Ira A. Fulton Schools of Engineering
Arizona State University

CSE 539: Applied Cryptography

Hash Project

Purpose

In this project, you will employ cryptographic libraries to hash strings using the MD5 library. You will gain the experience necessary to use any of the modern hash functions in C#'s hashing library. With an MD5 hashing function, you will learn how to convert a string to bytes and salt that string for more secure hashing. You will also learn how to find collisions in a hash function using a birthday attack.

Objectives

Students will be able to:

- Justify salting password hashes.
- Employ the Birthday Paradox to find a collision in a hash function.

Technology Requirements

Programming assignments should be done using .NET Core 6.0. While you can write your code on a Windows or Mac machine, the autograder will test your code on Ubuntu 22.04. It is highly recommended to create a Virtual Machine for writing your assignments.

Information on how to install the .NET Core SDK on Ubuntu can be found on Microsoft's website: Install .NET Core SDK or .NET Core Runtime on Debian

You can create a new project using this command: dotnet new console --output project_name

You can also run your project by going into your project directory and using: dotnet run

Directions

The use of hashes typically comes with the assumption that they cannot be reversed. As such, they are the perfect fit for storing passwords. Passwords can be verified by hashing the input and comparing them to the stored hash. As hashing is deterministic, the same hash will always be generated. Unlike encryption, hashing is not designed to be reversible. If an attacker obtains the hash of the password, they should be unable to find the original password. However, rainbow tables have been created for popular hashing functions and passwords. Thus, hashed passwords that are part of the rainbow table can be quickly looked up. Furthermore, since hashing is deterministic, two users with the same password would have the same hash. This gives additional information to an attacker.

To mitigate these issues, a salt can be added to the password. Typically, a different salt is used per user, causing users with the same password to have different hashes. Furthermore, salted passwords are much less likely to be found in a rainbow table, increasing the security of the password hashes.

In this exercise, you will employ MD5, which is an outdated hashing algorithm. The goal of this exercise is to find collisions using a birthday attack. As the logic is the same, you will be evaluated using a hashing function with a smaller output space.

First, write a program in C# to find the MD5 hash of a string. Documentation for MD5 in C# is available here: MD5 Class (System.Security.Cryptography)

To convert a string to a byte array, use the Encoding.UTF8.GetBytes method. You can test your code using an online MD5 calculator. Strings converted using that method will produce identical hashes as an MD5 calculator.

A 1 byte salt will be passed to you as a command line argument. You should append this byte to the end of your byte array before hashing.

For example, suppose you are hashing the string “Hello World!” and are passed the salt C5.

Bytes before hashing: 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 C5

Bytes after hashing: E6 D9 B0 B9 D1 78 B2 40 02 89 EB EA 33 E8 B8 82 You can

compare this result to hashing only “Hello World!” to see the difference: Bytes before

hashing (only “Hello World!”): 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 Bytes after hashing:

ED 07 62 87 53 2E 86 36 5E 84 1E 92 BF C5 0D 8C.

You will now perform a birthday attack with a function for calculating a hash with salt. As the standard version of MD5 is time-consuming to attack, you should modify the hashing result to

make it easier to attack. Instead of using the full hash, only compare the first 5 bytes (in this case, “Hello World!” with the salt C5 hashes to E6 D9 B0 B9 D1).

You may choose the length of the string, but you may only use alphanumeric characters in your string [A-Z][a-z][0-9]. Your program should output two strings that hash to the same value with MD5 and the given salt. These strings should be separated by a comma.

Continuing from the earlier example, suppose that the byte C5 is passed in as the salt. Here is a sample command and output from the program:

> Command: dotnet run “C5”
>
> Sample output: AQJCMW0DGL,I95ORWB1A7

In this example, both values hash to 2B 68 3B 65 7A when salted with C5.

Note: There will be many different possible solutions. Your answers will be verified by hashing them.

Submission Directions for Project Deliverables

Compress your project folder into a .zip archive and submit that file. The autograder will run the following commands on your archive for testing:

unzip submission_name.zip

cd submission_name

dotnet run

Try these commands before submitting to ensure your submission is properly set up. You can add the command line arguments from the provided examples to verify that your program works correctly.

Important Note: Your folder name must match the submission name. For example, if Coursera asks you to submit a file named P2.zip, running the unzip command as shown should produce a folder named P2. If you do not follow this convention, the autograder will be unable to find your project.

Project 2 "submission_name": P2