MENDOZA, John Robert T.
2014-90568
CS253 Computer Security
Final Project

I. Background/Introduction

The U.K. National Health Service (NHS) fell victim to a cyber-attack which froze its computers across the entire institution and threatened to lose millions of patient medical records stored in its server facilities [1]. NHS is the main institution of the UK government that provides healthcare and social care services to the country's diverse population. The security breach led to disruption of the NHS IT infrastructure which debilitated the delivery of healthcare services causing critical operations and appointments to doctors to be cancelled and patients diverted to emergency and accident departments of hospitals. The institution also faces the risk of having medical records leaked and exposed to the public. This is an incident that could have been averted have government institutions like NHS started on adopting the use of Linux-based operating system (OS) as primary workstation and server software platforms for running medical applications and storing highly-sensitive medical records.

GNU/Linux is a robust OS platform that provides tools for enabling secure computing and communications out of the box. OpenSSL, an open-source software library which provides cryptographic functions for a wide variety of encryption algorithms, is bundled by default in GNU/Linux OS distributions and is readily accessible for use. Encryption algorithms can easily be implemented to secure the storage of classified files, confidential documents and highly-sensitive information. Integrity of files and information can be ensured by using hash values calculated from a broad selection of hashing functions callable from the software library that can be used as well for cryptographically signing digital content for tamper-proofing and authentication. All of this security tasks can easily be performed by executing the OpenSSL command line tool on GNU/Linux.

In this paper, we will demonstrate the utility of OpenSSL's command line tool for implementing two modes of symmetric encryption operation, calculating hash values, and digital content signing and file encryption using public key methods. The commands used for the demonstration are made available through Github [2] and were tested to run on Ubuntu Linux 16.04 64-bit desktop operating system with OpenSSL version 1.0.2j.

II. Demonstration

A. Securely storing files

Common users and system administrators alike are often confronted with non-trivial task of remembering numerous credentials for different user accounts across several computer systems and software applications. For mature organizations with well-defined IT policies, users are regulated to only create complex alphanumeric combinations for passwords. This practice, in turn, have taught users to maintain passphrase keys in databases which are usually plain text files stored in their computers. This practice reinforces the notion that people and not technology are the weakest link in computer security chain. Once an attacker is able to gain access to the plaintext file, or obtains a copy of the notebook, the entire computer and network infrastructure could be compromised.

MENDOZA, John Robert T.
2014-90568
CS253 Computer Security
Final Project

An alternative method for storing password databases is to use symmetric encryption to obfuscate the content of the password database. OpenSSL supports a wide range of algorithms for performing encryption tasks. Advanced Encryption System (AES) is the latest and most secure block cipher widely implemented in the world today. Block ciphers, such as AES, are operated along with modes that define the scheme for which encryption is applied to blocks of data. Electronic Code Book (ECB) and Cipher Block Chaining (CBC) are the two most popular modes of operation for AES which will are to be demonstrated in the following sections. There are four essential components when performing encryption using OpenSSL:

- Command
- Encryption algorithm/mode
- Encryption key/IV
- Input and output files

```
LENA="data/lena512color.tiff"
LENA_ecb="enc/lena512color-aes-ecb.tiff.enc"
AES_KEY="6701bbd42acaac40cd4a"

openssl enc -aes-128-ecb -e -in ${LENA} -out ${LENA_ecb} -K ${AES_KEY}
```

openssl the binary program that is called to implement the cryptographic functions on the shell environment. ***enc*** is the command which instructs the program to determine the type of function the program will be operating in. In this case, ***enc*** is called to enable encoding using ciphers. ***-aes-128-ecb*** defines the type of cipher, key size and mode that will be used by ***-e*** which instructs the OpenSSL to perform encryption on the target input. The path of the target file to be encrypted is specified by the option ***-in*** while the output is determined by ***-out***. The key that will be supplied to the algorithm is indicated by preceeded by the option ***-K***.

