Technology Corner: Brute Force Password Generation -- Basic Iterative and Recursive Algorithms

Nick V. Flor, Haile Shannon
Anderson School of Management
University of New Mexico
nickflor@unm.edu, hshannon@unm.edu

ABSTRACT

Most information systems are secured at minimum by some form of password protection. For various reasons a password may be unavailable, requiring some form of password recovery procedure. One such procedure is software-based automated password recovery, where a program attempts to log into a system by repeatedly trying different password combinations. At the core of such software is a password generator. This article describes the basic iterative and recursive algorithms for generating all possible passwords of a given length, which is commonly referred to as *brute-force password generation*. The paper ends with a discussion of alternative password recovery procedures one should attempt before brute-force password recovery.

1. INTRODUCTION

It is common to forget passwords. A study of Yahoo users found that 2,149 out of 50,100 users (4.28%) forgot their passwords over a three-month period (Florencio & Herley, 2007). The same study reported that the average user had seven distinct passwords that were actively used. Another study of students reported that one-third of all respondents had forgotten passwords and that the respondents had an average of 4.45 different passwords, which they used to access 8.18 different applications (Brown, Bracken, Zoccoli, & Douglas, 2004).

When a user forgets a password for a system, some form of password recovery procedure is necessary. Brute-force password generation is one such procedure. Using brute-force, a program generates all possible combinations of passwords and attempts to login to the system using these combinations. This paper presents two algorithms for brute-force password generation. The paper ends with a discussion of the limitations of brute-force password generation and outlines other password recovery procedures that should be attempted prior to trying brute-force.

2. METHOD

Microsoft's Visual Studio 2010 Express development environment was used to

create two algorithms for automatically generating passwords of a user-specified length: (1) an iterative and (2) a recursive algorithm. The programming language used was C#, and the correctness of the algorithms was determined by visual inspection of the passwords generated for password sizes up to 8 characters. It should be noted that there are many different ways to implement automated password generators, and they are not difficult to write. Our goal was to create short algorithms that were both understandable and easy to modify.

3. RESULTS

There are a number of algorithms for generating all possible passwords when given both a password length, as well as an allowable character set, e.g., all upper/lower case letters and numbers. The general strategy is to try all characters in all positions, and this strategy can be implemented iteratively or recursively. The following reports both types of algorithms that we developed to generate passwords.

3.1 The Iterative Password Generator

To implement an iterative password generator one can use what we call the *counting algorithm* (see Appendix A for C# code that implements the counting algorithm). To understand the counting algorithm for automated password generation, imagine the allowable character set is numbers only (0-9, or ten character in total), and that the password length is four. Under these conditions, there are a total of 10*10*10*10 possible passwords ($10{,}000=10^4$). The passwords would range from 0000 through 9999, or 0 through 10^4-1. Moreover, you could generate all those passwords by starting at zero and adding one until you reached 9999.

To generalize this idea to a character set of size C, with a password length N, you simply count from 0 to C^N-1, and then map the numbers to the characters in the character set. For example, suppose your character set was capital letters only (26 in all), with a password length of 4. You would write a program that counted from 0 to 26^4-1, which is 0 to 456975; then the challenge is to map 0 to AAAA, 456975 to ZZZZ, and to map all the numbers in between to their letter-sequence equivalents.

To map a number X to its letter-sequence equivalent, first create a character array of size C, e.g., *chars*, containing all the allowable password characters. Then, loop N times, each time applying the operation X modulo C. This operation yields a number from 0 to C-1 that represents the rightmost character in the sequence, which you can then use to index into *chars* to retrieve the actual letter. Finally, divide X by C and continue with the loop—the division shifts the next character into the rightmost position. In pseudocode, this mapping looks as follows (see Figure 1).

```
// given:
//   X, the number to convert to a password string
//   N, the password length
//   C, the number of characters in the password
//   chars, the array of allowable password characters

Password = ""
For i=1 to N
    ch = chars[X mod C]
    Password = Password + ch
    X = X / C
Next
```

Figure 1. Pseudo-Code for Converting a Number to a Character Sequence—See Appendix A for a Complete Implementation in C#

One problem with the counting algorithm is that the maximum integer value in most programming languages is 2^{32}-1. This in turn limits the maximum password size that the algorithm can generate. For example, suppose that the allowable character set includes upper and lowercase letters (52), numbers (10), and both a comma and a period (2) -- for a total of 64 characters. Under these conditions, the maximum password size that the counting algorithm can generate is five letters, because $64^5 = (2^6)^5 = 2^{30}$, which is $< 2^{32}$-1. However, a 6-character password would require a maximum integer of $(2^6)^6 = 2^{36}$, which exceeds the maximum integer in most programming languages.

