Hashing: How do servers store passwords?

Topic Cryptography
Stimulus Computer Science

Contents

Introduction ..... 3.
Introduction to hashing ..... 4.
Hashing in detail ..... 5.
  Bitwise operations and syntax ..... 5.
  What is a “byte”? ..... 5.
  How do hash functions actually work? ..... 6.
Example hash function: Adler-32 ..... 7.
  Time complexity ..... 7.
Reflection ..... 9.
Bibliography ..... 10.

Introduction

How do servers store passwords?

The natural answer is a structure like this:

username	password
alice@email	sorry-bob-321

When a user attempts to log in, they transmit their credentials. If the provided password matches the password for the requested username, authentication succeeds.

While this works, there are glaring security issues with this model. Consider the consequences of a data leak: anyone with the data can now access every leaked user's accounts and the user's accounts on other servers, given how common non-unique passwords are.

To solve this conundrum, break the question into it's parts:

- Authentication involves the user inputting their username and password
- The server checks whether the received password is correct for the requested user

We know that we can compare the value of the sent password and the password in the database, but that involves the server knowing the password.

What if the value of the password could be stored without knowing the original password? To avoid the security problems, this encoded version would have to be non-reversible. A function like this would have a very useful property:

$$\text{if } f(a) = f(b), \text{ then } a = b \text{ and vice versa}$$

where f is the encoding function.

Introduction to hashing

A *Hash Function* is a function that maps inputs of any size to a fixed-size value. This output is called the *Digest*. This value has the property of being unique[1] for any given input. This value can be compared in place of the original input.

Using the popular hashing algorithm SHA-256, the above database can instead store the digest in the password field:

username	password
alice@email	eba9b6657ebb2ca702ba6feba680140a55f983523975a86ad8f7b812cab7be2c

Via hashing, the server never receives raw passwords from users. When the user inputs their password, the password is hashed with the same algorithm used by the server and is only then transmitted.

A *Cryptographic Hash Function* is a hash function which has additional security properties. For any length n, these include:

- The probability of any one digest being output by the function decreases exponentially as n increases (at least 2^{-n})
- It is unfeasible to derive the input string given a digest
- It is unfeasible to find 2 inputs that produce the same digest

Non-cryptographic hash functions are widely used for non-security-critical infrastructure. Common use cases include unique identifiers in a “hash map”, a datastructure that maps hashes of keys to values.

Cryptographic hash functions are widely used for digital security. Besides passwords, they are used to verify integrity. When a server sends a request to another computer, the response usually contains the hash of the response. If the hashed response does not match the provided hash, there is an issue with the integrity of the message. [1]

[1] To a degree. Depends on the particular algorithm and if it is designed for security or not. See the following Section for the distinction.

Hashing in detail

The "shape" of the input x of a hash function $f(x)$ is not the kind familiar to most mathematicians, numbers. The same hash function can be used on both text, images, and arbitrary datatypes in a computer program. This variety of uses is due to all of the listed uses being made of the same thing: *bytes*.

A bit can be thought of as an on/off switch or a boolean true/false. At the base level, computers are entirely built on such simple logic. Operations on bits are called *bitwise operations*.

Bitwise operations and syntax

In mathematical notation, the base of a number is expressed as a subscript. For example, the number 123 in base 2 is 1111011_2. In programming contexts, it is more often to logic around bitwise operations using hexadecimal (base 16). For this paper, only binary will be used. Numbers expressed without a base are considered to be base 10 (decimal).

Operation	Description	Symbol	Example
AND	Logical and	$\wedge$	$110_2 \wedge 011_2 = 010_2$
OR	Inclusive or	$\vee$	$1100_2 \vee 1001_2 = 1101_2$
XOR	Exclusive or	$\oplus$	$1100_2 \oplus 1001_2 = 0101_2$
NOT	Logical not	$\neg$	$\neg 1100_2 = 0011_2$
SHL	Shift bits left	$\ll$	$010_2 \ll 1 = 100_2$
SHR	Shift bits right	$\gg$	$010_2 \gg 1 = 001_2$
ROTL	Rotate bits left	$\lll$	$101_2 \lll 1 = 011_2$
ROTR	Rotate bits right	$\ggg$	$101_2 \ggg 1 = 110_2$

Table 3: A table of common bitwise operations

Leading zeros are significant for some operations. For example, $\neg 1_2 = 0$, but $\neg 01_2 = 10_2 = 2$. This is becase bitwise operations are (almost) always applied to a set of bits, not just individual ones. The NOT operation will flip all the leading zeros to ones, resulting in a different number.

