INFS3202/7202 – Web Information Systems

Lecture Week 11: Web Security

Dr Aneesha Bakharia (Senior Lecturer, EECS)
Email: a.bakharia1@uq.edu.au

Contents

Course Updates

Issue/Feedback	Change
RiPPLE Weekly Tasks	• Weekly Tasks have concluded.
Code Reviews in Week 10	• Thank you for attending the Code Review • If you have an extension please attend a Lab on Tuesday to complete the Code Review
Project Assessment Item	• Due Friday Week 12 at 3pm • 2 weeks remaining

What is Web Security?

What is Web Security?

- Web security encompasses the strategies, protocols, and tools designed to protect websites, web applications, and users from threats on the internet.

- **Definition:**
  Protecting web servers, web applications, and user data from attacks such as hacking, malware, and breaches.

- **Objective:**
  Ensure data integrity, confidentiality, and availability across web environments.

- **Challenges:**
  Dealing with threats like SQL injection, XSS (Cross-Site Scripting), and DDoS (Distributed Denial of Service) attacks.

- **Importance:**
  Vital for safeguarding sensitive information, maintaining user trust, and complying with regulations.

OPTUS Data Breach in 2022

Wikipedia The Free Encyclopedia

Search Wikipedia | Search | Create account | Log in

2022 Optus data breach

Add languages

Article | Talk | Read | Edit | View history | Tools

From Wikipedia, the free encyclopedia

Contents hide

(Top)
Background
Breach
Government response
Optus response
Legal action
See also
References

In September 2022, Australian telecommunications company Optus suffered a data breach that affected up to 10 million current and former customers comprising a third of Australia's population. Information was illegally obtained, including names, dates of birth, home addresses, telephone numbers, email contacts, and numbers of passports and driving licences. Conflicting claims about how the breach happened were made; Optus presented it as a complicated attack on its systems while an Optus insider and the Australian Government said a human error caused a vulnerability in the company's API. A ransom notice asking for A$1,500,000 to stop the data from being sold online was issued. After a few hours, the data thieves deleted the ransom notice and apologised for their actions.

Government figures, including Home Affairs and Cyber Security Minister Clare O'Neil, and Minister for Government Services Bill Shorten, criticised Optus for its role in the attack, and for being uncooperative with government agencies and the public. The government announced legislation, including the allowance of information-sharing with financial services and government agencies, and reforms to Australia's laws on security of critical infrastructure to help the government act in the event of future breaches.[1] In response to the data breach, Optus agreed to pay for the replacements of compromised passports, commissioned an external review, and gave seriously affected customers a subscription to a credit monitoring service. Optus also apologised for the breach. Customers criticized Optus for not being responsive and providing inadequate responses to those affected. As of June 2023, investigations into the breach and a class-action lawsuit from affected customers were ongoing.

Background [edit]

See also: Optus

Optus, an Australian telecommunications company, was founded in 1981 with the formation of the government-owned satellite-communications company AUSSAT.[2] AUSSAT was privatised in 1991 and sold to a consortium that included Mayne Nickless and AMP Limited.[3] In 2022, Optus was Australia's third-largest telecommunications company with a 13.1% market share.[4] In September 2022, Optus had around 10 million customers, comprising more than a third of Australia's population of around 26.12 million people.[5][6]

Breach [edit]

On 20 September 2022, Optus's technical team noticed and investigated suspicious activity on its network. The next day, Optus's systems were found to have sustained a data breach and regulators were informed. On 22 September, the company publicly announced the data breach and informed news agencies.[5][7] Optus advised the public to be vigilant for potential fraudulent activity but stated it did not know whether the breach had caused any harm to customers. Optus did not state how many customers were affected or whether the theft of data had caused harm.[8] Illegally obtained Information included names, dates of birth, home addresses, telephone numbers, email contacts, and passport and driving-licence numbers.[5]

https://en.wikipedia.org/wiki/2022_Optus_data_breach

What caused the OPTUS Data Breach?

- The Optus data breach of September 2022, occurred through an unprotected and publically exposed API. This API didn't require user authentication before facilitating a connection. A lack of an authentication policy meant anyone that discovered the API on the internet could connect to it without submitting a username or password.

- The API exposed sensitive private data
  e.g. Drivers License, Medicare number, etc

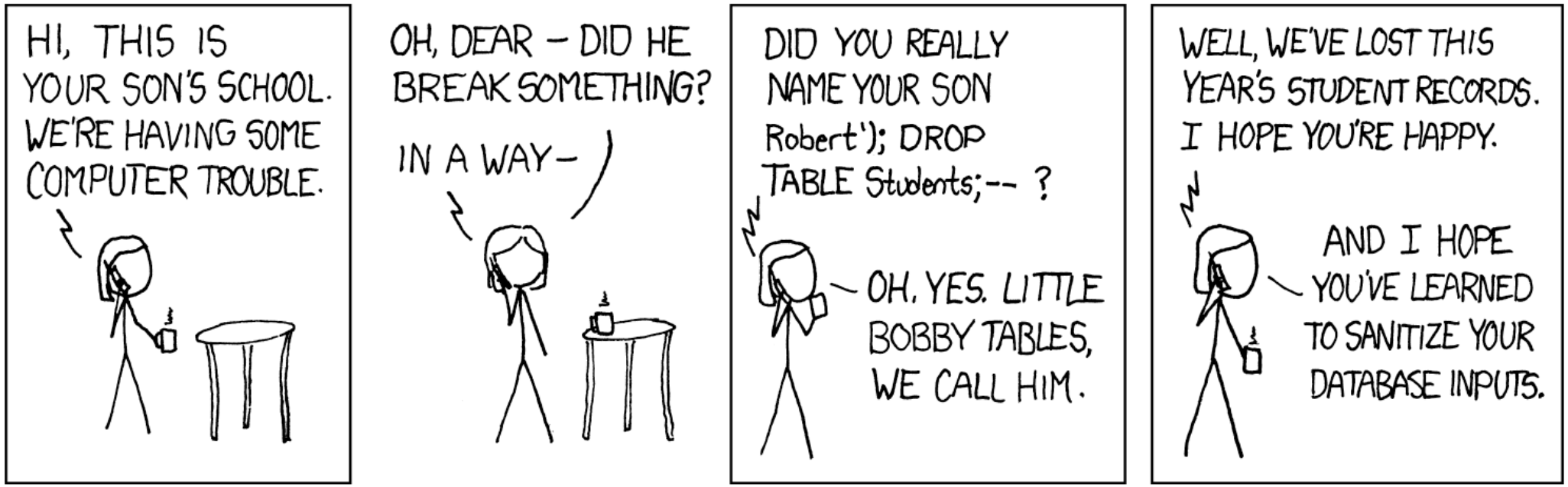
- The URL to the API had a auto-increment client number
  It was easy to automate calls to the API to get the data from another user