3.2 The Recursive Password Generator

A recursive password generator can be written that is independent of the total number of password possibilities. Given an array of allowable password characters, e.g., *chars*, loop through each character (*ch*) in the array, recursively placing the character in each position (*pos*), until you have built up a character sequence (*pwd*) of the maximum size (*siz*). The result will be all passwords of a given size. For example, suppose the recursive procedure is named *GenerateAllPasswords* and you wanted all passwords of size eight. You would call the recursive procedure as follows: *GenerateAllPasswords("", 0, 8).* Figure 2 depicts the pseudocode for a recursive brute-force password generator.

```
GenerateAllPasswords(pwd, pos, siz)
    if (pos < siz)
       foreach (char ch in chars)
           GenerateAllPasswords(pwd + ch, pos + 1, siz);
       next
    else
       print pwd;
    end if
```

Figure 2. Pseudo-Code for Recursively Generating All Passwords (pwd) of a Given Length (siz) — See Appendix B for a Complete Implementation in C#

4. DISCUSSION

The main problem with any brute-force, automated password generator is that the number of possible passwords generated is literally astronomical in size. Just to put matters in perspective, there are reportedly 200 billion stars in our Milky Way galaxy. If the allowable password characters were the 92 printable symbols on a keyboard (52 upper/lower case letters + 40 numbers/punctuations), then a six-character password has more than 600 billion possibilities—more than the number of stars in the Milky Way! Even with a fast processor, the automated password generator is limited by the response time of the information system that it is sending passwords to. Thus, brute-force password generation should be used as a last resort for all but the shortest passwords.

There are a number of alternative password-recovery procedures to try prior to brute-force. For example, in their classic paper on password security, Morris and Thompson (1979) recommended a dictionary list, a name list, dictionary words spelled backwards, a list of first names from a mailing list, last names, street names, city names, all valid license plate numbers in a given state, room numbers, social security numbers, telephone numbers, and other kinds of personal numbers.

More recent research reports that two-thirds of student passwords are designed around personal characteristics, with the remaining one-third relating to relatives, friends or lovers; proper names and birthdays form the basis of more than half of all passwords (Brown, Bracken, Zoccoli, & Douglas, 2004). Thus another password recovery procedure is to obtain personal information about a user and attempt various combinations of this information. However, if these procedures fail, brute-force password generation is the last recourse.

REFERENCES

Brown, A. S., Bracken, E., Zoccoli, S., & Douglas, K. (2004, September). Generating and Remembering Passwords. *Applied Cognitive Psychology, 18*(6), 641-651.

Florencio, D., & Herley, C. (2007). A Large-Scale Study of Web Password Habits. In: *Proceedings of the 16th International World Wide Web Conference* (pp. 657-665). Banff, Alberta: University of Calgary.

Morris, R., & Thompson, K. (1979, November). Password Security: A Case History. *Communications of the ACM, 22*(11), 594-597.

Appendix A: The Basic Non-Recursive Password Generating Algorithm

```
using System;

namespace ConsolePasswordGenerator
{
    class Program
    {
        static void Main(string[] args)
        {
            char[] chars = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
'K', 'L', 'M',
                             'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z'};

            Console.Write("Password Length? ");

            int iPasswordLength = Convert.ToInt32(Console.ReadLine());
            int iPossibilities = (int) Math.Pow((double) chars.Length, (double)
iPasswordLength);
            Console.WriteLine("{0} words total", iPossibilities);
            for (int i = 0; i < iPossibilities; i++)
            {
                string theword = "";
                int val = i;
                for (int j = 0; j < iPasswordLength; j++)
                {
                    int ch = val % chars.Length;
                    theword = chars[ch]+theword;
                    val = val / chars.Length;
                }
                Console.WriteLine(theword);
            }
            Console.ReadLine();
        }
    }
}
```

Appendix B: The Basic Recursive Password Generating Algorithm

```
using System;

namespace RecursiveConsolePasswordGenerator
{
    class Program
    {
        static char[]
chars={'a','b','c','d','e','f','g','h','i','j','k','l','m',

'n','o','p','q','r','s','t','u','v','w','x','y','z'};

        static void GenerateAllPasswords(string pwd, int pos, int siz)
        {
            if (pos < siz)
            {
                foreach (char ch in chars)
                {
                    GenerateAllPasswords(pwd + ch, pos + 1, siz);
                }
            }
            else
                Console.WriteLine(pwd);
        }

        static void Main(string[] args)
        {
            Console.Write("Password Length?");
            int iPasswordLength = Convert.ToInt32(Console.ReadLine());
            GenerateAllPasswords("", 0, iPasswordLength);
            Console.ReadLine();
        }
    }
}
```