Lab 5a: Cryptography

This lab uses the **Ubuntu 20.04** virtual machine (VM) as an OVA file **cscelab.ova** on Canvas. The credentials are as follows:

username: sec-lab

password: untccdc

1. **Section 1: Symmetric Encryption using OpenSSL**

The learning objective of this lab is for students to get familiar with the concepts in the secret key encryption. After finishing the lab, students should be able to gain a first-hand experience on encryption algorithms and their modes of operation. We will study tools and libraries for providing data confidentiality.

OpenSSL (<https://www.openssl.org/>) is toolkit for the Transport Layer Security (TLS) protocol, and also a general-purpose cryptographic library. Its latest full-featured version OpenSSL 1.1.1 is installed on the VM.

Encryption and decryption is performed using “openssl enc” and “openssl dec” commands, respectively. You may type “man openssl” to learn more.

1. In your home directory, create a text file *plaintext.txt* and write a sentence “This is my secret message” into it (make sure to close the file). To confirm, type:

cat plaintext.txt

**Q1: Attach a screenshot of the result.**

1. Let us now encrypt this file using a password. Type:

openssl enc -aes-256-ctr -pass pass:***euid*** -pbkdf2 -in plaintext.txt -out ciphertext.bin

The first option requests to use the AES-256 cipher in the counter (CTR) mode. The second option defined a password to be used for encryption, and the next option requests to use the PBKDF2 algorithm for generating a key from the password.

In this exercise, for simplicity, use your EUID as a password (for example, if your EUID is “aa0001”, then the respective option will be written as “-pass pass:aa0001”). Note that in practice, such a password should never be used as it is very weak (i.e., too short and too easy to guess). The remaining options define the filenames for input (the plaintext) and output (the ciphertext).

**Note:** If the “-pass” option was not used, then utility would request the password to be entered manually (two times – the second one for confirmation).

1. Display the contents of the ciphertext file:

hexdump -C ciphertext.bin

**Q2: Attach a screenshot of the result.**

1. For decryption:

openssl enc -aes-256-ctr -pass pass:***euid*** -pbkdf2 -d -in ciphertext.bin -out plaintext\_dec.txt

**Note:** If the “-pass” option was not used, then utility would request the password to be entered manually.

1. To confirm, type:

cat plaintext\_dec.txt

Note that the original messages have been decrypted.

**Q3: Attach a screenshot of the result.**

1. It is possible to encode the ciphertext using Base 64, in order to have it in the text format:

openssl enc -aes-256-ctr -a -pass pass:***euid*** -pbkdf2 -in plaintext.txt -out ciphertext.txt

1. Type:

cat ciphertext.txt

1. Verify that decryption works correctly:

openssl enc -aes-256-ctr -d -a -pass pass:***euid*** -pbkdf2 -in ciphertext.txt

**Note:** If the option “-out” is omitted, the standard output is used.

**Q4: Attach a screenshot of the result.**

1. Run the encryption again with the same password, but write the output into *ciphertext2.txt*:

openssl enc -aes-256-ctr -a -pass pass:***euid*** -pbkdf2 -in plaintext.txt -out ciphertext2.txt

1. Note that the decryption works correctly again:

openssl enc -aes-256-ctr -d -a -pass pass:***euid*** -pbkdf2 -in ciphertext2.txt

1. Now, note that the ciphertexts are different. To verify, type:

cmp ciphertext.txt ciphertext2.txt

cat ciphertext.txt ciphertext2.txt

**Q5: Explain why the ciphertexts in *ciphertext.txt* and *ciphertext2.txt* are different, even though the same password was used for encryption.**

**Hint:** Run decryption of both files again, no adding the “-p” option.

1. **Section 2: Public Key Encryption and Digital Signatures Using OpenSSL**

Let us focus on the RSA algorithm in this section.

1. First, let us generate an RSA private key (effectively, we will generate the public/private key pair). Type:

openssl genrsa -out ***euid***.key 3072

As usual, replace “***euid***” with your actual EUID.

This command generates the RSA private key and outputs it to the file ***euid****.key*.   
The key is stored in the PEM format. Display it:

cat ***euid***.key

1. Next, we extract and display the public key:

openssl rsa -in ***euid***.key -pubout -out ***euid***\_pk.key

cat ***euid***\_pk.key

**Q6: Attach a screenshot of the result.**

1. For encrypting the *plaintext.txt*, type:

openssl rsautl -encrypt -pubin -inkey ***euid***\_pk.key -in plaintext.txt -out rsa\_ciphertext.bin

1. For decryption, type:

openssl rsautl -decrypt -inkey ***euid***.key -in rsa\_ciphertext.bin -out rsa\_plaintext\_dec.txt

cat rsa\_plaintext\_dec.txt

**Q7: Attach a screenshot of the result.**

**Note:** The above method is suitable for encryption of short messages (up to about 1 kilobyte),   
for longer messages a hybrid encryption (KEM/DEM) should be used.

1. To digitally sign the file *plaintext.txt*, type:

openssl dgst -sign ***euid***.key -out sig.bin plaintext.txt

**Note:** As of the current version 1.1.1, OpenSSL signs messages directly when using the *rsautl* or *pkeyutl* commands. For this reason, it is simpler to deploy the *dgst* command, in order to hash and sign the message with one command.

Display the signature:

hexdump sig.bin

1. To verify the signature:

openssl dgst -verify ***euid***\_pk.key -signature sig.bin plaintext.txt

**Q8: Attach a screenshot of the result.**

1. Any changes in the message will invalidate the signature.  
   Let us replace the last letter in our message:

echo "This is my secret messagd" > plaintext.txt

cat plaintext.txt

1. Now, verification will fail:

openssl dgst -verify ***euid***\_pk.key -signature sig.bin plaintext.txt

**Q9: Attach a screenshot of the result.**

1. **Section 3: Public Key Certificates Using OpenSSL**

Let us now study handling of X.509 public key certificates using OpenSSL.

