Software Project, spring 2011 - Assignment 3

Due by 02.08.2011, 23:59

In this assignment you will implement an efficient attack against a commonly used password storage mechanism called *hashed-passwords*. This attack, called "rainbow-table", is a generalization of the exhaustive search, but with a different time-space trade-off. Note that the attack is a practical one: For example, WindowsME authentication was broken using a similar attack.

Its weight is 25% of your final course grade.

Outline

A rainbow-table based attack for breaking hashed passwords, consisting of two executables: "**create_rainbow_table"** and **"crack_using_rainbow_table"**.

1. A Rainbow-based attack

1.1. Background – Rainbow tables

Motivation:
The main problem with the cracking algorithm implemented in exercise 2 is the huge files created.

- It commonly exceeds ordinary disk space with relatively simple rules (i.e. rules that define a relatively small set of possible passwords).
- While pre-processing time is mostly spent on disk accessing, query time can be as fast as few milliseconds – much faster than what is needed.
- *Thus, to optimize the overall running time, these costs should be balanced –An algorithm doing so is called "Rainbow Tables".*

This section will give background information – same as taught in class. It consists of three parts:

- Presenting the concept of pre-computed hash-chains which handle the above problem.
- Presenting Rainbow-Tables that are a specially optimized variant of pre-computed hash chains.
- Pseudo-code of rainbow-table pre-processing & query. Note that the data structure is the same as in exercise 2 (DEHT), but there are several DEHT queries per single crack.

Pre-computed hash chains

"Hash chains" is a technique for decreasing disk storage requirement. Denote the set of possible ascii passwords (i.e. passwords that fit rule) as ***S***. For example, rule "**#4**" (i.e up to 4 digits) yields ***S*** of size 11,111. The idea is to create some *reduction function* ***R*** which maps "back" from the hashed-password domain into ***S***. Note that ***R*** is not an inverse of the cryptographic-hash ***H*** - such inverse cannot be calculated. R will distribute its range as evenly as possible over all of ***S***. Calculating R must be deterministic. A pseudo-code for a large family of possible R's is described below. Now, by alternating calls once for the hash function and once for the reduction function, and repeating these calls many times, *chains* of alternating passwords and hash values are formed. The chain length is a predetermined parameter. For example, if chain-length is **2** (that is, two reduction functions are used to build chain), rule is "**.6**" (i.e. up to 6 alphabetic lower case English letters) and the cryptographic-hash values are ***32*** bits long (i.e. 8 hexa chars), a chain might look like this:

aadquy —H→ 281DAF40 —R→ sgfnyd —H→86AAF4E31—R→ keibym—H→ **920ECF10**

To generate a table of many such chains, we randomly choose a set of *initial passwords*, compute chains of some fixed length *k* for each one, and store *only* the first and the last entries in each chain. More precisely (see pseudo-code), we insert to disk-embedded-hash-table a single condition of End-of-Chain to Beginning-of-chain. So given end-of-chain (some hash-value) we can quickly determine the beginning of the chain using a single query. In the example above, the DEHT action performed looks like **insert(key** : **920ECF10**, **data : aadquy)**.
Intuitively, the motivation is that during query it will be easy to go "down" the chain. So, by storing only a single pair, we have a simple way to go "down" the chain for any intermediate password of this chain.

The first password is called the *starting point* and the last hash result is called the *endpoint*. In the example chain above, "**920ECF10** " would be the endpoint, "**aadquy** " would be the starting point, and none of the other passwords (or the hash values) would be stored. Then endpoints are the "keys" of DEHT. Now, during query, given a hash value *h* that we want to crack (i.e., find the corresponding password for), compute a chain starting with *h* by applying R, then H, then R, then H, and so on. If at any point we observe a value matching one of the endpoints in the table, we get the corresponding starting point and use it to recreate the chain. There's a good chance that this chain will contain the value *h*, and if so, the immediately preceding value in the chain is the password *p* that we seek.

