

Symmetric and Public-key Crypto

Due April 14 2015, 11:59PM

Updated April 11: see Piazza for a list of errata.

Sections 1–4 are Copyright © 2006 - 2011 Wenliang Du, Syracuse University.
The development of this document is/was funded by three grants from the US National Science Foundation: Awards No. 0231122 and 0618680 from TUES/CCLI and Award No. 1017771 from Trustworthy Computing. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation. A copy of the license can be found at <http://www.gnu.org/licenses/fdl.html>.

1 Overview

The overall learning objective of this lab is to get familiar with the concepts with symmetric key encryption and hashing. You will be getting first-hand experience with using the OpenSSL commandline tool and library to encrypt/decrypt messages under different encryption modes, and to construct message digests with hash functions.

The tasks are set up in such a way that you not only gain experience with the *use* of these various mechanisms, but also with their *properties*. You are encouraged to think about what these tools are doing, and why you get the results you get. Some tasks ask you to write up these observations.

You will be using the same VM that you used for the previous projects. Also, you will have to read up on the OpenSSL documentation (we provide links throughout this document). Make this a habit: standards, interfaces, requirements, and recommended uses change over time.

2 Symmetric key encryption and decryption

2.1 Task 1: Encryption using different ciphers and modes

In this task, we will play with various encryption algorithms and modes. You can use the following `openssl enc` command to encrypt/decrypt a file. To see the manuals, and to see the various cipher modes that OpenSSL supports, you can type `man openssl` and `man enc`.

```
% openssl enc ciphertype -e -in plain.txt -out cipher.bin \  
-K 00112233445566778899aabbccddeeff \  
-iv 0102030405060708
```

Replace the `ciphertype` with a specific cipher type, such as `-aes-128-cbc`, `-aes-128-cfb`, `-bf-cbc`, etc. In this task, you should try at least 3 different ciphers and three different modes. We include some common options for the `openssl enc` command in the following:

<code>-in <file></code>	input file
<code>-out <file></code>	output file
<code>-e</code>	encrypt
<code>-d</code>	decrypt
<code>-K/-iv</code>	key/iv in hex is the next argument
<code>-[pP]</code>	print the iv/key (then exit if -P)

However, you will very rarely be typing in the key and initialization vector yourself: that is very prone to error, and is kind of a waste of time. Instead, we can use OpenSSL itself to help us generate random symmetric keys. Really, all we want from a symmetric key is that it be the right size and that it be random, so we generate them with OpenSSL's `rand` command:

```
% openssl rand -base64 16 > symm_key
```

This will generate a 16 byte (128 bit) random value in base 64 encoding. We can use this to encrypt as follows:

```
% openssl enc ciphertype -e -in plain.txt -out cipher.bin \  
-pass file:symm_key -salt
```

Note that the key we generated, `symm_key`, is being used instead of specifying the key by hand. Strictly speaking, we will not use this as our key, but rather as a sort of “password” that will be used to help generate the key. To see this, you can attach the `-P` option to view the key that is actually being used. In any event, decryption also uses the same `-pass file:symm_key` argument (it does not need the `-salt` argument — technically, neither does encryption, as it is the default, but this is just to reinforce that you should never use no salt).

Submission: Please submit a file `task1a.bin` using the above static key and initialization vector in AES 128-bit mode with CBC; this file should decrypt to a file consisting of your UID. Also, submit two additional files: the first should be `task1.key` which is a random key value as described above, and the second should be called `task1b.bin`. We should be able to decrypt, using 256-bit AES in CBC mode, `task1b.bin` using `task1.key` as the key file (as above) to get the plaintext file: a file consisting of your UID.

2.2 Task 2: ECB vs. CBC

The file `tux-large.bmp` contains a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Please encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes, and then do the following:

1. Let us treat the encrypted picture as a picture, and use a picture viewing software to display it. However, For a `.bmp` file, the first set bytes contains the header information about the picture, we have to set it correctly, so the encrypted file can be treated as a legitimate `.bmp` file and we can view it. We have provided `header.bin` for you that you can use to will replace the header of the encrypted picture with that of the original picture. You can use the command: `cat header.bin yourEncryptedFile.bin > tux-enc-{cbc,ecb}.bmp`. This will create a file appending the encrypted data onto the `header.bin`.
2. Display the encrypted picture using any picture viewing software.

