UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

DEPARTMENT OF ELECTRICAL, ELECTRONIC AND COMPUTER ENGINEERING

NETWORK SECURITY (EHN 410)

PRACTICAL 1 GUIDE

Contents

1 Introduction **3**

1.1 Public-key cryptography 3

1.2 X.509 Certificates and Certification Authorities (CA's) 4

1.2.1 X.509 Certificates 4

1.2.2 Certification Authorities and certificate signing 4

1.3 Secure Sockets Layer (SSL/TLS) 5

1.4 OpenSSL Library 6

2 Generating X.509 certificates **6**

2.1 Setting up a Certification Authority (CA) 6

2.2 Creating and signing a certificate 7

2.3 Installing the CA's certificate on Firefox 7

2.3.1 Manually adding a certificate 7

2.3.2 Alternative approach 8

3 OpenSSL's BIO library **9**

3.1 Making a new connection (client) 9

3.2 Accepting a connection (server) 9

3.3 Reading and writing from a connection 10

3.4 Closing and freeing the sockets 12

4 OpenSSL's SSL library **12**

4.1 Initialising the OpenSSL library 12

4.2 Creating an SSL context 12

4.2.1 Setting up the certificates for the web server 12

4.3 Setting up and accepting a connection 13

4.4 Reading and writing on the connection 14

5 Miscellaneous **14**

5.1 Linking against libssl 14

5.2 Suggested implementation order 14

5.3 Other notes about the practical 15

6 Recommended resources **15**

1 INTRODUCTION

1.1 PUBLIC-KEY CRYPTOGRAPHY

Public key cryptography can be used for message authentication and key distribution. A public key encryption scheme uses two separate keys. An example of using public key cryptography for encryption and authentication is given below

