Building A Versatile
Secure Socket Layer (SSL) Package

Advisors
Dr. Abdulkhader Al-Fantookh
Dr. Mohammed Al-affendi

Students
Hani Mohammed Al-mansour
Fahad Abdullah Al-qahtani

Graduate Project
Semester II 1419/1420
## Content

**PART I: Introduction To SSL** ........................................................................... 4

*Chapter 1: An Overview of SSL Protocol* .................................................. 5
  1.1 Introduction ......................................................................................... 6
  1.2 Goal Of SSL ..................................................................................... 7
  1.3 How SSL Works ............................................................................... 8

*Chapter 2: An Overview of Public Key Cryptography* ....................... 10
  2.1 Introduction ..................................................................................... 11
  2.2 Cryptographic Techniques Used In SSL ........................................ 11
  2.3 Public Key Cryptosystems ............................................................... 12

*Chapter 3: Cryptographic Techniques Used In SSL* ..................... 14
  3.1 Introduction ..................................................................................... 15
  3.2 ONE-WAY Functions ..................................................................... 15
  3.3 Digital Signature ........................................................................... 16
  3.4 Digital Enveloping ......................................................................... 17
  3.5 Digital Certification ....................................................................... 18
  3.6 Secret-Key Cryptography .............................................................. 19

*Chapter 4: RSA Public Key Cryptosystem* ........................................ 20
  4.1 Introduction ..................................................................................... 21
  4.2 Introduction To RSA Algorithm .................................................... 21
  4.3 Mathematical Background of RSA Cryptosystem ....................... 22
  4.4 RSA Algorithm ............................................................................ 24
  4.5 Probabilistic Primality Testing ....................................................... 25

**PART II: SSL Package Implementation** .................................................. 28
  II.1 Data Structure Representation ..................................................... 29
  II.2 SSL Package ............................................................................... 29

*Chapter 5: Secure Socket Layer Module* ............................................ 28
  5.1 Introduction ..................................................................................... 30
  5.2 Presentation Language ................................................................. 31
  5.3 Session and Connection States .................................................... 31
  5.4 Record Layer ................................................................................ 34
  5.5 Change Cipher Spec Protocol ...................................................... 39
  5.6 Alert Protocol ............................................................................... 40
  5.7 Handshake Protocol overview ................................................... 43
  5.8 Handshake Protocol .................................................................... 45
  5.9 Application Data Protocol ............................................................ 57
  5.10 Cryptography Computations ...................................................... 57
Chapter 6: Cryptography Module

6.1 Introduction ................................................................. 62
6.2 Public Key Encryption Algorithms .................................... 62
6.3 Symmetric Key encryption Algorithms .............................. 65
6.4 Message Digest Algorithms ............................................. 66
6.4 SSL Package Cipher Suites .............................................. 66

Chapter 7: X.509 Authentication Module ................................. 67
7.1 What is X.509? ............................................................... 69
7.2 X.509 Structure In ASN.1 ................................................ 70
7.3 The X.509 Object ............................................................ 73
7.4 How To use X.509 Object ................................................ 73
7.5 What is Base64? ............................................................. 74
7.6 The Base64 Object .......................................................... 76
7.7 How To use Base64 Object ................................................. 77

Chapter 8: Using The SSL Package ......................................... 78
8.1 Introduction ................................................................. 79
8.2 Creating The Package ..................................................... 79
8.3 Initializing The Package ................................................... 79
8.4 Connecting ................................................................. 80
8.5 Handshake ................................................................. 81
8.6 Exchanging Data .......................................................... 82
8.7 Closing Connection ....................................................... 82

Chapter 9: SSL Gateway Application ...................................... 83
9.1 SSL Gateway ............................................................... 84
9.2 How SSL Gateway Works .............................................. 84
9.3 How To Use SSL Gateway .............................................. 84
9.4 Security Analysis of SSL Gateway .................................. 85

Appendix A: SSL Tunneling .................................................. 86
Appendix B: SSL Presentation Language ................................. 91

References ........................................................................... 97
Introduction

Today, Internet security has become an important issue. Many companies try to incorporate commerce in the Internet to give its customers more flexibility. To incorporate commerce business in the Internet we have to make it more secure and more trusted. So Internet security have become a major research topic today. Many techniques have been developed to cover this hole in the Internet. One of the most famous and strong techniques is Secure Socket Layer (SSL) protocol, which have been developed by Netscape Corporation. It’s widely used by many Internet web browsers and Internet applications.

The SSL protocol utilizes both Public Key and Secrete Key Cryptography to achieve its goals. Therefore we divide our report into two major parts. In Part I, we are going to describe the Secure Socket Layer (SSL) and major topics in Cryptography. In Part II, we are going to describe our implementation of SSL Package. We will try to describe these two parts in simplified, descriptive, and technical manner to be easy to understand.
An Overview Of SSL Protocol
1.1: Introduction

The best candidate for application-layer security is Netscape Communications Secure Socket Layer (SSL) protocol, which is currently in its third revision [3]. SSL is a layered protocol. At each layer, messages may include fields for length, description, and content. SSL takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, and reassembled, then delivered to higher level clients.

The primary goal of the SSL Protocol is to provide privacy and reliability between two communicating applications. The protocol is composed of two layers. At the lowest level, layered on top of some reliable transport protocol (e.g., TCP) (see figure(1)), is the SSL Record Protocol. The SSL Record Protocol is used for encapsulation of various higher level protocols. One such encapsulated protocol, the SSL Handshake Protocol, allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. One advantage of SSL is that it is application protocol independent. A higher level protocol can layer on top of the SSL Protocol transparently. The SSL protocol provides connection security that has three basic properties:

- The connection is private. Encryption is used after an initial handshake to define a secret key. Symmetric cryptography is used for data encryption (e.g., DES, RC4, etc.)
- The peer's identity can be authenticated using asymmetric or public key, cryptography (e.g., RSA).
- The connection is reliable. Message transport includes a message integrity check using a keyed MAC. Secure hash functions (e.g., SHA [5], MD5 [4], etc.) are used for MAC computations.
1.2: Goals of SSL

The goals of SSL Protocol, in order of their priority, are:

1. *Cryptographic security* SSL should be used to establish a secure connection between two parties.

2. *Interoperability* Independent programmers should be able to develop applications utilizing SSL that will then be able to successfully exchange cryptographic parameters without knowledge of one another's code.

Note: It is not the case that all instances of SSL (even in the same application domain) will be able to successfully connect. For instance, if the server supports a particular encryption, and the client does not have such encryption, then the connection will not succeed.

3. *Extensibility* SSL seeks to provide a framework into which new public key and bulk encryption methods can be incorporated as necessary. This will also accomplish two sub-goals: to prevent the need to create a new protocol (and risking the introduction of possible new weaknesses) and to avoid the need to implement an entire new security library.
4. **Relative efficiency**

Cryptographic operations tend to be highly CPU intensive, particularly public key operations. For this reason, the SSL protocol has incorporated an optional session caching scheme to reduce the number of connections that need to be established from scratch. Additionally, care has been taken to reduce network activity.

### 1.3: How SSL Works

When the client connects the server and client exchange hello messages to establish the protocol, define optional encryption algorithms, exchange keys, and define optional data-compression parameters (See figure (2)). The server and client can mutually request certificates for authentication, including a complete chain of certificates leading to a certificate authority. The client generates the bulk encryption keys and sends them to the server encrypted with the server’s public key from its...
certificate. A total of four keys are used, with separate pairs for client-to-server and server-to-client communication.

Once SSL completes the initial handshake, it enters into data mode, in which application data is passed in encrypted, sequence chunks, each including a message-digest to prevent tampering. Multiple encryption algorithms including RC6 and DES are supported. Following the interaction, they perform a completion handshake and close the connection. Netscape has clearly design SSL to be generic protocol, so it can serve applications other than just HTTP, including (potentially) e-mail and database access.
2
An Overview of Public Key Cryptography
2.1: Introduction

Public Key Cryptography (also called asymmetric Cryptography) is a technique first identified by Diffie and Hellman in which encryption and decryption involve two keys. A user has two keys, the public and private key, and either can encrypt or decrypt data. The user gives his public key to other users, and keeps the private key. Data encrypted with the public key can be decrypted only with the corresponding private key, and vice versa.

Public Key Cryptography uses other techniques to make it more secure so that users on a network can share their secret information by exchanging it entirely in public.

2.2: Public Key Techniques

Public Key Cryptography uses a set of techniques that allow users on a network to share their secret information by exchanging it entirely in public. These techniques are as follow:

(i) Digital Signatures.
(ii) Digital Enveloping.
(iii) Digital Certificates.

And now we describe each one individually.

Digital Signatures

Digital Signature is a technique in which a signer, say “Ahmed”, signs a message in such a way that anyone can verify that the message was signed by no one other than Ahmed, and consequently that the message has not been modified since he signed it. The implementation of digital signature involves a message digest algorithm and public-key algorithm. (We will describe message digest algorithms and public-key algorithms in chapters 2 and 3).

Digital Enveloping

Digital Enveloping is a technique in which the user “seals” a message in such a way that no one other than the intended recipient can open the sealed message.
Digital Certificates

Digital Certificate is a technique in which a certificate authority “signs” a special message containing the name of some user and his public-key, in such a way that anyone can verify that the message was signed by no one other than the certificate authority and there by develop trust in the user public key. We will describe the algorithm in chapter 3.

2.3: Public-Key Cryptosystems

Public-Key Cryptosystem is a system that generates the public and private keys. Since public-key encryption was developed, several public-key systems have been proposed. Many of these systems rely their security on different computational problems. The most important and popular systems are as follows:

RSA Cryptosystem

This system was developed by Rivest, Shamir, and Adelman and is known as the RSA algorithm. The security of this approach is based on the fact that it can be relatively easy to multiply large primes together but almost impossible to factor the resulting product [9]. RSA has become the algorithm that most people associate with the notion of public-key cryptography. In this research we will use this algorithm as the main algorithm and cover all its aspects.

Merkle-Hellman knapsack

This system is based on the difficulty of the subset sum problem (which is NP-Complete). However, this system and all the various knapsack systems have been shown to be insecure, with exception of the Chord-Rives cryptosystem.[9]

McEliece

The McEliece cryptosystem is based on algebraic coding theory and is still regarded as being secure. Its based on the problem of decoding a linear code (which is NP-Complete).[9]

ElGamal
The ElGamal cryptosystem is based on the difficulty of the discrete logarithm problem for finite fields.[9]

**Elliptic Curve**

The Elliptic Curve Cryptosystems are modifications of other systems (such as the ElGamal Cryptosystem, for example) the work in the domain of elliptic curves rather than finite fields. The Elliptic Curve Cryptosystems appear to remain secure for smaller keys than other public-key cryptosystems. With a 160-bit modulus, an elliptic curve cryptosystem offers the same level of cryptographic security as RSA with 1024-bit module. [9]
3
Cryptographic Techniques used in SSL
3.1: Introduction

The main characteristic in Public Key Cryptography is that, it uses two keys each one can decrypt what is encrypted with the other key and the first key can’t be derived from the second and vice versa. So it requires minimal amount of information sharing among users and it allow an arbitrary user to send a key to another, specifically identified user. This can happen safely without the users having to exchange secrete keys manually. All the sender needs is the recipient’s public key and the sender can safely transmit a secret key that only the recipient can read.

Using this mechanism does not guarantee security at all, it will just make the job of an eavesdropper a little harder than before. The popular attack against public key cryptography is MAN-IN-THE-MIDDLE attack. There are many ways for success using this attack. The best method to illustrate these successful attacks is by example.

Assume Ahmed wants to talk with Ali and there is an eavesdropper Khalid that sees all the traffic that passes between Ahmed and Ali. When Ahmed requests Ali’s public key and he send it back to him, Khalid can replace Ali’s public key with his. Now, When Ahmed wants to send a message for Ali, He will encrypt it with Khalid’s public key and send it. Khalid will intercept this message, decrypt it, read it, encrypt it with Ali’s public key, and send it to Ali. In this way, Ahmed and Ali can’t know if there is an eavesdropper or not. Moreover, Khalid can change the sent message completely if he wants.

This was a small and simple example of how an attacker can success if we use public and private keys only. To make the connection between two parties more secure we have to use public key in conjunction with other methods. We will describe these methods in next sections.

3.2: ONE-WAY Functions

The notion of ONE-WAY function is central to public-key cryptography. One-way functions are fundamental building block for most of the security protocols. ONE-WAY functions are relatively easy to compute, but significantly hard to reverse. That is, given x its easy to compute f(x) but given f(x) its hard to compute x. One-way functions can not be used for encryption (no point in sending an encrypted message that no one can read).
3.2.1: ONE-WAY Hash Function

There are many other names for ONE-WAY Hash Function, it’s also called: compression functions, contraction function, message digest, fingerprint, cryptographic checksum.

Hash functions have been used in computer science for a long time. A hash function is a function that takes a variable-length input string called pre-image and converts it to fixed length output string called hash value, In such a way that its hard to find two messages with the same hash value.

Some examples of One-Way hash function algorithms are Secure Hash Algorithm (SHA) and Message Digest #5 algorithm (MD5). These two algorithms are used frequently for signing and authenticating a messages.

3.2.2: Message Authentication Codes (MAC)

A Message authentication code, or MAC, is a key-dependent one-way hash function. MACs have the same properties as the one-way hash functions discussed previously, but they also include a key. Only someone with the identical key can verify the hash. They are useful to provide authenticity without secrecy.

MACs can be used to authenticate files between users. They can also be used by a single user to determine if his files have been altered, perhaps by a virus. The user could compute the MAC of his files and store that value in a table. If the user used one-way hash function, then the virus could compute the new hash after infection and replace the table entry. A virus could not do that with a MAC, because the virus does not know the key.

An easy way to turn a one way hash function into MAC is to encrypt the hash value with a symmetric algorithm.

