Foundations of Information Assurance

Lab Assignment 2 Report

Team 20

Sonam Ghatode

Vishal Maurya

09.30.2019
CY5010

Index

Part 1 - Setup

- In the first step of setup, we basically fetch the files all the files from the **master server** under the directory **/home/SharedFolder/Lab_2** to our Linux VM. The command used for this purpose is:

  scp -r cy5010_master:/home/SharedFolder/Lab_2 /home/user/

  Here is the screenshot for the same:

```
user@ubuntu:~/Lab_2$ scp -r cy5010_master:/home/SharedFolder/Lab_2 /home/user/
sign2                                                 100%   70     1.7KB/s   00:00
file2.zip                                             100%   20MB   4.3MB/s   00:04
sign1                                                 100%  256     5.5KB/s   00:00
public_key.pem                                        100%  174     4.3KB/s   00:00
file1.zip                                             100%   20MB   4.8MB/s   00:04
user@ubuntu:~/Lab_2$ ls
file1.zip  file2.zip  public_key.pem  sign1  sign2
user@ubuntu:~/Lab_2$ _
```

- To verify the digital signature files fetched from the master server with their respective binary test files with the public key that was retrieved, we used the following commands:

  Command 1: *openssl dgst -verify public_key.pem -signature sign1 file1.zip*

  Command 2: *openssl dgst -verify public_key.pem -signature sign2 file2.zip*

  Following is the screenshot for the commands used:

```
user@ubuntu:~/Lab_2$ ls
file1.zip  file2.zip  pub_key  public_key.pem  sign1  sign2
user@ubuntu:~/Lab_2$ openssl dgst -verify public_key.pem -signature sign1 file1.zip
Error Verifying Data
140498018447808:error:0D0680A8:asn1 encoding routines:asn1_check_tlen:wrong tag:../crypto/asn1/tasn_
dec.c:1130:
140498018447808:error:0D07803A:asn1 encoding routines:asn1_item_embed_d2i:nested asn1 error:../crypt
o/asn1/tasn_dec.c:290:Type=ECDSA_SIG
user@ubuntu:~/Lab_2$ openssl dgst -verify public_key.pem -signature sign2 file2.zip
Verified OK
user@ubuntu:~/Lab_2$
```

  While we were trying to verify sign1 digitally signed file with the respective binary test file file1.zip, “Error Verifying Data” error was encountered. The

possible reasons for the verification failure:

- Either file1.zip was not signed with the digital signature at all.
- sign1 is not the digitally signed file of file1.zip.

However, sign2 is verified with its respective binary test file file2.zip.

Part 2 - Cryptographic Scheme:

Cryptographic Scheme in a shell:

The plaintext file is encrypted using the AES symmetric key algorithm to ensure the speed and efficiency of the encryption. The random 128 bit secret key used in AES algorithm is then encrypted using the receiver's RSA private key, to facilitate the transmission of the secret key over an insecure channel. The key size we chose as 128 bit since there is a trade-off between the size of the secret key and RSA encryption. That means, if the secret key is too large, the size of the secret key file will hamper the RSA 4096 bit encryption and we have to move to a higher bit RSA encryption version which is slower in operation. So to maintain the speed and security of the cryptographic scheme, 128 bit seem suitable to us. This maintains the confidentiality of the key as well as the plaintext. We used RSA 4096 bit version since RSA 2048 is still prone to attack, so to provide minimum security needed for the secret key to be confidential, RSA 4096 looked suitable for the encryption (as speed of encryption is also to be considered).

To ensure the integrity of the plaintext file and the secret key, hash digests of the plaintext file and the secret key is generated to be sent over the insecure channel. Since hashes are meant to be one-way function, the attacker cannot or should not be able to get the original files. Once the encryption and hashing is done, all the files are zipped and then digitally signed using SHA-512 digest and the sender's RSA private key to ensure the authenticity of the digital envelope. Fig. 2.1 shows a diagrammatic representation of the encryption scheme used for sending the message to the receiver securely over an insecure channel.