Electronic Code Book (ECB) is the simplest mode of operation used for block ciphers such as AES but it is also the weakest. Data encrypted in ECB fails to exhibit strong encryption properties. Similar blocks of data after in ECB mode will always produce the same cipher text. This is not ideal for cipher algorithms as it gives attackers leverage on performing various types of attacks and thus putting data confidentiality at risks. To address this issue, we shall implement a different mode of operation for AES.

```
LENA="data/lena512color.tiff"
LENA_cbc="enc/lena512color-aes-cbc.tiff.enc"
AES_KEY="6701bbd42acaac40cd4a"
AES_IV="c9fe4ad1af98eee54b0b"

openssl enc -aes-128-cbc -e -in ${LENA} -out ${LENA_ecb} -K ${AES_KEY} \
-iv {AES_IV}
```

The key differences in the range of options applied when operating in CBC mode is the change in the cipher mode and the introduction of the option *iv.* Initialization vector (IV) is a special sequence of hexadecimal characters used to augment the cipher key in order to increase stochasticity and randomness. It ensures that distinct patterns are produced for same blocks of data when applied with an algorithm having the same key. CBC is quite different from ECB as it uses block chaining for implementing the cipher. Unlike ECB which treats each block of data independently, CBC on the other hand introduces dependency between sequences of blocks of data which strengthens the quality of ciphertext generated. The cipher text produced in one block of data is linked to the cipher from the previous one which makes it function more like a stream cipher.

In the above example, the command attempts to encrypt a TIFF image file with AES-128 in ECB mode. The TIFF file is a standard test image for demonstrating image processing tasks [3]. In order to verify the effect of encryption to the image file, we will use a simple tool in Linux for classifying file types. ***`file`*** is a command line tool for determining file types using different types of tests. We feed the path of the input and output to file and inspect the generated results.

```
LENA="data/lena512color.tiff"
LENA_ecb="enc/lena512color-aes-ecb.tiff.enc"
LENA_cbc="enc/lena512color-aes-cbc.tiff.enc"

file ${LENA} ${LENA_ecb} ${LENA_cbc}
data/lena512color.tiff:            TIFF image data, big-endian
enc/lena512color-aes-ecb.tiff.enc: data
enc/lena512color-aes-cbc.tiff.enc: data
```

The original LENA image was identified as a TIFF image data encoded in big-endian format while the encrypted versions, applied with AES operated in CBC and EBC mode, are classified simply as a data. The application of AES encryption on the file has rendered the content to be unintelligible. Moreover, the encrypted file can no longer be read and processed by regular image processing applications thus achieving the goal of encryption; that is, to obfuscate data and conceal information.

In order to recover the data from its encrypted form, only a minor modification to the openssl options to perform decryption. The -e option is dropped and replaced with -d while the arguments for -out and -in are switched. Finally, the hash values are computed using the program *sha256sum* to show that the recovered files are identical to the original image.

```
LENA_ecb="enc/lena512color-aes-ecb.tiff.enc"
LENA_cbc="enc/lena512color-aes-cbc.tiff.enc"
LENA_ecb_d="dec/lena512color-aes-ecb.tiff"
LENA_cbc_d="dec/lena512color-aes-cbc.tiff"
AES_KEY="6701bbd42acaac40cd4a"
AES_IV="c9fe4ad1af98eee54b0b"

openssl enc -aes-128-ecb -d -in ${LENA_ecb} -out ${LENA_ecb_d} -K
${AES_KEY}
openssl enc -aes-128-cbc -d -in ${LENA_cbc} -out ${LENA_cbc_d} \
 -K ${AES_KEY} -iv {AES_IV}

sha256sum ${LENA} ${LENA_cbc_d} ${LENA_ecb_d}
c056da23302d2fb0d946e7ffa11e0d94618224193ff6e2f78ef8097bb8a3569b
data/lena512color.tiff
c056da23302d2fb0d946e7ffa11e0d94618224193ff6e2f78ef8097bb8a3569b
dec/lena512color-aes-cbc.tiff
c056da23302d2fb0d946e7ffa11e0d94618224193ff6e2f78ef8097bb8a3569b
dec/lena512color-aes-ecb.tiff
```

The hash values produced by sha256sum thus indicating that the original LENA image is a replica of the data recovered from the encrypted files.