3.3 Digital signatures

Digital signature is a technique in which a sender, say "Ahmed," "signs" a message m in such a way that anyone can "verify" that the message was signed by no one other than Ahmed, and consequently that the message has not been modified since he signed it.
The typical implementation of digital signature involves a message-digest algorithm and a public-key algorithm for encrypting the message digest (i.e., a message-digest encryption algorithm):

- Ahmed reduces the message m to a message digest d with a message-digest algorithm; then he encrypts the message digest d with his private key, obtaining an encrypted message digest s. He sends the message m and the encrypted message digest s to Ali; the two parts together form the digitally signed message.

- Ali decrypts the encrypted message digest s with Ahmed's public key, obtaining the message digest d; then he reduces the message m to a comparative message digest d' and compares it to the message digest d. If the two are the same, he accepts the message.

Notice that Ali's work does not involve any information specific to him. Indeed, anyone can verify at any time that Ahmed signed the message, without access to any secret information. This application assumes that Ali knows Ahmed's public key; methods of developing trust in users' public keys are covered by the digital certificate application.

### 3.4 Digital enveloping

Digital enveloping is a technique in which someone, say Ahmed, "seals" a message m in such a way that no one other than the intended recipient, say "Ali," can "open" the sealed message.

The typical implementation of digital enveloping involves a secret-key algorithm for encrypting the message (i.e., a content-encryption algorithm) and a public-key algorithm for encrypting the secret key (i.e., a key-encryption algorithm):

- Ahmed encrypts the message m with a randomly generated secret key k, obtaining an encrypted message c; then he encrypts the secret key k with Ali's public key, obtaining an encrypted secret key e. He sends the encrypted message c and the encrypted secret key e to Ali; the two parts together form the digitally enveloped message.
• Ali decrypts the encrypted secret key \( e \) with his private key, obtaining the secret key \( k \); then he decrypts the encrypted message \( c \) with the secret key \( k \), obtaining the message \( m \).

Notice that Ahmed's work does not involve any information specific to his. Indeed, anyone can seal a message at any time for Ali, without access to any secret information. This application assumes that Ahmed knows Ali's public key; methods of developing trust in users' public keys are covered by the digital certificate application.

### 3.5 Digital certification

Digital certification is an technique in which a certification authority "signs" a special message \( m \) containing the name of some user, say "Ahmed," and his public key in such a way that anyone can "verify" that the message was signed by no one other than the certification authority and thereby develop trust in Ahmed's public key.

The typical implementation of digital certification involves a signature algorithm for signing the special message. (A signature algorithm is chosen here, rather than a message-digest algorithm followed by a message-digest encryption algorithm, as in the digital signature application, because X.509 certificates only use a signature algorithm.)

• Ahmed sends a "certification request" containing his name and his public key to a certification authority.

• The certification authority forms a special message \( m \) from Ahmed's request and signs the special message \( m \) under its private key, obtaining a signature \( s \). The certification authority returns the message \( m \) and the signature \( s \) to Ahmed; the two parts together form a certificate.

• Ahmed sends the certificate to Ali to convey trust in his public key.

• Ali verifies the signature \( s \) under the certification authority's public key. If the signature verifies, he accepts Ahmed's public key.

As with an ordinary digital signature, anyone can verify at any time that the certificate was signed by the certification authority, without access to any secret information.

This application assumes that Ali knows the certification authority's public key. Ali can develop trust in the certification authority's public key recursively, if he has a
certificate containing the certification authority's public key signed by a superior certification authority whom he already trusts. In this sense, a certificate is a stepping stone in digital trust.

3.6 Secret-key cryptography

Secret-key cryptography (also called symmetric cryptography) is the technology in which encryption and decryption involve the same key, a secret key. Pairs of users share a secret key, keeping the key to themselves. Data encrypted with a secret key can be decrypted only with the same secret key.

A secret-key algorithm is an algorithm for encrypting or decrypting data with a secret key. A secret key is typically used to encrypt the content of a message; in such an application, the key is called a content-encryption key and the secret-key algorithm is called a content-encryption algorithm.

An Example of secret-key cryptography is RC4, which have been developed by Ronald L. Rivest. We are going to use this algorithm as our main secret-key algorithm.

Secret-key algorithms are 1000 times faster than public-key algorithms [8]. Also, if the message that will be sent is a set of $n$ possible messages then an attacker only has to encrypt all the $n$ possible messages and compare the result with the encrypted message. So public-key cryptography is usually used to establish secure connection and agree upon a secret-key algorithm for exchanging data.
RSA Public Key Cryptosystem
4.1: Introduction

RSA algorithm was introduced in 1978 by three scientists, Rivest, Shamir, and Adleman. It has remained secure, no serious flaws have yet been found. Although the amount of analysis is no guarantee of security of a method, it does suggest a confidence level [11].

The RSA encryption algorithm incorporates result from number theory, combined with the difficulty of determining the prime factors of a number. RSA operate with the arithmetic modular n.

RSA has become the algorithm that most people associate with the notion of public-key cryptography.

In this research we choose RSA to be the main cryptosystem for generating public and private keys. Reasons for that choice lies on its popularity, simplicity, and (the most important) documentation availability. Elliptic curve cryptosystem known to be more secure than RSA cryptosystem relative to the key size, but for the lack of documentation, time and no SSL server support it until now we choose RSA to be our main cryptosystem. The RSA cryptosystem is based on problems in finite field and Elliptic Curve is based on elliptic curve over the finite field.

4.2: Introduction to RSA algorithm

In this approach a plain text block is treated as an unsigned integer. Two keys, \( d \) and \( e \) are used for decryption and encryption respectively. The plain text \( P \) is encrypted as \( (P^e \mod n) \) where, \( n \) is a product of to large prime numbers. Because the exponentiation is performed in modulus \( n \), it’s very difficult to factor \( P^e \) to uncover the encrypted plain text. However, the decryption key \( d \) is carefully chosen so that \( ((P^e)^d \mod n = P) \). Thus the legitimate receiver who knows \( d \) simply computes \( ((P^e)^d \mod n = P) \) and recover \( P \) without having to factor \( P^e \).

The underlying problem on which the encryption algorithm is based is that of factoring large numbers (100-digit and higher). The factorization is not even known to be NP-Complete; the fastest known algorithm is exponential in time.
4.3: Mathematical background of RSA algorithm

Before we introduce RSA algorithm we have to describe the mathematical theory behind it in brief. Any further description can be found in the Number theory references (look at references section).

4.3.1: Greatest common divisor and prime Numbers

The greatest common divisor of two numbers \( a \) and \( b \) is the largest integer that divides both \( a \) and \( b \). the greatest common divisor is often written \( \gcd(a,b) \). For example \( \gcd(15,10) = 5 \).

Prime number is any positive number that is divisible by itself and 1. Means for any number \( q < p \), where \( p \) is prime, \( \gcd(q, p) = 1 \).[10]

4.3.2: Euclidean algorithm

The Euclidean algorithm is a procedure for computing the GCD of two numbers. This algorithm exploits the fact that if \( x \) divides \( a \) and \( b \), then \( x \) also divides \( a - (k*b) \) for every \( k \).

The Euclidean Algorithm. Let \( r_0 = a \) and \( r_1 = b \) be nonnegative integers with \( b \neq 0 \). If the division algorithm is successively applied to obtain \( r_j = r_{j+1} q_{j+1} + r_{j+2} \) with \( 0 < r_{j+2} < r_{j+1} \) for \( j = 0,1,2,\ldots, n-2 \) and \( r_n = 0 \), then \( \gcd(a, b) = r_{n-1} \), the last nonzero remainder.[10]

To understand the algorithm it’s best to illustrate it by a simple example. So, to find the \( \gcd(75,28) \), we first divide the large number by the smaller:

\[
75 /28 = 2 + 19/28
\]

Then we repeat this process on the remainder by getting the \( \gcd(28,19) \)

\[
28/19 = 1 + 9/19
\]

We repeat this until the remainder equal 0 which is guaranteed to happen sooner or later.

\[
19/9 = 2 + 1/9
\]

\[
9/1 = 1 + 0
\]

The gcd is the last divisor, 1 in our case. Thus \( \gcd(75,28) = 1 \), which is the correct answer.

4.3.3: Modular Arithmetic

Here we will only mention some of useful modular arithmetic theorems:
Theorem 1: If \( a \) and \( b \) and \( n \) are integers then \( a \mod n = b \) if \( a = c \cdot n + b \) for some integer \( c \) \([10]\).

Theorem 2: Two integers \( a \) and \( b \) said to be relatively prime to each other if and only if \( \gcd(a, b) = 1 \) \([10]\).

Theorem 3: Two different integers can have the same result under modulus \( n \). we denote this by:

\[ X \equiv Y \pmod{n} \]

Theorem 4: the inverse of integer \( a \) in modulus \( n \) is \( b \) if and only if \( ((a \cdot b) \mod n = 1) \). We denote the inverse \( b \) as \( a^{-1} \). \([10]\)

Theorem 5: let \( n \) be a positive integer. The Euler phi-function \( \phi(n) \) is defined to be the number of positive integers not exceeding \( n \) which are relatively prime to \( n \). \([10]\)

**4.3.4: Computing the inverse in modular arithmetic**

There are many methods for computing the inverse in modular arithmetic. One of these methods is an extension to Euclidean algorithm called extended Euclidean algorithm.

The Extended Euclidean algorithm uses the results of the quotients in the divisions of the Euclidean algorithm to find integers \( x, y \) with \( ax + by = \gcd(a, b) \). Thus if \( \gcd(a, b) = 1 \), then \( x \pmod{b} \) is \( a^{-1} \pmod{b} \).

To find \( x \) we define a sequence of numbers \( X_0, X_1, \ldots, X_{n-1} \), according to the following recurrence:

\[
X_0 = 0 \\
X_1 = 1 \\
\ldots \ldots \\
X_j = X_{j-2} - Q_{j-1} \cdot X_{j-1}
\]

Where \( n \) is the number of quotients in the division of Euclidean algorithm. And \( Q_j \) is the \( j \)th quotient in the divisions of Euclidean algorithm. To make this clear we will illustrate it with an example.

**Example:** find \( 28^{-1} \pmod{75} \).

**Solution:** from previous example of Euclidean algorithm we find that:

\[
Q_1 = 2, \quad Q_2 = 1, \quad Q_3 = 2, \quad n = 5
\]
Now we apply the rule to find $28^{-1} \pmod{75}$

- $X_0 = 0$
- $X_1 = 1$
- $X_2 = 0 - 2 \times 1 = -2$
- $X_3 = 1 - 1 \times -2 = 3$
- $X_4 = -2 - 2 \times 3 = -8$

We find that $28^{-1} \pmod{75} = -8 \pmod{75} = 67 \pmod{75}$

$\Rightarrow 28^{-1} \pmod{75} = 67 \pmod{75}$

To prove this we compute

$$(28 \times 67) \pmod{75} = 1876 \pmod{75} = 1$$

$\Rightarrow 28 \times 67 \equiv 1 \pmod{75}$

$\Rightarrow 28^{-1} = 67 \pmod{75}$ and $67^{-1} = 28 \pmod{75}$

With this simple algorithm we solve the problem of finding the inverse of integer, $a$, in modulus $b$. One thing must be noted here, that there is inverse for integer $a$ in modulus $b$ if and only if $(a,b) = 1$. That means if $(a,b) \neq 1$ then there is no inverse for the integer $a$ in modulus $b$. [7]

### 4.4: The RSA Algorithm

We can now describe the RSA cryptosystem in more detail. RSA cryptosystem uses computation in modulus $n$, where $n$ is multiple of two large prime numbers.

To generate two keys, choose two random large prime numbers, $p$ and $q$ (refer to section 2.5 on how to choose large prime number). For maximum security choose $p$ and $q$ of the same length and the length is more than 100-digit. Compute the product:

$$n = pq$$

Then choose encryption key, $e$, such that $e$ and $\phi(n)$ are relatively prime. Finally, using the extended Euclidean algorithm compute $d$ such that:

$$d = e^{-1} \pmod{\phi(n)}$$
At this stage we have produced two keys. The numbers e and n are the public key, and the number d is the private key. The two primes p and q are no longer needed.

To encrypt a message \( m \), first divide it into numerical blocks smaller than \( n \) (\( m_1, \ldots, m_i \)). To encrypt a message block \( m_j \) and produce the cipher message \( c_j \) we follow this simple formula:

\[ c_j = m_j^e \mod n \]

To decrypt a message, take each encrypted block \( c_j \) and compute:

\[ m_j = c_j \mod n \]

Proving that this formula will produce the original message is as follow:

\[ c_j^d = (m_j^e)^d = m_j^{ed} \]

\[ \therefore ed \equiv 1 \pmod{\phi(n)} \]

\[ \Rightarrow ed = k \ast (\phi(n)) + 1 \]

\[ \Rightarrow m = m_j^{k(\phi(n)) + 1} = m_j^{k(\phi(n))} + 1 = m_j^{k(\phi(n))} \]

in modulus n we get

\[ m_j^{k(\phi(n))} \mod n = [(m_j \mod n) \cdot (m_j^{k(\phi(n))} \mod n)] \mod n \]

\[ \therefore m_j \mod n = m_j \]

We assuming that \( m \) is relatively prime to \( n \) because The probability that \( m \) and \( n \) are not relatively prime is extremely small, Its about \( 10^{-99} \) when \( p \) and \( q \) are both larger than \( 10^{100} \) [10].

\[ \therefore m_j^{k(\phi(n))} \mod n = 1 \]

\[ \Rightarrow [(m_j \mod n) \cdot (m_j^{k(\phi(n))} \mod n)] \mod n \]

\[ = m_j \ast 1 \mod n = m_j \]

\[ \Rightarrow \text{we get the original message.}[9,10] \]

4.5: Probabilistic Primality Testing

In setting up the RSA Cryptosystem, its necessary to generate large primes (e.g. 100 digit). In practice, the way this is done is to generate large random numbers, and then test them for primality using a probabilistic polynomial-time Monte Carlo algorithm such as the Solovay-Strassen or Miller-Rabin algorithm. These algorithms
are fast (i.e. an integer $n$ can be tested in time that is polynomial in $\log_2 n$, the number of bits in the binary representation of $n$), but there is a possibility that the algorithm may claim that $n$ is prime when it’s not. However, by running the algorithm enough times, the error probability can be reduced below any desired threshold.

