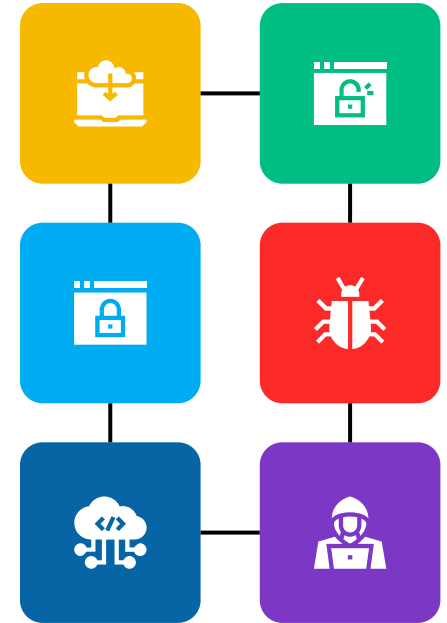


Web Application Attacks

MNU-2025

Dr. Ahmed Samy

Lecture 08



Module Contents

In this module, we will cover the below topics:



Introduction to Web Apps

Impact of Web Application Attacks



Common Web Application Attacks

Web App Attacks Mitigation

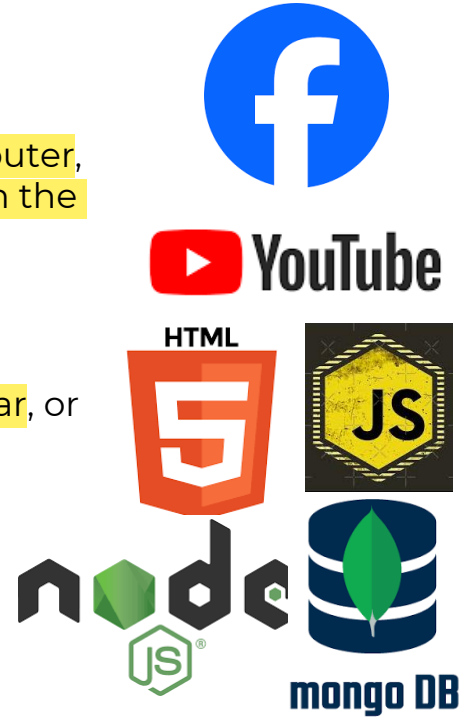


Web Applications



What is a Web Application?

- **A web application** is a software program that runs in a web browser and is accessed over the internet (or an intranet).
- Unlike traditional desktop applications that are installed on a personal computer, web applications are hosted on remote servers and require no installation on the user's device-just a browser and internet connection.
- **Technologies Used:**
- **Frontend (client-side):** HTML, CSS, JavaScript, frameworks like React, Angular, or Vue.
- **Backend (server-side):** Languages like Node.js, Python, PHP, Ruby, Java; databases like MySQL, MongoDB, PostgreSQL.
- **APIs:** For data exchange between client and server.

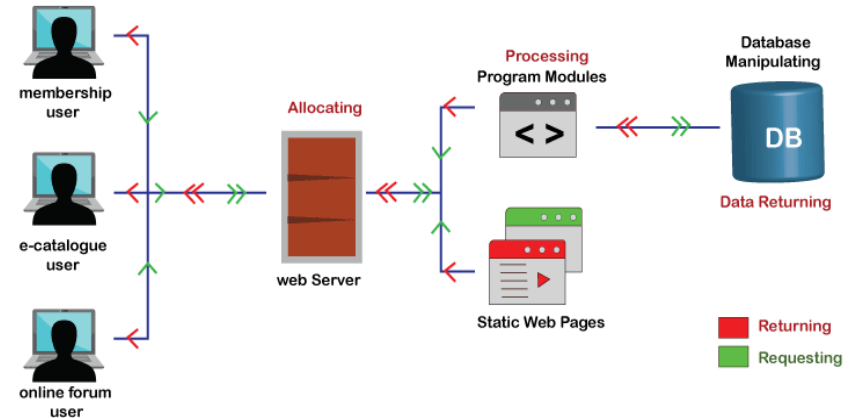


HTTP Methods

- The most common HTTP methods are:
 - GET: Retrieves data from the server.
 - POST: Sends data to the server to create a new resource.
 - PUT: Updates an existing resource on the server.
 - DELETE: Deletes a resource on the server.
 - PATCH: Partially updates a resource on the server.
 - HEAD: Retrieve the resource's header.
- A Uniform Resource Locators (URL) is a string of characters used to identify a resource on the internet. Web applications use URLs to direct users to specific pages or resources.
- URLs contain information such as the protocol, domain name, and path to the resource.
- Example: https://www.example.com/page1 —
 - Protocol: https
 - Domain name: www.example.com
 - Path: /page1

