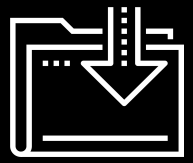
# Introduction to Web Vulnerabilities and Hardening

Cybersecurity

Web Vulnerabilities and Hardening, Day 1



# Class Objectives

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

Articulate the intended and unintended functionalities of a web application.

Identify and differentiate between SQL and XSS injection vulnerabilities.

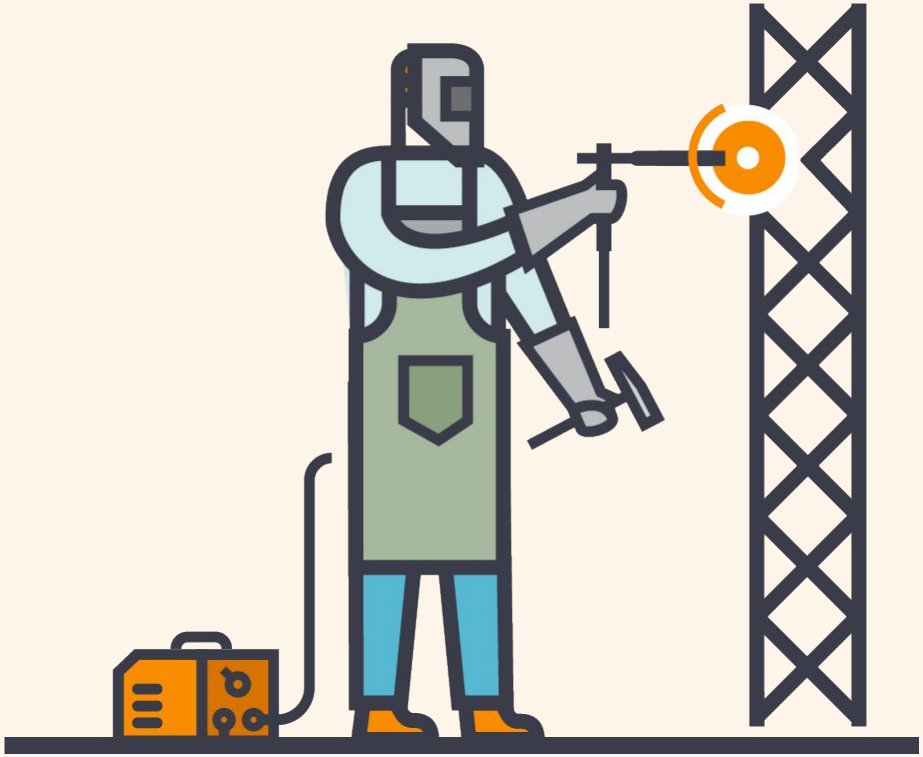
Design malicious SQL queries using DB Fiddle.

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.

# Introduction to Offensive Security

Suppose a demolition crew wants to dismantle a bridge efficiently and safely.



If certain sections of the bridge are handled incorrectly, it could harm the crew.

Therefore, knowledge of the bridge's weak points is essential for a safe and efficient project.

Similarly, a security professional needs to know the weak points in a network in order to effectively assess its security.

**Offensive security** means using proactive measures to protect networks, systems, and applications from malicious attacks.

# Introduction to Offensive Security

**Offensive security involves...**

Testing a technology for potential vulnerabilities.

**Offensive security requires...**

A thorough understanding of the technology being tested.

Everything that we've done in this course has built up our knowledge to prepare us for the offensive security modules.

# Introduction to Offensive Security

You have prepared for the offensive security module by learning how technologies properly operate. For example:

**By studying web development:**

You learned how front-end web applications interact with back-end services.

**By studying Linux and Windows OS:**

You learned about the infrastructure of these systems and where valuable data is located within them.

**By studying networking:**

You learned how devices exchange information with one another through ports and protocols.

**Now that you understand how these components properly operate, you can learn about the vulnerabilities that might exist within them.**

# Offensive Security Career Context

Offensive security is a significant part of the InfoSec community and includes several potential roles, such as:

**Network penetration tester**

Primarily works in networks and tests for vulnerabilities by exploiting missing patches, misconfigurations, etc.

**Red teamer**

Similar to a penetration tester, but red teamers also test the capabilities of the blue team by trying to not be caught. Red teaming comes the closest to actual threat-actor simulation.

**Vulnerability assessment analyst**

Conducts vulnerability scans throughout the network and validates findings.

**Web application penetration tester**

Someone who tests a web application's security.

# Offensive Security Module Preview

In the coming weeks, we will explore the following topics and tools:

| | Week 15 | Week 16 | Week 17 |
|---|---|---|---|
| | **Web Vulnerabilities** | **Penetration Testing I** | **Penetration Testing II** |
| Topics | Cross-site scripting (XSS), SQL injection, validation bypasses, directory traversal, brute forcing | OSINT, enumeration, port scanning, payloads, Linux | Windows, lateral movement, persistence |
| Tools | Burp Suite, BeEF, DB Fiddle | Metasploit, Nmap, Recon-ng, shodan.io, WHOIS | Metasploit, SearchSploit, msfvenom, Meterpreter |

# Questions?

# During Web Development Week...

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

- 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.

# Intended vs. Unintended

# Intended vs. Unintended

Intended functionalities for web applications could include:

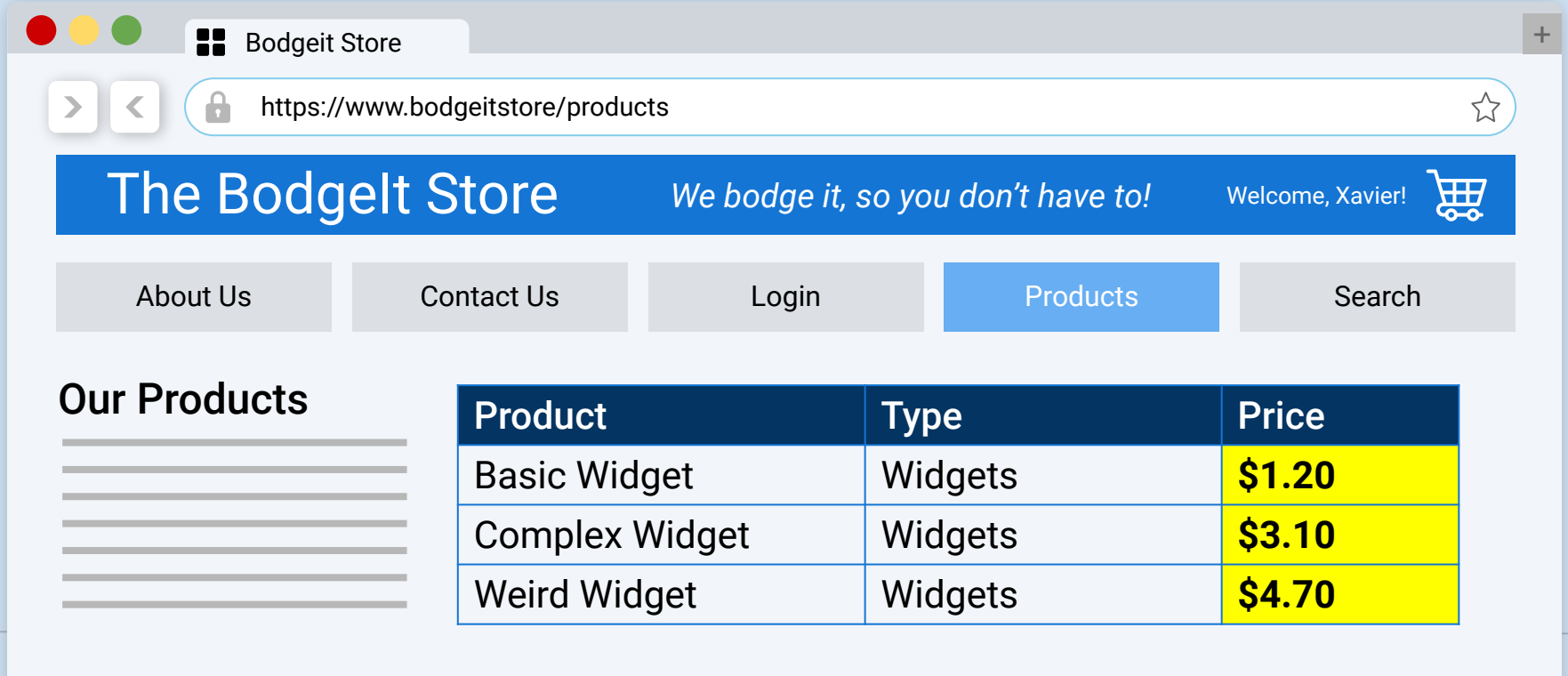
- Purchasing a product.
- Uploading and displaying an image.
- Posting a message in a guestbook.
- Viewing your balance on a bank account.

Malicious actors try to exploit weaknesses that exist within these functions, causing **unintended** consequences.

These weaknesses are called **web vulnerabilities**.

