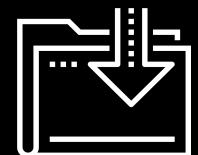




# Injection Vulnerabilities

Cybersecurity  
15.1 Web Vulnerabilities and Hardening



# Class Objectives

---

By the end of today's class, we'll be able to:



Articulate the intended and unintended functionalities of a web application.



Identify and differentiate between SQL and XSS injection vulnerabilities.



Design malicious SQL queries using DB Fiddle.



Create payloads from the malicious SQL queries to test for SQL injection against a web application.



Design malicious payloads to test for stored and reflected cross-site scripting vulnerabilities.

# Last Week...

---

We learned about the intended use and functionalities of web application:



How HTTP requests and responses work to make a web application function.



How to maintain a user's session using cookies.



The various stacks of a common web architecture and how the components within these stacks work together.



How the database component works within a web application.



How to write and run basic SQL queries against a database.

# This Week...

---

We will learn how to exploit those web applications for unintended uses and weaknesses.

Malicious actors will exploit weaknesses that might exist in these functions to cause unintended consequences.

These weaknesses are called **web vulnerabilities**.



# Intended

The intended functionality of the sample webpage allows users to purchase different widgets at various prices.

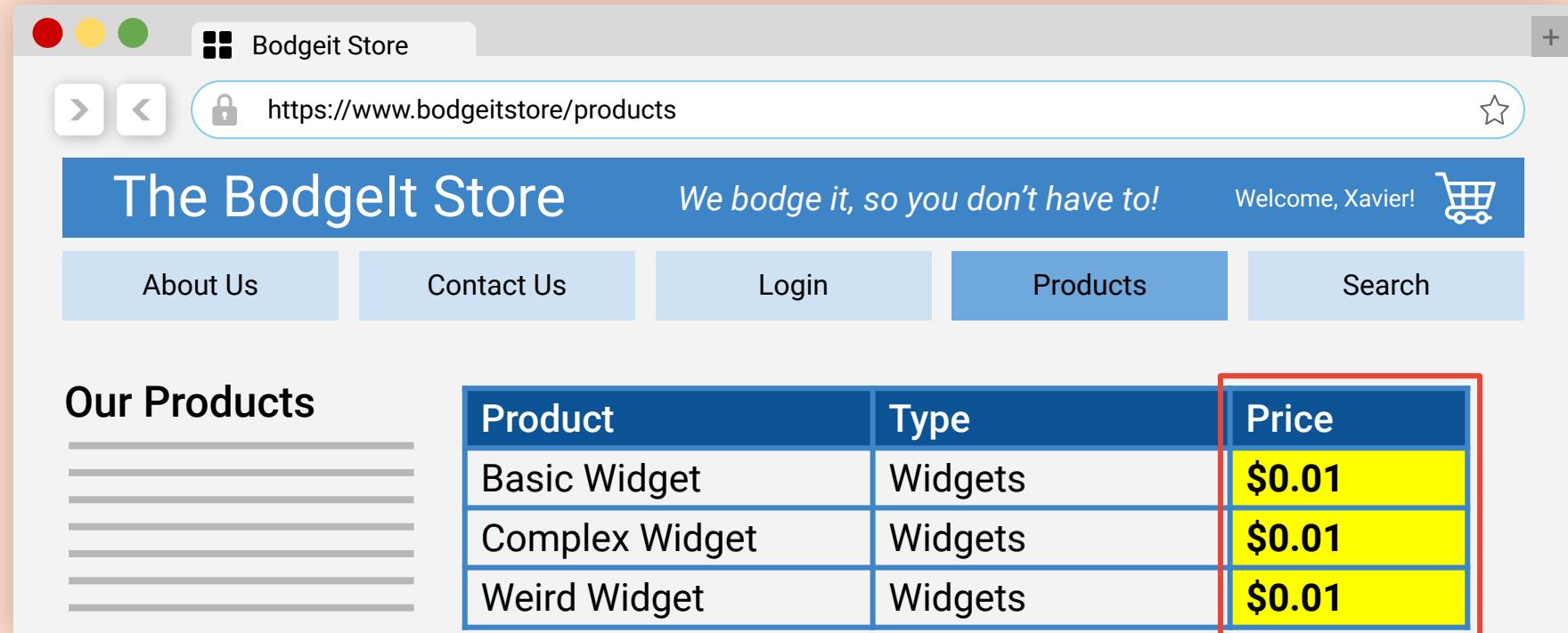
The screenshot shows a web browser window with the following details:

- Address Bar:** https://www.bodgeitstore/products
- Page Title:** Bodgeit Store
- Header:** The Bodgeit Store, We bodge it, so you don't have to!, Welcome, Xavier!, Shopping Cart icon
- Navigation:** About Us, Contact Us, Login, Products, Search
- Section:** Our Products (with five empty horizontal lines)
- Table:** A table titled "Our Products" showing three rows of widget information.

Product	Type	Price
Basic Widget	Widgets	\$1.20
Complex Widget	Widgets	\$3.10
Weird Widget	Widgets	\$4.70

# Unintended

A malicious user could exploit a web application vulnerability and create an unintended result by changing the price of all the widgets to \$0.01.



The screenshot shows a web browser window for "Bodgeit Store" at <https://www.bodgeitstore/products>. The page title is "The Bodgeit Store". The header includes a slogan "We bodge it, so you don't have to!" and a welcome message "Welcome, Xavier!". The navigation menu has links for "About Us", "Contact Us", "Login", "Products", and "Search". Below the menu, there's a section titled "Our Products" with a table. The table has columns for "Product", "Type", and "Price". The "Price" column is highlighted with a red border. All three rows in the table show a price of "\$0.01".

Product	Type	Price
Basic Widget	Widgets	\$0.01
Complex Widget	Widgets	\$0.01
Weird Widget	Widgets	\$0.01

# Impacts

---

## Financial impact

The malicious actor can exploit the web application and purchase widgets for a significantly reduced cost

## Legal impact

If a web application has a vulnerability that exposes confidential user data, a business could be subject to penalties.

## Reputational impact

If a web application has a reputation for being down, or dangerous due to web vulnerabilities, their customers will likely search for another business.

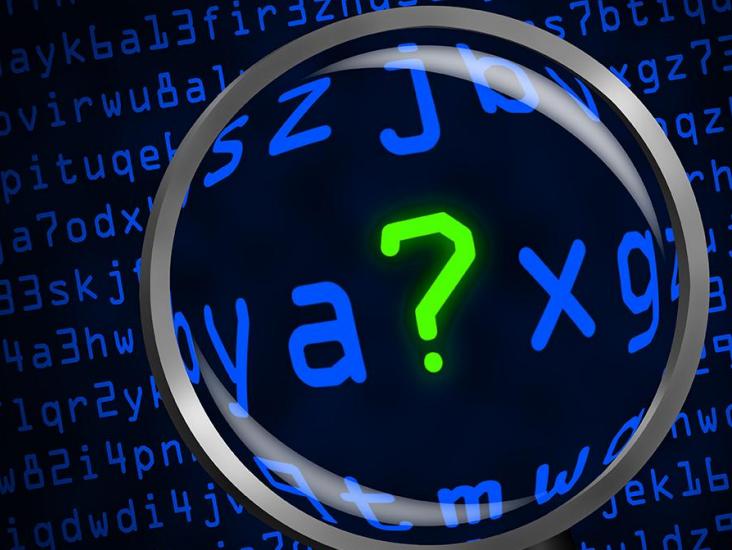
# OWASP Top 10



The expansiveness of the web and its various technologies, coding languages, and architectures leads to new capabilities. With these capabilities come many potential vulnerabilities.

## The Challenge Is...

**With so many potential vulnerabilities, how do security professionals identify and manage the top issues that could impact web applications?**



# Introducing the OWASP Top 10

---



# OWASP

Open Web Application  
Security Project

“ A nonprofit foundation that works to improve the security of software through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences. ”

The OWASP Top 10 represents a consensus from the OWASP community of the most common and critical security risks to web applications.

