A Pattern for the Secure Shell Protocol

Kaivan Kaighobadi and Eduardo B. Fernandez
Department of Computer and Electrical Engineering and Computer Science
Florida Atlantic University
Boca Raton, FL 33431, USA,
kkaighob@fau.edu, ed@cse.fau.edu

Abstract

As the use of web-based and remote services grows each year, the concern of securing data shared between these services and its users has become a primary concern. A commonly employed framework for securing information and connections is Secure Shell (SSH). Secure Shell provides for a secure connection to remote login stations and for many common web protocols used for delivering web services. In this paper, we present a security pattern for the SSH protocol. This pattern focuses on the cryptographic methods employed by the SSH protocol to secure, and authenticate a connection and information exchanged over that connection.

Introduction

As the use of web-based services grows each year, the issue of securing the connections and data shared between a web-based service and its clients has become an issue of increasing concern. With web-based services for activities such as banking, e-mail, file transfers, and remote desktop connections already commonplace in today's world, the amount of sensitive data that is transferred over remote connections has grown drastically. This growth comes with the inherent risk of a greater number of people who wish to illegally obtain that information. Given this increased risk, we must ask the question: How do we secure this information from eavesdroppers, and how do we verify that the persons we are talking to are indeed who they claim to be?

A solution to those questions is provided in the form of the Secure Shell (SSH) protocol, commonly employed for applications used on Linux and UNIX based systems. SSH provides a secure communication environment by providing a method for system authentication, data confidentiality and integrity, and non-repudiation. Our pattern focuses on the cryptographic mechanisms of SSH. The capabilities we include are a key exchange mechanism, a public key cryptography mechanism, a message authentication mechanism, and a symmetric key cryptography system. Each of these capabilities has appeared as a pattern before [relevant sources here], but SSH combines these individual patterns in a unique way that results in a composite security pattern.

In section 2 we present our pattern while in section 3 we present some conclusions.

2. Secure Shell (SSH)

2.1 Intent

Provide an authenticated environment where users may utilize programs and securely exchange information with remote locations. This environment provides confidentiality via symmetric encryption, authentication via public key encryption, and non-repudiation via digital signatures.

2.1 Example

Bob has a 2 hour commute to work each way. The long drive requires him to wake up early and arrive home very late at night. Bob has the option to do his work from home, but he needs a secure way to transfer any information that he needs for work to and from the company network. Managers at his job will not allow him to work remotely because the kind of work he does involves sensitive data, and he does not have a secure means of transferring that data. If Bob could find a way to secure the information exchanged between home and work, he would be able to work remotely.

2.2 Context

Users who wish to perform actions on a system, retrieve information, or use a service from a remote location where the information exchanged between the remote user and the system travels wholly or in part over an insecure medium such as the Internet. We may also wish to make users accountable for their actions and be able to authenticate the origin of messages.

2.3 Problem

For many business, home, and government users, the ability to remotely access computer systems and exchange information is an important function. Those who do this may need this functionality in order to access an important program or access certain documents when it is impractical to do so in person. These users, more often than not, are required to keep secret the information that they handle as a result of a business or government mandate. Given these requirements, we are faced with several questions: How can we ensure that information exchanged between two systems is secured from eavesdroppers; how do we make users accountable for their actions and ensure the origin of a message; how do we ensure that a system cannot be impersonated by an attacker?

Solutions to this problem are constrained by the following forces:

- *Confidentiality* – Information must be kept secret during transit. This must be accomplished even on insecure channels.
- *System Authentication* – Each system must be able to authenticate the identity of any other system it communicates with to avoid impersonation.
- *Message Authentication* – Each system must be able to authenticate the source and correctness of any messages that it receives.
- *Integrity* – Messages sent between two systems must be free of errors or intentional modification during transit.
- *Non-Repudiation* – A user cannot plausibly deny sending a message
- *Flexibility* – Security must be applicable in a number of different applications.
- *Transparency* – Security must be transparent once a secure connection has been established. Otherwise, use of the system would be inconvenient.
- *Low Overhead* – Securing a connection should not cause a sizable increase in the overhead of the connection, otherwise the security becomes a significant hindrance to the use of the connection.

2.5 Solution

The task of authenticating and securing system messages is accomplished using 3 separate mechanisms.

