Учреждение образования

Белорусский государственный университет

информатики и радиоэлектроники

Факультет компьютерных систем и сетей

Кафедра информатики

Отчет по лабораторной работе №5

**Хэш-функции**

|  |  |
| --- | --- |
| Выполнил:  студент группы 653501  Ковалевская Алина | Проверил:  Артемьев В.С. |

Минск, 2019

**Задача**

Реализовать программное средства контроля целостности сообщений с помощью вычисления хэш-функции и алгоритма HMAC.

**Введение**

Хеш-функция, или функция свёртки – функция, осуществляющая преобразование массива входных данных произвольной длины в (выходную) битовую строку установленной длины, выполняемое определённым алгоритмом. Преобразование, производимое хеш-функцией, называется хешированием. Исходные данные называются входным массивом, «ключом» или «сообщением». Результат преобразования (выходные данные) называется «хешем», «хеш-кодом», «хеш-суммой», «сводкой сообщения».

Хеш-функции применяются в следующих случаях:

* при построении ассоциативных массивов;
* при поиске дубликатов в сериях наборов данных;
* при построении уникальных идентификаторов для наборов данных;
* при вычислении контрольных сумм от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;
* при сохранении паролей в системах защиты в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся обратной по отношению к использованной хеш-функции);
* при выработке электронной подписи (на практике часто подписывается не само сообщение, а его «хеш-образ»);

и др.

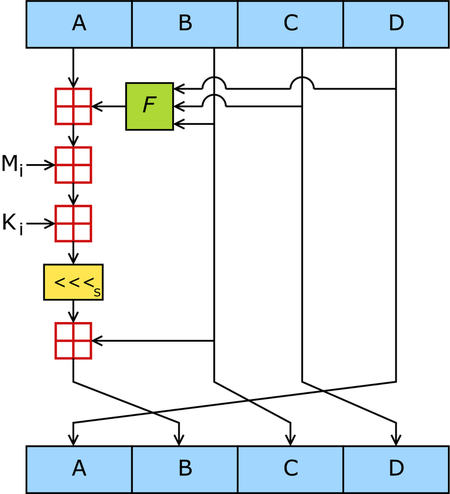
В общем случае (согласно принципу Дирихле) нет однозначного соответствия между хеш-кодом (выходными данными) и исходными (входными) данными. Возвращаемые хеш-функцией значения (выходные данные) менее разнообразны, чем значения входного массива (входные данные). Случай, при котором хеш-функция преобразует более чем один массив входных данных в одинаковые сводки, называется «коллизией». Вероятность возникновения коллизий используется для оценки качества хеш-функций.

Существует множество алгоритмов хеширования, отличающихся различными свойствами. Примеры свойств:

* разрядность;
* вычислительная сложность;
* криптостойкость.

Выбор той или иной хеш-функции определяется спецификой решаемой задачи. Простейшим примером хеш-функции может служить «обрамление» данных циклическим избыточным кодом (англ. CRC, cyclic redundancy code).

MD5 (англ. Message Digest 5) — 128-битный алгоритм хеширования, разработанный профессором Рональдом Л. Ривестом из Массачусетского технологического института (Massachusetts Institute of Technology, MIT) в 1991 году. Предназначен для создания «отпечатков» или дайджестов сообщения произвольной длины и последующей проверки их подлинности. Широко применялся для проверки целостности информации и хранения хешей паролей.



HMAC (сокращение от англ. hash-based message authentication code, код аутентификации (проверки подлинности) сообщений, использующий хеш-функции) – в информатике (криптографии), один из механизмов проверки целостности информации, позволяющий гарантировать то, что данные, передаваемые или хранящиеся в ненадёжной среде, не были изменены посторонними лицами (см. человек посередине). Механизм HMAC использует MAC, описан в RFC 2104, в стандартах организаций ANSI, IETF, ISO и NIST. MAC – стандарт, описывающий способ обмена данными и способ проверки целостности передаваемых данных с использованием секретного ключа. Два клиента, использующие HMAC, как правило, разделяют общий секретный ключ. HMAC – надстройка над MAC; механизм обмена данными с использованием секретного ключа (как в MAC) и хеш-функций. В зависимости от используемой хеш-функции выделяют HMAC-MD5, HMAC-SHA1, HMAC-RIPEMD128, HMAC-RIPEMD160 и т. п.

