Symmetric and Public-key Crypto

Due April 9, 2019, 11:59:59PM

The overall learning objective of this lab is to get familiar with concepts of symmetric-key and public-key cryptography. You will use command-line tools and libraries (e.g., OpenSSL and Libsodium) to encrypt and decrypt messages under different encryption modes, and to construct message digests with hash functions. The ultimate goal of this task is to prepare you for applying these techniques in the final project.

You are encouraged to think about what these tools are doing, and why you get the results you get.

You will be using the same VM you used in Project 1. Also, you will have to read up on the OpenSSL and Libsodium documentation (we provide links throughout this document). Make this a habit: standards, interfaces, requirements, and recommended uses change over time.

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
```

(The backslashes denote the fact that the command carries over onto the next line, and are of course not necessary in general.)

Replace the `<ciphertype>` with a specific cipher type, such as `-aes-128-cbc` (for AES with 128-bit keys in CBC mode), `-aes-128-cfb` (cipher feedback mode), `-bf-cbc` (Blowfish in CBC mode), etc. You can see the list of supported ciphers on your machine by running the command `openssl enc --help` (note that the `enc` command actually does not support the `--help` option—get used to the documentation for OpenSSL being rather poor—but that it will nonetheless show you its various options and supported ciphers). Familiarize yourself with this by trying at least three different ciphers and three different modes.

We include some common options for the `openssl enc` command in the following:

```
-in <file>      input file
-out <file>     output file
-e              encrypt
-d              decrypt
-K/-iv          key/iv in hex is the next argument
-[pP]           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 (`00112233...`) and initialization vector (`010203...`) 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.

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 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 can you learn by just looking at either picture?

Task 3 Hashing

So far, we have learned how to use the tools provided by `openssl` to encrypt and decrypt messages. From now on, we will learn how to use `openssl`'s crypto library and the `sodium` crypto library in programs.

In this task, we will investigate two properties of common hash functions: the one-way property and the collision-free property. 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 programs to invoke the message digest functions in `openssl`'s crypto library or `sodium` crypto library.

You must write your own C programs to complete this task twice with different libraries: once using `openssl`'s crypto library, once using the `sodium` crypto library.

When implementing with `openssl`*, you must use EVP for this task.* Documentation for EVP can be found here:

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

To install libsodium, please refer to the official documentation:

https://libsodium.gitbook.io/doc/installation.

or execute the following command:

sudo apt install libsodium-dev

Documentation of `libsodium` can be found here:

https://libsodium.gitbook.io/doc/.

Since modern 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 **most significant** k bits of the hash value, where k can vary from one invocation of your program to another.

Each of your programs (implemented with `openssl` and `libsodium`) should do the following:

Given a k-bit hash value (in ASCII hex; for simplicity, k will be a multiple of 8), finds a string (consisting only lower-case ASCII characters) with the same hash value. Your programs will have to repeatedly (1) generate random text, (2) hash it, (3) compare the *most* significant k bits to the input. You must also seed your random number generator differently with each invocation of your programs: there should be low probability that, if run twice on the same exact input, your programs returns the same output.

Write two programs called `task3-openssl.c` (using `openssl`) and `task3-libsodium.c` (using `libsodium`). Your programs will be compiled using `gcc` and called (say compiled as executable `task3`) as follows:

```
./task3 <digest name> <hash value>
```

For example, `./task3 sha256 2612c7` provides a 24-bit hash value (because `2612c7` is 24 bits), while `./task3 sha512 8db1` provides a 16-bit hash value (because `8db1` is 16 bits). Your programs must write the winning text to `task3.out`. Please ensure the output is readable and writable, i.e.:

```
open("task3.out", O_WRONLY | O_CREAT, 0644);
```

We will verify with command line tools, e.g., `openssl dgst -sha256 task3.out`.

Note 1: In this task, you are supposed to write your own program to invoke the crypto libraries. No credit will be given if you only use the `openssl` commands to do this task.

Note 2: To compile your code, you may need to include the header files in `openssl` and/or `libsodium`, and link to `openssl` and/or `libsodium` libraries. See the `Makefiles` in starter files for details.

Submission: Submit two files called `task3-openssl.c` and `task3-libsodium.c` containing your programs, as described above. Also, submit a file called `task3.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).

Task 4 Authenticated Encryption

Authenticated Encryption is a way to provide confidentiality and integrity of messages between two communicating parties. In this task, you will implement authenticated encryption *via whatever design you choose*. However, *you are required write your own C program to implement your design twice with different libraries*: once using `openssl`'s crypto library, once using the `sodium` crypto library. Documentation of `libsodium` can be found here:

`https://libsodium.gitbook.io/doc/`.

Write two programs called `task4-openssl.c` (using `openssl`) and `task4-libsodium.c` (using `libsodium`). Implement the same two commands for each program: `write` and `read`. We describe them as follows:

1. `write`: This will take two arguments: a message file and a key file. The key will be generated using `openssl`'s command line tool as in Task 1. You will write your output (in whatever format you so choose) to a file called `cipher.bin`.
2. `read`: This command will also take two arguments: the name of a cipher file and the name of a key file (corresponding to those from the `write` command). Given `cipher.bin` and a key `key.bin`, output the decrypted message if the key is correct and the tag matches. If not, output 'INVALID'.

