Laboratorium systemów wbudowanych i czasu rzeczywistego.

Program brute-force na systemy QNX

Wydział Elektryczny

POLITECHNIKA WARSZAWSKA

Wykonał: Paweł Drapiewski,
GR1 Informatyka, 6 sem, dzienne
Data: 05.04.2017 r

Spis treści

Wstęp

Program ten ma za zadanie łamać hasła programów uruchamianych przy pomocy CLI (ang. *command line intrepreter*) na systemie operacyjnym czasu rzeczywistego QNX. Wykorzystuje on prosty mechanizm brute-force, czyli sprawdzania kolejno wszystkich możliwych wzorców hasła.

Opis algorytmu

Przyjęty algorytm nazywa się metodą bruteforce i polega na generowaniu kolejno wszystkich kombinacji haseł na danym słowniku, sprawdzając ich poprawność poprzez próbę zalogowania.

W przypadku naszego programu generowane hasło będzie się składać ze znaków zdefionowanych w funkcji *init_char_set()* (linia 196 kodu programu *bruteforce.c* dostępnego w dodatku A) czyli dostępne zanki to małe litery, duże litery oraz cyfry.

Główna pętla programu to *start_bruteforce()* (linia 66 *bruteforce.c*) odpowiada ona za kolejne przejście po wygenerowanych wzorcach hasła, póki nie zostanie odnalzione dopasowanie lub skończą się kombinacje dla podanej przez użytkownika maksymalnej długości hasła.

Generacja hasła odbywa się w funkcji *generate_next_password_pattern()* (linia 94 *bruteforce.c*). Podmienia ona ostatni znak hasła na znak kolejny jaki występuje w tablicy *char_set* zdefionowanej w *init_char_set()*. W przypadku, gdy nie istnieje kolejny znak, to zwiększany jest jego poprzednik.

Walidacja poprawności danych

Program jest odporny na:

- podanie zbyt mało argumentów
- podanie argumentów w złej kolejności
- sprawdza czy podany program istnieje przed próbą włamywania się do niego
- dla prametru maksymalna długość hasła sprawdza on czy jest to liczba oraz czy jej zakres mieści się w przedziale [1; 1024]

Wnioski

Napisanie kodu w C zawsze nie jest zadaniem trywialnym, ponieważ wymaga pełnego zarządzania pamięcią (a brak uwalniania i inicjalizacji pamięci potrafi dokuczyć :)). Lecz sam język C słynie ze swojej szybkości, dlatego zdaje się być jezykiem idealnym dla postawionego problemu. Dodatkowo sprzyjającym czynnikiem okazał się system operacyjny QNX, ponieważ jest on zoptymalizowany na jak najszybsze przetwarzanie programu, co było widoczne porównując czasy wykonania z np. systemem opartym na dystrybucji Debian. Jednakże istnieje możliwość

uzyskania jeszcze lepszych rezultatów wykorzystując wątków lub obliczeń rozproszonych czy też poprzez przeniesienie obliczeń na kartę graficzną stosując platofrmę CUDA. A jeśli i tutaj wydajność nie spełni naszych oczekiwań, to należy wtedy zastosować bardziej wyrafinowane metody np. metodę słownikową, która nie zawsze da 100% skuteczności, ale na pewno w większości przypadków przyśpieszy proces odnajdywania hasła.

Dodatek A: Kod źródłowy

Program do shakowania program_to_hack.c:

```
/*********************************************************
 * Author: Pawel Drapiewski                              *
 * This program only validate user inputs as username and  *
 * password and return 0 if everything is ok             *
 *********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>

#define USERNAME "drapek"
#define PASSWORD "g8"
#define BUFF_SIZE 1024

int authorization( char * input_usrnm, char * input_psswd) ;

int main( int argc, char ** argv ) {
        char * input_username;
        char * input_password;

        if( argc < 3 ) {
                printf("Error, you need to 2 parametres: username and password!\n");
                return EXIT_FAILURE;
        }

        input_username = argv[1];
        input_password = argv[2];

        if( !authorization(input_username, input_password) ){
                printf("You gained access!\n");
                return EXIT_SUCCESS;
        }
        else {
            printf("Wrong username or password.\n");
                return EXIT_FAILURE;
        }
}

int authorization( char * input_usrnm, char * input_psswd) {
        if( strcmp( input_usrnm, USERNAME ) == 0 &&
            strcmp( input_psswd, PASSWORD ) == 0 ) {
                return 0;
         }
        else {
                return -1;
        }
}
```

