A Security-Enhanced Client-Server File Synchronisation System

Rui Hu, Zhangbin Cheng
The University of Melbourne

1. Introduction

In project 1, we have implemented a distributed system for synchronising files over client-server network. By employing block-level file concept, hashing table technology, as well as request-reply-acknowledge (RRA) communication paradigm, our system is capable of achieving bi-directional file synchronisation effective and efficiently. However, due to the lack of data security protection, our system is highly vulnerable to attacks by both insiders and intruders, such as information eavesdropping, masquerading and tampering. Therefore, in order to ensure data integrity and confidentiality, necessary security actions must be taken on the system.

In this report, we focus on addressing the following security treats and attacks: information leakage, message tampering, eavesdropping, replaying as well as client masquerading.

Based upon these security goals, we have developed a scheme, which relies on strong cryptography, to secure network file synchronisation against many types of attacks. Our system transfers all data encrypted, and only decrypts it at a client workstation. In this way, due to the lack of sufficient information, physically stealing the media will not enable an attacker to gain access to the data or to plant false data. In addition, access control is implemented in our system, to specify whether a server allows a client operation to take place on requested resources. Consequently, the risk of unauthorised operations on file data will be greatly mitigated.

Although the security of the system is dramatically enhanced by employing forementioned security scheme, the synchronisation performance will be affected to some extent. This is mainly due the time consumed by data encryption and decryption and the increased number of communication rounds. However in the context of achieving secure file synchronisation, a small dip in performance is acceptable.

The remainder of this report is organised as follows: The strengths and weaknesses of file security and cryptography technologies that have been employed in our system are discussed in section 2. A specific description of our system security design is presented in section 3. Conclusions are given in section 4.

2. Applied Security and Cryptography Technologies

2.1. Key Management

2.1.1. Secrete Key Cryptography

Secret key cryptography involves using the same cryptographic key (*Ks)* for both encryption of plaintext and the decryption of cipher text and thus it is also referred as symmetric encryption.

ciphertext = encrypt (plaintext, Ks)
plaintext = decrypt (ciphertext, Ks)

Since the secrete key is unique and only known by client and server, data can be transferred over the network privately and uniquely. This is mainly achieved by secret key encryption, which turns sensitive data, such as private file content, into useless bits of information and only the right recipient who has the encryption key can decode the information. This private communication channel assures that if someone else tries to intercept or eavesdrop the data, it will appear to be useless. Additionally, compared to public key algorithms which will be described in the next section, secret key encryption algorithms are extremely fast and are well suited for performing cryptographic transformations on large streams of data. Considering that it is highly possible for our system to transform large blocks of data in each synchronisation operation, we regard secrete key cryptography as a desirable option to ensure the confidentiality of both client's and server's file data. Meanwhile, apart from being utilised to encrypt block data, secrete key is also applied to encrypt other protocol messages to prevent leaking information such block size, file modified time as well as synchronisation directions.

On the other hand, a number of disadvantages of this method have to be taken into consideration when designing the system. First of all, since each communicating pair of client and server need to share the secrete key, the key must be transmitted through a given communication channel, which will arise a serious concern that there may be a chance that an attacker can discover the secrete key during transmission. Hence, both of the key and communication channel must be kept in secret. Second, a possible way to compromise data encrypted with this type of cipher is to perform an exhaustive search of every possible key. Even though depending on the size of the key, this type of search is extremely time consuming using even the fastest computer, but it is still possible for attackers to steal the key. Apparently, before implementing secrete key technique, we have to ensure the safety of the key first, which brings us to the next section.

2.1.2. Public Key Cryptography

Public key cryptography refers to a cryptographic system requiring two separate keys, one of which is secret - *Kprivate* and the other of which is public - *Kpublic*. To illustrate, when client wants to send a secure message to server, it uses server's *Kpublic* to encrypt the plaintext and server then utilises its *Kprivate* to decrypt the ciphertext.

Although it is computationally easy for the intended recipient to generate the key pair and employ it for encryption and decryption, it is extremely difficult for anyone to derive the private key based only upon their knowledge of the public key. Obviously, the primary benefit of public-key cryptography is increased security and convenience. Private keys never need to transmitted or revealed to anyone. Moreover, public-key encryption has a much larger range of possible values for the key, and is therefore less susceptible to exhaustive attacks that try out every possible key. Furthermore, public-key cryptography can also be used for digital signatures, in which a message is signed with the sender's private key and then can be verified by anyone who has access to the sender's public key. This ensures that any manipulation of the message will result in changes to the encoded message digest, which fundamentally prevents messages from being tampered.