There are also other methods that are called True Primality Tests. In these methods positive integers can be proven to be prime, and also these methods called primality proving algorithms. These primality tests are generally more computationally intensive than the probabilistic primality tests such as Miller-Rabin algorithm. Consequently, before applying one of these tests to a candidate prime $n$, the candidate should be subjected to a probabilistic primality test.

In this research we will use Miller-Rabin algorithm because it’s better than Solovay-Strassen in computational expensivity, implementation simplicity, and probabilistic error.

4.5.1: Miller-Rabin Test

The probabilistic primality test used most in practice is the Miller-Rabin test, also known as the strong pseudoprime test. Here we aren’t going to describe the theory behind it; we will just describe the algorithm of generating a prime number.

To generate a prime number, first choose a random number, $p$, to test. Calculate $b$, where $b$ is the number of times 2 divides $p-1$ (i.e. $2^b$ is the largest power of 2 that divides $p-1$). Then calculate $m$, such that $p = 1 + 2^b m$. After that set the security parameter, $t$, where $t$ is integer greater than or equal 1 [8]. The security parameter decreases the probability of compositeness of $p$ whenever it increases. Then follow the following procedure:

for $I$ from 1 to $t$ do the following
begin
choose a random number, $a$, such that $a$ is less than $p$.
compute $y = a^m \mod p$.
if $y \neq 1$ and $y \neq p-1$ then do the following
begin
$j = 1$
while $j < b-1$ and $y \neq n-1$ do the following
begin
compute $y = y^2 \mod n$.
if $y = 1$ then return (p is composite)
    $j = j + 1$
end
if $y \neq n - 1$ then return (p is composite)
end
end
return (p is prime)

◆
SSL package Implementation
II.1 Data structure Representation:

In the following chapters the representation language will include ASN.1 notation, XDR notation, and pseudo-Pascal for code implemented in Delphi™. We try to standardize our notation by using Delphi language but due to limitation in Pascal language, some structures can't be represented in Pascal language. So we use the standard notation for each module. However, we add an appendixes at the end of this report that describe each notation in a simple way.

II.2 SSL Package:

The package is divided into three modules:

1- SSL module: which encapsulates the main SSL protocol and provide suitable interfaces for overlying applications.

2- Cryptographic module: which is a library that includes all the needed cryptography algorithms that will be used by the SSL module.

3- X.509 Authentication module: Which is a standard protocol for authentication developed by ISO. The SSL module uses it to authenticate the server and the client (if needed).

We designed the cryptography and X.509 modules such that they are independent from the SSL package, which means that they can be used in other applications without using the SSL module.
5
Secure Socket Layer Module
5.1 Introduction

The SSL module implement the client side of the SSL protocol. The server side has been taken into account while developing this package so this package can be easily upgraded to implement both side.

The SSL Protocol Composed of two layers (see figure 3). At the lowest level (layered on top of some reliable transport protocol (e.g., TCP)) is the SSL Record Protocol. The SSL Record Protocol is used for encapsulation of various higher level protocols. One such encapsulated protocol, the SSL Handshake Protocol, allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. One advantage of SSL is that it is application protocol independent. A higher level protocol can layer on top of the SSL Protocol transparently.

![SSL Protocol Layers](image)

5.2 Presentation language

In This Specification we are going to deal with a structures that can't be specify in Pascal Language. So In Describing the SSL protocol we will adopt the presentation Language used in the original SSL document [3]. The language is very basic, it uses the programming language "C" in its syntax and XDR [16] in both its syntax and intent (see appendix B for Language Specification).

5.3 Session and connection states

An SSL session is stateful protocol. But this does not mean that the applications built on top of SSL layer must be stateful. HTTP for example, is stateless. It remains
statless when used over SSL. It is the responsibility of the SSL Handshake protocol to coordinate the states of the client and server, thereby allowing the protocol state machines of each to operate consistently, despite the fact that the state is not exactly parallel. Logically the state is represented twice, once as the current operating state, and (during the handshake protocol) again as the pending state. Additionally, separate read and write states are maintained. When the client or server receives a change cipher spec message, it copies the pending read state into the current read state. When the client or server sends a change cipher spec message, it copies the pending write state into the current write state. When the handshake negotiation is complete, the client and server exchange change cipher spec messages (see Section 5.5), and they then communicate using the newly agreed-upon cipher spec.

An SSL session may include multiple secure connections; in addition, parties may have multiple simultaneous sessions.

The session state includes the following elements:

<table>
<thead>
<tr>
<th>Session identifier</th>
<th>An arbitrary byte sequence chosen by the server to identify an active or resumable session state.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer certificate</td>
<td>X509.v3[14] certificate of the peer. This element of the state may be null.</td>
</tr>
<tr>
<td>Compression method</td>
<td>The algorithm used to compress data prior to encryption.</td>
</tr>
<tr>
<td>Cipher spec</td>
<td>Specifies the bulk data encryption algorithm (such as null, DES, etc.) and a MAC algorithm (such as MD5 or SHA). It also defines cryptographic attributes such as the hash_size. (See definition below)</td>
</tr>
<tr>
<td>Master secret</td>
<td>48-byte secret shared between the client and server.</td>
</tr>
<tr>
<td>is resumable</td>
<td>A flag indicating whether the session can be used to initiate new connections.</td>
</tr>
</tbody>
</table>

The CipherSpec is part of the Session State. References to fields of the CipherSpec are made throughout this document using presentation syntax. CipherSpec structure is as follow:

```c
enum { stream, block } CipherType;
```
enum { true, false } IsExportable;

enum { null, rc4, rc2, des, 3des, des40, fortezza } BulkCipherAlgorithm;

enum { null, md5, sha } MACAlgorithm;

struct {
    BulkCipherAlgorithm bulk_cipher_algorithm;
    MACAlgorithm mac_algorithm;
    CipherType cipher_type;
    IsExportable is_exportable
    uint8 hash_size;
    uint8 key_material;
    uint8 IV_size;
} CipherSpec;

The connection state includes the following elements:

<table>
<thead>
<tr>
<th>Server and client random</th>
<th>Byte sequences that are chosen by the server and client for each connection.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server write MAC secret</td>
<td>The secret used in MAC operations on data written by the server.</td>
</tr>
<tr>
<td>Client write MAC secret</td>
<td>The secret used in MAC operations on data written by the client.</td>
</tr>
<tr>
<td>Server write key</td>
<td>The cipher key for data encrypted by the server and decrypted by the client.</td>
</tr>
<tr>
<td>Client write key</td>
<td>The cipher key for data encrypted by the client and decrypted by the server.</td>
</tr>
<tr>
<td>Initialization vectors</td>
<td>When a block cipher in CBC mode is used, an initialization vector (IV) is maintained for each key. This field is first initialized by the SSL handshake protocol. Thereafter the final ciphertext block from each record is preserved for use with the following record.</td>
</tr>
<tr>
<td>Sequence numbers</td>
<td>Each party maintains separate sequence numbers for transmitted and received messages for each connection. When a party sends or receives a change cipher spec</td>
</tr>
</tbody>
</table>
message, the appropriate sequence number is set to zero. Sequence numbers are of type uint64 and may not exceed $2^{64}-1$.

### 5.4 Record layer

The SSL Record Layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size.

#### 5.4.1 Fragmentation

The record layer fragments information blocks into SSLPlaintext records of $2^{14}$ bytes or less. Client message boundaries are not preserved in the record layer (i.e., multiple client messages of the same ContentType may be coalesced into a single SSLPlaintext record). The structure of SSLPlaintext is as follow:

```c
struct {
    uint8 major, minor;
} ProtocolVersion;

enum {
    change_cipher_spec(20), alert(21), handshake(22),
    application_data(23), (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[SSLPlaintext.length];
} SSLPlaintext;
```

<table>
<thead>
<tr>
<th>Type</th>
<th>The higher level protocol used to process the enclosed fragment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>The version of protocol being employed.</td>
</tr>
<tr>
<td>Length</td>
<td>The length (in bytes) of the following SSLPlaintext.fragment.</td>
</tr>
<tr>
<td>Fragment</td>
<td>The application data. This data is transparent and treated as an independent block to be dealt with by the higher level protocol specified by the type field.</td>
</tr>
</tbody>
</table>

Note: Data of different SSL Record layer content types may be interleaved. Application data is generally of lower precedence for transmission than other content types.

The Algorithm used in our implementation to analyze SSLPlaintext structure is as follow:

```plaintext
Algorithm AnalyseSSLPlaintext( PlainText : StreamOfBytes);
begin
  Get PlainText Header;
  if Version < 3 then Raise an error and close the connection;
  else
    begin
      Msg_Body := Get PlainText Fragment;
      case Type of
        Handshake: (AnalyseHandShake(Msg_Body));
        Change_Cipher_Spec: (AnalyseChangeCipherSpec(Msg_Body));
        Application_Data: (AnalyseApplicationData(Msg_Body));
        Alert: (AnalyseAlert(Msg_Body));
      end
    end;
end;
```

5.4.2 Record compression and decompression

All records are compressed using the compression algorithm defined in the current session state. There is always an active compression algorithm, however initially it is defined as CompressionMethod.null. The compression algorithm translates an SSLPlaintext structure into an SSLCompressed structure. Compression functions erase their state information whenever the CipherSpec is replaced.

Compression must be lossless and may not increase the content length by more than 1024 bytes. If the decompression function encounters an SSLCompressed.fragment that would decompress to a length in excess of $2^{14}$ bytes, it should issue a fatal decompression_failure alert (Section 5.7.2). The structure of SSLCompressed is as follow:
struct {
    ContentType type; /* same as SSLPlaintext.type */
    ProtocolVersion version; /* same as SSLPlaintext.version */
    uint16 length;
    opaque fragment[SSLCompressed.length];
} SSLCompressed;

<table>
<thead>
<tr>
<th>Length</th>
<th>The length (in bytes) of the following SSLCompressed.fragment. The length should not exceed $2^{14} + 1024$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fragment</td>
<td>The compressed form of SSLPlaintext.fragment.</td>
</tr>
</tbody>
</table>

*Note:* A CompressionMethod.null operation is an identity operation; no fields are altered.

The Algorithm used in our implementation to analyze SSLCompressed structure is as follow:

```plaintext
Algorithm AnalyseSSLCompressed( Compressed : StreamOfBytes);
begin
    Get Compressed Header
    Buffer := Get Compressed Fragment;
    DataBuffer := Decompress fragment;
    If length(DataBuffer) > $2^{14}+1024$ then send alert and close the connection;
    else
        begin
            Update the fragment size in the header info;
            Append Data Buffer to fragment info;
            AnalyseSSLPlainText(fragment);
        end;
    end;
end;
```

### 5.4.3 Record payload protection and the CipherSpec

All records are protected using the encryption and MAC algorithms defined in the current CipherSpec. There is always an active CipherSpec, however initially it is SSL_NULL_WITH_NULL_NULL, which does not provide any security.

Once the handshake is complete, the two parties have shared secrets that are used to encrypt records and compute keyed message authentication codes (MACs) on their contents. The techniques used to perform the encryption and MAC operations
are defined by the CipherSpec and constrained by CipherSpec.cipher_type. The encryption and MAC functions translate an SSLCompressed structure into an SSLCiphertext. The decryption functions reverse the process. Transmissions also include a sequence number so that missing, altered, or extra messages are detectable.

The structure of SSLCiphertext is as follow:

```c
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    select (CipherSpec.cipher_type) {
        case stream: GenericStreamCipher;
        case block: GenericBlockCipher;
    } fragment;
} SSLCiphertext;
```

- **Type**: The type field is identical to SSLCompressed.type.
- **Version**: The version field is identical to SSLCompressed.version.
- **Length**: The length (in bytes) of the following SSLCiphertext.fragment. The length may not exceed $2^{14} + 2048$.
- **Fragment**: The encrypted form of SSLCompressed.fragment, including the MAC.

The Algorithm used in our implementation to analyze SSLCompressed structure is as follow:

```c
Algorithm AnalyseSSLCipherText( CipherText : StreamOfMemory);
begin
  Get CipherText Header
  if Version < 3 then raise and error and close the connection;
  else begin
    Buffer := Decipher the coming Data;
    if Cipher algorithm is Block cipher then
      begin
        Update Initialization Vector (IV);
        Remove Block Padding from the data;
      end;
      Verify the hashing value;
      if the hashing is equal then
        begin
          if Cipher Enabled then Increment Reading Message Sequence;
          AnalyseSSLCompressed(Buffer);
        end;
      else
        send alert and close the connection;
    end;
  end;
end;
```
5.4.3.1 Null or standard stream cipher

Stream ciphers (including BulkCipherAlgorithm.null) convert SSLCompressed.fragment structures to and from stream SSLCiphertext.fragment structures. The structure of GenericStreamCipher is as follow:

```c
stream-ciphered struct {
    opaque content[SSLCompressed.length];
    opaque MAC[CipherSpec.hash_size];
} GenericStreamCipher;
```

The MAC is generated as:

```
hash(MAC_write_secret + pad_2 + hash(MAC_write_secret + pad_1 + seq_num +
    SSLCompressed.type + SSLCompressed.length + SSLCompressed.fragment));
```

where "+" denotes concatenation.