Below is an example run of your program (compiled as executable `task4`). Note that calling the programs with `write` creates the `cipher.bin` file.

```
% ls
key.bin    msg.txt    task4
% ./task4 write key.bin msg.txt
% ls
key.bin    msg.txt    cipher.bin    task4
% ./task4 read key.bin  cipher.bin
Example message
% ./task4 read different-key.bin cipher.bin
INVALID
% ./task4 read key.bin different-cipher.bin
INVALID
% ./task4 reaaaddd key.bin cipher.bin
ERROR
```

Submission: Submit two files called `task4-openssl.c` and `task4-libsodium.c` containing your programs, as described above. Also submit a file called `task4.txt` explaining what you did to implement authenticated encryption: what encryption method and what integrity check did you use, and what was your format for the cipher file?

Now you have experience (tasks 3 and 4) using both libsodium and openssl for writing code dealing with encryption. Also submit a file called `task4-diff.txt` that contains a short (4 sentences or less), plain-English explaination on pros/cons of `openssl` vs. `libsodium`.

Task 5 Public key

Task 5.1 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_1.pem`

Task 5.2 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 5.1). 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 5.1.
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 `task5_2_sym_key.enc` and the encrypted file itself, `task5_2.enc`. We should be able to decrypt `task5_2_sym_key.enc` with our private key and we should be able to decrypt `task5_2.enc` with the symmetric key you provide. Use 256-bit AES in CBC mode to encrypt the file.

Task 5.3 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-512 and signed with the private key corresponding to the public key we provided you: `task5/f1` is a file and `task5/sig1` is the corresponding signature, and so on for 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 `task5_3.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 5.1 to sign the `largefile.txt` that we provided using SHA-512 as the hashing algorithm. Name your signature `task5_3.sig` — we should be able to verify it using your `task5_1.pem` file.

Extra Credit: Crack keys

It is very important to choose keys that are long enough to make it unlikely that an attacker can recover them while the key is still in active use.

Task: We will make available to you a set of RSA public keys (the exponent and the modulus) that are far too short to be of practical use, ranging from 32 bits to 512 bits. Each of you has been assigned your very own set of keys to crack: yours are available at:

`https://www.cs.umd.edu/class/spring2019/cmsc414/projects/keys/<ID>.tgz`

where `<ID>` is the last four least significant hex digits of the SHA-256 hash of your university ID (for instance, if your UID were 12345, your keys would be in the file `cfc5.tgz`). You can easily compute SHA-256 on the commandline via the command `echo -n "12345" | sha256sum`. Your goal is to determine the corresponding private key to as many of these public keys as you can. As the following background describes, this is really easy... but only if you can factor the large modulus.

Some background: Recall that generating an RSA key involves choosing two *large, random* primes p and q, and uses their product as the "modulus": $N = pq$, and all of the RSA operations take place "mod N". But recall also that all of the RSA operations are modular exponentiation; encrypting a message m with public key e involves computing $m^e \mod N$, and decrypting a ciphertext m^e with secret key d involves computing $(m^e)^d \mod N = m$. In a sense, what this means is that e and d are inverses of one another "in the exponent".

In modular arithmetic, if we are working over mod N, the arithmetic in the exponents operate over mod $((p-1)(q-1))$. This is known as Euler's totient function (covered in wonderful depth in CMSC 456, Cryptography). So to sum this up, if we were to know p and q, we could easily take a public key e and compute the corresponding private key d by simply computing:

$$d = e^{-1} \mod ((p-1)(q-1))$$

Fortunately, inverting a number in modular arithmetic is easy, but we have to know the modulus. And computing $(p-1)(q-1)$ is easy, but only if we know p and q. This is what gets us to the crux of RSA's security: N is made public, but p and q are not; RSA's security relies on the assumption that it is really difficult to factor a huge (thousands of bits) number N. But wouldn't you know it, we are giving you small N's, so suddenly factoring (and therefore computing a private key given a public key) becomes possible.

Some guidance: To crack the keys, you will need to factor the modulus, which is a product of two very large primes. These numbers are so large that simply storing them in an `unsigned int` would not suffice; you will need to look into using other libraries that can represent and perform mathematical operations over arbitrarily sized integers.

We discussed brute force attacks in class, which involves trying every possible value (in this case, that would mean trying every prime: does 2 divide the modulus? does 3 divide the modulus? does 5 divide it, 7, 11, ...?). For *very* small keys, this might suffice, but not for larger key sizes (e.g., 128-bit or more). Yet RSA keys even as large as 512-bits are still considered bad, even though such a brute force attack doesn't apply. The reason is because there are other methods that can vastly speed up factorization of the product of two primes. To be able to crack the larger keys, you will need to investigate these, and possibly install some extra libraries.

Note: You need not perform this extra credit within the VM.

Submission: If you choose to do this extra credit, include a directory called `extra-credit` in your submission, which should include all the code or scripts you used to launch this attack (but please do not include third-party libraries in your submission). This directory should also two files:

- `extra-credit/keys.txt` – a text file containing the public keys you attacked and their corresponding primes and private keys (please provide them in a human readable format).
- `extra-credit/writeup.txt` – a text file describing how you went about cracking the keys, the libraries you used, how long it took to crack the keys, and what limitations you ran into.

Summary of Submitted Files

Submit the following files through the submit server (only the latest submission counts):

- `task1a.bin`
- `task1.key`
- `task1b.bin`
- `task2.txt`
- `tux-enc-ecb.bmp`
- `tux-enc-cbc.bmp`
- `task3-openssl.c`
- `task3-libsodium.c`
- `task3.txt`
- `task4-openssl.c`
- `task4-libsodium.c`
- `task4.txt`
- `task4-diff.txt`
- `task5_1.pem`
- `task5_2_sym_key.enc`
- `task5_2.enc`
- `task5_3.txt`
- `task5_3.sig`
- `extra-credit/keys.txt` (optional)
- `extra-credit/writeup.txt` (optional)

Note: If your code compiles with warnings when `-Wall` is provided to `gcc`, you will be docked 2 points.