Lab passwords_solved

October 16, 2018

1 Password Authentication Using Python

In this lab activity, you will explore the logic of how Unix-based systems generate secure passwords by using the Python programming language.

2 Importing the Necessary Modules

To do this activity, you need to have the following python modules installed in your computer:

- passlib

Remember that you can use the command "!pip install . . . " to install these modules.

```
In [1]: !pip install passlib
```

```
Requirement already satisfied: passlib in c:\programdata\anaconda3\lib\site-package
```

Now import the installed module.

```
In [2]: import passlib
```

3 Creating a Unix Password

The **passlib.hash** module contains all the password hash algoriths that have been used in the history of UNIX systems, such as DES, MD5 and SHA-256. To use them, ypu only have to import the hashing function and use it with the correct parameters.

If you want to find out more about the currently active and inactive hashes in UNIX, check the following link: https://passlib.readthedocs.io/en/stable/lib/passlib.hash.html

3.1 DES

First, we wil test the classical DES hashing function. To do so, you need to import from the **passlib.hash** module the function called **des_crypt**:

- from passlib.hash import des_crypt

Afterwards, you can call the hashing function by performing the following command:

- deshash1 = des_crypt.hash(**string**)

```
In [3]: ## Use this cell to import the DES hashing function and hash the password "
        ## and print your result.
        from passlib.hash import des_crypt
        deshash1 = des_crypt.hash('a')
        deshash1

Out[3]: 'NPLeNggxxTNnU'
```

Notice that if you call the function again to hash the same password, the output will be different

```
In [4]: ## Use this cell to call the DES hashing function to hash password "a". Sto
        ## and print your result.
        deshash2 = des_crypt.hash('a')
        deshash2

Out[4]: 'pUxuat68NtMdE'
```

This is due to the fact that the algorithm has added **salt** and thus, it generates a different hash every time. You can specify the salt used by adding the salt keyword to the function as follows:

- deshash3 = des_crypt.hash('abc', salt=**string with two characters**)

Where the two characters used can be numbers 0-9, letters a-z, letters A-Z and special characters ./

In practice, it is better **NOT** to specify the salt, because the aim is to obtain a random hash.

```
In [5]: ## Use this cell to call the DES hashing function to hash password "a" with
        ## Store the result in variable 'deshash3' and print your result.
        deshash3 = des_crypt.hash('a', salt='08')
        deshash3

Out[5]: '0884v.jRPyncg'
```

You should obtain the hash **0884v.jRPyncg**. Notice that although salt was added, all passwords have the same length:

```
In [6]: ## Run this cell to print the length of the DES hashes
        print('Size of deshash1:', len(deshash1))
        print('Size of deshash2:', len(deshash2))
        print('Size of deshash3:', len(deshash3))

Size of deshash1: 13
Size of deshash2: 13
Size of deshash3: 13
```

3.1.1 Verifying a password

To validate an incoming password against your existing hashes, you can use the verify function.

- des_crypt.verify(**incoming password**, **hashed password**)

```
In [7]: ## Use this cell to verify different combinations of passwords against your
        print(des_crypt.verify('a', deshash1))
        print(des_crypt.verify('ab', deshash3))
```

```
True
False
```

3.1.2 Why is DES considered insecure?

DES is no longer considered secure, for a variety of reasons:

- Its use of the DES stream cipher, which is vulnerable to practical pre-image attacks, and considered broken, as well as having too-small key and block sizes.
- The 12-bit salt is considered to small to defeat rainbow-table attacks (most modern algorithms provide at least a 48-bit salt).
- The fact that it only uses the lower 7 bits of the first 8 bytes of the password results in a dangerously small keyspace which needs to be searched.

Source: https://passlib.readthedocs.io/en/stable/lib/passlib.hash.des_crypt.html

3.2 Other hashing functions

3.2.1 MD5

MD5 is more secure than DES and contains an option to specify the size of the salt. This is required by some standards such as Cisco compatible hashes, which require a salt size of four.

```
In [8]: ## Run this cell to generate the MD5 hash of password 'a' with a salt size
        from passlib.hash import md5_crypt
        md5hash = md5_crypt.hash("a", salt_size = 4)
        md5hash
```

```
Out[8]: '$1$UfGG$bTxxs6BbCY3LL.0xHzDms1'
```

This hash appears to be more secure than the DES ones!