Было замечено, что скорость работы хеш-функций (например, MD5, SHA-1, RIPEMD128, RIPEMD-160), обычно, выше скорости работы симметричных блочных шифров (например, DES). Возникло желание использовать хеш-функции в MAC, а наличие готовых библиотек с реализациями различных хеш-функций только подтолкнуло эту идею.

Но использовать некоторые хеш-функции в MAC было невозможно. Например, хеш-функция MD5 не может применяться в MAC, так как принимает только один аргумент — данные (строку, последовательность байт).

В HMAC данные «смешивались» с ключом и хеш-функция применялась дважды.

Были предложены и другие механизмы, позволяющие одновременно использовать данные и секретный ключ в существующих алгоритмах хеширования, но HMAC получил наибольшую поддержку.

Преимущества HMAC:

– возможность использования хеш-функций, уже имеющихся в программном продукте;

– отсутствие необходимости внесения изменений в реализации существующих хеш-функции (внесение изменений может привести к ухудшению производительности и криптостойкости);

– возможность замены хеш-функции в случае появления более безопасной или более быстрой хеш-функции.

Механизм HMAC был описан в стандартах организаций ANSI, IETF, ISO и NIST. Реализация HMAC является для протокола IPsec. HMAC используется и в других протоколах интернета, например, TLS. Ожидается, что TLS вскоре заменит SSL и SET.

**Алгоритм**

Обозначения:

– b, block\_size — размер блока в байтах;

– H, hash — хеш-функция;

– ipad — блок вида ( 0x36 0x36 0x36 ... 0x36 ), где байт 0x36 повторяется b раз; 0x36 — константа, магическое число, приведённое в RFC 2104;

– К, key — секретный ключ (общий для отправителя и получателя);

– K0 — изменённый ключ K (уменьшенный или увеличенный до размера блока (до b байт));

– L — размер в байтах строки, возвращаемой хеш-функцией H; L зависит от выбранной хеш-функции и обычно меньше размера блока;

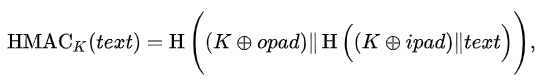
|  |  |  |
| --- | --- | --- |
| Хеш-функция H | b, байт | L, байт |
| MD5 | 64 | 16 |
| SHA-1 | 64 | 20 |
| out = H( in )  b = length( in )  L = length( out ) | | |

– opad — блок вида ( 0x5c 0x5c 0x5c ... 0x5c), где байт 0x5c повторяется b раз; 0x5c — константа, магическое число, приведённое в RFC 2104;

– text — сообщение (данные), которое будет передаваться отправителем и подлинность которого будет проверяться получателем;

– n — длина сообщения text в битах.

Алгоритм HMAC можно записать в виде одной формулы:



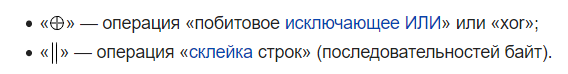
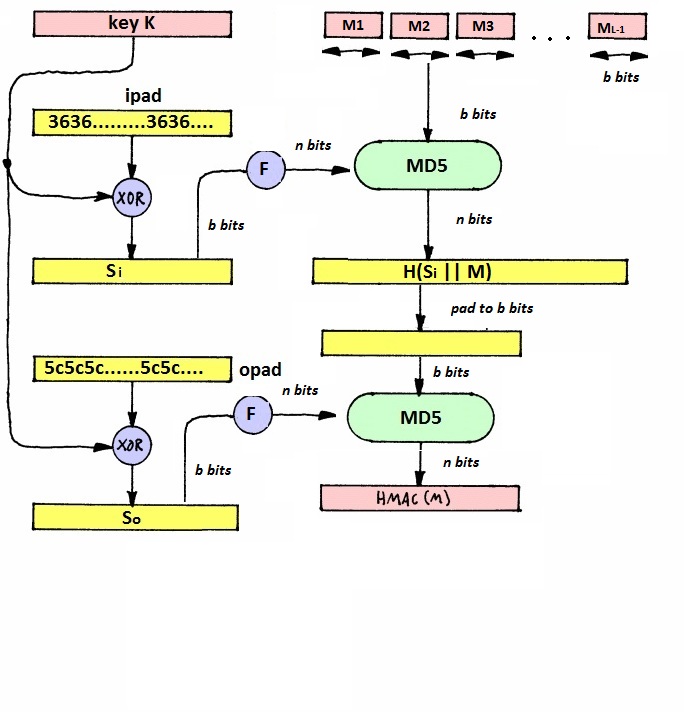


Схема работы алгоритма HMAC-MD5 приведена на рисунке.



