ENHANCING THE IKE PRESHARED KEY AUTHENTICATION METHOD

A Dissertation
presented to
the Faculty of the Graduate School
University of Missouri-Columbia

In Partial Fulfillment
Of the Requirements for the Degree

Doctor of Philosophy

---

by
RAED M. BANI-HANI

Dr. Gordon Springer, Dissertation Supervisor

DECEMBER 2006

The undersigned, appointed by the Dean of the Graduate School, have examined the thesis entitled

ENHANCING THE IKE PRESHARED KEY

AUTHENTICATION METHOD

presented by Raed Bani-Hani

a candidate for the degree of Doctor of Philosophy

and hereby certify that in their opinion it is worthy of acceptance.

_____

Dr. Gordon Springer

_____

Dr. Ye Duan

_____

Dr. Michael Jurczyk

_____

Dr. Hongchi Shi

_____

Dr. Harry Tyrer

To my mother, may her soul rest in peace, who passed away while I was pursuing my PhD degree.

ACKNOWLEDGEMENTS

The work for this dissertation started at the beginning of the year 2004. The first year was utilized to gather background information and experience of the IPsec and IKE protocols, followed by implementing a replacement module for the IKE protocol to remedy its weakness and enhance its security level. The writing for this thesis was completed during the first six months of 2006.

I have left the acknowledgments section of this dissertation unwritten until the very last moment so that I don't forget many people who supported me during these last three, very intensive years of my life.

All thanks are due to Allah (God) who facilitated for me all the means to finish this stage of my life. He granted me a very supportive wife and family whom I can not thank enough. The completion of this dissertation would not have been possible without their assistance and support.

I would like to thank Dr. Gordon Springer, my advisor, for the invaluable comments and input on the content of this dissertation. I would like also thank Mr. Larry Sanders for his assistance in setting up the testing environment of this dissertation. He was very patient and cooperative every time I jumped into his office. I'm also grateful to my committee members. They not only gave me the resources, help and guidance I needed but, in addition, they pushed me to discover and surpass my own limits.

Finally, I would like to thank all friends and families in Columbia, Missouri for supporting and encouraging me through out my stay here. Special thanks are due to Khenissi's daughters for proofreading the thesis. May Allah reward them the best.

This work is the result of a very long trip that has just started, and I hope it will not be as difficult.

Contents

Appendices

List of Figures

List of Tables

ABSTRACT

Over the past decade, the Internet has grown from a small, restricted network to a big, global network connecting people and organizations all over the world. People are using the Internet for on–line banking, e–commerce, and private communications. Furthermore, large organizations are using the Internet to provide many services to its users and to exchange sensitive data with other businesses and organizations.

With the growing amount of sensitive and valuable amount of information that is transferred across the Internet continuously, protection from unauthorized access has become a major concern. It is necessary to secure information exchanged against security threats such as loss of privacy, loss of data integrity, or identity theft. This necessity leads to the design of security protocols. One of the modern security protocols is the IP security (IPsec) protocol. Actually, it is a set of protocols and standards that provides the security services of data confidentiality, integrity, and authenticity for an Internet User.

IPsec depends on another protocol to establish shared keys before it can apply any security service. This protocol is called the Internet Key Exchange (IKE) protocol. In addition to generating and maintaining shared secrets, IKE is responsible for authenticating the parties that would like to use the IPsec services. One of authentication methods is the preshared key authentication method. In this dissertation, we explain this method, show how the keys negotiated through this method can be compromised, and propose an improvement to make the method more secure.

Chapter 1

Introduction

With its explosive growth and popularity, millions of people are using the Internet worldwide to exchange data and to communicate with each other. This communication can be by exchanging email messages, instant messages, sensitive data, such as bank accounts or credit card numbers, or simply by reading data from the web. Such communications, most of the time, are not secure. The lack of secure communication might cause a problem if someone other than the communicating parties can gain access to the data exchange, especially sensitive data. Therefore, data exchange between communicating parties needs to be secured in order to avoid this problem

Security is needed to provide users with assurance against several types of attacks. One such attack is the loss of privacy, or sniffing. In this attack an eavesdropper, who is not supposed to learn the content of any communication, is able to observe confidential data as it traverses the Internet. Therefore, without encryption, every message sent might be exposed to the eavesdropper. Another attack is the loss of data integrity. In this attack the content of a message is modified by an attacker to produce an unauthorized effect. For example, a message says ”deposit 1000 dollars in Alice’s account” could be changed to ”deposit 1000 dollars in Bob’s account”. Consequently, it is very important when a message is received to make sure that it has not been modified or altered in transit to

insure proper results. One more type of attack is identity spoofing, in which one entity impersonates or pretends to be a different entity so that it might have access to confidential data. For this reason the receiving entity, before processing any message, must make sure that the message is from the source that it claims to be. In another way, the receiver must authenticate messages.

There are many other types of attacks. However, the ones mentioned above are enough to show that there is a great risk in using the Internet without any type of protection against the threats outlined previously. In response to these threats, the Internet Engineering Task Force (IETF) has developed the IP Security (IPsec) protocol suite. It is a set of protocols and standards that provide the basic services of data confidentiality, integrity, and authenticity for an Internet user.

The Transport Layer Security (TLS) protocol [1] is another potential solution for the security threats. It enables secure data transfer between two devices over a public network. TLS protects applications running over the Transmission Control Protocol (TCP) [2], and is mostly utilized to protect the Hyper Transfer Text Protocol (HTTP) [3] communications. TLS provides authentication, data integrity, and confidentiality through the use of different cryptographic algorithms. The two primary functions of the TLS protocol is to provide privacy between a client and a server and to authenticate the server to the client. The IETF has standardized the use of TLS over HTTP. Request For Comments (RFC) 2818 describes how to use TLS to secure HTTP connections [4]. However, TLS can not protect applications that are using a transport layer other than the TCP. Unlike TLS, IPsec supports the User Datagram Protocol (UDP) [5] and is not limited to use TCP only. In fact, a major advantage of IPsec is that it can protect all traffic regardless of transport protocols because it offers protection at the IP layer. Moreover, IPsec allows Gateway–to–Gateway communication to protect traffic on behalf of users using the

gateways. Therefore, IPsec provides a security solution that is not dependent on the application or the transport layer on which it runs. Chapter 3 provides more features of the IPsec protocol.

A secure shared key must be established and agreed upon before any security service of IPsec can be applied. The security of these services depends on the security of the shared key itself and the method used to establish it. As a result, it is essential that this key be established securely. Otherwise, all messages and information protected by this key could be compromised. This is exactly what Internet Key Exchange (IKE) [6] does: establish a secure key for the use of IPsec services.

In addition to establishing a secret key between the communicating parties, IKE incorporates an authentication process to make sure parties that would like to communicate securely are actually what they claim to be. Authentication is accomplished through exchanging some type of identity data during the key establishment. One method of authentication is achieved using a preshared key. A preshared key is simply a string of characters that is agreed upon ahead of time between the communicating parties.

In this dissertation, we will show that the preshared key authentication method is insecure, and the means by which an attacker could compromise the shared key will be outlined. Once the shared key is compromised, all messages secured by that key could be compromised.

In the next chapter, a basic introduction to the subject of cryptography is provided. Chapter 3 gives a brief overview of IPsec and its operation. In Chapter 4, IPsec security protocols are explained. Establishing a shared key and the details of the IKE protocol are presented in Chapter 5. Chapter 6 illustrates the weakness of the preshared key authentication method and demonstrates the way through which a key could be compromised. Modifying the IKE preshared key authentication method to overcome its weakness and to

elevate its security level is presented in Chapter 7. Chapter 8 details the design and implementation process of a replacement module for the IKE phase one communication that implements the proposed enhancement. Finally, the results and analysis of the proposed enhancement are provided in Chapter 9.

Chapter 2

Cryptography Basics

2.1 Introduction

There are many aspects to security and many applications, ranging from secure commerce and payments to private communications and protecting passwords. One essential aspect for secure communications is that of *cryptography*, the focus of this introductory chapter. This chapter aims to provide two major goals. The first is to define some of the terms and concepts used in cryptographic methods, which help the reader to understand the operation of the IP Security (IPsec) protocol (see Chapter 3). The second is to introduce the reader to some algorithms and examples in use today.

In many of the descriptions below, two communicating parties will be referred to as Alice and Bob; this is the common nomenclature in the cryptography field and literature to make it easier to identify the communicating parties. If there is a third or fourth party to the communication, they will be referred to as Carol and Dave. Mallory is a malicious party, Eve is an eavesdropper, and Trent is a trusted third party [7].

2.2 What is Cryptography

Cryptography is the science of keeping a message confidential through the processes of encryption and decryption. A message is simply any unaltered data, called *plaintext*, which can take any form such as text, image, or video files. Encryption is the process of taking a plaintext message M and transforming it to a *ciphertext* C that cannot be easily viewed or understood by unintended parties. The main benefit of the encryption process is that it insures confidentiality and privacy by hiding data content from anyone for whom it is not intended, even those who have access to the encrypted data or ciphertext.

Decryption is the opposite of encryption. It is performed at the receiving end of a communication by converting the ciphertext C back into the plaintext message M. During the encryption and the decryption processes, two parties, the sender and the receiver, must have some shared data in place. This data is called a key K. Keys are typically very large numbers with particular mathematical properties so that when they are used in the encryption process they form a ciphertext that is difficult to decrypt without knowing these keys. Keys are independent of the plaintext and control the output of the algorithm used. The algorithm produces a different output depending on the specific key K being used at the time. Changing the key changes the output of the algorithm.

In addition to *data confidentiality*, cryptography provides three main types of services to insure proper treatment of data that is transmitted or stored. These services are *authentication*, *data integrity*, and *nonrepudiation*. Authentication assures the recipient that the message received is from the source that it claims to be from. This insures that an intruder is not able to masquerade as someone else. Data integrity verifies that the messages are received exactly as sent, with no modification or insertion. As a result, an intruder is not able to substitute a false message for a legitimate one. Nonrepudiation prevents an entity, a sender or a receiver, from denying a transmitted message that was authenticated as

coming from it. Therefore, when a message is sent, either entity can prove that the other entity in fact sent the message.

2.3 Types of Cryptographic Systems

Two forms of cryptography systems are common in use: *symmetric*, in which the processes of encryption and decryption are performed using the same key, and *asymmetric*, in which each process is performed using different keys.

2.3.1 Symmetric Key Cryptography

Symmetric key cryptography, sometimes called conventional cryptography, uses a single shared key for encryption and decryption. This method requires the two communicating parties to agree on a key before they can communicate securely. This key is an essential piece of information. It is called a shared or a secret key. To perform encryption, the encryption algorithm takes as an input the plaintext message, M, along with the shared key, K, to produce a ciphertext, C, that is sent to the other party. In order for the other party to bring back the ciphertext C to the original plaintext message M, it uses a decryption algorithm with the same shared key K. If the decryption algorithm uses a different key than the one used in encryption, the message produced after decryption will not match the original message sent. This indicates the importance of using the same shared key for the success of the encryption and the decryption processes. Moreover, it indicates the importance of keeping that key known only to the parties involved in the secure communicating. Otherwise, the content of any message encrypted using the shared key can be revealed to any one who knows the key and leads to losing the privacy of the communication. As a result, all of the security of symmetric key cryptography is based upon the key and its remaining secret. It depends on the secrecy of the key not the secrecy of the algorithm

used. This means that the algorithms can be published and analyzed without compromising any communication that uses the algorithm so long as the key employed remains private. Some known names of this type of cryptography is “shared key“ cryptography sine the sender and receiver must share the same key, or ”secret key“ cryptography since the key must be only known to the sender and receiver to maintain integrity.

2.3.1.1 Algorithms

There are many commonly used symmetric key algorithms today, most of which are based on block ciphers. A *block cipher* processes the plaintext input in fixed–sized blocks and produces a block of ciphertext of equal size for each plaintext block. Having the ciphertext block size the same as the plaintext block size is important since there is no data expansion with encryption. Three of the most important symmetric key block ciphers are *Data Encryption Standard* (DES), *Triple DES* (3DES), and *Advanced Encryption Standard* (AES).

DES was adopted as a Federal Information Processing Standard publication (FIPS PUB) in 1977 [8]. The official description of the standard can be found in [9]. DES encrypts and decrypts data in 64–bit blocks using a 56–bit secret key. It takes a 64–bit block of plaintext input and outputs a 64–bit block of ciphertext. Longer plaintext amounts are processed in 64–bit blocks. It always operates on blocks of equal size. In the case that the plaintext message does not align evenly on a 64–bit boundary, padding is used. Decryption in DES is essentially the same as encryption. The ciphertext is used as an input to the DES algorithm. It is broken into 64-bit blocks and processed with the same key used during encryption. This is an excellent feature because it means that there is no need to implement two different algorithms, one for encryption and one for decryption.

Security of DES can be divided into security of the algorithm itself and security (strength) of the shared key used. Over the years, there have been numerous attempts

to find and exploit weaknesses in the algorithm, making DES the most studied algorithm in existence. Despite numerous attempts, no one has succeeded, so far, in discovering a fatal weakness in the DES algorithm [8]. However, with a 56–bit key length, there are only 2^{56} possible keys, which is considered insecure nowadays. In 1998, the Electronic Frontier Foundation announced that it had broken DES encryption using a special purpose machine called "*DES Cracker*" [10]. This machine was able to try out all possible 2^{56} keys in less than three days. It cost less than $250,000 and searched over 88 billion bytes per second. As a result, and knowing that hardware prices will continue to drop as speed increases, DES is no longer considered to be secure.

Triple DES (3DES) was developed in response to the weakness of DES. It was standardized in 1999 with the publication of FIPS 46-3 [9]. Basically, it is based on the DES algorithm with the same 64–bit block size. However, it uses three 56–bit keys in three rounds of the DES algorithm making its actual key length of 168 bits. In the first round of operation, a block is encrypted with the first 56–bit key. In the second round, the resulting cipher block is decrypted using the second 56–bit key. This step will not produce the original plaintext message because the key used in decryption is different from the one used in encryption. Finally, the resulted block from the second round is encrypted using the last 56–bit key. The decryption process is simply the corresponding reverse operation using the appropriate keys. Thus, if B is a plaintext block, C is the corresponding ciphertext block, E is the encryption process, D is the decryption process, K_1 is the first 56–bit key, K_2 is the second 56–bit key, and K_3 is the last 56–bit key, the encryption process of 3DES can be summarized as $C = E_{K_3}(D_{K_2}(E_{K_1}(B)))$ and the decryption process as $B = D_{K_1}(E_{K_2}(D_{K_3}(C)))$. In practice, a user inputs the entire 168–bit key (21 character) rather than entering each of the three keys individually.

Because the underlying algorithm used is DES, 3DES can show the same strength

in attacking the algorithm itself. Moreover, with a 168–bit key length, it takes much longer time to enumerate all the 2^{168} possible keys, which makes it much more difficult to apply a brute–force attack. However, because 3DES applies the DES algorithm three times, it takes three times as long as DES to complete. Therefore, 3DES is relatively slow. Besides that, it uses a block size of 64 bits which is undesirable for the reasons of efficiency and security [8]. It is clear that with larger block size we will have less number of input blocks that will take less time to process. Therefore, larger block size makes processing the blocks more efficient. Furthermore, there is a good security related reason for the larger block size. This comes from the fact that we encrypt much larger pieces of data than just one block. For this, the so called modes of operation (see the next section) are used, of which the Cipher Block Chaining mode (CBC) is the most popular. An algorithm in CBC mode breaks the input data into input blocks of equal size. Assume the input data is broken into N blocks. Then after encryption we have N ciphertext blocks. These ciphertext blocks should look like random nonsense to anyone who does not know the key.

One feature observed about CBC is that if it ever happens that two ciphertext blocks encrypted under the same key are the same, then it is easy to compute the bit–wise XOR of the two corresponding plaintext blocks [7]. Therefore, if this happens, information about the plaintext leaks. This is not due to any weakness of the underlying encryption algorithm; it is simply a consequence of the way CBC is constructed. If only a small number of blocks is encrypted, the chance of two of the ciphertext blocks being equal is negligible. However, if the number of input blocks is very large, the chance of two equal blocks increases. Thus, larger block size reduces the number of input blocks which, in turn, decreases the chance that two ciphertext blocks are the same. As a result, it is desirable for better security to have larger block size.

To overcome the drawbacks of 3DES, a new algorithm, *Advanced Encryption Standard* (AES), was standardized and published in 2001 as FIPS PUB 197 [11]. Unlike DES, which was designed specifically for hardware implementations; one of the design criteria for the AES algorithm is that it can be efficiently implemented in both hardware and software. AES uses a block size of 128 bits and supports a variable key length, in contrast to the fixed key lengths of DES and 3DES. A key can be 128 bits, 192 bits, or 256 bits. Assuming that one could build a machine that could recover a DES key in a second (i.e. try 2^{56} keys per second), it would then take that machine approximately 149 thousand billion (149 trillion) years to crack a 128-bit AES key. In addition to increased security that comes with larger key lengths, AES can encrypt data faster than 3DES [12], which makes it more suitable for future applications in terms of security and performance. More about AES can be found in [8, 11, 13].

2.3.1.2 Modes of Operation

Cryptography algorithms usually operate on blocks of fixed length. In the case of the DES and the 3DES algorithms, the block length is 64 bits. It is 128 bits for the AES algorithm. To process longer messages, several modes of operations may be used [14, 15]. Electronic Codebook (ECB) is the simplest of the modes. In this mode the messages are split into blocks and each is processed separately. See Figure 2.1 and Figure 2.2 for illustration of the encryption and decryption processes, respectively. The drawback of this mode is that if the same block appears more than one time in the input, it always produces the same ciphertext block. Therefore, for lengthy messages, the ECB mode may not be secure because it does not hide data patterns or repetitions. To overcome this limitation, cipher–block chaining (CBC) mode is introduced. In CBC mode, each block of plaintext is XORed with the preceding block ciphertext. XOR is the function known as "exclusive–or." It is a logical operator that results in true if one of its two operands, but not both of

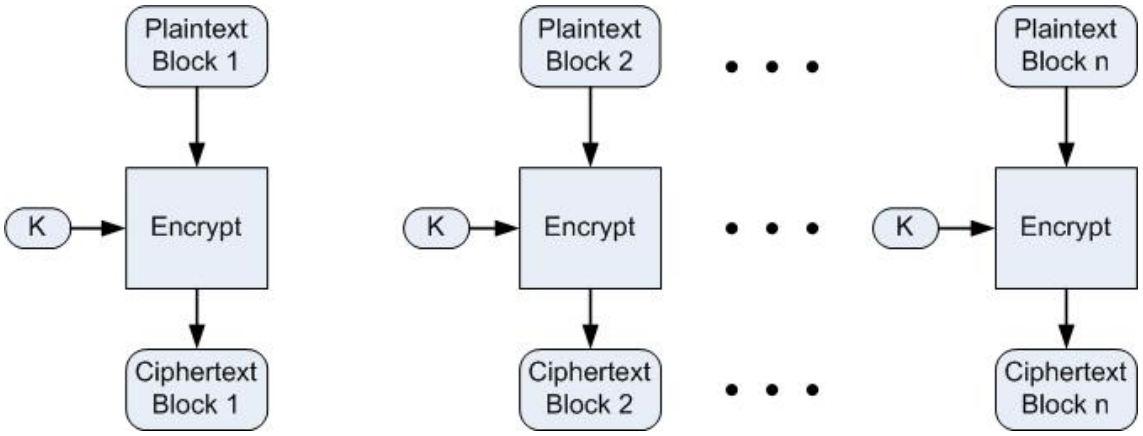

Figure 2.1: Electronic Codebook (ECB) Encryption Process

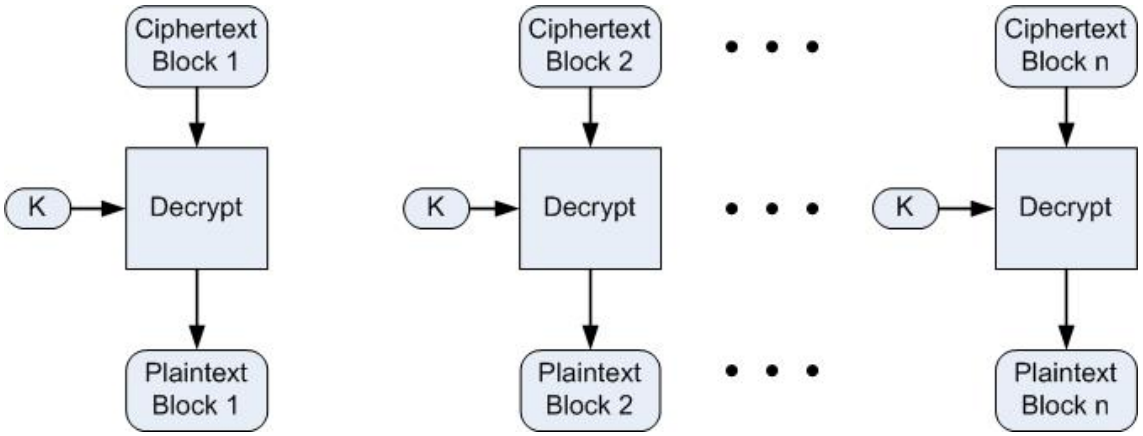

Figure 2.2: Electronic Codebook (ECB) Decryption Process

them, is true. In CBC mode, if some plaintext blocks are repeated, it produces a different ciphertext block for each of them. In other words, each ciphertext block is dependent on all plaintext up to this point. For the first ciphertext block, an initialization vector (IV) of the same block size is XORed with the first block of plaintext. On decryption, that same IV is used to recover the first plaintext block. An example showing the operation of CBC mode is shown in Figure 2.3 and Figure 2.4. CBC is the most commonly used mode of operation in security applications. For other cipher modes of operations see [7, 8, 14, 15].

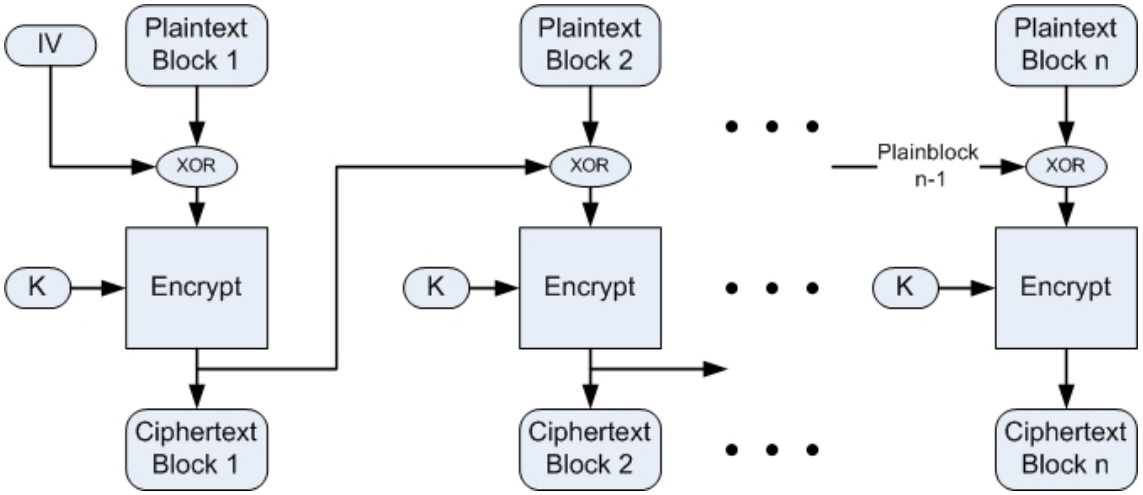

Figure 2.3: Cipher Block Chaining (CBC) Encryption Process

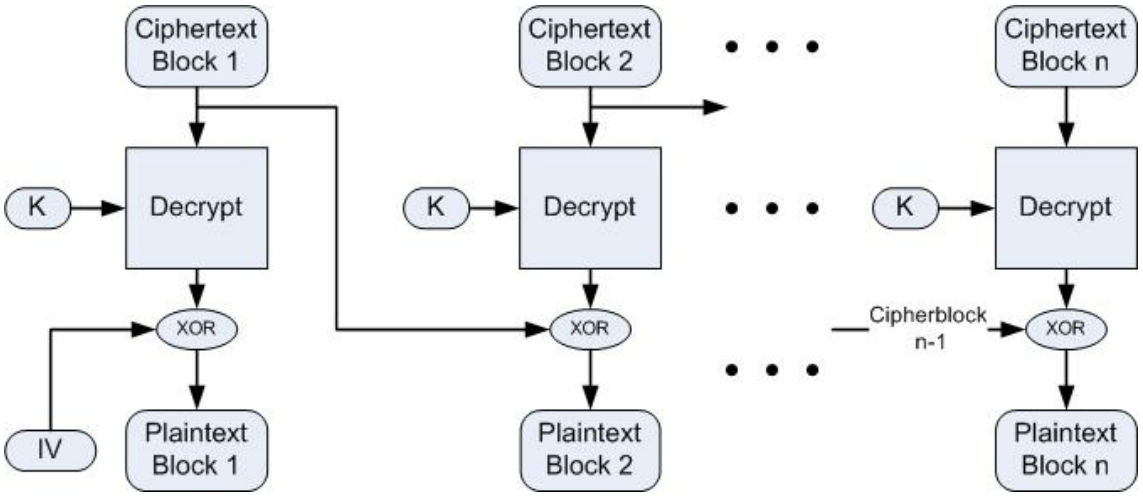

Figure 2.4: Cipher Block Chaining (CBC) Decryption Process

2.3.1.3 Authentication and Integrity

A tag, known as Message Authentication Code (MAC), is generated and appended to each message to provide authentication and integrity services. To generate this tag, it

is assumed that the two communicating parties share a common secret key. The tag is calculated as a function of the message and the shared key. Encryption algorithms, such as DES, could be used to generate this tag. The message is encrypted using an encryption algorithm together with the shared key, and the last number of bits of the ciphertext are used as the tag. Typically, a tag length is 16 or 32 bits. This approach, however, is not commonly used because it is inefficient. Each message needs to be encrypted just to produce the authentication code. An alternative to this approach is the use of one way hash functions. A hash function takes a large, variable length message M as input and produces a small, fixed length value called a hash value or a digest $H(M)$. The one way property means that starting with a hash output value, it is difficult to create a different input value that would generate the same output value. That is, it is computationally infeasible to find two messages M and M' such that $H(M) = H(M')$. Some popular hash functions are the US Secure Hash Algorithm 1 (SHA-1) [16] and the Message Digest Algorithm (MD5) [17].

To generate an authentication value of a message, a hash value is calculated over the concatenation of the message and a shared key. This method is called *Hashed Message Authentication Code* (HMAC) [18]. One main feature of this method is that it does not involve encryption. Therefore, it is more efficient since it requires less computation. Since the message together with the shared key is used as input to a hash function, this method would generate an HMAC authentication value that is not only dependent on the message, but also on the shared secret key. This insures that only a holder of the shared key could generate that authentication value. In addition to authentication, an HMAC authentication value also provides data integrity. If the message is altered during transmission in any way, the receiver will not be able to verify the received message and will discard it.

To illustrate the process of authenticating and verifying a message, see the example

in Figure 2.5. In this example, Alice uses the plaintext message M with the shared key K to calculate an HMAC authentication value and sends it together with the message M to Bob. Bob computes its own authentication value using the same shared key K and the received message. He then compares the computed value with the received authentication value. If it matches, Bob is assured that Alice knows the same shared key K, confirming the her identity. That is, Alice is authenticated. Moreover, Bob is assured that the message was not modified or altered during transmission. Otherwise, the computed authentication value would not match the received one because of the one way property of hash functions. Therefore, data integrity is satisfied.

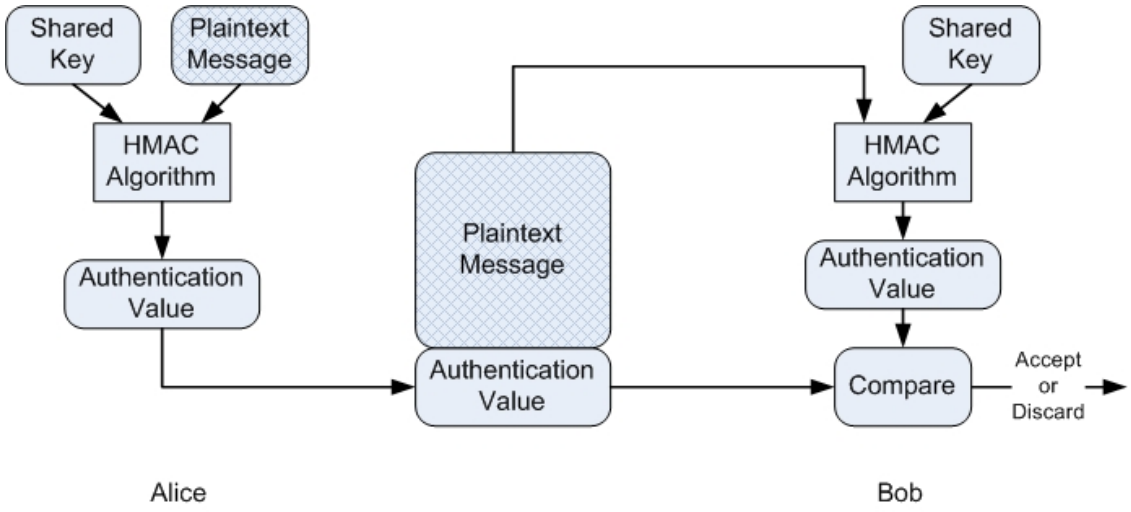

Figure 2.5: Symmetric Key Authentication and Integrity

2.3.2 Asymmetric Key Cryptography

In symmetric key algorithms, a shared key must be agreed upon in advance for a sender and a receiver to communicate securely. If they are in different physical locations, they must trust a courier or other secure communication medium to prevent the disclosure of

the secret key during the key's transmission. This indicates that key distribution is the main problem in symmetric key algorithms.

Asymmetric key cryptography, also known as public key cryptography, solves the problem of key distribution. It uses two related but different keys. One key is the private key, which is kept secret. The other key is the public key, which is published publicly and known to the outside world. The public and private keys are mathematically related. However, it is very difficult, in a reasonable amount of time, to compute one key knowing the other. Encryption is done using the public key. Therefore, any one knowing the public key, and virtually everyone can since the key is publicly available, could use it to decrypt a message and send it to the owner of the public key. On the other hand, only the one who knows the corresponding private key can decrypt and read that message. This guarantees message confidentiality during transit. The primary benefit of public key cryptography is that it eliminates the need to share a secret key in advance, and allows people who have no preexisting arrangement to exchange messages securely. Contrary to symmetric key algorithms, keys of public key algorithms are very large. A shared key is typically 128 bits, while a private (or public) key is typically 1024 bits. Despite that, both types provide comparable levels of security [8].

2.3.2.1 Algorithms

Two popular examples of public key cryptography are the RSA [8, 19] and the Diffie–Hellman [7, 20, 21] algorithms.

RSA is the best–known public key algorithm. It is named after its inventors, Rivest, Shamir, and Adleman. It offers both encryption and authentication (known as a digital signature, see section 2.3.2.2). Basically, the RSA algorithm is a block cipher in which the plaintext and ciphertext are treated as integers between 0 and $n-1$ for some integer n. The value of n depends primarily on a product of two very large primes numbers, p and q,

and it is used as a module in modulus arithmetic. Simply, a *modulus n arithmetic* means that integers are normally used with ordinary mathematical operations, such as addition, multiplication, and exponentiation, except that after each operation the result keeps only the remainder after dividing by n. In other words, to calculate X modulo Y (usually written $XmodY$), one determines the remainder after removing all multiples of Y from X. Clearly, the value $XmodY$ will be in the range from 0 to $Y - 1$.

The algorithm begins by selecting the values of p and q and computing their product n. Next the value $\phi(n) = (p - 1)(q - 1)$ is computed. This value represents the number of positive integers less than n and *relatively prime* to n [22]. A relatively prime number to n is a number that has no common divisor with n except 1. The algorithm continues by *selecting* an integer e that is relatively prime to $\phi(n)$ and less than n. Finally, another integer, d, is *calculated* such that $(de - 1)$ is divisible by $\phi(n)$. The public key is the pair (n, e), and the private key is (n, d).

To exchange secure messages using the RSA algorithm, both a sender and a receiver generate their private and public keys as described above. Then, each publishes its public key to the other so they can be used in encrypting messages exchanged. For example (see Figure 2.6), to encrypt a message M, Alice computes the ciphertext value $C = M^e modn$, where e and n are Bob's public value. To decrypt the message, Bob computes $M = C^d modn$ where d and n are Bob's private value.