A01

Broken Access Control

A06

Vulnerable and Outdated Components

A02

Cryptographic Failures

A07

Identification and Authentication Failures

A03

Injection

A08

Software and Data Integrity Failures

A04

Insecure Design

A09

Security Logging and Monitoring Failures

A05

Security Misconfiguration

A10

Server-Side Request Forgery (SSRF)

# OWASP Top 10

---

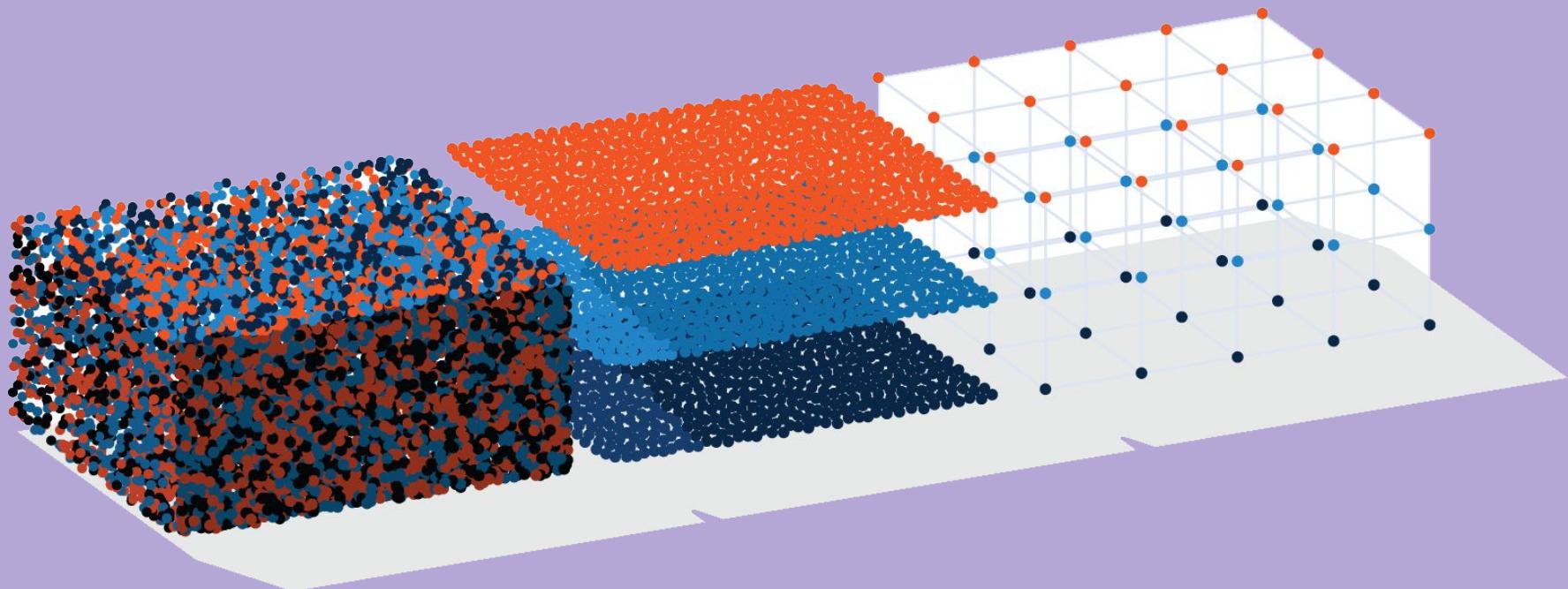
Created to educate software developers, designers, architects, managers, and organizations about the consequences of web application security weaknesses.



# OWASP Top 10

---

Developed through industry surveys completed by over 500 individuals from hundreds of organizations and over 100,000 real-world applications and APIs.



# This Week's Career Context

---

We will cover many web application vulnerabilities and demonstrate how to exploit them. Familiarity with these vulnerabilities and the methods used to exploit them are important skills for several cybersecurity careers:

## Web Application Developers

While web application developers might not work directly in cybersecurity, understanding vulnerabilities and their impact can help them create secure code

## Application Security Engineers

Application security engineers work alongside developers to ensure that they develop secure code.

## Penetration Testers

Penetration testers who test an organization's web applications need to understand these vulnerabilities, attack methods, and impacts to conduct a thorough penetration test.

# This Week

---

The week will proceed as follows:

Day 1

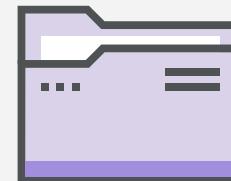
We will begin today with the number three risk from the OWASP list—**injection**.

We will cover injection vulnerabilities such as SQL injection and cross-site scripting.



Day 2

We will cover web application vulnerabilities that exist within back-end components, such as directory traversal and file inclusion.



Day 3

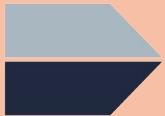
We will cover identification and authentication failures and learn how to exploit these vulnerabilities with advanced tools such as Burp Suite.



# How We Will Approach Each Vulnerability

---

As we introduce each web application vulnerability, we will cover the following:



The intended purpose of the original function



The vulnerability and method of exploit



Unintended consequences of the exploit



Mitigation of the vulnerability



The potential impact of the exploit

## Warning



The techniques we will learn throughout this unit can be used to cause serious damage to an organization's systems. This is ILLEGAL when done without permission.

## Warning



All of the labs we provide are safe locations to test the methods and tools taught during the week.

## Warning



NEVER apply any of these methods to any web applications you do not own or do not have clear, documented permission to be interacting with.

# Injections

# The OWASP Top 10: No.3 Injections

A01

Broken Access Control

A06

Vulnerable and Outdated Components

A02

Cryptographic Failures

A07

Identification and Authentication Failures

A03

**Injection**

A08

Software and Data Integrity Failures

A04

Insecure Design

A09

Security Logging and Monitoring Failures

A05

Security Misconfiguration

A10

Server-Side Request Forgery (SSRF)

# Injections

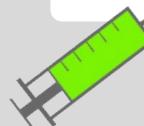
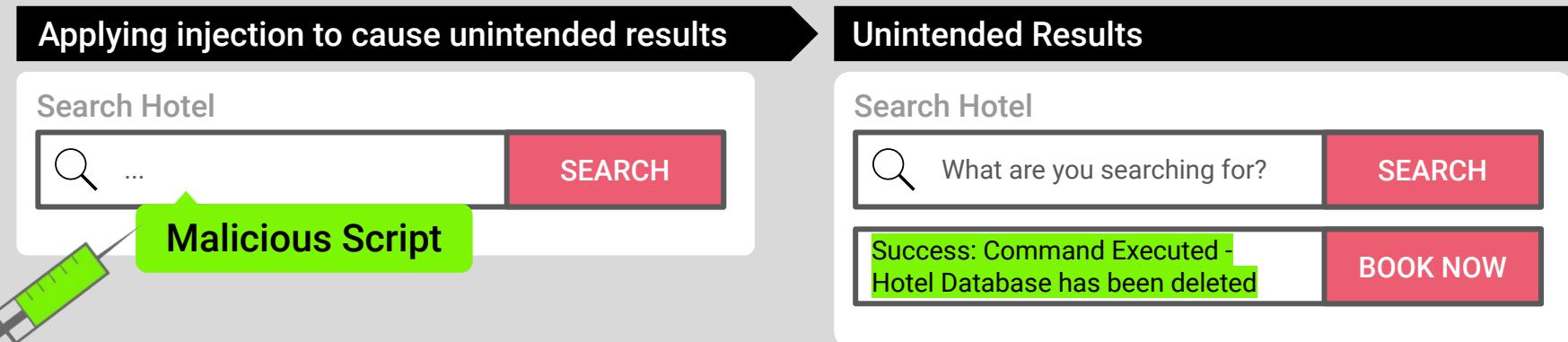
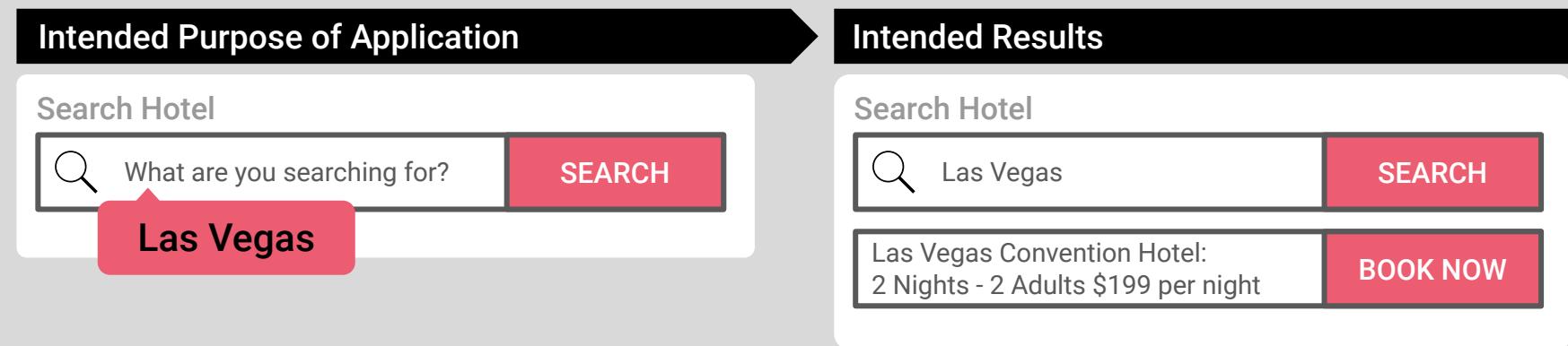
---

Injection attacks occur when an attacker supplies untrusted input to an application.