For example, if we're given the hash 86AAF4E31, and compute its chain, we would soon reach the password "sgfnyd" that is stored in the table, e.g.:

86AAF4E31 —R→ keibym - doesn't appear in the table
keibym—H→ **920ECF10** - appears in the table!

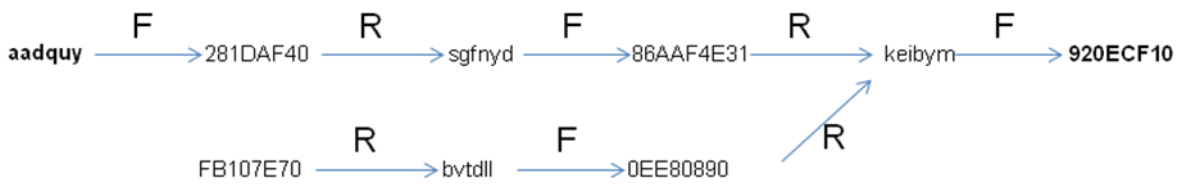
We then get the corresponding starting password "**aadquy**" . Next, follow this chain until 920ECF10 is reached:

aadquy —H→ 281DAF40 —R→ sgfnyd —H→ 86AAF4E31
So the password is "sgfnyd"!

The created table content does not depend on the hash value to be inverted. The table is created once and then repeatedly used without modification. Increasing the length of the chain decreases the size of the table. It also increases the time required to perform lookups, and this is the time-memory trade-off of the rainbow table. In a simple case of one-reduction chains, the lookup is very fast, but the table is very big. Once chains get longer, the lookup slows down, but the table size goes down. Note, however, that the chain resulted by such a DEHT query, does not always contain the desired hash value h. It may so happen that the chain with a completely unrelated starting point merges with the desired chain, with a meeting point that is after h (see diagram below). For example, we may be given a hash value FB107E70, and when we follow its chain, we get:

FB107E70 —R→ bvtdll —H→ 0EE80890 —R→ keibym —H→**920ECF10**

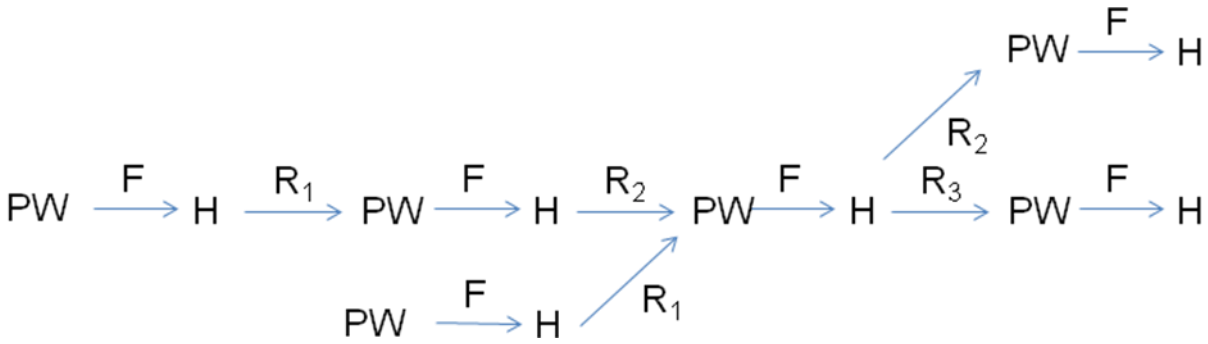
But querying DEHT with key “920ECF10” will return the chain starting at "**aadquy**", (see figure below) which does not contain hash value “FB107E70”. This collision is caused when the reduction function for different hash values (in this case 86AAF4E31 & 0EE80890) returns the same value (in this case keibym).When we get such a collision during query, we refer to it as a ***false alarm***. In such a case, we ignore the match and continue to extend the chain of *h* looking for another match. If the chain of *h* gets extended to length *k* with no good matches, then the password was never produced in any of the chains.



