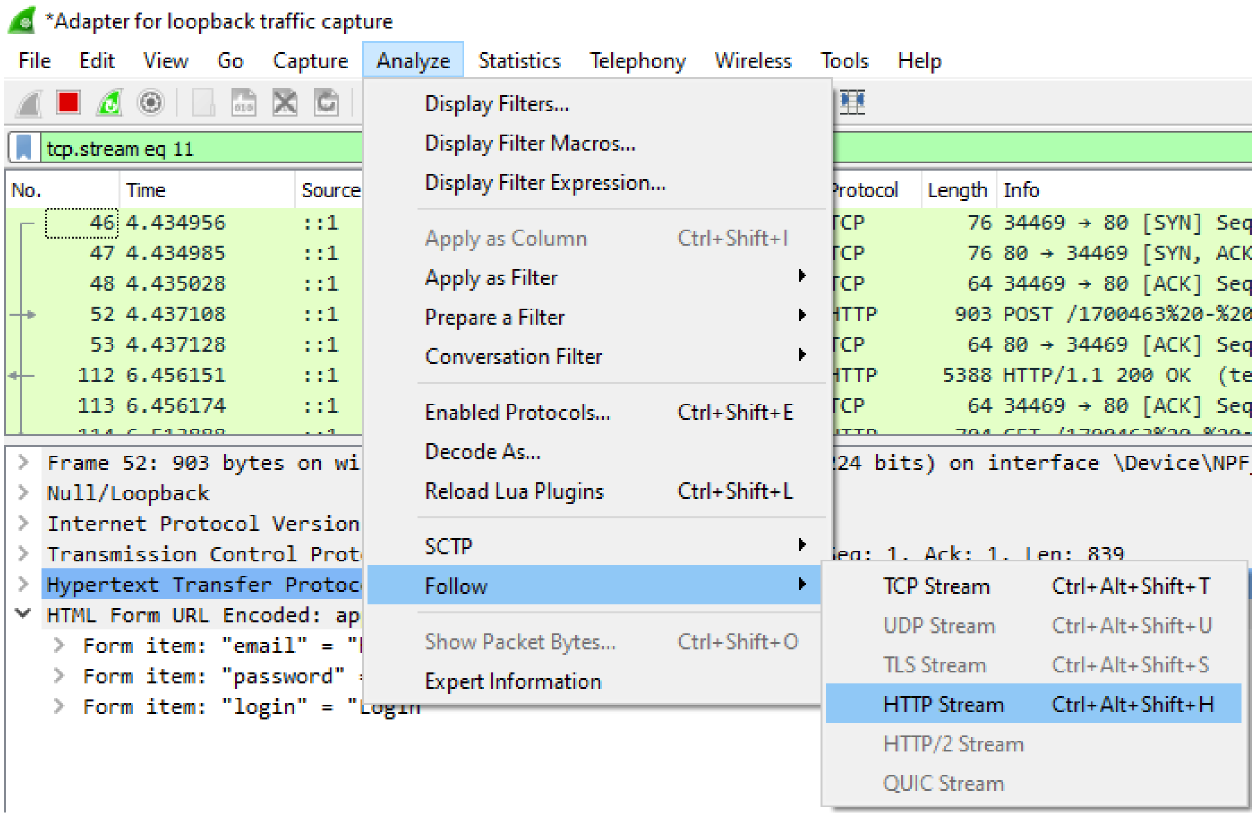
Clicking on the first packet in the list, it can be seen that the username and password that were entered can be seen in plain text. This can be seen in figure 5 below where the username reads 'hacklab@hacklab.com' and the password reads 'Hacklab1'.



```
No.  Time      Source  Destination  Protocol  Length  Info
 52 4.437108   ::1     ::1          HTTP       903 POST /1700463%20-%20original/ HTTP/1.1  (application/x-www-form-urlencoded)
112 6.456151   ::1     ::1          HTTP      5388 HTTP/1.1 200 OK  (text/html)
114 6.513888   ::1     ::1          HTTP       704 GET /1700463%20-%20original/ HTTP/1.1
163 8.531591   ::1     ::1          HTTP      5351 HTTP/1.1 200 OK  (text/html)

> Frame 52: 903 bytes on wire (7224 bits), 903 bytes captured (7224 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 6, Src: ::1, Dst: ::1
> Transmission Control Protocol, Src Port: 34469, Dst Port: 80, Seq: 1, Ack: 1, Len: 839
> Hypertext Transfer Protocol
v HTML Form URL Encoded: application/x-www-form-urlencoded
  > Form item: "email" = "hacklab@hacklab.com"
  > Form item: "password" = "Hacklab1"
  > Form item: "login" = "Login"
```

Figure 5: Captured Details

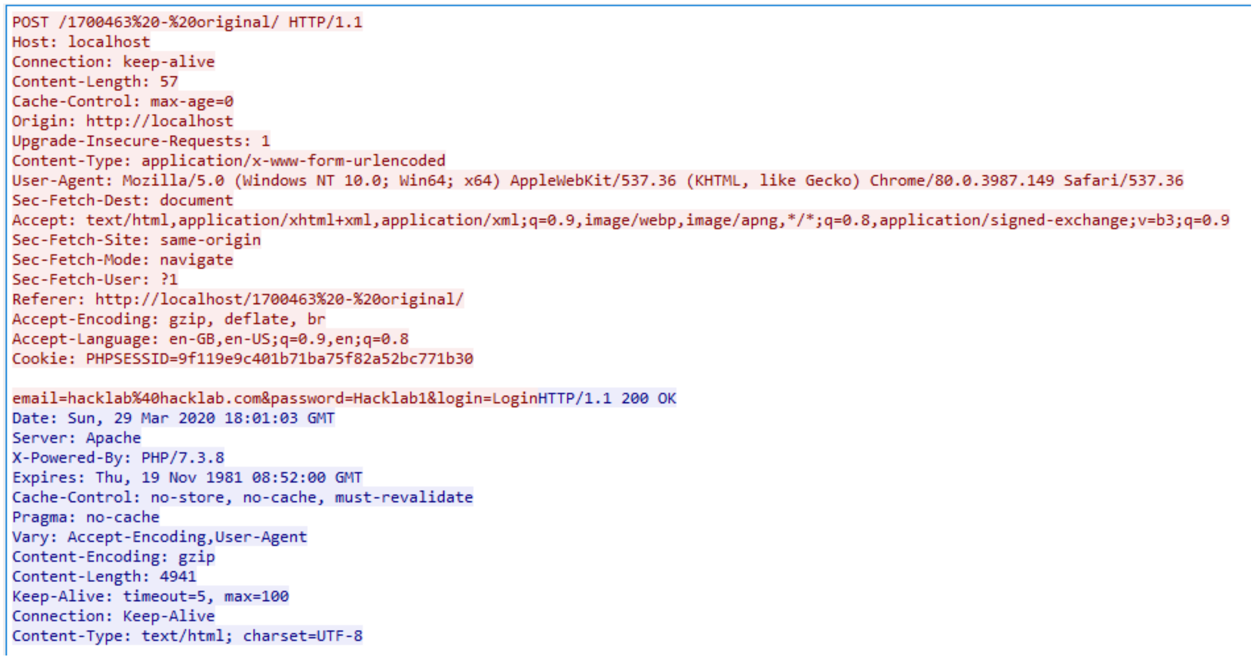
There is another method to view any data that has been captured if there are many more packets and are not able to find the exact one that represents the posting of the variables from the login page. This can be completed by navigating to the top menu and clicking on Analyze -> Follow -> HTTP Stream. This can be seen in figure 6 below.


*Adapter for loopback traffic capture
File
Edit
View
Go
Capture
Analyze
Statistics
Telephony
Wireless
Tools
Help
tcp.stream eq 11
Display Filters...
Display Filter Macros...
Display Filter Expression...
Apply as Column Ctrl+Shift+I
Apply as Filter
Prepare a Filter
Conversation Filter
Enabled Protocols... Ctrl+Shift+E
Decode As...
Reload Lua Plugins Ctrl+Shift+L
SCTP
Follow
Show Packet Bytes... Ctrl+Shift+O
Expert Information
TCP Stream Ctrl+Alt+Shift+T
UDP Stream Ctrl+Alt+Shift+U
TLS Stream Ctrl+Alt+Shift+S
HTTP Stream Ctrl+Alt+Shift+H
HTTP/2 Stream
QUIC Stream

```
No.  Time      Source   Protocol  Length  Info
 46 4.434956   ::1      TCP         76 34469 → 80 [SYN] Seq
 47 4.434985   ::1      TCP         76 80 → 34469 [SYN, ACK
 48 4.435028   ::1      TCP         64 34469 → 80 [ACK] Seq
 52 4.437108   ::1      HTTP       903 POST /1700463%20-%20
 53 4.437128   ::1      TCP         64 80 → 34469 [ACK] Seq
112 6.456151   ::1      HTTP      5388 HTTP/1.1 200 OK  (te
113 6.456174   ::1      TCP         64 34469 → 80 [ACK] Seq
```

Figure 6: Route to HTTP Stream

After following the HTTP stream, a window should appear with all the HTTP information that was captured from the packets. As can be seen highlighted in figure 7 below, the username and password that were entered into the prompted fields are clearly visible in plaintext with no security precautions to get past.

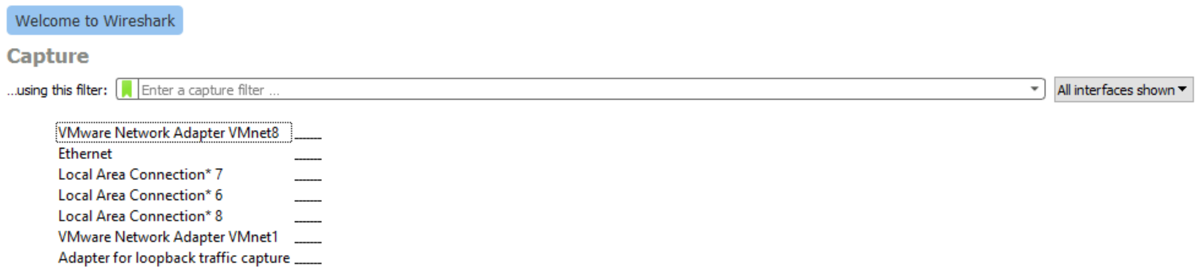


```
POST /1700463%20-%20original/ HTTP/1.1
Host: localhost
Connection: keep-alive
Content-Length: 57
Cache-Control: max-age=0
Origin: http://localhost
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.149 Safari/537.36
Sec-Fetch-Dest: document
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Referer: http://localhost/1700463%20-%20original/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Cookie: PHPSESSID=9f119e9c401b71ba75f82a52bc771b30

email=hacklab%40hacklab.com&password=Hacklab1&login=LoginHTTP/1.1 200 OK
Date: Sun, 29 Mar 2020 18:01:03 GMT
Server: Apache
X-Powered-By: PHP/7.3.8
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Vary: Accept-Encoding,User-Agent
Content-Encoding: gzip
Content-Length: 4941
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

Figure 7: Captured HTTP Stream

2.2.2 Using HTTPS

The first step in analyzing HTTPS packets in booting up Wireshark and capturing packets from the relative connection point. A list of connections will be displayed at startup. This can be seen in figure 8 below.



VMware Network Adapter VMnet8
Ethernet
Local Area Connection* 7
Local Area Connection* 6
Local Area Connection* 8
VMware Network Adapter VMnet1
Adapter for loopback traffic capture

Figure 8: Wireshark Home Screen

In this case the connection point is Ethernet. Clicking on this option then starts the capture process of all traffic going through the connection. Navigating to a website such as www.Amazon.co.uk and viewing the navigation bar shows that a padlock is visible. This can be seen in figure 9 below. This indicates that the website is using HTTPS.

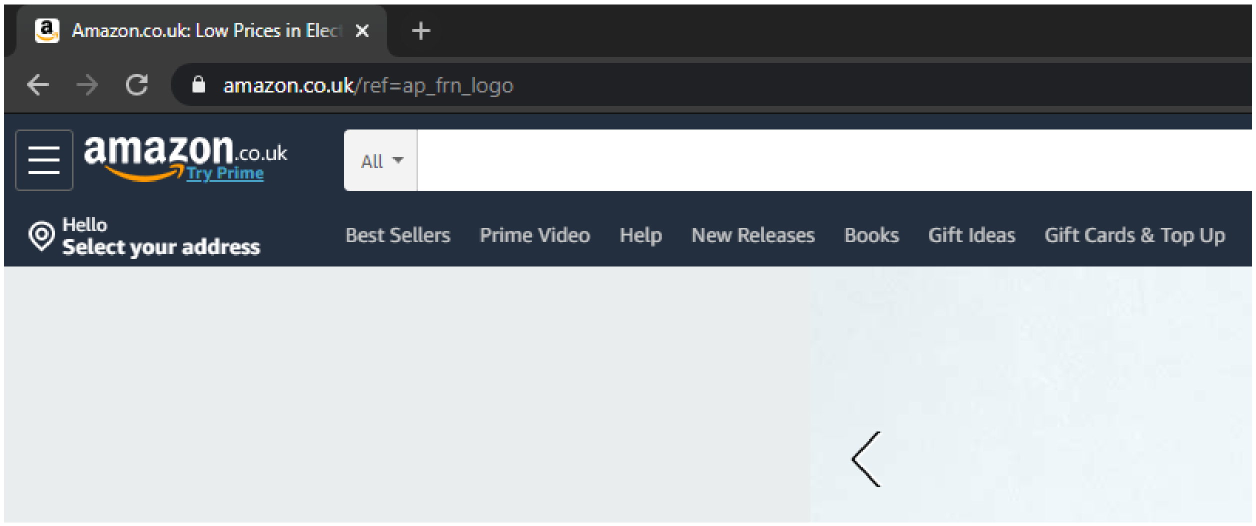
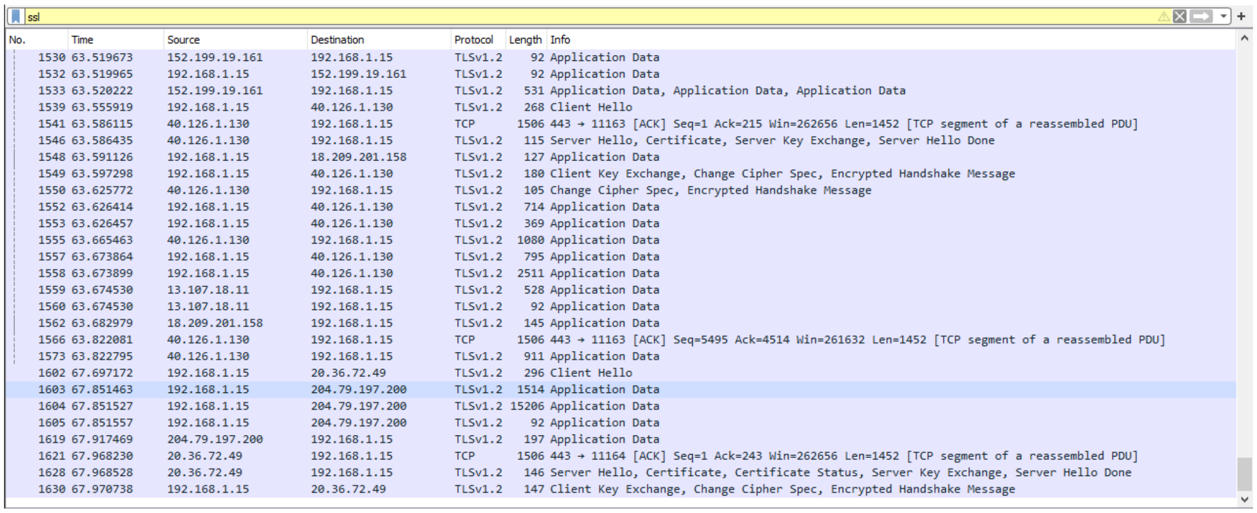


Figure 9: HTTPS Website

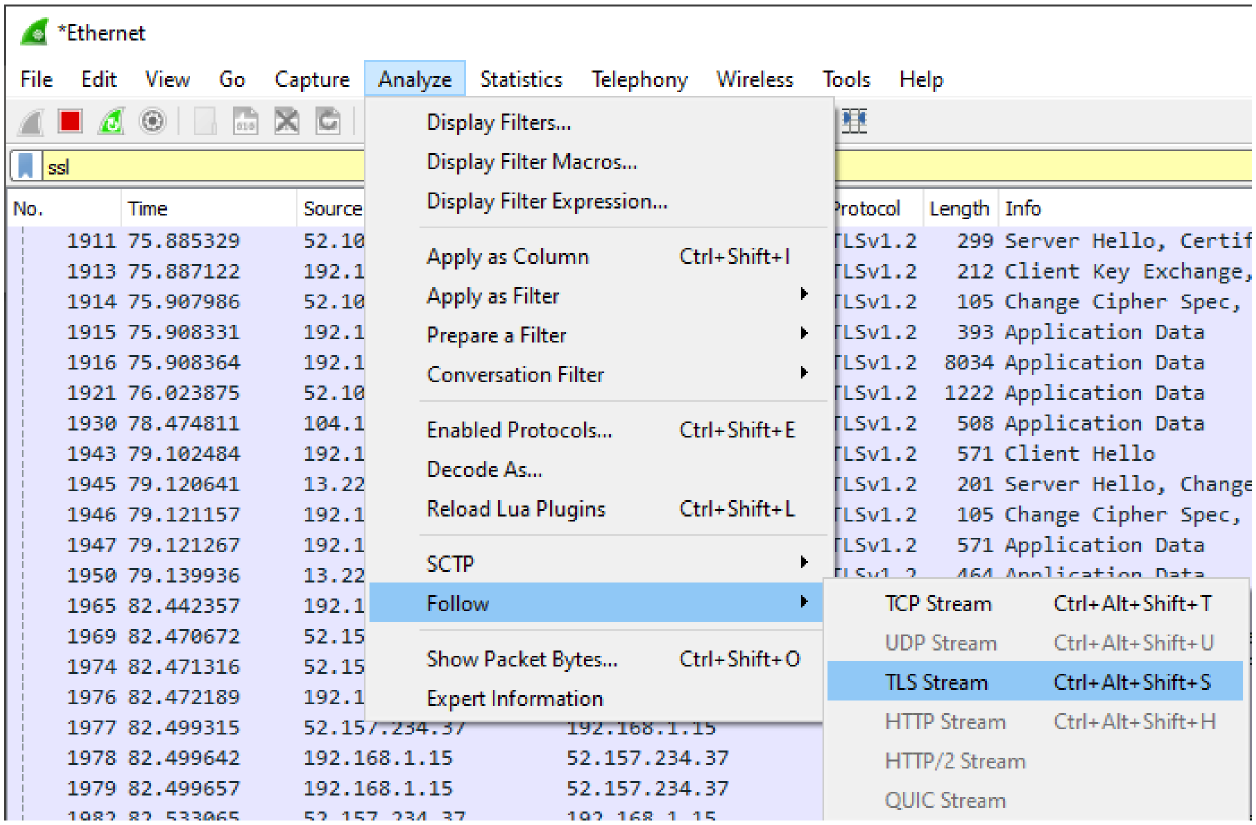
Once navigating to the login page and entering login details, these packets should be captured by Wireshark. Since HTTPS runs using SSL (Secure Socket Layer) encryption then a filter can be put in place to filter just the HTTPS traffic. This can be seen in figure 10 below.



```
ssl
No.   Time       Source          Destination     Protocol Length Info
 1530 63.519673  152.199.19.161  192.168.1.15    TLSv1.2    92 Application Data
 1532 63.519965  192.168.1.15    152.199.19.161  TLSv1.2    92 Application Data
 1533 63.520222  152.199.19.161  192.168.1.15    TLSv1.2   531 Application Data, Application Data, Application Data
 1539 63.555919  192.168.1.15    40.126.1.130    TLSv1.2   268 Client Hello
 1541 63.586115  40.126.1.130    192.168.1.15    TCP      1506 443 → 11163 [ACK] Seq=1 Ack=215 Win=262656 Len=1452 [TCP segment of a reassembled PDU]
 1546 63.586435  40.126.1.130    192.168.1.15    TLSv1.2   115 Server Hello, Certificate, Server Key Exchange, Server Hello Done
 1548 63.591126  192.168.1.15    18.209.201.158  TLSv1.2   127 Application Data
 1549 63.597298  192.168.1.15    40.126.1.130    TLSv1.2   180 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
 1550 63.625772  40.126.1.130    192.168.1.15    TLSv1.2   105 Change Cipher Spec, Encrypted Handshake Message
 1552 63.626414  192.168.1.15    40.126.1.130    TLSv1.2   714 Application Data
 1553 63.626457  192.168.1.15    40.126.1.130    TLSv1.2   369 Application Data
 1555 63.665463  40.126.1.130    192.168.1.15    TLSv1.2  1080 Application Data
 1557 63.673864  192.168.1.15    40.126.1.130    TLSv1.2   795 Application Data
 1558 63.673899  192.168.1.15    40.126.1.130    TLSv1.2  2511 Application Data
 1559 63.674530  13.107.18.11    192.168.1.15    TLSv1.2   528 Application Data
 1560 63.674530  13.107.18.11    192.168.1.15    TLSv1.2    92 Application Data
 1562 63.682979  18.209.201.158  192.168.1.15    TLSv1.2   145 Application Data
 1566 63.822081  40.126.1.130    192.168.1.15    TCP      1506 443 → 11163 [ACK] Seq=5495 Ack=4514 Win=261632 Len=1452 [TCP segment of a reassembled PDU]
 1573 63.822795  40.126.1.130    192.168.1.15    TLSv1.2   911 Application Data
 1602 67.697172  192.168.1.15    20.36.72.49     TLSv1.2   296 Client Hello
 1603 67.851463  192.168.1.15    204.79.197.200  TLSv1.2  1514 Application Data
 1604 67.851527  192.168.1.15    204.79.197.200  TLSv1.2 15206 Application Data
 1605 67.851557  192.168.1.15    204.79.197.200  TLSv1.2    92 Application Data
 1619 67.917469  204.79.197.200  192.168.1.15    TLSv1.2   197 Application Data
 1621 67.968230  20.36.72.49     192.168.1.15    TCP      1506 443 → 11164 [ACK] Seq=1 Ack=243 Win=262656 Len=1452 [TCP segment of a reassembled PDU]
 1628 67.968528  20.36.72.49     192.168.1.15    TLSv1.2   146 Server Hello, Certificate, Certificate Status, Server Key Exchange, Server Hello Done
 1630 67.970738  192.168.1.15    20.36.72.49     TLSv1.2   147 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
```

Figure 10: Filter SSL

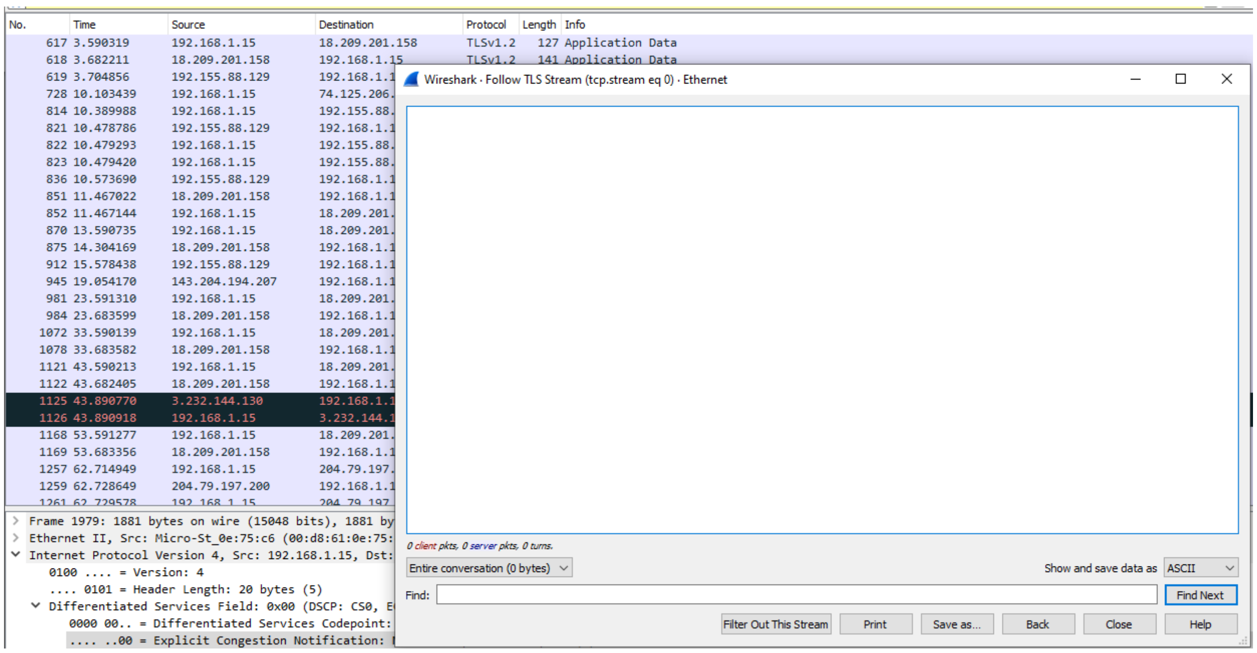
From here, you are able to view any content that can be deciphered by viewing the TLS (Transport Layer Security) stream. This can be done by navigating to the following Analyze -> Follow -> TLS Stream. This can be seen in figure 11 below.


*Ethernet
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
ssl
Display Filters...
Display Filter Macros...
Display Filter Expression...
Apply as Column Ctrl+Shift+I
Apply as Filter
Prepare a Filter
Conversation Filter
Enabled Protocols... Ctrl+Shift+E
Decode As...
Reload Lua Plugins Ctrl+Shift+L
SCTP
Follow
Show Packet Bytes... Ctrl+Shift+O
Expert Information
TCP Stream Ctrl+Alt+Shift+T
UDP Stream Ctrl+Alt+Shift+U
TLS Stream Ctrl+Alt+Shift+S
HTTP Stream Ctrl+Alt+Shift+H
HTTP/2 Stream
QUIC Stream

```
No.       Time        Source                                      Protocol   Length  Info
     1911 75.885329   52.10                                       TLSv1.2       299  Server Hello, Certif
     1913 75.887122   192.1                                       TLSv1.2       212  Client Key Exchange,
     1914 75.907986   52.10                                       TLSv1.2       105  Change Cipher Spec,
     1915 75.908331   192.1                                       TLSv1.2       393  Application Data
     1916 75.908364   192.1                                       TLSv1.2      8034  Application Data
     1921 76.023875   52.10                                       TLSv1.2      1222  Application Data
     1930 78.474811   104.1                                       TLSv1.2       508  Application Data
     1943 79.102484   192.1                                       TLSv1.2       571  Client Hello
     1945 79.120641   13.22                                       TLSv1.2       201  Server Hello, Change
     1946 79.121157   192.1                                       TLSv1.2       105  Change Cipher Spec,
     1947 79.121267   192.1                                       TLSv1.2       571  Application Data
     1950 79.139936   13.22                                       TLSv1.2       464  Application Data
     1965 82.442357   192.1
     1969 82.470672   52.15
     1974 82.471316   52.15
     1976 82.472189   192.1
     1977 82.499315   52.157.234.37       192.168.1.15
     1978 82.499642   192.168.1.15        52.157.234.37
     1979 82.499657   192.168.1.15        52.157.234.37
     1982 82.533065   52.157.234.37       192.168.1.15
```

Figure 11: Route to TLS Stream

A window should appear with any information that can be intercepted and displayed. As can be seen in figure 12 below, there is no information that can be seen due to the increase in security and encryption that is used.