Submission: Please submit the encrypted versions of the image `tux-enc-cbc.bmp` (for CBC) and `tux-enc-ecb.bmp` (for ECB). Along with the images please include `task2.txt` containing:

1. A description of what the encrypted images look like.
2. Why there is a difference between the two?
3. What if any useful data you can learn by just looking at either picture?

2.3 Task 3: Programming using the Crypto Library

So far, we have learned how to use the tools provided by `openssl` to encrypt and decrypt messages. In this task, we will learn how to use `openssl`'s crypto library to encrypt/decrypt messages in programs. Your task is to find an encryption key through a brute force *dictionary attack*.

OpenSSL provides an API called EVP, which is a high-level interface to cryptographic functions. Although OpenSSL also has direct interfaces for each individual encryption algorithm, the EVP library provides a common interface for various encryption algorithms. To ask EVP to use a specific algorithm, we simply need to pass our choice to the EVP interface. Docs are here:

http://www.openssl.org/docs/crypto/EVP_EncryptInit.html

You must use EVP for this task and the next.

You know that `aes-128-cbc` is used to generate the ciphertext from the plaintext, and you also know that the numbers in the IV are all zeros (not the ASCII character '0'). Another clue that you have learned is that the key used to encrypt this plaintext is an English word shorter than 16 characters; the word that can be found from a typical English dictionary. Since the word has less than 16 characters (i.e. 128 bits), space characters (hexadecimal value 0x20) are appended to the end of the word to form a key of 128 bits. Your goal is to write a program to find this key. You are provided an English word list with the starter files for this project.

The program must compile and run in the VM we have provided, with the Makefile we provided. Your program should read files `words.txt` (the word list), `task3.txt` (plaintext), and `task3.bin` (ciphertext) from the same directory, and output a single line with the key or the phrase "NO MATCH" (just the phrase, not the quotes). In either case, it should end in a new line. *Do not add any extraneous text or characters*; we will grade with `diff`.

Note 1: You must read the plaintext and ciphertext from the filenames as above, do not compile them into your program. Our testing will use different files, but the same filenames.

Note 2: We are using 128-bit encryption, meaning block size is 16 bytes. AES is a block-oriented cipher, meaning that it will only encrypt complete blocks. $21 \% 16$ leaves 5 bytes for a second block. By default, `openssl` uses PKCS padding to get 32 bytes, and then encrypts this. So on the decrypt side, you need to make sure you have room for 32 bytes, or bus errors could occur.

Note 3: In this task, you are supposed to write your own program to invoke the crypto library. No credit will be given if you simply use the `openssl` commands to do this task.

Note 4: To compile your code, you may need to include the header files in `openssl`, and link to `openssl` libraries. See the included Makefile for details.

Submission: Please submit a file called `task3.c`. Once compiled it will output the key or "NO MATCH" depending on the input it receives.

3 Task 4: Hashing

In this task, we will investigate the difference between two properties of common hash functions: one-way property versus collision-free property. *You must use EVP for this task and the previous.*

We will use the brute-force method to see how long it takes to break each of these properties. Instead of using `openssl`'s command-line tools, you are required to write your own C program to invoke the message digest functions in `openssl`'s crypto library. Documentation can be found here:

http://www.openssl.org/docs/crypto/EVP_DigestInit.html.

Since most of the hash functions are quite strong against the brute-force attack on those two properties, it will take us years to break them using the brute-force method. To make the task feasible, we reduce the length of the hash value. We can use any one-way hash function, but we only use the **least significant** k bits of the hash value, where k can vary from one invocation of your program to another.

Write a program that, given a k -bit hash value (in ASCII hex), finds a string (consisting only lower-case ASCII characters) with the same hash value. Your program will have to repeatedly (1) generate a random text, (2) hash it, (3) compare least significant bits to the input. You must also seed your random number generator differently with each invocation of your program.

