Python with Applications, PIC16
E. Ryu
Spring 2019

UCLA

Homework 3
Due 5pm, Monday, May 6, 2019

Name your file `hw3.py` and submit on CCLE. Comment your code adequately.

Problem 1: The file `hw3_unsalted_db.txt` contains a database of (fake) usernames and password hashes. The SHA-256 hash function was used without salting. Using `words.txt` provided for hw1, perform a dictionary attack on this database. For simplicity, use only lower case letters in the attack.

Remark. The true passwords are provided in `hw3_true_passwords.txt` just for fun. Your code may not use these true passwords.

Problem 2: The file `hw3_salted_db.txt` contains a database of (fake) usernames and password hashes. The SHA-256 hash function was used and the input was salted in the following manner

```
hsh = sha256(username + password)
```

Using `words.txt` provided for hw1, perform a dictionary attack on this salted database. How much longer does it take? You do not have to run this attack to completion. Find out how long the attack would take and write your response as a comment in `hw3.py`.

Problem 3: For this problem, perform experiments and write up your response in text as a comment in `hw3.py`. You will not submit code.

Consider the setup where a server transmits a message and its MD5 hash. For simplicity, assume the hash is transmitted accurately, but 40 characters in the message is changed to a different random English letter, capital or lower case. (There are 746 chracters in total.) The goal is to study how likely it is for this message to falsely pass the message integrity test.

The following code tries to find a random corruption of the original message that produces the same hash. For how long do you expect the code to run before completion?

```
from hashlib import md5
import random
import string

def verifyHash(msg,hsh):
    return md5(msg).hexdigest()==hsh

original = '''There are many variations of passages of Lorem Ipsum
available, but the majority have suffered alteration in some form,
by injected humour, or randomised words which don't look even
slightly believable. If you are going to use a passage of Lorem
Ipsum, you need to be sure there isn't anything embarrassing hidden
in the middle of text. All the Lorem Ipsum generators on the
Internet tend to repeat predefined chunks as necessary, making this
the first true generator on the Internet. It uses a dictionary of
over 200 Latin words, combined with a handful of model sentence
structures, to generate Lorem Ipsum which looks reasonable. The
generated Lorem Ipsum is therefore always free from repetition,
injected humour, or non-characteristic words.'''.replace('\n',' ')
msg_hash = md5(original).hexdigest()

while True:
    lst = list(original)

    #pick 40 random locations
    for randInd in random.sample(range(len(original)),40):
        #find random corruption different from original character
        corrupted_letter = lst[randInd]
        while (corrupted_letter == lst[randInd]):
            corrupted_letter = random.choice(string.ascii_letters)
        lst[randInd] = corrupted_letter

    print "".join(lst)
    if verifyHash("".join(lst),msg_hash):
        break

print "We found a random corruption resulting in the same hash!"
```

Problem 4: Consider a ledger where the debtor signs the hashes of the previous and current messages, i.e., the ledger contains entries

```
( message, sign(hash(previous_message)+hash(message)) )
```

This system is vulnerable to the following attack.

Alice manipulates Bob to quickly (and legitimately) send the same amount of money within 1 second. (Perhaps Alice purchases a product twice and refunds both purchases at the same time.) If Alice is successful, the ledger shall have two identical (legitimate) transactions

```
(previous message, sign( ... ) )
("Bob,Alice,10.00,04/01/19,11:11,refund",
    sign(hash(previous message)+hash("Bob,Alice...")) )
("Bob,Alice,10.00,04/01/19,11:11,refund",
    sign(hash("Bob,Alice...")+hash("Bob,Alice...")) )
(next message, sign(hash("Bob,Alice...")+hash(next message))
```

If Alice has access to the ledger, then she can duplicate the second transaction.

```
(previous message, sign( ... ) )
("Bob,Alice,10.00,04/01/19,11:11,refund",
    sign(hash(previous message)+hash("Bob,Alice...")) )
("Bob,Alice,10.00,04/01/19,11:11,refund",
    sign(hash("Bob,Alice...")+hash("Bob,Alice...")) )
("Bob,Alice,10.00,04/01/19,11:11,refund",
    sign(hash("Bob,Alice...")+hash("Bob,Alice...")) )
("Bob,Alice,10.00,04/01/19,11:11,refund",
    sign(hash("Bob,Alice...")+hash("Bob,Alice...")) )
("Bob,Alice,10.00,04/01/19,11:11,refund",
    sign(hash("Bob,Alice...")+hash("Bob,Alice...")) )
("Bob,Alice,10.00,04/01/19,11:11,refund",
    sign(hash("Bob,Alice...")+hash("Bob,Alice...")) )
(next message, sign(hash("Bob,Alice...")+hash(next message)) )
```

In a one paragraph writing, explain whether it is possible to detect this attack.

We can prevent this attack by adding a counter to each transaction, and requiring the signature to also sign the counter. When Bob wishes to send money to Alice, she looks at the prior entry and its counter in the ledger and sends to the central authority

```
( message, counter, sign(str(counter)+hash(previous message)+hash(message)) )
```

where `counter` is `1` larger than the `counter` of the previous entry. The central authority accepts this entry if it can verify the signature and the `counter` value is correct. Implement this system and pply it to `hw3_ledger.txt` to output a secure signed ledger `hw3_signed_ledger.txt`. Furthermore, implement a function that reads `hw3_signed_ledger.txt` and verfies the validity of all entries.

Problem 5: The previous ledger still has privacy concerns, as we can see the transaction history. One approach is to use pseudonyms, but this mechanism requires someone to keep track of a table matching pseudonyms with real names. Instead, we assign transactions not to users, but rather to owners of private signing keys. Also, we remove the transaction description from the ledger. (We could keep an encrypted description, but we remove it for simplicity.)

Consider the system where the recorded entries are

```
veri_key_A,veri_key_B,225,9/15/18,22:28
veri_key_C,veri_key_D,7,11/20/18,19:23
veri_key_E,veri_key_F,12.7,01/05/19,16:16
```

with the appropriate signature appended. When Alice (who owns `sign_key_A`) wishes to send money to the owner of `sign_key_B` (whomever that may be), she looks at the prior entry and its counter and sends

```
( "veri_key_A,veri_key_B,225,9/15/18,22:28", counter,
    sign(str(counter)+hash(previous_message)+hash(message)) )
```

to the central authority. The central authority accepts this entry if it can verify the signature using `veri_key_A` and the `counter` value is correct.

In a few paragraphs of writing, answer the following questions. To send money, does Alice have to reveal her true identity to the recipient of the money, the central authority, or the world? To receive money from Alice, what information do I need to provide her? How does the central authority keep track of how much money each individual owes?

Implement this system and pply it to `hw3_ledger.txt` to output an anonymous secure signed ledger `hw3_anon_signed_ledger.txt`. Furthermore, implement a function that reads `hw3_anon_signed_ledger.txt` and verfies the validity of all entries.