```
No.   Time        Source            Destination       Protocol  Length Info
  617 3.590319    192.168.1.15      18.209.201.158    TLSv1.2   127 Application Data
  618 3.682211    18.209.201.158    192.168.1.15      TLSv1.2   141 Application Data
  619 3.704856    192.155.88.129    192.168.1.1
  728 10.103439   192.168.1.15      74.125.206.
  814 10.389988   192.168.1.15      192.155.88.
  821 10.478786   192.155.88.129    192.168.1.1
  822 10.479293   192.168.1.15      192.155.88.
  823 10.479420   192.168.1.15      192.155.88.
  836 10.573690   192.155.88.129    192.168.1.1
  851 11.467022   18.209.201.158    192.168.1.1
  852 11.467144   192.168.1.15      18.209.201.
  870 13.590735   192.168.1.15      18.209.201.
  875 14.304169   18.209.201.158    192.168.1.1
  912 15.578438   192.155.88.129    192.168.1.1
  945 19.054170   143.204.194.207   192.168.1.1
  981 23.591310   192.168.1.15      18.209.201.
  984 23.683599   18.209.201.158    192.168.1.1
 1072 33.590139   192.168.1.15      18.209.201.
 1078 33.683582   18.209.201.158    192.168.1.1
 1121 43.590213   192.168.1.15      18.209.201.
 1122 43.682405   18.209.201.158    192.168.1.1
 1125 43.890770   3.232.144.130     192.168.1.1
 1126 43.890918   192.168.1.15      3.232.144.1
 1168 53.591277   192.168.1.15      18.209.201.
 1169 53.683356   18.209.201.158    192.168.1.1
 1257 62.714949   192.168.1.15      204.79.197.
 1259 62.728649   204.79.197.200    192.168.1.1
 1261 62.729578   192.168.1.15      204.79.197

> Frame 1979: 1881 bytes on wire (15048 bits), 1881 by
> Ethernet II, Src: Micro-St_0e:75:c6 (00:d8:61:0e:75:
v Internet Protocol Version 4, Src: 192.168.1.15, Dst:
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  v Differentiated Services Field: 0x00 (DSCP: CS0, E
      0000 00.. = Differentiated Services Codepoint:
      .... ..00 = Explicit Congestion Notification:
```

Wireshark · Follow TLS Stream (tcp.stream eq 0) · Ethernet

0 client pkts, 0 server pkts, 0 turns.

Entire conversation (0 bytes)

Show and save data as ASCII

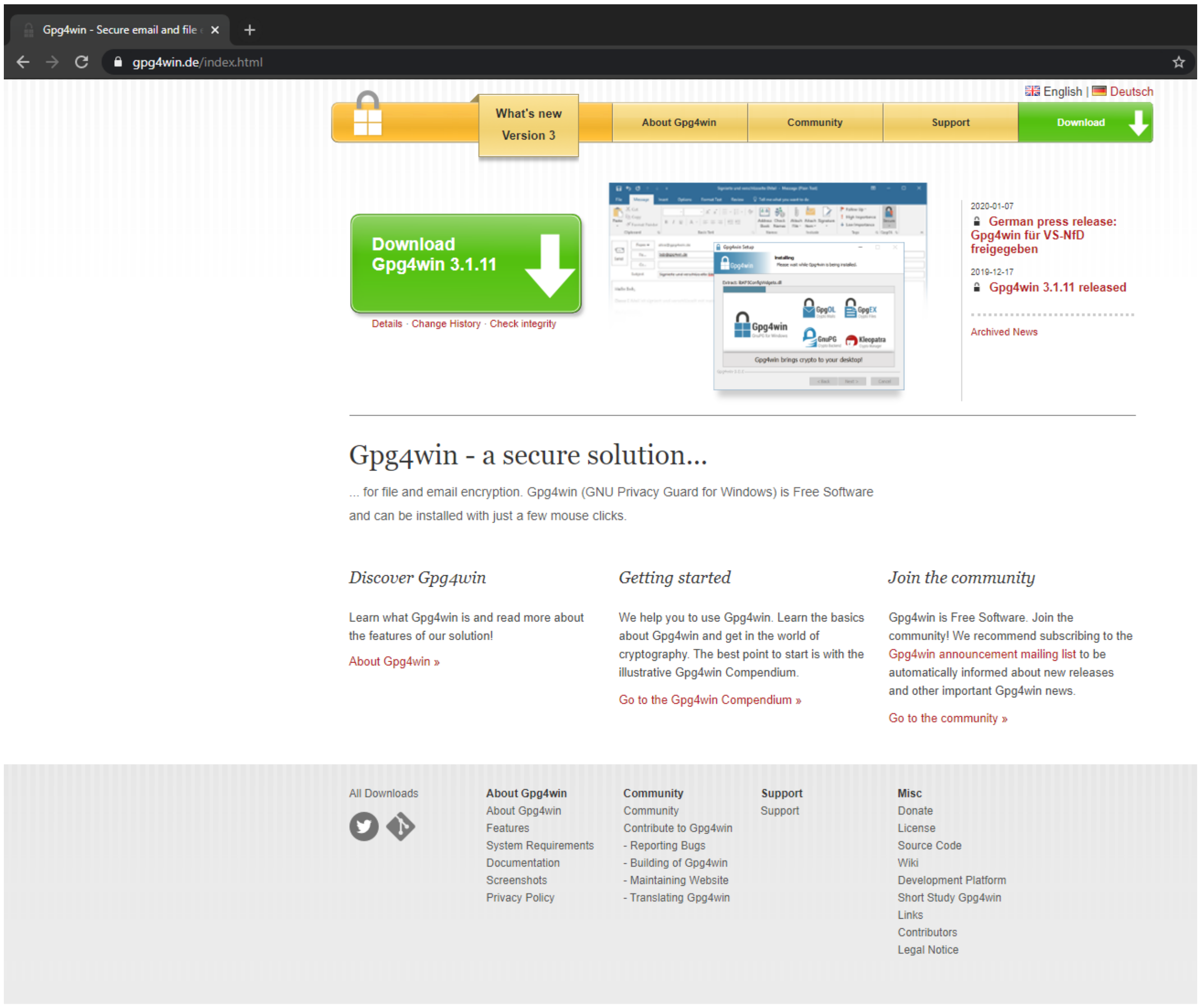
Find:

Figure 12: Captured Data

1.1 Alternative transmission Method – gpg4win Example

HTTPS works well for traffic that is being sent through a web browser, but what is you are sending data over a local network or using another form of transmitting the data to the receiver that may not use the same security protocols. RSA can be used out with a web browser as well. This method of encryption is commonly known as public-key encryption. The basis behind public key cryptography is using two ‘keys’, a private and a public key. One computer can hold the private key to decrypt data, and this computer can distribute a paired public key to anyone it may choose to allow for data to be encrypted for the paired private key. To be able to carry out and send files using public key encryption, the following procedure can be used.

The first step is to download the Gpg4win software from www.gpg4win.de/index.html. The page that should appear can be seen below in figure 13.


Gpg4win - Secure email and file
gpg4win.de/index.html

English | Deutsch

What's new Version 3 | About Gpg4win | Community | Support | Download

Download Gpg4win 3.1.11

Details · Change History · Check integrity

2020-01-07
German press release: Gpg4win für VS-NfD freigegeben

2019-12-17
Gpg4win 3.1.11 released

Archived News

Gpg4win - a secure solution...

... for file and email encryption. Gpg4win (GNU Privacy Guard for Windows) is Free Software and can be installed with just a few mouse clicks.

Discover Gpg4win

Learn what Gpg4win is and read more about the features of our solution!

About Gpg4win »

Getting started

We help you to use Gpg4win. Learn the basics about Gpg4win and get in the world of cryptography. The best point to start is with the illustrative Gpg4win Compendium.

Go to the Gpg4win Compendium »

Join the community

Gpg4win is Free Software. Join the community! We recommend subscribing to the Gpg4win announcement mailing list to be automatically informed about new releases and other important Gpg4win news.

Go to the community »

All Downloads

About Gpg4win
About Gpg4win
Features
System Requirements
Documentation
Screenshots
Privacy Policy

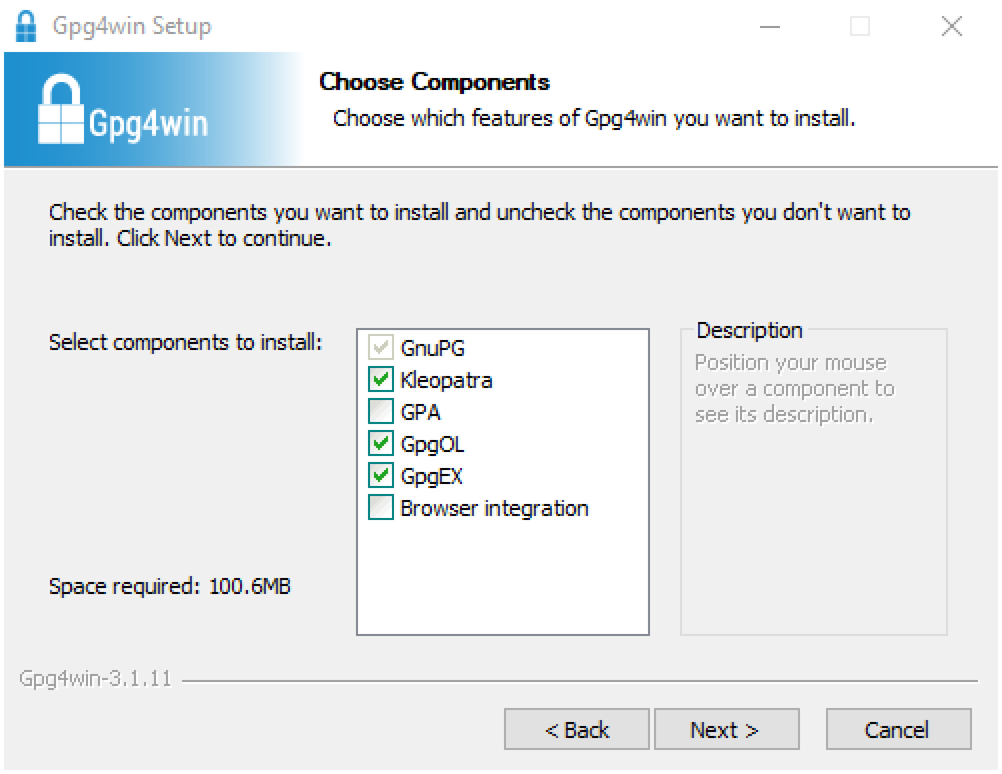
Community
Community
Contribute to Gpg4win
- Reporting Bugs
- Building of Gpg4win
- Maintaining Website
- Translating Gpg4win

Support
Support

Misc
Donate
License
Source Code
Wiki
Development Platform
Short Study Gpg4win
Links
Contributors
Legal Notice

Figure 13: Gpg4win Website

Once the installer is downloaded, install the application with the following choices of components, including the software Kleopatra, this can be seen in figure 14 below.


Gpg4win Setup

Choose Components
Choose which features of Gpg4win you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

- [✓] GnuPG
- [✓] Kleopatra
- [ ] GPA
- [✓] GpgOL
- [✓] GpgEX
- [ ] Browser integration

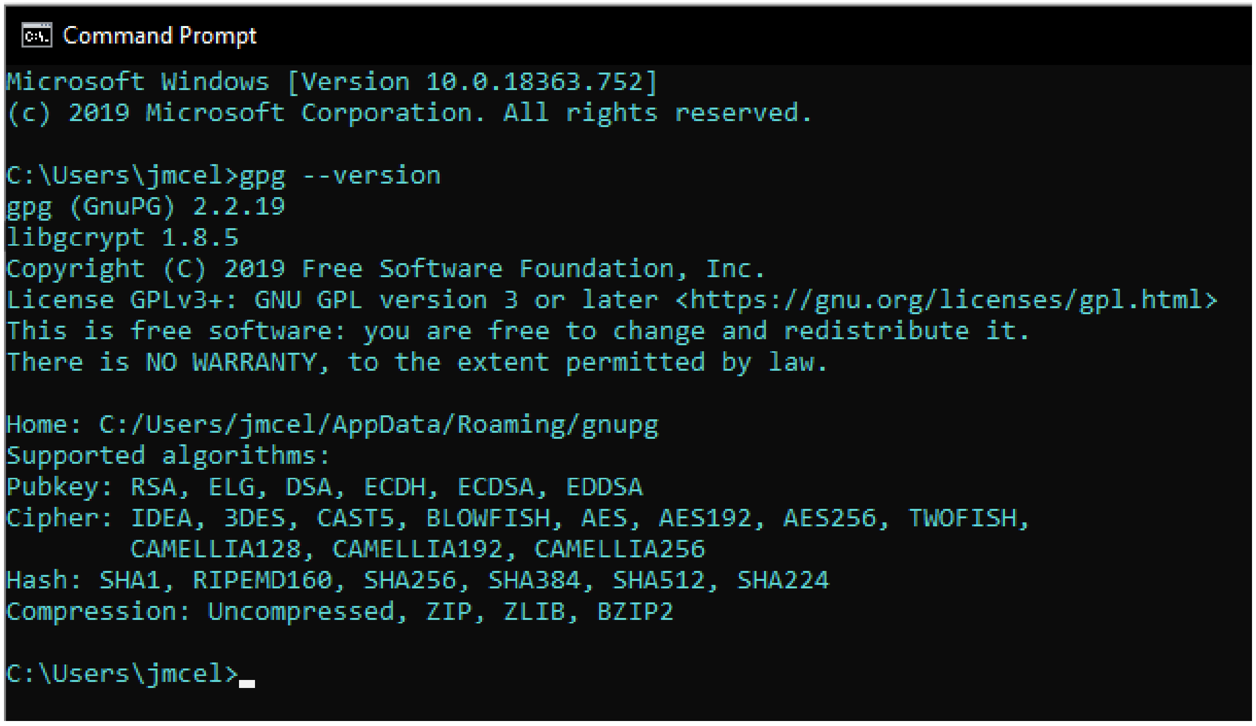
Description
Position your mouse over a component to see its description.

Space required: 100.6MB

Gpg4win-3.1.11

Figure 14: Setup

To ensure that the program has installed successfully. Opening command prompt and typing the command ‘gpg --version’ should display the following screen.



```
Microsoft Windows [Version 10.0.18363.752]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\jmcel>gpg --version
gpg (GnuPG) 2.2.19
libgcrypt 1.8.5
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Home: C:/Users/jmcel/AppData/Roaming/gnupg
Supported algorithms:
Pubkey: RSA, ELG, DSA, ECDH, ECDSA, EDDSA
Cipher: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH,
        CAMELLIA128, CAMELLIA192, CAMELLIA256
Hash: SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
Compression: Uncompressed, ZIP, ZLIB, BZIP2

C:\Users\jmcel>
```

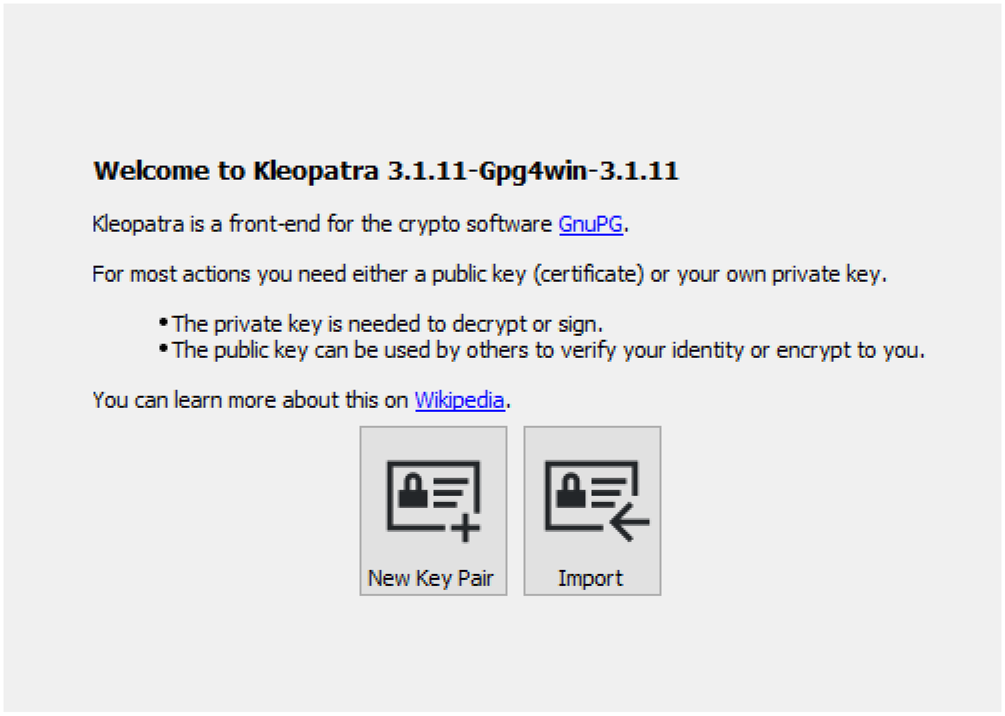
Figure 15: Verifying Install

From here there are two different ways to use the software. There is the command line interface or the graphical interface. These have both been covered in the following sections.

2.2.3 GUI (Graphical User Interface)

2.2.3.1 Generating Key Pair

The first step in using the graphical interface of gpg4win is to use the software that was previously installed, Kleopatra. Running this software, the following screen should appear.



Welcome to Kleopatra 3.1.11-Gpg4win-3.1.11

Kleopatra is a front-end for the crypto software GnuPG.

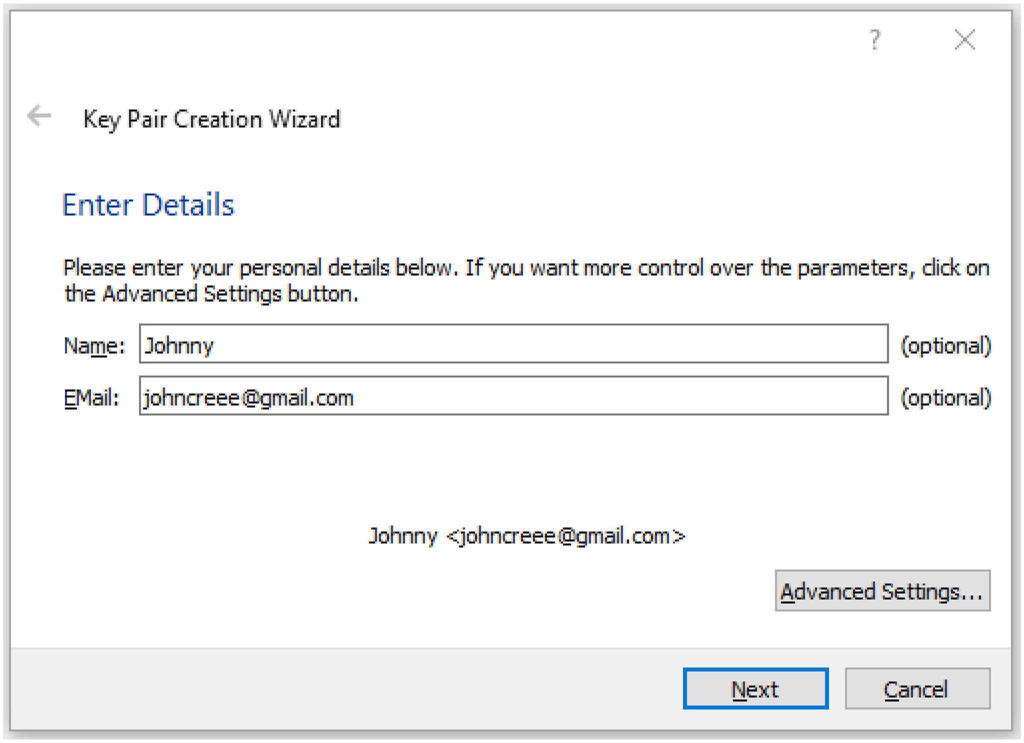
For most actions you need either a public key (certificate) or your own private key.

- The private key is needed to decrypt or sign.
- The public key can be used by others to verify your identity or encrypt to you.

You can learn more about this on Wikipedia.

Figure 16: Kleopatra Home Screen

Using the 'New Key Pair' button, another window will pop up asking for a name and email for the private key. The entries 'Johnny' and 'johncreee@gmail.com' were used. This can be seen in the figure below.



Key Pair Creation Wizard

Enter Details

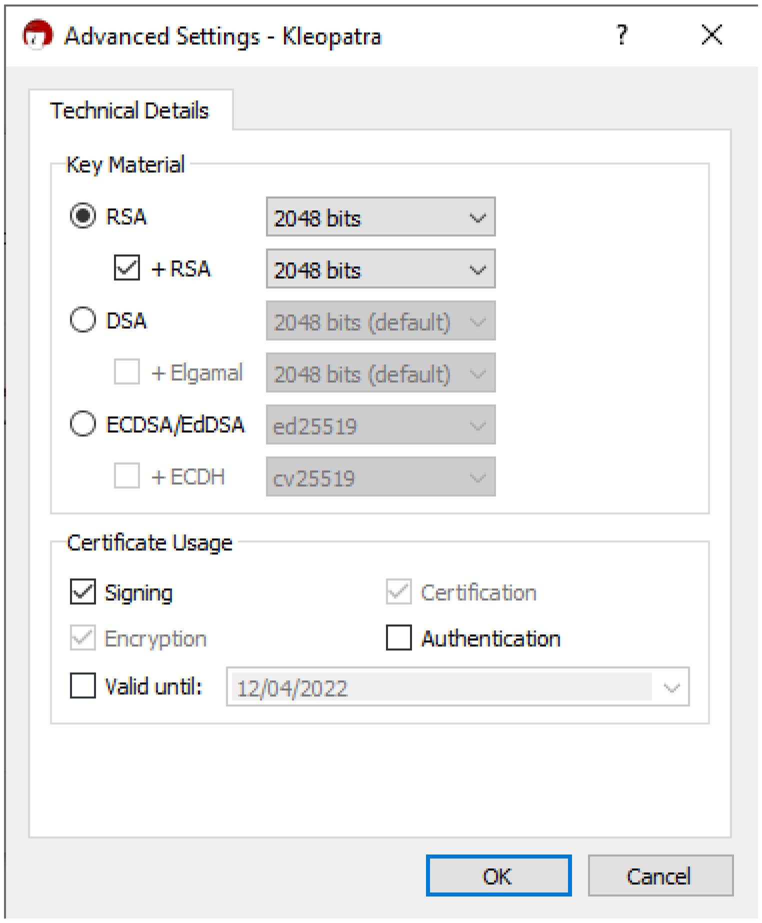
Please enter your personal details below. If you want more control over the parameters, click on the Advanced Settings button.

Name:	Johnny	(optional)
EMail:	johncreee@gmail.com	(optional)

Johnny <johncreee@gmail.com>

Figure 17: Entered Details

Selecting advanced options from the same window displays the more detailed options of the key pair that is to be generated. All values were let untouched apart from the date, unchecking this box allows the for key pair to be used indefinitely. This can be seen in figure 18 below.


Advanced Settings - Kleopatra

Technical Details

Key Material

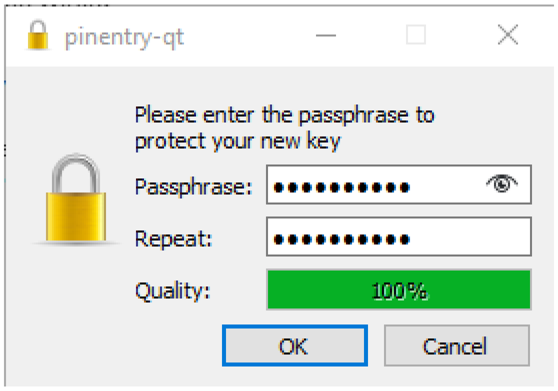
- [✓ RSA] 2048 bits
- [✓] +RSA 2048 bits
- [ ] DSA 2048 bits (default)
- [ ] +Elgamal 2048 bits (default)
- [ ] ECDSA/EdDSA ed25519
- [ ] +ECDH cv25519

Certificate Usage

- [✓] Signing
- [✓] Certification
- [✓] Encryption
- [ ] Authentication
- [ ] Valid until: 12/04/2022

Figure 18: Technical Details

Clicking 'OK', then 'Next', then 'Create', a window will appear asking for a passphrase. The value of 'Hacklab123' was used in this case and can be seen entered into the fields below.


pinentry-qt

Please enter the passphrase to protect your new key

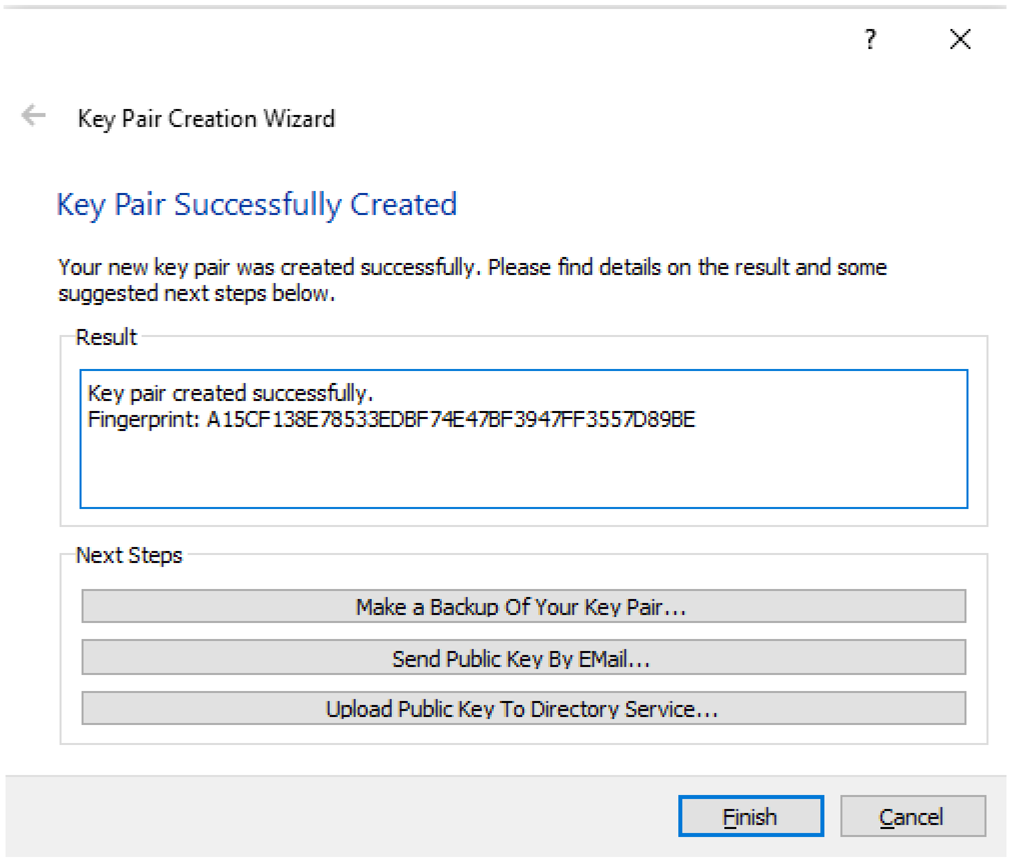
Passphrase: ●●●●●●●●●●

Repeat: ●●●●●●●●●●

Quality: 100%

Figure 19: Entered Passphrase

The next screen should indicate that the key pair has been generated successfully. This can be seen below.


?
Key Pair Creation Wizard
Next Steps
Make a Backup Of Your Key Pair...
Send Public Key By EMail...
Upload Public Key To Directory Service...
Finish
Cancel

Key Pair Successfully Created

Your new key pair was created successfully. Please find details on the result and some suggested next steps below.

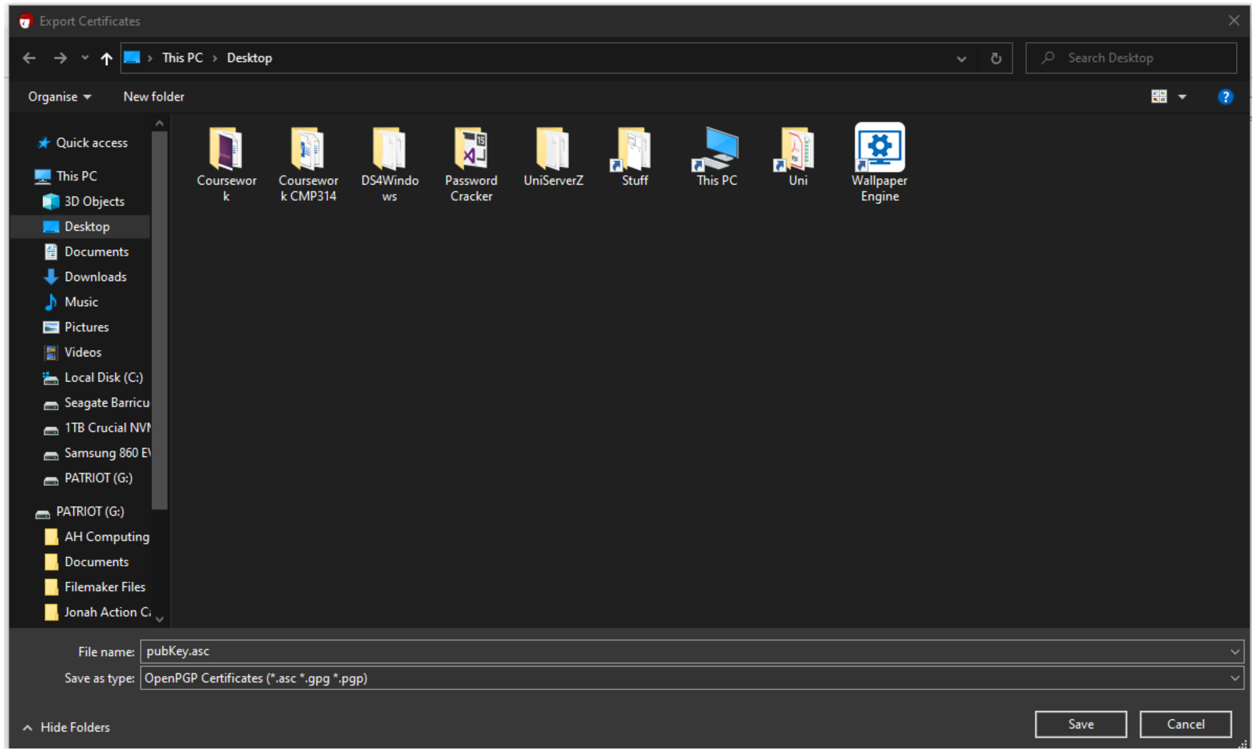
Result

Key pair created successfully.
Fingerprint: A15CF138E78533EDBF74E47BF3947FF3557D89BE

Figure 20: Successful Creation of Key

2.2.3.2 Exporting Public Key

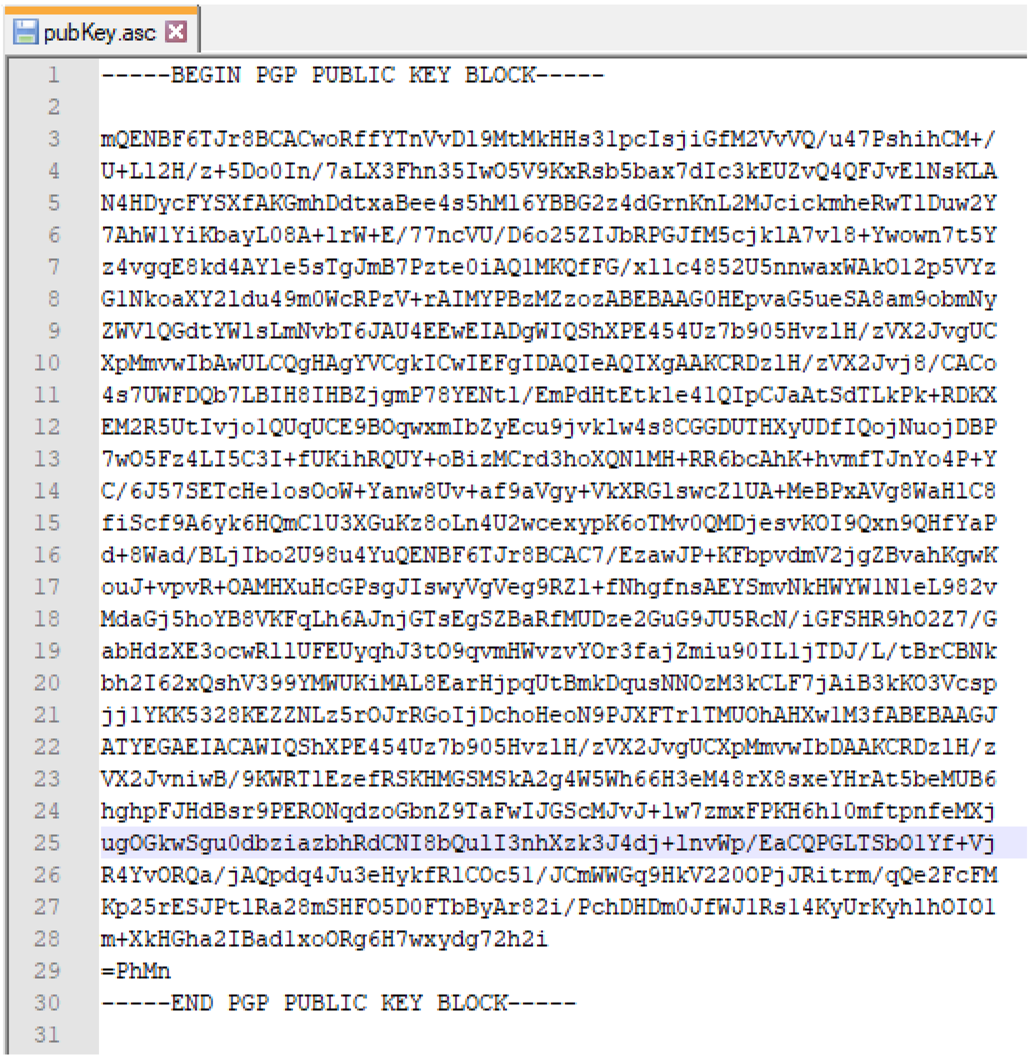
Now that the key pair has been created, the next step is to export the public key. This can be completed by clicking on the new key created and then going to the top navigation bar of the software and clicking the button lapelled 'Export'. A save window will appear; in this case the file was saved as 'pubKey.asc' in the desktop directory. This can be seen carried out below.



File name: pubKey.asc
Save as type: OpenPGP Certificates (*.asc *.gpg *.pgp)

Figure 21: Saving Public Key

Opening the file in notepad ++, it can be seen that the file does contain the public key. The contents of the file can be seen below.