First, the use of an Internet Key Exchange (IKE) mechanism [Kau05] is employed in order to set up a shared private key. In this mechanism, each system first calculates half of a shared key using an exchange algorithm. The two systems then exchange their half of the key with the other system. Finally, each system runs the algorithm on the received portion of the key. The result for both systems is a matching value that is then used in a later mechanism.

In SSH, this exchange method is employed through Diffie-Hellman key exchange [Ylo06b] [Sta06]. Each system calculates a portion of the key E and F. These values are calculated using a modulo function, a large prime number P, a generator G, and two random numbers, X and Y between 1 and P - 1, to produce E and F such that $E = G^X \bmod P$ and $F = G^Y \bmod P$. These values are then exchanged, where the same algorithm is used by the receiving system to produce a matching value, K, such that $K = E^Y \bmod P$ and $K = F^X \bmod P$. This value, K, serves as the shared private key used with the symmetric key encryption (SKE) algorithm.

Second, systems are mutually authenticated through the use of a Public Key Infrastructure (PKI) mechanism that utilizes public key cryptographic methods [Sta06] [Ylo06b]. A message is first hashed, creating a smaller size message known as a digest. This digest is then encrypted using the sender's private key and sent to the receiver, along with a second copy of the message. Once received, the hashed message is decrypted using the sender's public key and is compared to a hash of the second copy of the message that is calculated by the receiver. If the two values match, the receiver can be assured that the message came from a specific system.

In SSH, the PKI begins with a message M. The message is hashed producing a value H such that H = hash(M). This reduces the size of the eventual signature and reduces computational time. The hash is then encrypted (E) using the sender's private key P_{rk} producing a signature, S, such that $S = E_{Prk}(H)$. S is then sent to the receiver along with the sender's public key, P_{uk} and a copy of M. The receiver verifies P_{uk} by cross referencing the key with a certificate authority. If the received key is verified to be the correct key for the source, the receiver then decrypts (D) S using P_{uk} to extract H. This is done such that $H = D_{Puk}(S)$. The receiver then locally calculates H[1] from the copy of M and compares the two hash values. If the values match, the sender is authenticated to the receiver.

Third, message integrity is protected using a message authentication mechanism [Kra97]. Messages are hashed using a keyed hashing function which utilizes a shared secret key between the sender and receiver. The function produces a Message Authentication Code (MAC), which is appended to the original message and sent to the receiver. The receiver then uses the same keyed hashing function on the received message, and it compares the result to the received MAC. If the values match, then the receiver can be sure that the message was not modified in transit.

SSH utilizes this functionality via the private key calculated using the IKE and an agreed upon MAC algorithm. Before a message is encrypted, the message (which may be signed or unsigned), along with a unique sequence identifier is hashed using the agreed upon MAC algorithm that utilizes the shared secret key calculated during the IKE to produce a MAC for that message. After the message has been encrypted, the MAC is appended to the end of the message in an unencrypted form, and is sent to the receiver, where the unencrypted message is verified against the MAC using the same algorithm.

Finally, messages exchanged between two authenticated systems are encrypted using a mechanism to avoid eavesdropping [Sta06]. In this algorithm, a message is encrypted using a symmetric key which is generated by the IKE. The encrypted message is sent to the receiver where the message is decrypted using the same key, producing a readable message. This algorithm assumes that the key shared between a given sender and receiver is secret and no other party knows the key that is being used.

In SSH, the use of a SKE algorithm relies on the results from the Diffie-Hellman key exchange. Once a shared value K has been produced, a message M (can be signed or unsigned using PKI) is encrypted using a symmetric key encryption algorithm producing a ciphertext message C such that $C = E_K(M)$. The cipher message C is sent to the receiver where the receiver decrypts the message using the same key to produce the original message M such that $M = D_K(M)$.

[1] In some instances, no copy of M is sent because M is calculated from previously exchanged data between the sender and receiver.

For all four mechanisms, the two systems must agree upon common algorithms to be used. This information is negotiated in the initial communication stages between each system. More information on this selection process can be found in [Ylo06b].

Structure

Figure 1 shows a class diagram for the SSH protocol.

A **System** is either the server or the client attempting to connect to the server. A system can act as a **Sender** or **Receiver**.

Each system has access to a **KeyExcangeAlgorithmList**. This list is a collection of various **PrimeGroups,** which contain a prime number and a generator for each prime number. Each **Sender** then uses a **ModularFunction** to create their half of the key and sends their half to the **Receiver** where the same function is run, producing the private key.