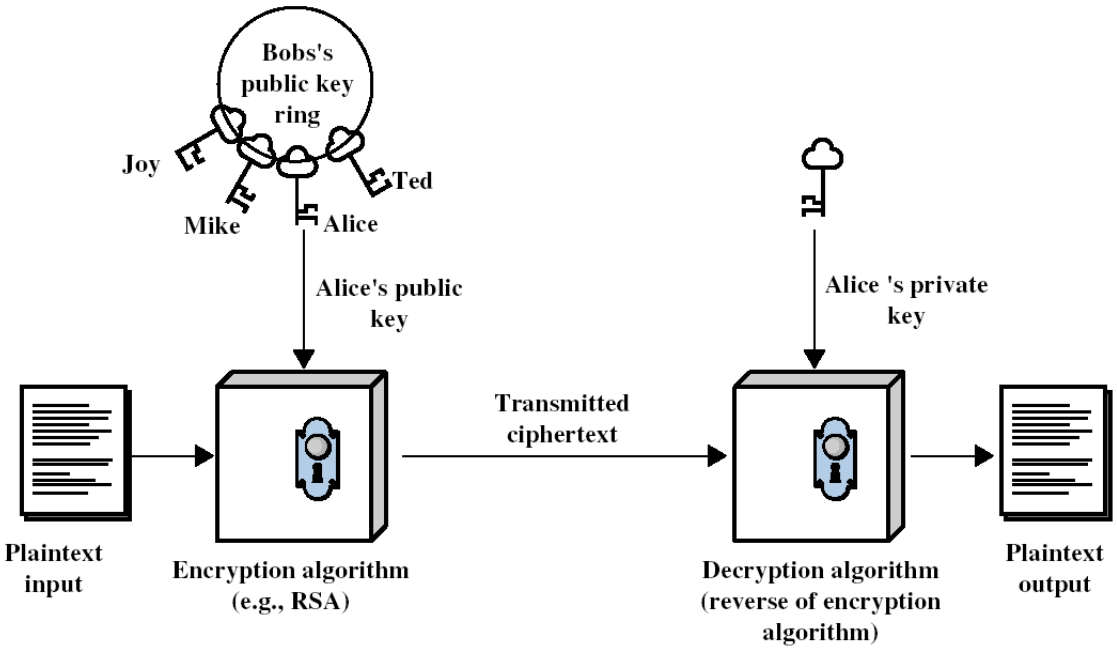

Figure 1: Encryption using public key cryptography

If one key in the public and private key pair is used for encryption, the other key must be used for decryption. In figure 1 the plaintext message is encrypted using Alice's public key. Only Alice can decrypt the message since only she knows her private key. In Fig 2 Bob encrypts a message with his private key. Alice decrypts the message with Bob's public key. Since the message is encrypted with Bob's private key, any person can decrypt the message, but only Bob can create the message. This provides authentication -Alice can be sure that Bob created the message. Public key cryptography systems can be classified in three categories:

- Encryption/decryption: The sender encrypts the message with the recipient's public key.
- Digital signature: The sender signs a message with his private key. An example is a sender that encrypts a fingerprint of the file with his private key.
- Key exchange: The sender or receiver cooperates to establish a session key. An example is the server that encrypts a session key with the receiver's public key. The session key is then used with conventional (symmetric) encryption.

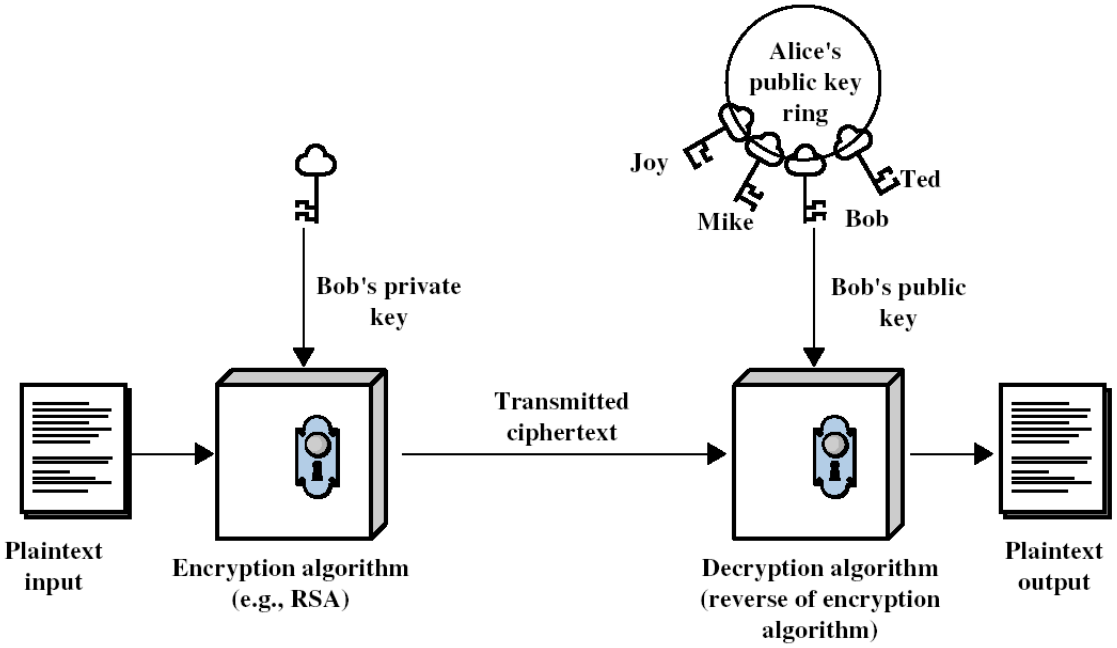

Figure 2: Authentication using public key cryptography

1.2 X.509 Certificates and Certification Authorities (CA's)

1.2.1 X.509 Certificates

ITU-T recommendation X.509 is part of the X.500 series of recommendations that defines a directory service. The X.509 specification defines a framework for the provision of authentication services to the X.500 directory users. The IETF's public key infrastructure X.509 (PKIX) working group adapted the infrastructure to the internet. The certificate format is described in RFC3280.