Diffie–Hellman is an algorithm to enable two users (or more) to establish a shared key securely without any prior shared information. It was developed in 1976 by Diffie and Hellman [20]. The algorithm is similar to asymmetric key algorithms in that the concept of public and private keys is used. However, it is not typically used in encrypting or decrypting messages as we normally might think. Instead, it is used to securely exchange and agree on a secret key to be used in symmetric algorithms. The concept of a *primitive*

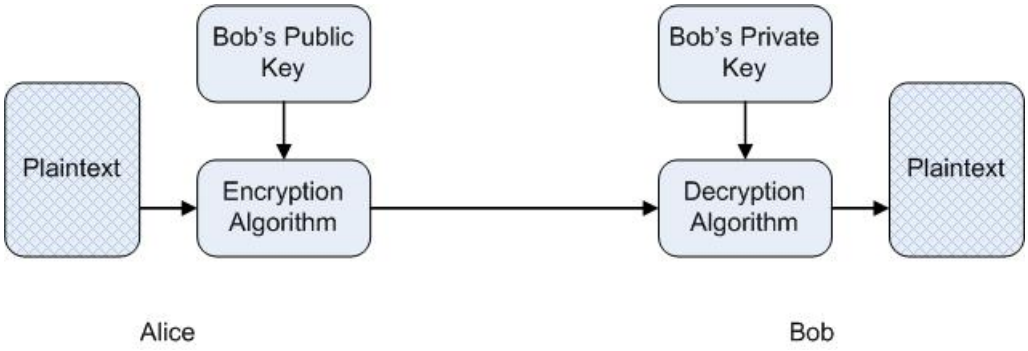

Figure 2.6: Asymmetric Key Encryption

root (also known as generator g) is used in describing the operation of the algorithm. A primitive root [8, 22] of a prime number p is one whose powers generate all the integers from 1 to p. That is, if g is a primitive root of a prime number p, then the numbers $gmodp, g^2modp, \cdots, g^{p-1}modp$ are all distinct and consist of the integers from 1 to $p-1$ in some permutation. To illustrate the algorithm, assume Alice and Bob want to establish a shared key. First they agree on two public parameters: a large number p and an integer g that is a primitive root of p. These two parameters do not need to be secret, they can be agreed upon over some insecure communication path. Next, Alice chooses a large random integer x such that $1 < x < p-1$, and Bob chooses independently a large random integer y such that $1 < y < p-1$. Then, the corresponding public values of these random integers are derived. Alice computes its public value $X = g^x modp$. Similarly, Bob computes its own public value $Y = g^y modp$. After that, both users exchange their public values while keeping the random numbers chosen previously secret or private. Finally, Alice uses her private value, x, and Bob's public value, Y , to compute the secret key . Similarly, Bob computes the secret key using his secret value, y, and Alice's public value, X. These computations produce identical keys. As a result, each side shares the same secret key with the other side after the algorithm finishes.

2.3.2.2 Authentication and Integrity

Public key encryption can be used to provide authentication and integrity of data. Instead of encrypting data using a public key, a private key is used. Data encrypted with a private key can be only decrypted with the associated public key. Therefore, providing authentication and data integrity can be accomplished by hashing data and encrypting the hash value using a private key to form what is called a *digital signature*.

To give an example, suppose that Alice wants to send a message to Bob provided that Bob is assured the message is indeed from Alice. In this case Alice uses her own private key to produce a digital signature. Alice sends the message and the associated digital signature value to Bob. At the receiving end, Bob uses Alice's public value to decrypt the message. This proves that the message was indeed encrypted with Alice's private value, and since only Alice owns the private value this insures the authenticity of the message. In addition, it is difficult to alter the message without knowing Alice's private value because this value is needed to encrypt the hash value of the modified message to make it a legitimate one. Therefore, Bob is assured that the message was not modified in transit. This insures that the data integrity is satisfied. Figure 2.7 explains the asymmetric key authentication and data integrity process.

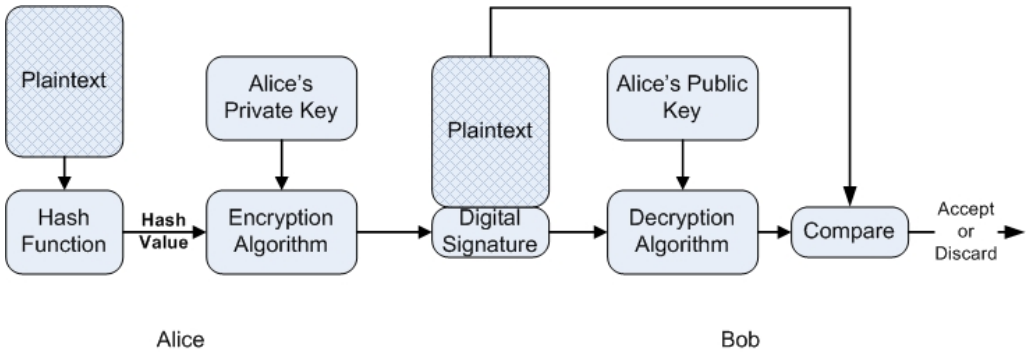

Figure 2.7: Asymmetric Key Authentication and Integrity

A well–known algorithm that provides authentication and data integrity based on asymmetric key cryptography is the Digital Signature Standard (DSS) algorithm. More information about this algorithm can be found in [23].

Due to its mathematical properties and shorter key length, symmetric key algorithms are computationally much faster than public key algorithms [22, 24]. Therefore, symmetric key algorithms are used to provide data confidentiality, while public key algorithms are mainly used for key establishment and distribution, authentication , and data integrity.

Chapter 3

IP Security

3.1 Introduction

The IP Security (IPsec) protocol suite is defined by the Internet Engineering Task Force (IETF) to provide security services for traffic at the IP layer. In general, IP packets have no inherent security, without which the packets are susceptible to unauthorized monitoring and access. For example, it is relatively easy for an eavesdropper to inspect the content of IP packets in transit, forge their address, and modify their content. As a result, IPsec was standardized as a method of protecting IP packets and ensuring secure private communication over IP networks. It provides the basic services of *confidentiality, origin authentication, and data integrity*.

3.2 IPsec Features

The fundamental strength of IPsec is that it works at the network layer level below the application layer. Therefore, just as the Internet Protocol (IP) is transparent to a user, so are the IPsec security services. This means that all the applications that use IP as the network layer protocol for transferring data can also use the security services of IPsec without any modification to these applications. For example, applications such as remote

login, email, file transfer, web access, and many others can be secured easily using IPsec.

Another feature of IPsec is that there is no need to train users for the processes of setting up the security mechanism, key creation, or key revoking since these operations are done by system administrators transparently to end users. Furthermore, IPsec is designed to provide security services for both the IPv4 [25] and the IPv6 [26] protocols. Its implementation is mandatory in IPv6 and optional in IPv4.

IPsec can provide prevention methods for different types of attacks. For instance, to protect against eavesdropping (or sniffing), IPsec encrypts data before transmission so that the data is viewable only by the peer who is able to decrypt it. In addition, prevention against data modification is achieved by attaching an HMAC or a digital signature code (see Chapter 1) to each output message, which is verified by the receiving end to detect any change that might have been made during transmission of the message. Furthermore, IPsec authenticates communicating peers before starting a secure communication to protect against identity spoofing and Man in The Middle Attacks (MITM) [7, 27, 28].

IPsec operates in end hosts or secure gateways. It provides security services to protect packets transmitted between a pair of hosts, between a pair of security gateways (routers or firewalls), or between a host and a security gateway. Most of the security services are provided through the use of two security protocols: the *Authentication Header* (AH) and the *Encapsulated Security Payload* (ESP) [see Chapter 4]. These protocols are designed to be independent of specific cryptographic algorithms to permit the selection of different algorithms as appropriate for the user's need, without redesigning or reimplementing the protocols. However, there are a set of default algorithms defined in RFC 4305 [29] for use with AH and ESP. Besides AH and ESP, IPsec uses a key exchange protocol to establish shared keys. IPsec–enabled devices must establish a shared key before applying any security services. A manual mechanism to setup and add keys is mandatory to implement.

However, a manual mechanism is not scalable and difficult to maintain. Therefore, a protocol for negotiating security services, authenticating peers, and generating a shared key dynamically is defined. This protocol is called the *Internet Key Exchange* (IKE) protocol (see Chapter 5).

3.3 Security Policy Database

The protection offered by IPsec is based on security requirements defined in a *Security Policy Database* (SPD) [30]. This database is established and maintained by a system administrator according to the security needs.

Each entry in the SPD defines the traffic to be protected, how to protect it, and with whom the protection is shared. For each packet entering or leaving an IPsec–enabled system, the SPD is checked to choose the proper action to apply to this packet. One of three possible actions might be taken: *apply*, *bypass*, or *discard*. The first option means to apply security services to outbound packets and expect inbound packets to have security services applied. For such action, the SPD entry contains the security services to apply, protocols to employ, and algorithms to use. Bypass means that the communication is insecure and needs no protection. Thus, security services are not applied to outbound packets, and inbound packets are expected not to have any security service applied. The discard action means the packet is not let in or out because it does not match any specific rule in the SPD.

In general, selecting the appropriate action to perform against IP packets is determined by matching information from the IP and transport layer headers of the packet against entries (polices) in the SPD. The pieces of information used to select the action are called *selectors*. Some of the selectors supported in IPsec are [30]: source and destination IP addresses, upper layer source and destination ports, and upper layer protocols. Using

these selectors, one can protect, for example, all traffic between two hosts or transport layer traffic (TCP or UDP). If more control is required, only certain types of traffic could be protected such as FTP or HTTP.

3.4 IPsec Modes of Operation

IPsec provides two modes of operation to exchange data across the Internet: *transport mode* and *tunnel mode* [24]. In transport mode, IPsec protects upper layer protocols. That is, the protection is applied to the payload of the IP packet, and the IP header is left intact. Typically this mode is used when end to end security is desired. It has the advantage of adding only a few bytes, the IPsec header, to each packet. However, since the IP header is sent in the clear, it also allows devices on the public network to see the ultimate source and destination of the packet. Unfortunately, by doing so, transport mode allows an attacker to perform some traffic analysis by knowing the source and final destination addresses of the packet.

The other mode of operation, tunnel mode, is used to provide data security between two networks (tunnel points). It provides protection for the entire IP packet, which becomes the payload in a new IP packet. The source and destination addresses of this new packet correspond to the two tunnel end points. This mode allows a network device, such as a router, to act as an IPsec proxy. That is, the router performs encryption on behalf of the hosts. The source router encrypts packets and adds a new IP header to each one. The source address in the new IP Header is the router's IP address. After that, the router forwards the packets along the IPsec tunnel. After receiving packets, the destination router decrypts the original IP packets and forwards them to the ultimate destination. The major advantage of the tunnel mode is that the end systems do not need to be modified to utilize the benefits of IPsec. Tunnel mode also protects against traffic analysis since the ultimate

source and destination in the original IP packet are encrypted. As a result, an attacker can determine only the tunnel end points.

3.5 Security Associates

Before IPsec can use the security services of ESP and/or AH protocols, such as confidentiality, authentication, and integrity, the communicating parties must determine exactly how they are going to use and apply these services. That is, the two parties must agree on which algorithm to use for encryption (for example DES or 3DES) and which to use for authentication (for example MD5 or SHA1). After deciding on the algorithms, the two parties must specify which mode of operation is to be used: transport mode or tunnel mode. Furthermore, the two parties must agree on session keys and their lifetimes. Therefore, there is a lot of information each party must keep track of and manage. To make such management easy, IPsec uses the concept of a Security Association (SA) [24,30,31] to manage all attributes of a secure communication channel. The SA attributes required and recommended for the use of IPsec protocols (AH, ESP) are defines in [30].

A SA is a relationship between two entities that describes how the entities will use security services to communicate securely. Basically, an SA groups together all the information needed in order to communicate securely with someone else. This includes information such as encryption algorithm, authentication algorithm and mechanism, and key information to name a few. In addition, an SA specifies the lifetime associated with it. At the end of that time, the SA expires and can no longer be used to protect IP traffic. As a result, a new SA for the same connection has to be created. All the active SAs are maintained in a database called the Security Associate Database (SADB) [24,30].

The IPsec SA is unidirectional, meaning that it is a one–way relation between the sender and the receiver. Consequently, if two hosts, A and B, are communicating securely

using ESP, then the host A will have an SA, SA_{out}, for processing outbound packets and will have a different SA, SA_{in}, for processing inbound packets. The host B will also create two SAs for processing its packets. The SA_{out} of the host A and the SA_{in} of the host B will share the same cryptographic algorithms and keys. Similarly, SA_{in} of the host A and SA_{out} of the host B will share the same set of cryptographic parameters. Figure 3.1 below shows the relation among different SAs. Since the SAs are unidirectional, a separate table (or database) is maintained for SAs used for outbound and inbound processing. The SAs are also protocol specific. That is, there is an SA for each IPsec protocol (ESP or AH). If two hosts, A and B, are communicating securely using both AH and ESP, then each host builds a different SA table for each protocol.

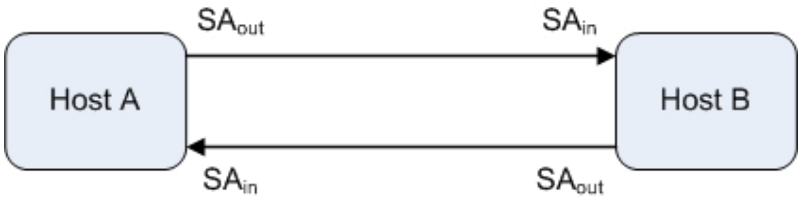

Figure 3.1: Security Association (SA) Establishment

Each SA is uniquely identified by the destination IP address of the received packet, the IPsec protocol, and the Security Parameter Index (SPI) [24, 30, 31]. An SPI is a 32-bit random number selected by the receiving end of the communication. It is sent in the IPsec header of each packet transmitted.

3.6 IPsec Processing

This section gives a brief overview of IPsec processing. The goal is to give the reader a general sense of the steps taken before a packet can be protected. Additional details and information can be found in the paper by Kent and Atkinson [30].

3.6.1 Outbound Processing

The outbound processing starts after receiving a packet from the transport layer. Next, the IPsec implementation accesses the SPD using selectors from the IP and transport layer headers to determine the appropriate action to apply to the outgoing packet. One of three possible actions might be taken:

- *Drop the packets*, in which case no more processing is needed.
- *Bypass security*, in which case normal processing of an IP packet is initiated.
- *Apply security*, in which case some security services need to be applied as indicated by the SPD entry.

A pointer to a SA(s) that represents the security services to apply is returned, if it is already established, when the SPD is accessed. If no SA is established yet, one is established and added to the SPD entry before any packet can be sent. After the SA(s) are established and the security services are determined, the IPsec implementation applies the required IPsec processing and adds the necessary IPsec headers (ESP and/or AH) to the outgoing packet. In addition, it adds the SPI value in these headers so that the receiver can determine the right SA(s) to process this packet.

3.6.2 Inbound Processing

The receiving end accesses its SPD to determine the appropriate policy to apply on the incoming packet. If the packets does not contain any IPsec headers, and the result of the SPD lookup is discard or apply but there is no established SA, the packet is dropped. Otherwise the packet is passed for normal IP processing.

If the packet contains an IPsec header, the IPsec implementation extracts the SPI, destination address, and IPsec protocol value from the received packet. It uses this infor-

mation to access its SADB to select the right SA to process the received packet. The IPsec protocol value is either ESP or AH. Based on this value, further processing is applied by handing the packet to the ESP or AH layer. Once the processing is done without any error, the IPsec header is stripped off, and the packet is forwarded to the IP layer for normal IP processing.

Chapter 4

IPsec Protocols

4.1 Introduction

IPsec uses two protocols to protect traffic at the IP layer: Encapsulating Security Payload (ESP), which provides encryption and/or authentication services; and Authentication Header, which provides a packet authentication service. These two protocols are the core of IPsec functionality, and each is described next. In addition to ESP and AH, IPsec uses a key management protocol to establish shared keys that are used in performing authentication and confidentiality. This protocol is called the Internet Key Exchange (IKE) Protocol. The IKE protocol is explained in Chapter 5.

4.2 Encapsulating Security Payload

Encapsulating Security Payload (ESP) [32] is the IPsec protocol that is responsible for providing confidentiality. In addition, it can provide data integrity and authentication services for IP packets. The ESP protocol inserts an ESP header and trailer into IP packets. It inserts the ESP header after the IP header and before the data to be protected. The ESP trailer is appended to the end of the protected packet. For example, to provide confidentiality for an IP payload, ESP takes the IP payload, such as a TCP segment, encrypts it

using a symmetric key algorithm, and encapsulates it within an ESP header and trailer.

ESP supports both confidentiality and authentication services. However, it is not necessary to use both services at the same time. According to the policy set by a system administrator, confidentiality, authentication, or both can be used. Therefore, it is possible to do ESP without encryption or ESP without authentication. However, it is illegal to have ESP without both encryption and authentication. At least one service must be used.

ESP is designed for use with symmetric encryption algorithms. All algorithms used must operate in CBC mode (see Section 2.3.1.2). CBC mode requires the data to be divided into blocks of equal size to process it. As a result, padding is added to the end of an IP packet to make it a multiple of the required block size, if necessary. Padding becomes part of the ciphertext and is stripped off by the recipient after being processed. Furthermore, CBC mode requires the use of an Initialization Vector (IV). The receiving end requires this IV to process the packet correctly. Thus, the IV is contained in the first bytes of the transmitted messages, depending on the algorithm requirements.

For an ESP implementation to be compatible with IETF specifications, DES in CBC mode [29, 33, 34] is mandatory–to–implement across all implementations of ESP for the confidentiality service. For Authentication, an Integrity Check Value (ICV) is computed after the encryption of the IP payload and is appended to the end of the packet. The computation of this value is based on one way hash functions. Compliant implementations of ESP are requires to support both HMAC–MD5 [35] and HMAC–SHA1 [36].

4.2.1 ESP Header

The ESP header [32] usually follows the IP header. In the case of IPv4, it immediately follows the IP header. In the case of IPv6, it immediately follows the IP header if no extension headers are used.

ESP protocol is assigned the number 50 to identify its existence. Therefore, the header immediately preceding the ESP header will contain the value 50 in its Next Header (IPV6) or Protocol (IPv4) fields to indicate that the following protocol is ESP. Figure 4.1 shows the ESP header format. It contains the following fields:

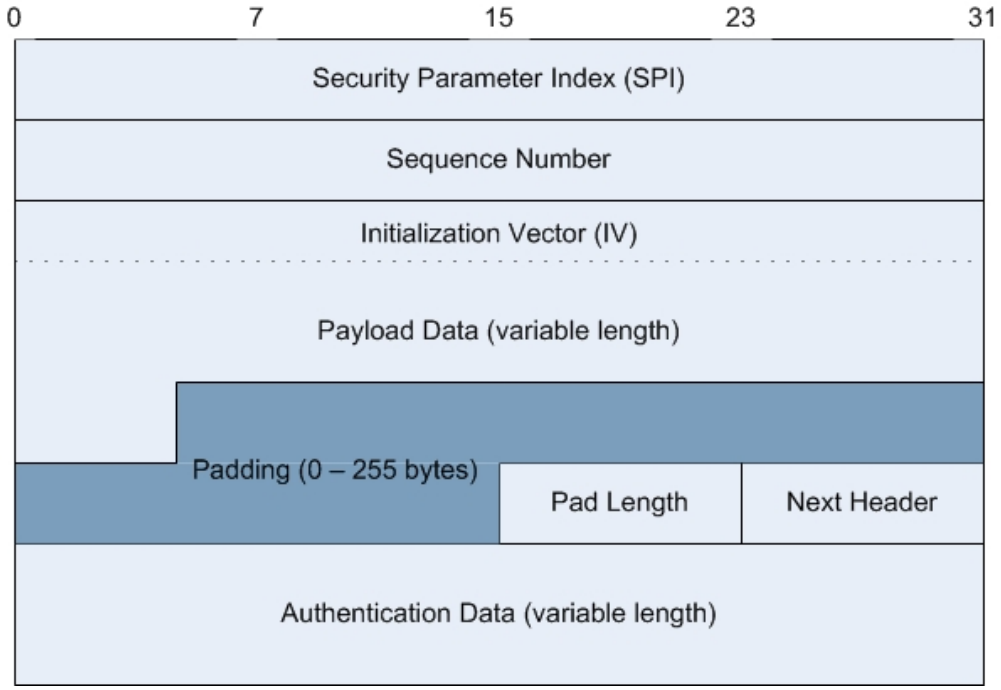

Figure 4.1: ESP Header Format

- *Security Parameter Index* (SPI): the SPI field is a 32–bit value that is set by the destination at the time of ESP Security Associate (SA) establishment. This value, together with the destination address and the protocol value (50 for ESP), identifies the proper SA to use in processing the packet at the destination. The SPI value is sent in the clear (not encrypted) because it is used to identify the algorithm and the key that will be used to decrypt the packet.

- *Sequence Number*: the sequence number field is a 32–bit monotonically increasing

number that provides anti–reply service to ESP. The sender must always set and transmit this field, but the receiver need not act upon it. The value of this field is set to zero when the sender and receiver establish the ESP SA. This value is incremented by one for each transmission of a packet. If this field reaches its maximum value (2^{32}), it is nor allowed to recycle. Instead, the sender and the receiver must establish a new SA for the connection.

- *Payload Data*: the actual data being protected by ESP is contained in the payload Data field. The length of this field depends on the length of the data contained in the field. This field is also used to contain the Initialization Vector (IV) that an encryption algorithm may require. For the mandatory–to–implement algorithm (DES in CBC mode), the IV is in the first eight bytes of this field. Note that the IV is not encrypted despite being part of the protected data.

- *Padding*: the padding field is used in two cases. The first case occurs when the encryption algorithm employed requires input to be a multiple of a certain block size. In this case the padding field is used to fill the input to the size required by the algorithm. In the second case, padding is used to right justify the Pad Length and Next Header fields when confidentiality service is not used. These two fields must terminate on a 4–byte boundary.

- *Pad Length*: this field is used to indicate how much pad has been added to the packet so that the recipient can determine the actual length of the payload data. The range of possible values of pad is between 0 and 255 bytes. If no pad is added, the sender still needs to include this field with a value of zero.

- *Next Header*: this 8–bit field identifies the type of data contained in the payload field. It could be an upper layer protocol data or an entire IP packet.

- *Authentication Data*: the authentication data field contains an Integrity Check Value (ICV) computed over the ESP packet minus the authentication data field. The length of this field depends on the output of the authentication algorithm employed. For the mandatory–to–implement algorithms, this field is 96 bits. If ESP is applied without the authentication service, this field is not used and will not exist in the ESP header.

4.2.2 ESP Modes

ESP protection is applied to an IP packet in one of two modes: the Transport Mode or the Tunnel Mode. The protection range applied to an IP packet differs according to the mode used. In transport mode, the ESP header is inserted between the original IP header and the upper layer protocol header. In addition, only the upper layer protocol data such as TCP or UDP is protected. That is, the IP header is not encrypted. In tunnel mode, the original IP packet is enclosed, or encapsulated, within an ESP header and trailer, and a new IP header is added to that. As a result, the original IP packet, including the original IP header, is encrypted because it becomes part of the payload of the new IP packet. The outer, unencrypted IP header contains the IP addresses of the tunnel end points, while the inner, encrypted IP header contains the ultimate source and destination addresses of the packet. This prevents an attacker from analyzing the network traffic between the ultimate source and destination addresses. Figure 4.2 shows an IP packet before and after applying ESP confidentiality service in transport and tunnel modes.

If authentication is used, an ICV is computed over the IP packet including the ESP header and trailer after the encryption and appended to the packet. The receiver uses this value to verify the integrity and authenticity of the packet. Figure 4.3 shows an IP packet before and after applying the ESP confidentiality and authentication services.

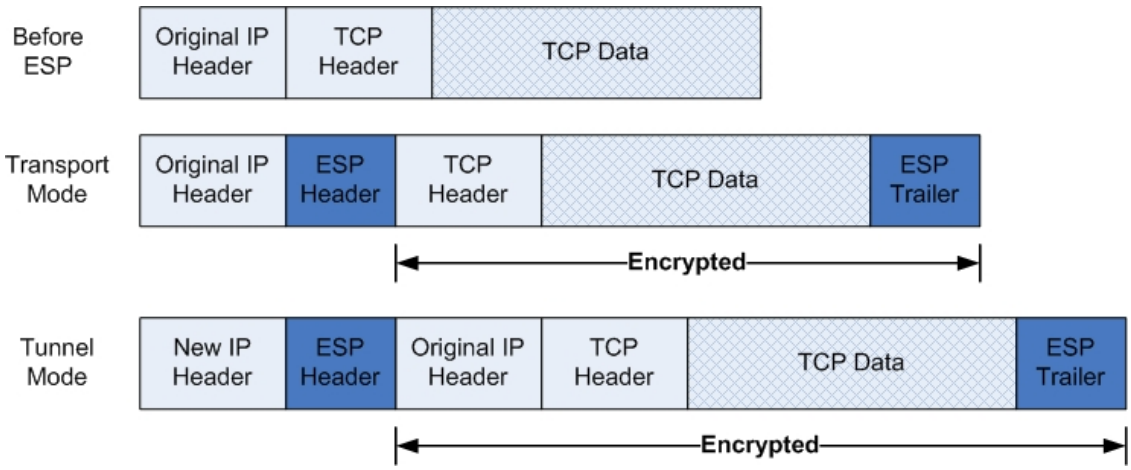

Figure 4.2: Applying ESP Confidentiality Service in Transport and Tunnel Modes

4.3 Authentication Header

The Authentication Header (AH) protocol is used to provide authentication, data integrity, and anti–reply services to IP traffic. It does not provide confidentiality. Therefore, it does not negotiate an encryption algorithm in its SA. AH defines two mandatory–to–implement authentication methods: HMAC–SHA1 [36] and HMAC-MD5 [35]. Like ESP, the output of these algorithms is truncated to 96 bits before inserted into the AH header. The authentication service provided by AH differs from that provided by ESP in the range of input it covers. While ESP does not authenticate any field in the outer IP header, AH does authenticate some fields. However, AH does not authenticate all fields of the outer IP header because some fields do not have constant values and change in transit such as hop limit, time to live, and header checksum. These fields are called mutable fields. AH calculates its authentication value over the *immutable* fields of the IP header as well as the IP payload. Mutable fields are set to zero for the calculation of the authentication value.

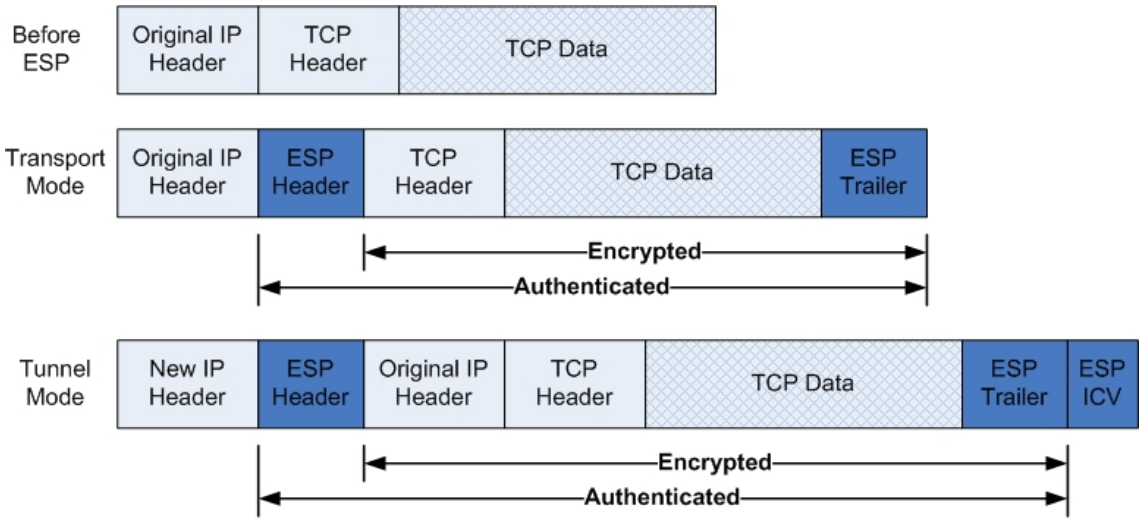

Figure 4.3: Applying ESP Confidentiality and Authentication Services in Transport and Tunnel Modes

4.3.1 AH Header

The AH protocol is assigned the number 51 to identify its existence. Therefore, the protocol header immediately preceding the AH header will contain the value 51 in its protocol (IPv4) or Next Header (IPv6) fields. The rules for inserting the AH header are similar to that of ESP. However, when ESP and AH are both protecting the same data (used at the same time), the AH is always inserted after the ESP header [24]. The AH header is simpler than the ESP header because it does not provide confidentiality. There is no trailer since there is no need for padding. In addition, there is no need for an IV. The AH header is shown in Figure 4.4. It contains the following fields:

- *Next Header*: this 8–bit field indicates what follows the AH header. It could be an upper layer protocol in transport mode or another IP packet in tunnel mode.
- *Payload Length*: is an 8–bit field that specifies the length of AH header in 32–bit words minus two. For example, when the standard authentication method of 96–bit

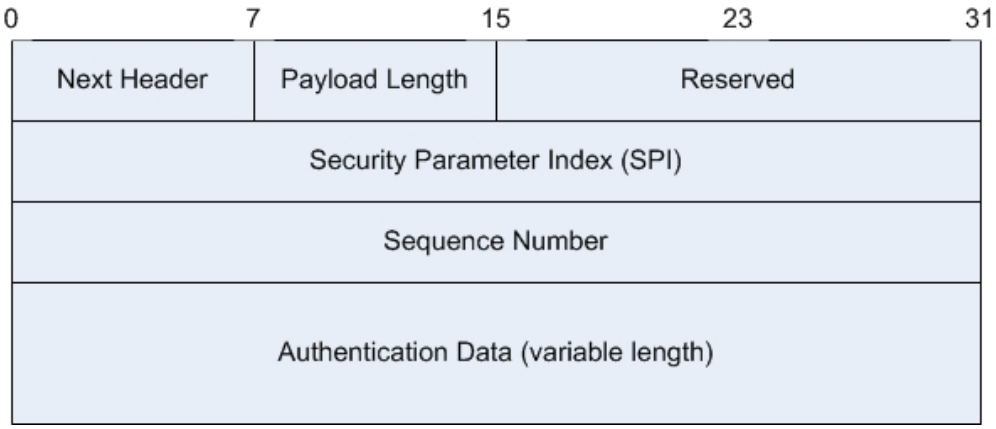

Figure 4.4: AH Header Format

length is used, this field's value will be $6 - 2 = 4$ since the header contains 3 32–bit fixed length fields plus 3 32–bit authentication value fields.

- *Reserved*: the 16–bit reserved field is not used and must be set to zero.
- *Security Parameter Index* (SPI): similar to ESP, this field is a 32–bit value that is used along with the destination address and the protocol value (51 for AH) to identify the SA used to authenticate this packet and the SA that will be used at the destination to process this packet.
- *Sequence Number*: is a 32–bit field containing a monotonically increasing number that is similar to the one used in ESP. Its main function is to provide the anti–reply service.
- *Authentication Value*: is a variable–length field that contains the Integrity Check value (ICV) resulting from applying an authentication algorithm over the IP packet, including the immutable fields of the IP header. For the mandatory–to–implement algorithms, 96 bits are inserted in this field.

4.3.2 AH Modes

The AH can be used in transport or tunnel modes. If AH is used in transport mode, the entire original IP packet (except for mutable fields) is authenticated. If used in tunnel mode, the entire new IP packet, including the original IP packet, is authenticated except for mutable fields in the new IP header. Figure 4.5 shows the IP packet before and after applying the AH in tunnel and transport modes.

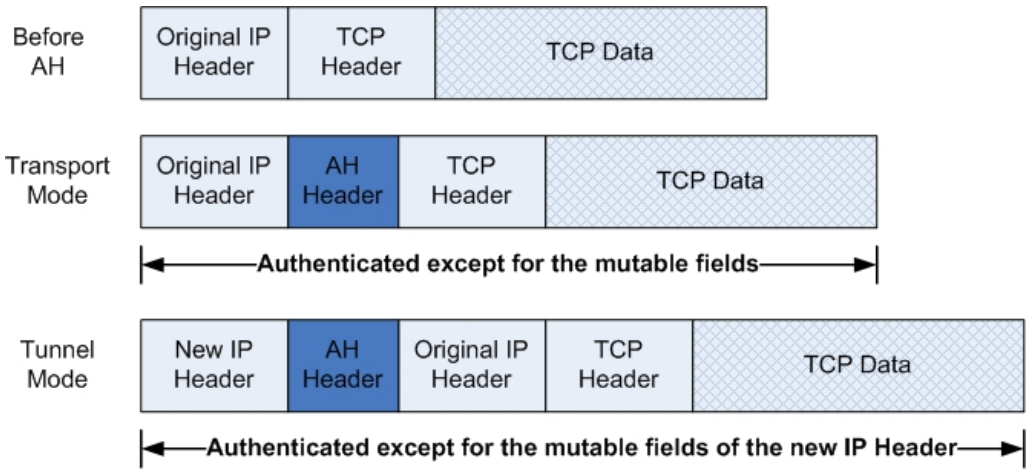

Figure 4.5: Authentication Service of AH Protocol in Transport and Tunnel Modes

4.4 Processing an IPsec Packet

When an IPsec packet matches an entry in the SPD denoting protection with ESP, AH, or both protocols, the SADB is required to see whether an SA exists. If not, one must be created. If there is an SA, the required security services are applied to the packet. These services might be satisfied using ESP or AH protocols. If both protocols are used, the authentication value must not be encrypted. This dictates that for outbound packets the ESP protocol is applied first to produce an ESP packet that is authenticated using the AH protocol. For inbound packets, the AH is applied first to verify the integrity and

authenticity of the packet. If it fails the authentication, the packet is discarded without decrypting it saving an unnecessary decryption operation. In summary, the AH protocol always protects ESP packets, not the other way around. Refer for Section 3.6 for general description of outbound and inbound processing of an IPsec packet.

Chapter 5

Internet Key Exchange Protocol

5.1 Introduction

Prior to an IP Packet being secured and protected by IPsec protocols, a security association (SA) must exist. If there is no SA currently active between the two peers wishing to communicate securely, one must be created. SAs may be created manually or automatically. In manual creation, a system administrator configures each system with the required keys, algorithms, and other required parameters. Even though this method is practical for a small, static environment, it is not well suited for large environments because it is not scalable. In addition, it takes much effort to reconfigure each system in case keys need to be renewed periodically to insure a higher level of security. Therefore, automatic creation of SAs is more favorable because it is more scalable and efficient. Furthermore, manual configuration reserves the resources required by SAs all the time regardless whether these SAs are frequently used or not, as opposed to on-demand creation of SAs. This is where the Internet Key Exchange (IKE) protocol comes into play. It establishes shared security parameters and authenticated keys dynamically as they are needed. In other words, IKE establishes SAs between IPsec peers.