  e.g. https://somedomain.com/getuserdetails?user_id=10

  From: https://www.upguard.com/blog/how-did-the-optus-data-breach-happen

- Authentication and Authorization are very important across all URL and APIs
- Secure all Routes by default, have well defined authorization and open only as needed making sure to test that the data make available is not sensitive.

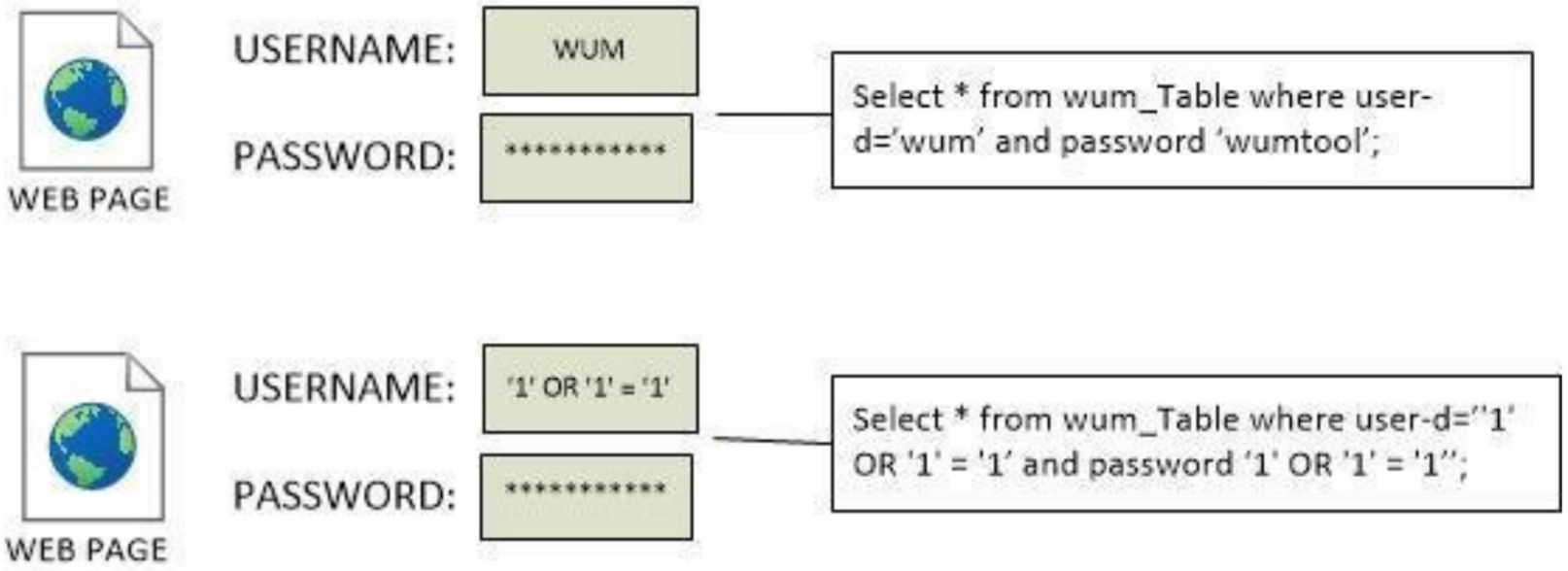
SQL Injection



https://xkcd.com/327/

SQL Injection

- SQL injection typically happens when a web application requests user input, such as a username or user ID. If the input provided is an SQL statement or part of an SQL statement instead of legitimate data, the application may execute this statement on the database without realizing its malicious intent.



https://www.researchgate.net/figure/A-SQL-injection-attack_fig3_322250414

SQL Injection

- An attacker can inject SQL control characters and command keywords (e.g.,single quote (‘), double quote (“), equal (=), comment (- -), etc.) to change the query structure.

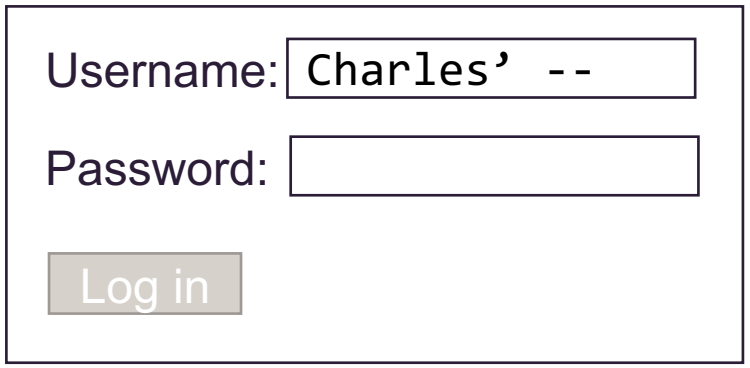
```
SELECT * FROM user WHERE name=‘Charles’ AND passwd=‘topsecret’
```