Program hakujący bruteforce.c:

```
1    /*********************************************************
2     * Author: Pawel Drapiewski                              *
3     * This program make simply brutforce attack on program      *
4     * specified by user in program parameter                *
5     * Execution: brutforce [program path] [username] [password] *
6     *********************************************************/
7     #include <stdio.h>
8     #include <stdlib.h>
9     #include <string.h>
10    #include <unistd.h>
11
12    #define WRONG_PASSWORD 0
13    #define CORRECT_PASSWORD 1
14    #define FILE_EXISTS 1
15    #define FILE_DOESNT_EXISTS 0
16    #define BUFF_SIZE 1024
17    #define DEBUG 1
18
19    char * create_execution_command(const char * program_path, const char * username, char * password );
20    int check_password(const char * program_path, const char * username, char * password );
21    int check_if_file_exist(const char * program_path);
22    char * init_char_set();
23    void print_array(char * array);
```

```c
int start_brutforce(const char * program_path, const char * username, int password_length, char * brutforce_char_set, int charset_length);
void init_actual_pswd_pattern(int * actual_pswd_pattern, int charset_length);
void transform_pswd_to_string(int * actual_pswd_pattern, int password_length, char * charset, char * store_pswd_here);
int generate_next_password_pattern(int * actual_password_indexes, int password_length, int charset_length);

int main(int argc, char const *argv[]) {
  int max_pswd_length;
  int charset_length;
  char * brutforce_char_set;
  int result;

  if( argc < 4 ) {
    printf("[ERROR] You need 3 parametres: [program_path] [user_login] [password_max_lengts]\n");
    return EXIT_FAILURE;
  }

  max_pswd_length = atoi(argv[3]);

  if( (max_pswd_length < 1) || (max_pswd_length > 1024)) {
    printf("[ERROR] Maximum password length must be greater than 0 and smaller than 1025\n");
    return EXIT_FAILURE;
  }

  if( check_if_file_exist(argv[1]) == FILE_DOESNT_EXISTS ) {
    printf("[ERROR] Program \'%s\' doesn't exists!\n", argv[1]);
    return EXIT_FAILURE;
  }

  charset_length = 0;
  brutforce_char_set = init_char_set(&charset_length);
  #if (DEBUG)
    printf("Charset length: %d\n", charset_length);
    print_array(brutforce_char_set);
  #endif

  printf("Processing in progress. Be patient, this operation can take long time.\n");
  result = start_brutforce(argv[1], argv[2], max_pswd_length, brutforce_char_set, charset_length);

  return result;
}


int start_brutforce(const char * program_path, const char * username, int password_length, char * brutforce_char_set, int charset_length) {
  int * actual_pswd_pattern;
  char * password_to_check;
  int is_all_patterns_checked;
  int i;

  actual_pswd_pattern = (int *) malloc((password_length) * sizeof(int)); /*stores password as indexes of brutforce_char_set array */
  init_actual_pswd_pattern(actual_pswd_pattern, password_length);
  password_to_check = (char *) malloc((password_length + 1) * sizeof(char)); /* stores password as string */

  is_all_patterns_checked = 0;
  while( !is_all_patterns_checked) {
    transform_pswd_to_string(actual_pswd_pattern, password_length, brutforce_char_set, password_to_check);
    if( check_password(program_path, username, password_to_check) == CORRECT_PASSWORD ) {
      printf("[SUCCESS] Password found: %s\n", password_to_check);
      return 0;
    }
    #if( DEBUG )
      printf("\tActual checked password: %s\n", password_to_check);
    #endif
    is_all_patterns_checked = generate_next_password_pattern(actual_pswd_pattern, password_length, charset_length);
  }

  printf("[FAIL] Sorry, password not found. May try with longer password or check username.\n");
  return 1;
}

/* generate next password pattern. Returns 1 when there is no next pattern, 0 if OK*/
int generate_next_password_pattern(int * actual_password_indexes, int password_length, int charset_length) {
   int i;
   int j;
   int k;

   i = 1;
    while( actual_password_indexes[i] != -1 && i < password_length) {
       i++;
    }

    i--; /* this is index of the last char in password pattern */

    if(actual_password_indexes[i] == (charset_length - 1)) {
      /* if end char or chars have maximum value than find previous element who isn't*/
      k = i;
      while( actual_password_indexes[k] == (charset_length - 1)) {
        k--;
        /*if all elemntes have maximum value than exapnd password */
