

DATA SECURITY

# WHAT WE'LL COVER TODAY

- SQL Injection: More on what it is and how to avoid it
- Hashing
- Encryption:
  - Symmetric Key Encryption
  - Asymmetric Encryption
- Encryption using Java

# MORE ABOUT SQL INJECTION

**SQL injection** occurs when untrusted data such as user data from application web pages are added to database queries, materially changing the structure and producing behaviors inconsistent with application design or purpose.

Clever attackers exploit SQL injection vulnerabilities to steal sensitive information, bypass authentication or gain elevated privileges, add or delete rows in the database, deny services, and in extreme cases, gain direct operating system shell access, using the database as a launch point for sophisticated attacks against internal systems.

# HOW DOES SQL INJECTION WORK?

Using our UnitedStates database, let's say we let someone enter a city id and then display the information for the city using this SQL String:

```
String query = "SELECT * FROM city WHERE city_id = " + cityId;
```

If the user enters **237** for the customerId we get this SQL:

```
SELECT * FROM city WHERE city_id = 237;
```

If you run this in PgAdmin, you will get the row for the city with `city_id` 237.

# HOW DOES SQL INJECTION WORK?

But what happens when the user enters **237 OR 1 = 1** ?

```
String query = "SELECT * FROM city WHERE city_id = " + cityId;
```

Gives us this SQL:

```
SELECT * FROM city WHERE city_id = 237 OR 1=1
```

What happens if you execute this query?

# HOW DOES SQL INJECTION WORK?

But what happens when the user enters

**237; DELETE FROM park\_state ?**

```
String query = "SELECT * FROM city WHERE city_id = " + cityId;
```

Gives us this SQL:

```
SELECT * FROM city WHERE city_id = 237; DELETE FROM park_state
```

What happens if you execute this query?

# HOW CAN WE PREVENT SQL INJECTION?

- **Parameterized Queries**

- The single most effective thing you can do to prevent SQL injection is to use parameterized queries. If this is done consistently, SQL injection will not be possible.

- **Input Validation**

- Limiting the data that can be input by a user can certainly be helpful in preventing SQL Injection, but is by no means an effective prevention by itself.

- **Limit Database User Privileges**

- A web application should always use a database user to connect to the database that has as few permissions as necessary. For example, if your application never deletes data from a particular table, then the database user should not have permission to delete from that table. This will help to limit the damage in the case that there is a SQL Injection vulnerability.

# PROTECTING SENSITIVE DATA

Many data breaches involve sensitive data that has not been stored in a safe manner, meaning if the attacker gets to the data they can easily understand it.

There are many cases in which we need to be able to store data such that it is not readable by unauthorized parties:

- Passwords
- PIN
- Credit card numbers
- Bank account numbers

Depending on the nature of the **data** and how it's used, it **can be protected using** one of two techniques: **hashing** or **encryption**.



# PASSWORDS

One type of sensitive data systems need to store is user passwords. If passwords are kept in plain text, anyone with internal access can see them and if the database gets breached, hackers would also see the credentials in plain view.

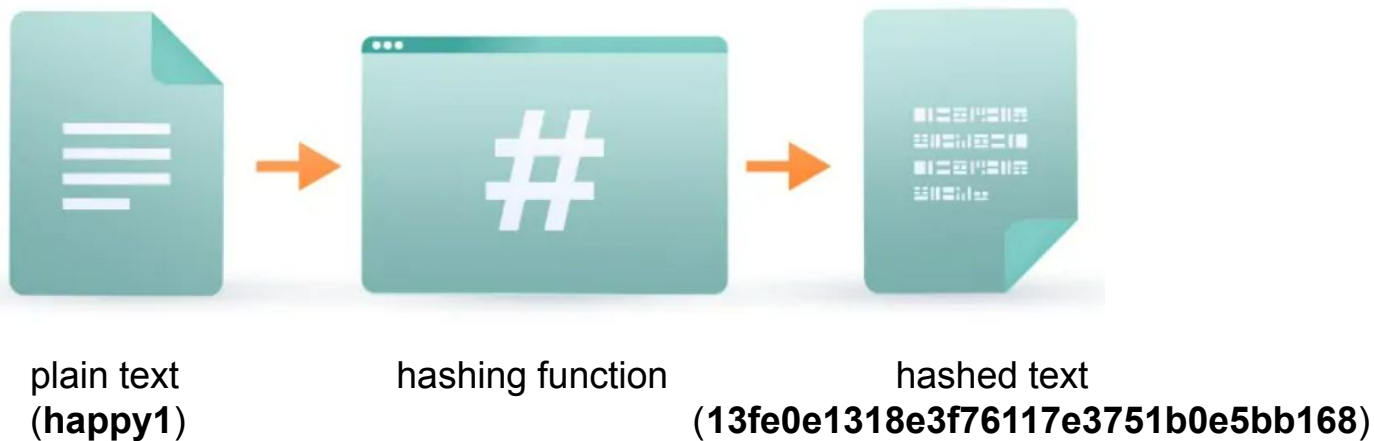
username	password
alice	happy1
bob	password123
jason	qwerty
mike	happy1

