Exploring MACs and Hash Functions

Getting Started

Boot your Linux system or VM. If necessary, log in and then open a terminal window and cd to the labtainer/labtainer-student directory. The pre-packaged Labtainer VM will start with such a terminal open for you. Then start the lab:

```
    labtainer macs-hash
```

Note the terminal displays the paths to three files on your Linux host:

1) This lab manual
2) The lab report template
3) A spreadsheet that you will populate as part of the lab.

On most Linux systems, these are links that you can right click on and select “Open Link”. **If you chose to edit the lab report and/or spreadsheet on a different system, you are responsible for copying the completed files back to the displayed path on your Linux system before using “stoplab” to stop the lab for the last time.**

Familiarize yourself with questions in the lab report template before you start.

In this lab, you will explore cryptographic hash functions and message authentication codes using `openssl`, `shasum`, and a couple of home-grown scripts.

Note: There is an appendix with commonly used Unix commands.

Use the “ll” command to list the content of the directory.

Task 1: Practice Generating Digests

For this task you will **not** be using OpenSSL for digest creation because there are easier ways of generating digests on a Unix system. Instead, you will be using the `shasum` command, which by default supports SHA-1 (with a 160-bit output), as well as other SHA-2 outputs as an option. You can see the options by entering the following[1]:

```
shasum --help | less
```

The `shasum` command uses the following syntax:

```
shasum –a ALGORITHM FILENAME
```

replacing:

- *ALGORITHM* with one of the supported options (i.e., **1, 224, 256, 384, 512, 512224, 512256**).
- *FILENAME* with the name of some file on your system.

For example, to generate the 160-bit SHA-1 digest:

```
shasum –a 1 foo.txt
```

Do the following:

1. Create a small text file.
2. Use `shasum` to try the seven algorithms shown above on the file you created above.

Task 2: Checking Software Digests

In this task you will be learning how to verify the integrity of a downloaded file using hash functions. You will use the text-based "lynx" browser to download two files. While rudimentary, this browser is not vulnerable to most malicious attacks on browsers and may be suitable for retrieving files from web sites of questionable providence.

1. Type "lynx verydodgy.com" at the command prompt.
2. Use the arrow keys and "d" key to download and save the **`floppy57.fs`**
3. From that same web page select the **SHA256.sdx** file and download it. .

4. Return to the terminal and generate the SHA256 digest of the file you just downloaded to see if it matches the value contained in the SHA256.sdx file.

Note that item #1 of the worksheet asks a follow-up question.

[1] If you try to enter "man shasum", then you will get the man page for a Perl command instead of what you want, which is very confusing.

Task 3: Exploring the "Avalanche Effect"

The "avalanche effect" is a description for how a small change in the input file can cause a drastic effect on the output digest.

To understand the pseudo-random properties of cryptographic hash functions, do the following things:

1. Create a file named `iou.txt` with the following content: "Bob owes me 200 dollars".
2. Generate a SHA256 digest of the `iou.txt` file.
3. Open `iou.txt` with `and editor, e.g., leafpad .`
4. Change the '2' to a '3'.

   This will result in one bit being changed.

   An explanation for why a change from '2' to '3' results in one bit being changed.
   In the ASCII format, a '2' is the number 50, while a '3' is the number 51. The number 50 in binary is 00110010, while the number 51 in binary is 00110011; the only difference in these two numbers is the right-most binary digit.

5. Save your changes and exit the text editor.
6. Generate another SHA256 digest of the modified `iou.txt` file.

   Record in item #2 of the report your observations of the differences between the two digests of `iou.txt`.

7. Try a few more times to change iou.txt to see if you can get a new version of the file to match the digest of the original version.

   Record in item #3 of the report your experience with trying to get a different message to match the digest of the original data.

Note that item #4 asks a follow-up question.

Task 4: Exploring Second Pre-Image Resistance

In this task you will investigate the second pre-image resistant properties of the SHA256 cryptographic hash function (i.e., SHA2 with a 256-bit output). Because SHA256 is a relatively secure cryptographic hash function, you will not use its full 256-bit output. Instead, you will try to find a file that matches only a much smaller part of the digest.

1. First, generate a SHA256 digest for `declare.txt`.

   Record in item #5 of your report the last four hex digits of the displayed digest.

2. You will next be using a script named `collide1.sh`. The `collide1.sh` script was written to find random data that will hash to the same value as a given input file. Even better, it will let you specify how much of the digest you want to try to match (all or part).

   Execute the following command to find some random data that will hash such that the last hex digit of its digest will match the last hex digit of the digest for `declare.txt`.

   ```
   ./collide1.sh  declare.txt  1
   ```

   In item #6 of the report, in the row for "Attempt 1", enter the displayed number of attempts it took to find a match on the last digit of the digest. Repeat the above command <u>nine more times</u> to fill out the table.

   Item #7 asks a follow-up question.

