# HACKER PREVENTION GROUP ASSIGNMENT

**Worth 10% of your final mark. 2% for each section (presentation & document)**

**Research the assigned security concern to create documentation and a class presentation to share with your colleagues. The documentation will be compiled into a Web Safety Manual for the class.**

## **REQUIREMENTS**

Both your documentation and your presentation will include:

* Definition of the security concern
* Potential targets or objectives of the hacker
* Notable cases or victims (examples from the news)
* Examples / Demonstrations (show the code, or demonstrate the process)
* Methods to protect your code and users

Be sure to divide up the work amongst the team: presenters, demonstration creation, document creation etc. as I will be looking for equal contribution.

Do not copy and paste the notes you find, and **be sure to cite your sources and provide links to articles or images that you reference or use**. Your colleagues will refer to this manual in the future and will want to know where to find more information. If you include images, be sure to add alt text to the image.

For the documentation, use the template provided below and **save it as a new Word document** with: security concern #, security concern, and your team lead name as the file name (e.g. ***5-sql-inject-Harpreet.docx***). One team member will submit this document on behalf of your team. Your document will be compiled with the others to create a class Web Security Manual.

***DELETE TEXT ABOVE AND REPLACE ALL RED TEXT BELOW***

|  |  |
| --- | --- |
| **Security Concern #** | **Security Concern Name** |
| Contributing Team Members: | TEAM MEMBERS’ NAMES HERE |

## DEFINITION

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database allowing an attacker to view data that they are not normally able to retrieve. This might include data that belongs to other users, or any other data that the application can access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.  
In some situations, an attacker can escalate a SQL injection attack to compromise the underlying server or other back-end infrastructure. It can also enable them to perform denial-of-service attacks.

## OBJECTIVE OF ATTACK:

A successful SQL injection attack can result in unauthorized access to sensitive data, such as:

* Passwords.
* Credit card details.
* Personal user information.

SQL injection attacks have been used in many high-profile data breaches over the years. These have caused reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period

There are lots of SQL injection vulnerabilities, attacks, and techniques, that occur in different situations which include

* [Retrieving hidden data](https://portswigger.net/web-security/sql-injection#retrieving-hidden-data), where you can modify a SQL query to return additional results.
* [Subverting application logic](https://portswigger.net/web-security/sql-injection#subverting-application-logic), where you can change a query to interfere with the application's logic.
* [UNION attacks](https://portswigger.net/web-security/sql-injection/union-attacks), where you can retrieve data from different database tables.
* [Blind SQL injection](https://portswigger.net/web-security/sql-injection/blind), where the results of a query you control are not returned in the application's responses.

## NOTABLE CASES/VICTIMS:

…

## EXAMPLES:

…

## PREVENTION:

We can prevent most instances of SQL injection using parameterized queries instead of string concatenation within the query which are also called "prepared statements".

For example the following code is vulnerable to SQL injection because the user input is concatenated directly into the query:

String query = "SELECT \* FROM products WHERE category = '"+ input + "'";

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery(query);

We can rewrite this code in a way that prevents the user input from interfering with the query structure:

PreparedStatement statement = connection.prepareStatement("SELECT \* FROM products WHERE category = ?");

statement.setString(1, input);

ResultSet resultSet = statement.executeQuery();

We can use parameterized queries for any situation where untrusted input appears as data within the query, including the WHERE clause and values in an INSERT or UPDATE statement. They can't be used to handle untrusted input in other parts of the query, such as table or column names, or the ORDER BY clause. Application functionality that places untrusted data into these parts of the query needs to take a different approach, such as:

* Whitelisting permitted input values.
* Using different logic to deliver the required behavior.

For a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant. It must never contain any variable data from any origin. Do not be tempted to decide case-by-case whether an item of data is trusted, and continue using string concatenation within the query for cases that are considered safe. It's easy to make mistakes about the possible origin of data, or for changes in other code to taint trusted data.