<table>
<thead>
<tr>
<th>Pad_1</th>
<th>The character 0x36 repeated 48 times for MD5 or 40 times for SHA.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pad_2</td>
<td>The character 0x5c repeated 48 times for MD5 or 40 times for SHA.</td>
</tr>
<tr>
<td>Seq_num</td>
<td>The sequence number for this message.</td>
</tr>
<tr>
<td>Hash</td>
<td>Hashing algorithm derived from the cipher suite.</td>
</tr>
</tbody>
</table>

Note that the MAC is computed before encryption. The stream cipher encrypts the entire block, including the MAC. For stream ciphers that do not use a synchronization vector (such as RC4), the stream cipher state from the end of one record is simply used on the subsequent packet. If the CipherSuite is SSL_NULL_WITH_NULL_NULL, encryption consists of the identity operation (i.e., the data is not encrypted and the MAC size is zero implying that no MAC is used). SSLCiphertext.length is SSLCompressed.length plus CipherSpec.hash_size.
5.4.3.2 CBC block cipher

For block ciphers (such as RC2 or DES), the encryption and MAC functions convert SSLCompressed.fragment structures to and from block SSLCiphertext.fragment structures. The structure of GenericBlockCipher is as follow:

```c
block-ciphered struct {
    opaque content[SSLCompressed.length];
    opaque MAC[CipherSpec.hash_size];
    uint8 padding[GenericBlockCipher.padding_length];
    uint8 padding_length;
} GenericBlockCipher;
```

The MAC is generated as described in Section 5.4.3.1.

<table>
<thead>
<tr>
<th>Padding</th>
<th>Padding that is added to force the length of the plaintext to be a multiple of the block cipher's block length.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Padding_length</td>
<td>The length of the padding must be less than the cipher's block length and may be zero. The padding length should be such that the total size of the GenericBlockCipher structure is a multiple of the cipher's block length.</td>
</tr>
</tbody>
</table>

The encrypted data length (SSLCiphertext.length) is one more than the sum of SSLCompressed.length, CipherSpec.hash_size, and padding_length.

**Note:** With CBC block chaining the initialization vector (IV) for the first record is provided by the handshake protocol. The IV for subsequent records is the last ciphertext block from the previous record.

5.5 Change cipher spec protocol

The change cipher spec protocol exists to signal transitions in ciphering strategies. The protocol consists of a single message, which is encrypted and
compressed under the current (not the pending) CipherSpec. The message consists of a single byte of value 1. The structure of ChangeCipherSpec is as follow:

```c
struct {
    enum { change_cipher_spec(1), (255) } type;
} ChangeCipherSpec;
```

The change cipher spec message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the just-negotiated CipherSpec and keys. Reception of this message causes the receiver to copy the read pending state into the read current state. The client sends a change cipher spec message following handshake key exchange and certificate verify messages (if any), and the server sends one after successfully processing the key exchange message it received from the client. An unexpected change cipher spec message should generate an unexpected_message alert (see the next section). When resuming a previous session, the change cipher spec message is sent after the hello messages.

### 5.6 Alert protocol

One of the content types supported by the SSL Record layer is the alert type. Alert messages convey the severity of the message and a description of the alert. Alert messages with a level of fatal result in the immediate termination of the connection. In this case, other connections corresponding to the session may continue, but the session identifier must be invalidated, preventing the failed session from being used to establish new connections. Like other messages, alert messages are encrypted and compressed, as specified by the current connection state. The Alert structure and the messages is as follow:

```c
enum { warning(1), fatal(2), (255) } AlertLevel;
```

```c
enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
```
decompression_failure(30),
handshake_failure(40),
no_certificate(41),
bad_certificate(42),
unsupported_certificate(43),
certificate_revoked(44),
certificate_expired(45),
certificate_unknown(46),
illegal_parameter (47)
(255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;

5.6.1 Closure alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack. Either party may initiate the exchange of closing messages.

| Close_notify | This message notifies the recipient that the sender will not send any more messages on this connection. The session becomes unresumable if any connection is terminated without proper close_notify messages with level equal to warning. |

Either party may initiate a close by sending a close_notify alert. Any data received after a closure alert is ignored.

Each party is required to send a close_notify alert before closing the write side of the connection. It is required that the other party respond with a close_notify alert of its own and close down the connection immediately, discarding any pending writes. It is not required for the initiator of the close to wait for the responding close_notify alert before closing the read side of the connection.
5.6.2 Error alerts

Error handling in the SSL Handshake protocol is very simple. When an error is detected, the detecting party sends a message to the other party. Upon transmission or receipt of an fatal alert message, both parties immediately close the connection. Servers and clients are required to forget any session-identifiers, keys, and secrets associated with a failed connection. The following error alerts are defined:

<table>
<thead>
<tr>
<th>Error Alert</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unexpected_message</td>
<td>An inappropriate message was received. This alert is always fatal and should never be observed in communication between proper implementations.</td>
</tr>
<tr>
<td>Bad_record_mac</td>
<td>This alert is returned if a record is received with an incorrect MAC. This message is always fatal.</td>
</tr>
<tr>
<td>Decompression_failure</td>
<td>The decompression function received improper input (e.g. data that would expand to excessive length). This message is always fatal.</td>
</tr>
<tr>
<td>Handshake_failure</td>
<td>Reception of a handshake_failure alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available. This is a fatal error.</td>
</tr>
<tr>
<td>No_certificate</td>
<td>A no_certificate alert message may be sent in response to a certification request if no appropriate certificate is available.</td>
</tr>
<tr>
<td>Bad_certificate</td>
<td>A certificate was corrupt, contained signatures that did not verify correctly, etc.</td>
</tr>
<tr>
<td>Unsupported_certificate</td>
<td>A certificate was of an unsupported type.</td>
</tr>
<tr>
<td>Certificate_revoked</td>
<td>A certificate was revoked by its signer.</td>
</tr>
<tr>
<td>Certificate_expired</td>
<td>A certificate has expired or is not currently valid.</td>
</tr>
<tr>
<td>Certificate_unknown</td>
<td>Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.</td>
</tr>
<tr>
<td>Illegal_parameter</td>
<td>A field in the handshake was out of range or inconsistent with other fields. This is always fatal.</td>
</tr>
</tbody>
</table>
5.7 Handshake protocol overview

The cryptographic parameters of the session state are produced by the SSL Handshake Protocol, which operates on top of the SSL Record Layer. When a SSL client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and use public-key encryption techniques to generate shared secrets. These processes are performed in the handshake protocol, which can be summarized as follows: The client sends a client hello message to which the server must respond with a server hello message, or else a fatal error will occur and the connection will fail. The client hello and server hello are used to establish security enhancement capabilities between client and server. The client hello and server hello establish the following attributes: Protocol Version, Session ID, Cipher Suite, and Compression Method. Additionally, two random values are generated and exchanged: ClientHello.random and ServerHello.random.

Following the hello messages, the server will send its certificate, if it is to be authenticated. Additionally, a server key exchange message may be sent, if it is required (e.g. if their server has no certificate, or if its certificate is for signing only). If the server is authenticated, it may request a certificate from the client, if that is appropriate to the cipher suite selected. Now the server will send the server hello done message, indicating that the hello-message phase of the handshake is complete. The server will then wait for a client response. If the server has sent a certificate request Message, the client must send either the certificate message or a no_certificate alert. The client key exchange message is now sent, and the content of that message will depend on the public key algorithm selected between the client hello and the server hello. If the client has sent a certificate with signing ability, a digitally-signed certificate verify message is sent to explicitly verify the certificate.

At this point, a change cipher spec message is sent by the client, and the client copies the pending Cipher Spec into the current Cipher Spec. The client then immediately sends the finished message under the new algorithms, keys, and secrets. In response, the server will send its own change cipher spec message, transfer the pending to the current Cipher Spec, and send its finished message under the new Cipher Spec. At this point, the handshake is complete and the client and server may begin to exchange application layer data. (See figure 4).
Note: To help avoid pipeline stalls, ChangeCipherSpec is an independent SSL Protocol content type, and is not actually an SSL handshake message.

When the client and server decide to resume a previous session or duplicate an existing session (instead of negotiating new security parameters) the message flow is as follows:

The client sends a ClientHello using the Session ID of the session to be resumed. The server then checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a ServerHello with the same Session ID value. At this point, both client and server must send change cipher spec messages and proceed directly to finished messages. Once the re-establishment is complete, the client and server may begin to exchange application layer data. (See Figure 5.) If a Session ID match is not found, the server generates a new session ID and the SSL client and server perform a full handshake.
The contents and significance of each message will be presented in detail in the following sections.

5.8 Handshake protocol

The SSL Handshake Protocol is one of the defined higher level clients of the SSL Record Protocol. This protocol is used to negotiate the secure attributes of a session. Handshake messages are supplied to the SSL Record Layer, where they are encapsulated within one or more SSLPlaintext structures, which are processed and transmitted as specified by the current active session state. The structure of Handshake Messages is as follow:

```c
enum {
    hello_request(0), client_hello(1), server_hello(2), certificate(11),
    server_key_exchange (12), certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16), finished(20), (255)
} HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
}```
select (HandshakeType) {
    case hello_request: HelloRequest;
    case client_hello: ClientHello;
    case server_hello: ServerHello;
    case certificate: Certificate;
    case server_key_exchange: ServerKeyExchange;
    case certificate_request: CertificateRequest;
    case server_hello_done: ServerHelloDone;
    case certificate_verify: CertificateVerify;
    case client_key_exchange: ClientKeyExchange;
    case finished: Finished;
} body;
} Handshake;

The handshake protocol messages are presented in the order they must be sent; sending handshake messages in an unexpected order results in a fatal error. The Algorithm used in our implementation to analyze Handshake structure is as follows:

```
Algorithm AnalyseHandshake(Buffer : StreamOfBytes);
begin
    while Buffer not empty do
        begin
            Get the handshake message header
            Msg_Type := handshake message type;
            Msg_Body := handshake message body;
            case Msg_Type of
                Hello_Request:(AnalyseHelloRequest);
                Client_Hello:(AnalyseClientHello(Msg_Body));
                Server_Hello:(AnalyseServerHello(Msg_Body));
                Certificate:(AnalyseCertificate(Msg_Body));
                Server_Key_Exchange:(AnalyseServerKeyExchange(Msg_Body));
                Certificate_Request:(AnalyseCertificateRequest(Msg_Body));
                Server_Hello_Done:(AnalyseServerHelloDone(Msg_Body));
                Certificate_Verify:(AnalyseCertificateVerify(Msg_Body));
                Client_Key_Exchange:(AnalyseClientKeyExchange(Msg_Body));
                Finished:(AnalyseFinished(Msg_Body));
            end;
        end;
    end;
end;
```

We are going to describe each message in more detail. However, the ServerKeyExchange message is used for anonymous servers and needs more
complicated algorithms, So we decide to restrict our project to non-anonymous servers. Because anonymous servers is vulnerable to MAN-IN-THE-MIDDLE attack.

5.8.1 Hello messages

The hello phase messages are used to exchange security enhancement capabilities between the client and server. When a new session begins, the CipherSpec encryption, hash, and compression algorithms are initialized to null. The current CipherSpec is used for renegotiation messages.

5.8.2 Hello request

The hello request message may be sent by the server at any time, but will be ignored by the client if the handshake protocol is already underway. It is a simple notification that the client should begin the negotiation process anew by sending a client hello message when convenient.

Note: Since handshake messages are intended to have transmission precedence over application data, it is expected that the negotiation begin in no more than one or two times the transmission time of a maximum length application data message.

After sending a hello request, servers should not repeat the request until the subsequent handshake negotiation is complete. A client that receives a hello request while in a handshake negotiation state should simply ignore the message.

The structure of a hello request message is as follows:

struct { } HelloRequest;

5.8.3 Client hello

When a client first connects to a server it is required to send the client hello as its first message. The client can also send a client hello in response to a hello request or on its own initiative in order to renegotiate the security parameters in an existing connection. The client hello message includes a random structure, which is used later in the protocol. The structure of the Random is as follow:
struct {
  uint32 gmt_unix_time;
  opaque random_bytes[28]; } Random;

<table>
<thead>
<tr>
<th>GMT_unix_time</th>
<th>The current time and date in standard UNIX 32-bit format according to the sender's internal clock. Clocks are not required to be set correctly by the basic SSL Protocol; higher level or application protocols may define additional requirements.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random_bytes</td>
<td>28 bytes generated by a secure random number generator.</td>
</tr>
</tbody>
</table>

The client hello message includes a variable length session identifier. If not empty, the value identifies a session between the same client and server whose security parameters the client wishes to reuse. The session identifier may be from an earlier connection, this connection, or another currently active connection. The second option is useful if the client only wishes to update the random structures and derived values of a connection, while the third option makes it possible to establish several simultaneous independent secure connections without repeating the full handshake protocol. The actual contents of the SessionID are defined by the server. The structure of SessionID is as follow:

opaque SessionID<0..32>;

The Client hello message have a CipherSuite list, passed from the client to the server in the client hello message, contains the combinations of cryptographic algorithms supported by the client in order of the client's preference (first choice first). Each CipherSuite defines both a key exchange algorithm and a CipherSpec. The server will select a cipher suite or, if no acceptable choices are presented, return a handshake failure alert and close the connection. The structure of CipherSuite and the Suites is as follow:

uint8 CipherSuite[2]; /* Cryptographic suite selector */
The client hello includes a list of compression algorithms supported by the client, ordered according to the client's preference. If the server supports none of those specified by the client, the session must fail.

```
enum { null(0), (255) } CompressionMethod;
```

**Issue:** Which compression methods to support are under investigation. Until now there is no compression method has been supported (even in TLS ver 1.0).

The structure of the client hello is as follows.

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-1>;
    CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;
```

<table>
<thead>
<tr>
<th><strong>Client_version</strong></th>
<th>The version of the SSL protocol by which the client wishes to communicate during this session. This should be the most recent (highest valued) version supported by the client. For this version of the specification, the version will be 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Random</strong></td>
<td>A client-generated random structure.</td>
</tr>
<tr>
<td>Session_id</td>
<td>The ID of a session the client wishes to use for this connection. This field should be empty if no session_id is available or the client wishes to generate new security parameters.</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Cipher_suites</td>
<td>This is a list of the cryptographic options supported by the client, sorted with the client's first preference first. If the session_id field is not empty (implying a session resumption request) this vector must include at least the cipher_suite from that session.</td>
</tr>
<tr>
<td>Compression_methods</td>
<td>This is a list of the compression methods supported by the client, sorted by client preference. If the session_id field is not empty (implying a session resumption request) this vector must include at least the compression_method from that session. All implementations must support CompressionMethod.null.</td>
</tr>
</tbody>
</table>

After sending the client hello message, the client waits for a server hello message. Any other handshake message returned by the server except for a hello request is treated as a fatal error.

