Crypto Lab – One-Way Hash Function and MAC

1 Overview

The learning objective of this lab is for students to get familiar with one-way hash functions and Message Authentication Code (MAC). After finishing the lab, in addition to gaining a deeper undertanding of the concepts, students should be able to use tools and write programs to generate one-way hash value and MAC for a given message.

2 Lab Tasks

2.1 Task 1: Generating Message Digest and MAC

In this task, we will play with various one-way hash algorithms. You can use the following `openssl dgst` command to generate the hash value for a file. To see the manuals, you can type `man openssl` and `man dgst`.

```
  % openssl dgst dgsttype filename
```

Please replace the `dgsttype` with a specific one-way hash algorithm, such as `-md5`, `-sha1`, `-sha256`, etc. In this task, you should try at least 3 different algorithms, and describe your observations. You can find the supported one-way hash algorithms by typing "`man openssl`".

2.2 Task 2: Keyed Hash and HMAC

In this task, we would like to generate a keyed hash (i.e. MAC) for a file. We can use the `-hmac` option (this option is currently undocumented, but it is supported by `openssl`). The following example generates a keyed hash for a file using the HMAC-MD5 algorithm. The string following the `-hmac` option is the key.

```
 % openssl dgst -md5 -hmac "abcdefg" filename
```

Please generate a keyed hash using HMAC-MD5, HMAC-SHA256, and HMAC-SHA1 for any file that you choose. Please try several keys with different length. Do we have to use a key with a fixed size in HMAC? If so, what is the key size? If not, why?

2.3 Task 3: The Randomness of One-way Hash

To understand the properties of one-way hash functions, we would like to do the following exercise for MD5 and SHA256:

1. Create a text file of any length.
2. Generate the hash value H_1 for this file using a specific hash algorithm.
3. Flip one bit of the input file. You can achieve this modification using `ghex` or `Bless`.
4. Generate the hash value H_2 for the modified file.
5. Please observe whether H_1 and H_2 are similar or not. Please describe your observations in the lab report. You can write a short program to count how many bits are the same between H_1 and H_2.

2.4 Task 4: One-Way Property versus strong Collision-Resistant Property

In this task, we will investigate the difference between hash function's two properties: one-way property versus strong collision-resistant property (breaking weak collision-resistance property is similar to breaking one-way 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 our own C programs to invoke the message digest functions in `openssl`'s crypto library. A sample code can be found from `https://wiki.openssl.org/index.php/Manual:EVP_DigestInit(3)`. Please get familiar with this sample code.

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 to 24 bits. We can use any one-way hash function, but we only use the first 24 bits of the hash value in this task. Namely, we are using a modified one-way hash function. Please design an experiment to find out the following:

1. How many trials it will take you to break the one-way property using the brute-force method? You should repeat your experiment for multiple times, and report your average number of trials.
2. How many trials it will take you to break the strong collision-resistant property using the brute-force method? Similarly, you should report the average.
3. Based on your observation, which property is easier to break using the brute-force method?
4. (10 Bonus Points) Can you explain the difference in your observation mathematically?

2.5 Task 5: Pseudo Random Number Generation

Generating random numbers is a quite common task in security software. In many cases, encryption keys are not provided by users, but are instead generated inside the software. Their randomness is extremely important; otherwise, attackers can predict the encryption key, and thus defeat the purpose of encryption. Many developers know how to generate random numbers (e.g. for Monte Carlo simulation) from their prior experiences, so they use the similar methods to generate the random numbers for security purpose. Unfortunately, a sequence of random numbers may be good for Monte Carlo simulation, but they may be bad for encryption keys. Developers need to know how to generate secure random numbers, or they will make mistakes. Similar mistakes have been made in some well-known products, including Netscape and Kerberos.

In this task, students will learn a standard way to generate pseudo random numbers that are good for security purposes.

Task 5.A: Measure the Entropy of Kernel

To generate good pseudo random numbers, we need to start with something that is random; otherwise, the outcome will be quite predictable. Software (i.e. in the virtual world) is not good at creating randomness, so most systems resort to the physical world to gain the randomness. `Linux` gains the randomness from the following physical resources:

```
void add_keyboard_randomness(unsigned char scancode);
void add_mouse_randomness(__u32 mouse_data);
void add_interrupt_randomness(int irq);
void add_blkdev_randomness(int major);
```

The first two are quite straitforward to understand: the first one uses inter-keypress timing and scancode, and the second one uses mouse movement and interrupt timing . The third one gathers random numbers using the interrupt timing. Of course, not all interrupts are good sources of randomness. For example, the timer interrupt is not a good choice, because it is predictable. However, disk interrupts are a better measure. The last one measures the finishing time of block device requests.

The randomness is measured using *entropy*, which is different from the meaning of entropy in the information theory. Here, it simply means how many bits of random numbers the system currently has. You can find out how much entropy the kernel has at the current moment using the following command.

```
% cat /proc/sys/kernel/random/entropy_avail
```

Please move and click your mouses, type somethings, and run the program again. Please describe your observation in your report.

Task 5.B: Get Pseudo Random Numbers from `/dev/random`

`Linux` stores the random data collected from the physical resources into a random pool, and then uses two devices to turn the randomness into pseudo random numbers. These two devices have different behaviors. In this subtask, we study the `/dev/random` device.

You can use the following command to get 16 bytes of pseudo random numbers from `/dev/random`. We pipe the data to `hexdump` to print them out.

```
% head -c 16 /dev/random | hexdump
```

Please run the above command several times, and you will find out that at some point, the program will not print out anything, and instead, it will be waiting. Basically, every time a random number is given out by `/dev/random`, the entropy of the randomness pool will be decreased. When the entropy reaches zero, `/dev/random` will block, until it gains enough randomness. Please show us how you can get `/dev/random` to unblock and to print out random data.

Task 5.C: Get Random Numbers from `/dev/urandom`

`Linux` provides another way to access the random pool via the `/dev/urandom` device, except that this device will not block, even if the entropy of the pool runs low.

You can use the following command to get 1600 bytes of pseudo random numbers from `/dev/urandom`. You should run it several times, and report whether it will block or not.

```
% head -c 1600 /dev/urandom | hexdump
```

Both /dev/random and /dev/urandom use the random data from the pool to generate pseudo random numbers. When the entropy is not sufficient, /dev/random will pause, while /dev/urandom will keep generating new numbers. Think of the data in the pool as the "seed", and as we know, you can use a seed to generate as many pseudo random numbers as you want. Theoretically speaking, the /dev/random device is more secure, but in practice, there is not much difference, because the "seed" is random and non-predictable. /dev/urandom does re-seed whenever new random data become available. The fact that /dev/random blocks may lead to denial of service attacks.

It is recommended that you use /dev/urandom to get random numbers. To do that in your program, you just need to read directly from this file. The following code snippet shows you how.

```
#define LEN 16 // 128 bits

unsigned char *key = (unsigned char *) malloc(sizeof(unsigned char)*LEN);
FILE* random = fopen("/dev/urandom", "r");
fread(key, sizeof(unsigned char)*LEN, 1, random);
fclose(random);
```

3 Submission

You need to submit a detailed lab report to describe what you have done and what you have observed; you also need to provide explanation to the observations that are interesting or surprising. In your report, you need to answer all the questions listed in this lab.