Spring Security

Table of Contents

[1 Spring Security Concepts 2](#_Toc78562972)

Resources used:

* Five Spring Security Concepts - Authentication vs authorization - Java Brains Brain Bytes: <https://www.youtube.com/watch?v=I0poT4UxFxE>
* Protecting passwords with Salt & Pepper | CISSP Answers: <https://www.youtube.com/watch?v=eicDtA9Yu-A>
* Adding Salt to Hashing: A Better Way to Store Passwords: <https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/>
* Passwords & hash functions (Simply Explained): <https://www.youtube.com/watch?v=cczlpiiu42M>

1. General Web Security Concepts
   1. Cryptography

Asd

* + 1. Hash Functions

They provide a mapping between an arbitrary length input, and a (usually) fixed length (or smaller length) output. It can be anything from a simple crc32, to a full blown cryptographic hash function such as **MD5** or **SHA1/2/256/512**. The point is that **there's a one-way mapping going on, they are ONE WAY functions.** It's **always a many:1** **mapping** (meaning there will always be collisions) since every function produces a smaller output than it's capable of inputting (If you feed every possible 1mb file into MD5, you'll get a ton of collisions). They are **hard** (or **impossible** **in practicality**) to **reverse**.

* + 1. Encryption Functions

They provide a **1:1 mapping** between an arbitrary length input and output**, TWO WAY function**. And **the result are always reversible**. The important thing to note is that it's reversible using some method. And it's always 1:1 for a given key. Now, there are multiple input:key pairs that might generate the same output (in fact there usually are, depending on the encryption function). Good encrypted data is indistinguishable from random noise. This is different from a good hash output which is always of a consistent format.

* + 1. Use Cases

Use a **hash function** when you want to compare a value but can't store the plain representation (for any number of reasons). Passwords should fit this use-case very well since you don't want to store them plain-text for security reasons (and shouldn't). But what if you wanted to check a filesystem for pirated music files? It would be impractical to store the 3 MB per music file in order to compare it later. So instead, take the hash of the file, and store that (md5 would store 16 bytes instead of 3MB). That way, you just hash each file and compare to the stored database of hashes (This doesn't work as well in practice because of re-encoding, changing file headers, etc, but it's an example use-case).

Use a **hash function** when you're checking validity of input data. That's what they are designed for. If you have 2 pieces of input, and want to check to see if they are the same, run both through a hash function. The probability of a collision is astronomically low for small input sizes (assuming a good hash function). That's why it's recommended for passwords. For passwords up to 32 characters, md5 has 4 times the output space. SHA1 has 6 times the output space (approximately). SHA512 has about 16 times the output space. You don't really care what the password was, you care if it's the same as the one that was stored. That's why you should use hashes for passwords.

**Hash** **functions** are also great for signing data. For example, if you're using HMAC, you sign a piece of data by taking a hash of the data concatenated with a known but not transmitted value (a secret value). So, you send the plain-text and the HMAC hash. Then, the receiver simply hashes the submitted data with the known value and checks to see if it matches the transmitted HMAC. If it's the same, you know it wasn't tampered with by a party without the secret value. This is commonly used in secure cookie systems by HTTP frameworks, as well as in message transmission of data over HTTP where you want some assurance of integrity in the data.

**Hashing** is useful if you want to send someone a file. But you are afraid that someone else might intercept the file and change it. So a way that the recipient can make sure that it is the right file is if you post the hash value publicly. That way the recipient can compute the hash value of the file received and check that it matches the hash value.

**Encryption** is good if you say have a message to send to someone. You encrypt the message with a key and the recipient decrypts with the same (or maybe even a different) key to get back the original message. Credits

Use **encryption** whenever you need to get the input data back out. Notice the word need. If you're storing credit card numbers, you need to get them back out at some point, but don't want to store them plain text. So instead, store the encrypted version and keep the key as safe as possible.

**A note on hashes for passwords:**

A key feature of cryptographic hash functions is that they should be very fast to create, and very difficult/slow to reverse (so much so that it's practically impossible). This poses a problem with passwords. If you store sha512(password), you're not doing a thing to guard against **rainbow tables** or **brute force attacks**. Remember, the hash function was designed for speed. So it's trivial for an attacker to just run a dictionary through the hash function and test each result.