```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQENBF6TJr8BCACwoRffYTnVvDl9MtMkHHs3lpcIsjiGfM2VvVQ/u47PshihCM+/
U+Ll2H/z+5Do0In/7aLX3Fhn35IwO5V9KxRsb5bax7dIc3kEUZvQ4QFJvElNsKLA
N4HDycFYSXfAKGmhDdtxaBee4s5hMl6YBBG2z4dGrnKnL2MJcickmheRwTlDuw2Y
7AhWlYiKbayL08A+lrW+E/77ncVU/D6o25ZIJbRPGJfM5cjklA7vl8+Ywown7t5Y
z4vgqE8kd4AYle5sTgJmB7Pzte0iAQlMKQfFG/xllc4852U5nnwaxWAkOl2p5VYz
GlNkoaXY2ldu49m0WcRPzV+rAIMYPBzMZzozABEBAAG0HEpvaG5ueSA8am9obmNy
ZWVlQGdtYWlsLmNvbT6JAU4EEwEIADgWIQShXPE454Uz7b905HvzlH/zVX2JvgUC
XpMmvwIbAwULCQgHAgYVCgkICwIEFgIDAQIeAQIXgAAKCRDzlH/zVX2Jvj8/CACo
4s7UWFDQb7LBIH8IHBZjgmP78YENtl/EmPdHtEtkle4lQIpCJaAtSdTLkPk+RDKX
EM2R5UtIvjolQUqUCE9BOqwxmIbZyEcu9jvklw4s8CGGDUTHXyUDfIQojNuojDBP
7wO5Fz4LI5C3I+fUKihRQUY+oBizMCrd3hoXQNlMH+RR6bcAhK+hvmfTJnYo4P+Y
C/6J57SETcHelosOoW+Yanw8Uv+af9aVgy+VkXRGlswcZlUA+MeBPxAVg8WaHlC8
fiScf9A6yk6HQmClU3XGuKz8oLn4U2wcexypK6oTMv0QMDjesvKOI9Qxn9QHfYaP
d+8Wad/BLjIbo2U98u4YuQENBF6TJr8BCAC7/EzawJP+KFbpvdmV2jgZBvahKgwK
ouJ+vpvR+OAMHXuHcGPsgJIswyVgVeg9RZl+fNhgfnsAEYSmvNkHWYWlNleL982v
MdaGj5hoYB8VKFqLh6AJnjGTsEgSZBaRfMUDze2GuG9JU5RcN/iGFSHR9hO2Z7/G
abHdzXE3ocwRllUFEUyqhJ3tO9qvmHWvzvYOr3fajZmiu90ILljTDJ/L/tBrCBNk
bh2I62xQshV399YMWUKiMAL8EarHjpqUtBmkDqusNNOzM3kCLF7jAiB3kKO3Vcsp
jjlYKK5328KEZZNLz5rOJrRGoIjDchoHeoN9PJXFTrlTMUOhAHXwlM3fABEBAAGJ
ATYEGAEIACAWIQShXPE454Uz7b905HvzlH/zVX2JvgUCXpMmvwIbDAAKCRDzlH/z
VX2JvniwB/9KWRTlEzefRSKHMGSMSkA2g4W5Wh66H3eM48rX8sxeYHrAt5beMUB6
hghpFJHdBsr9PERONqdzoGbnZ9TaFwIJGScMJvJ+lw7zmxFPKH6hl0mftpnfeMXj
ugOGkwSgu0dbziazbhRdCNI8bQulI3nhXzk3J4dj+lnvWp/EaCQPGLTSbOlYf+Vj
R4YvORQa/jAQpdq4Ju3eHykfRlCOc5l/JCmWWGq9HkV22OOPjJRitrm/qQe2FcFM
Kp25rESJPtlRa28mSHFO5D0FTbByAr82i/PchDHDm0JfWJlRsl4KyUrKyhlhOIOl
m+XkHGha2IBadlxoORg6H7wxydg72h2i
=PhMn
-----END PGP PUBLIC KEY BLOCK-----

```

Figure 22: Public Key

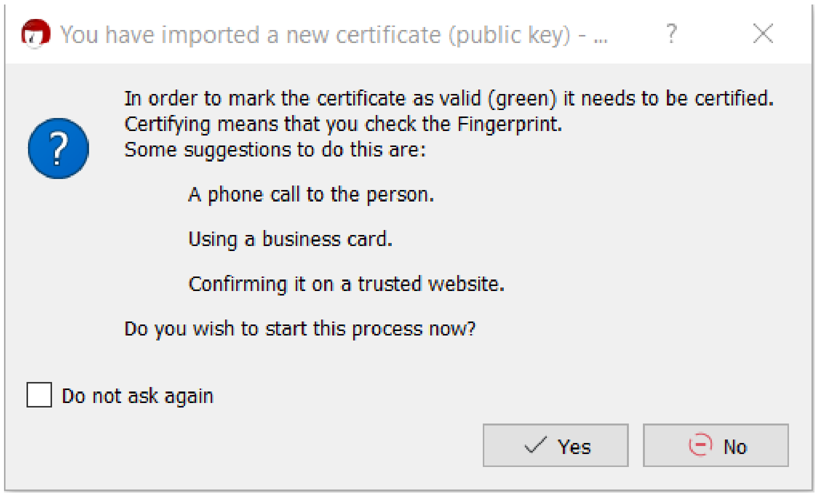
2.2.3.3 *Distributing Public Key*

The next step is to distribute the public key to another computer to allow for data to be encrypted using the new computer. This can be done my email or any other method of transferring files. In this case a USB stick was used to transfer the file from one computer to the other.

2.2.3.4 *Importing the Public Key*

Note: From here on in section 2.3.1, there is two computers. One will be referenced as desktop computer, the one with the private key, and the other a laptop which will have the public key.

Once the file has been transferred to the laptop form the desktop computer, running Kleopatra on the laptop, the same starting screen should be present as previously. The option 'Import' should be present. Using this option and selecting the file that has been transferred over, the following screen should appear.


You have imported a new certificate (public key) - ...

In order to mark the certificate as valid (green) it needs to be certified.
Certifying means that you check the Fingerprint.
Some suggestions to do this are:

A phone call to the person.

Using a business card.

Confirming it on a trusted website.

Do you wish to start this process now?

Figure 23: Create New Key Prompt

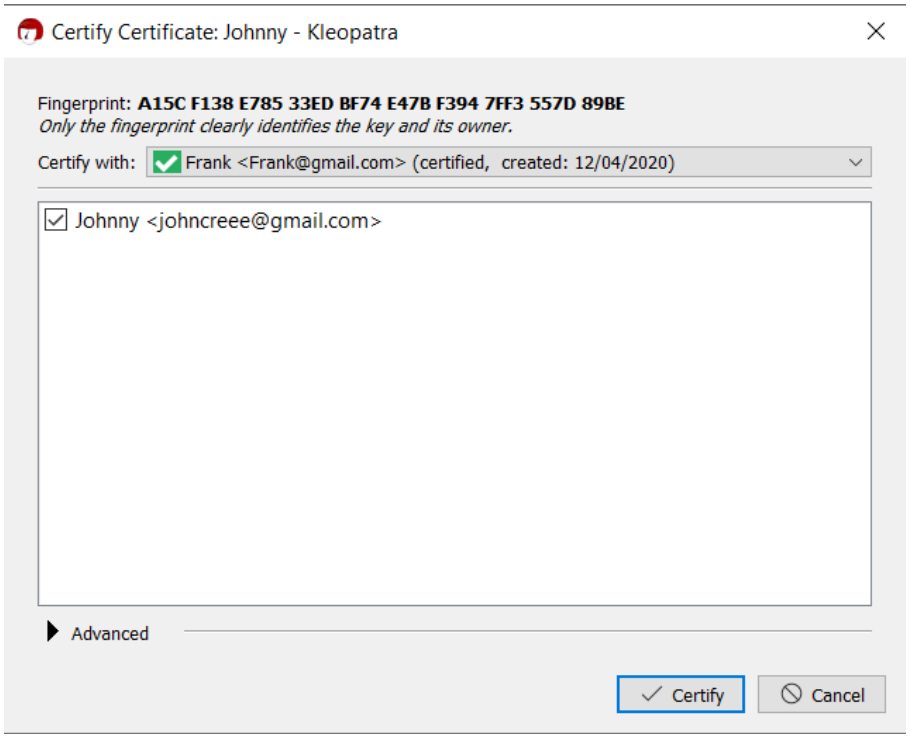
Selecting 'Yes', allows for the creation of a key pair on the laptop. The same process that was seen previously on the desktop computer is carried out. The entered details were as followed:

Name: Frank

Email: Frnak@gmail.com

Passphrase: Frank123

Once the local key pair had been created, the option to certify the imported key is shown as can be seen below in figure 24.


Certify Certificate: Johnny - Kleopatra

Fingerprint: **A15C F138 E785 33ED BF74 E47B F394 7FF3 557D 89BE**
Only the fingerprint clearly identifies the key and its owner.
Certify with: ✓ Frank <Frank@gmail.com> (certified, created: 12/04/2020)

☑ Johnny <johncreee@gmail.com>

Figure 24: Certify Key

Selecting 'Certify' and then entering the passphrase for the new 'Frank' user completes the importation process.

2.2.3.5 Encrypting a file

To encrypt a file, first a message must be crafted. A simple text file with some text inside will do the trick. A text file labelled TestMessage.txt was created. To encrypt the created text file, the 'Sign/Encrypt…' button at the top navigation bar of the window is selected. This can be seen in figure 25 below.

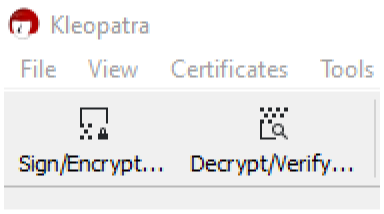
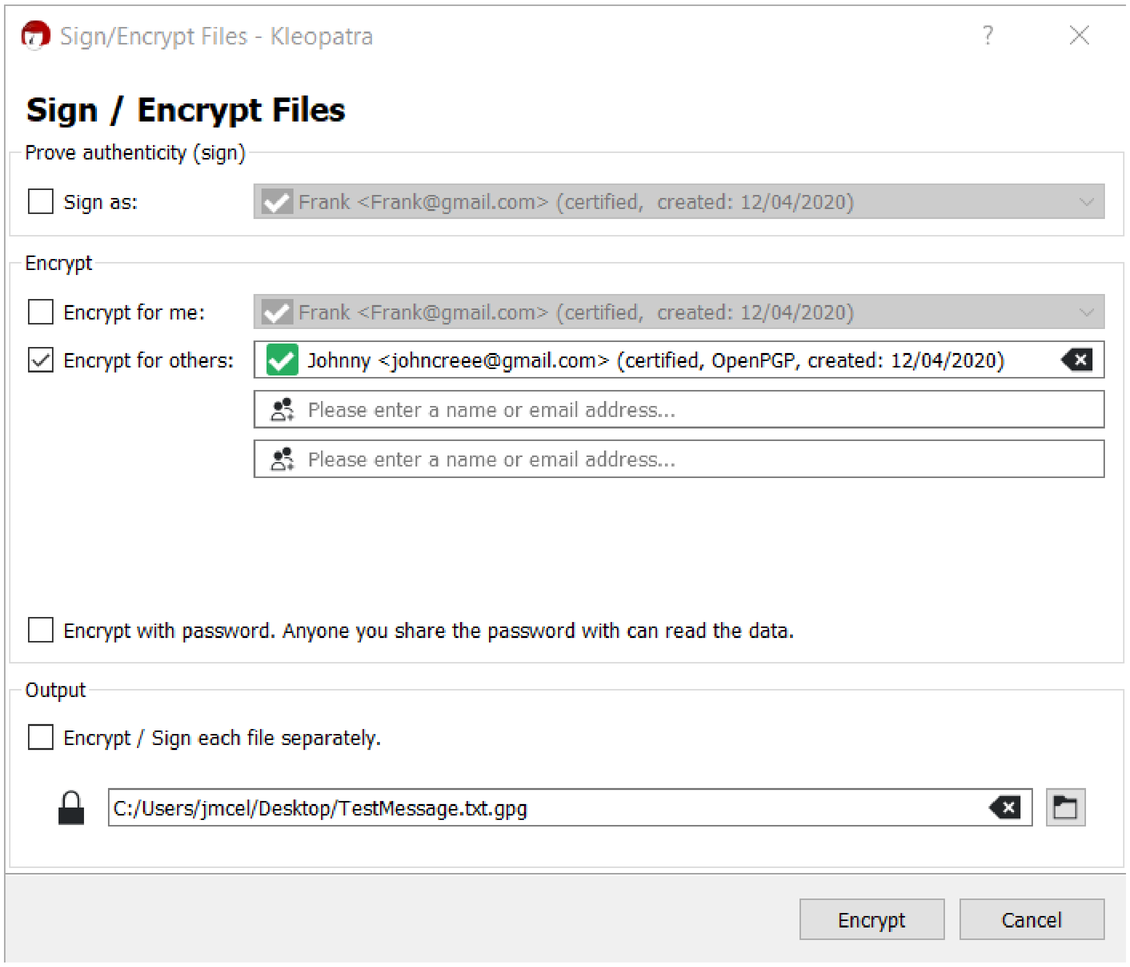


Figure 25: Encrypt Option

After selecting the button, a window will appear to select the file in which you would like to encrypt. After selecting the created text file, the following window should appear. From here the option to 'Encrypt for others:' is selected. Entering the username 'Johnny' should give the option to select the user 'Johnny' from the drop-down list. This can be seen below.


Sign/Encrypt Files - Kleopatra

Sign / Encrypt Files

Prove authenticity (sign)

☐ Sign as: Frank <Frank@gmail.com> (certified, created: 12/04/2020)

Encrypt

☐ Encrypt for me: Frank <Frank@gmail.com> (certified, created: 12/04/2020)

☑ Encrypt for others: Johnny <johncreee@gmail.com> (certified, OpenPGP, created: 12/04/2020)

Please enter a name or email address...

Please enter a name or email address...

☐ Encrypt with password. Anyone you share the password with can read the data.

Output

☐ Encrypt / Sign each file separately.

C:/Users/jmcel/Desktop/TestMessage.txt.gpg

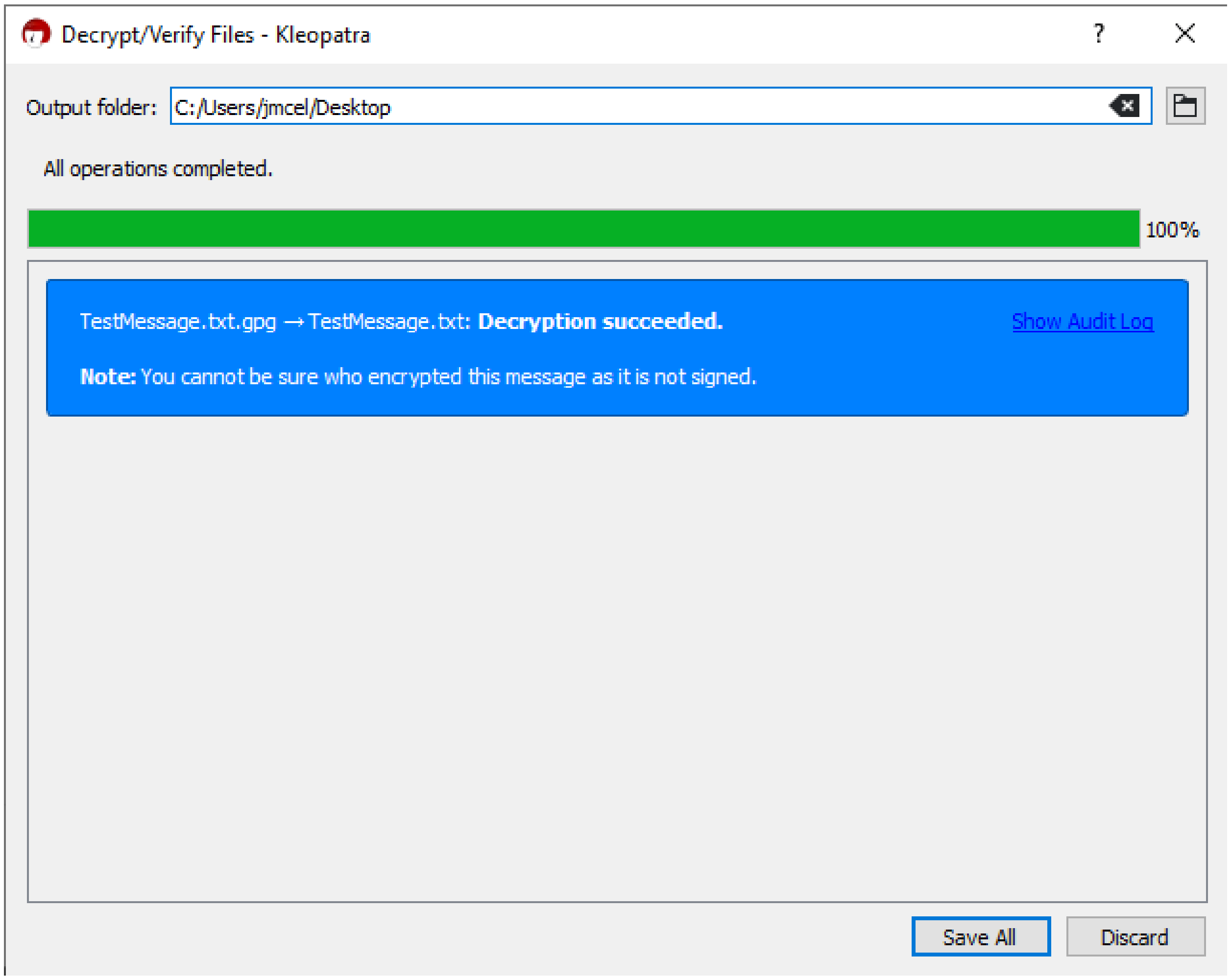
Figure 26: Encryption Options

Once all the necessary fields have been filled out, continuing the process encrypts the file creating another file called ‘TestMessage.txt.gpg’.

From here, the ‘TestMessage.txt.gpg’ is sent back to the desktop computer to then be decrypted.

2.2.3.6 Decrypting the file

To decrypt the ‘TestMessage.txt.gpg’ file, selecting the ‘Decrypt/Verify…’ button in Kleopatra opens a window to select the encrypted file. Once selected the file, the passphrase for the ‘Johnny’ user must be entered. Once the passphrase has been entered, the following window should appear indicating that the file has been decrypted successfully.



Output folder: C:/Users/jmcel/Desktop

All operations completed.

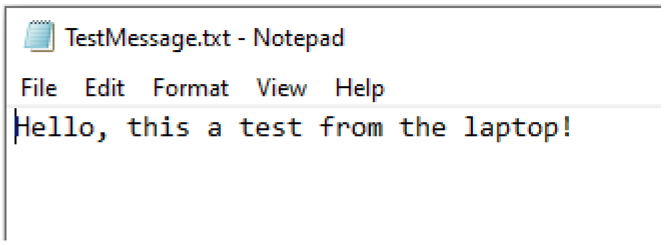
100%

TestMessage.txt.gpg → TestMessage.txt: **Decryption succeeded.**

Note: You cannot be sure who encrypted this message as it is not signed.

Figure 27: Successful Decryption

Saving the file to the desktop and opening it in Notepad ++, the contents of the original text file can be seen. The contents can be seen below.



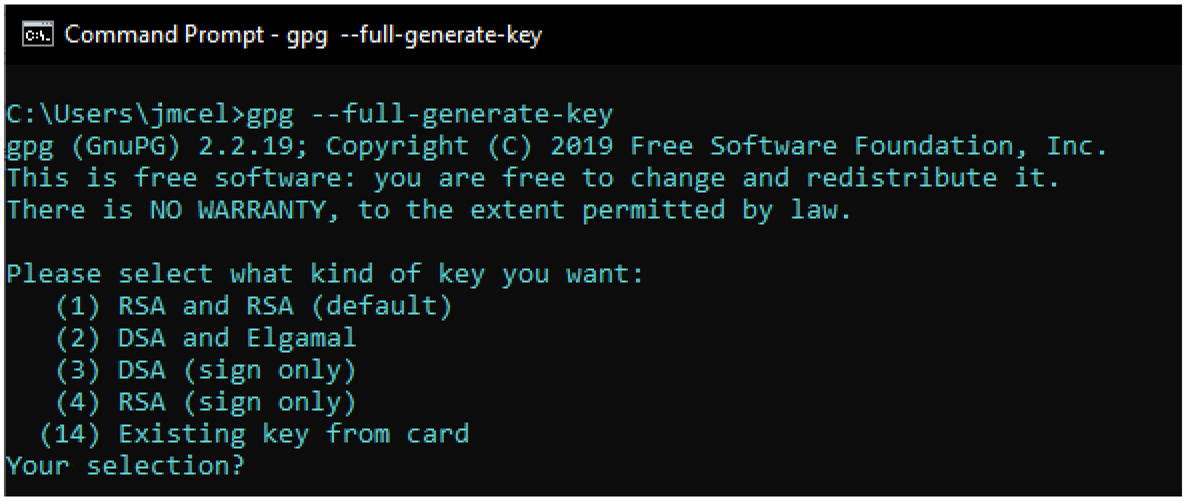
```
Hello, this a test from the laptop!
```

Figure 28: Plain Text Message

2.2.4 CLI (Command Line Interface)

2.2.4.1 Generating Key Pair

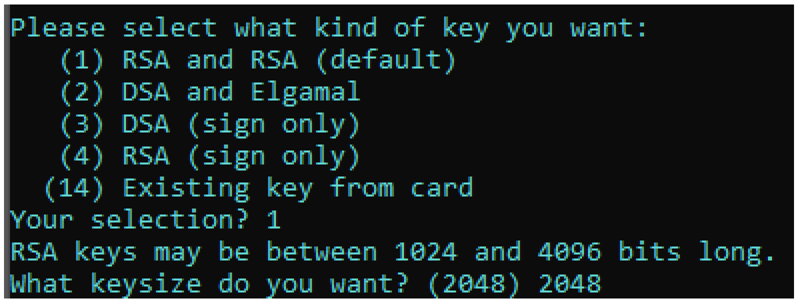
The first step in using gpg4win is to generate a private key. This can be done by using the command 'gpg --full-generate-key'. This will display the screen as shown below.