Attacks: `Charles’ --` ←

-- : the remainder of the statement is to be treated as a comment and not executed

The executed SQL statement becomes:

*SELECT * FROM user WHERE name=‘Charles’ -- ‘ AND passwd=‘ ’*



Unauthorized access to the backend database

An attacker can successfully bypass an application’s authentication mechanism

SQL Injection – Retrieving Hidden Data (1)

- URL

  https://somesite.com/products?category=handbags

- Category is included in a Query checking if released is 1

  SELECT * FROM products WHERE category = ‘handbags' AND released = 1

- URL with SQL Comment
  https://somesite.com/products?category=handbags' --

- Resulting Query will return all items

  SELECT * FROM products WHERE category = ‘handbags’--’ AND released = 1

SQL Injection – Retrieving Hidden Data (2)

- URL

  https://somesite.com/products?category=handbags

- Category is included in a Query checking if released is 1

  SELECT * FROM products WHERE category = ‘handbags' AND released = 1

- URL with SQL Comment
  https://somesite.com/products?category=handbags'+OR+1=1--

- Resulting Query will return all items

  SELECT * FROM products WHERE category = ‘handbags' OR 1=1--' AND released = 1

SQL Injection – Running Additional Queries (1)

- URL

  https://somesite.com/products?category=handbags

- Category is included in a Query checking if released is 1

  SELECT * FROM products WHERE category = ‘handbags' AND released = 1

- URL with SQL Comment
  https://somesite.com/products?category=handbags’;delete * from users --

- Delete Query will be executed on users table.

- Update queries as well a unions with other tables can be returned

SQL Injection – Running Additional Queries (2)

```
SELECT * FROM user WHERE name=‘Charles’ AND passwd=‘topsecret’
```

Attacks:

```
Charles’; INSERT INTO groupMembership (userID, group)
VALUES (SELECT userID FROM users
WHERE userName=‘Charles', 'Administrator'); --
```

The executed SQL statement becomes: Change authorization information

```
SELECT * FROM user WHERE name=‘ Charles’;
INSERT INTO groupMembership (userID, group)
VALUES (SELECT userID FROM users WHERE userName=‘Charles', 'Administrator');
```

PHP with SQL Injection Vulnerability

```php
<?php
// Assume a POST request with a username field
$username = $_POST['username'];

// SQL query that is vulnerable to SQL injection
$query = "SELECT * FROM users WHERE username = '$username'";

// Executing the query
$result = mysqli_query($conn, $query);

// Check if user exists
if ($row = mysqli_fetch_assoc($result)) {
    echo 'Logged in as: ' . $row['username'];
} else {
    echo 'User not found.';
}
?>
```

Prepared Statements in PHP

```php
<?php
// Assume a POST request with a username field
$username = $_POST['username'];

// Create a prepared statement
if ($stmt = $conn->prepare("SELECT * FROM users WHERE username = ?")) {

    // Bind the input parameter to the prepared statement as a string
    $stmt->bind_param("s", $username);

    // Execute the query
    $stmt->execute();

    // Get the result
    $result = $stmt->get_result();

    // Check if user exists
    if ($row = $result->fetch_assoc()) {
        echo 'Logged in as: ' . $row['username'];
    } else {
        echo 'User not found.';
    }

    // Close the statement
    $stmt->close();
} else {
    echo 'Prepared statement creation failed: ' . $conn->error;
}
?>
```

- This version of the code uses a prepared statement, which separates the data (user input) from the code (SQL query), thus preventing SQL injection.
- The bind_param function binds the user input to a placeholder in the SQL query (?), specifying "s" to treat it as a string.
- This ensures that the input is handled safely, regardless of its content.

Preventing SQL Injection in CodeIgniter (1)

Escaping Values

It's a very good security practice to escape your data before submitting it into your database. CodeIgniter has three methods that help you do this:

1. $db->escape()

This function determines the data type so that it can escape only string data. It also automatically adds single quotes around the data so you don't have to:

```
<?php

$sql = 'INSERT INTO table (title) VALUES(' . $db->escape($title) . ')';
```

2. $db->escapeString()

This function escapes the data passed to it, regardless of type. Most of the time you'll use the above function rather than this one. Use the function like this:

```
<?php

$sql = "INSERT INTO table (title) VALUES('" . $db->escapeString($title) . "')";
```

3. $db->escapeLikeString()

This method should be used when strings are to be used in LIKE conditions so that LIKE wildcards (`%` , `_`) in the string are also properly escaped.

```
<?php

$search = '20% raise';
$sql    = "SELECT id FROM table WHERE column LIKE '%" . $db->escapeLikeString($search) . "%' ESCAPE '!'";
```

- If you must create SQL directly and concatenate with variable that may come from a url querystring or user input then escape the values.

https://codeigniter.com/user_guide/database/queries.html

Preventing SQL Injection in CodeIgniter (2)

Where

$builder->where()

This method enables you to set **WHERE** clauses using one of five methods:

Note

All values passed to this method are escaped automatically, producing safer queries, except when using a custom string.

Note

`$builder->where()` accepts an optional third parameter. If you set it to `false`, CodeIgniter will not try to protect your field or table names.

1. Simple key/value method

```php
<?php

$builder->where('name', $name);
// Produces: WHERE name = 'Joe'
```

Notice that the equal sign is added for you.

If you use multiple method calls they will be chained together with **AND** between them:

```php
<?php

$builder->where('name', $name);
$builder->where('title', $title);
$builder->where('status', $status);
// WHERE name = 'Joe' AND title = 'boss' AND status = 'active'
```

- Use the QueryBuilder
- Values are by default escaped.

https://codeigniter.com/user_guide/database/query_builder.html#where

Preventing SQL Injection in CodeIgniter (3)

find()

Returns a single row where the primary key matches the value passed in as the first parameter:

```php
<?php

$user = $userModel->find($user_id);
```

The value is returned in the format specified in $returnType.

