# Lab 06

# Secure Hash Function

## Objective

This lab will introduce students to hashing  and hash based Message Authentication code MACs.  In particular we will implementa particular variant of hashing algorithms like MD5, SHA1, SHA256, etc.  .

## 1) Useful concepts:

For a one-way hash to be used in cryptographic systems, the algorithm must provide preimage resistance, secondary resistance, and collision resistance:

■ *Preimage resistance means that an attempt to find the original message that produces a hash is computationally unrealistic or for a given h in the output space of the hash function, it is hard to find any message x with H(x) = h.*

■ *Secondary resistance means that an attempt to find a second message that produces the same hash is computationally unrealistic or for a given message x2≠x1 with H(x1) =H(x2).*

■ *Collision resistance means that finding any two messages that will produce the same hash is computationally unrealistic for the message pair or s x1≠x2 with H(x1)=H(x2).*

In examining the rules, while the secondary resistance and collision resistance may appear very similar, they are slightly different. From a (second) preimage attack we also get a collision attack. The other direction doesn't work as easily, though some collision attacks on broken hash functions seem to be extensible to be almost as useful as second preimage attacks (i.e., we find collisions where most parts of the message can be arbitrarily fixed by the attacker).

The strength of the hash function does not equal the hash length. The strength of the hash is about half the length of the hash due to the probability produced by the **Birthday Attack**. *The birthday attack exploits the mathematics behind the birthday problem in probability theory.*

Consider the scenario in which a teacher with a class of 30 students (n = 30) asks for everybody's birthday to determine whether any two students have the same birthday. The birthday attack treats our birthdays as uniformly distributed values out of 365 days. The general intuition is that it takes √N samples from a space of size N to have 50% chance of collision. Imagine selecting some value (k) at random from N. Then out of the k values you picked there are k(k − 1)/2 pairs. For any given pair there is a 1/N chance of collision. This gives k(k − 1)/2N chance of collision. Therefore, k ~√N will lead to around 50% chance of collision. The *birthday attack* relies on any match coming from within a set and not a specific match to a specific value. That intuition should guide us as we approach Message Authentication Codes (MACs). This birthday attack gives us a generic approach for finding two messages that hash to the same value in far less time than brute force. The size that matters is the output size of the hash function, too.

HMAC can use a variety of hashing algorithms, like MD5, SHA1, SHA256, etc. The HMAC function is not process intensive, so it has been widely accepted, and it is easy to implement in mobile and embedded devices while maintaining decent security. The following code example shows how to generate an HMAC MD5 digest with Python:

## Practice -Lab Activity 1: HMAC MD5 digest generation in Python:

```
import hmac  # Import the hmac module to use the HMAC algorithm
from hashlib import md5  # Import the md5 function from hashlib to use as the hashing algorithm

# Define the secret key as a byte string (required for HMAC).
# HMAC works with bytes, so we prefix the string with 'b' to convert it to bytes.
key = b'DECLARATION'

# Create a new HMAC object using the secret key and MD5 as the hashing algorithm.
# The second argument is an optional initial message (here we provide an empty byte string b'').
h = hmac.new(key, b'', md5)

# Add the message to be hashed to the HMAC object.
# 'h.update()' adds the content (also in bytes) to the HMAC instance.
# Since HMAC works with bytes, we need to prefix the string with 'b' to convert it to bytes.
h.update(b'We hold these truths to be self-evident, that all men are created equal')

# Compute the HMAC digest and print it as a hexadecimal string.
# 'h.hexdigest()' returns the digest (hash value) in hexadecimal form.
print(h.hexdigest())
```

## Practice -Lab Activity 2: MD5 Hash

This hash function accepts sequence of bytes and returns 128 bit hash value, usually used to check data integrity but has security issues. Functions associated:

- encode(): Converts the string into bytes to be acceptable by hash function.
- digest(): Returns the encoded data in byte format.

- hexdigest(): Returns the encoded data in hexadecimal format.

***Note:***
*The md5 library was a Python library that provided a simple interface for generating MD5 hashes. This library has been deprecated in favor of the hashlib library, which provides a more flexible and secure interface for generating hashes.*

The below code demonstrates the working of MD5 hash accepting bytes and output as bytes.

```
# Python 3 code to demonstrate the
# working of MD5 (byte - byte)

import hashlib  # Import hashlib to use hash functions

# Encoding the string 'GeeksforGeeks' as bytes and hashing it using the MD5 hash function
# MD5 requires a byte input, so we prefix the string with 'b' to convert it to bytes.
result = hashlib.md5(b'GeeksforGeeks')

# Printing the equivalent byte value of the MD5 hash.
# The 'digest()' function returns the hash value as bytes.
print("The byte equivalent of hash is: ", end="")
print(result.digest())
```

**Output:**

```
The byte equivalent of hash is : b'\xf1\xe0ix~\xcetS\x1d\x11%Y\x94\\hq'
```

## Practice -Lab Activity 3:

**Explanation:** The above code takes byte and can be accepted by the hash function. The md5 hash function encodes it and then using digest (), byte equivalent encoded string is printed. Below code demonstrated how to take string as input and output hexadecimal equivalent of the encoded value.