B. Verifying Email Attachments

Electronic Mail (e-mail) is the most efficient method for communicating in the workspace. It should be used in government institutions more than ever in order to accelerate business processes and improve the speed for delivering services to its citizenry. However, there are still institutions who are not maximizing the benefits of this prevalent technology due to risks associated with maintaining and operating e-mail services. E-mail messages has been seen recently as the primary channel for disseminating malicious files such as viruses and malwares to computer networks. Computer users unwittingly open file attachments to these email messages believing that the messages originate from valid and official sources.

One way to mitigate this incident is by introducing integrity checks in the workflow. Hash values of the files can be sent along with or separately (through out-of-band channels) from the intended message or payload. The receiver of the file attachment can check whether the file have been tampered by obtaining the hash value that came from the trusted source and matching it with the calculated hash value by the recipient. The command below demonstrates the procedure for performing cryptographic hash functions using OpenSSL.

```
LENA="data/lena512color.tiff"

openssl dgst -sha1 ${LENA}
SHA1(data/lena512color.tiff)= e647d0f6736f82e498de8398eccc48cf0a7d53b9

openssl dgst -sha256 ${LENA}
SHA256(data/lena512color.tiff)=
c056da23302d2fb0d946e7ffa11e0d94618224193ff6e2f78ef8097bb8a3569b

openssl dgst -sha512 ${LENA}
SHA512(data/lena512color.tiff)=
2cb9d7df53eb8640dc48d736974f472a98d9c7186de7a972490455f5f3ed29dfc5b75c95c
cb3ed4596bc2bfc4b1e52cf4d76bcee27d334dd155bb426617392dc
```

Suppose the LENA image is the payload to an email message. The LENA image is to be transmitted to a personnel in need of the LENA image. Perhaps someone who is doing some image processing task and perhaps needs a test image to work with. Note that the payload is not confidential or classified; it is a plain and simple data that need to be shared. The sender computes for the hash value of the image and takes note of it. Before sending the email message containing the LENA image as attachment, an “out-of-band” message consisting of the last 4 digit of the hash value of the file is transmitted to the recipient using via an SMS or other means of private messaging other than email. The message proceeds to be sent over email to the intended recipient.

The recipient as soon as he receives the email message does not immediately open the attachment file. The recipient downloads the file and calculate its hash value and attempts to compare the last 4 characters to the reference characters obtained from the out-of-band message. If the values do not match, the file attachment can be suspected of having been tampered or replaced. The recipient, as precautionary measure must discard the file immediately. On the other hand, if the values are the same, the file can then be opened and used by the recipient.

In the above example, three types of cryptographic function was applied to compute for the hash values: SHA-1, SHA-256, and SHA-512. Secure Hash Algorithm (SHA) is a hash algorithm used in cryptography especially in public key encryption for generating distinct character sequences or message digests for files and plaintext data. Aside from the underlying mathematical operations used in computing the the hash values, the three types evidently differ in the length of the hash value. SHA-1 is the most popular and oldest hash algorithm but also the weakest. It produces a message digest consisting of 160 bits or 40 characters in length. SHA-256 which is set to replace SHA-1 as the prevailing standard, increases the length to 256 bits or 64 characters while SHA-512 doubles this number to 512 bits or 128 characters. It is assumed the longer the message digest, the more secure and harder for the hash function to be shattered or broken. Perhaps due to the fact that permutations of longer message digests are much larger thus increases the security level.

Using OpenSSL to compute the hash values, the *dgst* command is called along with the digest algorithm. There is no separate option to indicate the input file. Multiple files can be fed to the command to generate hash values all at once.

Cryptographic hash functions, unlike encoding or transformation, is a one-way function. Unlike encryption, message digest cannot be used to recover or reproduce data.

C. Facilitating Secure Communication

Having to utilize a separate out-of-band channel to support data verification might be cumbersome and expensive for many organizations. Instead of streamlining the use of encryption tool to facilitate secure email communications, incorporating such scheme in workflows conversely add to delays and contributes to unproductivity. The scheme may not be applicable for transmitting data of high value or confidential in nature. A separate solution exists that incorporates the message and hash value in a singular payload obviating the need for a separate out-of-band communication mechanisms.