You can specify more than one row to return by passing an array of primaryKey values instead of just one:

```php
<?php

$users = $userModel->find([1, 2, 3]);
```

Note

If no parameters are passed in, `find()` will return all rows in that model's table, effectively acting like `findAll()`, though less explicit.

- Uses Models
- ->find() by default provides protection

https://codeigniter.com/user_guide/models/model.html#find

Cross-site Scripting

What is Cross-site Scripting?

What is XSS?

Injection security attack
An attacker injects malicious client-side scripts into the content from trusted websites.

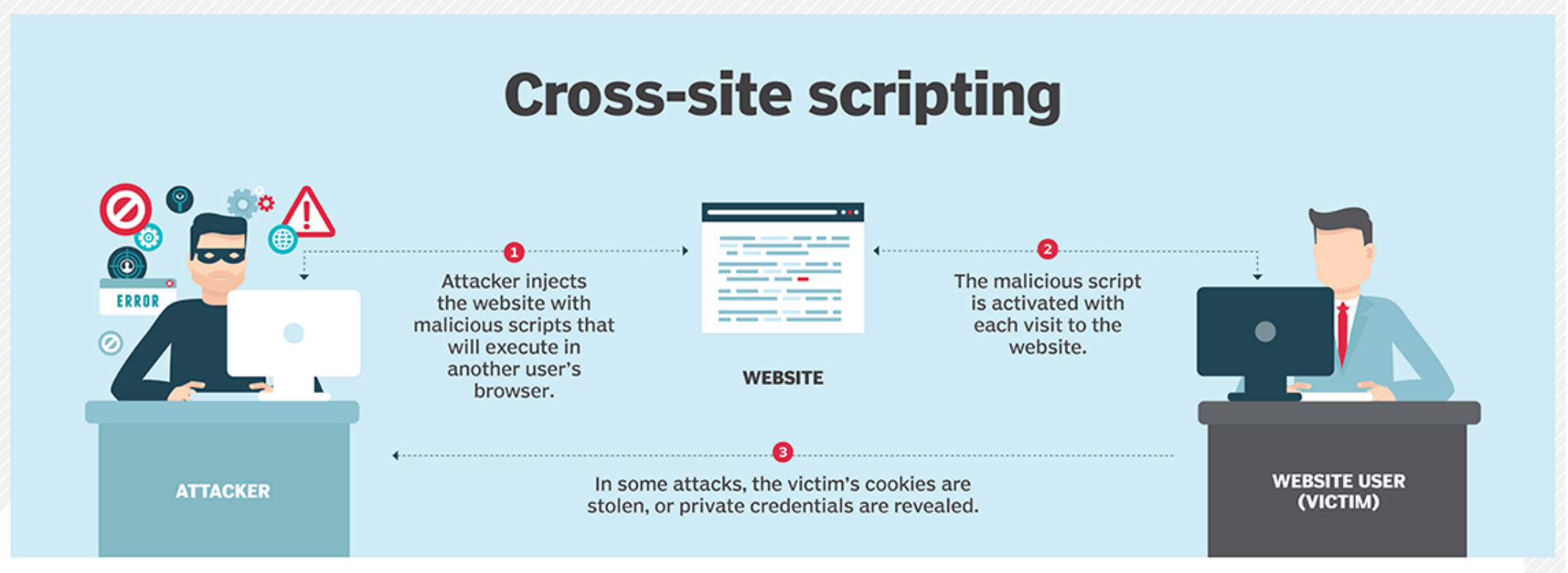
When does it occur?

An attacker injects malicious scripts into the web pages, which will be viewed by other users.
Malicious scripts are eventually executed by your Web browsers!
Most likely this is Javascript code.

How serious is it?

It has been estimated that approximately 65% of websites are vulnerable to an XSS attack in some form.

What is Cross-site Scripting?



https://searchsecurity.techtarget.com/definition/cross-site-scripting

Reflected XSS

Occurs when user input, such as search terms or other data, is immediately used by web pages without proper sanitization and reflected back to the user.

Example:
A user might enter a script into a search box that is immediately displayed on the result page without filtering, like so:

```
1
2   <!-- User input in a search form -->
3   <form action="/search">
4       Search: <input type="text" name="query">
5       <input type="submit">
6   </form>
7
8   <!-- If user inputs the following script in the search box -->
9   <script>alert('You have been hacked');</script>
10
11  <!-- The server includes the input in the response without sanitization -->
12  <h1>Results for: <script>alert('You have been hacked');</script></h1>
13
```

Stored XSS

Occurs when malicious scripts are permanently stored on target servers, such as in a database, message forum, visitor log, comment field, etc., and then later served to other users.

Example: A comment on a blog post that contains a script:

```
<!-- A user submits a comment that includes a malicious script -->
<div>User comment: <script>document.cookie='stealcookie='+document.cookie;</script></div>

<!-- This comment is stored and displayed to every visitor of the page -->
<p>Comments:</p>
<div>User comment: <script>document.cookie='stealcookie='+document.cookie;</script></div>
```

DOM-based XSS

Occurs when a script takes data from the client side, such as the URL, and processes it in a way that executes script without proper sanitization.

