Experiment 13

AIM: Write a Program to implement Cryptographic Hash Functions and Applications (HMAC): to understand the need, design and applications of collision resistant hash functions.

Roll No.	70
Name	MAYURI SHRIDATTA YERANDE
Class	D15-B
Subject	Internet Security Lab
LO Mapped	LO6: Demonstrate the network security system using open source tools.

AIM: Write a Program to implement Cryptographic Hash Functions and Applications (HMAC): to understand the need, design and applications of collision resistant hash functions.

THEORY:

A cryptographic hash function (CHF) is a mathematical algorithm that maps data of an arbitrary size (often called the "message") to a bit array of a fixed size (the "hash value", "hash", or "message digest"). It is a one-way function, that is, a function for which it is practically infeasible to invert or reverse the computation. Ideally, the only way to find a message that produces a given hash is to attempt a brute-force search of possible inputs to see if they produce a match, or use a rainbow table of matched hashes. Cryptographic hash functions are a basic tool of modern cryptography.

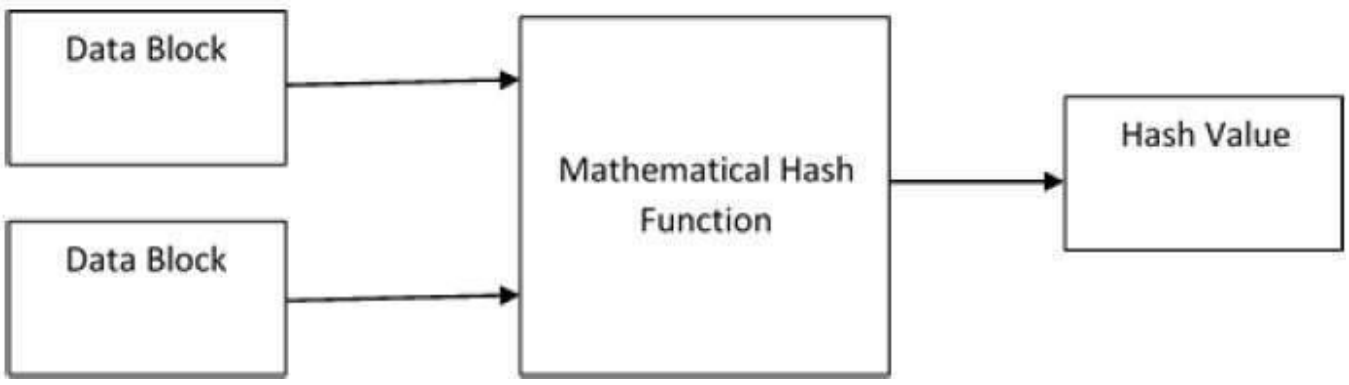
A cryptographic hash function must be deterministic, meaning that the same message always results in the same hash. Ideally it should also have the following properties:

- it is quick to compute the hash value for any given message
- it is infeasible to generate a message that yields a given hash value (i.e. to reverse the process that generated the given hash value)
- it is infeasible to find two different messages with the same hash value
- a small change to a message should change the hash value so extensively that a new hash value appears uncorrelated with the old hash value

Design of Hashing Algorithms

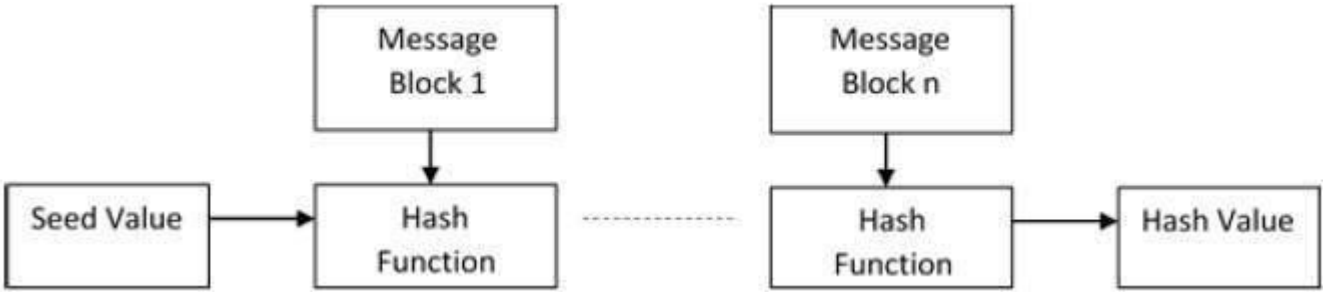
At the heart of a hashing is a mathematical function that operates on two fixed-size blocks of data to create a hash code. This hash function forms part of the hashing algorithm.

The size of each data block varies depending on the algorithm. Typically the block sizes are from 128 bits to 512 bits. The following illustration demonstrates hash function −



Hashing algorithm involves rounds of the above hash function like a block cipher. Each round takes an input of a fixed size, typically a combination of the most recent message block and the output of the last round.