That malicious input, also known as a **payload**, contains malicious data or code that is then processed as part of a query or command that alters the way a program is intended to function.



# Injections commonly occur in fields and forms on web applications, where malicious code can be injected.

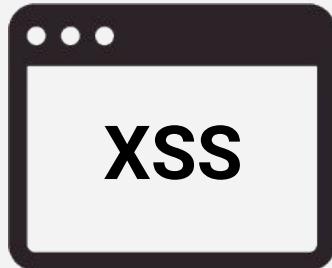


# Types of Injection

Injection involve submitting an untrusted input. How applications work with input depends on their design, architecture, and functionality. We'll focus on:

## Cross-Site Scripting

This injection is a submitted user input that can run malicious scripts against the website. It depends on the application modifying the client-side code with a user's input.



## SQL Injection

In this injection type, a submitted user input can run SQL commands against a database. It depends on the application running queries against a SQL database.



# SQL Refresher and Unintended SQL Queries

# SQL Injection

The first vulnerability we will cover is SQL injection.

SQL injection works against an application by sending requests to a SQL database through user input.

It is conducted by inserting malicious SQL statement into fields on a web application.



# SQL Injections

---

Before we learn how to conduct a SQL injection attack, we need to understand what happens behind the scenes as an application interacts with a database. We will do this as follows:

01

Cover how a web application connects to a database.

02

Review the structure of a SQL database and a SQL query.

03

Demonstrate an intended query and the intended results.

04

Cover modification of the intended query to create unintended results.

# How a Web Application Connects to a Database

Enter your userid:

 **GO**

Enter your userid:

 **GO**

Enter your userid:

 **GO**

userid	salary
jsmith	20000

The web application contains a field where a user can input their **user id**.

Julie enters her userid, "jsmith", and selects Go to submit.

The application returns Julie's salary of \$20,000.

# How a Web Application Connects to a Database

The web application took Julie's input ("jsmith") and sent it back to the web server.

The web server took the value of "jsmith" and submitted it to a database in a **pre-built** query to find that record.

The database contained a salary table with a list of users and salaries.

The query found the record for "jsmith" and returned the salary "20000" to Julie.

Enter your userid:

jsmith	GO
--------	----

userid	salary
jsmith	20000

userid	salary
lsmith	45000
wgoat	1000000
rjones	777777
manderson	65000
jsmith	20000

# Structure of a SQL Database and Query

SQL databases organize data like a spreadsheet.

Each **row** or **record** is an item in the database.

userid	salary
lsmith	45000
wgoat	1000000
rjones	777777
manderson	65000
jsmith	20000

Each **column** is a piece of data in the row.



A whole spreadsheet of rows and columns is called a **table**.

A collection of tables is a **database**.

# Structure of a SQL Database and Query

**UserId, salary** indicates the two fields requested to be displayed.

**where userid = 'jsmith'** instructs the program to only return records where the userid is "jsmith".

```
select userid, salary from salaries where userid = 'jsmith'
```

**select** is used to read or select data from a table.

**from salaries** instructs the program to select the data from the salaries table.

userid	salary
Ismith	45000
wgoat	1000000
rjones	777777
manderson	65000
jsmith	20000

# Intended Query and Results Demonstration

In the next demonstration, we will review and run an intended query against a database to visualize how they return intended results.

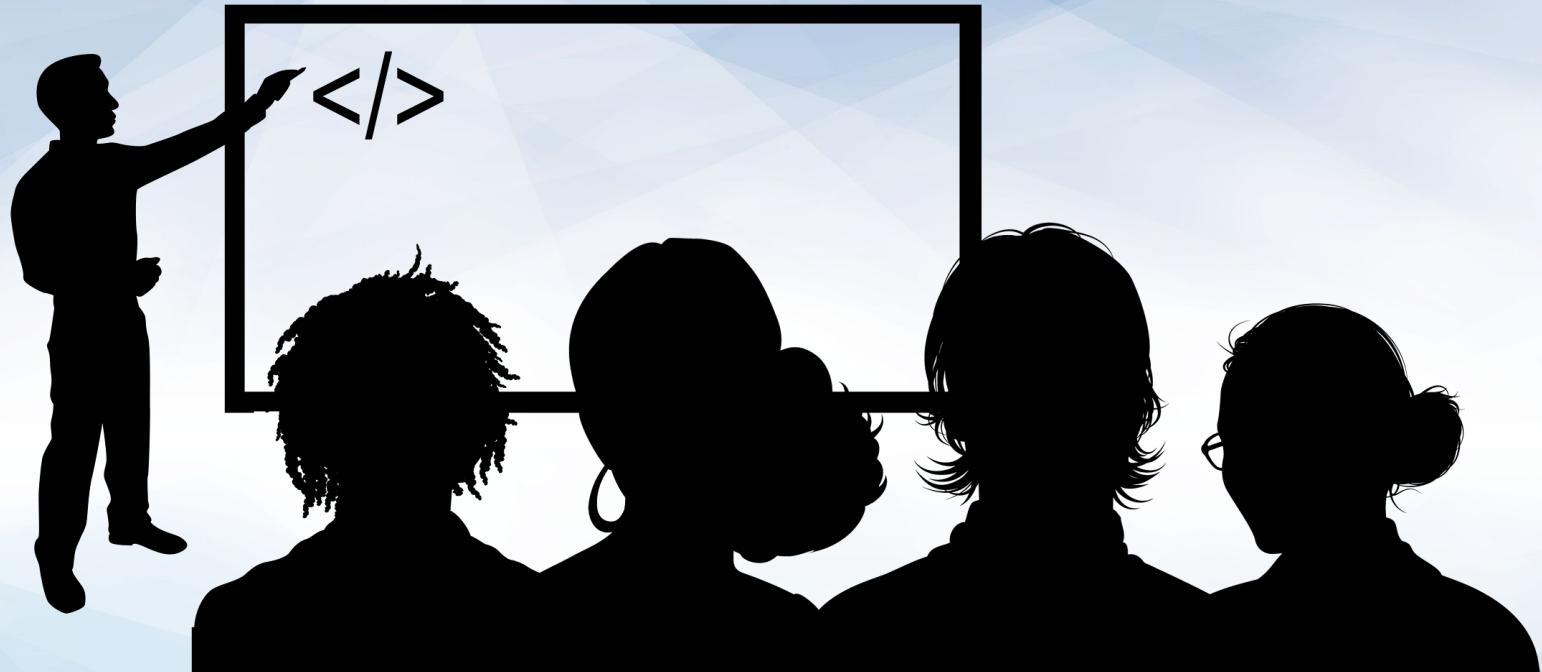
The screenshot shows the MySQL Fiddle interface. At the top, it displays "Database: MySQL v5.7". Below the header, there are tabs for "Run", "Update", "Fork", "Load Example", "Star", and "PRO". On the left, there's a "Fiddle Title" input field with a "PRO" button, a "Fiddle Description" input field with a character count of "50 characters remaining.", and a "Private Fiddle" toggle switch with a note: "This setting cannot be modified after saving the fiddle." In the center, the "Schema SQL" section contains the following code:

```
1 CREATE TABLE IF NOT EXISTS
`salaries` (
2   `userid` varchar(200) NOT NULL,
3   `salary` int(20) NOT NULL
4 );
5
6
7 INSERT INTO `salaries` (`userid`,
`salary`) VALUES
8   ('lsmith',45000),
9   ('wgoat',100000),
10  ('rjones',777777),
11  ('manderson',65000),
12  ('jsmith',20000);
13
```

The "Query SQL" section contains the following code:

```
1 select userid, salary from salaries
where userid = 'jsmith'
```

A green button labeled "Have any feedback?" is located next to the query section.