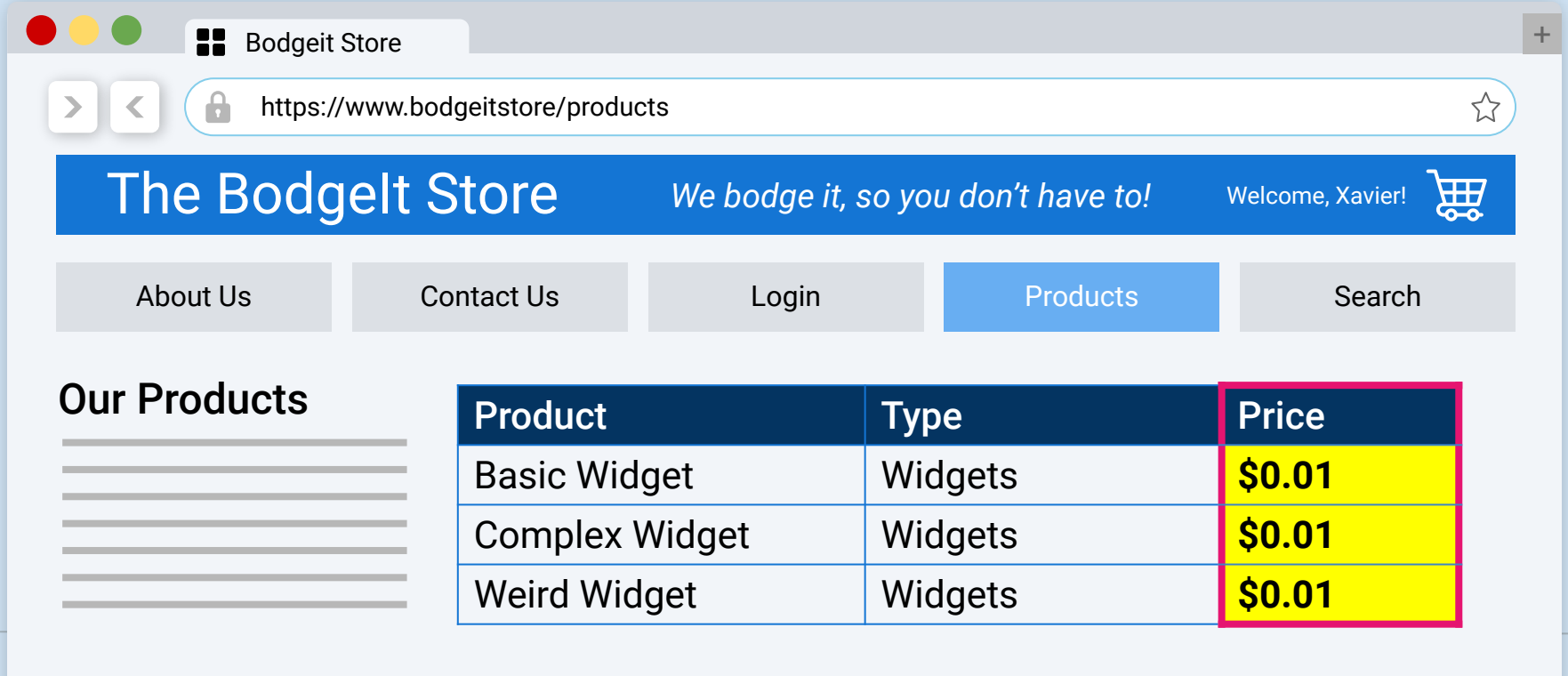
# Intended

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



# 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.



# Impacts

These vulnerabilities can have a significant impact, including:

## Financial impact

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

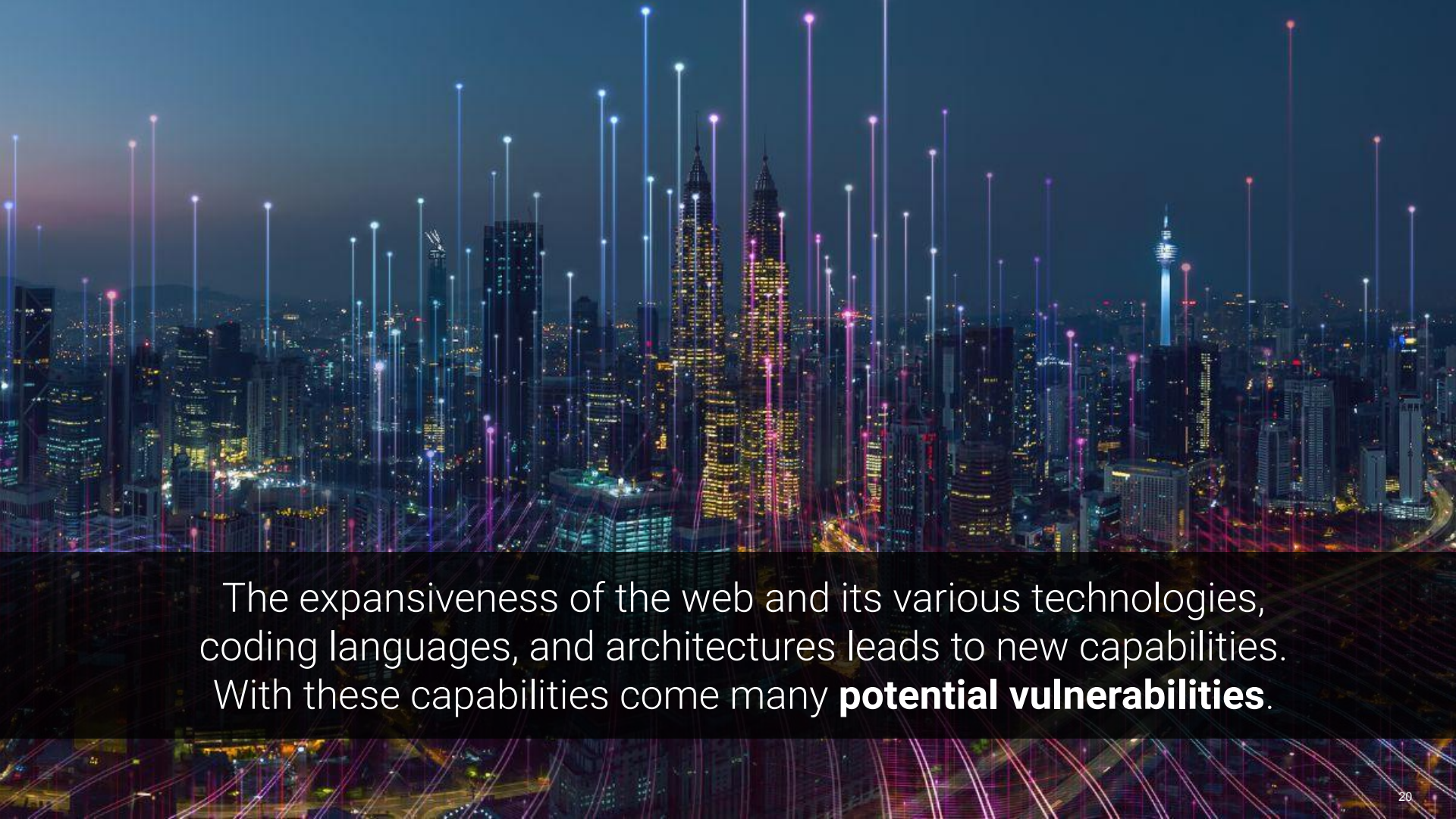
## Legal impact

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 gains a reputation for being down or dangerous due to web vulnerabilities, its 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**.

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.

Source: https://owasp.org/

# Pro Tip:

OWASP is well known for their list of the top ten web application risks:
OWASP: Top Ten Web Application Security Risks

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

| | |
|---|---|
| A01 | Broken Access Control |
| A02 | Cryptographic Failures |
| A03 | Injection |
| A04 | Insecure Design |
| A05 | Security Misconfiguration |
| A06 | Vulnerable and Outdated Components |
| A07 | Identification and Authentication Failures |
| A08 | Software and Data Integrity Failures |
| A09 | Security Logging and Monitoring Failures |
| A10 | Server-Side Request Forgery (SSRF) |

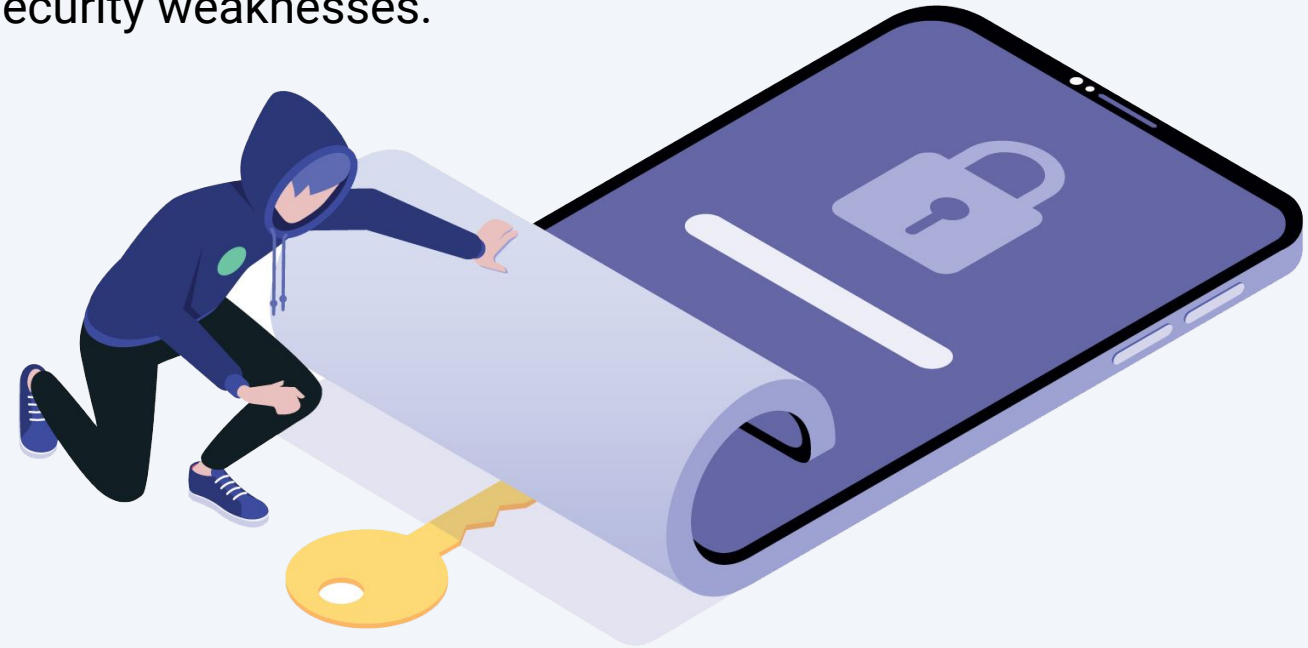
# OWASP Top 10

- Organizations can use this list to prioritize risks that might impact their applications.
- The list gets reviewed frequently and is updated when necessary. The latest version was updated in 2021.
- The latest update added three new categories, four categories had naming and scoping changes, and some items were consolidated.