Chain collisions can cause also loss of disk efficiency as we used two DEHT entries that are partially wasted on the same passwords. There is an algorithmic solution called Rainbow-Table that mitigates this problem. Note that H is almost collision free (as its range is at least 128 bits), but R creates a password within a limited domain (i.e. a sub-set of passwords that we want to pre-process) and hence collisions caused by R are inevitable.

Rainbow-tables, background

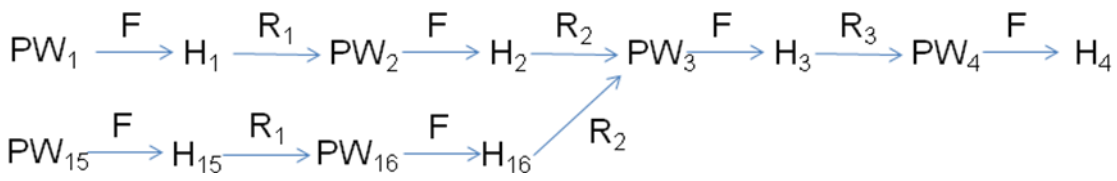
Rainbow-tables are simply pre-computed hash-chains, without the problem illustrated in the diagram above - that each collision causes a merge of chains for the entire way down stream. In other words, as we cannot reduce the amount of collisions between chains, what we want is to have a "recovery" whenever possible. The idea is to use a different reduction (i.e. a different R) at different locations of the chain, thus such merges may occur only if reduction collides on the same index –and so we reduce collision by "chain-length" times. In all common cases, a collision will cause minimal damage, as suggested by the following diagram:



This changes the way a query is done: because the hash value of interest may be found at any location in the chain, it's necessary to generate k different chains (k being the pre-computed chain length), where each such chain is a reconstruction of a potential pre-computed chain that holds the desired hash value in the k-th position. The first chain assumes the hash value is in the last hash position and just applies R_k; the next chain assumes the hash value is in the second-to-last hash position and applies R_{k-1},-H-R_k; and so on until the last chain, which applies to all the reduction functions, alternating with H. This creates a new way of producing a false alarm: if our "guess" of the position of the hash value is wrong, we may needlessly evaluate a chain.
Notice that even when we use an index-dependent reduction function, it is still common that several distinguished chains can end with the same hashed endpoint while starting in different passwords (i.e if we had a collision at the same reduction index). In these cases we get several identical keys (in *prefix*.key) pointing to different passwords (in *prefix*.data file). If we pick just one match (say the first chain that other merges into) we'll get up to 70% success rate in breaking passwords. In order to raise performance, we suggest a *multi-query scheme*: when performing a query on a key in the hash table we will return all potential starting points matching the same last hash. When all passwords are retrieved we treat each password as a potential starting point to the chain we are trying to recover until we find the matching chain.

For example, in the following scenario, when trying to crack "H_{16}" (corresponded to hash PW_{16}) you need to:

1. Create chain H_{16}…H_4 (applying R_2 and R_3)
2. Retrieve all passwords from DEHT that match key H_4 (i.e. PW_1 & PW_{15})
3. Create chain PW_1.. PW_2 by applying R_1 to chain starting with PW_1. PW_2 will yield "**false alarm**" when comparing F(PW_2) to H_{16}.
4. Create chain PW_{15}..PW_{16} by applying R_1 to chain starting with PW_{15}. PW_{16} will yield "**success**" when comparing F(PW_{16}) to H_{16}



"multi-query" requires a varied size answer. Since dynamic allocation is not recommended within tight loop you will use a "masrek" data structure to return all potential passwords. This structure holds two fixed size array, one is an array of pointer to char* of size multi_query (given in ini) that holds all possible pointers to password, and the other is a char array of size (multi_query *MaxPasswordLength) which holds all password retrieved. As multi_query size goes up more collided chains are to retrieve and raises the success rate.

How to create so many reduction functions:

To create many reduction functions you will use pseudo-random functions.
A pseudo-random function is a deterministic function (created by cryptographic techniques) that receives two inputs: a (random) seed s and another input x. Restating F(s,x) as $F_s(x)$ gives us a huge family of functions. Each seed yields a different function. For all practical purposes, when s and t are two independent random values, the functions $F_s(x)$ and $F_t(x)$ behave like completely independent random functions of x. It is easy to pick randomly many independent seeds before pre-process begins and thus get many independent random functions.