1.2.2 Certification Authorities and certificate signing

The Certification Authority's public key is distributed in a secure fashion to all the users. Fig. 3 shows how a CA signs a public key certificate. The unsigned certificate contains the user ID and the user's public key. The CA does a background check on the user. The CA signs the certificate by encrypting the hash code with it's private key. Any person who receives the certificate can easily verify that the CA signed the certificate. The user first computes the hash code of the certificate. Next the user decrypts the signature using the CA's public key. The decrypted signature and the hash code should match if the certificate wasn't modified.

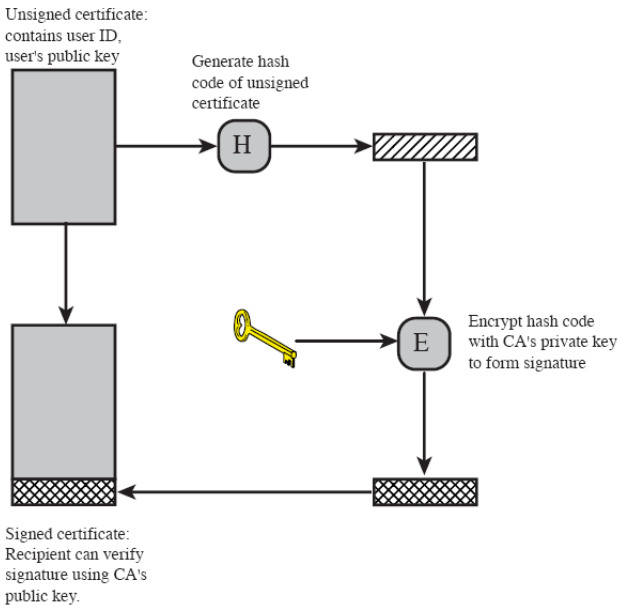

Figure 3: Public-key certificate signing

1.3 SECURE SOCKETS LAYER (SSL/TLS)

SSL provides a reliable and secure end-to-end service. SSL works on top of the TCP protocol. Some other protocols such as HTTP can be placed on top of the SSL protocol. SSL can use X.509 certificates to enable authentication between clients and servers. An example of the handshake between the SSL client and server is shown in figure 5. This practical does not require that the client (i.e. the browser) send a certificate to identify itself.

