**School of Computers and Information Engineering**

**STUDENT LEARNING ACTIVITY**

Fall Semester 2019

**Computer Algorithms (CIE3090)**

**Secure Password Hashing**

CA Learning Algorithm Assignment

Submitted by

U1710105, Shokirov Ismoil (Group#002)

U1710026, Khamraev Izzat (Group#002)

U1710115, Murodova Nozima (Group#002)

U1710040 Isroilova Azizabonu (Group#002)

U1710152, Muslimov Akhmadjon (Group#002)

U1710087, Akbarov Javohoir (Group#002)

|  |  |
| --- | --- |
| **Secure Password Hashing** | **1** |

**Secure Password Hashing**

What is Hashing?

Hashing is a type of algorithm which takes any size of data and turns it into a fixed-length of data. This is often used to ease the retrieval of data as you can shorten large amounts of data to a shorter string (which is easier to compare). For instance let’s say you have a DNA sample of a person, this would consist of a large amount of data (about 2.2 – 3.5 MB), and you would like to find out to who this DNA sample belongs to. You could take all samples and compare 2.2 MB of data to all DNA samples in the database, but comparing 2.2 MB against 2.2 MB of data cant take quite a while, especially when you need to traverse thousands of samples. This is where hashing can come in handy, instead of comparing the data, you calculate the hash of this data (in reality, several hashes will be calculated for the different locations on the chromosomes, but for the sake of the example let’s assume it’s one hash), which will return a fixed length value of, for instance, 128 bits. It will be easier and faster to query a database for 128-bits than for 2.2 MB of data.

Difference between hashing and encryption?

The main difference between hashing and encryption is that a hash is not reversible. When we are talking about cryptographic hash functions, we are referring to hash functions which have these properties:

* It is easy to compute the hash value for any given message.
* It is infeasible to generate a message that has a given hash.

|  |  |
| --- | --- |
| **Secure Password Hashing** | **2** |

* It is infeasible to modify a message without changing the hash.
* It is infeasible to find two different messages with the same hash.

The hash function should be resistant against these properties:

* Collisions (two different messages generating the same hash)
* Pre-image resistance: Given a hash h it should be difficult to find any message m such that h = hash(m).
* Resistance to second-preimages: given m, it is infeasible to find m’ distinct from m and such that MD-5(m) = MD-5(m’).

Existing Modern Hashing algorithms

Some hashing algorithms you may encounter are:

* MD-5
* SHA-1
* SHA-2
* SHA-3

**MD-5**

MD-5 is a hashing algorithm which is still widely used but cryptographically flawed as it’s prone to collisions. MD-5 is broken in regard to collisions, but not in regard of preimages or second-preimages. The first attacks on MD-5 were published in 1996, this was in fact an attack on the compression of MD-5 rather than MD-5 itself. In 2004 a theoretical attack was produced which allowed for weakening the pre-image resistance property of MD-5. In practice the attack is way too slow to be useful.

|  |  |
| --- | --- |
| **Hashing algorithms** | **3** |

**SHA**

SHA or Secure Hashing Algorithm is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS). Currently three algorithms are defined:

* SHA-1: A 160-bit hash function which resembles the earlier MD-5 algorithm. This was designed by the National Security Agency (NSA) to be part of the Digital Signature Algorithm. Cryptographic weaknesses were discovered in SHA-1, and the standard was no longer approved for most cryptographic uses after 2010.
* SHA-2: A family of two similar hash functions, with different block sizes, known as SHA-256 and SHA-512. They differ in the word size; SHA-256 uses 32-bit words where SHA-512 uses 64-bit words. There are also truncated versions of each standardized, known as SHA-224 and SHA-384. These were also designed by the NSA.
* SHA-3: SHA-3 is not yet defined. NIST is working on the exact parameters they will use; SHA-3 will be [Keccak](http://keccak.noekeon.org/), or “close enough”, but not necessarily the Keccak which was submitted (it is a configurable function, and they seem to want to tweak the parameters a bit differently than what was first proposed).

Note that while SHA-1 is “cryptographically broken” the properties we seek in a password hashing algorithm are still valid. In the real world finding a password hashing algorithm built on SHA-1 is still secure in the sense, that if it’s implemented there is no reason to assume it should be immediately changed to something newer.