3.2.2 Why is MD5 considered insecure?

- It relies on the MD5 message digest, for which theoretical pre-image attacks exist.
- Its fixed number of rounds (combined with the availability of high-throughput MD5 implementations) means this algorithm is increasingly vulnerable to brute force attacks.

Source: https://passlib.readthedocs.io/en/stable/lib/passlib.hash.md5_crypt.html

3.3 SHA-256

This algorithm includes fixes and advancements such as number of rounds and the use of cryptographic primitives, which are low level functions that further secure the password. In this case, the salt can be from 0 to 16 characters long.

```
In [9]: ## Run this cell to generate three hashes of password 'a'
        from passlib.hash import sha256_crypt

        # Regular hash
        shahash1 = sha256_crypt.hash("a")
        print(shahash1)

        # Same, but with explict number of rounds
        shahash2 = sha256_crypt.using(rounds=12345).hash("a")
        print(shahash2)

        # Same, but with explict salt
        shahash3 = sha256_crypt.hash("a", salt = '08')
        print(shahash3)
```

```
$5$rounds=535000$kQmDk805NPruXtZ9$o8Krd33cusgqFF7hia5rBudWriaxy4D1LPbQaw2r8I7
$5$rounds=12345$znuWLT7zJBf2TuJI$VoiTMp6cpIe3OYvgTHGolQdwwU4oUbHrQZzIq9ZFJW8
$5$rounds=535000$08$VYkpN7VWeleEgTXfcmOqffeeTmVk38mH.tIrIg9s6yA
```

Notice that the hash specifies that SHA256 has been used (indicated by the 5) and the number of rounds used. Moreover, the size of the hashes will vary according to the size and rounds used.

```
In [10]: ## Run this cell to print the length of the SHA-256 hashes
         print('Size of shahash1:', len(shahash1))
         print('Size of shahash2:', len(shahash2))
         print('Size of shahash3:', len(shahash3))
```

```
Size of shahash1: 77
Size of shahash2: 76
Size of shahash3: 63
```

3.3.1 Why is SHA-256 considered insecured?

Even SHA256 has some security issues:

- The algorithm's initialization stage contains a loop which varies linearly with the square of the password size; and further loops, which vary linearly as $passwordsize * rounds$.
- This means an attacker could provide a maliciously large password at the login screen to attempt a DOS on a publically visible login. For example, a 32kb password would require hashing 1Gb of data. Passlib mitigates this by limiting the maximum password size to 4k by default.

- An attacker could also theoretically determine a password's size by observing the time taken on a successful login, and then attempting verification themselves to find the size password which has an equivalent delay. This has not been applied in practice, probably due to the fact that (for normal passwords < 64 bytes), the contribution of the password size to the overall time taken is below the observable noise level when evesdropping on the timings of successful logins for a single user.

Source: https://passlib.readthedocs.io/en/stable/lib/passlib.hash.sha256_crypt.html

3.4 Cracking DES hashes

A very intuitive way to crack a hash is by doing a **brute force attack**. This would consist mainly in exhaustively trying combinations of passwords and salts until a match is achieved.

To implement a brute force attack for one-character long passwords, you can use the following code:

```
In [11]: ## This is a function that performs a brute force attack on DES hashe with
         ## In this scenario, you have the advantage of knowing that the password t
         ## Moreover, we will use the time module to calculate the time that the ma
         import time

         # 1. Declare strings "list_of_saltchars" and "list_of_passchars" (2 lines)
         list_of_saltchars = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
         list_of_passchars = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
         def bruteForce(hashtobreak):
         # 2. Iterate the list of salt chars using two fors so that you can test al
             for i in list_of_saltchars:
                 for j in list_of_saltchars:
                     salt = i+j
                     # 3. For all characters in list_of_passchars, hash with the sa
                     for k in list_of_passchars:
                         attempt = des_crypt.hash(k, salt=salt)
                         # 4. If there is a match, exit and return the charcter
                         if attempt == hashtobreak:
                             # 5. Return the broken password and the broken salt
                             return k, salt

         t = time.time()
         brokenhash, brokensalt = bruteForce(deshash3)
         print('The password is:', brokenhash)
         print('The salt is:', brokensalt)
         print('Elapsed time to break the hash: ', time.time() - t)

The password is: a
The salt is: 08
Elapsed time to break the hash:  0.5593645572662354
```

In a Windows 10 machine with 16 GB RAM, it takes around 0.55 seconds to break the password.