**Adding a** **salt** helps matters since it adds a bit of unknown data to the hash. So instead of finding anything that matches **md5(foo)**, they need to find something that when added to the known salt produces **md5(foo + salt)** (which is very much harder to do). But it still doesn't solve the speed problem since if they know the salt it's just a matter of running the dictionary through.

Text

Description automatically generatedSo, there are ways of dealing with this. One popular method is called **key strengthening** (or **key stretching**). Basically, you iterate over a hash many times (thousands usually). This does two things. First, it slows down the runtime of the hashing algorithm significantly. Second, if implemented right (passing the input and salt back in on each iteration) actually increases the entropy (available space) for the output, reducing the chances of collisions. A trivial implementation is:

* The bottom line, hash(password) is not good enough. hash(password + salt) is better, but still not good enough... Use a stretched hash mechanism to produce your password hashes…

**Another note on trivial stretching**

Text

Description automatically generatedDo not under any circumstances feed the output of one hash directly back into the hash function:

* 1. Storing Passwords, Salt and Pepper

Passwords can be basically stored in 3 ways in a database:

* **Plain text**: This is really reaaally bad for obvious reasons.
* **Encrypted Passwords**: Encrypting the password with a secret key and storing the result in a database is also a bad idea. The encryption key needs to be stored somewhere on the system. Since Encrypting is two way, if an attacker has access to the key, then he can decrypt ALL the stored passwords of every user.
* **Storing the hash value**: This is the preferred way. Since hashing is one way, the resulting string cannot be reversed to access the original password. Not even the system will know the plain password yet we can still easily authenticate users by hashing their passwords and then comparing the resulting string to the entries in the database.

But there is another attack vector called **Rainbow Tables**.

Hackers gather all the leaked/stolen passwords into a big database, which is called a rainbow table. In this table, there are a ton of commonly used and also leaked password in plain text and the results of all the hashing algorithms hashing that password. An attacker could get access to a companies database. It will be extremely fast to compare the hash values of all the entries.

**Rainbow Table versus Dictionary/Brute Force**

A rainbow table is generally an offline only attack. In a brute force attack or dictionary attack, you need to spend time either sending your guess to the real system to running through the algorithm offline. Given a slow hashing or encryption algorithm, this wastes time. Also, the work being done cannot be reused.

* + 1. Protecting Passwords with Salt and Pepper

Salt and pepper is used to solve the Rainbow Table problem.

* **Salt**: A unique, non-secret value which is appended to the user’s password before it is hashed.   
  Salts are stored in the database (in plain text). Each password in the database will have its own, randomly generated salt value associated with it.
* **Pepper**: A secret value which is appended to the user’s password before it is hashed.  
  Peppers are not stored in the database. Peppers are NOT unique for each password. It can be thought as a secret key that is stored in a secure system file. It is typically the same value appended to every password.

Most hash functions are optimized for speed. A relatively new system (SHA3 – GTX 970) can do 300 Million hashed/second. The speed of a hash function is a positive thing in certain areas however when it comes to storing/hashing passwords, we don’t want the hash algorithm to be fast.

If an attacker accesses a DB with salts, he won’t be able to do his ms fast rainbow table lookups because the hashes will be different. When the salt is unique for each hash, we inconvenience the attacker by now having to compute a hash table for each user hash. To make it even slower, the best practice is to use a hash algorithm which is deliberately being slowed down. This creates a big bottleneck for the attacker.

Examples for such “slow” hashing algorithms are: **bcrypt**, **scrypt**, **argon2**

* bcrypt(password, salt, cost)

These algorithms take the password, the salt and a cost argument. The cost defines the number of rounds the algorithm goes through. This effectively slows it down. We can increase the cost over time to counter computers getting faster.

While the attacker may be able to crack one password, cracking all passwords will be unfeasible. Regardless, when a company experiences a data breach, it is impossible to determine which passwords could have been cracked and therefore all passwords must be considered compromised. A request to all users to change their passwords should be issued by the company right away. Upon password change, a new salt should be generated for each user as well.

* More details: <https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/>

A picture containing text, iPod

Description automatically generatedA really good protection is done if multiple of these techniques are used together.   
For example, Dropbox uses this multi-layer protection.