Nonetheless, three main drawbacks of this type of encryption exist, one of which is that the speed of public-key cryptography is extremely slow and not well designed for transmitting large amounts of data, since it consumes lots of server resources, such as processor, bandwidth and RAM. The second demerit is that public-key cryptography may be vulnerable to impersonation attack. Even if private keys are not available, a successful attack on a certification authority will allow an adversary to impersonate whomever he or she chooses by using a public-key certificate from the compromised authority to bind a key of the adversary's choice to the name of another user. The third disadvantage is that if

The third disadvantage is that public-key cryptography is normally regarded as an effective defence because attackers can only see encrypted packets. However, Marlinspike [2] demonstrated a number of ways for an attacker to circumvent this protection. Particularly if the attacker is on the same local network as the server being compromised, Marlinspike's techniques make it very possible to perform the man-in-the-middle attack. Two popular example techniques are rogue wireless access point and DNS cache poisoning. Meanwhile, with the increased application of public-key cryptography today, hackers are finding more and more ways to hack and bypass this kind of security protection on network, which makes public-key technique potentially become less and less secure.

Based upon above descriptions of both secrete-key and public-key cryptography, it can be concluded that public-key cryptography is preferable to be employed with secret-key cryptography together to get the best of both worlds, such as the security advantage of the former technique and the speed advantage of the latter one.

Specifically in our system, the client needs to first generate a secret key that is going to be shared by the server, then server's public key is utilised to encrypt this shared key. After sever successfully receives and decrypts the key, all the data that are transferred later will be encrypted by this shared secrete key. In this way, both the system security and performance can be optimised.

2.1.3. Ephemeral Key Management

A cryptographic key is called ephemeral if it is generated for defining how long that a particular process can be operated. Typically, this kind of concept is designed for single secrete key (as known as session key) management. From prior sections' analysis, it is evident that in our system session key is crucial to the data security and integrity, since it is applied to encrypt nearly 90% of the data transferring between client and server. Even though session key is already secured by employing public-key cryptography, a number of security loopholes still exist. Typically, if the session key never expires, system would have an infinitely long-standing vulnerability to session hijacking. Once session key is stolen, all file data would be dangerously exposed to attackers. In addition, message replaying would become possible, since attackers can intercept the encoded messages and then send them later. In this case, replaying messages can still be decoded by receivers using the unexpired session key.

The best solution to this kind of security flaw is to keep a tight expiration timeframe on the session key and regenerate them frequently. Specifically in our system, based on the session key *Ks*, a new key *Kexp* will be encoded as a concatenation of three values:

$$Kexp = \{identity, Ks, expiration\}$$

where *identity* specifies the unique id used by the client and *expiration* is the time-based validity of the *Ks*. Hence, every time when client requests to proceed file synchronisation, the session key validity will be checked first, and if it is expired, a new *Ks* needs to be regenerated by the client. In this way, the risk of message replaying will be

significantly mitigated since replayed messages can not be interpreted by the receiver if the session key has already been expired.

Even though ephemeral key increases the communication rounds between client and server and makes the system network traffic become heavier, it is still worth employing this kind of technique in the sake of data security.

2.2. Access Control

Access control is referring to a selective restriction of access to certain resources. With requests of the form *<op, principal, resource>* given to a server from a client, the server determines if the access is permitted. Before making the determination, the server must authenticate the request message and the principal's credentials, which is commonly done by verifying the password that client provides.

To be specific in our system, *op* refers as "file synchronisation", *principal* is expressed as "client" and *resource* indicates that which file that client attempts to synchronise. Obviously, access control decently improves the security degree of our system by only allowing trusted clients to perform operations on authorised files, which prevents spoofing and masquerading attacks to a large extent. But on the other hand, like other security techniques, it will increase the network traffic and thus makes the system not suitable for low-bandwidth network.