Public key cryptography, also known as asymmetric encryption, is a cryptographic technique that uses pairs of keys to perform encryption and decryption operations. In contrast with asymmetric encryption, two separate keys are utilized in this type of system which allows multiple entities to perform encryption. A public key is a widely-known key that is used for encrypting data while a private key, which is a secret key and known only to a user, is used to decrypt and recover data encrypted with the public key.

Government institutions may use this cryptographic system to ensure the integrity of the data as well as verifying the source of the data. Suppose a person in the organization wishes to send a highly classified payload over email to a recipient working within the organization. Using public key cryptosystem, the receiver generates a key pair on his workstation and shares the public key to the sender. The approach introduced in the previous section can be applied to ascertain the integrity of the file and ensure that the correct public key is utilized for the encryption process. The sender obtains the public key and proceeds in achieving the objective. The payload is encrypted using symmetric encryption methods such as the approach introduced in the first section. The passphrase use to encrypt the payload is stored in a file. The public key is then used to encrypt this secret file. The secret file along with the encrypted version of the payload is transmitted to the recipient. The recipient, who possesses the private key of the key pair, can decrypt the secret file, recover the passphrase, and successfully extract the payload. The following OpenSSL commands demonstrate the procedure to accomplish this particular task. The RSA algorithm is selected to generate the key pair with AES-128-CBC as the algorithm for

Receiving Party:

```
PRIV_RSA_KEY="data/private-rsa.key"
PUB_RSA_KEY="data/public-rsa.key"
PASSPHRASE="CS253-WedE-2017"
SECRET_ENC="enc/secret.txt.enc"
SECRET_DEC="dec/secret.txt"


# The receiving party generates 2048-bit RSA key pair
openssl genrsa -aes256 -out ${PRIV_RSA_KEY} -passout pass:${PASSPHRASE} \
2048

# Extract the public key and disseminate to other users
openssl rsa -in ${PRIV_RSA_KEY} -outform PEM -pubout -out ${PUB_RSA_KEY}
\ -passin pass:${PASSPHRASE}
```

Sending Party:

```
SECRET_FILE="data/secret.txt"
PUB_RSA_KEY="data/public-rsa.key"
SECRET_ENC="enc/secret.txt.enc"
SECRET_DEC="dec/secret.txt"
LENA_rsa="enc/lena512color-rsa.tiff.enc"
```

```
# generate secret passphrase and store in a file
openssl rand -base64 128 | tee ${SECRET_FILE}

# encrypt the LENA image with the generated secret passphrase
openssl enc -aes-128-cbc -e -in ${LENA} -out ${LENA_rsa} -pass
file:${SECRET_FILE}

# also encrypt the secret file containing the passphrase using RSA
# then sends the two files to the "receiving" party
openssl rsautl -encrypt -inkey ${PUB_RSA_KEY} -pubin -in ${SECRET_FILE}
-out ${SECRET_ENC}
```

Employing public key crypto system, such as RSA, to directly encrypt the payload, i.e. LENA image, is not feasible [5]. OpenSSL produces the following error when attempting to encrypt the file using the public key:

```
RSA operation error:  020:error:0406D06E:rsa
routines:RSA_padding_add_PKCS1_type_2:data too large for key
size:.\crypto\rsa\rsa_pk1.c:151:
```

It is not possible to encrypt large files with RSA. If the key length used to create the RSA key pair limits the size of payload allowed to be encrypted. The most effective method of utilizing asymmetric cryptosystem is to encrypt the secret key used for symmetric cryptosystem which is more manageable in size.

Receiving Party:

```
PRIV_RSA_KEY="data/private-rsa.key"
PASSPHRASE="CS253-WedE-2017"
SECRET_ENC="enc/secret.txt.enc"
SECRET_DEC="dec/secret.txt"
LENA_rsa_d="dec/lena512color-rsa.tiff"

# receiving party attempts to recover the secret passphrase from the
# encrypted secret file using his private key
openssl rsautl -decrypt -inkey ${PRIV_RSA_KEY} -in ${SECRET_ENC} -out
${SECRET_DEC} -passin pass:${PASSPHRASE}

# secret file used to decrypt the LENA image
openssl enc -aes-128-cbc -d -in ${LENA_rsa} -out ${LENA_rsa_d} -pass
file:${SECRET_DEC}
```