**Implementation note:**

Application data may not be sent before a finished message has been sent. Transmitted application data is known to be insecure until a valid finished message has been received. This absolute restriction is relaxed if there is a current, non-null encryption on this connection.

**5.8.4 Server hello**

The server processes the client hello message and responds with either a handshake_failure alert or server hello message. The Structure of ServerHello is as follow:
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
} ServerHello;

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Server_version</strong></td>
<td>This field will contain the lower of that suggested by the client in the client hello and the highest supported by the server. For this version of the specification, the version will be 3.0.</td>
</tr>
<tr>
<td><strong>Random</strong></td>
<td>This structure is generated by the server and must be different from (and independent of) ClientHello.random.</td>
</tr>
<tr>
<td><strong>Session_id</strong></td>
<td>This is the identity of the session corresponding to this connection. If the ClientHello.session_id was non-empty, the server will look in its session cache for a match. If a match is found and the server is willing to establish the new connection using the specified session state, the server will respond with the same value as was supplied by the client. This indicates a resumed session and dictates that the parties must proceed directly to the finished messages. Otherwise this field will contain a different value identifying the new session. The server may return an empty session_id to indicate that the session will not be cached and therefore cannot be resumed.</td>
</tr>
<tr>
<td><strong>Cipher_suite</strong></td>
<td>The single cipher suite selected by the server from the list in ClientHello.cipher_suites. For resumed sessions this field is the value from the state of the session being resumed.</td>
</tr>
<tr>
<td><strong>Compression_method</strong></td>
<td>The single compression algorithm selected by the server from the list in ClientHello.compression_methods. For resumed sessions this field is the value from the resumed session state.</td>
</tr>
</tbody>
</table>
5.8.5 Server certificate

If the server is to be authenticated (which is generally the case), the server sends its certificate immediately following the server hello message. The certificate type must be appropriate for the selected cipher suite's key exchange algorithm, and is generally an X.509v3 certificate (see chapter 7). The same message type will be used for the client's response to a certificate request message. The Structure of Server Certificate is as follow:

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

    certificate_list  This is a sequence (chain) of X.509.v3 certificates, ordered with the sender's certificate first followed by any certificate authority certificates proceeding sequentially upward.

5.8.6 Certificate request

A non-anonymous server can optionally request a certificate from the client, if appropriate for the selected cipher suite. The Structure of Certificate Request is as follow:

enum {
    rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4), rsa_ephemeral_dh(5),
    dss_ephemeral_dh(6), fortezza_kea(20), (255)
} ClientCertificateType;

opaque DistinguishedName<1..2^16-1>;

struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    DistinguishedName certificateAuthorities<3..2^16-1>;
} CertificateRequest;
<table>
<thead>
<tr>
<th><strong>Certificate_types</strong></th>
<th>This field is a list of the types of certificates requested, sorted in order of the server's preference.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CertificateAuthorities</strong></td>
<td>A list of the distinguished names of acceptable certificate authorities.</td>
</tr>
</tbody>
</table>

Note: DistinguishedName is derived from [14].

Note: It is a fatal handshake_failure alert for an anonymous server to request client identification.

### 5.8.7 Server hello done

The server hello done message is sent by the server to indicate the end of the server hello and associated messages. After sending this message the server will wait for a client response. The Structure of ServerHelloDone is as follow:

```c
struct { } ServerHelloDone;
```

Upon receipt of the server hello done message the client should verify that the server provided a valid certificate if required and check that the server hello parameters are acceptable.

### 5.8.8 Client certificate

This is the first message the client can send after receiving a server hello done message. This message is only sent if the server requests a certificate. If no suitable certificate is available, the client should send a no_certificate alert instead. This alert is only a warning, however the server may respond with a fatal handshake failure alert if client authentication is required. Client certificates are sent using the Certificate defined in Section 5.8.5.

### 5.8.9 Client key exchange message

The choice of messages depends on which public key algorithm(s) has (have) been selected. In our case the only supported algorithm is RSA. The Structure of ClientKeyExchange is as follow:
struct {
    select (KeyExchangeAlgorithm) {
        case rsa: EncryptedPreMasterSecret;
        case diffie_hellman: ClientDiffieHellmanPublic;
        case fortezza_kea: FortezzaKeys;
    } exchange_keys;
} ClientKeyExchange;

The information to select the appropriate record structure is in the pending session state.

**RSA encrypted premaster secret message**

If RSA is being used for key agreement and authentication, the client generates a 48-byte pre-master secret, encrypts it under the public key from the server's certificate or temporary RSA key from a server key exchange message, and sends the result in an encrypted premaster secret message. The Structure of PreMasterSecret is as follow:

```c
struct {
    ProtocolVersion client_version;
    opaque random[46];
} PreMasterSecret;
```

<table>
<thead>
<tr>
<th>Client_version</th>
<th>The latest (newest) version supported by the client. This is used to detect version roll-back attacks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>46 securely-generated random bytes.</td>
</tr>
</tbody>
</table>

The PreMasterSecret is encrypted using RSA algorithm after forming it into PKCS#1 Block. (See chapter 6 on how to form a PKCS#1 Block).

```c
struct {
    public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;
```
Pre_master_secret

This random value is generated by the client and is used to generate the master secret, as specified in Section 5.12.

5.8.10 Certificate verify

This message is used to provide explicit verification of a client certificate. This message is only sent following any client certificate that has signing capability. The structure of CertificateVerify is as follow:

```plaintext
digitally-signed struct {
    select(SignatureAlgorithm) {
        case anonymous: struct { };
        case rsa:
            opaque md5_hash[16];
            opaque sha_hash[20];
        case dsa:
            opaque sha_hash[20];
    }
} Signature;
struct {
    Signature signature;
} CertificateVerify;
```

CertificateVerify.signature.md5_hash

MD5(master_secret + pad_2 + MD5(handshake_messages + master_secret + pad_1));

Certificate.signature.sha_hash

SHA(master_secret + pad_2 + SHA(handshake_messages + master_secret + pad_1));
Pad_1 This is identical to the pad_1 defined in section 5.4.3.1.

Pad_2 This is identical to the pad_2 defined in section 5.4.3.1.

Here handshake_messages refers to all handshake messages starting at client hello up to but not including this message.

5.8.11 Finished

A finished message is always sent immediately after a change cipher specs message to verify that the key exchange and authentication processes were successful. The finished message is the first protected with the just-negotiated algorithms, keys, and secrets. No acknowledgment of the finished message is required; parties may begin sending encrypted data immediately after sending the finished message. Recipients of finished messages must verify that the contents are correct. The Structure of Finished message is as follow:

```c
enum { client(0x434C4E54), server(0x53525652) } Sender;

struct {
    opaque md5_hash[16];
    opaque sha_hash[20];
} Finished;

md5_hash  MD5(master_secret + pad2 + MD5(handshake_messages + Sender + master_secret + pad1));
sha_hash   SHA(master_secret + pad2 + SHA(handshake_messages + Sender + master_secret + pad1));
```

| Handshake_messages | All of the data from all handshake messages up to but not including this message. This is only data visible at the handshake layer and does not include record layer headers. |

It is a fatal error if a finished message is not preceded by a change cipher spec message at the appropriate point in the handshake.
The hash contained in finished messages sent by the server incorporate Sender.server; those sent by the client incorporate Sender.client. The value handshake_messages includes all handshake messages starting at client hello up to, but not including, this finished message. This may be different from handshake_messages in Section 5.9.8 because it would include the certificate verify message (if sent).

*Note:* Change cipher spec messages are not handshake messages and are not included in the hash computations.

### 5.9 Application data protocol

Application data messages are carried by the Record Layer and are fragmented, compressed and encrypted based on the current connection state. The messages are treated as transparent data to the record layer.

### 5.10 Cryptographic computations

The key exchange, authentication, encryption, and MAC algorithms are determined by the cipher_suite selected by the server and revealed in the server hello message.

#### 5.10.1 Asymmetric cryptographic computations

The asymmetric algorithms are used in the handshake protocol to authenticate parties and to generate shared keys and secrets.

The algorithm that is used to convert the pre_master_secret into the master_secret is as follow:

```
master_secret =
    MD5(pre_master_secret + SHA('A' + pre_master_secret + ClientHello.random +
           ServerHello.random)) +
    MD5(pre_master_secret + SHA('BB' + pre_master_secret + ClientHello.random +
                              ServerHello.random)) +
```

MD5(pre_master_secret + SHA('CCC' + pre_master_secret + ClientHello.random + ServerHello.random));

**RSA**

When RSA is used for server authentication and key exchange, a 48-byte pre_master_secret is generated by the client, encrypted under the server's public key, and sent to the server. The server uses its private key to decrypt the pre_master_secret. Both parties then convert the pre_master_secret into the master_secret, as specified above.

RSA digital signatures are performed using PKCS #1 block type 1 (See chapter 6 for more detail). RSA public key encryption is performed using PKCS #1 block type 2.

### 5.10.2 Symmetric cryptographic calculations and the CipherSpec

The technique used to encrypt and verify the integrity of SSL records is specified by the currently active CipherSpec. A typical example would be to encrypt data using DES and generate authentication codes using MD5. The encryption and MAC algorithms are set to SSL_NULL_WITH_NULL_NULL at the beginning of the SSL Handshake Protocol, indicating that no message authentication or encryption is performed. The handshake protocol is used to negotiate a more secure CipherSpec and to generate cryptographic keys.

### 5.10.1 The master secret

Before secure encryption or integrity verification can be performed on records, the client and server need to generate shared secret information known only to themselves. This value is a 48-byte quantity called the master secret. The master secret is used to generate keys and secrets for encryption and MAC computations.

### 5.10.2 Converting the master secret into keys and MAC secrets

The master secret is hashed into a sequence of secure bytes, which are assigned to the MAC secrets, keys, and non-export IVs required by the current CipherSpec. CipherSpecs require a client write MAC secret, a server write MAC secret, a client write key, a server write key, a client write IV, and a server write IV, which are
generated from the master secret in that order. The following inputs are available to the key definition process:

opaque MasterSecret[48]
ClientHello.random
ServerHello.random

When generating keys and MAC secrets, the master secret is used as an entropy source, and the random values provide unencrypted salt material and IVs for exportable ciphers.

To generate the key material, compute

key_block =

MD5(master_secret + SHA(`A' + master_secret + ServerHello.random + ClientHello.random)) +
MD5(master_secret + SHA(`BB' + master_secret + ServerHello.random + ClientHello.random)) +
MD5(master_secret + SHA(`CCC' + master_secret + ServerHello.random + ClientHello.random)) + [...];

Until enough output has been generated. Then the key_block is partitioned as follows.

client_write_MAC_secret[CipherSpec.hash_size]
server_write_MAC_secret[CipherSpec.hash_size]
client_write_key[CipherSpec.key_material]
server_write_key[CipherSpec.key_material]
client_write_IV[CipherSpec.IV_size] /* non-export ciphers */
server_write_IV[CipherSpec.IV_size] /* non-export ciphers */

Any extra key_block material is discarded.

Exportable encryption algorithms (for which CipherSpec.is_exportable is true) require additional processing as follows to derive their final write keys:
final_client_write_key = MD5(client_write_key + ClientHello.random + ServerHello.random);
final_server_write_key = MD5(server_write_key + ServerHello.random + ClientHello.random);

Exportable encryption algorithms derive their IVs from the random messages:

client_write_IV = MD5(ClientHello.random + ServerHello.random);
server_write_IV = MD5(ServerHello.random + ClientHello.random);

MD5 outputs are trimmed to the appropriate size by discarding the least-significant bytes.

**Export key generation example**

SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5 requires five random bytes for each of the two encryption keys and 16 bytes for each of the MAC keys, for a total of 42 bytes of key material. MD5 produces 16 bytes of output per call, so three calls to MD5 are required. The MD5 outputs are concatenated into a 48-byte key_block with the first MD5 call providing bytes zero through 15, the second providing bytes 16 through 31, etc. The key_block is partitioned, and the write keys are salted because this is an exportable encryption algorithm.

client_write_MAC_secret = key_block[0..15]
server_write_MAC_secret = key_block[16..31]
client_write_key = key_block[32..36]
server_write_key = key_block[37..41]
final_client_write_key = MD5(client_write_key + ClientHello.random + ServerHello.random)[0..15];
final_server_write_key = MD5(server_write_key + ServerHello.random + ClientHello.random)[0..15];
client_write_IV = MD5(ClientHello.random + ServerHello.random)[0..7];
server_write_IV = MD5(ServerHello.random + ClientHello.random)[0..7];
Cryptography Module
6.1 Introduction

The Cryptography module offers SSL module a set of encryption and decryption algorithms to encrypt and decrypt the data that will be sent. The algorithms that will be offered by the Cryptography module are:

1. **Public (asymmetric) Key Encryption Algorithms**: Used for encrypting the negotiated key that will be used in data encryption.
   a. RSA.

2. **Symmetric Key Encryption Algorithms**: Used for encrypting the data that will be exchanged.
   a. RC4
   b. RC2
   c. IDEA
   d. DES

3. **Message Digest algorithms**
   a. MD5
   b. SHA-1

6.2 Public Key Encryption Algorithms

The cryptography module has the only supported public key by SSL protocol, which is RSA algorithm. It's used for exchanging the key that will be used for encrypting the data. However, there is a weak point in public key algorithms that prevent the protocol from encrypting data as its.

