Chapter 1. THE UNIX PASSWORD BASED AUTHENTICATION SYSTEM

This chapter is centred on a simple but relevant subject: password based authentication (PBA). Regardless of the system, be it UNIX based, Windows, or even a remote system requiring PBA, e.g., on-line networks such as Facebook, LinkedIn, the paradigm is almost always the same: *the users enters a password which is verified against an encrypted version of the password that is stored locally on the system*. This encrypted version of the password is not always the result of applying an encryption function on the password, but rather applying some cryptographic one-way function (OWF). An OWF is a function that is easy to apply on the password but from which it is computationally infeasible to find the password, i.e., computing from input to output is easy while from output to input infeasible. Any cryptographic primitive can be used: hash functions, encryption functions, etc., since all these cryptographic primitives are OWFs. These functions will allow only for a random looking sequence to be stored in the password file, from which it should not be easy (or hopefully impossible) to guess the password of the user. Since usually hash functions (not encryption functions) are used for this purpose, we will refer to this encrypted value of the password as hashed password (note however that an encryption function such as DES or Blowfish can be used for the same purpose, in fact these are ready to use alternatives in most Linux distributions despite the more common use of MD5 or SHA2). If you are not yet familiar with hash functions, all that you should know for the moment is that they are OWFs that takes as input a string of any length and turns it into a value of fixed size, e.g., 128 bits in case of MD5, 160 bits in case of SHA1, 256 bits in case of SHA256, etc. that is usually referred as tag or hash.

The necessity for encrypting the passwords before storing them comes from the need of protecting one user from another (usually from admins or super-users) that can snitch on the password file (this is usually the case for super-users). Indeed this protection is not perfect, one can plant a key-logger and record all user input, install a Trojan that records activity at login, etc. However, if we assume that the system is clean from such malicious objects (and this is a reasonable assumption in many situations), then the best one could do is to read the file in which the passwords are stored. Consequently, encrypting the passwords is a good security decision.

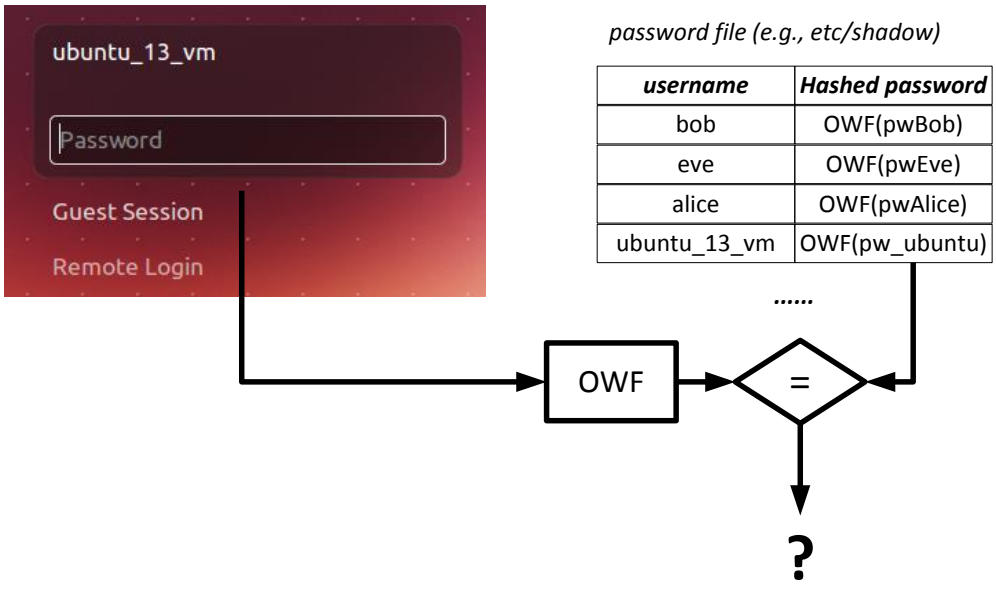

Figure 1. Password based authentication in Ubuntu

The way in which passwords are encrypted varies from one system to another, here we focus on how this is done under UNIX (and in particular the Ubuntu OS which we assume to be installed on your computer). The user authentication works in a straight-forward way: when the user enters his password at the login screen, the password is passed through a one-way function (the same which was used when it was stored) and the output is verified against the value stored in this passwords file. If the values are identical the users gains access, otherwise it is rejected (usually there is only a limited number of attempts and there is some delay after entering a wrong password in order to prevent attacks). This mechanism is suggested in Figure 1.