Instructor Demonstration  
DB Fiddle

# Modification of Intended Query to Create Unintended Results

---

We know the intended pre-built query for the web application:

```
select userid, salary from salaries where userid = ''
```

After Julie entered her user id of "jsmith", the intended query to pull the salary was updated:

```
select userid, salary from salaries where userid = 'jsmith'
```

# Modification of Intended Query to Create Unintended Results

SQL query uses the conditional `where` clause, like `where userid = 'jsmith'`, to specify the desired results of a query.

	<b>userid</b>	<b>salary</b>	<b>T/F</b>
Does <code>userid = jsmith?</code>	lsmith	45000	<b>False</b>
Does <code>userid = jsmith?</code>	wgoat	1000000	<b>False</b>
Does <code>userid = jsmith?</code>	rjones	777777	<b>False</b>
Does <code>userid = jsmith?</code>	manderson	65000	<b>False</b>
Does <code>userid = jsmith?</code>	jsmith	20000	<b>True</b>

{ `userid = jsmith`,  
so return salary. }

# Modification of Intended Query to Create Unintended Results

We can add a second condition, called an **always true statement**, to trigger true statements for the other columns.

	userid	salary	T/F
Does 1 = 1?	lsmith	45000	True
Does 1 = 1?	wgoat	1000000	True
Does 1 = 1?	rjones	777777	True
Does 1 = 1?	manderson	65000	True
Does 1 = 1?	jsmith	20000	True

Whereas  
`where userid = jsmith,`  
only applies to the  
`jsmith` row, an always true  
statement like `1=1` applies  
to all rows.

Therefore, the query  
returns every user id.

# Modification of Intended Query to Create Unintended Results

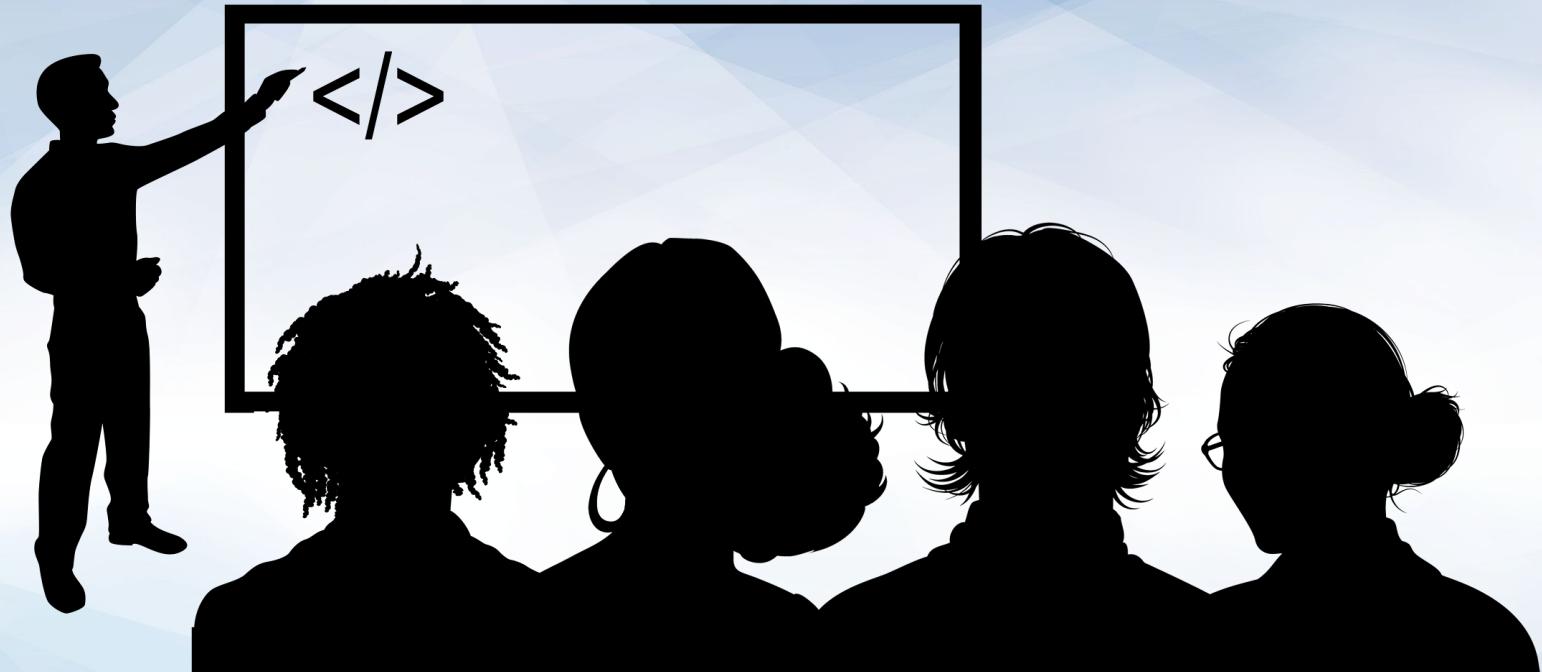
We can add a second condition, called an **always true statement**, to trigger true statements for the other columns.

```
select userid, salary from salaries where userid = 'jsmith' OR '1' = '1'
```

Let's take examine always true statements in [DB Fiddle](#).



Malicious payload



Instructor Demonstration  
Always True DB Fiddle

# Review

---

## The concepts covered so far...

01

Web vulnerabilities are weaknesses that exist within the intended functions of web applications.

02

The OWASP Top 10 is a published list of the top 10 most common and critical security risks to web applications.

03

**Injections** supply untrusted input, or the **payload**, to an application that is processed as part of a query or command.

04

**SQL injection** depends on web applications that apply user input to a database.

05

A web server can take the user input and apply it to a database using SQL queries.

06

A method to modify an intended SQL query is to add an **always true statement**.



## **Activity:** SQL Refresher and Unintended SQL Queries

In this activity, you will design several SQL queries to test directly against a database, which is represented by DB Fiddle.

**Suggested Time:**  
**20 Mins**





**Time's Up! Let's Review.**

# Testing SQL Injection on Web Applications

# Testing SQL Injection

Now we will focus on the web application and conduct the following steps:

01

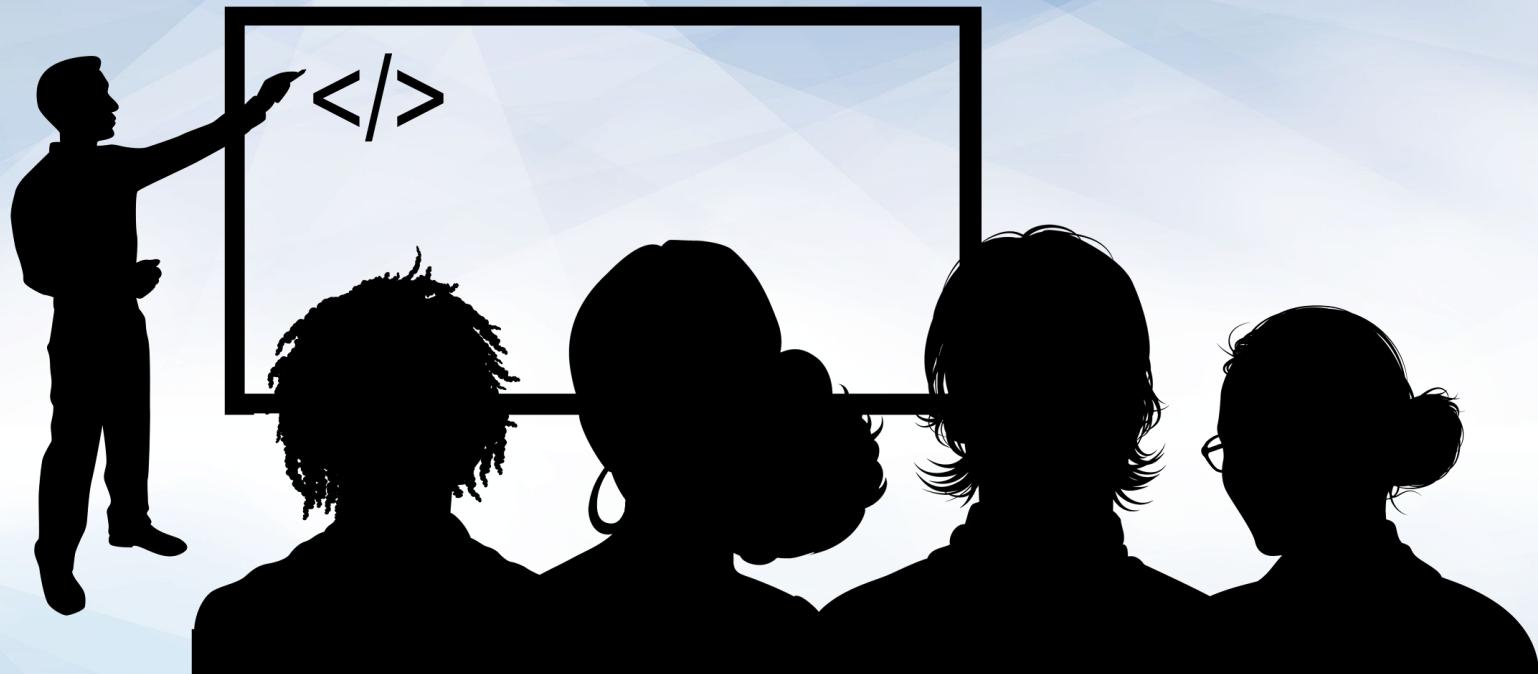
Test the intended purpose of the application.

02

Design malicious payloads by using the SQL queries built in the previous activity.

03

Test SQL injection on the application by using those payloads.



Instructor Demonstration  
SQL Injection

# Real-World Challenges

While the purpose of the lesson was to illustrate how a SQL injection payload can create unintended results, in the real world we need to be aware of several issues not covered in this lesson.

## Access to Database Structure

In the real world, application security analysts would not have access to the database tables or structure.

## Variety of Databases

In the real world there are multiple database types, in which SQL queries will not work the same way.

## Variety of Attacks

Other SQL injection attacks can delete or alter data, rather than just display it as we did.

# Impact

The reason that SQL injection is considered one of the most harmful attacks is due to the potential impact. If an attacker successfully launches a SQL injection attack against an organization's database, they might be able to display, change, or delete the organization's confidential data.

The screenshot shows a news article from Computerworld. The header includes the publication name "COMPUTERWORLD" and "UNITED STATES". A green "INSIDER" button is visible. The main headline reads "SONY CYBERATTACK" followed by "Sony Pictures falls victim to major data breach". Below the headline is a sub-headline: "Hacking group LulzSec claims it has accessed personal data on more than 1 million people". There are social media sharing icons for Facebook, Twitter, LinkedIn, Reddit, Email, and Print. A small profile picture of the author, Jaikumar Vijayan, is shown next to his name. The article was published on "JUN 2, 2011 7:20 PM PST".

# Mitigation Methods

Input validation is a common method used to mitigate this attack.

Input validation is a method to validate the data input with a predefined logic, ensuring that the input is what the application is expecting.

**EXAMPLE:**

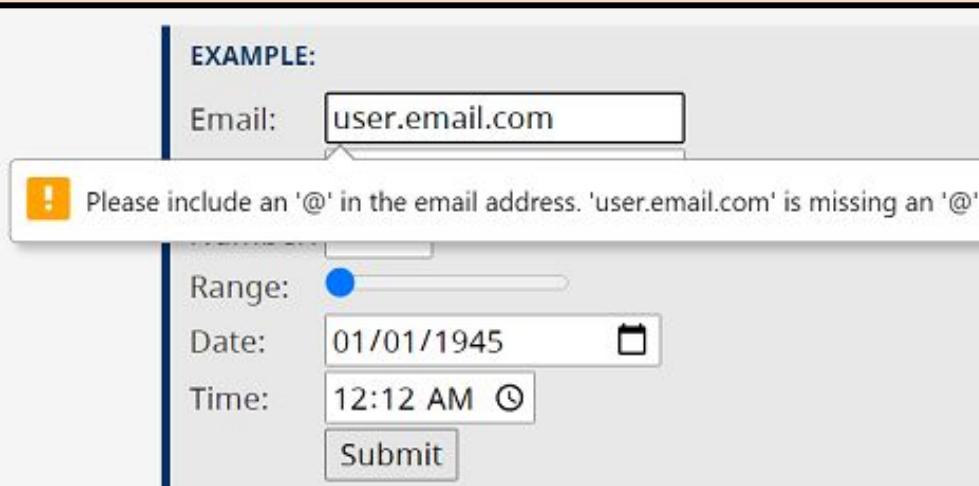
Email:

 Please include an '@' in the email address. 'user.email.com' is missing an '@'.

Range:

Date:  

Time:  



# Mitigation Methods

Input validation can be applied on the client side or the server side



## Client-side

**Client-side input validation** involves coding the predefined logic into the webpage.



## Server-side

**Server-side input validation** involves adding the predefined logic into the code on the web server.

### For example

An input can only be chosen from a predefined drop-down menu.

### For example

If a user enters a malicious SQL code and selects submit, then the web server will check and remove it after receiving this malicious input.



## **Activity:** Testing SQL Injection on Web Applications

In this activity you will use the SQL queries you crafted last activity to create payloads.

**Suggested Time:**  
**20 Mins**





**Time's Up! Let's Review.**

*Break*



# HTML & JavaScript



**Cross-site scripting (XSS)** is another injection type that occurs when an application takes in malicious user input which inadvertently executes *user-supplied* code.

# XSS Scenario

AwesomeBikes.com is an online message board intended for users to recommend bike models, ask questions, and chat about everything related to cycling.

The screenshot shows a web browser window for the 'Awesome Bikes' website. The URL in the address bar is <https://www.awesomebikes.com/discussionboard>. The page title is 'Our Discussion Board'. At the top, there are buttons for 'New Topic', 'My Topics', and 'Favorites'. A yellow arrow points from a callout box to the second post, which is by user 'Angie'. The callout box contains the text: 'The intended use involves users entering their messages into a text box. Those messages will then display on the message board.' The first post is by user 'Phil' and the second post is by user 'Angie'.

The intended use involves users entering their messages into a text box. Those messages will then display on the message board.

Awesome Bikes

<https://www.awesomebikes.com/discussionboard>

New Topic My Topics Favorites

Last visit was: Wed Feb 21, 2021 12:08 pm

## Our Discussion Board

Post a new message

Today's posts

Mark forums as read

Phil  
03-02-2021  
04:47 PM (CT)

I just got the new Cannondale A341, and it's amazing! Anyone else own one?

Reply

Angie  
03-02-2021  
04:47 PM (CT)

Not me, but I am looking for a new bike. I have \$1000 to spend.

Reply

# XSS Scenario

However, because this site is vulnerable to injection attacks, a malicious user can input a script in the text box.

Rather than post the text of the script to the message board, the webpage will interpret the script as code. This code will infect any user who subsequently visits AwesomeBikes.com.

The screenshot shows a web browser window for 'Awesome Bikes' at the URL <https://www.awesomebikes.com/discussionboard>. The page has standard navigation buttons (New Topic, My Topics, Favorites) and a 'New Topic' button. A red box highlights the text input field where a malicious script was entered. Another red box highlights the injected script code. A yellow callout box with a red border contains the text 'Malicious script to redirect to coolbikes.net'. A red arrow points from this callout to the injected script code. A blue button at the bottom right says 'Post a new message'.

Awesome Bikes

https://www.awesomebikes.com/discussionboard

New Topic    My Topics    Favorites

New Topic

```
<script> location.replace("https://www.coolbikes.net") </script>
```

Malicious script to redirect to coolbikes.net

Post a new message

# XSS Scenario

Once the attacker posts the redirect script, subsequent visitors to AwersomeBikes.com will be redirected to coolbikes.net, a malicious spoof site that tries to scam customers.