```
Command Prompt - gpg  --full-generate-key

C:\Users\jmcel>gpg --full-generate-key
gpg (GnuPG) 2.2.19; Copyright (C) 2019 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
  (14) Existing key from card
Your selection?
```

Figure 29: Generate Key

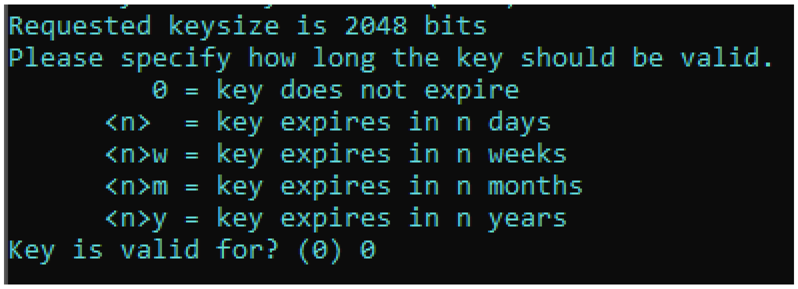
From here, the selection made is '1' for 'RSA and RSA (default)'. The keysize will then be asked, for example purposes the default value of 2048 will be used, but a bigger keysize could be used in practice. This can be seen below.



```
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
  (14) Existing key from card
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048) 2048
```

Figure 30: Type of Key

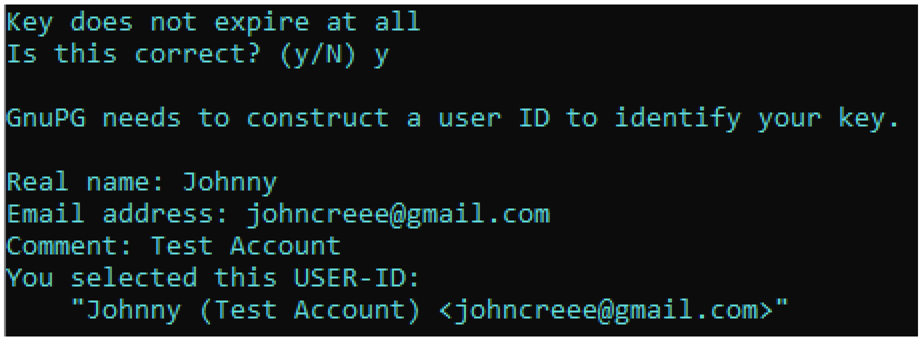
Then the time to live of the key is asked for, this can be days, weeks, months, years or indefinitely. For indefinitely '0' is entered. This was used in this case. This can be seen below.



```
Requested keysize is 2048 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 0
```

Figure 31: Keysize

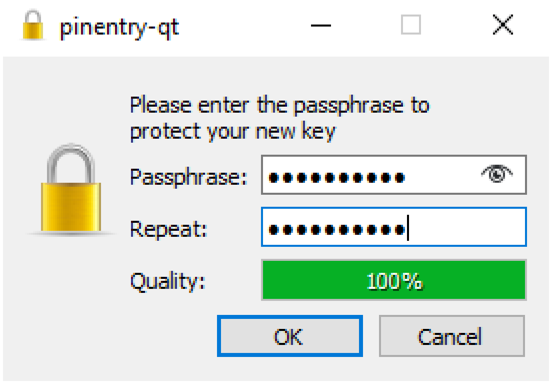
A confirmation message will appear on screen to confirm that the key is to last indefinitely. Typing ‘y’ passes this check. From there, details are required for linking the key. As can be seen in the screenshot below test details were entered for this example.



```
Key does not expire at all
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Real name: Johnny
Email address: johncreee@gmail.com
Comment: Test Account
You selected this USER-ID:
    "Johnny (Test Account) <johncreee@gmail.com>"
```

Figure 32: Entered Details

Once the details are correct, entering ‘o’ and pressing enter confirms the details and a window will appear prompting for the entry of a passphrase. In this case the passphrase ‘Hacklab123’ was used. This was entered into the window as can be seen below in figure 33.



pinentry-qt

Please enter the passphrase to protect your new key

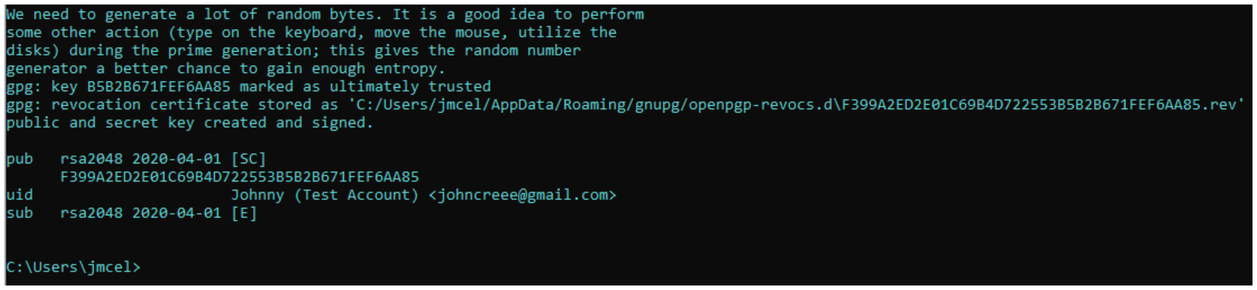
Passphrase: ••••••••••

Repeat: ••••••••••

Quality: 100%

Figure 33: Passphrase

Once completed the following message should appear in the command prompt.

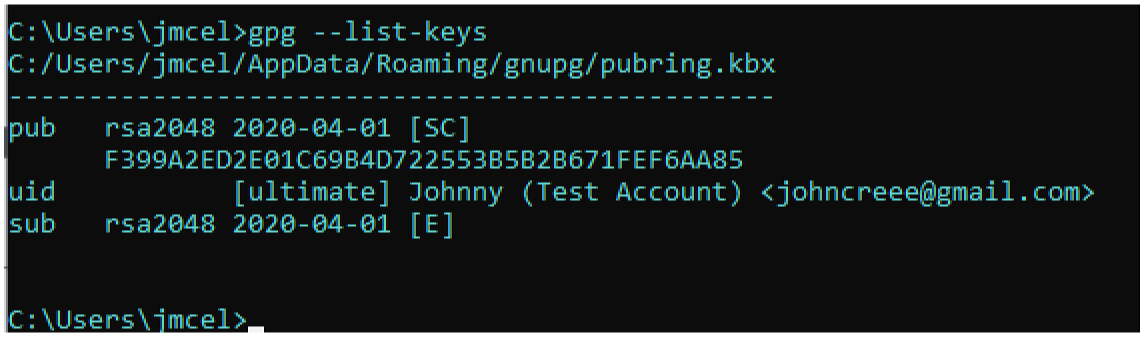


```
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: key B5B2B671FEF6AA85 marked as ultimately trusted
gpg: revocation certificate stored as 'C:/Users/jmcel/AppData/Roaming/gnupg/openpgp-revocs.d\F399A2ED2E01C69B4D722553B5B2B671FEF6AA85.rev'
public and secret key created and signed.

pub   rsa2048 2020-04-01 [SC]
      F399A2ED2E01C69B4D722553B5B2B671FEF6AA85
uid                      Johnny (Test Account) <johncreee@gmail.com>
sub   rsa2048 2020-04-01 [E]


C:\Users\jmcel>
```

Figure 34: Successful Creation of Key Pair

This indicates that a public and private key have been signed to the created account and where the certificate has been stored on the local machine.

To ensure that the key has been created successfully, entering the command 'gpg --list-keys' will display the following screen with the list of stored keys that have been generated.

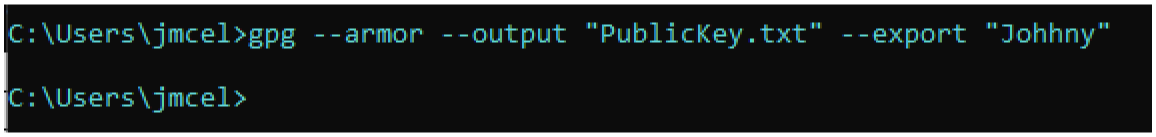


```
C:\Users\jmcel>gpg --list-keys
C:/Users/jmcel/AppData/Roaming/gnupg/pubring.kbx
------------------------------------------------
pub   rsa2048 2020-04-01 [SC]
      F399A2ED2E01C69B4D722553B5B2B671FEF6AA85
uid           [ultimate] Johnny (Test Account) <johncreee@gmail.com>
sub   rsa2048 2020-04-01 [E]


C:\Users\jmcel>
```

Figure 35: Confirm Key Creation

Navigating to the file path specified in figure 35 above, the revocation certificate can be seen. The file contents can be seen in appendix A.

The next step is to export the public key to be able to distribute it to other people. This can be done using the following command: 'gpg --armor --output "PublicKey.txt" --export "Johhny". This indicates to output the public key from the user "Johhny" to the file "PublicKey.txt" in the current directory. The command can be seen being used in the screenshot below.



```
C:\Users\jmcel>gpg --armor --output "PublicKey.txt" --export "Johhny"

C:\Users\jmcel>
```

Figure 36: Output Public Key

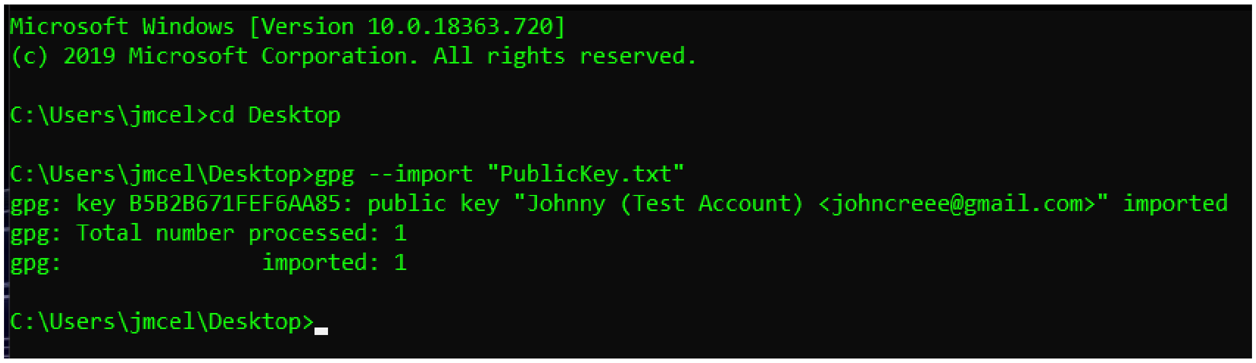
The full contents of the PublicKey.txt file can be found in appendix B.

2.2.4.2 Sending Public Key and Encrypting Data

Note: From here on in section 2.3.2, there is two computers. One will be referenced as desktop computer, the one with the private key, and the other a laptop which will have the public key.

Now that a public key has been generated and saved. To then start using it, first the key must be distributed to who is required to send data to you. In this case, another computer was installed with the same software using the same method, but with no sets of keys registered.

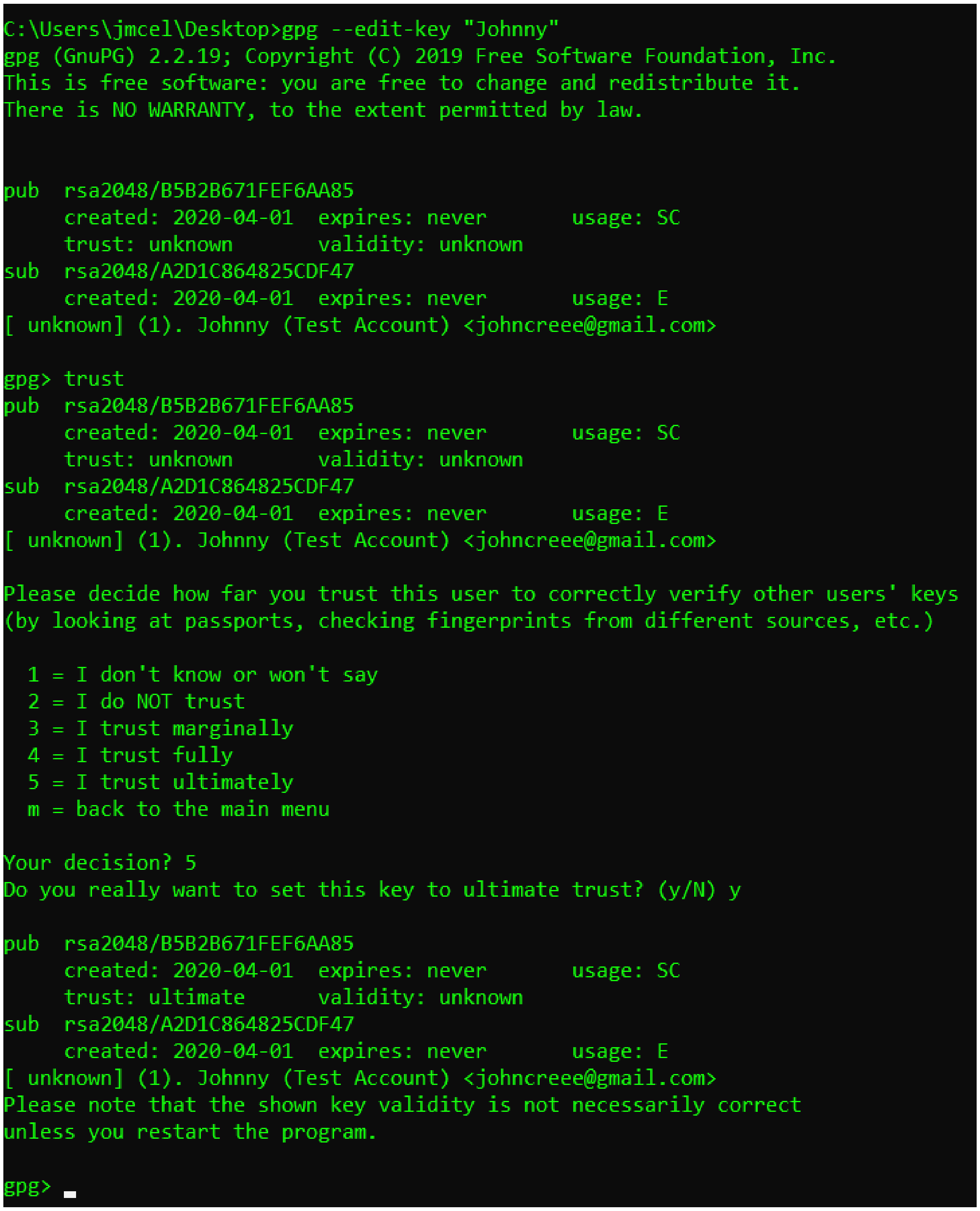
To import the key onto the laptop, first the PublicKey.txt file was copied over from the Desktop. From here, typing the command 'gpg –import "PublicKey.txt"' imports the public key from the user 'Johnny'. This can be seen below in figure 37.



```
Microsoft Windows [Version 10.0.18363.720]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\jmcel>cd Desktop

C:\Users\jmcel\Desktop>gpg --import "PublicKey.txt"
gpg: key B5B2B671FEF6AA85: public key "Johnny (Test Account) <johncreee@gmail.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1

C:\Users\jmcel\Desktop>
```

Figure 37: Import Public Key

Once the key has been imported, the 'trust' attribute of the key needs to be configured. To configure the program to trust the imported key, the command 'gpg --edit-key "Johnny" is used. This will open the gpg edit command prompt. From here typing 'trust' then choosing the option '5 = I trust ultimately', the prompt will then ask If you are sure you would like to trust this key completely. Typing 'y' confirms the edit. The whole process can be seen in figure 38 below.

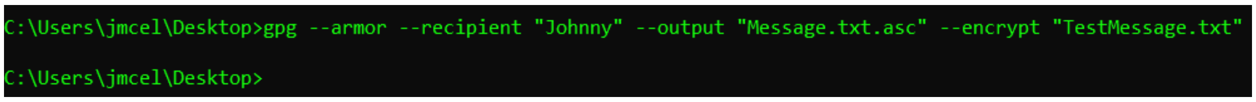


```
C:\Users\jmcel\Desktop>gpg --edit-key "Johnny"
gpg (GnuPG) 2.2.19; Copyright (C) 2019 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.


pub  rsa2048/B5B2B671FEF6AA85
     created: 2020-04-01  expires: never       usage: SC
     trust: unknown       validity: unknown
sub  rsa2048/A2D1C864825CDF47
     created: 2020-04-01  expires: never       usage: E
[ unknown] (1). Johnny (Test Account) <johncreee@gmail.com>

gpg> trust
pub  rsa2048/B5B2B671FEF6AA85
     created: 2020-04-01  expires: never       usage: SC
     trust: unknown       validity: unknown
sub  rsa2048/A2D1C864825CDF47
     created: 2020-04-01  expires: never       usage: E
[ unknown] (1). Johnny (Test Account) <johncreee@gmail.com>

Please decide how far you trust this user to correctly verify other users' keys
(by looking at passports, checking fingerprints from different sources, etc.)

  1 = I don't know or won't say
  2 = I do NOT trust
  3 = I trust marginally
  4 = I trust fully
  5 = I trust ultimately
  m = back to the main menu

Your decision? 5
Do you really want to set this key to ultimate trust? (y/N) y

pub  rsa2048/B5B2B671FEF6AA85
     created: 2020-04-01  expires: never       usage: SC
     trust: ultimate      validity: unknown
sub  rsa2048/A2D1C864825CDF47
     created: 2020-04-01  expires: never       usage: E
[ unknown] (1). Johnny (Test Account) <johncreee@gmail.com>
Please note that the shown key validity is not necessarily correct
unless you restart the program.

gpg>
```

Figure 38: Editing Trust Configuration

To encrypt a message, first open notepad and type in a message. In this case, the file was saved as "TestMessage.txt". Using the command 'gpg –armor –recipient "Johnny" –output "Message.asc"

–encrypt "TestMessage.txt"' can be used to encrypt the created message file. The command can be seen in figure 39 below.



```
C:\Users\jmcel\Desktop>gpg --armor --recipient "Johnny" --output "Message.txt.asc" --encrypt "TestMessage.txt"

C:\Users\jmcel\Desktop>
```

Figure 39: Encrypting Message

This will create a file called 'Message.txt.asc'. This can be seen in the screenshot below.

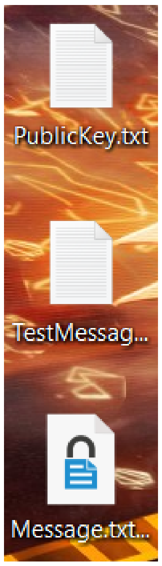
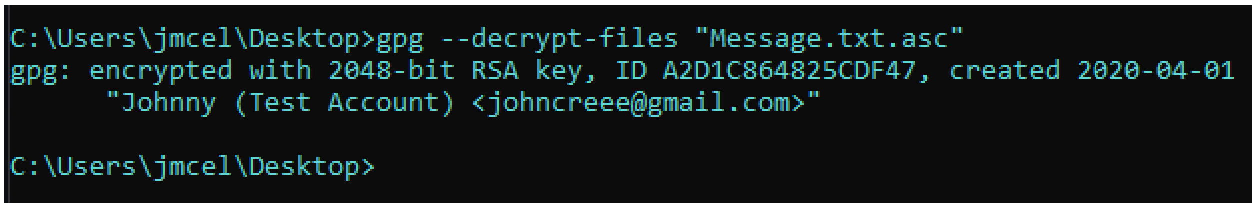


Figure 40: Outputted File

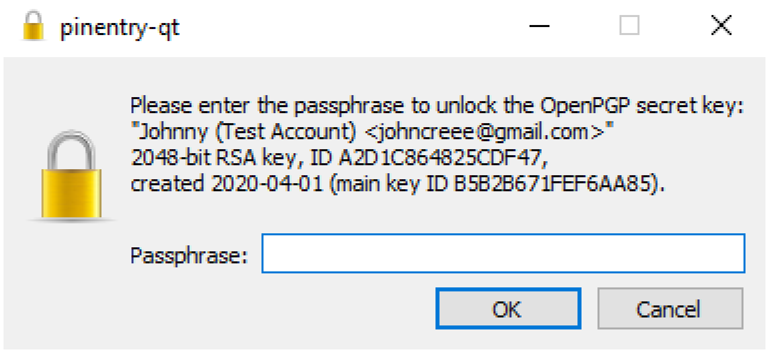
From here, the 'Message.txt.asc' file must be transferred over to the desktop computer. In the real world, this can be done by email or USB stick or any other method of transferring files. Once the file has been moved over, in the command prompt, using the command 'gpg --decrypt-files "Message.txt.asc"', will decrypt the file. This can be seen in the screenshot below labelled figure 41.



```
C:\Users\jmcel\Desktop>gpg --decrypt-files "Message.txt.asc"
gpg: encrypted with 2048-bit RSA key, ID A2D1C864825CDF47, created 2020-04-01
      "Johnny (Test Account) <johncreee@gmail.com>"

C:\Users\jmcel\Desktop>
```

Figure 41: Decrypt File

A password prompt may pop up for the 'Johnny' user, in which the password implemented was 'Hacklab123'. This can be seen below.



Please enter the passphrase to unlock the OpenPGP secret key:
"Johnny (Test Account) <johncreee@gmail.com>"
2048-bit RSA key, ID A2D1C864825CDF47,
created 2020-04-01 (main key ID B5B2B671FEF6AA85).

Passphrase:

Figure 42: Enter Private Key Passphrase

The message file should then be decrypted, and another file should have been generated with the message inside. This generated file can be seen in figure 43 below.

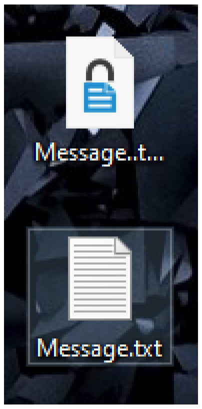
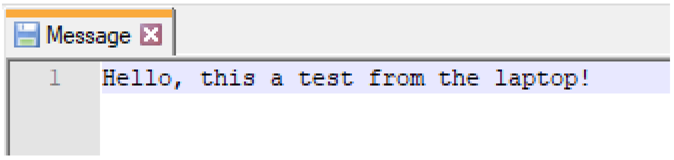


Figure 43: Decrypted File

Opening the file, it can be seen that the message has been decrypted successfully. The message reads 'Hello, this is a test from the laptop!". The file and contents can be seen in the figure below.



