Crypto Lab – One-Way Hash Function and MAC

Saturday, April 5 2014, 8:41 PM

Task 1: Generating Message Digest and MAC

Created a shell file to generate several message digests through OpenSSL. This will create the message digest of a text file with a single character '0' (2 bytes) in it.

Shell file:

```
touch hashes.txt
touch test.txt
echo 0 > test.txt
openssl dgst -md4 test.txt > hashes.txt
openssl dgst -md5 test.txt >> hashes.txt
openssl dgst -ripemd160 test.txt >> hashes.txt
openssl dgst -sha test.txt >> hashes.txt
openssl dgst -sha1 test.txt >> hashes.txt
openssl dgst -sha224 test.txt >> hashes.txt
openssl dgst -sha256 test.txt >> hashes.txt
openssl dgst -sha384 test.txt >> hashes.txt
openssl dgst -sha512 test.txt >> hashes.txt
openssl dgst -whirlpool test.txt >> hashes.txt
```

Output file (2 byte input):

MD4(test.txt)= 035d94b6e5f8acde38e471976eea3600
MD5(test.txt)= 897316929176464ebc9ad085f31e7284
RIPEMD160(test.txt)= a8755c0bf76e3153e5573fae5257a2b5550db03c
SHA(test.txt)= 913f266bd38e826c9db4798776d84fc520460a83
SHA1(test.txt)= 09d2af8dd22201dd8d48e5dcfcaed281ff9422c7
SHA224(test.txt)= 51d17b7c777114691588f549eb084256fb6fc05c641289d486bf8367
SHA256(test.txt)= 9a271f2a916b0b6ee6cecb2426f0b3206ef074578be55d9bc94f6f3fe3ab86aa
SHA384(test.txt)= c0210b6eb9e16e86f212ff5bbe7cf2e7259eba652916602a9dbc8d3a65e199d0e902f355de2bd48d87fda1d7f31bedb7
SHA512(test.txt)=
a546d1300f49037a465ecec8bc1ebd07d57015a5ff1abfa1c94da9b30576933fb68e3898ff764d4de6e6741da822a7c93adc6e845806a266a63aa14c8bb09ebb
whirlpool(test.txt)=
f8915b1311b3264790a79e8082b71de83c2b1bd6866197e92fe320d68e962f32bcbc5895938c54155c75d2874d07c76917f969ca9c46d4a429ee54827037054c

The input file to be hashed was then changed from a 1-character file to a string of consecutive '1's totaling 1.4 MB, and the message digests were generated again.

Output file (1.4 MB output):

MD4(test.txt)= 43522dc2a14fb85f3fbb4133f848e400
MD5(test.txt)= e5b6179f6e5134b8ce7a580d73e35fae
RIPEMD160(test.txt)= a6ceee7a24b05ae1db6a79a2f5ad1f7845c0ee56
SHA(test.txt)= 34ed1cd5623a36e18c3666d3f54cd1d84de42bec
SHA1(test.txt)= 6ff8ed92d4dabc55df12cbc9031973bc28c17e95
SHA224(test.txt)= 017f558fb9b8661354dcb1b7cb6ebc0cc636f3e29ae9d63f7eaf5844
SHA256(test.txt)= 1b91c12b8907d57c46a49112798923dfa3e077e02de384c43e541a8c39ea0995
SHA384(test.txt)= e3a4394dfabd4db4220c311be6b30292c7d84ef9f19418e0ae328ce4f1732fd50406bdcd88def88ebca9fd9ede89e9d2
SHA512(test.txt)=
1ad4c4f1f8e4ac42ed5164a578ebddf1aae4cb85e1e0c5769e1d8ecfdaed311ea87135456fa1ebc73e462cfa3b5ef3f9416cf56193e5c7125cf446d51e6d5ce9
whirlpool(test.txt)=
2249e4feb138316dc819aace0de632203f41a2a385a2cc8ac24bf882bb6fb7bad4a61af0cd628b2d339ec9d17035c08d3e1cd2b325ca83107b6940ef525c6822

Task 2: Keyed Hash and HMAC

test.txt is a file containing a single '0' character (2 bytes).