Example: Using URL parameters to modify the DOM:

In this case, the JavaScript will execute the script content of the name parameter as it is directly included in the DOM.

```
// URL: http://example.com/page?name=<script>alert('XSS')</script>

// JavaScript that takes the 'name' URL parameter and sets it as innerHTML of a div
document.getElementById('nameDiv').innerHTML = decodeURIComponent(location.search.split('name=')[1]);

```

Preventing XSS

- Strip HTML tags and JS before storing
- Escape all user entered data eg <?= esc($userEnteredData) ?>

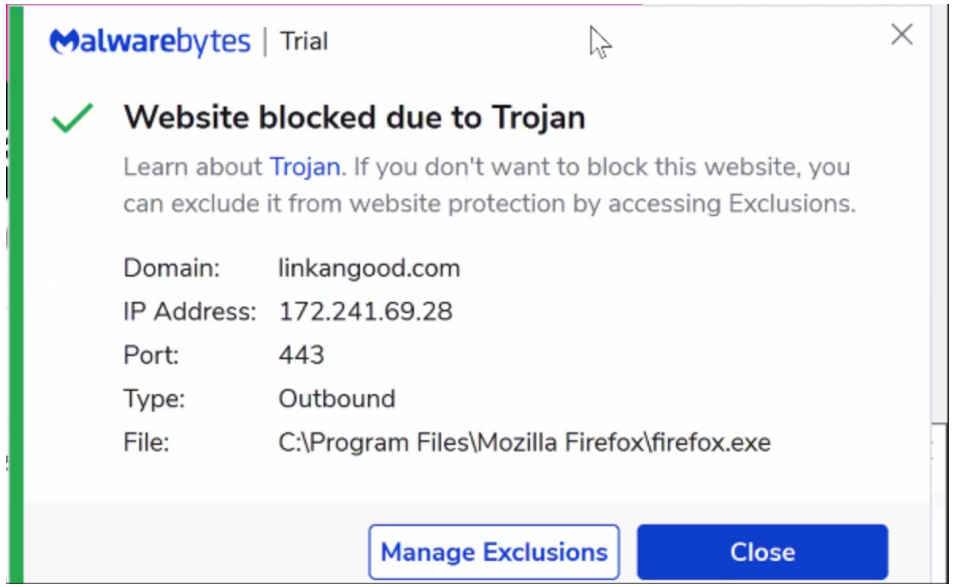
```
<table class="table table-striped">
    <thead>
        <tr>
            <th>Name</th>
            <th>Email</th>
            <th>Phone</th>
            <th>Status</th>
            <th>Actions</th>
        </tr>
    </thead>
    <tbody>
    <?php foreach ($users as $user): ?>
        <tr>
            <td><?= esc($user['name']) ?></td>
            <td><?= esc($user['email']) ?></td>
            <td><?= esc($user['phone']) ?></td>
            <td><?= esc($user['status']) ?></td>
            <td>
                <a class="btn btn-sm btn-info me-2" href="<?= base_url('
                <a class="btn btn-sm btn-primary me-2" href="<?= base_ur
                <a class="btn btn-sm btn-warning me-2" href="<?= base_ur
            </td>
            <!-- Add more user details as needed -->
        </tr>
    <?php endforeach; ?>
    </tbody>
</table>
```

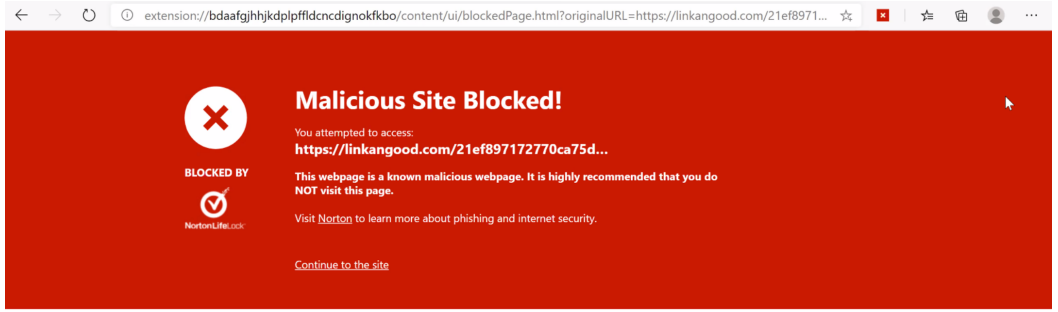
- **esc()**: This is a function call. In the context of many PHP frameworks, including CodeIgniter, esc() is a function used for escaping output to prevent Cross-Site Scripting (XSS) attacks. It performs character escaping suitable for HTML output, similar to PHP's native htmlspecialchars() function.

- Apply the esc() function to this name to escape any special HTML characters (like <, >, ", ', and &). This conversion prevents potentially malicious scripts embedded in the user data from executing in the browser, thus thwarting XSS attacks.

Example XSS in a Learning Management System

- Students reported getting blocked by their virus protection when they visited course content
- The site editor's laptop was infected and everything they edited on the web was inserting the Trojan javascript





Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS)

CORS, or Cross-Origin Resource Sharing, is a security feature implemented by web browsers to manage how resources on a web page can be requested from another domain outside the domain from which the first resource was served.

It is a set of HTTP headers that define whether browsers permit web applications running at one origin to access resources from a different origin.

How CORS works?

Origin Concept:

An origin is defined by the scheme (protocol), host (domain), and port of a URL.

For instance, the origin of https://example.com:443 is different from https://example.com:8443 because they have different ports.