```
Hello, this a test from the laptop!
```

Figure 44: Plain Text File

1.2 RSA Encryption

Both HTTPS and Kleopatra uses an asymmetric key structure protocol through SSL/TLS, this is known as the RSA encryption method. This method of encryption uses a public key to encrypt the data and a private key to decrypt the data.

To craft the public key, the algorithm uses two prime numbers (p & q), calculates the product of them (n) and the result is one of two parts of the public key. The second part is calculated by taking an exponent (e), that is co-prime with (n). This means that (n) and (e) have no common factors. The values (n) and (e) are used in conjunction with each other to create the public key.

The first step in crafting the private key, the totient of (n) is calculated. This is the product of (p-1) and (q-1). The next step is to create the key itself; this is done by taking the totient calculated, multiplying it by two, adding one and dividing it by (e). The value calculated is assigned (d) and is the value for the private key.

From here, to encrypt the data. The equation $c = M^{e} mod\, n$ is used. This is where (e) and (n) have already been calculated for the public key, (M) is the unencrypted message and (c) is the encrypted message.

To decrypt the message, the equation $D = c^{d} mod\, n$ is used. This is where (n) and (d) have already been calculated for the private key, (c) is the encrypted message and (D) is the decrypted message.

Below is a working example of how the mathematics work. To make it easier to follow, smaller numbers have been used. In practice, larger numbers would be used to make it harder to guess and allow it to be more secure.

Generating Public Key

Let's take p = 67, q = 79

$n = pq$

$n = 67 \times 79$

$n = 5293$

Small exponent e:

Must be coprime with n

Not a factor of n

1 < e < ϕ(n)

Let's take: e = 7

Generating Private Key

Calculate $\phi(n)$:

$\phi(n) \ = \ (p \ - \ 1) \ (q \ - \ 1)$

$\phi(n) \ = \ (66) \ (78)$

$\phi(n) \ = \ 5148$

Calculate Private key d:

$d \ = \ \frac{2(\phi(n))+1}{e}$

$d \ = \ \frac{2(5148)+1}{7}$

$d \ = \ \frac{10297}{7}$

$d \ = \ 1471$

Encrypting Data with public key

Encrypt the numbers '1413'

Calculate Encrypted data:

$c \ = M^{e} mod\ n$

$c \ = 1413^{7} mod\ 5293$

The mod function present in the equation above means that
$c \ = \ remainder\ of \left\{ \frac{1413*1413*1413*1413*1413*1413*1413}{5293} \right\}$

This can also be calculated using the Extended Euclidean Algorithm

$c \ = \ 1124592370792022535671 7\ mod\ 5293$

$c \ = \ 1350$

Decrypt the data

So far, we have c being the encrypted data, d being the primary key and n being the public key.

Calculate D the Decrypted data:

$D \ = c^{d} mod\ n$

$D \ = 1350^{1471} mod\ 5293$

$D \ = \ 1413$

D = 1413, The decrypted message.

M = 1413, The starting unencrypted message.

1.3 Storing data securely

Not only is transmitting data important but storing data is one of the most important parts of data security. This is due to the data being sat in one place, this makes it a target for malicious attackers as they have one point to attack rather than trying to grab the data in transit. If a website was using HTTPS to securely transmit data to and from the webserver, what happens if sensitive information is needing to be stored on the webserver? Like a password for a login portal for example. One of the most common pieces of data that is targeted is passwords, this is due to the information and accounts that could be accessed using these passwords. To try and store passwords safely and securely, hashing algorithms were developed to obscure the representation of the password. There are many different hashing algorithms available in the present day. In this paper, the following hashing algorithms will be discussed: MD5 (Message-Digest 5), the SHA (Secure Hashing Algorithm) family including SHA-224, SHA-256, SHA-384, SHA-512. Each one of these hashing algorithms can hash sensitive information and appear to obscure the data, however some of the algorithms are more secure than others.

The MD5 hashing algorithm was first implemented in 1991 by Ronald Rivest. When released, the hashing algorithm was widely used for data security. Although now in the present day, the algorithm is found to be unfit for securing data. This is due to MD5 being a fast algorithm to use and run. MD5 produces a hash that is 128-bits in size, this is a relatively small hash when compared to other algorithms.

The SHA family of hashing algorithms have been in development for many years. The family consists of SHA-0, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512. The first four of these from SHA-0 to SHA-256 work on a 512-bit message segment divided into 32-bit words and the last two working on a 1024-bit message segment divided into 64-bit words. Each of the algorithms produce a different length of output. SHA-1, which replaced SHA-0, produces an output of length 160 bits. SHA-224 produces a 224-bit output, SHA-256 a 256-bit output, SHA-384 a 384-bit output and SHA-512 a 512-bit output.

Taking into consideration the above hashing algorithms. Looking at the hash of a simple string can give the idea of a level of security they provide. Listed below is the hashing algorithms with the simple string 'Hello' being hashed.

MD5 - 8b1a9953c4611296a827abf8c47804d7

SHA-1 – f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0

SHA-224 - 4149da18aa8bfc2b1e382c6c26556d01a92c261b6436dad5e3be3fcc

SHA-256 - 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969

SHA-384 –

3519fe5ad2c596efe3e276a6f351b8fc0b03db861782490d45f7598ebd0ab5fd5520ed102f38c4a5ec834e9
8668035fc

SHA-512 –

3615f80c9d293ed7402687f94b22d58e529b8cc7916f8fac7fddf7fbd5af4cf777d3d795a7a00a16bf7e7f3fb 9561ee9baae480da9fe7a18769e71886b03f315

Previously in CMP202 Data Structures and Algorithms 2 in 2nd year Ethical Hacking, I implemented a multi-threaded password cracker as my coursework. This worked by using one CPU (Central Processing Unit) thread as a dictionary attack where a word list file was read from and each word was placed through the specified hashing algorithm from the choice of MD5 or SHA256. The other threads use a brute force method by attempting all possibilities for each number of characters up to the maximum number of threads the CPU has. E.g. First thread tries all 1-character passwords, second thread tries 2-character passwords and so on.

During the development of this paper however, the algorithm has had multiple improvements and changes made to it. First, more hashing options were added. The algorithm now includes the following hashing algorithms: MD5, SHA224, SHA 256, SHA384, SHA512. The mathematical files to be able to hash values were taken from (Code for Monkeys - C++, PHP, CSS, Programming Resource :: zedwood.com, 2020). This allows for a wider range of hashes to be cracked using this algorithm. The program has the option to run using just one hashing algorithm to try and crack the inputted hash, or to run using all the hashing algorithms at the same time to try and crack the inputted hash. This allows for a more functional and modular program. The full source code for the password cracker can be seen in appendix C.

To demonstrate which hashing algorithm is more secure than the other, a selection of phrases/passwords were chosen to be attempted to be cracked. These included: ‘password’, ‘qwerty’, ‘h4ck. These phrases were tested in the password cracker with MD5, SHA-224, SHA-256, SHA-384 and SHA-512 hashes being tested. The inputted hashes can be seen below:

<u>MD5:</u>

password - 5f4dcc3b5aa765d61d8327deb882cf99

qwerty – d8578edf8458ce06fbc5bb76a58c5ca4

h4ck – 0ed5f1f056b1d96122afaae306d3dd65

<u>SHA-224:</u>

password – d63dc919e201d7bc4c825630d2cf25fdc93d4b2f0d46706d29038d01

qwerty – 5154aaa49392fb275ce7e12a7d3e00901cf9cf3ab10491673f97322f

h4ck – 906022a0f5bff263c5b9cfaa11a6c9270a456d2c7b30e6b50a0a25cc

<u>SHA-256:</u>

password - 5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8

qwerty – 65e84be33532fb784c48129675f9eff3a682b27168c0ea744b2cf58ee02337c5

h4ck – caf33ed88e8d64d24f58a675f8da6491cd91eec1c2cde5fbeb0773cec2d1fb63

<u>SHA-384:</u>

password –

a8b64babd0aca91a59bdbb7761b421d4f2bb38280d3a75ba0f21f2bebc45583d446c598660c94ce680c47d19c30783a7

qwerty –

1ab60e110d41a9aac5e30d086c490819bfe3461b38c76b9602fe9686aa0aa3d28c63c96a1019e3788c40a14f4292e50f

h4ck -
ad6851ee3f95d51d3b818c8c34d1cc862fa5180c41a3876ad52ac11f76c7d9bb9efc751583555e6cf9d950e11d7fad79

<u>SHA-512:</u>

password –

b109f3bbbc244eb82441917ed06d618b9008dd09b3befd1b5e07394c706a8bb980b1d7785e5976ec049b46df5f1326af5a2ea6d103fd07c95385ffab0cacbc86

qwerty –

0dd3e512642c97ca3f747f9a76e374fbda73f9292823c0313be9d78add7cdd8f72235af0c553dd26797e78e1854edee0ae002f8aba074b066dfce1af114e32f8

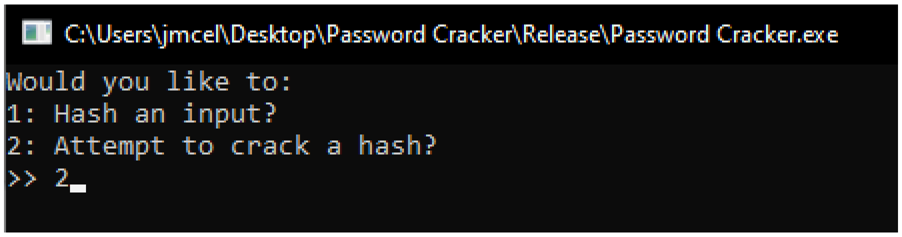
h4ck –

4d99f965b71a0de9bc8a0c5b9e153291e7aede241d8487662c7527429c8537842138280fb9f1262cc56d73e6d27b20a4a5b39af1af5c807522bb3d56a7b7856e

Using the password cracker algorithm that had been implemented along with these hashes will give an indication on the amount of time that requires to crack each hash through brute force and dictionary attacks. The dictionary attack was carried out using a wordlist from online (danielmiessler/SecLists, 2020). This has 10,000 common passwords that can be hashed and compared to the input hash, this file was saved and renamed to 'Wordlist.txt' in the same directory as the password cracker program.

To be able to carry out the comparison, the program must be used to crack each of these hashes using individual hashing algorithms. Below are the steps taken to carry out one of these actions.

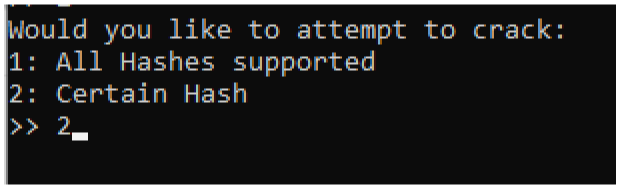
When first running the program, the program will ask if the user would like to attempt to crack a hash, or to hash a string. Entering '2' to crack a hash is used.



```
C:\Users\jmcel\Desktop\Password Cracker\Release\Password Cracker.exe
Would you like to:
1: Hash an input?
2: Attempt to crack a hash?
>> 2
```

Figure 45: Function Selection

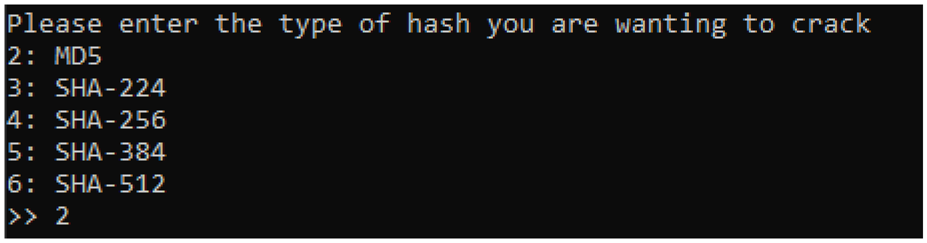
From here, the options that follow allow the user to either compare that entered hash to all possible hash types this program supports or to compare against a certain type of hashing algorithm. Since in this case it is to determine which singular hashing algorithm is better than the other, '2' is entered.



```
Would you like to attempt to crack:
1: All Hashes supported
2: Certain Hash
>> 2
```

Figure 46: Hash Function Selection

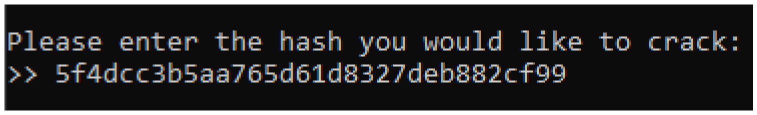
Displayed is a list of the hashing algorithms that can be tested listed from 2-6. In this case MD5 is being used, entering '2' allows for the use of testing against MD5 hashes.



```
Please enter the type of hash you are wanting to crack
2: MD5
3: SHA-224
4: SHA-256
5: SHA-384
6: SHA-512
>> 2
```

Figure 47: Hash Selection

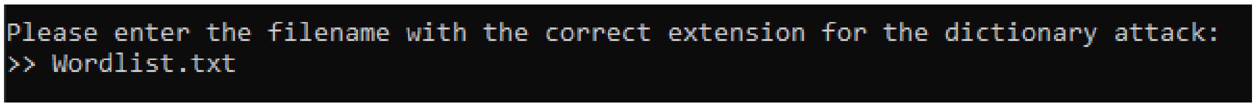
The next step is to input the hash that is wanting to be cracked, this can be seen below with the hash of the phrase 'password' being entered.



```
Please enter the hash you would like to crack:
>> 5f4dcc3b5aa765d61d8327deb882cf99
```

Figure 48: Enter Hash

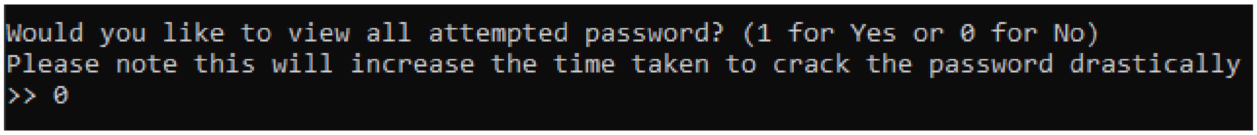
Next is to input the filename of a wordlist file for the dictionary attack. In this case the file that was downloaded and being used is the 'Wordlist.txt' file. This can be seen entered in the screenshot below.



```
Please enter the filename with the correct extension for the dictionary attack:
>> Wordlist.txt
```

Figure 49: Enter Dictionary File

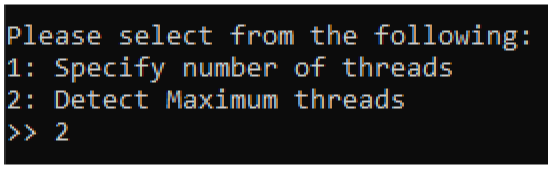
The next option is to display all attempted passwords. Selecting yes at this stage would dramatically increase the time taken to crack any hashed phrase. So to give more accurate sense of which algorithm is faster, no is selected. This is carried out by entering '0' at this stage.



```
Would you like to view all attempted password? (1 for Yes or 0 for No)
Please note this will increase the time taken to crack the password drastically
>> 0
```

Figure 50: View Attempted Passwords

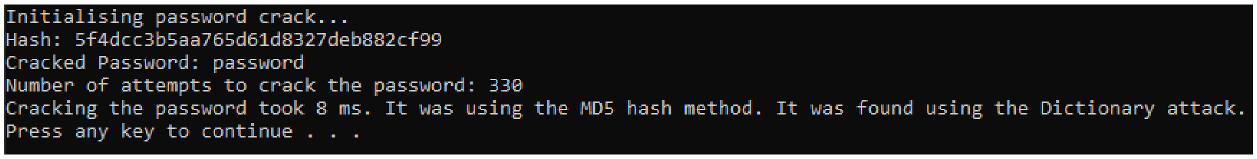
The program will then ask the user if they would like to specify the number of CPU threads to use. To make it fair for all tests, the option to detect the maximum number of threads on the computer and use them will be selected. This can be carried out by entering '2' as seen below in figure 51.



```
Please select from the following:
1: Specify number of threads
2: Detect Maximum threads
>> 2
```

Figure 51: Select Threads

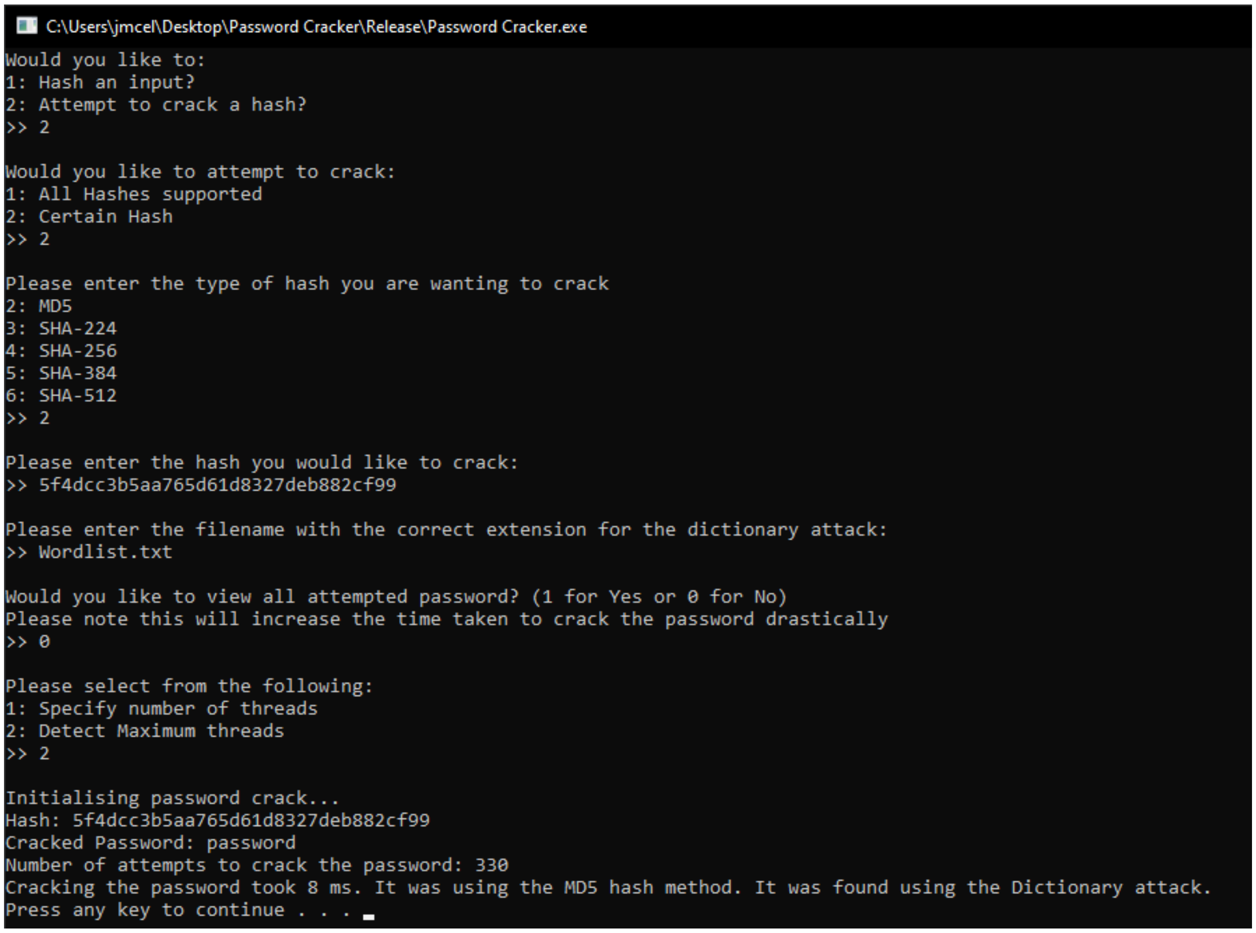
Hitting enter after this prompt will start the password cracking. Seen below is the output format of the program with the entered hash, the cracked password, the number of attempts it took to crack the password, the number of milliseconds it took to crack, the hashing algorithm that was used and the type of attack the password was found using.



```
Initialising password crack...
Hash: 5f4dcc3b5aa765d61d8327deb882cf99
Cracked Password: password
Number of attempts to crack the password: 330
Cracking the password took 8 ms. It was using the MD5 hash method. It was found using the Dictionary attack.
Press any key to continue . . .
```

Figure 52: Output

The full procedural output of the steps carried out above can be seen in figure 53 below.



```
C:\Users\jmcel\Desktop\Password Cracker\Release\Password Cracker.exe
Would you like to:
1: Hash an input?
2: Attempt to crack a hash?
>> 2

Would you like to attempt to crack:
1: All Hashes supported
2: Certain Hash
>> 2

Please enter the type of hash you are wanting to crack
2: MD5
3: SHA-224
4: SHA-256
5: SHA-384
6: SHA-512
>> 2

Please enter the hash you would like to crack:
>> 5f4dcc3b5aa765d61d8327deb882cf99

Please enter the filename with the correct extension for the dictionary attack:
>> Wordlist.txt

Would you like to view all attempted password? (1 for Yes or 0 for No)
Please note this will increase the time taken to crack the password drastically
>> 0

Please select from the following:
1: Specify number of threads
2: Detect Maximum threads
>> 2

Initialising password crack...
Hash: 5f4dcc3b5aa765d61d8327deb882cf99
Cracked Password: password
Number of attempts to crack the password: 330
Cracking the password took 8 ms. It was using the MD5 hash method. It was found using the Dictionary attack.
Press any key to continue . . .
```

Figure 53: Full Procedure

These steps are repeated 5 times for each of the hashes listed above for each hashing algorithm. The results of the carried-out procedure can be seen below. These numbers are relative to the machine they are running on as the CPU speed and core/thread count is an essential aspect to the program running. The following results were carried out on an Intel Core i5 9600k, 6 Core 6 Thread CPU, overclocked to 4.8 GHz on all cores, with 32GB of 2666MHz DDR4 RAM. The results have been split into two separate tables, one of which records the average time that is taken to crack each hash, and one with the average number of attempts it takes to crack each hash.

MD5	Time 1	Time 2	Time 3	Time 4	Time 5	Average Time
password	8	9	10	9	8	8.8
qwerty	10	9	8	9	8	8.8
h4ck	13486	13514	13494	13596	13582	13534.4
SHA-224	Time 1	Time 2	Time 3	Time 4	Time 5	Average Time
password	9	10	10	8	8	9
qwerty	8	10	8	9	9	8.8
h4ck	25637	25728	25543	25677	25574	25631.8
SHA-256	Time 1	Time 2	Time 3	Time 4	Time 5	Average Time
password	8	8	10	10	10	9.2
qwerty	11	9	10	10	11	10.2
h4ck	28535	28738	28673	29715	29580	29048.2
SHA-384	Time 1	Time 2	Time 3	Time 4	Time 5	Average Time
password	8	9	9	10	11	9.4
qwerty	10	8	11	9	8	9.2
h4ck	43302	41663	41997	41515	41545	42004.4
SHA-512	Time 1	Time 2	Time 3	Time 4	Time 5	Average Time
password	11	12	11	9	9	10.4
qwerty	11	8	9	9	10	9.4
h4ck	54320	55396	52433	53170	56286	54321

Figure 54: Average Time Taken

MD5	Number 1	Number 2	Number 3	Number 4	Number 5	Average Attempts
password	330	266	643	449	491	435.8
qwerty	967	598	275	296	341	495.4
h4ck	6178847	6190666	6193037	6186701	6177655	6185381.2
SHA-224	Time 1	Time 2	Time 3	Time 4	Time 5	Average Attempts
password	457	291	1018	704	152	524.4
qwerty	136	622	166	338	489	350.2
h4ck	6130058	6115808	6112918	6107792	6118382	6116991.6
SHA-256	Time 1	Time 2	Time 3	Time 4	Time 5	Average Attempts
password	132	148	198	395	208	216.2
qwerty	416	216	459	515	503	421.8
h4ck	5681083	5762855	5769557	5932745	5967011	5822650.2
SHA-384	Time 1	Time 2	Time 3	Time 4	Time 5	Average Attempts
password	149	132	150	168	536	227
qwerty	260	92	505	89	94	208
h4ck	5692543	5413609	5393001	5403932	5409344	5462485.8
SHA-512	Time 1	Time 2	Time 3	Time 4	Time 5	Average Attempts
password	484	785	647	445	335	539.2
qwerty	481	180	351	229	413	330.8
h4ck	5424822	5436279	5375960	5446452	5451421	5426986.8

Figure 55: Average Attempts Taken

Taking the above results and plotting graphs from them shows the trend with each hashing algorithm. Looking at figure 56 and 57, it can be seen that the number of attempts to be able to crack the password drastically decreases with the complexity of the algorithm, with the number of attempted passwords for MD5 being over 6 million in the space of 13 seconds and SHA-512 having 5.4million attempts in the space of 54 seconds. This is due to the drastic increase in time that it takes for one hash to be calculated for a more complex algorithm such as SHA-512, meaning that less attempts can be made in the same length of time for an easier algorithm to run.

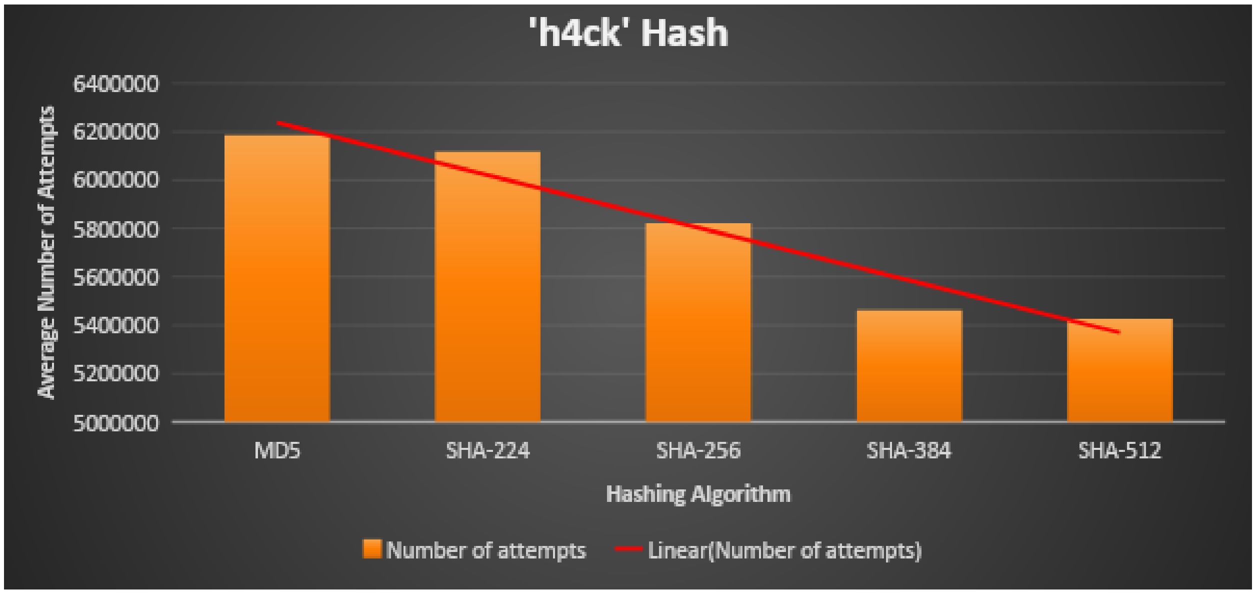


Figure 56: Number of Attempts again Hashing Algorithm

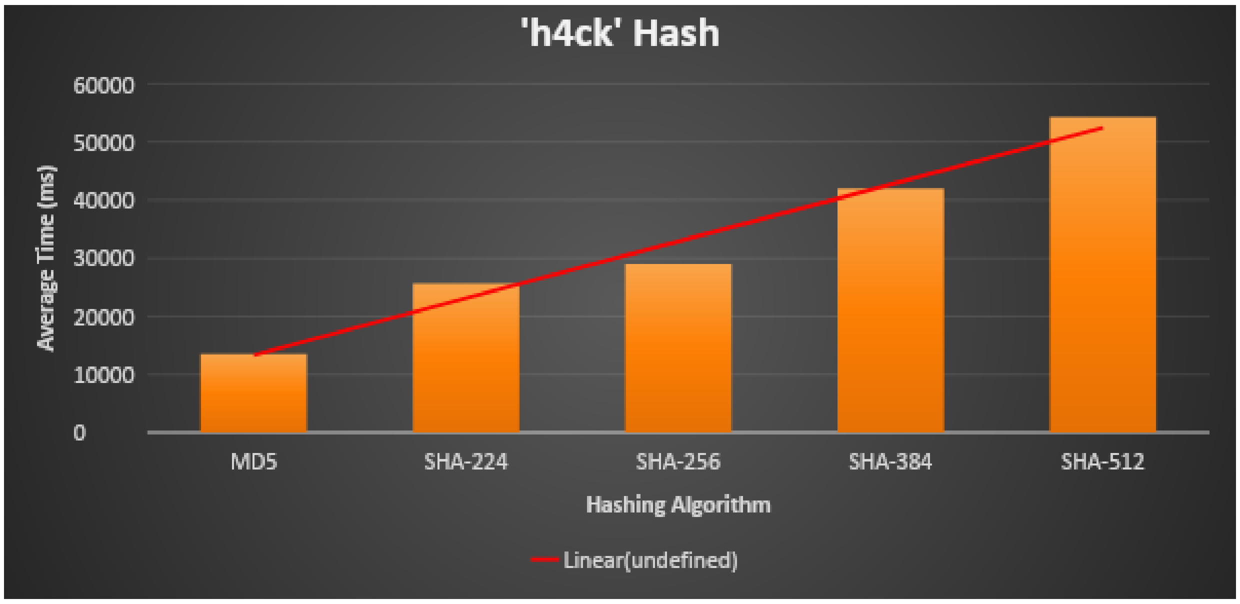
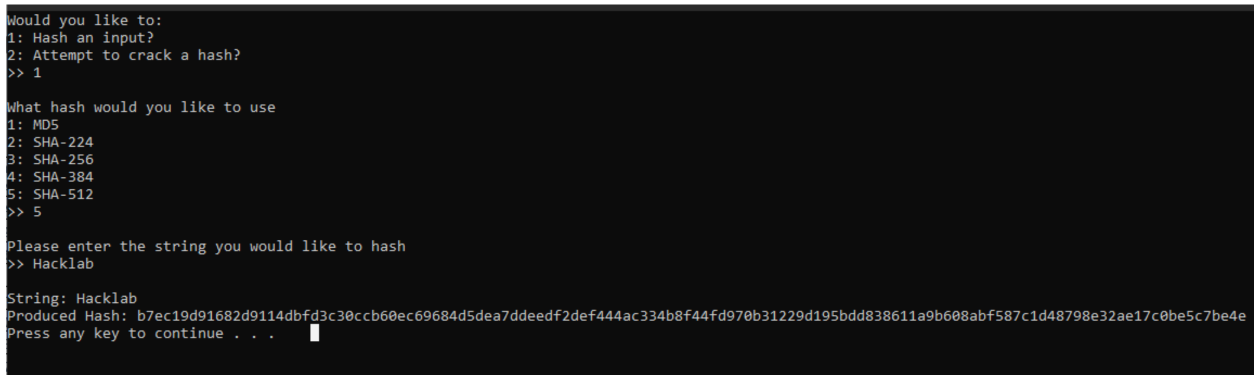


Figure 57: Time Taken for Each Hashing Algorithm

As can be seen from the gathered results, the SHA family of hashes, from the ones tested above, require a much longer time to be able to crack the same password used in all hashing algorithms. This means that there is a much lower number of passwords being attempted per second. MD5 can be seen to have many more attempts in a shorter time than SHA-256 and above algorithms. From the results, when attempting to crack a MD5 hash, the program attempts around 450,000 hashes per second, whereas SHA-256 attempts around 200,000 hashes and SHA-512 attempting around 100,000 per second. A lower number of hashes attempted per second means that it would take a much longer time to be able to crack any hash.

Along with the high number of hashes that can be tested per second, another reason as to why MD5 being considered insecure for use is that as it has been out for a long time, there are large dictionary tables with cracked hashes in them showing the passwords in plain text. An example of an online dictionary is MD5Online (https://www.md5online.org/), they have over 1,150 billion cracked passwords using MD5 hashes, this means that 1,150 billion passwords have been cracked and are available to be checked online at any time. MD5 is also known to have hash collisions, this is where different data, when hashed, has the same hashed value as each other. In terms of a webserver and passwords in a database, this might mean that a different password could be used instead of the actual password, as they turn out to have the same hash value. This in turn indicates that the SHA family are a much more suitable hashing algorithm for securing sensitive data. SHA-512 is the most secure hashing algorithm out of the collection tested.

The implemented password cracker program allows for any inputted string to be hashed using one of the listed algorithms in the program. This can allow the user to keep their information secure by hashing any data that could be considered sensitive. The simple process of using the password cracker program to hash the entered value of "Hacklab" and return the hash: "b7ec19d91682d9114dbfd3c30ccb60ec69684d5dea7ddeedf2def444ac334b8f44fd970b31229d195bdd838611a9b608abf587c1d48798e32ae17c0be5c7be4e" can be seen below in figure 58.