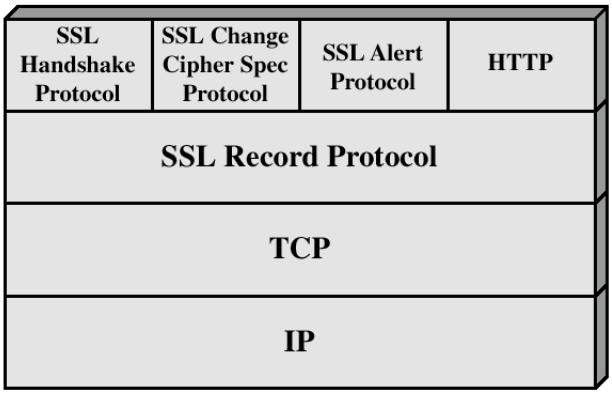

Figure 7.2 SSL Protocol Stack

Figure 4: SSL protocol stack

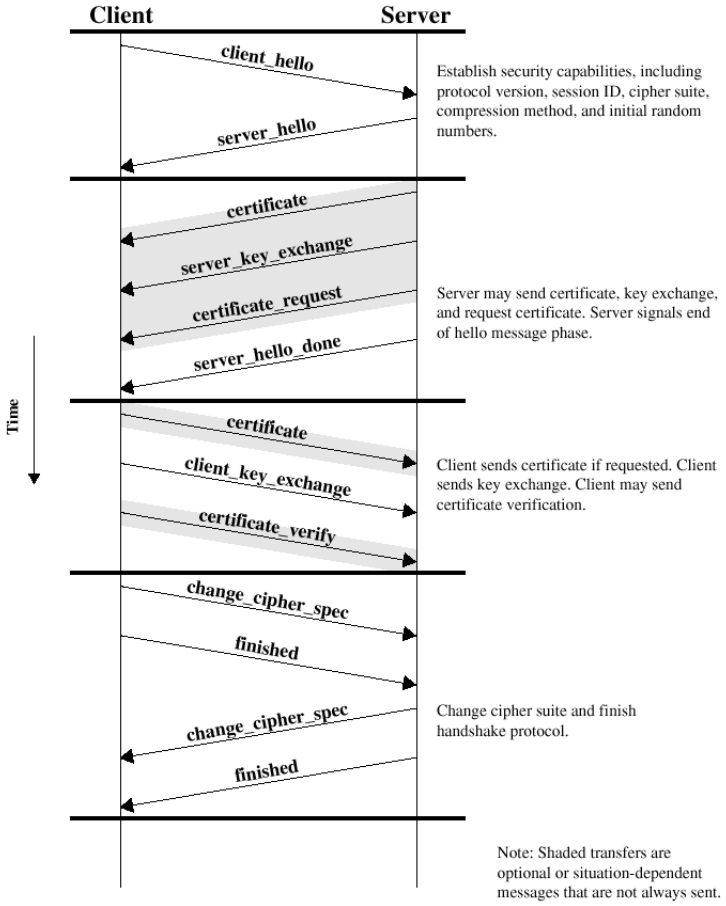

Figure 5: SSL handshaking

1.4 OpenSSL Library

OpenSSL is an open source implementation of the SSL/TLS protocols. Most Unix versions include the OpenSSL library. OpenSSL is based on SSLeay by Eric A. Young and Tim Hudson. OpenSSL also includes notoriously bad documentation.

2 Generating X.509 certificates

The following subsections provide the general procedure followed to generate X.509 certificates. Links to more information regarding these commands can be found in section 6

2.1 Setting up a Certification Authority (CA)

The first step in setting up a CA is creating a public and private key pair for the certification authority. This can be done with the following command:

```
openssl genrsa -des3 -out cert.key 1024
```

This command will create a 1024 bit RSA key pair. The key pair will be encrypted using

triple DES. This password (or the key if no password was set up) should be protected. The next step is to generate the CA's certificate. The certificate contains information about the certificate authority, the public key and a signature. The certificate can be generated with the following command:

```
openssl req -new -key cert.key -x509 -days 1095 -out cert.crt
```

You will be prompted for a password to decrypt the CA's key. The *-days* parameter specifies the number of days for which the certificate is valid. The CA's certificate needs to be distributed in a secure fashion to all the users. For example, the CA's certificate will be emailed to all the users and the fingerprint of the certificate will be confirmed over a secure channel. Once confirmed, the fingerprint of the certificate can be retrieved using:

```
openssl x509 -fingerprint -noout -in cert.crt
```

2.2 CREATING AND SIGNING A CERTIFICATE

The administrator for the web server should generate their key pair. This key does not have to be encrypted (as it will be used every time that someone requests an HTTPS web-page). The key is generated in almost exactly the same way as the CA's key:

```
openssl genrsa -des3 -out webServ.key
```

The next step is to generate a Certificate Signing Request (CSR). This request is sent to the CA for signing. A signing request is generated with the command:

```
openssl req -new -key webServ.key -out webServReq.csr
```

The final step is to have the CA created sign the web-server's CSR with its key and certificate. This can be achieved using the following command:

```
openssl x509 -req -days 365 -in webServReq.csr
     -CA <path to the CA's certificate>/cert.crt
     -CAkey <path to the CA's key>/cert.key -CAcreateserial
     -out webServCert.crt
```

2.3 INSTALLING THE CA'S CERTIFICATE ON FIREFOX

2.3.1 MANUALLY ADDING A CERTIFICATE

To manually add a certificate to a web browser, the following procedure can be followed:

- Go to the Certificate Manager in Firefox, click on Tools → Options → Advanced, click on the Encryption Tab and click on View Certificates.