Key Characteristics of Web Applications

- **Interactivity:** User input and dynamic responses.
- **Data-Driven:** Rely on databases to store and retrieve information.
- **User Interface:** Rich UI built with HTML, CSS, and JavaScript.
- **Accessibility:** Can be accessed from various devices with a browser.
- **Connectivity:** Operate over a network (Internet).



Impact of Web Application Attacks

- Web application attacks can have severe consequences, affecting individuals, businesses, and organizations.
- The potential impact of web attacks are:
- **Data Breaches:** Attacks like SQL injection and cross-site scripting can expose sensitive information, including personal data, financial records, and intellectual property.
- **Financial Loss:** Attacks can lead to direct financial losses through theft, fraud, and disruption of business operations.
- **Reputational Damage:** A successful attack can erode trust in an organization, damaging its brand and reputation.
- **Legal and Regulatory Implications:** Organizations may face legal action and fines for failing to protect user data, as mandated by regulations like GDPR.



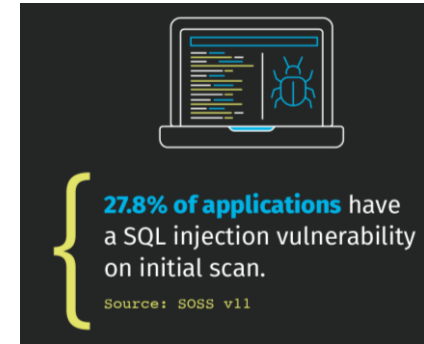
Common Web Application Attacks

Common Web Application Attacks

- Web applications are frequent targets for cyberattacks due to their exposure to the internet and interaction with sensitive data.
- Here are some of the most common web application attacks:
 1. **SQL Injection (SQLi)**
 2. **Cross-Site Scripting (XSS)**
 3. **Cross-Site Request Forgery (CSRF)**
 4. **Server-Side Request Forgery (SSRF)**
 5. **Broken Access Control**
 6. **File Upload Vulnerabilities**

1. What is SQL Injection?

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



```
MariaDB [demo]> SELECT * FROM users UNION SELECT * FROM articles;
+-----+-----+-----+
| id | username | password |
+-----+-----+-----+
| 1 | admin | Vaada7aPa55w0rd! |
| 1 | Article 1 | First article |
| 2 | Article 2 | Second article |
| 3 | Article 3 | Third article |
+-----+-----+-----+
4 rows in set (0.003 sec)
```

SQL Injection Example

- There are lots of SQL injection vulnerabilities, attacks, and techniques, that occur in different situations. Some common SQL injection examples include:
 1. **Retrieving hidden data:** where you can modify a SQL query to return additional results.
 2. **Subverting Application Logic:** where you can change a query to interfere with the application's logic.
 3. **UNION Attacks:** where you can retrieve data from different database tables.
 4. **Blind SQL Injection:** where the results of a query you control are not returned in the application's responses.



Retrieving hidden data

- Imagine a shopping application that displays products in different categories. When the user clicks on the Gifts category, their browser requests the URL:

```
https://insecure-website.com/products?category=Gifts
```

- This causes the application to make a SQL query to retrieve details of the relevant products from the database:

```
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
```

- This SQL query asks the database to return:
 - all details (*)
 - from the products table
 - where the category is Gifts
 - and released is 1.
- The restriction `released = 1` is being used to hide products that are not released. We could assume for unreleased products, `released = 0`.

Retrieving hidden data Cont.