Each system has access to an **EncryptionAlgorithmList** that contains the different symmetric encryption methods. A **Sender** will encrypt a message using **SymmetricAlgorithm** and the private key. A **Receiver** will decrypt a message using **SymmetricAlgorithm** and the same private key.

Each **System** also has a **KeyPair** associated with it, one public and one private. Each **System** *may* also have a **Certificate** associated with it as well. Each public key and certificate is registered to a **CertificateAuthority**, while any private keys are kept secret on the system. A **system** may check with a **CertificateAuthority**, which may or may not be a local authority, to verify whether a given certificate or public key is valid for a given **System**.

Each **System** also has access to a **MACAlgorithmList** that contains the list supported message authentication algorithms. A **Sender** will use a **MACAlgorithm** to create a MAC for an unencrypted message, and will append the MAC to the end of the message after encryption. The **Receiver** will decrypt the message and create a MAC of that message by using the same algorithm, and will compare it to the received MAC

A given **Message** sent between two **Systems** may be a plain **Message, EncryptedMessage**, or a **SignedMessage** imbedded in an **EncryptedMessage**, with all **EncryptedMessages** including a MAC. If a message is signed, the **sender** will request that a **Signer** uses the **HashAlgorithm** and the **SignatureAlgorithm** to create a **Signature** for a given signed message. If the **Receiver** receives a signed message, they will request that a **Verifier** confirm the legitimacy of that signed message. This is started by accessing the **CertificateAuthority** to verify the legitimacy of a given public key, assuming a certificate exists for that key. The **Verifier** will then use the same **HashAlgorithm** and **SignatureAlgorithm** used by the **Sender** and compare the results with the hash decrypted using the sender's public key. If a message is encrypted, the **Sender** will create a MAC using the **MACAlgorithm**, and will append the MAC to the end of the encrypted **Message** or **SignedMessage,** created by the **SymmetricAlgorithm**. The **Receiver** will decrypt the received with the same algorithm, and will utilize the same authentication algorithm to verify the **Message** or **SignedMessage** against the MAC.