To get the reduction function we suggest to concatenate the input (x) with the seed (s) and then hash the result into a 64bit number.
Thus, for each index inside chain j=1,..., chain-length you should pre-compute some random seed, seed[j] , stored in a seed table at *prefix*.seed output file.
An example for a pseudo-random function will be provided in supplementary materials for your use.

Pseudo-code for rainbow table creation:

Pre-process:

Receive a definition of passwords that we want to crack (i.e a rule). Mark this set as ***S***. For clarity, assume that the cryptographic hash to crack is MD5.

To build the rainbow table we iterate many times (about 10 times size of ***S/chain length.***)

```
Generate a random password within S:(e.g. aadquy). Name it firstPass.
Init curHash := MD5(firstPass)
For j=1 to chain-length do
        k = pseudo-random-function with seed[j] and input curHash;        } Reduction
        NewPassword = get_kth_password*(k,S)
        curHash = MD5(NewPassword);
end

Insert into disk embedded hash table the following pair: key=curHash, data=firstPass
end
```

Query:

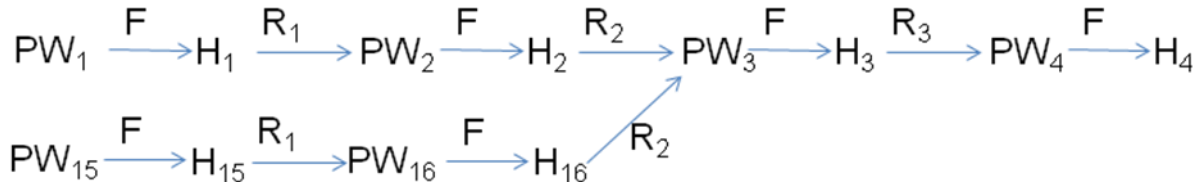
Pseudo code for cracking a hashed password referred as "*target"*, assuming cryptographic hash is MD5 for clarity

```
For j = chain_length to 1 do
        //Gamble that our password is in location "j" in some chain as follow:
        curHash=target
```

```
// go down the chain (chain_length-j ) steps (till curHash = end-point hash).
  For i=j to chain_length do
        k = pseudo-random-function with seed seed[i] and input curHash;
        NewPassword = get_kth_password*(k,S)
        curHash = MD5(NewPassword);
  end // going down the chain.
  Multi-query in disk-embedded hash table with key: curHash.
        Get data (passwords set) to array: tryThisPassword[0..n]
  For k =0 to n-1 // if n=0 (no password is found), we guessed wrong j, continue loop other j.
     //assume tryThisPassword[k] is beginning of correct chain
     curPass = tryThisPassword[k]
     Go j-1 steps down // (till curPass is the password before the hash we are looking for).
     Check whether MD5(curPass)==target
     If so, return curPass
     Else, continue loop // false alarm.
  End // looping multiple query
End //main loop on j
```

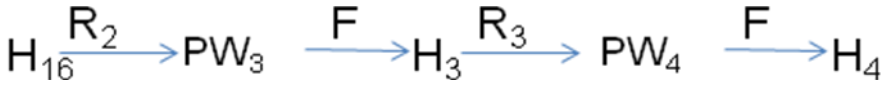
If you arrived here, it means that the target does not exist in either 1st or 2nd or 3rd… location of any-chain, in other-words, not in our Rainbow-Table.