- The application doesn't implement any defenses against SQL injection attacks. This means an attacker can construct the following attack, for example:

```
https://insecure-website.com/products?category=Gifts'--
```

- This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts'--' AND released = 1
```

- Crucially, note that '--' is a comment indicator in SQL. This means that the rest of the query is interpreted as a comment, effectively removing it.
- In this example, this means the query no longer includes AND released = 1.
- As a result, all products are displayed, including those that are not yet released.

Retrieving hidden data Cont.

- You can use a similar attack to cause the application to display all the products in any category, including categories that they don't know about:

```
https://insecure-website.com/products?category=Gifts'+OR+1=1--
```

- This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1
```

- The modified query returns all items where either the category is Gifts, or 1 is equal to 1.
- As 1=1 is always true, the query returns all items.

Subverting application logic

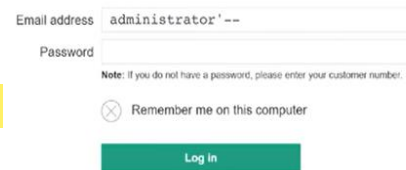
- Imagine an application that lets users log in with a username and password.
- If a user submits the username `wiener` and the password `bluecheese`, the application checks the credentials by performing the following SQL query:

```
SELECT * FROM users WHERE username = 'wiener' AND password = 'bluecheese'
```

- If the query returns the details of a user, then the login is successful. Otherwise, it is rejected.
- In this case, an attacker can log in as any user without the need for a password.
- They can do this using the SQL `comment sequence --` to remove the password check from the WHERE clause of the query.

```
SELECT * FROM users WHERE username = 'administrator'--' AND password = ''
```

- This query returns the user whose username is administrator and successfully logs the attacker in as that user.



Email address

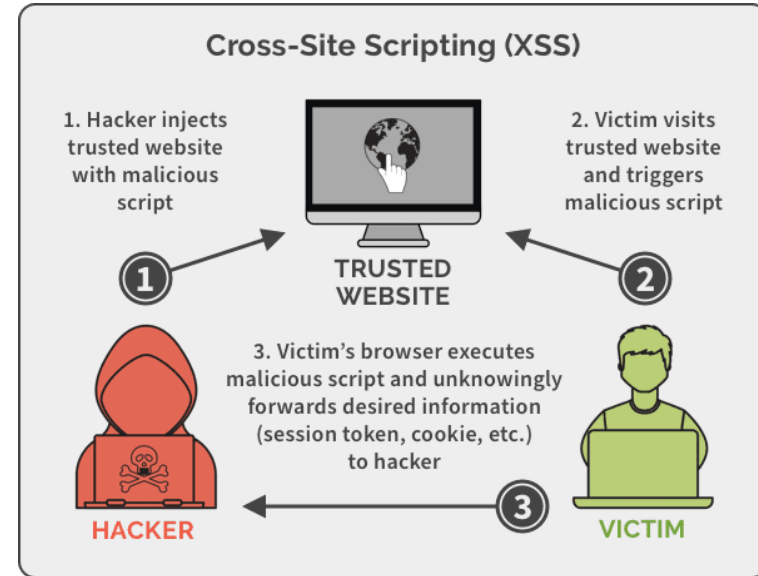
Password

Note: If you do not have a password, please enter your customer number.

☐ Remember me on this computer

2. What is Cross-Site Scripting (XSS)?

- **Cross-site scripting** works by manipulating a vulnerable web site so that it returns malicious JavaScript to users. When the malicious code executes inside a victim's browser, the attacker can fully compromise their interaction with the application.
- There are three main types of XSS attacks:
 1. **Reflected XSS:** where the malicious script comes from the current HTTP request.
 2. **Stored XSS:** where the malicious script comes from the website's database.
 3. **DOM-based XSS:** where the vulnerability exists in client-side code rather than server-side code.



