Software Project, spring 2011 - Assignment 2

Due by 02.06.2011, 23:59

In this assignment you will implement the second part of your project.

The Exercise will allow you to practice data structures, files, pointers to functions, strings and more. Its weight is 30% of your final course grade.

Outline

Exhaustive search with pre-processing attack, consisting of two executables:
exhaustive_table_generator and **exhaustive_query.**

One interface you must implement during the exercise is a Disk Embedded Hash Table – DEHT. You must follow the data structures and algorithms learned in class. An interface deht.h is supplied and you must create a matching implementation deht.c. Another supplied interface is misc.h – miscellaneous bit-level functions. Read carefully the definition and examples in misc.h. We will also provide relevant web-links for your convenience. We will not create any special purpose tests for this module, but we recommend you to do your own unit testing.

Background

1.1. Pre-processing based attack using exhaustive search

The notion of “pre-processing” based attack, along with efficient algorithms for "disk embedded hash table" were explained in class. Recall that there are two stages in this process.

- Pre-processing – a time consuming process that loops over possible passwords (w) and calculates their cryptographic-hash (h). It inserts the pair (h,w) into a disk-embedded hash-table.
- Cracking – a very quick process that given a cryptographically-hashed password (h), queries the disk-embedded-hash-table and finds a matching password (w) that will be authenticated.

As an important subroutine for this process, you should create a function that given a 64-bit ("long" in nova) number k, generates the k-th password that fits a pre-specified rule. Thus, you can easily generate a random password or loop over all passwords that satisfy the rule. When looping over all possible k, some passwords may appear more than once, but you must make an effort to minimize these cases as they impair cracking efficiency. We expect that the function will be 1:1. When a rule is a union of several terms, and these terms intersect, it is fine that passwords which satisfy ***m*** of the terms will be generated by at most ***m*** different values of k.

Another important module is the heavily used data structure DEHT. Please read carefully deht.h and implement deht.c accordingly. As learned in class, the data-structure is purely disk-embedded and its pointers are offsets within files. Algorithmically, it is based on a hash-table with linked-list buckets, each link-element consisting of blocks of a predefined size. Key and data are saved to different files: *prefix*.key and *prefix*.data, respectively.

1.2. Technical description

exhaustive_table_generator receives a single arguments of file prefix in the command line. This prefix is the one used to create *prefix*.key and *prefix*.data, a third file *prefix*.ini contains other initialization parameters:

- Rule ("rule", see syntax in appendix).
- Lexicon file name ("lexicon_name")
- Hash name ("hash_name", either MD5 or SHA1).
- Flag ("flag", either "all" or a number n).
    - "all" means you scan all possible passwords one by one until covering all options, each one exactly once (if your scanning loop "visits" a small fraction of passwords more than once it is OK).
    - A number n means that you loop n times. Each iteration randomly chooses an index, calculates its corresponding password and hash value, and outputs to a file.

Both ini file and lexicon file are small enough to be loaded into RAM.

The program creates two files (prefix.key and prefix.data) which represent a data-structure that enables to crack a given hashed-password very quickly using **exhaustive_query**. You will use default DEHT-optimization parameters:

- Use 10 pairs (key,data) per block.
- Use a signature of 64bit per key (i.e. 8 Bytes).
- Use 65536 (2^16) entries in the table-of-pointers (i.e. 16-bit hash function).

exhaustive_query receives a single argument of a prefix for a DEHT pair of files. The program should:

- Verify that the files (.key and .data) exist and read the hash name from its header.
- Load the header into memory, but not the table of pointers (i.e. do not call read_DEHT_pointers_table).
- Repeatedly print "**>>**" and read the hashed passwords typed by the user (using fgets(buffer,stdin,256) and **hexa2binary**). If the input is not a hexadecimal number (i.e. not a sequence of hexa digits), it prints "Non hexa\n", ignores the last typed command and continues. When the user types "quit", it exits properly.
- Print either "**Try to login with password \"%s\"\n**" or "**Sorry but this hash doesn't appear in pre-processing\n**" (and continue to loop).

1.3. Notes & tips

- You must use the DEHT interface with your own implementation.
- You must be restrictive with disk space consumption. For example, store binary data instead of textual characters, and all other disk-saving tricks suggested in class. These pre-processing modules, when used in practice, push system resources to the limit. Thus, disk-space should be treated accordingly.
- Recall that hashed passwords serve as keys for the DEHT i.e., hashed passwords are used to map the data into a specific index in the array. Moreover, the keys should be saved in hash-table for comparison and validation vs. the query key when retrieving data. For efficiency, we will save only the 8 lower bytes of the hashed key to the hash table, which will save space and won't affect validation performance.

- Unlike querying, during the creation of the table (**exhaustive_table_generator)**, you should hold the relevant table-of-pointers (both head and tails) in memory as it is frequently accessed during creation. Namely, before such a long phase of many insertions, call "calc_DEHT_last_block_per_bucket" and after the phase is completed, call "write_DEHT_pointers_table".
- During **exhaustive_table_generator**, the function that returns the k-th password is run many times. Thus, you may pre-process the "rule" and "lexicon" as much as needed. For example, create an array of strings describing the words in the lexicon, split the rule into terms, etc. It is recommended to create a data-structure that holds pre-processed rules, but again, its memory consumption should be few KB, at most 1MB for very complex input, even for rules that produces billion of passwords (e.g. keep some string manipulations on the rule itself, and an array of string that holds the lexicon).

- Please note, that if flag "all" is used, program is purely deterministic. It shall have 100% success for passwords that fit rule and 0% with those doesn't.

- Save 8 lower bytes of hashed key into the "key" field in *prefix*.key file. The "data" filed in th*e* *prefix*.key is holds both data offset in *prefix*.data file and data length.

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. > … "
  - **Error: Usage exhaustive_table_generator <filenames prefix>\n**

  - **Error: Usage exhaustive_query <filenames prefix>\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. Example for **exhaustive_query**:
  Error: Commands are either "quit" or <hashed password>.\n
- If any of the parameters in *prefix*.ini file is illegal or missing print an informative error message, for example:
  - If the given hash name is neither "MD5" nor "SHA1", quit with an error message of **Error: Hash "%s" is not supported\n**
  - If the given flag is neither "all" nor positive number, quit with an error message of **Error: flag "%s" is not supported\n**
  - If the rule does not fit the format, quit with an error message (to *stderr*): **Error: rule "%s" does not fit syntax.\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 (e.g. writing in **exhaustive_table_generator**, reading in **exhaustive_query**) 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 relevant to both executable

- 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 interface files.
- 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. Be careful 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. In case a file creation has failed, you must erase (defected) files from the disk.
- 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 software use the same INI.
- Note that INI file parameters are not constrained to a specific order.

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 digits (1-9).

The lexicon is a text file consisting of lines. Each line 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. Long). At least one of the lines is empty, and the empty word (i.e. string with length 0) is always legal. Similarly, empty digit sequences are legal, and even common. Notice, however, that even though each substring in the rule can be an empty sequence, the entire rule must not be empty, that is, an empty password which theoretically can be invoked by a rule is illegal!

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/ex2
- 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

Makefile is provided at the website. Your exercise should pass the compilation test, which will be performed by running the make all command in a UNIX terminal window.

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

Good Luck!