Rainbow Table Generator Documentation

Introduction

The Rainbow Table Generator is a C program designed to create a rainbow table, a precomputed table for reversing cryptographic hash functions. This generator supports various hashing algorithms, including SHA1, SHA256, SHA512, and Blowfish. It uses a reduction function and iterative hashing to construct a rainbow table with specified parameters.

Usage

To utilize the Rainbow Table Generator, compile the provided C code and execute the resulting binary. The program will prompt you for input parameters, including the hashing algorithm, reduction length, character set range, password length, chain length, and the number of chains. These parameters will be discussed later in this document.

To compile the code, you need to download the sha.h header file and specify its location to your compiler. Failure to do so will result in an error. To obtain the header file, search for "sha.h" online and download it. If you can't find it online, follow this approach below as it is faster and what I used. Then, during code compilation, point the compiler to the downloaded header file (downloaded via the terminal). For instance, here is how I directed the compiler to the location where I downloaded OpenSSL with Homebrew on my Mac:

Step 1: brew openssl info
This command provides information and indicates whether OpenSSL is installed on your system. It also provides some caveats. If it is not installed, proceed to step 2.

Step 2: brew install openssl
This installs OpenSSL on your Mac. After installation, note the path where it was stored and copy that path. Also, remember the version you downloaded, as you will use it in the next step.

Step 3: In your VSCode terminal, use this command to compile the rainbow table code:
gcc your_code.c -o your_executable -I /usr/local/Cellar/openssl/{version}/include -L /usr/local/Cellar/openssl/{version}/lib -lssl -lcrypto

Note: Replace {version} with the version you installed. The path should be the one indicating where it was installed after successful installation on your system.

This is how mine was linked:

gcc -o rainbowTable rainbowTable.c -lcrypto -lssl -I /opt/homebrew/Cellar/openssl@3/3.2.1/include -L /opt/homebrew/Cellar/openssl@3/3.2.1/lib

Step 4: Run your executable binary file, e.g., ./rainbowTable, and your code should execute successfully.

Input Parameters

Hashing Algorithm: Choose from sha1, sha256, sha512, or blowfish.

Reduction Length: Specify the length of the reduction function.

Character Set: Enter the range of character sets used for generating random passwords.

Password Length: Define the length of the passwords in the rainbow table.

Chain Length: Set the number of iterations for each password in a chain.

Number of Chains: Determine the total number of chains to generate.

Algorithm and Hashing

The Rainbow Table Generator supports multiple hashing algorithms for securing passwords. It uses the OpenSSL library for hash functions, providing robust security. In this implementation, we are comparing four different Hashing Algorithms, namely: Sha1, Sha256, sha512, and Blowfish algorithms.

Length of Reduction Function

The reduction length is a crucial parameter that influences the structure and diversity of the generated rainbow table. It determines how many characters from the hash are considered when applying the reduction function to generate a new password in the chain.

Formula

The formula used to calculate the reduction index in the code is:

int reduction_index = j % reduction_length;

Here, j represents the current iteration in the chain, and % is the modulo operator, which returns the remainder of the division of j by reduction_length (giving by the user). This operation ensures that the reduction index stays within the bounds defined by the reduction length.

Explanation

Initialization: The process begins by initializing a random password and calculating its hash using a specified hashing algorithm.
Chain Generation Loop: The reduction function is applied in a loop for each iteration in the chain. The reduction index is calculated based on the current iteration, which ensures a cyclic pattern determined by the reduction length.
Applying Reduction Function: The reduction function takes the calculated reduction index, fetches a character from the hash, and combines it with the current iteration (j). This result is then used as an index to select a character from the character set. The selected character is appended to the reduced password.
Hash Update: After obtaining the reduced password, it undergoes hashing again, creating a new hash that serves as the starting point for the next iteration in the chain.
Chain Length Completion: This process repeats for the defined chain length, effectively creating a chain of passwords and hashes. Each password in the chain is generated by applying the reduction function to the previous hash.

