Παπαπαναγιωτάκης-Μπουσύ Ιάσων

ΑΜ: 1115201000201

ΥΣ13 ΕΑΡΙΝΟ 2014

Project #2

Extra μέρες: 0

Rainbow Table attack on Blake hash function

First I studied what a rainbow table is, how it is defined by the reduction function and the hash function used and why it's granting the attacker a way to vastly reduce the size of the tables compared to a dictionary attack (time-space tradeoff).

Source1, source2, source3, source4.

The idea is fairly simple and provide an easy way for an attacker who have hashed passwords to get the original plaintext password.

Advantages of this technique include:

1. The much smaller file size over a dictionary attack.
2. Because the construction of the table is done offline it greatly reduce the online time required compared to brute force attacks.

Disadvantages are:

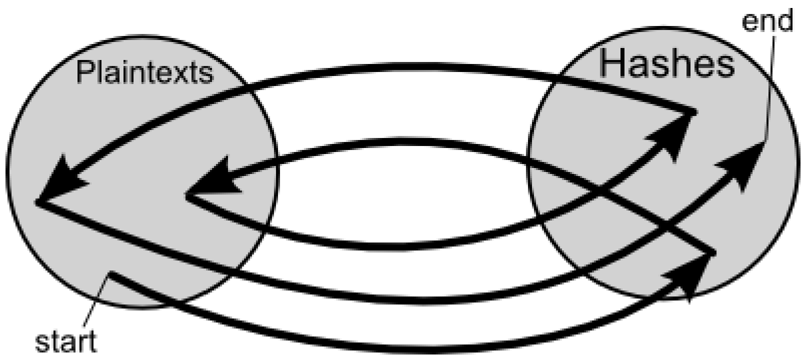
1. The attacker must have the hash of the password he want to "break".
2. It is exponentially harder if there is a salt applied to each password because for each possible value of the salt the attacker must generate a new rainbow table.

About the hash function Blake

Blake is one of the finalist functions submitted to the SHA-3 competition by Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. For more information click here.

The magic function "R"

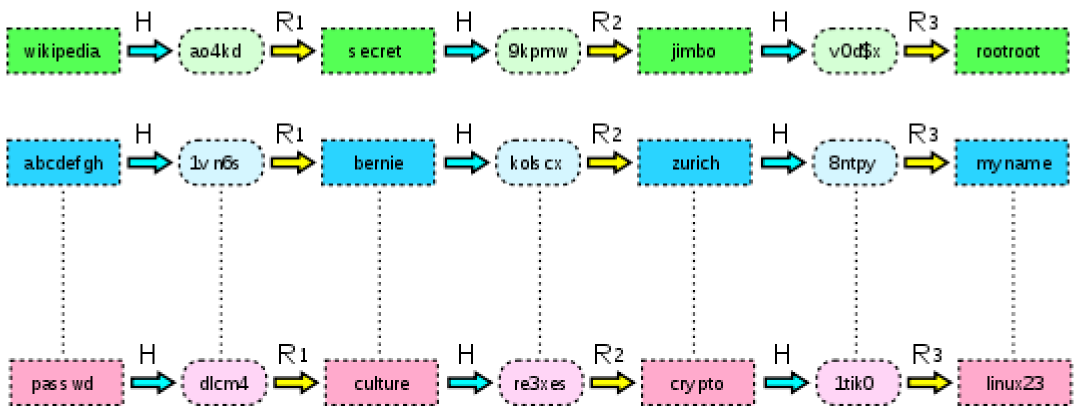
The reduction function is the most important part of a rainbow table. It takes hashes and outputs possible passwords within a given set of characters. A good rainbow table is one where there are few duplicates of plaintext which is equivalent to the reduction function to be close to collision-less.



Rainbow tables are called like that because it is common to apply more than one reduction function in order to achieve better results. By having a unique reduction function for each step of the generated chains if there is a collision (of the reduction function) that happens between to plaintext that are on different columns the chains wont merge.

The usual technique is to have a family of reduction functions defined by a function and the place in the chain it is applied. i.e.

R1() = (f(),1), R2() = (f(),2) etc.



Cracking with Rainbow Tables

The algorithm to look for a plaintext password **P** given the plaintext's hash **H** in a rainbow table **RT(h,p)** with chain length **l** is quite simple.