Same-Origin Policy:

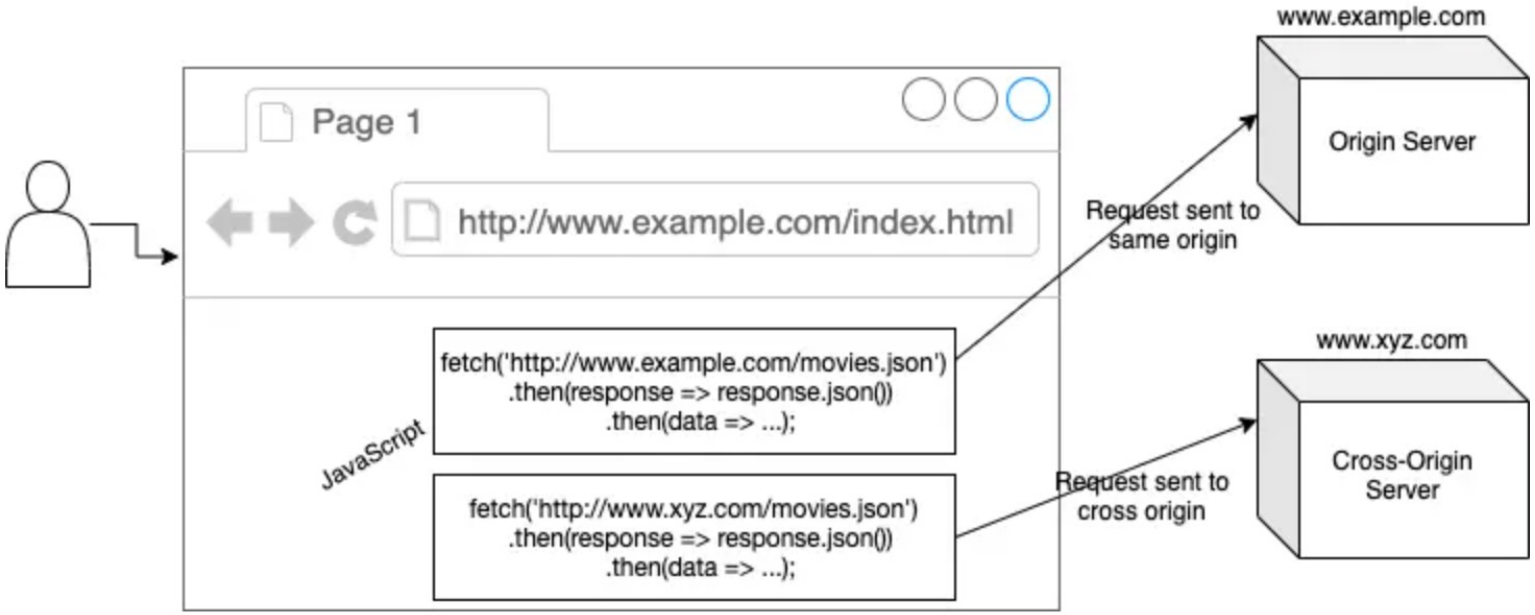
Browsers implement a same-origin policy by default. This policy restricts how a document or script loaded from one origin can interact with resources from another origin. Its primary purpose is to prevent malicious websites from interacting with other sites on which the user might be authenticated (like banking sites).

Cross-Origin Requests:

When a script tries to request resources from a different origin (cross-origin), the browser blocks the request unless the server on the other end allows it explicitly.

CORS Example

- Most likely currently experienced if you are developing locally can calling an API eg Locally developing React or Vue.js frontend
- Calling an API on your UQCloud Zone using FetchAPI
- Setting CORS headers in CodeIgniter
  https://gist.github.com/kenjis/e757d2b4193b6843724e447e6eaa1254



https://reflectoring.io/complete-guide-to-cors/

Why is CORS important?

- The victim visits evilwebsite.com while logged into goodwebsite.com.
- Evilwebsite.com loads a malicious script onto the victim’s computer that interacts with goodwebsite.com.
- Unaware, the victim runs the malicious script, which then makes a cross-origin request to goodwebsite.com. In this scenario, assume the request aims to access credentials needed for a sensitive action, like revealing the user's password.
- Upon receiving this cross-origin request, goodwebsite.com checks the CORS header to determine whether to allow the data transfer. In this case, let's assume that CORS is configured to permit requests with credentials (Access-Control-Allow-Credentials: true).
- The request passes validation, and consequently, the data is transmitted from the victim’s browser back to evilwebsite.com.

Cross Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF or XSRF) is a type of security vulnerability typically found in web applications. It allows an attacker to induce users to perform actions that they do not intend to do while they are authenticated. This vulnerability exploits the trust that a site has in the user's browser.

The danger of CSRF lies in exploiting the trust that a web application has in the user's browser. An attacker can manipulate a user’s browser to perform unwanted actions on a web application where they are authenticated. This can lead to unauthorized changes or transactions, data theft, and account compromise.

CSRF - Example

Here’s a step-by-step breakdown of a typical CSRF attack:

1. **User Login:** The user logs into a website, www.example.com, which authenticates the user and stores a session cookie on their browser.

2. **Malicious Request:** The user, without logging out from www.example.com, visits a malicious website, www.evil.com. This site executes a harmful action by requesting www.example.com to perform a specific task (e.g., changing the user's email address or transferring funds).

3. **Browser Submits Request:** The user’s browser automatically includes cookies pertaining to www.example.com with the request. If the session is still active, the server at www.example.com might execute the request without any additional verification.

4. **Action Executed:** Without the user’s consent, the action is carried out as if the user had intended it.

CSRF Prevention in CodeIgniter

Enable CSRF Protection

You can enable CSRF protection by altering your **app/Config/Filters.php** and enabling the *csrf* filter globally:

```
<?php

namespace Config;

use CodeIgniter\Config\BaseConfig;

class Filters extends BaseConfig
{
    public $globals = [
        'before' => [
            // 'honeypot',
            'csrf',
        ],
    ];

    // ...
}
```

Select URIs can be whitelisted from CSRF protection (for example API endpoints expecting externally POSTed content). You can add these URIs by adding them as exceptions in the filter:

```
<?php

namespace Config;

use CodeIgniter\Config\BaseConfig;

class Filters extends BaseConfig
{
    public $globals = [
        'before' => [
            'csrf' => ['except' => ['api/record/save']],
        ],
    ];

    // ...
}
```

HTML Forms

If you use the form helper, then `form_open()` will automatically insert a hidden csrf field in your forms.