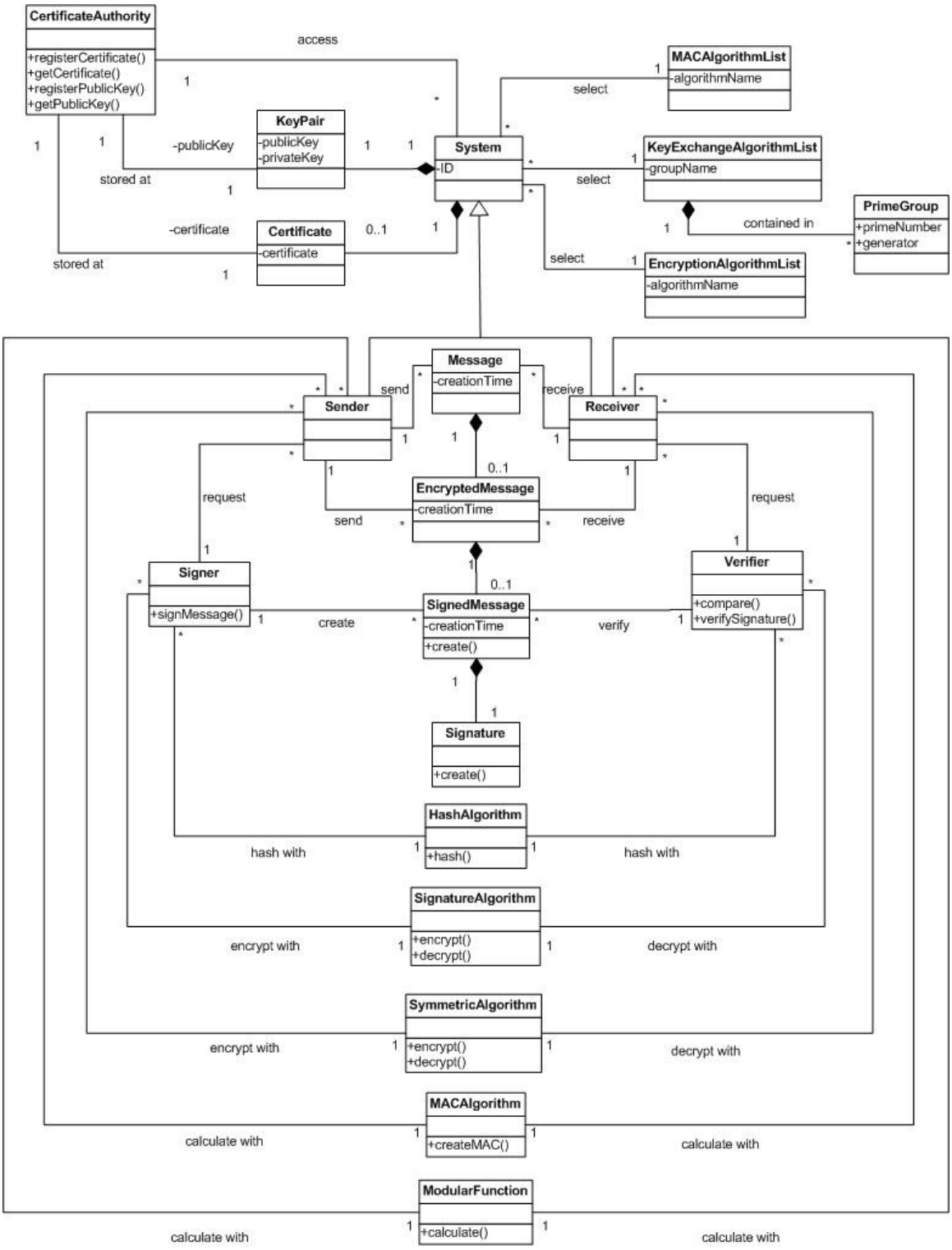

Figure 1: Class Diagram for Secure Shell Encryption, Authentication, and Key Exchange based on [Bau97]

Dynamics

We describe the dynamics of the use cases to create a shared private key, sign a message, and verify a signed message.

Create a shared private key (Figure 2)

Summary: Two systems wish to calculate a shared private key.
Actors: Two systems, both of which act as sender and receiver.
Precondition: The systems have negotiated what prime number and what generator will be used.
Description

1. Each system sends its prime number, generator, and random value to its modular function.
2. The modular function calculates, and returns, values 1 and 2 to its respective system.
3. Each system sends a message to the other system with their calculated value.
4. Each system again sends its prime number, generator, and the received value to its modular function.
5. The modular function calculates and returns matching key values to each system.

Postcondition: A value is created that is a shared private key.

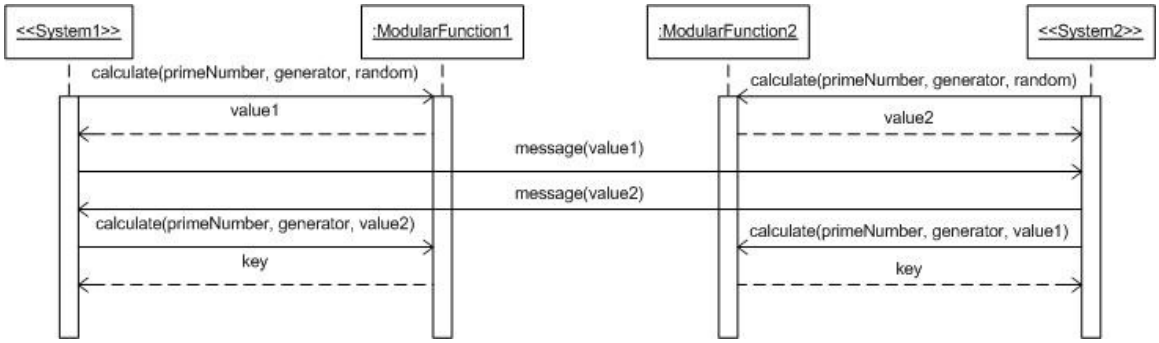

Figure 2: Sequence Diagram for creating a secret shared key.

Sign a message (Figure 3)

Summary: A sender wants to sign a message before sending it.
Actors: A sender
Precondition: A sender must have a public/private key pair with the public key registered to a certificate authority, and the sender and receiver have agreed upon a pair of PKE algorithms.
Description

1. A sender sends the message and its private key to the signer.
2. The signer calculates the hash value of the message using the HashAlgorithm.
3. The signer encrypts the hash value using the sender's private key and the SignatureAlgorithm. This creates the digital signature.
4. The signer creates the SignedMessage which contains the signature and the original message.