hash.sh:

```
openssl dgst -md5 -hmac "0" test.txt > hashes.txt
openssl dgst -sha1 -hmac "0" test.txt >> hashes.txt
openssl dgst -sha256 -hmac "0" test.txt >> hashes.txt
```

```
openssl dgst -md5 -hmac "abcdefg" test.txt >> hashes.txt
openssl dgst -sha1 -hmac "abcdefg" test.txt >> hashes.txt
openssl dgst -sha256 -hmac "abcdefg" test.txt >> hashes.txt
openssl dgst -md5 -hmac "abcdefgABCDEFG0123456" test.txt >> hashes.txt
openssl dgst -sha1 -hmac "abcdefgABCDEFG0123456" test.txt >> hashes.txt
openssl dgst -sha256 -hmac "abcdefgABCDEFG0123456" test.txt >> hashes.txt
openssl dgst -md5 -hmac "01234567891113151719212325272931333537394143454749515355575961636567 69" test.txt >> hashes.txt
openssl dgst -sha1 -hmac "01234567891113151719212325272931333537394143454749515355575961636567 69" test.txt >> hashes.txt
openssl dgst -sha256 -hmac "01234567891113151719212325272931333537394143454749515355575961636567 69" test.txt >> hashes.txt
```

hashes.txt:

```
HMAC-MD5(test.txt)= 0a497a812fcd689f330ac92135e3b5cc
HMAC-SHA1(test.txt)= 01482a4fe1d656e38d3038c6ca9bb71f7b7b02bb
HMAC-SHA256(test.txt)= 7506c8c00373a084e71648a8b400f8dff414a1039935718f839aef2ef1d81eb4
HMAC-MD5(test.txt)= 1f0dfa0210e1f8fa201c8d52017ec741
HMAC-SHA1(test.txt)= 97847cb29ac76d6ecd9ecdffa77b72223db2073b
HMAC-SHA256(test.txt)= 92584255946bb72dfc010aa5a6ee8309450af558655d0e0ef574dd38a8e5e1e0
HMAC-MD5(test.txt)= 9b916229616ac627a94d0144d7a08146
HMAC-SHA1(test.txt)= 9b0ecb4688d2babe2ce1e1aecf6676fc8d9d5f01
HMAC-SHA256(test.txt)= 44feece9394bbc5960e208ab5bc925250062abf59e9fcd193450a463e1655491
HMAC-MD5(test.txt)= cfbdee6f44be87695e422794d4b40443
HMAC-SHA1(test.txt)= 9a94b8dbcec87bc6fecaa6e13786e341cf4bfdca
HMAC-SHA256(test.txt)= 7516498b1e11b5260d95e9e6d9a3a3e701fce38ebdf7f74198f5ca0db0cbe2f7
```

The length of the key used for generating HMAC digest does not matter, since the output digest for a specific algorithm always has a consistent length. The reason we can enter a key of any length is that the algorithm will take care of the length issues on its own. If a key is greater than a block size, only the relevant portion (i.e. 256 bits for SHA-256) will be taken, and the rest cropped. If a key length is less than the first block size, it will be padded to reach a certain length. In other words, the length does matter, but not to the user since there is an underlying mechanism to handle key length in the implementation of the HMAC generators.

Task 3: The Randomness of One-way Hash

In this task, generate a hash H1 using MD5 for a text file containing a single '0' character. We then change the content of the file to be a single '1' character and generate H2 using the same algorithm.
Binary representation of character '0': 00110000
Binary representation of character '1': 00110001
Therefore flipping a bit of the text file actively means changing the content from '0' to '1'.

```
[04/08/2014 14:05] seed@ubuntu:~/Desktop$ echo "0" > test.txt
[04/08/2014 14:05] seed@ubuntu:~/Desktop$ openssl dgst -md5 test.txt
MD5(test.txt)= 8973169291764 64ebc9ad085f31e7284
[04/08/2014 14:05] seed@ubuntu:~/Desktop$ echo "1" > test.txt
[04/08/2014 14:05] seed@ubuntu:~/Desktop$ openssl dgst -md5 test.txt
MD5(test.txt)= b026324c6904b2a9cb4b88d6d61c81d1
```

The two resulting digests are completely different.
We repeat the same experiment using SHA256:

```
[04/08/2014 14:10] seed@ubuntu:~/Desktop$ echo "0" > test.txt
[04/08/2014 14:10] seed@ubuntu:~/Desktop$ openssl dgst -sha256 test.txt
SHA256(test.txt)= 9a271f2a916b0b6ee6cecb2426f0b3206ef074578be55d9bc94f6f3fe3ab86aa
[04/08/2014 14:10] seed@ubuntu:~/Desktop$ echo "1" > test.txt
[04/08/2014 14:10] seed@ubuntu:~/Desktop$ openssl dgst -sha256 test.txt
SHA256(test.txt)= 4355a46b19d348dc2f57c046f8ef63d4538ebb936000f3c9ee954a27460dd865
```

The results are the same. We conclude that if we flip even 1 single bit from the input file, the whole message digest will differ. If we repeat the experiment for different input sizes and algorithms, the digests are always different.
On the other hand, MD5 and SHA1 are known to be broken in terms of collision detection. Therefore if we test thoroughly enough over many inputs, we might eventually find two equal digests for two different inputs.