# PASSWORDS

- We need to be able to **verify** a password but not **recover** it.
- A system administrator with access to credential data should not be able to determine a password.
- Any hacker that steals a database or set of credentials should not be able to read the passwords.
- Even with super-computing capabilities, no one should be able to access the data within any reasonable amount of time.

# HASHING

- Use a **one-way function to scramble** the plain-text password prior to storage.
- Transforms the password into a completely different string of characters with a set length.



# HASHING

- When you log into your account, the password you enter is hashed using the password hashing function.
- The hashed password is compared to the hash stored in the database for your user account.
- If the password hash matches, you're granted access to your account.

username	password_hash
alice	3b2ba79307ae54030de4185d09171a9e
bob	86861c4aec4eb67df463fb0fe96a3127

**bob, password123 -> 86861c4aec4eb67df463fb0fe96a3127**



# CRACKING A HASH

- If you run a specific word like 'java' through a hashing function, the hashed output will always be the same for that word.
- Hackers can use software to generate hashes for millions of password guesses by running them through known hashing functions. A password which is a direct entry in the English dictionary is especially vulnerable.
- Once a database is breached, the hackers can compare the password hash values from the database to hashes from their password guesses.
- If a match is found then the system can be accessed.
- These attacks are known as “dictionary attacks”.
- Adding “salt” to a hash helps defend against these types of attacks.

# NEEDS SALT

- The output of a hash function is deterministic; when given the same input, the same hash is always produced.
- If two users have the same password their hash is also the same.
- Once a password is known, the same password can be used to access all the accounts that use that hash.

username	password_hash
alice	3b2ba79307ae54030de4185d09171a9e
bob	86861c4aec4eb67df463fb0fe96a3127
jason	3b2ba79307ae54030de4185d09171a9e

**If Alice's password is 'happy1', what do you think Jason's password is?**

# SALTING A HASH

- To limit the possibility of a successful dictionary attack, we salt the passwords.
- **Salt** is a generated value that is added to the input of hash functions to create unique hashes for every input.
- A salt makes a hash function look non-deterministic; when given the same input, the same hash is not produced.


username	password_hash	salt
alice	060b5087575363068163e6e39e0f7b5f	f1nd1ngn3m0
bob	34285ac3f310665d46be96563330df6f	f1nd1ngn22o
jason	08af9b4a340e6367f0793e75eadac596	f1nd1ngn4z1

# SALTING A HASH

- To limit the possibility of a successful dictionary attack on passwords.
- **Salt** is a generated value that is added to the password before hashing to create unique hashes for every input password.
- A salt makes a hash function look non-deterministic. For the same input, the same hash is not produced.

Alice and Jason still have the same 'happy1' password but appending a different salt value to the password before hashing gives us different password hashes.

username	password_hash	salt
alice	060b5087575363068163e6e39e0f7b5f	f1nd1ngn3m0
bob	34285ac3f310665d46be96563330df6f	f1nd1ngn22o
jason	08af9b4a340e6367f0793e75eadac596	f1nd1ngn4z1