- Click on the Authorities tab and click Import.
- Select your created certificate and provide the necessary permissions (usually you can just use it for http authentication).

Fig 6 shows a screen shot of where to add the certificates. Similar steps can be followed to add the certificates to other browsers.

2.3.2 Alternative approach

If you have already set up the server-side of your SSL connection, the domain address can be inserted into the browser and Firefox should automatically detect that the connection requires a certificate. It will therefore ask you to allow/disallow the certificate and set the permissions accordingly.

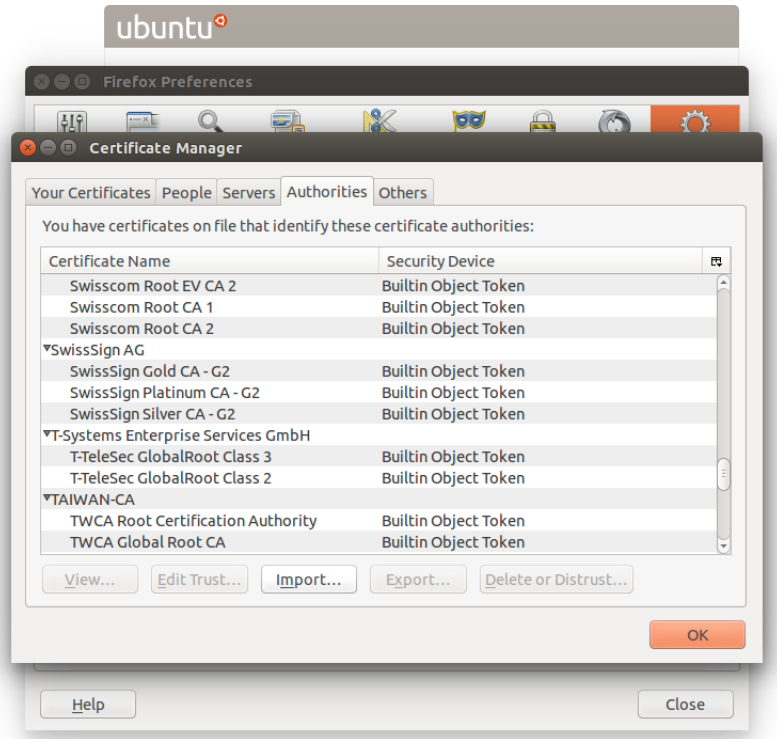

Figure 6: Installing the CA's certificate on Firefox

3 OpenSSL's BIO library

The BIO library is part of the OpenSSL library. This library is an abstraction for all the communication libraries. The library also includes implementations of all the supported platform's socket libraries (i.e. BSD sockets under Linux and Winsock under Windows). This means that the same API can be used on different platforms. Code that reads and writes to a BIO object does not have to be changed if it uses a secure or normal socket connection.

3.1 Making a new connection (client)

Making a new connection with the BIO library requires initialising the OpenSSL library and creating a new BIO object to connect to use as a connection. Once initialised, it is also prudent to check whether the connection succeeded. The following code snipped provides an example of how this may be done:

```
// init SSL library
SSL_library_init();
/* connect the BIO object
 without SSL authentication */
BIO_new_connect(ctx);
int connected = BIO_do_connect(bio);
```

3.2 Accepting a connection (server)

When accepting a connection, the first thing that needs to be done is to create the socket and bind it to a port to listen to. Creating a listening socket can be done in one call as shown below.

```
BIO *abio;
/* create a new listening BIO object (on port 80 - HTTP) */
abio = BIO_new_accept("80");
```