Your program (name it `task4.c`) will be compiled using `gcc` and called as follows:

```
./task4 <digest name> <hash value>
```

For example, `./task4 sha256 2612c7` provides a 24-bit hash value, while `./task4 sha512 8db1` provides a 16-bit hash value. Your program must write the winning text to `task4.out`. Please ensure the output is readable and writable, i.e.:

```
open("task4.out", O_WRONLY | O_CREAT, 0644);
```

We will verify with command line tools, e.g., `openssl dgst -sha256 task4.out`.

Submission: Submit a file called `task4.c` containing your program, as described above. Also, submit a file called `task4.txt` that describes how many texts your program has to try before matching an input, and how that varies with the length of the input. Back this up with data (averages of three trials for various values of k will suffice).

4 Task 5: Public key generation

Our next few tasks will make use of public/private key pairs. First, you will be generating your own key pairs and sending us your public key. As always, remember to keep your private key private!

There are a few ways to generate key pairs; many of you are probably familiar with `ssh-keygen`, which is commonly used to generate key pairs `.ssh/id_rsa` and `.ssh/id_rsa.pub` to facilitate logging into an SSH server (Google for more info, if you are not already familiar with this and would like to not have to enter your passwords so often when logging into a server from a machine you trust). But here, we will gain some familiarity with OpenSSL's key generation scheme.

Your task is to generate an RSA key pair. This is done in two phases in OpenSSL; first you generate your private key, as follows:

```
% openssl genrsa -aes128 -out private_key.pem 1024
```

A couple things to note here about this input:

- `genrsa`: As you've come to see by now, OpenSSL's command line tool takes a command as a first argument (`dgst`, `genrsa`, and later we will see `rsautl`). There are often multiple ways to do the same thing, so it is best to get to know whichever way is most reliable and least prone to error (you will investigate two ways to sign later).

- `-out private_key.pem`: Ultimately, the goal of the above command is to generate a private key. This argument specifies the name of the file to save it to. I've given this a `.pem` file extension to denote the fact that OpenSSL defaults to the PEM file format when working with public and private keys. (The name comes from Privacy Enhanced eMail, but the only thing that stuck from that system was the file format, so nobody ever calls it anything but PEM.)
- `-aes128`: This option informs OpenSSL not to store your private key in plaintext, but rather to first encrypt it with 128-bit AES before writing it to disk. A natural question to ask is: but what the heck is the key, and won't I need to store that key, and a key to encrypt that key, and a key to encrypt that key, and... After you run the command, you'll notice that it asks you for a pass phrase. OpenSSL uses "password-based encryption," using your password as input to determine an AES key. What this means is that every time you use your private key, you must be able to provide your password so that OpenSSL can decrypt the part of your `private_key.pem` that contains the private key. Of course, as with the `dgst` command, you get many options here for encrypting (if you Google for examples, you will see that `des3` is a common option, despite 3DES's shortcomings).
- `1024`: Finally, we provide our desired key size. I've generated a 1024-bit key, but you will be generating a 2048-bit key.

Now that we have our private key, we can generate the corresponding public key:

```
% openssl rsa -in private_key.pem -out public_key.pem -pubout
```

Note that this is using the `rsa` command as opposed to the `genrsa` command we used to generate our private key in the first place. Also, since we encrypted the private key, we should expect this command to ask us for our pass phrase.

Most of the arguments are pretty straightforward. `-in` denotes the input file: our private key, while `-out` denotes the output file: our public key. The `-pubout` option makes explicit that we will be generating a public key as our output.

Submission: Submit a 2048-bit RSA public key in PEM format called `task5.pem`

5 Task 6: Public key encryption/decryption

Public key encryption and decryption in OpenSSL make use of the `rsautl` command. First, encryption:

```
% openssl rsautl -encrypt -inkey public_key.pem -pubin \  
-in plaintext -out ciphertext
```

Recall that public key encryption is performed with the public key (so that only the holder of the private key can decrypt the message). This is captured by the fact that we provided the public key to the `-inkey` option, and included the `-pubin` option to inform OpenSSL that that file should be interpreted as a public key.

Decryption works similarly:

```
% openssl rsautl -decrypt -inkey private_key.pem -in ciphertext
```

This will output the plaintext to stdout. Again, because we are using the private key that we encrypted with password based encryption, we will be asked for our pass phrase to complete the above command.

