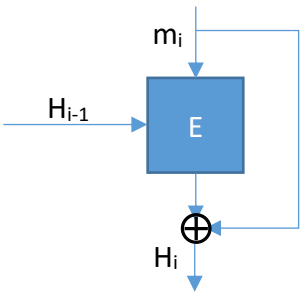
Thomas Mackintosh
ECE 4802
21 November 2016

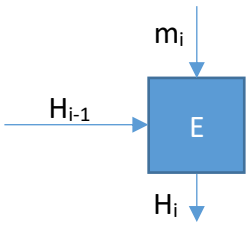
Assignment 4

1. 12.3:
   1. For $c = e_k(x \| h(x))$, if an attacker knows the whole plaintext x then they know the entire input into the encryption. With knowledge of both the input and ciphertext output of the encryption function, the key to the encryption function could be brute forced if the key space is small enough. With knowledge of the key k, an attacker could potentially verify any message x^1. This would only work in a situation where the key k does not change, i.e. this attack would not work with a one-time pad.
   2. For $c = e_{k1}(x \| MAC_{k2}(x))$, the above attack would not work. Assuming the length of k_2 is long enough that the attacker cannot reasonable brute force it, the attacker will not be able to crack the key k_2 to get the correct output of the MAC. Without knowledge of the key to the MAC the attacker cannot get the key k_1 and thus will not be able to verify any message x^1.
2.
   (a) Draw the block diagram for both constructions

Matyas-Meyer-Oseas



Matyas-Meyer-Oseas (modified)



   (b) Show why the modified construction is not secure by using the decryption function of the block cipher to obtain a (second) preimage. Assume H_0 is all zeroes.

   If we use the decryption function on the modified Matyas-Meyer-Oseas block, we get the message directly as an output: $Dec(Enc_{H_0}(m_1)) = m_1$ for the first block of the block chain H_0. If we move to the next block, we know the input to the block will be some message m_2 and the hash result from the previous block, H_1. We can easily use the decryption function on this block to obtain a second preimage: $Dec(Enc_{H1}(m_2) = m_2$.

   (c) Given the above method for finding preimages, describe how to find collisions.

   We can do $Enc_{Hi-1}(m_i) = H_i$ to find an m such that m_i != m_i^1 and compute whether $H(m_i) = H(m_i^1)$.

3. Below is the code for my implementation of an AES CBC-MAC:

```
### This is a template file for a simple CBCMAC for Python 3
##
##  Please implement the provided functions and assure that your code
##  works correctly for the example given below
##
##  Name: Thomas Mackintosh
##

from Crypto.Cipher import AES
import binascii

key = b'Sixteen byte key'
iv  = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
out = b''

m1  = b'The quick brown fox jumps over the lazy dog'
m2  = b'The quick brown fox jumps over the lazy doh'
c1  = b'\x94maSb\x14\x08\x15\xef<\x8c:\xbe\xb9LF'
c2  = b'|K\x8b\x06\x96K#\x1d\x87\xdd\x1e\xca\xa9o\xad\x83'

def padding(m):
    """Append padding to message in order to
       become multiple of 16 in order to fit in
       AES input"""

    # single-1 padding (a single 1 followed by zeroes)
    # combined with length strengthening where the length in
    # bits is appended as a 64-bit number.

    m_hex = binascii.hexlify(m)
    m_hex += b'80' #single-1 padding appended first
    while len(m_hex) < 120:
        m_hex += b'0'
    m_hex += b'00000158'

    # print ("Message as hex: ", m_hex, "\n")
    return m_hex

def CBCMACbasedOnAES(message, key, init_vec):
    """This function computes the MAC of message using key.
       The MAC function is CBC-MAC with AES and both single-1
       padding and length strengthening provided by the
       padding function.
       key must be convertible to bytes of length 16
       message must be convertible to bytes type"""

    cbc = AES.new(key, AES.MODE_CBC, init_vec)

    out = cbc.encrypt(message)[-16:]
    return out

def main():
    #For M1
    print("Message: ",m1)
    pad_m = bytes.fromhex(hex(int(padding(m1),16))[2:])
    print("Need: ", c1)
    CBC = CBCMACbasedOnAES(pad_m, key, iv)
    print("Have: ", CBC, "\n")

    #For M2
    print("Message: ",m2)
    pad_m = bytes.fromhex(hex(int(padding(m2),16))[2:])
    print("Need: ", c2)
    CBC = CBCMACbasedOnAES(pad_m, key, iv)
    print("Have: ", CBC)



    """ Two testvectors are given below:
    m1 =          b'The quick brown fox jumps over the lazy dog'
    CBC-MAC1 =    b'\x94maSb\x14\x08\x15\xef<\x8c:\xbe\xb9LF'

    m2 =          b'The quick brown fox jumps over the lazy doh'
    CBC-MAC2 =    b'|K\x8b\x06\x96K#\x1d\x87\xdd\x1e\xca\xa9o\xad\x83'
    """

if __name__ == '__main__':
    main()
```