Alternate Flow

- The message sent to the signer in step 1 is a composite message consisting of previously exchanged information and is not from a new message (optional choice within the standard).
- The SignedMessage from step 5 does not contain the original message. Instead it includes the public key, and an optional certificate, and the composite message (optional choice within the standard).

Postcondition: A SignedMessage object has been created

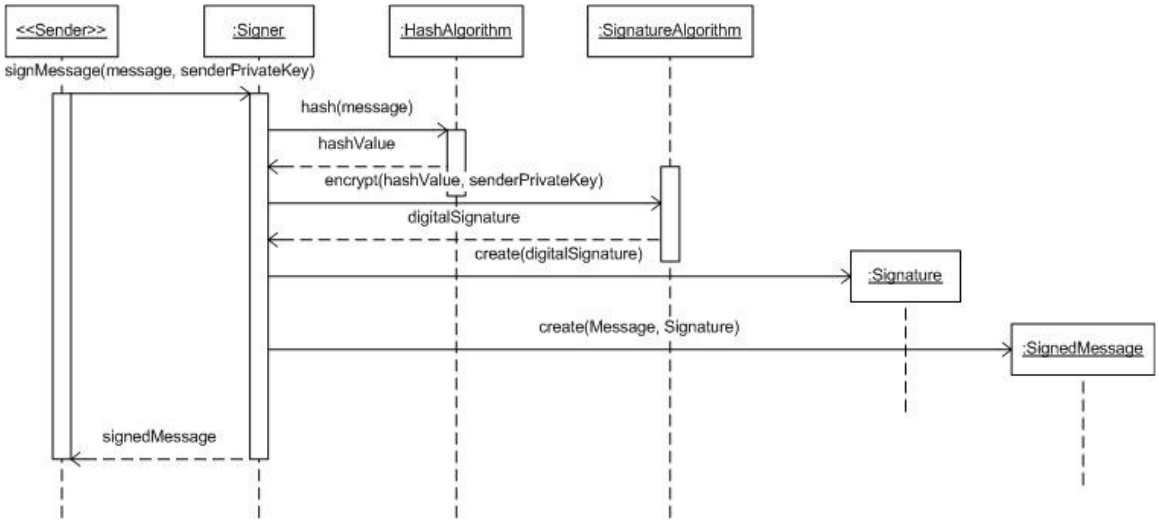

Figure 3: Sequence Diagram for signing a message

Verify a signed message (Figure 4)

Summary: A receiver wants to verify that a message was sent by a specific system.
Actors: A receiver
Precondition: The sender and receiver have agreed upon a pair of PKE algorithms.
Description

1. A receiver verifies the public key of the sender with the certificate authority.
2. A receiver sends the signature, original message, and the sender's public key to the verifier.
3. The verifier decrypts the signature using the using the sender's public key.
4. The verifier hashes the message.
5. The verifier compares the message hash and the decrypted signature.
6. The verifier sends an acknowledgement to the receiver that the signature is valid.

Alternate Flow 1

- If the alternate flow for signing a message is followed, instead of hashing the received message in step 4, previously exchanged information is hashed instead.

Alternate Flow 2

- If the hash in step 4 and the decrypted signature from step 3 do not match, the verifier sends an acknowledgement to the receiver that signature verification failed.