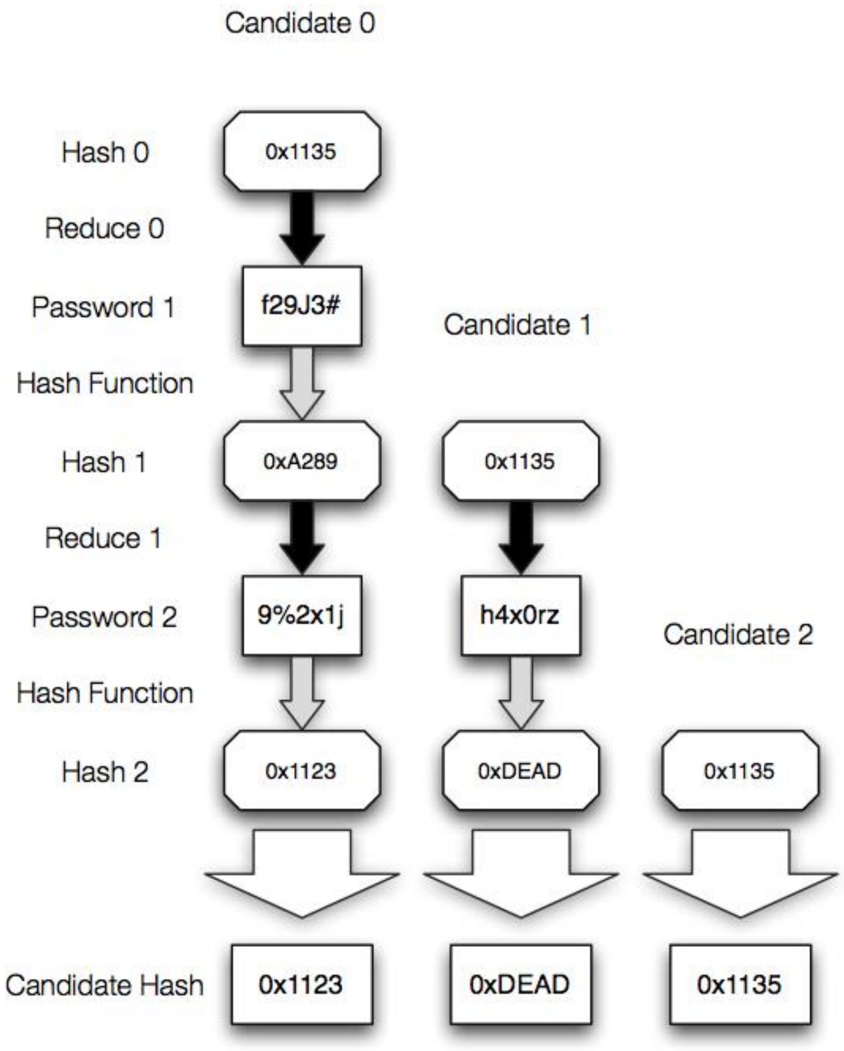
1. If **H** is in **RT** then run the chain from **p** (**l-1**) times to find the plaintext password.
2. Else if hash(R(**H**)) is in **RT** run the chain from **p** (**l-2**) times to find the plaintext password.
3. Else if R(hash(hash(R(**H**)))) is in **RT** run the chain from **p** (**l-3**) times to find the plaintext password.

   .

   .

   .
4. Do that at most **l-1** times, if the password is still not found then it wasn't in the table.

If the reduction function is different for each step of the chain we have to take that into account also.



Coding for the Rainbow Table

- C# -

I started by coding with C# as it is an easy to use programming language witch is flexible between high level programming and lower level access. I found this implementation of the Blake hash function to use but the more I coded the more problems I had with formatting between the Blake's output and the program's output to the terminal. As it was still early I decided to go for an even easier way.

Coding for the Rainbow Table

- Python –

As a very high level programming language, Python helped me a lot to focus on the essence of the rainbow table creation as I didn't have to bother with encoding details. The code to generate Rainbow Tables was very small.

I then started to create my own Rainbow Tables.

Given that the passwords I was trying to crack were 6-char long with characters a-zA-Z0-9!@ (64 different characters) the number of possible different passwords are just over 68 billion. To start, I needed at least a rainbow table with 1 billion entries so that after taking into account the collisions I would have a chance of 1% approximately to find the password. This is a very low probability but at least a start.

Sadly is soon became obvious that python was very slow. I took 10 minutes on my computer to generate a table of 1000*1000. After that disappointing result my first thought was to make it parallel so I found a way to create threads in Python and run the code again only to see it even slower. It appears that the because the Python interpreter is running on a single hardware thread (one logical core) the threads generated within the Python program are all running alternately on the same core so there isn't any speedup.

I came back to the original program for one thread and had the idea to run multiple instance of it at the same time. That way I could take advantage of my hardware to accelerate the creation of the rainbow table. In order to avoid duplicate original passwords I first generated the unique passwords for each chain to start and then gave 200.000 passwords to each of the six instances of the program. That would give me a table of 1.200.000*1.000 (chain length set to 1.000). After more than 24 hours I decided to stop it. Python was proven way too slow for this task. The code is on the folder "Python".