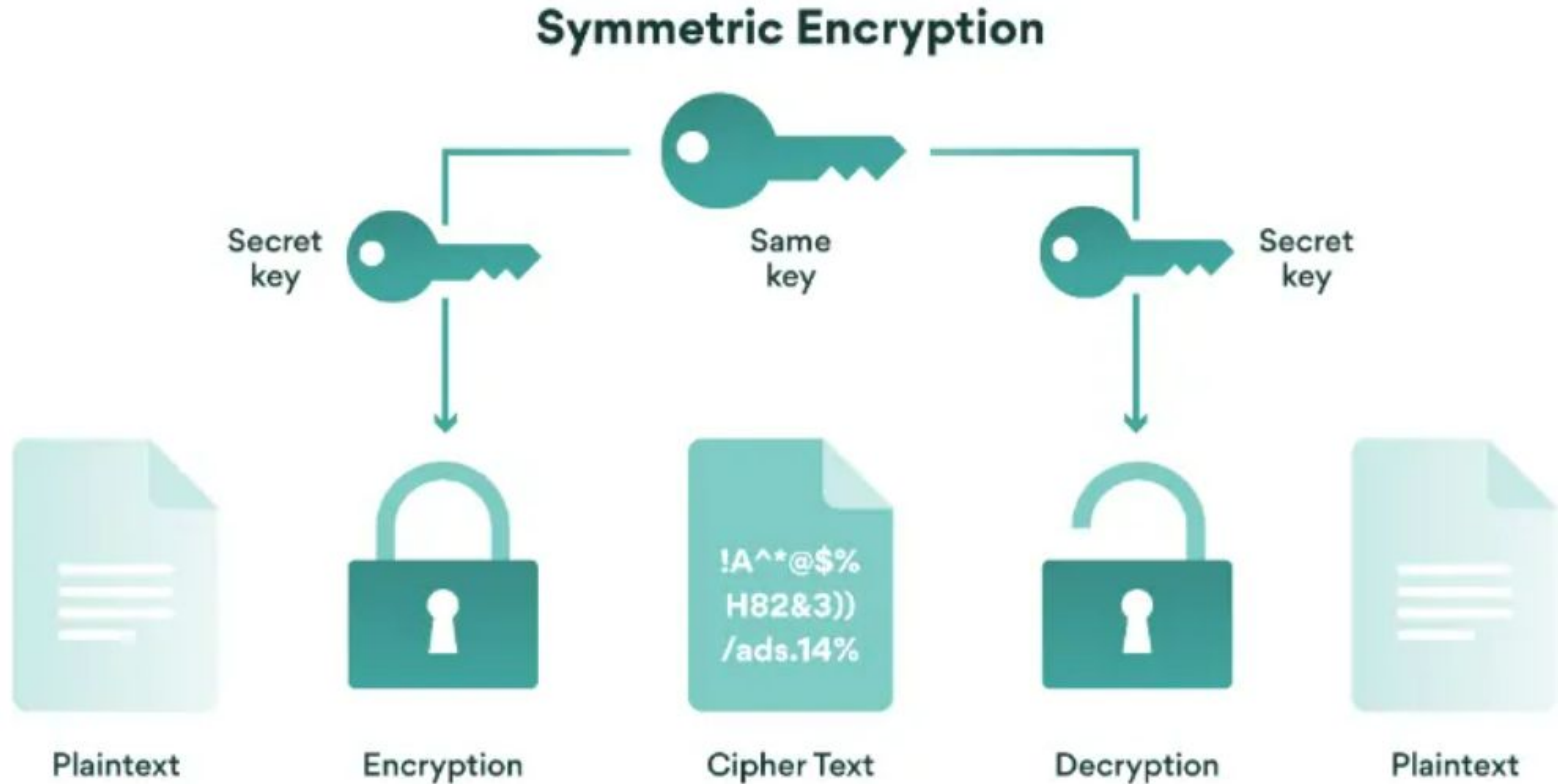
# ENCRYPTION

- Encryption is the most effective way to achieve data security.
- Information is encoded in such a way that only authorized parties can gain access to it.
- A combination of keys and algorithms is used to encode and decode sensitive information.

# SECURING DATA AT REST

- Data at rest can use a form of encryption called **symmetric key encryption**
- Also known as private key encryption.
- Requires both parties to use a private key to encrypt and decrypt data.
- Any party possessing the key can read the data.
- Difficult to secure the symmetric key between multiple parties.