**Реализация HMAC**

Этапы работы алгоритм HMAC перечисленные ниже.

1. Получить K0 путём уменьшения или увеличения ключа K до размера блока (до b байт).

1.1. Если длина ключа K равна размеру блока, то копируем K в K0 без изменений и переходим к шагу 2.

IF length( K ) == b THEN :

K\_0 = K

END\_IF

1.2. Если длина ключа K больше размера блока, то к ключу K применяем хеш-функцию H, получаем строку размером в L байт, добавляем нули к правой части этой строки для создания строки размером в b байт, копируем результат в K0 и переходим к шагу 2.

IF length( K ) > b THEN :

x = H( K ) // length( x ) == L

K\_0 = zeros( x, b - L )

END\_IF

1.3. Если длина ключа K меньше размера блока, то добавляем нули к правой части K для создания строки размером в b байт, копируем результат в K0 (например, если length( К ) = 20 (в байтах) и b = 64 (в байтах), то к правой части К будет добавлено 64 - 20 = 44 нулевых байта (0x00)) и переходим к шагу 2.

IF length( K ) < b THEN :

K\_0 = zeros( K, b - length( K ) )

END\_IF

2. Получить блок Si размером в b байт с помощью операции «побитовое исключающее ИЛИ»

Si = xor( K0, ipad )

3. Получить блок So размером в b байт с помощью операции «побитовое исключающее ИЛИ»:

So = xor( K0, opad ) = K0

4. Разбить сообщение (данные, набор байт) text на блоки размером b байт.

5. Склеить строку (последовательность байт) Si с каждым блоком сообщения М.

6. К строке, полученной на прошлом шаге, применить хеш-функцию Н.

7. Склеить строку So со строкой, полученной от хеш-функции H на прошлом шаге.

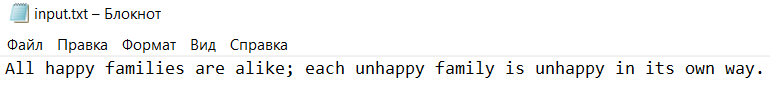
8. К строке, полученной на прошлом шаге, применить хеш-функцию Н.

Ключи размером, меньшим L байт, считаются небезопасными. Рекомендуется выбирать ключи случайным образом и регулярно их менять. Ключи размером, большим L байт, существенно не увеличивают стойкость функции, могут использоваться, если имеются сомнения в случайности данных, используемых для создания ключа и получаемых от генератора случайных чисел.

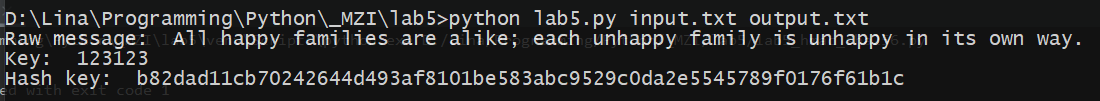
Размер ключа К должен быть больше или равен L/2 байт.

**Демонстрация работы программы**

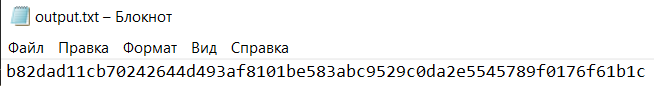
**input.txt**



**k = "123123"**



**output.txt**



**Вывод**

HMAC (сокращение от англ. hash-based message authentication code, код проверки подлинности сообщений, использующий односторонние хеш-функции) — в криптографии, один из механизмов проверки целостности информации, позволяющий гарантировать то, что данные, передаваемые или хранящиеся в ненадёжной среде, не были изменены посторонними лицами (атака типа «man in the middle»).

К таким данным могут относиться, например, данные, передаваемые в запросах API, когда критически важна целостность передаваемой информации, или же при передаче данных из Web-форм.

Была получена простая реализация, позволяющая подписывать любые данные и проверять переданные подписанные данные. Теперь можно подписывать данные, передаваемые через HTTP/REST API, или же создавать продвинутые CSRF-токены для форм и быть уверенными в том, что получаемые данные оригинальны и консистентны.