Coding for the Rainbow Table

- C++ -

With a good understanding of the algorithm I started to write C++ code for the rainbow table generation. I used a `set<string>` to store passwords already generated at the start of a chain and then a `map<string,string>` to store the password and final hash of each chain. When the map is filled its content is then written to an output file.

Generating random passwords:

Each row have to start with a different password, if two rows have the same starting password they will collide and take time to compute for no reason.

In order to make sure a start plaintext-password wasn't already used I stored all of them on a `set`.

My first attempt to create random password was with the C function rand(). Given an array with all the allowed characters (of length 64), for each character I selected the output of this:

```
char newChar = Array[rand()%64];
```

This work for small numbers but I soon hitted the limits. Rand() is a pseudo random number generator but it have low entropy on lower bits as I found here. So instead I used the proposed method:

```
char newChar = Array[rand()/(RAND_MAX / 64)];
```

Profiling the code:

I used Microsoft Visual Studio to analyze the performance of the code and found that over 90% of the time was spent within the Blake hash function. This is expected as hashing is an expensive operation. This also meant that no matter how optimized my code would be I could only affect the 10% of the time.

Choosing a good Reduction function:

As mentioned above, it is critical to choose a good reduction function.

My first thought was to create a different R function for each row. This approach can't work because when searching a hash in the table I would have to apply to the input hash every reduction function for each step of the search.

After that I tried having one reduction function for each column of the table witch is a common practice. So I had a basic reduction function with a salt, the number of the column. The basic idea is to have something random enough in order to avoid collisions. Using Blake inside the reduction function was one idea but it would almost double the time needed so instead I used the hash given to the reduction function. Blake is good hash function with a very good distribution so I used it to get 6 indexes from 0 to 63 and then assigned to each letter of the new password a char from my char-dictionary array.

```
index = HashValue[i] + HashValue[i+6] + HashValue[i+12] +
HashValue[i+18] + HashValue[i+24] + salt;
```

```
PreHashValue[i] = charset[index%64];
```

In order to make it more "random" I used several bytes of the hash for each character generated plus the salt.

With this reduction function I was able to have around 6% collisions. i.e. a table 1.000.000*1.000 ended up 937.000*1.000.

Password search:

I implemented the simple algorithm explained above for searching the plaintext of a given hash.

Execution:

I started by collecting some hashed from the dbus-monitor on the sbox.

Then I created a rainbow table 10.000.000*8.000 which ended up 6.500.000*8.000 due to collisions. As expected the more rows I ask for and the longer those chains are, there are more collisions. This took around 6:30 hours and ended up on a file 470 MB.

Finally I searched each hash I had collected on the rainbow table. The search for each hash took up to 2 minutes. Two minutes when the hash could not be found and from 0 to 2 for each hash I could crack. At first I thought those times was acceptable but I turns out there was a requirement to crack the hashes in under 15 seconds.

Optimizations:

My code was already optimized and compiled with -O3 for full speed and as mentioned before was only 10% of the execution time, the rest being taken by the hash function Blake.

So I coded the search to be done with threads, all of them working together. I chose the pthread library as it is cross platform and I had some experience with it. The number of threads was set to 8 to fully take advantage of my hardware.

The search algorithm didn't have to change. Explanation:

H is the input hash

R(x,y) is the reduction function applied to the hash x with salt y (salt is the step of the chain)

B(p) is the Blake function applied on a plaintext p

N is the length of the chains

P is the plaintext of H

SC(a,b) search on the chain with final hash a for the plaintext of the hash b

map<h,p> is a map with all the final hashes h and their starting passwords p

```
Thread i:

for( j = N - 1 - i; j >=0; j+=8 )

      for ( z = 0; z < j; z++)

            p = R(H,N-1)

            h = B(p)

      if( h in map )

            SC(h,H)
```

This fully utilize 8 threads without any overlap or blocking conditions. It gave a great boost to search times that now took no longer that 10-15 seconds. Also if a thread find the password it immediately prints it so I don't have to wait for all of them to finish.

Having done that I went a step further and parallelized the rainbow table creation. If M are the rows requested each thread creates M/8 rows (some locking required for critical sections).