TASK: Modify the code to be able to crack hashes created with passwords more than one character long.

```
In [12]: # Use this cell to write your improved code
         import time

         # 1. Declare strings "list_of_saltchars" and "list_of_passchars" (2 lines)
         list_of_saltchars = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
         list_of_passchars = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
         def bruteForce(hashtobreak):
         # 2. Iterate the list of salt chars using two fors so that you can test al
             for i in list_of_saltchars:
                 for j in list_of_saltchars:
                     salt = i+j
                     # 3. For all characters in list_of_passchars, create a passwor
                     password_size = 0
                     while True: # This while increments the size of the password i
                         password_size+=1
                         indexes = [0]*password_size # create a list to store the i
                         while sum(indexes)<(len(list_of_passchars)-1)*password_siz
                             k = ''
                             for a in indexes: # This for creates the password base
                                 k = k + list_of_passchars[a]
                             attempt = des_crypt.hash(k, salt=salt)
                             if attempt == hashtobreak:
                                 return k, salt
                             # Increment the password vector
                             flag = False
                             for b, a in enumerate(indexes):
                                 if a<len(list_of_passchars)-1 and flag == False:
                                     indexes[b]+=1
                                     if b>0:
                                         for c in range(len(indexes[:b])):
                                             indexes[c]=0
                                     flag = True
                     k = ''
                     for a in indexes: # This for creates the password based on the
                         k = k + list_of_passchars[a]
                     attempt = des_crypt.hash(k, salt=salt)
                     if attempt == hashtobreak:
                         return k, salt

         ###### 1-DIGIT PASSWORD
         t = time.time()
         ## hash a new password
         deshash4 = des_crypt.hash('0', salt='00')
```

```
        ## crack the password
        brokenhash, brokensalt = bruteForce(deshash4)
        print('The password is:', brokenhash)
        print('The salt is:', brokensalt)
        print('Elapsed time to break the hash: ', time.time() - t)

        ###### 2-DIGIT PASSWORD
        t = time.time()
        ## hash a new password
        deshash4 = des_crypt.hash('00', salt='00')
        ## crack the password
        brokenhash, brokensalt = bruteForce(deshash4)
        print('The password is:', brokenhash)
        print('The salt is:', brokensalt)
        print('Elapsed time to break the hash: ', time.time() - t)

        ###### 3-DIGIT PASSWORD
        t = time.time()
        ## hash a new password
        deshash4 = des_crypt.hash('000', salt='00')
        ## crack the password
        brokenhash, brokensalt = bruteForce(deshash4)
        print('The password is:', brokenhash)
        print('The salt is:', brokensalt)
        print('Elapsed time to break the hash: ', time.time() - t)

        ###### 4-DIGIT PASSWORD
        t = time.time()
        ## hash a new password
        deshash4 = des_crypt.hash('0000', salt='00')
        ## crack the password
        brokenhash, brokensalt = bruteForce(deshash4)
        print('The password is:', brokenhash)
        print('The salt is:', brokensalt)
        print('Elapsed time to break the hash: ', time.time() - t)
```

```
The password is: 0
The salt is: 00
Elapsed time to break the hash:  0.002001047134399414
The password is: 00
The salt is: 00
Elapsed time to break the hash:  0.07004237174987793
The password is: 000
The salt is: 00
Elapsed time to break the hash:  4.823193311691284
The password is: 0000
The salt is: 00
Elapsed time to break the hash:  347.9849593639374
```

You can see there is a vast difference in time, particularly between cracking a 3-digit and a 4-digit password.