Security

The Ideas of Security

Information Security

As software engineers, we must understand information security:

- Data is digitized, stored, and shared.
- Digital information can be easily leaked, hacked, or altered, it requires strong protection.
- Software tools are using security tools extensively.

CIA Triad

We have a principle in information security called the **CIA Triad**.

The CIA stands for:

- **C**onfidentiality
- **I**ntegrity
- **A**vailability

Understanding Each Component

- **Confidentiality**: Only authorized people can see or change the data.
- **Integrity**: Data stays accurate and unmodified.
- **Availability**: Users can access data anytime they need it easily.

Example: ATM and Bank Software

- **Confidentiality**: Only password holders can access accounts.
- **Integrity**: Keeps transaction records correct.
- **Availability**: The ATMs are everywhere and accessible whenever they need them.

Together, these form the **CIA Triad**—the key principles of information security.

One more Factor: Authenticity

The **CIA Triad** (Confidentiality, Integrity, Availability) explains how to keep data safe —
but in the real world, we also need **Authenticity**.

Authenticity means we can be sure **who** sent the data or message, and that it really came from the right person — not a fake or hacker.

Simple Example

Imagine your friend sends you a message:

> "Let's meet at 5 PM!"

- If it's really your friend, that's fine.
- But what if someone pretends to be your friend and sends the same message?
- You need a way to **verify the sender** — that's **authenticity**.

We need CIAA in a Real-World

- **Confidentiality** → Only the right people can read it.
- **Integrity** → No one has changed it.
- **Availability** → It’s there when you need it.
- **Authenticity** → You know *who* it came from.

Integrity Tools: Hash, Salt, & HMAC

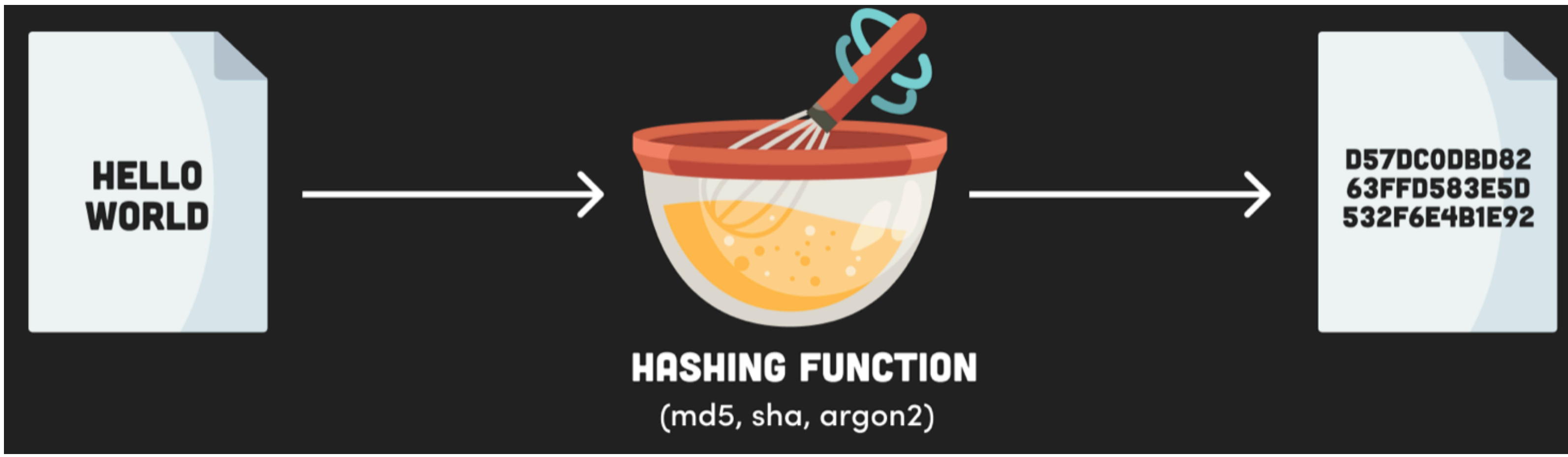
1. Hash: The Magic Blender

Imagine you put a cookie into a *magic blender*: Once blended, you get a unique cookie dust — but you can **never get the cookie back!**

If even one sprinkle changes, the dust looks totally different — so we know if someone messed with it!

> That's what a **hash** does: it turns data into secret code that can't be reversed.