IKE was standardized by the Internet Engineering Task Force (IETF) as the official protocol for establishing and configuring SAs for IPsec [6, 24]. It uses UDP port 500 or

TCP port 500 to exchange IKE messages between the two peers. Therefore, these ports must be permitted on any IP interface involved in exchanging IKE packets.

The IKE protocol is a combination of the Oakley [37] and SKEME [38] protocols, and operates inside a framework defined by Internet Security Association and Key Management Protocol (ISAKMP) [31]. Oakley is a key exchange protocol that introduced the concept of modes. The modes are different key exchanges. Each produces the same result of establishing a shared, authenticated key. IKE borrowed the idea of different modes, described in Section 5.3, and incorporates it in the protocol operations. SKEME is another key exchange protocol, which defines a type of authenticated key exchange where the parties use public key cryptography to authenticate each other. IKE borrowed this technique exactly from SKEME for one of its authentication methods that is described in Section 5.4. IKE depends heavily on ISAKMP. It uses the payloads, messages, exchanges, and phases of ISAKMP to establish the required security parameters and keys. ISAKMP is described in the next section.

5.2 Internet Security Association and Key Management Protocol

ISAKMP defines producers and packet formats to authenticate a peer, to exchange data for secret keys generation, and to negotiate security services and attributes that involve the creation and management of SAs. It provides a consistent framework for transferring data and constructing messages. However, it does not define the content of these messages since it is not bound to any specific cryptographic algorithm, key generation method, or authentication mechanism. This independence provides better support to accommodate new improvements and algorithms. For example, if improved cryptographic or authentication algorithms are developed or if new attacks against current algorithms

or key exchange methods are discovered, ISAKMP can update the current algorithms without having to develop a new protocol or redesign the current one.

5.2.1 ISAKMP Messages and Payloads

ISAKMP is responsible for the creation and management of SAs. Recall that a SA contains all the information (SA attributes) necessary to execute security services such as encryption algorithms, authentication mechanism, key lengths and lifetimes, and modes of operation.

ISAKMP SAs are established and created through the exchange of some ISAKMP messages between the peers. Each message is composed of an ISAKMP Header followed by a variable number of payloads. The number and ordering of these payloads differ according to the type of exchange being used. Each exchange (see Table 5.1) dictates a number of payloads in some predefined order to accomplish its goals.

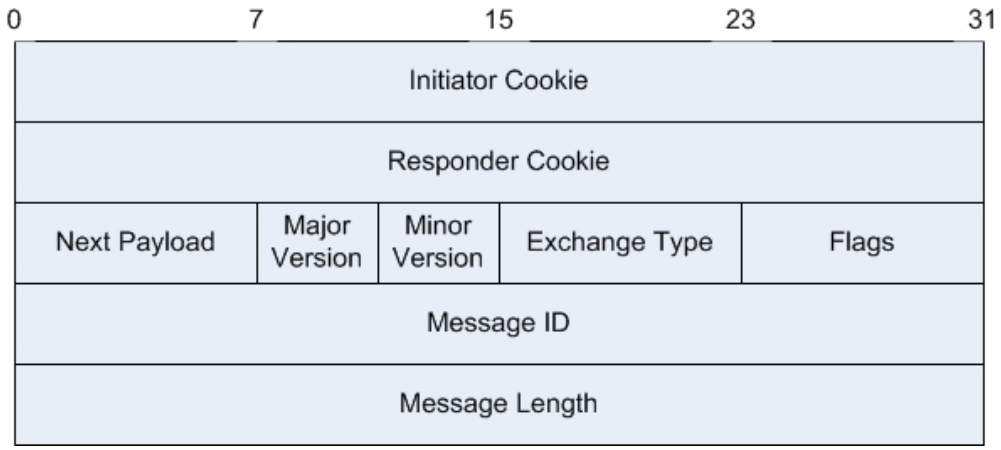

Figure 5.1: ISAKMP Header Format

The ISAKMP header has a fixed format. It is shown in Figure 5.1. The Initiator and Responder cookies are four–octet numbers that are associated with the initiator and responder peers, respectively. These cookies are used by either peer (as opposed to the

normal way in which the tuple [SPI, destination address, protocol] is used to identify an SA) to identify the ISAKMP SA created in phase one of communication.

The Next Payload field is a one-octet field that indicates which of the various payloads immediately follows the ISAKMP Header. These payloads are described below. The Major and Minor Version fields, each is four bits, are used to identify the version of the ISAKMP. The one-octet exchange field specifies the type of the ISAKMP exchanges being used in the current negotiation. It dictates the number and ordering of payloads exchanged. Table 5.1 shows the exchange types that are defined in the ISAKMP protocol [31].

Exchange Type	**Value**
None	0
Base	1
Identity Protection	2
Authentication Only	3
Aggressive	4
Informational	5

Table 5.1: ISAKMP Exchanges

Flags is a one-octet field that indicates some specifications for the exchange being used. It is represented in a bit mask scheme, in which each bit signifies the presence of an option. Currently, only 3 bits (options) are defined: the encryption flag, which signifies that the payloads following the ISAKMP header are encrypted; the commit flag, which signifies that a peer wishes a notification of negotiation completion; and the authentication-only bit, which is used primarily to add key recovery to ISAKMP.

The Message ID is a two-octet unique identifier used for message identification in the IKE phase two negotiation. This two-octet value is generated by the initiator of the phase two. The entire length in bytes of an ISAKMP message, including the length of the

header itself, is determined by the four-octet Message Length field. This field is the last part of an ISAKMP header.

ISAKMP defines 13 different payload types [24]. They all begin with the same generic header shown in Figure 5.2. This header provides the ability to chain different payloads together to form an ISAKMP message(see Figure 5.3). Moreover, it defines the boundaries between the payloads of an ISAKMP message.

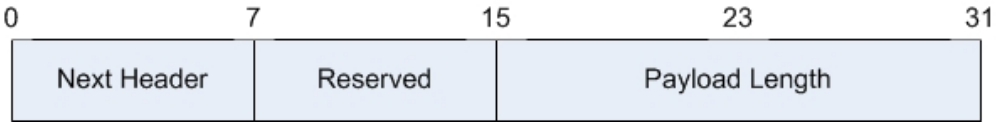

Figure 5.2: ISAKMP Generic Header

The Next Header is a one-octet field that specifies the type of the next payload that follows the current payload in the message. Table 5.2 summarizes the payload types defined in ISAKMP. If the current payload is the last in the message, then the Next Header field is set to zero. The one-octet Reserved field is unused and is set to zero in the sender and is ignored by the receiver. The Payload Length is a two-octet field that specifies the length of the current payload, including the generic payload header.

The Security Association payload is used to negotiate security attributes, which includes the encryption algorithm, the hash algorithm, the authentication method, and lifetime. Moreover, it specifies the protocol for which the SA is being established. It could be a phase one ISAKMP SA or a phase two IPsec SA. Security attributes and mechanisms are transferred through the Proposal and Transform payloads. The Proposal and Transform payloads are dependent on the Security Association payload, and may not be used or transmitted alone. Instead, they are encapsulated in a SA payload. The proposal payload contains data used during SA negotiation. Its payload indicates the security protocol (ISAKMP, ESP, or AH) for which the SA is being negotiated. The payload also indicates

Next Payload Type	Value
None	0
Security Association (SA)	1
Proposal (P)	2
Transform (T)	3
Key Exchange (KE)	4
Identification (ID)	5
Certificate (Cert)	6
Certificate Request (CR)	7
Hash (HASH)	8
Signature (SIG)	9
Nonce (NONCE)	10
Notification (N)	11
Delete (D)	12
Vendor ID (VID)	13

Table 5.2: ISAKMP Payload Types

the sending entity's SPI and the number of algorithms (transforms) proposed for each protocol. The Transform payload contains the specific values of the security attributes that are negotiated in the SA establishment. These values are represented in a type/value scheme. For instance, an attribute of type ”encryption algorithm” might have the value of ”DES”. These values are, of course, not English words but numbers that are assigned by IANA, the Internet Assigned Number Authority [24].

The Key Exchange payload contains data necessary for the two peers to agree on a session key. The interpretation of this date is specified by the negotiated key exchange algorithm. This payload can be used to support a variety of key exchange methods. Typically, it includes a Diffie-Hellman [20,24] public value. The Identification payload is used to exchange identification data that is used to determine (authenticate) peers. The Certificate payload is used to carry certificates or certificate–relevant information. It may appear in any ISAKMP message. The Certificate Request payload provides a means to request a certificate from a peer. Similar to a certificate payload, it may appear in any ISAKMP

message. The Hash payload contains data generated by a hash function selected during the SA negotiation. The data transmitted in a hash payload is used to authenticate peers and provide data integrity. In a Signature payload, the data contained is generated by a digital signature function selected during the SA negotiation. Basically, it is an output of a hash function that is encrypted by the private key of the sender using a digital signature algorithm such as DSS algorithm [23]. This payload is used to verify the integrity of the data in a message and to authenticate peers. The Nonce payload contains random data generated by the transmitting entity to guarantee liveness during an exchange. That is, to guarantee some uniqueness and prevent the replay of old packets into a new connection. Notification messages, such as error conditions, are transmitted in the Notification Payload. The Delete Payload indicates an SA that the sender has deleted from its database (SADB) and is no longer valid. This informs the receiver to delete the specified SA. In the Vendor IP payload, vendor identification data is used by different vendors to identify their implementation of ISAKMP on the network.

To create an ISAKMP message, payloads are chained together by using the Next Header field in the ISAKMP generic header. The ISAKMP Header describes the first payload following it, and each payload describes which payload comes next. The example in Figure 5.3 shows how two payloads (Key Exchange and Nonce) are chained together to construct an ISAKMP message.

5.2.2 ISAKMP phases

ISAKMP describes two separate phases of negotiations. In the first phase, peers establish an authenticated and secure channel between themselves. In other words, phase one establishes an ISAKMP SA, agrees on the keying material, and authenticates peers. In the second phase, that authenticated and secure channel of phase one is used to negotiate

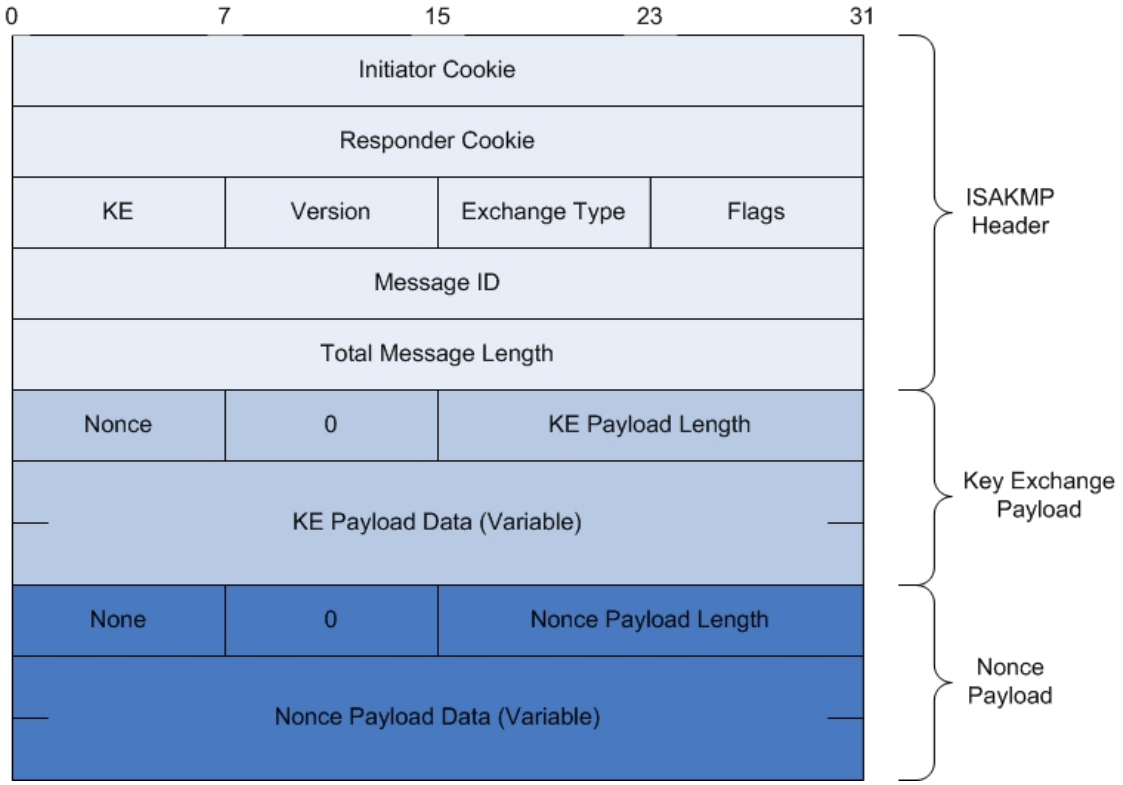

Figure 5.3: ISAKMP Payloads Chained to Form an ISAKMP Message

security services for a different protocol such as IPsec. Since the ISAKMP SA is already authenticated, it can be used to provide source authentication, integrity, and confidentiality for all messages exchanged in phase two. Once phase two finishes, the state associated with it in the ISAKMP process is destroyed, but the ISAKMP SA itself can live on (until its life time expires) to secure future phase two exchanges. This justifies the use of a two phase approach. It allows many second phase SAs to be established without the need to start the negotiation process from the beginning. This reduces the cost of SA management by eliminating the need to go through the costly authentication process every time a second phase (usually IPsec) SA is established [24].

5.2.3 Security Associate Establishment

Establishing a SA (an ISAKMP SA or an IPsec SA) requires the negotiation of security attributes and parameters. This negotiation is done using the security associate, proposal, and transform payloads. A message for negotiating and establishing a SA consists of a single security association payload which includes one or more proposal payloads. Each proposal payload contains one or more transform payloads. Therefore, the security association, proposal, and transform payloads are considered together and treated as one unit. Thus, the Next Header field of the security associate payload points to any following payload not to the proposal payload.

Through the proposal payload, the initiator presents to the responder the security protocol to use with the security association being negotiated. For example, in phase one the initiator specifies ISAKMP as the protocol for which the current security association is being negotiated. If the initiator needs to negotiate multiple protocols, as in the case of using ESP *and* AH to protect phase two communication, it presents one proposal for each protocol provided that all proposals have the same proposal number. This is equivalent to the logical ***AND*** operation. It tells the responder to treat these proposals as one unit and either accept or reject them all.

The initiator may propose a set of proposals that the responder has to select one. In this case the initiator presents to the responder more than one proposal, each with a different proposal number. This is equivalent to the logical ***OR*** operation. If more than one proposal are presented to the responder, they are presented according to the initiator's preference order with the first proposal being the most desirable.

For each proposal payload offered in the SA negotiations, there must be at least one accompanying transform payload. The transform payload provides the initiator with the capability to present multiple values for a protocol (or algorithm) to the responder. The

responder must select only one transform for each protocol (algorithm) presented in the proposal or reject the whole proposal.

To see an example, assume an initiator would like to setup an ISAKMP SA. The ISAKMP SA has various parameters that must be agreed upon between the peers. Since some IKE messages are encrypted and authenticated, the peers must agree on the encryption and authentication methods. In addition, they must agree on how to exchange and authenticate each other's identity and which key exchange method to use. These parameters (the encryption algorithm, the hash algorithm, the authentication method, the Diffie-Hellman parameters, and SA lifetime) are referred to as a protection suite. Protection suites are negotiated as units by exchanging SA payloads. Each proposal payload in the SA payload represents a single protection suite. The attributes (parameters) of that protection suite is represented in a transform payload. Therefore, suppose the initiator wants to protect its communications using two different protocols: ESP and AH. For the ESP protocol, the initiator proposes two transforms: 3DES and DES. For the AH protocol, the initiator proposes two transforms: SHA1 and MD5. The responder must select one transform from the two transforms proposed for ESP and one transform from the two proposed transforms for AH. The resulting protection suites will be either 3DES with SHA1, 3DES with MD5, DES with SHA1, or DES with MD5 depending on which ESP and AH transforms are selected by the responder.

To complete the security association establishment, the responder replies with an SA payload containing the selected proposals and their corresponding transform payloads. In this case, each proposal must contain only one transform. This approach shows that the responder is the entity which controls the choice of the security attributes and algorithms. If the initiator wants to control this process because, for example, it has a restricted policy in which a certain algorithms must be used and nothing else, the initiator presents the

responder with proposals that have a single transform payload. Therefore, the responder will have no choice and will either accept or reject the proposal.

After an ISAKMP SA is established in phase one, a SA for the IPsec protocols (ESP and AH) is established in phase two under the protection of the ISAKMP SA. ISAKMP SA differs from that of IPsec in that it is bidirectional. Even though one party is assigned the role of an initiator, and one is assigned the role of a responder, once an ISAKMP SA has been established it may be used to protect both inbound and outbound traffic. Moreover, regardless of who initiated the phase one exchange that established the ISAKMP SA, either party may initiate a phase two exchange and protect it with the ISAKMP SA. Note that the cookies in the ISAKMP Header are not swapped if the responder in phase one becomes the initiator in phase two because the cookie pair is used to identify the ISAKMP SA in the SAs database (SADB).

5.3 IKE Phases and Modes

IKE is an implementation (instantiation) of the ISAKMP. It uses the two phases of the ISAKMP. The first phase establishes an IKE Security Associate (called ISAKMP SA), and the second phase uses that SA to negotiate SAs for IPsec protocols. To establish these SAs IKE defines two exchanges (modes) for phase one, and one exchange (mode) for phase two. Phase one exchanges (modes) are called Main Mode and Aggressive Mode. Either mode can be used to establish an ISAKMP SA . Phase two exchange (mode) is called Quick Mode. In this mode an IPsec SA is established.

Both phase one modes accomplish the same thing, yet in different steps and numbers of messages. They establish an ISAKMP SA with the required parameters to provide confidentiality, message integrity, and message source authentication for IKE messages. In addition both modes conform to standard ISAKMP payload format, attribute encoding,

and method of processing. There are no requirements for the order of payloads in an ISAKMP message except that an SA payload must precede all other payloads in phase one modes.

5.4 IKE Authentication Methods

From among the parameters negotiated in the ISAKMP SA, the parameter that has the most impact on the IKE modes is the authentication method. An IKE exchange may actually change depending on the authentication method negotiated by the two peers. Three types of authentication are supported in IKE (phase one negotiation). One is authentication with digital signature, in which a hash value is computed over some of the payloads exchanged between the two peers, such as the nonce and the Diffie–Hellman public values, and signed using a digital signature algorithm like the Digital Signature Standard (DSS). After that it is sent to the other peer to verify it. Another method is authentication using public key encryption, in which the nonces that were negotiated in the ISAKMP SA establishment are encrypted using the other party's public key. The public keys must be provided in some fashion beforehand. The last method is authentication using preshared keys, in which a key (password) manually setup between the peers is used along with nonces exchanged to create a hash value that is used to authenticate peers. Authentication using preshared keys is the only method that MUST be supported in any IKE implementation. For more information about authentication using digital signatures or public key encryption see Harkins and Carrel paper [6] and Doraswamy and Harkins book [24]. Authentication using a preshared key is described in Section 5.6.

During IKE message exchanges, various parameters are transmitted in the clear. However, both sides of the communication maintain some secret information that will not be visible for an observer or an attacker. These secretes are:

- SKEYID: is the first secret computed, and on which all subsequent keys are based. It is derived from secret material known only to the communicating parties.
- SKEYID_d: is used to derive keying material for IPsec SA. (Phase two SA).
- SKEYID_a: is used to provide data integrity and data source authentication to IKE messages.
- SKEYID_e: is used to encrypt IKE messages. For example, it is used in the last two messages of the Main Mode exchange.

5.5 Secrets Generation

The generation of SKEYID is dependent on the authentication method negotiated. However, all other SKEYID–based secrets are generated identically regardless of the authentication method. Each side of the communication contributes a cookie and a nonce to the secret generation. The initiator contributes his cookie, *CKY–I*, and his nonce, N_i. Similarly, the responder contributes *CKY–R* and N_r. In addition, the peers share the Diffie-Hellman secret, g^{ir}, as a result of the Diffie–Hellman key exchange in phase one. Using | to denote concatenation, SKEYID is generated as follows:

- For signature authentication

$$SKEYID = PRF(N_i \mid N_r,\ g^{ir})$$

- For public key authentication

$$SKEYID = PRF(hash(N_i \mid N_r),\ CKY\text{–}I \mid CKY\text{–}R)$$

- For preshared keys authentication

$$SKEYID = PRF(preshared\,key,\ N_i \mid N_r)$$

PRF, the Pseudo Random Function, is the HMAC version of the negotiated hash function. In another way, $PRF = Hash(key, message)$. Once SKEYID has been generated, the remaining secrets can be generated regardless of the authentication method:

- $SKEYID\_d = PRF(SKEYID,\ g^{ir} \mid CKY\text{–}I \mid CKY\text{–}R \mid 0)$
- $SKEYID\_a = PRF(SKEYID,\ SKEYID\_d \mid g^{ir} \mid CKY\text{–}I \mid CKY\text{–}R \mid 1)$
- $SKEYID\_e = PRF(SKEYID,\ SKEYID\_a \mid g^{ir} \mid CKY\text{–}I \mid CKY\text{–}R \mid 2)$

  Where the numbers 0, 1, and 2 are represented as one octet.

Phase one messages are authenticated by each side computing a hash that the other side can verify. The computation of the hash value is identical regardless of the authentication method negotiated. The initiator's authenticating hash is:

$$HASH\text{–}I = PRF(SKEYID,\ g^i \mid g^r \mid CKY\text{–}I \mid CKY\text{–}R \mid SA\text{–}offer \mid ID_i)$$

The responder authenticating hash is:

$$HASH\text{–}R = PRF(SKEYID,\ g^r \mid g^i \mid CKY\text{–}R \mid CKY\text{–}I \mid SA\text{–}offer \mid ID_r)$$

Where g^i and g^r are the Diffie-Hellman public values of the initiator and the responder, respectively. SA–offer is the SA payload that was offered by the initiator to the responder (minus the ISAKMP generic header). ID_i and ID_r are the identities of the initiator and the responder in phase one communication, respectively.

5.6 Preshared Key Authentication Method

Phase one communication can be authenticated by a preshared key. The key is derived by some out–of–bound mechanism and agreed upon beforehand. Moreover, the key SKEYID_e is the one used to encrypt all payloads following the ISAKMP header, if required. Encryption must be applied using CBC mode which requires the use of an Initialization Vector (IV)(see Section 2.3.1.2). The IV is generated by hashing the two Diffie-Hellman public values together,g^i and g^r.

5.6.1 Main Mode

Six messages are negotiated to establish an ISAKMP SA in the phase one exchange when the Main Mode is used. The first two messages are used for negotiating the security policy (SA parameters) for the exchange. They are also used for exchanging cookies of the initiator and responder. The next two messages are used for the Diffie-Hellman keying material exchange. After these messages, the peers have exchanged the Diffie-Hellman public values, and they are ready to compute the shared key and generate the SKEYID–based secrets. The last two messages are used for authenticating the peers. They include each peer's identity together with its authenticating hash. The last two authentication messages are encrypted with the SKEYID_e key using the negotiated encryption algorithm. As a result, the identities of the peers are protected from eavesdroppers. Figure 5.4shows the messages negotiated during the Main Mode.

In Figure 5.4, HDR represents an ISAKMP header that describes the exchange type (mode) being used (Main Mode in this case). HDR* indicates that all payloads after the header are encrypted. SA, KE, Nonce, Hash, and ID are security associate, key exchange, nonce, hash, and identity payloads respectively. The KE payload contains the Diffie-Hellman public values g^i and g^r. The i and r subscripts refer to the initiator and responder

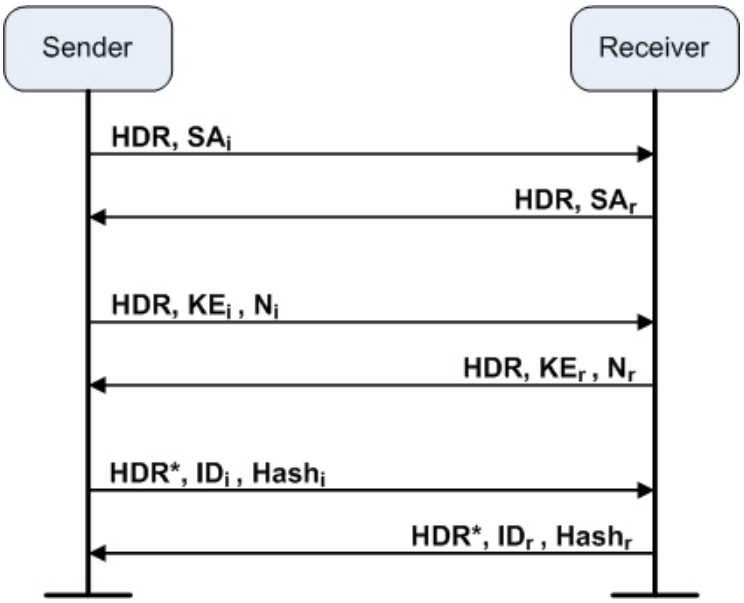

Figure 5.4: Main Mode Exchange using Preshared Key

values, respectively. The $Hash_i$ payload contains the HASH–I value, while the $Hash_r$ payload contains the HASH–R value described previously. Note that while the last two messages are encrypted, the first four messages, which contain most of the necessary information to compute the shared secret state, are sent in the clear. Therefore, an attacker can get them easily.

5.6.2 Aggressive Mode

The purpose of Aggressive Mode is the same as Main Mode. The difference is that Aggressive Mode takes half the number of messages as Main Mode does. As a result, the Aggressive Mode limits its negotiating power and does not provide identity protection as Main Mode does. In Aggressive Mode exchange, the initiator offers the SA parameters, its Diffie-Hellman public value, its nonce, and its identity in the first message. The responder replies with the selected SA parameters, its Diffie-Hellman public value, its nonce, its identity, and a hash payload for authentication purposes. The hash payload contains the

HASH–R value. At this point all the information to compute the shared secrets are exchanged. The last message is used for authenticating the initiator and provides a proof of participation in the exchange. It contains the HASH–I value. Figure 5.5 summarizes the Aggressive Mode exchange.

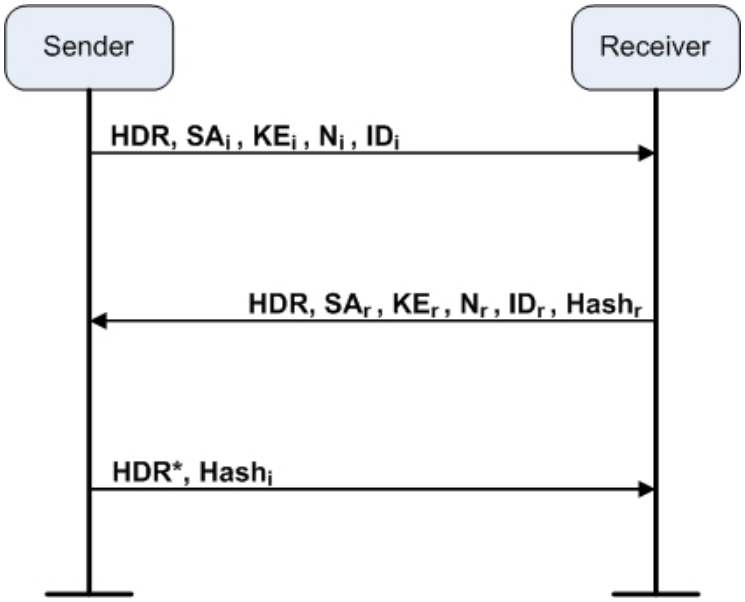

Figure 5.5: Aggressive Mode Exchange using Preshared Key

5.7 Quick Mode

Once an IKE SA is established, via Main or Aggressive Mode exchange, in phase one it can be used to generate SAs for other security protocols such as IPsec. These SAs are established using Quick Mode. In addition, Quick Mode is used to derive keying material for phase two communication. Quick Mode is done under the protection of the previously established IKE SA. The SKEYID_a value computed in IKE SA is used as a key to authenticate all Quick Mode messages, and the SKEYID_e value from the same ISAKMP SA is used to encrypt all the payloads exchanged, except the ISAKMP Headers.

Figure 5.6 shows the messages negotiate during Quick Mode exchange. The information between brackets is optional. Note that in Quick Mode a hash payload must immediately follow the ISAKMP header, and a SA payload must immediately follow the hash payload. Other than that there are no restrictions on the payload ordering in the Quick Mode.

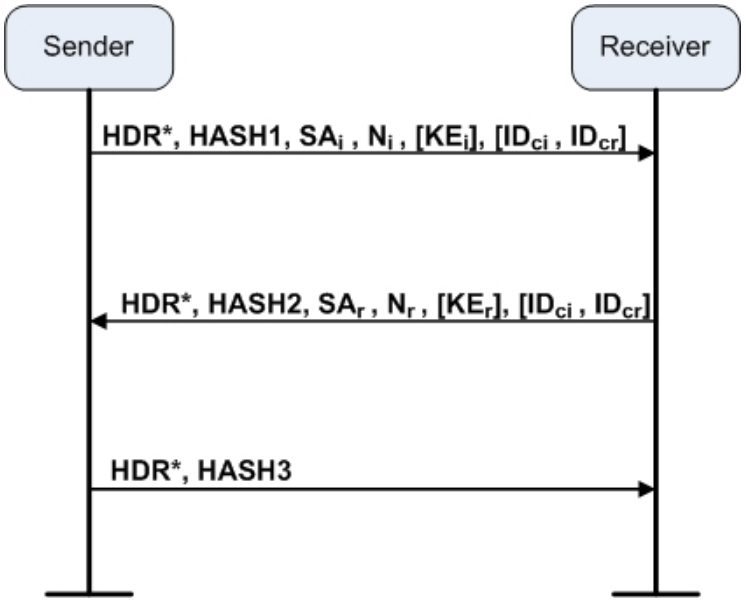

Figure 5.6: Quick Mode Exchange

Quick Mode derives the keys used for the IPsec SA from the ISAKMP SA key SKEYID_d , and computes some authenticating hashes [6, 24]: HASH(1), HASH(2), and HASH(3). HASH(1) is computed over the message ID from the ISAKMP header concatenated with the entire message following the hash payload. This includes all generic payload headers, but excluding any padding added as a result of encryption. HASH(2) is computed in the same way except that the initiator's nonce payload (N_i) minus its generic header is added after the message ID and before the remaining parts of the message. HASH(3) is computed over the value zero represented as a single octet concatenated with the message ID followed by the initiator's and the responder's nonce payloads respec-

tively minus their generic headers. Therefore, the authenticating hashes are computed as follows:

- $HASH(1) = PRF(SKEYID\_a,\ M\text{–}ID \mid SA \mid N_i\ [|KE]\ [|\ ID_{ci} \mid ID_{cr}])$
- $HASH(2) = PRF(SKEYID\_a,\ M\text{–}ID \mid N_i \mid SA \mid N_r\ [|KE]\ [|\ ID_{ci} \mid ID_{cr}])$
- $HASH(3) = (SKEYID_a, 0 \mid M\text{–}ID \mid N_i \mid N_r)$

The optional identities exchanged in this mode (ID_{ci} and ID_{cr}) can be used by the receiver to determine which services are applicable to which clients.

As mentioned earlier, the ISAKMP SA is bidirectional. This means that either party can initiate a quick mode exchange in phase two regardless of who initiated the Main or Aggressive mode exchanges of phase one. When this happens, the initiator of phase one exchange might become a responder in phase two exchange and vice versa. However, regardless of the role played in phase two, the initiator cookie and the responder cookie are determined based on the role played in the phase one exchanges (Main Mode or Aggressive Mode). This is important since these cookies are used to identify the ISAKMP SA used to protect the Quick Mode exchange in phase two.

Chapter 6

Weakness of IKE Preshared Key Authentication Method

6.1 Introduction

Some of the authentication methods defined in the IKE protocol require the use of public key cryptography. Therefore, if the communicating peers have no public/private keys assigned to them, or if it is unaffordable to obtain such keys, then the only authentication method available in the IKE protocol is the preshared key authentication method. In fact, the preshared key authentication method is the only mandatory–to–implement authentication method in the IKE protocol. However, authentication based on preshared keys is subject to off–line dictionary and brute–force attacks that can compromise the security provided by the phase one security associate, ISAKMP SA. As a result, all phase two negotiations, which are used to generate SAs for IPsec, are compromised also. In this chapter each mode of the phase one exchange is analyzed under the preshared key authentication method to show its weaknesses and the way it can be compromised. Moreover, the speed with which a preshared key could be compromised is also computed.

6.2 Attacking the Aggressive Mode

Both modes of phase one, Main Mode and Aggressive Mode, are subject to brute–force or off–line dictionary attacks [39] through which the preshared key can be determined. However, with Aggressive Mode it is much easier to implement or apply such attacks. To illustrate, the reader can refer to Figure 5.5, which shows the Aggressive Mode Exchange under the preshared key authentication method. In this mode all the information necessary to compute the phase one SKEYID–based secrets (SKEYID, SKEYID_d, SKEYID_a, SKEYID_e) is sent in the clear. Therefore, an attacker can easily intercept the initiator's cookie, nonce, identity, Diffie–Hellman public key, and SA parameters from the first message. Moreover, he/she can intercept the responder's cookie, nonce, identity, public key, and HASH–R value from the second message. Note that the intercepted HASH–R value is not encrypted.