        if( k == -1)  {
            /* when password have maxium length with all maxiumum values than there is no more options */
            if(i == (password_length - 1))
                return 1; /* we can't generete next pattern because our password has already max lenght */

            j = i + 1;
            for(; j >= 0; j--) {
              actual_password_indexes[j] = 0; /*assign index of first char in char set array */
            }
        } else {
          /* add value of first non maximum elements, and from this char to end assign first char value*/
          actual_password_indexes[k] =+ actual_password_indexes[k] + 1;
```

```
            j = k + 1;
            while(actual_password_indexes[j] != -1 && j < password_length) {
              actual_password_indexes[j] = 0;
              j++;
            }
          }
        } /*while closing*/
      } else {
        actual_password_indexes[i] = actual_password_indexes[i] + 1;
      }

      return 0;
  }

  void init_actual_pswd_pattern(int * actual_pswd_pattern, int charset_length){
    int i;
    for(i = 0; i < charset_length; i++){
      actual_pswd_pattern[i] = -1;
    }
  }


  void transform_pswd_to_string(int * actual_pswd_pattern, int password_length, char * charset, char * store_pswd_here) {
    int i = 0;
    while( actual_pswd_pattern[i] != -1 && i < password_length ) {
      store_pswd_here[i] = charset[actual_pswd_pattern[i]];
      i++;
    }

    store_pswd_here[i] = '\0';
  }

  /* Check if username and password for given program is correct */
  int check_password(const char * program_path, const char * username, char * password ) {
    char * command = create_execution_command(program_path, username, password);

    if (system(command) == 0) {
      return CORRECT_PASSWORD;
    }
    else {
      return WRONG_PASSWORD;
    }
  }

  /* Creates command with parameters to execute it in system() function */
  char * create_execution_command(const char * program_path, const char * username, char * password ) {
    char * command_to_execute = malloc(BUFF_SIZE * sizeof(char));
    memset(command_to_execute, '\0', BUFF_SIZE * sizeof(char));
    char * spacebar = " ";
    char * program_runner = "./";

    strcat(command_to_execute, program_runner);
    strcat(command_to_execute, program_path);
    strcat(command_to_execute, spacebar);
    strcat(command_to_execute, username);
    strcat(command_to_execute, spacebar);
    strcat(command_to_execute, password);
    strcat(command_to_execute, " 1> /dev/null"); /*ignores stdout of hacking program*/

    return command_to_execute;
  }

  /* chec if exists program under giver program path */
  int check_if_file_exist(const char * program_path) {
    if(access(program_path, F_OK) != -1) {
        return FILE_EXISTS;
    }
    else {
      return FILE_DOESNT_EXISTS;
    }
  }

  char * init_char_set(int * charset_length) {
    char * result_char_set;
    int array_elem_nmb;
    int char_a_code;
    int char_z_code;
    int range;
    int i;
    int start_index;


    result_char_set = (char *) malloc(BUFF_SIZE * sizeof(char));
    array_elem_nmb = 0;

    /*put small letters into array*/
    char_a_code = 97;
    char_z_code = 122;
    range = char_z_code - char_a_code;
    i;
    for (i = 0; i <= range; i++){
      result_char_set[i] = (char)(char_a_code + i);
      array_elem_nmb++;
    }

    /*put big letters into array*/
    char_a_code = 65;
    char_z_code = 90;
    range = char_z_code - char_a_code;

    start_index = array_elem_nmb;

    for (i = 0; i <= range; i++){
```

```
227         result_char_set[i + start_index] = (char)(char_a_code + i);
228         array_elem_nmb++;
229       }
230
231       /*put numbers into array*/
232       char_a_code = 48;
233       char_z_code = 57;
234       range = char_z_code - char_a_code;
235
236       start_index = array_elem_nmb;
237
238       for (i = 0; i <= range; i++){
239         result_char_set[i + start_index] = (char)(char_a_code + i);
240         array_elem_nmb++;
241       }
242
243       *charset_length = array_elem_nmb;
244       result_char_set[array_elem_nmb] = '\0'; /*the last element of array shoud be null*/
245
246       return result_char_set;
247     }
248
249     void print_array(char * array) {
250       int i;
251       printf("Charset for brutforce attack:\n");
252       i = 0;
253       while(array[i] != '\0') {
254         printf("[%d] %c (addr: %d)\n", i, array[i], &array[i]);
255         i++;
256       }
257     }
```