ECE 4802, Project 4

Calvin Figuereo-Supraner

November 22 2016

Versioning

```
$ python3 --version
Python 3.5.2
```

Usage

```
$ ./q3.py
$ ./q4.py
```

Packages

The program `q3.py` requires the package below.

```
$ pip3 install pycrypto
```

The program `q4.py` requires the packages below.

```
$ pip3 install hashlib timeit binascii
```

Problem 1

1a

The technique to compute c is given below, where e_k is a stream cipher:

$$c = e_k(x||h(x))$$

Assuming s is our stream, we can also write:

$$c = s \oplus (x||h(x))$$

If Oscar is given x, he can compute $h(x)$, allowing him to recover s:

$$s = c \oplus (x||h(x))$$

Oscar has a replacement x' he wants to use, so he computes $h(x')$, and generates a valid c':

$$c' = s \oplus (x'||h(x'))$$

This attack will not work if k is a one-time pad: Oscar can still recover s, but is unable to use s again to create c'.

1b

No – Oscar can recover the first $length(x)$ bits of s, but not the remaining bits that were used in the MAC. He also doesn't know k_2.

Problem 2

2a

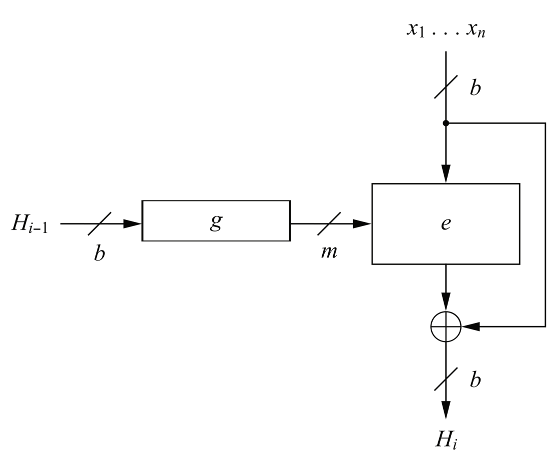

Figure 1: *Matyas-Meyer-Oseas* mode

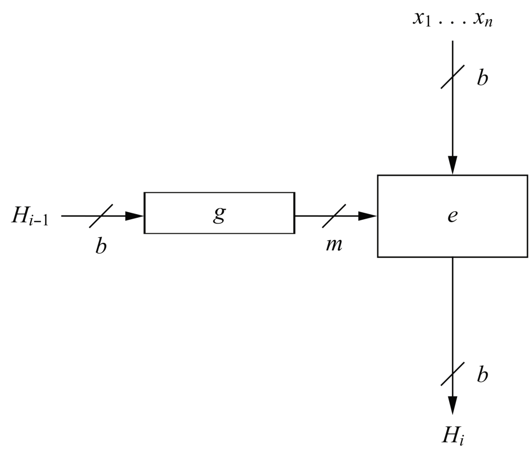

Figure 2: Modified *Matyas-Meyer-Oseas* mode

2b

The above diagram is represented by:

$$H_i = \mathsf{Enc}_{H_{i-1}}(m_i)$$

The modified construction is not secure, because it allows an attacker to prepend blocks to the message m.

2c

If an attacker can obtain a second preimage, they can obtain a 'third' preimage by the same method. Those two possible preimages are then considered a collision under that hash scheme.

Problem 3

The file `q3.py` implements CBC-MAC using AES.

Problem 4

The file `q4.py` recovers all passwords from 'passwords.txt'. Below is the output with `ENABLE_TIMING` set to `False`, then `True`. The benchmarks are on an Ubuntu VM with 768MB RAM and one core of an Intel m3 0.9GHz CPU.

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

```
Airmont           1.96634444600204 sec
Ansonia           7.982376846004627 sec
Anguilla          6.823614074004581 sec
Apple Grove       7.839919405996625 sec
Altus             5.326477777998662 sec
Algonquin         3.4030003007501364e-05 sec
Algerita          1.2102995242457837e-05 sec
Annandale         1.9330000213813037e-05 sec
Alvwood           1.3857999874744564e-05 sec
Allenhurst        1.1113006621599197e-05 sec
Ambler            1.3826000213157386e-05 sec
Alamance          8.384995453525335e-06 sec
Allen City        1.0727999324444681e-05 sec
Anselma           1.762100146152079e-05 sec
Ambridge          1.4170997019391507e-05 sec
Agency            8.351534323999658 sec
Adgateville       7.010622478999721 sec
Accord            2.5236064480050118 sec
Abeytas           1.460132076004811 sec
Advance           9.291494026001601 sec
```

The average time-to-crack for each password policy is as follows:

Default rounds, no salt 0.100 sec

Default rounds, salted 5.988 sec

Extra rounds, salted 5.727 sec

The default rounds without salt password policy is the weakest, since hashing `dictionary.txt` once creates a quick lookup table. Despite the benchmarks, the extra-rounds policy is the strongest: the lower time-to-crack is because the plaintexts were closer to the start of `dictionary.txt`.