The first call to the `BIO_do_accept()` function sets up the socket. The second call to the function accepts a new connection.

```
/* set up the socket (first call) */
if (BIO_do_accept(abio)<=0) {
  printf("error setting up the listening socket\n");
  BIO_free(abio);
  return 0;
}
/* set the mode to blocking mode */
BIO_set_nbio_accept(abio,0);
/* wait for the incoming connection */
if (BIO_do_accept(abio)<=0) {
  printf("error accepting the socket\n");
  BIO_free(abio);
  return 0;
}
```

The `BIO_set_nbio_accept()` function sets up the listening socket as blocking or non-blocking. When the listening BIO is set to blocking, the function will not return until a connection is made. When the listening BIO is set to non-blocking, the function will exit if there are no connections to be made. The socket that is created when the connection is accepted is retrieved with the `BIO_pop()` function as shown below.

```
/* retrieve the bio for the connection */
cbio = BIO_pop(abio);
```

3.3 READING AND WRITING FROM A CONNECTION

Reading and writing to the BIO object is done with the `BIO_read()` and `BIO_write()` calls. An example of code that reads a buffer from the BIO object is given below.

```
/* read any data from the bio */
bytesread = BIO_read(cbio,buffer,sizeof(buffer));
if (bytesread == 0)
{
  printf("client closed the connection\n");
  BIO_free(cbio);
  return 0;
}
```

A simple function that reads a text file and writes it to the BIO object is shown below. This function is trivial and will not be explained.

```
/* Writes a page to the BIO object.
   returns 1 if successful,
   else returns 0.*/
int write_page(BIO *bio,const char *page)
{
  FILE *f;
  int bytesread;
  unsigned char buf[512];
  /* open the file */
  f = fopen(page,"rt");
  if (!f)
  {
    printf("could not open the page\n");
    return 0;
  }

  /* read the page from file and write
  the page to the  BIO socket */
  while (1)
  {
    /* read a number of bytes */
    bytesread = fread(buf,sizeof(unsigned char),512,f);
    /* check if it is the end of the file */
    if (bytesread == 0)
    break;
    /* write the buffer to the socket */
    if (BIO_write(bio,buf,bytesread) <= 0)
    {
      printf("write failed\n");
      break;
    }
  }
  /* close the file */
  fclose(f);
}
```

3.4 CLOSING AND FREEING THE SOCKETS

The `BIO_free()` function can be used to free the connected sockets.

4 OPENSSL'S SSL LIBRARY

4.1 INITIALISING THE OPENSSL LIBRARY

The code to initialize the SSL library is shown below.

```
SSL_load_error_strings();
SSL_library_init();
```

The `SSL_load_error_strings()` function ensures that errors returned by the SSL library are readable while the `SSL_library_init()` function registers all the available ciphers and message digests.

4.2 CREATING AN SSL CONTEXT

The SSL context (`SSL_CTX` structure) holds the default values for the SSL structures from which later connections are created. It also initializes the list of ciphers to be used. The `SSL_CTX` structure is created once per program life-time.

```
/* create the SSL context */
ctx = SSL_CTX_new(SSLv23_server_method());
if (ctx==NULL) {
  printf("failed to create the SSL context\n");
return 0;
}
```

The `SSL_CTX_new()` function creates a new context. `SSL_CTX_new()` takes a function as a parameter.

4.2.1 SETTING UP THE CERTIFICATES FOR THE WEB SERVER

The certificate of the web server should be added to the context with the `SSL_CTX_use_certificate_file()` function. The public and private key is loaded with the `SSL_CTX_use_PrivateKey_file()` function. The file-type of the keys that were generated in section 2 is `SSL_FILETYPE_PEM()`.