The amount of digits can be considered the "size" of the set of bits. In a computing context, bits are usually addressed in terms of *bytes*.

What is a "byte"?

A *byte* is the fundamental unit of computer memory. For all modern architectures, a byte is 8 bits in size.

Most computers only allow adressing bytes, not bits. Any operations to read or write to bits are done as operations on a particular bit of a particular byte.

Bytes are how all computer data is stored. This is the encoding for all text, images, archives, documents, etc. that hash algorithms take as input. Because of this, it is the least-common denominator for all computer memory. As such, it is the input to the hash function.

How do hash functions actually work?

The kind of operations to perform on bytes to achieve good hashes are ones that:

- Will vary in value/bits after application
- Is hard to undo

Usually, mathematical operations are less optimal in that they usually have an inverse operation ($+$ and $-$, $\times$ and $\div$, x^n and $\log_x(n)$, etc.).

The modulus operator (expressed as mod or $\%$) is the remainder after integer division. It is useful for adding randomness to the hash while being difficult to undo, as there is no inverse to modulus.

Some example operations:

Operation	Input	Output	Comment
$\ll 3$	011001001_2	1100100100_2	Useful for mixing up the hash. Usually used in conjunction with other operations for more randomness
$\ggg 3$	011001001_2	001101001_2	Similar to SHR in purpose, except ROTR doesn't lose data when exceeding the size of the set of bits.
$\bmod 7$	011001001_2	000000101_2	Introduces a lot of small variations when combined with other operations (like bit shifts).

After any combination of mixing operations are performed on the hash for each byte, it is common to do a final bit shift or union to create the final digest.

Example hash function: Adler-32

Adler-32 is a hash function used for checksumming. It finds use in the compression library Zlib [2]. It can be expressed mathematically:

$$
\begin{aligned}
f(x) &= a_n \vee (b_n \ll 16) \\
&\text{where} \\
a_0 &= 1 \\
b_0 &= 0 \\
a_i &= (a_{i-1} + x_i) \bmod 65521 \\
b_i &= (b_{i-1} + a_i) \bmod 65521
\end{aligned}
$$

where n is the length of the sequence of bytes x and a and b are 32-bit unsigned integers.

Time complexity

In the domain of mathematics, there is no aspect of "execution time", only values. In a computing context, this algorithm is very inefficient.[2] Every time that a_i is calculated, a_{i-1} must also be calculated, and the same for b.

The calculations for $f(x)$ for $n = 2$ and $n = 3$:

n	2	3
Calcs.	$\ldots a_2 \vee (b_2 \ll 16)$ $\quad \ldots a_2 = (a_1 + x_2) \bmod 65521$ $\quad\quad \ldots a_1 = (a_0 + x_1) \bmod 65521$ $\quad \ldots b_2 = (b_1 + a_2) \bmod 65521$ $\quad\quad \ldots a_2 = (a_1 + x_2) \bmod 65521$ $\quad\quad\quad \ldots a_1 = (a_0 + x_1) \bmod 65521$ $\quad\quad \ldots b_1 = (b_0 + a_1) \bmod 65521$ $\quad\quad\quad \ldots a_1 = (a_0 + x_1) \bmod 65521$	$\ldots a_3 \vee (b_3 \ll 16)$ $\quad \ldots a_3 = (a_2 + x_3) \bmod 65521$ $\quad\quad \ldots a_2 = (a_1 + x_2) \bmod 65521$ $\quad\quad\quad \ldots a_1 = a_0 + x_1) \bmod 65521$ $\quad \ldots b_3 = (b_2 + a_3) \bmod 65521$ $\quad\quad \ldots a_3 = (a_2 + x_3) \bmod 65521$ $\quad\quad\quad \ldots a_2 = (a_1 + x_2) \bmod 65521$ $\quad\quad\quad\quad \ldots a_1 = a_0 + x_1) \bmod 65521$ $\quad\quad\quad \ldots b_2 = (b_1 + a_2) \bmod 65521$ $\quad\quad\quad\quad \ldots a_2 = (a_1 + x_2) \bmod 65521$ $\quad\quad\quad\quad\quad \ldots a_1 = (a_0 + x_1) \bmod 65521$ $\quad\quad\quad\quad \ldots b_1 = (b_0 + a_1) \bmod 65521$ $\quad\quad\quad\quad\quad \ldots a_1 = (a_0 + x_1) \bmod 65521$
#	8	13

The cost of this implementation is clear: not only does every computation of a_i require computing every previous one, every evaluation of b_i requires computing a_{i-1} for every b_i. Without state, the same a_i values are calculated multiple times.