2.1 What is Reflected cross-site scripting?

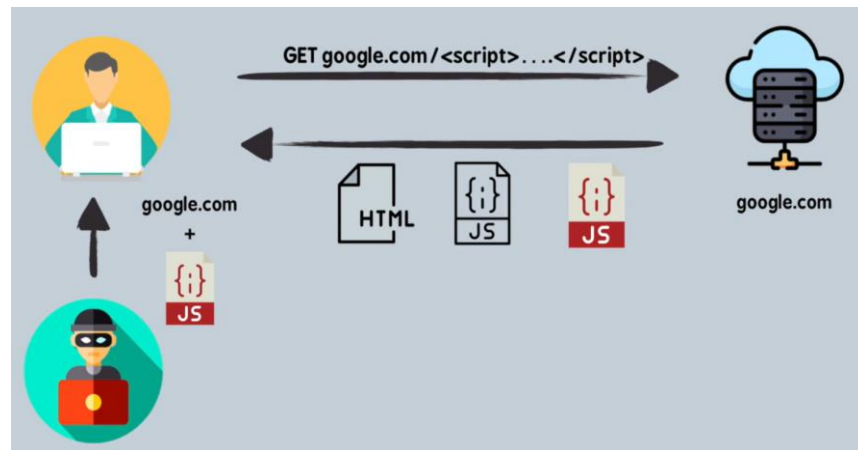
- Reflected XSS is the simplest variety of cross-site scripting. It arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.
- The hacker send a URL contains a script to the victim (www.google.com + `<script> Bad Script </script>`).
 - When the victim click on the link, the victim's browser will request the web page (www.google.com) from the web server.
 - The web server will send the HTML, Java Script, and the malicious script to the victim browser.

- Try the below site to try the XSS**

<https://xss-game.appspot.com/level1>

- Type below script in the search:
`<script>alert("xss is here")</script>`
- If anyone click the below URL it will run the script.

[https://xss-game.appspot.com/level1/frame?query=<script>alert\("xss+is+here"\)</script>](https://xss-game.appspot.com/level1/frame?query=<script>alert('xss+is+here')</script>)



Reflected cross-site scripting Example

- Suppose a website has a search function which receives the user-supplied search term in a URL parameter:

```
https://insecure-website.com/search?term=gift
```

- The application echoes the supplied search term in the response to this URL:

```
<p>You searched for: gift</p>
```

- Assuming the application doesn't perform any other processing of the data, an attacker can construct an attack like this:

```
https://insecure-website.com/search?term=<script>/*+Bad+stuff+here...+*/</script>
```

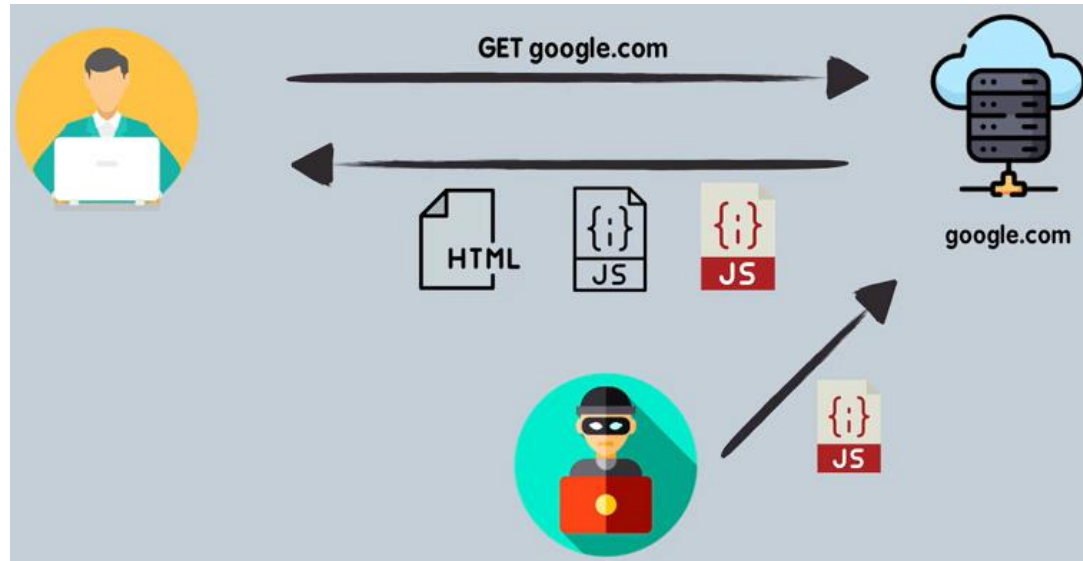
- This URL results in the following response:

```
<p>You searched for: <script>/* Bad stuff here... */</script></p>
```

- If another user of the application requests the attacker's URL, then the script supplied by the attacker will execute in the victim user's browser, in the context of their session with the application.