```
Would you like to:
1: Hash an input?
2: Attempt to crack a hash?
>> 1

What hash would you like to use
1: MD5
2: SHA-224
3: SHA-256
4: SHA-384
5: SHA-512
>> 5

Please enter the string you would like to hash
>> Hacklab

String: Hacklab
Produced Hash: b7ec19d91682d9114dbfd3c30ccb60ec69684d5dea7ddeedf2def444ac334b8f44fd970b31229d195bdd838611a9b608abf587c1d48798e32ae17c0be5c7be4e
Press any key to continue . . .
```

Figure 58: Hashing a value

3 Discussion

3.1 General Discussion

All sensitive data in the modern day must be secured to prevent from malicious attackers gaining access to any sensitive information that may be transferred over a network or being stored on a computer and or database. This can be carried out using the methods discussed in this paper, using RSA through HTTPS or Kleopatra to transfer data and using hashing algorithms to store data securely. Without securing data properly, information can be captured and cracked by attackers with a malicious intent. Be this over a network or through a database, every aspect of the data transport life must be taken into consideration, without this user's information would be at risk.

Not hashing passwords with a suitable hashing algorithm can have a devastating effect. If a password database was to use the MD5 hashing algorithm as standard, if exploited and leaked. The passwords could easily be cracked in a very short time when compared to other algorithms. Using algorithms such as Sha-256, SHA-384, SHA-512 should be the standard in the modern day. These algorithms take more time to run which means that less attempts can be made per second at cracking the hash. These algorithms also do not currently have the large databases of already cracked hashes available like MD5 does. This in turn allows for a much more secure method in storing passwords.

3.2 Countermeasures

When trusting a website for using SSL/TLS encryption, it is possible to gain a fake SSL certificate. This means that it may appear that the network traffic is going to be sent encrypted and securely, this would however not be the case as the webserver would not actually have the encryption activated, it would just appear as it did. In conjunction with this, it is possible to have a fake 'padlock' at in the URL bar of a website. Again, this appears to be genuine but can trick the user into thinking that their traffic will be secure when being transmitted.

One of the possible countermeasures to the hashing algorithms discussed above has already been presented and tested in section 2.5 above. Password crackers can be used to brute force and/or dictionary attack any found hashes. Depending on the hardware used to run the password crackers, hundreds of thousands of passwords can be attempted per second to try and crack these passwords.

3.3 Conclusions

In conclusion to this paper, all sensitive data being sent or stored should be handled correctly and either be encrypted, hashed or obscured in a way that makes it very difficult for the attacker as can be seen using HTTP in the modern day to send data between points is not a secure way of transmitting data. HTTPS should always be used over HTTP when necessary. If unable to use HTTPS, then another encryption method such as Kleopatra should be used as a secure method in transferring data. Both of these methods use RSA encryption and are considered a secure way of transmitting data.

In conclusion to the gathered results from the password cracker, it can be seen that SHA-512 is the most secure hashing algorithm tested above, however it may be more suitable for smaller businesses that may not have the required computing power to use SHA-256 instead of SHA-512 as it takes longer to hash. SHA-256 is still considered a secure hashing algorithm but with more secure options available, consideration into what is being stored must be taken. MD5 should not be used to store any sensitive information as it is no longer considered secure due to the many vulnerabilities found in the algorithm.

3.4 Future Work

If more time was available, more research would be put into the mathematics and equations behind the different hashing algorithms such as MD5 and the SHA family of algorithms. This would allow for a further insight into the differences in mathematics between the algorithms and why one would be considered more complex and secure that the other.

More research could be carried out to find other methods of sending data securely over a network. This could include other types of encryption methods such as AES (Advanced Encryption Standard) or TripleDES (Triple Data Encryption Standard). Demonstrations of these methods along with explanations of the mathematics behind these methods could be researched and demonstrated to allow for a fuller understanding and comparison between each method.

Methods in which to try and break these types of encryption methods could be researched into and possibly demonstrated to indicate how strong each method is for securing data. Tools could possibly be developed like the password cracker used in this paper.

References

For URLs, Blogs:

Store.hp.com. (2020). *What Are the Different Types of Encryption? | HP® Tech Takes*. [online] Available at: https://store.hp.com/us/en/tech-takes/what-are-different-types-of-encryption [Accessed 2 Mar. 2020].

Anon, (2020). [online] Available at: https://www.cloudflare.com/learning/ssl/what-is-https/ [Accessed 2 Mar. 2020].

Md5online.org. 2020. *3 Reasons Why MD5 Is Not Secure | Md5online*. [online] Available at: https://www.md5online.org/blog/why-md5-is-not-safe/ [Accessed 31 March 2020].

En.wikipedia.org. 2020. *MD5*. [online] Available at: https://en.wikipedia.org/wiki/MD5 [Accessed 31 March 2020].

En.wikipedia.org. 2020. *SHA-2*. [online] Available at: https://en.wikipedia.org/wiki/SHA-2 [Accessed 31 March 2020].

KeyCDN. 2020. *What Is The Difference Between HTTP And HTTPS? - Keycdn*. [online] Available at: https://www.keycdn.com/blog/difference-between-http-and-https [Accessed 31 March 2020].

Hashed Out by The SSL Store™. 2020. *What Is The Difference Between SHA-1, SHA-2 And SHA-256?*. [online] Available at: https://www.thesslstore.com/blog/difference-sha-1-sha-2-sha-256-hash-algorithms/ [Accessed 31 March 2020].

Medium. 2020. *Why An Unsalted MD5 Hash Is Bad Practice*. [online] Available at: https://medium.com/@svanas/why-an-unsalted-md5-hash-is-bad-practice-6a0d7d017856 [Accessed 31 March 2020].

Vocell, J., 2020. *A Beginner's Guide To SSL: What It Is & Why It Makes Your Website More Secure*. [online] Blog.hubspot.com. Available at: https://blog.hubspot.com/marketing/what-is-ssl [Accessed 1 April 2020].

Streetdirectory.com. 2020. *What Is The Strongest Hash Algorithm?*. [online] Available at: https://www.streetdirectory.com/etoday/what-is-the-strongest-hash-algorithm-ejcluw.html [Accessed 27 April 2020].

2020. [online] Available at: https://www.cloudflare.com/learning/ssl/what-is-https/ [Accessed 27 April 2020].

Robert Heaton. 2020. *How Does HTTPS Actually Work? | Robert Heaton*. [online] Available at: https://robertheaton.com/2014/03/27/how-does-https-actually-work/ [Accessed 27 April 2020].

SearchSecurity. 2020. *Secure Data Transmission Methods*. [online] Available at: https://searchsecurity.techtarget.com/tip/Secure-data-transmission-methods [Accessed 27 April 2020].

Schneider Electric Blog. 2020. *Transporting Data Securely - Schneider Electric Blog*. [online] Available at: https://blog.se.com/industrial-software/2015/04/21/transporting-data-securely/ [Accessed 27 April 2020].

Md5online.org. 2020. *3 Reasons Why MD5 Is Not Secure – Md5online*. [online] Available at: https://www.md5online.org/blog/why-md5-is-not-safe/ [Accessed 28 April 2020].

Namecheap.com. 2020. *What Is An RSA Key Used For? - SSL Certificates - Namecheap.Com*. [online] Available at: https://www.namecheap.com/support/knowledgebase/article.aspx/798/67/what-is-an-rsa-key-used-for [Accessed 28 April 2020].

Digicert.com. 2020. *All About SSL Cryptography | Digicert.Com*. [online] Available at: https://www.digicert.com/ssl-cryptography.htm [Accessed 28 April 2020].

Difference between SHA 512, S., Mohan, V. and Ossifrage, S., 2020. *Difference Between SHA 512, SHA 512 Half, SHA 256*. [online] Cryptography Stack Exchange. Available at: https://crypto.stackexchange.com/questions/55658/difference-between-sha-512-sha-512-half-sha-256 [Accessed 29 April 2020].

integrity, S., S., D., Climent, M. and Rock, G., 2020. *SHA-256 Or MD5 For File Integrity*. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/14139727/sha-256-or-md5-for-file-integrity [Accessed 29 April 2020].

Maria, G., 2020. *4 Common Encryption Methods And Use Cases*. [online] GetApp Lab. Available at: https://lab.getapp.com/common-encryption-methods/ [Accessed 29 April 2020].

2020. [online] Available at: https://www.cloudflare.com/learning/ssl/why-is-http-not-secure/ [Accessed 1 May 2020].

SelfKey. 2020. *All Data Breaches In 2019 & 2020 - An Alarming Timeline - Selfkey*. [online] Available at: https://selfkey.org/data-breaches-in-2019/ [Accessed 1 May 2020].

Greenberg, A., 2020. *Hackers Are Passing Around A Megaleak Of 2.2 Billion Records*. [online] Wired. Available at: https://www.wired.com/story/collection-leak-usernames-passwords-billions/ [Accessed 1 May 2020].

Mathematical files for Password Cracker: Zedwood.com. 2020. *Code For Monkeys - C++, PHP, CSS, Programming Resource :: Zedwood.Com*. [online] Available at: http://www.zedwood.com/ [Accessed 30 April 2020].

Dictionary file: GitHub. 2020. *Danielmiessler/Seclists*. [online] Available at: https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-10000.txt [Accessed 30 April 2020].

APPENDICES

APPENDIX A – OPENPGP CERTIFICATE

This is a revocation certificate for the OpenPGP key:

```
pub   rsa2048 2020-03-31 [S]
      11221E6CD14332A558C0331490B6F1F5066EFBA3
uid          Johhny (Test Account) <johncreee@gmail.com>
```

A revocation certificate is a kind of "kill switch" to publicly
declare that a key shall not anymore be used. It is not possible
to retract such a revocation certificate once it has been published.

Use it to revoke this key in case of a compromise or loss of
the secret key. However, if the secret key is still accessible,
it is better to generate a new revocation certificate and give
a reason for the revocation. For details see the description of
of the gpg command "--generate-revocation" in the GnuPG manual.

To avoid an accidental use of this file, a colon has been inserted
before the 5 dashes below. Remove this colon with a text editor
before importing and publishing this revocation certificate.

:-----BEGIN PGP PUBLIC KEY BLOCK-----
Comment: This is a revocation certificate

iQE2BCABCAAgFiEEESIebNFDMqVYwDMUkLbx9QZu+6MFAl6DdasCHQAACgkQkLbx

9QZu+6PHzwf/VmS34ctOwp1JcABpyKdWE3P0TmwJURGqsw1ttYzT6bwtbBrlWcj1
qm6fLKrHrZTDe826kCHKrjRWwgPwqssZDBktyEP50vHiBsqBjlVhMA2/uXW2qvom
RryiBXGgaCihPTm5SKflxY6QEb/hgX3o65mKGG29sGlLswPjgmNcnLVnRnrjkVAm
Hz+9YJf5KrjNmZEJnLHqv8mlT5O6q24KpvHBpk/pX6FAuxpU43K3nqNi+NQD4H4H
x22tYQhWJRls7CUjNa5/XopTkm4MLh4++9t1iKVE2aPX9wXAI7A5rEh+PqT08DKl
J4rWuIRebebGLdL8u38WlXHh2820d0LzNA==
=jMNU
-----END PGP PUBLIC KEY BLOCK-----

Appendix B – Public Key

-----BEGIN PGP PUBLIC KEY BLOCK-----

mQENBF6DdYgBCAC0n7AC719PcazllMfLZi4U0ThJcO7kc/FpHkvPsM/Faoab7CIS
Xuv66DZGGN/Axyk92CYrlBBqwHQHS0aPSMSqJi2CwEke/C3ne2YybPGe3p7s4Fjp
AI2wDCtEtr6PoztmA8P9BTuzw/4mV15XQRkSbWaJZX84o3a0n2NXXqTLxudr5sSa
AealgXd0lcxXDINX+Rs0l4udfWYJpmDn8E3J5NVfeqPu4sqL0qWNWBQLCxolsh/L
8S02xKXyhdmzBlv8DJ4rAY7O5vHSHHT0LuZCFeNnA6bc+CriRTWS0MmaxWG6KRcq
xaLrKoa30ZHkeQZzEOg2+gCNV6/5Ui5m9PqHABEBAAG0K0pvaGhueSAoVGVzdCBB
Y2NvdW50KSA8am9obmNyZWVlQGdtYWlsLmNvbT6JAU4EEwEIADgWIQQRIh5s0UMy
pVjAMxSQtvH1Bm77owUCXoN1iAIbAwULCQgHAgYVCgkICwIEFgIDAQIeAQIXgAAK
CRCQtvH1Bm77o3bmCACenGKUng6jUIBezjSJC7ZEEyUxiFKeDKxsgLLSpQAK3wVi
U0P19R1hhY2jDs79hFRNvZYTk2+scZtk3s9VawIMC69kuf5ZFtVRuoEMz9t+cOsk
+QsSP3GWfeQ95qPqjNX124y1A0IyNh923bzgFvfQhRB19py6YOGz0648k+Gig/W1
PKJM5pcHY7ehOoglN+zDUraHYNGI+kvA1X+UeaC3TRqevwGtSFmo6eWJkvuKobzJ
96DmL+8983YTAdXgLU6gSs3L9IakIJcOiHAC7il7721kEIR937LljQz8F/ulRkBx
4a/uY4QWBE4eju1HXdNoPtG3AE/BOTiMy8+hFBIWuQENBF6DdYgBCAC7Eur2vbmG
WyuaoYN5DsukGxllXM6Ij75ItN1b5QUbHuJa7+8SMtr9fHNvyNEgxkv/a76R+ucq
zLRJZcL4SGbEnnhmNei+fgb8RTLNkpyysNtD9iejQAEzU7Dn1/fjnX4rJ3P0iZr/

MBj/42KxIX20Awgwa30jeBYDOaQSo2ixVZTT513SdJLImmVIh6J1PUJgJ9JQzF3o
0TKHAYRogUOPH6j/ra+zcjcHGGxvkDO3XMmBJuqqhK4FTbMO3I0gWLKHdDx+ivqv
6mDh0fBUxqoEhy4Dkt+pDWnOHGkDoRamAb1OWf0qPqN5U9pdpTTdibIUoNFvj0ks
Cv8pARuwSaQXABEBAAGJATYEGAEIACAWIQQRIh5s0UMypVjAMxSQtvH1Bm77owUC
XoN1iAIbDAAKCRCQtvH1Bm77o3sICACgaaDTcTE1eE2M6a3xn1F+vCCUZEZgDETo
LFeI/rm+Zhgp4mRFr7Y0p1Y6B+l9o6wNzDFUoZOx/avxojEyA7qbkCKE/6p6hQ2B
ObEFLsc0tWOoAb5k2iA5TgztMs4VCDihFtX+PQw3fAraIDZL988kp/y3w9pEnG75
KDwCHH4FeLiYdhnokFYFB7EkFC4mWZlzLW6s2QKmiXSeN9I9f1AJSXCfz2ZUqTbs
MnRtDb1+GN4V2ShbvPEbVSFZkE14vC7BVXqMzEZSqXwLUUss+nKPFxDCBlKd3Qiy
3x6CpABCYTbshRNqPHxH48d0rhEDkxgLzS7mE/RroY+kGjbg+m+/
=zuC4
-----END PGP PUBLIC KEY BLOCK-----

Appendix C – Multi-Threaded Password Cracker

Source.cpp

```
// Multi-Threaded password cracker implemented by Jonah McElfatrick
// Uses a mixture of dictionary attack and brute force methods to attempt to crack the inputted hash
// The hash algorithms compatible with this algorithm are MD5, SHA224, SHA256, SHA384 and
SHA512
#include <iostream>
#include <string>
#include <fstream>
#include <thread>
#include <mutex>
#include <vector>
#include "md5.h" // NOT MY OWN WORK, THIS FILE WAS TAKEN FROM
http://www.zedwood.com/article/cpp-md5-function
#include "sha224.h" // NOT MY OWN WORK, THIS FILE WAS TAKEN FROM
http://www.zedwood.com/article/cpp-sha224-function
#include "sha256.h" // NOT MY OWN WORK, THIS FILE WAS TAKEN FROM
http://www.zedwood.com/article/cpp-sha256-function
#include "sha384.h" // NOT MY OWN WORK, THIS FILE WAS TAKEN FROM
http://www.zedwood.com/article/cpp-sha384-function
#include "sha512.h" // NOT MY OWN WORK, THIS FILE WAS TAKEN FROM
http://www.zedwood.com/article/cpp-sha512-function

using namespace std;

// Import clock
using std::chrono::duration_cast;
using std::chrono::milliseconds;

// Define the alias "the_clock" for the clock type we're going to use.
typedef std::chrono::steady_clock the_clock;

// Mutex for when calculating the hash
mutex hash_mutex;

// Mutex for when displaying the current attempted password
mutex cout_mutex;

// Counter condition variable to count the number of attempts been carried out
condition_variable counter_cv;
mutex counter_mutex;
bool counter_bool;

// Bool to identify if the password has been found or not
bool done = false;
```

```
// String for the found password
string FoundPassword;

// Counter
int counter = 0;

// Method of found
string method;


// Used method
string hashMethod;

// Uppercase letters array
const char CapitaLetters[26] =
{
		'A', 'B', 'C', 'D', 'E', 'F', 'G',
		'H', 'I', 'J', 'K', 'L', 'M', 'N',
		'O', 'P', 'Q', 'R', 'S', 'T', 'U',
		'V', 'W', 'X', 'Y', 'Z'
};

// Lowercase letters array
const char LowerCaseLetters[26] =
{
		'a', 'b', 'c', 'd', 'e', 'f', 'g',
		'h', 'i', 'j', 'k', 'l', 'm', 'n',
		'o', 'p', 'q', 'r', 's', 't', 'u',
		'v', 'w', 'x', 'y', 'z'
};

// Symbols array
const char Symbols[22] =
{
		'!','£','$','%','^','&','*','(',')'
		,'@','~','#','|','?','¬','`','{',
		'}','[',']',';',':'
};

// Numbers array
const char Numbers[10] =
{
		'1','2','3','4','5','6','7','8','9','0'
};

// Takes each line in the word file, hashes it then compares it to the entered hash to see if they match
// If they match then the password has been found, else it continues on till the end of the file
void DictionaryAttack(string filename, string inputHash, int hashChoice, int display) {
    // Variable for the contents of the file
    string contents;

    // Opens the text file for reading
    ifstream file(filename, ios_base::binary);

    // Check to see if the file was successfully opened
    if (!file.good()) {
        cout << "Unable to open Text File";
        system("pause");
    }
    else {
        // Takes each line in the text file and hashes the contents, then compares it to the
entered hash
        while (file >> contents && !done) {
            // Notify the counter condition variable to allow the addition of another
attempt to the counter variable
            counter_bool = true;
            counter_cv.notify_one();

            // Check to see if the user wants to print out current password being
attempted
            // Mutex's used to allow for a clean output on which password is being
attempted
            if (display == 1) {
                cout_mutex.lock();
                cout << contents << endl;
                cout_mutex.unlock();
            }

            // Initialise hash variable to store the hashed attempted password
            string hash = "";

            if (hashChoice == 1 || hashChoice == 2) {
                // Hash the password using MD5
                hash_mutex.lock();
                hash = md5(contents);
                hash_mutex.unlock();

                // Checks to see if the current hashed attempted password is equal to
the inputted hash in MD5
                if (hash == inputHash) {
                    hashMethod = "MD5";
                    cout << endl << "Hash: " << hash << endl;
                    cout_mutex.lock();
                    done = true;
                    FoundPassword = contents;
                    method = "Dictionary";
                    cout_mutex.unlock();
                }
            }

            if (hashChoice == 1 || hashChoice == 3) {
                // Hash the password using sha224
                hash_mutex.lock();
                hash = sha224(contents);
                hash_mutex.unlock();

                // Checks to see if the current hashed attempted password is equal to
the inputted hash in SHA224
                if (hash == inputHash) {
                    hashMethod = "SHA-224";
                    cout << endl << "Hash: " << hash << endl;
                    cout_mutex.lock();
                    done = true;
                    FoundPassword = contents;
                    method = "Dictionary";
                    cout_mutex.unlock();
                }
            }

            if (hashChoice == 1 || hashChoice == 4) {
                // Hash the password using sha256
                hash_mutex.lock();
                hash = sha256(contents);
                hash_mutex.unlock();

                // Checks to see if the current hashed attempted password is equal to
the inputted hash in SHA256
                if (hash == inputHash) {
                    hashMethod = "SHA-256";
                    cout << endl << "Hash: " << hash << endl;
                    cout_mutex.lock();
                    done = true;
                    FoundPassword = contents;
                    method = "Dictionary";
                    cout_mutex.unlock();
                }
            }

            if (hashChoice == 1 || hashChoice == 5) {
                // Hash the password using sha384
                hash_mutex.lock();
                hash = sha384(contents);
                hash_mutex.unlock();

                // Checks to see if the current hashed attempted password is equal to
the inputted hash in SHA384
                if (hash == inputHash) {
                    hashMethod = "SHA-384";
                    cout << endl << "Hash: " << hash << endl;
                    cout_mutex.lock();
                    done = true;
                    FoundPassword = contents;
                    method = "Dictionary";
                    cout_mutex.unlock();
                }
            }

            if (hashChoice == 1 || hashChoice == 6) {
                // Hash the password using sha512
                hash_mutex.lock();
                hash = sha512(contents);
                hash_mutex.unlock();

                // Checks to see if the current hashed attempted password is equal to
the inputted hash in SHA512
                if (hash == inputHash) {
                    hashMethod = "SHA-512";
                    cout << endl << "Hash: " << hash << endl;
                    cout_mutex.lock();
                    done = true;
                    FoundPassword = contents;
                    method = "Dictionary";
                    cout_mutex.unlock();
                }
            }
        }
    }
    // Closes the input textfile
    file.close();
    return;
}

// Tries every iteration or possibility from lowercase letters, uppercase letters, numbers and symbols.
Hashes these itterations and then compares them to the entered hash value
// If the calculated hash is equal to the entered hash then the password has been found, else it
continues on till the end of the possible itterations
// For each thread running this function they are calculating it for different lengths, for example thread
```

```
one will calculate for a password string length of 1
// thread two will calculate for a password string length of 2, thread three will calculate for a password string length of 3 and so on.
void BruteForce(int stringlength, string s, string inputHash, int hashChoice, int display) {
	// Check to see if the password has been found, if so will start the return process on all iterations of the function
	if (done) return;

	string pwordattempt;

	// Checks to see if the attempted password length for the thread has been reached to then check if the attempted password is equal to the actual password
	if (stringlength == 0)
	{
		// Notify the counter condition variable to allow the addition of another attempt to the counter variable
		counter_bool = true;
		counter_cv.notify_one();

		// Check to see if the user wants to print out current password being attempted
		// Mutex's used to allow for a clean output on which password is being attempted
		if (display == 1) {
			cout_mutex.lock();
			cout << s << endl;
			cout_mutex.unlock();
		}

		// Initialise hash variable to store the hashed attempted password
		string hash = "";

		if (hashChoice == 1 || hashChoice == 2) {
			// Hash the password using MD5
			hash_mutex.lock();
			hash = md5(s);
			hash_mutex.unlock();

			// Checks to see if the current hashed attempted password is equal to the inputted hash in MD5
			if (hash == inputHash) {
				hashMethod = "MD5";
				cout << endl << "Hash: " << hash << endl;
				cout_mutex.lock();
				done = true;
				FoundPassword = s;
				method = "Brute Force";
				cout_mutex.unlock();
			}
		}

		if (hashChoice == 1 || hashChoice == 3) {
			// Hash the password using sha224
			hash_mutex.lock();
			hash = sha224(s);
			hash_mutex.unlock();

			// Checks to see if the current hashed attempted password is equal to the
inputted hash in SHA224
			if (hash == inputHash) {
				hashMethod = "SHA-224";
				cout << endl << "Hash: " << hash << endl;
				cout_mutex.lock();
				done = true;
				FoundPassword = s;
				method = "Brute Force";
				cout_mutex.unlock();
			}
		}

		if (hashChoice == 1 || hashChoice == 4) {
			// Hash the password using sha256
			hash_mutex.lock();
			hash = sha256(s);
			hash_mutex.unlock();

			// Checks to see if the current hashed attempted password is equal to the
inputted hash in SHA256
			if (hash == inputHash) {
				hashMethod = "SHA-256";
				cout << endl << "Hash: " << hash << endl;
				cout_mutex.lock();
				done = true;
				FoundPassword = s;
				method = "Brute Force";
				cout_mutex.unlock();
			}
		}

		if (hashChoice == 1 || hashChoice == 5) {
			// Hash the password using sha384
			hash_mutex.lock();
			hash = sha384(s);
			hash_mutex.unlock();

			// Checks to see if the current hashed attempted password is equal to the
```

```
inputted hash in SHA384
            if (hash == inputHash) {
                hashMethod = "SHA-384";
                cout << endl << "Hash: " << hash << endl;
                cout_mutex.lock();
                done = true;
                FoundPassword = s;
                method = "Brute Force";
                cout_mutex.unlock();
            }
        }

        if (hashChoice == 1 || hashChoice == 6) {
            // Hash the password using sha512
            hash_mutex.lock();
            hash = sha512(s);
            hash_mutex.unlock();

            // Checks to see if the current hashed attempted password is equal to the
inputted hash in SHA512
            if (hash == inputHash) {
                hashMethod = "SHA-512";
                cout << endl << "Hash: " << hash << endl;
                cout_mutex.lock();
                done = true;
                FoundPassword = s;
                method = "Brute Force";
                cout_mutex.unlock();
            }
        }
        return;
    }

    for (int i = 0; i < 26 && !done; i++) // iterate through alphabet
    {
        // Append new character onto the string
        // Recursively call function again untill string has reached its length
        // Loop for lowercase letters
        pwordattempt = s + LowerCaseLetters[i];
        BruteForce(stringlength - 1, pwordattempt, inputHash, hashChoice, display);
    }

    // Loop for capital letters
    for (int j = 0; j < 26 && !done; j++) {
        pwordattempt = s + CapitaLetters[j];
        BruteForce(stringlength - 1, pwordattempt, inputHash, hashChoice, display);
    }
```

```
		// Loop for symbols
		for (int x = 0; x < 22 && !done; x++) {
			pwordattempt = s + Symbols[x];
			BruteForce(stringlength - 1, pwordattempt, inputHash, hashChoice, display);
		}

		// Loop for numbers
		for (int y = 0; y < 10 && !done; y++) {
			pwordattempt = s + Numbers[y];
			BruteForce(stringlength - 1, pwordattempt, inputHash, hashChoice, display);
		}
}

// Simple function to check if an integer is within a certain range
int validateInt(int min, int max, int choice) {
		while (choice < min || choice > max) {
			cout << "Invalid Input" << endl;
			cout << "Please enter a value from the range above: " << endl << ">> ";
			cin >> choice;
			cout << endl;
		}
		return choice;
}