Postcondition: The signature has been verified

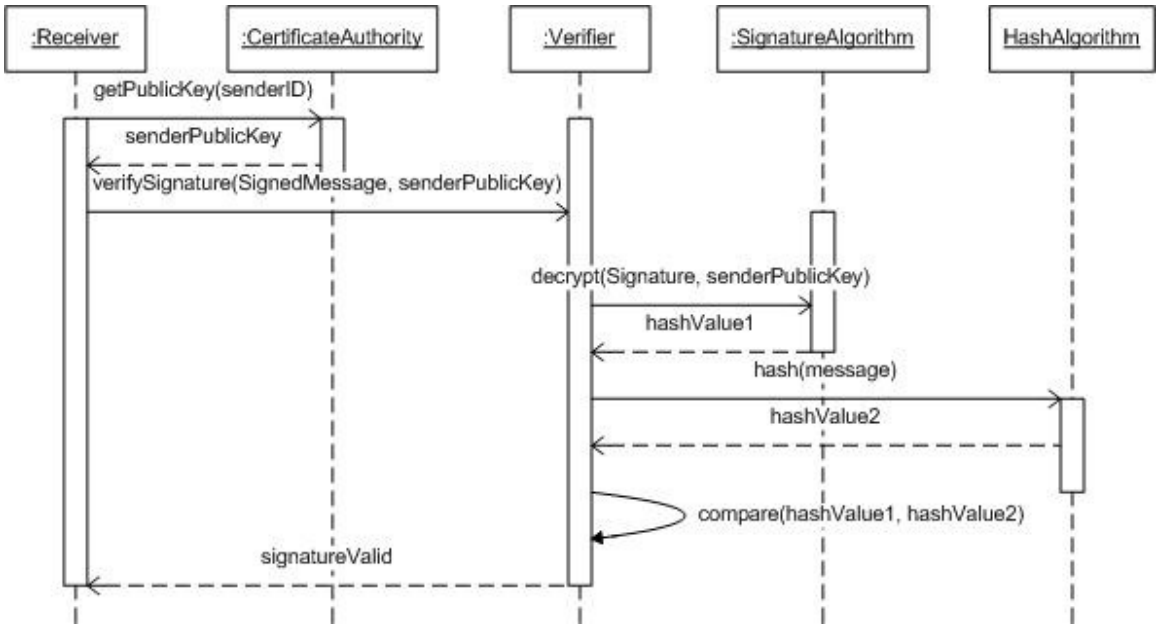

Figure 4: Sequence Diagram for verifying a signature

Authenticate and symmetrically encrypt a message (Figure 5)

Summary: A sender wants to symmetrically encrypt and authenticate a message.
Actors: A sender.
Precondition: A private shared key must exist between the sender and receiver.
Description

1. A sender sends the message and the key to the MAC algorithm.
2. The MAC algorithm produces and returns MAC.
3. A sender sends the message and the key to the symmetric algorithm.
4. The symmetric algorithm encrypts the message using the provided key.
5. The sender sends the cipherMessage concatenated with the MAC to the receiver.

Postcondition: An encrypted message is created and sent.

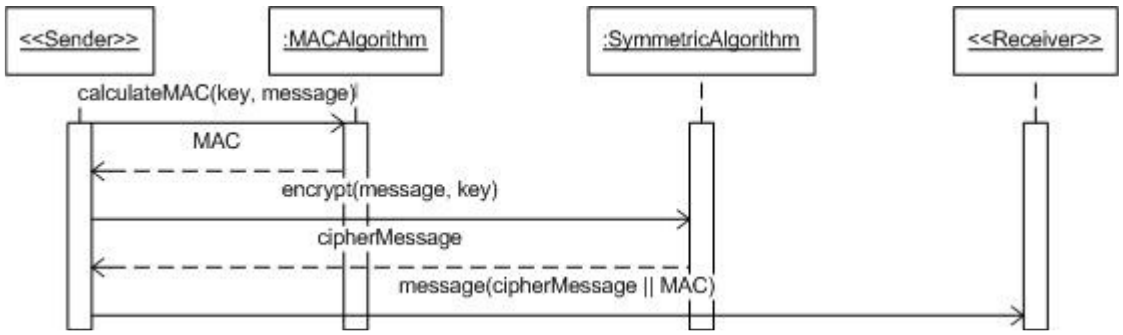

Figure 5: Sequence Diagram for authenticating, symmetrically encrypting, and sending a message.

Symmetrically decrypt and authenticate a message (Figure 6)

Summary: A receiver wants to decrypt a cipherMessage and authenticate the original message.
Actors: A receiver
Preconditions: A shared private key must exist between a sender and receiver
Description

1. A receiver sends the cipherMessage and its key to the symmetric algorithm.

2. The symmetric algorithm decrypts the cipherMessage into the original message.
3. The receiver sends the message and the key to the MAC algorithm.
4. The MAC algorithm produces and returns a MAC.
5. The receiver compares the newly calculated MAC with the received MAC to determine if the message was altered.

Postcondition: The receiver recovers and authenticates the original message.