Suppose that we would like to create a certificate signing request (CSR) to the Certificate Authority for the RSA key that we generated earlier. The following command can be used (do not type it yet):

openssl req -key ***euid***.key -new -out ***euid***\_domain.csr

Then, the utility will request some additional information, which is called a Distinguished Name (DN). An important field in the DN is the Common Name (CN) —it should be the exact domain name of the host for which the certificate will be used. Below is an example of the prompt:

Country Name (2 letter code): The two-letter country code where your company/organization is legally located. Example: US

State or Province Name (full name): Example: Texas

Locality Name (e.g., city): Example: Denton

Organization Name (e.g., company): University of North Texas

Organizational Unit Name (e.g., section): Department of Computer Science and Engineering (this field is optional)

Common Name (e.g. server FQDN): Fully Qualified Domain Name; Example: www.unt.edu

Email Address: Example: webmaster@unt.edu (this field is optional)

It is possible to enter all of the above information from the command line as described below.

1. Type:

openssl req -key ***euid***.key -new -out ***euid***\_domain.csr \

-subj "/C=US/ST=Texas/L=Denton/O=UNT/OU=CSE/CN=www.***euid***.edu"

1. Let us verify the result:

openssl req -text -in ***euid***\_domain.csr -noout -verify

**Q10: Attach a screenshot of the result.**

**Note:** The CSR file “***euid***\_domain.csr” will need to be sent to CA that will check the user information. If the check is successful, CA will issue the certificate file. We will omit this step in this lab. Instead, we will obtain and verify the certificate of the Google webserver. For that, we will use the *s\_client* program (of the OpenSSL suite) which implements a generic SSL/TLS client.

1. Type:

openssl s\_client -connect google.com:443 </dev/null

**Note:** The redirection from the null device immediately closes the *s\_client* program, as in general it expects commands to establish the TLS connection.

1. In order to display the whole certificate chain, type:

openssl s\_client -connect google.com:443 -showcerts </dev/null

1. Since the output of the previous command takes several screens to be display, making a picture of the last screen may not be very informative. The “more” command will be helpful in this case. Type:

openssl s\_client -connect google.com:443 -showcerts </dev/null | more

**Note:** Scrolling is done by pressing “Space” to advance the whole screen down, or “Enter” to advance one line.

**Q11: Attach two screenshots: The first and the last screen displayed as a result of the above command.**

1. **Section 4: SSH Authentication Using Public Keys**

OpenSSH can use public key cryptography for authentication. We will use a freshly generated RSA key pair. Note that if you already have a generated key, you may use the *ssh-keygen* command with “-i” option and then specify the key file name.

**Important note:** In this lab, we will use the earlier generated key pair, only to demonstrate the conversion of key formats.

1. Type:

ssh-keygen -t rsa

**Note:** You will see the following prompts—you may just press “Enter” for all of them—and the following messages will be displayed:

Enter file in which to save the key (/home/sec-lab/.ssh/id\_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /home/sec-lab/.ssh/id\_rsa

Your public key has been saved in /home/sec-lab/.ssh/id\_rsa.pub

The key fingerprint is:

SHA256:lsfwRd7HQmfPZ+rjQOGjLkyr+M67/xImKZRtVD31T9s sec-lab@vm1

(**Note:** The above value and the below randomart image will be different each time for each student. The “randomart” image is a visualization of the SHA256 hash value to make it easier to compare.)

The key's randomart image is:

+---[RSA 3072]----+

| ... .o . o|

| . oo + =.|

| + . .+ + O|

| o o = o . B+|

| . . .S + + ..E|

| . o.+. o o |

| . = o. . o |

| o =. o . |

| .oB=.+o . |

+----[SHA256]-----+

**Q12: Attach a screenshot of the result.**

1. Let us verify the result:

ls ~/.ssh/

You may expect to see two files: *id\_rsa* and *id\_rsa.pub*, which should contain the private and public keys, respectively.

1. Let us try to establish an SSH connection (for simplicity, we will connect to our own host):

ssh localhost

The SSH server will request a password. Press “Control+Z” to escape. Suppose that we want to allow trusted users to access our host without entering a password. Such a user needs to possess a private key corresponding to the public key communicated to the server in a trusted manner. Such the public key are called “authorized keys”.

1. Designate your public key as the OpenSSH authorized key as follows:

cp ~/.ssh/id\_rsa.pub ~/.ssh/authorized\_keys

1. This should allow us to establish an SSH connection (to our own host) without the use of passwords:

ssh localhost (if the prompt about adding to the known hosts appears, then accept it)

(If successful, a welcome message will be displayed.)

Close the SSH connection:

exit

**Q13: Attach a screenshot of the result.**

1. **References:**
2. OpenSSL version 1.1.1 manual: <https://www.openssl.org/docs/man1.1.1/>
3. OpenSSL Cookbook, 3ed online: <https://www.feistyduck.com/library/openssl-cookbook/online/>
4. OpenSSL Quick Reference Guide:  
   <https://www.digicert.com/kb/ssl-support/openssl-quick-reference-guide.htm>
5. M. Anicas: OpenSSL Essentials: Working with SSL Certificates, Private Keys and CSRs: <https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs>
6. Wikibook on OpenSSH/Cookbook/Public Key Authentication: <https://en.wikibooks.org/wiki/OpenSSH/Cookbook/Public_Key_Authentication>