This code provides the following output:

```
Message:  b'The quick brown fox jumps over the lazy dog'
Need:  b'\x94maSb\x14\x08\x15\xef<\x8c:\xbe\xb9LF'
Have:  b'\x94maSb\x14\x08\x15\xef<\x8c:\xbe\xb9LF'

Message:  b'The quick brown fox jumps over the lazy doh'
Need:  b'|K\x8b\x06\x96K#\x1d\x87\xdd\x1e\xca\xa9o\xad\x83'
Have:  b'|K\x8b\x06\x96K#\x1d\x87\xdd\x1e\xca\xa9o\xad\x83'
```

4. Below is the code for cracking hashed passwords using SHA512:

```
from Crypto.Hash import SHA512
from timeit import default_timer as timer

def crack_passwords(passwords_from_file):
    with open('dictionary.txt', 'r') as fh:

        dictionary = [line.rstrip('\r\n') for line in fh.readlines()]

        dictl = len(dictionary) - 1
        pt = []

        for i in range (0,len(passwords_from_file),1):
            #start = timer() can be used to measure how long it takes to crack each password
            vec = passwords_from_file[i].split('$') #Separates each line into its individual parts

            if i < 15:
                for j in range(0, dictl, 1):
                    Hash = SHA512.new(vec[2].encode('utf-8') + dictionary[j].encode('utf-8'))

                    for k in range(0,4999,1): #All are SHA512 and default to 5000 rounds
                        Hash = SHA512.new(Hash.hexdigest().encode('utf-8'))

                    if (Hash.hexdigest() == vec[3]):
                        pt.append(dictionary[j])
                        end = timer()
                        print(dictionary[j]) #Print dictionary word on successful crack
                        break

            else:
                for j in range(0, dictl,1):
                    Hash = SHA512.new(vec[3].encode('utf-8')+dictionary[j].encode('utf-8'))

                    for k in range(0,24999,1):
                        Hash =SHA512.new(Hash.hexdigest().encode('utf-8'))

                    if (Hash.hexdigest() == vec[4]):
                        pt.append(dictionary[j])
                        end = timer()
                        print(dictionary[j]) #Print dictionary word on successful crack
                        break

    return pt # return list of plaintexts

with open('passwords.txt', 'r') as fh:
    plaintexts = crack_passwords([line.rstrip('\r\n') for line in fh.readlines()])
    # for i in fh:
    #     print i
    # Can use this notation to print the final list of plaintexts after all calculations finish
    # Instead I print the words as they are found withing the crack_passwords function.
```

The above code provides the following output:

```
Airmont
Ansonia
Anguilla
Apple Grove
Altus
Algonquin
Algerita
Annandale
Alvwood
Allenhurst
Ambler
Alamance
Allen City
Anselma
Ambridge
Agency
Adgateville
Accord
Abeytas
Advance
```

If we output the code with the time taken to crack each password, we get the following output:

```
PASSWORD: Airmont    Time taken: 5.550299601018196 seconds
PASSWORD: Ansonia    Time taken: 20.517417489987565 seconds
PASSWORD: Anguilla    Time taken: 18.958472188998712 seconds
PASSWORD: Apple Grove    Time taken: 23.13965288698091 seconds
PASSWORD: Altus    Time taken: 14.389990436990047 seconds
PASSWORD: Algonquin    Time taken: 9.697414041991578 seconds
PASSWORD: Algerita    Time taken: 9.250819558015792 seconds
PASSWORD: Annandale    Time taken: 19.625334457989084 seconds
PASSWORD: Alvwood    Time taken: 14.420188821997726 seconds
PASSWORD: Allenhurst    Time taken: 22.174967784027103 seconds
PASSWORD: Ambler    Time taken: 15.48839113197755 seconds
PASSWORD: Alamance    Time taken: 6.359459736995632 seconds
PASSWORD: Allen City    Time taken: 10.89969251700677 seconds
PASSWORD: Anselma    Time taken: 20.171272645995487 seconds
PASSWORD: Ambridge    Time taken: 17.4873442449898 seconds
PASSWORD: Agency    Time taken: 22.654370269010542 seconds
PASSWORD: Adgateville    Time taken: 16.702048188017216 seconds
PASSWORD: Accord    Time taken: 6.616238359012641 seconds
PASSWORD: Abeytas    Time taken: 2.9110286129871383 seconds
```

PASSWORD: Advance Time taken: 18.245870003011078 seconds

From the above results, we see that it takes 81.04014845800702 seconds to get through the first 5 passwords which are preceded with a salt at 5000 rounds (average 16.2080296916 seconds per password.) We also see it takes 130.680724372 seconds to get through the next 10 passwords without a hash at 5000 rounds (average 13.068072437 seconds per password.) Lastly, it takes 44.840311862 seconds to get through the last 5 passwords which are hashed at 25000 rounds without a salt (average 8.9680623724 seconds.) From this data we can see the most secure form of SHA512 is with a salt at 5000 rounds. SHA512 hashed at 25000 rounds is the least secure as it can be broken the quickest.