**Программный код**

**lab5.py**

import sys

K = [

0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,

0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,

0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,

0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,

0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,

0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,

0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,

0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

]

def calc\_sigma0(num):

num = (calc\_rotate\_right(num, 7) ^

calc\_rotate\_right(num, 18) ^

(num >> 3))

return num

def calc\_sigma1(num):

num = (calc\_rotate\_right(num, 17) ^

calc\_rotate\_right(num, 19) ^

(num >> 10))

return num

def calc\_capsigma0(num):

num = (calc\_rotate\_right(num, 2) ^

calc\_rotate\_right(num, 13) ^

calc\_rotate\_right(num, 22))

return num

def calc\_capsigma1(num):

num = (calc\_rotate\_right(num, 6) ^

calc\_rotate\_right(num, 11) ^

calc\_rotate\_right(num, 25))

return num

def calc\_ch(x, y, z):

return (x & y) ^ (~x & z)

def calc\_maj(x, y, z):

return (x & y) ^ (x & z) ^ (y & z)

def calc\_rotate\_right(num, shift, size=32):

return (num >> shift) | (num << size - shift)

def sha256(message):

if isinstance(message, str):

message = bytearray(message, 'ascii')

elif isinstance(message, bytes):

message = bytearray(message)

elif not isinstance(message, bytearray):

raise TypeError

length = len(message) \* 8

message.append(0x80)

while (len(message) \* 8 + 64) % 512 != 0:

message.append(0x00)

message += length.to\_bytes(8, 'big')

blocks = []

for i in range(0, len(message), 64):

blocks.append(message[i:i + 64])

h0 = 0x6a09e667

h1 = 0xbb67ae85

h2 = 0x3c6ef372

h3 = 0xa54ff53a

h5 = 0x9b05688c

h4 = 0x510e527f

h6 = 0x1f83d9ab

h7 = 0x5be0cd19

for message\_block in blocks:

message\_schedule = []

for t in range(0, 64):

if t <= 15:

message\_schedule.append(bytes(message\_block[t \* 4:(t \* 4) + 4]))

else:

term1 = calc\_sigma1(int.from\_bytes(message\_schedule[t - 2], 'big'))

term2 = int.from\_bytes(message\_schedule[t - 7], 'big')

term3 = calc\_sigma0(int.from\_bytes(message\_schedule[t - 15], 'big'))

term4 = int.from\_bytes(message\_schedule[t - 16], 'big')

schedule = ((term1 + term2 + term3 + term4) % 2 \*\* 32).to\_bytes(4, 'big')

message\_schedule.append(schedule)

a = h0

b = h1

c = h2

d = h3

e = h4

f = h5

g = h6

h = h7

for t in range(64):

t1 = ((h + calc\_capsigma1(e) + calc\_ch(e, f, g) + K[t] +

int.from\_bytes(message\_schedule[t], 'big')) % 2 \*\* 32)

t2 = (calc\_capsigma0(a) + calc\_maj(a, b, c)) % 2 \*\* 32

h = g

g = f

f = e

e = (d + t1) % 2 \*\* 32

d = c

c = b

b = a

a = (t1 + t2) % 2 \*\* 32

h0 = (h0 + a) % 2 \*\* 32

h1 = (h1 + b) % 2 \*\* 32

h2 = (h2 + c) % 2 \*\* 32

h3 = (h3 + d) % 2 \*\* 32

h4 = (h4 + e) % 2 \*\* 32

h5 = (h5 + f) % 2 \*\* 32

h6 = (h6 + g) % 2 \*\* 32

h7 = (h7 + h) % 2 \*\* 32

return ((h0).to\_bytes(4, 'big') + (h1).to\_bytes(4, 'big') +

(h2).to\_bytes(4, 'big') + (h3).to\_bytes(4, 'big') +

(h4).to\_bytes(4, 'big') + (h5).to\_bytes(4, 'big') +

(h6).to\_bytes(4, 'big') + (h7).to\_bytes(4, 'big'))

def hmac\_md5(key, message):

i\_key = bytearray()

o\_key = bytearray()

key = key.encode()

message = message.encode()

blocksize = 64

if len(key) > blocksize:

key = bytearray(sha256(key))

elif len(key) < blocksize:

i = len(key)

while i < blocksize:

key += b"\x00"

i += 1

for i in range(blocksize):

i\_key.append(0x36 ^ key[i])

o\_key.append(0x5C ^ key[i])

return sha256(bytes(o\_key) + sha256(bytes(i\_key) + message)).hex()

if \_\_name\_\_ == "\_\_main\_\_":

k = "123123"

if len(sys.argv) < 2:

exit(1)

file\_in = sys.argv[1]

file\_out = sys.argv[2]

with open(file\_in, "r+") as f:

m = f.read()

print("Raw message: ", m)

print("Key: ", k)

h\_key = hmac\_md5(k, m)

print("Hash key: ", h\_key)

with open(file\_out, "w+") as f:

f.write(h\_key)