Source Code

q3.py

```
#!/usr/bin/python3

from Crypto.Cipher import AES

def pad(message):
    """Append padding to message for AES input."""
    m_len = len(message)
    s1_pad = b'\x80' + bytes(15 - (m_len % 8)) # single-1 padding
    ls_pad = (m_len*8).to_bytes(8, byteorder='big') # length strengthening
    return message + s1_pad + ls_pad

def CBCMAC_AES(message, key):
    """Compute the CBC-MAC of the message under AES."""
    enc_message = AES.new(key, AES.MODE_CBC, bytes(16)).encrypt(pad(message))
    return enc_message[-16:]

def main():
    tests = [
        {'key': b'Sixteen byte key',
         'msg': b'The quick brown fox jumps over the lazy dog',
         'mac': b'\x94maSb\x14\x08\x15\xef<\x8c:\xbe\xb9LF'},
        {'key': b'Sixteen byte key',
         'msg': b'The quick brown fox jumps over the lazy doh',
         'mac': b'|K\x8b\x06\x96K#\x1d\x87\xdd\x1e\xca\xa9o\xad\x83'}
    ]
    for t in tests:
        assert CBCMAC_AES(t['msg'], t['key']) == t['mac']
    print("Success!")
    return
```

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

q4.py

```
#!/usr/bin/env python3

from timeit import default_timer as timer
from hashlib import sha512

ENABLE_TIMING = False # global; if enabled, print crack time for each password

WORDLIST = [] # global; lines in dictionary.txt
HASHLIST = [] # global; hashes for dictionary.txt

class PasswordDetails:
    """Class that describes password policy as fields."""
    def __init__(self, rounds, salt, hash_data):
        self.rounds     = rounds
        self.salt       = salt
        self.hash_data  = hash_data

def crack_passwords(passwords_from_file):
    """Return plaintext passwords recovered from `passwords.txt`."""
    global WORDLIST, HASHLIST
    with open('dictionary.txt', 'r') as fh:
        WORDLIST = [line.rstrip('\r\n') for line in fh.readlines()]
    HASHLIST = [my_hash(word, 5000, '') for word in WORDLIST]
    plaintexts = [brute(password) for password in passwords_from_file]
    return plaintexts

def my_hash(message, rounds, salt):
    """SHA-512 hash with rounds and salt parameters."""
    digest = sha512((salt + message).encode('utf-8')).hexdigest() # first round
    for _ in range(rounds-1): # remaining rounds
        digest = sha512(digest.encode('utf-8')).hexdigest()
    return digest # potential speed-up without .encode() and .hexdigest()?

def parse(line):
    """Parse the password line into a PasswordDetails class."""
    fields = line.split('$')
    # Custom rounds with salt
    if line.count('$') == 4:
        (_, _, rounds, salt, hash_data) = fields
        rounds = int(rounds.strip('rounds='))
    # Default rounds without salt
    elif '$$' in line:
```

```
        (_, _, _, hash_data) = fields
        rounds, salt = 5000, ''
    # Default rounds with salt
    else:
        (_, _, salt, hash_data) = fields
        rounds = 5000
    return PasswordDetails(rounds, salt, hash_data)

def brute(line):
    """Return the plaintext that generates the given line."""
    start = timer()
    ################## Start timed block
    details = parse(line) # minor speed-up by fetching fields only once
    salt, rounds, hash_data = details.salt, details.rounds, details.hash_data
    if salt == '': # weakest policy
        pt = '{:12}'.format(WORDLIST[HASHLIST.index(hash_data)])
    else: # either salted policy
        for word in WORDLIST:
            if my_hash(word, rounds, salt) == hash_data:
                pt = '{:12}'.format(word); break
    ################## End timed block
    end = timer()
    if ENABLE_TIMING: pt += '\t{} sec'.format(end - start)
    return pt

def main():
    """Crack all passwords in `passwords.txt` and print to STDOUT."""
    with open('passwords.txt', 'r') as fh:
        plaintexts = \
            crack_passwords([line.rstrip('\r\n') for line in fh.readlines()])
        print('\n'.join(plaintexts))
    return

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