More on Password Encoding Algorithms

Password encoding algorithms are fundamental for securely storing passwords within a system. They convert plaintext passwords into a hashed or encrypted format to enhance security.

Hashing

Hashing solves the problem of immediate access to the system with exposed passwords.

Hashing is a one-way function that converts the input to a line of symbols. Usually, the length of this line is fixed.

If the data is hashed, it's tough to convert the hash back to the original input, and it's also tough to find the input to get the desired output.

We have to hash the password in two cases:

- When the user registers in the application, we hash the password and save it to the database.
- When the user wants to authenticate, we hash the provided password and compare it with the password hash from the database.

Salting the Password

- A salt is a sequence of randomly generated bytes that is hashed along with the password. The salt is stored in the storage and doesn't need to be protected.

- Whenever the user tries to authenticate, the user's password is hashed with the saved salt, and the result should match the stored password.

- Since the salt is not a secret, attackers can still start a brute-force attack.

- A salt can make the attack difficult for the attacker, but hardware is getting more efficient. We must assume fast-evolving hardware with which the attacker can calculate billions of hashes per second.

Password Hashing Functions

Password hashing functions are cryptographic algorithms designed to securely convert passwords into a fixed-length, irreversible string of characters. Here are some widely used password-hashing functions:

1. **BCryptPasswordEncoder:** Bcrypt is a popular adaptive hashing function designed for password hashing. It incorporates salt and a cost factor to make the hashing process slow and computationally intensive, thereby increasing the difficulty of brute-force attacks.

   Properties:
   1. Slow hashing with a tunable cost factor.
   2. Generates salt automatically.

```
BCryptPasswordEncoder bCryptPasswordEncoder= new BCryptPasswordEncoder();
String encodedPassword = bCryptPasswordEncoder.encode(plainPassword);
```

2. **PBKDF2 (Password-Based Key Derivation Function 2):** PBKDF2 applies a hash function multiple times to the password and a salt to generate a secure hash. It allows iteration counts to increase the complexity and, therefore the security of the hash.

   Properties:
   1. Iterative hashing with a customisable iteration count.
   2. Uses salt to prevent rainbow table attacks.

```
String secret = "MySecret"; // secret key used by password encoding
int iterations = 500;  // number of hash iteration
int hashWidth = 512;      // hash width in bits

Pbkdf2PasswordEncoder pbkdf2PasswordEncoder =
  new Pbkdf2PasswordEncoder(secret, iterations, hashWidth);
pbkdf2PasswordEncoder.setEncodeHashAsBase64(true);
String encodedPassword = pbkdf2PasswordEncoder.encode(plainPassword);
```

3. **Argon2:** Argon2 is a memory-hard hashing function that provides resistance against GPU and ASIC attacks. It allows for efficient memory usage, making it challenging for attackers to crack hashed passwords.

   Properties:
   1. Memory-hard function, resistant to parallel computation attacks.
   2. Provides configurable memory and iteration parameters.

```
int saltLength = 32; // salt length in bytes
int hashLength = 8; // hash length in bytes
int parallelism = 2; // currently not supported by Spring Security
int memory = 512;   // memory costs
int iterations = 5;

Argon2PasswordEncoder argon2PasswordEncoder = new Argon2PasswordEncoder(
  saltLength,
  hashLength,
  parallelism,
  memory,
  iterations);
String encodePassword = argon2PasswordEncoder.encode(plainPassword);
```

4. **SCryptPasswordEncoder:** Scrypt is a password-based key derivation function designed to be more secure against hardware-based attacks than many other password-hashing algorithms. It was specifically designed to be "memory-hard," making it more costly to perform large-scale custom hardware attacks by requiring a significant amount of memory.

```
int cpuCost = (int) Math.pow(2, 14); // factor to increase CPU costs
int memoryCost = 8;       // increases memory usage
int parallelization = 1; // currently not supported by Spring Security
int keyLength = 32;       // key length in bytes
int saltLength = 64;      // salt length in bytes

SCryptPasswordEncoder sCryptPasswordEncoder = new SCryptPasswordEncoder(
  cpuCost,
  memoryCost,
  parallelization,
  keyLength,
  saltLength);
String encodedPassword = sCryptPasswordEncoder.encode(plainPassword);
```

References:

1. Bcrypt Password Encoder
2. Pbkdf2 Password Encoder
3. Argon2 Password Encoder
4. Scrypt Password Encoder