The HASH–R value is a very important piece of information an attacker can obtain from intercepting the responder's second message. Recall that the value of $HASH\text{–}R = PRF(SKEYID,\ g^r \mid g^i \mid CKY\text{–}R \mid CKY\text{–}I \mid SA\text{–}offer \mid ID_r)$, where $\mid$ is the concatenation operation. Hence, it is clear that the attacker has all the necessary information to compute the HASH–R value except $SKEYID = PRF(preshared\ key,\ N_i \mid N_r)$, which depends on the preshared key. However, the attacker can determine the preshared key by applying an off–line dictionary attack (or a brute–force attack) on the preshared key. The attacker enumerates all possible candidates for the preshared key, and for each one computes:

$SKEYID_{new} = PRF(guessed\ key,\ N_i \mid N_r)$, and

$HASH\text{–}R_{new} = PRF(SKEYID_{new},\ g^r \mid g^i \mid CKY\text{–}R \mid CKY\text{–}I \mid SA\text{–}offer \mid ID_r)$.

In the case that $HASH\text{–}R_{new} = HASH\text{–}R$(the intercepted one), then there is a very high probability that the guessed key equals the preshared key because of the one–way

property of the hash functions. Note that the PRF, which is the HMAC version of the negotiated hash function, can be easily known from the parameters of the SA offer sent in the clear in the first message. Thus, given enough time and resources, an attacker can produce the preshared key and compute the SKEYID value, on which all the remaining secrets depend.

In addition to the preshared key, the attacker needs the Diffie–Hellman shared secret, g^{xy}, to reveal the remaining secrets (SKEYID_d, SKEYID_a, and SKEYID_e). The Diffie–Hellman shared secret is very difficult to compute from the two public values g^r and g^i that are obtainable from the intercepted messages (if the Diffie–Hellman key is 1024 bits, the probability of guessing it is $\frac{1}{2^{1024}}$). However, the attacker can use the Man in The Middle (MITM) attack, after compromising the preshared key, to compromise the Diffie–Hellman shared secret (see Figure 6.1) and all future connections. The purpose of this attack, as a result, is to discover the shared secret resulting from the Diffie–Hellman exchange between the two peers.

To compromise a connection successfully, the MITM attack must be launched after compromising the preshared key. In the MITM attack, the attacker intercepts the first initiator's message to the responder and changes the initiator's Diffie–Hellman public key g^i to his/her public key g^a, and then sends the message to the responder with g^a as the Diffie–Hellman public key as if it is coming from the initiator. The responder replies with its Diffie–Hellman public key g^r in the second message. Again the attacker intercepts this message and changes the responder's Diffie–Hellman public key g^r to his/her public key g^a. In addition, the responder's authenticating hash, HASH–R, is replaced with a new value using the parameters shared between the attacker and the initiator. After that the message with the attacker's Diffie–Hellman public key and the new authenticating hash is sent to the initiator as if it is coming from the responder. The authenticating hash will

pass the verification and the attacker will be authenticated to the initiator. At this point the attacker shares the g^{ia} secret key with the initiator, and the g^{ar} secret key with the responder.

Based on this situation caused by the MITM attack, and armed with the value of the preshared key previously compromised in a separate attack, the attacker computes two sets of secret keys (SKEYID, SKEYID_d, SKEYID_a, SKEYID_e) and two sets of the authenticating hashes (HASH–I, HASH–R). One set is computed using the shared parameters with the initiator, and the other set is computed using the shared parameters with the responder. In addition, the initiator and the responder compute their four secret keys and their authenticating hashes normally, with the exception that the shared keys with the attacker are used in the computations.

When the initiator sends its authentication hash, HASH–I, it will be intercepted by the attacker and replaced with an authenticating hash value that is based on the parameters shared between the attacker and the responder. This authentication value will pass the verification at the responder and the attacker will be authenticated at the responder.

After authentication is done, the initiator assumes it is talking with the responder. Therefore, it uses its SKEYID_d, which is actually shared with the attacker, to encrypt phase two communications to the responder. The attacker intercepts these messages, decrypts them with the shared key with the initiator, and reencrypts them using the shared key with the responder before sending them to the responder. The responder will accept the messages as if they are coming from the initiator and process them. The final result is that all the communication between the initiator and the responder is compromised, and all the attacker needs to do is repeated decryption and reencryption of the messages intercepted.

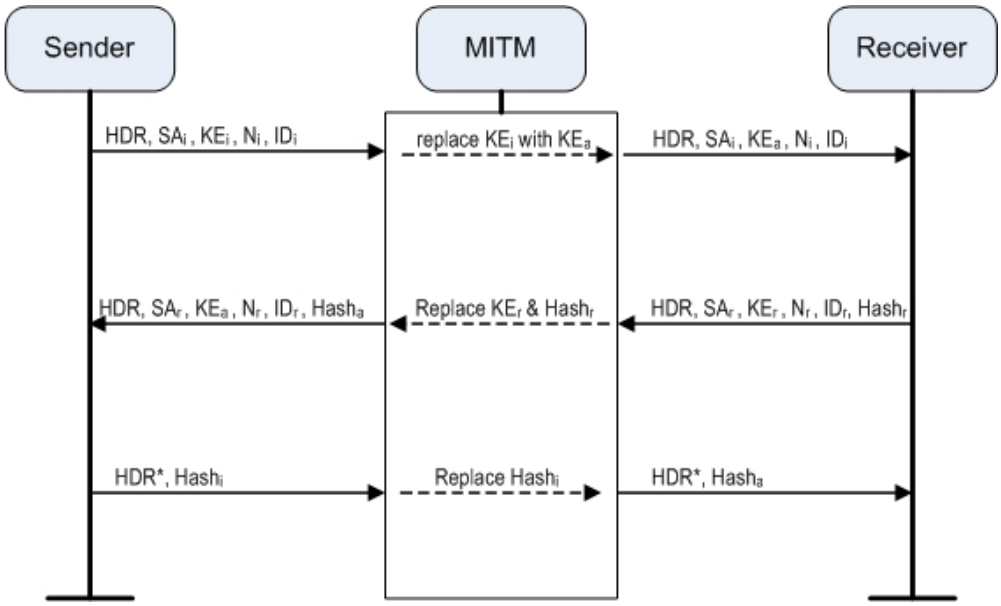

Figure 6.1: MITM Attack to Compromise a Connection in the Aggressive Mode

6.3 Attacking the Main Mode

For the Main Mode exchange, it is more difficult to discover the preshared key value, yet it is not theoretically difficult. This results from the fact that the hashes, HASH–I and HASH–R, in the Main Mode are encrypted before being sent (see the last two messages of Figure 5.4) in contrast to the Aggressive Mode where the HASH–R value is sent in the clear.

To overcome this problem, the attacker must fool one peer (usually the initiator) using MITM attacks in order to trick that peer into encrypting the authenticating hash using the shared information with the attacker not the responder. This way the attacker can compromise the preshared key. This is shown in Figure 6.2. The attack begins when the attacker intercepts the fourth message from responder to initiator that contains the Diffie–Hellman public key of the responder, g^r. The attacker changes that key to his/her public key, g^a, and sends the message to the initiator. The initiator thinks this message is from

the responder because up to this point authentication has not been started. As a result, the initiator computes the four secrets SKEYID, SKEYID_d, SKEYID_a, SKEYID_e, and the authenticating hash, HASH–I, based on the shared information with the attacker, g^{ia}, and not that of the responder. Then, the initiator sends its ID_i and authenticating hash, HASH–I, encrypted with a key that is derived from SKEYID_e. The attacker intercepts this encrypted message and starts an off–line dictionary attack (or a brute–force attack) on the preshared key by enumerating all the candidates for the preshared key, and for each one computes:

$SKEYID_{new} = PRF(guessed\,key,\ N_i \mid N_r), and$

$SKEYID\_d_{new} = PRF(SKEYID_{new},\ g^{ia} \mid CKY\text{–}I \mid CKY\text{–}R \mid 0), and$

$SKEYID\_a_{new} = PRF(SKEYID_{new},\ SKEYID\_d_{new} \mid g^{ia} \mid CKY\text{–}I \mid CKY\text{–}R \mid 1), and$

$SKEYID\_e_{new} = PRF(SKEYID_{new},\ SKEYID\_a_{new} \mid g^{ia} \mid CKY\text{–}I \mid CKY\text{–}R \mid 2), and$

$HASH\text{–}I_{new} = PRF(SKEYID_{new},\ g^i \mid g^a \mid CKY\text{–}I \mid CKY\text{–}R \mid SA\text{–}offer \mid ID_i).$

Now, armed with $SKEYID\_e_{new}$, the attacker decrypts the intercepted message and obtains the HASH–I value that was sent by the initiator. If the new computed hash value equals the decrypted hash value, i.e. if $HASH\text{–}I_{new} = HASH\text{–}I$, then there is a very high probability that the guessed key equals the preshared key due to the one–way property of hash functions.

After compromising the preshared key, the same MITM attack used in the Aggressive Mode (see Figure 6.1) can be used to compromise the shared Diffie–Hellman key and all future connections between the two peers. Note that in the Main Mode two MITM attacks are required: one to compromise the preshared key, and one to compromise the Diffie–Hellman shared key, while only one MITM attack is required in the Aggressive Mode. Note also that after sending the fifth message containing its ID and authenticating hash, the initiator will not receive a reply because the message will fail the authentication

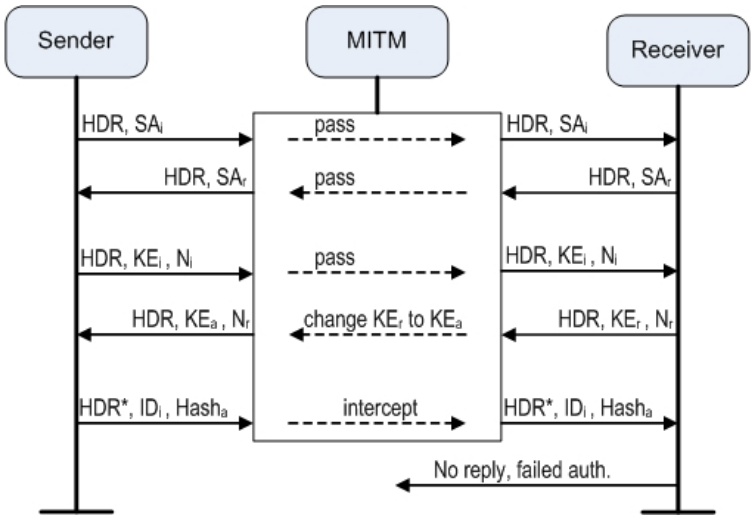

Figure 6.2: MITM Attack to Compromise the Preshared Key in the Main Mode

process at the responder since parameters shared with the attacker rather than the responder are used in computing the authenticating hash. Therefore, the initiator assumes that the message is lost and initiates a new connection with the responder. This shows that it is difficult to detect such attacks because the initiator has no way of telling whether it was communicating with a legitimate responder or an attacker.

6.4 Cracking Speed of the IKE Preshared Key

The preshared key authentication method is subject to off–line dictionary or brute–force attacks through which the preshared key is compromised. In this section I will briefly introduce the Data Encryption Standard (DES) cracker machine [10] which is used for calculating the time it takes to compromise the preshared key value. The performance of the DES cracker machine in terms of hashing speed is computed. Based on that speed, the time required to compromise different preshared key lengths is calculated.

Aggressive Mode exchange is used in order to determine the time required to compromise the preshared key value. In the Aggressive Mode an attacker can intercept all

the packets exchanged between any two communicating parties (initiator and responder). These packets contain all the necessary parameters required for computing the SKEYID–based encryption and authentication keys except the preshared key itself. However, among the intercepted parameters is the HASH–R value, which is a function of the preshared key. Two hash algorithms (MD5 and SHA–1) might be used to compute the HASH–R value. With this hash value in hand, an attacker can apply an off–line dictionary attack (or brute force attack) against the authenticating hash value until the preshared key is determined. Once the preshared keys is determined, the remaining secrets (the encryption and authentication keys), which depend on the preshared key among other parameters, can be compromised using the MITM attack explained in Section 6.2.

The DES cracker is a very effective machine built by the Electronic Frontier Foundation (EFF) to perform a brute–force search of DES's key space. It decrypts an encrypted message by trying every possible key until the right key used in the encryption process is successfully determined. The aim in doing this is to prove that the DES's key is not long enough to be secure. The same machine, with its huge processing power and high degree of parallelism, can be also used to show that the preshared key used for authenticating the phase one exchange of the IKE protocol is insecure. Most of the description found below about the DES cracker machine is taken from the book "Cracking DES - Secrets of Encryption Research, Wiretap Politics and Chip Design" [10].

The DES cracker machine was built in 1998 by the EFF. It cost about $200,000 to build. It consists of an ordinary personal computer connected with a large array of custom chips. Software in the personal computer instructs the custom chips to begin searching. The strength of the machine is that it replicates a simple but useful search circuit thousands of times, allowing the task of searching a given key space to be distributed among the search circuits. This means that the problem of searching the given key space (such as

brute–forcing a preshared key) can be usefully solved by many machines working in parallel. For example, a single DES cracker machine can find the key in a certain amount of time. A thousand DES cracker machines can solve the same problem in one thousandth of the time. A million DES cracker machine could theoretically solve the problem in about a millionth of the time.

The DES cracker machine built in 1998 was able to perform 250,000,000,000 DES decryption operations per second [10, 40]. The machine is composed of 1856 custom DES chips, housed on 29 circuit boards of 64 chips each. The speed of each custom chip was 40 MHz.

In attacking the preshared key, hashing operations are used rather than DES decryption operations. In order to compare the performance of the DES operations with the performance of the hashing operations two simple performance tests have been conducted. In fact, similar tests were conducted by Roe in 1993 and published in his paper "Performance of Symmetric Ciphers and One–Way Hash Functions" [41]. However, the paper is dated being more than 10 years old and processors have increased in performance enormously since that time. Therefore, we considered making a run of a similar task on a current generation machine.

The first test is for the MD5 and the SHA–1 hashing algorithms. It measures the time taken to hash a 128 character message. The length is chosen to reflect a typical length of a Diffie–Hellman shared secret that is hashed to generate a new preshared key (see Section 7.4 for details on generating the new preshared key). Typically, the Diffie–Hellman key length is 1024 bits which is 128 bytes worth of data. The test is implemented by clearing a 128 character buffer, and then 100,000 times calling the routines to start hashing, hash the buffer, and finish hashing. The measurements were carried out on a 500 MHz HP/Compaq True64 Unix system 200 times to produce an average value. It

takes 0.334961 seconds on average to hash the 128 buffer 100,000 times using the SHA–1 algorithm, and 0.274414 seconds on average to hash the same buffer 100,000 times using the MD5 algorithm. Using these results, one can calculate the rate at which hashing operations can be performed per second. Table 6.1 shows such a rate for both the MD5 and the SHA–1 algorithms.

Hash Algorithm	**Speed (hashes/second)**
MD5	364412
SHA–1	298542

Table 6.1: Average Hash Algorithms Performance on a Tru64 500 MHz Machine

The second performance test measures the performance of the DES decryption operation on the same machine. It measures the time it takes to decrypt a single data block under 100,000 different keys. This situation is similar to the way the DES cracker machine works. The DES cracker machines tries to decrypt a ciphertext block using a different key in each trial. The size of the block is set to 64 bits (8 bytes). In each run of the test, the routines to initialize the decryption process, decrypt the block, and finish decryption are called. To get an average value, the test was run 200 times. It takes 7.373047 seconds on average to decrypt an 8–byte block 100,000 times on a 500 MHz HP/Compaq True64 Unix system. Table 6.2 shows the rate per second at which decryption operations can be performed.

Encryption Algorithm	**Speed (decryptions/second)**
DES	13562

Table 6.2: DES Decryption Performance on a Tru64 500 MHz Machine

The hashing speed using the DES cracker machine can be estimated based on the above results. The MD5 algorithm is almost 27 times faster than DES decryption, and the

SHA–a algorithm is almost 22 times faster. Therefore, the DES cracker machine is expected to perform $250,000,000,000 * 27 = 6,750,000,000,000$ MD5 hashing operation per second and $250,000,000,000 * 22 = 5,500,000,000,000$ SHA–1 hashing operation per second. These figures are used in all subsequent calculations to determine the required time to compromise different preshared key lengths using a brute–force attack.

The possible set of characters a byte in the preshared key can take is variable. It depends on the person that sets up the preshared key. It might be 26 characters (upper or lower characters). It might be the 52 combined upper and lower characters. Another option for the character set is the combination of upper, lower, and numeric characters. This set is 62 characters in size. If all printable characters are considered, the set is composed of 96 characters. On the other hand, if all the ASCII characters are to be considered, the size of the character set is 128. However, in practicality a preshared key depends on the interface through which the key is set up. The interface might limit the number of characters a user can input for a key. A good example of that can be found in the paper by Mudge [42]. It shows an example of a Windows NT key that is limited to 14 characters by the interface. Furthermore, the preshared key length is limited by the keys found on a regular keyboard when being setup. Therefore, theoretically a preshared key can have any ASCII character, but practically it is not the case.

In addition to that, no one can exclude any of the above possible sets and consider the set of ASCII characters only. All of them are valid according to the IKE standard specifications. There are no requirements on the set of possible characters a preshared key can take in the IKE RFC's [6, 31] nor in the IKEv2 RFC [43]. The only requirement is found in the IKEv2 that requires the preshared key length to be equal to the length of the output of the underlying hash function(can be 16 bytes if MD5 is used). If that key is composed of lower case and/or upper case characters only, it is insecure.

Character Set	Preshared key Length	Brute Force Attack Timing	
		SHA–1	**MD5**
Upper or Lower Case	8	0.019 sec	0.015 sec
	10	12.83 sec	10.45 sec
	12	2.41 hours	1.96 hours
	16	125.71 years	102.43 years
Upper and Lower Case	8	4.86 sec	3.95 sec
	10	3.65 min	2.97 min
	12	1.12 years	0.91 years
	16	$8.2 * 10^6$ years	$6.7 * 10^6$ years

Table 6.3: Brute Force Timing in the IKE Protocol (U and/or L Case Letters)

Users and system administrators tend to use passwords, without looking into their lengths or degree of randomness, as preshared keys. In fact, in IKEv2 specifications it is acknowledged that the preshared key can be of any length (see Section 2.15 in RFC 4306 [43]), and that it is common to derive preshared keys from a user–chosen password without incorporating a source of randomness in the key. Therefore, a 17 ASCII character string is padded to the preshared key before using it according to the IKEv2 specifications. The pad string is added so that if the shared secret is derived from a password, it adds some unpredictability to it so that it can resist off–line or social engineering attacks. However, the specifications do not state how this padding string is generated or distributed. It is assumed that the padding string is there and ready to be used. Practically, if this string is dynamic, generating and distributing it brings us back to the same problem of generating and distributing the preshared key securely. On the other hand, if it is static, it must be known to every one in advance. In this case, the padding string is useless from a security point of view since it is well known to the peers, public, and the outside world.

Table 6.3 shows the brute–force attack timing required to compromise different preshared key lengths when using the DES cracker machine. The calculations in this table are based on the assumption that the possible set for a character in the preshared key is

Character Set	Preshared key Length	Brute Force Attack Timing	
		SHA–1	**MD5**
Printable ASCII (96 char.)	8	10.93 min.	8.90 min.
	10	69.95 days	56.99 days
	12	1761.43 years	1435.24 years
	16	$14.9 * 10^{10}$ years	$12.1 * 10^{10}$ years

Table 6.4: Brute Force Timing in the IKE Protocol (Printable ASCII)

the set of upper and/or lower case characters only. Table 6.4 shows the brute–force attack timing using the DES cracker machine when the set of printable characters is considered to be the set of possible characters for a preshared key character. An entry in the above tables is computed as follows:

$$Attacking\ Time = 0.5 * \frac{set\ of\ possible\ characters^{preshared\ key\ length}}{hashing\ speed}$$

For example, the brute–force attacking time of a preshared key that is 8–lower case characters hashed with SHA algorithm is:

$$Attacking\ Time = 0.5 * \frac{26^8}{5.5 * 10^{12}} = 0.019\ sec$$

.

When conducting a brute–force search, the right preshared key might be found in the first thousand tries, or it might be found in the last thousand tries. On average, the preshared key is found after trying half the possible values. Therefore, the timing for the brute–force search is generally given as the average time to find a key. The maximum time is double the average time.

It is clearly seen that the preshared key could be compromised. Even though in some cases the time required to compromise the preshared key is long, it is still feasible to crack the key for four reasons. The first is that the preshared key is static. This means that its lifetime is infinite, which imposes no limit on the time frame during which the attacker

must apply or launch the attack. Consequently, the attacker has an infinite time through which he/she can try every possible value the preshared key can take, and eventually he/she will successfully compromise the preshared key.

The second reason is that the results in the table are based on the DES cracker machine, which was built in 1998 with a processor speed of 40 MHz. Due to huge improvements in the processing power and speed since that time, it would definitely take much less time to compromise the preshared key using the current technology. Current chips are much faster than the ones used in the original DES cracker machine design. Modern processor chips can run at more than 3.6 GHz, ninety times faster than the original design.

Third, the DES cracker machine is used here only to give the reader an approximate bound on how long it would take to crack a preshared key hash. However, other ”cracking” systems have and continue to be developed whose performance will outperform the DES cracker machine in many times. This would result in reducing the time it takes to crack the preshared key. Thus, the time bound in the Table 6.3 and the Table 6.4 shows the possibility and the feasibility of cracking the preshared key, not the exact time it takes to do so. Newer improved machines will definitely increase the risk of compromising the preshared key.

Finally, by deploying more machines in a distributed environment to work on the preshared key search in parallel, the collective work of these machine will reduce the time needed to crack the preshared key further. Theoretically, there is no limit on the number of DES cracker machines that can be put together to work collectively on compromising the preshared key except the cost associated of building and putting these machines together. Recall that the DES cracker machine cost about $200,000 back in 1998. However, taking into consideration that the cost of hardware is getting lower and lower (along with increased processing power), the cost factor might not be an issue in the near future, espe-

cially if we are talking about governmental institutions, not normal users, who are willing to spend millions of dollars to compromise that essential preshared key. Therefore, combining the infinite preshared key lifetime with more processing power for each machine in a large distributed environment using contemporary up–to–date machines would definitely reduce the time to compromise a preshared key dramatically.

The above analysis shows that the preshared key can be compromised. One might suggest using strong preshared keys that are difficult to be guessed or that are not found in a dictionary. This might increase the time and effort to compute that preshared key. However, once it is computed, it is effortless to compromise all subsequent communications because the same preshared key is used again and again to authenticate the phase one exchange and establish the phase one SA, which protects all subsequent IPsec communications. Therefore, using the IKE preshared key authentication method in its current design is not safe. We need to find a remedy to overcome this weakness. The next chapter proposes and discusses a modification for the IKE phase one exchange to enhance its preshared key authentication method.

Chapter 7

Enhancing the IKE Preshared Key Authentication Method

7.1 The Problem

The analysis of the preshared key authentication method in Chapter 6 shows the importance of having a strong authentication service in IKE phase one communications. Without being able to authenticate the peer at the other end, the SAs and session keys established are suspect. One can not trust an identity's identification without being authenticated. Even though encryption (ESP) and integrity (AH) services will protect subsequent communications in phase two from being compromised, it is possible, without strong authentication, that the SA and the session key may have been established with an attacker who performed a MITM attack in phase one and is now revealing all data communicated in phase two. Thus, depending on a weak phase one authentication followed by very strong authentication and encryption in phase two can not prevent all communications from being compromised. Consequently, in order to secure phase two data communications, the system must provide very strong phase one authentication that can not be compromised easily.

7.2 Performance Analysis of the Preshared Key Authentication Method

Authentication in IKE Phase one exchanges can be achieved using preshared keys or using public key cryptography (RSA signatures or digital signatures). Based on the analysis in Chapter 6 the preshared key authentication method can not be considered to be secure. It is subject to off–line dictionary or brute–force attacks.

A direct and simple solution to this security risk is to avoid using the preshared key authentication method. Instead, one recommendation is to use methods that depend on public key cryptography, such as RSA signatures. These algorithms are known for their strength from a security point of view. However, when it comes to performance, the public key cryptography methods suffer from the fact that they are slow.

To give an exact relation between the performance of the preshared key authentication method and the performance of the public key authentication methods, the time it takes to finish the IKE phase one exchange using both methods is measured. Only phase one timing is considered in this measurement since authentication occurs in this phase. By determining the time required to finish the phase one exchange it is clear which authentication method is faster and more effective for the use of the phase one authentication.

The hardware required to run the test which measures phase one timing is composed of two Windows–based systems. One is a Windows 2000 machine. The other is a Windows XP machine. After installing the IPsec software on the Windows 2000 machine, since it is not installed by default, both systems are configured to use the preshared key authentication method for one set of tests and the authentication using RSA signatures for a second set of tests [44, 45]. RSA signatures require the use of certificates. A certificate is a file that binds a system's identity to its associated public keys enabling the system to send encrypted and digitally signed electronic messages.

The certificate identifies the system and is required to verify its digital signature. Each certificate contains a public key, along with other information such as the owner's common name and the certificate's expiration date. Information is verified in the certificate by relying on a trusted third party called a Certificate Authority (CA). Each certificate is signed with the private key of the CA. Moreover, the CA's authenticity is verified by its certificate, which is generally available to the public. The CA certificate must be generated before any other certificates can be generated. Moreover, the certificate must be installed on both systems in order to sign and verify other client certificates. Certificates for the CA and the two systems are generated locally using the *openssl* utility program [46]. See openssl documentation [47] and Hirsch's paper [48] to see how to setup your own CA and generate client certificates.

Table 7.1 shows the time measurements obtained by capturing the packets transmitted between the two Windows–based systems in each authentication method. The table shows the average values of measuring the phase one exchange timing five times a day for a period of two weeks. Packets were captured using the Ethereal tool [49]. Ethereal is a free network protocol analyzer for UNIX and Windows. It allows a user to intercept and display packets being transmitted or received over a network to which the computer is attached. It prints out many details about the packets intercepted through its Graphical User Interface (GUI). For example, it shows the content of each packet, time of transmission or interception, source address, destination address, and much more. For more information about Ethereal see the paper by Hards [50].

The table shows that the time required to finish the phase one exchange using the RSA signature method is much more than the time required to finish the phase one exchange using the preshared key authentication method. It takes almost seven times as long to finish the phase one exchange using the RSA signature. Therefore, authentication using

Trial No.	RSA Signature	Preshared Key	% (RSA/Preshared)
1	2.5838	0.3883	665.41 %
2	2.5609	0.3846	665..86 %
3	2.5242	0.3676	686.67 %
4	2.5178	0.3616	696.29 %
5	2.7283	0.3930	694.22 %

Table 7.1: Time to Finish Phase One Using Different Authentication Methods

preshared keys is much less time consuming than using the RSA signatures. The next scenario illustrates how this might be helpful in improving the performance of systems using IPsec for protecting their communications with other systems.

Considering that thousands of clients (initiators) might try to initiate a connection with a certain machine (responder) per second, the time to finish phase one becomes a crucial factor in determining the overall system performance. For example, assume that we have a web server that is setup to use IPsec to protect its traffic. If we have thousands of users who are trying to access some web pages on that server from different systems, these systems must be authenticated. Using the RSA signatures for the authentication requires all user systems to have public and private keys assigned to them which might be difficult or unaffordable. Furthermore, it takes much more time and processing power to authenticate these systems using the RSA signatures authentication method. Using the same hardware setup, processing power, and the preshared key authentication method, the same web server can theoretically support a number of users that is 7 times greater than the number of users supported in the RSA signature authentication method. In addition, taking into consideration that the visits to the web server are usually of short duration, after which the SA is discarded to claim its reserved resources, a user must be reauthenticated each time he/she accesses the web server again.

Since the authentication process can possibly run thousands of times each second (depending on the number of users), if we can save some of the time spent in the au-

thentication process itself, that saving is replicated thousands of times. This opens the possibility to do more useful work and serve more users using the same underlying hardware setup without any associated cost. Furthermore, it enhances the response time and the throughput of the server system.

Therefore, from a performance point of view, the preshared key is the preferred phase one authentication method to use. However, since this method is subject to off–line dictionary or brute–force attacks, it can not be used securely. Thus, without overcoming its vulnerability, the preshared key authentication method is unusable.

7.3 Proposed Enhancement to Secure the Preshared Key Authentication Method

To utilize the preshared key authentication method performance on a system without risking the system's security, the method must be modified to resist the brute–force and off–line dictionary attacks. This can be achieved by limiting the preshared key lifetime. That is, use a dynamic preshared key instead of the static one currently used in the IKE. By doing so, we impose a time bound to limit the amount of time an attacker has to successfully obtain the preshared key value.

Without the time limit on the life of a preshared key, an attacker has an indefinite time frame through which he/she can apply the off–line dictionary attack (or brute–force attack). Because of this indefinite time the attacker can eventually compromise the preshared key. After that it becomes easy to compromise all future connections. Thus, all it takes to compromise a connection is one successful attack against the preshared key that can be applied at the leisure of the attacker. However, by introducing a limited life time on the preshared key, an attacker must compromise the preshared key within the time bound. Otherwise, the efforts and time spent on compromising a preshared key is useless

after that key is replaced with a fresh, new one to be used in the authentication process of the next phase one exchange. This introduces a new dimension on the preshared key security. Besides the possibility of having a longer key length, the key is valid only for a short period of time. This limits the vulnerability period to a maximum or hours or minutes rather than years.

The proposed enhancement in this dissertation does exactly that. It is very simple and effective. Instead of using the same constant preshared key over and over in each authentication process of the phase one exchange, a new, different preshared key is used each time. This new key is generated after the end of a successful phase one authentication process to be used in the next phase one exchange authentication process. This enhancement imposes a time limit on the liveness of the preshared key and makes it a frequently changed, dynamic, one–time key.

The maximum lifetime of the preshared key is either the lifetime of the current session in which the key is used or the default lifetime designated by an IPsec implementation for the phase one SA if the session is longer than that (the default value is eight hours [51]). Thus, by using this proposed enhancement, it is guaranteed that the maximum time allowed for an attacker to compromise a preshared key is eight hours or less. This time is very short based on the processing power available nowadays and in the near future (see Table 9.1 to see the resistance of the proposed enhancement to the off–line dictionary and brute force attacks).

The proposed enhancement provides another level of security in terms of the amount of information leak that can happen if a preshared key is compromised. This is due to the limited lifetime a preshared key can have in the proposed enhancement compared to the infinite lifetime of the permanent preshared keys in the original IKE protocol. This short lifetime ensures that even if an attacker is able to compromise the communication

protected by one preshared key, he would have access only to the information that was protected and exchanged while using that preshared key. New preshared keys used in consequent sessions will protect the information that is sent in the future. That is, only the session that used the compromised preshared key *might* be compromised. Future sessions will not be compromised depending on the compromised preshared key since they are using new, fresh and different preshared keys in their communications. Furthermore, if the time required to break one preshared key (within the time limit of the eight hours) exceeds the useful lifetime of the information exchanged, this scheme of one–time preshared keys becomes very valuable.

7.4 New Preshared Key Generation

The new preshared key is generated as a function of the key used to authenticate all phase one messages (SKEYID_a), the established Diffie–Hellman secret, which is randomly generated in each session, and a master key known only by the initiator and the responder. The new preshared key is generated after the current phase one exchange finishes successfully. When the next phase one exchange starts, it uses this newly generated key as the preshared key for authenticating the exchange.

The master key is used to prevent an attacker from generating the new preshared key in the same way it is generated at the initiator or the responder. It is always possible, in the worst case scenario, that the preshared key (the currently used one) is compromised even in the short lifetime of eight hours imposed by the proposed enhancement (or any value less than that setup by a system administrator). In this case, an attacker can launch a MITM attack against the connection between the initiator and the responder as described in Section 6.2. If the master key is not used, the initiator and the responder will compute the new preshared key after the end of the compromised session based on the Diffie–

Hellman shared secret and the SKEYID_a value that are shared with the attacker not between them. At the same time, the attacker can compute the new preshared key in the same way since he/she knows the parameters that are used in the computation. Therefore, the attacker needs to compromise only one preshared key before he/she can generate all future preshared keys and compromise all subsequent communications between the initiator and the responder.

For this reason the master key is used. Assuming in the worst case that a preshared key *might* be compromised during the life time of a session (maximum of eight hours), the attacker will be only able to compromise information exchanged in the current session only. By the end of this session, a new preshared key will be in place for the use of the next phase one session. This new preshared key is generated by a secret known only to the initiator and the responder (the master key). Therefore, the attacker will not get the new preshared key based on compromising the current one, and he/she has to start the attacking process against the new preshared key again in order to compromise the new session.

The master key is setup the first time a preshared key is setup. This master key is used again and again in the computation of the preshared key. It is hashed with the negotiated hash function, and the resulting hash value is concatenated with the current Diffie–Hellman shared secret to form a message whose HMAC value is computed using the SKEYID_a value as the key for that HMAC operation. That is, the new preshared key=HMAC(SKEYID_a, Diffie–Hellman shared secret | hash(master key)), where | is the concatenation operation.

The method of generating the new preshared key is not different whether using the Main Mode or the Aggressive Mode in the phase one exchange. Generating the preshared key is supported in both modes in the same way. The length of the generated preshared

key depends on the output length hash function used. For example, it is 20 bytes if the SHA-1 algorithm is used. However, the minimum length generated is 16 bytes when MD5 is used.

There is no security risk in using the master key over and over in the new preshared key generation since neither the master key itself nor any data directly related to it is exchanged in any part of the communication. To add more security to the master key, its hash value, using the negotiated hash function, is used in the computation of the new preshared key. Furthermore, this master key can be changed according to some policy set between the initiator and the responder to insure even more security.

7.5 New Preshared Key Synchronization

In order for the proposed enhancement to work correctly, we must insure that the initiator and the responder do not go out of synchronization in terms of the preshared key. The generation and the updating of the preshared key must be done consistently in a correct and timely fashion at both peers. Otherwise, the generated preshared keys in the initiator and the responder might be different.

