Session _____

Don't be a Brute: A Programming Assignment for Exposing Undergraduate Students to Computational Complexity and Heuristic Problem-Solving Approaches

Tom Wulf

College of Applied Science, University of Cincinnati

Abstract:

A programming assignment that was developed for an undergraduate course in computer security in which students simulate a brute force password guessing attack is described. This exercise helps students to understand issues of computability, re-enforces the basic mathematics of combinatorics, and provides a springboard for discussions of heuristic versus brute-force problem-solving approaches. In the context of a course on computer security, the assignment serves to tangibly demonstrate issues with password selection and user policies that apply to this issue.

Undergraduate students in Computer Science Technology and Information Engineering Technology do not receive the same training in the formal analysis of algorithms that students in standard theory based Computer Science programs do. It is clear, however, that IT students must develop a basic understanding of problem complexity issues and heuristic problem solving approaches to be successful in their careers. The exercise described in this paper gives students a hands-on feel for computational complexity through a programming assignment which simulates a brute-force, password-guessing attack.

Preliminaries: Class Lecture and Readings

In class lecture and assigned readings, students were first introduced to a brute-force password guessing approach in which every possible password string using a finite symbol set for strings of a specified length or range of lengths is generated. Testing for the actual password string then is simply an exhaustive linear search through this generated set. The first issue required in order to examine the magnitude of this task is to determine the cardinality of the set to be tested. This provides an opportunity to review the multiplication rule for the cardinality of permutation sets with the students. A hypothetical operating system that restricts the user to choose password strings of length 8 from the symbol set consisting of the uppercase letters is described and the cardinality of this password set is calculated as an example.

The next step is to consider real world operating systems which allow password strings composed

of a greater number of symbols including the lowercase letters, digits, and other non alpha-numeric symbols. This produces symbols sets with cardinalities of 52, 62, and 92.

In existing operating systems, password strings can be a range of lengths from some minimum (5) to some maximum (8 or more) and the symbol set has roughly 96 usable characters.[1] This fact naturally leads to a review of the formula for determining the sum of a geometric progression in order to determine the cardinalities of sets composed of a range of lengths.

Students are then introduced to (a review of) the concept of expected best, average and worst case algorithm performance. Intuitively, students should now understand that the brute-force attack will have to test about half of the possible password strings in the set before finding a match. It is useful to challenge the students to informally prove that the brute force guessing attack will actually succeed.

In the context of an undergraduate security course, user strategies for choosing a password string that maximizes the amount of computation that an attacker must perform to guess the password are then related to the cardinality of the various symbol sets. Rules for creating strong passwords are then reviewed. These arise directly out of the analysis of the symbol set size.[2]

A basic analysis of the feasibility of generating all the password strings for some of the larger password string sets is now presented to the students in an informal way. After showing that this is not feasible, the heuristic *dictionary attack* and the related *hybrid-dictionary* attack is presented to the students as a better algorithmic approach to guessing a password. [Stallings 1998] [Skoudis 2002]

Finally, a description of how cracking programs and attacks are actually conducted is explained. Many students initially assume that the attack will be done at the login prompt when it is most often conducted on a stolen copy of the system password file. [Skoudis 2002]

The Assignment:

In order to solidify student understanding of the concepts presented in lecture, students are given a programming assignment in which they are to develop a crude simulation of a brute-force password guessing attack by generating the entire set of password strings for some of the smaller symbol sets, recording the required times to do so, and then after calculating the average time required to generate a single password string extrapolating to estimate the time required to generate the larger sets. Results of the calculations are submitted electronically in a report in the form of an MS Excel spreadsheet along with the source code for the program. In order to do this,

[1] Recently, Microsoft Corporation described the use of 2-octet Unicode characters for system passwords. Here there are 65535 possible symbols for each character in the password! This was in the context of the annual OpenHack competition sponsored by E-week magazine. See http://www.eweek.com/category2/1,3960,600431,00.asp for details.

[2] Strong passwords should be at least 8 characters in length and contain each of the following: 1) Mix of upper and lowercase letters, 2) digits, 3) punctuation symbols. [Skoudis 2002]

students must understand the mathematics of the combinatorics presented in the lecture and have basic programming skills.

An iterative programming approach relying on nested fixed-loop control structures is sketched out in lecture as one solution to the problem and a recursive solution is also discussed.

For the assignment, students are not required to store the generated password strings, and merely display them in sequence to the console as they are generated using standard output functions. Students are permitted to use any programming language they wish. Some discussion about the validity of this approach as a crude simulation is presented as well as a general discussion about the validity of comparing the derived results from the use of different languages, (compilation vs. interpretation for instance), methods, and platforms.

Results:

The assignment proved very successful in giving students an appreciation for computational complexity. Some were intrigued enough by the discussion of the validity of comparing the results of different languages that they actually implemented additional solutions using different languages and compared the results of the various solutions. Other students submitted results from running the same solution on machines with different hardware capabilities and platforms.

One variation to this assignment might be to have the students work in teams to produce several such solutions, each using a different programming language and/or compiler and comparing the results in a more rigorous way.

Conclusions:

The assignment described above proved most successful for giving students a solid grasp of a less tangible, abstract mathematical concept.

References:

Skoudis, Ed Counter Hack: a Step by Step Guide to Computer Attacks and Effective Defenses, 2002 Prentice Hall

Skoudis, Ed The Hack-Counter Hack Training Course, 2002 Prentice Hall

Stallings, William Cryptography and Network Security Principles and Practice 2nd Ed., 1998 Prentice Hall