2.2 What is Stored cross-site scripting?

- **Stored XSS** (also known as persistent or second-order XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.
1. The hacker stores a malicious code (Java Script code) inside a specific web page.
 2. When the users visit this page, the server will reply with the normal content of this page and the hacker's Java Script.



Reflected cross-site scripting Example

- Suppose a website allows users to submit comments on blog posts, which are displayed to other users. Users submit comments using an HTTP request like the following:

```
POST /post/comment HTTP/1.1
Host: vulnerable-website.com Content-Length: 100

postId=3&comment=This+post+was+extremely+helpful.&name=Carlos+Montoya&email=carlos%40normal-user.net
```

- After this comment has been submitted, any user who visits the blog post will receive the following within the application's response:

```
<p>This post was extremely helpful.</p>
```

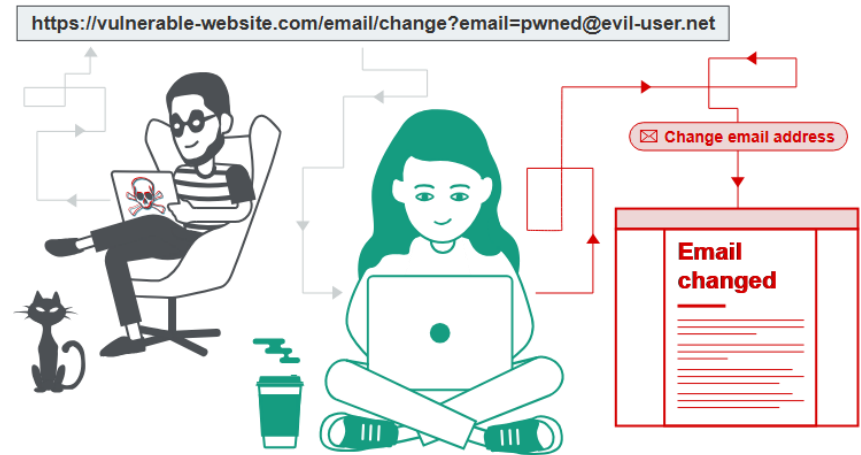
- Assuming the application doesn't perform any other processing of the data, an attacker can submit a malicious comment like this:

```
<script>/* Bad stuff here... */</script>
```

- The script supplied by the attacker will then execute in the victim user's browser, in the context of their session with the application.

3. What is Cross-Site Request Forgery (CSRF)?

- **Cross-site request forgery** allows an attacker to induce users to perform actions that they do not intend to perform.
- **Example:**
 - The attacker causes the victim user to carry out an action unintentionally. This might be to change the email address on their account, to change their password, or to make a funds transfer.
 - Depending on the nature of the action, the attacker might be able to gain full control over the user's account.
 - If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality.



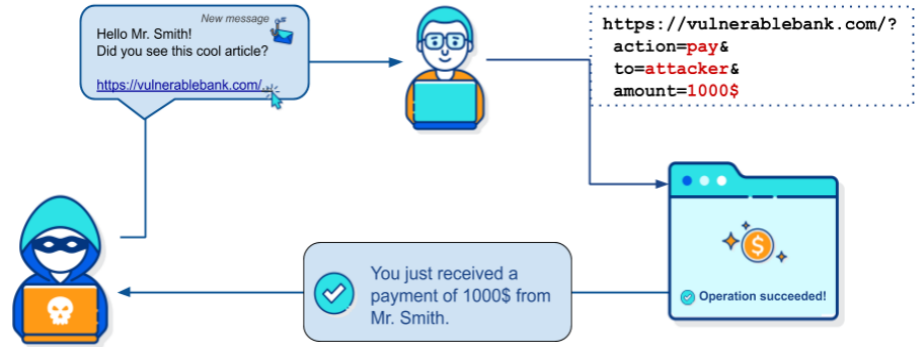
Cross-Site Request Forgery Example

- CSRF attacks typically comprise two actions: **tricking the victim** into clicking a link or loading page by social engineering or phishing, **then sending a legitimate-looking, crafted request** to the website from the victim's browser.