For example, we have pre-computed these two chains:

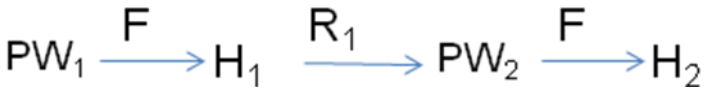


Then our DEHT will have two entries for key H_4: PW_1 & PW_{15}.
Now assuming that we want to crack hash H_{16}: we should loop over all its possible locations in the chain, e.g. all possible reduction function, first assuming that H_{16} is stored as is in the DEHT, after failing to retrieve a matching chain (or retrieving a false alarm) we will assume that H_{16} is the hash that generated the call for R_3, thus building the chain

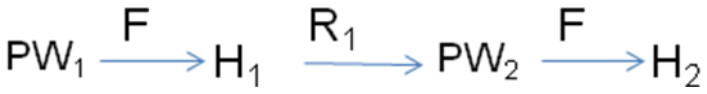
We receive H_{26} as the end point and query H_{26} which will result in a second NULL password or false alarm, since H_{16} is assumed to be in the wrong place in chain.
We continue the loop, assuming now that H_{16} is in the position that generated the call for R_2 thus building chain:



Multi-query(H_4) will return both PW_1 & PW_{15}.
Now we have to reconstruct the other part of the pre-computed chain in order to find the matching password. PW_1 produces this chain:



i.e – another false alarm, while P_{15} will produce this chain:



Revealing that PW_{16} is the password we were looking for.

a. Detailed description

Similarly to exhaustive search, you should deliver one executable file doing pre-processing (**create_rainbow_table**), and another one doing query (**crack_using_rainbow_table**).

create_rainbow_table receives in command line parameters a *prefix* (preferences parameters filename) and search for *prefix*.ini file, give in the ini file format with relevant parameters. It creates 3 files with the same given prefix (*prefix*.key, *prefix*.data & *prefix*.seed).

create_rainbow_table creates rainbow files to be read by **crack_using_rainbow_table**. Parameter file of preferences will have a syntax of <parameter name>=<parameter value> each in a line of its own (similar to ".ini" in windows). Empty lines are possible. Tabs and spaces should be ignored unless in a middle of filename. The parameters are:

- **rule** = string describes rule as shown in class (see appendix).
- **lexicon_name** = filename of all words supported by rule (see appendix).
- **hash_name** = hash function to be used (MD5 \ SHA1)
- **num_of_R** = number of times we use reduction function to create the chain for a single entry in rainbow table.
- **hash_size** = range of DEHT hash-table, i.e. the number of entries in the table of pointers.
- **bucket_size** = number of pairs <key> <data> in each allocated block
- **main_rand_seed** = string describes source of randomness for entire program. No use of "rand" at all during its run!! Thus, to reconstruct same run, use same seed. To get a different run, change seed and nothing else within ini file.
- **multi_query** = number of password to return for a single key query

If any error occurs you exit properly given informative error, for example

- Ini file does not exists
- Either of the files to be created already exists.
- The given cryptographic hash-name is unknown.
- The rule does not fit syntax.
- Ini file does not fit above syntax.

crack_using_rainbow_table receives one argument of the same ini file name that was used in **create_rainbow_table**. If ini file was opened and parsed successfully and the DETH file was found, it repeatedly prints ">>" and gets either:

- Hexa string. Then, query rainbow table.
    - If found print "**Try to login with password \"%s\"\n**"
    - If not, print "**Sorry but this hash doesn't appears in pre-processing\n**"
- String starts with '!'. Then '!' is neglected and the string is considered as a text password. Hash it using cryptographic hash-function and query the result

- Print: "\tIn hexa password is%s\n".
- Continue to act the same as above using the hexa string (i.e. either crack or fail, by feeding the binary hashed password to the cracker).

- String "quit" → exit properly.
- Otherwise: Print *stderr*: "**Commands are either hexa, !password or quit.\n**" and continue the loop.