The Transmission Control Protocol (TCP) [2] is used to insure synchronous key updating. It provides a reliable, connection-oriented, byte stream, transport layer service. *Reliability* is the main feature of TCP that makes it a straight forward candidate for implementing the proposed enhancement. In general, TCP sets a timeout anytime it sends data, retransmits data if it is lost, acknowledges data received by the other end, and reorders out-of-order data. This insures that data received by the other end is free of errors and in the correct order or the connection times out.

The deployment of the new preshared key is a two–step process. The key must be generated. Then the old preshared key (the one currently used) must be replaced (updated)

with the new generated preshared key. Key generation is explained in Section 7.4. In this section we describe how the update process of the preshared key happens in order to keep the preshared keys synchronized at both ends of the communication. Preshared keys are saved in local files. Refer to Section 7.6 for discussion about securing these files from unauthorized access.

Generation of the preshared key is not started until after one endpoint authenticates the other endpoint. Otherwise, a denial of service attack is possible. An attacker sending many bogus IKE messages will force an endpoint to update its preshared key. When these messages fail the authentication, the endpoint has to recover from that by resetting the preshared key to the old value that was used before the updating process occurred. This consumes the processing power of the endpoint and prevents it from doing other useful work. This attack can be avoided if generating and updating of the preshared key happens after the authentication succeeds.

7.5.1 Aggressive Mode Synchronization

Generating and updating the preshared key in the Aggressive Mode (see Figure 5.5) starts after the initiator authenticates the responder in the second message. As a result, the initiator computes the preshared key as explained in the Section 7.4. Furthermore, the initiator updates the preshared key assuming that the phase one exchange will finish successfully. This assumption is based on the increasing reliable services today's networks are providing. At the same time, the initiator retains the old preshared key to reset the preshared key file in case the phase one exchange does not finish successfully or to use it in case the two endpoints of the communications go out of synchronization in terms of the preshared key. By sending the third message, the initiator informs the responder that it has updated its preshared. It waits for that message to be acknowledged by the responder.

If the responder did, it means that the message has been received by the responder which will generate and update its preshared key file.

If the message is not delivered to the responder or no acknowledgment is received back from the responder (after some number of retransmissions from the initiator side), the TCP protocol on the initiator's side will time out and report that the responder is not reachable. Since the responder does not update its preshared key because of the loss of the third message, the initiator has to recover from updating its preshared key by resetting the preshared key file with the old preshared key that was retained previously. This shows the importance of implementing the proposed enhancement over the TCP protocol. It provides a reliable transport service that can inform an entity whether a message has been successfully transmitted or not.

In the rare case that the third message reached the responder and got acknowledged, but the responder crashed before updating its preshared key, the next phase one exchange between the initiator and the responder will fail because they are using different preshared keys. The initiator uses the newly generated preshared key while the responder uses the old preshared key because it was not physically written to the disk.

Instead of reporting that directly to the system administrators to set up a new preshared key manually, which might take a while and requires the intervention of humans, the initiator reinitiates the phase one exchange using the retained (old) preshared key. This key is the same at both endpoints of the communication because it was used in a successful authentication in the previous phase one exchange. Therefore, there is a great chance that the exchange succeeds in this time. At this point, a new preshared key is generated and updated in the same way described above. However, if the authentication process fails again, then there is a good reason to get the system administrators involved in setting a new preshared key between the initiator and the responder.

Trying to authenticate the phase one exchange again using the retained (old) preshared key must happen only once and within a very short duration (seconds) from the failure of the first try of the phase one exchange so that an attacker does not benefit from having two consecutive phase one exchange using the same preshared key. Note that this solution requires the most recent preshared key to be saved in addition to the newly generated preshared key. See Section 7.6 for the security of the files containing these keys.

The loss of the first message in the Aggressive Mode is recovered from easily. All it needs is to reinitiate the connection again since the preshared key has not been generated or updated at this point. Likewise, the loss of the second message can be recovered from in the same way and the two peers will not be brought out of synchronization in terms of the preshared key. However, if the loss of the second message is a result of an attacker intercepting the message and forbidding it from reaching the initiator, he/she might benefit from that.

The attacker can launch an off–line dictionary or brute–force attack against the preshared key in the same way described in Section 6.2 by intercepting the second message. During that time, the initiator and the responder might try to reconnect again using the same preshared key the attacker is trying to compromise. If they succeed before the key is compromised, the attacker will not benefit from trying to compromise the preshared key because by the end of the session a new preshared key will be in place to be used in the next session. However, if the attacker keeps forbidding the initiator and the responder from completing the phase one authentication until the preshared key is compromised, he/she can launch the MITM attack after the key is compromised as described in Section 6.2 . However, in this case the attacker will have access only to the information exchanged in this session and will not be able to compromise future connections using the compromised preshared key since by the end of the current session a new preshared

key will be in place that the attacker can not compute (see Section 7.4 for more details).

Note that in this case (a successful MITM attack) the initiator and the responder will go out of synchronization. The computed preshared keys at both ends will be different since the Diffie–Hellman shared keys used in the computation are not the same (the initiator shares a Diffie–Hellman shared key with the attacker that is different from the Diffie-Hellman shared key between the attacker and the responder). The next phase one between the initiator and the responder will fail, and that would be reported to the system administrators to take action (most probably resetting the preshared key manually).

A simple solution to prevent this situation and forbid an attacker from having access even to one session can be enforced. Note that the time it takes to compromise the preshared key is not usually short. During that time the initiator and the responder will try many times to establish a connection. If the connection fails a certain number of times successively, for example 10 times, the initiator reports that to the system administrators who should become suspicious and most probably need to establish a new preshared key with the responder manually. This way, we do not give the attacker the chance to compromise the preshared key and launch the MITM attack against the connection because the key being compromised is useless after the system administrators update their keys.

7.5.2 Main Mode Synchronization

The loss of the four first messages in the Main Mode (see Figure 5.4) can be easily recovered from. These messages do not trigger generating or updating the preshared key at either endpoint of the communication. Therefore, it is just a matter of retrying the connection when one of these four message is lost. Recall that when a message is lost or not acknowledged in the TCP protocol, the connection terminates. Thus, the connection has to be reinitiated again to establish a connection.

However, if the loss of the fourth message is caused by an attacker who is trying to apply the attack described in Section 6.3, he/she might succeed if the initiator and the responder could not establish the connection for a time longer than the one needed to compromise the preshared key. The attacker can enforce this situation by forbidding the connection establishment between the initiator and the responder as long as the preshared key has not been compromised. Once the key is compromised, the attacker permits the fourth message, and all other messages, to finish in order to apply the MITM attack described in Section 6.3.

A possible remedy for this situation is to warn the system administrators after failing to establish the connection for a predefined number of times successively, say 10 times. The system administrators should act upon this warning by behaving conservatively through resetting the preshared key between the initiator and the responder. The time it takes to try reconnecting a predefined number of times is much shorter that the times it takes to compromise a preshared key. Therefore, the attacker will not benefit from compromising the preshared key. In the worst case, the attacker will be able to compromise only one session if he compromised the key before the system administrators reset the preshared key. Future communications will be protected by new preshared keys as described in Section 7.4 and Section 7.5.1.

Generating and updating the preshared key in the Main Mode starts at the responder after receiving the fifth message. This message is supposed to authenticate the initiator to the responder. If the message fails to get to the responder, the initiator will retransmit it until it gets to the responder or until the connection times out. If the connection times out, the phase one exchange starts from the beginning with no risk of bringing the two endpoints out of synchronization since the new preshared key has not been generated or updated yet at either side of the communication.

If the fifth message reaches the responder and passes the authentication, the responder generates a new preshared key as explained in the Section 7.4, updates its preshared key file, and retains the old preshared key for recovery purposes. Then, it sends the sixth message to the initiator. If the message fails to get to the initiator, the responder keeps retransmitting it until it is received or until the connection times out. If the connection times out, the responder recovers by using the retained (old) preshared key to reset its preshared key file and exits. If the message reaches the initiator, the initiator generates the new preshared key and updates its preshared key file.

In the rare case that the initiator crashes before it updates its preshared key file, the two peers will be out of synchronization. When the two peers try to establish a connection after that, the authentication will fail. A similar solution to the one described in Section 7.5.1 can be used. The initiator reinitiates the phase one exchange using the retained (old) preshared key. There is a good chance that the exchange finishes successfully this time because the old preshared key was successfully used in the previous phase one exchange. If it does, a new preshared key is generated and the two endpoints are brought to synchronization again. However, if the phase one exchange fails again, it is reported to the system administrators. They will probably need to reset the preshared key manually before the connection can be successfully established.

7.6 Securing the Key Files

Local files at each endpoint of the communication are used to save the values of the old preshared key, the newly generated preshared key, and the master key. The old preshared key is used in authenticating the endpoints and bringing them to a synchronized state in terms of the preshared key is for some reason they become unsynchronized.

The new preshared key is used to authenticate the next phase one exchange. It is

generated and updated after the end of the current phase one authentication. The master key is shared only between the initiator and the responder. It is used to insure that the newly generated preshared key can be computed only by the initiator and the responder (see Section 7.4 for more information about the master key).

It is assumed during this dissertation that the files that contain these key are protected from unauthorized access. The keys are very sensitive data and must be protected. Otherwise, an attacker does need to launch any sort of off–line (or brute–force) attacks in order to compromise the keys. If an attacker can gain access to these keys, he/she can apply the MITM attack described in Section 6.2 easily. Therefore, compromising all connections between the initiator and the responder becomes easy and effortless.

Therefore, the contents of these files must be protected independently of the protocol or the environment they are using in. The files might be encoded in some way such that even if an attacker gains access to their contents, he/she benefits little, if any. For example, system administrator's public/privae strong encryption keys can be used to encrypt the contents of the files on each system to keep attackers from getting the preshared key even on a public system.

7.7 Compatibility With the Original IKE Protocol

The proposed enhancement is not compatible with the original IKE protocol version 1.0. Peers which implement the enhanced version of the IKE protocol (the proposed enhancement) will need to take a different version number than the one currently used in the IKE protocol. It might take the version number of 1.1. However, it is likely that there will be a transition period in which some implementations will want to support both versions in the future. This section explains a way that can be used for the two versions to work together.

Whenever an endpoint receives an offer to use version number x of the IKE protocol,

it means for the receiver that the sender supports all previous versions of the protocol up to x. For example, when an initiator offers version 1.1 to a responder, the responder can choose either version 1.0 or version 1.1 since both are supposed to be supported by the initiator.

When the initiator negotiates the security parameters with the responder during the transition period, it offers the responder the highest version number of the IKE protocol it can support. Suppose the initiator offers version 1.1. When the responder receives that offer, it checks the version number to see whether it supports it or not. If it does support the enhanced version of the IKE protocol (version 1.1), it replies with that version number in the SA reply. This tells the initiator that the responder supports the enhanced version. Therefore, both peers will use the enhanced version of the IKE in their negotiation and will generate a new preshared key for use after the current session finishes.

If the responder supports only version 1.0 of the IKE protocol, it will reply with that version in the SA payload sent back to the initiator. In this case, the initiator knows that the responder supports only version 1.0. Therefore, the initiator will not generate a new preshared key even if it can do that since the other peer can not. If the initiator supports only version 1.0 of the IKE, the responder has no choice but to use that version even if it supports version 1.1.

The ability to support both versions of the IKE protocol, the original IKE protocol and the proposed enhancement, might open a possibility for an attacker to defeat the security gain of the proposed enhancement. An attacker can force the communicating peers to scale down to use the original IKE protocol even though both peers support the enhanced version. This is explained in Section 7.8.

7.8 Security Analysis of the Proposed Enhancement

In order for an attacker to compromise an IPsec connection, the SKEYID–based secrets and the Diffie–Hellman shared secret must be compromised. These secrets can be compromised in the original IKE protocol using a two–step attacking process. Each step must take place in a separate phase one exchange. That is, two separate phase one exchanges are needed for the attack to succeed. In the first phase one exchange an attacker compromises the preshared key through a brute–force or off–line dictionary attack. In the second phase one exchange the attacker compromises the Diffie–Hellman shared secret through the MITM attack (see Section 6.2 and Section 6.3 for the attack procedure). This two–step attack is successful because the same preshared key that is used in the first phase one exchange is used again in the second phase one exchange. In fact, the preshared key in the original IKE standard is used over and over in every phase one exchange.

This situation can never happen when the proposed enhancement is used at both ends of the communication since a new preshared key is used each time a phase one exchange is initiated (assuming the previous phase one session finishes successfully). Even if an attacker is able to compromise the currently used preshared key using an off–line dictionary or brute–force attack, this key is useless after the current phase one exchange finishes. When the next phase one exchange starts, which the attacker is waiting to launch a MITM attack, a new preshared key and totally new SKEYID–based secrets and a Diffie–Hellman secret are in place. Therefore, the attacker is unable to utilize the compromised key for authenticating him/herself to the initiator or the responder in the next phase one exchange to gain access to restricted data.

Furthermore, the way the new preshared key is generated guarantees that if an attacker compromises the preshared key of a phase one exchange, he/she will not be able to predict the new preshared key that will be used in the next exchange due to the use of a master

key in generating the new preshared keys. The master key is only known for the initiator and the responder of the communication.

In addition, if the current phase one exchange was forced not to finish successfully (see Section 7.5 and Section 7.7 for possible cases), a new preshared key will not be generated, and the communicating peers will use the old preshared key for the next phase one exchange. This situation *might* benefit the attacker since two consecutive phase one exchanges use the same preshared key in communicating. However, the time between a phase one exchange fails to finish and another one starts is usually very short. It is very unlikely that an attacker is able to compromise the preshared key in that short time. However, in the worst case, if an attacker is lucky and compromises the preshared key before the next phase one exchange begins, the attacker *might* be able to apply the two–step attacking process. However, in that case the attacker will have access only to the information exchanged during that session under the protection of compromised key. He/She will not be able to compromise future sessions as new preshared keys will be in place for each future session that can not be computed by the attacker.

The proposed enhancement provides a solution to disallow the attacker from even having access to only one session worth of information. It limits the number of incomplete (unsuccessful) phase one exchanges a peer might experience. After a predefined number of incomplete phase one exchanges takes place, for example 5 times, the proposed enhancement suspects that this failure is due to an attacker trying to compromise the preshared key. Therefore, it stops phase one exchange initiation and informs the system administrator to take action. Most probably the system administrators will reset the preshared keys. This solution is discussed in the Section 7.5 and the Section 7.7.

In order for the proposed enhancement to be backward compatible with the original IKE protocol, a transition period in which both versions of the IKE protocol can commu-

nicate successfully with each other is required. However, during that period an attacker might force the communicating peers to scale down to use the original IKE version. This might help the attacker in compromising the connection since enforcing the peers to use the original IKE protocol will result in using the same preshared key over and over in the authentication process. So, the attacker will need to compromise that key once and keep enforcing the two peers to use the IKE original protocol (version 1.0) to compromise all future communication.

Suppose that the initiator offers to use the enhanced version of the IKE protocol, and suppose that the responder replied that it can also use the enhanced version. An attack can be launched as follows: the attacker intercepts the SA reply being sent from the responder to the initiator. The SA indicates that the responder can use the enhanced version (version 1.1) of the IKE protocol. However, the attacker modifies the reply to indicate the original IKE protocol (version 1.0) and sends the SA reply to the initiator. Consequently, the initiator and the responder scale down to communicate through the original IKE protocol even though they can use the enhanced, more secure IKE version. Neither the initiator nor the responder will be aware of such attack. This is caused by the definition of the authenticating hashes, HASH-I and HASH-R, which do not authenticate (include) the responder's SA reply in their calculations.

A possible solution for this problem can be achieved by including the SA reply of the responder in the calculation of the authenticating hashes, HASH-I and HASH-R. Therefore, after receiving the initiator's authenticating hash, the responder can check whether the initiator received the correct SA reply. If not, it can abort the protocol. Recall that the initiator's SA offer is already included in calculating the authentication hashes. Therefore, an attacker can not modify the initiator's offer to indicate version 1.0 instead of version 1.1 without being caught by the communicating peers. The rationale of this solution is

explained more in Zhou's paper [52] "Further analysis of the Internet key exchange protocol." If this solution is not adopted, the proposed enhancement will not provide a lower level of security than the existing IKE version. In the worst case, if an attacker forced the communicating peers to scale down the IKE version, the security level provided in this case will be equivalent to what currently exists. Hence, from a security point of view, this limitation does not create a security level worse than the original IKE protocol security level. Furthermore, this problem exists only as long as both versions of the protocol is supported. When all peers are updated to use the enhanced version of the IKE protocol, the attacker can not apply this attack.

Finally, if an attacker tries to compromise the connection without applying the two–step attacking process described in Section 6.2 and Section 6.3, he/she has to guess the currently used preshared key and Diffie–Hellman secret g^{ir} in the current session. However, he/she has to do so within the lifetime frame of that session (maximum of eight hours). Since g^{ir} is a large number (usually 1024 bits in length) and the preshared key is a number of 160 bits (depending on the hash function output length) the probability of guessing the preshared key and g^{ir} is less than $\frac{1}{2^{1024}} \times \frac{1}{2^{160}}$. That is, the attacker has to try $\frac{2^{(1024+160)}}{2}$ possible values on average within the eight hours limit. This is computationally infeasible even with very high processing power machines. Therefore, it is computationally infeasible for an attacker to compromise the connection within the allotted time frame. Furthermore, the probability of guessing the preshared key and the Diffie–Hellman secret can be made smaller in the proposed enhancement by using a hash function or Diffie–Hellman key with longer outputs. For example, if the SHA–512 hash algorithm and a 2048 Diffie–Hellman key are used, the probability of guessing the preshared key and g^{ir} becomes less than $\frac{1}{2^{2048}} \times \frac{1}{2^{512}}$.

In addition the security level of the proposed enhancement can be made higher, even

without extending the key lengths, by limiting the lifetime of a phase one session. Instead of using the default lifetime value of eight hours, a suspicious user can reinitiate the phase one exchange at any time forcing the use of a new preshared key, a new Diffie–Hellman secret, and totally new SKEYID–based secrets. In this case the attacker is faced with shorter time during which he/she must compromise the keys. Thus, the proposed enhancement works and provides a higher level of security than the current IKE standard for the preshared key method of authentication.

Chapter 8

Design and Implementation of the Proposed Enhancement

8.1 Design Overview

This chapter describes the design and implementation of a replacement module for the IKE phase one communication protocol. The module demonstrates the viability of the proposed enhancement. It exchanges actual IKE phase one messages and goes through the normal process of IKE preshared key authentication method. By the end of the exchange, a new preshared key is generated that is used in the next phase one session.

The module is not an application by itself. It replaces part of the IKE phase one communication to obtain a higher degree of security. Assuming that we have an existing application that is utilizing the IKE and IPsec protocols to protect its data, we are only refining the part of the IKE protocol that deals with the preshared key authentication method by replacing it with the module described in this chapter. This generates a new IKE phase one protocol with a higher level of security, but without requiring any code or behavior changes from the applications that depends on IPsec protocols to protect their communications. Figure 8.1 shows an overall picture of the proposed enhancement module compared to the other components. The replacement module sends and receives

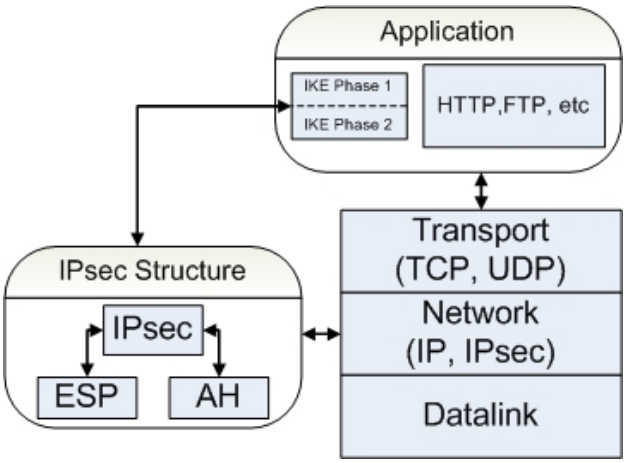

Figure 8.1: Position of the Replacement Module

phase one IKE messages through a traditional Client/Server model. The client represents the initiator's side of the protocol, and the server represents the responder's side. Basically, each side composes a message, sends it, and waits for a reply from the other side. The next sections explain the design and implementation of the module in more detail.

8.2 Design Requirements and Criteria

The design of the replacement module that implements the proposed enhancement supports the following characteristics. First, it must work with both modes of operation of the IKE phase one using the preshared key authentication method. The preshared key itself is saved in a local file on each side and must be protected from being accessed by unauthorized entities. Otherwise, the preshared key might be easily compromised. Supporting both modes of the phase one exchange aims to provide practical evidence that the proposed modification is applicable in each mode and is not tied to a specific one. Therefore, the security level of both modes can be elevated after replacing the original IKE phase one preshared key authentication method with the proposed enhancement module.

Second, the module must work transparently. Except when setting up the preshared key for the first time, the new preshared key is generated automatically each time a session of the phase one exchange ends. Neither the generation nor the setup of the new preshared key require the intervention of end users or system administrators. It is handled by the proposed enhancement software implementation , which will be part of the IKE protocol after being adapted.

Third, the design must have minimum impact on the current architecture of the IKE protocol so that it can be easily adapted and integrated, without significant code change, to current implementations. Therefore, the current architecture of the IKE protocol [6] is considered in designing the application. Furthermore, the design avoids introducing new messages or payload types. The same number of messages in each phase one mode is used with the same data and format as described in RFC 2408 [31].

Fourth, the generation of the new preshared key needs to be effective and securely accomplished without exchanging any extra data between the peers. It is necessary to have the minimum impact on the performance of the phase one exchange when producing a new preshared key. This is important to make the proposed enhancement scalable. Therefore, the new preshared key is the output of a hash function operation. Hash function operations are known to be very fast. Besides that, different hash functions produce different output lengths. As a result, the generated preshared keys can be of different lengths, which satisfies the requirements of different security policies.

Fifth, the proposed enhancement implementation must not create a deadlock situation. By deadlock we mean that a session between the initiator and the responder is never established because the preshared keys are different on both sides of the communication. This might be a result of unsynchronized key update between the communicating endpoints. Therefore, the programming module that implements the proposed enhancement

is designed to run over the Transmission Control Protocol (TCP) instead of the User Datagram Protocol (UDP). Most of the current IKE implementations run over UDP. However, according to the IKE specification, TCP can be used to transmit IKE messages. In RFC 2408 [31], it is stated in Section 2.5.1 that ISAKMP, the protocol on which IKE messages are based, can be implemented over any transport protocol.

The requirement to have a deadlock free environment suggests the use of TCP for phase one exchange. If UDP is used instead, it is always possible that the final message of the exchange is lost. For example, if the third message is lost in the Aggressive Mode (see Figure 5.5), the initiator will update its preshared key file with the generated preshared key. However, if this message is lost, the responder will never receive this message and assume the communication is not successful and hence retain the old preshared key. In this case the two peers will have different preshared keys and subsequent authentications will never succeed unless the preshared keys are reset.

By running the phase one exchange over TCP, reliability is added to the phase one exchange to prohibit such a scenario (see Section 7.5 for complete discussion of the rationale of using TCP). However, if UDP is still needed to be used, the implementer must take care of the transmission reliability. Every message must be acknowledged and retransmitted in case of errors. This requires the sender to keep a copy of each message for retransmission purposes until it is acknowledged. In addition, messages must be received in the right order. This requires the sender to append a sequence number in some way to each message in order to know which is being successfully sent and which is not. Moreover, the receiver must acknowledge every message received. Therefore, more messages are exchanged to avoid deadlock in UDP than in TCP.

Finally, the design must not allow the peers to go out of synchronization in terms of the preshared key. Therefore, updating the preshared key file with the new preshared

key in one peer is not done until after the other peer is authenticated. Otherwise, an attacker could bring the two peers out of synchronization by forcing either peer to update its preshared key file before authentication is confirmed. Moreover, an attacker could apply a denial of service attack if the new preshared key is generated before the other party is authenticated. An attacker can send millions of bogus requests that would consume the target's processing power and prevent it from doing other useful work. These requests force the target machine to generate new preshared keys, update the preshared key file, and recover from that situation by resetting the preshared key file. Therefore, this requires the update of the preshared key file to be started by the *initiator* after the second message in the Aggressive Mode and by the *responder* after the fifth message in the Main Mode.

The process of preshared key update occurs as follows: in the Aggressive Mode the initiator updates its preshared key file with the new preshared key before sending the third message. At the same time, the initiator retains the old preshared key to be used if the initiator and responder are out of synchronization for some reason. When the responder receives the third message, it updates its preshared key file with the new preshared key and retains the old preshared key . For the Main Mode, the update process starts at the responder after receiving the fifth message. It updates its preshared key file with the new preshared key, retains the old preshared key, and sends the sixth message to the initiator. at that point the initiator updates its preshared key file with the new preshared key file and retains the old preshared key file.

The reason for retaining the old preshared key file is to recover from unsynchronized preshared keys between the initiator and the responder. If for some reason the two peers are out of synchronization (one peer might crash in the middle of updating its preshared key file), the preshared keys at the endpoints will be different and they will not be able to connect successfully. Instead of notifying the system administrators to set up a new

preshared key manually, which might take a while and requires the intervention of humans, the two peers restore the preshared key files using the old preshared key. This key is the same in both machines because it was used in a successful authentication in the previous communication. The restore process along with a complete discussion about the synchronization issue is found in the Section 7.5.

8.3 Supported Cryptographic Parameters

The replacement module which implements the functionality of the proposed enhancement supports two encryption algorithms. One is the Data Encryption Standard (DES). The other is the Triple DES (3DES). Two hash algorithms, SHA–1 and MD5, are supported also. Moreover, the application supports two Diffie–Hellman groups. Diffie-Hellman groups are used to determine the length of the base prime numbers used during the key exchange process. Group 1 provides 768 bits of keying strength, Group 2 provides 1024 bits. The replacement module is also designed to accommodate other encryption and hashing algorithms with minimum effort. For example, encryption algorithms such as AES can be easily supported in the replacement module by including a system call to the AES library code. Similarly, different hash functions with different lengths, such as SHA–256 and SHA–512, are easily integrated into the replacement module in the same way. In addition, more Diffie–Hellman groups can be supported to provide more key strength if desired. Therefore, the application might be used as a testing bed for integrating more algorithms and options, and to test the functionality and the correctness of the proposed enhancement using more security algorithms.

8.4 The Initiator Design

The Initiator design follows the requirements described in Section 8.2. Moreover, the design is subdivided into two main parts based on the exchange mode. This helps in achieving a design that can be easily tested, debugged, traced for errors, modified, and maintained. Part one deals with the design of the Main Mode exchange, and part two deals with the design of the Aggressive Mode exchange. Furthermore, each part is divided into different stages according to the number of messages exchanged. For example, the Main Mode module is implemented in six stages, each corresponding to one of the messages exchanged in this mode, while the Aggressive Mode module part is implemented in three stages. Regardless of the mode used, each stage is comprised of two main tasks (functions):

1. Create the appropriate message with the right format and send it to the responder.

2. Receive the reply of the corresponding message, verify, decode, and process its content.

A Finite State Machine (FSM) is used to describe the states of the initiator's design [53]. The FSM is developed by following the sending and receiving of the messages exchanged. However, since phase one of the IKE protocol can be archived through the Aggressive Mode or the Main Mode, a separate FSM is required for each mode. Figure 8.2 depicts the FSMs of the Main Mode and the Aggressive Mode in the IKE phase one communication.

In the Main Mode FSM, the initiator starts in the *Init* state. When it creates and sends the first message, it moves to the next state in which it waits for a reply for that message. If a reply is received and passes the verification process, the initiator moves to the *key exchange (KE)* state. In this state, the KE and Nonce payloads are built and sent.

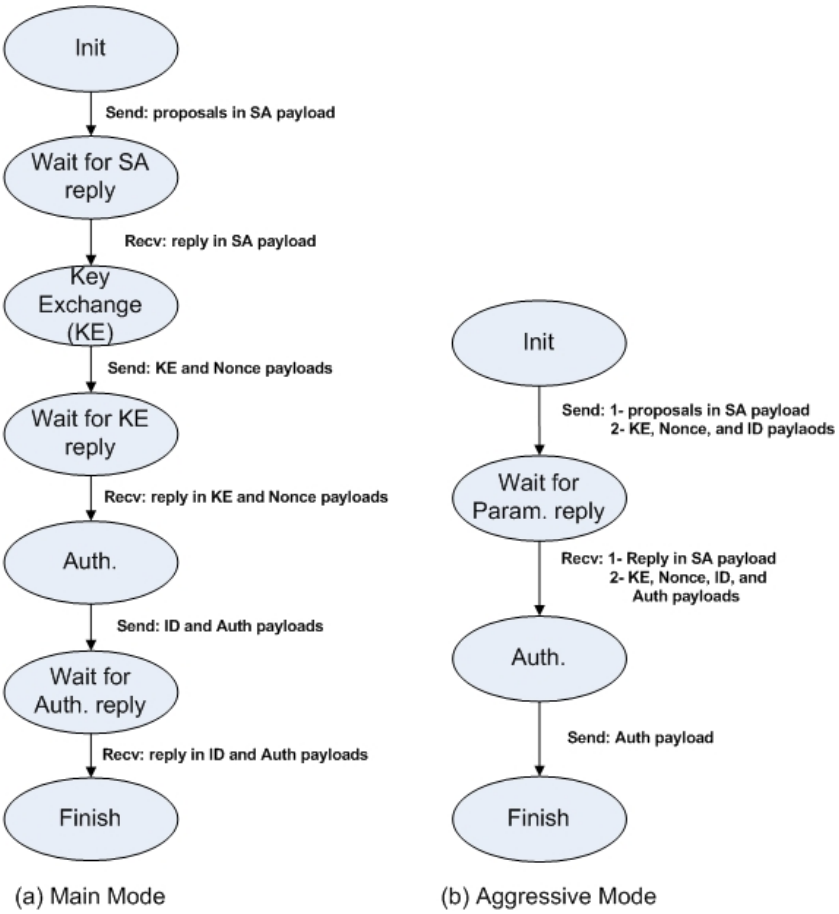

Figure 8.2: Initiator's Finite State Machine in IKE Phase One

When the message which contains them is sent, the initiator moves to the *wait for KE reply* state. In this state the initiators waits for a message containing the KE and Nonce payloads of the responder. If the message is received and passes the verification process, the initiator moves to the *Auth.* state. In this state the authenticating hashes are computed and added to the ID payload to form a message that is sent to the responder. After sending the message, the initiator move to the *wait for Auth. reply* state expecting a reply for the authentication message. If the reply is received and verified, the initiator updates its preshared key file and finishes the phase one exchange. Note that in the FSMs, details

of unexpected messages or errors are left out until the discussion of the implementation details. In addition, the details of generating the new preshared has been described in Section 7.4.

The same description of the states applies to the Aggressive Mode FSM. The initiator starts in the *Init* state. After sending the first message it moves to the *wait for reply* state, in which a reply for the sent message and the authenticating hash of the responder is expected. When received and verified, the initiator updates its preshared key file, prepares its authenticating hash, and moves to the *start authentication* state. After sending its authenticating hash, the initiator exits and finishes phase one.

8.5 The Responder Design

The responder is represented by the server side of the application. It is designed following the design requirement described in Section 8.2. Moreover, a FSM similar to the initiator's FSM describes the responder's design architecture [53]. Similar to the initiator's design, the responder is divided into two main parts. One represents the Main Mode, and one represents the Aggressive Mode. Each part is implemented in a number of stages according to the number of messages required to finish each mode. In each stage, the responder waits for a request from the initiator. When this happens, it processes the request and sends the appropriate reply to the initiator. Figure 8.3 depicts the FSM of the responder in the IKE phase one communication.

In the Main Mode FSM the responder starts in a waiting state called *wait for a request*. It waits for the initiators to establish a connection. If the request is received, the responder moves to a processing state, called *process proposal*, in which the request is verified, decoded, and processed accordingly. After a reply is sent for the request the responder moves to the *wait for KE exchange* state. In this waiting state, the key exchange and nonce

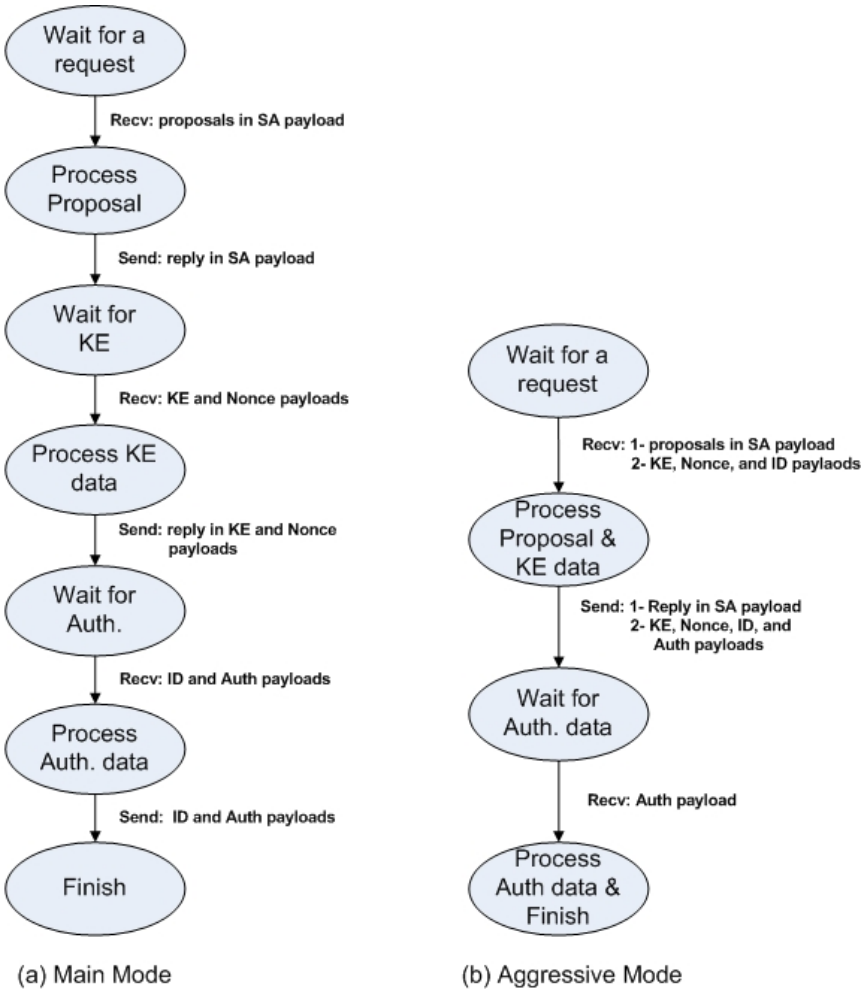

Figure 8.3: Responder's Finite State Machine in IKE phase One

payloads of the initiator are expected. The responder keeps waiting for these payloads until they are received, at which time it moves to the next state, *process KE data*, to process them. In this state, the responder creates its nonce and key exchange payloads and sends them back to the initiator. By doing so, the responder moves to the next state, *wait for auth. data*, waiting for the initiator's authentication and identification payloads. When received, the responder processes them in the *process auth. data* state. In this state the responder updates its preshared key file, sends its authentication and identification

payloads to the initiator, and finishes the phase one exchange.