The problem with public key encryption. Recall that *public key encryption can only operate over inputs the same size as its key*, so if we use the above commands, the plaintext (and ciphertext) are both limited to the size of the key (2048 bits from Task 5). Of course, some times you will have to send files larger than the RSA key size: for this very task, in fact! The general approach to this is to:

1. Generate a random symmetric key. (As in Task 1, using the `openssl rand` command.)
2. Encrypt the large file with this symmetric key (as in the previous tasks).
3. **Encrypt the symmetric key** with the public key, and then immediately delete the symmetric key (the sender does not need it in plaintext anymore). Encryption follows the same as in Task 6.
4. Send the encrypted file and the encrypted symmetric key.

The recipient can use his or her private key to decrypt the symmetric key (using `openssl rsautl -decrypt . . .`), and then use that to decrypt the file.

Submission: We have provided you with our own public key, `public.key.pem`, and a file, `largefile.txt`, that is too large to be encrypted using our public key alone. Encrypt `largefile.txt` using the method described above, and submit two files: the encrypted symmetric key `task6_sym_key.enc` and the encrypted file itself, `task6.enc`. We should be able to decrypt `task6_sym_key.enc` with our private key and we should be able to decrypt `task6.enc` with the symmetric key you provide. Use 256-bit AES in CBC mode to encrypt the file.

6 Task 7: Public key signatures/verification

Rounding out this assignment, we will be performing signatures and signature verification.

Like with `encrypt/decrypt`, `rsautl` provides a way to sign and verify, but also like `encrypt/decrypt`, it only operates over inputs no larger than the key. This is pretty useless, so let's just skip to the version that really does what we want!

```
% openssl dgst -sha1 -sign private_key.pem -out sig.bin plaintext
```

Let's break down the arguments:

- `dgst`: This should look familiar! Instead of just hashing, however, we will be informing it to also sign with a private key (or verify with a public key).
- `-sha1`: This is the hashing algorithm we are instructing it to use before signing. This is how we will reduce the size of the input, while also making sure every bit of the input still affects the signature, with high probability.
- `-sign private_key.pem`: These are the main difference from just generating hashes; it says we wish to sign it, and to use this private key to do so. It should come as no surprise at this point that, because we are using our encrypted private key file as input, this will ask us for our pass phrase.
- `-out sig.bin`: The name of the file to which to save the signature.
- `plaintext`: The input file that we wish to sign (note that we do not need to give a `-in` option here; so much for consistency across different OpenSSL commands).

To verify the signature, we need the corresponding public key, the original input file, and we need to know what the hashing algorithm used was. So to verify the above, we would run:

```
% openssl dgst -sha1 -verify public_key.pem \  
    -signature sig.bin plaintext
```

The main differences here are that we are providing `-verify` and the public key, and instead of specifying an output, we provide the signature along with the `-signature` option.

Submission: We have provided you with five different files and signatures, all created using SHA-256 and signed with the private key corresponding to the public key we provided you: `task7/fl` is a file and `task7/sig1` is the corresponding signature, and so on files 2–5, as well. Some of these are signed correctly, while others are not. Your task is to try to verify each one, and determine which are correct and not. Submit a file named `task7.txt` which consists of exactly five lines of text: line i should be the *exact* text “Verified OK” if file i ’s signature verifies, or the *exact* text “Verification Failure” if it does not. Do not add any extraneous characters: we have a reference file and will be grading it by running `diff`.

You will also be generating your own signature. Use your public/private key pair from Task 5 to sign the `largefile.txt` that we provided using SHA-256 as the hashing algorithm. Name your signature `task7.sig` — we should be able to verify it using your `task5.pem` file.

7 Summary of Submitted Files

Make a gzipped tarball named `<firstname>.<lastname>.tgz` that includes the following files:

- `task1a.bin`
- `task1.key`
- `task1b.bin`
- `task2.txt`
- `tux-enc-ecb.bmp`
- `tux-enc-cbc.bmp`
- `task3.c`
- `task4.c`
- `task4.txt`
- `task5.pem`
- `task6_sym_key.enc`
- `task6.enc`
- `task7.txt`
- `task7.sig`

So Jo Smith would do:

```
tar cvfz jo.smith.tgz task1b.bin task2.txt ... task4.txt
```

and submit `jo.smith.tgz` via the submit server.

Note: Only the latest submission counts.