Example
Let's illustrate with an example using a reduction length of 3 and a chain length of 4:

Initial Password: Randomly generated, let's say "abcde."
Initial Hash: Obtained by hashing the initial password.
Reduction Function Iterations:

Iteration 1: Reduction index = 1 (1 % 3), character selected from hash determines a new character from the character set, and the password is updated.
Iteration 2: Reduction index = 2 (2 % 3), another character is selected and added to the password.
Iteration 3: Reduction index = 0 (3 % 3), character selected, password updated.
Iteration 4: Reduction index = 1 (4 % 3), the process continues until the chain length is reached.
Caveats
The reduction length is one of the parameters that determine the complexity and uniqueness of the generated rainbow table. The tradeoff in the reduction length is that a shorter reduction length may result in faster table generation but reduced password diversity, while a longer reduction length may increase diversity but also increase computation time. It's a balance that depends on the specific use case and security requirements.

Reduction Function

In the process of generating rainbow tables, the reduction function serves as a fundamental component, composing the transformation of hashed values back into plaintext passwords. The root of this function lies in its formula: r(x) = x mod N, where N denotes the size of the input set.

Functionality Overview

The reduction function operates by extracting a portion of the hash and manipulating it to form a new password. The cyclic nature of the modulo operation ensures a repetitive yet deterministic pattern, crucial for the generation of coherent password chains.

Formula Explanation

x - Hashed Value: Represents the input hash generated from a previous iteration in the rainbow table chain.
N - Input Set Size: Signifies the size of the character set used for password generation. This could include alphanumeric characters, symbols, and other specified characters.
Modulo Operation (mod): The remainder of x divided by N is calculated, resulting in an index within the bounds of the character set.

Example Illustration
Hash: "f2a9c"
Character Set: "ABC"
Password Length: 4
Reduction Length: 3
Chain Length: 3
Number of Chains: 1
Iteration 1:

Password: "ABCA" (randomly generated)
Hash: "f2a9c"
Reduction Index Calculation: (0 + 0) % 3 = 0 (for the first iteration)
Reduction Function Calculation:
First character in Reduced Password: "A" (calculated using the reduction index)
Iteratively append characters until the Reduced Password is of length 4: "AAAA"
Hash the Reduced Password: "AAAA"

Update the hash for the next iteration.
Iteration 2:
Reduction Index Calculation: (1 + 1) % 3 = 2 (for the second iteration)
Reduction Function Calculation:
First character in Reduced Password: "A" (calculated using the reduction index)
Iteratively append characters until the Reduced Password is of length 4: "CCC"
Hash the Reduced Password: "CCC"
Update the hash for the next iteration.
Iteration 3:
Reduction Index Calculation: (2 + 2) % 3 = 1 (for the third iteration)
Reduction Function Calculation:
First character in Reduced Password: "C" (calculated using the reduction index)
Iteratively append characters until the Reduced Password is of length 4: "BCBC"
Hash the Reduced Password: "BCBC"

This character is then iteratively appended until the reduced password reaches the specified password length.

Note this is just an illustration. You can uncomment the debug line in the code to print how the logic works in the code if you want to visualize it.

The process continues until the chain length is reached, with each iteration involving the calculation of a new reduction index, applying the reduction function to form a reduced password until it reaches the specified password length, and then rehashing for the next iteration.

For every reduced password, it keeps adding characters until it matches the specified password length. Afterward, it hashes the newly formed reduced password to generate the next set of hashes. This process repeats according to the total chain length specified by the user.

Operational Flow
Initialization: Start with a randomly generated password and its corresponding hash.
Index Calculation: Compute the reduction index using the modulo operation, introducing a cyclic pattern based on the reduction length.

Character Selection: Pick a character from the hash, combining it with the current iteration.

Password Update: Append the selected character to the reduced password, gradually constructing a password of the specified password_length.

Hash Update: The reduced password undergoes hashing anew, yielding a fresh hash for the subsequent iteration in the chain.

Iterative Process: Repeat the cyclic process for the defined chain length, creating a coherent chain of passwords and their corresponding hashes.

The reduction function, summarized by the formula r(x) = x mod N, serves as a key mechanism in the generation of the rainbow tables. Its careful use guarantees that the generated password chains are varied and predictable, which is essential for successful password recovery approaches.