It is quite common to get interview questions about the OWASP Top 10.

# OWASP Top 10

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



# OWASP Top 10

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

# Attack vs. Vulnerability

# Attack vs. Vulnerability

Sometimes, it might seem like the words **attack** and **vulnerability** are used interchangeably. For example, you might see the terms:

SQL injection **attack**

SQL injection **vulnerability**

# Can anyone explain the difference?

# Attack vs. Vulnerability

SQL injection **attack**

The method or action used to exploit the vulnerability

SQL injection **vulnerability**

The weakness within a function of the web application that can be exploited

# 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 their impact to conduct a thorough penetration test. |

# Week Overview

The week will proceed as follows:

## Day 1:

**We will:**

- Begin today with the number three risk from the OWASP list—injection.
- 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 the exploit.
- The unintended consequence of the exploit.
- The mitigation of the vulnerability.
- The potential impact of the exploit.

# This Week's Activity

You will play the role of application security analysts at a bioengineering company called Replicants.

While Replicants does bioengineering well, the company faces challenges developing a secure website.

You were hired as an application security analyst to keep its web applications secure!

Replicants just received an anonymous email stating that its web application has many web vulnerabilities. Unfortunately, the email does not identify what these vulnerabilities are.

Because you are responsible for keeping the company's data secure, you will be tasked with testing for vulnerabilities on its application and providing recommendations for mitigation.

## ⚠️ Important ⚠️

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

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

# Questions?

# Injections

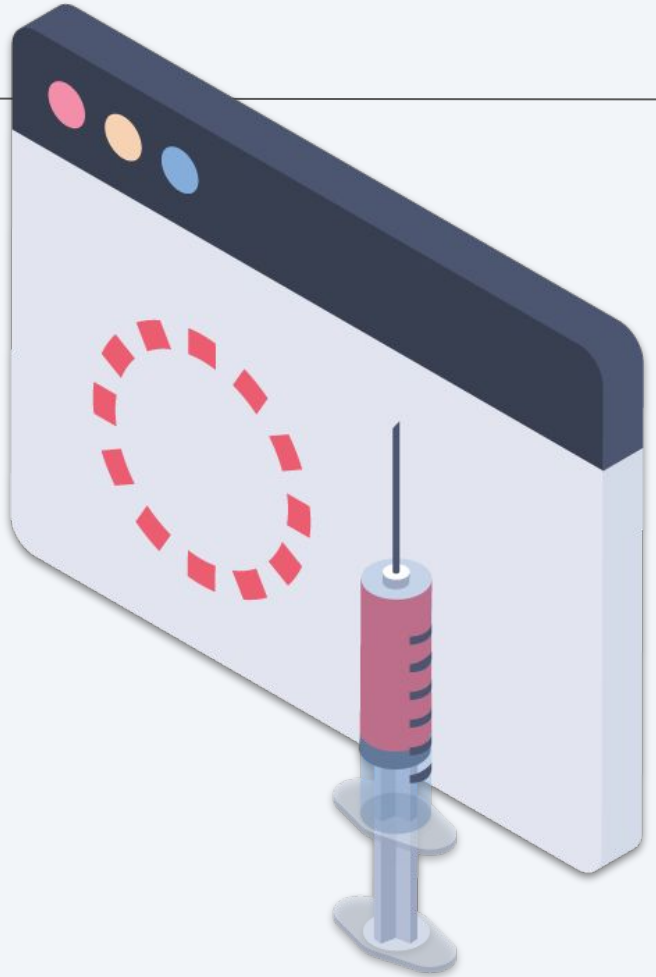
# The OWASP Top 10: No.3 Injections

| | | | |
|---|---|---|---|
| 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.



# Malicious code can be injected in fields and forms on web applications.

## Intended purpose of application

**Search Hotel**

What are you searching for? | SEARCH

Las Vegas

## Intended results

**Search Hotel**

Las Vegas | SEARCH

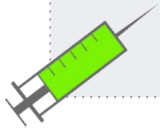
Las Vegas Convention Hotel:
2 Nights - 2 Adults $199 per night | BOOK NOW

## Applying injection to cause unintended results

**Search Hotel**

What are you searching for? | SEARCH

Malicious Script



## Unintended results

**Search Hotel**

Las Vegas | SEARCH

Success: Command Executed -
Hotel Database has been deleted | BOOK NOW

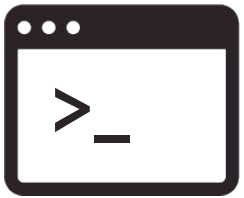
# Injection Types

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

## Cross-Site Scripting (XSS)

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.

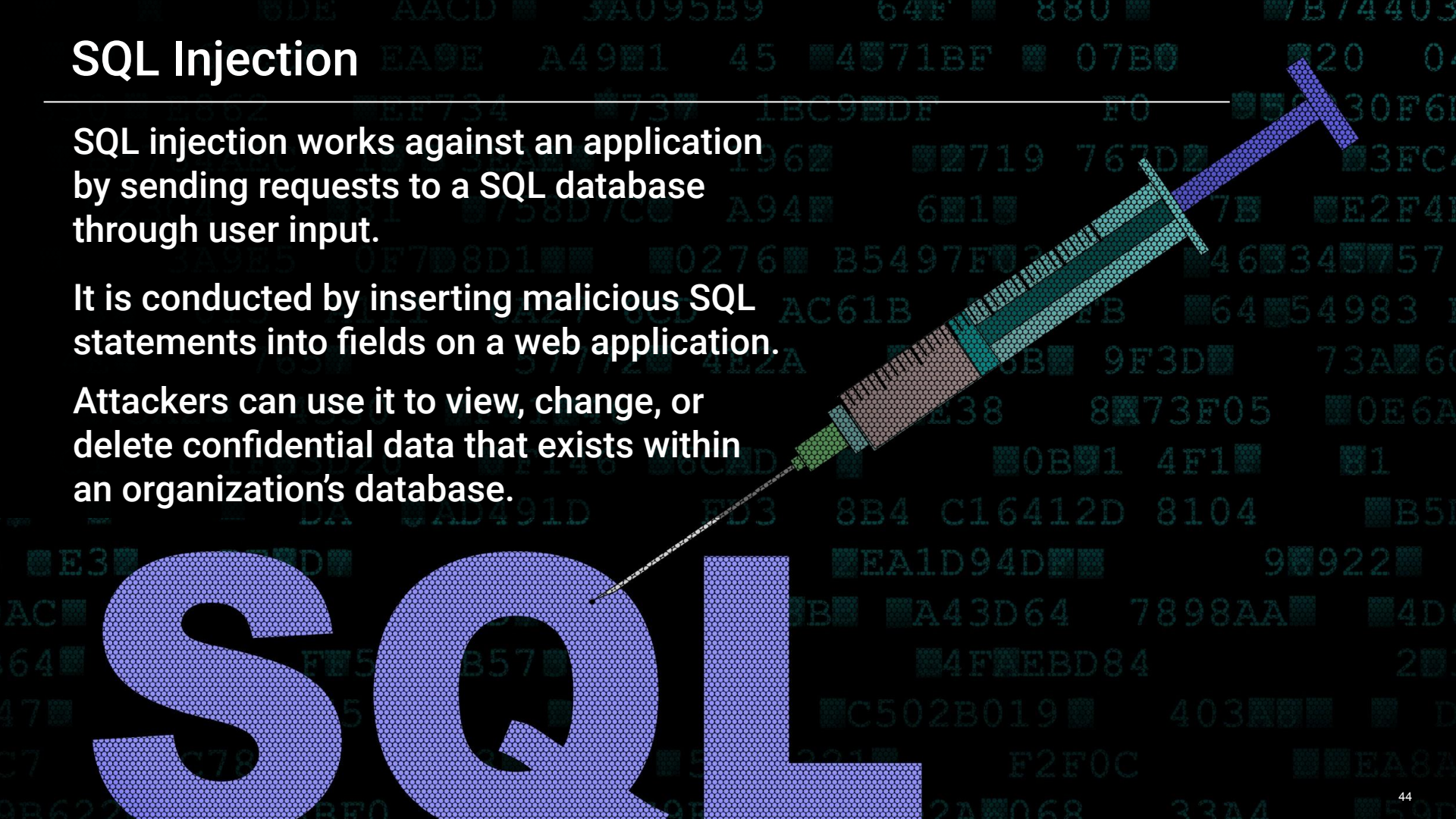
# SQL Refresher and Unintended SQL Queries

# SQL Injection

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

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

Attackers can use it to view, change, or delete confidential data that exists within an organization's database.



# 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. So, we will:

01 Cover how a web application connects to a database.

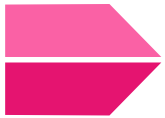
02 Review the structure of a SQL database and a SQL query.

03 Demonstrate an intended query and the intended results.

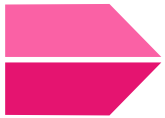
04 Cover modification of the intended query to create unintended results.

# How a Web Application Connects to a Database

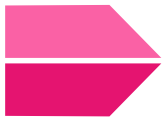
For this demonstration, we will play the role of security analysts at an organization that developed a new salary lookup application.