1.1 The passwd and shadow Files

Traditionally, in UNIX based operating systems the hashed passwords were stored in the file */etc/passwd* (a text file). On almost all recent distributions (including Ubuntu 13 which we assume to be deployed on your computer) the *passwd* file contains only some user related information while the hashed passwords are not here but in the */etc/shadow* file (also a text file, but with limited access, e.g., it cannot be accessed by regular users). This is done in order to increase security by disallowing regular users from reading it. The *passwd* file can be accessed by all users in read mode, however the *shadow* file is accessible only to super-users.

Adding users and passwords. To play a bit with the *password* and *shadow* files we first add some users, say *tom*, *alice* and *bob*. To add users use the command *sudo useradd –m username* (-m creates the home directory of the user) then to set the password use *sudo passwd username* (*sudo* allows you to run the *usearadd* and *passwd* commands with super-user privileges). If you need help on any of this commands use *man useradd* or *man passwd*.

```
ubuntu@ubuntu:~$ sudo useradd -m tom
ubuntu@ubuntu:~$ sudo passwd tom
Enter new UNIX password:
Retype new UNIX password:
```

Table 1. Creating a user named tom and setting his password

Accessing the shadow file. To access the *shadow* file you also need super-user privileges, for this, in the terminal run *sudo gedit* and open the file from *gedit*. If you successfully managed to create these accounts then the *passwd* and *shadow* files should look similar to what you can see in Tables 2 and 3 (note the user names and their hashed passwords).

```
ubuntu:x:1000:1000:ubuntu_13_vm,,,:/home/ubuntu:/bin/bash
tom:x:1001:1001::/home/tom:/bin/sh
alice:x:1002:1002::/home/alice:/bin/sh
bob:x:1003:1003::/home/bob:/bin/sh
```

Table 2. Example of *passwd* file with 4 users: ubuntu, tom, alice and bob

```
ubuntu:$1$js9ai3VX$iFbR5uTfv3JMmFCladdcn1:16459:0:99999:7:::
tom:$6$vIkXOyrz$CMiFB8meMfTANianaS7z5f8yMfplk/TtncZs/7b0es65XZSIyz3k
aiSwN/61sBdrPhT9B0RulJ9tWEnE7kpJC/:16470:0:99999:7:::
alice:$6$gpOJXcSy$AVrdUKBdSM8NlGmrbexoyetS2LhRgg3qkaTbZdMh4mj.Yps3
UxIkrtGDQfEGA.yNDhlIPG3m1hupX3b0I0Vs3.:16470:0:99999:7:::
bob:$6$5IPGOooA$J6rZ74NUpCVx9C2mpesKKr0iBjkHNCxz8Io3aPj5W6mwVKKv
nhWIbq0H91T9bDcWmDE3/6Ageoa3olcVe2nKY0:16470:0:99999:7:::
```

Table 3. Example of *shadow* file with 4 users: ubuntu, tom, alice and bob

Structure of the passwd and shadow files. In the *passwd* file, the first field is the user name, while the x indicates that the passwords are not here but in the *shadow*

file. Subsequently you can see the user identifier, group identifier, the user full name (and other potential information such as phone number, contact details, etc.), the home directory and the program that is started at login. The *shadow* file contains the information that is more relevant to us. Note the "$" sign in this file. Following the user name, in the *shadow* file, we have a *id* field which identifies the particular algorithm used to encrypt/hash the passwords. The following options are supported in your Ubuntu distribution:

- *1* - a version based on MD5 which is a hash function with 128 bit output that is no longer cryptographically secure (more details available in the lectures) but can still be somewhat safely used for this purpose,
- *$2a$* - Blowfish, a symmetric encryption algorithm, but not a usual option for this purpose,
- *5* - SHA-256 a hash function with 256 bit output,
- *6* - SHA-512 a hash function with 512 bit output which should give the maximum level of security.