The problem arises when the sent message is a set of $n$ possible messages. A cryptanalyst can encrypt all the $n$ possible messages (as the encryption key is public) and compare the result with the encrypted incoming encrypted message. In this way he can know which was the message that has been sent. So this type of attack has been taken into account in the standard RSA encryption algorithm which is PKCS#1. This standard describes how to encrypt messages using RSA algorithm. We are going here to present what we are interesting in and any further description can be found in
The main topic that we are interested in is the format of the messages that will be encrypted by RSA. Any message that will be encrypted by RSA algorithm in SSL 3.0 protocol must follow PKCS#1 ver1.5 standard. Each message is converted to one or more blocks, the format of the block is as follows (which has extracted from PKCS#1 ver 1.5 standard):

The block consists of: A block type $BT$, a padding string $PS$, and the data $D$ shall be formatted into a stream of bytes $EB$, the encryption block.

$$ EB = 00 \parallel BT \parallel PS \parallel 00 \parallel D $$

$X \parallel Y$  concatenation of $X, Y$

$||X||$  length in bytes of $X$

The block type $BT$ shall be a single byte indicating the structure of the encryption block. For version 1.5 of PKCS#1 standard it shall have value 00, 01, or 02. For a private-key operation, the block type shall be 00 or 01. For a public-key operation, it shall be 02.

The padding string $PS$ shall consist of $k-3-||D||$ bytes. For block type 00, the bytes shall have value 00; for block type 01, they shall have value FF; and for block type 02, they shall be pseudorandomly generated and nonzero. This makes the length of the encryption block $EB$ equal to the length of the modulus.

**Notes.**

1. The leading 00 byte ensures that the encryption block, converted to an integer, is less than the modulus.

2. For block type 00, the data $D$ must begin with a nonzero octet or have known length so that the encryption block can be parsed unambiguously. For block types 01 and 02, the encryption block can be parsed unambiguously since the padding string $PS$ contains no bytes with value 00 and the padding string is separated from the data $D$ by a byte with value 00.

3. Block type 01 is recommended for private-key operations. Block type 01 has the property that the encryption block, converted to an integer, is guaranteed to be large.
4. Block types 01 and 02 are compatible with PEM RSA encryption of content-encryption keys and message digests as described in RFC 1423.

5. For block type 02, it is recommended that the pseudorandom byte be generated independently for each encryption process, especially if the same data is input to more than one encryption process.

6. For block type 02, the padding string is at least eight byte long, which is a security condition for public-key operations that prevents an attacker from recovering data by trying all possible encryption blocks. For simplicity, the minimum length is the same for block type 01.

7. This standard may be extended in the future to include other block types.[2]

After formatting the block(s) it will be encrypted with the public key using RSA component that has been developed.

6.2.1 RSA Component

The RSA component used for encrypting messages using the algorithm described in chapter 4. The component contains two main subjects. First, Big Numbers Library. Second, RSA Algorithm.

(a) Big Numbers: For more Security, The RSA Algorithm uses very large Numbers (e.g. 1024-bit numbers) and because there is no such primitive operations in computer that can handle these numbers, a library that encapsulate all primitive operations on these numbers (e.g. +, *, / and mod) must be developed.

(b) RSA Algorithm: It uses Big Number library to do the encryption procedure described in chapter 4.
6.3 Symmetric Key Encryption Algorithms:

There are two basic types of symmetric algorithms: block ciphers and stream ciphers. Block ciphers operate on blocks of plaintext usually of 64 bits but some times longer. Stream ciphers operate on stream of plaintext one bit or byte at a time. With block cipher, the same plaintext block will always encrypt to the same ciphertext block, using the same key. With stream cipher, the same plaintext bit or byte will encrypt to different bit or byte every time its encrypted.

6.3.1 Stream Ciphers:

Stream cipher converts plaintext to ciphertext bit or byte at atime. It uses a key stream generator to output a stream of bits. This key stream is XORed with a stream of plaintext to produce the stream of ciphertext bits. SSL supports only one stream cipher, which is RC4.

RC4 Algorithm Component

The RC4 Component implements the RC4 Algorithm described in [13]. For technical description of the algorithm refer to [8]. Here we are just going to describe how to use this component. Using the component is very simple, it needs at most two steps to do the work.

1) Initializing RC4: using procedure \texttt{RC4Initialize} to initialize the key as a parameter. This step done only once at the beginning.

2) Encryption and Decryption: two procedures used for encryption and decryption, \texttt{RC4Encrypt} and \texttt{RC4Decrypt}. It takes as a parameter \texttt{Key} and \texttt{Data block}.

6.3.2 Block Ciphers:

There are many types of block cipher. The supported type by SSL protocol is Cipher Block Chaining (CBC) mode. This type uses feedback mechanism, which means that the result of the encryption of previous blocks is fed back into the
encryption of the current block. In other word, the plaintext is XORed with the previous ciphertext block before it is encrypted. The first block in our case in XORed with the Initialization Vector (IV) that has been agreed upon between the SSL client and server during the handshake. All cipher supported by SSL protocol are block ciphers except RC4.

6.4 Message Digest Algorithms

To allow the client and the server to verify that the message comes from the other party and has not been tampered with by another person in the middle, each message is hashed using the hashing algorithm agreed on between the client and server. There are many hashing algorithms available but the SSL 3.0 protocol supports only two hashing algorithms, which are MD5 and SHA-1.

Note: Due to vulnerability in the hashing procedure used in SSL 3.0 protocol (which is MAC), the new standard TLS 1.0 protocol uses different procedures for hashing called HMAC.

6.5 SSL Package Cipher suites:

There are many cipher suites supported by the SSL protocol, but due to U.S. export restrictions many of these ciphers are not allowed to be used outside the U.S. so until an SSL server is implemented that accepts these band ciphers we can't test them on the Internet. This with the time limitations limited the implemented cipher suites to the following:

<table>
<thead>
<tr>
<th>Cipher Suite</th>
<th>Cipher</th>
<th>Cipher type</th>
<th>Key size</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSL_NULL_WITH_NULL_NULL</td>
<td>NULL</td>
<td>Stream</td>
<td>0</td>
<td>NULL</td>
</tr>
<tr>
<td>SSL_RSA_WITH_NULL_MD5</td>
<td>NULL</td>
<td>Stream</td>
<td>0</td>
<td>MD5</td>
</tr>
<tr>
<td>SSL_RSA_WITH_NULL_SHA</td>
<td>NULL</td>
<td>Stream</td>
<td>0</td>
<td>SHA</td>
</tr>
<tr>
<td>SSL_RSA_EXPORT_WITH_RC4_40_MD5</td>
<td>RC4</td>
<td>Stream</td>
<td>40</td>
<td>MD5</td>
</tr>
<tr>
<td>SSL_RSA_WITH_RC4_128_MD5</td>
<td>RC4</td>
<td>Stream</td>
<td>128</td>
<td>MD5</td>
</tr>
<tr>
<td>SSL_RSA_WITH_RC4_128_SHA</td>
<td>RC4</td>
<td>Stream</td>
<td>128</td>
<td>SHA</td>
</tr>
<tr>
<td>SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5</td>
<td>RC2</td>
<td>Block</td>
<td>40</td>
<td>MD5</td>
</tr>
<tr>
<td>SSL_RSA_WITH_IDEA_CBC_SHA</td>
<td>IDEA</td>
<td>Block</td>
<td>128</td>
<td>SHA</td>
</tr>
<tr>
<td>SSL_RSA_EXPORT_WITH_DES40_CBC_SHA</td>
<td>DES</td>
<td>Block</td>
<td>40</td>
<td>SHA</td>
</tr>
<tr>
<td>SSL_RSA_WITH_DES_CBC_SHA</td>
<td>DES</td>
<td>Block</td>
<td>128</td>
<td>SHA</td>
</tr>
</tbody>
</table>
In case of IDEA the algorithm has been tested on test vectors, and proved to be error free, but we could not find a server that accept it, so its until now untested on an actual SSL session.
X.509 Authentication Module
Brief Introduction:

SSL authentication is based on the ITU-T recommendation X.509, which specifies how authentication should be performed and gives a general format that should be followed. In this chapter we will cover the main fields of the X.509 certificate that are critical to the success of the SSL connection, with an explanation of each of these fields. Also we will explain the X.509 object and the Base64 object, the latter is used by the former to encode and decode the certificates.
7.1 What is X.509? :

ITU-T Recommendation X.509 specifies the authentication service for X.500 directories, as well as the widely adopted X.509 certificate syntax. The initial version of X.509 was published in 1988, version 2 was published in 1993, and version 3 was proposed in 1994 and considered for approval in 1995. Version 3 addresses some of the security concerns and limited flexibility that were issues in versions 1 and 2.

Directory authentication in X.509 can be carried out using either secret-key techniques or public-key techniques; the latter is based on public-key certificates. The standard does not specify a particular cryptographic algorithm, although an informative annex of the standard describes the RSA algorithm.

An X.509 certificate consists of the following fields:

- version
- serial number
- signature algorithm ID
- issuer name
- validity period
- subject (user) name
- subject public key information
- issuer unique identifier (version 2 and 3 only)
- subject unique identifier (version 2 and 3 only)
- extensions (version 3 only)
- signature on the above fields

This certificate is signed by the issuer to authenticate the binding between the subject (user's) name and the user's public key. The major difference between versions 2 and 3 is the addition of the extensions field. This field grants more flexibility as it can convey additional information beyond just the key and name binding. Standard extensions include subject and issuer attributes, certification policy information, and key usage restrictions, among others.

To authenticate its self the SSL server sends an X.509 Certificate to the SSL client. The server can also ask for a client to authenticate its self with a certificate.
7.2 X.509 structure in ASN.1:

The following is just the outer shell of X.509, which encapsulates the ASN.1 structure of X.509. For the complete ASN.1 structure for X.509 see ITU-T Recommendation X.509.

Certificate ::= SEQUENCE {
  tbsCertificate TBSCertificate,
  signatureAlgorithm AlgorithmIdentifier,
  signature BIT STRING
}

TBSCertificate ::= SEQUENCE {
  version [ 0 ] Version DEFAULT v1(0),
  serialNumber CertificateSerialNumber,
  signature AlgorithmIdentifier,
  issuer Name,
  validity Validity,
  subject Name,
  subjectPublicKeyInfo SubjectPublicKeyInfo,
  issuerUniqueID [ 1 ] IMPLICIT UniqueIdentifier OPTIONAL,
  subjectUniqueID [ 2 ] IMPLICIT UniqueIdentifier OPTIONAL,
  extensions [ 3 ] Extensions OPTIONAL
}

The Certificate is a SEQUENCE of:

1- **TBScertificate**

It is a SEQUENCE of the following:

(a) **version**

Version ::= INTEGER { v1(0), v2(1), v3(2) }

The certificate has three versions, when extensions are used, use X.509 version 3 (value is 2). If no extensions are present, but a UniqueIdentifier is present, use version 2 (value is 1). If only basic fields are present, use version 1 (the value is omitted from the certificate as the default value). If this field not present then as indicated be DEFAULT tag the Version will be v1(0).

The presence of Version 1 is indicated by the tag [0].

(b) **serialNumber**

CertificateSerialNumber ::= INTEGER

This contains the serial number of the Certificate. It’s an ASN.1 Integer.

It should be unique for every certificate issued by the CA.

(c) **issuer**

Contains the algorithm identifier for the signature algorithm used by the CA to sign the certificate.
(d) validity

Validity ::= SEQUENCE {
  notBefore      Time,
  notAfter       Time }

Time ::= CHOICE {
  utcTime        UTCTime,
  generalTime    GeneralizedTime }

The certificate validity period is the time interval during which the CA warrants that it will maintain information about the status of the certificate. The field is represented as a SEQUENCE of two dates: the date on which the certificate validity period begins (notBefore) and the date on which the certificate validity period ends (notAfter). Both notBefore and notAfter may be encoded as UTCTime or GeneralizedTime.

UTCTime specifies the year through the two low order digits and time is specified to the precision of one minute or one second. UTCTime includes either Z (for Zulu, or Greenwich Mean Time) or a time differential. For the purposes of this profile, UTCTime values MUST be expressed Greenwich Mean Time (Zulu) and MUST include seconds (i.e., times are YYMMDDHHMMSSZ), even where the number of seconds is zero. The year field (YY) is interpreted as follows:

Where YY is greater than or equal to 50, the year shall be interpreted as 19YY; and where YY is less than 50, the year shall be interpreted as 20YY.

GeneralizedTime values MUST be expressed Greenwich Mean Time (Zulu) and MUST include seconds (i.e., times are YYYYMMDDHHMMSSZ), even where the number of seconds is zero. GeneralizedTime values MUST NOT include fractional seconds.

(e) subject

The subject field identifies the entity associated with the public key stored in the subject public key field (the CA). The subject name may be carried in the subject field and/or the subjectAltName extension.

(f) subjectPublicKeyinfo

SubjectPublicKeyInfo ::= SEQUENCE {
  algorithm            AlgorithmIdentifier,
  subjectPublicKey     BIT STRING }
This field is used to carry the public key and identify the algorithm with which the key is used. The algorithm is identified using the AlgorithmIdentifier structure. The subjectPublicKey that holds the RSA key.

```plaintext
RSAPublicKey ::= SEQUENCE {
  modulus            INTEGER,  -- n
  publicExponent     INTEGER  -- e -- }
```

(g) **issuerUniqueID & subjectUniqueID**

```plaintext
UniqueId ::= BIT STRING
```

These fields may only appear if the version is 2 or 3. The subject and issuer unique identifiers are present in the certificate to handle the possibility of reuse of subject and/or issuer names over time. This profile recommends that names not be reused for different entities and that Internet certificates not make use of unique identifiers.