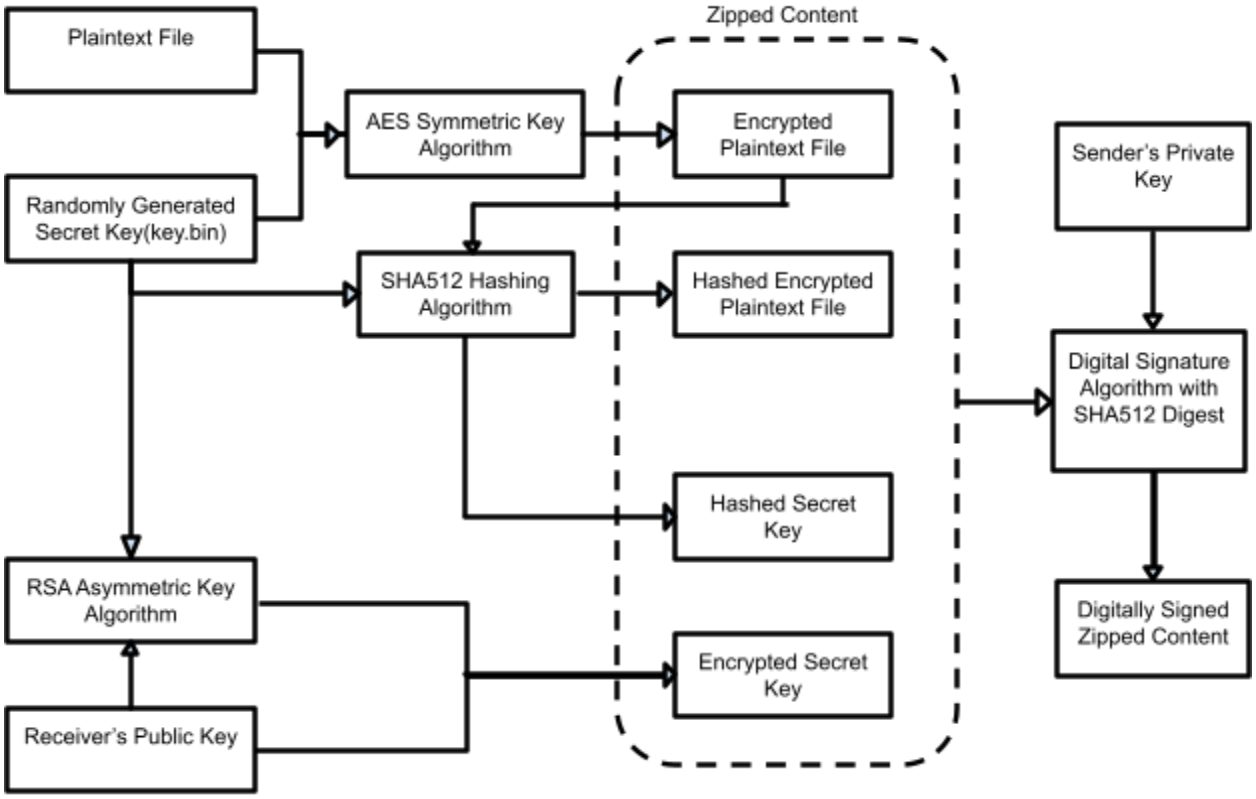

Fig. 2.1: Encryption at Sender's Side

At the receiver's end, the digital signature of the digital envelope is verified to check the authenticity of the received envelope. Once the verification is successful, the secret key is decrypted using the RSA private key of the receiver. After decryption, a SHA-512 hash of the decrypted secret key is generated and is compared with the hash of the secret key sent by the sender. If the hashes are not equal, integrity of the key is compromised over the channel and if the hashes are equal, the integrity of key is intact. Once the integrity check is done, this key is used to decrypt the encrypted plaintext file using AES algorithm. After decryption, a SHA-512 hash of the plaintext file is generated to be compared with the hash of the plaintext file sent by the sender. If the integrity is intact, the receiver can be assured of the successful transmission of the message over an insecure medium.

Fig 2.2 shows a diagrammatic representation of the decryption process carried out at the receiver's end.