This process is repeated for as many rounds as are required to hash the entire message. Schematic of hashing algorithm is depicted in the following illustration −



Since, the hash value of the first message block becomes an input to the second hash operation, output of which alters the result of the third operation, and so on. This effect, known as an avalanche effect of hashing.

Avalanche effect results in substantially different hash values for two messages that differ by even a single bit of data.

Understand the difference between hash function and algorithm correctly. The hash function generates a hash code by operating on two blocks of fixed-length binary data.

Hashing algorithm is a process for using the hash function, specifying how the message will be broken up and how the results from previous message blocks are chained together.

Popular Hash Functions

Message Digest (MD)

MD5 was the most popular and widely used hash function for quite some years The MD family comprises hash functions MD2, MD4, MD5 and MD6. It was adopted as Internet Standard RFC 1321. It is a 128-bit hash function. MD5 digests have been widely used in the software world to provide assurance about integrity of transferred file. For example, file servers often provide a pre-computed MD5 checksum for the files, so that a user can compare the checksum of the downloaded file to it.

Secure Hash Function (SHA)

Family of SHA comprises four SHA algorithms; SHA-0, SHA-1, SHA-2, and SHA-3. Though from the same family, they are structurally different.

- The original version is SHA-0, a 160-bit hash function, was published by the National Institute of Standards and Technology (NIST) in 1993. It had few weaknesses and did not become very popular. Later in 1995, SHA-1 was designed to correct alleged weaknesses of SHA-0.
- SHA-1 is the most widely used of the existing SHA hash functions. It is employed in several widely used applications and protocols including Secure Socket Layer (SSL) security.
- In 2005, a method was found for uncovering collisions for SHA-1 within a practical time frame making long-term employability of SHA-1 doubtful.
- The SHA-2 family has four further SHA variants, SHA-224, SHA-256, SHA-384, and SHA-512 depending on the number of bits in their hash value. No successful attacks have yet been reported on SHA-2 hash function.
- Though SHA-2 is a strong hash function. Though significantly different, its basic design still follows the design of SHA-1. Hence, NIST called for new competitive hash function designs.

Applications of Hash Functions

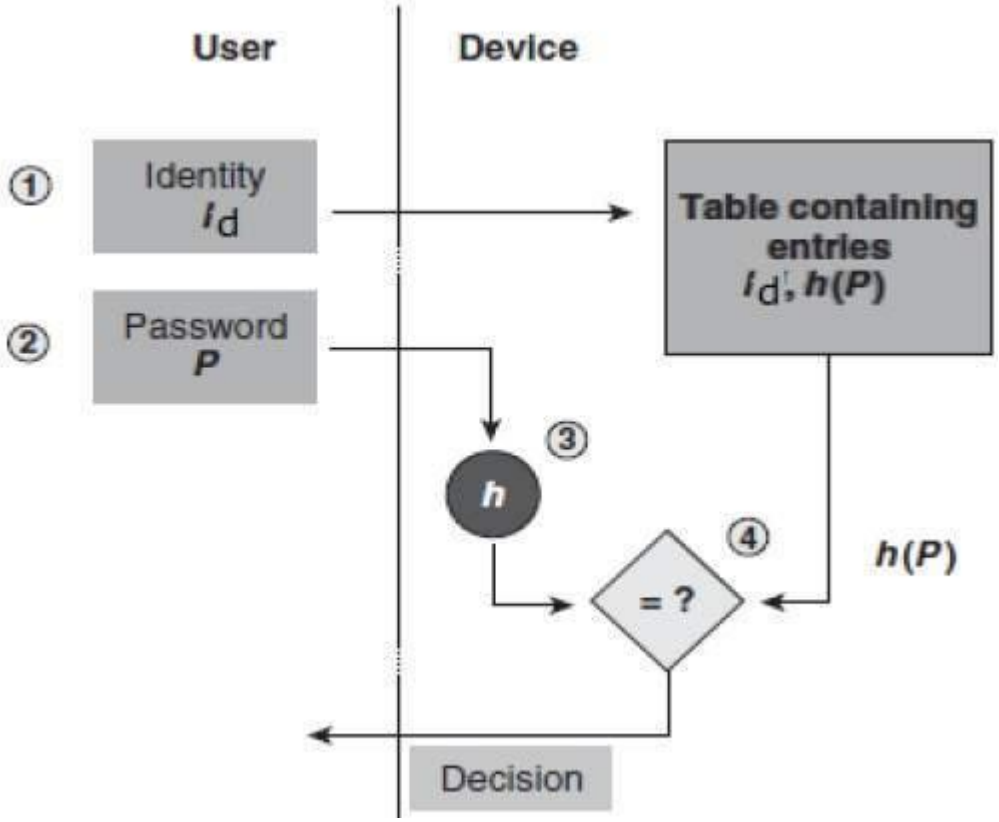
There are two direct applications of hash function based on its cryptographic properties.

Password Storage

Hash functions provide protection to password storage.

- Instead of storing passwords in clear, mostly all logon processes store the hash values of passwords in the file.
- The Password file consists of a table of pairs which are in the form (user id, h(P)).