// Receive the choice of which hashing algorithm the password was hashed with
int ReceiveChoiceInput() {
		int choice;
		cout << "Would you like to attempt to crack:" << endl << "1: All Hashes supported" << endl <<
"2: Certain Hash" << endl << ">> ";
		cin >> choice;
		cout << endl;
		validateInt(1, 2, choice);
		if (choice == 2) {
			cout << "Please enter the type of hash you are wanting to crack" << endl << "2: MD5"
<< endl << "3: SHA-224" << endl << "4: SHA-256" << endl << "5: SHA-384" << endl << "6: SHA-512" <<
endl << ">> ";
			cin >> choice;
			cout << endl;
			validateInt(2, 6, choice);
		}
		return choice;
}

// Receive the hash that is trying to be cracked
string ReceiveHashInput() {
		string hash;
```

```
	cout << "Please enter the hash you would like to crack: " << endl << ">> ";
	cin >> hash;
	cout << endl;
	return hash;
}

// Receive the filename for the wordlist used in the dictionary attack
string ReceiveFilename() {
	string filename;
	cout << "Please enter the filename with the correct extension for the dictionary attack: " <<
endl << ">> ";
	cin >> filename;
	cout << endl;
	return filename;
}

// Function for receiving the number of threads that will run depending on how many characters the
user wants to try and crack the password for
int ReceiveThreads() {
	int threadNumber;
	int choice;
	// Gives the user the choice to use the maximum number of threads available by the
computer's CPU
	// Or to use a specified number of threads
	cout << "Please select from the following: " << endl << "1: Specify number of threads" << endl
<< "2: Detect Maximum threads" << endl << ">> ";
	cin >> choice;
	cout << endl;
	// Validate choice is one from the list above
	validateInt(1, 2, choice);

	if (choice == 1) {
		cout << "Please enter the number of characters you would like to try to crack: " <<
endl << ">> ";
		cin >> threadNumber;
		cout << endl;

		// Validate the input is no more than 6 characters long
		validateInt(1, 10, threadNumber);

		// Returning threadNumber + 2 as there is 2 threads needed extra, one for the
dictionary
		// attack and one to run the counter function to count the number of times a attempt
has been made to crack the password
		threadNumber += 2;
	}
	else {
		threadNumber = thread::hardware_concurrency();
	}

	return threadNumber;
}

// Function to receive the choice to display the current attempted password
int receiveDisplayChoice() {
	int choice;
	cout << "Would you like to view all attempted password? (1 for Yes or 0 for No)" << endl <<
"Please note this will increase the time taken to crack the password drastically" << endl << ">> ";
	cin >> choice;
	cout << endl;
	choice = validateInt(0, 1, choice);
	return choice;
}

int ReceiveMethodChoice() {
	int choice;
	cout << "Would you like to:" << endl << "1: Hash an input? " << endl << "2: Attempt to crack a
hash? " << endl << ">> ";
	cin >> choice;
	cout << endl;
	choice = validateInt(1, 2, choice);
	return choice;
}

// Function to record the number of attempts that have been made to crack the entered hash
void numberCounter() {
	while (!done)
	{
		unique_lock<mutex> mylock(counter_mutex);
		while (!done && counter_bool == false) {
			counter_cv.wait(mylock);
		}
		if (done)
			return;
		counter += 1;
		counter_bool = false;
	}
	return;
}

// Function to hash the users input and return the hashed value to the screen
void hashFunction() {
	// Initialise variables for selecting the hash choice, the string to be hashed and the calculated
hash
```

```
	int hashChoice;
	string stringToHash;
	string hash;

	cout << "What hash would you like to use" << endl << "1: MD5" << endl << "2: SHA-224" << 
endl << "3: SHA-256" << endl << "4: SHA-384" << endl << "5: SHA-512" << endl << ">> ";
	cin >> hashChoice;
	cout << endl;
	hashChoice = validateInt(1, 5, hashChoice);
	cout << "Please enter the string you would like to hash" << endl << ">> ";
	cin >> stringToHash;
	cout << endl;

	switch (hashChoice)
	{
	case 1: hash = md5(stringToHash); break;
	case 2: hash = sha224(stringToHash); break;
	case 3: hash = sha256(stringToHash); break;
	case 4: hash = sha384(stringToHash); break;
	case 5: hash = sha512(stringToHash); break;
	default:
		break;
	}

	cout << "String: " << stringToHash << endl << "Produced Hash: " << hash << endl;
	return;
}


int main()
{
	//Initialise variable for storing the filename of the wordlist being used
	string filename;

	// Choice of which hash is being used
	int hashChoice = 0;

	// The inputted hash that is the target goal to crack
	string inputHash = "";

	// Number of attemtps to crack the password
	int attempts;

	// Length of the string being attempted as the password
	int stringlength = 1;

	// Number of threads that are being initiated
```

```
		int threadNumber;

		// Choice to display attempted passwords or not
		int display;

		// Choice of hash or crack
		int methodChoice;

		// Receive the choice to hash an input or crack a hash
		methodChoice = ReceiveMethodChoice();

		// Check to see what method has been chosen
		if (methodChoice == 1) {
			hashFunction();
		}
		else {
			// Receive the which hashing algorithm is being used
			hashChoice = ReceiveChoiceInput();

			// Receive the hash that is being broken
			inputHash = ReceiveHashInput();

			// Receive the name of the text file that is being used in the dictionary attack
			filename = ReceiveFilename();

			// Receive the choice to display the attempted passwords or not
			display = receiveDisplayChoice();

			// Receive the number of threads to be initiated
			threadNumber = ReceiveThreads();

			cout << "Initialising password crack...";

			// Start the clock
			the_clock::time_point start = the_clock::now();

			// Initialise vector of threads
			vector<thread> tvector = {};

			// Initialise the thread to run the counter loop to count the number of attempts to 
crack the password
			tvector.push_back(thread(numberCounter));

			// Initialising threads each cracking the password for a different length
			while (stringlength < (threadNumber - 1)) {
				tvector.push_back(thread(BruteForce, stringlength, "", inputHash, 
hashChoice, display));
```

```
				stringlength++;
			}

			//Initialise thread for Dictionary attack
			tvector.push_back(thread(DictionaryAttack, filename, inputHash, hashChoice,
display));

			// Join all working threads
			for (int i = 1; i < threadNumber; i++) {
				tvector[i].join();
			}

			// Display the cracked password if found and the number of tries to crack it
			if (done) {
				cout << "Cracked Password: " << FoundPassword << endl;
				cout << "Number of attempts to crack the password: " << counter << endl;
			}
			//Display password not found
			else {
				cout << "Password not found" << endl;
			}
			the_clock::time_point end = the_clock::now();
			// End the clock

			// Compute and display the difference between the start and end times in
milliseconds
			auto time_taken = duration_cast<milliseconds>(end - start).count();
			cout << "Cracking the password took " << time_taken << " ms. " << "It was using the "
<< hashMethod << " hash method. " << "It was found using the " << method << " attack." << endl;

			// Notify and join the thread that is running the counter loop to count the number of
password attempts
			counter_cv.notify_one();
			(tvector.front()).join();
		}

		system("pause");
		return 0;
}
```

Md5.h

```
/* MD5
 converted to C++ class by Frank Thilo (thilo@unix-ag.org)
 for bzflag (http://www.bzflag.org)

   based on:

   md5.h and md5.c
   reference implementation of RFC 1321

   Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.

*/

#ifndef BZF_MD5_H
#define BZF_MD5_H
#define _CRT_SECURE_NO_WARNINGS
#include <cstring>
#include <iostream>


// a small class for calculating MD5 hashes of strings or byte arrays
// it is not meant to be fast or secure
//
// usage: 1) feed it blocks of uchars with update()
//      2) finalize()
//      3) get hexdigest() string
//      or
//      MD5(std::string).hexdigest()
```

```
//
// assumes that char is 8 bit and int is 32 bit
class MD5
{
public:
	typedef unsigned int size_type; // must be 32bit

	MD5();
	MD5(const std::string& text);
	void update(const unsigned char *buf, size_type length);
	void update(const char *buf, size_type length);
	MD5& finalize();
	std::string hexdigest() const;
	friend std::ostream& operator<<(std::ostream&, MD5 md5);

private:
	void init();
	typedef unsigned char uint1; //  8bit
	typedef unsigned int uint4;  // 32bit
	enum { blocksize = 64 }; // VC6 won't eat a const static int here

	void transform(const uint1 block[blocksize]);
	static void decode(uint4 output[], const uint1 input[], size_type len);
	static void encode(uint1 output[], const uint4 input[], size_type len);

	bool finalized;
	uint1 buffer[blocksize]; // bytes that didn't fit in last 64 byte chunk
	uint4 count[2];   // 64bit counter for number of bits (lo, hi)
	uint4 state[4];   // digest so far
	uint1 digest[16]; // the result

	// low level logic operations
	static inline uint4 F(uint4 x, uint4 y, uint4 z);
	static inline uint4 G(uint4 x, uint4 y, uint4 z);
	static inline uint4 H(uint4 x, uint4 y, uint4 z);
	static inline uint4 I(uint4 x, uint4 y, uint4 z);
	static inline uint4 rotate_left(uint4 x, int n);
	static inline void FF(uint4 &a, uint4 b, uint4 c, uint4 d, uint4 x, uint4 s, uint4 ac);
	static inline void GG(uint4 &a, uint4 b, uint4 c, uint4 d, uint4 x, uint4 s, uint4 ac);
	static inline void HH(uint4 &a, uint4 b, uint4 c, uint4 d, uint4 x, uint4 s, uint4 ac);
	static inline void II(uint4 &a, uint4 b, uint4 c, uint4 d, uint4 x, uint4 s, uint4 ac);
};

std::string md5(const std::string str);

#endif
```

Sha224.h

```
#ifndef SHA224_H
#define SHA224_H
#include <string>

class SHA224
{
protected:
        typedef unsigned char uint8;
        typedef unsigned int uint32;
        typedef unsigned long long uint64;

        const static uint32 sha256_k[];
        static const unsigned int SHA224_256_BLOCK_SIZE = (512 / 8);

public:
        void init();
        void update(const unsigned char *message, unsigned int len);
        void final(unsigned char *digest);
        static const unsigned int DIGEST_SIZE = (224 / 8);

protected:
        void transform(const unsigned char *message, unsigned int block_nb);
        unsigned int m_tot_len;
        unsigned int m_len;
        unsigned char m_block[2 * SHA224_256_BLOCK_SIZE];
        uint32 m_h[8];
};

std::string sha224(std::string input);

#define SHA2_SHFR(x, n)    (x >> n)
#define SHA2_ROTR(x, n)   ((x >> n) | (x << ((sizeof(x) << 3) - n)))
#define SHA2_ROTL(x, n)   ((x << n) | (x >> ((sizeof(x) << 3) - n)))
#define SHA2_CH(x, y, z)  ((x & y) ^ (~x & z))
#define SHA2_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
#define SHA256_F1(x) (SHA2_ROTR(x,  2) ^ SHA2_ROTR(x, 13) ^ SHA2_ROTR(x, 22))
#define SHA256_F2(x) (SHA2_ROTR(x,  6) ^ SHA2_ROTR(x, 11) ^ SHA2_ROTR(x, 25))
#define SHA256_F3(x) (SHA2_ROTR(x,  7) ^ SHA2_ROTR(x, 18) ^ SHA2_SHFR(x,  3))
#define SHA256_F4(x) (SHA2_ROTR(x, 17) ^ SHA2_ROTR(x, 19) ^ SHA2_SHFR(x, 10))
#define SHA2_UNPACK32(x, str)                 \
{                                   \
   *((str) + 3) = (uint8) ((x)      );       \
   *((str) + 2) = (uint8) ((x) >>  8);       \
   *((str) + 1) = (uint8) ((x) >> 16);       \
   *((str) + 0) = (uint8) ((x) >> 24);       \
}
#define SHA2_PACK32(str, x)                   \
{                                   \
   *(x) =   ((uint32) *((str) + 3)      )    \
         | ((uint32) *((str) + 2) <<  8)    \
         | ((uint32) *((str) + 1) << 16)    \
         | ((uint32) *((str) + 0) << 24);   \
}

#endif
```

Sha256.h

```
#ifndef SHA256_H
#define SHA256_H
#define _CRT_SECURE_NO_WARNINGS
#include <string>

class SHA256
{
protected:
	typedef unsigned char uint8;
	typedef unsigned int uint32;
	typedef unsigned long long uint64;

	const static uint32 sha256_k[];
	static const unsigned int SHA224_256_BLOCK_SIZE = (512 / 8);
public:
	void init();
	void update(const unsigned char *message, unsigned int len);
	void final(unsigned char *digest);
	static const unsigned int DIGEST_SIZE = (256 / 8);

protected:
	void transform(const unsigned char *message, unsigned int block_nb);
	unsigned int m_tot_len;
```

```
        unsigned int m_len;
        unsigned char m_block[2 * SHA224_256_BLOCK_SIZE];
        uint32 m_h[8];
};

std::string sha256(std::string input);

#define SHA2_SHFR(x, n)    (x >> n)
#define SHA2_ROTR(x, n)   ((x >> n) | (x << ((sizeof(x) << 3) - n)))
#define SHA2_ROTL(x, n)   ((x << n) | (x >> ((sizeof(x) << 3) - n)))
#define SHA2_CH(x, y, z)  ((x & y) ^ (~x & z))
#define SHA2_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
#define SHA256_F1(x) (SHA2_ROTR(x,  2) ^ SHA2_ROTR(x, 13) ^ SHA2_ROTR(x, 22))
#define SHA256_F2(x) (SHA2_ROTR(x,  6) ^ SHA2_ROTR(x, 11) ^ SHA2_ROTR(x, 25))
#define SHA256_F3(x) (SHA2_ROTR(x,  7) ^ SHA2_ROTR(x, 18) ^ SHA2_SHFR(x,  3))
#define SHA256_F4(x) (SHA2_ROTR(x, 17) ^ SHA2_ROTR(x, 19) ^ SHA2_SHFR(x, 10))
#define SHA2_UNPACK32(x, str)                 \
{                                             \
    *((str) + 3) = (uint8) ((x)      );       \
    *((str) + 2) = (uint8) ((x) >>  8);       \
    *((str) + 1) = (uint8) ((x) >> 16);       \
    *((str) + 0) = (uint8) ((x) >> 24);       \
}
#define SHA2_PACK32(str, x)                   \
{                                             \
    *(x) =   ((uint32) *((str) + 3)      )    \
           | ((uint32) *((str) + 2) <<  8)    \
           | ((uint32) *((str) + 1) << 16)    \
           | ((uint32) *((str) + 0) << 24);   \
}
#endif
```

Sha384.h

```
#ifndef SHA384_H
#define SHA384_H
#include <string>

class SHA384
{
protected:
        typedef unsigned char uint8;
        typedef unsigned int uint32;
        typedef unsigned long long uint64;

        const static uint64 sha512_k[];
        static const unsigned int SHA384_512_BLOCK_SIZE = (1024 / 8);

public:
        void init();
        void update(const unsigned char *message, unsigned int len);
        void final(unsigned char *digest);
        static const unsigned int DIGEST_SIZE = (384 / 8);

protected:
        void transform(const unsigned char *message, unsigned int block_nb);
        unsigned int m_tot_len;
        unsigned int m_len;
        unsigned char m_block[2 * SHA384_512_BLOCK_SIZE];
        uint64 m_h[8];
};


std::string sha384(std::string input);

#define SHA2_SHFR(x, n)    (x >> n)
#define SHA2_ROTR(x, n)   ((x >> n) | (x << ((sizeof(x) << 3) - n)))
#define SHA2_ROTL(x, n)   ((x << n) | (x >> ((sizeof(x) << 3) - n)))
#define SHA2_CH(x, y, z)  ((x & y) ^ (~x & z))
#define SHA2_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
#define SHA512_F1(x) (SHA2_ROTR(x, 28) ^ SHA2_ROTR(x, 34) ^ SHA2_ROTR(x, 39))
#define SHA512_F2(x) (SHA2_ROTR(x, 14) ^ SHA2_ROTR(x, 18) ^ SHA2_ROTR(x, 41))
#define SHA512_F3(x) (SHA2_ROTR(x,  1) ^ SHA2_ROTR(x,  8) ^ SHA2_SHFR(x,  7))
#define SHA512_F4(x) (SHA2_ROTR(x, 19) ^ SHA2_ROTR(x, 61) ^ SHA2_SHFR(x,  6))
#define SHA2_UNPACK32(x, str)                 \
{                                             \
    *((str) + 3) = (uint8) ((x)      );       \
    *((str) + 2) = (uint8) ((x) >>  8);       \
    *((str) + 1) = (uint8) ((x) >> 16);       \
    *((str) + 0) = (uint8) ((x) >> 24);       \
}
#define SHA2_UNPACK64(x, str)                 \
{                                             \
    *((str) + 7) = (uint8) ((x)      );       \
    *((str) + 6) = (uint8) ((x) >>  8);       \
    *((str) + 5) = (uint8) ((x) >> 16);       \
    *((str) + 4) = (uint8) ((x) >> 24);       \
    *((str) + 3) = (uint8) ((x) >> 32);       \
    *((str) + 2) = (uint8) ((x) >> 40);       \
    *((str) + 1) = (uint8) ((x) >> 48);       \
    *((str) + 0) = (uint8) ((x) >> 56);       \
}
#define SHA2_PACK64(str, x)                   \
{                                     \
    *(x) =   ((uint64) *((str) + 7)      )    \
           | ((uint64) *((str) + 6) <<  8)    \
           | ((uint64) *((str) + 5) << 16)    \
           | ((uint64) *((str) + 4) << 24)    \
           | ((uint64) *((str) + 3) << 32)    \
           | ((uint64) *((str) + 2) << 40)    \
           | ((uint64) *((str) + 1) << 48)    \
           | ((uint64) *((str) + 0) << 56);   \
}

#endif
```

Sha512.h

```
#ifndef SHA512_H
#define SHA512_H
#include <string>

class SHA512
{
protected:
        typedef unsigned char uint8;
        typedef unsigned int uint32;
        typedef unsigned long long uint64;

        const static uint64 sha512_k[];
        static const unsigned int SHA384_512_BLOCK_SIZE = (1024 / 8);

public:
        void init();
        void update(const unsigned char *message, unsigned int len);
        void final(unsigned char *digest);
        static const unsigned int DIGEST_SIZE = (512 / 8);

protected:
        void transform(const unsigned char *message, unsigned int block_nb);
        unsigned int m_tot_len;
        unsigned int m_len;
        unsigned char m_block[2 * SHA384_512_BLOCK_SIZE];
        uint64 m_h[8];
};


std::string sha512(std::string input);

#define SHA2_SHFR(x, n)    (x >> n)
#define SHA2_ROTR(x, n)   ((x >> n) | (x << ((sizeof(x) << 3) - n)))
#define SHA2_ROTL(x, n)   ((x << n) | (x >> ((sizeof(x) << 3) - n)))
#define SHA2_CH(x, y, z)  ((x & y) ^ (~x & z))
#define SHA2_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
#define SHA512_F1(x) (SHA2_ROTR(x, 28) ^ SHA2_ROTR(x, 34) ^ SHA2_ROTR(x, 39))
#define SHA512_F2(x) (SHA2_ROTR(x, 14) ^ SHA2_ROTR(x, 18) ^ SHA2_ROTR(x, 41))
#define SHA512_F3(x) (SHA2_ROTR(x,  1) ^ SHA2_ROTR(x,  8) ^ SHA2_SHFR(x,  7))
#define SHA512_F4(x) (SHA2_ROTR(x, 19) ^ SHA2_ROTR(x, 61) ^ SHA2_SHFR(x,  6))
#define SHA2_UNPACK32(x, str)                 \
{                                             \
    *((str) + 3) = (uint8) ((x)      );       \
    *((str) + 2) = (uint8) ((x) >>  8);       \
    *((str) + 1) = (uint8) ((x) >> 16);       \
    *((str) + 0) = (uint8) ((x) >> 24);       \
}
#define SHA2_UNPACK64(x, str)                 \
{                                             \
    *((str) + 7) = (uint8) ((x)      );       \
    *((str) + 6) = (uint8) ((x) >>  8);       \
    *((str) + 5) = (uint8) ((x) >> 16);       \
    *((str) + 4) = (uint8) ((x) >> 24);       \
    *((str) + 3) = (uint8) ((x) >> 32);       \
    *((str) + 2) = (uint8) ((x) >> 40);       \
    *((str) + 1) = (uint8) ((x) >> 48);       \
    *((str) + 0) = (uint8) ((x) >> 56);       \
}
#define SHA2_PACK64(str, x)                   \
{                                             \
    *(x) =   ((uint64) *((str) + 7)      )    \
           | ((uint64) *((str) + 6) <<  8)    \
           | ((uint64) *((str) + 5) << 16)    \
           | ((uint64) *((str) + 4) << 24)    \
           | ((uint64) *((str) + 3) << 32)    \
           | ((uint64) *((str) + 2) << 40)    \
           | ((uint64) *((str) + 1) << 48)    \
           | ((uint64) *((str) + 0) << 56);   \
}
```

```
#endif
```

Md5.cpp

```
/* MD5
converted to C++ class by Frank Thilo(thilo@unix-ag.org)
for bzflag(http ://www.bzflag.org)

            based on :

md5.h and md5.c
reference implemantion of RFC 1321

Copyright(C) 1991 - 2, RSA Data Security, Inc.Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc.MD5 Message - Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc.makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose.It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.

* /

/* interface header */
#include "md5.h"

/* system implementation headers */
#include <cstdio>
#define _CRT_SECURE_NO_WARNINGS
```

```
// Constants for MD5Transform routine.
#define S11 7
#define S12 12
#define S13 17
#define S14 22
#define S21 5
#define S22 9
#define S23 14
#define S24 20
#define S31 4
#define S32 11
#define S33 16
#define S34 23
#define S41 6
#define S42 10
#define S43 15
#define S44 21

///////////////////////////////////////////////

// F, G, H and I are basic MD5 functions.
inline MD5::uint4 MD5::F(uint4 x, uint4 y, uint4 z) {
	return x & y | ~x&z;
}

inline MD5::uint4 MD5::G(uint4 x, uint4 y, uint4 z) {
	return x & z | y & ~z;
}

inline MD5::uint4 MD5::H(uint4 x, uint4 y, uint4 z) {
	return x ^ y^z;
}

inline MD5::uint4 MD5::I(uint4 x, uint4 y, uint4 z) {
	return y ^ (x | ~z);
}

// rotate_left rotates x left n bits.
inline MD5::uint4 MD5::rotate_left(uint4 x, int n) {
	return (x << n) | (x >> (32 - n));
}

// FF, GG, HH, and II transformations for rounds 1, 2, 3, and 4.
// Rotation is separate from addition to prevent recomputation.
inline void MD5::FF(uint4 &a, uint4 b, uint4 c, uint4 d, uint4 x, uint4 s, uint4 ac) {
	a = rotate_left(a + F(b, c, d) + x + ac, s) + b;
}
```

```
inline void MD5::GG(uint4 &a, uint4 b, uint4 c, uint4 d, uint4 x, uint4 s, uint4 ac) {
	a = rotate_left(a + G(b, c, d) + x + ac, s) + b;
}

inline void MD5::HH(uint4 &a, uint4 b, uint4 c, uint4 d, uint4 x, uint4 s, uint4 ac) {
	a = rotate_left(a + H(b, c, d) + x + ac, s) + b;
}

inline void MD5::II(uint4 &a, uint4 b, uint4 c, uint4 d, uint4 x, uint4 s, uint4 ac) {
	a = rotate_left(a + I(b, c, d) + x + ac, s) + b;
}

//////////////////////////////////////////////

// default ctor, just initailize
MD5::MD5()
{
	init();
}

//////////////////////////////////////////////

// nifty shortcut ctor, compute MD5 for string and finalize it right away
MD5::MD5(const std::string &text)
{
	init();
	update(text.c_str(), text.length());
	finalize();
}

//////////////////////////////

void MD5::init()
{
	finalized = false;

	count[0] = 0;
	count[1] = 0;

	// load magic initialization constants.
	state[0] = 0x67452301;
	state[1] = 0xefcdab89;
	state[2] = 0x98badcfe;
	state[3] = 0x10325476;
}
```

```
/////////////////////////////

// decodes input (unsigned char) into output (uint4). Assumes len is a multiple of 4.
void MD5::decode(uint4 output[], const uint1 input[], size_type len)
{
	for (unsigned int i = 0, j = 0; j < len; i++, j += 4)
		output[i] = ((uint4)input[j]) | (((uint4)input[j + 1]) << 8) |
		(((uint4)input[j + 2]) << 16) | (((uint4)input[j + 3]) << 24);
}

/////////////////////////////

// encodes input (uint4) into output (unsigned char). Assumes len is
// a multiple of 4.
void MD5::encode(uint1 output[], const uint4 input[], size_type len)
{
	for (size_type i = 0, j = 0; j < len; i++, j += 4) {
		output[j] = input[i] & 0xff;
		output[j + 1] = (input[i] >> 8) & 0xff;
		output[j + 2] = (input[i] >> 16) & 0xff;
		output[j + 3] = (input[i] >> 24) & 0xff;
	}
}

/////////////////////////////

// apply MD5 algo on a block
void MD5::transform(const uint1 block[blocksize])
{
	uint4 a = state[0], b = state[1], c = state[2], d = state[3], x[16];
	decode(x, block, blocksize);

	/* Round 1 */
	FF(a, b, c, d, x[0], S11, 0xd76aa478); /* 1 */
	FF(d, a, b, c, x[1], S12, 0xe8c7b756); /* 2 */
	FF(c, d, a, b, x[2], S13, 0x242070db); /* 3 */
	FF(b, c, d, a, x[3], S14, 0xc1bdceee); /* 4 */
	FF(a, b, c, d, x[4], S11, 0xf57c0faf); /* 5 */
	FF(d, a, b, c, x[5], S12, 0x4787c62a); /* 6 */
	FF(c, d, a, b, x[6], S13, 0xa8304613); /* 7 */
	FF(b, c, d, a, x[7], S14, 0xfd469501); /* 8 */
	FF(a, b, c, d, x[8], S11, 0x698098d8); /* 9 */
	FF(d, a, b, c, x[9], S12, 0x8b44f7af); /* 10 */
	FF(c, d, a, b, x[10], S13, 0xffff5bb1); /* 11 */
	FF(b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */
	FF(a, b, c, d, x[12], S11, 0x6b901122); /* 13 */
	FF(d, a, b, c, x[13], S12, 0xfd987193); /* 14 */
FF(c, d, a, b, x[14], S13, 0xa679438e); /* 15 */
FF(b, c, d, a, x[15], S14, 0x49b40821); /* 16 */

/* Round 2 */
GG(a, b, c, d, x[1], S21, 0xf61e2562); /* 17 */
GG(d, a, b, c, x[6], S22, 0xc040b340); /* 18 */
GG(c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */
GG(b, c, d, a, x[0], S24, 0xe9b6c7aa); /* 20 */
GG(a, b, c, d, x[5], S21, 0xd62f105d); /* 21 */
GG(d, a, b, c, x[10], S22, 0x2441453); /* 22 */
GG(c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */
GG(b, c, d, a, x[4], S24, 0xe7d3fbc8); /* 24 */
GG(a, b, c, d, x[9], S21, 0x21e1cde6); /* 25 */
GG(d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */
GG(c, d, a, b, x[3], S23, 0xf4d50d87); /* 27 */
GG(b, c, d, a, x[8], S24, 0x455a14ed); /* 28 */
GG(a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */
GG(d, a, b, c, x[2], S22, 0xfcefa3f8); /* 30 */
GG(c, d, a, b, x[7], S23, 0x676f02d9); /* 31 */
GG(b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */