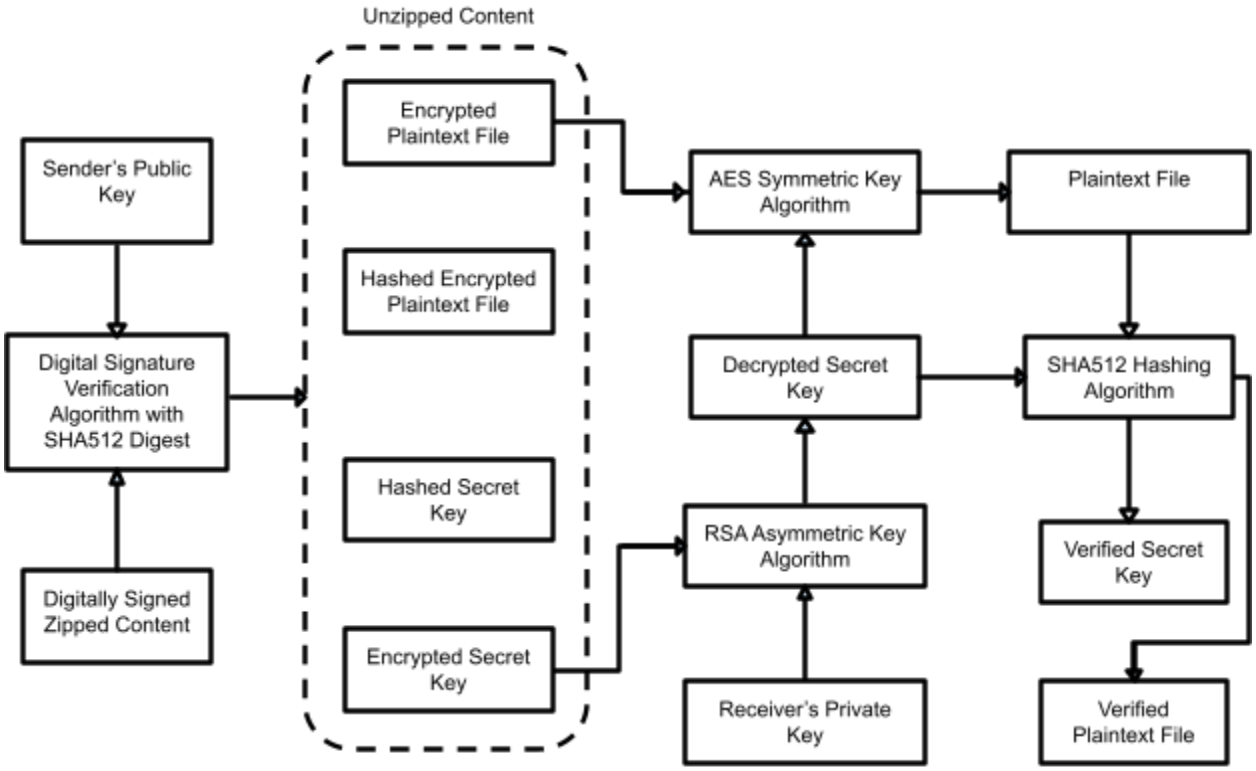

Fig. 2.2: Decryption at Receiver's Side

- **Encryption at the sender's end:**
    1. **Generation of random key:**

       We generated a random base-64 encoded key to be used as a secret key for the encryption of the plaintext file. Base-64 encoding helps facilitate the transfer of binary data over channels more efficiently. The command used to create a random base-64 encoded key:

       openssl rand -base64 128 > key.bin

       rand generates a random 128 bit key and -base64 attribute is used to encode this generated key into base-64 encoding.

    2. **Encryption of secret key key.bin:**

After the generation of key, the key needs to be encrypted in order to send it over insecure channel. The encryption of key needs to be strong enough to make it impossible for the attacker to decrypt the key and use it for the attack. For the encryption of key, we used RSA asymmetric encryption algorithm. In this case, we will be using RSA with 4096 bits key.