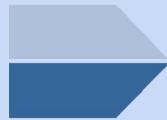
The screenshot shows a web browser window with the following details:

- Address Bar:** Displays "https://www.coolbikes.net".
- Page Title:** "Cool Bikes".
- User Interface:** Includes standard browser controls (red, yellow, green buttons) and a star icon for bookmarks.
- Forum Header:** Features "New Topic", "My Topics", and "Favorites" buttons.
- Last Visit:** Shows "Last visit was: Wed Feb 21, 2021 1:10 pm".
- Main Content:** A forum topic titled "Cool Bikes" with a "Post a new message" button and a "DOWNLOAD" button with a downward arrow.
- Post 1:** By user "Janet" on "03-02-2021 01:35 AM (PST)". Content: "Cheap bikes for sale! Click link here." with a "Reply" button.
- Post 2:** By user "Sam" on "03-02-2021 04:47 PM (CT)". Content: "Send BitCoin for good bike! Act now!" with a "Reply" button.
- Navigation:** A "Todays posts" button is visible below the main content area.

# XSS Impacts

---

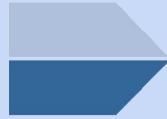
Depending on the specific script that the user inputs, the impact on subsequent visitors can include a number of potential actions, such as the following:



Redirecting to a spoof page where the malicious user can then try to sell fake products and capture credentials.



Stealing the user's cookies.



Adding a keylogger onto the user's machine.



Downloading malware to the user's machine.

# HTML and JavaScript

---

To learn how XSS exploits affect webpages, we first need to go behind the scenes of a webpage and learn how the source code creates the displays that we interact with when visiting.

**To better understand how the webpage source code works, we will cover the following:**



**HTML:**

A language used to build the structure of a webpage.



**Javascript:**

A language used to make webpages interactive and dynamic.

# HTML

---

When a user enters a URL in a browser and the browser displays a webpage, several high level-steps occur behind the scenes.

STEP 1

You enter "google.com" in the search bar on the top of your browser and press Enter.

STEP 2

Your browser sends a request to Google's web server.

STEP 3

Google's web server returns an HTML file to your browser.

STEP 4

Your browser renders the HTML file to display the items on the webpage.

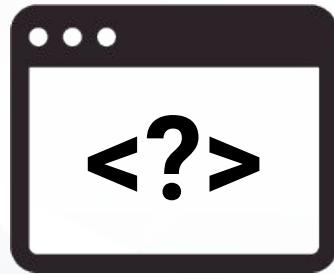


What is HTML?



Google Search

I'm Feeling Lucky



# What Do You Know about HTML?

# HTML



HTML, or Hypertext Markup Language, is a language used to display the content on a webpage.

 It is considered a **client-side language**, as it is designed to run on the user's client, the browser.

 HTML contains **elements**, which can define the following:

- The boundaries of a paragraph on a webpage, like where a paragraph starts and ends.
- The size and boldness of headings on a webpage.
- The placement of embedded images, video, or audio.

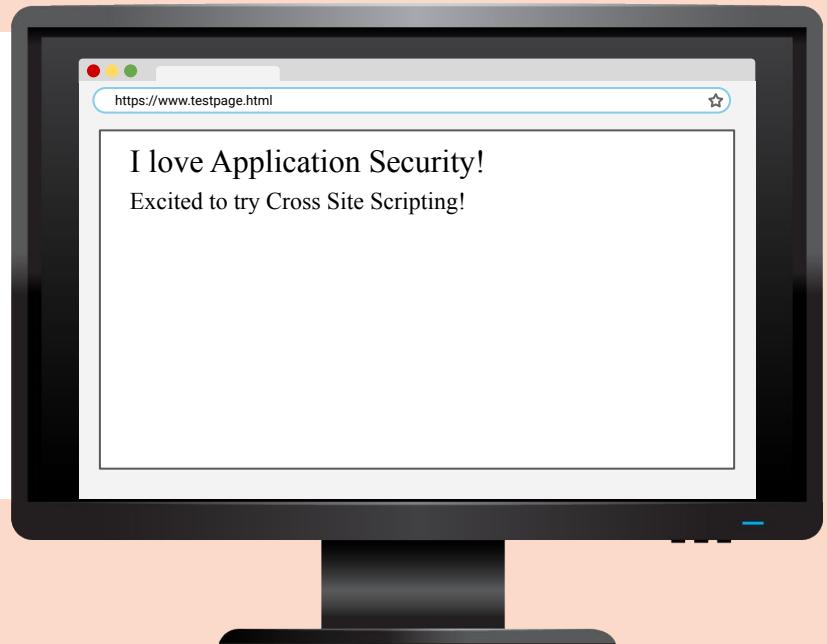
 HTML elements use **tags** and **angle brackets** < > to delineate the HTML structure of a webpage. For example, the <html>, <body>, and <h1> tags introduce content into a webpage.

# Sample HTML

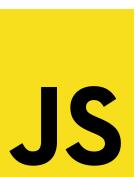
---



```
<html>
<body>
<h1>I love Application Security!</h1>
<h2>Excited to try Cross Site Scripting!</h2>
</body>
</html>
```



# JavaScript



While HTML provides options that can improve the aesthetic and design of a web page, it still has limitations to static improvements and is unable to create more dynamic features on a web page.

JavaScript is a programming language that allows web developers to add complex web features that dynamically update web content and add life to a web page.

**These features include:**



Animations



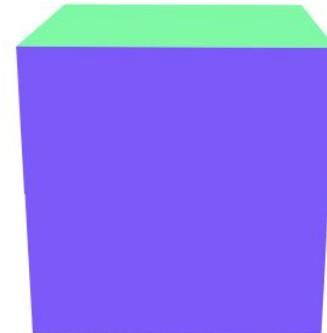
Interactive games and maps



Audio and Video playback



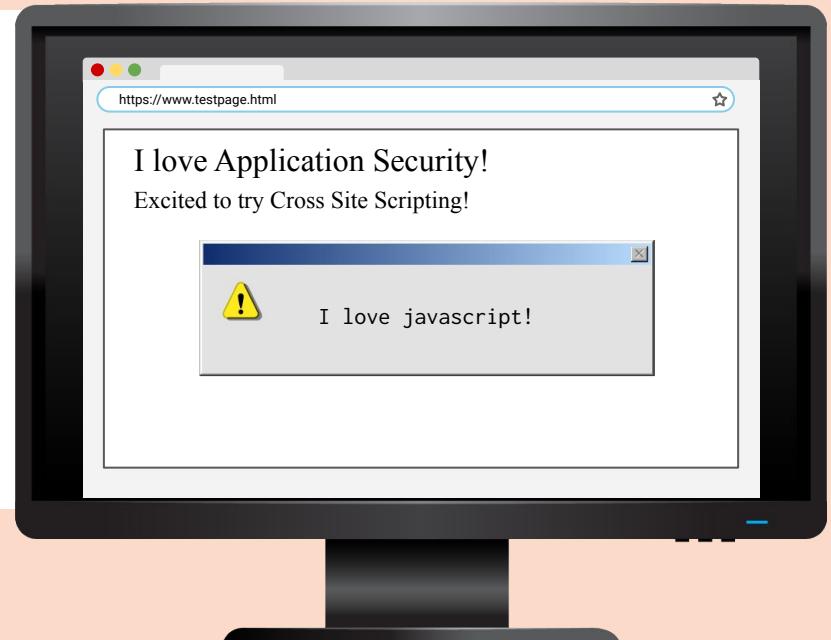
Online Chats



# JavaScript

JS

```
<html>
<body>
<h1>I love Application Security!</h1>
<h2>Excited to try Cross Site Scripting!</h2>
<script>alert("I love javascript!")</script>
</body>
</html>
```



- <script> indicates the start of the JavaScript.
- alert("I love javascript") is a JavaScript alert script used to create a pop-up that states "I love javascript".
- </script> closes the script JavaScript.