Because the Aggressive Mode finishes phase one exchange with a fewer messages, it is represented by fewer states in Figure 8.3(b). In this mode, the responder waits for the initiator's request in the *wait for a request* state. The initiator's request contains the proposal, key exchange, nonce, and identification payloads. The reception of this message moves the responder to the *process proposal and KE data* state. In this state, the correctness of the received request is verified and an appropriate reply is sent to the initiator. Sending the reply moves the responder to the *wait for Auth.* state, in which the responder is waiting for the initiator's authenticating hash. After the hash is received and processed, the responder updates it preshared key file, finishes the exchange, and exits.

8.6 Implementation Overview

The FSM's described above are followed to implement the IKE phase one proposed enhancement module. Details of how to compose and process each message of the IKE phase one exchange are provided in the next sections. The proposed enhancement's module is implemented using the C programming language. It runs over the TCP protocol and uses port 5500 to exchange the required messages. A user has control over the initiator and responder parts of the replacement mode using commandline arguments. The arguments and defaults are discussed in Appendix A.

8.7 Related Work

There are some open source projects that implement the IKE protocol. Three widely known implementations are the OpenBSD project (*isakmpd*), the Linux Free S/WAN project (*pluto*), and the KAME project (*Racoon*) [54]. These projects implement the full specifications of the IKE protocol described in RFC 2408 [31] and RFC 2409 [6]. They

implement both phases of communication and all supported authentication methods.

Among the three projects, OpenBSD is the most up–to–date and generic implementation. Therefore, it is chosen for further analysis and study of the implementation code. After more than a month of code analysis, it was difficult to extract some code of the project that only implements the preshared key authentication method and reuse it in the replacement module that implements the IKE preshared key authentication method proposed enhancement. The OpenBSD code is huge and most of the time is undocumented or barely documented. This makes it difficult to figure out what code does which authentication method of the IKE protocol. Therefore, attention was shifted into looking for a library code that can be utilized in building the application. Such a library exists and is called the *libike* library. According to its implementation [55], libike is a C library that allows its users to engage in IKE exchanges. It implements IKE exchanges and utilizes a callback mechanism to delegate tasks of the actual packet transmission and security policy management to the external code.

The library provides a clear interface and design. However, the implementation is not complete. The responder's functionality is not implemented. Furthermore, some functions are simply a return statement with a note for users to implement them while some are partially implemented. Therefore, the library is of limited usability. Despite that, the library gives an excellent overview of what an IKE implementation might look like. In fact, it provides an excellent reference example to the OpenSSL library [46].

OpenSSL is a full–strength general purpose cryptography library. It provides cryptographic functionality to applications by implementing a wide range of standard cryptographic algorithms such as DES, 3DES, MD5, and SHA–1. More about OpenSSL can be found in its online documentation [47]. Based on the foregoing, the IKE phase one replacement module is built from scratch using the OpenSSL library.

8.8 Replacement Module Programming Structure

The replacement module is composed of several resource files that collectively build the IKE preshared key authentication method code. In fact, the files are of two types: C language source code files and header files. Three main header files are used in developing the application code. The first file is *common.h*. It contains header files, constants, and function prototypes required by and used in other files. The second file is *const.h*. This file contains the Diffie–Hellman prime numbers that correspond to the group numbers used in the IKE. These numbers are taken from RFC 2409 [6] and RFC 3526 [56]. The third file is *isakmp.h*. It defines all the constants corresponding to the supported cipher and hash algorithms. Moreover, it defines the supported payload and identification types besides the header's structure and format for all payload types.

The source code files are composed of four main files. The first one is the *initiator.c* file. It defines all the necessary variables and uses various functions to implement the functionality of the initiator part of the application. The second file, which implements the responder's functionality, is the *responder.c* file. The third file, *crypt.c*, is dedicated for cryptography functions. It contains a collection of functions that implement different services of cryptography: it implements the functions of encryption, decryption, hashing, Diffie–Hellman key generation, and SKEYID secrets computations. Finally, the message creation and processing functions are implemented in the file *isakmp.c*. This file collects and implements all the functions that provide the Application Programming Interface (API) to create and process different ISAKMP phase one messages. It includes functions that support both the Aggressive Mode and the Main Mode. Functions in this file are named according to the mode used. For Aggressive mode the functions are named *aggressive_createx* and *aggressive_processx*, where x is the number of the message created or processed. Similarly, the Main Mode functions are called *main_createx*

and *main_processx*. These functions are the core of the application. They are described later.

8.9 Implementation Details of the Initiator

The main goal of the initiator is to request the establishment of a SA. Following the initiator's FSM that is described in Section 8.4, the initiator composes and processes messages that lead to the establishment of the SA. Figure 8.4 shows the overall algorithm of the initiator. Each part is described next.

8.9.1 First Message

The purpose of the first message is to negotiate SA parameters between the initiator and the responder. Its contents depend on the mode used in the negotiation. In the Main Mode, the message contains an ISAKMP header followed by a Security Association (SA). In the Aggressive Mode, a key exchange (KE), nonce, and identification (ID) payloads are added to that. The first message is created using the functions *main_create1* and *aggressive_create1* for the Main Mode and the Aggressive Mode, respectively. Figure 8.5 shows the flowchart for creating the first message in both modes of operation.

Creation of the first message is the most difficult task. The initiator has to encode eight different transforms in the offer sent to the responder. Recall that two cipher algorithms, two hash algorithms, and two Diffie–Hellman groups are supported in this application. Two approaches can be used to encode the transform payloads in the first message. The first one is to build the payloads from top to bottom. This requires the transform payloads to be chained in a linked list. The list must be traversed each time a transform payload is added, and traversed back to update the length field of each upper transform payload to reflect the new length of the lower payloads. This approach is difficult to implement

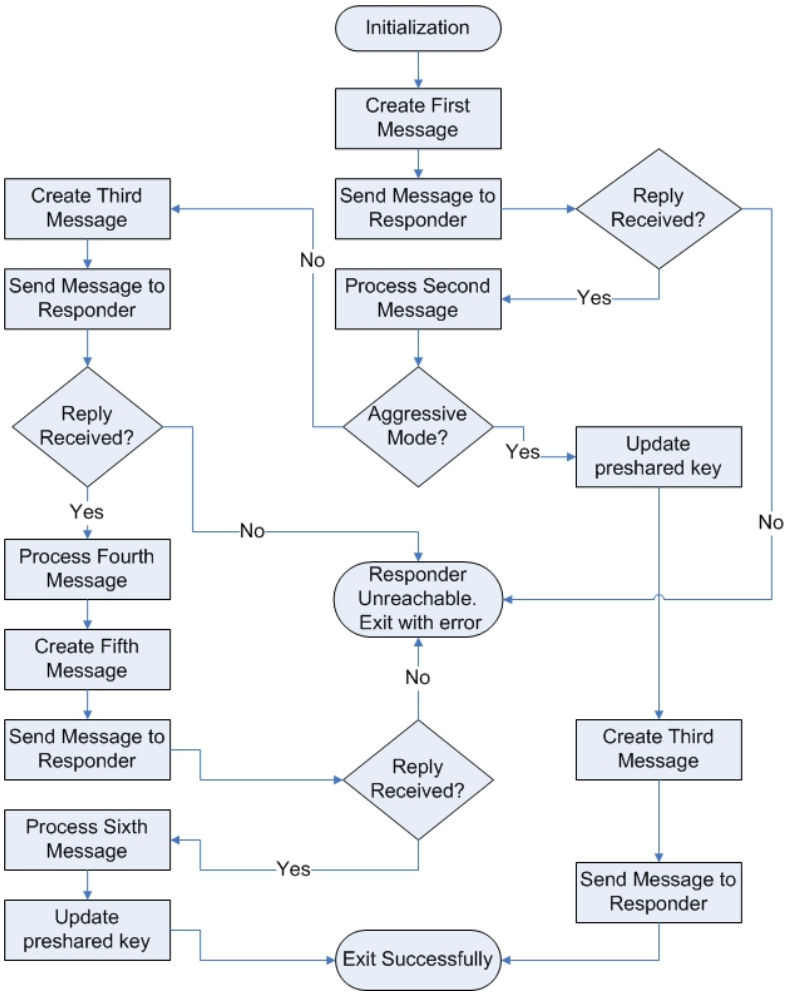

Figure 8.4: Initiator's Flowchart

and is not effective if many payloads are to be added. The other approach is to build the message backward from bottom to top. The initiator builds the last payload first. In this way the type and length of the current payload are in hand before building (adding) the next payload. Thus, it becomes easier to set the ”next payload” and ”length” fields of the next payload. This process continues up until the ISAKMP header is added.

The second approach is used to construct the SA payload in the first message. Figure 8.5 depicts the steps taken to do so. Part (a) of the figure shows the creation of the

first message in the Main Mode, and part (b) shows the creation of the first message in the Aggressive Mode.

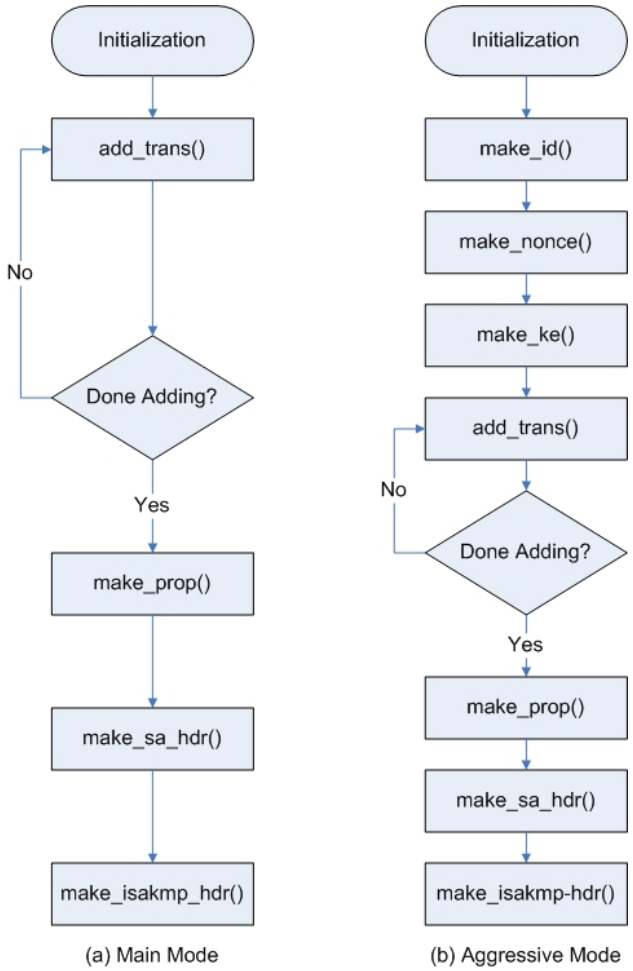

Figure 8.5: Creation of the Initiator's First Message

To have an easy–to–follow design, each step is achieved through a call to some function that is implemented in the application. In Figure 8.5(a) the function *add_trans* builds and adds a transform payload to a set of transforms. The function is called each time a transform payload is to be added. A flag is set in the call to the function when adding all payloads is finished in order for the function to finalize the process and return a pointer to the constructed transform payload(s). The function *make_prop* constructs a proposal

payload. It takes the number of transform payloads constructed earlier along with their lengths, assigns the corresponding fields in the proposal header, and returns a pointer to the constructed header along with its total length. The *make_sa_hdr* function uses the returned values from the make_prop function to construct a SA header. The next payload field of the SA header points to a proposal header and the length field is the total length of the proposal header and the SA header. The final step in building the first message is adding the ISAKMP header. This is accomplished by the function *make_isakmp_hdr*. It sets the initiator's cookie, exchange type, next payload type (SA in this case), and the total length of the message. Cookies are 8–byte random numbers generated using the *RAND_bytes()* library function defined in the OpenSSL library. This function generates *strong* pseudorandom numbers, which are difficult to be guessed or determined by an attacker except by trying all possibilities (brute–force).

The same functions are used to build the first message in the Aggressive mode except that the process starts with building different payloads. Following the bottom–up approach, the identification payload is built first through the function *make_id*. It sets the ID data, type, and length in the ID header. The nonce payload is built next by the function *make_nonce*. Similar to the cookies, the nonce is a 20–byte random number generated in the same way. The *build_ke* function constructs a key exchange payload. The body of the payload is filled with the Diffie–Hellman public value corresponding to the selected group number. The remainder part of the message (the SA payload and ISAKMP header) is constructed the same way as in the Main Mode. After finishing message one creation, control goes back to the initiator to continue phase one exchange communication.

8.9.2 Second Message

The initiator is concerned with processing, not creating, the second message. Creation of the second message is done at the responder and is explained later (see Section 8.10.2). The second message contains the responder's reply to the initiator's offer. In the Main Mode, it contains the chosen SA parameters and the ISAKMP header. In addition to that, it contains the responder's KE, nonce, and ID data in the Aggressive Mode. Processing the second message is done using the function *main_process2* and *aggressive_process2* in the Main Mode and the Aggressive Mode, respectively. Figure 8.6 shows the required steps to process the second message.

The processing in the Main Mode starts by verifying the received message for possible errors. In fact, the verification process is the first step in processing all received messages. The length of the received message is checked to see if it is valid. If the length is less than the ISAKMP header size, the payload length of the message ($message\ length - ISAKMP\ header\ size$) is less than ISAKMP header size, or there is no next payload, the received message is too short to encode. Thus, the process exits with a suitable error message. If the length is valid, the initiator's cookie in the received message is verified against the one that was sent in the first message. If they do not match, the process exits with a suitable error message. If they match, the verification process continues to validate the exchange type in the received message.

The "exchange type" field of the ISAKMP header in the received message must match the mode that the initiator is expecting. If the initiator starts in the Aggressive Mode or the Main Mode, the reply must be in the Aggressive Mode or the Main Mode, respectively. Moreover, the next payload after the ISAKMP header must be a SA payload. Otherwise, an error message is displayed before exiting the process. When verification is finished successfully, the initiator goes to the next step, *Process CKY–R*, in which the responder

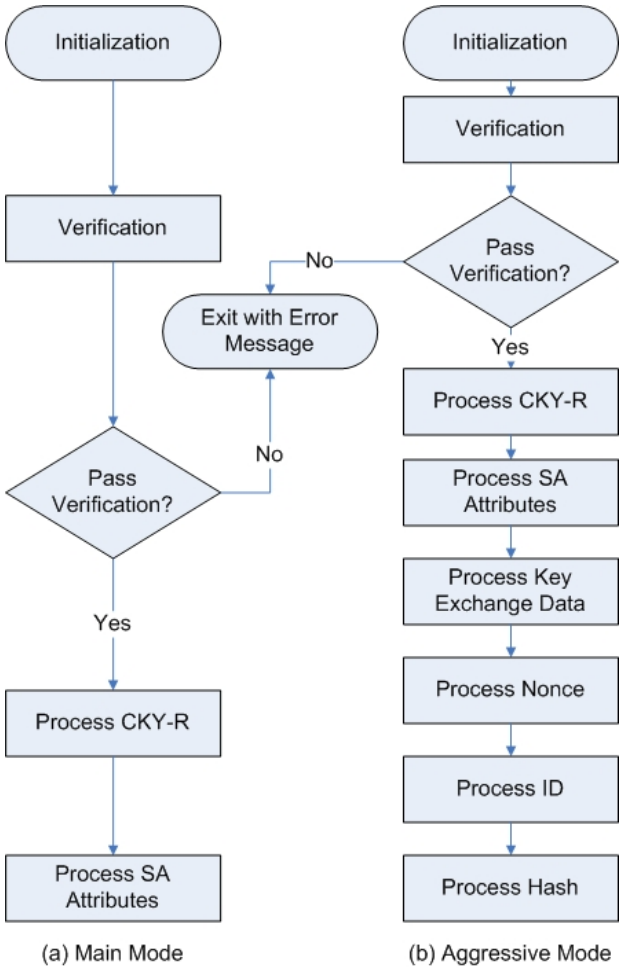

Figure 8.6: Processing of the Initiator's Second Message

cookie (CKY–R) is saved. The next step is to process the SA attributes. The initiator verifies that there is one and only one transform payload in the reply. If not, a warning message is displayed, and the initiator processes the first transform payload only, from which the SA attributes that the responder has chosen is extracted. These attributes are checked to see if they are supported by the initiator to make sure the responder has not replied with an attribute that was not offered. If they are supported, processing of the second message in the main Mode ends and control returns to the initiator to continue phase one exchange communication.

For the Aggressive Mode, the verification process is longer. However, it follows the same procedure as described above in the Main Mode processing. It starts with verifying the received message's length, cookies, and SA attributes. After that, the initiator processes the key exchange data. The public key of the responder is saved for later use when the Diffie–Hellman shared secret is generated. Similarly, the responder's nonce is saved to be used later. The responder's ID data and type is processed next and saved for later use. The important step comes next. The initiator has to verify the responder's authenticating hash, HASH–R. It computes the shared Diffie–Hellman key, SKEYID, and SKEYID_a values. Then it computes the expected value of HASH–R. If this value matches what is received in the message, the responder is authenticated. If not, an error message is displayed before exiting.

At this stage the initiator is ready to generate and update its preshared key in the Aggressive Mode. If the responder's authenticating hash passes the verification process, the initiator generates a new preshared key, updates the preshared key file, and retains the old preshared key to recover from an unsynchronized situation in case the preshared key update process does not finish successfully. Then, the initiator sends the third message to the responder.

8.9.3 Third Message

This is the last common message between the Aggressive Mode and the Main Mode in the phase one exchange. After that, all the messages are for the Main Mode only since there is no message after the third one in the Aggressive Mode.

Generating the third message is easy in both modes of operation of the phase one exchange. Figure 8.7 shows the steps necessary to build the third message by the initiator. The functions *main_create3* and *aggressive_create3* are responsible for creating the third

message in the Main Mode and the Aggressive Mode, respectively.

In the Main Mode, the initiator simply builds its nonce payload using the function *make_nonce*. The nonce is a 20–byte strong random number. Next, the *make_ke* function

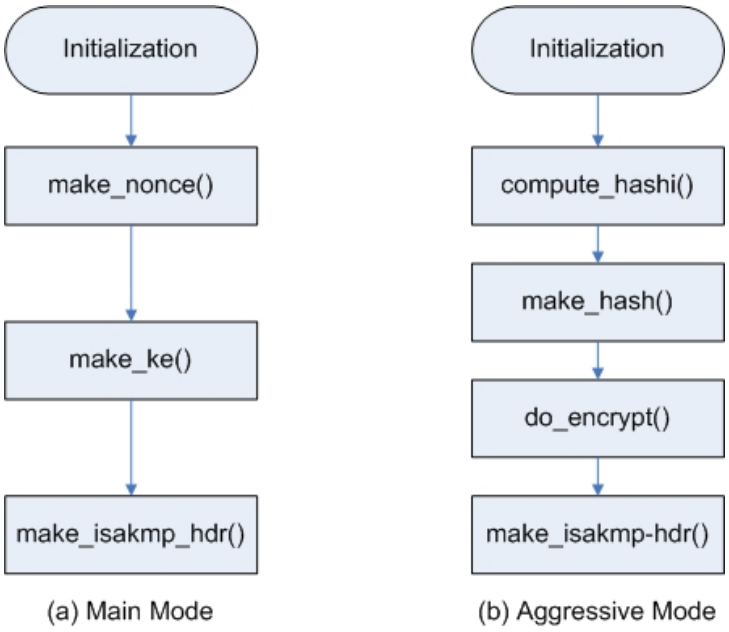

Figure 8.7: Creation of the Initiator's Third Message

builds the key exchange payload. It generates the Diffie–Hellman value that corresponds to the negotiated group number. Finally, the ISAKMP header is added to form the Main Mode third message. In the Aggressive Mode the process is different. The third message is encrypted and contains the initiator's authenticating hash. The authenticating hash value is computed first using the function *compute_hashi* then the hash payload is constructed using the function *make_hash*. Before the ISAKMP header is added, the hash payload is encrypted in the Cipher Block Chaining (CBC) mode (see Section 2.3.1.2) using the SKEYID_e key and the negotiated cipher algorithm. This is done using the *do_encrypt* function.

Among other parameters, the do_encrypt function requires an encryption key and an

initialization vector (IV). The computation of encryption key is dependent on the cipher algorithm negotiated. In RFC 2409 [6] it is stated that the encryption key for the DES algorithm is the first 8 bytes of the base encryption key (SKEYID_e). However, For the 3DES algorithm it is the first 24 bytes. Since the SKEYID_e is either 16 or 20 bytes (depending on the negotiated hash algorithm), it needs to be extended. Assuming K is the required encryption key, | denotes concatenation, and 0 is a one–octet representation of zero:

$K = K1 \mid K2 \mid K3$, where

$K1 = hash(SKEYID\_e, 0)$,

$K2 = hash(SKEYID\_e, K1)$, and

$K3 = hash(SKEYID\_e, K2)$.

For 3DES it is enough to compute K1 and K2 and get the first 24 bytes. The IV is the result of hashing the initiator's and responder's Diffie–Hellman public keys. Since only eight bytes are needed in either DES or 3DES, the IV is the first eight bytes of the resulting hash. After the hash payload is encrypted, the ISAKMP header is added with the encryption flag is set in the header to indicate that the message is encrypted.

8.9.4 Fourth Message

The fourth message is applicable only in the Main Mode. Processing the message starts at the initiator's side by the verification process. The message's length, exchange type, and cookies are all verified. If verified successfully, the responder's public key and nonce are extracted and saved by processing the key exchange and nonce payloads, respectively. Actually, processing the fourth message is similar to processing the last part of the second message of the Aggressive Mode (see Section 8.9.2).

8.9.5 Fifth Message

The fifth message contains the initiator's authenticating hash, HASH–I, and ID. Before constructing the message, the initiator must compute the Diffie–Hellman shared key, the SKEYID–based secrets, the initialization vector, and the encryption key. Then, the initiator's authenticating hash value (HASH–I) is computed using the function *compute_hashi* in the same way it is computed in the Aggressive Mode third message. This value is sent to the *make_hash* function to construct the initiator's hash payload. In fact, it is the body of the payload. ID payload is constructed next using the function *make_id*. The payload carries the IP address of the initiator as an identifier.

Recall that the fifth message is encrypted. Therefore, the part consisting of the hash and ID payloads is encrypted next. The encryption function returns a pointer to the encrypted data along with its length. The length of the returned data might be different from the length of the data that was encrypted due to padding bytes that are added to the data before encryption. This is the length that is added to the ISAKMP header length to form the message's total length. The ISAKMP header, with the encryption bit is set, is added to the encrypted data to make the fifth message. Figure 8.8 shows the steps required to create the fifth message in The Main Mode.

8.9.6 Sixth Message

The sixth message is the last message in the exchange. It contains the responder's hash and ID payloads. Processing steps are shown in Figure 8.9.

The first step is the verification process. Besides the length, exchange type, and cookies verification, the encryption flag in the ISAKMP header is verified also since the message is expected to be encrypted. If it is not set, an error message is displayed before exiting. The next step is to decrypt the received message to recover the responder's ID

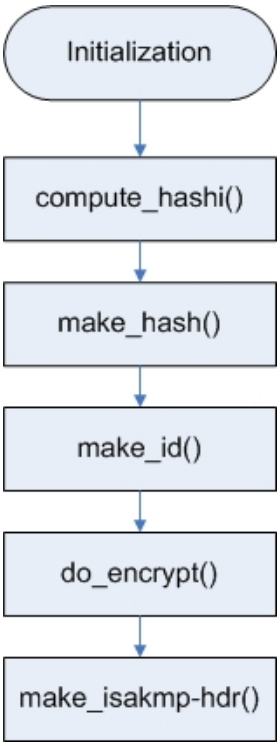

Figure 8.8: Creation of the Initiator's Fifth Message in the Main Mode

and hash payloads. The initiator uses the negotiated cipher algorithm and the encryption key computed in message five to decrypt the received message. Then, it gets the responder ID and compares it with the responder's IP address. In the Main Mode, the ID payloads contain the IP address of its sender. If there is no match, the initiator displays an error message and exits. If the ID payload is processed successfully, the initiator moves to verify the responder's authenticating hash, HASH–R. It computes an expected value for HASH–R and compares it with the received one. If they match, the responder is authenticated. In this case, the initiator updates its preshared key file, retains the old preshared key, and finishes the phase one communication. If the responder fails the acknowledge the last message, the initiator resets it preshared key file with the retained preshared key and display a warning message to the system administrators.

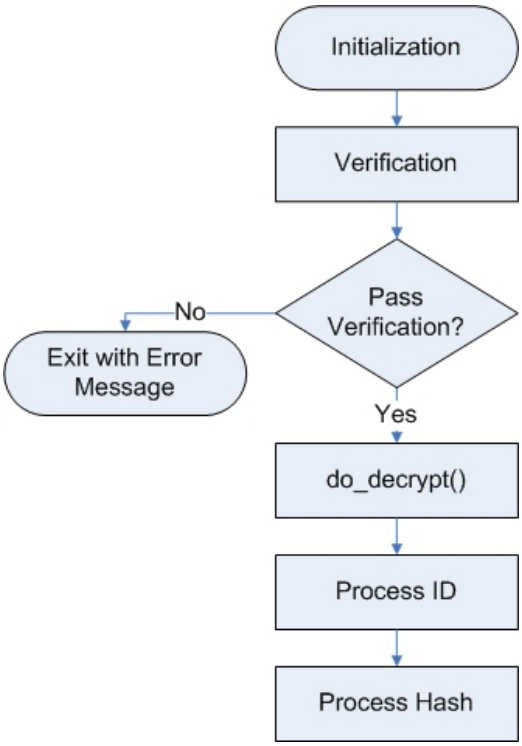

Figure 8.9: Processing of the Initiator's Sixth Message in the Main Mode

8.10 Implementation Details of the Responder

Most of the implementation details of the responder are similar to that of the initiator. Basically, the responder starts and waits for the initiator's request to establish a SA. Then it starts the creation and processing of the messages. Figure 8.10 depicts the overall flowchart of the responder. Many of the initiator functions are reused in the implementation of the responder due to the similarity in both creating and processing of the messages.

8.10.1 First Message

The first message contains the initiator's request to establish a SA using any of the offered transforms. When the first message is received in the Main Mode the responder verifies its length and the existence of a next payload. Moreover, the next payload must be a SA

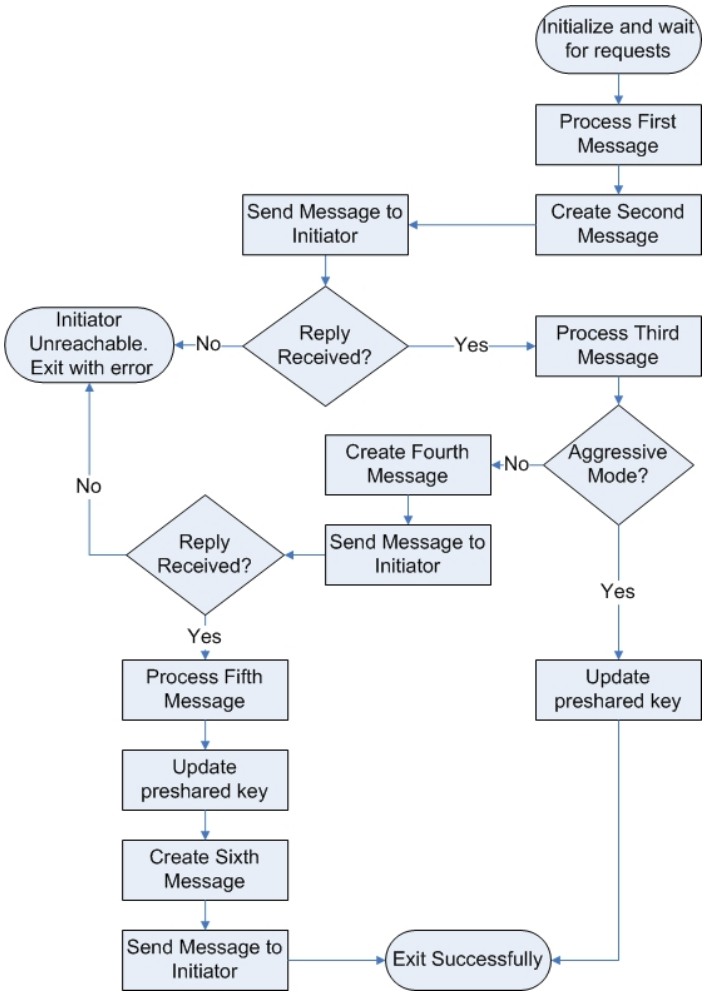

Figure 8.10: Responder's Flowchart

payload. Otherwise, an error message is displayed causing the responder to exit. Next, the responder saves the initiator's cookie and extracts the body of the SA payload. The SA body contains the transform payload, which has a field that indicates the number of transforms offered by the initiator. Each transform contains a set of proposed SA parameters that the responder has to choose from. To do so, the responder loops over these proposed transforms sequentially. For each one, the proposed cipher algorithm, hash algorithm, and Diffie–Hellman group number are extracted and checked whether they are supported

according to the responder's security policy or not. If they are supported, the looping process stops returning these parameters for the responder to use them in establishing the SA. If they are not supported, the looping process continues. If the looping process finishes without finding any supported parameters, the responder displays an error message and exits.

The first message in the Aggressive Mode contains more payloads than the first message in the Main Mode. However, the common part (ISAKMP header and SA payload) are processed similarly. After verifying the ISAKMP header and processing the SA payload as explained above in the Main Mode processing, the initiator's Diffie–Hellman public value is obtained from the key exchange payload and saved for later use. Similarly, the initiator's nonce is obtained from the nonce payload. Finally, the ID payload is processed and the initiator's ID is obtained. Figure 8.11 shows the steps for processing message one at the responder.

8.10.2 Second Message

The responder constructs the second message after successfully finishing the processing of the first message. It is important to know that the exchange moves sequentially. A message can not be created unless the previous message is processed. Composing the responder's second message in the Main Mode is exactly the same as composing the initiator's first message in the Main Mode except that the responder's Diffie–Hellman public value and nonce are used in constructing the corresponding payloads. In addition, the responder's cookie is added to the ISAKMP of the message.

Similarly, the responder's second message in the Aggressive Mode is constructed using the responder's ID, Diffie–Hellman public value, and nonce in the same way as the initiator's first message in the Aggressive Mode is constructed (see Section 8.9.1). Fur-

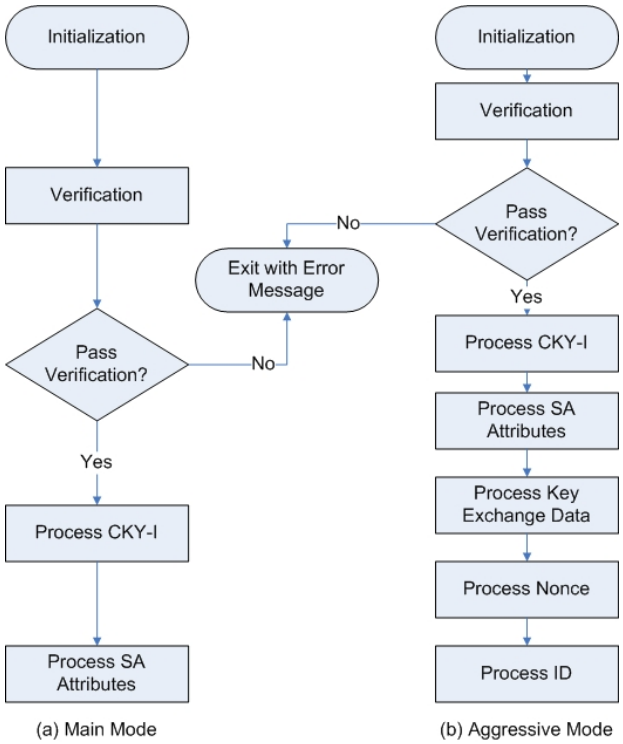

Figure 8.11: Processing the Responder's First Message

thermore, the message contains the responder authenticating hash, HASH–R, in a hash payload. The hash payload is built exactly the same way the initiator's hash payload is built except that the HASH–I value is replaced with the HASH–R value. That is, the HASH–R value is computed using the function *compute_hashr* then the hash payload is constructed using the function *make_hash*. This procedure is reflected in Figure 8.12.