/* Round 3 */
HH(a, b, c, d, x[5], S31, 0xfffa3942); /* 33 */
HH(d, a, b, c, x[8], S32, 0x8771f681); /* 34 */
HH(c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */
HH(b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */
HH(a, b, c, d, x[1], S31, 0xa4beea44); /* 37 */
HH(d, a, b, c, x[4], S32, 0x4bdecfa9); /* 38 */
HH(c, d, a, b, x[7], S33, 0xf6bb4b60); /* 39 */
HH(b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */
HH(a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */
HH(d, a, b, c, x[0], S32, 0xeaa127fa); /* 42 */
HH(c, d, a, b, x[3], S33, 0xd4ef3085); /* 43 */
HH(b, c, d, a, x[6], S34, 0x4881d05); /* 44 */
HH(a, b, c, d, x[9], S31, 0xd9d4d039); /* 45 */
HH(d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */
HH(c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */
HH(b, c, d, a, x[2], S34, 0xc4ac5665); /* 48 */

/* Round 4 */
II(a, b, c, d, x[0], S41, 0xf4292244); /* 49 */
II(d, a, b, c, x[7], S42, 0x432aff97); /* 50 */
II(c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */
II(b, c, d, a, x[5], S44, 0xfc93a039); /* 52 */
II(a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */
II(d, a, b, c, x[3], S42, 0x8f0ccc92); /* 54 */
II(c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */
```

```
		II(b, c, d, a, x[1], S44, 0x85845dd1); /* 56 */
		II(a, b, c, d, x[8], S41, 0x6fa87e4f); /* 57 */
		II(d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */
		II(c, d, a, b, x[6], S43, 0xa3014314); /* 59 */
		II(b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */
		II(a, b, c, d, x[4], S41, 0xf7537e82); /* 61 */
		II(d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */
		II(c, d, a, b, x[2], S43, 0x2ad7d2bb); /* 63 */
		II(b, c, d, a, x[9], S44, 0xeb86d391); /* 64 */

		state[0] += a;
		state[1] += b;
		state[2] += c;
		state[3] += d;

		// Zeroize sensitive information.
		memset(x, 0, sizeof x);
}

//////////////////////////////

// MD5 block update operation. Continues an MD5 message-digest
// operation, processing another message block
void MD5::update(const unsigned char input[], size_type length)
{
		// compute number of bytes mod 64
		size_type index = count[0] / 8 % blocksize;

		// Update number of bits
		if ((count[0] += (length << 3)) < (length << 3))
				count[1]++;
		count[1] += (length >> 29);

		// number of bytes we need to fill in buffer
		size_type firstpart = 64 - index;

		size_type i;

		// transform as many times as possible.
		if (length >= firstpart)
		{
				// fill buffer first, transform
				memcpy(&buffer[index], input, firstpart);
				transform(buffer);

				// transform chunks of blocksize (64 bytes)
				for (i = firstpart; i + blocksize <= length; i += blocksize)
```

```
                        transform(&input[i]);

                index = 0;
        }
        else
                i = 0;

        // buffer remaining input
        memcpy(&buffer[index], &input[i], length - i);
}

//////////////////////////////

// for convenience provide a verson with signed char
void MD5::update(const char input[], size_type length)
{
        update((const unsigned char*)input, length);
}

//////////////////////////////

// MD5 finalization. Ends an MD5 message-digest operation, writing the
// the message digest and zeroizing the context.
MD5& MD5::finalize()
{
        static unsigned char padding[64] = {
          0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        };

        if (!finalized) {
                // Save number of bits
                unsigned char bits[8];
                encode(bits, count, 8);

                // pad out to 56 mod 64.
                size_type index = count[0] / 8 % 64;
                size_type padLen = (index < 56) ? (56 - index) : (120 - index);
                update(padding, padLen);

                // Append length (before padding)
                update(bits, 8);

                // Store state in digest
                encode(digest, state, 16);
                // Zeroize sensitive information.
                memset(buffer, 0, sizeof buffer);
                memset(count, 0, sizeof count);

                finalized = true;
        }

        return *this;
}

//////////////////////////////

// return hex representation of digest as string
std::string MD5::hexdigest() const
{
        if (!finalized)
                return "";

        char buf[33];
        for (int i = 0; i < 16; i++)
                sprintf(buf + i * 2, "%02x", digest[i]);
        buf[32] = 0;

        return std::string(buf);
}

//////////////////////////////

std::ostream& operator<<(std::ostream& out, MD5 md5)
{
        return out << md5.hexdigest();
}

//////////////////////////////

std::string md5(const std::string str)
{
        MD5 md5 = MD5(str);

        return md5.hexdigest();
}
```

Sha224.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include <cstring>
#include <fstream>
#include "sha224.h"

const unsigned int SHA224::sha256_k[64] = //UL = uint32
{ 0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
 0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
 0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
 0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2 };

void SHA224::init()
{
        m_h[0] = 0xc1059ed8;
        m_h[1] = 0x367cd507;
        m_h[2] = 0x3070dd17;
        m_h[3] = 0xf70e5939;
        m_h[4] = 0xffc00b31;
        m_h[5] = 0x68581511;
        m_h[6] = 0x64f98fa7;
        m_h[7] = 0xbefa4fa4;
        m_len = 0;
        m_tot_len = 0;
}

void SHA224::update(const unsigned char *message, unsigned int len)
{
        unsigned int block_nb;
        unsigned int new_len, rem_len, tmp_len;
        const unsigned char *shifted_message;
        tmp_len = SHA224_256_BLOCK_SIZE - m_len;
        rem_len = len < tmp_len ? len : tmp_len;
        memcpy(&m_block[m_len], message, rem_len);
        if (m_len + len < SHA224_256_BLOCK_SIZE) {
                m_len += len;
                return;
        }
        new_len = len - rem_len;
        block_nb = new_len / SHA224_256_BLOCK_SIZE;
        shifted_message = message + rem_len;
        transform(m_block, 1);
        transform(shifted_message, block_nb);
        rem_len = new_len % SHA224_256_BLOCK_SIZE;
        memcpy(m_block, &shifted_message[block_nb << 6], rem_len);
        m_len = rem_len;
        m_tot_len += (block_nb + 1) << 6;
}

void SHA224::final(unsigned char *digest)
{
        unsigned int block_nb;
        unsigned int pm_len;
        unsigned int len_b;
        int i;
        block_nb = (1 + ((SHA224_256_BLOCK_SIZE - 9)
                < (m_len % SHA224_256_BLOCK_SIZE)));
        len_b = (m_tot_len + m_len) << 3;
        pm_len = block_nb << 6;
        memset(m_block + m_len, 0, pm_len - m_len);
        m_block[m_len] = 0x80;
        SHA2_UNPACK32(len_b, m_block + pm_len - 4);
        transform(m_block, block_nb);
        for (i = 0; i < 7; i++) {
                SHA2_UNPACK32(m_h[i], &digest[i << 2]);
        }
}

void SHA224::transform(const unsigned char *message, unsigned int block_nb)
{
        uint32 w[64];
        uint32 wv[8];
        uint32 t1, t2;
        const unsigned char *sub_block;
        int i;
        int j;
        for (i = 0; i < (int)block_nb; i++) {
                sub_block = message + (i << 6);
                for (j = 0; j < 16; j++) {
                        SHA2_PACK32(&sub_block[j << 2], &w[j]);
                }
                for (j = 16; j < 64; j++) {
                        w[j] = SHA256_F4(w[j - 2]) + w[j - 7] + SHA256_F3(w[j - 15]) + w[j -
16];
                }
                for (j = 0; j < 8; j++) {
                        wv[j] = m_h[j];
                }
                for (j = 0; j < 64; j++) {
                        t1 = wv[7] + SHA256_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
                                + sha256_k[j] + w[j];
                        t2 = SHA256_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
                        wv[7] = wv[6];
                        wv[6] = wv[5];
                        wv[5] = wv[4];
                        wv[4] = wv[3] + t1;
                        wv[3] = wv[2];
                        wv[2] = wv[1];
                        wv[1] = wv[0];
                        wv[0] = t1 + t2;
                }
                for (j = 0; j < 8; j++) {
                        m_h[j] += wv[j];
                }
        }
}

std::string sha224(std::string input)
{
        unsigned char digest[SHA224::DIGEST_SIZE];
        memset(digest, 0, SHA224::DIGEST_SIZE);
        SHA224 ctx = SHA224();
        ctx.init();
        ctx.update((unsigned char*)input.c_str(), input.length());
        ctx.final(digest);

        char buf[2 * SHA224::DIGEST_SIZE + 1];
        buf[2 * SHA224::DIGEST_SIZE] = 0;
        for (int i = 0; i < SHA224::DIGEST_SIZE; i++)
                sprintf(buf + i * 2, "%02x", digest[i]);
        return std::string(buf);
}
```

Sha256.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include <cstring>
#include <fstream>
#include "sha256.h"



const unsigned int SHA256::sha256_k[64] = //UL = uint32
{ 0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
 0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
 0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
 0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2 };

void SHA256::transform(const unsigned char *message, unsigned int block_nb)
{
	uint32 w[64];
	uint32 wv[8];
	uint32 t1, t2;
	const unsigned char *sub_block;
	int i;
	int j;
	for (i = 0; i < (int)block_nb; i++) {
		sub_block = message + (i << 6);
		for (j = 0; j < 16; j++) {
			SHA2_PACK32(&sub_block[j << 2], &w[j]);
		}
		for (j = 16; j < 64; j++) {
			w[j] = SHA256_F4(w[j - 2]) + w[j - 7] + SHA256_F3(w[j - 15]) + w[j -
16];
		}
		for (j = 0; j < 8; j++) {
			wv[j] = m_h[j];
		}
		for (j = 0; j < 64; j++) {
			t1 = wv[7] + SHA256_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
```

```
                                        + sha256_k[j] + w[j];
                        t2 = SHA256_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
                        wv[7] = wv[6];
                        wv[6] = wv[5];
                        wv[5] = wv[4];
                        wv[4] = wv[3] + t1;
                        wv[3] = wv[2];
                        wv[2] = wv[1];
                        wv[1] = wv[0];
                        wv[0] = t1 + t2;
                }
                for (j = 0; j < 8; j++) {
                        m_h[j] += wv[j];
                }
        }
}

void SHA256::init()
{
        m_h[0] = 0x6a09e667;
        m_h[1] = 0xbb67ae85;
        m_h[2] = 0x3c6ef372;
        m_h[3] = 0xa54ff53a;
        m_h[4] = 0x510e527f;
        m_h[5] = 0x9b05688c;
        m_h[6] = 0x1f83d9ab;
        m_h[7] = 0x5be0cd19;
        m_len = 0;
        m_tot_len = 0;
}

void SHA256::update(const unsigned char *message, unsigned int len)
{
        unsigned int block_nb;
        unsigned int new_len, rem_len, tmp_len;
        const unsigned char *shifted_message;
        tmp_len = SHA224_256_BLOCK_SIZE - m_len;
        rem_len = len < tmp_len ? len : tmp_len;
        memcpy(&m_block[m_len], message, rem_len);
        if (m_len + len < SHA224_256_BLOCK_SIZE) {
                m_len += len;
                return;
        }
        new_len = len - rem_len;
        block_nb = new_len / SHA224_256_BLOCK_SIZE;
        shifted_message = message + rem_len;
        transform(m_block, 1);
        transform(shifted_message, block_nb);
        rem_len = new_len % SHA224_256_BLOCK_SIZE;
        memcpy(m_block, &shifted_message[block_nb << 6], rem_len);
        m_len = rem_len;
        m_tot_len += (block_nb + 1) << 6;
}

void SHA256::final(unsigned char *digest)
{
        unsigned int block_nb;
        unsigned int pm_len;
        unsigned int len_b;
        int i;
        block_nb = (1 + ((SHA224_256_BLOCK_SIZE - 9)
                < (m_len % SHA224_256_BLOCK_SIZE)));
        len_b = (m_tot_len + m_len) << 3;
        pm_len = block_nb << 6;
        memset(m_block + m_len, 0, pm_len - m_len);
        m_block[m_len] = 0x80;
        SHA2_UNPACK32(len_b, m_block + pm_len - 4);
        transform(m_block, block_nb);
        for (i = 0; i < 8; i++) {
                SHA2_UNPACK32(m_h[i], &digest[i << 2]);
        }
}

std::string sha256(std::string input)
{
        unsigned char digest[SHA256::DIGEST_SIZE];
        memset(digest, 0, SHA256::DIGEST_SIZE);

        SHA256 ctx = SHA256();
        ctx.init();
        ctx.update((unsigned char*)input.c_str(), input.length());
        ctx.final(digest);

        char buf[2 * SHA256::DIGEST_SIZE + 1];
        buf[2 * SHA256::DIGEST_SIZE] = 0;
        for (int i = 0; i < SHA256::DIGEST_SIZE; i++)
                sprintf(buf + i * 2, "%02x", digest[i]);
        return std::string(buf);
}
```

Sha384.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include <cstring>
#include <fstream>
#include "sha384.h"

const unsigned long long SHA384::sha512_k[80] = //ULL = uint64
{ 0x428a2f98d728ae22ULL, 0x7137449123ef65cdULL,
 0xb5c0fbcfec4d3b2fULL, 0xe9b5dba58189dbbcULL,
 0x3956c25bf348b538ULL, 0x59f111f1b605d019ULL,
 0x923f82a4af194f9bULL, 0xab1c5ed5da6d8118ULL,
 0xd807aa98a3030242ULL, 0x12835b0145706fbeULL,
 0x243185be4ee4b28cULL, 0x550c7dc3d5ffb4e2ULL,
 0x72be5d74f27b896fULL, 0x80deb1fe3b1696b1ULL,
 0x9bdc06a725c71235ULL, 0xc19bf174cf692694ULL,
 0xe49b69c19ef14ad2ULL, 0xefbe4786384f25e3ULL,
 0x0fc19dc68b8cd5b5ULL, 0x240ca1cc77ac9c65ULL,
 0x2de92c6f592b0275ULL, 0x4a7484aa6ea6e483ULL,
 0x5cb0a9dcbd41fbd4ULL, 0x76f988da831153b5ULL,
 0x983e5152ee66dfabULL, 0xa831c66d2db43210ULL,
 0xb00327c898fb213fULL, 0xbf597fc7beef0ee4ULL,
 0xc6e00bf33da88fc2ULL, 0xd5a79147930aa725ULL,
 0x06ca6351e003826fULL, 0x142929670a0e6e70ULL,
 0x27b70a8546d22ffcULL, 0x2e1b21385c26c926ULL,
 0x4d2c6dfc5ac42aedULL, 0x53380d139d95b3dfULL,
 0x650a73548baf63deULL, 0x766a0abb3c77b2a8ULL,
 0x81c2c92e47edaee6ULL, 0x92722c851482353bULL,
 0xa2bfe8a14cf10364ULL, 0xa81a664bbc423001ULL,
 0xc24b8b70d0f89791ULL, 0xc76c51a30654be30ULL,
 0xd192e819d6ef5218ULL, 0xd69906245565a910ULL,
 0xf40e35855771202aULL, 0x106aa07032bbd1b8ULL,
 0x19a4c116b8d2d0c8ULL, 0x1e376c085141ab53ULL,
 0x2748774cdf8eeb99ULL, 0x34b0bcb5e19b48a8ULL,
 0x391c0cb3c5c95a63ULL, 0x4ed8aa4ae3418acbULL,
 0x5b9cca4f7763e373ULL, 0x682e6ff3d6b2b8a3ULL,
 0x748f82ee5defb2fcULL, 0x78a5636f43172f60ULL,
 0x84c87814a1f0ab72ULL, 0x8cc702081a6439ecULL,
 0x90befffa23631e28ULL, 0xa4506cebde82bde9ULL,
 0xbef9a3f7b2c67915ULL, 0xc67178f2e372532bULL,
 0xca273eceea26619cULL, 0xd186b8c721c0c207ULL,
 0xeada7dd6cde0eb1eULL, 0xf57d4f7fee6ed178ULL,
 0x06f067aa72176fbaULL, 0x0a637dc5a2c898a6ULL,
 0x113f9804bef90daeULL, 0x1b710b35131c471bULL,
 0x28db77f523047d84ULL, 0x32caab7b40c72493ULL,
 0x3c9ebe0a15c9bebcULL, 0x431d67c49c100d4cULL,
 0x4cc5d4becb3e42b6ULL, 0x597f299cfc657e2aULL,
 0x5fcb6fab3ad6faecULL, 0x6c44198c4a475817ULL };

void SHA384::init()
{
	m_h[0] = 0xcbbb9d5dc1059ed8ULL;
	m_h[1] = 0x629a292a367cd507ULL;
	m_h[2] = 0x9159015a3070dd17ULL;
	m_h[3] = 0x152fecd8f70e5939ULL;
	m_h[4] = 0x67332667ffc00b31ULL;
	m_h[5] = 0x8eb44a8768581511ULL;
	m_h[6] = 0xdb0c2e0d64f98fa7ULL;
	m_h[7] = 0x47b5481dbefa4fa4ULL;
	m_len = 0;
	m_tot_len = 0;
}

void SHA384::update(const unsigned char *message, unsigned int len)
{
	unsigned int block_nb;
	unsigned int new_len, rem_len, tmp_len;
	const unsigned char *shifted_message;
	tmp_len = SHA384_512_BLOCK_SIZE - m_len;
	rem_len = len < tmp_len ? len : tmp_len;
	memcpy(&m_block[m_len], message, rem_len);
	if (m_len + len < SHA384_512_BLOCK_SIZE) {
		m_len += len;
		return;
	}
	new_len = len - rem_len;
	block_nb = new_len / SHA384_512_BLOCK_SIZE;
	shifted_message = message + rem_len;
	transform(m_block, 1);
	transform(shifted_message, block_nb);
	rem_len = new_len % SHA384_512_BLOCK_SIZE;
	memcpy(m_block, &shifted_message[block_nb << 7], rem_len);
	m_len = rem_len;
	m_tot_len += (block_nb + 1) << 7;
}

void SHA384::final(unsigned char *digest)
{
	unsigned int block_nb;
	unsigned int pm_len;
	unsigned int len_b;
	int i;
	block_nb = (1 + ((SHA384_512_BLOCK_SIZE - 17)
                    < (m_len % SHA384_512_BLOCK_SIZE)));
          len_b = (m_tot_len + m_len) << 3;
          pm_len = block_nb << 7;
          memset(m_block + m_len, 0, pm_len - m_len);
          m_block[m_len] = 0x80;
          SHA2_UNPACK32(len_b, m_block + pm_len - 4);
          transform(m_block, block_nb);
          for (i = 0; i < 6; i++) {
                    SHA2_UNPACK64(m_h[i], &digest[i << 3]);
          }
}

void SHA384::transform(const unsigned char *message, unsigned int block_nb)
{
          uint64 w[80];
          uint64 wv[8];
          uint64 t1, t2;
          const unsigned char *sub_block;
          int i, j;
          for (i = 0; i < (int)block_nb; i++) {
                    sub_block = message + (i << 7);
                    for (j = 0; j < 16; j++) {
                              SHA2_PACK64(&sub_block[j << 3], &w[j]);
                    }
                    for (j = 16; j < 80; j++) {
                              w[j] = SHA512_F4(w[j - 2]) + w[j - 7] + SHA512_F3(w[j - 15]) + w[j -
16];
                    }
                    for (j = 0; j < 8; j++) {
                              wv[j] = m_h[j];
                    }
                    for (j = 0; j < 80; j++) {
                              t1 = wv[7] + SHA512_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
                                        + sha512_k[j] + w[j];
                              t2 = SHA512_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
                              wv[7] = wv[6];
                              wv[6] = wv[5];
                              wv[5] = wv[4];
                              wv[4] = wv[3] + t1;
                              wv[3] = wv[2];
                              wv[2] = wv[1];
                              wv[1] = wv[0];
                              wv[0] = t1 + t2;
                    }
                    for (j = 0; j < 8; j++) {
                              m_h[j] += wv[j];
                    }
```

```
		}
}

std::string sha384(std::string input)
{
		unsigned char digest[SHA384::DIGEST_SIZE];
		memset(digest, 0, SHA384::DIGEST_SIZE);
		SHA384 ctx = SHA384();
		ctx.init();
		ctx.update((unsigned char*)input.c_str(), input.length());
		ctx.final(digest);

		char buf[2 * SHA384::DIGEST_SIZE + 1];
		buf[2 * SHA384::DIGEST_SIZE] = 0;
		for (int i = 0; i < SHA384::DIGEST_SIZE; i++)
			sprintf(buf + i * 2, "%02x", digest[i]);
		return std::string(buf);
}
```

Sha512.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include <cstring>
#include <fstream>
#include "sha512.h"

const unsigned long long SHA512::sha512_k[80] = //ULL = uint64
{ 0x428a2f98d728ae22ULL, 0x7137449123ef65cdULL,
 0xb5c0fbcfec4d3b2fULL, 0xe9b5dba58189dbbcULL,
 0x3956c25bf348b538ULL, 0x59f111f1b605d019ULL,
 0x923f82a4af194f9bULL, 0xab1c5ed5da6d8118ULL,
 0xd807aa98a3030242ULL, 0x12835b0145706fbeULL,
 0x243185be4ee4b28cULL, 0x550c7dc3d5ffb4e2ULL,
 0x72be5d74f27b896fULL, 0x80deb1fe3b1696b1ULL,
 0x9bdc06a725c71235ULL, 0xc19bf174cf692694ULL,
 0xe49b69c19ef14ad2ULL, 0xefbe4786384f25e3ULL,
 0x0fc19dc68b8cd5b5ULL, 0x240ca1cc77ac9c65ULL,
 0x2de92c6f592b0275ULL, 0x4a7484aa6ea6e483ULL,
 0x5cb0a9dcbd41fbd4ULL, 0x76f988da831153b5ULL,
 0x983e5152ee66dfabULL, 0xa831c66d2db43210ULL,
 0xb00327c898fb213fULL, 0xbf597fc7beef0ee4ULL,
 0xc6e00bf33da88fc2ULL, 0xd5a79147930aa725ULL,
 0x06ca6351e003826fULL, 0x142929670a0e6e70ULL,
 0x27b70a8546d22ffcULL, 0x2e1b21385c26c926ULL,
 0x4d2c6dfc5ac42aedULL, 0x53380d139d95b3dfULL,
 0x650a73548baf63deULL, 0x766a0abb3c77b2a8ULL,
 0x81c2c92e47edaee6ULL, 0x92722c851482353bULL,
 0xa2bfe8a14cf10364ULL, 0xa81a664bbc423001ULL,
 0xc24b8b70d0f89791ULL, 0xc76c51a30654be30ULL,
 0xd192e819d6ef5218ULL, 0xd69906245565a910ULL,
 0xf40e35855771202aULL, 0x106aa07032bbd1b8ULL,
 0x19a4c116b8d2d0c8ULL, 0x1e376c085141ab53ULL,
 0x2748774cdf8eeb99ULL, 0x34b0bcb5e19b48a8ULL,
 0x391c0cb3c5c95a63ULL, 0x4ed8aa4ae3418acbULL,
 0x5b9cca4f7763e373ULL, 0x682e6ff3d6b2b8a3ULL,
 0x748f82ee5defb2fcULL, 0x78a5636f43172f60ULL,
 0x84c87814a1f0ab72ULL, 0x8cc702081a6439ecULL,
 0x90befffa23631e28ULL, 0xa4506cebde82bde9ULL,
 0xbef9a3f7b2c67915ULL, 0xc67178f2e372532bULL,
 0xca273eceea26619cULL, 0xd186b8c721c0c207ULL,
 0xeada7dd6cde0eb1eULL, 0xf57d4f7fee6ed178ULL,
 0x06f067aa72176fbaULL, 0x0a637dc5a2c898a6ULL,
 0x113f9804bef90daeULL, 0x1b710b35131c471bULL,
 0x28db77f523047d84ULL, 0x32caab7b40c72493ULL,
 0x3c9ebe0a15c9bebcULL, 0x431d67c49c100d4cULL,
 0x4cc5d4becb3e42b6ULL, 0x597f299cfc657e2aULL,
 0x5fcb6fab3ad6faecULL, 0x6c44198c4a475817ULL };

void SHA512::transform(const unsigned char *message, unsigned int block_nb)
{
        uint64 w[80];
        uint64 wv[8];
        uint64 t1, t2;
        const unsigned char *sub_block;
        int i, j;
        for (i = 0; i < (int)block_nb; i++) {
                sub_block = message + (i << 7);
                for (j = 0; j < 16; j++) {
                        SHA2_PACK64(&sub_block[j << 3], &w[j]);
                }
                for (j = 16; j < 80; j++) {
                        w[j] = SHA512_F4(w[j - 2]) + w[j - 7] + SHA512_F3(w[j - 15]) + w[j -
16];
                }
                for (j = 0; j < 8; j++) {
                        wv[j] = m_h[j];
                }
                for (j = 0; j < 80; j++) {
                        t1 = wv[7] + SHA512_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
```

```
                                        + sha512_k[j] + w[j];
                                t2 = SHA512_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
                                wv[7] = wv[6];
                                wv[6] = wv[5];
                                wv[5] = wv[4];
                                wv[4] = wv[3] + t1;
                                wv[3] = wv[2];
                                wv[2] = wv[1];
                                wv[1] = wv[0];
                                wv[0] = t1 + t2;
                        }
                        for (j = 0; j < 8; j++) {
                                m_h[j] += wv[j];
                        }

                }
}

void SHA512::init()
{
                m_h[0] = 0x6a09e667f3bcc908ULL;
                m_h[1] = 0xbb67ae8584caa73bULL;
                m_h[2] = 0x3c6ef372fe94f82bULL;
                m_h[3] = 0xa54ff53a5f1d36f1ULL;
                m_h[4] = 0x510e527fade682d1ULL;
                m_h[5] = 0x9b05688c2b3e6c1fULL;
                m_h[6] = 0x1f83d9abfb41bd6bULL;
                m_h[7] = 0x5be0cd19137e2179ULL;
                m_len = 0;
                m_tot_len = 0;
}

void SHA512::update(const unsigned char *message, unsigned int len)
{
                unsigned int block_nb;
                unsigned int new_len, rem_len, tmp_len;
                const unsigned char *shifted_message;
                tmp_len = SHA384_512_BLOCK_SIZE - m_len;
                rem_len = len < tmp_len ? len : tmp_len;
                memcpy(&m_block[m_len], message, rem_len);
                if (m_len + len < SHA384_512_BLOCK_SIZE) {
                        m_len += len;
                        return;
                }
                new_len = len - rem_len;
                block_nb = new_len / SHA384_512_BLOCK_SIZE;
                shifted_message = message + rem_len;
```

```
        transform(m_block, 1);
        transform(shifted_message, block_nb);
        rem_len = new_len % SHA384_512_BLOCK_SIZE;
        memcpy(m_block, &shifted_message[block_nb << 7], rem_len);
        m_len = rem_len;
        m_tot_len += (block_nb + 1) << 7;
}

void SHA512::final(unsigned char *digest)
{
        unsigned int block_nb;
        unsigned int pm_len;
        unsigned int len_b;
        int i;
        block_nb = 1 + ((SHA384_512_BLOCK_SIZE - 17)
                < (m_len % SHA384_512_BLOCK_SIZE));
        len_b = (m_tot_len + m_len) << 3;
        pm_len = block_nb << 7;
        memset(m_block + m_len, 0, pm_len - m_len);
        m_block[m_len] = 0x80;
        SHA2_UNPACK32(len_b, m_block + pm_len - 4);
        transform(m_block, block_nb);
        for (i = 0; i < 8; i++) {
                SHA2_UNPACK64(m_h[i], &digest[i << 3]);
        }
}

std::string sha512(std::string input)
{
        unsigned char digest[SHA512::DIGEST_SIZE];
        memset(digest, 0, SHA512::DIGEST_SIZE);
        SHA512 ctx = SHA512();
        ctx.init();
        ctx.update((unsigned char*)input.c_str(), input.length());
        ctx.final(digest);

        char buf[2 * SHA512::DIGEST_SIZE + 1];
        buf[2 * SHA512::DIGEST_SIZE] = 0;
        for (int i = 0; i < SHA512::DIGEST_SIZE; i++)
                sprintf(buf + i * 2, "%02x", digest[i]);
        return std::string(buf);
}
```