The process of logon is depicted in the following illustration −



IMPLEMENTATION:

CODE:

```
#include <stdio.h>
#include <windows.h>
#include <wincrypt.h>

int main()
{

HCRYPTPROV  hProv       = NULL;
HCRYPTHASH  hHash       = NULL;
HCRYPTKEY   hKey        = NULL;
HCRYPTHASH  hHmacHash   = NULL;
PBYTE       pbHash      = NULL;
DWORD       dwDataLen   = 0;
BYTE        Data1[]     = {0x70,0x61,0x73,0x73,0x77,0x6F,0x72,0x64};
BYTE        Data2[]     = {0x6D,0x65,0x73,0x73,0x61,0x67,0x65};
HMAC_INFO   HmacInfo;


ZeroMemory(&HmacInfo, sizeof(HmacInfo));
HmacInfo.HashAlgid = CALG_SHA1;

if (!CryptAcquireContext(
    &hProv,                 // handle of the CSP
    NULL,                   // key container name
    NULL,                   // CSP name
    PROV_RSA_FULL,          // provider type
    CRYPT_VERIFYCONTEXT))     // no key access is requested
{
  printf(" Error in AcquireContext 0x%08x \n",
        GetLastError());
  goto ErrorExit;
}


if (!CryptCreateHash(
    hProv,                  // handle of the CSP
    CALG_SHA1,              // hash algorithm to use
    0,                      // hash key
    0,                      // reserved
    &hHash))                // address of hash object handle
{
  printf("Error in CryptCreateHash 0x%08x \n",
        GetLastError());
  goto ErrorExit;
}

if (!CryptHashData(
    hHash,                  // handle of the hash object
```

```
    Data1,                  // password to hash
    sizeof(Data1),          // number of bytes of data to add
    0))                     // flags
{
   printf("Error in CryptHashData 0x%08x \n",
         GetLastError());
   goto ErrorExit;
}

if (!CryptDeriveKey(
    hProv,                  // handle of the CSP
    CALG_RC4,               // algorithm ID
    hHash,                  // handle to the hash object
    0,                      // flags
    &hKey))                 // address of the key handle
{
   printf("Error in CryptDeriveKey 0x%08x \n",
         GetLastError());
   goto ErrorExit;
}


if (!CryptCreateHash(
    hProv,                  // handle of the CSP.
    CALG_HMAC,              // HMAC hash algorithm ID
    hKey,                   // key for the hash (see above)
    0,                      // reserved
    &hHmacHash))            // address of the hash handle
{
   printf("Error in CryptCreateHash 0x%08x \n",
         GetLastError());
   goto ErrorExit;
}

if (!CryptSetHashParam(
    hHmacHash,              // handle of the HMAC hash object
    HP_HMAC_INFO,           // setting an HMAC_INFO object
    (BYTE*)&HmacInfo,       // the HMAC_INFO object
    0))                     // reserved
{
   printf("Error in CryptSetHashParam 0x%08x \n",
         GetLastError());
   goto ErrorExit;
}

if (!CryptHashData(
    hHmacHash,              // handle of the HMAC hash object
    Data2,                  // message to hash
    sizeof(Data2),          // number of bytes of data to add
    0))                     // flags
{
   printf("Error in CryptHashData 0x%08x \n",
         GetLastError());
   goto ErrorExit;
```

```
}


if (!CryptGetHashParam(
    hHmacHash,              // handle of the HMAC hash object
    HP_HASHVAL,             // query on the hash value
    NULL,                   // filled on second call
    &dwDataLen,             // length, in bytes, of the hash
    0))
{
  printf("Error in CryptGetHashParam 0x%08x \n",
        GetLastError());
  goto ErrorExit;
}

pbHash = (BYTE*)malloc(dwDataLen);
if(NULL == pbHash)
{
  printf("unable to allocate memory\n");
  goto ErrorExit;
}

if (!CryptGetHashParam(
    hHmacHash,              // handle of the HMAC hash object
    HP_HASHVAL,             // query on the hash value
    pbHash,                 // pointer to the HMAC hash value
    &dwDataLen,             // length, in bytes, of the hash
    0))
{
  printf("Error in CryptGetHashParam 0x%08x \n", GetLastError());
  goto ErrorExit;
}

// Print the hash to the console.

printf("The hash is:  ");
for(DWORD i = 0 ; i < dwDataLen ; i++)
{
  printf("%2.2x ",pbHash[i]);
}
printf("\n");

// Free resources.
ErrorExit:
    if(hHmacHash)
        CryptDestroyHash(hHmacHash);
    if(hKey)
        CryptDestroyKey(hKey);
    if(hHash)
        CryptDestroyHash(hHash);
    if(hProv)
        CryptReleaseContext(hProv, 0);
    if(pbHash)
        free(pbHash);
```

```
    return 0;
}
```

OUTPUT:

CONCLUSION: We have successfully implemented a program on Cryptographic Hash Functions and Applications (HMAC): and understood the need, design and applications of collision resistant hash functions.