4.3 SETTING UP AND ACCEPTING A CONNECTION

The first step in setting up the socket to accepting connections is to create a new SSL BIO. The new BIO is created from the `SSL_CTX` as shown below.

```
bio = BIO_new_ssl(ctx,0);
if (bio == NULL) {
  printf("failed retrieving the BIO object\n");
}
```

To disable retries the following code is used:

```
BIO_get_ssl(bio,&ssl);
SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);
```

The reason why this should be done is explained in the `BIO_f_ssl(3)` man page under notes. The next step is to create a BIO that listens on the correct port. An example of this is given below:

```
/* set the listening bio to accept the connections */
abio = BIO_new_accept("443");
BIO_set_accept_bios(abio,bio);
```

The `BIO_set_accept_bios()` function chains the SSL BIO to the accept BIO. This enables the SSL encryption.

NOTE: Only the accept BIO needs to be freed. The other BIOs that are chained to the accept BIO will be automatically freed. The next step is to set up the socket. This can be done with a call to the `BIO_do_accept()` function. The second call to this function will wait for new connections. The new connections can be retrieved using the `BIO_pop()` function. This part of the setup is exactly the same as the example in the example in section 3.2.

After the connection is accepted, the SSL handshake needs to be done. An example of how a handshaking is implemented is shown below:

```
/* do the SSL handshake and check if it was successful */
if (BIO_do_handshake(connectbio) <= 0) {
  printf("handshake failed\n");
return 0;
}
```

If the client did not accept the server's certificate, the handshake will fail.

4.4 READING AND WRITING ON THE CONNECTION

Reading and writing on the SSL BIO is done exactly the same as described in section 3.3.

5 MISCELLANEOUS

5.1 LINKING AGAINST LIBSSL

The OpenSSL library is correctly installed on most distributions of Linux. Linking against the OpenSSL library can be done as shown in the excerpt below.

```
LINKFLAGS = -lssl
$(PROGNAME): $(OBJS)
gcc $(OBJS) $(LINKFLAGS) -o $(PROGNAME)
```

NOTE: In some cases the *-lssl* does not link automatically to the "libcrypto" libraries. To rectify this the `-lcrypto` flag must also be included in the linker flag.

5.2 SUGGESTED IMPLEMENTATION ORDER

The suggested implementation order of the assignment is:

1. Implement a normal web server using only the BIO functions. Test with telnet on port 80. Read the “GET /” request from the client but ignore it and write a simple page to the client.
2. Generate the certificates
3. Implement a simple client and server using the BIO and SSL library. Do not use any certificates.
4. Install the CA’s certificate in a browser and set the server to deliver a certificate. Test the web page. This is to test that your web server works properly.
5. Redesign and refactor the web server to use threads and different GET strings.
6. Implement a simple client to connect to the web server. This client should then verify the certificate of the web server and retrieve a web page and display some information of the web page.
7. Generate Doxygen html and Latex files of your code.

5.3 OTHER NOTES ABOUT THE PRACTICAL

Look into using the `chroot` function. Most versions of Linux includes both a chroot command and a `chroot` function. Using this command increases security and can even simplify the implementation.

Use a high numbered port (i.e. 4001) because you do not have permissions to open a lower numbered port. Please use original and unique names for your CA and web server's details in the certificates.

Write your program so that it takes the certificate and key name it should open as arguments from the command line. e.g. the program should be called with:

```
./mySecureWebServer <path_to_key>/webserverskey.key
     <path_to_certificate>/webserverscertificate.crt
```

or something similar. The client should be executed using:

```
./client <path_to_key>/certificate.crt <filename>
```

6 RECOMMENDED RESOURCES

1. IBM developerWorks OpenSSL tutorial: http://www- 128.ibm.com/developerworks/linux/library/l-openssl.html
2. OpenSSL documentation: http://www.openssl.org/docs/ssl/
3. Generating certificates: http://gagravarr.org/writing/openssl-certs/index.shtml
4. Doxygen coding format: http://www.stack.nl/ dimitri/doxygen/