Hash means "chop and mix," and that describes what a hashing function does.



- **Hashing** (e.g., SHA) turns data into a unique, fixed-length code.
- The same input always gives the same result, so it's used to check if two values match (like passwords).

Code Example: Using Hash

There are many algorithms available for hashing functions, but we use SHA256 in this example.

Hash is used for Integrity — making sure nothing has changed.

```javascript
const { createHash } = require('crypto');

function hash(input) {
    return createHash('sha256').update(input).digest('hex');
}

let password = 'hello';
const hash1 = hash(password);
console.log(hash1);

const hash2 = hash(password);
console.log(hash2);

const hash3 = hash(password + "2");
console.log(hash3);

console.log(hash1 == hash2); // true
console.log(hash2 == hash3); // false
```

Why Hashing Helps

Let’s see what happens in your code first:

```
const hash1 = hash('hello');  // always same result
const hash2 = hash('hello');  // same again
const hash3 = hash('hello2'); // different
```

- The same input always makes the same hash.
- That’s great for checking passwords — but also a big clue for hackers.

Why Rainbow Tables Are Dangerous

A rainbow table is like a giant cheat sheet of common passwords and their hash results.

For example:

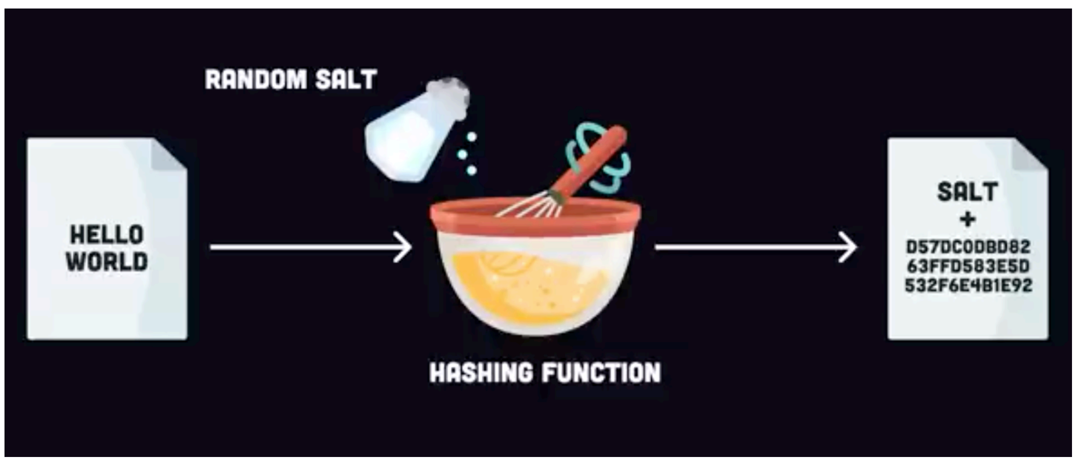
```
Password:  SHA256 Hash
123456:    8d969eef6ecad3c29a3a629280e
xpassword: 5e884898da28047151d0e56f8d7
hello:     2cf24dba5fb03232r3fsfsdfsdf
...
```

So if hackers steal a database of hashed passwords, they can look up each hash in their rainbow table to find the original password!

2. Salt: The Secret Ingredient

That’s why we use Salt — a random value added before hashing:

> A **salt** is a random string that is added to the input before hashing.
> This makes the hash more unique and harder to guess.



Code Example: Generating Secure User Info

This is the `signup()` function to return user information with email and password (salt + hashedPassword).

```
const { randomBytes, scryptSync } = require('crypto');

const users = [];

function signup(email, password) {
    const salt = randomBytes(16).toString('hex');
    const hashedPassword = scryptSync(password, salt, 64).toString('hex');
    const user = {
        email: `${email}`,
        password: `${salt}:${hashedPassword}`
    };
    users.push(user);
    return user;
}
```

When a user **signs up**, their password isn't saved directly; Instead, the program:

- Creates a random **salt** (like a secret spice).
- Mixes the password and salt → runs through **scryptSync** (a hashing algorithm).
- Saves both the salt and the hashed result, not the real password.

Salt Usage Example: Login Function

This is a login function with an email address and password.