The new web application is called Salary Finder.



Employees can enter their user id and confirm their salary.

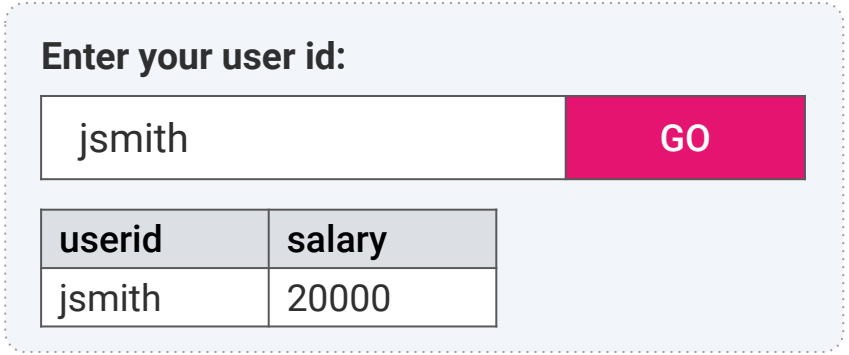


As security analysts, we are tasked with determining whether this application is vulnerable to SQL injection.

# How a Web Application Connects to a Database


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


Julie enters her user id, “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 user id:**

jsmith GO

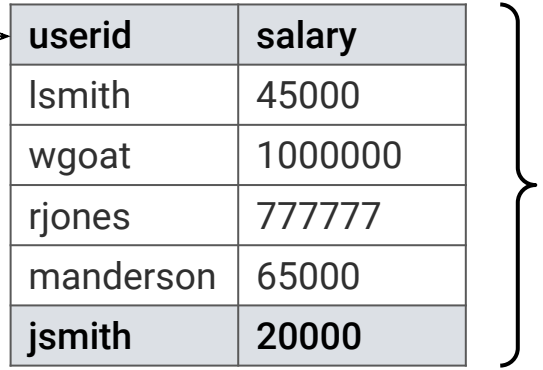
| userid | salary |
|---|---|
| jsmith | 20000 |

| 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** |

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

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

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

# Structure of a SQL Database and Query

`Userid, salary` indicates the two fields requested for display.

`where userid = 'jsmith'` instructs the program to only return records where the userid is “jsmith”.

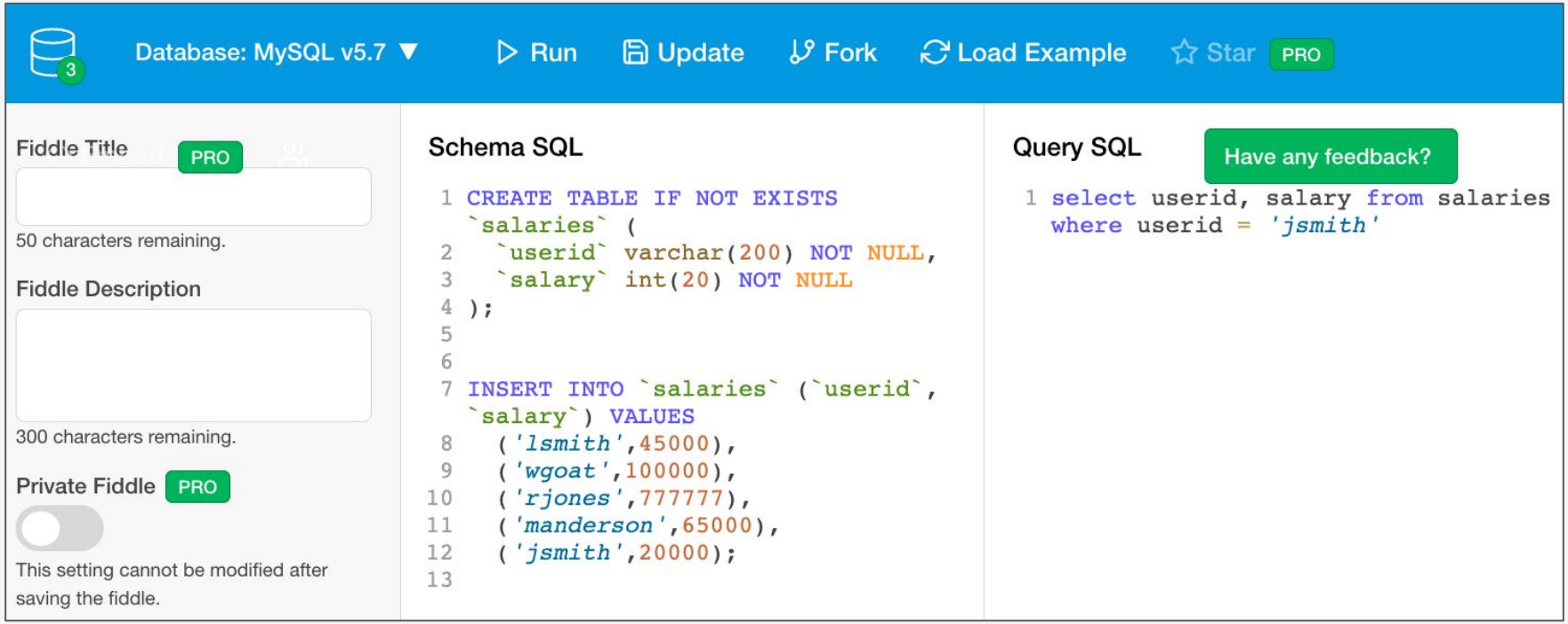

`select` is used to read or select data from a table.

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

| userid | salary |
|---|---|
| lsmith | 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 it returns intended results.



# 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

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

| | userid | salary | T/F |
|---|---|---|---|
| `Does userid = jsmith?` | lsmith | 45000 | **False** |
| `Does userid = jsmith?` | wgoat | 1000000 | **False** |
| `Does userid = jsmith?` | rjones | 777777 | **False** |
| `Does userid = jsmith?` | manderson | 65000 | **False** |
| `Does userid = jsmith?` | jsmith | 20000 | **True** |

`userid = jsmith`, so return salary.

# Always True

**Always true** statements are used to surpass all other conditions in a SQL query.

“Always true” describes a condition that always has a true result.

The most common always true statement is `1 = 1`.

If you had to give me a quarter every time `1 = 1`, you would always end up giving me a quarter, because 1 will always equal 1.

# Modification of Intended Query to Create Unintended Results

So we can add a second condition, an **always true statement**, to a query 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

To add this always true statement to the SQL query, we add `OR` and the always true statement as the second condition, as follows:

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

Malicious payload

# Instructor Demonstration

## Always True DB Fiddle

# Review

We've covered the following concepts so far:

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

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

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

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

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

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 Minutes

# Time's Up! Let's Review.

# Questions?

# 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 | Variety of Databases | Variety of Attacks |
| --- | --- | --- |
| In the real world, application security analysts would not have access to the database tables or structures. | In the real world, there are multiple database types in which SQL queries will not work the same way. | Other SQL injection attacks can delete or alter data, rather than just display it as we did. |

# Impact

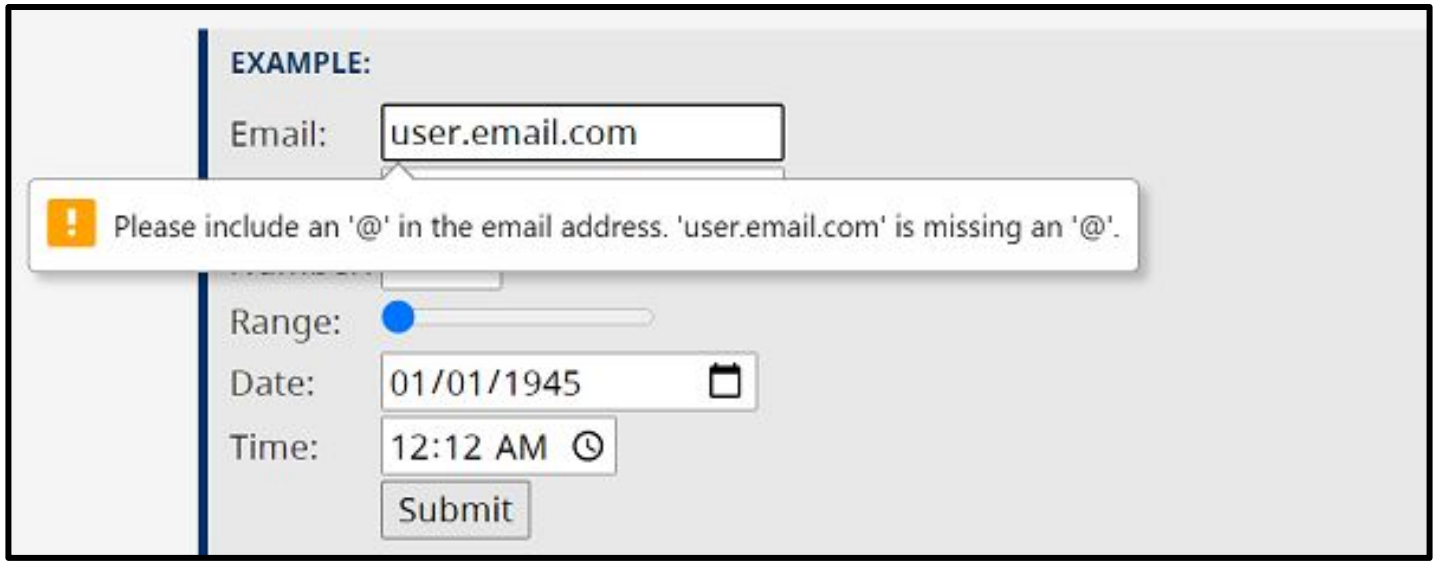
The reason that SQL injection is considered one of the most harmful attacks is due to its potential impact.

