

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.....	3
Opis algorytmu.....	3
Walidacja poprawności danych.....	3
Wnioski.....	3
Dodatek A: Kod źródłowy.....	4
Program do shakowania program_to_hack.c:.....	4
Program hakujący bruteforce.c:.....	4

Wstęp

Program ten ma za zadanie łamać hasła programów uruchamianych przy pomocy CLI (ang. *command line interpreter*) 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 kolejne próby zalogowania.

W przypadku naszego programu generowane hasło będzie się składać ze znaków zdefiniowanych w funkcji *init_char_set()* (linia 196 kodu programu *bruteforce.c* dostępnego w dodatku A). Czyli znakami, z których może składać się hasło są 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 odnaleziona dopasowana kombinacja 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* zdefiniowanej w *init_char_set()*. W przypadku, gdy nie istnieje kolejny znak, to zwiększany jest jego poprzednik i tak do wyczerpania wszystkich kombinacji dla zadanej maksymalnej długości hasła.

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 parametru 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 językiem idealnym dla postawionego problemu. Dodatkowo sprzyjającym czynnikiem okazał się system operacyjny QNX, ponieważ jest on zoptymalizowany na jak najszybsze przetwarzanie, co było zauważalne dzięki porównaniu

czasu wykonania programu z np. systemem opartym na dystrybucji Debian.

Jednakże istnieje możliwość uzyskania jeszcze lepszych rezultatów od tych osiągniętych przez zaprezentowany program. Przykładowo można podnieść wydajność przetwarzania wykorzystując m.in. wątki lub obliczenia rozproszone czy też poprzez przeniesienie obliczeń na kartę graficzną stosując platformę CUDA. A jeśli i tutaj wydajność nie spełni naszych oczekiwań, to należy wtedy zastosować bardziej wyrafinowane metody łamania haseł np. metodę słownikową, która pomimo, że nie zawsze da 100% skuteczności to na pewno w większości przypadków znacznie przyspieszy 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 <unistd.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);
24 int start_brutforce(const char * program_path, const char * username, int password_length, char * brutforce_char_set, int
charset_length);
25 void init_actual_pswd_pattern(int * actual_pswd_pattern, int charset_length);
26 void transform_pswd_to_string(int * actual_pswd_pattern, int password_length, char * charset, char * store_pswd_here);
27 int generate_next_password_pattern(int * actual_password_indexes, int password_length, int charset_length);
28
29 int main(int argc, char const *argv[]) {
30     int max_pswd_length;
31     int charset_length;
32     char * brutforce_char_set;
33     int result;
34
35     if( argc < 4 ) {
36         printf("[ERROR] You need 3 parametres: [program_path] [user_login] [password_max_lengths]\n");
37         return EXIT_FAILURE;
38     }
39
40     max_pswd_length = atoi(argv[3]);
41
42     if( (max_pswd_length < 1) || (max_pswd_length > 1024) ) {
43         printf("[ERROR] Maximum password length must be greater than 0 and smaller than 1025\n");
44         return EXIT_FAILURE;
45     }
46
47     if( check_if_file_exist(argv[1]) == FILE_DOESNT_EXISTS ) {
48         printf("[ERROR] Program '%s\' doesn't exists!\n", argv[1]);
49         return EXIT_FAILURE;
50     }
51
52     charset_length = 0;
53     brutforce_char_set = init_char_set(&charset_length);
54     #if (DEBUG)
55         printf("Charset length: %d\n", charset_length);
56         print_array(brutforce_char_set);
57     #endif
58
59     printf("Processing in progress. Be patient, this operation can take long time.\n");
60     result = start_brutforce(argv[1], argv[2], max_pswd_length, brutforce_char_set, charset_length);
61
62     return result;
63 }
64
65 int start_brutforce(const char * program_path, const char * username, int password_length, char * brutforce_char_set, int
charset_length) {
66     int * actual_pswd_pattern;
67     char * password_to_check;
68     int is_all_patterns_checked;
69     int i;
70
71     actual_pswd_pattern = (int *) malloc((password_length) * sizeof(int)); /*stores password as indexes of brutforce_char_set
array */
72     init_actual_pswd_pattern(actual_pswd_pattern, password_length);
73     password_to_check = (char *) malloc((password_length + 1) * sizeof(char)); /* stores password as string */
74
75     is_all_patterns_checked = 0;
76     while( !is_all_patterns_checked ) {
77         transform_pswd_to_string(actual_pswd_pattern, password_length, brutforce_char_set, password_to_check);
78         if( check_password(program_path, username, password_to_check) == CORRECT_PASSWORD ) {
79             printf("[SUCCESS] Password found: %s\n", password_to_check);
80             return 0;
81         }
82         #if( DEBUG )
83             printf("\tActual checked password: %s\n", password_to_check);
84         #endif
85         is_all_patterns_checked = generate_next_password_pattern(actual_pswd_pattern, password_length, charset_length);
86     }
87
88     printf("[FAIL] Sorry, password not found. May try with longer password or check username.\n");
89     return 1;
90 }
91
92 /* generate next password pattern. Returns 1 when there is no next pattern, 0 if OK*/
93 int generate_next_password_pattern(int * actual_password_indexes, int password_length, int charset_length) {
94     int i;
95     int j;
96     int k;
97
98     i = 1;
99     while( actual_password_indexes[i] != -1 && i < password_length ) {
100         i++;
101     }
102
103     i--; /* this is index of the last char in password pattern */
104
105     if(actual_password_indexes[i] == (charset_length - 1)) {
106         /* if end char or chars have maximum value than find previous element who isn't*/
107         k = i;
108         while( actual_password_indexes[k] == (charset_length - 1) ) {
109             k--;
110             /*if all elemntes have maximum value than exapnd password */
111             if( k == -1 ) {
112                 /* when password have maxium length with all maximum values than there is no more options */
113                 if(i == (password_length - 1))
114                     return 1; /* we can't genereate next pattern because our password has already max lenght */

```