1. Get your password and hash it without a salt
2. They run the resulting value through a bcrypt algorithm, with a salt and a cost of 10 (slows down brute force attacks)
3. The resulting hash is then encrypted with AES (Advanced Encryption Standard). The encryption key is not stored in the database but kept separately.

If an attacker breaches the dropbox database, they will have to peel away each protective layer to get to the plain text password.

* Password Hashing: add salt + pepper or is salt enough?: <https://security.stackexchange.com/questions/3272/password-hashing-add-salt-pepper-or-is-salt-enough>
  1. Basic access authentication (Http Basic Auth)

In the context of an HTTP transaction, basic access authentication is a method for an HTTP user agent (e.g. a web browser) to provide a **user name** and **password** when making a request. In basic HTTP authentication, a request contains a **header field** of the form Authorization: Basic <credentials>, where credentials is the **base64** encoding of id and password joined by a single colon (:).

HTTP Basic authentication (BA) implementation is the simplest technique for enforcing access controls to web resources because **it does not require** cookies, session identifiers, or login pages; rather, HTTP Basic authentication uses standard fields in the HTTP header, removing the need for handshakes.

The BA mechanism provides **no confidentiality protection** for the transmitted credentials. They are merely encoded with Base64 in transit, but not encrypted or hashed in any way. Therefore, Basic Authentication is typically used in conjunction with HTTPS to provide confidentiality.

* Because the BA field has to be sent in the header of each HTTP request, the web browser needs to cache credentials for a reasonable period of time to avoid constantly prompting the user for their username and password. **Caching policy differs between browsers**. Microsoft Internet Explorer caches the credentials for 15 minutes by default.

HTTP does not provide a method for a web server to instruct the client to "**log** **out**" the user. However, there are a number of methods to clear cached credentials in certain web browsers. One of them is redirecting the user to a URL on the same domain containing credentials that are intentionally incorrect. However, this behavior is inconsistent between various browsers and browser versions. Microsoft Internet Explorer offers a dedicated JavaScript method to clear cached credentials:



**Server Side**

When the server wants the user agent to authenticate itself towards the server, the server must respond appropriately to unauthenticated requests. To unauthenticated requests, the server should return

* a response whose header contains a **HTTP 401 Unauthorized** status and
* a **WWW-Authenticate** field: The WWW-Authenticate field for basic authentication is constructed as following: WWW-Authenticate: Basic realm="User Visible Realm" The server may choose to include the charset parameter from RFC 7617: WWW-Authenticate: Basic realm="User Visible Realm", charset="UTF-8" This parameter indicates that the server expects the client to use UTF-8 for encoding username and password (see below).

**Client Side**

When the user agent wants to send authentication credentials to the server, it may use the **Authorization** field. The Authorization field is constructed as follows:

1. The **username** and **password** are combined with a **single colon** (:). This means that the username itself cannot contain a colon.
2. The resulting string is encoded into an octet sequence. The character set to use for this encoding is by default unspecified, as long as it is compatible with US-ASCII, but the server may suggest use of UTF-8 by sending the charset parameter.
3. The resulting string is encoded using a variant of **Base64**.
4. The authorization method and a space (e.g. "Basic ") is then prepended to the encoded string.

For example, if the browser uses Aladdin as the username and OpenSesame as the password, then the field's value is the base64-encoding of Aladdin:OpenSesame, or QWxhZGRpbjpPcGVuU2VzYW1l. Then the Authorization header will appear as:



**URL Encoding**

A client may avoid a login prompt when accessing a basic access authentication by prepending **username:password@** to the hostname in the URL. For example, the following would access the page index.html at the web site www.example.com with the secure HTTPS protocol and provide the username Aladdin and the password OpenSesame credentials via basic authorization:

This has been deprecated by RFC 3986: Use of the format "user:password" in the user info field is deprecated. **Some modern browsers therefore no longer support URL encoding of basic access credentials**. This prevents passwords from being sent and seen prominently in plain text, and also eliminates (potentially deliberately) confusing URLs.

Asd

asd

asd

asd

1. Spring Security Concepts
   1. Authentication

Knowledge Based

Possession Based

Multi Factor = Knowledge Based + Possession Based

* 1. Authorization
  2. Principal
  3. Granted Authority
  4. Roles

Asd

asd