Kotsomitopoulos Aristotelis
Project #3 ΥΣ13 ΕΑΡΙΝΟ 2014 (computer system security)

I want to hack a password of **6** chars, those chars must have been created from the given alphabet.

Alphabet:(64)
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@"
so a password example can be "AriS1@".

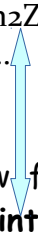
In order to hack this password i will use **rainbow** tables!

passwords are stored as the output of a hash function. Hashes are one-way operations. Even if an attacker gained access to the hashed version of your password, it's not possible to reconstitute the password from the hash value alone but its possible to attack the hashed value of your password using rainbow tables.

We have 64^6 different password = 68.719.476.736 = 69 billion different passes

First of all i create for example 1.000.000 different passwords 6 char long and this will be the start of my chain something like :

!!!7Bo
!!!8o!
QBLfQU
VM7Ucl
eORyLL
lXO8rC
t9p48R
8Qi!cm
8QjgAU
I3m2Zn
.....
...



now for every plaintext of the above we will create a chain
plaintext ->
hash = blake(plaintext) ->
new_plaintext = my_reduction_function(**hash**) ->
hash = blake(**new_plaintext**) -> repeat.....->.....->....
in my program i create a chain 10.000 long so for every plaintext i use

blake+my_reduction_function (10000 times).
Finally i will have almost 1.000.000 x 10000 = 10 billion different values *(without collisions) so if we had 1 password given i have 1/7 possibilities to hack it but now i have more than 15 password given so that size is enough with a bit of luck...*

to generate that size of rainbow table took more than 16 hours... in my ophinion the best size for that problem would be 5 million x 10.000 but that need alot lot lot of time to be generated even with threads and multicore processor!

So after that huge generating i place the results in a .txt file an my search-find password program will use that table to hack the password.. the file will look like this :

output_1mX10000.txt

```
!!!7B0    77240e13ebff7ea020e62d90975cd17963fa818bec3f55836e2fcd4780ea48d9(1)
!!!80!   f55715be2179b0fb9d3f83bdb39fed8374afcbaea9213e4f2001a0b8734a0765(2)
!!!80Z   3410c7d80d27f5bd489f5dd5c0533043c6ce17c363f8ab7acf7b91511df5ac96(3)
!!!YI2   1657ca495a203164edc363700c86adbe54374edc05cf474e70030a842f76da68(4)
!!!oC@   0f1e9b44a2441c46f0c525f5c22777c4f009ec1b0f5776604b42639cf230648f(5)
!!!uZW   d2953ba6e7074c76c0461eb8ec7d7c36bcb434001284abba8d0c92d6bd892f7a(6)
!!07fO   5b9ef9c41b5444476e2c66e050c795031185f42bfc40bf8f174f1e0e3bb8f581(7)
!!0JB0   7a73cb5357eb35ecfc82ebc351112824c54bdaa70603fe5262e21f22fd13cd60(8)
!!0R7U   6ffdb183126ca9e70b659ea45f4dcb0ca78c7136f72fa733bb960d315eca19a0(9)
!!0XJ6   85b0c65a867e08b6ae7d7d65ddae3c6e4def38ad805cbb96b75a49884928852b(10)
!!0fNM   790fe297f6ca448cdd97e083176ad3e492258ea2167f9048119da9dcbfaa12d9(11)
!!0kyY   5498173ef57774579a8095eb6d32a2bee8cac4bf9153ddda011ab33d600d6257(12)
!!0rik   73288f5ab7e7fc5b1d2a387dacc6401b24c422494d5d4f6ad280e4d046b83327(13)
!!1wh0   98f12f574918ff51e92369c38d9369d474b2f2bf0291d79e6bb3be71ed7abd2d(14)
.......
..............
.......................
.........................
zzxEpx   f9042996c416d9f32395b0bb967aca9df277f4e8dcd14794bc60a820aa5d286f(999988)
zzxMxc   c1409a47dab4ce96a013a7033c868d56888a748cf670c28b86c1413c6801fd92(999989)
zzxpFE   e6dba8c5b25946070e7f680e2255a08aba46ca92fdf2e818f245ad01fb8fba03(999990)
zzxvEY   f028262e61fee77d2104ef928d195d6b401fdb50522bdacad088ab0df197a756(999991)
zzyw9@   0861fc88106fffb2e9731b5a6b67b5ffc891d19d2b8b8334c57a97ce50bf3e0a(999992)
zzz4sv   bf6aef7de87249ecaee1bb8c83b1025d3c238226f1ce47999b1dbc24fd2bab6e(999993)
zzzAuV   dd6e7cfa2031148cf1c22c1756bd7f5a0cb8c96e581e87cf1fb3dada8963cfee(999994)
zzzGGs   8872e669d2284297bbfe4494ced287de58ff0b60c66d3a1b46a8b7657cc1188f(999995)
zzzRqC   d5859daaf422a4c73dafe382292c5e1c9dba17361d9176663658a8a48b9e3a08(999996)
zzzTe5   946a862717e78677d6eb96b25f952cdee0a91de870789b129abc1d8e4ff15eca(999997)
zzzZ67   9750c53137b46208040c93b08cc7ed1454a4fe0e1583511153291bcd0e40b393(999998)
zzzgB3   4a5af0f0a7c1b61137bcd1143d447a05c8ada3d63eade369286e807f0b9c554a(999999)
zzzyqE   a36a2403b3adb9c50094866613fd2e956f500868cbc2a6ee9a99ba8ec8c41b79(1000000)
```