If a malicious actor 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.


https://www.computerworld.com

# Mitigation Methods

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



**Input validation** is a method of validating the data input with a predefined logic, ensuring that the input is what the application is expecting.

# Mitigation Methods

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

| | Client-side | Server-side |
|---|---|---|
| **Description:** | Client-side input validation involves coding the predefined logic into the webpage. | Server-side input validation involves adding the predefined logic into the code on the web server. |
| **Example:** | An input can only be chosen from a predefined drop-down menu. | 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. |

It is considered best practice to use server-side input validation, as methods can be applied to bypass client-side input validation.

# SQL Injection Summary

| | |
|---|---|
| **The intended purpose of the original function:** | Web applications interact with SQL databases by placing user input into SQL queries to display or modify data. |
| **The vulnerability and method of exploit:** | SQL injection involves a malicious user inserting a payload to run unexpected results. |
| **The unintended consequence of the exploit:** | The unintended consequence is the SQL query running a different query than intended. |
| **The mitigation of the vulnerability:** | Mitigation can be achieved by applying input validation code logic to the client- and server-side code. |
| **The potential impact of the exploit:** | The impact could include the unauthorized viewing, modification, or deletion of confidential data within an organization's databases. |

# Activity: Testing SQL Injection on Web Applications

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

Suggested Time:

20 Minutes

# Time's Up! Let's Review.

# Questions?

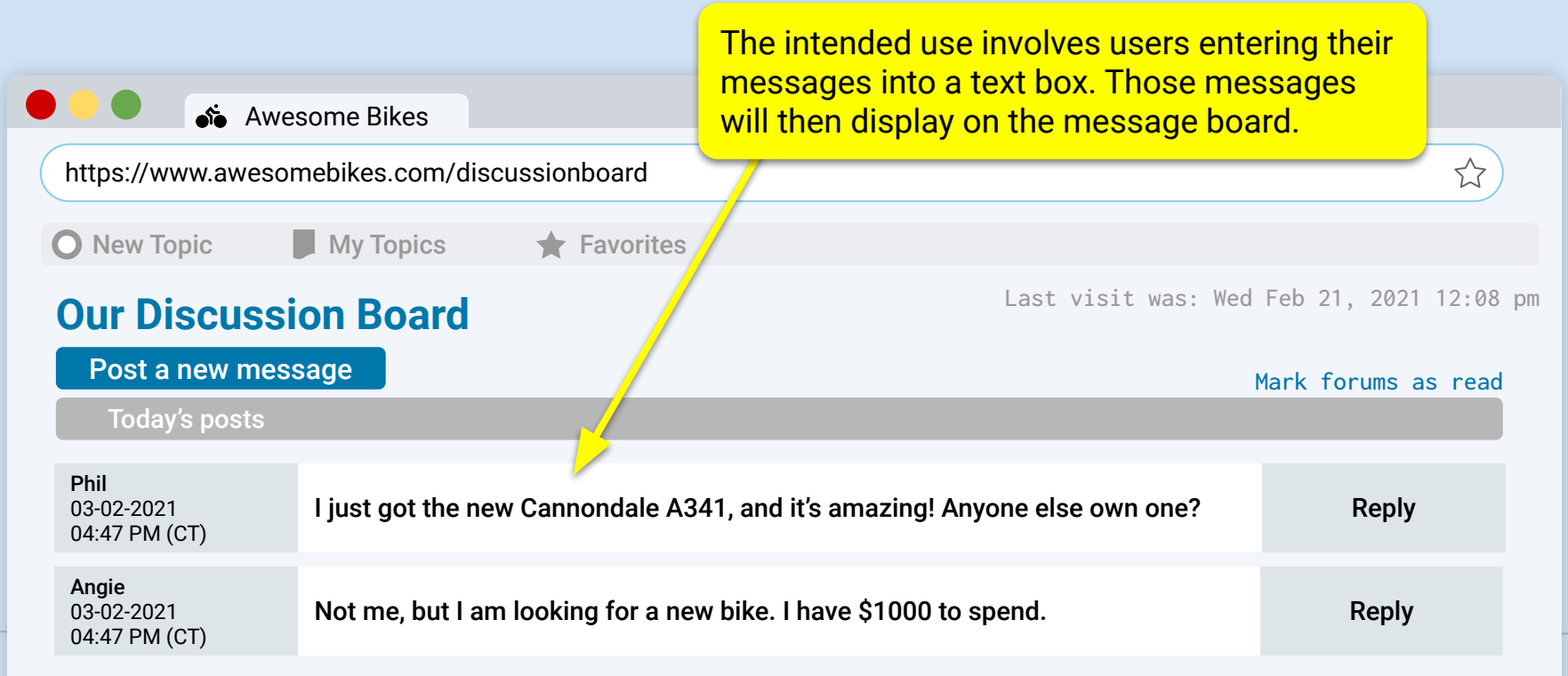

Break

# HTML & JavaScript

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

# XSS Scenario

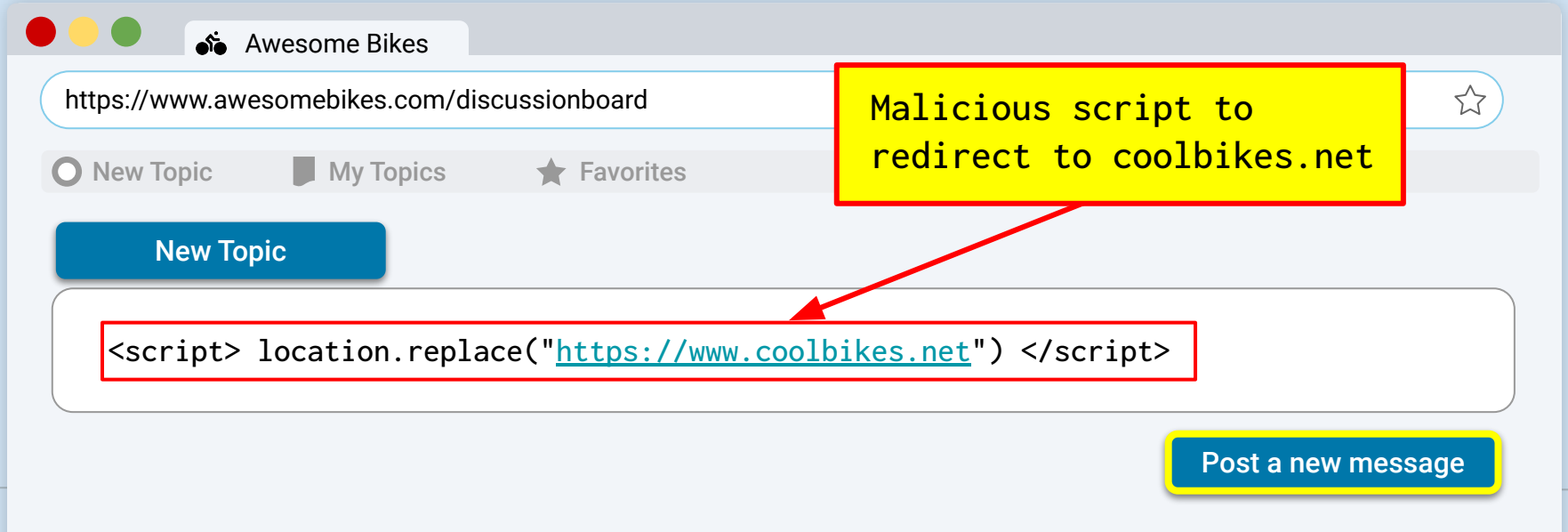
AwesomeBikes.com is an online message board intended for users to recommend bike models, ask questions, and chat about everything related to cycling.



# XSS Scenario

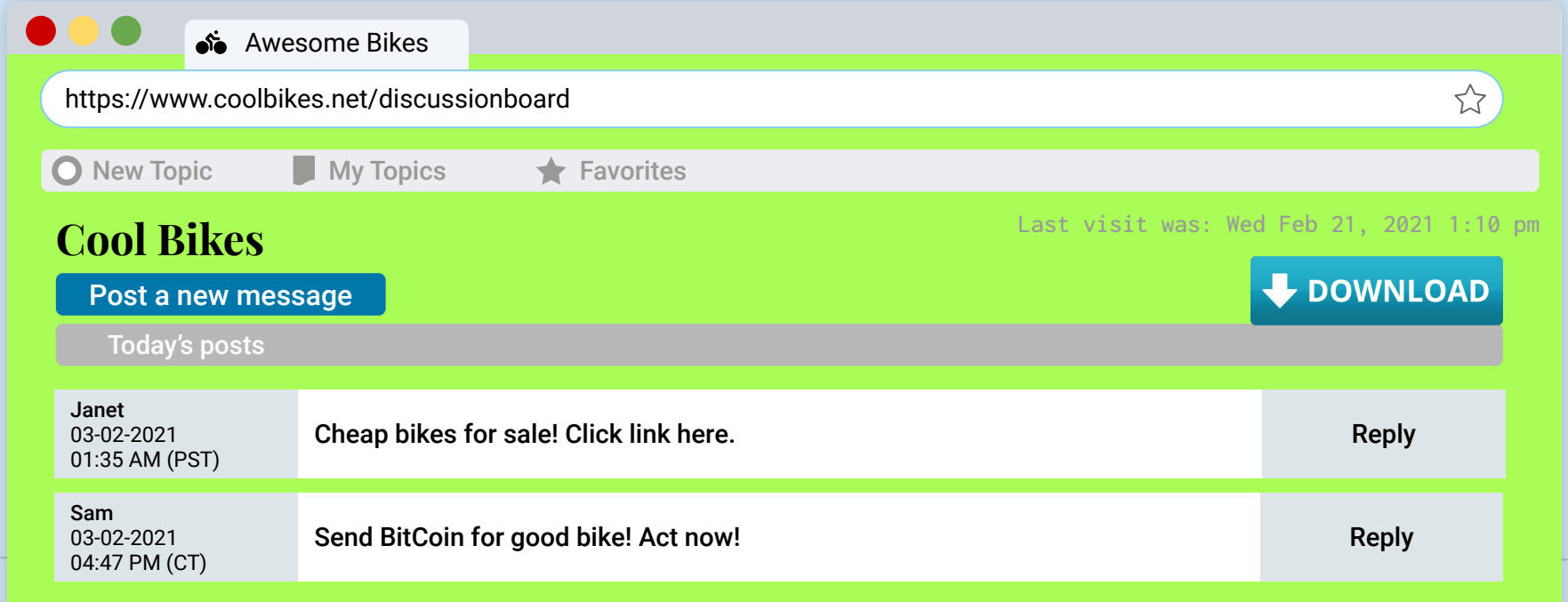
However, because this site is vulnerable to injection attacks, a malicious user can input a script in the text box.