RSA Encryption Algorithm:

RSA (Rivest–Shamir–Adleman) is an asymmetric cryptographic algorithm based on the fact that finding the factors of a large composite number is difficult: when the integers are prime numbers, the problem is called prime factorization. RSA uses a key pair for encryption and decryption: public key and private key. Public key, as the name suggests, can be distributed to multiple people and private should be kept only by the key holder. If a person wants to send a message to a receiver(who has RSA private key), he/she will encrypt the message using the public key of the receiver and send it over an insecure channel. Since the message can only be decrypted by the receiver's private key, it can be sent using any medium. Asymmetric key encryption tend to be secure than symmetric key encryption because of the involvement of two keys, but are slower in operation.

To secure the secret key over the transmission channel, we used receiver's public key to encrypt the secret key key.bin. Commands used were:

To create RSA private key:

openssl genrsa -out filename [numbits]

genrsa is used to generate the RSA private key, from which public key can be extracted later. *out* attribute specifies the file in which key should be saved and last option *[numbits]* specifies the size of the private key to be generated in bits. This must be the last option specified.

To extract the public key from RSA private key:

At Sender's end:
openssl rsa -in rsa_private.key -out rsa_public.key -pubout -outform PEM

rsa is used to extract the public key from generated RSA private key, *in* and *out* attributes specifies input private key file and output public key file respectively. By default a private key is output, with *pubout* option a public key will be output instead.

To encrypt the secret key key.bin using public key of receiver:

openssl rsautl -encrypt -inkey id_rsa.pub.pem -pubin -in key.bin -out key.bin.enc

rsautl command can be used to sign, verify, encrypt and decrypt data using the RSA algorithm.

Options	Description
-encrypt	encrypt the input data using an RSA public key.
-inkey filename	the input key file, by default it should be an RSA private key.
-pubin	the input file is an RSA public key.
-in filename	This specifies the input filename to read data from or standard input if this option is not specified.
-out filename	specifies the output filename to write to or standard output by default.

3. **Encrypt the plaintext with secret key key.bin:**

   The encryption method we used to encrypt the plaintext file is Advanced Encryption Standard (AES) symmetric key encryption algorithm. AES works well on low RAM and with high speed.

   Advanced Encryption Standard (AES):

   AES is based on a design principle known as a substitution–permutation network, and is efficient in both software and hardware. The key size used for an AES cipher specifies the number of transformation rounds that

convert the input, called the plaintext, into the final output, called the ciphertext. The number of rounds are as follows:

- 10 rounds for 128-bit keys.
- 12 rounds for 192-bit keys.
- 14 rounds for 256-bit keys.

Since plaintext file can be of any size and using public key encryption in such case is time consuming and costly, we used AES to encrypt the plaintext file using secret key key.bin. Command used for encryption:

openssl enc -aes-256-cbc -salt -in SECRET_FILE -out SECRET_FILE.enc -pass file:./key.bin

enc command is a symmetric cipher command that allows data to be encrypted or decrypted using various block and stream ciphers using keys based on passwords or explicitly provided.

Options	Description
-aes-256-cbc	the ciphername, can change with encryption algorithm to be used for encryption.
-salt	use a salt in the key derivation routines.
-pass source	the password source.
-in filename	This specifies the input filename to read data from or standard input if this option is not specified.
-out filename	specifies the output filename to write to or standard output by default.

4. **Hashing the plaintext file using SHA-512 Hashing Algorithm:**

   To allow the receiver to check the integrity of decrypted plaintext, we hashed the plaintext file using SHA512 hashing algorithm and sent this

hashed file along with other files. Since hashes are one-way functions, attacker won't be able to get the plaintext back from the hash but receiver can check the integrity of the decrypted plaintext file by comparing the hash of the decrypted plaintext file with the hashed plaintext.

SHA-512 Hashing Algorithm:

SHA-512 (Secure Hash Algorithm 2) is a hash function computed with 64-bit words. The digest length of SHA-512 is 512 bits.

The command used to hash the plaintext file:

openssl dgst -sha512 plain_text_file.txt | awk {'print $2'} > hashed_plain_text.hash

dgst command is used to specify a digest function. Digest functions output the message digest of a supplied file or files in hexadecimal. *dgst* may be used with an option specifying the algorithm to be used, by default it is *sha256.*

5. **Hashing the secret key file key.bin using SHA-512 Hashing Algorithm:**

   To allow the receiver to check the integrity of decrypted secret key, we hashed the secret file using SHA512 hashing algorithm and sent this hashed file along with other files.