- A user logs into **www.vulnerablebank.com** using forms authentication.
- The server authenticates the user. The response from the server includes an **authentication cookie**.
- Without logging out, the user visits a malicious web site, e.g. **www.attackerwebsite.com**. The malicious site contains the following HTML form:

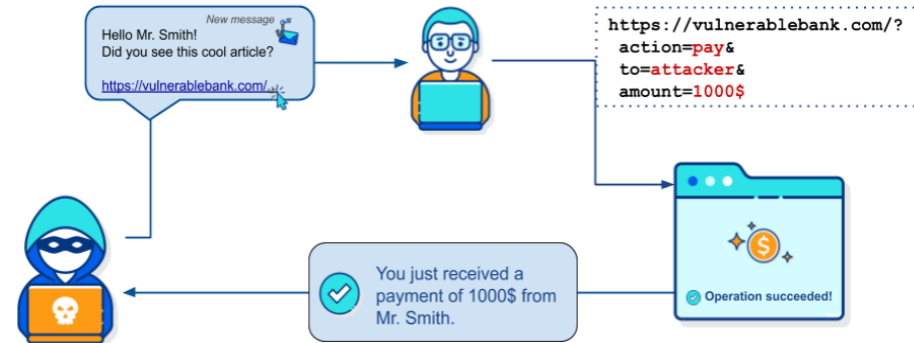
```
<form action="https://www.vulnerablebank.com/api/account" method="POST">
<input type="hidden" name="action" value="pay">
<input type="hidden" name="amount" value="1000">
<input type="submit" value="Click Me"> </form>
```

- Notice that the form action posts to the vulnerable site, **not to the malicious site**. This is the 'cross-site' part of CSRF.



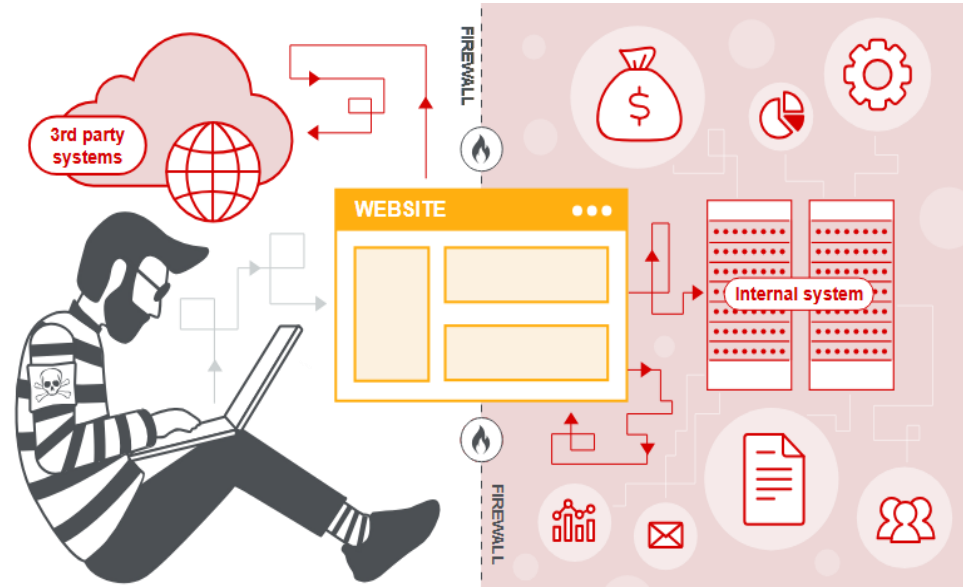
Cross-Site Request Forgery Example Cont.

4. The user clicks the submit button. The browser includes the authentication cookie with the request.
5. The request runs on the server with the user's authentication context and can do anything that an authenticated user is allowed to do.