After the algorithm identifier a random value *$salt$* follows. This value is called *salt* and is a randomly generated value, non-secret, that is used to prevent pre-computed attacks, i.e., you cannot compute the hash over a dictionary of passwords in an off-line manner since you do not know the salt and all your off-line computations will be of no use for a distinct salt value (it also prevents two users with the same password from having the same hash value in the *shadow* file). Finally, the *$hash$* value is the actual hash of the password. Other fields follow but not of much importance for this work: days since last change, days until change allow, days before change required, days warning for expiration, days before account inactive, days since epoch when account expires.

1.2 Verifying Passwords Programmatically

To generate the hash of a password, the *crypt()* function must be used. This function takes the password and the salt as character arrays, i.e., *char **, and returns a character array which is the hash of the password. The id in the salt dictates the particular algorithm that is to be used. This function can be called from any C/C++ program, but usually you will have to include *crypt.h* in order to work.

```
char *crypt(const char *key, const char *salt);
```

Table 4. The UNIX crypt function

Programs that use this function must be linked with the *–lcrypt* option, the sequence for compiling and running the program is in Table 5. Note that we assume the program test.cpp to be in the current directory and we specify the output file as *test* then run this file with *./test*.

```
ubuntu@ubuntu:/mnt/hgfs/VM_Shared$ g++ -o test test.cpp –lcrypt
ubuntu@ubuntu:/mnt/hgfs/VM_Shared$ ./test
```

Table 5. Compiling and running the program

1.3 EXHAUSTIVE SEARCH, A TRIVIAL ATTEMPT

Various programs for cracking passwords exist, but the purpose of this assignment is to help you in building your own. The program in Table 6 performs an exhaustive search for passwords of length at most MAX_LEN where the characters are chosen from a predefined set *char* charset.* How the code works should easily follow from the comments. The main idea is that we test each password that is generated by passing it through *crypt*, see *int check_password(char* pw, char* salt, char* hash)*. To generate all possible passwords from the predefined character set, i.e., *charset*, we take passwords of 1 character at the beginning and gradually apply to them each possible character, etc. All this is done inside *char* exhaustive_search(char* charset, char* salt, char* target)*.

```
#include <iostream>
#include <list>
#include <cstring>
#include <crypt.h>

using namespace std;

//this is an example line from the shadow file:
```

```
//$6$Iy/hHRfM$gC.Fw7CbqG.Qc9p9X59Tmo5uEHCf0ZAKCsPZuiYUKcejrsGu
ZtES1VQiusSTen0NRUPYN0v1z76PwX2G2.v1l1:15001:0:99999:7:::
// the salt and password values are extracted as

string target_salt = "$6$Iy/hHRfM$";
string target_pw_hash =
"$6$Iy/hHRfM$gC.Fw7CbqG.Qc9p9X59Tmo5uEHCf0ZAKCsPZui
YUKcejrsGuZtES1VQiusSTen0NRUPYN0v1z76PwX2G2.v1l1";

// define a null string which is returned in case of failure to find the password
char null[] = {'\0'};

// define the maximum length for the password to be searched
#define MAX_LEN 6

list<char*> pwlist;

// check if the pw and salt are matching the hash
int check_password(char* pw, char* salt, char* hash)
{
char* res = crypt(pw, salt);
cout << "password " << pw << "\n";
cout << "hashes to " << res << "\n";
for (int i = 0; i<strlen(hash); i++)
if (res[i]!=hash[i]) return 0;
cout << "match !!!" << "\n";
return 1;
}

// builds passwords from the given character set
// and verifies if they match the target
char* exhaustive_search(char* charset, char* salt, char* target)
{
char* current_password;
char* new_password;
int i, current_len;

// begin by adding each character as a potential 1 character password
for (i = 0; i<strlen(charset); i++){
      new_password = new char[2];
new_password[0] = charset[i];
```

```
new_password[1] = '\0';
pwlist.push_back(new_password);
}

while(true){

// test if queue is not empty and return null if so
if (pwlist.empty()) return null;

// get the current current_password from queue
current_password = pwlist.front();
current_len = strlen(current_password);

// check if current password is the target password, if yes return the
current_password
if (check_password(current_password, salt, target)) return
current_password;

// else generates new passwords from the current one by appending each
character from the charlist
// only if the current length is less than the maxlength
if(current_len < MAX_LEN){
for (i = 0; i < strlen(charset); i++){
        new_password = new char[current_len + 2];
        memcpy(new_password, current_password, current_len);
        new_password[current_len] = charset[i];
        new_password[current_len+1] = '\0';
        pwlist.push_back(new_password);
}
}
// now remove the front element as it didn't match the password
pwlist.pop_front();
}
}

main()
{
 char* salt;
 char* target;
 char* password;
 // define the character set from which the password will be built
```

```
 char charset[] = {'b', 'o', 'g', 'd', 'a', 'n', '\0'};
 //convert the salt from string to char*
 salt = new char[target_salt.length()+1];
 copy(target_salt.begin(), target_salt.end(), salt);
 //convert the hash from string to char*
 target = new char[target_pw_hash.length()+1];
 copy(target_pw_hash.begin(), target_pw_hash.end(), target);
 //start the search
 password = exhaustive_search(charset, salt, target);
 if (strlen(password)!= 0) cout << "Password successfuly recovered: " <<
password << " \n";
 else cout << "Failure to find password, try distinct character set of size \n";
}
```