**Solution**

```
# Python 3 code to demonstrate the
# working of MD5 (string - hexadecimal)

import hashlib  # Import hashlib to use hash functions

# Initializing string to hash
str2hash = "GeeksforGeeks"

# Encoding the string using encode() to convert it to bytes
# Then sending it to the md5() function to compute the MD5 hash
result = hashlib.md5(str2hash.encode())
# Printing the equivalent hexadecimal value of the MD5 hash
# The 'hexdigest()' function returns the hash value in hexadecimal format
```

```
print("The hexadecimal equivalent of hash is: ", end="")
print(result.hexdigest())
```

**Output:**

```
The hexadecimal equivalent of hash is: f1e069787ece74531d112559945c6871
```

## Practice -Lab Activity 4: SHA, (Secure Hash Algorithms)

SHA, (Secure Hash Algorithms) are set of cryptographic hash functions defined by the language to be used for various applications such as password security etc. Some variants of it are supported by Python in the "**hashlib**" library. These can be found using "algorithms guaranteed" function of hashlib.

```
# Python 3 code to check
# available algorithms
import hashlib
# prints all available algorithms
print ("The available algorithms are : ", end ="")
print (hashlib.algorithms_guaranteed)
```

**Output:**

```
The available algorithms are: {'sha256', 'sha384', 'sha224', 'sha512', 'sha1', 'md5'}
```

To proceed with, lets first discuss the functions going to be used in this article.

**Functions associated:**

- **encode() :** Converts the string into bytes to be acceptable by hash function.
- **hexdigest() :** Returns the encoded data in hexadecimal format.

**SHA Hash**

The different SHA hash functions are explained below.

- **SHA256:** This hash function belong to hash class SHA-2, the internal block size of it is 32 bits.
- **SHA384:** This hash function belong to hash class SHA-2, the internal block size of it is 32 bits. This is one of the truncated version.
- **SHA224:** This hash function belong to hash class SHA-2, the internal block size of it is 32 bits. This is one of the truncated version.
- **SHA512:** This hash function belong to hash class SHA-2, the internal block size of it is 64 bits.
- **SHA1:** The 160 bit hash function that resembles MD5 hash in working and was discontinued to be

used seeing its security vulnerabilities.

Below code implements these hash functions.

```
# Python 3 code to demonstrate
# SHA hash algorithms.

import hashlib

# initializing string
str = "GeeksforGeeks"

# encoding GeeksforGeeks using encode()
# then sending to SHA256()
result = hashlib.sha256(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA256 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "GeeksforGeeks"

# encoding GeeksforGeeks using encode()
# then sending to SHA384()
result = hashlib.sha384(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA384 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "GeeksforGeeks"

# encoding GeeksforGeeks using encode()
# then sending to SHA224()
result = hashlib.sha224(str.encode())

# printing the equivalent hexadecimal value.
```

```
print("The hexadecimal equivalent of SHA224 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "GeeksforGeeks"

# encoding GeeksforGeeks using encode()
# then sending to SHA512()
result = hashlib.sha512(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA512 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "GeeksforGeeks"

# encoding GeeksforGeeks using encode()
# then sending to SHA1()
result = hashlib.sha1(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA1 is : ")
print(result.hexdigest())
```

**Output:**

```
The hexadecimal equivalent of SHA256 is :
f6071725e7ddeb434fb6b32b8ec4a2b14dd7db0d785347b2fb48f9975126178f

The hexadecimal equivalent of SHA384 is :
d1e67b8819b009ec7929933b6fc1928dd64b5df31bcde6381b9d3f90488d253240490460c0a5a1a873da8236c12ef9b3

The hexadecimal equivalent of SHA224 is :
173994f309f727ca939bb185086cd7b36e66141c9e52ba0bdcfd145d

The hexadecimal equivalent of SHA512 is :
0d8fb9370a5bf7b892be4865cdf8b658a82209624e33ed71cae353b0df254a75db63d1baa35ad99f26f1b399c31f3c666a7fc67ecef3bdcdb7d60e8ada90b722
```

```
The hexadecimal equivalent of SHA1 is :
4175a37afd561152fb60c305d4fa6026b7e79856
```

**Explanation:** The above code takes string and converts it into the byte equivalent using encode() so that it can be accepted by the hash function. The SHA hash functions encode it and then using hexdigest(), hexadecimal equivalent encoded string is printed.

## Graded Task 1: Simulating a Birthday Attack

**Objective:** Create a program that simulates a birthday attack on a simplified hash function.

- **Step 1:** Use a reduced hash size (e.g., truncate the output of SHA-256 to 16 bits).
- **Step 2:** Generate random strings and compute truncated hashes.
- **Step 3:** Find two different strings that produce the same truncated hash (collision).

## Graded Task 2: Exploring Hash Stretching (PBKDF2)

**Objective:** Implement hash stretching using PBKDF2 to demonstrate its defense against brute-force attacks.

- **Step 1:** Implement PBKDF2 to hash a password with multiple iterations.
- **Step 2:** Compare how different iteration counts affect the computation time.

## Graded Task 3: Collision Detection

**Objective:** Demonstrate the difficulty of finding a collision in SHA-256.

- **Step 1:** Write a Python program to generate two different random strings.
- **Step 2:** Compute their SHA-256 hash and check if the hashes are the same (collision).
- **Step 3:** Loop through many iterations to see how unlikely a collision is with a strong hash function like SHA-256.

## Graded Task 4: Password Hashing with Salt

**Objective:** Implement password hashing with a salt to demonstrate how salt improves security.

- **Step 1:** Generate a random salt for each password.
- **Step 2:** Hash the concatenation of the password and the salt.
- **Step 3:** Store both the salt and the hash for future verification.

## Graded Task 5: Hash Stretching

11. **Task 11: Key Derivation and Hash Stretching**

- Implement a password-based key derivation function (e.g., PBKDF2) using a hash function.
- Demonstrate how increasing the number of iterations affects the time taken to compute the hash.Discuss the role of hash stretching in defending against brute-force attacks.