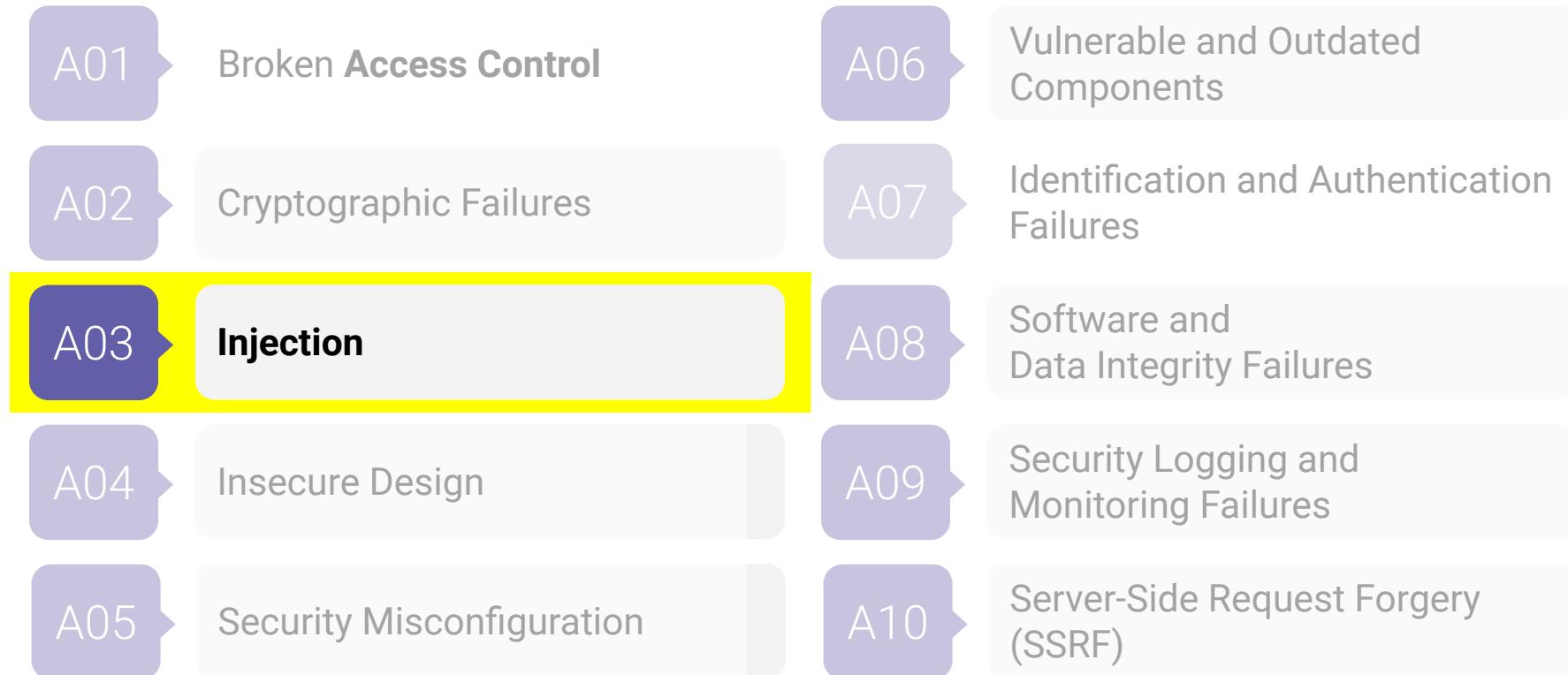


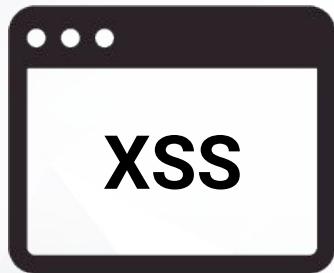
Instructor Demonstration  
Testing Guestbook Web Application

# Testing XSS on Web Applications

# The OWASP Top 10: Injection

Now we'll take a look at **cross-site scripting**, a risk that is part of category A03: Injection.





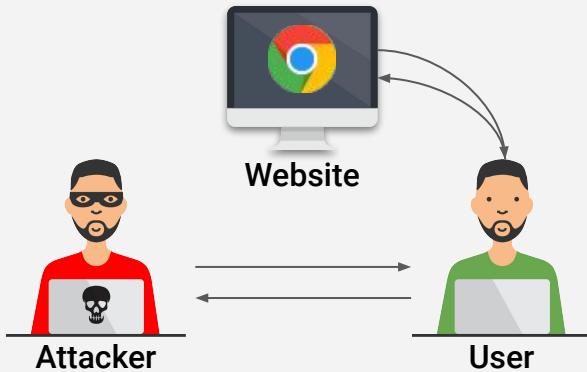
**Cross-Site Scripting (XSS)** is a web application attack that injects malicious scripts into vulnerable web applications, impacting the users of the web application.

# Testing XSS on Web Applications

There are multiple types of cross-site scripting, depending on how the application is designed:

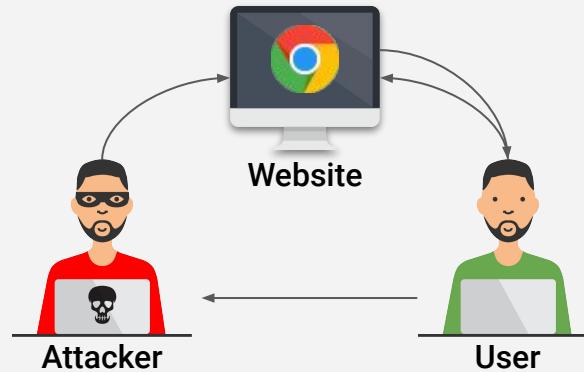
## Reflected (Non-Persistent) XSS

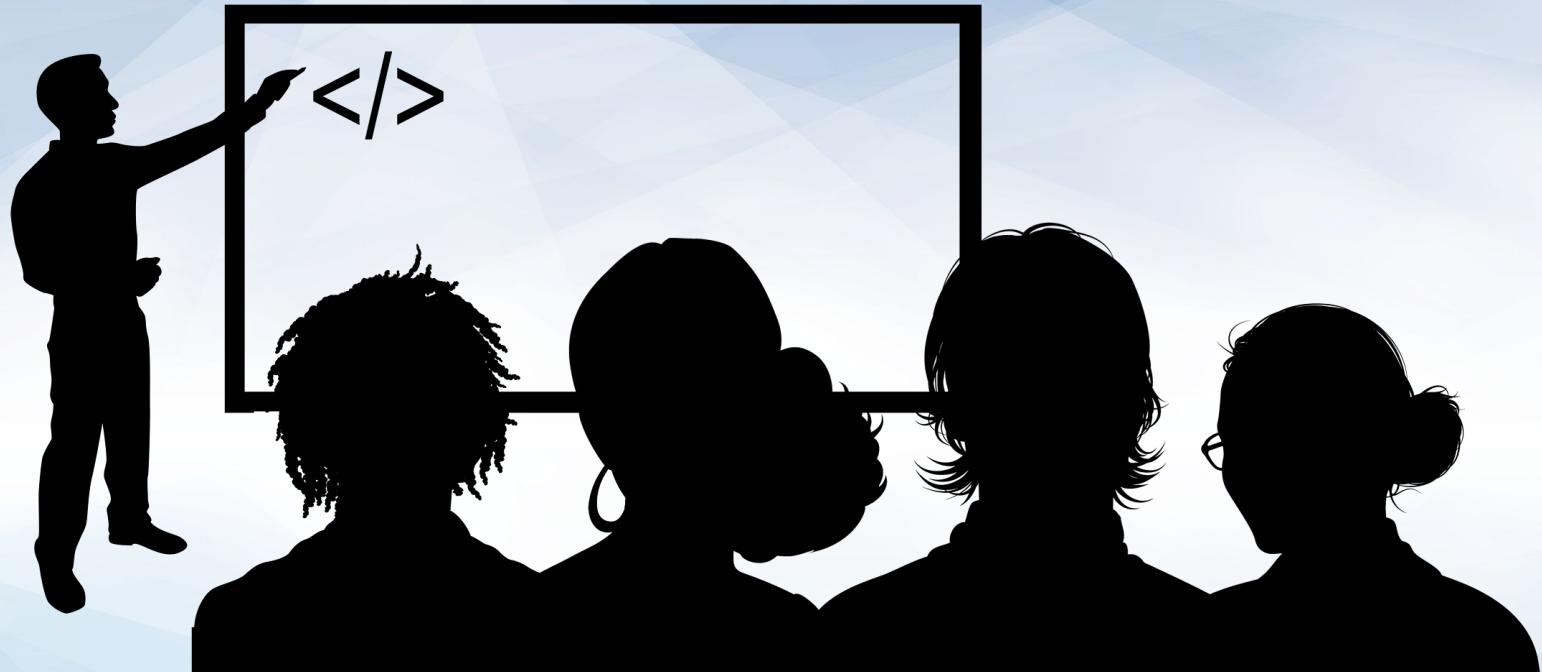
depends on the user input being immediately returned to the user and not stored on the application's server.