Task 4: One-Way Property versus Collision-Free Property

To compile, install the libssl-dev package and compile with gcc using the -lcrypto argument.

To break the one-way property, we are given a hash and we keep generating hashes of different 3-character plaintexts untill one of them yields the desired hash. Each character will span from ASCII value 32 to 126, namely ' ' (space character) to '~'. The input was hardcoded in the source this case.

```
mugl@mugl-VirtualBox ~/Desktop $ gcc hash.c -lcrypto
mugl@mugl-VirtualBox ~/Desktop $ ./a.out
MD5 digest is: 649266 (first 24 bits only)
Brute-forcing plaintext ...
One-way property broken: Plaintext found after 852175 tries!
String "~H9" yields hash "649266"
mugl@mugl-VirtualBox ~/Desktop $ gcc hash.c -lcrypto
mugl@mugl-VirtualBox ~/Desktop $ ./a.out
MD5 digest is: 3eec72 (first 24 bits only)
Brute-forcing plaintext ...
One-way property broken: Plaintext found after 589858 tries!
String "aB#" yields hash "3eec72"
mugl@mugl-VirtualBox ~/Desktop $ gcc hash.c -lcrypto
mugl@mugl-VirtualBox ~/Desktop $ ./a.out
MD5 digest is: e30a3f (first 24 bits only)
Brute-forcing plaintext ...
One-way property broken: Plaintext found after 189972 tries!
String "5$c" yields hash "e30a3f"
mugl@mugl-VirtualBox ~/Desktop $
```

To break the collision-free property, we generate digests of all possible 3-character plaintexts untill two of them have the same hash. There were many collisions in this simplified experiment (first 24 bits only instead of the whole hash value was used). This hints that if we brute forced longer plaintexts into their correct and full hash value, we would, after a long time, also find many collisions.

```
mugl@mugl-VirtualBox ~/Desktop $ gcc hash3.c -lcrypto
mugl@mugl-VirtualBox ~/Desktop $ ./a.out
Testing collision-free property ...
Collision detected: "  8" and "kV7" yield the same hash value "e3a983"
Collision detected: "  I" and "c#!" yield the same hash value "64d41a"
Collision detected: "  N" and "px@" yield the same hash value "7d2d0e"
Collision detected: "  P" and "$^M" yield the same hash value "7810d5"
Collision detected: "  w" and "X|v" yield the same hash value "8bcc16"
Collision detected: " !R" and "o]m" yield the same hash value "d2d5ef"
Collision detected: " !X" and "G1P" yield the same hash value "7be3a6"
Collision detected: " !i" and "6b." yield the same hash value "71bcb6"
Collision detected: " !t" and "2EN" yield the same hash value "070bde"
Collision detected: " ," and "@{n" yield the same hash value "71cbc2"
Collision detected: " "6" and "oN>" yield the same hash value "835a50"
Collision detected: " "C" and "W+=" yield the same hash value "4e7850"
Collision detected: " "i" and "U+ " yield the same hash value "7c2bdd"
Collision detected: " "z" and "T`k" yield the same hash value "172f04"
Collision detected: " #(" and "@'m" yield the same hash value "407d47"
Collision detected: " #C" and "QX5" yield the same hash value "c3d0e9"
Collision detected: " #R" and "Be3" yield the same hash value "37133c"
Collision detected: " #e" and "<.'" yield the same hash value "0dd346"
```

The one-way property is clearly much easier to break then the collision-free property since we are aiming the brute force attack at one single string, whereas breaking the collision-free property requires testing out all the possible 3-character strings for the same hash value.

The two source codes are available at http://mu.gl/hash2.c and http://mu.gl/hash3.c, and an extended list of collisions at http://mu.gl/collisions.txt (which is still about 5% of the complete list, as I stopped it after several hours).