Asymmetric cryptosystems can further improve the security posture of any government organization by introducing digital signatures in their communication workflow. Suppose the recipient, to whom the classified payload was destined to, demands the sender to authenticate the payload to validate its true source. The sender, from whom the classified payload was generated, could sign the document to validate its authenticity. This method also aids in identifying the source of information or payload. The following commands demonstrate the procedure for digitally signing a document using OpenSSL. For this sequence, the ECDSA algorithm is employed to generate the key pair and SHA-256 to compute for the message digest.

Sending Party:

```
PUB_ECDSA_KEY="data/public-ecdsa.key"
PRIV_ECDSA_KEY="data/private-ecdsa.key"
LENA="data/lena512color.tiff"
LENA_sign="enc/lena512color-sign.der"
LENA_sign_d="dec/lena512color-sign.der"

# generate ecdsa key
openssl ecparam -genkey -name secp384r1 -noout -out ${PRIV_ECDSA_KEY}

# extract public and share to recipient
openssl ec -in ${PRIV_ECDSA_KEY} -pubout -out ${PUB_ECDSA_KEY}

# sign the hash of the file
openssl dgst -sha256 -sign ${PRIV_ECDSA_KEY} ${LENA} > ${LENA_sign}
```

The process of digital signing involves the implementation of an asymmetric cryptosystem to create the keypair to sign the message digest of a file or payload. ECDSA is a signature generation algorithm which uses elliptic curve cryptography. To generate ECDSA keys in OpenSSL, *ecparam* and *ec* commands need to be called. *ecparam* is used to create the private key and *ec* to extract the public key from it. Similar to the process of calculating message digest in the previous sections, *dgst* is passed to openssl to activate the options for hash generation. In the above example, the SHA-256 digest of the message was signed and stored in a separate DER file. A DER file is a file format for storing key and signing information.

Once the signature is generated, it can be passed along with the encrypted payload, secret file, and the public key to the receiving party. The receiving party may then use the ECDSA public key to decode the digital signature file and retrieve the digest of the payload. Consequently, it can be matched with the hash value computed from the recovered payload. The recipient is guaranteed that the message originated from the sender and was not intercepted or tampered while being moved over the network.

Receiving Party:

```
PUB_ECDSA_KEY="data/public-ecdsa.key"
LENA="data/lena512color.tiff"
LENA_sign="enc/lena512color-sign.der"

openssl dgst -sha256 -verify ${PUB_ECDSA_KEY} -signature ${LENA_sign} ${LENA}
Verified OK
```

Conclusion

MENDOZA, John Robert T.
2014-90568
CS253 Computer Security
Final Project

The OpenSSL library along with GNU/Linux operating system offers a convenient and viable platform for government institutions to achieve a secure computing and networking environment. The wide range of cryptographic functions available provide users the ability to implement different modes of encryption techniques for protecting data and mitigating the threats of cyberattacks. Creating a highly secure networking environment ensures that government institutions can maximize the full benefits of the Internet to improve the delivery of its services and engage more with their citizens.

Bibliography

[1] Griffin, A. (2017, May 15). NHS hack could be about to become far worse as people switch on computers after weekend. Retrieved May 23, 2017, from http://www.independent.co.uk/life-style/gadgets-and-tech/news/nhs-hack-cyber-attack-am-i-safe-appointments-latest-updates-weekend-security-a7736001.html
[2] J. (2017, May 23). Jrtmendoza/cs253. Retrieved May 23, 2017, from https://github.com/jrtmendoza/cs253
[3] Lenna. (2017, May 16). Retrieved May 23, 2017, from https://en.wikipedia.org/wiki/Lenna
[4] EVP Symmetric Encryption and Decryption. (n.d.). Retrieved May 23, 2017, from https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption
[5] Encrypt and decrypt files to public keys via the OpenSSL Command Line. (n.d.). Retrieved May 23, 2017, from https://raymii.org/s/tutorials/Encrypt_and_decrypt_files_to_public_keys_via_the_OpenSSL_Command_Line.html