2.3. Cryptographic Hash Function

A cryptographic hash function is an algorithm that takes an arbitrary block of data and returns a fixed-size bit string, which is the hash value. Ideally, there are three properties that a hash function has. First, it is very easy to compute the hash value for any given message. Second, given a hash *h*, it is extremely difficult to find any message *m* such that *h = hash(m).* Third, it is infeasible to modify a message without changing the hash. From these properties, it is clear that cryptographic hash function is a desirable technique that can be employed in a file synchronisation system. Because all the block data could be hashed so that even though hashed block data is eavesdropped by attackers, they cannot interpret it unless they can steal the original hash table as well, which is almost impossible.

However, it is worth noting that hash function cannot prevent some common attacks, such as message tampering, because hash values of block data could be intercepted and tampered by attackers. A possible solution to this issue is to apply a particular algorithm to encrypt the hash value of each block, in this way, even if the encrypted hash value is tampered, when server attempts to decrypts it, decryption error will come up.

Specifically in our system, SHA-256 hash function is utilised because this algorithm generates a relatively large hash size which is 256 bits. Whereas among all the hash functions, SHA-256 is one of the slowest functions. For a system with high transaction rate, SHA-256 can take a significant toll on the CPU. Considering that data confidentiality and integrity are the paramount properties of a file synchronisation system, we still choose SHA-256 as one of the security protection measures.

2.4. File Encryption

3. Our Security Implementation

In general, we implement a system which is taking advantage of both symmetric cryptography and asymmetric cryptography. The symmetric algorithm using public-private key pair is considerably secure and convenient because private keys never need to be transmitted or revealed to anyone else. On the flip side, as previously stated, a drawback of this type of encryption is its slow speed, which will significant deteriorate system performance when large volume of data is involved. Therefore, we combined it with the symmetric secret key cryptography to achieve a security-speed trade off.

A detailed communication paradigm between the server and client is illustrated in Figure 1. As Figure 1 shows, the server first checks whether its public and private key files present when a file syncretisation starts, and generates a new key pair if key pair does not exist. Meanwhile, the client requests a connection to the server and sends a message *Request Public Key.* Since the server's public key is distributable, it is passed to the client without encryption. Upon obtaining the server's public key, a shared session key is generated using a symmetric algorithms *AES* at client side. This shared key is sent to the server after encrypted by the public key and will be involved in later communication. The server decrypts received shared session key using its private key which is paired with the public key distributed to the client. After that, the client passes a request JSON like *<fileSync, client, file.text>*
to inform the server of which resource it attempts to access. Importantly, the server asks for a password for the resource and therefore client prompts user to input the password. The password entered by user is also encrypted by the generated

shared key to secure the transmission. If the password verification is successful, the server sends confirmation to the client and begins file synchronisation process. Otherwise, the server rejects the request and informs the client of the access denial. As for the client, if access granted, the file synchronisation process proceeds, or if the access is denied by the server, then it will keep prompting user until a valid password is provided. Above mechanism is referred as Access Control. When everything goes well, a similar file synchronisation procedure to last project (without security implementations) continues. However it is worth noting that every message passed through the socket is encrypted by the secret key shared by both parties. For example, besides block instructions such as hash value (copy block) and actual bytes (new block), the plain protocol messages such as *New Block Request* or *Process Acknowledgement* are also encrypted in order to prevent information leakage.