3. Now execute the following command to try to match the last <u>two</u> hex digits of the digest for `declare.txt`.

   ```
   ./collide1.sh  declare.txt  2
   ```

   In item #8 of the report, in the row for "Attempt 1", enter the displayed number of attempts it took to find a match on the last two digits of the digest. Repeat the above command <u>nine more times</u> to fill out the table.

4. Now execute the following command to try to match the last <u>three</u> hex digits of the digest for `declare.txt`.

   ```
   ./collide1.sh  declare.txt  3
   ```

   In item #9 of the report, in the row for "Attempt 1", enter the displayed number of attempts it took to find a match on the last three digits of the digest. Repeat the above command <u>nine more times</u> to fill out the table.

5. Transfer the information from the three completed tables to the spreadsheet named "Collide1.xlsx". A predefined graph will show your average results compared against the theoretical results.

   Item #10 asks a follow-up question about the graph shown in the completed spreadsheet.

Task 5: Exploring Collision Resistance

You will next be using a Python script called `collide2.py`. The `collide2.py` script tries to solve a different collision problem. It generates random data and hashes it, and then saves each digest in a table. It continues doing this until it has found two random strings of data whose digests match on the last byte (i.e., the last two hex digits).

1. Execute the above following command:

   ```
   ./collide2.py
   ```

   In item #11 of the report, in the row for "Attempt 1", enter the displayed number of attempts it took to find two random messages whose digests match on the last byte. Repeat the above command nine more times to fill out the table.

   Item #12 asks a follow-up question.

Task 6 Exploring Message Authentication Codes

Within OpenSSL you will be using the hash-based message authentication code (HMAC) options. To get help on the command-line enter the following:

```
man dgst
```

You would enter the following to create a SHA1-based HMAC using OpenSSL:

```
openssl dgst -sha1 -hmac KEY FILENAME
```

replacing:

- *KEY* with any string of your choosing, **as long as it does not have spaces**. It appears that this key is first hashed in some fashion to create the actual key.
- *FILENAME* with the name of a file on your system

For example, to generate a SHA-1-based HMAC on a file named foo.txt (with a key of "mykey"), you would do the following:

```
openssl dgst -sha1 -hmac mykey foo.txt
```

There are other hash functions available other than "-sha1"; see "`man dgst`" for more information.

Do the following steps:

1. Use `openssl` as described above to create an HMAC for the file you created in Task 1.

2. Repeat the command with the same file but **use a different key** on the second try.

   Record in item #13 of the report your observations about the HMAC outputs when different keys are used on the same file.

3. Suppose you intercepted a transmission that included: 1) `declare.txt` (one of the files in your home directory) and 2) its SHA1-based HMAC. The MAC tag you intercepted is:

   ```
   HMAC-SHA1(declare.txt)= 986eb8a92e561f550a911352c8b2cf5fd0465342
   ```

   Through some other means you find out that the key is **only** a single digit, i.e., one of the following: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

   Assume that you need to figure out the key (given the information learned above).

   Try all possible keys to determine the key that was used to generate the above HMAC value on the `declare.txt` file.

   Note that item #14 asks a follow-up question.

Submission

After finishing the lab, go to the terminal on your Linux system that was used to start the lab and type:

```
  stoplab macs-hash
```

If you modified the lab report or spreadsheet on a different system, you must copy those completed files into the directory paths displayed when you started the lab, and you must do that before typing “stoplab”.   When you stop the lab, the system will display a path to the zipped lab results on your Linux system. Provide that file to your instructor, e.g., via the Sakai site.

Appendix – Some Unix Commands

`cd` Change the current directory.

```
cd destination
```

With no “destination” your current directory will be changed to your home directory. If you “destination” is “..”, then your current directory will be changed to the parent of your current directory.

`cp` Copy a file.

```
cp source destination
```

This will copy the file with the “source” name to a copy with the “destination” name. The “destination” can also include the path to another directory.

`clear` Erase all the output on the current terminal and place the shell prompt at the top of the terminal.

`less` Display a page of a text file at a time in the terminal. (Also see `more`).

```
less file
```

To see another page press the space bar. To see one more line press the Enter key. To quit at any time press ‘q’ to quit.

`ls` List the contents and/or attributes of a directory or file

```
ls  location
ls  file
```

With no “location” or “file” it will display the contents of the current working directory.

`man` Manual

```
man command
```

Displays the manual page for the given “command”. To see another page press the space bar. To see one more line press the Enter key. To quit before reaching the end of the file enter ‘q’.

`more` Display a page of a text file at a time in the terminal. (Also see `less`).

```
more file
```

To see another page press the space bar. To see one more line press the Enter key. To quit at any time press ‘q’ to quit.

`mv` Move and/or Rename a file/directory

```
mv source destination
```

The “source” file will be moved and/or renamed to the given “destination.”

`pwd` Display the present working directory

```
pwd
```