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 name and then display the population for the city using this SQL String:

```
String query = "SELECT population from CITY WHERE city_id = "
 +  cityId;
```

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

```
SELECT population from CITY
WHERE city_id = 237;
```

If you run this in DbVisualizer, you will get the population for city with city_id 237.

How Does SQL Injection Work?

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

```
String query = "SELECT population from CITY WHERE city_id = "
    + cityId;
```

Gives us this SQL:

```
SELECT population from CITY
WHERE city_id = 237 OR 1=1
```

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**.

Dealing With Passwords

One type of sensitive data systems need to store is user 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

Hashing process:

- Use a **one-way function to obfuscate** the plain-text password prior to storage.
- Use the password supplied by the user, **re-hash** it, and compare it to the stored password hash value.
- **Salt the passwords** in order to make it take longer to calculate all possibilities.

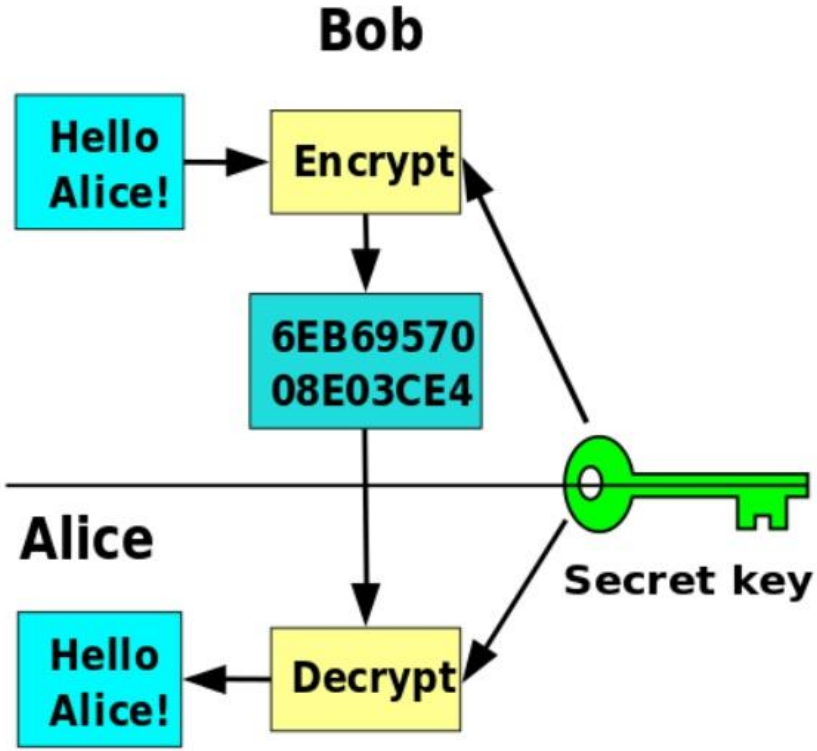
Encryption

Encryption is the most effective way to achieve data security. When data is sent between two parties or stored, it is stored in an encrypted non-human readable format that requires the key to properly decrypt and understand.

Securing Data at Rest

- Data at rest can use a form of encryption called **symmetric key encryption**
- Requires both parties to use the key to encrypt and decrypt data.
- Any party possessing the key can read the data.
- Has difficulties of securing the symmetric key amongst multiple parties.

Symmetric Encryption



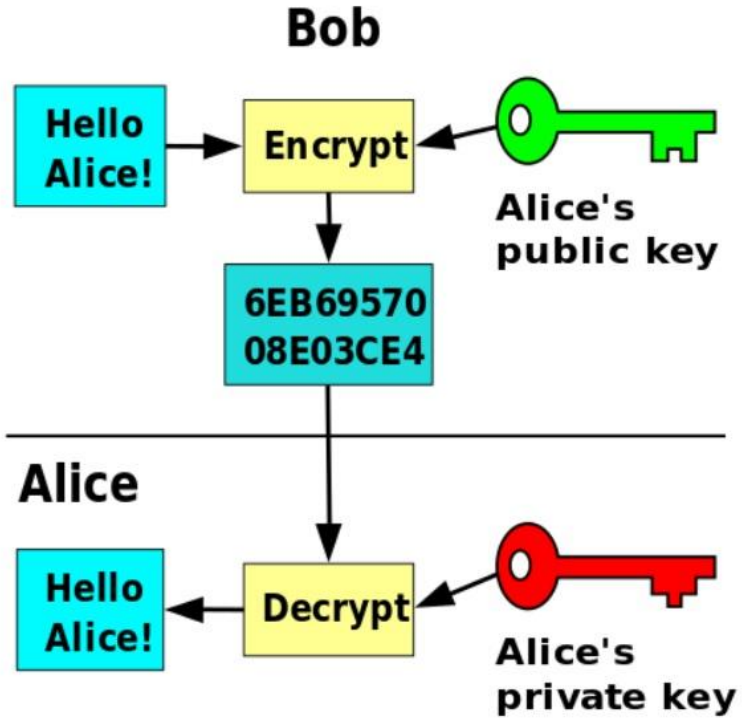
Encryption

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.

.

Asymmetric Encryption



Examples

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** and **SSL** are recognized as **protocols** to provide **secure HTTP(S)** for internet transactions. It supports **authentication**, **encryption**, and **data integrity**.

Examples

Digital Certificates

- Ownership of a public key is **certified** by use of a **digital certificate** allowing parties to rely upon the signature generated by the private key.
- 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.

.

Examples

Man-in-the-Middle Attack

- Even communication performed over **SSL** is **suspect** to a **MITM 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