4. What is Server-Side Request Forgery (SSRF)?

- **Server-side request forgery** is a web security vulnerability that **allows an attacker to cause the server-side application to make requests to an unintended location.**
- In a typical SSRF attack, the attacker might cause the server to make a connection to internal-only services within the organization's infrastructure.
- In other cases, they may be able to force the server to connect to arbitrary external systems. This could **leak sensitive data**, such as authorization credentials.



Server-Side Request Forgery Example

- Imagine a shopping application that lets the user view whether an item is in stock in a particular store.
- To provide the stock information, the application must query various back-end REST APIs. It does this by passing the URL to the relevant back-end API endpoint via a front-end HTTP request. When a user views the stock status for an item, their browser makes the following request:

```
POST /product/stock HTTP/1.0 Content-Type: application/x-www-form-urlencoded
Content-Length: 118
```

```
stockApi=http://stock.weliketoshop.net:8080/product/stock/check%3FproductId%3D6%26storeId%3D1
```

- This causes the server to make a request to the specified URL, retrieve the stock status, and return this to the user.
- In this example, an attacker can modify the request to specify a URL local to the server:

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118
```

```
stockApi=http://localhost/admin
```

Server-Side Request Forgery Example Cont.

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://localhost/admin
```

- The server fetches the contents of the `/admin URL` and returns it to the user.
- An attacker can visit the `/admin URL`, but the administrative functionality is normally only accessible to authenticated users. This means an attacker won't see anything of interest.
- However, if the request to the `/admin URL` comes from the local machine, the normal access controls are bypassed.
- The application grants full access to the administrative functionality, because the request appears to originate from a trusted location.

5. What is Broken Access Control Attack?

- Access control enforces policy such that users cannot act outside of their intended permissions.
- **Common access control vulnerabilities include:**
 1. Violation of the principle of least privilege or deny by default.
 2. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool modifying API requests.
 3. Permitting viewing or editing someone else's account, by providing its unique identifier (insecure direct object references)
 4. Accessing API with missing access controls for POST, PUT and DELETE.
 5. Elevation of privilege. Acting as a user without being logged in or acting as an admin when logged in as a user.
 6. Force browsing to authenticated pages as an unauthenticated user or to privileged pages as a standard user.

Broken Access Control Example

- Administrative functions might be linked from an administrator's welcome page but not from a user's welcome page. However, a user might be able to access the administrative functions by browsing to the relevant admin URL.
- For example, a website might host sensitive functionality at the following URL:

```
https://insecure-website.com/admin
```

- This might be accessible by any user, not only administrative users who have a link to the functionality in their user interface. In some cases, the administrative URL might be disclosed in other locations, such as the robots.txt file:

```
https://insecure-website.com/robots.txt
```

- Even if the URL isn't disclosed anywhere, an attacker may be able to use a wordlist to brute-force the location of the sensitive functionality.

Broken Access Control Example cont.

- In some cases, sensitive functionality is hidden by giving it a less predictable URL. This is an example of so-called "security by obscurity".
- However, hiding sensitive functionality does not provide effective access control because users might discover the obfuscated URL in a number of ways.
- Imagine an application that hosts administrative functions at the following URL:
`https://insecure-website.com/administrator-panel-yb556`
- This might not be directly guessable by an attacker. However, the application might still leak the URL to users. The URL might be disclosed in JavaScript that constructs the user interface based on the user's role:

```
<script>
  var isAdmin = false;
  if (isAdmin) {
    ...
    var adminPanelTag = document.createElement('a');
    adminPanelTag.setAttribute('href', 'https://insecure-website.com/administrator-panel-yb556');
    adminPanelTag.innerText = 'Admin panel';
    ...
  }
</script>
```

Broken Access Control Example cont.

```
<script>
  var isAdmin = false;
  if (isAdmin) {
    ...
    var adminPanelTag = document.createElement('a');
    adminPanelTag.setAttribute('href', 'https://insecure-website.com/administrator-panel-yb556');
    adminPanelTag.innerText = 'Admin panel';
    ...
  }
</script>
```

- This script adds a link to the user's UI if they are an admin user. However, the script containing the URL is visible to all users regardless of their role.

Broken access control resulting from platform misconfiguration.