Big-O Notation is a method for describing time complexity of algorithms in terms of a function O in terms of n. For example, the notation for constant time is $O(1)$ and logarithmic time is $O(\log(n))$. [3]

[2] *This* is inefficient. Other implementations can be much better time-wise.

Big-O notation disregards the actual time each calculation takes, instead representing the overall complexity. In the above example, the calculation $a_1 = (a_0 + x_i) \bmod 65521$ equals 1 calculation. Thus, Big-O notation is the amount of individual units of work (calculations) required to compute an algorithm for any size n.

The Big-O notation for the above function $f(x)$:

- a_i executes once for each i. This makes it $O(n)$.
- b_i executes a_i n times and itself n times.
- The overall $a_n \vee (b_n \ll 16)$ is a single calculation, so $O(1)$.

In total, $O(5(n-1)+3)$. Refactoring that, $f(x)$ has a time complexity of $O(5x-2)$, or $\begin{cases} O(1) \text{ if } n{=}0 \\ O(5x{-}2) \text{ else} \end{cases}$ if including the base case where no recursion is needed.

In practice, Big-O notation doesn't usually use this much precision in expressing complexity. This algorithm's complexity would be more likely to be expressed as being $O(n)$.[3]

[3]For the purpose of this paper, I treat the notation as a function.

Reflection

I had expected writing this paper to be easier, a confidence mistake I make often. I have existing knowledge of most of the topics covered here, so i assumed that it would be easy to figure out how to form a mathematical connection.

I originally planned on writing the math section on collision attacks on MD-5 (an algorithm that used to be used for cryptographic hashes that has since been proven unsafe), but in the process of writing I realized that was unfeasible. While there is a lot of existing literature on the topic, such as initial paper that described the attack [4], but it is much too long to integrate into a paper like this, and it is less connected to the kind of math that would be well-suited for this assignment.

For the future, I will attempt more research and testing before I start writing my paper so as to avoid this struggle.

In addition to what's mentioned above:

- I'd like to write in a way that flows better.
- I thought my function would be be non-linearly complex, turns out it's just linear. This led to my point about efficiency of algorithms to be less impactful.
- I could have connected time complexity to derivatives/integrals.

Overall, I am happy with my deepened understanding of the complexities of the software that backs every, even trivial, online interaction.

Bibliography

Sources not directly cited were used in the research of this paper.

[1] "Cryptographic hash function". Accessed: Nov. 22, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Cryptographic_hash_function

[2] "Adler-32". Accessed: Nov. 26, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Adler-32

[3] "Time complexity". Accessed: Dec. 17, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Time_complexity

[4] Tao Xie, Fanbao Liu, and Dengguo Feng, "Fast Collision Attack on MD5". Accessed: Nov. 22, 2023. Available: https://eprint.iacr.org/2013/170.pdf

[5] Wouter Penard and Tim van Workhoven, "On the Secure Hash Algorithm family". Accessed: Nov. 21, 2023. Available: https://web.archive.org/web/20160330153520/https://www.staff.science.uu.nl/~werkh108/docs/study/Y5_07_08/infocry/project/Cryp08.pdf

[6] "Anouncing the Secure Hash Standard". Accessed: Nov. 22, 2023. Available: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf

[7] "The cryptographic hash function SHA-256". Accessed: Nov. 22, 2023. Available: https://helix.stormhub.org/papers/SHA-256.pdf

[8] "MD5". Accessed: Nov. 23, 2023. [Online]. Available: https://en.wikipedia.org/wiki/MD5

[9] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner, "Secure Applications of Low-Entropy Keys". Accessed: Nov. 21, 2023. Available: https://www.schneier.com/wp-content/uploads/2016/02/paper-low-entropy.pdf

[10] Brad Conte, "crypto-algorithms". Available: https://github.com/B-Con/crypto-algorithms

[11] Theresa Maxino, "Revisiting Fletcher and Adler Checksums". Accessed: Nov. 26, 2023. Available: http://www.zlib.net/maxino06_fletcher-adler.pdf

[12] "Hash function". Accessed: Dec. 17, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Hash_function