- This function checks if the given email and password generates the same hashedPassword stored in the users list.

```
function login(email, password) {
    const user = users.find(v => v.email === email);
    const [salt, pw] = user.password.split(':');

    // Use the salt to make hash from the given password
    const created_pw = scryptSync(password, salt, 64).toString('hex');

    return created_pw === pw;
}
```

When the user **logs in**, it repeats the process:

- Takes the stored salt and the new password.
- Hashes them together again.
- If the new hash matches the saved one → correct password!

Why This Prevents Rabinbow Table Attack?

If hackers steal the `users` list:

```
{ email: "me@example.com", password: "f2a1...:7a9b..." }
```

They don’t have the real password — only the salt and the hash.

Without knowing the original password:

- They can’t reverse the hash (it’s one-way).
- Hackers can't reuse pre-made rainbow tables; they could build one for a specific salt, but it’s extremely expensive and practically useless.

HMAC or MAC

Salt protects *stored data* (like passwords in a database), but what about *messages sent across the network*?

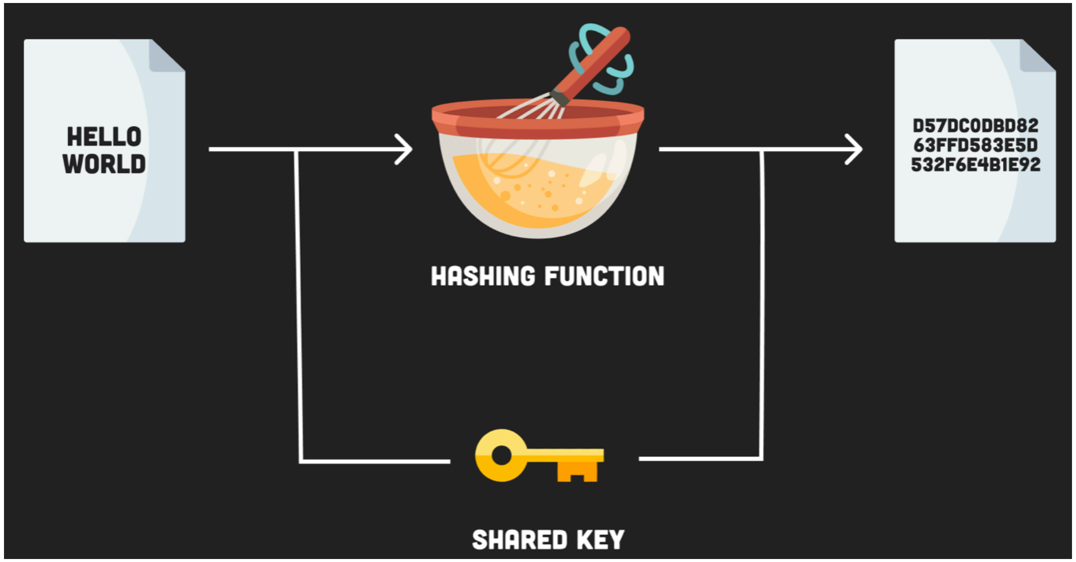
- If someone changes the message while it’s traveling, the salt and hash won’t stop it — the receiver can’t tell who sent it or if it was changed.

Key Differences

- **Hash and Salt** focus only on **Integrity**
- **HMAC** provides both **Authenticity** and **integrity** (verifies the originator of the data)

HMAC (Hash-based Message Authentication Code) is a keyed hash of data - like a hash with a password.

- HMAC is also called MAC.
- HMAC uses a different key to produce a different output.



Code Example: Using HMAC

In this example, we send a message with HMAC. A sender and a receiver share the password.

Sender Side:

```
const { createHmac } = require('crypto');

const password = 'hello!';
const message = 'hello world';

// Generate HMAC
// We need to generate the same hash only when we know the password
// Server can be sure the hmac hash is correct only when
// the sender has the correct password
const hmac = createHmac('sha256', password)
    .update(message)
    .digest('hex');

// Send message and hmac
```

Receiver Side:

The receiver already has a shared key, so the receiver can generate the HMAC from the key and the message.

- The receiver can compare the received HMAC and generated HMAC to check if the message is compromised or not (integrity check).

```
// message and hmac are received
const hmac_created = createHmac('sha256', password)
    .update(message)
    .digest('hex');

if (hmac_created === hmac) {
    console.log(`${message} is not compromised.`);
} else {
    console.log(`${message} is compromised.`);
}
```