in order to reduce collisions i tried a lot of different reduction functions with a lot of tests , in my final function in order to produce 6 char long random passwords from a given 64 char long hash :
i map every char from the hash(16 different chars *0123456789abcdef*) with my alphabet "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@"
i add the first 10 hash chars (ex. 0Fa2.. = 0+15+10+2..) and then i make a XOR with the next 10 in that way i produce a number than number MOD 64 give me a number inside my alphabet for example 63 is '@'! and for the 6th char i use the last 4 characters + 6 first in that way i produce a deterministic random string (plaintext) for my chain!

```
string plaintext_from_hash3(string my_hash){
```

```
string charset = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@";

    string result;
    int sum = 0;
    int counter=0;
        /* this is for our last XOR 50-60 with the last 4 chars + the 6 firsts! */
    int last_digits = my_hash[60]+my_hash[61]+my_hash[62]+my_hash[63];
    for(int i=0; i<6; i++)
        last_digits += my_hash[i];
        /* this is for our 5 XOR */
    int for_XOR[6];
    for(int i=0; i<6; i++){
            sum=0;
        for(int j=0; j<10; j++){
            if(my_hash[counter] == '0')
                sum += 0;
            else if(my_hash[counter] == '1')
                sum += 1;
            else if(my_hash[counter] == '2')
                sum += 2;
            else if(my_hash[counter] == '3')
                sum += 3;
            else if(my_hash[counter] == '4')
                sum += 4;
            else if(my_hash[counter] == '5')
                sum += 5;
            else if(my_hash[counter] == '6')
                sum += 6;
            else if(my_hash[counter] == '7')
                sum += 7;
            else if(my_hash[counter] == '8')
                sum += 8;
            else if(my_hash[counter] == '9')
                sum += 9;
            else if(my_hash[counter] == 'a')
                sum += 10;
            else if(my_hash[counter] == 'b')
                sum += 11;
            else if(my_hash[counter] == 'c')
                sum += 12;
            else if(my_hash[counter] == 'd')
                sum += 13;
            else if(my_hash[counter] == 'e')
                sum += 14;
            else if(my_hash[counter] == 'f')
                sum += 15;
            counter++;
        }
        for_XOR[i] = sum;
    }
    int res;
    res = for_XOR[0] ^ for_XOR[1];      //1st XOR
    if(res<0)
        res=-res;
    result+=charset[res%64];
    res = for_XOR[1] ^ for_XOR[2];      //2nd XOR
    if(res<0)
        res=-res;
    result+=charset[res%64];
    res = for_XOR[2] ^ for_XOR[3];      //3rd XOR
    if(res<0)
        res=-res;
    result+=charset[res%64];
    res = for_XOR[3] ^ for_XOR[4];      //4th XOR
    if(res<0)
        res=-res;
    result+=charset[res%64];
    res = for_XOR[4] ^ for_XOR[5];      //5th XOR
    if(res<0)
        res=-res;
    result+=charset[res%64];
    res = for_XOR[5] ^ last_digits;      //6th XOR
    if(res<0)
        res=-res;
    result+=charset[res%64];

    return result;

}
```

Lets assume for example after we use **dbus-monitor** in sbox.di.uoa.gr we get the