(h) **extensions**

```plaintext
Extensions ::= SEQUENCE SIZE (1..MAX) OF Extension
```

2- **signatureAlgorithm**

```plaintext
AlgorithmIdentifier ::= SEQUENCE {
  algorithm               OBJECT IDENTIFIER,
  parameters              ANY DEFINED BY algorithm OPTIONAL  }
```

This field may only appear if the version is 3. If present, this field is a SEQUENCE of one or more certificate extensions. The extensions defined for X.509 v3 certificates provide methods for associating additional attributes with users or public keys and for managing the certification hierarchy.

3- **signature**

The signature field contains a digital signature computed upon the ASN.1 DER encoded tbsCertificate. The ASN.1 DER encoded tbsCertificate is used as the input to the signature function. This signature value is then ASN.1 encoded as a BIT STRING and included in the Certificate's signature field. By generating this signature, a CA certifies the validity of the information in the tbsCertificate field. In particular, the CA certifies the binding between the public key material and the subject of the certificate.
7.3 The X.509 Object:

The X.509 object is implemented in a Pascal unit called X509v3.pas. We will show first the Object Methods and Properties.

P = Property, M = Method.

<table>
<thead>
<tr>
<th>P/M</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>CertData : TmemoryStream;</td>
<td>The Certificate in its raw Binary form</td>
</tr>
<tr>
<td>M</td>
<td>Function Initilize ( Buf : TmemoryStream) : Boolean;</td>
<td>A polymorphic function that:</td>
</tr>
<tr>
<td></td>
<td>OR Function Initilize ( FileName : String): Boolean;</td>
<td>If input is Buf it stores Buf into CertData else it decodes the Base64 encoded Certificate in the file specified by FileName and stores it in CertData. Then it extracts needed information from CertData. It returns True if its successful otherwise it returns False.</td>
</tr>
<tr>
<td>M</td>
<td>Function GetVersion : integer</td>
<td>Returns the Certificate Version number</td>
</tr>
<tr>
<td>M</td>
<td>Function GetSerialNumber : TmemoryStream;</td>
<td>Returns Serial Number of the certificate</td>
</tr>
<tr>
<td>M</td>
<td>Function GetValidity : Tvalidity;</td>
<td>Returns the NotBefor and NotAfter validity dates</td>
</tr>
<tr>
<td>M</td>
<td>Function GetPK : TRSAKey;</td>
<td>Returns the RSA key which will be used in Key exchange.</td>
</tr>
</tbody>
</table>

As you can see this X.509 object currently implements retrieval of fields which are critical to the completion of the SSL handshake.

7.4 How to use X.509 object:

1- Include X509v3 to the uses clause.
2- Make a variable (Say Cert) of type X509.
3- Create Cert by typing Cert := X509.Create;
4- Initialize Cert from Memory or file like this
If Cert.Initialize(Buf) then
begin
  //use Cert to get needed information
end
else
  //Give ERROR Message

If you want to load the Certificate from a file send FileName : string instead
of Buf : TmemoryStream as explained earlier

5- Destroy Cert by means of Cert.Free;

7.5 What is Base64? :

The Base64 Content-Transfer-Encoding is designed to represent arbitrary sequences of octets in a form that need not be humanly readable. The encoding and decoding algorithms are simple, but the encoded data are consistently only about 33 percent larger than the UUencoded data. This encoding is virtually identical to the one used in Privacy Enhanced Mail (PEM) applications, as defined in RFC 1421. The Base64 encoding is adapted from RFC 1421, with one change: Base64 eliminates the "*" mechanism for embedded clear text.

A 65-character subset of US-ASCII is used, enabling 6 bits to be represented per printable character. (The extra 65th character, ",", is used to signify a special processing function.)

Note: This subset has the important property that it is represented identically in all versions of ISO 646, including US ASCII, and all characters in the subset are also represented identically in all versions of EBCDIC. Other popular encodings, such as the encoding used by the uuencode utility and the base85 encoding specified as part of Level 2 PostScript, do not share these properties, and thus do not fulfill the portability requirements a binary transport encoding for mail must meet.

The encoding process represents 24-bit groups of input bits as output strings of 4 encoded characters. Proceeding from left to right, a 24-bit input group is formed by concatenating 3 8-bit input groups. These 24 bits are then treated as 4 concatenated 6-bit groups, each of which is translated into a single digit in the Base64 alphabet.
When encoding a bit stream via the Base64 encoding, the bit stream must be presumed to be ordered with the most-significant-bit first.

That is, the first bit in the stream will be the high-order bit in the first byte, and the eighth bit will be the low-order bit in the first byte, and so on.

Each 6-bit group is used as an index into an array of 64 printable characters. The character referenced by the index is placed in the output string. These characters, identified in Table 1, below, are selected so as to be universally representable, and the set excludes characters with particular significance to SMTP (e.g., ".", CR, LF) and to the encapsulation boundaries defined in this document (e.g., ":-").

<table>
<thead>
<tr>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
<th>Value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>18</td>
<td>S</td>
<td>36</td>
<td>k</td>
<td>54</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>19</td>
<td>T</td>
<td>37</td>
<td>l</td>
<td>55</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>20</td>
<td>U</td>
<td>38</td>
<td>m</td>
<td>56</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>21</td>
<td>V</td>
<td>39</td>
<td>n</td>
<td>57</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>22</td>
<td>W</td>
<td>40</td>
<td>o</td>
<td>58</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>23</td>
<td>X</td>
<td>41</td>
<td>p</td>
<td>59</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>24</td>
<td>Y</td>
<td>42</td>
<td>q</td>
<td>60</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>H</td>
<td>25</td>
<td>Z</td>
<td>43</td>
<td>r</td>
<td>61</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>I</td>
<td>26</td>
<td>a</td>
<td>44</td>
<td>s</td>
<td>62</td>
<td>+</td>
</tr>
<tr>
<td>9</td>
<td>J</td>
<td>27</td>
<td>b</td>
<td>45</td>
<td>t</td>
<td>63</td>
<td>/</td>
</tr>
<tr>
<td>10</td>
<td>K</td>
<td>28</td>
<td>c</td>
<td>46</td>
<td>u</td>
<td>(pad)</td>
<td>=</td>
</tr>
<tr>
<td>11</td>
<td>L</td>
<td>29</td>
<td>d</td>
<td>47</td>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>M</td>
<td>30</td>
<td>e</td>
<td>48</td>
<td>w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>N</td>
<td>31</td>
<td>f</td>
<td>49</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>O</td>
<td>32</td>
<td>g</td>
<td>50</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>P</td>
<td>33</td>
<td>h</td>
<td>51</td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Q</td>
<td>34</td>
<td>I</td>
<td>52</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>R</td>
<td>35</td>
<td>j</td>
<td>53</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table showing the Base64 Alphabet.

The output stream (encoded bytes) must be represented in lines of no more than 76 characters each. All line breaks or other characters not found in Table 1 must be ignored by decoding software. In Base64 data, characters other than those in Table 1, line breaks, and other white space probably indicate a transmission error, about which a warning message or even a message rejection might be appropriate under some circumstances.
Special processing is performed if fewer than 24 bits are available at the end of the data being encoded. A full encoding quantum is always completed at the end of a body. When fewer than 24 input bits are available in an input group, zero bits are added (on the right) to form an integral number of 6-bit groups. Padding at the end of the data is performed using the '=' character. Since all Base64 input is an integral number of octets, only the following cases can arise:

1. The final quantum of encoding input is an integral multiple of 24 bits; here, the final unit of encoded output will be an integral multiple of 4 characters with no "=" padding,

2. The final quantum of encoding input is exactly 8 bits; here, the final unit of encoded output will be two characters followed by two "=" padding characters, or

3. The final quantum of encoding input is exactly 16 bits; here, the final unit of encoded output will be three characters followed by one "=" padding character.

Because it is used only for padding at the end of the data, the occurrence of any '=' characters may be taken as evidence that the end of the data has been reached (without truncation in transit). No such assurance is possible, however, when the number of octets transmitted was a multiple of three.

Any characters outside of the Base64 alphabet are to be ignored in Base64-encoded data. The same applies to any illegal sequence of characters in the Base64 encoding, such as "=====".

7.6 The Base64 object:

The Base64 class TB64 is implemented in the unit Base64.pas. This object implements Decoding of Base64 string either read from a file or assigned from a memory string, and the encoding of a memory stream in to a string. Now we will show the object Methods and properties:
P = Property, M = Method.

<table>
<thead>
<tr>
<th>P/M</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>A : String</td>
<td>The Base64 encoded string to be decoded. Can be assigned directly, and the result of the encoding is assigned to A.</td>
</tr>
<tr>
<td>M</td>
<td>Procedure GetText(FileName : string);</td>
<td>A procedure that gives the user the option of loading the Base64 encoded data into A from a file named FileName..</td>
</tr>
<tr>
<td>M</td>
<td>Function Encode(B : TmemoryStream): Integer</td>
<td>returns the number of bytes encoded from B, and stores the encoded string in A.</td>
</tr>
<tr>
<td>M</td>
<td>Function Decode(Var B: TmemoryStream): boolean;</td>
<td>Decodes A into B and returns True if it succeeds and False if it fails.</td>
</tr>
</tbody>
</table>

This object is used by X509 object to load a certificate from a file.

### 7.7 How to use Base64 object:

1- Include Base64 to the uses clause.
2- Make a variable (Say B64) of type TB64.
3- Create B64 by typing B64 := TB64.Create;
4- Load the Base64 encoded file into A using B64.GetText(FileName);
5- Decode A by into buf:TmemoryStream by doing the folwoing

```pascal
buf := TmemoryStream.Create;
if B64.Decode(buf) then
    //use buf
else
    //give an error message
```
8
Using The SSL Package
8.1 Introduction

The package is presented as a component in Delphi™ 4.0. We are going to show how to use it in any application that needs an SSL client. The following diagram describes how to use this component.

![Diagram showing the SSL handshake process]

8.2 Creating the Package

You can create the package by two ways:

1- Drag and Drop the component into the application form, during design time only.
2- Create the component at design time using the method create.

8.3 Initializing the Package:

After creating the package it will be automatically initialized to certain standard that provide maximum available security. User can change these settings by calling `InitSock` procedure with the appropriate parameters. Its prototype is:

```pascal
Procedure InitSock(Init: ConnectionInit);
```

The structure of `ConnectionInit` is a record that takes all required initialization settings. Its structure is:

```pascal
TSSLConnectionInit = Record
    Resumed : Boolean;
    SSLVersion : TSSLVersion;
    ConnectionEnd : TConnectionEnd;
    CipherSuites : TCipherSuites;
    Compression : TCompressionMethods;
    Certificate : X509;
    PrivateKey,
    PrivateKeyExport : TRSAKey;
    RequestClientCert : TRequestClientCert;
end;
```
<table>
<thead>
<tr>
<th><strong>Resume</strong></th>
<th>True if Client want to resume previous connection else False</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSLVersion</strong></td>
<td>Type of SSL Version to be used. ( ver 3.0 in this Package)</td>
</tr>
<tr>
<td><strong>ConnectionEnd</strong></td>
<td>The Side of the SSL protocol. (Client side in this Package)</td>
</tr>
<tr>
<td><strong>Cipher Suites</strong></td>
<td>Array that contain the cipher suites acceptable by the user</td>
</tr>
<tr>
<td><strong>Compression</strong></td>
<td>Array that contain the Compression Methods acceptable by the user</td>
</tr>
<tr>
<td><strong>Certificate</strong></td>
<td>The certificate of the user. If there is no certificate then its nil</td>
</tr>
<tr>
<td><strong>PrivateKey</strong></td>
<td>The private key for the user (used in server side only). No use for this field now.</td>
</tr>
<tr>
<td><strong>PrivateKeyExport</strong></td>
<td>Used as private key if the original key is greater than 512-bit</td>
</tr>
<tr>
<td><strong>RequestClientCert</strong></td>
<td>Define if the SSL server needs a client certificate or not. (not used in this package now.</td>
</tr>
</tbody>
</table>

### 8.4 Connecting

The client can connect to a server by specifying its DNS or IP in Host property and the port in Port property. After that the client choose one of the following:

1. Connecting without handshake.
2. Connecting and performing handshake.
3. Connect through proxy server and perform handshake.

#### 8.4.1 Connecting without handshake

The client can connect to the server without using the SSL protocol. This feature is helpful for applications that don't want to immediately begin a secure connection. They want first to exchange non-secret data and after that they may upgrade their connection to be secure using SSL.

The client can use this by calling procedure Connect. Its prototype is:

Procedure Connect;

#### 8.4.2 Connecting and performing Handshake

The client in this case wants an immediate secure connection. So after connecting, the handshake will immediately take the place until the secure connection
is established. To open this connection the client calls SSLConnect procedure. This procedure will establish the connection and perform the handshake. Its prototype is:
Procedure SSLConnect;

8.4.3 Connect through proxy server and perform handshake

In some cases the client is behind a proxy server. In this case the SSL protocol needs to be tunneled through this proxy. The SSL tunneling protocol is described in Appendix A. The client in this case must first initialize the Proxy and ProxyPort properties with proxy name and proxy port, respectively. Second, the client calls SSLProxyConnect procedure to establish the connection through proxy and perform handshake. Its prototype is:
procedure SSLProxyConnect;

8.5 Handshake

There are two cases where the client needs to initiate the handshake explicitly:

1. Upgrading the connection to SSL connection.
2. Resuming a session.

8.5.1 Upgrading the connection to SSL connection

When the client and the server agree to upgrade their connection to SSL connection and the server is ready the client call InitHandshake procedure to perform the handshake. After finishing, the connection is now upgraded to SSL connection. The prototype of the procedure is:
Procedure InitHandshake;

8.5.2 Resuming a session

In this case the client wants to resume a previously established session with the server. Reconnecting with the server needs the previous session information to establish the SSL connection - if it's not already there. This done by calling SSLResumeHandshake procedure. Its prototype is:

Procedure SSLResumeHandshake[(Previous : TSSLClientSock)];
The parameter is optionally. If the session is already initialized don't call it with the parameter.