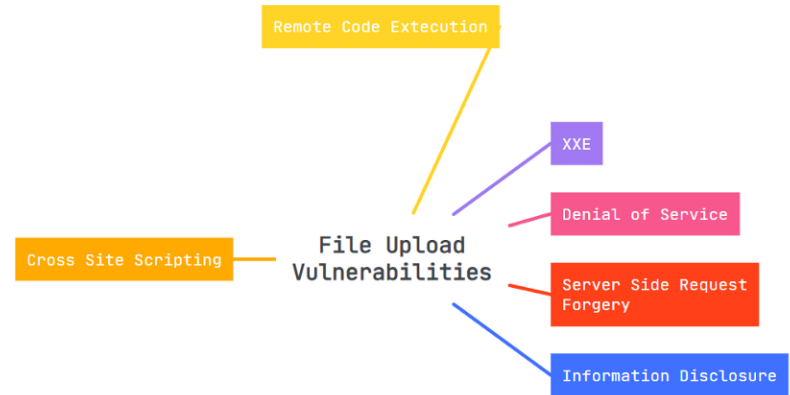
- Some applications enforce access controls at the platform layer. They do this by restricting access to specific URLs and HTTP methods based on the user's role. For example, an application might configure a rule as follows:

```
DENY: POST, /admin/deleteUser, managers
```
- This rule denies access to the POST method on the URL /admin/deleteUser, for users in the managers group. Various things can go wrong in this situation, leading to access control bypasses.
- Some application frameworks support various non-standard HTTP headers that can be used to override the URL in the original request, such as X-Original-URL and X-Rewrite-URL.
- If a website uses strict front-end controls to restrict access based on the URL, but the application allows the URL to be overridden via a request header, then it might be possible to bypass the access controls using a request like the following:

```
POST / HTTP/1.1
X-Original-URL: /admin/deleteUser
...
```

6. What is File Upload Vulnerabilities?

- File upload vulnerabilities are when a web server allows users to upload files to its filesystem without sufficiently validating things like their name, type, contents, or size.
- Failing to properly enforce restrictions on these could mean that even a basic image upload function can be used to upload arbitrary and potentially dangerous files instead.
- This could even include server-side script files that enable remote code execution.



Exploiting unrestricted file uploads to deploy a web shell

- From a security perspective, the worst possible scenario is when a website allows you to upload server-side scripts, such as PHP, Java, or Python files, and is also configured to execute them as code. This makes it trivial to create your own web shell on the server.

A web shell is a malicious script that enables an attacker to execute arbitrary commands on a remote web server simply by sending HTTP requests to the right endpoint.

- If you're able to successfully upload a web shell, you effectively have full control over the server.
- For example, the following PHP one-liner could be used to read arbitrary files from the server's filesystem:

```
<?php echo file_get_contents('/path/to/target/file'); ?>
```

- Once uploaded, sending a request for this malicious file will return the target file's contents in the response.

Web Application Attacks Mitigation

Mitigation Techniques

- **Secure Development Practices (Security by Design):**
 - **Input Validation and Sanitization:** This is paramount. Always validate and sanitize all user-supplied input (from forms, URLs, headers, APIs, etc.) on the server-side.
 - **Secure Coding Standards:** Adhere to secure coding guidelines and best practices for the programming languages and frameworks you are using.
 - **Static and Dynamic Application Security Testing (SAST/DAST):** Integrate SAST tools into your development pipeline to analyze code for vulnerabilities without executing it. Use DAST tools to test the running application for security flaws by simulating attacks.
- **Web Application Firewall (WAF):** WAFs can protect against common web application attacks like SQL Injection, XSS, Cross-Site Request Forgery (CSRF), and more
- **Access Control and Authentication:** Implement robust authentication mechanisms to verify user identities.
- **Data Protection:**
 - **Encryption**
 - **Data Minimization**
 - **Data Masking and Tokenization**

Mitigation Techniques

- **Infrastructure Security:**
 - **Network Segmentation**
 - **Firewalls**
 - **Intrusion Detection System**
 - **Regular Security Patching**
 - **Hardening**
- **Monitoring and Logging**

Thanks!

Do you have any questions?