Table 5. An exhaustive search algorithm for finding the password.

1.4 EXERCISES

1. Consider passwords of 20 characters and that they are hashed through MD5 which outputs 128 bits. How many passwords of 20 characters are there for a single 128 bit output? How many users should be expected until a collision occurs with probability ½ ? (note that since hash functions are collision resistant, it is actually computationally infeasible to find such passwords, but it is good to understand that they do exist)

2. Find the password that corresponds to the following shadows entry, having in mind that the character set is {a, b, c, 1, 2, !, @, #} and the non-alphanumerical symbols occur only at the end of the password

tom:6SvT3dVpN$lwb3GViLl0J0ntNk5BAWe2WtkbjSBMXtSkDCtZUkVhVPiz5
X37WflWL4k3ZUusdoyh7IOUlSXE1jUHxIrg29p.:16471:0:99999:7:::

3. Consider a 14 character password that ranges over all possible ASCII symbols. On your current computer, how much time will you need to break such a password?

4. Consider the same context as previously, but this time we are concerned with memory usage. Could you provide a rough estimation of the amount of memory that is used to break the password in the previous example? Can you implement a solution that improves on this amount?

5. The following *shadow* entry was generated by a password formed by an arbitrary arrangement of the following words: *red*, *green*, *blue*, *orange*, *pink*. Find the password.

tom:$6$9kfonWC7$gzqmM9xD7V3zzZDo.3Fb5mAdM0GbIR2DYTtjYpcGkXVWat TC0pa/XVvKTXLb1ZP0NG9cinGRZF7gPLdhJsHDM/:16471:0:99999:7:::

6. Now a more demanding exercise. All of the following passwords start with)):@$*!:((and the rules defined below for each user apply only for the predefined character set:

$$Alpha = \{ a, b, c \ldots, x, y, z \}$$ [1]

$$Num = \{0, 1, 2, \ldots, 9\}$$

$$Sym = \{!,@,\#,\$,\%,\^,\&,*,(,)\}$$

   a. ***tom_easy*** has a password from all characters in *Alpha*, *Num* and *Sym*, which gives a total of: 26 letters, 10 numbers and 10 symbols, summing up to 46 characters. The password contains at most 4 such characters, i.e., 46^4 = 4,477,456. [2]
   b. ***tom_harder*** has a password constructed from the same set *Alpha x Num x Sym* except that after the starting characters ")):@$*!:((" it has an additional number from 1..10. Suggestion: to solve this, you may consider running 10 instances of the previous program with passwords starting with ")):@$*!:((1", ")):@$*!:((2", ")):@$*!:((3", ")):@$*!:((4" and ")):@$*!:((5",