```

116
117     j = i + 1;
118     for(; j >= 0; j--) {
119         actual_password_indexes[j] = 0; /*assign index of first char in char set array */
120     }
121     } else {
122         /* add value of first non maximum elements, and from this char to end assign first char value*/
123         actual_password_indexes[k] += actual_password_indexes[k] + 1;
124         j = k + 1;
125         while(actual_password_indexes[j] != -1 && j < password_length) {
126             actual_password_indexes[j] = 0;
127             j++;
128         }
129     }
130     } /*while closing*/
131     } else {
132         actual_password_indexes[i] = actual_password_indexes[i] + 1;
133     }
134
135     return 0;
136 }
137
138 void init_actual_pswd_pattern(int * actual_pswd_pattern, int charset_length){
139     int i;
140     for(i = 0; i < charset_length; i++){
141         actual_pswd_pattern[i] = -1;
142     }
143 }
144
145
146 void transform_pswd_to_string(int * actual_pswd_pattern, int password_length, char * charset, char * store_pswd_here) {
147     int i = 0;
148     while( actual_pswd_pattern[i] != -1 && i < password_length ) {
149         store_pswd_here[i] = charset[actual_pswd_pattern[i]];
150         i++;
151     }
152
153     store_pswd_here[i] = '\0';
154 }
155
156 /* Check if username and password for given program is correct */
157 int check_password(const char * program_path, const char * username, char * password ) {
158     char * command = create_execution_command(program_path, username, password);
159
160     if (system(command) == 0) {
161         return CORRECT_PASSWORD;
162     }
163     else {
164         return WRONG_PASSWORD;
165     }
166 }
167
168 /* Creates command with parameters to execute it in system() function */
169 char * create_execution_command(const char * program_path, const char * username, char * password ) {
170     char * command_to_execute = malloc(BUFF_SIZE * sizeof(char));
171     memset(command_to_execute, '\0', BUFF_SIZE * sizeof(char));
172     char * spacebar = " ";
173     char * program_runner = "./";
174
175     strcat(command_to_execute, program_runner);
176     strcat(command_to_execute, program_path);
177     strcat(command_to_execute, spacebar);
178     strcat(command_to_execute, username);
179     strcat(command_to_execute, spacebar);
180     strcat(command_to_execute, password);
181     strcat(command_to_execute, " 1> /dev/null"); /*ignores stdout of hacking program*/
182
183     return command_to_execute;
184 }
185
186 /* chec if exists program under giver program path */
187 int check_if_file_exist(const char * program_path) {
188     if(access(program_path, F_OK) != -1) {
189         return FILE_EXISTS;
190     }
191     else {
192         return FILE_DOESNT_EXISTS;
193     }
194 }
195
196 char * init_char_set(int * charset_length) {
197     char * result_char_set;
198     int array_elem_nmb;
199     int char_a_code;
200     int char_z_code;
201     int range;
202     int i;
203     int start_index;
204
205
206     result_char_set = (char *) malloc(BUFF_SIZE * sizeof(char));
207     array_elem_nmb = 0;
208
209     /*put small letters into array*/
210     char_a_code = 97;
211     char_z_code = 122;
212     range = char_z_code - char_a_code;
213     i;
214     for (i = 0; i <= range; i++){
215         result_char_set[i] = (char)(char_a_code + i);
216         array_elem_nmb++;
217     }
218

```

```

219  /*put big letters into array*/
220  char_a_code = 65;
221  char_z_code = 90;
222  range = char_z_code - char_a_code;
223
224  start_index = array_elem_nmb;
225
226  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 should 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 }

```