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.



# 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.



# 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.


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

# HTML and JavaScript

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

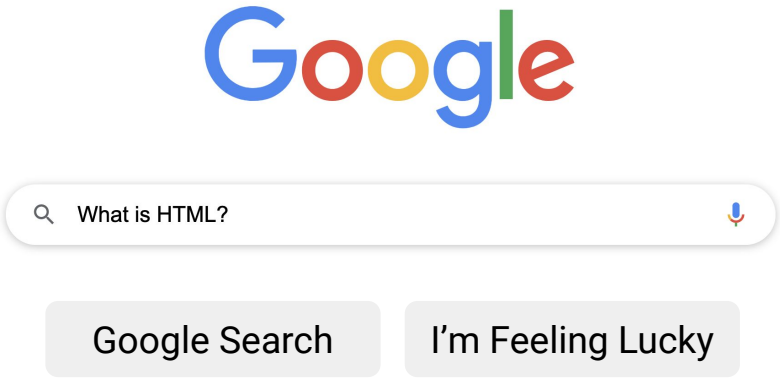
A language used to make webpages interactive and dynamic.

# HTML

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

**For example, if you want to access Google.com:**

| | |
|---|---|
| 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. |

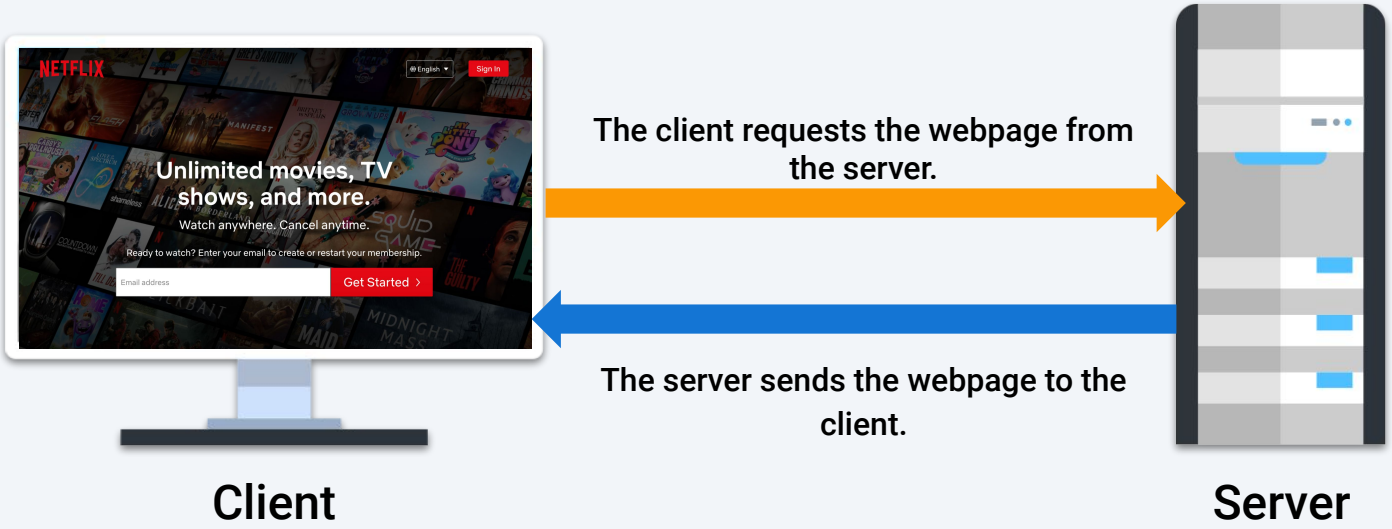


# What do you know about HTML?

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

# HTML

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



# HTML

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

The boundaries of a paragraph on a webpage, like where a paragraph starts and ends.

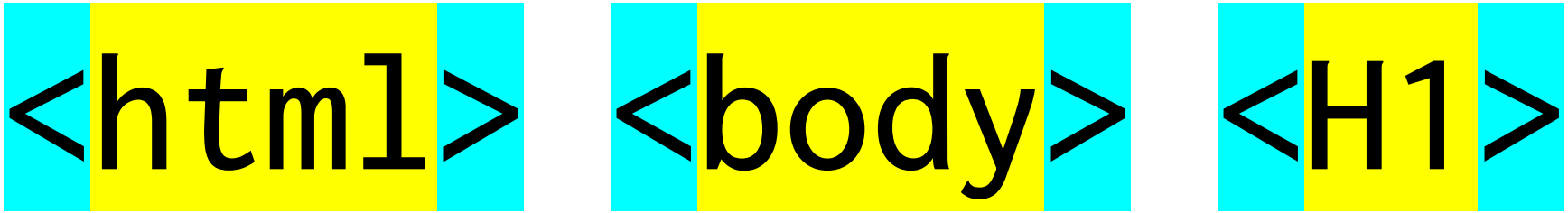
The size and boldness of headings on a webpage.

The placement of embedded images, video, or audio.

# HTML

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

Let’s look at an example of HTML code:

**<html>** indicates the start of HTML code.

**<body>** indicates the contents or attributes of the webpage.

**<h1>** indicates a heading on a website.

**I love Application Security!**
Indicates the text to be displayed in the heading.

**<h2>** indicates a heading, but at a smaller size.

**Excited to try Cross Site Scripting!**
Indicates the text to be displayed in the heading.

**</body>** closes the body of the HTML script.

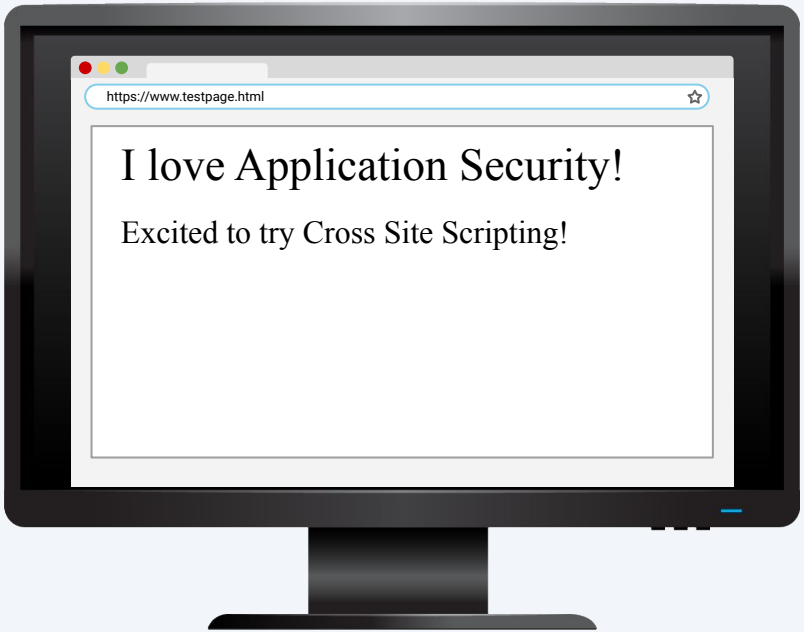
**</html>** closes the HTML.
A forward slash / indicates end of script.

```
<html>

<body>

<h1>I love Application Security!</h1>

<h2>Excited to try Cross Site Scripting!</h2>

</body>

</html>
```

# Sample HTML

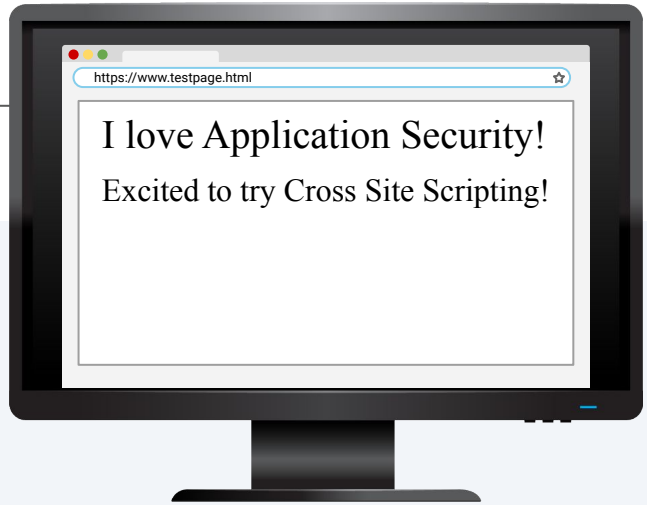
Let’s see what the webpage will look like when the browser renders this code:

```
<html>

<body>

<h1>I love Application Security!</h1>

<h2>Excited to try Cross Site Scripting!</h2>

</body>

</html>
```



# Sample HTML

Note that none of the tags themselves are displayed on the page, only the text between the tags.




```
<html>

<body>

<h1>I love Application Security!</h1>

<h2>Excited to try Cross Site Scripting!</h2>

</body>

</html>
```

The **<h1>** tag tells the browser to display the text “I love Application Security!” at a specific size.