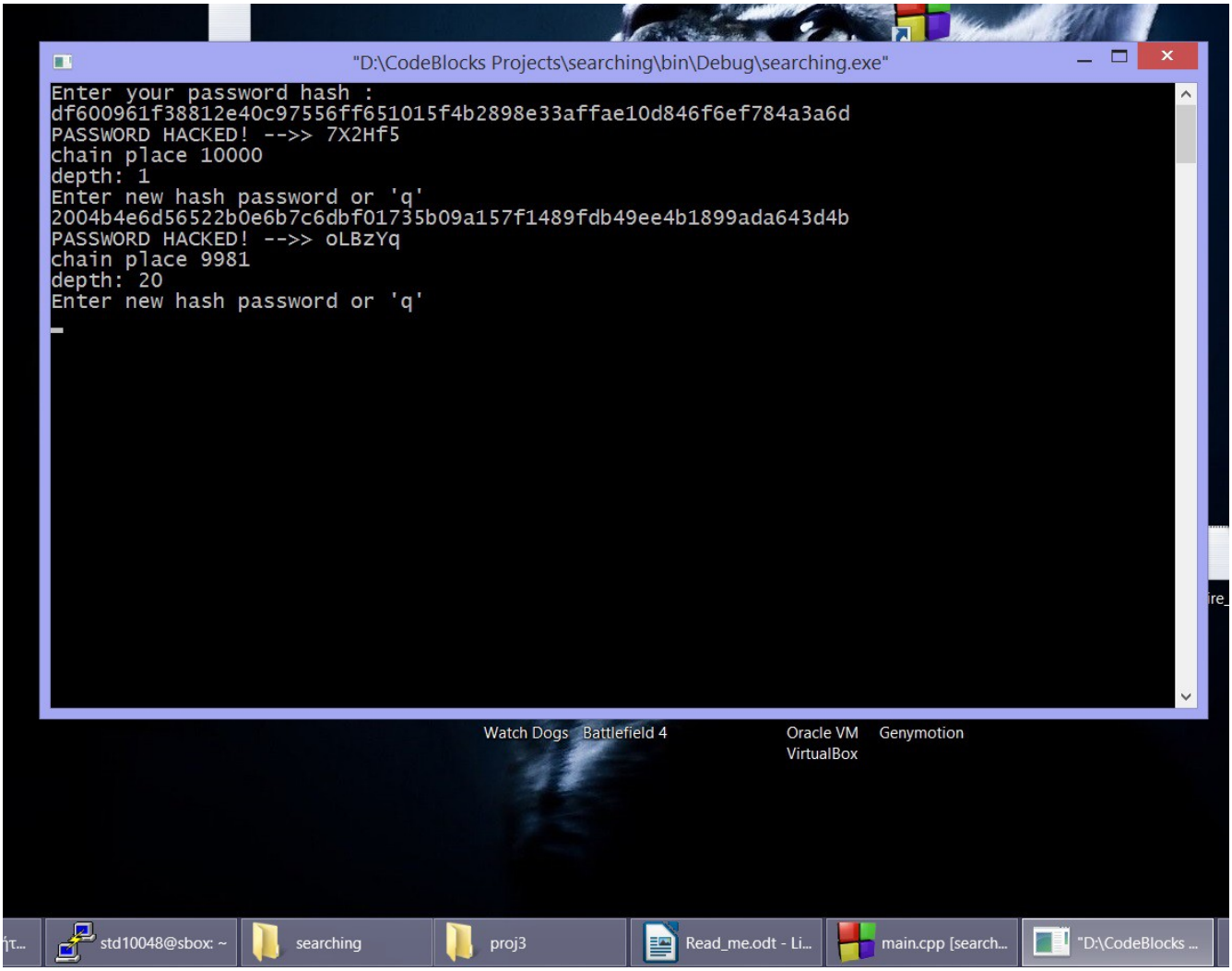
following hash :
9b465fce3fd32aa5531f801d4c63f31380adc84c62f0d170aaf535619b85a928

so we will search if the above hash is in our table IF WE FIND IT we will use the chains started plaintext to obtain the password we will use blake->reduction->blake->reduction.. until we find our first hash in our case it would be the last one so the **plaintext** that produced this hash would be our hacked password !.. if we dont match the above hash with our **rainbow** table hashes then we will use reduction function to our hash then blake and then we will try to find it again if we find it then we will search again the correct chain until we find the **FIRST** hash(not the new one)..if we fail again we will continue hashing and reducing our given hash until we found a chain match in rainbow table.. if we use in my situation 10.000 loops and dont find a match then we failed and that password cant be hacks with my table..now if we find a match but after expanding the appropriate rainbow table 10.000 and we dont find our **FIRST** hash then we dont stop we continue exapning and searching for another chain MATCH!! if we check every possible chain and the **FIRST** isnt there then the password cannot be hacked with that rainbow table..
if we had used a bigger one it would be almost 100% possibility our password would be hacked!

After we hack the password we use nc localhost 4433 and we enter the password! We have only 10 seconds to copy hash + paste it to our program + use the above command + paste the result so its a bit hard.. my search function can make an average of 7 second to find the result so its kinda HARD!!!

The exercise was developed using C++ in windows (CodeBlocks) and tested in linux ubuntu all my .cpp and .h would are included in my ZIP! My rainbow table has size 100MB so i cant include it!

So here is an example of hacking a password from hash!
with my search program..
but again my rainbow table is 10 billion without collisions so its a bit small.


"D:\CodeBlocks Projects\searching\bin\Debug\searching.exe"
Enter your password hash :
df600961f38812e40c97556ff651015f4b2898e33affae10d846f6ef784a3a6d
PASSWORD HACKED! -->> 7X2Hf5
chain place 10000
depth: 1
Enter new hash password or 'q'
2004b4e6d56522b0e6b7c6dbf01735b09a157f1489fdb49ee4b1899ada643d4b
PASSWORD HACKED! -->> oLBzYq
chain place 9981
depth: 20
Enter new hash password or 'q'
Watch Dogs
Battlefield 4
Oracle VM VirtualBox
Genymotion
std10048@sbox: ~
searching
proj3
Read_me.odt - Li...
main.cpp [search...
"D:\CodeBlocks ...