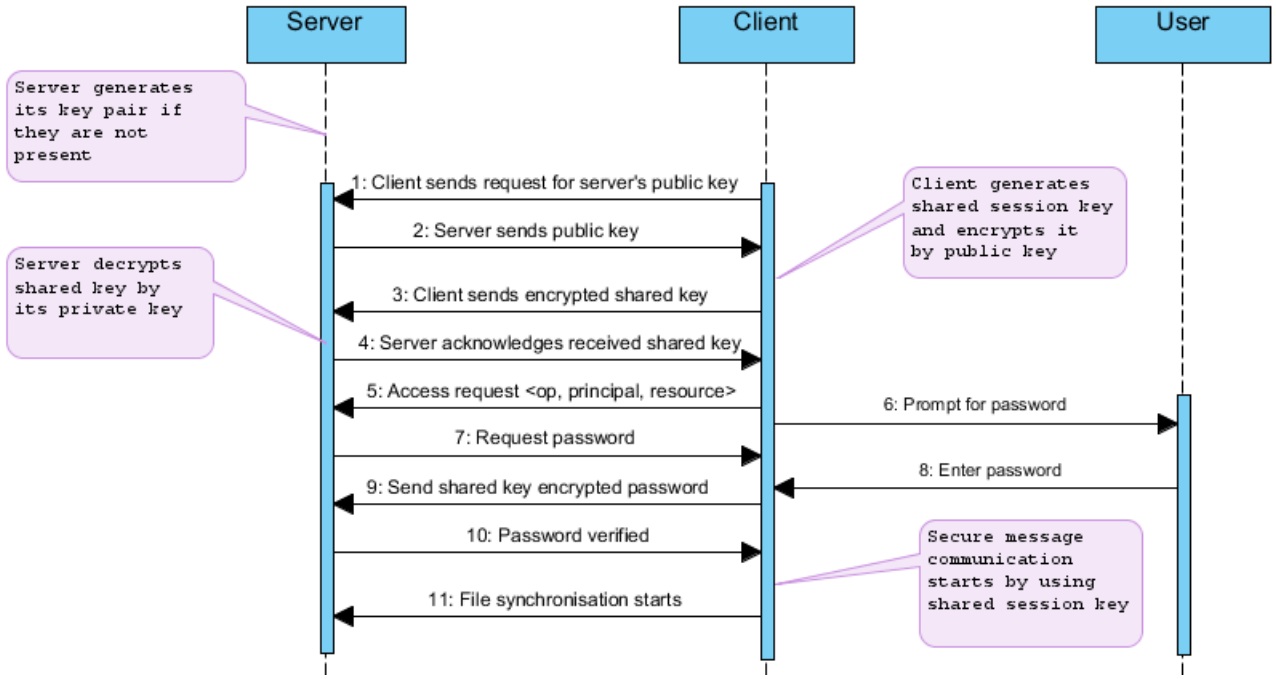

Figure 1

4. Conclusion

We have reviewed and experimented different security and cryptography technologies to seek for desired options for our file synchronisation system. A final combination of public-key cryptography and secret-key cryptography is adpoted to hit a balance between security and transmission speed. Such a multi-layer key management significantly strengthen security coefficient for specific operations. Firstly, the public/private key pair is utilized to secure shared session key transmission. Secondly, resources are well-protected using access control. Lastly, the shared key and cryptographic hash function are involved to guarantee a safe and efficient file synchronisation.

Importantly, we presented the details of all the techniques that are adopted in our system's security scheme, showing that all of the potential security attacks and treats mentioned in section 1 are successfully mitigated and prevented. However, the system is still vulnerable to some attacks that our security scheme is not capable of addressing, such as Denial of Service (DoS) attack. A typical example would be that adversaries could overload the server's resources by flooding the communication channel, such as large amounts of repeated invalid login attempts.

We believe our implementation achieves a desirable level of security although more available technologies can be embedded in the future.

References

[1] Reed, B. C., Smith, M. A., & Diklic, D. (2002, December). Security considerations when designing a distributed file system using object storage devices. In *Security in Storage Workshop, 2002. Proceedings. First International IEEE* (pp. 24-34). IEEE.

[2] Larry Seltzer. "Protecting Your Code

Updates: How to Defend Against SSL Spoofing Attacks." Retrieved 20 May, 2013, from https://www.secure128.com/documents/Code_Signing_White_Paper_Protecting_Your_Code.pdf

[3] Miller, E., Long, D., Freeman, W., & Reed, B. (2001, April). Strong security for distributed file systems. In *Performance, Computing, and Communications, 2001. IEEE International Conference on.* (pp. 34-40). IEEE.

[4] Joshua Kissoon. "Secure Socket Layer - An Overview". In *Code Project*. Retrieved 19 May, 2013, from http://www.codeproject.com/Articles/292839/Secure-Socket-Layer-An-Overview

[5] "Cryptography Overview". In *MSDN*. Retrieved 19 May, 2013, from http://msdn.microsoft.com/en-us/library/92f9ye3s(v=vs.71).aspx

[6] "How to Create Client/Server Keystores using Java Keytool". Retrieved 19 May, 2013, from http://ruchirawageesha.blogspot.com.au/2010/07/how-to-create-clientserver-keystores.html