The **<h2>** tag tells the browser to display the text “Excited to try Cross Site Scripting!” at a smaller size.

# Sample HTML

These are just a few of the many available HTML tags.

`<html>` `<body>` `<h1>` `<h2>`

We will soon explore other tags, including:

| | |
|---|---|
| `<b>` | Bolds text. |
| `<u>` | Underlines text. |

HTML provides options that can improve the aesthetic and design of a webpage.

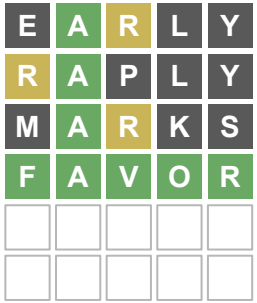
But it is limited to static improvements and cannot create more dynamic features.

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

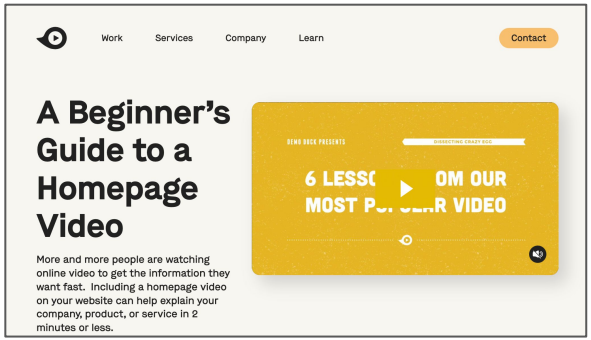
# JavaScript features include the following:

## Animations

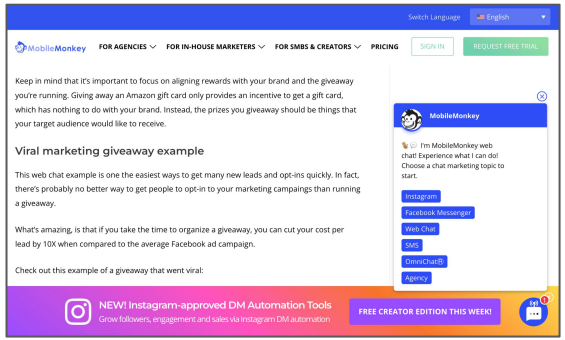

## Interactive games and maps




## Audio and video playback



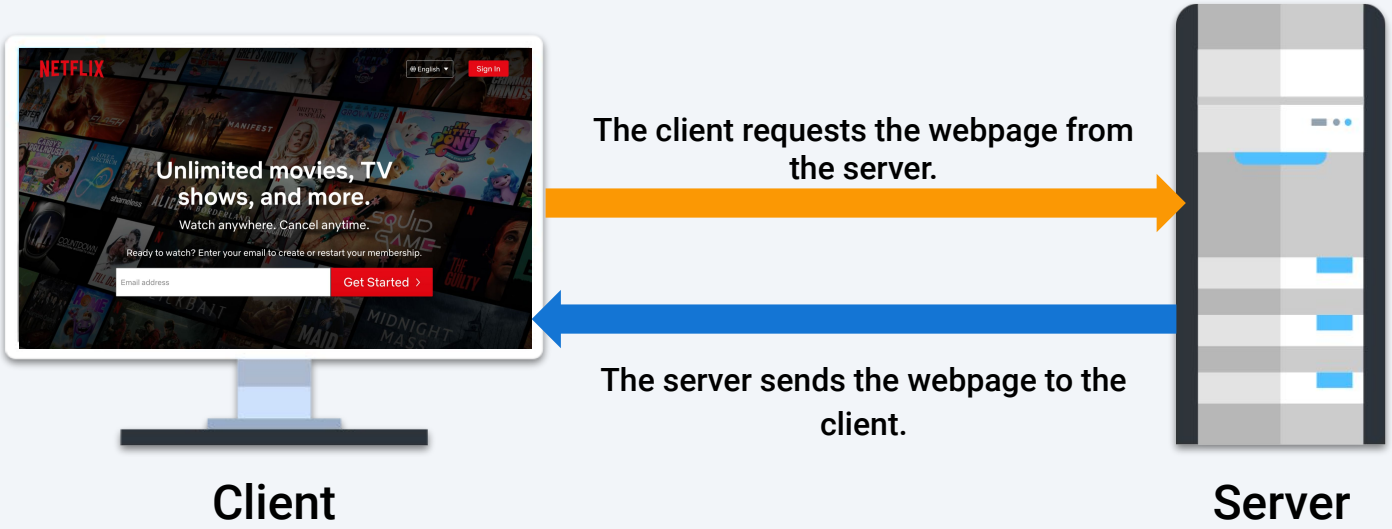

## Online chats




(Animation) (Video) (Online chat)

# HTML

JavaScript is also considered a **client-side language**, as it is designed to run on the user's browser. We add it into the HTML code by including a **<script>** tag.

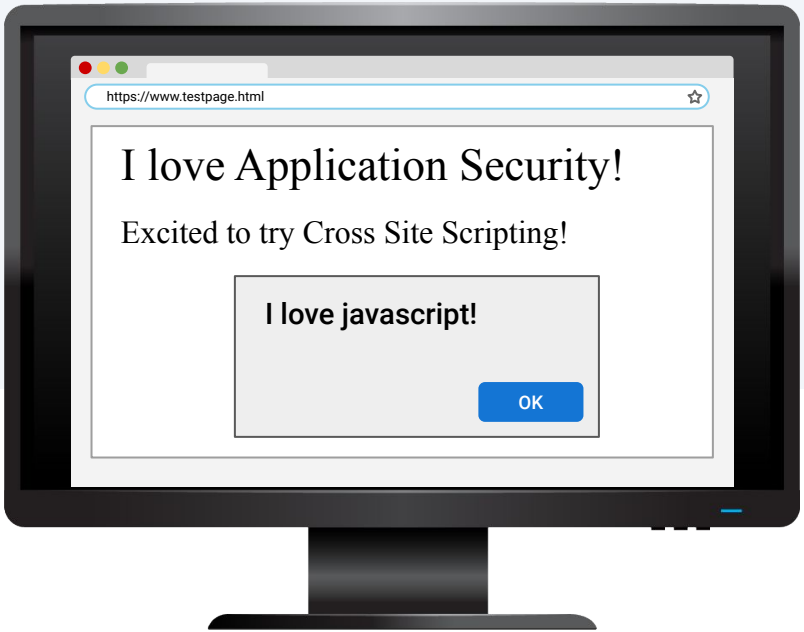


Let’s consider the same sample HTML code again, this time with an added JavaScript tag.