b. Notes & tips

- This mission is not trivial to debug. First make sure that both **create_rainbow_table** and **text_export_rainbow_table** work properly on short chains and small simple rules before digging into the tricky **crack_using_rainbow_table.** Please consider beginning with seed[j] identical for every j, so the reductions will all be the same. When this version works well, change seed[j] to be independent and thus get better performance (and a small set of new bugs to handle).
- Rainbow-table crack may have many hash-table queries. Thus, you should cache (i.e. load into RAM) the table of pointers, when **crack_using_rainbow_table** is initialized.
- There are two levels of headers – a struct you read when beginning, and the table of pointers.
- **crack_using_rainbow_table** receives ini file but it shall not use the variable *main_rand_seed* at all! Seeds for reduction functions are stored in *prefix*.seed file.
- A key function in this section is "get_kth_password" function (which you will implement). The function is called each time you need to invoke a password that is, at the start of each chain, every reduction call etc. You need to determine an upper-bound for number of passwords that fits a rule analytically. The function "get_kth_password" will return a unique password for each k (k< number of password that fits rule) in O(1) time without preprocessing a heavy data-structure.
- Recall that seed[j] (for each j) is a 64 bit register kept as separate seed list.
- **crack_using_rainbow_table** are deterministic and not dependent in main_rand_seed. **create_rainbow_table** is using pseudo random function which is your randomizer implementation (see misc.h). It first random seeds, and then random each initial password of chains using main_rand_seed and a running index (say loop-index + number-of-seeds). This enable you to reconstruct the same run again and get same results if you keep same ini file. However, if you change main_rand_seed you shall get an independent run. Not just different, independent. It means, that if you detect, say 80% of passwords in some test run with the seed AAB, the seed AAC shall, theoretically, detect different 85%, so amount of passwords to be detected by both runs shall be, theoretically, 97%. Of course, in practice, change more than one bit in the seed to get this effect, and you will get ~95% success rate. In our automatically tests, we try to verify that no password fail systematically, namely we will change seeds revealingly so all test passwords will be detected at least once.
- Notice that in this part we introduced a new output file: *prefix*.seed. This file will hold all different random seeds in binary format according to main_rand_seed and to num_of_R provided in ini file.

2. Miscellaneous

a. Error handling

- If a wrong number of arguments is given, quit with error message (to stderr) of the format "Error: Usage <executable name> <argument1 description> <arg2 desc. > … " for example:
  - **Error: Usage create_rainbow_table <prefix filename>\n**
  - **Error: Usage crack_using_rainbow_table < prefix filename>\n**
- **Most executables** receive commands from the user via prompt (">>"). If a command is an empty line, ignore it, with no message. If the command structure is different from the instructions, then ignore this command, print to *stderr* an informative message and continue to loop. For example:
  Error: Commands are either "quit" or hashed password.\n
- If an output file name already exists, **do not overwrite** it. Quit with error message of
  Error: File "%s" already exist\n
- If you cannot open a file with relevant permissions then use ***perror***(filename) and quit properly. Do it as soon as possible, i.e., do not let the user to enter commands if the file cannot be opened.
  - ***Perror*** prints an informative error message to ***stderr***.
  - Type "man perror" in Unix command line for a detailed description.
- When implementing DEHT, the following errors may occur while trying to open files:
  - create_empty_DEHT may try to open an existing file for writing. In this case return NULL and print to stderr **Error: file "%s" already exist.\n**
  - create_empty_DEHT may fail to create files (for example, non existing directory). In such a case use ***perror***(*prefix*) and return NULL.
  - load_DEHT_from_files may fail to open files for binary reading (e.g. no permissions or non existing files, etc.) In such a case use ***perror(****prefix)*and return NULL.
  - If a DEHT creation function returns NULL, quit properly.
  - If any DEHT function returns FAIL (-1) due to a disk operation (e.g. fwrite) that fails, then quit properly using ***perror***(*filename*).

b. Notes & tips

- You must use **function-pointers** for cryptographic hash functions, so delivery of hash function is by pointer rather than by name, id or enum. Thus, if someone wants to add another hash function, update is easy. The data-type is defined in misc.h .
- Whenever binary data is exported to text use **binary2hexa**.
- Note that lines are terminated (in Unix) with a single char: \n. This is how files are supplied and tested. In windows, two chars are present: \n\r. Use conversion when porting files across platforms.
- Recall that in "nova" where your project must run on, "long" means a 64bit register. Its analog on Visual studio is __int64. See in misc.h and deht.h two relevant typedefs you may need to change when porting.
- Note that throughout all the interfaces provided to you we kept the convention of

defining an array of binary data as **unsigned char *.** We used **char *** for textual strings (null terminated).