Note

To use auto-generation of CSRF field, you need to turn CSRF filter on to the form page. In most cases it is requested using the `GET` method.

If not, then you can use the always available `csrf_token()` and `csrf_hash()` functions

```
<input type="hidden" name="<?= csrf_token() ?>" value="<?= csrf_hash() ?>" />
```

Additionally, you can use the `csrf_field()` method to generate this hidden input field for you:

```
// Generates: <input type="hidden" name="{csrf_token}" value="{csrf_hash}" />
<?= csrf_field() ?>
```

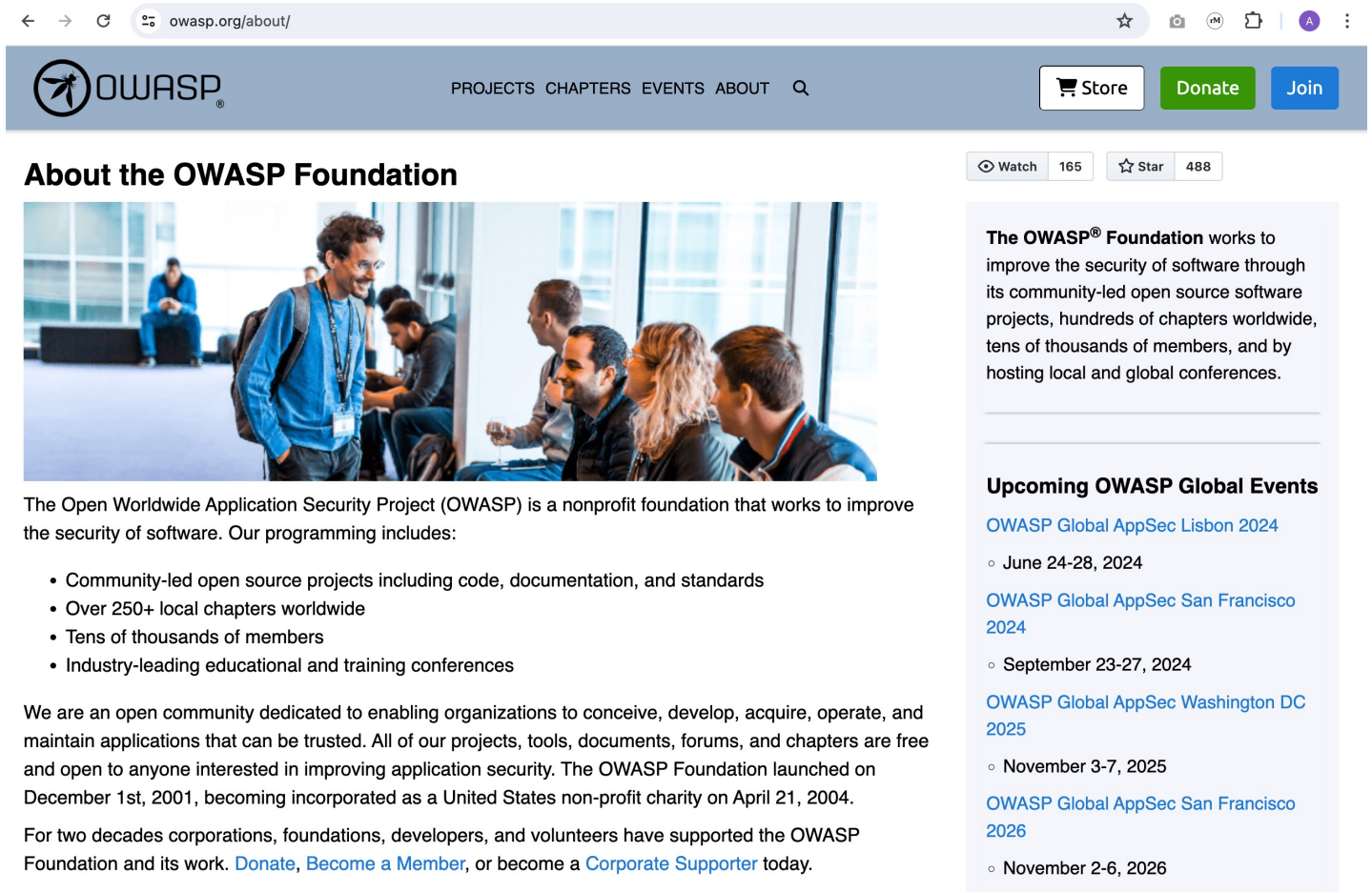
When sending a JSON request the CSRF token can also be passed as one of the parameters. The next way to pass the CSRF token is a special Http header that's name is available by `csrf_header()` function.

Additionally, you can use the `csrf_meta()` method to generate this handy meta tag for you:

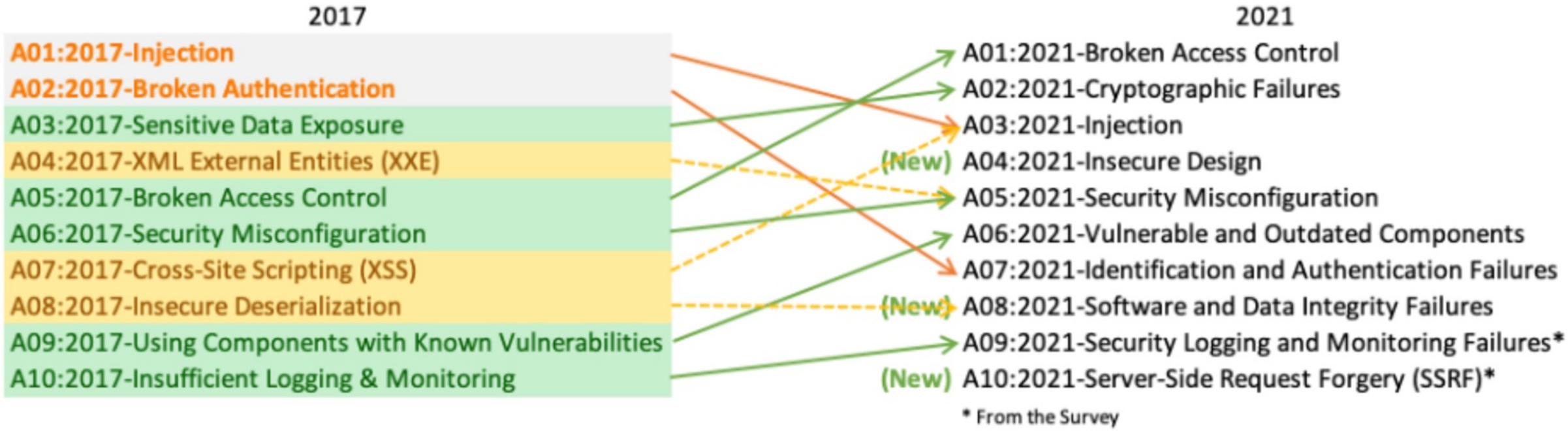
```
// Generates: <meta name="{csrf_header}" content="{csrf_hash}" />
<?= csrf_meta() ?>
```

Open Web Application Security Project

OWASP is an online community (https://owasp.org/) that produces freely-available articles, methodologies, documentation, tools, and technologies in the field of web application security.

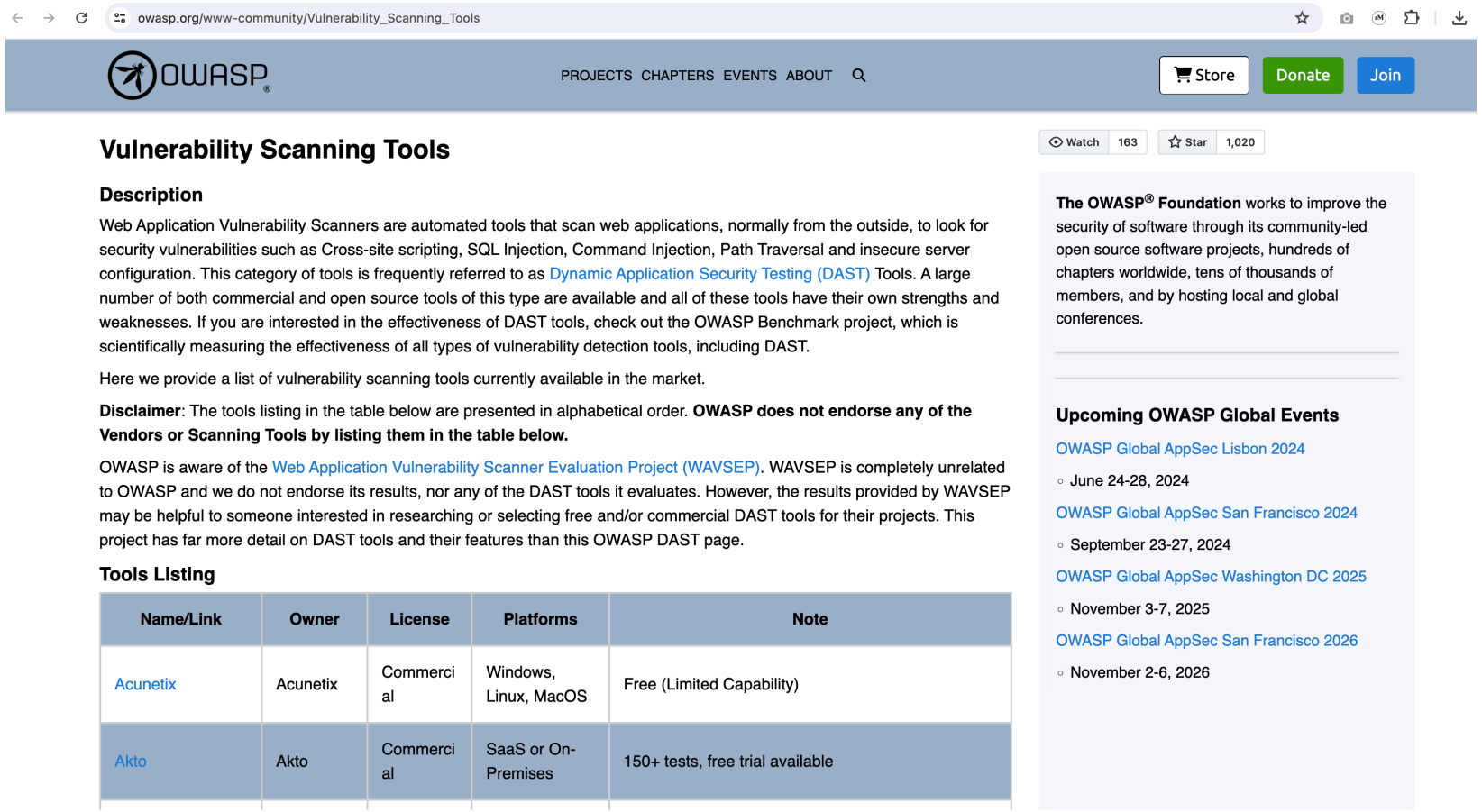


Top 10 Web Security Risks



https://owasp.org/www-project-top-ten/

Vulnerability Scanning Tools



https://owasp.org/www-community/Vulnerability_Scanning_Tools

Week 11: Todo

- Continue with your Project Assessment Item
- If you don’t have a Student Card, please get one!
  - You will need a Student Card for the Exam

Thank you