   The command used to hash the plaintext file:

   openssl dgst -sha512 key.bin | awk {'print $2'} > key_bin.hash

6. **Creating a zip of all encrypted things:**

   All the encrypted and hashed files are zipped to be sent to the receiver and further signing of the zip. The command used to zip the contents:

   zip -r9 ./zipped_content.zip ./encrypted_text_file.enc ./encrypted_secret_file.enc ./encrypted_secret_file.enc ./key_bin.hash

7. **Digitally sign the zip with sender's private key:**

   For authentication, the created zip is digitally signed using the sender's private key. At the receiver's end, the receiver will verify the signature using the public key of the sender.

We used SHA-512 algorithm to digitally sign the zip folder. The command used for signature:

openssl dgst -sha512 -sign filename -out ./basic_signature_zip.sign zipped_content.zip

sign option is used to digitally sign the digest using the private key in "filename".

- **Decryption at the receiver's end:**
  1. **Verify the digital signature using the sender's public key:**

     At the receiver's end, digital signature on the file is verified using the public key of the sender. This step is important to ensure the authenticity of the file received.

     The command used to verify the digital signature:

     openssl dgst -sha512 -verify filename -signature ./basic_signature_zip zipped_content.zip

     verify option in the dgst command is used to verify the signature using the public key in "filename". The output is either "Verification OK" or "Verification Failure".

  2. **Unzip the zip file after verification:**

     Once the verification is successful, the receiver unzips the zip file sent by the sender. The command used to unzip the contents:

     Unzip filename

  3. **Decrypt the secret key with the receiver's private key:**

     After unzipping, the secret key is decrypted using the receiver's RSA private key. This step is important to ensure the confidentiality of the AES symmetric key, that should only be shared with the receiver. The command used for the decryption of the key:

     openssl rsautl -decrypt -inkey id_rsa.pem -in key.bin.enc -out key.bin

     decrypt option in the rsautl command is used to decrypt the input data using an RSA private key.

4. **Compare hash of the decrypted key with key_bin.hash:**

   Once the encrypted secret key is decrypted using RSA asymmetric key algorithm, a SHA-512 hash of the decrypted key is generated to be compared with the hash sent in the zipped file. Once the hash is generated, it is compared with the hash sent by the sender, if the hash matches, the integrity of the file is maintained and if it doesn't match, the receiver needs the sender to send the files again.

   The command used to generate SHA-512 hash of the decrypted key:

   openssl dgst -sha512 key.bin | awk {'print $2'} > key_bin_verify.hash # key_bin_verify.hash == key_bin.hash

5. **Decrypt the encrypted plain text with decrypted secret key:**

   After the decryption of the secret key and verification of the hash, the encrypted plaintext is decrypted using AES algorithm and the secret key. The command used to decrypt the encrypted plaintext:

   openssl enc -d -aes-256-cbc -in filename -out filename -pass file:./key.bin

   d option in enc command is used to decrypt the input file.

6. **Compare hash of the decrypted plain text with original plain text:**

   After the decryption of the encrypted plaintext, a SHA-512 hash of the plaintext is generated. This hash is then compared with the hash sent by the sender. This step is necessary to ensure the integrity of the message sent in the plaintext file.

   The command used to generate the SHA-512 hash of the decrypted plaintext file:

   openssl dgst -sha512 SECRET_FILE | awk {'print $2'} > SECRET_FILE_verify.hash # SECRET_FILE_verify.hash == hashed_plain_text.hash

References :

Advanced Encryption Standard, Wikipedia :
https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

Openssl rsautl man page: https://www.openssl.org/docs/man1.0.2/man1/rsautl.html

Openssl rsa man page: https://www.openssl.org/docs/man1.0.2/man1/openssl-rsa.html

Openssl enc man page: https://www.openssl.org/docs/man1.0.2/man1/enc.html

Openssl dgst man page:https://www.openssl.org/docs/manmaster/man1/openssl-dgst.html