## 8.6 Exchanging Data

When reaching this state the client and server can know exchange their secret information securely. Two kind of procedures used for this purpose:

1. Receiving procedures.
2. Sending procedures.

### 8.6.1 Receiving

A set of receiving procedures has been implemented for adding more flexibility to the package. The best way to list them is by providing their prototype:

```plaintext
Procedure SSLReadStream(Size: Integer): TmemoryStream;
Procedure SSLReadString(Size: Integer): String;
```

If $Size = 0$, the function return immediately with the available Data in the buffer. Else the function return the number of bytes. If the available data is less than $size$ the function will block until the remaining bytes received

### 8.6.2 Sending

A set of sending procedures has been implemented for adding more flexibility to the package. The best way to list them is by providing their prototype:

```plaintext
Procedure SSLSendStream(Fragment: TmemoryStream);
Procedure SSLSendString(Fragment: String);
```

The client is applicable to use anyone of these procedures to send a data to the server.

## 9.7 Closing the Connection

After exchanging data is finished the client need to close the connection. Closing the connection is done by calling SSLClose procedure. Its prototype is:

```plaintext
Procedure SSLClose;
```
9
SSL Gateway (SSLG)
9.1 **SSL Gateway**

SSL Gateway (SSLG) is a program that works as a server for the user application and as SSL client for the SSL Server. In another way, it stays between the user application that wants to use a secure connection and the server that user application wants to connect to.

This SSLG is useful when the user doesn't want to use the SSL Component in his application or the application has been developed and can't be modified.

9.2 **How SSL Gateway Works**

SSLG accept TCP connection from the user application on one side and from the other side it initiate SSL Connection with the SSL server. After that, all data that comes from the user application will be send to server using the pre-established SSL Connection with it. Also all data received from the SSL server will be decrypted, verified and send to the user application.

9.3 **How to use SSL Gateway**

SSLG must be setup before it's used. The setup steps are as follow: (see figure 6)

A) Application Side:

1) Specify the port that will accept user application connections.

2) Specify which protocol will the user application use, so that the SSLG can get the host name by parsing the incoming messages. If its unknown protocol or its not supported by SSLG then the user must specify a standard host for the SSL Connection.

b) SSL Side:

1) If the connection of SSL will be made through proxy then the user must specify the proxy name and port. If the proxy need an authentication then user specifies the User ID and Password.

2) If the user application protocol is not known or not supported by SSLG then the user must specify a standard host for the session.
Now, after doing these steps the user need only to press the "START SSLG" to turn on the SSL Gateway.

9.4: Security Analysis of SSLG

The main purpose of this application is to provide a secure connection for applications that do not implement SSL. However, to provide the user with full security, the two applications SSLG and user application must be in the same computer. If they are in two different computers then the connection path between the SSLG and the user application must be trusted to ensure full security, because the data between the two is exchanged without any encryption.
SSL Tunneling
Tunneling SSL Through a WWW Proxy

Status of this Memo
This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

To learn the current status of any Internet-Draft, please check the `1id-abstracts.txt` listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

Table of Contents
- Overview
- General Considerations
- The CONNECT Method
- Proxy Response
- Security Considerations
- Extensibility

Overview
The wide success of SSL (Secure Sockets Layer from Netscape Communications Corporation) has made it vital that the current WWW proxy protocol be extended to allow an SSL client to open a secure tunnel through the proxy.

The HTTPS protocol, which is just HTTP on top of SSL, can be proxied in the same way that currently other protocols are handled by the proxies: to have the proxy initiate a secure session with the remote HTTPS server, and then perform the HTTPS transaction. This is the way FTP and Gopher get handled by proxies, too. However, this approach has two disadvantages:
The connection between the client and the proxy is normal HTTP, and hence, not secure. This is, however, often acceptable if the clients are in a trusted subnet (behind a firewall).

The proxy will need to have full SSL implementation incorporated into it. This document describes a way to do SSL tunneling without an implementation of SSL, in a way that is compatible with the current WWW proxy protocol (as defined by the HTTP/1.0 specification). The existing implementation of SSL tunneling in the Netscape Navigator and the Netscape Proxy Server conform to this specification. A patch implementing this feature also exists for the CERN proxy server (CERN httpd).

General Considerations

When tunneling SSL, the proxy must not have access to the data being transferred in either direction, for sake of security. The proxy merely knows the source and destination addresses, and possibly, if the proxy supports user authentication, the name of the requesting user.

In other words, there is a handshake between the client and the proxy to establish the connection between the client and the remote server through the proxy. In order to make this extension be backwards compatible, the handshake must be in the same format as HTTP/1.0 requests, so that proxies without support for this feature can still cleanly determine the request as impossible for them to service, and give proper error responses (rather than e.g. get hung on the connection).

SSL tunneling isn't really SSL specific -- it's a general way to have a third party establish a connection between two points, after which bytes are simply copied back and forth by this intermediary.

When SSL tunneling support is put into the SSL source code, it will work for any SSL enhanced application.

The CONNECT Method

The client connects to the proxy, and uses the CONNECT method to specify the hostname and the port number to connect to. The hostname and port number are separated by a colon, and both of them must be specified.

The host:port part is followed by a space and a string specifying the HTTP version number, e.g. HTTP/1.0, and the line terminator (CR LF pair, or a single LF).
After that there is a series of zero or more of HTTP request header lines, followed by an empty line. Each of those header lines is also terminated by the CR LF pair, or a single LF. The empty line is simply another CR LF pair, or another LF.

After the empty line, if the handshake to establish the connection was successful, SSL data transfer can begin.

Example:

CONNECT home.netscape.com:443 HTTP/1.0
User-agent: Mozilla/1.1

The advantage of this approach is that this protocol is freely extensible; for example, the proxy authentication can be used.

Example:

CONNECT home.netscape.com:443 HTTP/1.0
User-agent: Mozilla/1.1
Proxy-authorization: basic aGVsbG86d29ybGQ=

Proxy Response

After the empty line in the request, the client will wait for a response from the proxy. The proxy will evaluate the request, make sure that it is valid, and that the user is authorized to request such a connection. If everything is in order, the proxy will make a connection to the destination server, and, if successful, send a "200 Connection established" response to the client. Again, the response follows the HTTP/1.0 protocol, so the response line starts with the protocol version specifier, and the response line is followed by zero or more response headers, followed by an empty line. The line separator is CR LF pair, or a single LF.

Example:

HTTP/1.0 200 Connection established
Proxy-agent: Netscape-Proxy/1.1

After the empty line, the proxy will start passing data from the client connection to the remote server connection, and vice versa. At any time, there may be data coming from either connection, and that data should be forwarded to the other connection immediately.
If at any point either one of the peers gets disconnected, any outstanding data that came from that peer will be passed to the other one, and after that also the other connection will be terminated by the proxy. If there is outstanding data to that peer undelivered, that data will be discarded.

Security Considerations

CONNECT is really a lower-level function than the rest of the HTTP methods, kind of an escape mechanism for saying that the proxy should not interfere with the transaction, but merely forward the data. This is because the proxy should not need to know the entire URI that is being accessed (privacy, security), only the information that it explicitly needs (hostname and port number). Due to this fact, the proxy cannot verify that the protocol being spoken is really SSL, and so the proxy configuration should explicitly limit allowed connections to well-known SSL ports (such as 443 for HTTPS, 563 for SNEWS, as assigned by the Internet Assigned Numbers Authority).

Extensibility

The SSL tunneling handshake is freely extensible using the HTTP/1.0 headers; as an example, to enforce authentication for the proxy the proxy will simply use the 407 status code and the Proxy-authenticate response header (as defined by the HTTP/1.0 specification) to ask the client to send authentication information:

HTTP/1.0 407 Proxy authentication required
Proxy-authenticate: ...

The client would then send the authentication information in the Proxy-authorization header:

CONNECT home.netscape.com:443 HTTP/1.0
User-agent: ...
Proxy-authorization: ...

Multiple Proxy Servers

This specification applies also to proxy servers talking to other proxy servers. As an example, double firewalls make this necessary. In this case, the inner proxy is simply considered a client with respect to the outer proxy.
SSL Presentation Language
**B.1 Basic block size**

The representation of all data items is explicitly specified. The basic data block size is one byte (i.e. 8 bits). Multiple byte data items are concatenations of bytes, from left to right, from top to bottom. From the bytestream a multi-byte item (a numeric in the example) is formed (using C notation) by:

\[
\text{value} = (\text{byte}[0] << 8*(n-1)) | (\text{byte}[1] << 8*(n-2)) | ... | \text{byte}[n-1];
\]

This byte ordering for multi-byte values is the commonplace network byte order or big endian format.

**B.2 Miscellaneous**

Comments begin with "/*" and end with "*/". Optional components are denoted by enclosing them in "[ [ ] ]" double brackets. Single byte entities containing uninterpreted data are of type opaque.

**B.3 Vectors**

A vector (single dimensioned array) is a stream of homogeneous data elements. The size of the vector may be specified at documentation time or left unspecified until runtime. In either case the length declares the number of bytes, not the number of elements, in the vector. The syntax for specifying a new type \( T' \) that is a fixed length vector of type \( T \) is

\[ T \ T'[n]; \]

Here \( T' \) occupies \( n \) bytes in the data stream, where \( n \) is a multiple of the size of \( T \). The length of the vector is not included in the encoded stream.

In the following example, Datum is defined to be three consecutive bytes that the protocol does not interpret, while Data is three consecutive Datum, consuming a total of nine bytes.
opaque Datum[3]; /* three uninterpreted bytes */
Datum Data[9]; /* 3 consecutive 3 byte vectors */

Variable length vectors are defined by specifying a subrange of legal lengths, inclusively, using the notation \(<floor..ceiling>\). When encoded, the actual length precedes the vector's contents in the byte stream. The length will be in the form of a number consuming as many bytes as required to hold the vector's specified maximum (ceiling) length. A variable length vector with an actual length field of zero is referred to as an empty vector.

\[ T T' <floor..ceiling>; \]

In the following example, mandatory is a vector that must contain between 300 and 400 bytes of type opaque. It can never be empty. The actual length field consumes two bytes, a uint16, sufficient to represent the value 400 (see Section 4.4). On the other hand, longer can represent up to 800 bytes of data, or 400 uint16 elements, and it may be empty. Its encoding will include a two byte actual length field prepended to the vector.

opaque mandatory<300..400>;
   /* length field is 2 bytes, cannot be empty */
uint16 longer<0..800>;
   /* zero to 400 16-bit unsigned integers */

### B.4 Numbers

The basic numeric data type is an unsigned byte (uint8). All larger numeric data types are formed from fixed length series of bytes concatenated as described in Section 4.1 and are also unsigned. The following numeric types are predefined.

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```
B.5 Enumerateds

An additional sparse data type is available called enum. A field of type enum can only assume the values declared in the definition. Each definition is a different type. Only enumerateds of the same type may be assigned or compared. Every element of an enumerated must be assigned a value, as demonstrated in the following example. Since the elements of the enumerated are not ordered, they can be assigned any unique value, in any order.

```
enum { e1(v1), e2(v2), ... , en(vn), [[(n)]] } Te;
```

Enumerateds occupy as much space in the byte stream as would its maximal defined ordinal value. The following definition would cause one byte to be used to carry fields of type Color.

```
enum { red(3), blue(5), white(7) } Color;
```

One may optionally specify a value without its associated tag to force the width definition without defining a superfluous element. In the following example, Taste will consume two bytes in the data stream but can only assume the values 1, 2 or 4.

```
enum { sweet(1), sour(2), bitter(4), (32000) } Taste;
```

The names of the elements of an enumeration are scoped within the defined type. In the first example, a fully qualified reference to the second element of the enumeration would be Color.blue. Such qualification is not required if the target of the assignment is well specified.

```
Color color = Color.blue;   /* overspecified, legal */
Color color = blue;         /* correct, type implicit */
```

For enumerateds that are never converted to external representation, the numerical information may be omitted.
enum { low, medium, high } Amount;

B.6 Constructed types

Structure types may be constructed from primitive types for convenience. Each specification declares a new, unique type. The syntax for definition is much like that of C.

struct {
    T1 f1; T2 f2;
    ...
    Tn fn; } [[T]];

The fields within a structure may be qualified using the type's name using a syntax much like that available for enumerateds. For example, T.f2 refers to the second field of the previous declaration. Structure definitions may be embedded.

B.6.1 Variants

Defined structures may have variants based on some knowledge that is available within the environment. The selector must be an enumerated type that defines the possible variants the structure defines. There must be a case arm for every element of the enumeration declared in the select. The body of the variant structure may be given a label for reference. The mechanism by which the variant is selected at runtime is not prescribed by the presentation language.

struct {
    T1 f1;
    T2 f2;
    ....
    Tn fn;
    select (E) {
        case e1: Te1;
        case e2: Te2;
        ....
        case en: Ten;
    } [[fv]];
} [[Tv]];
For example

```c
enum { apple, orange } VariantTag;
struct {
    uint16 number;
    opaque string<0..10>; /* variable length */
} V1;

struct {
    uint32 number;
    opaque string[10];    /* fixed length */
} V2;
struct {
    select (VariantTag) { /* value of selector is implicit */
        case apple: V1;   /* VariantBody, tag = apple */
        case orange: V2;  /* VariantBody, tag = orange */
    } variant_body;       /* optional label on variant */ } VariantRecord;
```

Variant structures may be qualified (narrowed) by specifying a value for the selector prior to the type. For example, a

```c
orange VariantRecord
```

is a narrowed type of a VariantRecord containing a variant_body of type V2.

### B.7 Constants

Typed constants can be defined for purposes of specification by declaring a symbol of the desired type and assigning values to it. Under-specified types (opaque, variable length vectors, and structures that contain opaque) cannot be assigned values. No fields of a multi-element structure or vector may be elided.

For example,

```c
struct {
    uint8 f1;
    uint8 f2;
} Example1;
```

```c
Example1 ex1 = {1, 4}; /* assigns f1 = 1, f2 = 4 */
```
References

    PKCS#1: RSA Cryptography Standard ver 1.3 and ver 2.0