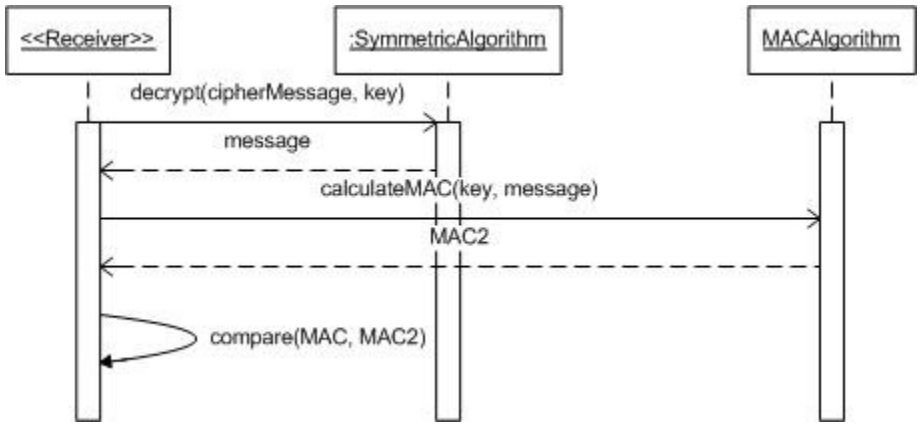

Figure 6: Sequence Diagram for symmetrically decrypting and authenticating a message.

2.6 Implementation

- All SSH implementations must support a basic set of authentication, encryption, and key exchange algorithms, but other algorithms may be used in place of the required algorithms.
- When choosing an IKE [Kau05] algorithm, be sure to choose an algorithm that produces a reasonable length key to avoid brute force attacks. In the required Diffie-Hellman [Ylo06b] [Kiv03] [Sta06] algorithm, the strength of the key is equivalent to a 70-80 bit symmetric key.
- When selecting a user authentication algorithm, be sure to select a strong security algorithm. The only required method is DSS [FIPS09] but SSH supports other methods including RSA [RSA02].
- For IKE and authentication algorithms, be sure to choose a hashing algorithm that is *collision resistant*. That is, it is highly unlikely that two different sets of data will produce the same hash value [Sta06].
- For symmetric encryption algorithms, the minimum recommended length for the key should be 128 bits or more. This requirement is met by several algorithms such as RSA and RC4 [Sta06].
- Considerations for interoperability must be made for legacy versions of SSH. The current 2.0 version of SSH [Ylo06b] does not support older versions of SSH by default.
- Both a public key and certificates for a sender should be verifiable through a certificate authority.

2.7 Known uses

- Since SSH is a protocol, SSH is used in tandem with other protocols and applications to create secure versions of those applications. A very common version of the SSH protocol is OpenSSH [OSSH10].
- Replaces or enhances the following remote applications in Linux:
  - Used as a replacement for the remote host shell logins.
  - Used as a replacement for the remote shell commands.
  - Used to forward X window connections for X applications.
- Some of the applications that utilize SSH:
  - SSH File Transfer Protocol (SFTP) [Gal08].
  - Used for tunneling or forwarding a port, and is commonly used for creating fully secured Virtual Private Networks (VPN) [VPN08]
  - PuTTY SSH/Telnet client for remote logins [Tat10]

2.8 Consequences

This pattern produces the following advantages.

- *Confidentiality* – Because SSH employs symmetric encryption, messages exchanged between two systems are secure from eavesdroppers as long as the encryption method for the system is not known and/or the key length is a sufficient size.
- *System Authentication* – Because a system is authenticated using either a PKI or SKE, we can prove that a given message came from a specific system.
- *Message Authentication* – Because a message is either signed using PKI or encrypted using SKE along with a MAC, we can be sure of the source, and correctness of any received message.
- *Integrity* – If a message is modified in transit, the locally calculated MAC of that message will be very different from the received MAC, which will invalidate the sent message. This is highly dependent on using a collision resistant hashing algorithm.
- *Non-Repudiation* – When a message that is signed by a system is sent, the sender cannot deny sending the message because only the sender should possess the private key to encrypt such a message.
- *Flexibility* – Because of the fact that SSH is a protocol, it can be used as an underlying security measure in a number of applications.
- *Transparency* – SSH provides a completely transparent method of security to the user.
- *Low Overhead* – SSH provides these security features at a relatively low overhead cost by using SKE to provide most of the security

The following disadvantages come with this pattern:

- Because messages are encrypted, there is an increased overhead in message exchange. However, this can be mitigated by choosing an encryption algorithm that uses a smaller key size or is less complex in its implementation.
- Programs must be configured on a case by case basis to use SSH. Because it is not a standard for communication, it is not normally implemented in most programs.
- SSH is typically deployed on Linux/UNIX based systems, and only has a few options for Windows users, making it somewhat OS dependent.