## Stored (Persistent) XSS

depends on the user input being stored on the application's server and later retrieved by a victim accessing the web application.





Instructor Demonstration  
Stored XSS

# Stored XSS Recap

---

Stored XSS depends on the application storing the user input on the web application's server.

An attacker submits a malicious input (or payload) into the web application.

The malicious input is returned to ANY subsequent user that visits the infected page on the web application.

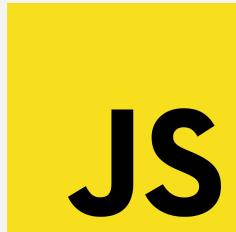


# Stored XSS Recap

Stored XSS depends on the application storing the user input on the web application's server.

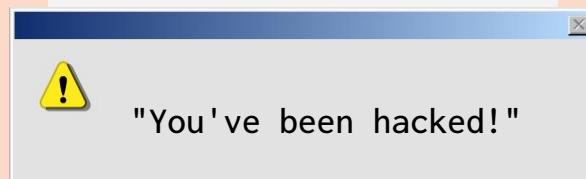
01

In the example, we added a JavaScript tag to the HTML.



02

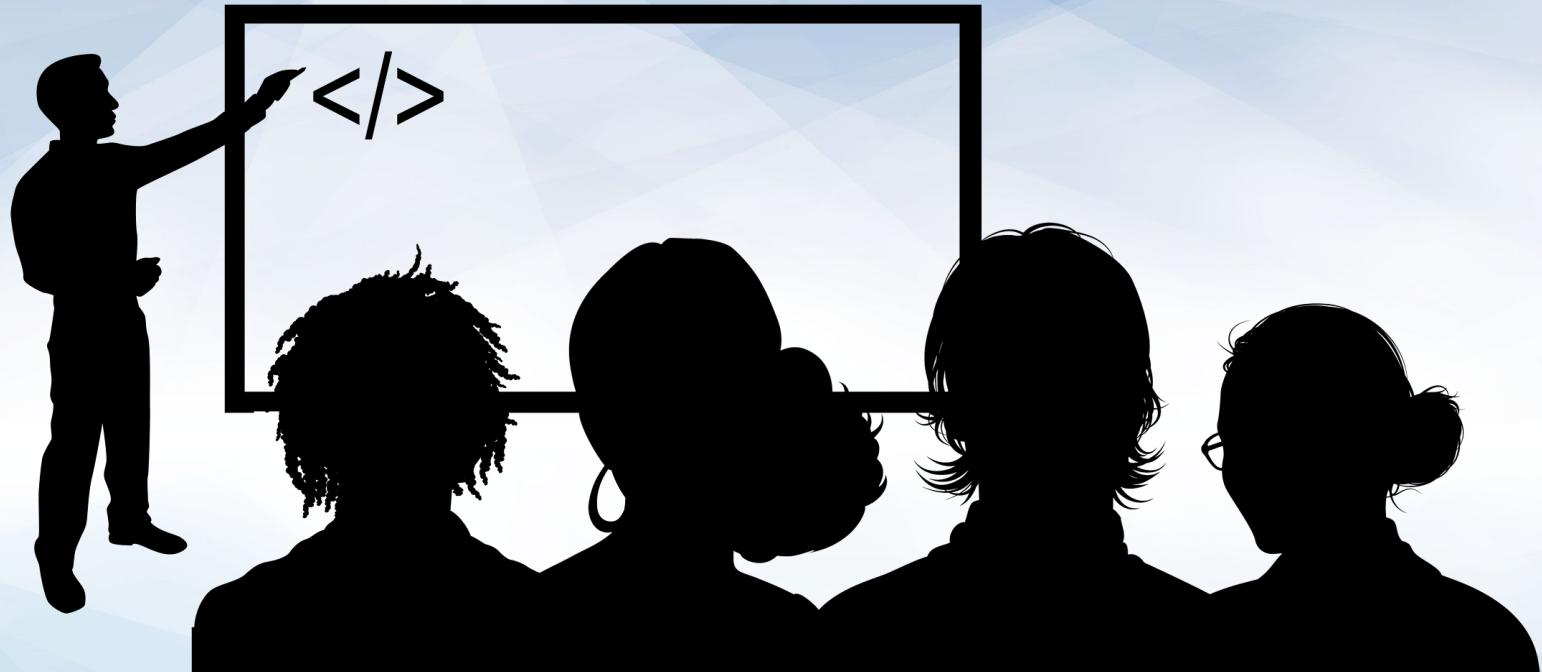
Anytime a user visits the page, a pop-up will be displayed.



03

An attacker could apply much more malicious scripts, such as the following:

- Redirecting the user to a spoof webpage.
- Installing a keylogger to capture what the user enters on other webpages.
- Capturing the user's cookies and sending them to the attacker.



Instructor Demonstration  
Reflected XSS

# Delivering a Reflected XSS Attack

Delivering a stored XSS attack simply requires inputting malicious scripts into the application. Then anyone who visits that application will have that script run.

Reflected XSS scripts need to be delivered differently, because the script is not stored on the web application. The URL must be sent to the victim, and the victim must be deceived into clicking the link.

**Phishing attacks are a common delivery method.**

The screenshot shows a Gmail inbox interface. On the left, there are navigation links for 'Mail' (with a plus sign) and 'Inbox' (highlighted with a red badge showing '936'). Below that is a 'Chat' section with a plus sign. The main area displays an incoming email from 'Security Alert'. The subject line reads 'Security Alert: Please update your billing information'. The body of the email says 'Dear Valued Customer, We've detected unusual activity on your account and suspended access. In order to reactivate, please go to the following link and confirm your billing information.' At the bottom of the email is a large blue button with the text 'CLICK HERE'.

# Impact

---

Cross-site scripting is considered a harmful attack due to the serious potential impact.

01

## Stored XSS

If a malicious actor applies \*\*stored XSS\*\* to a web application, every subsequent user that visits the infected web application will have the malicious script run, with the following impact:

- The victim's cookies stolen and session hijacked.
- The victim's computer infected with malware.



02

## Reflected XSS

If a malicious actor applies \*\*reflected XSS\*\* to a web application by sending a phishing email to a victim, the attacker can also do the following:

- Steal the victim's cookies and hijack their session.
- Infect the victim's computer with malware.



# Mitigation Methods

Input validation is a common method used to mitigate cross-site scripting.



Input validation is a method used to validate the data input with a pre-defined logic to ensure that the input is what the application is expecting.

Text Input

Submit

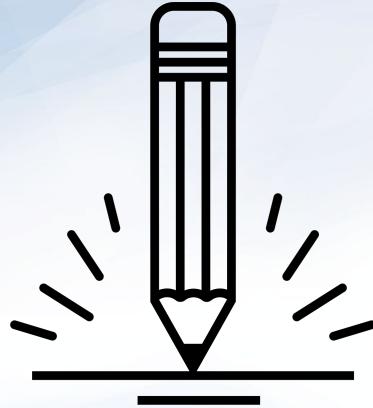


For stored XSS, we can use **server-side input validation** to prevent scripts from being stored on the application's web server.

For example, it would not accept, at any time, submitted input that might include `<script>` or `</script>`.



For reflected XSS, the input is not stored on the web server, so it would be best to use a **client-side input validation**, in which the code might not allow a malicious user to input scripts.



## **Activity:** Testing XSS on Web Applications

In this activity you will test the web application for cross-site scripting vulnerabilities.

**Suggested Time:**  
**20 Mins**





**Time's Up! Let's Review.**

# Class Objectives

---



Articulate the intended and unintended functionalities of a web application.



Identify and differentiate between SQL and XSS injection vulnerabilities.



Design malicious SQL queries using DB Fiddle.



Create payloads from the malicious SQL queries to test for SQL injection against a web application.



Design malicious payloads to test for stored and reflected cross-site scripting vulnerabilities.

*The  
End*