Strong passwords

Apart from choosing a good hashing algorithm you should also force your users to choose a password which is built up of at least eight, random characters. Unfortunately people aren’t designed to remember and generate random sequences of characters. This is why we force our users to make passwords which contain numbers, letters, signs and at least one capital letter. But how does this help in regard to password hashing?

To attack hashed passwords there are different strategies:

* Dictionary Attacks
* Bruteforce

|  |  |
| --- | --- |
| **Strong passwords** | **4** |

* Rainbow Tables (generate everything upfront in a database and do a look up for each hash)

With a dictionary attack you will try to use word lists, these can consist of mostly used passwords, words, names, years, etc. For each word you will run the hashing algorithm and see if the generated hash is the same as the hash in the database. If this is the case then you know that the word from which you derived the hash is the password.

With a bruteforce attack you will try all possible combinations of characters. When using passwords of at least eight characters long, only using the ASCII characters set, there are 128^8 possibilities of passwords.

To show the importance of the length of a password:

These days, using a single, modern GPU, you can process about 10.323.000.000 passwords per *second* when bruteforcing plain MD-5. With this speed, when using a password of eight random characters, it will take about eighty days to generate every single possibility. This single GPU only costs about 500 USD (AMD Radeon 6990). [People have actually constructed clusters which contain 25 of these cards](http://arstechnica.com/security/2012/12/25-gpu-cluster-cracks-every-standard-windows-password-in-6-hours/), optimized it and managed to generate 350 *billion* passwords per second. This means they can generate all possible passwords of eight random characters long in less than two days.

Now when you add one character to the password, the possibilities will be 128^9. With previous calculation of 350 billion it will now take 305 days. 10 characters -> 106 years. This seems long, but we need to take into account Moore’s law:

Moore’s law is the observation that, over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years. The period often quoted as “18 months” is due to Intel executive David House, who predicted that period for a doubling in chip performance (being a combination of the effect of more transistors and their being faster).

|  |  |
| --- | --- |
| **Analysis and complexity** | **5** |

Computers have become faster and faster over the years, which is something we need to take into account. From a cryptographical point of view, 106 years is still a short period. We want infinity (something which will take several hundred-thousand to millions of years).

Hashing Passwords

#### Why do we hash passwords?

We hash passwords because in the event an attacker gets read access to our database, we do not want him to retrieve the passwords plain text. Remember often we store usernames, email addresses and other personal information in our database. Security rule #1 dictates that users need to be protected from themselves. We can make them aware of the risks, we can tell them not to re-use passwords, but we all know that in the end there will still be people who use the same password for their Facebook, Gmail, Linkedin and corporate email. What you do not want is that when the attacker gets his hand on your database, he immediately has access to all the above accounts (usernames/email addresses will be the same).

Hashing passwords is to prevent this from happening, when the attacker gets his hands on your database, you want to make it as painful as possible to retrieve those passwords using a brute-force attack. Hashing passwords will not make your site any more secure, but it will perform **damage containment** in the event of a breach.

|  |  |
| --- | --- |
| **Subject importance** | **999** |

Importance of password hashing

Storing user passwords is a critical component for any web application. When you store a user’s password, you must ensure that you have it secured in such a way that if your data is compromised, you don’t expose your user’s password.

There have been many high profile cases of websites and web applications that have had their user details compromised. This is made even worse when the developers of the website have not stored the user’s password in a secure way.

**What is the problem?**

When developing websites or web applications it is important to always have the outlook that everyone is out to get you. Generally you should assume that every point of your code is vulnerable to attack. You might be thinking, “there’s no way that anyone could get access to my database!”, but this is not the right attitude to have.

When you store a password in a database, you never want to store it in plain text. Storing a password in plain text would mean that anyone who looked through the database would be able to just read the user’s passwords.

Similarly, there are many ways to secure a password from prying eyes, but not all of them should be used.

If you do not take the right precautions when storing passwords, you could expose your user passwords to an attacker. Many people use the same email and password combination all over the Internet. If you expose their password, they could be left vulnerable on other websites or web applications as well.

It is your responsibility to take the right precautions!