2.9 Example Resolved

By implementing a secure system with authentication, Bob may utilize a secure VPN client which employs SSH to securely connect with a remote system at his job (such as PuTTY). All information, including algorithm selection are handled automatically by the SSH protocol and the entire system is transparent to the user. This allows Bob to perform much of his work directly from home, saving him the commute time.

2.10 Related Patterns

Digital signatures [FIPS09] [Has09a]
Encryption and decryption with public key cryptographic schemes [RSA02] [Has09a]
Symmetric key encryption [Has09b]
Certificates [Die08]
Internet Key Exchange [Kau05]
Role pattern [Bau97]

3 Conclusions

Acknowledgements

We thank Rosanna Andrade, our shepherd, for her insightful suggestions.

References

[Bau97] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf, "The Role Object Pattern", Procs. of PLoP 1997, http://hillside.net/plop/plop97/Proceedings/riehle.pdf

[Die08] T. Dierks, Independent, E. Rescorla, RFTM Inc., "RFC – 4526 The Transport Layer Security (TLS) Protocol Version 1.2", August 2008, http://tools.ietf.org/html/rfc5246

[FIPS09] Federal Information Processing Standards, "Digital Signature Standard (DSS)", June 2009, http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf

[Gal08] J. Galbraith, VanDyke Software, O. Saarenmaa, F-Secure, "Internet draft – SSH File Transfer", August 2008, http://tools.ietf.org/html/draft-ietf-secsh-filexfer-13

[Has09a] K. Hashizume, E.B.Fernandez, and S. Huang, "Digital Signature with Hashing and XML Signature patterns", *Procs. of the 14th European Conf. on Pattern Languages of Programs, EuroPLoP 2009.*

[Has09b] K. Hashizume and E.B.Fernandez, "Symmetric Encryption and XML Encryption Patterns", *Procs. of the 16th Conf. on Pattern Languages of Programs (PLoP 2009)*

[Kau05] C. Kaufman, Microsoft, "RFC 4306 – The Internet Key Exchange (IKEv2) Protocol", December 2005, http://tools.ietf.org/html/rfc4306

[Kiv03] T. Kivinen, M. Kojo, SSH Communication Security, "RFC 3526 - More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", May 2003, http://tools.ietf.org/html/rfc3526

[Kra97] H. Krawczyk, IBM, M. Bellware, USCD, R. Canetti, "RFC – 2104 HMAC: Keyed-Hashing for Message Authentication", February 1997, http://tools.ietf.org/html/rfc2104

[OSSH10] "OpenSSH", accessed March 28, 2010, http://www.openssh.com/

[RSA02] RSA Laboratories, "PKCS#1 v2.1: RSA Cryptography Standard", June 14, 2002, ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf

[Sta06] W. Stallings, *Cryptography and network security* (4th Ed.), Pearson Prentice Hall, 2006.

[Tat10] S. Tatham, "PuTTY: a free telnet/ssh client", accessed March 2010, http://www.chiark.greenend.org.uk/~sgtatham/putty/

[VPN08] VPN Consortium, "VPN Technologies: Definitions and Requirements", July 2008, http://www.vpnc.org/vpn-technologies.html

[Ylo06a] T. Ylonen, SSH Communications Security Corp, C. Lonvick Ed, Cisco Systems Inc, "RFC 4252 - The Secure Shell (SSH) Authentication Protocol", January 2006, http://tools.ietf.org/html/rfc4252

[Ylo06b] T. Ylonen, SSH Communications Security Corp, C. Lonvick Ed, Cisco Systems Inc, "RFC 4253 - The Secure Shell (SSH) Transport Layer Protocol", January 2006, http://tools.ietf.org/html/rfc4253

Authorization and Disclaimer

Authors authorize LACCEI to publish the paper in the conference proceedings. Neither LACCEI nor the editors are responsible either for the content or for the implications of what is expressed in the paper.