- In any case of program-termination (either successful or not) you must free all allocated memory, and close all open files.
- If you fail on reading INI file, please make sure message contain the exact parameter caused problem, so if you fail to read our INI file, we can fix it for you.
- If INI file contain more parameters than necessary, skip them. This how INI are treated, as in real life many softwares use the same INI.
- If INI file does not contain one or more necessary parameters, quit properly raising appropriate error.

Submission guidelines

Please check the course webpage for updates and Q&A.
Any question should be posted to the course forums.

Review the supplementary input and output files in order to fully understand the exercise. You can find information about developing C under UNIX (e.g. working with make files) in the course webpage. Also, please follow the detailed submission guidelines carefully.

Files

- Create a directory ~/soft-proj11/ex3
- Copy the files of the assignment from the course website to the above directory
- Set permissions using
  chmod 755 ~
  *chmod -R 755 ~/*soft-proj11

Compilation

In this part, YOU should write your own makefile, providing at least following targets:

- all – builds all obj files and the executable
- Clean – deletes all binary files

Code submission

Although the submission is in pairs, every student must have all the exercise files under his home directory as described above. The exercise files of both partners must be identical. Each exercise directory (~/soft-proj11/ex2) must contain a file named **partners.txt** that contains the following information:

Full Name: your-full-name
Id No 1: your-id
User Name 1: your-user-name
Id No 2: partner-id
User Name 2: partner-user-name
Assignment No: the-assignment-number

Printout submission

You will not be required to print your code, instead you should submit one to three pages of your project design containing a description of your .c files and their dependencies. For each file you should specify its name, in which part of the project it is called (can be several parts), a list of central function in the file, a short description of each function and finally a list of dependencies to other files. Example is listed in appendix below.
Notice that even though you don't submit your code, code style check will be performed as usual. To that end you should also submit a printed table in which the checker will list his notes regarding your code style, bugs, errors etc. The code will be checked directly from your project directory. The table to print is attached at appendix below.
Both design form and table should be submitted to the checker's mailbox (first floor in front of the elevators, mail box # 296, Reuven Aronashvili). The printout should include the id-numbers and user names of both partners (one printout submission per pair).

Good Luck!

Appendix:

Formal definitions of rule & lexicon:

A rule is a string, consisting of the letters @#.$ and digits. It represents a regular expression of all passwords we wish to crack.

Rule is a concatenation of one or more of the following substring:

- "@n" means a sequence of up to n case sensitive word from lexicon (e.g. @1 can yield "Hello", "world", "Helloworld" etc)
- "#n" means a sequence of up to n digits. (e.g.#3 can yield: "", "70", "007"etc)
- ".n" means a sequence of up to n English characters, lower case. (e.g. "awxx", "aa", "europe" for .6).
- $n means a sequence of up to n characters (i.e. ascii code 32-126). (e.g. "Awesome!", "OMG:/", "#$%@#$" for $8).

Note than n in all 4 substring will be one digit (1-9).

In this part you are required to implement a complicated rule, which means that the password can fit at least one rule, the rules are delimited by the "&" sign, for example:

- .2&#2 can yield "70", "07", "er" or "g" but not "4f"

The lexicon is a text file consisting of lines. Each line is considered a word. Note that few of the words will have spaces (e.g. tom and jerry) but most will be English words (e.g. horse). At least one of the lines is empty, and the empty word (i.e. string with length 0) is also legal. Similarly, empty digit sequences are legal, and even common.

Design form example:

File name: crack_using_rainbow_table.c
Used in: part 3
Central functions:

- main() : parse input from user, load DEHT rainbow table and call crack function for each hashed password.
- crack(): receive a hashed password and return the cracking password if exists using rainbow table algorithm.

Dependencies:

- deht.c – this file is used to query the rainbow table created in createrainbowtable
- misc.c – this file is used to calculate reduction function.

Note table to print:

file	function	line	Note	score