**Hashing passwords**

“Hashing” passwords is the common approach to storing passwords securely. A “Hash” is a one-way function that generates a representation of the password. So when a user signs up for an account and they choose a password, the password is stored as the generated hash, rather than the actual characters that the user typed in. When you run a password through a hashing function, it will always produce the same output. When the user tries to log in with their email and password, the entered password is hashed again and then compared to what is stored in the database. If the two hashes are the same, the user has entered the correct password.

Hashes are impossible to convert back into plain text, but you don’t need to convert them back in order to break them. Once you know that a certain string converts to a certain hash, you know that any instance of that hash represents that string.

Hashing a password is good because it is quick and it is easy to store. Instead of storing the user’s password as plain text, which is open for anyone to read, it is stored as a hash which is impossible for a human to read.

Unfortunately, hashing a password is not nearly enough. It does not take very much computational power to generate a table of hashes of combinations of letters, numbers and symbols. Once you have this store of hashes, you can then compare the hash you want to crack and see if it matches. Once you find a match, you know the password.

**Salting**

In order to make it more difficult to expose a hash, you also need to salt it. Salting is where you add an extra bit of data to the password before you hash it. So for example, you would append every password with a string before hashing it. This would mean the string prior to hashing would be longer and therefore harder to find a match.

When the user comes to log back into your system, you simply take their entered password, append the salt and then hash it to see if it matches the hash you have stored.

**Why is Salting important?**

Salting is important because it adds a whole new level of required computational power in order to expose the hash. By adding a salt, you effectively render any lookup table useless. Hashing a password is not 100% secure as hashing alone is not that difficult to break. When you add a salt to the hash, you make it much more difficult to crack.

**What should I use as a Salt?**

Choosing the right strategy for salting is very important. Salting works best when the salt is completely unique for that user and for that instance of setting the password.

Many insecure systems do not use completely random salts. For example, I’ve seen systems that either use just the email address as a salt, or ever worse, they use the exact same string as a salt for every user. This is effectively pointless because once the salt is known for every user, it does not take much to generate a hash look up table. You also want the salt to be completely unique too. People use the same emails or usernames across many different websites. By using the email or username as the salt, you are opening the opportunity for a pre-made lookup table to be used.

Instead you should generate a unique salt for each user, and a unique salt whenever that user requires one, for instance when they change their password. There are many ways to do this which I will cover in the sample code at the end of this post.

**How should we store a Salt?**

You might be thinking, “If someone gets a hold of my hashed passwords, how can I also stop them from getting my salts? Where do I store them?”. The simple answer is, you can store your salt in the same User record as the User’s password. The added complexity of storing the salt in a different database is not worth the hassle.

If your user table is exposed by an attacker, having the salt and the hash still means they need an incredible amount of computational power in order to find the password. It is common practice to just store the two fields in the same table.

**Types of hashing algorithm**

There are a couple of different types of hashing algorithm that you should be aware of. Whilst hashing a password with any of these algorithms will render the password impossible to read for a human, you should still be careful about which one you choose because they are not all secure.

For a long time, MD5 was commonly used throughout the internet. However, MD5 is now accepted as being broken and too vulnerable to attack.

Another set of hashing algorithms is the SHA family.

Whilst it’s important to understand which hashing algorithms to avoid, I don’t think it is necessary to know the ins and outs of every single one. Generally you shouldn’t be making your own password encryption code because without expert knowledge of how the whole thing works you could be leaving yourself exposed. It’s much easier to just use a prewritten safe solution that I will show you below.

|  |  |
| --- | --- |
| **PPT slides** | **999** |

PPT Slides

|  |  |
| --- | --- |
| **Reference** | **999** |

Reference:

<https://www.loginradius.com/engineering/password-secure/>

<https://en.wikipedia.org/wiki/SHA-1>

<https://security.blogoverflow.com/2013/09/about-secure-password-hashing/>

<https://docs.oracle.com/cd/E26180_01/Platform.94/ATGPersProgGuide/html/s0506passwordhashing01.html>

**Index Page**

|  |  |
| --- | --- |
| **Title** | **Page No** |

* Problem Introduction
* Survey - Existing Solutions and Gap/Limitations
* Proposed Augmentation (if any)
* Proposed Algorithm (Solution)
* Analysis of Proposed Algorithm (complexity Analysis & comparison)
* Subject Importance
* Ppt slides
* Reference