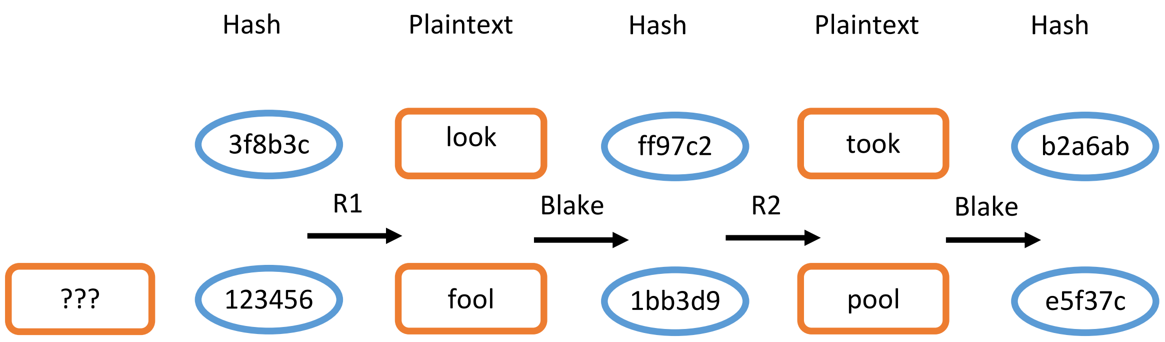
With the new parallel table creator it took my computer 2:30 hours to create a rainbow table 15.000.000*8.000 that ended up 8.000.000*8.000 due to collisions (file size 578MB). The final C++ code is on the "C++" folder.

Search complexity:

In order to understand why it took two minutes at first for a complete search I had to find its complexity.

On an ideal rainbow table (magic reduction function) it should take no more than $\frac{N(N-1)}{2}$ steps, each step have one call of the reduction function and one call of the Blake hash function.

The problem is that the reduction function in practice have collisions, for two different hashes it gives the same plaintext.



Input hash **H** is "123456"

H is not on the final hashes so we apply R2.

If R2(H) = "took" and Blake(took) = "b2a6ab" and this hash exists on the final hashes then according to the searching algorithm the plaintext we are looking for is on that chain, so we have to run the chain and stop just a step before (look/ff97c2) but now we see that ff97c2 ≠ **H** so "look" is not the plaintext we were looking for.

Each time the search algorithm find a collision like this one it have to run the chain. The more collisions our reduction function have the more chains will need to be unfolded.

This is the hidden complexity of the search algorithm. I didn't find a way to calculate it exactly but it can be done by running many searches on a rainbow table and count the steps it takes to find the right password.

Code structure:

I wrote a C++ class named RainbowTable and this the public API:

1) RainbowTable(vertical size, password length, chain length) the constructor of the class initialize some internal variables.
2) CreateTable() creates the rainbow table with size predefined by the constructor.
3) Print() prints the rainbow table, the first plaintext and the last hash.
4) Get_Size() returns the size of the table witch is equal to the number of rows it have.
5) Find_Password(hash) search for the plaintext of the input hash value.
6) Find_Password_Parallel(hash) open 8 threads to search for the plaintext of the input hash value in parallel.
7) Save_to_file(filename) save the Rainbow table generated (or loaded) to a file with the given name.
8) Load_from_file(filename) load a rainbow table from a file.

I used those functions to each time create the main I needed, to be complete the code needs a command line interface offering all those options plus some modifications to let the user choose the charset used for the plaintexts and the hash function.

Attempts to crack the hashes and get the secret:

After my first attempts where the search was taking around 2min max with the help of the find_password_parallel() and the same rainbow table 6,5millions * 8 thousands the search time was up to 15 seconds. When a password was found it could be just 7-8 seconds. After trying for 5 days to get the secret I still couldn’t beat the 15 seconds because some time is needed to copy the hash from one window to the other an then write the plaintext found back to the first window. During those five days I tried every night and almost every morning. Each time I tried to crack 10 hashes success rate of 10% but I always was late 2-3 seconds. Each time I verified I had found the right plaintext by hashing it on the python command line to make sure the hashed plaintext was equal to the hash served on the dbus-monitor.

In order to speed up things I created a table “taller” and “narrower” (31millions*1,6thousands). It took 1:35 hours to create and ended up on a file 2,2GB. This time searching was under one second so I tried and cracked the hash.

The secret:

I successfully found the plaintext corresponding to this hash:

87481c4c3c2208a21285cfde5d87c1fb4be6d3ee37e1dc0de9b429add37db051

Plaintext: **L40WHj**

And then typed it to the service to get the secret.

```
sdi1000201@sbox:~$ nc localhost 4433

Type the password:

L40WHj

W3ll d0n3!

-------------------------------------------------------

SERIAL:20140627230330-6de16ed727515bc78e6bd8a8d76cc21f

-------------------------------------------------------
```

Probably the ideal size where somewhere in between the 6,5millions*8thousands (just 1-2 seconds too slow) and the 31millions*1,6thousands ("too fast").

Execution variables:

All test were made on a windows 8 PC, CPU i7 4770k and 16 GB of RAM. Compiler: g++ with -O3 optimization.