[1] Note that there are no upper-case letters

[2] You should be able to crack this in ~12 hours (assuming that your computer can perform 5x10^6 passwords/day, check the exact running time with the *time* command)

etc. Since only the first solver gets the points, you may consider running these on distinct computers.

c. ***tom_split*** – has the first 4 characters from *Alpha* and the last 2 from *Num* & *Sym*. Suggestion, you should search separately for the first 4 and last 2 chars.
d. ***tom_wordy*** – has a concatenation in some random order of the following 8 words {the, big, brown, fox, or, small, grey, elephant}. Words may repeat but there are only 8 words.
e. ***tom_wordy_harder*** – has a concatenation in some random order of the following 10 words {the, big, brown, fox, or, small, gray, elephant, yesterday, today}. Words do not repeat.
f. ***tom_math*** – has a password of the form ")):@$*!:(($N_1N_2N_3N_4$" where N_i is a number generated $N_i = N_{i-1} +$ *seed* **mod** 255 where N_0 and *seed* are random values in {0, 255}. The numbers are written as characters, i.e., if N1 = 234 then password is ")):@$*!:((234 … "
g. ***tom_more_math*** – same as previously but the operation is performed modulo 4096, as well as the seed and N_0 and *seed* are random values in {0, 4096}

Note: remember passwords start with:)):@$*!:((

tom_easy:6JcQryNT4$Nydvd9w9kpwwkTTU93uMulS9noTyiLmheUnyNrVaNoVjAyyFAAXAXP1.EePMdYlohOyVAxcuplfZMQD7VixY7/:16497:0:99999:7:::

tom_hard:6tamx8Uvr$tMfa8QsdrJnDa6n40tVVy7kRaFbbgevr4rFz/rFNDTmaUcKnZiiBSGVkO/uS1/M513Z0BVuBELrhDrwr9EJRY0:16497:0:99999:7:::

tom_split:6Z9VfBmUG$MhG7XlzZnBxdgRjDf1utb7fJZSc8hvzPJhCjcBd.lN.HoMvsGT1.wn0ACl.AydYq5oVw9uFCTtpH4oOa1s/bT1:16497:0:99999:7:::

tom_wordy:6GHuikUus$T8/C1Ed6QLBkHhMWJB/nFPgY/tujpMqkpy8tmG.ovijy960HrWSkPQWU8h062SuR/NIDIJhCbszMIycdILr7p1:16497:0:99999:7:::

tom_wordy_harder:6p0sCvjGG$ZH9..sdjHFaWux9lgVWm44USpVawFheB8I4PJcA7ep9nj6ISwcCbO7/SvuTS9LdreyMO./zFPKyE06zR5G/Bw1:16497:0:99999:7:::

```
tom_math:$6$SMF7niTS$HuLhlRyIAnhLhNRtqqd/OSkye3fEsnd9i2trxx53Mji/hYZQ8
ywnIiUMa6hgSax/SOeCYTootE649Zzblt4Fq1:16497:0:99999:7:::

tom_math_harder:$6$/agnX0ga$kY0EejIuThPUH/DeTYJZIAPzxMA3WXYZjHOF/YKQ
a6jEM9lHNAKt9fRyVWGntpG/BPH3sZCZkKmFCHx1lZX8k0:16497:0:99999:7:::
```

Remarks. To view memory usage use the command *free –m*, the free command displays the amount of free and used memory, *-m* displays this in megabytes. If you want to repeat it each second use *watch –n 1 free –m*, the watch command executes periodically what follows, in this case *–n* means that repetition time is given in seconds. To get the running time of a program use the *time* command, e.g., *time ./test*.