# Sample HTML

The browser displays the following webpage when it renders this code:

```
<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

# Questions?

# Testing XSS on Web Applications

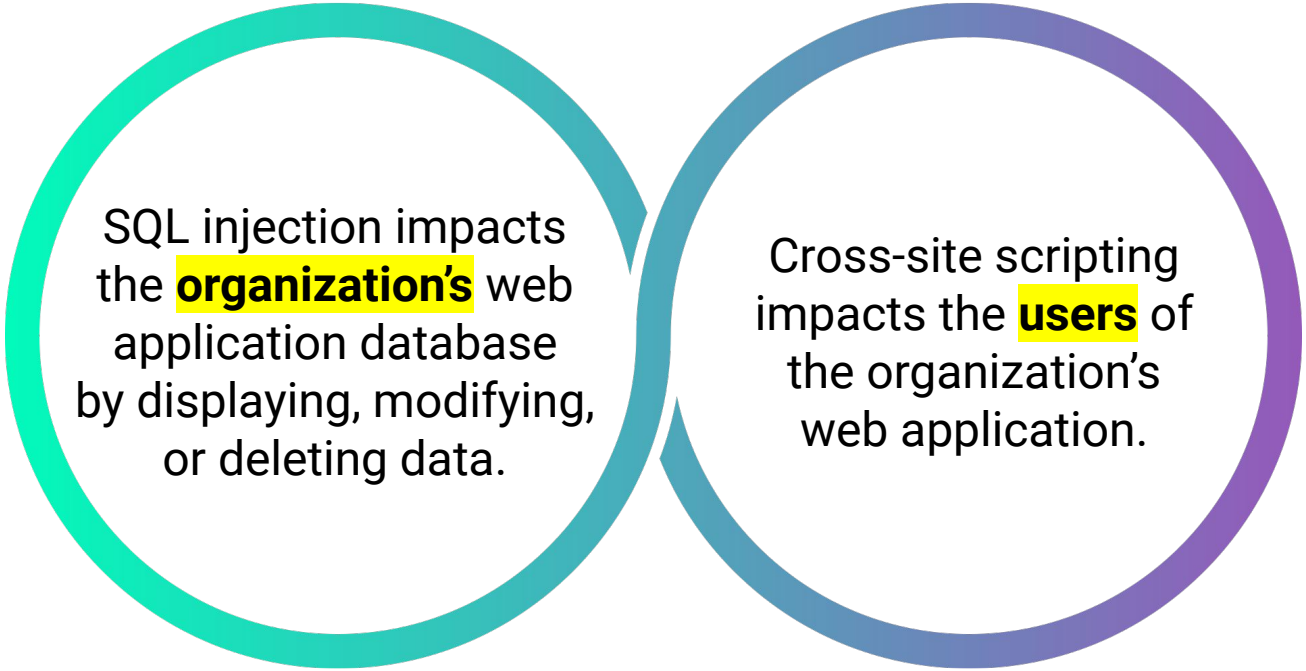
# The OWASP Top 10: Injection

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

| | | | |
|---|---|---|---|
| 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) |

# Cross-Site Scripting

While cross-site scripting is an injection attack similar to SQL injection, the impact is different for the following reasons:

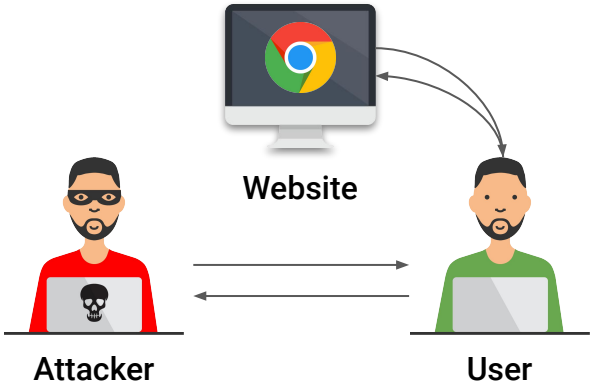


# Testing XSS on Web Applications

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

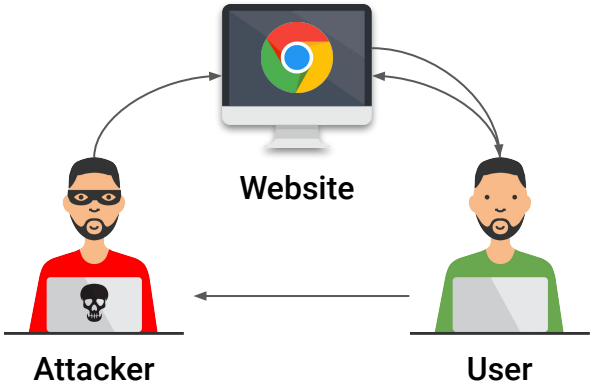
## Reflected (non-persistent) XSS

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 user accessing the web application.

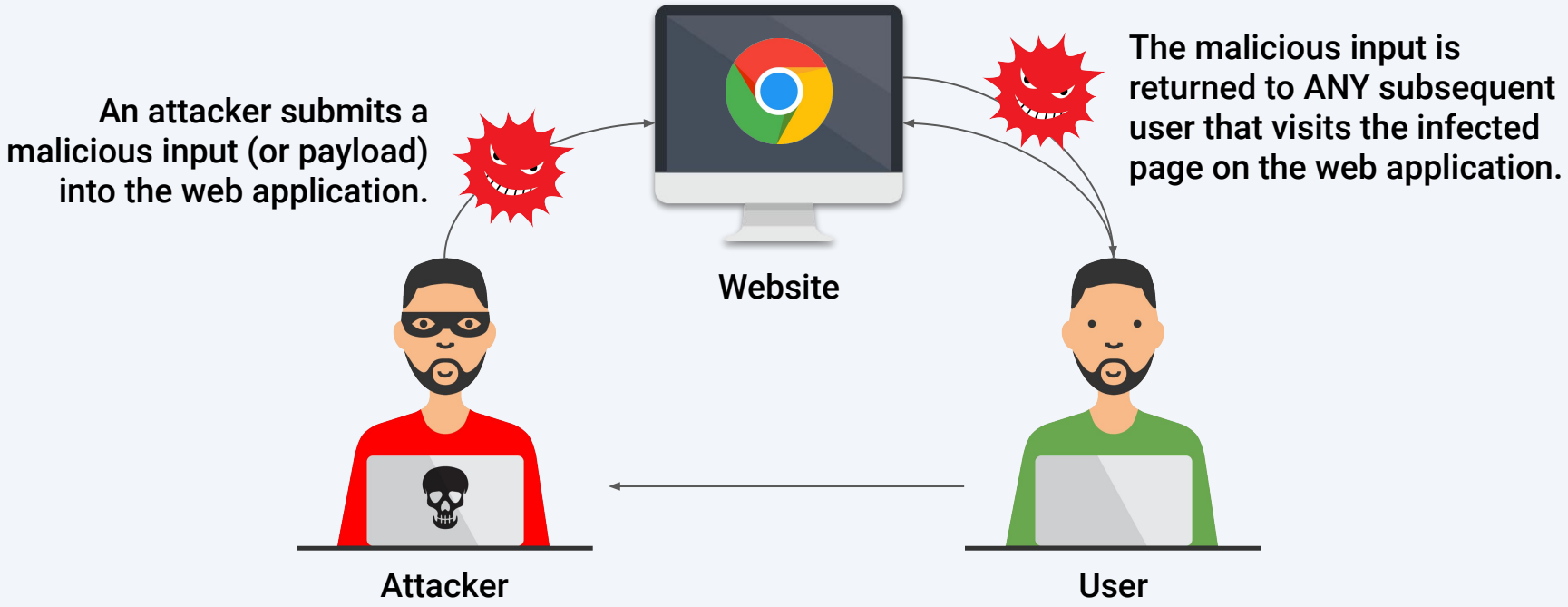


# Instructor Demonstration

## Stored XSS

# Stored XSS Recap

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



# Stored XSS Recap

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

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


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

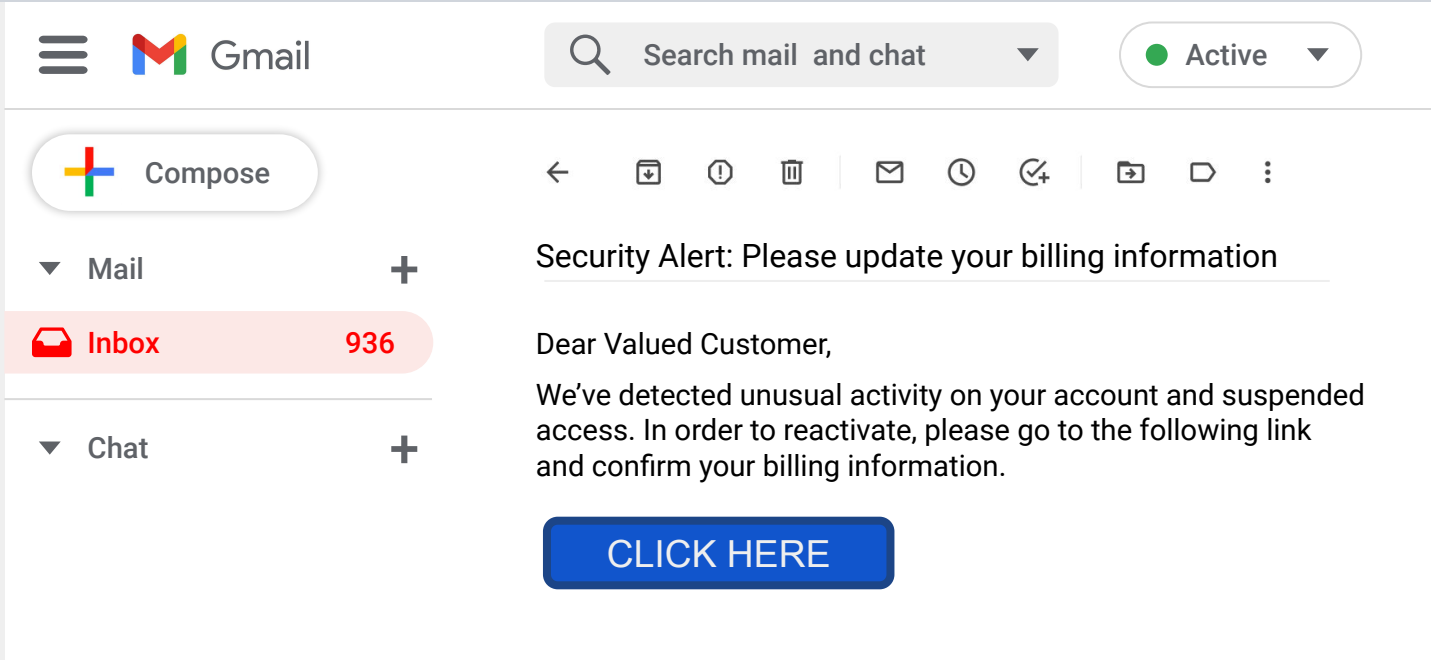
## Reflected XSS

# Delivering a Reflected XSS Attack

Delivering a stored XSS attack simply requires inputting malicious scripts to 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 user, and the user must be deceived into clicking the link.

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



# 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 user's cookies will be stolen and the session will be hijacked.
- The user's computer will be infected with malware.

## 02 Reflected XSS

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

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

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

# Mitigation Methods

**Input validation** is a method used to authenticate the data input with a predefined logic to ensure that the input is what the application is expecting.


## 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.

## 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, a submitted input that includes `<script>` or `</script>`.

# Cross-Site Scripting Summary

| | Stored XSS | Reflected XSS |
|---|---|---|
| The intended purpose of the original function: | A user's input is stored on the web application's server, and when returned, it modifies the source code of the webpage. | A user's input is immediately returned to the user to modify the source code of the webpage. |
| The vulnerability and method of exploit: | An attacker inputs a malicious script to a field, and subsequent users who access the web application will have the script run. | An attacker sends a crafted URL to a user, likely by a phishing email. Then the user accesses the URL, and a malicious script runs. |
| The unintended consequence of the exploit: | For stored and reflected XSS, the unintended consequence is that the source code will change on the webpage and a malicious script will run. | |
| The mitigation of the vulnerability: | For stored and reflected XSS, mitigation can be accomplished by applying input validation code logic to either the client- or server-side code. | |
| The potential impact of the exploit: | For stored and reflected XSS, the impact could include infecting the user's computer with malware or stealing their session cookies. | |

# Questions?

# Activity: Testing XSS on Web Applications

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

Suggested Time:

20 Minutes

# Time's Up! Let's Review.

# Questions?


The End