8.10.3 Third Message

The third message is easy to process. In the Aggressive Mode, the message is encrypted and contains the initiator's authenticating hash. The responder starts by verifying the message length and the existence of a next payload. Moreover, the exchange type is checked to make sure it indicates the Aggressive Mode, and the encryption flag is verified

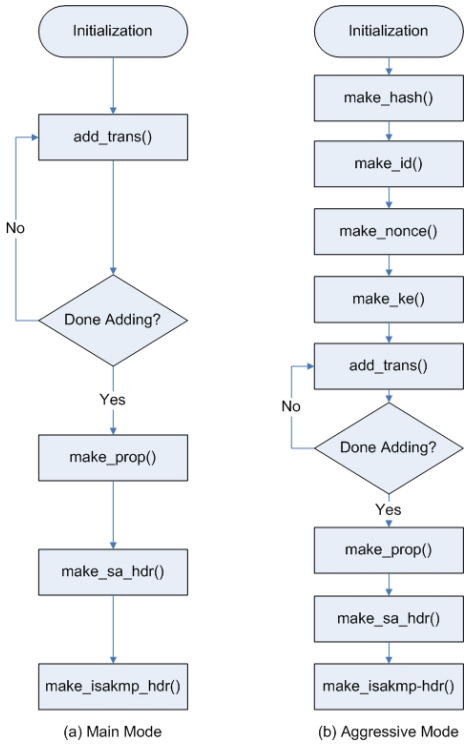

Figure 8.12: Creation of the Responder's Second Message

to be set. If the verification succeeds, the next step is to compute the IV, SKEYID–based secrets, and encryption key in order to decrypt the message. The HASH–I value is then extracted from the hash payload (after decryption) and verified. If it passes the verification process, the initiator identity is authenticated. Thus, the new preshared key is computed and the preshared key file is updated with the new key. Processing the third message in the Main Mode is straightforward. After passing the verification stage, the initiator's Diffie–Hellman public value and nonce are extracted and saved. Figure 8.13 depicts the processing steps of the third message at the responder side in both the Main Mode and the Aggressive Mode.

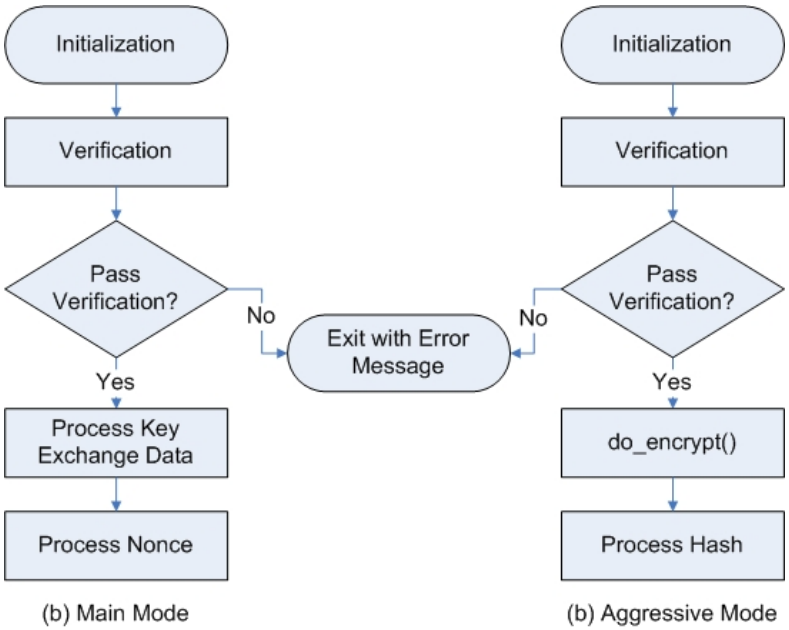

Figure 8.13: Processing of the Responder's Third Message

8.10.4 Fourth Message

The fourth message sent by the responder contains the responder's Diffie–Hellman public value and nonce. It is only applicable in the Main Mode since the Aggressive Mode is composed of three messages only. The construction of this message is exactly the same as the construction of the initiator's third message in the Main Mode. Figure 8.14 shows the creation of the fourth message process at the responder side. First, the nonce payload is created. A nonce is a 20–byte randomly generated value that is inserted in the body of the nonce payload. Next, the responder's Diffie–Hellman public value is generated based on the group number that is negotiated between the initiator and the responder in the first message. If the default group number is used, a 1204–bit Diffie–Hellman public key is generated and inserted in the body of the key exchange payload. The default group number is 2. After that, an ISAKMP header is added to the Nonce and Key Exchange payloads to form the forth message that is sent to the responder.

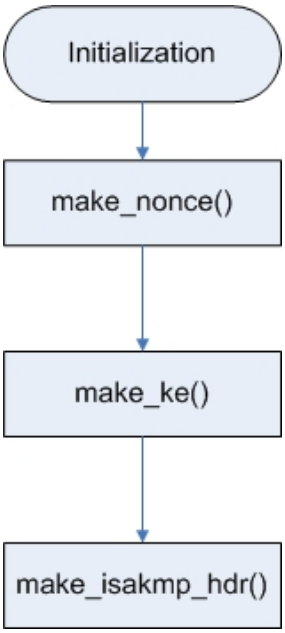

Figure 8.14: Creation of the Responder's Fourth Message

8.10.5 Fifth Message

The steps taken to process the fifth message at the responder is depicted in Figure 8.15. The message contains the initiator's authenticating hash, HASH–I, and its ID. The message is expected to be encrypted and the exchange type is expected to be set. These are verified first along with the message length and the existence of a next payload at the beginning. In addition, the cookies of the received message are checked to see if they match the original saved cookies of the initiator and the responder. If not, the responder issues an error message and exits. If they match, the process continues to decrypt the message. SKEYID–based secrets, IV, and decryption key are computed to do so. After successful decryption, the processing of the ID and hash payloads begins. The initiator's ID value is obtained from the ID payload carried in the message, and the HASH-I value is verified against the expected one. If the received ID value and authenticating hash match what is expected by the responder, the initiator is authenticated and the process of updating the

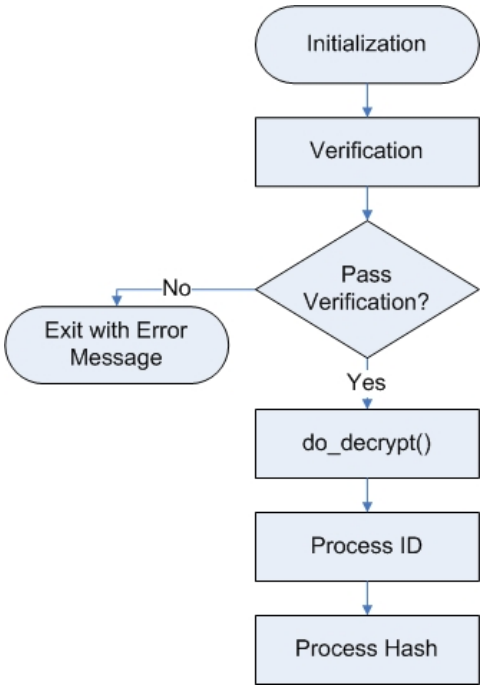

Figure 8.15: Processing of the Responder's Fifth Message

preshared key is started. If not, an error message is displayed before exiting the responder program. As a result of responder's exiting, the initiator will exit also and care is taken to make sure the preshared keys at both endpoints are not brought out of synchronization.

8.10.6 Sixth Message

The responder composes the sixth message by appending its authenticating hash, HASH–R, and ID payloads to an ISAKMP header. The message must be encrypted before being sent to the initiator. The process of creation the message is shown in Figure 8.16. The responder computes its HASH–R value and builds its hash payload. Furthermore, it builds its ID payload. The payload contains the responder's IP address. Before adding the ISAKMP header, the part of the message consisting of the hash and ID payloads is encrypted. Next, the encryption flag in the ISAKMP header is set before adding the header

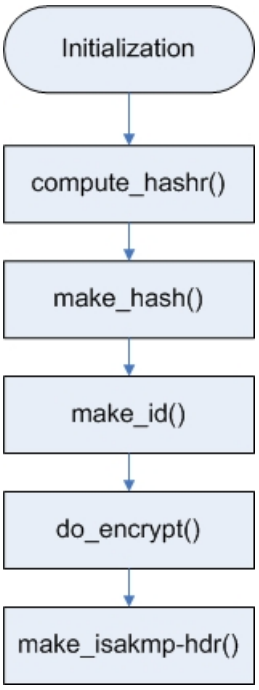

Figure 8.16: Creation of the Responder's Sixth Message

to the encrypted data to form the sixth message.

8.11 Testing the Replacement Module

This section presents the reader with some output samples of running the IKE preshared key authentication method replacement module. The replacement module implements the proposed enhancement to the IKE protocol. The module is fully described in Chapter 7. To verify the viability and the applicability of the proposed enhancement, some output results from running the replacement module are shown here. In this section, samples of the initiator's output are shown. The responder side has similar output results.

The output is generated by running two programs that the replacement module is composed of: the initiator program and the responder program. Each program represents one side of the communication. The programs are run on different machines. They can not

be run on the same machine since they are using the same port number for communication by default. Unless the default port numbers are overridden, trying to run the two programs on the same machine will not succeed.

The programs are controlled by commandline arguments and options. For example, a user can override the default ports or select the mode of phase one operation: Main Mode or Aggressive Mode. For a complete list and discussion of the commandline options see Appendix A.

Different conditions under which the initiator and the responder might work are tested here to show the robustness and the completeness of the proposed enhancement module design. First, the replacement module is tested under normal conditions in which the connection between the initiator is assumed to finish successfully. This verifies the viability and shows the main functionality of the proposed enhancement. Basically, the proposed enhancement module uses the current preshared key in authenticating the phase one exchange, generates a new preshared key after the authentication process finishes successfully, and uses the newly generated key in the next phase one exchange communication. This error–free situation is shown in Figure 8.17.

The figure shows the output of running the module for the first time in the Main Mode. It uses the current preshared key, goes through the authentication process, and generates a new preshared key. The responder is started on *garnet.rnet.missouri.edu* and the initiator is started on *mendel.rnet.missouri.edu*. After the end of the exchange, the mode used, the time it took to finish the exchange, the old preshared key, and the new preshared key are displayed. Then, the initiator's side of the replacement module is run again in the Main Mode (can be run in the Aggressive Mode with no difference). This is shown in the bottom half of Figure 8.17. It shows running the module again in the Main Mode using the result of the previous run. The preshared key that was generated in the previous run

```
mendel.rnet.missouri.edu> ./initiator garnet.rnet.missouri.edu

IKE MAIN MODE
Message 1 ===> i > HDR, SAi sent

Message 2 <=== i < HDR, SAr received

Message 3 ===> i > HDR, KEi, Ni  sent

Message 4 <=== i < HDR, KEr, Nr  received

Message 5 ===> i > HDR*, IDi, Hashi  sent

Message 6 <=== i < HDR*, IDr, Hashr  received

:) :) :) :) :) IKE Phase One Done

It took 0.1169770 seconds to finish phase one exchange
Old PSK  = 11166DACD154B352C42EBF874967107BBC61194F
New PSK= BC85D9F35F4E1BFDA9F96A043AFA52E22C242723



mendel.rnet.missouri.edu> ./initiator garnet.rnet.missouri.edu

IKE MAIN MODE
Message 1 ===> i > HDR, SAi sent

Message 2 <=== i < HDR, SAr received

Message 3 ===> i > HDR, KEi, Ni  sent

Message 4 <=== i < HDR, KEr, Nr  received

Message 5 ===> i > HDR*, IDi, Hashi  sent

Message 6 <=== i < HDR*, IDr, Hashr  received

:) :) :) :) :) IKE Phase One Done

It took 0.1170870 seconds to finish phase one exchange
Old PSK  = BC85D9F35F4E1BFDA9F96A043AFA52E22C242723
New PSK= 7990BAA307C7F81E4C431579555AF0A153AA94EC
```

Figure 8.17: Running the Replacement Module in the Main Mode Twice

is used in this run, and a new preshared key is generated for the next run. This is the way the proposed enhancement to the IKE protocol is supposed to work.

To show that the module works in both modes of phase one exchange, Figure 8.18 shows the initiator's output when running the module in the Aggressive Mode. Three

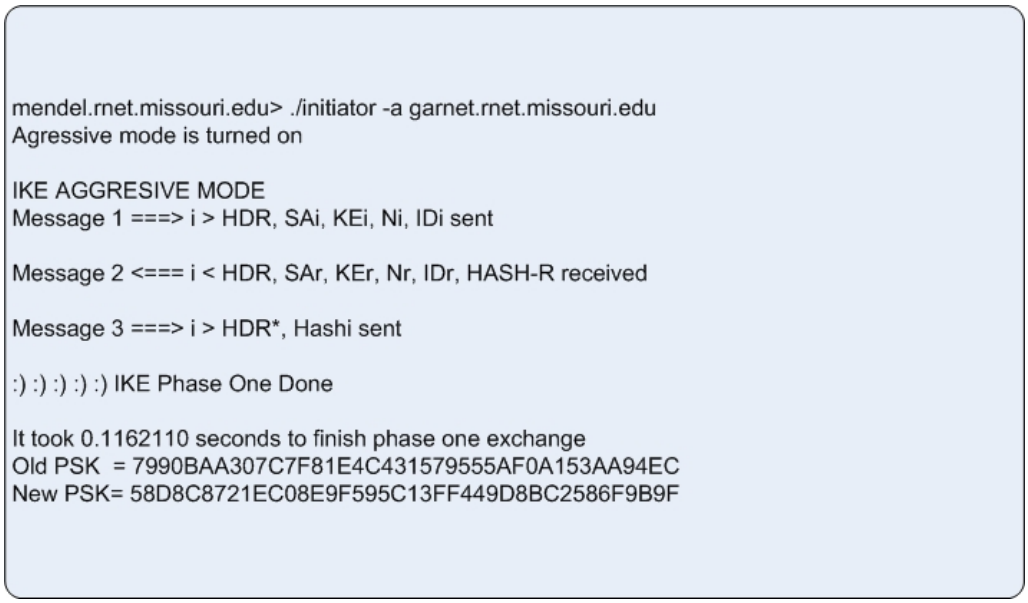

Figure 8.18: Running the Replacement Module in the Aggressive Mode

messages are exchanged in the Aggressive mode after which a new preshared key is generated.Note the *-a* flag in the commandline that instructs the initiator to run in the Aggressive Mode. The responder will run according to the mode specified in the "Exchange Type" field of the ISAKMP header of the received message (see Figure 5.1). Therefore, the responder will run in the Aggressive mode in this case. Note that the preshared key that was generated in the last run of the replacement module in the Main Mode (see Figure 8.17) is used as the preshared key in the Aggressive Mode. This indicates that it does not matter in which mode the new preshared key is generated. It will be used in any mode the next session will be run in.

Figure 8.19 shows the output sample when the debugging flag is enabled. The debugging flag is used to trace and pinpoint any error that might occur while running the

```
mendel.rnet.missouri.edu> ./initiator -b garnet.rnet.missouri.edu
Destination Host: garnet.rnet.missouri.edu
Current Preshared Key=58d8c872 1ec08e9f 595c13ff 449d8bc2  586f9b9f
Destination IP: 128.206.118.141 Initiator ID=80ce768e   Responder ID=80ce768d
Ci=0ad77d52 54aa7c0a  Ni=a158cc31 de66cf15 14bfbd79 be6bb98e 853b1290
IKE MAIN MODE
+++++++++ MESSAGE ONE PARAMETERS ++++++++++
Transform# 1=03000024 01010000 80010005 80020002 80030001 80040002 800b0001 000c0004 00007080
Transform# 2=03000024 02010000 80010005 80020001 80030001 80040002 800b0001 000c0004 00007080
Transform# 3=03000024 03010000 80010001 80020002 80030001 80040002 800b0001 000c0004 00007080
Transform# 4=03000024 04010000 80010001 80020001 80030001 80040002 800b0001 000c0004 00007080
Transform# 5=03000024 05010000 80010005 80020002 80030001 80040001 800b0001 000c0004 00007080
Transform# 6=03000024 06010000 80010005 80020001 80030001 80040001 800b0001 000c0004 00007080
Transform# 7=03000024 07010000 80010001 80020002 80030001 80040001 800b0001 000c0004 00007080
Transform# 8=03000024 08010000 80010001 80020001 80030001 80040001 800b0001 000c0004 00007080
Proposal Hdr=00000128 01010008  SA Hdr=00000134 00000001 00000001
ISAKMP Hdr=0ad77d52 54aa7c0a 00000000 00000000  01100200 00000000 00000150
Message 1 ===> i > HDR, SAi sent

Encryption Algorithm: 3DES Hash Algorithm :SHA1 Diffi_Hellman Group: 2:modp1024 Cr=68BA6771C77BD14A
Message 2 <=== i < HDR, SAr received

++++++++++ MESSAGE THREE PARAMETERS ++++++++++
Nonce Payload=00000018 a158cc31 de66cf15 14bfbd79  be6bb98e 853b1290
KE Payload=0a000084 b68292a1 7fbc4b6f 50c19c83  ebfc3d05 29802486 7b711b78 f841e1f4 ce456e67 ec642372
655ee7e0 42398b0d d6818c1b 4077a195 14302b3b 87a5fadd ce6a619a 296a5d1b dda0a24f 5ab31798 92eb1ebe
1ff47db2 fb9b85cc f46da519 acb3b257 173ee2af f44014d0 347624c2  fa6fd17d 080ce7f8 a17ef2cb 94b64e8c
9ee8489a
ISAKMP Hdr=0ad77d52 54aa7c0a 68ba6771 c77bd14a 04100200 00000000 000000b8
Message 3 ===> i > HDR, KEi, Ni  sent

Nr=DDF55368499F418C31C8E3374F554B46355C56EE
gy=B2D5D1ACCEE146406F9BDF6518A6D84AA3C4D717F296212305ADE692AF7A18BF4AC4B1B720B3FE8936
42E67CC5800678112117F963EF195F2EC4E7BB245ED25359FE453356C7732D867F13009FEC87D521939170D
9ECE4A67C0790E7207E5604EE550720C0D382FD5D0EA16609AD827ED3AB9C37575A54FE03EA20BD6138862C
Message 4 <=== i < HDR, KEr, Nr  received

IV=1758e156 06c15865 SKEYID=e0e72839 866d6856 1e023d91 39305f34 16011fbc
SKEYID_d=1d2f61dd 1ac08f1b 5b6a15cc 8a5f9e27 9c25483d
SKEYID_a=61b4baba 37bb423f efc371fe 4110e934 df336ca3
SKEYID_e=f769a259 5bf06978 e9e769d9 8ba56990 6f8491f8
Encryption Key=4b43ca1c ae400ba1 f28cceb3 0975d5ea 49359bb2 c70dc822

++++++++++ MESSAGE FIVE PARAMETERS ++++++++++
Hashi=e03c5bd4 0307b5a1 766707c8 2c2d03b5 1f98e8a2
Hash Paylaod=00000018 e03c5bd4 0307b5a1 766707c8 2c2d03b5 1f98e8a2
ID Payload=0800000c 011101f4 80ce768e
Unencrypted Data=0800000c 011101f4 80ce768e 00000018 e03c5bd4 0307b5a1 766707c8 2c2d03b5 1f98e8a2
Encrypted Data=07b1cc9d 399b1ea7 48659fbd ffc8c20b 39d9de51 f3c5dad8 662c93bb 483c338e 9ab40d5e f008d577
ISAKMP Hdr=0ad77d52 54aa7c0a 68ba6771 c77bd14a 05100201 00000000 00000044
Message 5 ===> i > HDR*, IDi, Hashi  sent

Undecrypted data=07b1cc9d 399b1ea7 7366392e 137febc0 6eacd5a1 565cbb1e 94eeaa37 eef88262 033adbf7
ee8ba0d6
Decrypted Data=0800000c 011101f4 80ce768d 00000018  f2270bdb 8e50d4c1 a81170ad e935fe9d 64449593
ID(Type=ID_IPV4_ADDR, Value=128.206.118.141)   Hashr=f2270bdb 8e50d4c1 a81170ad e935fe9d 64449593
Message 6 <=== i < HDR*, IDr, Hashr  received

:) :) :) :) :) IKE Phase One Done

It took 0.1250000 seconds to finish phase one exchange
Old PSK  = 58D8C8721EC08E9F595C13FF449D8BC2586F9B9F
New PSK= 27607AF99BA025E36E45B491F94EA2807F8D0173
mendel.rnet.missouri.edu>
```

Figure 8.19: Enable the Debugging Flag in the Main Mode

replacement module. The output gives a detailed content of each message sent or received. For example, it shows the current preshared key, the exchange mode, the content of each payload, the SKEYID–based secrets, encrypted and unencrypted data, and authenticating hashes. By comparing this output with the output of the responder's program (when running in the debugging mode also), a programmer can determine the cause of errors, if any, and fix it.

All the previous outputs of running the replacement module indicated the time it took to finish phase one exchange. It measures the time required to finish the IKE phase one preshared key authentication method under the proposed enhancement. This includes the time it takes to negotiate security policy, exchange key data, authenticate the communicating endpoints, and update the preshared key. To compare the performance of the enhanced

```
mendel.rnet.missouri.edu> ./initiator garnet.rnet.missouri.edu

IKE MAIN MODE
Message 1 ===> i > HDR, SAi sent

Message 2 <=== i < HDR, SAr received

Message 3 ===> i > HDR, KEi, Ni  sent

Message 4 <=== i < HDR, KEr, Nr  received

Message 5 ===> i > HDR*, IDi, Hashi  sent

Message 6 <=== i < HDR*, IDr, Hashr  received

:) :) :) :) :) IKE Phase One Done

It took 0.1171870 seconds to finish phase one exchange
Old PSK  = 27607AF99BA025E36E45B491F94EA2807F8D0173
New PSK= 27607AF99BA025E36E45B491F94EA2807F8D0173
```

Figure 8.20: Disable Key Generation in the Main Mode

IKE phase one preshared key authentication method with the performance of the original IKE phase one preshared key authentication method, the part that deals with generating

and updating the preshared key in the replacement module is disabled. Figure 8.20 shows the output result of this test. The initiator code is modified to remove the new preshared key generating code, then the code is recompiled and run.

The debugging flag is disabled in this case. Note that the old preshared key is the same as the new preshared key because key generation is disabled. It took almost the same time to finish phase one exchange when key generation is disabled compared to when it is enabled. This means that the proposed enhancement with its much higher level of security introduces very negligible impact, if any, on the performance of the phase one preshared key authentication method considering the timer resolution of the system clock in this example. See Section 9.5 for a complete analysis of the proposed enhancement performance.

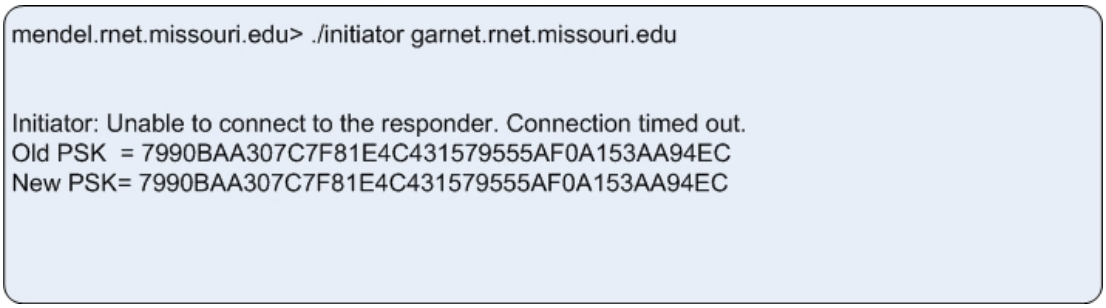

Figure 8.21: Output When Running the Initiator Only

To show the robustness of the proposed enhancement in keeping the two endpoints synchronized in terms of the preshared key, a simulated environment in which the IKE messages are lost (due to a MITM attack or unreliable connection) is simulated . The first scenario assumes that the responder is unreachable. This might be the result of applying a denial of service attack on the responder. This is simulated by not running the responder's program at all on the other side of the communication. In this case when the initiator's program calls the "connect" system call, it will time out before that connection is setup.

This forces the initiator to exit. However, it must exit without modifying the preshared key so that it stays synchronized with the other peer. This is shown in Figure 8.21. The preshared key before starting the initiator's program is the same after exiting the program. The preshared key at the responder is not changed also since the responder has not been run yet.

```
mendel.rnet.missouri.edu> ./initiator garnet.rnet.missouri.edu

IKE MAIN MODE
Message 1 ===> i > HDR, SAi sent

Initiator: error: connection lost with the responder.
Old PSK  = 7990BAA307C7F81E4C431579555AF0A153AA94EC
New PSK= 7990BAA307C7F81E4C431579555AF0A153AA94EC
```

Figure 8.22: Output When Exiting the Responder After the Second Message

A second scenario (see Figure 8.22) that might occur is an attacker intercepting the second message from the responder and blocking it from reaching the initiator. This is simulated by adding an exit statement in the responder's program after receiving the first message so as not to send the second message to the initiator. In this case, the "recv" system call at the initiator's side will wait for that message to arrive. After some time, the initiator's times out and exits indicating that the connection with the responder has been lost. However, before it exits, it makes sure that the preshared key has not been modified to keep synchronization with the responder.

Finally, sixth message from the responder might not arrive at the initiator. This is simulated by exiting the responder's program after receiving the fifth message. Figure 8.23 shows that the initiator recovered from that situation by restoring the preshared key file

```
mendel.rnet.missouri.edu> ./initiator garnet.rnet.missouri.edu

IKE MAIN MODE
Message 1 ===> i > HDR, SAi sent

Message 2 <=== i < HDR, SAr received

Message 3 ===> i > HDR, KEi, Ni  sent

Message 4 <=== i < HDR, KEr, Nr  received

Message 5 ===> i > HDR*, IDi, Hashi  sent

Initiator: error: connection lost with the responder.
Old PSK  = 7990BAA307C7F81E4C431579555AF0A153AA94EC
New PSK= 7990BAA307C7F81E4C431579555AF0A153AA94EC
```

Figure 8.23: Output When Exiting the Responder After the Fifth Message

to its old value. This guarantees that the two peers are synchronized in terms of the preshared key next time a connection is initiated. In the worst case that the two endpoints get out of synchronization, system administrators can reset the preshared keys when they are notified.

The implementation details of both the initiator and the responder along with some output samples from running the replacement module that implements the proposed enhancement have been presented and outlined in this chapter. The next chapter concludes with the security features of the proposed enhancement along with its performance analysis

Chapter 9

Conclusions

9.1 The Problem

After analyzing the preshared key authentication method in the original IKE protocol (see Chapter 6), it was shown that the method is not secure. The preshared key can be determined by applying off–line dictionary or brute–force attacks. Once the preshared key is compromised in the original IKE protocol, all subsequent communications are also compromised because the same preshared key is used over and over in the authentication process. In other words, the preshared key in the original IKE protocol is a constant, static key that has an infinite lifetime.

In this situation, the attacker has an infinite time frame to launch the off–line dictionary or brute–force attacks against the preshared key. Eventually, he/she will determine the preshared key in this unbounded time frame. In addition, this situation allows the attacker to benefit from the compromised key forever. It is enough for the attacker to launch the attack only once to compromise the static preshared key. Once the key is compromised, it is easy to compromise the channel between the initiator and the responder and access restricted data every time they communicate using a MITM attack.

This problem presented the need for modifying the preshared key authentication method. The attacker must not have the advantage of a compromised preshared key forever. He/she

must be faced with a limited time frame in which the off–line dictionary or brute–force attacks can be launched. As a result, the work on this dissertation was started.

9.2 The Proposed Solution

The goal of this dissertation is to present a simple and efficient way of modifying the IKE preshared key authentication method to enhance its security level. The proposed enhancement is based on the concept of dynamic preshared keys. Instead of using the same preshared key over and over in the authentication process of the phase one exchange, each exchange uses a new, different preshared key in the authentication process. The key is generated at the end of the phase one exchange successful authentication to be used in the authentication process of the next phase one exchange.

The generation of the new preshared key depends on the use of hash functions. The exact hash function to use is determined during the negotiation of the SA parameters. Input used to generate the new preshared key depends on a secret, among other parameters, shared only between the initiator and the responder (called the master key, see Section 7.4 for more details). The use of this secret overcomes the possibility of the new preshared key being generated by an attacker if a session is compromised.

The use of fresh, different preshared keys in each communication session limits the lifetime of a key. The preshared key is valid only for a short period of time, after which it is discarded and a new key is generated and used. The lifetime of a preshared key is the shorter of a session's lifetime (duration) or the 8–hour default lifetime for a key imposed by IKE protocol implementations.

Using the method of dynamic, limited lifetime preshared keys, the attacker is faced with a limited time frame to attack and compromise the preshared key. The key must be compromised during the lifetime of one session rather than an infinite lifetime as in the

original IKE protocol. Thus, by imposing a time bound on the preshared key, it becomes more difficult for the attacker to compromise the key since he/she must have many more resources to do so in a limited time frame that is at most 8 hours.

In addition, even if the preshared key is compromised someway or another in one session, the attacker cannot benefit from that compromised key forever since the key is used only in that specific session. After that session finishes (or passes the 8 –hour limit), a new preshared key is generated for the use of the next session. Therefore, only the information exchanged during the compromised session might be compromised. Future session are protected by different preshared keys that the attacker cannot compute or compromise based on compromising the currently used preshared key.

9.3 Security Features of the Proposed Enhancement

The major benefit that the proposed enhancement adds to the IKE protocol is increasing its security level dramatically. This is translated through the ability of the proposed enhancement to highly resist brute–force or off–line dictionary attacks. This resistance is shown in Table 9.1.

The table calculates the time it takes to compromise different preshared key lengths using a brute force attack assuming the proposed enhancement of the IKE protocol is deployed. The results in the table are based on the DES cracker machine, which is capable of trying $2.5 * 10^{11}$ DES operations per second [10]. Two different hashing algorithms are used in the calculations. The expected hashing performance of The DES cracker machine studied in Section 6.4. The machine is capable of trying $5.5 * 10^{12}$ preshared key hashes per second (on average) for the SHA–1 hashing algorithm and $6.75 * 10^{12}$ preshared key hashes per second (on average) for the MD5 hashing algorithm. (See Section 6.4 for details on how these numbers are calculated).

Relative Processing Power	Preshared Key Length (bytes)	Brute–Force Timing (years)	
		SHA–1	MD5
1	16	$9.7 * 10^{17}$	$7.9 * 10^{17}$
	20	$4.2 * 10^{27}$	$3.4 * 10^{27}$
10	16	$9.7 * 10^{16}$	$7.9 * 10^{16}$
	20	$4.2 * 10^{26}$	$3.4 * 10^{26}$
100	16	$9.7 * 10^{15}$	$7.9 * 10^{15}$
	20	$4.2 * 10^{25}$	$3.4 * 10^{25}$
1000	16	$9.7 * 10^{14}$	$7.9 * 10^{14}$
	20	$4.2 * 10^{24}$	$3.4 * 10^{24}$
10,000	16	$9.7 * 10^{13}$	$7.9 * 10^{13}$
	20	$4.2 * 10^{23}$	$3.4 * 10^{23}$
100,000	16	$9.7 * 10^{12}$	$7.9 * 10^{12}$
	20	$4.2 * 10^{22}$	$3.4 * 10^{22}$
1,000,000	16	$9.7 * 10^{11}$	$7.9 * 10^{11}$
	20	$4.2 * 10^{21}$	$3.4 * 10^{21}$

Table 9.1: Resistance of the Proposed Enhancement to Brute–Force Attacks

The resistance of the proposed enhancement to the brute–force and off–line dictionary attacks is verified against some hypothetical, very fast machines. Each of those machines has a processing power that is measured relatively to the processing power of the DES cracker machine. The more the relative processing power the more powerful a hypothetical machine is assumed to perform. The overall processing power can be increased either by increasing the processing power of the machine itself or combining the processing powers of a set of machines in a distributed environment. For example, a processing power of 100 means that we have a machine that is 100 times faster than the base DES cracker or that we have 100 DES cracker machines that are working collectively.

The table clearly shows that even with a processing power that is 1,000,000 times more powerful than the DES cracker machine, the proposed enhancement to the IKE protocol is still highly resistant to the brute–force and off–line dictionary attacks. For example, compared to the 8–hour (at most) period during which the preshared key is re-

quired to resist the brute–force and the off–line dictionary attacks, the proposed enhancement insures that it would take $9.7 * 10^{11}$ years, on average, to compromise the minimum preshared key length (16 bytes) hashed with the SHA–1 hashing algorithm even when a 1,000,000 times faster machine than the DES cracker machine is used. Furthermore, the preshared key lifetime does not have to last for the whole 8–hour period. If for any reason during that time concern is raised about the security of the communication, phase one can be terminated and reinitiated. In this case, a new preshared key is used. Therefore, it is enough to produce a key that can not be compromised during a session's life time (eight hours at most) in order for the key to resist brute–force and off-line dictionary attacks.

This huge security gain in the proposed enhancement is due to three main features added by the proposed enhancement to the original IKE protocol. The first one is the limit imposed on the lifetime of the preshared key. Due to that, the time frame in which the attacker has to successfully compromise the preshared key is brought down from an infinite time frame to an 8–hour period at most. This means that the proposed enhancement is deemed resistant to the brute–force and off–line dictionary attacks if the preshared key can not be compromised during the maximum eight–hour period. At the same time , the proposed enhancement insures that the attacker would take millions of billions of years to compromise a key generated by the proposed enhancement as shown in Table 9.1. Therefore, using a preshared key with a limited lifetime, as in the proposed enhancement, is very far from being compromised even when very fast and powerful machines are used.