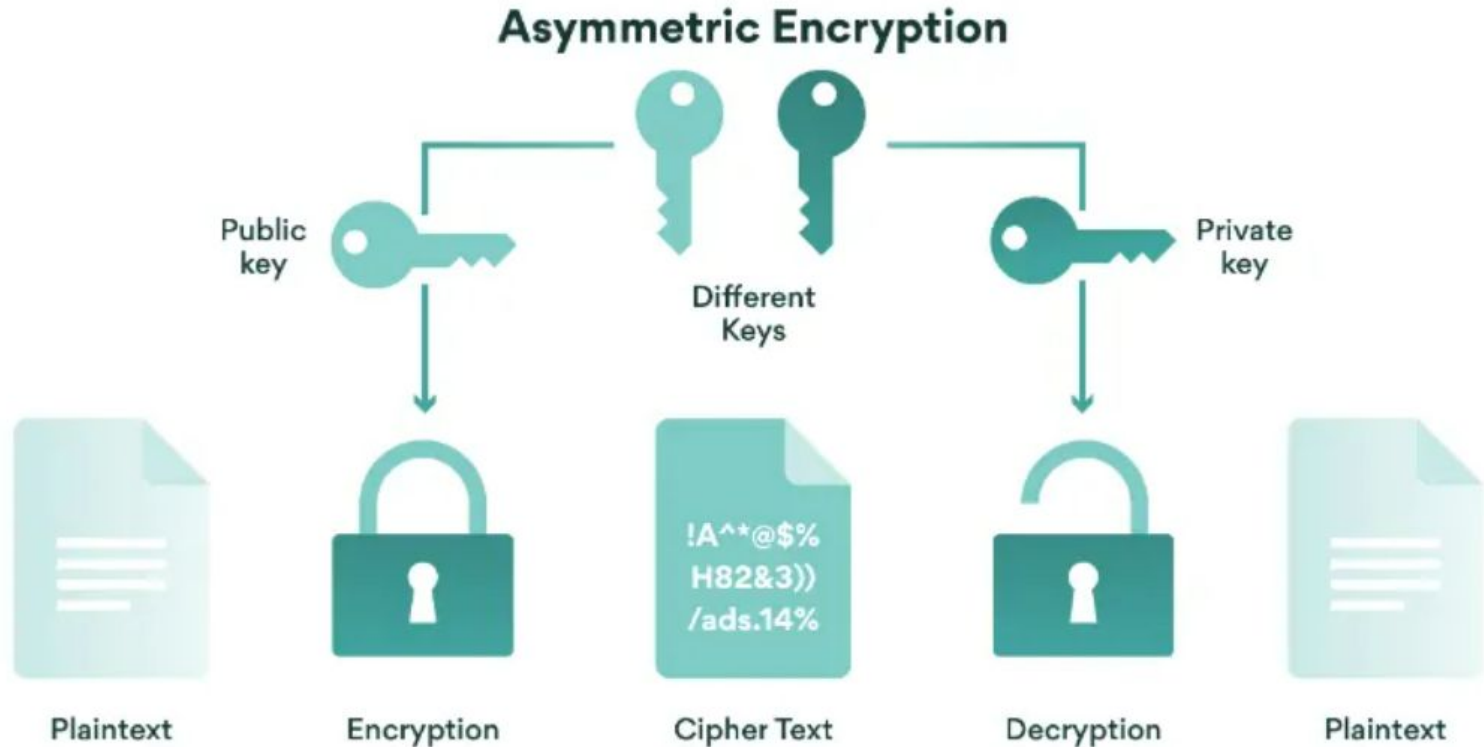
# SYMMETRIC ENCRYPTION



# SECURING DATA IN TRANSIT

- It may be necessary to allow others to send you secure data without worrying that it be intercepted.
- Giving the secure key away would not be a good decision.
- **Asymmetric algorithms** allow us to create a **public key** and a **private key**
- The public key is distributed freely.
- The private key is kept secret by the owner.
- If the public key is used for encryption, then the related private key is used for decryption. If the private key is used for encryption, then the related public key is used for decryption.

# ASYMMETRIC ENCRYPTION



# WEB ENCRYPTION

- Secure Socket Layer (**SSL**) and Transport Layer Security (**TLS**) are examples of **asymmetric key encryption**.
- **SSL** was developed by **Netscape** in 1994 to secure transactions over the WWW.
- **TLS** was developed to address some of the security risks of **SSL**. These days, when people talk about **SSL** as a protocol, they are usually actually referring to **TLS**.
- **TLS** and **SSL** are recognized as **protocols** to provide **secure HTTP(S)** for internet transactions. It supports **authentication**, **encryption**, and **data integrity**.

# DIGITAL CERTIFICATES

- Ownership of a public key is **certified** by use of a **digital certificate**.
- A **certificate authority** is a trusted third-party that provides the certificate.
- The CA **prevents** the attacker from **impersonating a server** by indicating that the certificate belongs to a particular domain.
- The CA issues digital certificates that contain a **public key and the identity of the owner**.
- When you can verify the CA's signature, then you can assume that a the public key in the certificate does indeed belong to the owner of the certificate.

# PERSON-IN-THE-MIDDLE ATTACK

- Even communication performed over **SSL** is subject to a **PITM attack**.
- The browser will set a SSL session with the attacker while the attacker sets an SSL session with the web server.
- The **browser** will try and **warn** the **user** that the **digital certificate is not valid**, but the **user** often **ignores** the warning because the threat is not understood.





LET'S LOOK AT SOME  
JAVA CODE TO DEAL WITH  
STORING PASSWORDS

# SET UP EXAMPLES

- Create a database called **user-manager**
- Run the **user-manager.sql** script, which is located in today's lecture code under the **database** directory.
- Import today's **lecture** into IntelliJ

.