Code Implementation Snippet

```
for (int i = 0; i < num_chains; ++i) {
        char password[password_length + 1];
        generate_random_password(password, password_length);

        char hash[SHA512_DIGEST_LENGTH * 2 + 1];
        char current_hash[SHA512_DIGEST_LENGTH * 2 + 1];

        generate_hash(password, algorithm, hash);

        // Inside the main loop for chain iterations
        for (int j = 0; j < chain_length; ++j) {
            int reduction_index = j % reduction_length;
            int char_set_length = strlen(char_set);
            char reduced_password[password_length + 1];

            // Reset reduced_password at the start of each chain iteration
            reduced_password[0] = '\0';

            // Calculate the initial length of reduced_password
            int current_length = 0;

            // Iteratively append characters to reduced_password until it reaches
password_length
            for (int k = 0; k < password_length; k++) {
                // Calculate the reduction index based on the current iteration
                int current_reduction_index = (reduction_index + current_length) %
reduction_length;

                // Append the selected character to reduced_password
```

```
                reduced_password[current_length++] =
char_set[(hash[current_reduction_index] + j) % char_set_length];
                reduced_password[current_length] = '\0';  // Null-terminate the string

            // Hash the reduced_password for the current iteration
            generate_hash(reduced_password, algorithm, current_hash);

            // Update the hash for the next iteration
            strcpy(hash, current_hash);

            // Save the password and its final hash to the array
            strcpy(pairs[i].password, password);
            strcpy(pairs[i].hash, hash);
```

code Logic

The reduction_index is calculated, which is used to select characters from the hashed value. In the reduction function, the reduction function takes the reduction index and the current iteration count modulo the char_set_length to get the index from where it would pick the characters from the character set to form a reduced_password. The reduced password length would be exactly the same length as the password_length produced in the code before it is rehashed for the next iteration. This means the reduction function has to calculate and append the characters to the reduced password until it is equal to password_length. Then, it would save it as the reduced password, and the rehashing would be performed again until the chain length is achieved.

Character Set

The character set in password-related scenarios refers to the collection of characters that can be used to compose passwords. It includes uppercase and lowercase letters, numbers, and special symbols. The choice of a character set has implications for both security and usability. In a nutshell, the total input size used is 72 characters which span the character set below:

- Lowercase letters (a-z)
- Uppercase letters (A-Z)
- Numerical digits (0-9)
- Special symbols (!@#$%^&*()_+{}[];:'<>,.?/)

Security Implications

The comprehensive character set significantly enhances the complexity of generated passwords, making them more resistant to various password attacks, including brute-force and dictionary attacks. The inclusion

of special symbols contributes to the creation of strong and unpredictable passwords, crucial for robust security.

Usability Tradeoffs

While the broad character set ensures stronger passwords, it may pose challenges for users in terms of memorization. Users might find it more difficult to remember passwords containing a diverse set of characters.

Code Implementation:

The character set is explicitly defined in the code, and during password generation, characters are randomly selected from this set to create diverse and secure passwords.

Password Length

Password length is a critical factor in determining the strength of a password. It represents the number of characters in a password and is directly linked to its security.

Security Implications:

Longer passwords generally provide higher security, as they increase the number of possible combinations. Longer passwords are more resilient to brute-force attacks and are considered a fundamental aspect of robust password policies.

Usability Tradeoffs:

Longer passwords can be more challenging for users to type accurately and remember.

Chain Length

Chain length, in the context of password generation, typically refers to the number of iterations in the process of creating and hashing passwords within a chain.

Security Implications:

A longer chain length increases the complexity and diversity of generated passwords, making it more challenging for attackers to reverse-engineer the chain. Longer chains contribute to the creation of more secure password tables or rainbow tables.

Usability Tradeoffs:

Longer chains might require more computational resources and time during the initial setup and subsequent password generation processes.

Number of Chains :

The number of chains represents the total quantity of unique password chains generated by the system.

Security Implications:

A higher number of chains contributes to a larger pool of precomputed password-hash pairs, making it more difficult for attackers to crack passwords.

Usability Tradeoffs:

Generating and managing a large number of chains may require more storage and computational resources.

Output

The program outputs a file named "rainbow_table.txt" containing sorted password-hash pairs. Each line represents a password in the rainbow table along with its corresponding hash.

Conclusion :

The Rainbow Table Generator provides a flexible and customizable tool for generating rainbow tables. Also, I modified the code to ensure that each reduced password in a chain is consistent in length with the specified password length, enhancing the utility of the generated rainbow table. Balancing security and usability is crucial. Optimal choices for character set, password length, chain length, and the number of chains depend on the specific security requirements and user experience goals of the system.