The second feature added by the proposed enhancement is the use of hash functions for generating the new preshared key. This imposes a minimum length on the generated preshared key. The length of the generated preshared key in the proposed enhancement depends on the output length of the underlying hash function. The minimum length a generated preshared key can take in the proposed enhancement is 16 bytes when the

MD5 hashing algorithm is used. If the SHA–1 hashing algorithm is used the length of the generated preshared key will be 20 bytes. Furthermore, longer lengths for the generated preshared key, to provide more security, can be generated if other hash functions that produce longer output are used. Table 9.1 uses a minimum length of 16 bytes for the preshared key in calculating the brute–force attack timing.

The third added feature is the expanding of the possible set of characters a byte in the preshared key can take. Using hash functions to generate new preshared keys extends the possible choices a byte (character) can take. The preshared key is no longer limited by the set of characters found on a regular keyboard (small letters, capital letters, digits, and special characters) when it is setup by humans. All the 256 possible choices for a byte are possible since the key in the proposed enhancement is generated by the machine itself. Therefore, enumerating all the possible candidates for the preshared key in the proposed enhancement using a brute–force attack is made much more difficult because the set of possible choices for a character is larger, and the security level grows exponentially with the size of the possible set of characters.

For example, the time required to determine a 16–byte preshared key using the brute force attack with SHA–1 as the hashing algorithm is computed as follows in the original IKE protocol, where there are 52 possible characters for each byte (see Table 6.3):

$$Time = 0.5 \times \frac{52^{16} possible\ keys}{(5,500,000,000,000\ key/second)} = 8.2 * 10^6\ years$$

while in the proposed enhancement, using the same key length, the same hashing algorithm, and the 256 possibilities for each byte, the time required to compromise the preshared key is computed as follows:

$$Time = 0.5 \times \frac{256^{16} possible\ keys}{(5,500,000,000,000\ key/second)} = 9.7 * 10^{17}\ years$$

Recall that an attacker tries half the possible preshared keys on average to compromise the key using a brute–force attack. Therefore, by extending the possible set a byte in the preshared key can take, a huge security gain is a achieved. It is almost 6.5 million times more secure in this example just by extending the possible set of values a byte in the preshared key can take.

In addition, in the proposed enhancement the preshared key is no longer setup by humans (except the first time). New preshared keys are generated and setup by machines which make the setup process more effective, error–free, and scalable. Furthermore, the output of the hash functions is random, making the generated preshared key more resistant to the off–line dictionary attacks. This forces the attacker to use the brute–force attack which is very time consuming and extremely unlikely to succeed as shown in Table 9.1.

Another major security benefit of the proposed enhancement is that an attacker cannot apply the MITM attacks described in the Section 6.2 and the Section 6.3. To apply this attack in the original IKE protocol, an attacker has to compromise two consecutive sessions. In the first one, the attacker compromises the preshared key. In the second one, the MITM attack is launched to compromise the connection itself. The attack is successful because the same preshared key used in the first session is used again in the second session and all future sessions. This situation can never happen in the proposed enhancement since the next session will use a totally different preshared key than the one compromised in the previous session. Therefore, an essential and necessary condition for the success of the MITM attack can never happen in the proposed enhancement. As a result, this attack is completely disabled and blocked in the proposed enhancement.

Even if an attacker, in the worst case, is able to compromise a preshared key for

some reason or another (extremely unlikely, see Table 9.1), the proposed enhancement guarantees that the attacker will not be able to generate the new preshared key that will be used in the next phase one authentication. Only the initiator and the responder are able to generate the new preshared key because they use a secret in the calculation of the new preshared key that is only shared between them. This secret is called the master key. The rationale behind using the master key and its use in the new preshared key calculation are explained in the Section 7.4. Furthermore, the attacker cannot benefit from compromising the key after a new session, in which the preshared key is replaced by a new key, is started. However, in some scenarios the attacker *might* benefit from the compromised key. These scenarios and possible solutions to remedy them are explained in Chapter 7.

In all the discussion in this dissertation it is assumed that the preshared keys are saved in local files at each endpoint of the communication. These files are protected from unauthorized access by attackers. Otherwise, attackers can gain access to these keys and compromise all connections between the initiator and the responder. Therefore, files continuing secret data are encoded in some way that an attacker gains nothing if he/she can access them. Section 7.6 explains this issue and suggest a possible solution to protect the files contents.

9.4 Proposed Enhancement Prototype

The proposed enhancement in this dissertation is designed, implemented, and tested to show the viability of the proposed enhancement to be functional. It is implemented as a programming module that is intended to replace part of the IKE phase one exchange code that deals with the preshared key authentication method. When that part is replaced, existing applications can use the modified IKE, which has a higher level of security, together with the IPsec protocols to protect their communication in the same way being done now.

The programming module that implements the proposed enhancement preserves the architecture of the IKE phase one messages so that it can be easily adapted in future implementations of the IKE protocol without any significant code change. It uses the same architecture as the current IKE protocol and supports both modes of operation of the phase one exchange. It does not involve an increase in the number of messages exchanged, change in the order in which the messages are exchanged, or require the creation of new types of messages. All of this is preserved while at the same time providing a much higher level of security.

The proposed enhancement is designed and implemented through a traditional client/server application over the TCP protocol. It is composed of two programs: the initiator and the responder. Each program runs on a different machine (cannot run both at the same machine since they use the same port number for communication). A user has control over the program through commandline arguments (see Appendix A). The proposed enhancement programs are not used by themselves as a complete application. The purpose of creating the programs is to provide a proof that the proposed enhancement can work as expected. They use one preshared key in the current session, generate another key for the next session before exiting the current session, and use that generated key in the next session. This is demonstrated in Figure 9.1. The application is run twice to show that the generated preshared key in the first run is used in the second run. The figure shows only the output result of initiator's side. Preshared keys 20–bytes long are used in this example since the SHA–1 hashing algorithm is used for generating the preshared key.

The programs were successfully and smoothly tested more that 200 times. Each time a new preshared key was generated in one session and used in the next session. At all times the two endpoints of the communications continued to be synchronized in terms of the generated preshared key. To test the synchronization under different situations, the

```
mendel.rnet.missouri.edu> ./initiator garnet.rnet.missouri.edu

IKE MAIN MODE
Message 1 ===> i > HDR, SAi sent

Message 2 <=== i < HDR, SAr received

Message 3 ===> i > HDR, KEi, Ni  sent

Message 4 <=== i < HDR, KEr, Nr  received

Message 5 ===> i > HDR*, IDi, Hashi  sent

Message 6 <=== i < HDR*, IDr, Hashr  received

:) :) :) :) :) IKE Phase One Done

It took 0.1169770 seconds to finish phase one exchange
Old PSK  = 11166DACD154B352C42EBF874967107BBC61194F
New PSK= BC85D9F35F4E1BFDA9F96A043AFA52E22C242723


mendel.rnet.missouri.edu> ./initiator garnet.rnet.missouri.edu

IKE MAIN MODE
Message 1 ===> i > HDR, SAi sent

Message 2 <=== i < HDR, SAr received

Message 3 ===> i > HDR, KEi, Ni  sent

Message 4 <=== i < HDR, KEr, Nr  received

Message 5 ===> i > HDR*, IDi, Hashi  sent

Message 6 <=== i < HDR*, IDr, Hashr  received

:) :) :) :) :) IKE Phase One Done

It took 0.1170870 seconds to finish phase one exchange
Old PSK  = BC85D9F35F4E1BFDA9F96A043AFA52E22C242723
New PSK= 7990BAA307C7F81E4C431579555AF0A153AA94EC
```

Figure 9.1: Running the Replacement Module in the Main Mode

initiator was run once alone without starting the responder in the other side. Another time, the responder program was modified to exit after receiving the fifth message to see what the final result would be. Again, the two peers kept their synchronization in terms of the preshared key. Figure 9.2 shows the output of this test. More output results from running the proposed enhancement programs can be found in the Section 8.11.

```
mendel.rnet.missouri.edu> ./initiator garnet.rnet.missouri.edu

IKE MAIN MODE
Initiator: Unable to connect to the responder. Connection time out.
Old PSK  = 7990BAA307C7F81E4C431579555AF0A153AA94EC
New PSK= 7990BAA307C7F81E4C431579555AF0A153AA94EC



mendel.rnet.missouri.edu> ./initiator garnet.rnet.missouri.edu

IKE MAIN MODE
Message 1 ===> i > HDR, SAi sent

Message 2 <=== i < HDR, SAr received

Message 3 ===> i > HDR, KEi, Ni  sent

Message 4 <=== i < HDR, KEr, Nr  received

Message 5 ===> i > HDR*, IDi, Hashi  sent

Initiator: error: the responder is unreachable
Old PSK  = 7990BAA307C7F81E4C431579555AF0A153AA94EC
New PSK= 7990BAA307C7F81E4C431579555AF0A153AA94EC
```

Figure 9.2: Key Synchronization in the Replacement

9.5 Performance Impact of the Proposed Enhancement

The performance of the proposed enhancement has been analyzed to determine its impact on the IKE phase one preshared key authentication method performance. This is done by determining the total processing time for the IKE phase one communication to finish. The Main Mode exchange is chosen to run this test. The Aggressive Mode can be used, and it gives similar results.

The time required to finish the IKE phase one communication in the Main Mode is measured twice using the programming module that is built in this dissertation. The generation of the preshared key is disabled in the first time to simulate the current IKE phase one communication. In this case, the measured value represents the time it takes the IKE

protocol to go through the actual negotiation and authentication of the phase one communication without generating the new preshared key. In the second case, generating the new preshared key is enabled to simulate the modified IKE phase one exchange, and the time to finish the phase one exchange is measured. The measured value represents the time it takes to finish the phase one exchange in the Main Mode using the proposed enhancement. This includes going through the actual negotiation and authentication of the phase one exchange plus the time it takes to generate the new preshared key. This helps to determine the impact of the proposed enhancement if it is adopted in the current IKE protocol. A HP/Comapq machines running at 500 MHz was used to run the tests. At the time of testing, the machine was lightly loaded since it was not running any production code or jobs that might affect the timing results. Table 9.2 shows the timing results obtained.

Trial No. No.	**Time (sec) (Without Preshared) (Key Generation)**	**Time (sec) (With Preshared) (Key Generation)**
1	0.117188	0.117187
2	0.117188	0.117187
3	0.117187	0.117187
4	0.117187	0.117187
5	0.117187	0.117187

Table 9.2: Time to Finish Phase One Exchange (in sec)

The timing data clearly shows that the proposed enhancement, which enables a much higher level of security, has a negligible effect on the time required to finish the phase one exchange. Considering the timer resolution, it takes almost the same time to finish the modified IKE phase one exchange as it takes the current, unmodified protocol. Therefore, the performance of systems that adopt the IKE phase one proposed enhancement will not be affected, yet at the same time its security level will be much higher.

The timing measurements are collected by running the program that implements the proposed enhancement on two HP/Compaq Tru64 Unix Machines. The initiator part of the program runs on one machine and the responder part runs on the other. Table 9.2 is a month summary of measurements. during this period, timing measurements were taken five times a day, and at the end of the month an average value for each of the five times was computed.

9.6 Shortcomings and Future Work

The proposed enhancement must be incorporated on each peer involved in the phase one exchange before it can be used successfully. This imposes a problem if one side supports the enhancement and another does not. The peers will go out of synchronization after the first run of the protocol because they will update their preshared keys inconsistently. One peer updates its preshared key while the other does not, which results in different preshared keys at the endpoints of the communication.

Unless all entities using the IKE are updated to use the proposed enhancement, there will be a transition period in which some entities are updated and some are not. To enable communication between these two sets of peers, a flag is introduced in the SA offer sent by the initiator. If set by the initiator in the SA offer, and set also in the SA reply from the responder, the two peers can use the modified version of the protocol. If the flag is not set in the sender's reply or is not offered originally by the initiator, the two peers use the original, unmodified IKE protocol.

The functionality of the flag can be easily adopted in the "reserved" field of the current SA generic header (see Figure 5.2). Currently this field carries the value of 0, which can be used to indicate the original IKE protocol (version 1.0). Any other value in this field, such as 1, can be used in IKE implementations that adopt the proposed enhancement to

indicate the modified, enhanced version of the IKE protocol (version 1.1). If the new version is not supported in either peer, the peers scale down to use the original IKE protocol. A complete explanation of this compatibility issue is found in Section 7.7.

The introduction of a transition period might be utilized by an attacker to force the communicating peers to scale down to use the original IKE protocol even though both support the enhanced version. This is discussed in Section 7.8. However, this problem is applicable only as long as both versions of the IKE protocol are supported. That is, during the transition period only. After the transition period ends and the original IKE protocol is turned off, the higher security level of the proposed enhanced will be fully utilized.

Another limitation resulting from the use of the proposed enhancement of the IKE protocol is disallowing users to share a preshared key (i.e., a group key). Each user must have a separate preshared key when communicating with the responder machine to utilize the security level of the proposed enhancement. When a phase one exchange finishes, this preshared key is updated and kept synchronized between the user's side and the responder's side.

However, the original IKE protocol allows a set of users to have the same ID and preshared key to remotely connect with the responder. This is not due to the IKE protocol itself, but an application's use of the IKE protocol that assumes that the preshared key is static. In fact, this feature is utilized to setup Virtual Private Networks (VPNs) for users to access remote resources. A group of users who share a common group name and group preshared key can remotely access an institution's VPN using this feature.

However, in this case the proposed enhancement of the IKE protocol can not be used since users will be out of synchronization in terms of the preshared key. Using the enhanced IKE protocol described in this thesis, any remote user that successfully connects to the responder machine will update the preshared key on the responder's side. This

update takes effect only between the current user and the responder machine. Other users will not be synchronized and will have no knowledge of the new preshared key generated. Therefore, they will not be able to connect to the responder machine in the future.

This limitation can be avoided simply by providing each user with a separate preshared key. In this case, the responder's machine must keep track of many more preshared keys rather than a single group preshared key. However, if it is not desirable to prohibit the group preshared key on which some applications depend to work correctly, the proposed enhancement can accommodate the group preshared key by introducing a flag in the SA payload. When this flag is set by the initiator, it tells the responder that a group preshared key is used in this exchange and consequently prohibits the responder from updating the preshared key. Any malicious modification to this flag will be noticed since the initiator's SA offer is used as an input in calculating the authenticating hashes. Therefore, an attacker can not force a responder to update its preshared key, by clearing the flag, if the client is not asking for that. This solution allows the current applications that use a group preshared key to keep working over the enhanced version of the IKE protocol. However, these applications will not benefit from the security features of the enhanced IKE version. In this case, a security level equivalent to the original IKE protocol security level is provided. Therefore, the proposed enhancement can accommodate the group preshared key. However, doing so forbids the communicating peers from a higher level of security the proposed enhancement is trying to provide.

These limitations of the proposed enhancement were realized early on. However, their solution necessitates modification of the IKE protocol architecture and the payloads exchanged contrary to the requirements and criteria of the proposed enhancement design described in Section 8.2. Therefore, the limitations can be overcome if no transition period in which the two versions of the IKE protocol coexist is provided, or if changing

the protocol architecture or the environment in which it is used is permitted. However, since the premise of this thesis was not to change the protocol, and provide for a transition period, the limitations exist and further work would be required to accommodate the variety of applications that depend on the original protocol with a static or group preshared key.

Despite the noted limitations, the enhanced IKE version that limits the exposure to attack should be given serious consideration by those desiring enhanced security protection. Furthermore, while there may be some who might not benefit from the full security level of the proposed enhancement due to these limitations, the majority of the users will enjoy a higher level of security. For those who cannot use the enhanced protocol due to the limitations, the security exposure will be no worse than the existing IKE protocol.

Additional potential future work is to build a working version of the proposed enhancement in a real implementation of the IKE protocol. The code that is built in this dissertation can be linked to an existing code of the IKE implementation in the kernel to replace the part of the phase one exchange that deals with the preshared key authentication method. This can be done by taking open source code for the IKE implementation, such as the Linux Free S/WAN implementation, and replacing the necessary IKE code with the modified version, then link the code to the kernel. Based on the security level the proposed enhancement guarantees, it is highly recommended that the enhancement be adapted by all IKE implementations.

Using TCP to implement the proposed enhancement increases the time required to finish the IKE phase one exchange due to the slow connection setup time of the TCP protocol compared to the UDP protocol. One might argue that this introduces a delay in setting up the phase one SA. However, requiring a faster protocol does not justify the use of an insecure one. The design goal of the IKE is to provide a secured environment for the

parties involved in an exchange. Therefore, satisfying this goal justifies the introduction of a little delay in the setup process.

9.7 Summary

We have shown that there is a weakness in the IKE preshared key authentication method. An attacker can obtain the preshared key value using a brute–force or off–line dictionary attack. After that, a MITM attack can be launched to compromise all future connections. To remedy this weakness, we have proposed an improvement that makes the preshared key dynamic. Instead of using the same preshared key repeatedly, a new one is created and agreed upon each time a new phase one exchange is initiated. This enhancement imposes a strict time limit on the preshared key life compared to the infinite lifetime used in the original IKE protocol. Given infinite time, the key length will have to keep getting longer and longer, but putting a time frame of the exposure considerably reduces the vulnerability of brute–force and off–line dictionary attacks.

The security level of the proposed enhancement was verified to be much more secure compared to the original IKE protocol. The security gain resulting from using the proposed enhancement over the original IKE protocol is much higher. This is shown in Table 9.1.

The performance of the proposed module has been analyzed. It has been shown that the module has almost no effect on the time it takes to finish the IKE phase one exchange. At the same time, the module provides a much higher level of security since the new generated key can resist the off–line dictionary attacks. Therefore, It is highly recommended to import the proposed enhancement to real–life implementations of the IKE protocol.

Appendices

Appendix A

Running The Phase One Exchange Replacement Module

This appendix serves as a guide for users on how to run the phase one exchange replacement module. The replacement module is an application written in the C programming language. It implements the proposed enhancement discussed in Chapter 7, which improves the security level of the IKE preshared key authentication method.

The proposed enhancement depends on the use of dynamic preshared keys. Instead of using the same preshared key over and over in the authentication process of each phase one exchange, a new key is generated after a successful phase one authentication and used in the next phase one authentication process. This insures that a preshared key will have a limited lifetime determined by the session's lifetime (maximum of eight hours). Therefore, the amount of time an attacker has to successfully compromise the key value is brought down from an indefinite value to eight hours at a maximum, which is a very short time for the attacker to compromise the key.

Users have control over the replacement module through commandline arguments. The module is composed of two programs, *initiator* and *responder*, that are run on different machines at the same time to build an IKE communication channel between them. The programs must be run on different machines because they use the same port number

for exchanging IKE massages. The two programs cannot run on the same machine unless the default port numbers they use for communication are overridden. If the two programs run on the same machine without changing the default port numbers, an error message is generated indicating that the prot number is already in use.

The format of the initiator's command line is:

```
prompt> initiator [options] <host>
```

The `<host>` is the responder's hostname or IP address. It must be included in the command since it is not an optional argument. Currently, only IPv4 addresses are supported. However, IPv6 could be easily incorporated if desired for additional testing. Different options can be specified at the commandline. Unless overridden by the user, the default values of these options are used. Following are the supported options in the initiator's program.

`-s <source port number>` This option specifies a port number for the initiator to use in communicating with the responder. If the user does not specify this option, the initiator's programm uses the default port of 5500. If the user wants to use a different port, he/she must specify that port number. For example, the command `"initiator -s 5000 mendel.rnet.missouri.edu"` sets the initiator to use port 5000 to communicate with the responder which is supposed to be running at mendel.rnet.missouri.edu over the default port of 5500.

`-d <destination port number>` This option specifies which port number the responder is using to accept incoming IKE messages from the initiator. In this case it is assumed that the initiator knows the port number on which the responder is running. For example, running the following command at the initiator's side `"initiator -s 5000 -d 6000 mendel.rnet.missouri.edu"` sets

the initiator to use port 5000 to communicate with the responder which is supposed to be running at mendel.rnet.missouri.edu over the port number 6000. Note that in this case, the two programs can be run on the same machine since they are using different port numbers.

`-f <preshared key filename>` This option specifies what filename is used to read and save the preshared key value. The default value for filename is *secret.txt*.

`-g <group number>` This option is used to determine the group number that will be used in this session of communication. Diffie-Hellman groups are used to determine the length of the base prime numbers used during the key exchange process. Group 1 provides 768 bits of keying strength, Group 2 provides 1024 bits. If not overridden, the default group number is 2.

`-a` This option enables the Aggressive Mode. The default mode used in communication is the Main Mode. This allows the users to choose either mode for communication between the initiator and responder programs.

`-b` This option enables the debugging mode. It is disabled by default. This option is helpful in tracing and determining errors when running the initiator's program. It changes the output format of the results to include more details and information about the exchanged messages. For example, it outputs the values of cookies, nonces, Diffie–Hellman keys, SKEYID secrets, and each message content.

The responder is run on another machine (unless the default port number is overridden) with the following command line format:

```
prompt> responder [options]
```

The responder program does not require the user to specify a host name since it does not initiate connections. It waits for connections to arrive from the initiator. The optional arguments in the command include the following:

`-s <source port number>` This option specifies the port on which the responder will be running. Note that the source port of the responder is the destination port of the initiator. If this option is not used by the user, the responder runs on the default port of 5500. For example, the command ***responder -s 5000*** runs the responder on the current machine on port 5000. The initiator must use this port number in its command to connect to the responder.

`-f <preshared key filename` This option enables users to specify a filename from which the preshared key value is read and written. This name should not necessarily be the same as the one on the initiator's side. The filename has a local effect only.

`-b` This option enables the debugging mode, which is disabled by default. It is similar to the `-b` option in the initiator commandline options. It outputs the values of cookies, nonces, Diffie–Hellman keys, SKEYID secrets, and each message content.

Note that the responder does not specify the mode it has to run in. This is specified when the first message from the initiator arrives. That message contains the mode which the initiator is offering for communication. The responder switches to work in that mode accordingly. In addition, note that the responder does not specify a destination port number since it does not initiate any connection. It knows the port number of the initiator's side from the TCP header of the first message. It uses that port number to send replies to the initiator.

Finally, Figure A.1 shows the help message displayed when a user enters a wrong

commandline option on the initiator's side. More output results from running the replacement module are provided in Section 8.11.

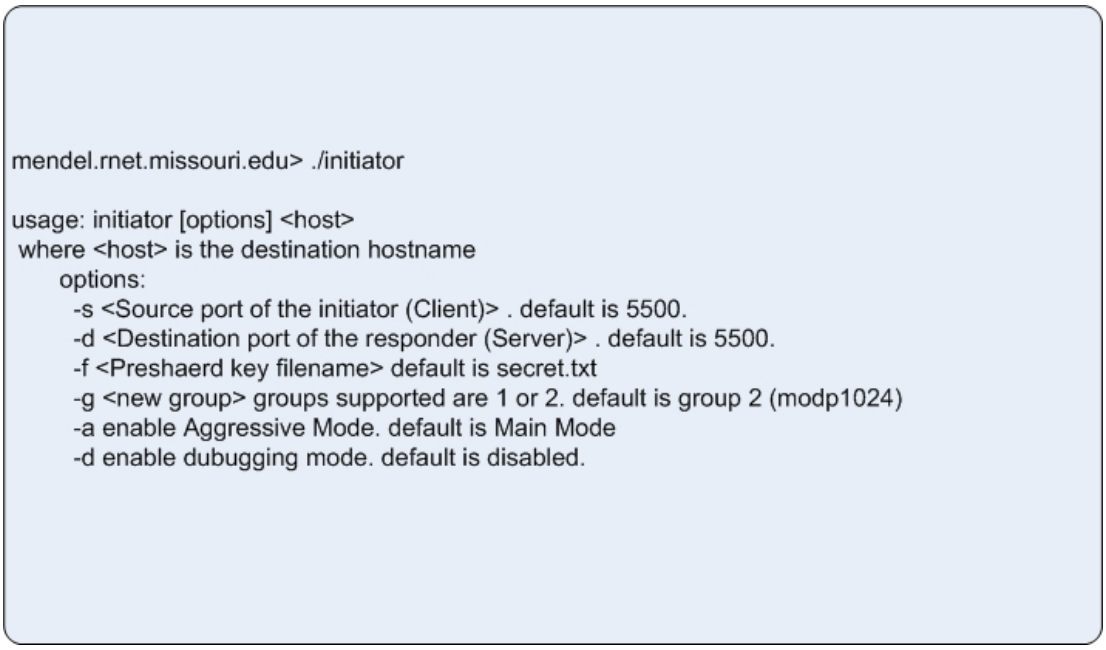

Figure A.1: Usage Message

Bibliography

[1] **T. Dierks and C. Allen**. *The TLS Protocol Version 1.0*. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFC 3546.

[2] **J. Postel**. *Transmission Control Protocol*. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[3] **R. Fielding and J. Gettys and J. Mogul and H. Frystyk and L. Masinter and P. Leach and T. Berners-Lee**. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[4] **E. Rescorla**. *HTTP Over TLS*. RFC 2818 (Informational), May 2000.

[5] **J. Postel**. *User Datagram Protocol*. RFC 768 (Standard), August 1980.

[6] **D. Harkins and D. Carrel**. *The Internet Key Exchange (IKE)*. RFC 2409 (Proposed Standard), November 1998. `http://www.ietf.org/rfc/rfc2409.txt`.

[7] **Bruce Schneier**. *Applied Cryptography*. John Wiley and Sons, 2nd edition, 1996.

[8] **William Stallings**. *Network Security Essentials*. Prentice-Hall, 2nd edition, 1999.

[9] **U.S. National Institute of Standards and Technology (NIST)**. *Data Encryption Standard. Federal Information Processing Standards Publication 46-3-2 (FIPS PUB 46-3)*, October 1999. `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.

[10] **Electronic Frontier Foundation**. *Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design*. O'Reilly, first edition, 1998.

[11] **U.S. National Institute of Standards and Technology (NIST)**. *Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197 (FIPS PUB 197)*, November 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[12] *Advanced Encryption Standard, The Latest Encryption Algorithm*. `http://www.secureage.com/webpages/enews1.shtml`.

[13] **Tariq Jamil**. *The Rijndeal Algorithm*. *IEEE Potentials*, 23(2):36–38, April/May 2004.

[14] **U.S. National Institute of Standards and Technology (NIST)**. *DES Modes of Operation*. *Federal Information Processing Standards Publication 81 (FIPS PUB 81)*, December 1980. `http://csrc.nist.gov/publications/fips/fips81/fips81.htm`.

[15] **U.S. National Institute of Standards and Technology (NIST)**. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. *Federal Information Processing Special Publication 800-38A*, 2001. `http://csrc.nist.gov/CryptoToolkit/modes/800-38_Series_Publications/SP8%00-38A.pdf`.

[16] **D. Eastlake 3rd and P. Jones**. *US Secure Hash Algorithm 1 (SHA1)*. RFC 3174 (Informational), September 2001. `http://www.ietf.org/rfc/rfc3174.txt`.

[17] **R. Rivest**. *The MD5 Message-Digest Algorithm* . RFC 1321 (Informational), April 1992. `http://www.ietf.org/rfc/rfc1321.txt`.

[18] **H. Krawczyk and M. Bellare and R. Canetti**. *HMAC: Keyed-Hashing for Message Authentication.* RFC 2104 (Informational), February 1997. `http://www.ietf.org/rfc/rfc2104.txt`.

[19] **Martin E. Hellman**. *An Overview of Public Key Cryptography*. *IEEE Communications Magazine*, pages 42–49, May 2002.

[20] **Diffie, Whitfield and Hellman, Martin E.** *New Directions in Cryptography*. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976. `citeseer.ist.psu.edu/diffie76new.html`.

[21] **E. Rescorla**. *Diffie-Hellman Key Agreement Method*. RFC 2631 (Proposed Standard), June 1999. `http://www.ietf.org/rfc/rfc2631.txt`.

[22] **Douglas R. Stinson**. *Cryptography: Theory and Practice*. CRC Press, third edition, 1995.

[23] **U.S. National Institute of Standards and Technology (NIST)**. *Digital Signature Standard. Federal Information Processing Standards Publication 186-2 (FIPS PUB 186-2)*, October 2001. `http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf`.

[24] **Naganand Doraswamy and Dan Harkins**. *IPSec: The New Security Standard for the Internet, Intranets and Virtual Private Networks*. Prentice-Hall, 1st edition, 1999.

[25] **J. Postel**. *Internet Protocol*. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[26] **S. Deering and R. Hinden**. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard), December 1998. `http://www.ietf.org/rfc/rfc2460.txt`.

[27] **Michael S. Borella**. *Methods and Protocols for Secure Key Negotiation Using IKE*. *IEEE Network*, 14(4):18–29, July/August 2000.

[28] **John Viega and Matt Messier**. *Secure Programming Cookbook for C and C++*. Oreilly, First edition, 2003.

[29] **D. Eastlake 3rd**. *Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH)*. RFC 4305 (Proposed Standard), December 2005. `http://www.ietf.org/rfc/rfc4305.txt`.

[30] **S. Kent and R. Atkinson**. *Security Architecture for the Internet Protocol*. RFC 2401 (Proposed Standard), November 1998. `http://www.ietf.org/rfc/rfc2401.txt`.

[31] **D. Maughan and M. Schertler and M. Schneider and J. Turner**. *Internet Security Association and Key Management Protocol (ISAKMP)*. RFC 2408 (Proposed Standard), November 1998. `http://www.ietf.org/rfc/rfc2408.txt`.

[32] **S. Kent and R. Atkinson**. *IP Encapsulating Security Payload (ESP)*. RFC 2406 (Proposed Standard), November 1998. `http://www.ietf.org/rfc/rfc2406.txt`.

[33] **P. Karn and P. Metzger and W. Simpson**. *The ESP DES-CBC Transform*. RFC 1829 (Proposed Standard), August 1995. `http://www.ietf.org/rfc/rfc1829.txt`.

[34] **C. Madson and N. Doraswamy**. *The ESP DES-CBC Cipher Algorithm With Explicit IV*. RFC 2405 (Proposed Standard), November 1998. `http://www.ietf.org/rfc/rfc2405.txt`.

[35] **M. Oehler and R. Glenn**. *HMAC-MD5 IP Authentication with Replay Prevention*. RFC 2085 (Proposed Standard), February 1997. `http://www.ietf.org/rfc/rfc2085.txt`.

[36] **C. Madson and R. Glenn**. *The Use of HMAC-SHA-1-96 within ESP and AH*. RFC 2404 (Proposed Standard), November 1998. `http://www.ietf.org/rfc/rfc2404.txt`.

[37] **H. Orman**. *The OAKLEY Key Determination Protocol*. RFC 2412 (Informational), November 1998. `http://www.ietf.org/rfc/rfc2412.txt`.

[38] **Hugo Krawczyk**. *SKEME: A Versatile Secure Key Exchange Mechanism for the Internet*. pages 114–127. IEEE Proceedings of the 1996 Symposium on Network and Distributed Systems Security, 1996. `citeseer.ist.psu.edu/krawczyk96skeme.html`.

[39] **John Pliam**. *Authentication Vulnerabilities in IKE and Xauth*. `http://www.ima.unm.edu/~pliam/xauth/`, October 1999.

[40] *Brute force attacks on cryptographic keys*. `http://www.cl.cam.ac.uk/~rnc1/brute.html`.

[41] **Michael Roe**. *Performance of Symmetric Ciphers and One-way Hash Functions*. pages 83–89. Fast Software Encryption, Cambridge Security Workshop Proceedings, LNCS Springer Verlag, 1994.

[42] **Mudge**. *L0phtcrack 1.5 Lanman/NT Password Hash Cracker*. http://www.insecure.org/sploits/l0phtcrack.lanman.problems.html.

[43] **C. Kaufman**. *Internet Key Exchange (IKEv2) Protocol*. RFC 4306 (Proposed Standard), December 2005.

[44] **Microsoft TechNet**. *Step–by–Step Guide to Internet Protocol Security (IPSec)*, February 2000. http://www.microsoft.com/technet/prodtechnol/windows2000serv/howto/isps%tep.mspx.

[45] **Microsoft TechNet**. *Exploring Peer–to–Peer IPsec in Windows 2000*, May 2001. http://www.microsoft.com/technet/community/columns/cableguy/cg0501.mspx%.

[46] *openssl Utility Program*. http://www.openssl.org.

[47] *openssl Online Documentation*. http://www.openssl.org/docs/.

[48] **Frederick J. Hirsch**. *Introducing SSL and Certificates using OpenSSL*. *World Wide Web Journal*, 2(3), 1997.

[49] *Ethereal Network Protocol Analyzer*. http://www.ethereal.com.

[50] **Brad Hards**. *A Guided Tour of Ethereal*. *Linux Journal*, (118), February 2004.

[51] **Microsoft TechNet**. *Key management and protection*, January 2005. http://technet2.microsoft.com/WindowsServer/en/Library/ec4bc2a7-3e89-48%c1-a16c-7dab4a2a11901033.mspx.

[52] **J. Zhou**. *Further Analysis of the Internet Key Exchange Protocol*. *Computer Communications*, 23(17):1606–1612, 2000.

[53] **Pau-Chen Cheng**. *An architecture for the Internet Key Exchange Protocol*. *IBM Systems Journal*, 40(3):721–746, 2001.

[54] **Niklas Hallqvist and Angelos D. Keromytis**. *Implementing Internet Key Exchange (IKE)*. Proceedings of the 9th USENIX Security Symposium. USENIX Denver, Colorado, August 2000.

[55] *libike Library Code*. `http://libike.cipherica.com/`.

[56] **T. Kivinen and M. Kojo**. *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*. RFC 3526 (Proposed Standard), May 2003.

VITA

Raed Bani-Hani was born October 02, 1976, in Irbid, Jordan. He attended public school in Jordan until 1994 when he joined the Jordan University of Science and Technology. He received his BS degree in Electrical and Computer Engineering in 1999. Then he pursued his graduate studies at the University of Missouri–Columbia. He received his MS in Computer Engineering in 2003, and PhD in Electrical and Computer Engineering in 2006. Presently he is a faculty member of the Computer Engineering Department at Jordan University of Science and Technology.